CAB201 Object Oriented Design and Implementation

5/31/2024

48.5/50 Points

Offline Score: **48.5/50**

View feedback

Anonymous grading: **no**

3/11/2024

Details

Overview

Task description:

Our client is an information-gathering agency that we cannot name for legal and contractual reasons. This client employs field agents that often find themselves gathering information in highly secure facilities.

Your task is to write a computer program in C# and .NET 8.0 that presents a menu allowing the agent to specify certain obstacles to the agent's mission, and provides the agent with information upon request, including the ability to request a safe path to a particular objective.

You are required to use an object-oriented approach to implementing the solution, and you are required to produce high quality, thoroughly documented source code so that the software can be maintained later. Your code must also feature proper use of exception handling.

Unit Learning Outcomes assessed:

ULO 3: Document software designs and computer code to ensure it is easy to maintain and complies with industry standards.

ULO 4: Apply object-oriented design and programming techniques to create software solutions.

Overview of assessment item

Estimated time for completion	Weighting	Group or Individual	How I will be assessed
25-50 hours	**50%** of final grade	Individual	See marking criteria below

What you need to do

Your software will present a command-based user interface to the agent, displaying a list of commands and presenting an interface for the inputting of commands. Different commands will require different parameters, that will be entered by the user and checked for correctness by your program. The user will enter in a command to direct the software to perform one of the available actions. Here is one such example of a session with a user (user input is shown ***in bold and italics***):

```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
```

```
add guard 2 1
Successfully added guard obstacle.
Enter command:
check 2 2
You can safely take any of the following directions:
North
East
West

Enter command:
exit
Thank you for using the Threat-o-tron 9000.
```

In this example, the user specifies that they know there is a guard located at (2,1) - which means 2 klicks east and 1 klick north of the map origin. The user then requests a list of safe directions, giving their own location as (2,2) - 2 klicks east and 2 klicks north of the origin. The program then reports that safe directions to travel in are **north**, **east** and **west** (as going south would result in the agent being caught by the guard. The user then enters `exit` to close the program.

Another use of the program is navigation:

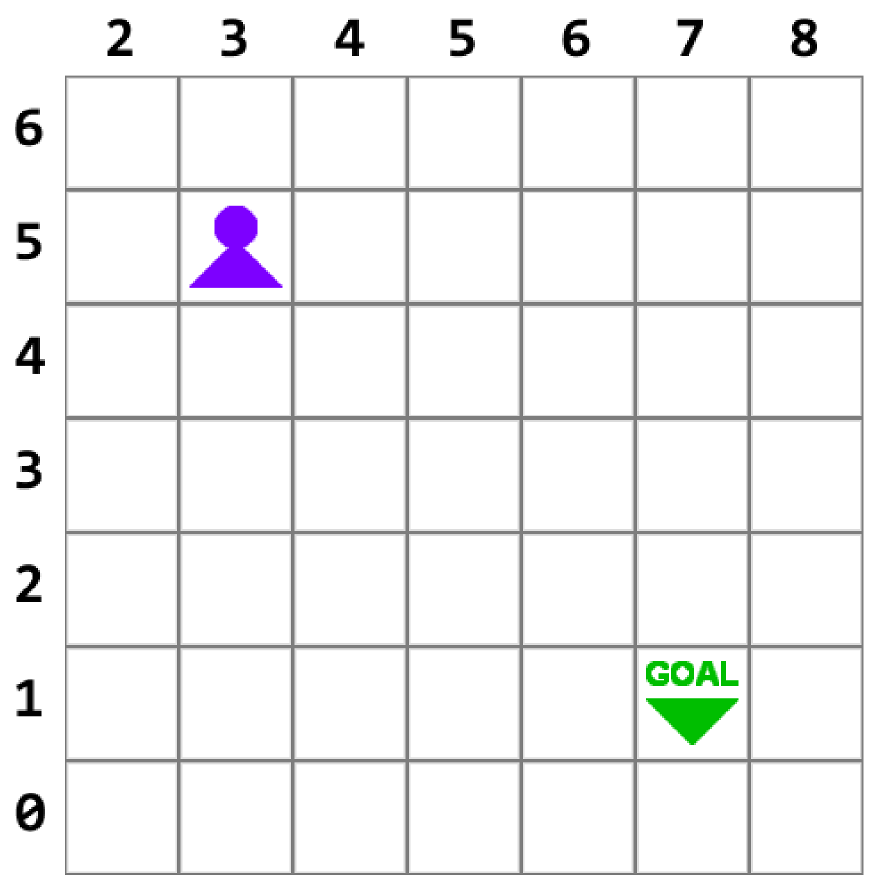
```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
path 3 5 7 1
The following path will take you to the objective:
Head south for 4 klicks.
Head east for 4 klicks.
Enter command:
exit
Thank you for using the Threat-o-tron 9000.
```

In this example, the agent is located 3 klicks east and 5 klicks north of the origin point, and the mission objective is located 7 klicks east and 1 klick north of the origin point. The agent has not observed any obstacles to the mission. The system then recommends that the agent go south for 4 klicks and east for 4 klicks.

