RMIT UNIVERSITY

Programming Studio 2
COSC 2804
Assignment 3

Assessment Type	Group assignment. Submit online via Canvas → Assignments → Assignment 3. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date	5.00pm, Friday 03 November 2023
Marks	45.

Contents

1 Overview

This assignment will introduce you to developing software around a high-level API, while also serving as an opportunity to apply the C++ programming skills you learned in Bootcamp 2 and advanced concepts you learned in Studio 2 to a larger, team-based project.

Your main task is to design an "Expansion pack" to Minecraft that allows a user to solve mazes. You will achieve this by leveraging Minecraft's C++ API, "mcpp". This task is deliberately open-ended, and you are encouraged to be creative with the solution. However, there are some specific constraints that must be adhered to, as outlined in the specification below. The assignment will be marked on four criterion's:

- **Base Implementation & Testing (30 marks)**: You will write a program that can create (or read) a random perfect maze, build it in the Minecraft world, and implement "help" functionality that can guide a player who is inside the maze towards the exit. This also includes writing tests to verify if your program is accurate.
- **Enhancements (9 marks)**: You will extend upon your base implementation with additional functionality. *The enhancements are for those who aim for HD grade.*
- **Video (6 marks)**: A video demonstrating how your final program works.

While this is a group project, **your individual final grade will depend on your contribution**, which will be evaluated through **in-class checkpoints** (see Section 4.3) and your **GitHub commit history**.

2 Learning Outcomes

This assignment covers the following course learning outcomes:

1. Analyse and solve computing problems; design and develop suitable algorithmic solutions; implement and debug algorithmic solutions using modern skills and practices in the C++ programming language;
2. Demonstrate the ability to communicate effectively with industry professionals and peers;
3. Demonstrate skills for self-directed learning, reflection and evaluation of your own and your peers work to improve professional practice;
4. Demonstrate adherence to appropriate standards and practice of Professionalism and Ethics.

3 Specification

3.1 Setting up the Minecraft API

Before you can begin work on the program, you will need to acquire a copy of Minecraft and follow the installation steps to set up the C++ API. See the link Setting up the Minecraft C++ API (mcpp) on the front page of the course Canvas shell.
It is recommended to set the level-seed of the Minecraft server to 1. This can be done by:

- Go to the folder where you installed the "Minecraft server" and delete the "world" folder.
- Locate server.properties file and set level-seed=1 inside it.

3.2 Base Program Functionality

The objective is to write a command line program that allows a user to solve mazes. The base program should have the ability to:

- Generate or load arbitrary-sized perfect mazes. A perfect maze is a maze where every point is reachable and where there is only one single path from one point in the maze to any other point.
- Build the maze in Minecraft world at a location specified by the user. In the base program, you should flatten the terrain before building the maze.
- Place a player at a random location inside the maze so that the player can navigate to the exit (i.e., solve the maze).
- Show a solution to the maze (path to exit) when requested by the player.
- The program should execute without any errors and upon exit it should reverse any "modifications" done to the Minecraft world (clean-up).

Launch:
The base program should have two modes of operation: a "`normal`" mode and a "`testing`" mode. The program is launched using the following terminal command (the square brackets indicate optional parameters):

```
>> ./mazeRunner [mode]
```

The optional parameter (`[mode]`) is left empty when running the program in `normal` mode, and it is set to "`-testmode`" when running in the test mode.
On launch, the program should display a welcome message:

```
Welcome to MineCraft MazeRunner!
--------------------------------
```

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

```
------------- MAIN MENU -------------
1) Generate Maze
2) Build Maze in MineCraft
3) Solve Maze
4) Show Team Information
5) Exit

Enter Menu item to continue:
```

Main Menu:

In the main menu, the user can input one of five options. If the user enters a number between 1-5, then the program should take the user to the respective menu or action (see sections below). In case the user enters anything other than digits 1 to 5, then the program should NOT crash or exit. Instead, it should display the following message and then take the user back to the main menu.

```
Input Error: Enter a number between 1 and 5 ....
```

Note: *Throughout the program, any input errors should be handled in a similar manner, i.e., inform the user that an input error has been made, indicate what are the expected inputs, and safely move to an appropriate program state without exiting or crashing.*

Generate Maze Menu:

In generate maze, the program should display three options:

```
------------- GENERATE MAZE -------------
1) Read Maze from terminal
2) Generate Random Maze
3) Back

Enter Menu item to continue:
```

When the user selects one of the three options, the following actions should take place:

1) **Read Maze from terminal:** The program should prompt the user to input maze details and capture the entered information. First it should prompt for the "Base Point" of the maze (i.e., [X Y Z] coordinate of the north-west corner of the maze in Minecraft world):

   ```
   Enter the basePoint of maze:
   ```

   Note: The y-coordinate entered must be one unit above the ground
   i.e., `getHeight(X, Z) + 1`.

