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	11.59 pm, Sunday 2nd November 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 create "villages" (buildings connected by paths and surrounded by a fence). 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. There are five components and criteria by which your final mark will be derived:

- **Testing (10 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 3.6. Testing is to be done in '`testmode`' which will be explained later.
- **Implementation (20 marks)**: Your team will develop a program that can prepare a space for, design, and then build a "village" within the Minecraft world. 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 3.6.
- **Code organization and style (10 marks)**: the above criteria only details the functionality of your developed program; however, you must also write "good" code, not just code that is functional - see Sec 3.5.
- **Integration (5 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. This is required to ensure repeatable testing on a known and constant environment. To reset the world, navigate to the server folder and delete or rename the "world" folder. When restarting the server it will automatically regenerate the world (given the seed is the same, it will generate the exact same world).

We require that the assignment be **compiled using the G++ 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 G++ 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 generate villages. The base program should have the ability to:

- **Determine the suitability of certain chunks of terrain for placement of a house or other structure, and prepare the terrain to smoothly integrate the houses you build by terraforming it.** Suitability checks should include verifying steepness or the presence of ravines or other extreme terrain and surface water coverage. Terraforming is to be done by *blurring* the heights across an area surrounding each building, or applying a similar terraforming algorithm.

- **Procedurally create and place houses with unique room layouts.** This is done by recursively subdividing the area inside the house with walls to form rooms, and making sure that each room is connected by placing doors. Each room should be *themed* to contain different kinds of furniture (such as a bedroom, dining room, or living room).

- **Create paths between the placed houses that avoid obstacles and create an interconnected network in the village.** This is done using path finding algorithms. A complete path should connect all houses together, while avoiding water and steep slopes. It should also prefer to generate on flat terrain, and, if necessary, interpolate heights to create a walkable path.

Note: Aesthetics are not the purpose of this assignment! While it is nice to generate a very nice looking village, there are no requirements for or marks associated with aesthetics or style. However, making aesthetic improvements is appreciated, and may be useful if you want to use this project to demonstrate your skills to future employers.

Launch:

The program shall be run from the command line, and have the following interface:

```
$ gen-village [OPTIONS]
```

Where `OPTIONS` includes any combination of these options and in any order:

`--loc=x,z` // default set to player coordinates
Determines the coordinate at which the village is to be centred.

`--village-size=int` // default `200`
Determines area around base point to explore, e.g. if the village centre is at `(100, 100)` and `village-size` is `100`, the two corners of the village area would be `(50, 50)` and `(150, 150)`.

`--plot-border=int` // default `10`
The size of the border around plots. Used to determine the area to be terraformed to smooth out the landscape and how close other plots and the village wall can be.

`--seed=int` // default `time(nullptr)`
The seed used for randomness engine for the random number generator. By default the current time should be used.

`--testmode`
If present, the algorithms used must be the test-mode specific algorithm specified in each respective algorithm's description section.

If any incorrect flags are provided, or arguments are supplied incorrectly, display an error message to the command line interface (CLI). You should also test that user provided values are valid, e.g. that a negative village size is not used.

3.3 Algorithms for Generating, Validating and Building Villages

3.3.1 Task A — Finding House Plots & Terraforming

Task A is devoted to generating the high-level structure of the village and comprised of four components:

1. Finding suitable building locations,

2. Terraforming the land surrounding the building locations,
3. Building a wall surrounding the village, and
4. Placing "waypoints" around the village to assist in path generation.

Plot Validation:

The village area shall be centered around the player's `(x,z)` coordinate, the height determined by the height of the highest non-air block at that `(x,z)` coordinate (or an alternative location provided by the `loc` argument). The `village-size` argument shall be used to determine the area, by using it as both the length and width. For example, if the `village-size` is set to `200` and the player is located at coordinates `(100, 100)`, then the village area shall stretch between the corners `(0, 0)` and `(200, 200)`.