Visualised, this problem would look like this:



(Note that the dimensions are not restricted to the 7x7 area depicted above – this is purely for the purposes of this visualisation)

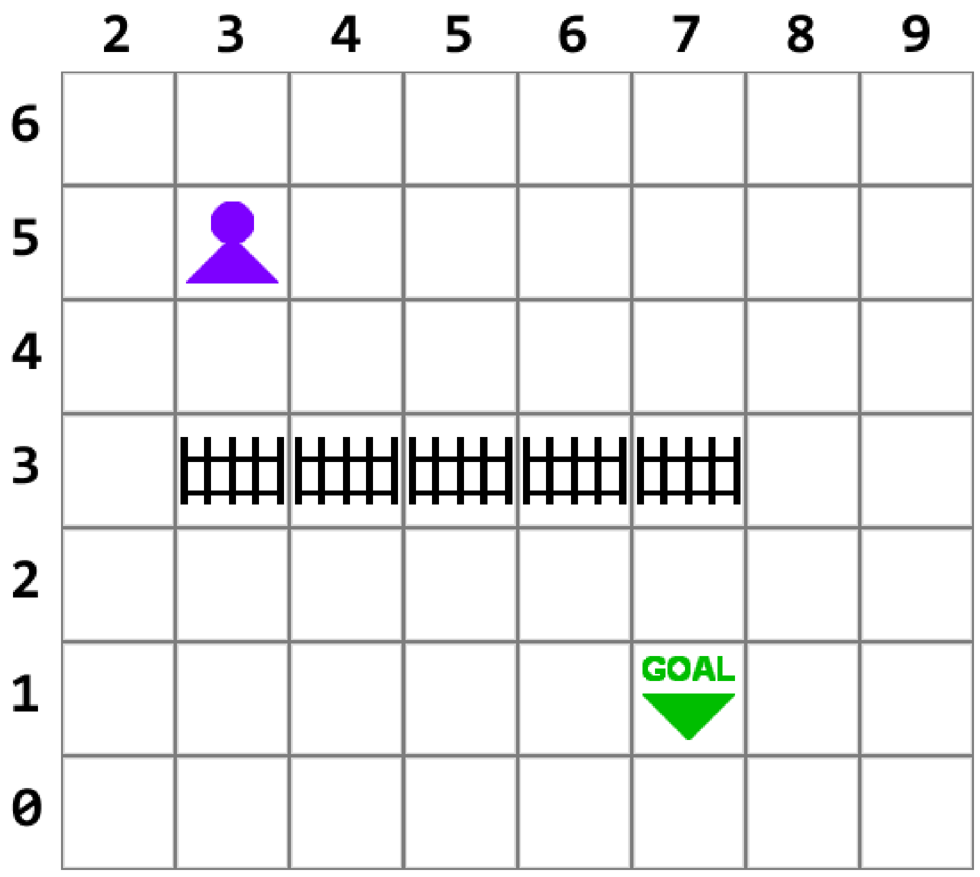
As there is nothing blocking the agent's path, any path that results in the agent moving 4 klicks south and 4 klicks east will be an acceptable solution. A path that results in the agent getting some of the way (but not all of the way) towards the goal will result in partial credit.

Most of the time, getting to the objective will be less straightforward, as there will be obstacles obstructing the agent's path. Each obstacle requires the agent to specify different information when adding it to the scenario. This information is supplied in the form of parameters, which make up part of the command entered by the agent. The previous example (a guard) is an obstacle that occupies a single location. Another example of an obstacle is a fence:

```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
add fence 3 3 east 5
Successfully added fence obstacle.
Enter command:
path 3 5 7 1
The following path will take you to the objective:
Head south for 1 klick.
Head east for 5 klicks.
Head south for 3 klicks.
Head west for 1 klick.
Enter command:
exit
Thank you for using the Threat-o-tron 9000.
```



In this case, the agent has observed a 5-klick long east-oriented fence at a location of 3 klicks east and 3 klicks north. The fence is considered to obstruct all of the squares that it occupies, so the agent will have to move around it (the agent cannot enter the squares at 3,3 or 4,3 or 5,3 or 6,3 or 7,3 at all).

The software can also be used to provide a simple text-based visualisation of the obstacles that have been defined so far:

```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
add fence 3 3 east 5
```

```
Successfully added fence obstacle.
Enter command:
add guard 4 5
Successfully added guard obstacle.
Enter command:
add guard 6 5
Successfully added guard obstacle.
Enter command:
map 0 0 9 8
Here is a map of obstacles in the selected region:
.........
.........
....G.G..
.........
...FFFFF.
.........
.........
.........
Enter command:
exit
Thank you for using the Threat-o-tron 9000.
```

In this case, a map is displayed of the specified area, showing a `G`, `F`, `S` or `C` depending on whether that location is protected by a guard, fence, sensor or camera, and a `.` for all safe locations.

