Programming Studio 2 – C++

Week 5 Class 2

Dr Ruwan Tennakoon
Dr Steven Korevaar

Preparation

Download the starter code on canvas (Modules -> Week5 -> Class 2)

Open VSCode

- On mac Just open and, in the VScode terminal navigate to your working directory. Copy the starter code to that same working directory.
- On Windows. Open VSCode in WSL2 and copy the starter code to your working directory.

Make sure you see the files copied in terminal (type ‘ls’ in terminal)

Build the software: type ‘make’ in terminal

Go through the code.

Programming Studio 2 – C++ Stream

Recap from class 1:

Writing large programs, C++ Overview

Class 2 overview:

1. We had a lot of issues with compilation, importing MCPP, and Makefiles! **Has everyone managed to get a basic program up and running?**
2. Go over the solutions from class 1!
3. Extend the solution from class 1 with better programming practices
    b) Testing paradigms (defensive vs contractual programming)
    c) Black box testing
4. Structured programming

Class 3:

Checkpoint 1!
We will be asking questions about the assignment specification, responsibilities, black box testing, and checking your progress on the main menu.

Class 2 – Testing Programs

How do we know the code we have written is correct?

We test it!

What do we test?

Everything! But mainly where there is potentially unexpected input

How do we test?

1. **Unit tests:** tests that are done inside code, that test specific functionality of the code.
2. **Black box testing:** testing that is done on the final program, mimics user input, and verifies that the program's output is correct.

We focus on black box testing in this course.

Class 2 – Black Box Testing

What is it?

Mimics how we test our programs during development: we give the program some input and check its output against the correct answer.

- If a program outputs the correct answer for all your test cases, then the program should be fully functional.

What are the pitfalls of this approach?

- Your test cases must be comprehensive!

How to in C++:

Black box testing can be done by using input/output redirection (with < and >):

```
./programName < input.txt > output.txt
```

Then you can test the actual output file with an expected output file with “diff”:

```
diff -w actual_output.txt expected_output.txt
```

Class 2 – Black Box Testing – Example!

Read the following into a program to build a 2D Char array called `env` containing only 'x's representing walls and '.'s representing open spaces:

E.G.: For the following input:

```
// the size of the env
5 6
xxxxxx
x....x
xxx.xx
x....x
xxxxxx
```

The program should output to the console:

```
Please enter the size of the env: 5 6
Height: 5, Width: 6
Please enter the env:
xxxxxx
x....x
xxx.xx
x....x
xxxxxx
The entered env was:
xxxxxx
x....x
xxx.xx
x....x
xxxxxx
```

What can we actually test with Black box testing?
What are the limitations of it?

Step 1: What inputs are there?
Step 2: What variations of the inputs could there be?
Step 3: What should the expected behaviour be given those variations?

Good tests should cover as much of that variation as possible!

Class 2 – Programming Paradigms To Avoid Errors

Defensive Programming

Programming such that all possible variations of inputs are handled properly.

Typically used for user input handling.

```
if (height <= 0 || width <= 0){
        std::cout << “Height or width is not correct!” << std::endl;
}
```

Programming by Contract

Assuming the user of the program (or the programmer themselves) handles the validity of input themselves.

Typically used for the inputs to functions that do not come from users directly.

```
// Contract:
// length and width must be integers greater than zero
Env(unsigned int length, unsigned int width);
```

We separate out our defensive programming from programming by contract to make code easier to read:

Do validation of the inputs -> do the main logic of our code.

Doing them at the same time can be confusing to understand.

RMIT UNIVERSITY

Class 2 – Structured Programming

Structured Programming is the concept every block of code has a:

- Single point of entry
- Single point of exit

Structured Programming restricts code to:

- Sequence
- Selection
- Iteration (repetition, recursion)
- Sub-routines (functions, methods, classes)
- Blocks

Theoretical basis:

- Any function computable by a Turing machine can be solved using only these above structures

Style guide is adapted from Google, it is not the be-all-end-all in terms of good programming, but it serves as a strong foundation on how to avoid major programming failures.

NOTE: THE STYLE GUIDE IS ENFORCED FULLY! Even if you have a "justifiable reason" to use multiple returns, you still may lose marks.

RMIT UNIVERSITY

Class 2 – Exception Handling

If we have the following code:

```
std::cout << "Please enter a number from 0-9: ";
std::string in_str ;
std::cin >> in_str ;
int in_num = std::stoi(in_str) ;
std::cout << std::endl ;
if (in_num >= 0 && in_num <= 9) {
    std::cout << "You have entered the number: " << in_num << std::endl ;
}
else {
    std::cout << in_num << " is not between 0-9!" << std::endl ;
}
```

What happens if we input something that's not a number?

Class 2 – Exception Handling

We can add "exception handling" to "try" a block of code and "catch" any exceptions that occur in that block:

```
std::cout << "Please enter a number from 0-9: ";
std::string in_str ;
std::cin >> in_str ;
int in_num = -1 ;

try {
   int in_num = std::stoi(in_str) ;
}
catch (...) {
   std::cout << "Please enter in a number! " << in_str << " is clearly not a number." << std::endl
;
   return 0 ;
}

std::cout << std::endl ;
if (in_num >= 0 && in_num <= 9) {
   std::cout << "You have entered the number: " << in_num << std::endl ;
}
else {
   std::cout << in_num << " is not between 0-9!" << std::endl ;
}
return 0;
```

RMIT UNIVERSITY

Class 2 – What to work on next?

Additional Implementation Exercises:

1. Flatten the terrain and build your structure on it. You should only flatten an area equal to the area of your structure.
2. Update the code so that you do not need to input length and width. Infer from the inputted environment.

Testing and Error Handling Exercises:

1. Extend error handling to all other user input areas.
2. Identify interesting examples to test – edge cases.
3. Identify edge cases for assignment 1 milestone 3.
4. Identify errors in the code completed so far. Specifically, in the Env class. (Hint: what happens in the program when certain elements are improperly initialised?)

Class 3:

Prepare for checkpoint 1!
We will be asking questions about the assignment specification.

Distribution of responsibility, ought to be fair! Don't have one student do the hard work, and others.

An attempt at writing black box test cases for the assignment.

Get the menu showing as a minimum!

RMIT UNIVERSITY