The program will then semi-randomly sample points within this area and select a plot size semi-randomly between 14 and 20 blocks inclusive, and test if that point can contain a valid plot. Each plot must also have a border specified by the `--plot-border` argument. A valid plot is determined by:

- A surface water coverage of $\leq 15\%$ (i.e. average 3 water blocks in a 20 block area). Does not include plot border. What water is within an acceptable plot must be replaced with land of some form.
- A maximum slope delta of 15 **excluding trees** (i.e. the highest and lowest blocks in the plot can have a maximum difference in Y of 15).
- The plot border must not be intersecting a wall or any other blocks at the height of the building.
- Plots must not intersect each other. The `plot-border` may intersect with other plot borders, but not plots themselves (you don't want to terraform an existing house or the flat ground it will be built on!).

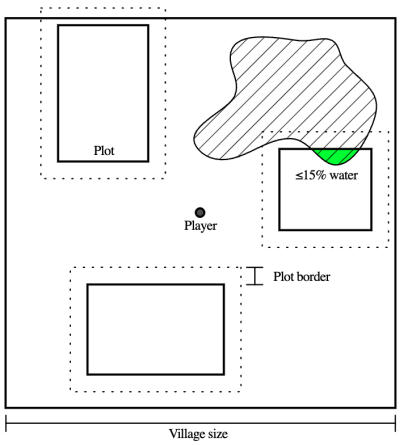

Figure 1: Village layout (note: top left plot is invalid as the plot border intersects the village wall.

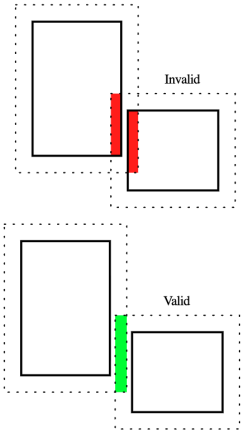

Figure 2: Intersecting plot borders

Your program should attempt to sample 1000 random plots at a maximum. If no valid plots are found within 1000 attempts an error message should be displayed to the user. You may place as many plots on a given village area that will fit; however, you should aim to have at least one plot per 50 blocks in the village size. For example: given a `village-size` of `50`, there should be at least one plot, given a `village-size` of `100` there should be at least two plots, etc (i.e. `num_plots >= village_size / 50`). You *may* remove all leaf blocks and/or trees before plot validation.

After the plot is selected and validated, select a semi-random point on the edge of the plot to serve as the entrance. Entrances should generally face towards the centre of the village, to ensure path generation works consistently, however you are free to use a different approach as long as they are not all predictably placed. The entrance must be fully accessible (this means not placed on a plot corner, at an intersection of multiple walls, or adjacent to furniture). You may need to discuss with your team how to handle this.

The plot validation step should result in a list of `Plot` objects for use in further parts of the program. At a minimum the `Plot` class should contain the following member variables.

```
class Plot {
public:
    mcpp::Coordinate origin;   // minimum/north-west (0, 0) corner
    mcpp::Coordinate bound;    // maximum/south-east (x, z) corner, whe
    mcpp::Coordinate entrance; // for paths and house generation

    // additional fields and methods as needed
} ;
```

Terraforming

After plot bounds are determined, you must pick the maximum height (ignoring trees) within your plot bounds to place the flat plot, i.e. placing a flat rectangle of blocks at the specified height. An area around the plot (with a size specified by `--plot-border`) must be terraformed to make the newly created plot fit in with its surroundings, i.e. the pre-existing terrain. This can be done by applying a function (be it linear, sine wave, or any other of your choice provided it is documented in the README.md) to modify the column according to its proximity to the edge of the plot. The choice of function is purely an aesthetic choice. As long as the terrain appears smooth (as seen in Figure 3) you can achieve full marks for this component.

