CO2402 Advanced Programming with C++
Assignment 2021-2022: MONOPOL*ish*

Hand in Date:

Introduction

For this assignment, you will work towards implementing a simplified version of the classic board game Monopoly. This is not an interactive game you can play, but rather a simulation of two players taking turns over a set number of rounds. Play is to be automated according to a sequence of pseudo-random numbers - there is no artificial intelligence, and no user input. All output will be text based and directed towards the console – there are no graphical elements to this assignment.
If you are not familiar with the game Monopoly, you can find a description here.

I expect you to complete this project in your own time outside of scheduled labs.

- This is an **individual** project and no group work is permitted.
- Do not diverge from the assignment specification. If you do not conform to the assignment specification then you will **lose marks**. Ask for clarification if you are unsure!

Avoiding Plagiarism

- You will be held responsible if someone copies your work - unless you can demonstrate that have taken reasonable precautions against copying.

Learning Outcomes Assessed (see module descriptor for full list)

- Make an informed choice of implementation method for a given problem (e.g. procedural, console, event-driven, object-oriented etc.)
- Implement and document a structured program to meet a given specification
- Select and apply appropriate data structures and algorithms to a given problem

Deliverables, Submission and Assessment

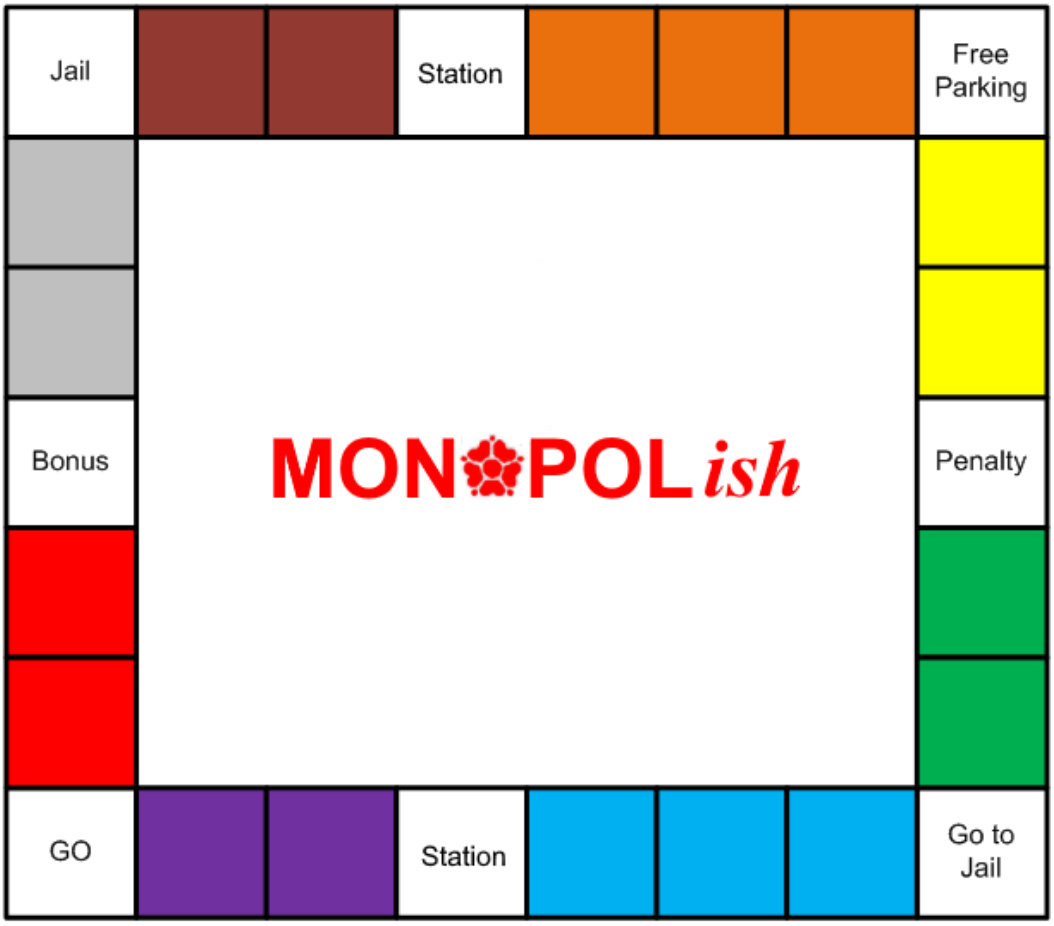
- There are **three** deliverables – your **source code**, a **report**, and a **video demonstration**
  - To get a mark above 0 you must submit all 3 deliverables.
- Submission is electronic, via Blackboard.
  - For the **source code**, upload **ONLY** the **.cpp** and **.h** files. DO NOT submit the solution file!
  - Make sure you include your name as a comment on the first line of all your source code files. Also include your name at the top of your **report**.
  - See the section on page 7 for a summary of what to include in your report and video.
- Assessment will be in-person in your scheduled lab during the week following the deadline.
  - Failure to attend the lab assessment will result in your mark being capped at 40.
- You may use whichever IDE you prefer to develop your solution. I am only interested in the source code and output.

