RMIT UNIVERSITY

C++ Programming Studio
COSC 2804
Assignment 3

Assessment Type	Group assignment (Three students per group). Submit online via Canvas → Assignments → Assignment 3 & GitHub Classrooms. Marks awarded for meeting requirements as closely as possible. Clarifications/updates may be made via announcements/relevant discussion forums.
Due Date	5.00pm, Friday 13 June 2025
Marks	50.

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 C++ Bootcamp and the advanced concepts you learned in C++ Programming 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 criteria:

- **Testing (8 marks)**: Design black-box test cases to verify that *all components* of your program work as intended - see Sec 3.4 for more details. Each team member is responsible for creating test cases that cover their assigned portion of the project - see task allocation in Sec 3.5. Testing is to be done in '`testing`' mode explained later.
- **Implementation (30 marks)**: Your team will develop a program that can either generate or read a perfect random maze, construct it within the Minecraft world, and implement a "help" feature to guide a player through the maze to the exit. Although this is a group task, each team member will be *individually evaluated based on their contributions* to specific components of the overall project - see task allocation in Sec 3.5.
- **Integration (7 marks)**: Combine the individual components into a cohesive software program that fulfills all requirements outlined in the specification.
- **Video (5 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**. *We are aware of attempts by students to artificially enhance their commit history, such as making trivial modifications to the code of other group members (like adding spaces or editing comments, etc.). This will be viewed as misconduct and we strongly discourage such behaviour.*

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 & C++ compiler

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.

We require that the assignment be **compiled using the GCC compiler**. You will find instructions on how to configure this for both Linux and Mac environments on Canvas. **Producing code that does not compile on GCC may result in zero marks for the assignment.**

3.2 Overall 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) Create a 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.*

Create a Maze Menu:

In creating a maze, the program should display three options:

```
------------- CREATE A 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 move to where the maze is to be built. Once the user is ready, the program should read the location of the player and set it as the "Base Point" of the maze (i.e., [X Y Z] coordinate of the corner of the maze in Minecraft world):

```
In Minecraft, navigate to where you need the maze
to be built in Minecraft and type - done:
```

Note: The y-coordinate entered must be one unit above the ground i.e., `getHeight(X, Z) + 1`. You can assume that the location is reasonable to build a maze. i.e., on land, no huge cliffs, etc.

When in `-testmode`, the player should be teleported to coordinate `(4848, 71, 4369)`. This assumes you have set the level-seed to 1 as mentioned in section 3.1.

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.

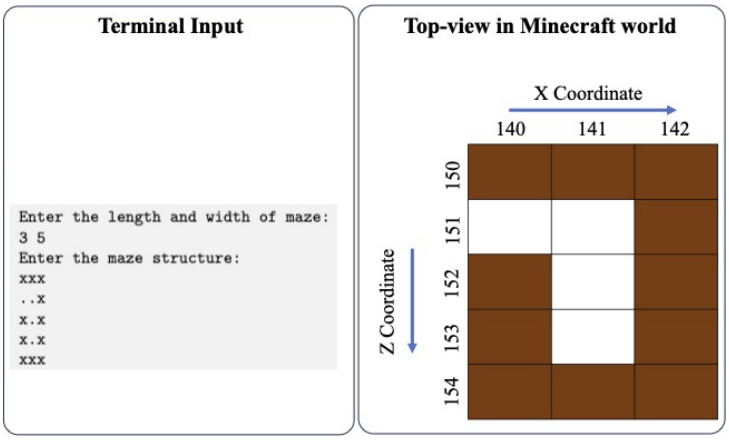
```
Enter the length and width of maze:
```

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.

See the below example that shows a sample input from the user and the corresponding expected maze structure in Minecraft world (viewed from above):



Once all inputs are read, the program should validate the user-defined maze to ensure the following:

(a) The maze contains exactly one entrance.

(b) The maze is perfect—i.e., it contains no loops or isolated sections.

(c) The maze structure adheres to the specified height and width, and uses only valid characters.

If errors related to the **first two conditions** are detected, the program should prompt user

```
Errors detected. Would you like to automatically fix them? (y/n)
```

if `yes`, the program should automatically correct them to produce a valid, perfect maze - instructions on how to do this is provided in Section 3.3. For any other type of error, the program should follow a defensive programming approach by rejecting the input and prompting the user to re-enter a valid maze.