   Next it should prompt the user for the length and width of the maze. Length (number of blocks in x direction) and width (number of blocks in z direction) should be odd numbers.

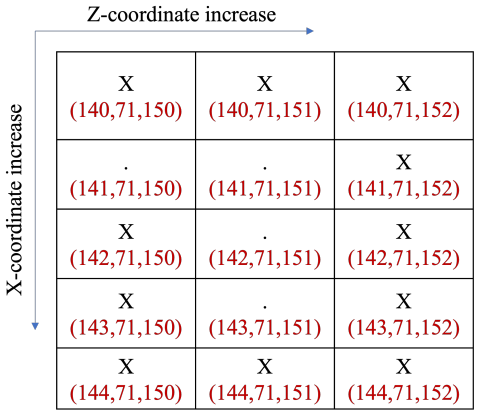
   ```
   Enter the length and width of maze:
   ```

Finally, the program should prompt for the structure of the maze.

```
Enter the maze structure:
```

The structure should be entered as a grid of characters 'x' and '.' where 'x' stands for a wall and '.' stands for empty space. The maze should only have one exit and the passages should have a width equal to one block and no loops (i.e., perfect maze).

If the base point is given as $(140, 71, 150)$, the length is 5 and the width is 3, then the mapping between maze structure and MineCraft coordinates is:



Once all the inputs are read, the program should print out the massage "Maze read successfully" followed by the maze information. Then it should return to the main menu. An example sequence of reading the maze is shown below:

```
------------- GENERATE MAZE -------------
1) Read Maze from terminal
2) Generate Random Maze
3) Back

Enter Menu item to continue:
1
Enter the basePoint of maze:
48 71 48
Enter the length and width of maze:
5 5
Enter the maze structure:
x.xxx
x.x.x
x.x.x
x...x
xxxxx
Maze read successfully
**Printing Maze**
BasePoint: (48, 71, 48)
Structure:
x.xxx
x.x.x
x.x.x
x...x
xxxxx
**End Printing Maze**
```

2) **Generate Random Maze:** Similar to the "Read maze from terminal" option, the program should prompt the user for the base-point and the length and width of the maze. Next it should generate a perfect maze according to the input parameters, using the "**Recursive backtracking**" algorithm. An overview of the Recursive backtracking algorithm is provided in Section 3.3. Your program must not use any other algorithm to generate a maze.

   The behavior of the maze generation algorithm will differ depending your program is running in the "`normal`" mode or the "`testing`" mode. More details are provided in Section 3.3.

   Once the maze is generated, the program should print out the massage "Maze generated successfully" followed by the maze information. Then it should return to the main menu. An example sequence of reading the maze is shown below:

```
------------ GENERATE MAZE ------------
1) Read Maze from terminal
2) Generate Random Maze
3) Back

Enter Menu item to continue:
2
Enter the basePoint of maze:
48 71 48
Enter the length and width of maze:
5 5
Maze generated successfully
**Printing Maze**
BasePoint: (48, 71, 48)
Structure:
x.xxx
x.x.x
x.x.x
x...x
xxxxx
**End Printing Maze**
```

3) **Back:** The program should return to the main menu.

Build Maze in MineCraft Menu:

When the user enters this option,

- First, the program should teleport the Minecraft player to a position that is 10 units higher than the base point of the maze. To do this, add 10 to the y coordinate of the base point and use that value as the new y coordinate for the player.
- Flatten the terrain: Change the height of the blocks in the area where you want to build the maze, so that they all match the y coordinate of the base point. You can do this by either breaking blocks or placing blocks of the same type as the ground.
- Build the maze structure by placing Acacia wood planks in the shape of the maze layout (`mcpp::Blocks::ACACIA_WOOD_PLANK`). The walls should be 3 blocks high, meaning that you need to stack 3 planks on top of each other for each wall segment.
- Once the maze is built, the program should go back to the main menu.