Resources

- This assignment has an associated set of support files.
  - monopolish.txt – A text file for reading in the game configuration
  - random.cpp – Code to generate a sequence of pseudo-random numbers from a seed
  - seed.txt – A text file containing the random number seed for testing
  - code style guide
- A set of sample output files for the various levels of the assignment will be provided for you to compare your own results against.

Overview

Although there is no graphical element to the program, it may be helpful for you to think of the game as being played on a board comprising 26 **squares** arranged as shown below:



Each of the squares has a name (and some other information) which is contained within the data file monopoly.txt.

In your simulation, the game should have two players, each with a playing **piece**. Player 1 has a *hamster* playing piece, and player 2 has a *pumpkin*. At the start of the game, the pieces are placed on the "GO" square.

Players also have an amount of **money**. Each player starts the game with £1,500. During the game, players are allowed to have negative amounts of money – they do not go bankrupt unless you are aiming for a high first class mark.

A game is played as a series of **rounds**. During a round, each player takes one turn to roll a single die. You will represent rolling the die by generating a random number between 1 and 6 (using the code I have provided – see appendix). The player then moves their piece clockwise round the board by the number of squares indicated on the die.

Different things will happen according to the square landed on which may result in the player gaining or losing money. Exactly what happens on each square depends on how many marks you want…

The game will be played for 20 rounds. At the end of the game, the player with the most money is declared the winner.

Program Specification

You should implement the features described below **in order**. To be eligible for a mark within any classification, you must have at least *attempted* **all** the features for **all** the previous classifications.

Basic Scenario = bare pass (40% max)

If you implement only what is in this section and provide video evidence of it running to completion, you will get 40%. No more, no less.

- Create a class called CSquare to represent the squares on the board.
- Read in the data file Monopoly.txt (*see appendix on the last page*) and use it to set up an array of 26 CSquare objects.
    - You do not need to use pointers or dynamic memory allocation at this level.
    - Each line in the file represents one square.
        - The first number in each line represents the square's type, but for a bare pass, you can treat all squares the same.
        - Following the type number is the square's *name*. You will need to store the name as data in the CSquare objects as you create them.
- For each player you need to store their *name*, *money*, and *position* on the game board.
    - You could use variables for this, but use of a structure or class would be much nicer.

- In your main program, simulate playing 20 rounds of the game as follows:
- When the game is started, a welcome message is displayed. The format of the message is:
  `'Welcome to Monopol-ish'`
- At the start of each round, the round number should be output:
  `'round <round number>'`
- For each turn, the name of the player and the generated number is displayed. Use the description of the player's piece as their name (i.e. Hamster, Pumpkin). Output the message:
  `'<Player> rolls <number>'`
- On the next line the name of the player and the name of the square that the player landed on are displayed. Output the message:
  `'<Player> lands on <square name>'`
- When a player passes over or lands on GO, the player receives £200. You must output the message:
  `‘<Player> passes GO and collects £200'`
- When a player lands on any other square, nothing should happen.
- At the end of the game you should output the final amount of money each player has, and also which player has won. Output the following messages:
  `'Game Over'`
  `'<Player1> has £<amount>`
  `'<Player2> has £<amount>`
  `‘<Player> wins.’`

- Your code must follow the style guide precisely, and also be well commented throughout.

- When the code runs, there should be no user interaction. Specifically, there should be no need to press a key in-between rounds. You may wish to introduce such a feature to help with development/debugging, but it should be disabled in the code/demo you submit.

Pass Mark = third classification (40% +)

- All objects should now be stored in dynamic memory
    - Squares should be accessed via an array (or better still a vector) of *pointers*.

