CompSys202/MechEng270
Object-oriented design and programming
Assignment #1 (5% of final grade)
Due: 11:00am, 7th August 2019

Learning outcomes

The purpose of this assignment is to target the following learning outcomes:

- Gain confidence coding using basic C++ syntax.
- Gain confidence using basic C++ objects/classes.
- Gain confidence compiling in Linux with makefiles.

1 Coding versus programming

This assignment is all about quickly getting to grips with the fundamentals of C++. This is more of a *coding* assignment, rather than a *programming* assignment. When it comes to programming, we would focus on the *design* of our program and the computational thinking that goes into the program (which can be done independently of a particular programming language). However, when it comes to coding, we are interested is getting our hands dirty with the technicalities of a particular programming language. Here, it is C++. For this reason, the assignment will be fully auto-marked – so expect your marks back very soon! But that also means you need to carefully set it up as expected, to avoid it failing in the compilation step.

2 What can we do with Money?

Well, that's a bit of a silly question. But in this assignment, we are interested in modeling some very basic operations on money. Each money amount has a *dollar* and a *cents* component. To be a valid money amount, we are requiring that each of these values is a natural number (i.e. 0, 1, 2, 3, ...). With a valid money amount, we can calculate the total amount of cents. We can also add more dollars and/or cents to it, but we have to be careful with subtracting dollars/cents to ensure it doesn't become invalid. Once we have two valid money amounts, we add them together or subtract one from another (assuming we don't invalidate one of them). We can also split tow money amounts, such that the two amounts become equal to each other. Each of these steps will be described in more details in the respective steps below.

3 Files provided

You have been given files containing method declarations, as well as one class definition (Money). You are welcome to add helper methods or data members to these files where necessary to structure and simplify your code, but you should not change any existing method declarations.

Makefile This Makefile will allow you to build and run your project. You can either run your project via Eclipse (see the course help videos to help you), or you can use the command line. Ultimately, once you finish your project and before you submit, you will want to **test your project using the command line in the ECSE Ubuntu machines** as this is how your project will be marked. The main commands relevant to this file are:

`make` Builds the `money_test` executable.

`make run` Builds the `money_test` executable and runs it.

`make all` This is needed by Eclipse, as it looks for a target named `all`.

`make clean` Removes all executable files from the working directory.

test.cpp This file contains some tests to make sure you have implemented some parts of the assignment correctly. It does not contain tests for the whole assignment, so do not assume that because all of the tests pass you will pass the assignment! You are encouraged to add more tests for parts of the assignment not already covered, but the tests will not be marked at all. To run all of the tests execute the command `make run` from a command-line terminal, or through Eclipse. You are also welcome to share your **tests** on Canvas or Piazza if you like (but obviously do not sure any of your other files!).

basics.h Declares some very basic methods used in the first part of the assignment. You may add to this file if you want to, but do not change the existing declarations.

basics.cpp Defines the basic methods that were declared in **basics.h**. You will see stubbed method implementations, which you will be completing in the first part of this assignment.

Money.h Declares a Money class, used for the second and latter parts of the assignment. You will need to modify this file (e.g. to declare instance variables to use for the Money class), but do not change the existing method declarations.

Money.cpp Defines the Money class that was declared in **Money.h**. You will see stubbed method implementations, which you will be completing in the second and latter parts of this assignment.

Particular sections of the **basics.cpp**, **Money.h** and **Money.cpp** files that you are expected to modify to complete this assignment are marked with comments such as:

```
// TODO: Implement
```

See Section 4 for more information about completing these implementation files.

4 Tasks

4.1 Complete the basic functions *[Task 1]*

As soon as you compile and run your code, you will notice the following output:

```
-------------------------------------------------
Total Run: 6
Total PASSED: 0
Total FAILED: 6
-------------------------------------------------
```

This is telling you that 6 test cases were run where none of them passed and all 6 of them failed. If you scroll up a little bit in that output, you will find more details on which particular tests failed, e.g:

```
FAILURE: test.cpp:36
    Expression 'totalCents(1, 5) == 105' evaluated to false
FAIL: Test 0 failed.
```

This tells you than an expression on line 36 of **test.cpp** did not evaluate as expected. If you then look into **test.cpp** you will see the following on line 36:

```
TestResult test_functionTotalSimple() {
    ASSERT(totalCents(1, 5) == 105);      // <--- This is line 36, failing here
    ...
}
```