```
Errors detected. Please try again.
```

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

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

Enter Menu item to continue:
1
In Minecraft, navigate to where you need the maze
to be built in Minecraft and type - done:
done
Enter the length and width of maze:
3 5
Enter the maze structure:
xxx
..x
x.x
x.x
xxx
Maze read successfully
**Printing Maze Structure**
xxx
..x
x.x
x.x
xxx
**End Printing Maze**
```

2) **Generate Random Maze:** Similar to the "Read maze from terminal" option, the program should prompt the user to navigate to 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 division**" algorithm. An overview of the Recursive division 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:

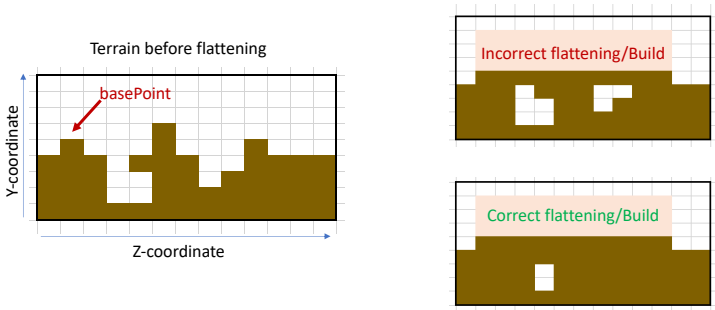
```
------------ CREATE A MAZE ------------
1) Read Maze from terminal
2) Generate Random Maze
3) Back

Enter Menu item to continue:
2
In Minecraft, navigate to where you need the maze
to be built in Minecraft and type - done:
done
Enter the length and width of maze:
5 5
Maze generated successfully
**Printing Maze 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,

- Flatten the terrain: Change the height of the blocks in the area where you want to build the maze (This is the area for the maze and one block border), so that they all match the y coordinate of the base point. You can do this by either removing blocks or placing blocks of the same type as the ground. **When flattening the terrain to build the maze you need to consider gravity. Do not place floating blocks**; all new blocks should be supported by at least one block below. See the below illustration.



- 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.
- The exit should be marked by a `mcpp::Blocks::BLUE_CARPET` block.
- 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 - blocks removed or added to the world by your program need to be saved in a suitable data structure that is memory efficient (i.e., do not store blocks that have not been changed).

In case a maze already exists when the "Build Maze in Minecraft" command is issued, the program should first clean the Minecraft world (revert all changes: removing the maze and unflattening the 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 furthest from the base point (lower-right edge 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 **Breadth-First Search (BFS) algorithm** 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 option "2) Show Escape Route" should have access to the maze structure, size, or basePoint - the agent solving the maze is not given the plan of the maze. This is similar to how a human is meant to solve a maze when they are inside - your software agent will do the same. 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) Create a 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 message "The End!" and safely quit.

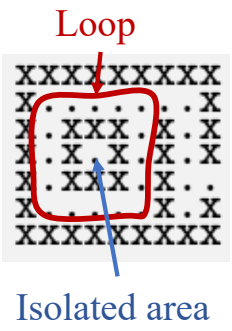
In order to remove/place back blocks you will need to save changes made to the Minecraft world in a suitable data structure. *When selecting the data structure, things including memory efficiency and efficiency of data access should be considered.* Saving Air blocks is not a good use of memory. You **must implement an appropriate data structure yourself utilising primitive C++ data types** to store the changes. **Should NOT use C++ STL Containers for this task.**

```
------------- MAIN MENU -------------
1) Create a 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, Validating and Solving Mazes

Validating & fixing the user Input Maze
The maze structure entered by the user under `Read maze from terminal` may not be perfect. In line with the defensive programming approach applied to user inputs, the maze should undergo validation when it is read in from the terminal. Specifically, the



function should detect any isolated areas or loops. In an isolated maze, there are passage sections that are inaccessible from the rest of the Maze. In mazes with loops, some walls are detached from the rest of the walls in the maze. An example maze with an isolated area and a loop is shown above.

Requirements:

1. In this task you are only required to identify/fix `isolations`, `loops` and entrance in a given maze.

2. For full marks, you need to do both detecting imperfections in the maze and fixing them.

Hints:
`Isolation`: Start with a copy of the Maze, then flood fill (you will learn flood fill in Studio class) the passage at the entrance. Scan the Maze (preferably in a random order that still hits every possible cell) for any unfilled cells within the maze - if there are, then the maze has isolation. To fix, scan the maze for any unfilled spaces that are adjacent to a filled cell. Remove a wall segment in the original Maze at that point, flood the Maze at this new point, and repeat until every section is filled.