- Declare a class to implement the coloured squares on the board. These squares represent properties (Real estate).
    - A property square is *A Kind Of* square.
        - The property class should be derived from the square class.
    - In the Monopoly.txt file, rows beginning with '1' denote property squares. Set up the correct object types as you read in the file.
    - As well as their type and their name, properties also have a *cost*, and a *rent*.
        - They also have a *colour group*, but you can ignore the colour group at this stage.
    - The cost and rent (as found in the Monopoly.txt file) are given in the table below:

Property Name	Cost (£)	Rent (£)	Colour Group
Craggs Row	60	5	0 (Red)
North Street	80	10	0 (Red)
Maudland Road	100	15	1 (Grey)
Brook Street	120	15	1 (Grey)
Kirkham Street	140	20	2 (Brown)
Leighton Street	160	20	2 (Brown)
Bhailok Road	180	25	3 (Orange)
Adelphi Street	180	25	3 (Orange)
Moor Lane	200	25	3 (Orange)
Victoria Street	220	30	4 (Yellow)
Harrington Street	240	30	4 (Yellow)
Marsh Lane	260	35	5 (Green)
Hope Street	280	35	5 (Green)
Friargate	300	45	6 (Blue)
Corporation Street	300	45	6 (Blue)
Walker Street	320	45	6 (Blue)
Fylde Road	400	50	7 (Purple)
Ring Way	420	50	7 (Purple)

- When a player lands on a on a property square the following should happen:
    - If the property is not owned by anyone, the player who landed on the property will buy it *if and only if* they have a positive amount of money.
        - If purchased, the *price* of the property is deducted from the player's money (even if it means the player now has a negative balance) and the player becomes that property's owner.
    - If the property is already owned by the player who landed on it, nothing happens.
    - If the property is owned by another player, the player who landed on the property must pay its owner *rent*.
        - The rent value of the property is deducted from the player's balance and added to the owning player's balance.

- If the property is bought by the player on this turn, then output the message:
  `'<Player> buys <square name> for <cost>'`
- If the property is owned by another player then output the message:
  `'<Player> pays <rent>'`

- For all the other non-coloured squares (except GO – whose row in the file begins with '2' – and where the players collect £200) nothing should happen when a player lands on them.
  - In order to implement this you should treat all of the special squares as being already owned and with a rent of £0.

- At the end of each player's turn you must output the message:
- '<Player> has <current balance>'

Lower second classification (50% +)

- For this grade you need to attempt a polymorphic solution.
  - You will need a hierarchy of classes for the different square types.
  - When it comes to implementing the different behaviour of the squares, the functionality is devolved to the lower levels of the hierarchy:
    - You should have a collection of pointers of type CSquare*, but the bespoke implementation code should be written in the derived classes.
    - You should not call methods on the derived classes directly.
- You *must* use an STL container to store/access the CSquare pointers.

- Implement additional classes derived from CSquare for the Jail square, Go to Jail square and Free Parking square (types 6, 7 and 8 in the text file).

- If a player lands on the 'Jail' square after rolling the die, the player is considered to be 'just visiting', and nothing special happens. Output the message:
  ```
  '<Player> lands on Jail'
  ‘<Player> is just visiting'
  ```

- If a player lands on "Go to Jail" then their piece immediately moves to the "Jail" square and £50 is automatically deducted from their money. Output the message:
  ```
  '<Player> lands on Go to Jail'
  '<Player> goes to Jail'
  '<Player> pays £50
  ```
- The turn after a player has been moved to Jail is treated as a regular turn. The player has already paid to get out of jail. No special message is needed.

- If a player lands on "Free Parking" then nothing happens. Output the message:
  ```
  '<Player>lands on Free Parking'
  '<Player> is resting'
  ```

- Implement a further derived class for Station squares (type 3).
  - The rules for *buying* stations are the same as for buying properties.
    - The cost of buying a station is always £200
  - When landing on a station which is owned by another player, the owner should be paid a fee of £10. The following message should be output:
  `'<Player> pays £10 for ticket'`

Upper second classification (60% +)

- For this grade, the polymorphic aspect of your solution must be fully correct, and the polymorphism must be serving a genuine purpose.
    - You will not be awarded marks if you just plonk the keyword "virtual" into your code, whilst the work within your code is still done in a procedural fashion.
- Players must be implemented as classes (not variables in main or structs).

- Derive additional classes from CSquare to implement the Bonus and Penalty squares.
- If a player lands on either the Bonus or Penalty squares, the die must be rolled again and action taken according to the following table:

Roll	Bonus		Penalty	
1	Find some money.	Gain £20	Buy a coffee in Starbucks.	Lose £20
2	Win a coding competition.	Gain £50	Pay your broadband bill.	Lose £50
3	Receive a rent rebate.	Gain £100	Visit the SU shop for food.	Lose £100
4	Win the lottery.	Gain £150	Buy an assignment solution.	Lose £150
5	Sell your iPad.	Gain £200	Go for a romantic meal.	Lose £200
6	It's your birthday.	Gain £300	Pay some university fees.	Lose £300

- Output the following including the bonus / penalty message as given above, e.g.

```
'<Player> lands on Bonus'
'Find some money. Gain £20'
'<Player> has <current balance>'
```

- At this level there should be no global variables in your code. (Global constants are allowed.)
- There must be no memory leaks when you run your code.
    - Proof of this should be included in the document/video demonstration.

