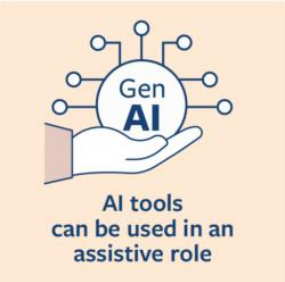
UNIVERSITY OF LEEDS



School of Computing: Assessment brief

Module title	Programming Project
Module code	XJCO1921
Assignment title	Assignment 2 – Project Code
Assignment type and description	You will now implement the maze game, using test-driven development to create a defensively designed program following a given specification.
Rationale	Using modern and professional development techniques, you will work on a simple project demonstrating your understanding of different techniques, such as dynamic memory allocation and structure usage. You will focus on creating good quality code which is well documented, modular, and maintainable; these are all important considerations when writing code in the industry.
Word limit and guidance	You should spend 16-20 hours working on this assessment.
Weighting	60%
Submission deadline	Thursday, 15 May 2025, 12:00 UK BST (19:00 Beijing Time)
Submission method	Gradescope
Feedback provision	Marked rubric
Learning outcomes assessed	- apply professional programming practices to programming projects. - design, implement, debug and test a modular programming solution to a real-world problem.
Module lead	Zheng Wang, Xiuying Yu

1. Assignment Guidance

You will produce, in C, a program which fits the following specifications:

Maze Game

```
Usage: ./maze <mazefile>
```

You are creating a basic game where players navigate through a maze.

The maze will be loaded from a file, the filename for which is passed as a command line argument. Mazes are made up of four characters:

Mazes are made up of four characters:

Character	Purpose
'#'	A wall which the player cannot move across
' ' (a space)	A path which the player can move across
'S'	The place where the player starts the maze
'E'	The place where the player exits the maze

A maze has a height and a width, with a maximum of 100 and a minimum of 5. Your program will **dynamically allocate** an appropriate data structure to store the maze.

The height and width do not have to be equal – as long as both are within the accepted range.

Within a maze, each 'row' and 'column' should be the same length – the maze should be a rectangle.

When the game loads, the player will start at the starting point ‘S’ and can move through the maze using WASD movement:

Key	Direction
W/w	Up
A/a	Left
S/s	Down
D/d	Right
Q/q	Quit the game

Note: For this assignment, each input will be separated with a newline character – this is not keypress triggered. This is for the purpose of autograding.

The player can move freely through path spaces (‘ ‘) but cannot move through walls or off the edge of the map. Some helpful prompts (messages) should be provided if this is attempted.

The map should NOT be shown to the player every time they make a move (again for the purpose of autograding), but they can enter‘M’/’m’ to view an image of the map, with their current location shown by an ‘X’.

When the user reaches the exit point ‘E’, the game is over and will close. The player should be given some message stating that they have won. There is no ‘lose’ condition.

Maze file specification

A valid maze:

- Has a single starting point ‘S’
- Has a single exit point ‘E’
- Contains only the start and exit characters, spaces (‘ ‘), walls (‘#’), and newline (‘\n’) characters
- Has every row the same length
- Has every column the same height
- Has a maximum width and height of 100
- Has a minimum width and height of 5
- Does **not** require every row and column to start or end with a ‘#’
- May have a trailing newline at the end of the file (one empty row containing only ‘\n’)

A selection of valid mazes is provided in your starting repository – you should ensure that your code accepts all these mazes.

Note that file extension is not important – there is no requirement for a maze file to be stored as a .txt file provided that the contents of the file are valid.

Standard Outputs

To allow some automatic testing of your functionality, we require some of your outputs to have a specific format. To prevent you from being overly restricted, this will only be the **final returned value** of your code rather than any print statements.

Return Codes

Scenario	**Value to be returned by your executable**
Successful running	0
Argument error	1
File error	2
Invalid maze	3
Any other non-successful exit **Note: it is unlikely that you will need to use this code**	100

Maze Printing Function

The maze printing function (‘M’/’m’) **must** output the maze in the following way:

- No additional spaces added
- Newline before the first row is printed
- Newline after the final row
- If the player’s current position overlaps with the starting point, this should display ‘X’ rather than ‘S’

The code required to do this is provided in the template as print_maze() and may be used without referencing me.

Additional Challenge Task – Maze Generator

This is an optional additional task that will involve researching and developing a more complex piece of code – you do not need to complete this section to achieve a 2:1/2:2 grade. This task may take longer than the recommended time given above – I recommend only attempting any part of it if you found the original task trivial to complete.

The task

In addition to allowing users to solve mazes, you will create an additional program `mazegen` which allows users to generate a **valid** and **solvable** maze with the specified width and height, to be saved in 'filename'.

