RMIT UNIVERSITY

School of Science

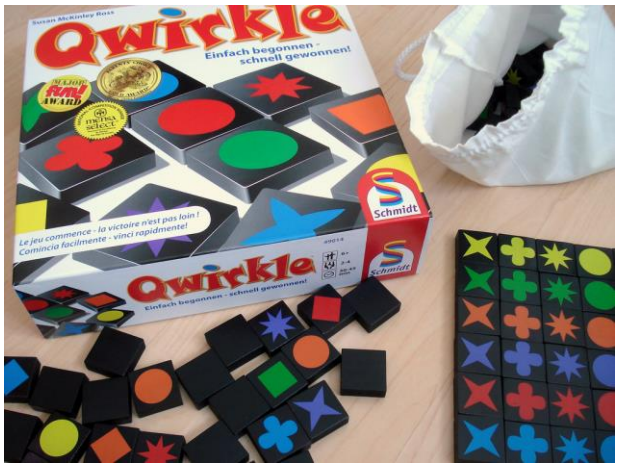
COSC1076 Advanced Programming Techniques

Assignment 2

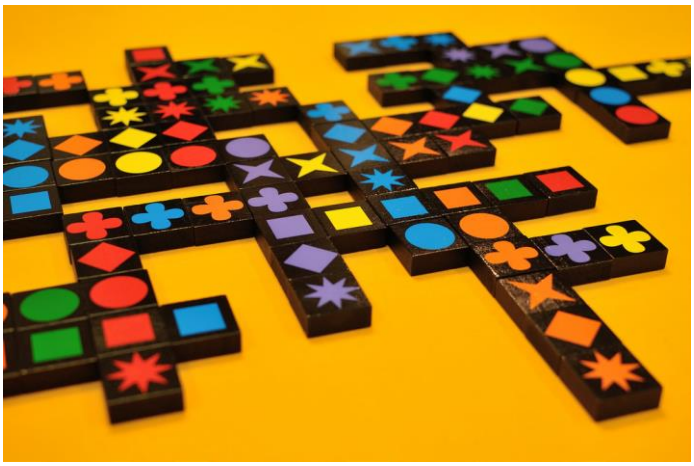
	Assessment Type: Group assignment; (4 members from same tute). Submit online via Canvas→Assignments→Programming Assignment 2. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
	Due dates: Group Registrations: Week 7 lab Milestone 1: 23:59pm, 25/Sep/2019; Milestone 2-3: 23:59pm, 9/Oct/2019 Progress Updates: During weekly labs Presentation & Marking: Week 12, by registered time slot Deadlines will not be advanced but they may be extended. Please check Canvas→Syllabus or via Canvas→Assignments→Programming Assignment 2 for the most up to date information. As this is a major assignment in which you demonstrate your understanding, a university standard late penalty of 10% per each working day applies for up to 5 working days late, unless special consideration has been granted.
	Weighting: 25 marks

1.1 Overview

In this assignment you will implement a 2-player text-based version of the Board Game **`Qwirkle`**.



(a) Qwirle box and pieces


(b) Example game state

For an explanation of the rules and gameplay:

- TableTop rules explanation and full game: https://youtu.be/Hp3IwPbZYSE?t=60
- Rules: Available on Canvas.

However, this assignment will use a modified version of the rules, detailed in Section 2.3.

In this assignment you will:

- Practice the programming skills covered throughout this course, such as:
  - ADTs
  - Linked Lists
  - Pointers
  - Dynamic Memory Management
  - File Processing
  - Program State Management
  - Exception Handling
- Practice the use of testing
- Implement a medium size C++ program:
  - Use features of C++14
  - Use elements of the C++ STL
- Work as a team
  - Use group collaboration tools

This assignment is divided into four Milestones:

- Milestone 1: Test Cases, to be developed to ensure your Qwirkle implementation is correct.
- Milestone 2: A fully functioning implementation of the base Qwirkle gameplay, which pass Milestone 1 tests.
- Milestone 3: Enhancements (minor & major) to the basic implementation. This may *only* be commenced if Milestone 2 is fully functional and error-free.
- Milestone 4: Written report (no more than 4 pages) analysing the design and implementation of your software, and the use of your test cases.

