CO2402: Advanced Programming with C++: 2014-2015
Assignment: Monopoly

Panayiotis Andreou

Release Date: 10/12/2014

Hand in Date: 3pm 16/03/15 for file submission
Demonstration: 17/03/15 during the class

Introduction

For this assignment you must implement a simplified version of the game *Monopoly*. If you are not familiar with the game you can find a description here:
http://en.wikipedia.org/wiki/Monopoly_(board_game)

There are no visual elements to the program. The entire program runs using a console window. All output will be text based and directed towards the console window. There is no player input into the program. There is no artificial intelligence within the program. All play is automated.

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**. If you do want to make some addition to the game and you are unsure whether the change will break the specification then check it with me first.

The format of the program output is given in the specification. You must adhere to this format. I will provide some example output. You can therefore test your output against the expected output.

Plagiarism

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

Learning Outcomes

- 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
- Critically evaluate a computer program with regard to robustness, usability, maintainability, readability and efficiency

Deliverables

- **Assessment will be by demonstration in your usual lab session during the week of the hand-in** (i.e. the week beginning 17/03/14).
- Upload all of your files by 3pm, 16/03/14 to the module page on Blackboard containing:
    - Executable program of your solution
    - All the source and project files required to build the executable. The project must be set up to run in Visual Studio 2010 or 2013.

    - All of the other material.
- All files should be sensibly named and in working order
- The data file “Monopoly.txt” should be placed into the working directory so that I can just run the program.
- Note that the program includes a random number generator. The seed for the generator is read from the file "seed.txt". This file should be in your working directory and included with your program.
- A printed report about your program.
- Testing strategy and results.

Project Files

This assignment has three associated files:

- There is a data file. The file can be found inside the module page on Web-CT. The file is called "Monopoly.txt". The data file is an ASCII file containing data about the squares on the board.
- There is code which you must use to generate random numbers. The code is given in “random.cpp”.
- There is a style guide. The style guide can be found inside the module page on Web-CT. The file is called "style guide.doc".

Assignment notes

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 “Retail Park”, one of the special squares, etc.

Code	**Meaning**
1	Property
2	Go
3	Retail Park
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.

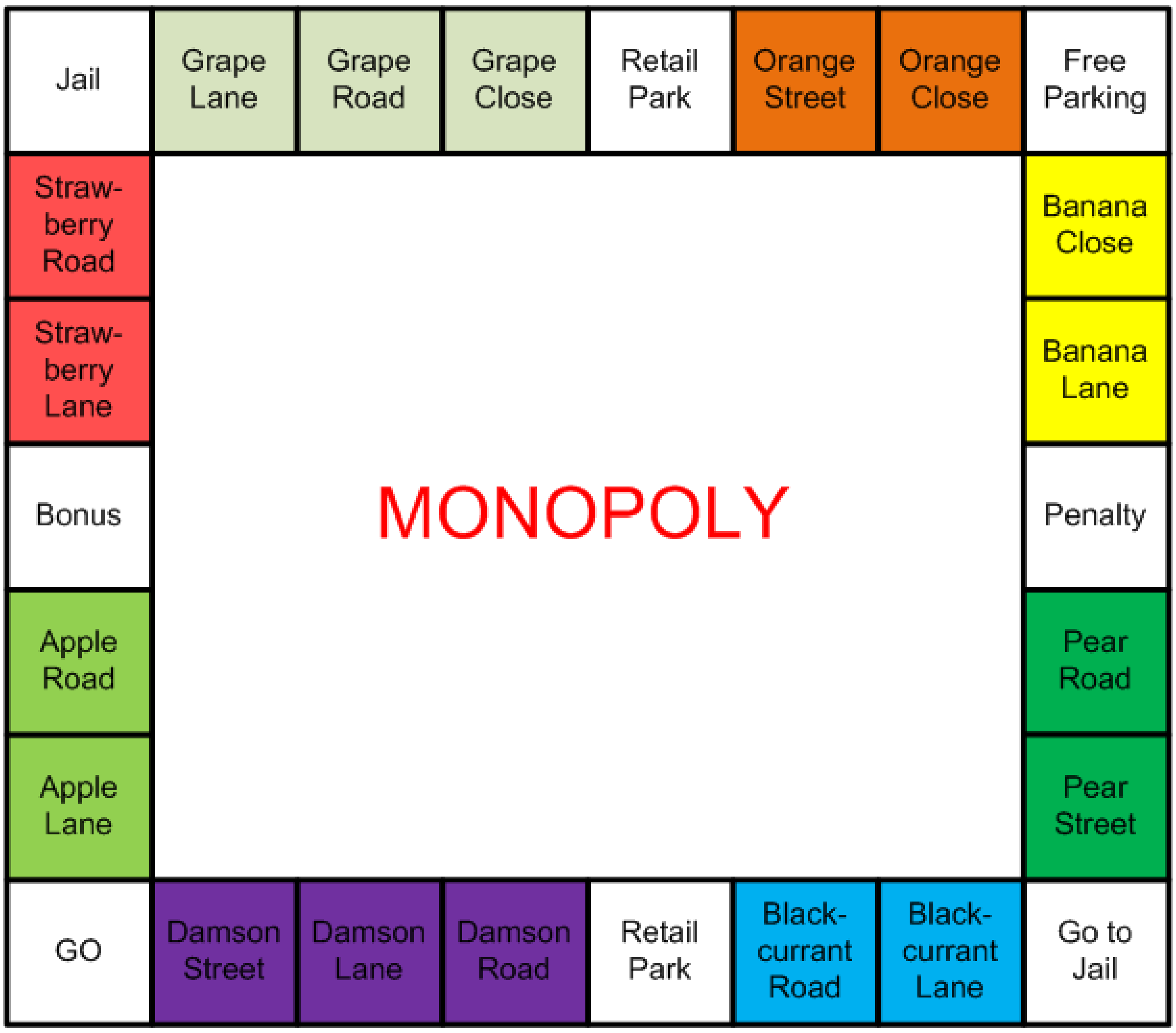
Report
- You need to write a report about the program. The report must be succinct and to the point. The report is not a story. The report is not a chronological record of the process of development. The *maximum* size of the report is 800 words (about two sides of A4). A single side of A4 would be sufficient with around 300 words.
- The report will state:
  - What was done in the final implementation of the program.
  - How you did what you did in the final implementation of the program.
  - What grade you expect to get