`Loops`: The way to do this is almost identical to the isolation remover, just treat walls as passages and vice versa. Start with a copy of the Maze, then flood across the top of the outer walls. Scan the Maze (preferably in a random other that still hits every possible wall vertex) for any unfilled walls adjacent to a filled wall. Add a wall segment to the original Maze at that point connecting the two wall sections, flood the Maze at this new point, and repeat until every section is filled. This utility is used in the creation of template Mazes, and can be used to convert a braid Maze to a perfect Maze that still looks similar to the original.

`Entrance`: Once other errors are resolved, scan the outer wall. If no entrance exists, create one by breaking the wall at a location adjacent to a path. If multiple openings are found, close them all and then add a single entrance as above.

Generating Random "Perfect" Maze - Recursive division Algorithm:
There are many algorithms to generate mazes. You can learn about different maze generation algorithms from this website. "Recursive division" is one such algorithm. Here is a high-level overview of this 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 the Maze:* Begin by creating a rectangular area with only the outer boundary walls in place. The area's interior should be completely open, with no dividing walls.

2. *Divide the Area:* Randomly select to place a dividing wall either horizontally or vertically within the area. This wall should be placed along an odd-numbered row or column (assume cols/rows are numbered starting from 1).

3. *Create a Passage:* In the wall you've just placed, open up a passage by removing a single section of the wall. This passage should be created along an even-numbered row or column to maintain the maze's structure.

4. *Recursive Division:* Apply steps #2 and #3 to the newly created sections on either side of the dividing wall. Continue this process recursively, further subdividing each section by adding walls and passages.

5. *Finalize the Maze:* Continue the recursive process until the sections cannot be divided any further. This is achieved when the empty space in a given area has either the length or the width equal to a single cell.

6. *Add an entrance/exit:* To complete the maze, add an entrance/exit by removing a random wall block from the outer walls. Ensure that the entrance/exit is connected to the internal passages, allowing for a clear path into and out of the maze. Place a `mcpp::Blocks::BLUE_CARPET` just next to the exit as a marker.

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

"Testing" mode: This mode introduces three 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. In step #2 of the above algorithm, When considering a specific area, follow these guidelines to select the division direction (either horizontally or vertically):

   - If the current area was created by dividing a larger area horizontally, then choose to divide the current area vertically.
   - Conversely, if the current area was created by dividing a larger area vertically, then opt to divide the current area horizontally.
   - If current area is the entire field, as in the start of the algorithm, divide horizontally.

2. In step #2 of the above algorithm, place the dividing wall so that it splits the area into two segments of equal or nearly equal areas. You should only place walls on odd-numbered row or column

3. In step #3 of the above algorithm, place the passage at the block closest to the centre of the current wall.

Worked example (normal mode): Let's create a maze with $H = 7$ and $W = 7$.
Step #1 of the algorithm is to *initialize the maze* with outer walls:

```
xxxxxxx
x     x
x     x
x     x
x     x
x     x
xxxxxxx
```

Next, step #2 - *Divide the area.* We first need to select a random direction to split. Let's select *Horizontal.* Then we need to select an odd row number randomly. This can be a number from the set $\{3, 5\}$ for our example. Let's select row 3 and place a wall:

```
xxxxxxx
x     x
xxxxxxx
x     x
x     x
x     x
xxxxxxx
```

Next, step #3 - *Create a Passage*, remove a wall block in an even-numbered column. We can select a number from set $\{2, 4, 6\}$. Let's select 4 and create a passage.

```
xxxxxxx
x     x
xxx xxx
x     x
x     x
x     x
xxxxxxx
```

We have now created two areas: the area above the wall and the area below the wall. We have to run the above process (steps #2 and #3) for both these areas. Lets select the area above the new wall - this area cannot be further divided as the empty space has length one. Therefore, we do not process this area any further.
Let go to the area below the new wall. This area has an empty space with a length and width larger than one, so we can divide it further. Lets assume we randomly selected to divide *Vertically* at column 3:

```
xxxxxxx
x     x
xxx xxx
x x   x
x x   x
x x   x
xxxxxxx
```