All placing or removing blocks should follow an approximate 50ms delay so that the user can see the build happening. See the video introduction on Canvas for more details. Remember to keep track of any modifications to the Minecraft world, so that you can reverse them when exiting the program.

In case a maze already exists when the "Build Maze in MineCraft" command is issued, the program should first clean the Minecraft world (remove the old maze and the flattening of terrain) and then start building the new maze.

Solve Maze Menu:

The program should display the Solve Maze menu.

```
------------- SOLVE MAZE -------------
1) Solve Manually
2) Show Escape Route
3) Back

Enter Menu item to continue:
```

When the user selects one of the three options, the following actions should take place:

1) **Solve Manually:** The program should place the Minecraft player in a random location within the maze (avoid walls!). If the program is operating in the "`testing`" mode, the player should always be placed in the empty cell that is closest to the lower-right edge (furthest from the base point) of the maze.

2) **Show Escape Route:** The program should find out the player's position in the maze and guide them to the exit. The program should use the **Wall Follower algorithm (Right-hand version)** to calculate the path from the player to the exit. This algorithm is explained in Section 3.3. You must not use any other algorithm for this task. The program should show the escape route to the player by placing `mcpp::Blocks::LIME_CARPET` blocks on the ground along the path. The program should place one block at a time, with a delay of approximately one second between each block. The previous block should be removed when the new block is placed. See the video introduction on Canvas for more details.

   Your program should also output the path coordinates to the terminal. see the example below:

```
------------- SOLVE MAZE -------------
1) Solve Manually
2) Show Escape Route
3) Back

Enter Menu item to continue:
2
Step[1]: (4853, 71, 4373)
Step[2]: (4853, 71, 4372)
Step[3]: (4852, 71, 4372)
Step[4]: (4851, 71, 4372)
Step[5]: (4851, 71, 4373)
Step[6]: (4851, 71, 4374)
Step[7]: (4850, 71, 4374)
Step[8]: (4849, 71, 4374)
Step[9]: (4849, 71, 4373)
Step[10]: (4849, 71, 4372)
Step[11]: (4849, 71, 4371)

------------- SOLVE MAZE -------------
1) Solve Manually
2) Show Escape Route
3) Back

Enter Menu item to continue:
```

The behavior of the algorithm will differ depending your program is running in the "`normal`" mode or the "`testing`" mode. More details are provided in Section 3.3.

None of the operations done under this menu should have access to the maze structure, size, or basePoint. This means that the only way for the player (or the solving algorithm) to know the structure of the maze (where the walls and empty spaces are) is to sense the Minecraft world (i.e., query Minecraft using `getBlock()` `getHeight()` commands). This property should incorporated into your code.

3) **Back:** The program should return to the main menu.

Show Team Information Menu:
The program should display the names and RMIT email addresses of all team members. Then the program should return to the main menu.

```
------------- MAIN MENU -------------
1) Generate Maze
2) Build Maze in MineCraft
3) Solve Maze
4) Show Team Information
5) Exit

Enter Menu item to continue:
4

Team members:
         [1] Ruwan Tennakoon (ruwan.tennekoon@rmit.edu.au)
         [2] Steven Korevaar (steven.korevaar@rmit.edu.au)
```

Change the team member information in the above example accordingly.

Exit:
The program should clean any modifications done to the Minecraft world. This means any blocks added should be removed and any removed blocks should be placed back. Additionally, any memory allocated by the program should be de-allocated. Once completed the program should display the massage "The End!" and safely quit.

```
------------- MAIN MENU -------------
1) Generate Maze
2) Build Maze in MineCraft
3) Solve Maze
4) Show Team Information
5) Exit

Enter Menu item to continue:
5

The End!
```

Note: In Minecraft there are some elements that cannot be restored. For example, there are some plants and flowers that grow on the surface; water/lava flowing; chests with items in them, etc. It is not expected that such elements will be restored. However, the

mainland shape and terrain should be preserved as best as possible. This means that any block that you can sense using `getBlock()` should be restored.