In additional, you will provide regular updates on your progress in this assignment to your tutor during your weekly lab classes.

1.2 Group Work

This assignment will be completed groups of 4. All members of the group **must** be from the same Lab. Your group must be registered on Canvas, by your Week 7 Lab.

If you are unable to find a group, please discuss this with your lab demonstrator **as soon as possible and before** the group registration deadline.

If at any point you have problems working with your group, please inform your lab demonstrator as soon as possible, so that any issues may be resolved. The weekly progress updates should assist in communicating the progress of your group work.

1.3 Relevant Lecture/Lab Material

To complete this assignment (especially some of the enhancements), you will requires skills and knowledge from lecture and lab material for Weeks 7 to 10 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Note that grade for your group work requires **consistent** work throughout all weeks. Thus, do the work you can initially, and continue to build in new features as you learn the relevant skills.

1.4 Start-up Code

On Canvas you will find start-up code to help you get running with the assignment. This code includes:

- Header file definitions for common gameplay aspects
- Example Test Case

The use of the start-up code is described in Section 6.3.

2. Learning Outcomes

This assessment is relevant to the following Learning Outcomes:

1. Analyse and Solve computing problems; Design and Develop suitable algorithmic solutions using software concepts and skills both (a) introduced in this course, and (b) taught in pre-requisite courses; Implement and Code the algorithmic solutions in the C++ programming language.
2. Discuss and Analyse software design and development strategies; Make and Justify choices in software design and development; Explore underpinning concepts as related to both theoretical and practical applications of software design and development using advanced programming techniques.
3. Discuss, Analyse, and Use appropriate strategies to develop error-free software including static code analysis, modern debugging skills and practices, and C++ debugging tools.
4. Implement small to medium software programs of varying complexity; Demonstrate and Adhere to good programming style, and modern standards and practices; Appropriately Use typical features of the C++ language include basic language constructs, abstract data types, encapsulation and polymorphism, dynamic memory management, dynamic data structures, file management, and managing large projects containing multiple source files; Adhere to the C++11/C++14/C++17 ISO language definition and features.
5. Develop skills for further self-directed learning in the general context of software engineering and computer science, including decision-making about guided design of software; Adapt programming experience and language knowledge to and from other programming language contexts.
6. Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.

3. Base Program Gameplay & Functionality

The base Qwirkle program implements a 2-player text-based version of Qwirkle, using a reduced rule-set. In the base game, the players take turns placing tiles from their hand onto the board. The rule changes for the base Qwirkle game are described in Section 3.5. In addition, the game has been modified from placing tiles horizontally and vertically, to **diagonally**.

This section details the behaviour of the base Qwirkle program. What is presented in this spec is a description of the main functionality of your Qwirkle program. Some parts are left open for you to decide the best course of action.

This spec does not give the rules of Qwirkle. Canvas contains a link to the rules.

!	**Aspects of this specification are flexible and open to your interpretation. In general, where there is flexibility, it is up to you to determine the best course of action. You may ask questions on the forum for clarity. Make sure that your tests are written to ensure your program works correctly based on any decisions you make.**

3.1 Launch

Your base Qwirkle program will be run using the following terminal command:

```
$ ./qwirkle
```

On launch, the program should display a welcome message:

```
Welcome to Qwirkle!
-------------------
```

Following the welcome message, the program should continue to the **main menu**.

3.2 Main Menu

The main menu shows the options of your Qwirkle program. By default there should be 4 options. The menu is followed by the **user prompt**.

```
Menu
----
1. New Game
2. Load Game
3. Show student information
4. Quit

>
```

The user selects an option by typing a number, and pressing enter. Each menu option is described below. The user prompt is described in Section 3.4, including what to do for invalid input.The

3.2.1 New Game

The program should:

1. Print a message for starting a new game
2. Ask for the player names
3. Create a new game of Qwirkle,
4. Proceed with normal gameplay.

As an overview, this process may look like:

```
> 1

Starting a New Game

Enter a name for player 1 (uppercase characters only)
> <user enters name>

Enter a name for player 2 (uppercase characters only)
> <user enters name>

Let's Play!
<normal gameplay continues from here>
```

The players should only consist of letters (no numbers or symbols). Your program should validate (check) that the player name is valid.
The Qwirkle gameplay is described in Section 3.3. Make sure you take note of the requirements for starting a new game, described in Section 3.3.10.

