The Game of Life

PHAS0100 2022/23: Research Computing with C++ Assignment 1

Errata

- `gof` -> `gol` in Section 2.2a.
- Clarify that the `CMakeLists.txt` (and other files/folders) can be freely edited as long as any names specified in these instructions are kept.
- Add that exceptions can be used when handling errors associated with loading a grid from a file.
- Add how paths can be handled in unit tests in Section **Assignment Submission**.
- Add that the zip file should be unzipped as a test inside the docker container in Section **Assignment Submission**.
- Clarify that *any* still lifes on a 4 by 4 grid count, even if they would not be still lifes on larger grids.
- Clarify that Section 5 contains only extra, optional hints; there are no specific instructions or associated marks.

If you believe you have found a mistake in the assignment, please contact the module leader.

Late Submission Policy

Please be aware of the UCL Late Submission Policy (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 Extenuating Circumstances information website. The Department EC Panel 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 exactly `PHAS0100Assignment1`. All code, images, results and documentation should be inside this git repository. You must include the `.git` folder otherwise we cannot mark your git commit log. Your folder structure should look similar to (but not exactly the same as):

```
- 314159   // YOUR STUDENT NUMBER
  - PHAS0100Assignment1
    - .git
    - README.md
    - src
    - test
    - ...
```

You should test that your zip file produces this structure **from inside the docker container**. This will guarantee that your zip file will extract correctly during the submission process (which uses the docker container).

You can use GitHub for your work if you wish, although we will only mark the code you submit on Moodle. Due to the need to avoid plagiarism, do not use a public GitHub repository for your work. We have provided a private repo unique to you accessible through Github Classroom. **To access the starting project, using the following link**:

https://classroom.github.com/a/UsaODd0l.

After you accept the permissions, you will have access to a repository named `game-of-life-<gh_username>`. This will contain the starting project for your own work.

Important: You may have already cloned a similar starting repository from last year. **It is vital you use the above link to access the starting project.**

It is important that you follow the file/folder/executable structure 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. Where it is not specified, you can and should create new files, functions, classes and edit the `CMakeLists.txt` as you like.

Your code should be written inside `src`, as is typical of C++ projects, and you should note that there are two folders, `lib` and `app`. This is to simulate a real-life project where you are writing both a **library** (containing code another developer might want to use if they were developing their own Game of Life) and an **application** (containing code related to an application a user might use to run the Game of Life). All app-related code should go in `app` while all code that could be part of the library should go in `lib`. You should take care to chose an appropriate location for each piece of code you write.

CTest will be used to run all unit tests. You can use CTest by running `ctest` from inside the `build` directory as described in `PHAS0100Assignment1/README.md`. We have already provided two files containing sample tests in the `test` directory. You can add new tests inside the existing test files and these will be run without any other changes, however if you add an entirely new test file (i.e. a new `.cpp` file in `test`) you will have to edit `PHAS0100Assignment1/test/CMakeLists.txt` to ensure the new file is built into a test executable and exposed to CTest. Instructions are provided inside `PHAS0100Assignment1/test/CMakeLists.txt` indicating which lines need to be edited to add new test files.

When writing tests involving paths (e.g. when testing loading a grid from a specific file) CTest will run the test executables from the `build/test` directory. In order to use files stored in `test/data` you may hardcode relative paths as `../../test/data/glider.txt`, for example. If you are unused to using unix paths, `..` refers to the folder above the current working directory, so if the current working directory is `build/test`, `../../test/data/glider.txt` correctly refers to the file stored in `test/data`.

We recommend you use Visual Studio Code IDE with the supplied devcontainer running Ubuntu 22.04 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 22.04 and with g++ 11.3.0 (enforcing C++17) and CMake 3.22.1 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.

Reporting Errors

Contact the lecturer directly if you believe that there is a problem with the starting 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. Any corrections or clarifications made after the initial release of this coursework will be written in the errata section and announced.

Assignment: The Game of Life

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 that emerge in the Game of Life given particular initial conditions.

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.

1 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 the status of each cell in the grid. First familiarise yourself with 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. See section 2.1 for the precise rules we will be implementing.

1.1 Storing the grid

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 number of columns and rows in the grid as a constructor argument. The class should

- make use of an appropriate C++ standard library container to store the cell data
- have methods to print the current grid to screen
- have methods to get and set individual cell contents

