Worksheet 4

COMP110: Principles of Computing

Ed Powley

January 2016

Introduction

In this assignment, you will create three small C++ programs:

A. A console application implementing the "terminal hacking" word-guessing game from the Fallout games;

B. A console application implementing the 2-player board game Connect 4;

C. A graphical application which generates and displays the Mandelbrot fractal.

This worksheet tests your ability to translate various program notations (pseudocode, flowcharts, mathematics, narrative descriptions) into C++ code.

Submission instructions

The GitHub repository at the following URL contains skeleton projects for the three parts of this worksheet.

`https://github.com/Falmouth-Games-Academy/comp110-worksheet-4`

Fork this repository into your own GitHub account. To submit, create a GitHub pull request.

For the most part, this worksheet requires you to edit C++ code in the skeleton projects provided. **Please do not rename or delete the projects or files provided, and please do not create new projects.**

Marking

Timely submission: 40%

The deadline for this worksheet is **12noon, Monday 8th February 2016**. To obtain the marks for timely submission, you must submit your progress towards all three parts

of the worksheet by this time. As with other worksheets, you may resubmit after these deadlines in order to collect extra correctness or quality marks. This 40% is awarded as long as you submit *something* for each part by the deadline, even if your submission has bugs or other issues.

Correctness: 30%

To obtain the marks for correctness, you must submit working solutions for the following:

- Part A, Sections 1–2;
- Part B, Sections 1–4;
- Part C, Sections 1–2.

Note that this is a threshold: the full 30% is awarded for work which is *complete* and contains *no clear errors*. In particular you will not be penalised for trivial errors which do not affect the overall functioning of your programs, nor will you receive extra credit for highly polished solutions.

Quality: 30%

The extra quality criteria for this worksheet are as follows:

1. **Presentation.** Your solutions are appropriately presented in GitHub, with descriptive commit messages and appropriate documentation in `readme.md` files. You have edited the provided skeleton projects, and refrained from renaming the provided files or creating new projects (except where instructed to do so to work on stretch goals). You have checked code into GitHub regularly whilst working on the worksheet.
2. **Part A stretch goal.** You have submitted a working solution for Part A, Section 3 (an improved word selection algorithm for the terminal hacking game).
3. **Part B stretch goal.** You have submitted a working solution for Part B, Section 5 (a rule variant for Connect Four).
4. **Part C stretch goal.** You have submitted a working solution for Part C, Section 3 (a generator for a fractal other than the Mandelbrot set).
5. **Sophistication.** Your solution for **any one** of the stretch goals is particularly sophisticated, demonstrating mastery of C++ programming concepts appropriate to the task at hand.

Figure 1: A screenshot of the hacking minigame in *Fallout 4*. Image from `http://fallout.wikia.com/wiki/Terminal`.

Part A.
Terminal Hacking

In this part, you will implement a version of the "terminal hacking" minigame from *Fallout 4* (Bethesda, 2015); see Figure 1. In this minigame you must guess a secret n-letter word, one of several options presented to you. On choosing an option, you are told the *likeness*: the number of letters which match the secret word (i.e. the same letter in the same position). For example if the secret word is `HOUSE` and your guess is `MOUSE`, the likeness is 4 out of 5. If your guess is `HOPES`, the likeness is 2 out of 5 (the letters `S` and `E` do not count as they are in the wrong positions).

The skeleton solution contains a project named `PartA_TerminalHacking` for you to build upon. This contains code to read words from a file, and choose the secret word and other words. You will implement the rest of the game.

1.

Algorithm 1 takes a guessed word and the secret word, and returns the likeness score as described above.

Implement the algorithm as a C++ function in `PartA_TerminalHacking.cpp`, choosing appropriate data types for the parameters, return value, and any variables.

Algorithm 1 An algorithm for calculating the likeness score for the terminal hacking minigame.

```
procedure GetLikeness(guessedWord, secretWord)
    result ← 0
    for i = 0, 1, ..., secretWord.length − 1 do
        if secretWord[i] = guessedWord[i] then
            increment result
        end if
    end for
    return result
end procedure
```

2.

Implement the main loop of the game, structured according to the flowchart shown in Figure 2.

3. Stretch goal

In the skeleton project, the words are chosen at random. This may lead to instances of the game which are unsatisfying, for example where all words have a low likeness score with respect to the secret word.