3.3 Algorithms for Generating and Solving Mazes

Wall Follower Algorithm:

Escaping a maze can be done in different ways. One way is to choose a random direction at each junction, but this is not very efficient and may not work at all. In this assignment, we will implement a better algorithm to escape a maze called "Wall follower". Wall following is a simple algorithm that can be used by either a human or a robot to escape a maze without knowing its layout. It does not require any memory and it always finds a solution if the maze has no loops. To implement wall following in a robot (agent), we can use the following procedure:

1. **Initialize player:** Read the block where the player is currently and assume an orientation such that the player's right hand is touching a wall (i.e., set the orientation in code so that the player's right hand is facing a wall - you may need to sense where walls are by using `getBlock()`).

2. **Move:** Follow the wall on the right-hand side to exit. For a human this would mean: keeping the right hand touching the wall and following that wall until the player reaches the exit.

Why does this simple algorithm work? Let's discuss it in class.

When the program is running in the "`testing`" mode, the player should be oriented using the following procedure: Read the block where the player is currently at, and assume the player is oriented such that it is right hand is facing the positive x-axis direction in Minecraft. Then, turn the player clockwise until its right hand is towards a wall. *You don't have to change the orientation of the player in Minecraft. This is only done in your code.*

Recursive Backtracking Algorithm:

There are many algorithms to generate mazes. You can learn about different maze generation algorithms from this website. "Recursive backtracking" is one such algorithm. Here is a high-level overview of the Recursive backtracking algorithm.

"Normal" mode: The procedure below explains how your program should generate a maze in the "`normal`" operation mode. Later, we will see how the "`testing`" mode should operate.

1. Initialize a rectangular maze - all cells are isolated (i.e., surrounded) by walls.

2. Choose a random starting point. The starting point should be an empty cell adjacent to the outer wall. Break the outer wall to create the entrance/exit.

3. Randomly choose a wall at that point and carve a passage through to the adjacent cell, but only if the adjacent cell has not been visited yet. This becomes the new current cell.

4. If adjacent cells in all directions have been visited, back up to the last cell that has un-carved walls and repeat.

5. The algorithm ends when the process has backed all the way up to the starting point.

There are many ways to implement the algorithm above. In assignment 3 you **must use the recursive methodology**.

"Testing" mode: This mode introduces two main changes to the algorithm explained above. These changes remove any randomness in the above procedure and make the program produce the same maze every time it runs for a given size.

1. The starting point is always set to (1, 1) and the gate is towards the left of this - (1, 0).

2. The directions for carving the passage in step 3 above are always selected in the following order: `UP, RIGHT, DOWN, LEFT`.

   If the current location is (x, z) then: `UP` : $(x-1, z)$; `RIGHT` : $(x, z+1)$; `DOWN` : $(x+1, z)$; and `LEFT` : $(x, z-1)$.

Worked example (normal mode): Let's create a maze with $H = 7$ and $W = 5$. Step one is to initialize the maze:

```
xxxxx
x x x
xxxxx
x x x
xxxxx
x x x
xxxxx
```

Next, we need to randomly select a starting point (step 2). let's select cell (3, 1) as the random starting point. The current location is denoted by a '*' - just for illustration. The gate should be the closest exit which is (3, 0) for this example.

```
xxxxx
x x x
xxxxx
.*x x
xxxxx
x x x
xxxxx
```

Next, we need to randomly select a wall to carve a passage (step 3). There are three options `UP, RIGHT, DOWN`. None of the adjacent cells have been visited before. Let's randomly select `RIGHT` direction:

```
xxxxx
x x x
xxxxx
...*x
xxxxx
x x x
xxxxx
```

We now have two options, move `UP` or `DOWN`. Let's move `UP`.

```
xxxxx
x x*x
xxx.x
....x
xxxxx
x x x
xxxxx
```

We now have only one option. Move `LEFT`.

```
xxxxx
x*..x
xxx.x
....x
xxxxx
x x x
xxxxx
```

There are no more directions to move at the current location. If we move `DOWN` it will create a loop - other directions exit the maze.

