PHAS0100 2021/22: Research Computing with C++

Assignment 1: The Game of Life Simulator

Late Submission Policy: Please be aware of the UCL Late Submission Policy, which is here: https://www.ucl.ac.uk/academic-manual/ *(chapter 4, section 3.12)*

Extenuating Circumstances
The course tutors are not allowed to grant Extenuating Circumstances for late submission. You must apply by submitting an Extenuating Circumstances form which can be found on the 2021/22 EC information website. The Department EC Panel (DECP) will then decide whether or not to grant extenuating circumstances and email the module lead confirming that Extenuating Circumstances have been granted and providing a revised submission date.

Assignment Submission
For this coursework assignment, you must submit by uploading a single Zip format archive to Moodle. You must use only the zip tool, not any other archiver, such as .tgz or .rar. Inside your zip archive, you must have a single top-level folder, whose folder name is your student number, so that on running unzip this folder appears. This top-level folder must contain all the parts of your solution. Inside the top-level folder should be a git repository named PHAS0100Assignment1. All code, images, results and documentation should be inside this git repository. You should git commit your work regularly as the exercise progresses.

You should clone the following repository and use as the starting point for the coursework: https://github.com/UCL/PHAS0100Assignment1

Due to the need to avoid plagiarism, do not use a public GitHub repository for your work - instead, use git on your local disk (with git commit but not git push), and **ensure the hidden .git folder is part of your zipped archive** as otherwise the git history is lost. We will be providing access to a private GitHub classroom repository that you can use as a remote backup option – there will an email/moodle announcement with details on this. It is important you do not push to a public GitHub repository as pieces of identical (or suspiciously similar) work will both be treated as plagiarised. Whilst it is fine to look up definitions and general documentation for C++/libraries online it is not acceptable to use code taken from the internet or other sources or provided by someone else for the problem solutions. The exception to this is the initial PHAS0100Assignment1 project that should be used as a starting point for your project.

We recommend you use Visual Studio Code IDE with the Ubuntu 20.04 docker image for development (the correct Dockerfile is included for you in the above repository). This is not a requirement but if you do use a different setup then you need to ensure that your code compiles and runs on Ubuntu 20.04 and with g++ 9.3.0 (enforcing C++17) and CMake 3.16.3 as this is the environment the markers will use to test the code.

Marks will be deducted for code that does not compile or run. Marks will also be deducted if code is poorly formatted. You can choose to follow your own formatting style but it must be applied consistently. See the Google C++ style guide for a good set of conventions regarding code formatting. In general, if adding to an existing project it is best to follow the existing style.

It is important that you follow the file/folder/executable structure of the PHAS0100Assignment1 of the starting project as well as the form and name of any functions/classes/executables described in any questions below. This is because we will be automating basic checks of the submitted work so if you have not followed the pre-defined structure those checks will fail and you will lose marks. For a similar reason you should make sure your unit tests are included when `ctest` is run by ensuring each new test file you create has an appropriate `add_test` entry in `PHAS0100Assignment1/Testing/CMakeLists.txt`

Preliminaries
In this coursework you will create a simulation program that models Conway's Game of Life, a 2D cellular automaton devised in 1970 by the mathematician John Conway. You will construct the

relevant classes and unit tests to store, print and manipulate the 2D cellular data. You will then build a simulation application that can start the game with different initial conditions based on either a set of random starting cell values or by reading in the initial conditions from a text input file. In the final part you will explore stationary patterns (known as *still lifes*) that emerge in the Game of Life given particular initial conditions.

Reporting Errors
Contact the lecturer directly if you believe that there is a problem with the example CMake code, or the build process in general. To make error reporting useful, you should include in your description of the error: error messages, operating system version, compiler version, and an exact sequence of steps to reproduce the error. Questions regarding the clarity of the instructions are allowed, but obvious "Please tell me the answer" type questions will be ignored. Any corrections or clarifications made after the initial release of this coursework will be highlighted like this.

Important: Read all these instructions before starting coding. In particular, some marks are awarded for a reasonable git history (as described in Part D), that should show what you were thinking as you developed.

Part A: Data structure for 2D grid of cells (20 marks)
The Game of Life takes place on a 2D grid of cells where each cell can be either alive or dead. In this first part you will create and a test a class to store status of each cell in the grid.

1. First familiarise yourself with the rules of Conway's Game of Life and take a look at some of the common pattern examples so that you know what to expect when running your simulation: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

   [0 marks]

2. Create a class (with an appropriate name) to store the status (alive or dead) of the 2D grid of cells. The user should be able to specify the size of the grid in terms of columns and rows as a constructor argument. The class should make use of the appropriate C++ standard library containers to store the cell data and should have methods to print the current grid to screen and to get and set individual cell contents. Include a unit test to check instantiation of your class.

   This is an example of how the print to screen should be formatted for a 5 by 5 grid with 4 cells alive shown as letter 'o' and with a '-' dash for dead cells):

   ```
   - - - - -
   - - o - -
   o - o - -
   - o o - -
   - - - - -
   ```

   Hints:

   The gol file name prefix and namespace name comes from **G**ame **o**f **L**ife.

   When implementing the print function use \n to create each of the newlines instead of std::endl as this avoid buffer flushing so that the output for all rows will show at the same time. You will need a final std::endl to flush the buffer.

   [4 marks implementation, 2 marks unit test, **7 total**]

3. Next add functionality to your class to initialise the grid such that each cell's status chosen randomly. Add a second constructor for the class with arguments that specify the overall size

of the grid (number or rows and columns) as well as the total number of alive cells to place randomly. Which particular cells are alive should then be chosen randomly.

When unit testing this new functionality check that the total number of placed cells is correct and confirm that different instances generate different patterns of alive cells.