3.2.2 Load Game

The program should first ask the user for a filename from which to load a game.

```
> 2

Enter the filename from which to load a game
>
```

The user enters the relative path to the saved game file, and presses enter.

After the filename is provided, the program must then conduct two validation checks:

1. Check that the file exists.
2. Check that the format of the file is correct. The format for saved games is described in Section 3.3.7.

If the filename passes both checks, the program should print a message, then load the game as described in Section 3.3.12, and continue with normal gameplay as described in Section 3.3.

```
> <filename>

Qwirkle game successfully loaded
<game play continues from here>
```

3.2.3 Show student information

The program should print the name, student number, and email address of each student in the group, separated by new lines. Note that you should replace **<full name>**, **<student number>** and **<email address>** sections with your full name, student number and student email address.

After printing the student details, the program should return to the main menu.

```
> 3

---------------------------------
Name: <full name>
Student ID: <student number>
Email: <email address>

Name: <full name>
Student ID: <student number>
Email: <email address>

<Student 3, etc.>
---------------------------------

<main menu>
```

3.2.4 Quit

The program should print a goodbye message, and safely terminate **without** crashing.

```
> 4

Goodbye
```

3.3 Base Gameplay

During base Qwirkle gameplay, the 2 players take turns placing tiles from their hand onto the board.

At the start of the player's turn, the program should show (in order):

1. The name of the current player
2. The scores of both players
3. The state of the board
4. The tiles in the current player's hand
5. The user prompt

The current player may then take one of two actions:

1. Place a tile onto the board
2. Replace on tile in their hand

Once the player successfully takes their action, their turn ends, and the other player's turn starts.
Alternatively, the player may perform one of two game functions:

1. Save the game to a file
2. Quit the game

Below is an example of a sequence of actions and game functions for two players, named **A** and **B**. The next sub-sections, describe the individual aspects of the base gameplay.

```
< previous output >
A, it's your turn
Score for A: 6
Score for B: 6

     0    2    4    6
   ------------------------
A  |    |    | B5 |    |
B    |    | B6 |    |    |
C  |    | B4 |    |    |
D    |    | R4 | Y2 |    |
E  |    |    | Y4 |    |
F    |    | Y1 | P4 |    |
   ------------------------
          1    3    5    7

Your hand is
Y5,G5,R5,O2,B1,P6

> place G5 at B5

B, it's your turn
Score for A: 8
Score for B: 6

     0    2    4    6
   ------------------------
A  |    |    | B5 |    |
B    |    | B6 | G5 |    |
C  |    | B4 |    |    |
D    |    | R4 | Y2 |    |
E  |    |    | Y4 |    |
F    |    | Y1 | P4 |    |
   ------------------------
          1    3    5    7

Your hand is
P2,P3,O6,G1,Y4,B2

> replace G1
< gameplay continues >
```

3.3.1 Tile Codes

Tiles are represented by a 2-character code:

`<colour><shape>`

To make it easier to visually differentiate tiles, colours are represented by letters, and shapes are represented by integers. The codes are given in the table below

Colour	Colour Code		Shape	Shape Code
Red	R		Circle	1
Orange	O		4-Star	2
Yellow	Y		Diamond	3
Green	G		Square	4
Blue	B		6-Star	5
Purple	P		Clover	6

For example, the **Yellow Square** tile is represented by the code: **`Y4`**

!	**The Start-up code provides useful #define statements for these codes.**

3.3.2 The Board

The display of the game board consists of two features:

1. Tile Display
2. Grid Co-ordinates

The board is a 2D grid of tiles, up to a maximum size of 26x26. However, some locations of the 2D grid may be empty, meaning that no tile has been placed there. When the board is displayed, all locations that contain a tile are filled with the tile code, and empty locations filled with two-spaces.

