National University of Singapore
School of Continuing and Lifelong Education

TIC1001: Introduction to Computing and Programming I
Semester I, 2019/2020

Lab 2
Incremental Coding

Quick Review

We learned that:

1. It is important to understand the computation state, i.e. the collection of values that are manipulated by your program.
2. We should spend time figuring out the algorithms (the steps in solving a problem) before coding start.
3. We can test a program more effectively by using equivalent class testing. In this lab, we are going to expand on the key ideas above and show you a useful coding practice that can be used to tackle large piece of coding.

Common Coding Issues

Most of you have tried (or finished) coding problem set 2 by now. You should also noticed that the difficulty is seemingly much higher than problem set 1. This perceived difficulty comes mainly from complexity, i.e., too much details to take care at any point in time. Some of the issues you may have encountered are:

1. Too many syntax errors, your code cannot compile.
2. Your code compiled but produce very different result(s).
3. Your code produced some correct answers, but failed other test cases.
4. (For task 2) You do not know how to start as the algorithm is hard to figure out.

The first three issues can be mitigated by a practice known as **incremental coding**. This practice advocates the following procedure in coding:

1. **[Code]** Implement one small chunk of logic (e.g. usually correspond to one step in the algorithm, or one simple function). To verify the logic correctness, you should include **debug messages** (e.g. `printf`, `cout`) that prints out key values (e.g. values computed/changed by this piece of code etc). This helps to verify the computation state during the execution of your program.
2. **[Compile]** Compile to clear all syntax error. This is much more manageable now since we only code a small chunk—syntax error must be within these few lines of new code. Fix the syntax errors until the code compiles.
3. **[Test]** Test this chunk of logic. Correct any logic errors found before proceeding. You should test all possible **execution paths** in your code. E.g., make sure you test both the `if` and `else` branch of code, make sure looping works for the minimum and maximum (if well defined) iterations, etc. Fix the logic errors.

4. Go back to step 1 for another step.

To illustrate the idea, we will use incremental coding for the first two steps for task 1 in problem set 2. For ease of reference, we have duplicated the steps below:

Converting from RGB to HSL

For this task, you will convert an RGB colour to its HSL equivalent.

1. Scale the RGB values to the range 0–1. Since the inputs are on a 0–255 scale, we can simply divide by 255.
2. Find the maximum and minimum value amongst the R, G and B component.

1. **Code:** We will attempt to translate the first step as follows:

```
HSL rgb_to_hsl(int red, int green, int blue) {
    int hue, sat, lum;

    // Your answer here.

    // Scale RGB to [0...1]
    double rRed, rGreen, rBlue;

    rRed   = red   / 255;  // Logic error purposely added
    rGreen = green / 255;  //
    rBlue   = blue / 255;  //

printf("R:%f G:%f B:%f\n", rRed, rGreen, rBlue); // Debug Message
```

   Notice that we added a debug message to show the result **of this step**, i.e. the three values that we are trying to convert to 0–1. In general, you should print values that are computed/changed by this new piece of code. **Also, debug message should be added as you code, rather than an afterthought.**

2. **Compile:** If you compile the above and find compilation error (syntax error), you should find it is relatively easy to clear the errors as you only need to focus on these few line of codes.

3. **Test:** Now, when we run the above using the sample values in the code template given (i.e. **R is 24, G is 98 and B is 118**), we see:

   Output

```
[Running].......
R:0.000000 G:0.000000 B:0.000000
The result is Hue:0, Sat:4200112, Lum:0
```

   The first line is the printout from our debug message. We should noticed the issue right away: **R, G and B should not be zero** for the sample values!

Again, the small piece of code allows you to focus on the potential issue. You should be able to quickly correct it (the integer division is the culprit). Correct the code and repeat the first three steps **[Code->Compile->Test]**:

```
HSL rgb_to_hsl(int red, int green, int blue) {
    int hue, sat, lum;

    // Your answer here.

    // Scale RGB to [0...1]
    double rRed, rGreen, rBlue;

    rRed   = red   / 255.0;  // Logic error corrected
    rGreen = green / 255.0;  //
    rBlue  = blue  / 255.0;  //

printf("R:%f G:%f B:%f\n", rRed, rGreen, rBlue); // Debug Message
```

You will see the following output:

Output

```
[Running].......
R:0.094118 G:0.384314 B:0.462745
The result is Hue:0, Sat:4200064, Lum:0
```

A quick verification on the calculator shows that the values are now correctly converted to real number between 0 and 1. Test a few more values before you move on, applying equivalent class testing if appropriate/possible. In this case, you can test the values 0 and 255 in addition of the sample values.