User interface

Your software will be tested via Gradescope and will therefore need to implement the following user interface requirements reasonably **precisely**. Remember that you can test your software by submitting via Gradescope at any time, and it is highly encouraged that you do so **often**. - minor issues can be caught very easily, but if you wait until the last week to submit you may find yourself losing many marks due to minor deviations from the spec.

Your program will begin by displaying a welcome message, followed by a listing of commands.

The welcome message is **optional** and you can display whatever you like here. In these examples we have chosen to go with "Welcome to the Threat-o-tron 9000 Obstacle Avoidance System." - feel free to put what you like here.

The program will then display 'Enter command:' and the user will be able to type their command in on the following line.

```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
```

The program will then wait for input from the user. If the first word of the user's input is something other than `add`, `check`, `map`, `path`, `help` or `exit`, the program will display `Invalid option: ` followed by the word entered, followed by the line `Type 'help' to see a list of commands.`, followed by `Enter command:` again.

```
Welcome to the Threat-o-tron 9000 Obstacle Avoidance System.

Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
blah
Invalid option: blah
Type 'help' to see a list of commands.
Enter command:
hello world
Invalid option: hello
Type 'help' to see a list of commands.
Enter command:

Invalid option:
```

```
Type 'help' to see a list of commands.
Enter command:
```

If the provides input corresponding to a valid command, the program's next action will depend on the command entered.

`add` (Add obstacle)

The `add` command is expected to be followed by the name of an obstacle: `guard`, `fence`, `sensor` or `camera`, followed by parameters that depend on the obstacle chosen. For details on each obstacle's parameters and how the obstacles function, see the Obstacles section below. Each obstacle will accept different parameters, and will obstruct the agent in different ways. For instance, if the obstacle is a guard, its parameters will specify the guard's location. For a fence, parameters are used to specify its origin, length and whether the fence is facing north or east. For a security camera, its location and direction.

If `add` is not followed with any parameters, the message `You need to specify an obstacle type.` will be displayed:

```
Enter command:
add
You need to specify an obstacle type.
Enter command:
```

If the first parameter to `add` is not one of the four obstacle types, the message `Invalid obstacle type.` will be displayed:

```
Enter command:
add pit
Invalid obstacle type.
Enter command:
```

Any number of obstacles can be specified in this way, and the same type of obstacle can be specified multiple times - e.g. there might be multiple fences.

`check` (Show safe directions)

The `check` command is expected to be followed by a pair of coordinates, specifying the agent's location (or a location the agent plans to arrive at):

```
check 0 0
```

If `check` is followed by fewer than 2 parameters or more than 2 parameters, the program will display `Incorrect number of arguments.`

If either or both parameters are not valid 32-bit signed integers, the program will display Coordinates are not valid integers.

```
Enter command:
check 5
Incorrect number of arguments.
Enter command:
check 2 4 8
Incorrect number of arguments.
Enter command:
check a b
Coordinates are not valid integers.
Enter command:
check 9999999999999 4
Coordinates are not valid integers.
Enter command:
```

If the location specified is blocked by an obstacle (e.g. a guard was specified at 0,0, or an east-oriented 7-klick long fence was specified at -3,0, the program will display `Agent, your location is compromised. Abort mission.`

```
Enter command:
add guard 0 0
Successfully added guard obstacle.
Enter command:
check 0 0
Agent, your location is compromised. Abort mission.
Enter command:
```

Otherwise, if the location is itself safe but obstructed on all four sides by obstacles, the program will instead display:

```
You cannot safely move in any direction. Abort mission.
```

Otherwise, the program will display the text `You can safely take any of the following directions:` followed by a list of directions (in the order `North`, `South`, `East`, `West`), each on its own line, that the agent can safely move in from their current location. If all four directions are unobstructed:

```
You can safely take any of the following directions:
North
South
East
West

Enter command:
```

If, on the other hand, there are obstacles blocking the squares north and east of the agent's location:

```
You can safely take any of the following directions:
South
West

Enter command:
```

`map` (Display obstacle map)

This command is followed by four parameters - the coordinates of the south-west cell of the map and the width and height of the map, in klicks.

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the coordinates are not both valid signed 32-bit integers, the program will display `Coordinates are not valid integers.`
- If the width and height of the map are not both valid signed 32-bit integers with values greater than 0, the program will display `Width and height must be valid positive integers.`

Otherwise, the program will display the line `Here is a map of obstacles in the selected region:` followed by a text-based map of the current obstacle configuration. This map will be oriented so that the top-left character printed corresponds with the north-western-most cell of the map, and the bottom-right character printed corresponds with the south-eastern-most cell of the map. Hence, the coordinate pair supplied by the user corresponds with the character printed in the bottom-left of the text map.

```
Enter command:
map 0 0 8 8
Here is a map of obstacles in the selected region:
........
........
........
........
........
........
........
........
Enter command:
```

Valid characters that can appear in the map are: `.` `G` `F` `S` `C`, for safe squares, guards, fences, locations protected by sensor and locations protected by camera respectively.