We now go to step 4 - backup until the place where we had multiple options. In this example, the last place we had another option is (3, 3). Backup to there:

```
xxxxx
x...x
xxx.x
...*x
xxxxx
x x x
xxxxx
```

Now we randomly select from the remaining options. We only have one option remaining - move `DOWN`.

```
xxxxx
x...x
xxx.x
....x
xxx.x
x x*x
xxxxx
```

Only one move remaining here - move `LEFT`

```
xxxxx
x...x
xxx.x
....x
xxx.x
x*..x
xxxxx
```

No more options - we back up and end up in the start location. This is the end of the algorithm. We have created a perfect maze! No unconnected empty cells and no loops.

```
xxxxx
x...x
xxx.x
....x
xxx.x
x...x
xxxxx
```

3.4 Enhancements

This section is designed *for students who are striving for an HD grade*. It consists of two challenges. First, you will enhance the maze-generating algorithm from base implementation to enable the creation of mazes on any type of terrain, not just flat surfaces. Second, you will improve the maze-solving algorithm to find the shortest path out of the maze.

Do not overwrite the work in base implementation when doing enhancements. Both stages need to be functioning in the final code to get full marks. You may use another command line flag to enable enhancements. Document how to run the enhanced version of your code.

E1: Create a Maze Without Flattening the Terrain

You will relax the assumption made in base implementation: "The terrain is flattened and devoid of any obstacles". The extended program should be able to build a maze at any reasonable location in the Minecraft world without modifying the terrain (it is reasonable to assume that the location is not underwater or contains cliffs). Furthermore, You should also avoid removing any obstacles like trees, holes, buildings, etc. that are within the maze area, and instead build around them. Since this stage is open-ended, you should come up with your own plan, following these requirements:

Requirements:

1. The program should be able to build a maze in any "*reasonable*" location without any modification to the terrain. In class, you can ask the teaching team about what locations are reasonable or not
2. Only needs to do this for automated maze generation (not for reading from the terminal).
3. Do not remove any obstacles within the maze area, but build around them.
4. Do not carve paths in directions with slopes of more than 1 block. Minecraft players cannot jump over 1 block.
5. The maze should be "perfect" (no loops and no isolations).

Hint: You may want to change how the recursive backtracking initializes the maze. This may involve sensing the environment in which the maze is to be built.

E2: Find the Shortest Path to Exit

While the wall-following algorithm you developed in the base program can find a solution to any perfect maze, there is no guarantee that the path found will be the shortest. In this milestone, your task is to write a new function that can find the shortest path to exit from any given location in the maze.

Requirements:

1. The Minecraft player does not have access to the structure of the maze. The only way to get information is by sensing in Minecraft world.
2. In cases where there are multiple shortest paths, your program has to find **only one** of those paths.
3. The program needs to be efficient. At least it should be better than a program where the agent walks to every empty location in the maze, calculates the distances, and then the shortest path.

Hint: You may want the agent (player) to explore all the locations that are one unit away from the current location, then two units away from the current, and so on until the agent finds the exit. This can be implemented using two lists: "visited" (initialized empty) and "queue" (initialized with current position). Each step involves taking the first element from the "queue" that is not in the "visited" list and adding all its directly reachable neighbors to the back of the "queue". When adding a particular location to the "queue" you should also attach the information about which current location lead to discovering that particular location (previous). When you find the exit, you can use the information about the previous to find the path from the exit to the current location - which in reverse will be the shortest path from the current location to the exit.

3.5 Testing for Correctness

Before starting out on implementation, it is good practice to write some tests. We are going to use I/O-blackbox testing which is discussed during Week 5 - class 2.

You need to write test cases to determine if all elements of your code are correct. This involves designing appropriate inputs and working out what should be the output if our program is 100% correct. The "`testing`" mode is designed so that your program does not have any randomness, enabling you to pre-determine the output.

A test consists of two text files:

1. `testname.input` - The *input* for the program.
2. `testname.expout` - The *expected output* if our program is 100% correct.

The tests should be named to convey the aim of the test. A test *passes* if the output of your program *matches* the expected output.
A test is run using the following sequence of commands.

```
>> ./mazeRunner -testmode < testname.input > testname.out
>> diff -w testname.expout testname.out
```

If this command displays any output, then the test has failed. Testing uses the `diff` command. This command checks to see if two files have any differences. The `-w` options ignore any white-space.