One the above is done, we can proceed to the next step (find max and min). As the steps is quite involved, we can code only the "find max" as the next chunk of code:

```
HSL rgb_to_hsl(int red, int green, int blue) {
    int hue, sat, lum;

    // Your answer here.
    // Step 1. Scale RGB to [0...1]
    double rRed, rGreen, rBlue;

    rRed   = red   / 255.0;  // Logic error corrected
    rGreen = green / 255.0;  //
    rBlue  = blue  / 255.0;  //

    printf("R:%f G:%f B:%f\n", rRed, rGreen, rBlue); //Debug Message

    // Step 2a. Find Max
    double maximum;

    maximum = rRed;

    if (maximum < rGreen) {
        maximum = rBlue;  // Logic Error purposely added!
    }
    if (maximum < rBlue) {
        maximum = rBlue;
    }
    printf("Maximum = %f\n", maximum);  // Debug Message
```

Compile and test. You find that:

Output

```
[Running].......
R:0.094118 G:0.384314 B:0.462745
Maximum = 0.462745
```

This example shows the importance of **testing all execution paths**. If you just test this one case, you will wrongly concluded that your code is correct! To throughly test this piece of code, you should at least try three cases where each of the R, G or B is the largest.

The logic error will be discovered if you use a test case where G is the largest, below is the debug message for R=20, G=240, B= 40:

Output

```
[Running].......
R:0.078431 G:0.941176 B:0.156863
Maximum = 0.156863
```

Maximum is wrong! It should be should be 0.941176 (Green's value)

You should be able to find and fix the logic bug. Once the bug is fixed and you are satisfied. Move on to the next piece of logic, e.g. "find minimum". You will continue add and test code in small incremental chunk until you finish the entire function.

Frequently Asked Questions

1. What qualifies as a "small chunk of logic"?

   Ans: We usually code one *step* in the logic. As you grow more proficient, your *step* can be larger. However, you should never code more than 15-20 lines of code without a [Code-Compile-Test] cycle.

2. Wont those *debug messages* get a bit cluttered afterwards?

   Ans: If you pay attention to the debug messages above, you will notice that they are not indented (i.e. always flushed to the left). This is one simple way to make them *stick out* from your other code. You can easily find them and comment them off once you are very sure about a piece of code. You should refrain from deleting them as you can easily comment/uncomment them if needed.

3. How do I debug a loop using debug message?

   Ans: You should print the loop counter (the variable(s) that control the loop iteration), the result computed by the loop.

4. How do I debug a function?

   Ans: You should print out the function parameters (make sure the values are passed properly by the caller) as well as the return value (just before the `return ...;` statement) if applicable.

5. Isn't incremental coding very slow?

   Ans: It is actually not much of a hassle once you get used to it. With Visual Studio Code or any IDE, it is actually quick easy (code a bit, press run, correct bugs, repeat). The key tenant of incremental coding is to keep complexity in check: debugging 5–10 lines of code is much easier compared to debugging 20–30 lines of code in one go.

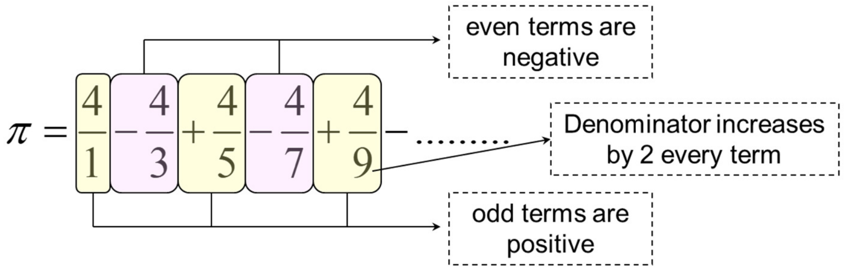
   "Go slower in order to move fast"

Exercise in using debug messages

The value of π constant can be approximated as follows:

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$$

On closer inspection, we can observe the following pattern:



$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots\cdots$$

It is also clear that we can get better pi approximation by calculating more terms. The example above uses 5 terms and give only 3.34. However, if you calculate 10,000 terms, you get an approximation of 3.14159. . . !

Let us define a function `double calculatePi( int n )` which returns the approximation of π by using n number of terms. If you want to challenge yourself, please do not read the following page and try to figure out the algorithm yourself.

Suppose you figured out the algorithm as follows:

```
1. pi starts at 0, denom starts at 1
2. As long as we have more terms to calculate:
     3. term is calculated as 4 / denom
     4. If it is the even numbered term, subtract term from pi
     5. Else it is an odd numbered term, add term to pi
     6. Increase denom by 2
7. Go to step (2) and repeat
```

Again, you can challenge yourself by translating the above into C/C++ code. Otherwise, you can open up the given template `pi.cpp` and see a sample translation.

Unfortunately, the sample translation has a number of logic bugs in it. Try to insert a few suitable debug messages to figure out the bug. Note that this is for you to learn what/where/when to print the variable values.

Submit the corrected function into Coursemology.