For example, applying a linear function over a plot border of size 10 would influence columns 1 block away from the plot edge to move 90% of the height difference towards the plot height, while columns 10 blocks away would move 10% of the height difference towards the plot height. In mathematical terms this can be expressed as follows, where d is the distance from the current block the nearest plot block, y_g is the height coordinate of the ground beyond the plot border, y_p is the height of the plot, and p is the size of the plot.

$$block\_height(d, y_g, y_p, p) = round(y_g + (y_p - y_g) \cdot \frac{p - d}{p}) \tag{1}$$

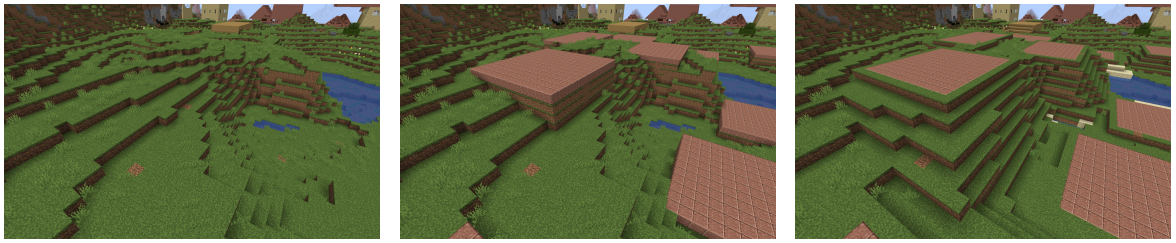

Figure 3: Three stages of terraforming, with `--plot-border=8` linear function

You may also opt for a more complex function to make the terraforming more aesthetically pleasing, a good example is using a sine function as seen in Figure 4.

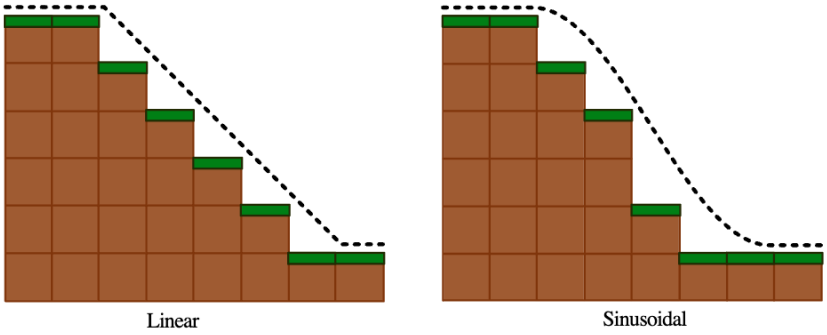

Figure 4: Linear vs Sine functions

Village Wall Placement

After terraforming, the program shall place a wall on the village boundary. The wall should be 3-4 blocks high, and impossible to cross by jumping/parkour from neighbouring blocks (Figure 5).

Figure 5: Wall Example

Waypoints

Waypoints are points in the village which serve as intermediaries for paths between plots. However, in this task, you only need to decide on the *location* of these structures, they should not be placed until the path generation step, Task C — Pathfinding. Waypoints should be positioned such that they fill *empty space* between plots, and should not contain any randomness. This can be done in many ways, but one way to do it is as follows:

1. Group neighboring plots into 3's. Prefer groups with a small total area (i.e. plots of a group should be close together).

2. Find the centre point of each plot group.

3. If the center point is suitable for a waypoint, add it to the list. A waypoint position is suitable if it is within no plot boundaries.

4. Continue this process until a suitable number of waypoints are found.

There should be at least one waypoint for every three plots within a village +- one or two in the case of complex village plots, with a minimum of one waypoint for every five plots. If no valid waypoints can be found, then display an error message and continue.

3.3.2 Task B — Building Houses

Task B is devoted to randomly generating the exterior and decorating the interiors of the buildings on the plots generated in Task A — Finding House Plots & Terraforming and comprised of four components:

1. Building a house,
2. Room subdivision,
3. Placing doors, and
4. Furniture placement.

You may assume in your code that the plot generation segment will return a list of `Plot` objects as described in Task A — Finding House Plots & Terraforming. As such, you now have an empty area on which to build your house!