Create a passage at a random location by breaking the wall at either 4 or 6. Let's do at 6:

```
xxxxxxx
x     x
xxx xxx
x x   x
x x   x
x     x
xxxxxxx
```

Again we have created two areas. The one on the left of the new wall cannot be further divided, so let's do the one on the right.
Lets do the division *Horizontally.* There is one option for us to put a wall - at row 5:

```
xxxxxxx
x     x
xxx xxx
x x   x
x xxxxx
x     x
xxxxxxx
```

Now break this wall at random at column 4 or 6. Let's do at 6:

```
xxxxxxx
x     x
xxx xxx
x x   x
x xxx x
x     x
xxxxxxx
```

Both areas that are created cannot be further divided. There are no more areas to be divided. So we have concluded the algorithm.
Finally we will create an entrance by removing a wall block from the outer wall:

```
xxxxxxx
x
xxx xxx
x x   x
x xxx x
x     x
xxxxxxx
```

Solving a Maze - Breadth-First Search (BFS)

The maze constructed in the Minecraft world is ideally a perfect maze, which makes it easier to solve. However, if the user chooses not to correct their input, the resulting maze may not be perfect. Therefore, we will implement an algorithm that can solve *any* maze, regardless of its structure or whether its layout is known in advance.

To implement the `Show escape route` feature, you must use the **Breadth-First Search (BFS)** algorithm. Instructions for implementing BFS are provided below.

Your task is to write a function that finds the shortest path to an exit in any maze, including non-perfect mazes.

Requirements:

1. The Minecraft player does not have access to the maze layout. The agent must explore the maze by sensing its surroundings in real-time.
2. If multiple shortest paths exist, the algorithm only needs to find **one** of them.
3. If no path to an exit exists (e.g., the player is in an isolated region), your program must inform the user: `Sorry, no path, you are trapped!`

Breadth-First Search (BFS) is a widely used algorithm in computer science for finding the shortest path in unweighted graphs. Since a maze can be represented as a grid-based graph, BFS is well-suited for this task. You will learn more about graphs in later courses - for now we are only interested in implementing BFS. If you are curious, read `https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/`

The core idea behind BFS is to begin at the player's current position and explore all reachable positions that are one hop (unit) away, then all positions two units away, and so on, expanding outward layer by layer until the exit is found. To implement this, you will use two lists:

- `visited` – a list to track which positions have already been explored.

- `queue` – a list (or queue data structure) to manage which positions to explore next.

Start by initializing `visited` as empty and `queue` with the player's starting position. At each step:

1. Remove the first position from the queue.
2. If it has not been visited, mark it as visited by adding it to the list.
3. Add all directly reachable neighboring positions to the end of the queue.
4. For each new position added to the queue, store the position it was discovered from (the `previous` node).

When the exit is found, you can reconstruct the shortest path by tracing back from the exit to the start using the `previous` mapping. Reversing this path yields the correct route from the starting position to the exit.

3.4 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. **Each student must contribute to testing by writing tests for their own tasks - cannot appoint one team member to write testing.**

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).

3.5 Task Allocation

Although this is a team project, each team member is expected to take ownership of distinct tasks as outlined below. This allocation ensures that every team member engages with key course concepts and can earn a strong grade based on individual effort. It is mandatory to follow this division of work. While collaboration during implementation is allowed, the bulk of each task should be completed by the team member to whom it is assigned.

- **Team Member 1:**
  1. Implement `Read Maze from Terminal`, including validation, correction of input maze, and handling all related error cases.
  2. Implement `Build Maze in Minecraft`, which involves flattening terrain, constructing the maze, and cleaning up the world upon exit.
  3. Develop black-box test cases to verify the correctness of the above functionalities.
- **Team Member 2:**
  1. Implement the `Generate Random Maze` feature using the Recursive Division algorithm.
  2. Handling error cases related to `Generate Random Maze`.
  3. Develop black-box test cases to verify the correctness of the above functionalities.
- **Team Member 3:**
  1. Implement the `Solve Maze` functionality, including both `Solve Manually` and `Show Escape Route`.
  2. Handle error cases related to maze solving.
  3. Develop black-box test cases to verify the correctness of the above functionalities.

Each of these tasks can be developed and tested independently. In addition, we expect the team to collaborate on defining the overall program architecture, choosing appropriate abstract data types (ADTs), and integrating individual contributions into a cohesive application (marks awarded under rubric category "Integration").