!	**The examples is this spec use an expandable board (see the Minor Enhancements in Section 5.1). That is, the board is only as large enough as necessary, and expands as the players add tiles.** **For the base game-play you may use a fixed size board.**

The grid co-ordinates use:

- Uppercase Letters for rows
- Integers for columns

Board locations are always referenced in row-column fashion:

`<row><column>`

For example, the blue square is at grid co-ordinate C2, and the yellow circle is at grid co-ordinate F3.

```
    0    2    4    6
  ------------------------
A |    |    | B5 |    |
B    |    | B6 | G5 |    |
C |    | B4 |    |    |
D    |    | R4 | Y2 |    |
E |    |    | Y4 |    |
F    |    | Y1 | P4 |    |
  ------------------------
       1    3    5    7
```

As a hint, you could store the board as:

- A 2D array of Tiles
- A vector of vectors of Tiles

3.3.3 The Player's Hand

The player's hand is an ordered **linked list** of tiles.

> **!** **In your implementation you must use a linked list to store the tiles in the player's hand. You must implement your own version of a Linked List.**

The player's hand is displayed as a comma separated list of tiles.

```
Your hand is
Y5,G5,R5,O2,B1,P6
```

The order of tiles in the player's hand is important for testing purposes!

- When **adding** a tile, it is always added at the end of the list.
- When removing a tile, the **remaining** tiles stay in the **same order**.

3.3.4 The Tile Bag

The tile bag, contains the rest of the tiles that are not on the board or in player's hands. The tile bag must be stored as an ordered **linked list**. The contents of the tile bag is **never** displayed to the users. However, the contents of the tile bag is stored in the saved game file.

> **!** **In your implementation you must use a linked list to store the tiles in the tile bag. You must implement your own version of a Linked List.**

The order of the tile bag is determined when generating a new game. When a tile is drawn from the bag, it is taken from the **front** of the **linked list**. If tiles are added to the bag, they are added to the **end of the linked list**.

3.3.5 Player Action: Place a Tile

The current player may place a tile onto the board using the command:

```
place <tile> at <grid location>
```

The command contains two elements:

1. A tile to place
2. The grid location to place the tile

For example, using the above hand and board, if the player performs:

```
> place G5 at B5
```

This results in the board:

```
     0    2    4    6
   ------------------------
A  |    |    | B5 |    |
B     |    | B6 | G5 |    |
C  |    | B4 |    |    |
D     |    | R4 | Y2 |    |
E  |    |    | Y4 |    |
F     |    | Y1 | P4 |    |
   ------------------------
          1    3    5    7
```

After the command is given, the program must:

1. Check that the command is correctly formatted.
2. Check that the placement of tile is legal according to the rules of Qwirkle.

If the player's action is **legal**, the program should:

1. Place the tile onto the board
2. Update the player's score
3. Draw a replacement tile from the tile bag and add it to the player's hand, if there are available tiles
4. Continue with the other player's turn

3.3.6 Player Action: Replace a Tile

The current player may replace one tile in their hand using the command:

replace <tile>

For example, using the above hand the player may take the following replace action:

```
> replace P6
```

After the command is given, the program must:

1. Check that the command is correctly formatted.
2. Check that the tile is in the player's hand.

If the player's action is legal, the program should:

1. Remove the tile from the players hand and place it in the tile bag. (If the player has two tiles with the same code, the first tile in the list should be replaced)
2. Draw a new tile from the tile bag and add it to the player's hand
3. Continue with the other player's turn

3.3.7 Function: Saving the Game

The current player may save the game to a file using the command:

save <filename>

The program should save the current state of the game to the provided filename (overwriting the file if it already exists). Then the program should display a message and continue with the gameplay. The current player does not change, so that a player may save the game and then take a turn.

```
> save savedGame

Game successfully saved
>
```

If the program has problems saving the file, it should display a message, and continue with normal gameplay **without crashing.**

The format of the saved file is as given below. Each item is saved on a new line.

```
<player 1 name>
<player 1 score>
<player 1 hand>
<player 2 name>
<player 2 score>
<player 2 hand>
<board>
<tile bag contents>
<current player name>
```

The format for each of the items is:

- Name: ASCII text
- Score: Integer
- Player hand and tile bag: comma separated ordered list
- Board: Same as printed when showing the game board, including the co-ordinates

For example, if the game in Section 2.3 was saved the saved game file will look like:

```
A
8
Y5,R5,O2,B1,P6,Y3
B
6
P2,P3,O6,Y4,B2,O3
     0    2    4    6
   ------------------------
A  |    |    | B5 |    |
B    |    | B6 | G5 |    |
C  |    | B4 |    |    |
D    |    | R4 | Y2 |    |
E  |    |    | Y4 |    |
F    |    | Y1 | P4 |    |
   ------------------------
         1    3    5    7
P2,B5,.....
A
```

3.3.8 Function: Quit

The program should quit **without crashing**, as per the instructions in Section 3.2.4.