In this case, the `totalCents()` function should have returned 105 since we are trying to calculate the total number of cents for a money amount that has 1 dollar and 5 cents. So if you open up **basics.cpp**, you will see the reason why! Complete the code here, and then re-run your tests (remember to compile your code before running the tests again). Once the error is fixed, the test will pass, and now you have 5 more tests to pass for Task 1:

```
PASS: Test 0 passed.
...
-------------------------------------------------
Total Run: 6
Total PASSED: 1
Total FAILED: 5
-------------------------------------------------
```

Repeat this until you clear all the test cases. For Task 1, if you have passed all the test cases provided to you, then you can be rest assured you have earned all the marks for Task 1! Once you see the following output, then you are ready to move onto Task 2:

```
-------------------------------------------------
Total Run: 6
Total PASSED: 6
Total FAILED: 0
-------------------------------------------------
```

Some notes on the expected behaviour:

- If the `totalCents()`, `sumAsCents()` or `split()` methods are provided with an invalid *dollars* or *cents* (i.e. something less than zero), then the value returned should be `-1`.
- For the `split()` method, if the total cents is an odd value, then we ignore the extra cent. For example, if the two amounts total `153` cents comes out to `76` cents).

4.2 Implement the Money class *[Task 2]*

We are now ready to tackle object-oriented programming, which is what our `Money` class is all about. The problem with the code developed in Task 1, is that these functions are a bit awkward and not very powerful if we want to model more complex operations with money.

Go to the top of **test.cpp**, and uncomment the following line:

```
// #define ENABLE_T2_TESTS
```

Compile and run your code again, and you will now notice the following output:

```
-------------------------------------------------
Total Run: 9
Total PASSED: 6
Total FAILED: 3
-------------------------------------------------
```

The 6 tests that have passed are those from Task 1. There are now 3 more test cases enabled, and they are all failing. Follow the same steps as in Task 1 to find out where those errors are, and complete the code. **However, this time, you have not been provided with all the test cases to get full marks for Task 2**. You will need to read the specifications below carefully, and create test cases of your own to make sure your implementation is correct:

- For a default Money (where neither a *dollars* or *cents* is provided), the *dollars* and *cents* will both be 0 (and therefore its *asCents()* with also be 0).
- The value stored in a Money object must never be negative. All Money must have *dollars* and *cents* values that are natural non-negative integers (0, 1, 2, 3, ...). If a Money is constructed with an invalid *dollars* or *cents*, then that value is set to 0 (e.g. if a Money of dollars=3 and cents=-2 is created, then the resulting components are dollars=3 and cents=0).
- If a *cents* value greater than 99 is provided, then you need to correctly "wrap" to the correct *dollar* and *cents* representation. For example, if a Money of dollars=3 and cents=201 is created, then the resulting components are dollars=5 and cents=1.
- Keep in mind that the above wrapping might also happen in combination with invalid values (e.g. dollars=-3 and cents=210 will give us $2 and 10c).

Remember, just because you pass the 3 tests provided in Task 2, this does not mean you will get all the marks for this section. You need to ensure you take all the specifications above into account.

4.3 Adding/subtracting dollars/cents to the Money class *[Task 3]*

Once Task 2 is complete, you are ready to improve on your `Money` class in Task 3. Go to the top of **test.cpp**, and uncomment the following line:

```
// #define ENABLE_T3_TESTS
```

Compile and run your code again, and observe any failing tests. The specifications are:

- The `Money::addDollars()` method will increase the money's dollar component by the respective amount. If the value is a negative, then nothing changes. This means, calling *addDollars()* can never decrease the money amount.
- The `Money::addCents()` method will increase the money's cents component by the respective amount. If the value is a negative, then nothing changes. This means, calling *addCents()* can never decrease the money amount. You need to keep in mind that wrapping might be needed, for example:
  - Adding even one cent might mean the cent component goes over 99.
  - The parameter to *addCents()* might itself have a value over 99.
- The `Money::subtractDollars()` and `Money::subtractCents()` methods will decrease the money's dollar and cents components respectively. As above, negative values are ignored and the money remains unchanged. This means, calling *subtractDollars()* and *subtractCents()* can never increase the money amount. What's more, attempts to decrease the money by an amount that is larger than the current money is again ignored (we can never have negative money values, but zero money is OK).
- Keep in mind that subtracting cents means we might need to wrap down. For example, subtracting 20c from $1.10 means we end up with $0.90.

Remember, just because you pass the tests provided in Task 3, this does not mean you will get all the marks for this section. You need to ensure you take all the specifications above into account.