Note: If a student deviates from the assigned tasks and their contribution is found to be below the expected standard, their grade may be significantly reduced.

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.
- Implement your own data structure to hold the changes made to the world instead of using C++ STL containers.
- 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.
- You should provide a comprehensive `README.md` file detailing the task allocation of each team member, the details of the branches used by each member, etc.

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

4.2 Implementation

The project implementation is organized into a series of milestones. Students are expected to follow the sequence outlined below to ensure smooth progress and timely completion.

- **Milestone 1 – Team Formation, Task Allocation & Black-Box Test Cases:** Finalize team members, assign tasks according to the project guidelines, and write black-box test cases as described in Section 3.4.
- **Milestone 2 – Initial Implementation:** Each team member should complete a basic, working version of their assigned tasks. The focus is on establishing core functionality, even if incomplete.
- **Milestone 3 – Complete Task Implementation:** Each member should substantially complete their assigned components, with all key features and error handling implemented.
- **Milestone 4 – Integration & Demonstration Video:** Integrate individual modules into a single, cohesive program. Conduct thorough testing and record a demonstration video showcasing the project.

We will only assess the last commit to GitHub before the deadline.

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**: Significant progress in Milestone 1.
- **Week 06**: Significant progress in Milestone 2.
- **Week 07**: Significant progress in Milestone 3.
- **Week 08**: Significant progress in Milestone 4.

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 3-7 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.
`MazeReadWriteUtils.h`	Support functions to read maze.
`mazeRunner.cpp`	Skeleton code for menu system.
`Makefile`	Simple Makefile for compiling.
`README.md`	File to add task allocation ant other details.
`Tests`	Contains files for one example test

The provided code should compile and produce a basic version of the menu system when executed. You are free to modify any part of the starter code. **Do NOT assume that all components of the starter code are correct**—it's your responsibility to adjust the code as needed to fulfill the assignment requirements.

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 **13th June 2024**. 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

Teams must consist of exactly three students. Under no circumstances will teams with more than three members be approved.

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. 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 following rubric will be used to assess your assignment:

- **Testing (8 marks)** — Assessed individually. Each student must create black-box test cases that thoroughly evaluate the components of their assigned tasks (see Section 3.5). Tests should be placed in the `Test` folder in your GitHub repository, with descriptive filenames. The purpose and assumptions of each test should be documented in the `README.md` file within the `Test` folder.

  - 0 marks: No attempt, or tests do not follow the specifications in Section 3.4.
  - 4 marks: Tests cover standard use cases but miss important edge cases.
  - 6 marks: Most edge cases are tested, though a few are still missing.
  - 8 marks: Tests comprehensively cover normal and edge cases.

- **Implementation (30 marks)** — Assessed individually. Each student is responsible for implementing the functionality assigned to them (see Section 3).
  - 0 marks: No attempt, or the program does not compile.
  - 5 marks: Code compiles; partial implementation exists but key features are missing.
  - 15 marks: Most functionality is implemented, but there are logical errors or missing edge cases (e.g., maze validation fails in some scenarios, maze generation produces imperfect mazes, or generated paths are not always shortest).
  - 25 marks: Functionality is complete and correct, but the code lacks efficiency, uses suboptimal data structures, or does not follow the style guide.
  - 30 marks: Fully meets all specifications, is efficient, uses appropriate data structures, and adheres to the style guide.

- **Integration (7 marks)** — Assessed as a group. Teams must integrate all individual components into a fully functional program that meets all assignment specifications.
  - 0 marks: Program does not compile, or key components are missing.
  - 5 marks: Program compiles and functions correctly, but integration is inefficient or poorly modularized.
  - 7 marks: Complete, efficient, and well-structured integration of all components.

- **Video (5 marks)** — Assessed individually.
  - 0 marks: Student did not appear in the video or did not explain their assigned tasks.
  - 2 marks: Student explained their work, but the presentation lacked clarity or completeness.
  - 5 marks: Clear, complete explanation and demonstration of all assigned components and corresponding test cases.

Final grades will reflect individual contributions, which will be evaluated based on in-class checkpoints and GitHub commit history.

Note: This rubric rewards demonstrated progress and understanding—not just starting the task. By this stage, you are expected to have strong programming skills. Minimal or incomplete work will not earn partial credit.