Design (as pseudocode) and **implement** (within your C++ project) an improved word choosing algorithm.

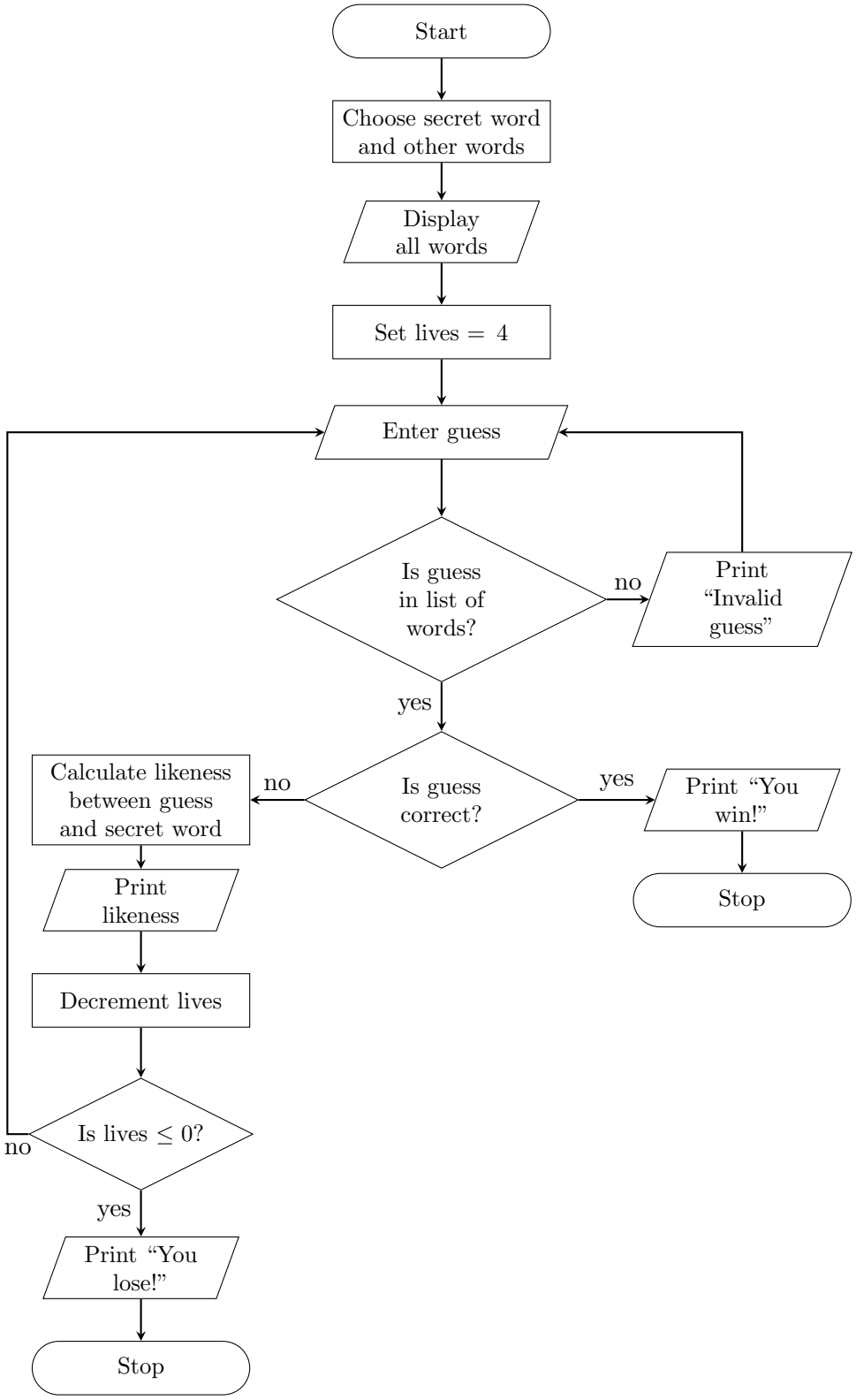

Figure 2: Flowchart for the Terminal Hacking game

Figure 3: A *Connect Four* board. The red player has won the game with a diagonal line starting in the lower left corner. Image from `https://commons.wikimedia.org/wiki/File:Connect_Four.jpg`.