This component will interact a lot with the quirks of the `mcpp` API. For additional information on rotating blocks, placing doors and beds (and other special blocks) please access the `mcpp` wiki.

Building a House The first component of this task is to place blocks to form the externals of a house: which includes four walls and a roof. A basic house is made of four vertical walls on the border of the plot (inside the plot, not on the border) and a roof. These can be any proper construction material you like (wood or stone recommended). The roof may be flat, but if you want aesthetics (despite there being no marks for it) you could use stairs to make a triangular roof! Houses should be a minimum of 6 blocks, a single storey of a building should be 20 blocks high at a maximum, and should be chosen semi-randomly.

Roof height: Each floor should be have at least five vertical blocks of space at a minimum inside to leave room for furniture.

Room Subdivision Once the structure of the building is complete we must fill out the inside of the building: for this you are to use the recursive subdivision algorithm to generate internal walls for each floor of the building. **Each room must be at least 4×4 blocks in size, but may be larger**. Each building may only have one room larger than 8×8 blocks. This must be implemented using recursive subdivision, you may make minor alterations to the algorithm to generate nicer looking buildings, but recursion must still be used.

The algorithm for recursive subdivision is as follows:

1. Start with an empty area representing the entire floor.
2. Split the area into two *sub-areas* along either the vertical or horizontal axes (chosen at random) at a random interval along the wall, provided doing so will not make the sub-areas smaller than 4 blocks in any direction.
3. Continue recursively splitting each sub-area until it creates areas which are small enough to be called rooms.

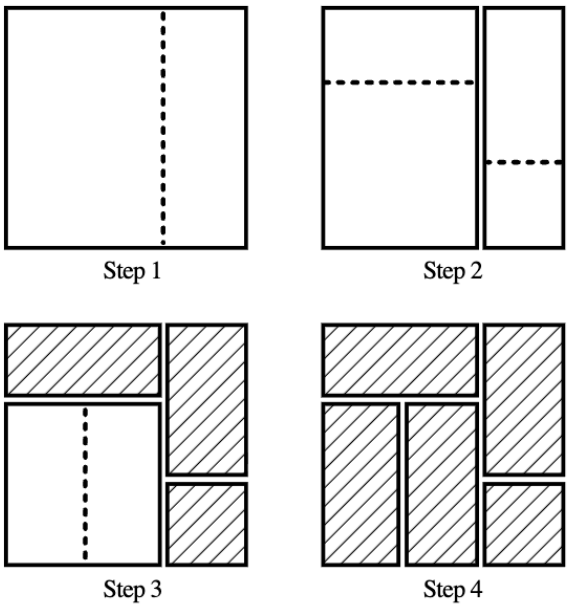

Figure 6: Recursive subdivision overview

Multiple floors: Your building may have multiple floors, if doing so you must connect them with stairs to ensure they are accessible, however multiple floors is not required for full marks (just style). You may need to modify the above algorithm to ensure adequate space to place stairs between levels.

Placing Doors

There's more to a house than four walls and even more walls inside – you need to be able to walk between rooms and be actually able to access each room. Place doors along the walls you just used in your subdivision algorithm so that every room is accessible from the entrance.

The simplest solution is to connect every room to each of its neighbours. For every room created by your subdivision algorithm, place one door on each wall. Before you do so, make sure to check that there isn't already a door placed along the wall, since the algorithm may have been run already for neighboring rooms. Also check that the wall connects two rooms (i.e. is not an exterior wall).