4.4 Splitting and adding/subtracting Money to the Money class *[Task 4]*

Once Task 3 is complete, you are ready to implement the rest of the functionality inside the `Money` class for Task 4. Go to the top of **test.cpp**, and uncomment the following line:

```
// #define ENABLE_T4_TESTS
```

Compile and run your code again, and observe any failing tests. The specifications are:

- The `Money::add(const Money &other)` method will add the `other` money's amount to the `Money` object being operated on. The add does not affect the `other` Money object, the reason it is declared as `const` in the parameter). Again, make sure all correct wrapping takes place.
- The `Money::subtract(const Money &other)` method will subtract the `other` money's amount from the `Money` object being operated on. Again, the subtract does not affect the `other` Money object and make sure all correct wrapping (down this time) takes place. If the other Money is larger this the money object being operated on, then nothing should happen (i.e. never end up with negative money).
- The `Money::split(Money &other)` method will change the amount for both this `Money` object being operated on as well the `other` money's amount (notice it is note declared as `const`). What this method does is take the two amounts and redistributes it between the objects so they end up with an equal amount. If the total number of cents is odd, we ignore the last cent. Remember wrapping. R

Remember, just because you pass the tests provided in Task 4, this does not mean you will get all the marks for this section. You need to ensure you take all the specifications above into account.

Important: how your code will be marked

- Your code will be marked using a semi-automated setup. If you fail to follow the setup given, your code **will not be marked**. All submitted files must compile without requiring any editing. Use the provided tests and `Makefile` to ensure your code compiles and runs without errors. Any tests that run for longer than 10 seconds will be terminated and will be recorded as failed.
- Although you may add more to them (e.g. instance fields or methods, `#include` statements, etc), you must not modify the existing methods defined within the following files:
  - `basics.h`
  - `Money.h`
- Do not move any existing code files to a new directory, and make sure all of your new code files are created inside the existing directory.

- You may modify `test.cpp` as you please (for your own testing purposes); this file will not be marked at all. Be aware that your code must still work with the original `test.cpp`.
- Although for **this** assignment your code will not be inspected for good programming practices, you should still think about naming conventions for any variables and functions you declare. Also get into the habit of commenting your code where necessary to help the marker understand why you wrote a piece of code a specific way, or what the code is supposed to do. Use consistent indentation and brace placement. For future assignments in this course, you will be marked for such coding style – so it is a good idea to get into a good habit by practicing now.

Submission

You will submit via Canvas. **Make sure you can get your code compiled and running via command line with the test.cpp file provided in the ECSE Department Linux (Ubuntu) computers** (i.e. where your labs are). Submit the following, in a single ZIP archive file:

- A **signed and dated Cover Sheet** stating that you worked on the assignment independently, and that it is your own work. Include your name, ID number, the date, the course and assignment number. You can generate and download this in Canvas, see the Cover Sheet entry.
- The entire contents of the **src_for_students** folder you were given at the start of the assignment, including the new code you have written for this assignment. Ensure you **execute `make clean` before zipping the folder** so your submission does not include any executable files (your code will be re-built for marking).
- Do NOT nest your zip files (i.e. do not put a zip file inside a zip file).

You must double check that you have uploaded the correct code for marking! There will be no exceptions if you accidentally submitted the wrong files, regardless of whether you can prove you did not modify them since the deadline. No exceptions. Get into the habit of downloading them again, and double-checking all was submitted correctly.

Academic honesty

- The work done on this assignment must be your own work. Think carefully about any problems you come across, and try to solve them yourself before you ask anyone for help (struggling a little with it will help you learn, especially if you end up solving it). If you still need help, check on Canvas (if it is about interpreting the assignment specifications) or ask in the Lab help clinics (if you would like more personal help with C++). Under no circumstances should you take or pay for an electronic copy of someone else's work.
- All submitted code will be checked using automated software similarity tools specifically designed for programming assignments. Submissions with suspicious similarity will result in an Investigative Meeting and will be forwarded to a Disciplinary Committee.
- Penalties for copying will be severe – to avoid being caught copying, don't do it.
- To ensure you are not identified as cheating you should follow these points:
  - Always do individual assignments by yourself.
  - Never show or give another person your code.
  - Never put your code in a public place (e.g. Reddit, public Github repos, forums, your website).
  - Never leave your computer unattended. You are responsible for the security of your account.
  - Ensure you always remove your USB flash drive from the computer before you log off.

Late submissions

Late submissions will incur the following penalties:

- 15% penalty for zero to 24 hours late
- 30% penalty for 25 to 48 hours late
- 100% penalty for over 48 hours late (dropbox automatically closes)