For example:

```
./mazeGen maze4.txt 20 45#
```

will save a maze that is 20 x 45 into 'maze4.txt', creating that file if it does not already exist.

A valid maze means that it fits the rules given in the "maze file specification section", as well as being **solvable** (there is at least one solution to the maze- it is possible to start at S and exit at E).

There are some existing algorithms that can create mazes, and you should experiment with using these to produce 'quality' mazes that are not trivial to solve and present some challenges to the player. You should **document** your process of developing the maze creation algorithm, as this will form a part of the assessment.

It is recommended that you keep a log including some maze files generated by each iteration, what you intend to change for the next iteration based on these maze files, and just some general comments about what you think was good or bad about this particular solution.

Some things to consider for each iteration are:

- Did the program produce a variety of designs of maze?
- Did the program produce only valid mazes?
- How did the program perform with larger dimensions (100 x 100 for example)
- What did the program do particularly well?
  - Can you identify what part of the code caused this?
- What did the program do particularly poorly?
  - Can you identify what part of the code caused this?
- What will you try next time?

For this task, you will present your maze generation program to a member of the module team during a lab session and discuss:

- How your program works
- How you iteratively developed it
- The limitations of your solution
- Any improvements you would like to make to it in the future

2. Assessment tasks

Produce the C code for a program that solves the tasks detailed above.

You should ensure that your code is:

- Structured sensibly
- Modular
- Well-documented
- Defensive
- Working as intended

You can use the code skeleton you produced in Assignment 1, or a basic skeleton is provided via Minerva.

You can use any number of additional header and C files and should produce or adapt a makefile that allows the program to compile successfully on a GitHub Codespace instance. You may not use any non-standard C libraries. **You should make sure your code runs in Linux.**

You should also use your test script and data from assignment 1 to help you produce defensive and working code. You can adapt and add to these throughout your development process. If you did not create a test script, or your test script does not work, then you can manually test your code.

Good programming practices

You should follow good software development practices. For this exercise, you are asked to:

- Follow modular development by making sure the code is modular and well structured; code with proper comments
- Use Makefile for code compilation;
- Use a git repository for version control

Notes:

1. **Makefile**: You should also submit a Makefile with at least two targets: *all* and *clean*. "*make all*" compiles your code to generate an executable binary, while "*make clean*" removes all object files (.o), all executables, and any *temporary* files created during your run.
2. **Version control**: We will check the commit logs of your git repository. We expect to see steady progress towards completion, as revealed in the pattern of git commits. One of the implications of this is that we will be penalising any student who develops their code without git and then dumps it all into git at the last minute.

3. General guidance and study support

You should refer to the lab exercises and lecture notes to support you. Use the lab sessions to ask for help.

4. Assessment criteria and marking process

A full breakdown of the assessment criteria can be found in section 8.

Your code will be tested with different maze files and user inputs containing errors - the exact nature of these errors will not be told to you before marking, so ensure that you validate a wide range of potential user errors. Note that we will be running our own test cases to mark the assignments and doing our best to devise creative ways to break your code! You should therefore develop your test cases to beat us to it.

Your code will be manually checked for code quality.

If you complete the additional challenge task, you will submit your code for plagiarism checking but will present your code to a member of the SWJTU module staff for assessment.

5. Presentation and referencing

If you need to reference any resources, use a simple comment, for example:

```
// This test is adapted from an example provided on https://byby.dev/bash-exit-codes
```