3.3.9 Special Operation: QWIRKLE!

If the player scores a Qwirkle (see the game rules) on their turn, then the program should print out an additional message, before displaying the game information. Remember to update the player's score accordingly.

```
> place .. at ..

QWIRKLE!!!

<display game>

>
```

3.3.10 Special Operation: Starting a New Game

When a new game is started, a special sequence of operations must be conducted:

1. Create the ordering for the tile bag
2. Set up the initial player hands
3. Start with an empty board, with player 1 as the starting player

You will need to devise **your own algorithm** to "shuffle" the bag of tiles to create a "random" initial order. This is left up to your own invention. The lectures will talk about randomness in C++ programs.

Then the initial tiles are added to the player's hands. 6 tiles are drawn from the tile bag and placed in the 1st player's hand. Then 6 tiles are drawn from the tile bag and placed in the 2nd player's hand.

Finally, the board starts with no tiles placed, so that when displayed, it should be empty.

3.3.11 Special Operation: Ending a Game

The game ends when:

1. The tile bag is empty, and
2. One player has no more tiles in their hand

If the game ends, the program should:

- Display the end game message
- Display the scores
- Display the name of the winning player
- Then quit, according to Section 3.2.4.

For example:

```
Game over
Score for <player 1 name>: 000
Score for <player 2 name>: 000
Player <winning player name> won!

Goodbye
```

3.3.12 Special Operation: Loading a Game

To load a game from a saved game file, the program should read the contents of the saved game file, and update all data structures in the program using the information in the saved game file. See Section 3.3.7 for the format of the saved game file. Specifically, the program should take note of:

- The player's name and scores
- The tiles in each players hand
- The state of the board
- The order of the tiles in the tile bag
- The current player - the next player to take a turn

Once the game has been loaded, gameplay continues resumes with the current player.

3.4 User Prompt

The user prompt is displayed whenever input is required from the user. It is a greater-than symbol (>), followed by a space. It is assumed that all user inputs are provided as a single line of input.

When shown the prompt, the user should see in their terminal window:

```
> 
```

If at any point the user enters **invalid input**, or the validation checks of the input fail (see each section) then the program should print **`Invalid Input`** and re-show the prompt.

```
> qwerty
Invalid Input
> 
```

3.4.1 EOF Character

If an any time the user enters the `EOF` (end-of-file) character, the the program should Quit, following the procedure in Section 3.2.4. That is:

```
> ^D

Goodbye
```

!	**This behaviour with the EOF character is necessary to ensure your program terminates at the end of every test case.**

3.5 Rule Changes

For the base Qwirkle implementation, the following rules have been modified:

- A New game always begins with Player 1 and an empty board
- Tiles are placed **one at a time**.
- Players can only replace one tile at a time
- Maximum board size of 26x26 (note that the board shown in the examples in section 3.3 is considered 8 x 6)
- There are only 2 tiles of each type (rather than 3 of each type). This makes it highly likely the game will fit within the 26x26 board, and reduce the complexity of your test cases.

!	**For your Milestone 3 enhancements, you may restore some or all of these rules to their original form.**

4 Deliverables

4.1 Mandatory Requirements

As part of your implementation, you **must**:

- Implement your own Linked List
- Use your Linked List implementation to store the player's hands and the tile bag.

!	**If you fail to comply with these mandatory requirements, marks will be deducted.**

4.2 Milestone 1: Test Cases

For Milestone 1, you must develop test cases for your Qwirkle implementation, including your enhancements.
These test cases will help ensure that your Qwirkle implementation is correct.
A single test case consists of 3 files, 2 mandatory and 1 optional.