Furniture Placement

Once your rooms are subdivided, pick a *theme* for each room (such as bedroom, kitchen, living room), and place at least one item of corresponding furniture into the room (such as a bed, sink, couch). Don't worry if your furniture is a bit abstract – just make it recognizable. The main point is that it fits into the room and is aware of the room bounds, which you should have kept track of during the subdivision process. Again, there are no marks for aesthetics, as long as there is something identifiable for each room.

The following are some recommended themes and furniture pieces are listed here (there should be plenty of guides on how to design these using Minecraft blocks) this is not an exhaustive list, please feel free to design your own rooms if you choose to:

Don't forget to be creative! As long as the rooms and house structure are present and work as expected you can achieve full marks but for style and fun you can make any visual improvement to your houses that you wish. Put them on stilts, change the

Kitchen: • Sink (cauldron) • Countertop • Stone floor	**Living room:** • Red Carpet • Table • Bookshelf
Bedroom: • Bed • Wardrobe • Bedside table • Blue Carpet	**Bathroom:** • Toilet • Sink • Tiled floor • Bath

colour scheme, decorate them from the outside, give them a fancy roof or make them into skyscrapers!

3.3.3 Task C — Pathfinding

Task C surrounds finding and building paths to connect different parts of the village together. It is comprised of four components:

1. Connecting waypoints,
2. Connecting houses,
3. Building paths
4. Building waypoint structures.

As with Task B — Building Houses, you may assume in your code that the plot generation segment will return a list of `Plot` objects and a list of waypoint positions, as described in Task A — Finding House Plots & Terraforming. In this task you will connect all waypoints together, and each plot to a waypoint.

All houses and all waypoints must be connected to the same network of paths. This should be possible on all types of terrain; however, you may need to change the height of the path.

The generation of this network can be performed in two steps: Connecting Waypoints 3.3.3 and Connecting Houses 3.3.3.

Connecting Waypoints

You can choose which waypoints to connect in any way, as long as your solution does not leave any waypoints inaccessible or isolated. To check for isolated waypoints you can consider using a flood-fill algorithm. Though this may not be necessary if your waypoint connection algorithm is properly implemented.

The recommended algorithm is as follows:

1. Start at any waypoint.
2. Find the closest waypoint which is not connected.
3. Connect these waypoints by creating a path between them (see Finding a Path 3.3.3 for how to do so).
4. Add this new waypoint to a list of connected waypoints.
5. Repeat from step 2 until all waypoints are connected.

This process (known as Primm's Algorithm) creates a tree connecting all waypoints without redundant or cyclic paths. You can learn more about it **here**.

Connecting Houses

For each house, find the nearest waypoint, and create a path between it and the house entrance (see Finding a Path 3.3.3 for how to do so).

Finding a Path

To find a path between two points in the world you can use an algorithm called "Breadth First Search" (BFS), more details can be found **here**. The general algorithm for breadth first search is as follows:

1. Add the first node to a `to_explore` queue.
2. Pop the first node from the `to_explore` from the queue, denote as `current_node`.
3. Check if `current_node` is the goal node, if it is: `GOTO: END` 7.
4. Add a triplet of information: `current_node`, the previous node, and the current "depth" (increasing by +1 for each node between this node and the start) to a `visited` list (the previous node and depth are used to be able to backtrack later to generate the path).
5. Check nodes surrounding `current_node` to see if they are valid, if they are valid, add them to the back of the `to_explore` queue, along side the `current_node` to denote where this node was explored from, and +1 to the current depth.
6. `GOTO` 2
7. `END`

Once the end has been discovered, we must then backtrack along the searched nodes to find the path.

1. Set `current_node` to the discovered end node from the BFS algorithm.
2. Append `current_node` to the `path`.
3. If `current_node` is the `start`, `GOTO` 6
4. Loop through nodes in the `visited` list, searching for the parent of `current_node` with one less depth than `current_node`.
5. When found, set `current_node` to the node found in the `visited` list and `GOTO` 2.
6. `END`

Once this loop is finished (and implemented correctly), you will have a list of nodes in reverse order that makes up the shortest path from the start to the end.