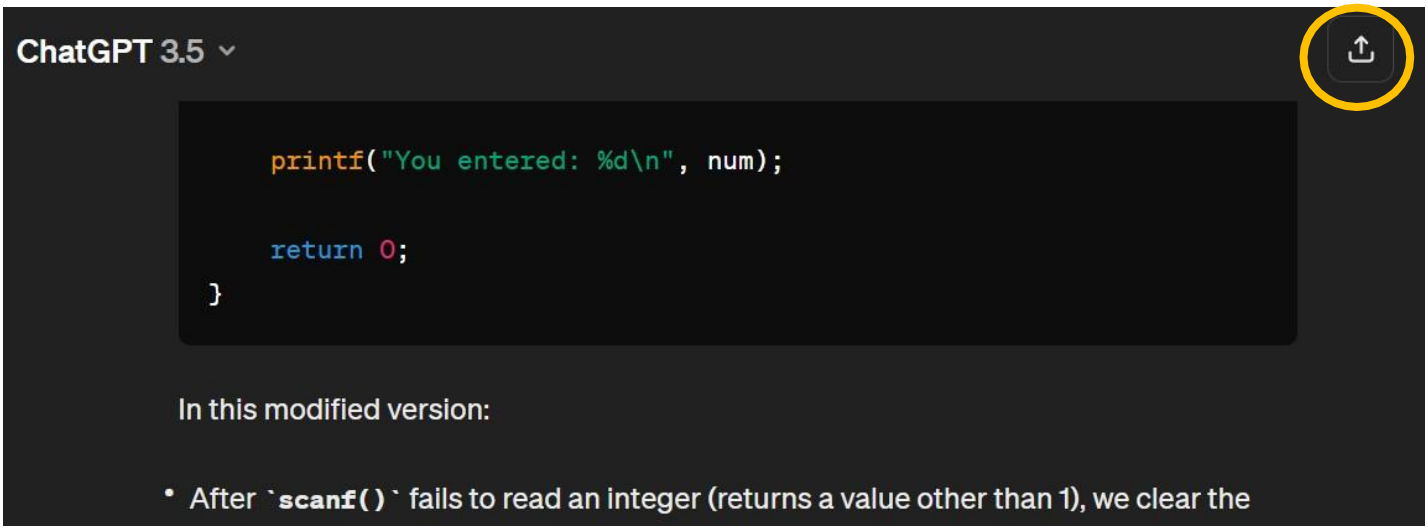
You should not be directly copying any code from external resources, even with a reference.

<u>Use of Gen AI (Generative Artificial Intelligence) instructions</u>:

This assessment is an amber category. AI tools can be used in an assistive role. Under this category, you are permitted to use AI tools for specific defined processes within the assessment: ChatGPT and Copilot

If you are referencing a Generative AI model, you must provide the full conversation.

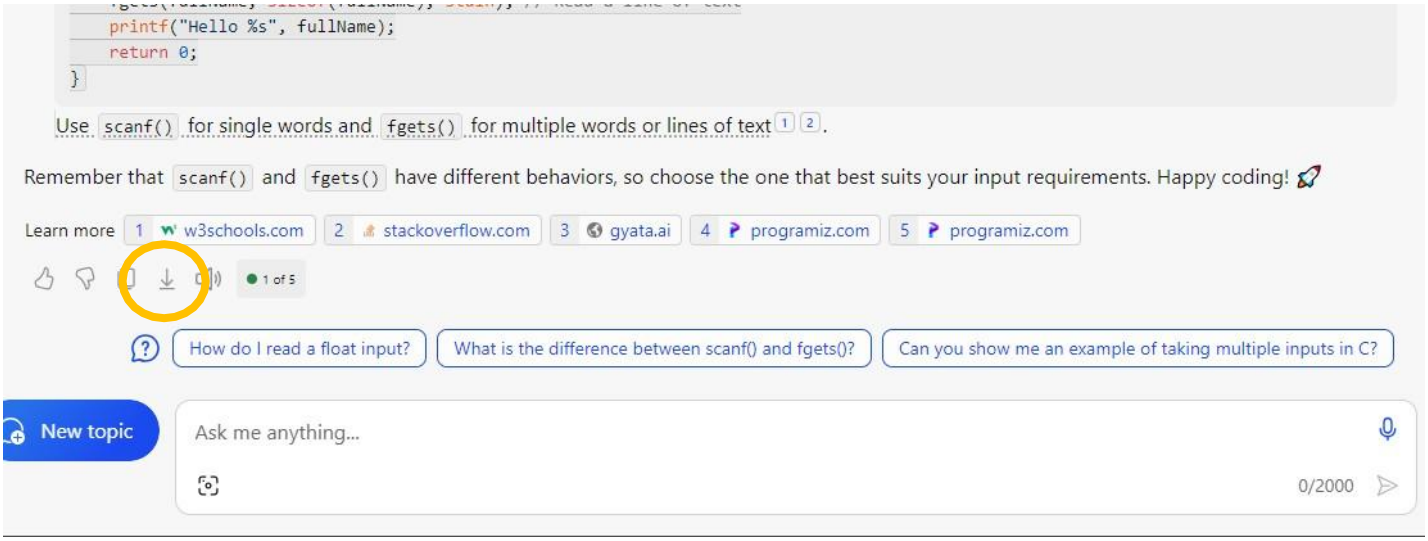
In **ChatGPT**, you can generate a link to the full conversation:



And provide the reference as follows:

```
// Lines 1 – 7 were adapted from code provided by the following conversation with
chatGPT: https://chat.openai.com/share/c356221d-fb88-4970-b39e-d00c87ae1e0b
```

In **Copilot**, you will need to export the conversation as a text file:



Save this with a filename including the date and 2-3 word summary of what the conversation was about (’11-03 inputs in C.txt’) and ensure this is submitted with your work.