First classification (70% +)

- Properties are grouped by colour (as read in from the file).
    - The property rent is doubled if the same owner owns *all* of properties of a colour group.

- The implementation should make use of object-oriented methods throughout (not just the polymorphic parts). This will mean the implementation of several classes.
    - One of the classes must be the monopoly game itself, which will act as a manager class.
    - All your classes should have their own header file and source file.

- The STL syntax for vectors is ugly and cumbersome.
    - Use typedef and auto to clean up the ugly syntax of vector declaration.

- Use smart pointers such that you do not directly invoke dynamic memory allocation anywhere in your code.
    - Use the C++14 syntax to avoid the keyword new.
    - The only allowable use of raw pointers is if you need to implement an *observer*.

High First classification

- Implement the following extra rules about mortgaging properties and bankruptcy, to be applied at the end of each player's turn:
    - If a player's balance has fallen below zero, they must mortgage one or more properties until their balance rises above zero.
        - Properties should be mortgaged in value order, starting with the lowest valued property first.
        - When a property is mortgaged the player is lent the original value of the property.
    - If a player is unable to reach a value of zero or more by mortgaging *all* their properties, they are immediately declared bankrupt and lose the game.
    - If a player's funds go above the cost of their lowest valued mortgaged property, they must repay that mortgage in full.
        - Repaying a mortgage cannot result in a player having negative funds
    - A mortgaged property does not collect any rent.
        - Output appropriate messages when properties are mortgaged or redeemed, e.g. `'<Player> mortgages <property> for <amount>'`

- Add an additional two players to the game. Their playing pieces are *Dog* and *Jaguar*.
    - If, before the 20 rounds have finished, three of the four players are bankrupt, the last solvent player, having a monopoly, is declared the winner.

- There should be a finite amount of money in circulation (£20,000 including the each player's initial £1,500). The bank's reserves cannot be negative. If the bank runs out of money, players cannot receive £200 on passing GO, and cannot receive bonuses.

Report Contents

For any grade:

- Include the functionality checklist (which will be available on Blackboard) as the first page of your report. (You can use it as a report template if you like) This will tell me which grade level you are working towards, and which sample output I should compere your output to.
- Copy the entire console output of your program running from the seed I supplied. Use `Consolas 10pt` font for this.

For a grade above 50%:

- Include a UML class diagram.

For a grade above 60%

- Take a screen-shot of the Visual Studio Debug Output window showing evidence that your program has no memory leaks.

Video Demonstration Contents

The video should be short. You can either upload an .mp4 file to Blackboard with your submission, or else upload it to YouTube (or similar) and provide a link.

- It needs to begin by showing something on the screen that shows it is you logged in.
- It should then show your assignment **compiling** without errors and **running** to completion.
- The console output should be the same as that which is copied into your report. The point of the video is to demonstrate that you haven't faked it.
- There is no need for any commentary.

Appendix

Reading the "Monopoly.txt" file

For all of the scenarios the file can be read in a straight forward fashion.
In order to make reading the file straightforward a code number is used. At the beginning of each line there is a number which identifies the type of square stored on that line.
Use the code number in order to determine whether the data refers to a property, a Station, one of the special squares, etc.

Code	Meaning
1	Property
2	Go
3	Station
4	Bonus
5	Penalty
6	Jail
7	Go to Jail
8	Free Parking

The file will always be in the same format. You can assume that a Property has a name consisting of two words, followed by its cost, then rent, then group, etc.

Number generation (the dice throw)

There is a code file labelled "random.cpp" This implements a function called "Random". Random returns a randomly generated number in the range 1 to 6.
Random number generators only generate a pseudo-random sequence of numbers. If the generator is seeded with the same number then an identical sequence of numbers is generated.

- The seed for the random number generator is read from the file "seed.txt".
- The use of a seed will allow me to check your work against a known number sequence.
- I will supply examples of play using a particular seed. Seed the generator with the given number and the play will be exactly the same each time. You can use the examples of play to test your program.

The pound symbol

The pound symbol was not defined in the original ASCII code. Output can depend on the computer system. The following works on my machine:

```
const char POUND = 156;
```

If this doesn't work then try a value of 35 or 163. Substitute the hash symbol for the pound symbol if you are unable to get the pound symbol to output.