Rules for path generation:

- Paths *must* never intersect with houses or the village wall.
- Paths *must* be walkable by a player. Any two adjacent blocks of a single path must not have a height difference larger than 1. If necessary, part of a path can be raised or lowered to gradually descend/ascend harsh slopes, however this *should* be avoided where possible. When the path network is complete, the player should be able to walk from any house to any other house.
- Paths *should* avoid water. If no path exists without crossing water, then you may place additional blocks such that the player can walk across it.

- Paths *should* avoid any changes in height. Of course it is often necessary for a path to have changes in height, but it should always *prefer* a flat route.
- Two paths *may* intersect or cross over.

When implementing the path-finding algorithm, you will need to check these rules when deciding the viable neighbours for a cell. Additionally, to allow your algorithm to prefer certain characteristics (those labeled *should* above), you can add an additional cost value to each cell (rather than have a static +1 depth for each node traversed), which is higher when these characteristics are violated. This will naturally bias the algorithm to avoid undesirable routes, without entirely prohibiting them.

Implementation requirement: In order to find the shortest path you will need to be able to save several different lists of coordinates and related information. **For the Breadth First Search implementation in Finding a Path 3.3.3 you must not use any C++ STL Containers**. Instead you must choose and create a suitable data structure that preserves order and is efficient for adding and removing an unknown quantity of nodes at the start and end of lists. You **must implement an appropriate data structure yourself utilising primitive C++ data types** to store the necessary information.

Building Paths

Paths must be made out of gravel, and thus each block will need to be supported by another block; either the existing ground or an additional solid block.

Building Waypoint Structures

Waypoints need to have some structure to identify them. This may be anything you like. Options include wells, market stalls, village bells, or street lamps. Waypoint structures must not collide with plots, and must not be placed on uneven terrain (you may need to discuss with your team how to handle this).

Figure 7: Example of a waypoint structure

3.4 Testing for Correctness

A core component of software development is ensuring that the code you write adequately meets specifications. As such, before you begin programming you should design **test-cases**. These test-cases should be extensive enough such that you should be confident your program is fully working and meets this nearly 30 page document full of requirements. However, there are two core sources of randomness in this program which must be accounted for: 1) Your program's random algorithms. 2) The Minecraft world. As tests are designed in advance, randomness cannot be handled appropriately. As such, randomness must be removed or addressed. The following section: Test Mode Changes describes how to alter your algorithms for testing. However, we cannot handle the Minecraft world's randomness in this way. As such for the Minecraft world, you should set the worlds seed to 1 (while not ideal is the best we can do), then find suitable locations in that seed for testing. All locations used for testing must be on an unaltered world with a seed of 1, and must be documented in `Tests/TESTING.md`.

3.4.1 Test Mode Changes

When the `--testmode` argument is present the random aspects of the program (not the Minecraft world), should be replaced with fully deterministic and predictable components. **Using a fixed seed is not acceptable**, instead the algorithm itself should change. The purpose of a deterministic change is so that the result can be predicted in advance, while a fixed random seed will give the same result each time, you cannot predict its result in advance.

The design of your program should allow for sub-components to be tested separately, i.e.: task A, task B, and C should be able to be run and tested independently in test mode. This may require changes to the inputs to the program, such as having plot objects, or waypoint locations, or other details be inputted for task B and C to use in order to be independent to task A's functionality.

When running in test mode: additional inputs via command line can be used to input data into the program that would be generated by other components. For example: task A generated plot objects that task B and C require. It may be worthwhile including a mechanism in test mode that can read plot objects from the command line input such that tasks B and C can be run and tested independently from task A.