Part B.
Connect Four

In this part, you will implement a version of the board game *Connect Four* (Milton Bradley, 1974); see Figure 3. This is a 2-player game, in which players take turns dropping coloured counters into the top of an upright grid. The counter falls into the lowest unoccupied space in the chosen column. The winner is the first player to make a horizontal, vertical or diagonal line of four counters of her colour.

1.

Implement the method `Board::performMove`. This method should simulate the player dropping a counter into the top of the given column `x`; i.e. it should determine the lowest unoccupied square (the largest value of `y` such that `getSquare(x, y)== 0`) and place the player's counter there. It should return `false` if the column is full, otherwise it should return `true`.

2.

Algorithm 2 checks for a single row of 4 counters on the board. The coordinates s_x, s_y give the starting point of the line, and d_x, d_y (which should both be -1, 0 or $+1$) give

Algorithm 2 An algorithm for checking the presence of a single line on the Connect Four board.

```
procedure CheckLine(board, s_x, s_y, d_x, d_y)
    if board[s_x, s_y] is empty then
        return false
    end if
    for i = 1, 2, 3 do
        t_x ← s_x + d_x × i
        t_y ← s_y + d_y × i
        if t_x, t_y is outside the bounds of the board then
            return 0
        else if board[t_x, t_y] ≠ board[s_x, s_y] then
            return 0
        end if
    end for
    return board[s_x, s_y]
end procedure
```

the direction. For example $s_x = 1, s_y = 2, d_x = 1, d_y = 0$ checks the line starting at position $(1, 2)$ and moving horizontally to the right.

Implement this algorithm as the method `Board::checkLine` in `Board.cpp`.

3.

Given the above function, it is possible to check the entire board for a winning line as follows. Loop through each square on the board. For each square, use CheckLine() to check for the following four lines:

- A horizontal line ($d_x = 1, d_y = 0$)
- A vertical line ($d_x = 1, d_y = 0$)
- A diagonal line ($d_x = 1, d_y = 1$)
- A diagonal line in the other direction ($d_x = 1, d_y = -1$)

If CheckLine() returns true for any of these, then the winner is the player who owns the square currently being considered. If CheckLine() never returns true for any square on the board, then there is no winner.

Implement this algorithm as the method `Board::checkWin` in `Board.cpp`.

4.

Implement the main loop of the game, structured according to the flowchart shown in Figure 4.

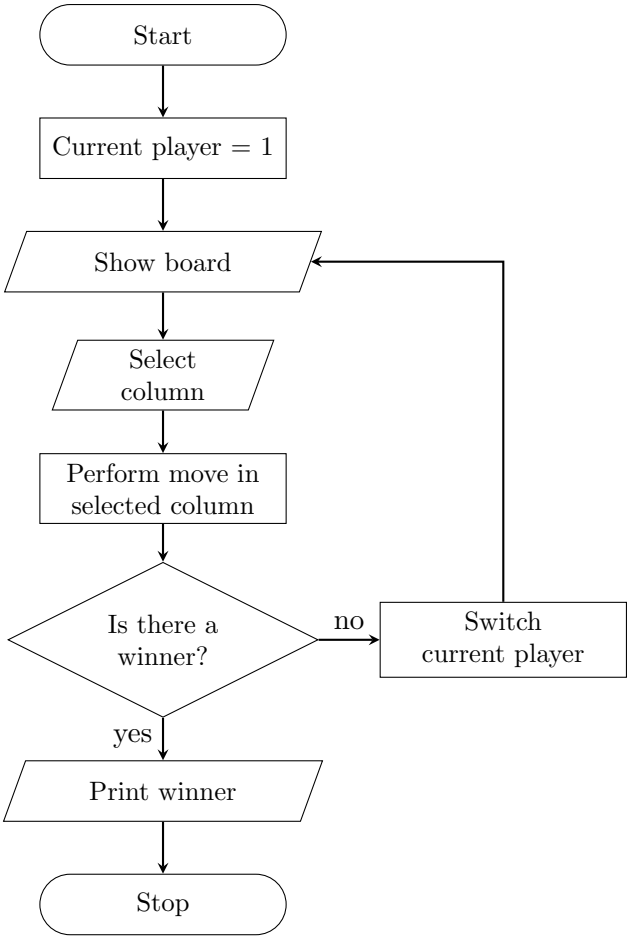

Figure 4: Flowchart for the Connect 4 game.