You must 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 - -
- - - - -
```

When implementing the print function use the newline character `\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.

[5 marks implementation, 2 marks unit test, 7 total]

1.2 Initialising the grid at random

Next add functionality to your class to initialise the grid such that each cell's status is 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. 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]

1.3 Initialising the grid from a file

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/test/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.
- Don't assume the number of rows and columns are always equal.

Consider consistency checks and data validation for the input data. You may wish to use exceptions here to handle errors.

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

1.4 Fetching live neighbours

Add a method to your class that for a given cell (row, column) returns the number of neighbouring cells that are alive. In this assignment neighbours include those diagonally closest to a cell, so each cell has 8 neighbours. As an example, consider the cell in the centre of the same example as before, this cell has 3 neighbouring cells that are alive:

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

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]

2 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.

2.1 Implementing the game of life

a. Choose an appropriate name for your class and create the corresponding source and header files. **[1 mark]**

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

   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.

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 neighbours.
- 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]

2.2 Creating a command-line application

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

a. Create a skeleton command line application `gol_simulator` that compiles and can be run from the build directory as `./bin/golSimulator` **[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.

   The command line options must allow a number of user interactions:

   - In the case of the text file input the command line arguments should allow the user to specify the input file.
   - 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]

2.3 Running the application

Provide screenshots of the result of your application after 4 evolutions for the following cases:

1. After running with `glider.txt` input file.
2. After starting a 7 by 7 grid with random initial cell values (15 cells alive).

[1 marks each screenshot, 2 marks total]

3 Finding stationary patterns (15 marks)

One feature of the Game of Life is the presence of so called still lifes. These are stationary patterns that do not change despite stepping from one generation to another. In this part you will write a new command line app to find still lifes by comparing the current and next generation of cells and calculating if there has been no change between them.

a. Create 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 lifes. **[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 provide screenshots of at least two still lifes that your code found (2 marks). There may be still lifes that exist on a 4 by 4 grid with the specified boundary conditions which would not exist on a larger grid. These are considered still lifes for this assignment. See how many still lifes in total you can find and write up your result in the `README.md` (2 marks). **[4 marks]**

[a-d, 15 marks total]

4 Additional Considerations (15 marks)

4.1 Good practices

We expect you to engage with typical good practice shown throughout the course. Specifically you will be marked on:

- Appropriate naming for functions and classes.
- Good code organisation (i.e. appropriate choice of files and folders)
- Informative and useful git commit log.
- Effective in-code commenting.
- Code formatting.

Hints:

- See section 5
- Familiarise yourself with writing good git commit messages.
- Familiarise yourself with Best practices for writing code comments.
- You can use formatting tools like `clang-format` from inside VS Code to help format your code.

[10 marks]

4.2 User instructions

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

[5 marks]

5 Additional Hints

This section contains only extra, optional hints; there are no specific instructions or associated marks.

5.1 Code Layout

- Make sure all header (`.h`) files have an include guard to prevent accidental double declaration of variables/functions/classes. Alternatively you can use`#pragma once`; it's less typing but it is not part of standard so not guaranteed to be available for every compiler.
- Header files should only contain function/class/variable declarations (the interface) not their definitions (the implementation). Definitions should go in a corresponding source (`.cpp`) file; this keeps interface and implementation separate, speeds up compilation and avoids double definitions of functions/classes/variables.
- Choose a consistent formatting style (for example the google C++ style guide) and stick to it. The focus should be on readability.
- Use descriptive variable, function and class names that describe the purpose/intent of an object and would be understandable to someone reading the code for the first time.
- Be consistent with indentation. Use 2, 4, etc spaces and stick to it. Avoid tabs as these can show up differently on different editors. You can use automatic tooling to help you with that.
- Comments in the header files should describe what a class/function is for and how it should be used (unless it is obvious from the function name). Avoid stating the obvious or describing literally what the code does when commenting in the implementation (`.cpp`) source files. Use comments for *tricky, non-obvious, interesting, or important parts of your code* (from cppguide#Comments).

5.2 Unit Tests

- Each unit test `TEST_CASE` should at least one meaningful assertion to check a condition has been met. Avoid meaningless checks such as `REQUIRE(true)` that will always pass. To check that a class can be instantiated you can use `REQUIRE_NOTHROW` or `REQUIRE_THROWS` to check that an exception is not/is thrown as expected, for example:

```
TEST_CASE( "FileReader file reader instantiation" , "[FileReader]")
{
  REQUIRE_NOTHROW(FileReader("file_that_exists.txt"));
  // should throw an exception for non-existent file
  REQUIRE_THROWS(FileReader("file_that_does_not_exist.txt"));
}
```

- To improve coverage of your unit tests consider whether you have fully tested the behaviour of the object or function that you are unit testing:
  - Have you tested for known solutions that you can compare to?
  - Have you tested a range of representative use cases, not just one specific case?
  - Have you tested for failure (like invalid inputs where you expect the constructor to throw an exception) and edge cases (special cases like the square root of 0)?
- Use the `Approx` catch2 method when comparing floating point numbers, for example:

```
TEST_CASE( "Average", "[MathsUtils]" ) {
  // to compare within floating point precision
  REQUIRE(GetAverage(10.0,20.0) == Approx(15.0));
  // or to compare a result to within your own defined precision
```

```
  REQUIRE(GetAverage(10.1,10.2) == Approx(10.0).epsilon(0.02));
}
```

5.3 Commit Log

- Commit messages should describe the changes being made and have enough information that someone could come back and find out where a specific change was introduced
- Avoid bundling many changes into a single commit: this makes it harder to understand what has been changed in each commit and also harder to use tools such as `git bisect` to locate a change that may have introduced a bug
- Keep the first line of commit message concise (i.e. brief but with lots of info) and add more details as part of the message body if needed