Task A

In this task there are two potentially random components which must be modified:

1. Plot location selection: This should "scan" over a grid of possible plot locations starting at the bottom-left, ending at the top-right, of the village, in 5 block increments. I.E., given the bottom-left corner of the village is at (x, z) and the village size is v, the first point to check will be in the bottom-left corner +5 blocks in both x and z directions: $(x + 5, z + 5)$. The program should then iterate across the x direction in 5 block increments until it reaches $x + v - 5$. Then add 5 to z, and move back to $x + 5$, and repeat.

2. Plot size selection: This should increment the plot size by 1 each valid plot found across the inclusive range of $(14, 20)$, then wrapping back to 14, after a plot size of 20 is generated.

Task B

As this task relies on plot and terrain generation you may hard-code a series of `plot` objects that can be used to test this section of work. Ensure you document in `Tests/TESTING.md` what scenarios you are testing on and why they were chosen.
In this task there are four potentially random components which must be modified:

1. Building a house: each building made should have an increasing height starting at 6 blocks, and increasing by one each building, with a maximum of 20 blocks. If you included multiple floors, decide on a method of deterministically deciding the number of floors for each building and document in `Tests/TESTING.md`.

2. Room subdivision: Division should occur in the direction of the largest axis, i.e., if a room is 20×10, the length of 20 should be divided. The division should occur as close to equal as possible.

3. Placing doors: doors should be placed as close to the halfway point of the wall as possible.

4. Furniture placement: the program should iterate sequentially over all possible room types such that each room type is chosen, with as little repetition as possible. Within each room the possible furniture types should also be iterated over sequentially. Each piece of furniture should be placed in the top-left corner of the room (provided there are no obstructions).

Task C

As with Task B, Task C also relies on plot and terrain generation. As such you may hard-code a series of `plot` objects that can be used to test this section of work. Ensure you document in `Tests/TESTING.md` what scenarios you are testing on and why they were chosen.
There are no random elements in this task, so no test-mode changes are necessary.

3.4.2 Formal Testing

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. Where `[a]` are the command line arguments for the test which must be documented in `Tests/TESTING.md`.

```
$ ./gen-village --testmode [a] < testname.input > testname.out
$ diff -w testname.expout testname.out
```

The above command takes the content of `testname.input` and redirects it into the program via `std::cin`, and all output of the program via `std::cout` to `testname.out`. Such that you only need to use `cin` and `cout` to perform black-box testing.

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. Your tests should be concise and discrete: each test should only test one edge case of a particular piece of functionality. I.E. testing user input should not simply have one test named "`testing_user_input.input`" that covers all possible inputs, instead they should be separated out into separate files for each edge case such that if a particular test case fails, you can easily identify under what circumstances the program fails.

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).

As part of testing, you must design your program in a way that can be tested and verified. While the functional requirements specify that the output must be in Minecraft, black-box testing is done via CLI, as such it is necessary to create additional outputs to the CLI as well as Minecraft world to ensure all components can be tested. How you do this is up to you, but it must be done with enough detail to be repeatable and useful.

Non-Black-Box Testing

If your task cannot be entirely black-box tested (such as building in Minecraft components or environmental interactions), you may also (but are not required to) include other forms of testing provided that the tests are fully documented. However, **all components must have some black-box testing**. You must think and implement ways of printing out all key information about your algorithm such that your black-box testing can be performed adequately.

A fully documented test must include: the purpose of the test (what is being tested), instructions to fully reproduce the test (program arguments, world seed/location, etc), and explicit instructions on how to validate whether the program operated correctly or not.

3.5 Code Organization and Style

Programming is not solely about making code that "works" but about making code that meets expectations of end users while also being maintainable by future programmers. As such, it is necessary to follow a set of guidelines and conventions while programming

to ensure that other programmers can understand your code and can fix bugs that may occur in the future, or alter the program as the needs of end users shift.

Throughout this assignment you must follow the C++ Style Guide as seen on `Canvas > Modules > Week 5 > C++ Style Guide`.

Throughout this assignment you must also follow a series of programming paradigms discussed in week 5 class 2.