5. Stretch goal

The Wikipedia page for Connect Four lists several variants of the game with modified rules (`https://en.wikipedia.org/wiki/Connect_Four#Rule_variations`).

Implement a new program, based on your Connect Four program, implementing a rule variant of your choice.

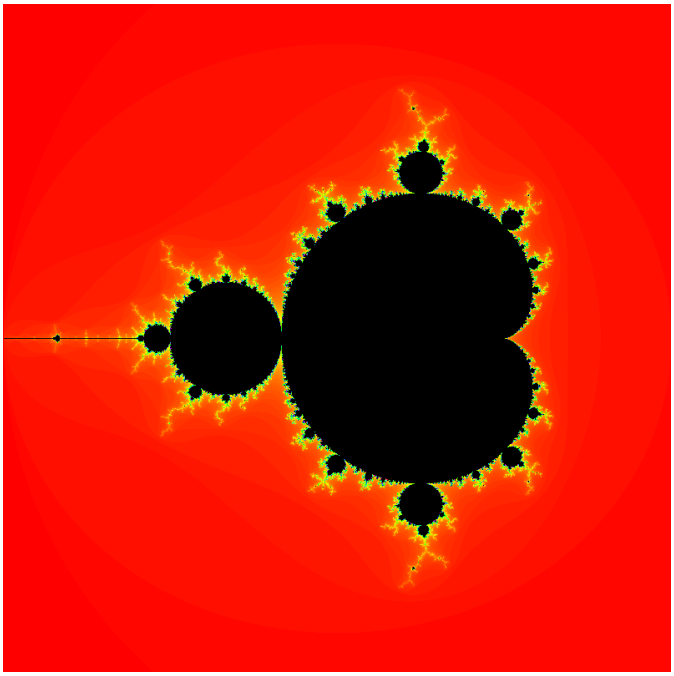

Figure 5: The Mandelbrot set fractal.

Part C.
Mandelbrot

In this part, you will use the CImg library (`http://cimg.eu/`) to write a program to generate and display the *Mandelbrot set* fractal; see Figure 5. This fractal colours each pixel of the image according to an iterated mathematical formula, as described below.

1.

To generate an interesting fractal, the on-screen x and y coordinates must first be rescaled. In the skeleton project the pixel coordinates range from 0 to 800, whereas the Mandelbrot set fractal is most interesting in the region $-2 \leq x \leq 1$ and $-1.5 \leq y \leq 1.5$.

Let p_x be the x coordinate of the pixel. This can be remapped into the range $x_{\min}$ to $x_{\max}$ using the following formula:

$$x_0 = \frac{p_x}{\text{image.width}} \times (x_{\max} - x_{\min}) + x_{\min}$$

The y coordinate can be remapped using a similar formula.

Implement the above calculations for the x and y coordinates, at the indicated parts of `PartC_Mandelbrot.cpp`.

2.

The Mandelbrot set is based on the following sequence of numbers. Let x_0 and y_0 be the coordinates of a point in the image. Then the sequence $x_1, y_1, x_2, y_2, x_3, y_3, \ldots$ is defined[1] recursively for $i = 0, 1, 2, 3, \ldots$ by:

$$x_{i+1} = (x_i)^2 - (y_i)^2 + x_0$$
$$y_{i+1} = (2 \times x_i \times y_i) + y_0$$

The points are coloured according to the *smallest* value of i for which $(x_i)^2 + (y_i)^2 \geq 4$. If such a value of i is not found after a large number of iterations (for example $i = 200$), the pixel is coloured black.

Implement an algorithm which performs the above computation, determining the smallest value of i for which $(x_i)^2 + (y_i)^2 \geq 4$ and selecting the appropriate pixel colour. Implement the algorithm in `PartC_Mandelbrot.cpp` so that the program generates the Mandelbrot set fractal (Figure 5) when it is run.

3. Stretch goal

Mathematicians have discovered many other fractals besides the Mandelbrot set.

Implement a new program, based on your Mandelbrot program, to generate a different type of fractal which you have found online or in the academic literature.

[1] If you are familiar with complex numbers, you may notice that this is equivalent to $z_{j+1} = z_j^2 + z_0$, where $z_j = x_j + y_j i$.