You would need to read the base game specifications, carefully before attempting to write tests. We will mark your tests based on how suitable they are for testing that your program is 100% correct. Just having trivial tests is not enough for full marks. Identify scenarios where your program might break and construct tests accordingly.

The starter code contains a folder with one sample test case. This should give you an idea of how your tests should be formatted (the sample will not be counted when marking).

4 Deliverables

4.1 Mandatory Requirements

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

- Adhere to all the specifications. Any assumptions made should be clearly documented.
- You must only use the C++17 STL. You must not incorporate any additional libraries (except for mcpp).
- Your program should compile in c++17 with the following flags.

```
>> g++ -Wall -Werror -std=c++17 -O -g -o mazeRunner ./*.cpp -lmcpp
```

- Your program must use defensive programming when interacting with users and appropriate contracts for any other interface.
- Your program must use good coding practices discussed in class. This includes the use of appropriate data structures or ADTs, memory management, and efficiency.

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

4.2 Implementation

The overall project implementation is broken down into several milestones. It is expected that the students will implement the project in the order of the milestones mentioned below.

- **Milestone 1 - Black box test cases & menu navigation**: This milestone involves writing test cases as described in section 3.5 and implementing the basic menu navigation logic. For Milestone 1, you are only expected to test the base program (not the enhancements).
- **Milestone 2 - Reading maze and finding the path**: At this milestone, your program should have the capability to read a maze through the terminal, build in Minecraft, and place the player inside to solve manually. Additionally, it should also include the wall follower to find the path to exit.

- **Milestone 3 - Generating the maze & cleanup**: This milestone involves implementing the recursive backtracking algorithm to generate a maze and implementing the functionality to revert back any changes done to the Minecraft world upon exit.
- **Milestone 4 - Enhancements**: This milestone involves implementing the enhancements described in Section 3.4.

4.3 Weekly Checkpoints

Each week in the second half of the course will contain a project checkpoint. This will happen in the third class of the week. During the checkpoints, students will demo their current progress to the instructors and answer any questions by them. Students should also explain the individual contributions of each group member. This may include showing the commit history of each member. The objectives for each week are listed below:

- **Week 05**: Thorough understanding of the specifications and an initial attempt at implementing the menu navigation. Identified each team member's responsibilities.
- **Week 06**: Significant progress in Milestone 1.
- **Week 07**: Significant progress in Milestone 2.
- **Week 08**: Significant progress in Milestone 3.

While the checkpoints themselves do not carry any marks explicitly, your final grade will be influenced by the consistent progress showcased during these checkpoints. Therefore, attending them is strongly advised. Additionally, checkpoints serve as a platform to identify and address group-related challenges at an early stage. The teaching staff can provide timely interventions to resolve such issues, preventing them from adversely affecting individual final grades.

If required the teaching staff will review your individual GitHub commits. This is partly to ensure that all team members are contributing evenly, but also to assess whether you are following best practices with GitHub. Some words of warning: If you have zero commits on GitHub, you will receive zero marks for the entire assignment, no exceptions! Similarly, if you have made some commits, but your overall contribution appears very thin, we will penalise you as many marks as seems fair. You must not email your code to your teammates and ask them to commit it for you; this will not count as a valid excuse for having few commits. Please summarise each team member's contributions in the README.md file of the main branch: a high-level summary of what each member worked on, in dot-point form.

You should commit early and commit often. By the end of the assignment, you should not just have a few, huge commits, each containing big chunks of the solution. Instead, your work should be broken up across many incremental commits. Use meaningful commit messages. Just like the comments in your code, a commit message should clearly summarise the purpose of the commit. Extremely terse messages such as "fix", "work", "commit", and "changes" are poor and will not help us understand what was done. Note that peer programming, which we encourage, does not mean that one member can always (or mostly) act as the "driver" and commit; all members should take turns at being the "driver" and committing to the repo. Where necessary, use comments to specify the team members involved in the contribution, e.g., "Fixed an issue where external wall of the

maze is removed. Contributors: Anna and Tom". In general, try to make things easy for the markers. If your GitHub username does not correspond with your real name or student ID, please indicate who you are in the README.md file. If the commits are split across multiple branches, please make a note of this in the main branch's README.md file so that the markers do not miss any of your contributions.