1. Defensive programming for user inputs,
2. Structured programming paradigms for general code structure,
3. Programming by contract for data passing between functions.

3.6 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:** Task A — Finding House Plots & Terraforming
    1. Implement `village plot finding and validation`, handling all related error cases.
    2. Implement `plot terraforming`, which involves constructing the base of the plot and smoothing the terrain around the plot.
    3. Implement `village wall placing`, which involves placing a wall to surround the village.
    4. Implement `waypoint location selection`, which involves determining and selected valid locations for waypoints in the village.
    5. Develop black-box test cases to verify the correctness of the above functionalities.
- **Team Member 2:** Task B — Building Houses
    1. Implement the `house exterior building`, which involves deciding on the height and creating the walls and roof of each building in the village.
    2. Implement the `house interior building` feature using the Recursive Subdivision algorithm.
    3. Implement `house decoration` features, involving placing doors between subdivided rooms and placing furniture in each room.
    4. Develop black-box test cases to verify the correctness of the above functionalities.
- **Team Member 3:** Task C — Pathfinding

1. Implement `breadth first search` with a custom data structure, for finding the shortest path between two points in the Minecraft world.
2. `Finding paths` to connect waypoints together and to connect houses to waypoints.
3. `Building paths` in the Minecraft world.
4. `Building waypoint structures` in Minecraft.
5. 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.

3.7 Verbiage

- *Semi-random*: You can choose to either select completely at random from the available set of choices, or pick with a custom strategy that you believe will achieve better results (as long as it's non-deterministic).

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 gen-village ./*.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.

There are four main points each student's section should address:

1. What task you worked on and demonstrate its function.
2. Use of ADTs in your program and why you chose them.
3. How you tested your task's functionality.
4. Use of defensive programming, structured programming, and programming by contract.

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
`main.cpp`	The main file; holds the overall logic for the program.
`paths.h`	File to hold implementation details for path finding.
`plots.h`	File to hold class structure for plots.
`utils.h`	File to hold minor supporting code.
`task_(a/b/c).(h/cpp)`	Files to hold structure of the different tasks.
`Makefile`	Simple Makefile for compiling.
`README.md`	File to add student details, task allocation, etc.
`Tests/*`	Contains all files for tests and testing details.
`Tests/TESTING.md`	Details about what testing you have done.

The provided code should compile and produce a outline of the program. You are free to (and should) 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 a week before the deadline. 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. These checkpoitns are also used to gauge your understanding of your own assignment, you may be asked to explain your work and demonstrate that you fully understand your final submission.

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 (10 marks)** — Assessed individually. Each student must create black-box test cases that thoroughly evaluate the components of their assigned tasks (see Section 3.6). 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 a `TESTING.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.
  - 10 marks: Tests comprehensively cover normal and edge cases.
- **Implementation (20 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.
  - 10 marks: Most functionality is implemented, but there are logical errors or missing edge cases.
  - 15 marks: Functionality is complete and correct, but the code lacks efficiency, uses suboptimal data structures, or does not follow the style guide.
  - 20 marks: Fully meets all specifications, is efficient, uses appropriate data structures, and adheres to the style guide.

- **Code organization and style (10 marks)** — Assessed individually. Each student is responsible for writing code that follows the recommendations/requirements in Section 3.5, notable by following the style guide and properly applies the main programming paradigms (defensive, structured, and contractual).

  - 0 marks: Code is difficult to read and understand, with no or minimal proper use of following the style guide and programming paradigms.
  - 4 marks: Some meaningful attempts at following the programming paradigms and adhering to the style guide.
  - 6 marks: Most code is understandable and well documented with mostly proper use of programming paradigms.
  - 10 marks: Code is highly understandable with proper use of all required programming paradigms, and no violations of the style guide.

- **Integration (5 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.
  - 3 marks: Program compiles and functions correctly, but integration is inefficient or poorly modularized.
  - 5 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.