`path` (Find safe path)

This command is followed by four parameters - the coordinates of the agent's current location and the coordinates of the agent's mission objective.

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the agent coordinates are not both valid signed 32-bit integers, the program will display `Agent coordinates are not valid integers.`
- If the objective coordinates are not both valid signed 32-bit integers, the program will display `Objective coordinates are not valid integers.`
- If the mission objective's location is equal to the agent's current location, the program will print `Agent, you are already at the objective.`
- If the location of the mission objective is obstructed by an obstacle (e.g. the objective is at 3,3 and there is a guard at 3,3) the program will print `The objective is blocked by an obstacle and cannot be reached.`

If there is a safe path between the agent's location and the objective, the program will print out:

```
The following path will take you to the objective:
```

...on a line by itself, followed by a series of lines consisting of `Head <direction> for <distance> klick(s).`, indicating that the agent must travel in that direction for that number of klicks. (If the distance is 1, 'klick' must be written - if it's greater than 1, 'klicks' must be written.)

```
Enter command:
path 0 0 4 4
The following path will take you to the objective:
Head north for 4 klicks.
Head east for 4 klicks.
Enter command:
```

If there is no safe path to the objective (due to obstacles), the program will print `There is no safe path to the objective.`

The path must navigate around any obstacles, ensuring that the agent is not obstructed or spotted. For example, if there is a guard on the most direct path, the path given must navigate around this guard:

```
Enter command:
add guard 3 0
Successfully added guard obstacle.
Enter command:
path 0 0 5 0
The following path will take you to the objective:
Head north for 1 klick.
Head east for 4 klicks.
Head south for 1 klick.
Head east for 1 klick.
Enter command:
```

`exit` (Exit program)

If `exit` is entered, the program will terminate, displaying a farewell message. You can write whatever you like for the farewell message - in this example, we have gone with `Thank you for using the Threat-o-tron 9000.`:

```
Enter command:
exit
Thank you for using the Threat-o-tron 9000.
```

`help` (Show a list of valid commands and their parameters)

If `help` is entered, the program will display the same list of valid commands that is displayed when the program is initially started:

```
Enter command:
help
Valid commands are:
add guard <x> <y>: registers a guard obstacle
add fence <x> <y> <orientation> <length>: registers a fence obstacle. Orientation must be 'east' or 'north'.
add sensor <x> <y> <radius>: registers a sensor obstacle
add camera <x> <y> <direction>: registers a camera obstacle. Direction must be 'north', 'south', 'east' or 'west'.
check <x> <y>: checks whether a location and its surroundings are safe
map <x> <y> <width> <height>: draws a text-based map of registered obstacles
path <agent x> <agent y> <objective x> <objective y>: finds a path free of obstacles
help: displays this help message
exit: closes this program

Enter command:
```

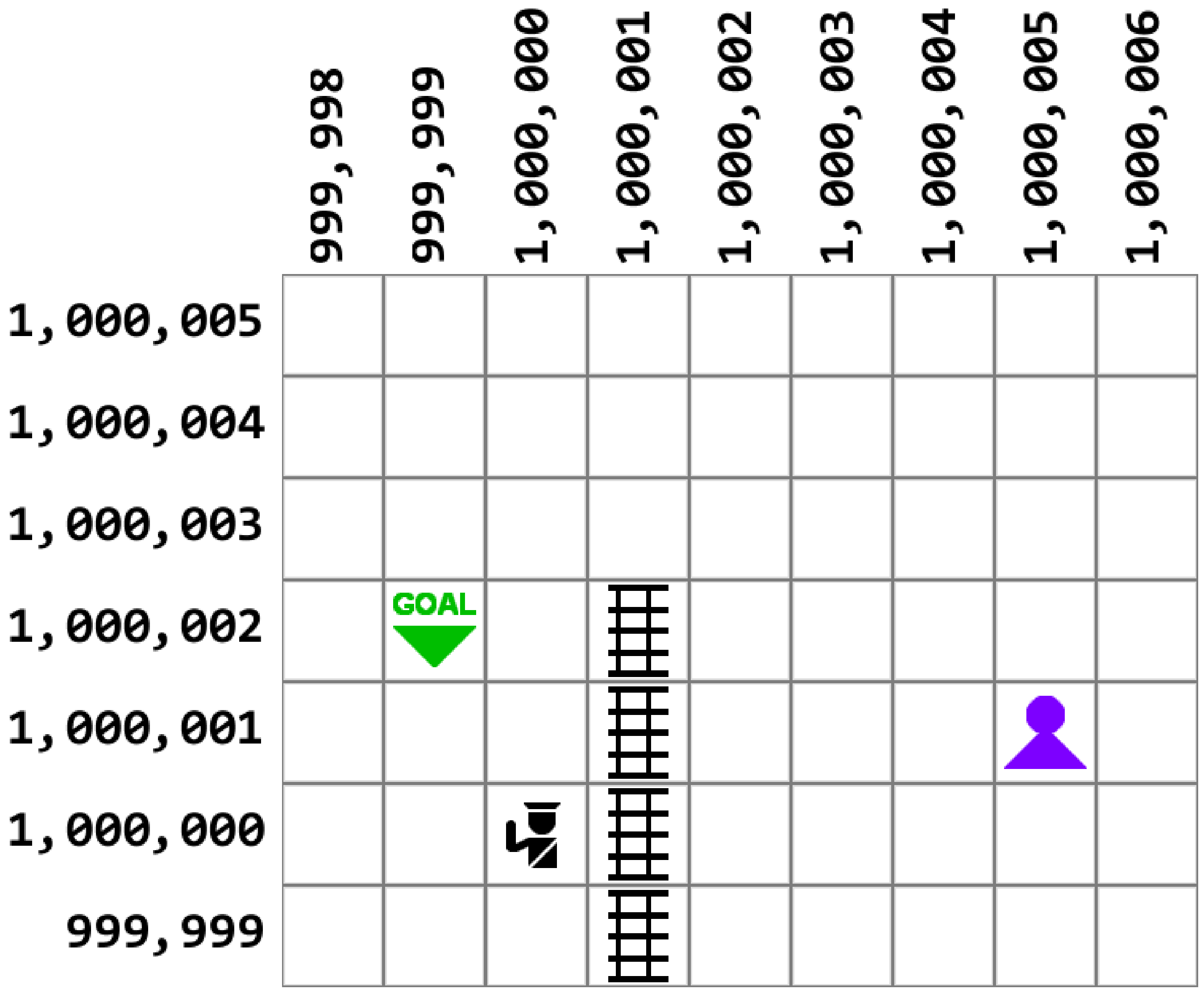
The help text needs to be identical to the text shown here.