4.4 Video

Teams are required to submit a short video presentation with the implementation. The purpose of the video is to provide a quick overview to the markers. Bear this in mind and try to show off the most impressive aspects of the implementation that you want the markers to notice. To highlight the randomized aspects of the maze, edge cases handled and, consider showing results from multiple runs.

You should explain the top five design choices you made to make the program efficient and accurate (or testable). This may include discussing the appropriateness of the data structures and ADTs used in your program. How you ensure correctness of your program in a team environment etc.

All students must present, and each presenter should speak for around 3 minutes. While we do not expect the video to be brilliantly edited, you should rehearse the presentation prior to recording it, and edit/re-record sections if they are sloppy. Try to make the presentation engaging by using video captures of the generated game world, rather than just using slides. To submit the video, you have a couple of choices: create a Microsoft team meeting amongst the team members and record the session - then you can move the recording to oneDrive and share the link (preferred method). Alternatively, you can host it on a video-sharing platform, such as YouTube. Please explain where to find the video in the README.md file.

5 Getting Started

5.1 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 implement 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 with your ideas and progress in the classes (especially on checkpoints). This is where you can bring your ideas to your tutor and ask them what they think. They will give some ideas for your progress.

5.2 Starter Code

The start-up code is very limited. It contains the following files. You can add/remove files as required:

File	Description
`menuUtils.h`	Support functions to print menu items.
`Maze.h/cpp`	Skeleton definition of a maze
`Agent.h/cpp`	Skeleton definition of a agent (player)
`mazeRunner.cpp`	Empty Main function
`Makefile`	Simple Makefile for compiling

6 Submission

Submission via GitHub class rooms.

After the due date, you will have 5 business days to submit your assignment as a late submission. Late submissions will incur a penalty of 10% per day. After these five days, Canvas will be closed and you will lose ALL the assignment marks.

Assessment declaration:
When you submit work electronically, you agree to the assessment declaration - `https://www.rmit.edu.au/students/student-essentials/assessment-and-exams/assessment/assessment-declaration`

6.1 Silence Period

A silence policy will take effect from **27th October 2023**. This means no questions about this assignment will be answered, whether they are asked on the discussion board, by email, or in person. Make sure you ask your questions with plenty of time for them to be answered.

7 Teams

Team membership will be decided by the course coordinators, though effort will be made to adhere to specific requests or constraints from students. Any issues that result within the team should be resolved within the team if possible; if this is not possible, then this should be brought to the attention of the course coordinators as soon as possible. **Marks are awarded to the individual team members, according to the contributions made towards the project.**

We expect each team to demonstrate the progress each week in the in-class checkpoint. **Checkpoints serve as a platform to identify and address group-related challenges at an early stage. The teaching staff can provide timely interventions to resolve such issues, preventing them from adversely affecting individual final grades.** Emails or personal messages after the due date will not be considered by the teaching staff to identify group issues/contributions.

The instructors will establish a GitHub classroom and establish a private repository for each team. The team will be required to use this repository to develop the application. The instructors will use the repository logs to monitor the activity and contributions

of each individual member of the team. Each team is also encouraged to use other collaborative tools such as Microsoft Teams.

8 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 following: `https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity`.

We will run code similarity checks.

9 Getting Help

There are multiple venues for getting help. The first places to look are the Canvas modules and discussion forum. You are also encouraged to discuss any issues you have in class with your tutors. Please refrain from posting solutions (full or partial) to the discussion forum.

10 Marking guidelines

The rubric that will be used to grade the assignment is viewable on Canvas.

- Base Implementation & Testing - 30 marks
  - Milestone 1 (8 marks).
  - Milestone 2 (10 marks).
  - Milestone 3 (12 marks).

- Enhancements (Milestone 4) - 9 marks
  - E1: Create a Maze Without Flattening the Terrain (4 marks).
  - E2: Find the Shortest Path to Exit (5 marks).
- Video - 6marks

The Final grade of each individual will depend on their contribution, which will be evaluated through in-class checkpoints and your GitHub commit history.

For Milestones 1-4, your marks will depend on **functionality** as well as using **good programming practices** and **advanced concepts** discussed in class. See Assignment 3 main page on Canvas for a more detailed rubric.

You will also Notice there are not many marks for "trying" or just "getting started". This is because by now you should be highly skilled in programming. You need to make significant progress on solving the task in this assignment before you get any marks.