\You can reference this in your code:

```
// Lines 1 – 7 were adapted from code provided by the CoPilot conversation
recorded in ’11-03 inputs in C.txt’
```

If you are using a different Generative AI model, these instructions may differ – you must still provide a link to or copy of the full conversation and reference in the same manner above.

Use of Generative AI in this Assessment

This assessment is rated ‘amber’ according to the university guidelines around generative AI.

This means that you can use genAI models such as ChatGPT or CoPilot to **explain concepts** that may be useful in this assessment, but you **must NOT ask it to write your code for you** nor **give it any part of my specification**.

The following link is an example of what I would consider ‘reasonable use’ of chatGPT for this assessment:

https://chat.openai.com/share/c356221d-fb88-4970-b39e-d00c87ae1e0b

6. Submission requirements

Submit your source code via Gradescope. There is a separate submission point for the extension work.

Ensure that:

- Any .c or .h files are not inside a subdirectory
- The makefile is not inside a subdirectory
- Your executables are named: maze and mazegen
- You have followed the return code instructions above
- Your code compiles on Linux

You will receive some instant feedback which should confirm that your upload is in the correct format and is using the correct return values – please ensure you correct any failing tests.

Note: Passing these tests is not a guarantee that your code will gain full marks from the autograder – just that it is the correct format/returns for the grader to run.

7. Academic misconduct and plagiarism

Leeds students are part of an academic community that shares ideas and develops new ones.

You need to learn how to work with others, how to interpret and present other people's ideas, and how to produce your own independent academic work. It is essential that you can distinguish between other people's work and your own, and correctly acknowledge other people's work.

All students new to the University are expected to complete an online Academic Integrity tutorial and test, and all Leeds students should ensure that they are aware of the principles of Academic integrity.

When you submit work for assessment it is expected that it will meet the University's academic integrity standards.

If you do not understand what these standards are, or how they apply to your work, then please ask the module teaching staff for further guidance.

By submitting this assignment, you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

8. Assessment/ marking criteria grid

Category	1st	2:1 / 2:2	Pass / 3rd	Fail
Basic Task (70)				
Auto-graded Questions				
Functionality (25)	Fully functional, meeting all specified requirements.	Mostly functional but with minor bugs or incomplete features.	Core functionality has been implemented with some major errors.	Significant errors, or does not meet specification.
Defensive design (10)	Robust error handling and defensive programming practices were implemented	Adequate error handling but lacks coverage for certain edge cases.	Error handling for a variety of cases has been implemented. Code does not regularly crash.	Lack of defensive programming; code is prone to errors or crashes.
Manually Marked Questions				
Code Structure (10)	Well-organized structure that enhances readability and comprehension.	Clear structure but lacks some cohesion affecting readability.	Structure exists but causes confusion or detracts from readability.	Lack of coherent structure, code is hard to follow.
Documentation (Comments / readme) (10)	Thorough documentation, including explanations of code logic and usage.	Adequate documentation, but some parts lack clarity or completeness.	Limited documentation, making it difficult to understand code intentions.	Absence of documentation which makes code unreadable.
Modular breakdown (5)	Clearly defined modules with distinct functionalities, promoting easy maintenance.	Modules mostly defined but may lack clear separation or functionality.	Poor division into modules, leading to confusion or inefficiencies.	Lack of modular design; code is monolithic and hard to manage.
Memory Management (5)	Optimal memory handling; efficient allocation and deallocation with no leaks.	Mostly efficient memory handling but may have minor leaks or inefficiencies.	Inefficient memory handling causing noticeable leaks or performance issues.	Severe memory leaks or grossly inefficient memory usage.

Use of git for version control (5)	A good use of git for version control with steady updates	Use of GitHub but with occasional commits toward the end of the deadline	One or two git commits before submission	No use of git for version controls

Maze Generator (30)				
Functionality (10)	Generates high-quality, challenging mazes meeting all criteria.	Mostly generates challenging mazes but may have some limitations.	Generates mazes with significant limitations or are too simplistic.	Fails to generate valid or challenging mazes.
Algorithm Development (20)	It is clear how the final algorithm was reached, with a clear progression from more simple algorithms to a final algorithm. Students are able to articulate this development with reflective language and clear justifications for choices made. Students were able to articulate the limitations of the solution and any further work which could improve it.	There is a clear progression of algorithm design, from a simpler algorithm to a more complex final algorithm that builds on the previous solutions. Students can articulate this development but may be lacking in reflective thinking or clear justifications. The student has made some attempts to articulate limitations and further work.	Some evidence of progression to a final solution, though students may struggle to explain and justify choices made and lack reflection. Students can explain basic, surface-level limitations.	No evidence of any progression to the solution, or the student is unable to explain how the algorithm was developed.