Coordinate system

This program utilises integer Cartesian coordinates, where the X coordinate indicates that many klicks **east** of an arbitrary origin point and the Y coordinate indicates that many klicks **north** of that point. **Negative coordinates are acceptable**. These scenarios can be considered to take place on an arbitrarily large plane, so coordinates can feature very large numbers (providing they fit within the range of a signed 32-bit integer.) You will need to take this into account when deciding how to store obstacles, draw maps and calculate safe paths:

```
Enter command:
add guard 1000000 1000000
Successfully added guard obstacle.
Enter command:
add fence 1000001 999999 north 4
```

```
Successfully added fence obstacle.
Enter command:
path 1000005 1000001 999999 1000002
The following path will take you to the objective:
Head north for 2 klicks.
Head west for 5 klicks.
Head south for 1 klick.
Head west for 1 klick.
Enter command:
```



Obstacles

There are a variety of obstacles that may be present. Obstacles are individually configured - for example, the Fence obstacle has a location, an orientation and a length - and they obstruct the agent in different ways. Your program is expected to take an object-oriented approach to handling these obstacles, such that new obstacles could be added to your program with minimal effort and disruption to the rest of the program.

This section presents a catalogue of different obstacles that may be present:

`add guard <x> <y>`

Guards take a pair of coordinates indicating the location of the guard.

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the coordinates are not both valid signed 32-bit integers, the program will display `Coordinates are not valid integers.`

The guard will obstruct the agent from entering the specified square. For instance, if the guard is located at (7,3), the agent will not be able to enter that square (and will be given paths that route around it, for example).

Upon successfully adding a guard, the message `Successfully added guard obstacle.` will be displayed.

`add fence <x> <y> <orientation> <length>`

Fences take four parameters - a pair of coordinates, an orientation (which must be either `east` or `north`) and a length (which must be at least 1).

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the coordinates are not both valid signed 32-bit integers, the program will display `Coordinates are not valid integers.`
- If the orientation is not east or north, the program will display `Orientation must be 'east' or 'north'.`
- If the length is not a valid signed 32-bit integer with a value greater than 0, the program will display `Length must be a valid integer greater than 0.`

The fence will then extend from the location indicated by the coordinates, in the specified direction, to cover a number of cells indicated by the length.

The fence will obstruct the agent from entering any square within its bounds. For example, a north-oriented fence at (4,3) with a length of 4 will stop the agent from entering (4,3), (4,4), (4,5) and (4,6).