Testing
- You need to write a comprehensive and clear testing strategy which complements your program.
- The testing strategy should perform all of the possible sensible tests with your code.
- The testing strategy should clearly describe the test and the final test results.
- You do not need to include intermediate test results, i.e. from previous builds.

Code Style and Layout
- Your code needs to adhere to the style guide.
- The style guide can be found inside the module page on Web-CT.

Full Description

- You are required to implement a simplified version of the Monopoly board game. The game will be played on a board containing twenty-six squares laid out as shown in the Figure below.



- The game has two players. Each player has a piece. Player one has a dog and player two has a car. At the start of the game the pieces are placed on the "GO" square.
- When a player passes over or lands on GO, the player receives £200. When the player passes over or lands on GO you must output the message:
  '<Player> passes GO and collects £200'
- Each player receives £1500 at the beginning of the game. Consider the game to have an unlimited amount of money.
- Players do not go bankrupt – their balance can be negative.
- A game is played as a series of rounds. During a round, each player takes one turn. In each turn a number between 1 and 6 is generated (which represents the player throwing a six side dice). The player's piece moves clockwise around the board by a number of squares equal to the number generated.
- Each square has a name. There are 6 special squares: GO, Penalty, Jail, Free Parking, Bonus and Go to Jail. There are 18 property squares. The property squares are numbered from 1 to 18. The property squares are grouped into 6 groups. The

groups each have a different colour: brown, light green, red, blue, green, light blue, yellow and purple. There are two airports.

- The game is run as a simulation requiring no user input.
- Play the game for only 20 rounds.
- When the game is started, a welcome message is displayed. The format of the message is:
  'Welcome to Monopoly'
- For each turn, the name of the player and the generated number is displayed. 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 lands on a property the following happens:
  - If the property is not owned, the player who landed on the property will buy it if they have a positive amount of money. If purchased, the price of the property is deducted from the player's money and the player becomes its owner.
  - If the property is owned by the player that landed on it, nothing happens.
  - If the property is owned by another player, the player that landed on the property must pay its owner rent.
- When a player lands on an airport the following happens:
  - If the airport is not owned, the player who landed on the airport will buy it if they have a positive amount of money. The cost is always £200. If purchased, the price of the airport is deducted from the player's money and the player becomes its owner.
  - If the airport is owned by the player that landed on it, nothing happens.
  - If the airport is owned by another player, the player that landed on the airport must pay the owner a fee of £10.
- Players always buy un-owned properties and airports they land on if they have a positive amount of money.
- If the property is un-owned then output the message:
  '<Player> buys <square name> for <cost>'
- If the property is owned then output the message:
  '<Player> pays <rent>'
- If the airport is owned then output the message:
  '<Player> pays £10 flying costs'
- If the square is a special square and special squares are not being implemented then output the message
  '<Player> pays £0'
- The cost and the rent of the properties are given in the following table:

Property Name	**Cost**	**Rent**	**Group**
Apple Lane	60	5	1
Apple Road	80	10	1
Strawberry Lane	100	15	2
Strawberry Road	120	15	2
Grape Lane	140	20	3
Grape Road	160	20	3
Grape Close	180	25	3
Orange Street	190	25	4

Orange Close	200	25	4
Banana Close	220	30	5
Banana Lane	240	30	5
Pear Road	260	35	6
Pear Street	280	35	6
Blackcurrant Lane	300	45	7
Blackcurrant Road	300	45	7
Damson Road	400	45	8
Damson Lane	400	50	8
Damson Street	420	50	8