1. `<testname>.input` - Input to provide to the Qwirkle program via stdin
2. `<testname>.output` - Expected output from the Qwirkle program on stdout
3. `(Optional) <testname>.save` - Expected output of the saved game file.

A test is run using the following sequence of commands.

```
./qwirkle < <testname>.input > <testname>.gameout

diff -w <testname>.output <testname>.gameout

if [-e <testname>.save] diff -w -y <testname>.save <actual_gamesave>
```

If this command displays any output, then the test has failed.

To make testing reliable, you should note if the test evaluates the saved game output, then ensure the test uses a suitable filename in place of `<actual_gamesave>`.

!	**Testing uses the `diff` command. This command checks to see if two files have any differences. The `–w` option ignores any whitespace.**

4.3 Milestone 2: Basic Implementation

For Milestone 2, you must implemented the base Qwirkle program as described in Section 3. Your base implementation should pass all test cases that you developed in Milestone 1.

!	**Your base implementation will only be considered to be sufficient if it is fully functional and error-free.**

4.4 Milestone 3: Enhancements

For Milestone 3, you need to develop one or more enhancements to the base Qwirkle program. **Milestone 3 is optional**. You should only commence enhancements if your base implementation is fully functional.

Enhancements may be either **minor** or **major**. To get a higher grade you will need to implement one or more minor or major enhancements. This is described in the marking rubric. Enhancements only count towards this Milestone if they are fully functional and error-free.

4.5 Milestone 4: Report

You are required to submit a report (of no more than 4 pages), that analyses and evaluates what your group has done in this assignment. Your report should comment on:

- Your use of ADTs, such as your Linked List, use of arrays and vectors.
- Design of your software
- Efficiency of your implementation
- Effectiveness of your Test Cases
- Group co-ordination and project management

Your report should note the strengths and weakness of these elements. Good analysis provides factual statements, evidence and justifications for conclusions that you draw. A statements such as:

> "We did <xyz> because we felt that it was good"

is not analysis. This is an unjustified opinion. Instead, you should aim for statements such as:

> "We did <xyz> because it is more efficient. It is more efficient because . . . "

4.6 Presentation

During Week 12, your group will present and discuss your Qwikle program to your lab demonstrator and/or the lecturer. In your presentation you should:

- Demonstrate your Qwirkle implementation and enhancements
- Demonstrate how your test cases prove your implementation is correct
- Discuss the design and efficiency of your software

Each presentation will be 10 minutes long, with 5 minutes for questions.

It is up to your group to decide how to best conduct this presentation. The purpose of the presentation is to demonstrate and convince the assessor of the quality of your group's software and overall work.

4.7 Weekly Progress Updates

Every 1-2 weeks in labs, you will have a brief update with your lab demonstrator. You can discuss:

1. Your group's progress
2. Ask questions about your software design and implementation
3. Raise issues
4. Demonstrate your consistent and regular contribution to the group

!	**One of the criteria in the marking rubric is for consistent and regular group work. These lab updates will help inform your final grade for your group work.**

5 Enhancement Suggestions

These are some suggestions of enhancements you could implement. If you have your own ideas, discuss these with your tutor or on the forum to make sure you know whether they count as minor or major.

5.1 Minor enhancements

!	**Minor enhancements are smaller in scope, and require only a small modification to your software design.**

5.1.1 Help!

Whenever there is a user prompt, the user may type "`help`" and the program should display some text to help the user determine what commands they may execute.

5.1.2 Better Invalid Input

Whenever the user enters invalid input at the user prompt, the program should show a useful error message to explain why the input was invalid.

5.1.3 Colour

Use colour to display tiles on the board and in the player's hand. The Linux, Mac (and most similar) terminals support the use of colour through the use of **escape codes**. A simple internet search will show you tutorials for working with escape codes.

5.1.4 Unicode/Emoji Tile Symbols

Use Unicode or Emoji for displaying tiles when printing information for the users to stdout. You can embed Unicode symbols or Emoji symbols directly into C++ files when used for writing information to output streams.
Most modern terminals also support the display of unicode symbols and emoji.

However, the saved game file must still be plain-text, and use the tile codes listed in Section 3.3.1.

5.1.5 Simple Hints

Provide a hint to the user of where a given tile could be placed on the board, and how many points the user will score. You will need to invent your own command so the user can ask for a hint.