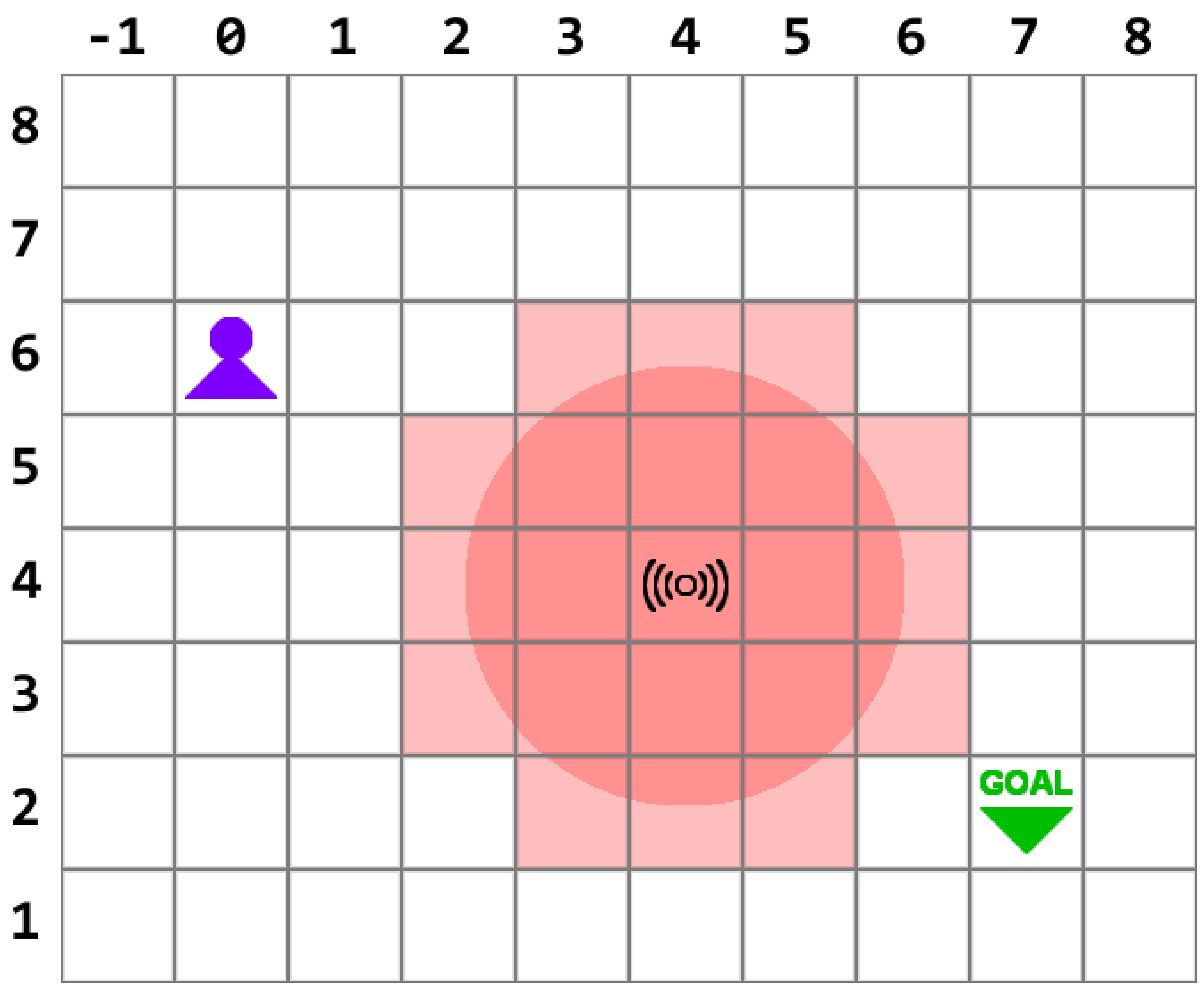
Upon successfully adding a fence, the message `Successfully added fence obstacle.` will be displayed.

`add sensor <x> <y> <range>`

An acoustic sensor takes three parameters - a pair of coordinates and a range. If the agent moves within the range of the sensor, the agent will be caught. The sensor's range is a **floating point value** and a square is considered to be obstructed by the sensor if it is within the sensor's range, measured in Euclidean distance.

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the coordinates are not both valid signed 32-bit integers, the program will display `Coordinates are not valid integers.`
- If the range is not a valid number or is less than or equal to 0, the program will display `Range must be a valid positive number.`

For example, if the sensor is at (4,4) and has a range of 2.5:



Using the Pythagorean formula, we can compute the distance between each square and the sensor

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

So, for example, given that the sensor is at (4,4), we can check if (2,6) is within the range of the sensor:

$$\sqrt{(2-4)^2 + (6-4)^2} = 2.828\ldots$$

2.828... is greater than 2.5, so (2,6) is not within the sensor's range.

Now consider (3,6):

$$\sqrt{(3-4)^2 + (6-4)^2} = 2.236\ldots$$

2.236... is less than 2.5, so (3,6) is within the sensor's range (and therefore the agent cannot enter that square).

With this, we can see that squares (3,2), (4,2), (5,2), (2,3), (3,3), (4,3), (5,3), (6,3), (2,4), (3,4), (4,4), (5,4), (6,4), (2,7), (3,7), (4,7), (5,7), (6,7), (3,8), (4,8) and (5,8) are all within 2.5 klicks of the sensor. This means the agent cannot enter any of those squares.