- The table data is given in “Monopoly.txt”.
- At the end of each player’s turn you must output then message:
  '<Player> has <current balance>’

Special Squares

- The special squares are not used in the basic scenario.
- If a player lands on “Go to Jail” then their piece immediately moves to the “Jail” square and £50 is automatically deducted. Output the message:
  '<Player> lands on Go to Jail'
  '<Player> goes to Jail'
  '<Player> pays £50
- If a player lands on the 'Jail' square during a normal roll, the player is considered to be 'just visiting', and nothing special happens. Output the message:
  '<Player> lands on Jail'
  <Player> is just visiting'
- 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'
- If a player lands on "GO" then they collect £200 as normal, but nothing else happens. Output the message:
  '<Player> lands on GO'
- If a player lands on “Penalty” then one of the 6 following random events occurs:
  - Pay food bill. Player loses £20.
  - Pay phone bill. Player loses £50.
  - Pay heating bill. Player loses £100.
  - Pay vehicle tax. Player loses £150.
  - Pay fuel bill. Player loses £200.
  - Pay windfall tax. Player loses £300.
- Output the following including the penalty message as given above, e.g.
  '<Player> lands on Penalty'
  'Pay food bill. Player loses £20'
  '<Player> has <current balance>'
- If a player lands on “Bonus” then one of the 6 following random events occurs:
  - Find some money. Player gains £20.
  - Win competition. Player gains £50.
  - Tax rebate. Player gains £100.
  - Win lottery. Player gains £150.

    - Bequest. Player gains £200.
    - Birthday. Player gains £300.
- Output the following including the bonus message as given above, e.g.
  '<Player> lands on Bonus'
  'Find some money. Player gains £20'
  '<Player> has <current balance>'

Overview of the marking scheme for the program

Bare pass mark

- You may implement a simplified version of the assignment to obtain a bare pass mark of 40%. If you follow this route then you cannot gain more than a pass mark of 40%.
- File read of the 'Monopoly.txt' file in order to set up the square class.
- Implement one or more classes.
- Use an array to store the squares. You do not need to use pointers or dynamic memory allocation.
- A player should collect £200 if they pass or land on "GO".
- Players move around the board. Output the name of the player and the square they've landed on. When a player lands on a square output the message:
  '<Player> has landed on <square name>'
- Ignore all of money actions for all of the other squares, e.g. there is no payment of rent, buying of properties, special squares or whatever.

Pass mark

- Implementation of the basic scenario, e.g. no use of polymorphism.
- File read of the data file in order to set up the square class.
- Use of a class to implement the properties of the board. You only have actions for the properties.
- The only other square you should implement is "GO"; ignore all of the other squares. If a player lands on any other square then nothing happens. In order to implement this you should treat all of the special squares as being already owned and with a rent of 0.
- A player should collect £200 if they pass or land on "GO".
- Implementation of the properties using an array of pointers or a vector of pointers.

Lower second classification

- Attempt to meet the upper second classification but with significant errors or elements missing.
- For example, development of part of the full scenario that successfully implements at least two types of square (property and Jail) using polymorphism.

Upper second classification

- Implementation of the full scenario (except for the doubling of rent). The full scenario introduces the special squares and property groups.
- File read of the data file in order to set up the game board data.
- Implement all of the special squares.
- Use of classes.
- Use of a STL container class to store the square class, e.g. use of the vector class.
- Use of polymorphism in order to implement all of the squares.
- Note: polymorphism should be implemented for a reason. The polymorphic solution is

a natural implementation of the hierarchical structure of the scenario. You will not be awarded marks in this category if you just plonk the keyword "virtual" into your code whilst the work within your code is still done in a procedural fashion. If you don't understand this, then ask!

First classification

- Implementation of the upper second classification.
- Properties are grouped by colour. 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. This will mean the implementation of several classes.
- One of the classes must be the monopoly game itself. The monopoly game will act as a manager class.
- The monopoly game class should be implemented as a singleton.
- Expand the game so that 2 to 6 players can compete.
- The player must be implemented as a class.

Marking scheme

Bare pass mark	**Available marks**	**Result**	**Comments**
Implementation and report	**40 only**	**pass**	

Pass mark	**Available marks**	**Result**	**Comments**
Basic scenario	35	/ 35	
Lower second		**Result**	
Full scenario but with significant errors or elements missing.	15	/ 15	
Upper second		**Result**	
Full scenario	10	/ 10	
First		**Result**	
Full scenario with the addition of complete object-orientation, doubling of rent, 2-6 players, manager class, etc.	10	/ 10	
Report		**Result**	
Quality	10	/ 10	
Testing		**Result**	
Quality	10	/ 10	
Code Style and layout		**Result**	
Conformance to the style guide. Use of good code layout. Commented throughout.	10	/ 10	

Overall mark

Comments
If you lose marks for any reason then this will be stated here.

Regulations

The coursework regulations can be found on Blackboard in the “Computing Noticeboard” in the course handbook.