5.1.6 Expandable Board

The board should only display enough rows and columns to fully show the board, plus one additional empty row and column around the edge of the board. Any additional empty rows and columns should not be displayed.

5.1.7 High Scores

The program maintains a list of high scores achieved by any player who has played your game of Qwirkle. You will need to devise a way to save, load, and display the high scores.

5.1.8 Forfeit

Provide functionality to allow a player (whose current turn it is) to forfeit the game. The tiles in their possesion are returned randomly into tile bag. Their score is set to forfeit and the remaining players can continue the game.

5.1.9 Wild power tile

At the start of each game, a tile is cosen at random and becomes the special wild power tile . When this tile is drawn from the bag, the player in possession of it can change either the colour or shape of the tile to their chosen value immediately before placing it on the board.

5.2 Major enhancements

!	**Major enhancements are large in scope, and require significant modifications to your software design.**

5.2.1 3-4 Player modes

Implement the 3 or 4-player mode of Qwirkle. You will need to change the number of tiles in each player's hand according to the rules of Qwirkle. This will also affect the format of the saved game file.

5.2.2 Place Multiple Tiles

Allow a player to place multiple tiles. You will need to invent your own command(s).

5.2.3 Write an AI

Develop an AI so a your program can be used in single-player mode, and a person can play against the computer. The AI should only place one tile at a time. If the AI cannot place any tiles, then it should replace one tile from its hand.

If you would like an additional challenge, you may allow the AI to place as multiple tiles.

5.2.4 Undo

Each player has up to 2 undo actions (that they can initiate per match). Each undo reverses the actions of one entire round of player actions. The current player can undo one round of each player turns (restoring placed tiles back to their original players and collected tiles back into the bag). At the end of the undo, it is the current players turn (and they can use this opportunity to counter/block or change the outcome of the round). (must be implemented in memory)

5.2.4 Instant Replay

At the end of the game, the winner can choose to replay all the moves in order for the entire match gameplay. (must be implemented in memory)

6 Getting Started

6.1 Managing Group Work

To help manage your group work, and demonstrate that you are consistently contributing to your groups, we recommend that you use the following tools:

- Git (for source code sharing and management)
- Trello (for allocating tasks)
- Slack/Facebook/WhatsApp (for group discussion)

These tools are not mandatory, but are highly recommended.

6.2 Designing your Software

This assignment requires you and your group to design the ADTs and Software to complete your implementation.
It is up to your group to determine the "best" way to implementing the program. There isn't necessarily a single "right" way to go about the implementation. The challenge in this assignment is mostly about software design, not necessarily the actual gameplay.

Trying to solve the whole program at once is too large and difficult. So to get started, the best thing to do is start small. You don't have to figure out the whole program at once. Instead, start with the smallest working program. Then add a small component, and make sure it is working. Then keep adding components to build up your final program.

You can get help about your ideas and progress through the lab updates. This is where you can bring your ideas to your tutor and ask them what they think. They will give some and ideas for your progress.

6.3 Starter Code

The start-up code is very limited. It contains the following files:

File	Description
TileCodes.h	Header file that give #define statements for tile codes
Tile.h/cpp	Skeleton definition of a tile
Node.h/cpp	Skeleton definition of a Linked List of tiles
LinkedList.h/cpp	Skeleton definition of a Linked List of tiles
qwirkle.cpp	Empty Main function
Makefile	Simple Makefile for compiling

7. Academic integrity and plagiarism (standard warning)

Academic integrity is about honest presentation of your academic work. It means acknowledging the work of others while developing your own insights, knowledge and ideas. You should take extreme care that you have:

- Acknowledged words, data, diagrams, models, frameworks and/or ideas of others you have quoted (i.e. directly copied), summarised, paraphrased, discussed or mentioned in your assessment through the appropriate referencing methods,
- Provided a reference list of the publication details so your reader can locate the source if necessary. This includes material taken from Internet sites.

If you do not acknowledge the sources of your material, you may be accused of plagiarism because you have passed off the work and ideas of another person without appropriate referencing, as if they were your own. RMIT University treats plagiarism as a very serious offence constituting misconduct. Plagiarism covers a variety of inappropriate behaviours, including:

- Failure to properly document a source
- Copyright material from the internet or databases
- Collusion between students

For further information on our policies and procedures, please refer to the University website.