Upon successfully adding a sensor, the message `Successfully added sensor obstacle.` will be displayed.

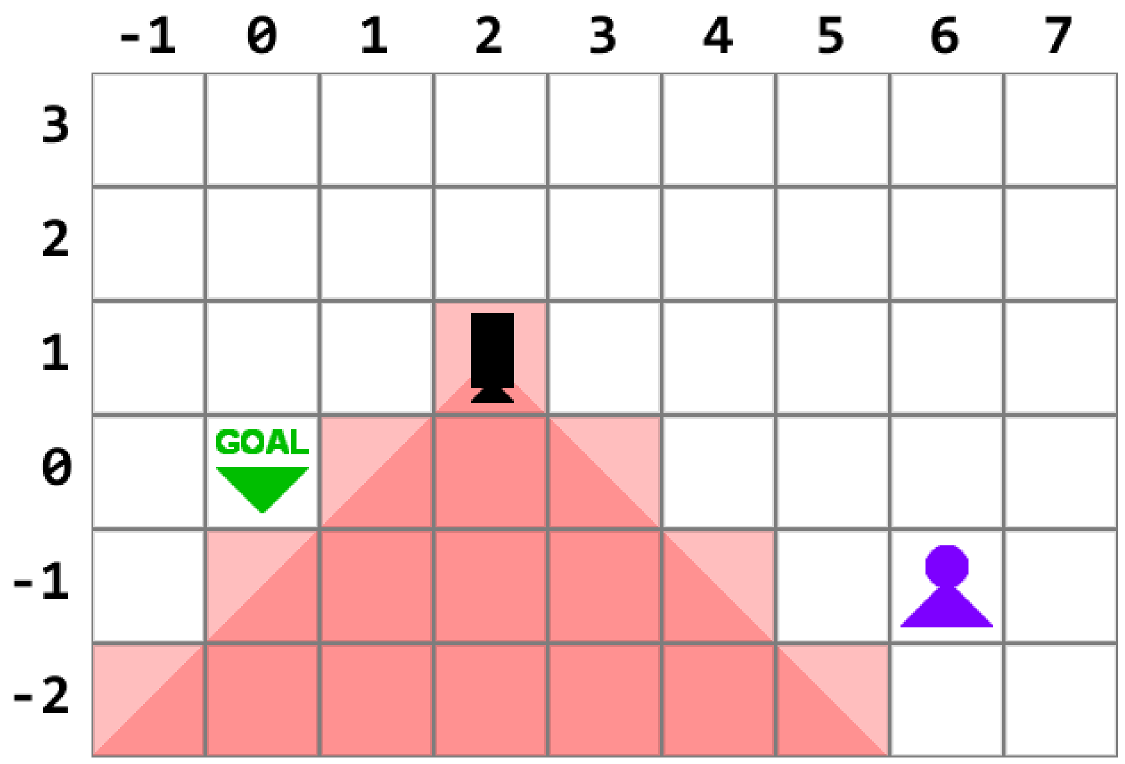
`add camera <x> <y> <direction>`

Cameras take three parameters - a pair of coordinates specifying a location, and a direction (which must be `north`, `south`, `east` or `west`), and can spot anything within a cone of vision extending out in that direction. Cameras have infinite range.

- If a different number of parameters are provided, the program will display `Incorrect number of arguments.`
- If the coordinates are not both valid signed 32-bit integers, the program will display `Coordinates are not valid integers.`
- If the direction is not one of `north`, `south`, `east` or `west`, the program will display `Direction must be 'north', 'south', 'east' or 'west'.`

Cameras have infinite range and will obstruct the agent entering its cone of vision. The cone of vision includes the camera's location itself (so if the camera is at 5,5 the location 5,5 is always considered obstructed, irrespective of what direction the camera is facing.) The cone of vision extends between the adjacent anticlockwise diagonal to the adjacent clockwise diagonal, covering all tiles between those diagonals and including the diagonals themselves. So, for a north-facing camera, this means all of the tiles between (and including) the north-west diagonal to the north-east diagonal.

Visualised, the camera's cone of influence looks like this (for a camera at (2,1) facing south):