Hints: Consider checks on the user inputs.

[3 marks implementation, 2 marks unit tests, **5 total**]

4. Next add functionality to your class to allow it to initialise cell contents based on the values provided in an input text file. As with the random initialisation you should add this as a new constructor, this time with a std::string argument for the file name to load.

   A set of input text files `glider.txt`, `oscillators.txt` and `still_lifes.txt` are provided in `PHAS0100Assignment1/Testing/Data`. You are free to test your class using your own files but you should ensure your class can successfully read these files. You should assume the following: each line in the file represents a new row and then along each line the character 'o' indicates an alive cell and a '-' dash represents a dead cell. Each cell in a row is separated by a single whitespace.

   Hints: watch out for whitespace between each element in a row and elements and don't assume the number of rows and columns are always equal.

   Consider consistency checks and data validation for the input data.

   [3 marks implementation, 2 marks unit tests, **5 marks total**]

5. Add a method to your class that for a given cell (row, column) returns the number of neighbouring cells that are alive. As an example, consider the cell in the centre of the example in Part A.2, this cell has 3 neighbouring cells that are alive. When calculating the number of neighbouring cells for a cell on the edge of the grid you should only include contributions from neighbours within the grid and assume other neighbours outside of the grid are dead. As an example, the cell in the middle row and leftmost column has 1 alive neighbour.

   [2 marks implementation, 1 marks unit tests, **3 total**]

Part B: Game of Life Simulator (20 marks)
In this part you will build a Game of Life class that will model the evolution of the cells based on the rules of the game of life and then create a command line application that brings all the components together.

6. Implement a Game of Life class:
   a. Choose an appropriate name for your class and create the corresponding source and header files.

      [1 marks]

   b. The class should store an instance of the 2D grid of cells class from Part A to represent the current status of cells.

      [2 marks]

   c. Implement a `TakeStep` member function that takes the current grid of cells and works out whether each cell should be alive or dead on the next iteration based on the rules of the Game of Life:

      i. A dead cell with exactly three live neighbours should become a live cell.
      ii. A live cell with two or three live neighbours should stay alive.
      iii. A live cell with less than two or more than three live neighbours should die.

      [3 marks implementation, 2 marks unit tests, 5 total]

d. The class should also have a `PrintGrid` function that prints to screen the current cell grid.

[2 marks]

Hints:
Consider whether it is simpler to pass pre-initialised instances of the 2D grid of cells class to the Game of Life class constructor as opposed to configuring and initializing them from within the Game of Life.

You may need two instances of the 2D grid of cells class in order to implement the `TakeStep` algorithm: one instance that represents the cell content for the current step and then another instance to temporarily store the content for the next step based on the Game of Life rules and nearest neighbors.

In order to unit test the `TakeStep` method you may need an additional method to get the current status for a given cell (row, column).

When implementing the `PrintGrid` you should reuse the print function from the 2D grid class and may want to add an additional delimitator to the screen to make clear when a new iteration occurs.

[a-d, **10 marks total**]

7. Next create a command line application to run the Game of Life simulation

   a. Create a skeleton command line application `gofSimulator` that compiles and can be run from the build director as `./bin/gofSimulator`

   [1 marks]

   b. Then implement command line arguments that allow the user to choose between the following two options:
   1: specifying a text file input or
   2: starting with random cell contents for the initial conditions.

   In the case of the text file input the command line arguments should allow the user to specify the input file; and in the case of random initial cell values the user should be able to specify the size of the grid and the total number of initial alive cells to place.

   The app should also have an argument to control the number of generations to simulate.

   The app should print a useful help message when run with no arguments and also when run with `--help or -h` options.

   [3 marks]

   c. The app should configure the Game of Life class appropriately depending on whether the user has selected random initial or text file input.

   [2 marks]

   d. The app should then loop through the requested number of steps (generations) printing to screen the current grid of cells.

   [2 marks]

Hints:

You can use basic C++ command line parsing:
http://www.cplusplus.com/articles/DEN36Up4/
Or you could try integrating a command line parser such as:
https://github.com/CLIUtils/CLI11(which is header only).

Consider adding a sleep between each generation (see https://www.cplusplus.com/reference/thread/this_thread/sleep_for) so that you can see the simulation progressing in real time.

[a-d, **8 marks total**]

8. Provide screengrabs of the result of your application after a number of evolutions for the following cases: 1) after running with `glider.txt` input file and 2) after starting a 7 by 7 grid with random initial cell values (15 cells alive).

[1 marks each screengrab, **2 marks total**]

Part C: Finding stationary patterns (15 marks)
One feature of the Game of Life is the presence of so called *Still Life*'s. These are stationary patterns that do not change despite stepping from one generation to another.

9. In this part you will write a new command line app to find *Still Life*'s by comparing the current and next generation of cells and calculating if there has been no change between them.

   a. Create a new command line app arguments to control the number of alive cells, the grid size and the number of iterations to try for each configuration when searching for *Still Life's*.

   [2 marks]

   b. Given the inputs the command line app should try out different random initial conditions and for each, run the user specified number of calls to the Game of Life class' `TakeStep` method.

   [4 marks]

   c. The program should calculate if any `TakeStep` results in no change and if so print the resulting *Still Life* pattern to the screen.

   [5 marks]

   d. Run the program for a 4 by 4 grid of cells and upload screen grabs of at least two Still Life's that your code found (2 marks). See how many still life's in total you can find and writeup your result in the README.md (2 marks).

   [4 marks]

[a-d, **15 marks total**]

Part D: Additional Considerations (15 marks)

10. Nice git commit log. Effective in-code commenting. Code formatting.

[**10 marks**]

11. Update the front page README.md to give clear build instructions, and instructions for use.

[**5 marks**]