8. Assessment declaration

When you submit work electronically, you agree to the assessment declaration.

RMIT UNIVERSITY

9. Rubric/assessment criteria for marking

	HD+	HD	DI	CR	PA	FL	NN
Test Cases (15%)	Outstanding	Comprehensive Unit tests for Milestone 2 and enhacements, covering the majority of common use cases, and most edge cases	Comprehensive Unit tests for Milestone 2, covering the majority of common use cases, and most edge cases	Sufficient Unit tests for Milestone 2, covering the majority of common use cases.	Sufficient Unit tests for Milestone 2, covering the majority of common use cases.	Poor (or missing) Unit tests for any part of the assignment, with poor coverage	Not Completed
		Test cases are well-design, with clear consideration given to the individual purpose of each test case.	Test cases are suitably design,but flawed, with consideration given to the individual purpose of each test case.	Test cases are well-design, with clear consideration given to the individual purpose of each test case.	Test cases are comprehensible, but explanation is required		
		Tests cases are largely, as simple as necessary for the purpose of the test.		Test cases are comprehensible, but explanation is required.			
		Unit tests contain no errors	Unit tests contain no errors	Unit tests contain no errors	Unit tests contain no errors	Unit tests may contain errors	
Milestone 2 (Base Implementation) (30%)	Outstanding	Complete and error free implementation of minimum requirements for Milestone 2.	Complete and error free implementation of minimum requirements for Milestone 2.	Complete and error free implementation of minimum requirements for Milestone 2	Complete and error free implementation of minimum requirements for Milestone 2.	Incomplete or error ridden implementation of the minimum requirements for Milestone 2..	Not Completed
		Exceptional coding style and suitably documented code. No input from the developers is required to comprehend the code.	Exceptional coding style and suitably documented code	Suitable coding style and suitably documented code. Code is readable, but parts of the software may not be clear	Suitable coding style and suitably documented code, but code is confusing to comprehend without further explanation from the developers	Poor and messy coding style with minimal documented code	
Milestone 3 (Enhancements) (30%)	Outstanding	Complete and error free implementation of multiple extensions, with at least one major extension	Complete and error free implementation of at least one major extension.	Complete and error free implementation of multiple minor extensions.	Complete and error free implementation of multiple minor extensions.	Incomplete or error ridden implementation all attempted extensions.	Not Completed
Group Work, Report (Mile 4), Present, & Ethical Dev (25%)		Style, and language features as per HD for Milestone 2	Style, and language features as per DI for Milestone 2	Style, and language features as per CR for Milestone 2	Style, and language features as per PA for Milestone 2	Style, and language features as per FL for Milestone 2	
	Outstanding	Exceptional group participation and work ethic in the group for the entire duration of the assignment.	Good group participation and suitable work ethic in the group for the entire duration of the assignment.	Sufficient group participation and work ethic in the group for a portion of the duration of the assignment.	Sufficient group participation and work ethic in the group for a limited duration of the assignment.	Poor group work, lacking in participation	Not Completed
		Exceptional management, documentation, and delegation of tasks across the group	Sufficient management, documentation, and delegation of tasks across the group, but with flaws.	Sufficient management, documentation, and delegation of tasks across the group	Minimal management, documentation, and delegation of tasks across the group.	Poor group management, documentation, and delegation of tasks across the group.	
		Exceptional, clear and concise Report & Presentation, with exceptional analysis of the performance, design, and implementation of the software.	Clear and concise Report & Presentation, with good but flawed analysis of the performance, design, and implementation of the software.	Suitable but Report & Presentation, with limited but suitable analysis of the performance, design, and/of implementation of the software.	Suitable but basic Report & Presentation, but with minimal analysis of the performance, design, and implementation of the software.	Poor, unclear, and confusing Report & Presentation, with minimal analysis of the performance, design, and implementation of the software	
		Exceptional demonstration and adherence to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.	Exceptional demonstration and adherence to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.	Sufficient demonstration and adherence to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.	Sufficient demonstration and adherence to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.	Violation of the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.	