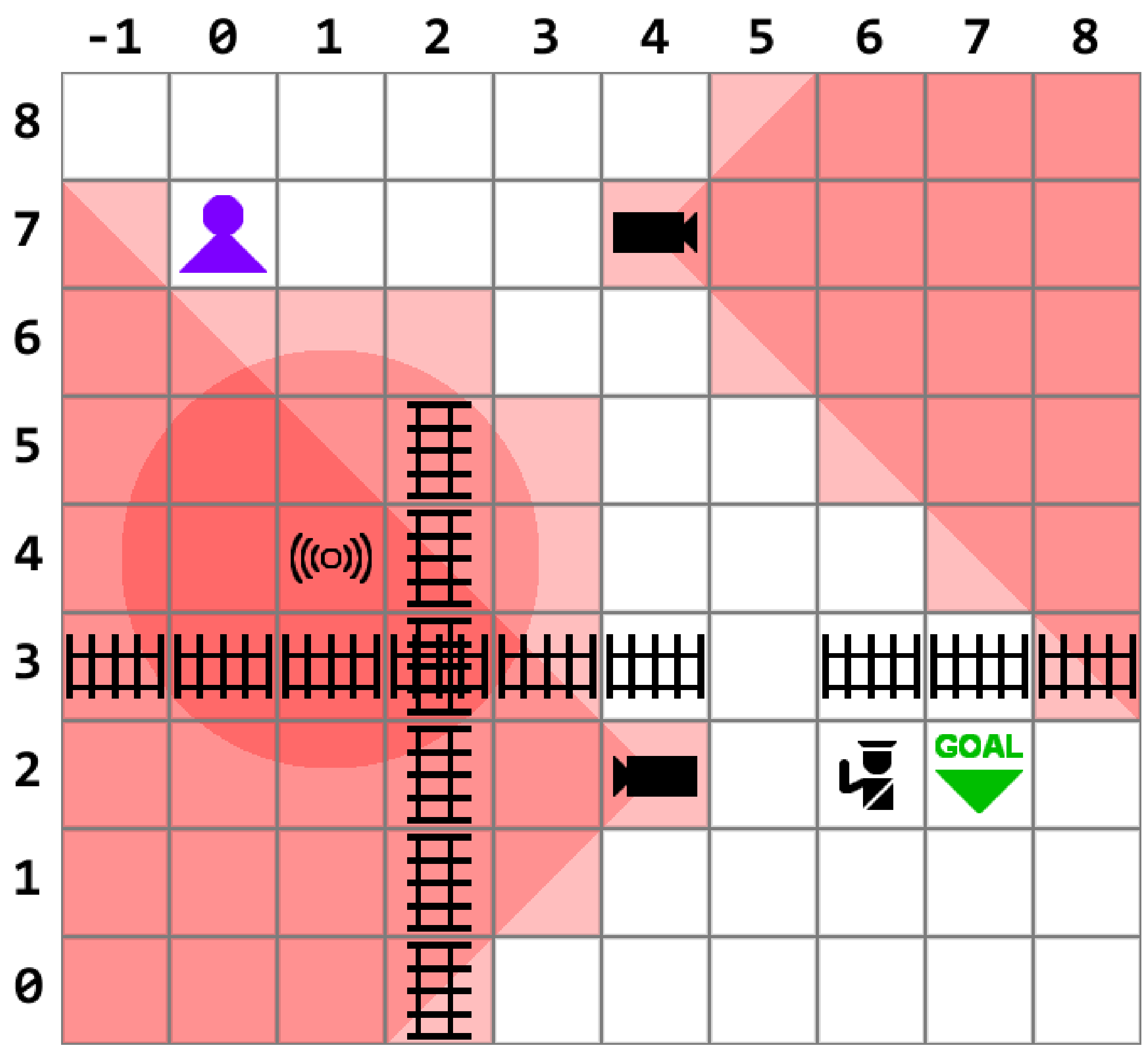


The agent is unable to enter the camera's square, nor any of the squares that the camera can see. If the camera is located at (2,1) and facing south it will obstruct (2,1) and (1,0), (2,0), (3,0) and (0,-1), (1,-1), (2,-1), (3,-1), (4,-1) and so on, extending infinitely.

Upon successfully adding a camera, the message `Successfully added camera obstacle.` will be displayed.

Multiple and overlapping obstacles

Many scenarios will involve multiple obstacles being present. When multiple obstacles are present, they all contribute to preventing the agent from entering certain squares. Multiple obstacles do not interfere with each other - for instance, sensors and cameras can overlap with fences, guards and each other, and sensors and cameras can detect the agent through fences. Multiple fences can overlap, as well.



When multiple obstacles are overlapping, the map display can choose to display any of the obstacles obstructing a particular tile. For example, the above scenario might look like this in the map:

```
Enter command:
map -1 0 10 9
Here is a map of obstacles in the selected region:
......CCCC
C....CCCCC
CSSS..CCCC
SSSSS..CCC
SSSSS...CC
SSSSSF.FFC
CSSSCC.G..
CCCCC.....
CCCC......
Enter command:
```

Marking criteria

There are a total of 50 marks available for this assessment item. Your submission will be marked with a mix of automated and manual processes.

Functionality (core task): 25 marks

When you submit your code to Gradescope, it will be automatically run against a battery of one hundred (100) test cases, checking your user interface, input validation and many different combinations of obstacles, agent locations and objective locations. These have been designed to vary in complexity, ensuring that you can pass some test cases even with a very early implementation. For this reason it is recommended that you submit to Gradescope **early and often**, to ensure that your code is successfully compiling and running on our grading platform.

80 of the test cases are designed to test your `check`, `map` and `path` commands.

Each of these test cases has the same structure:

- It will use your interface to add some number of obstacles to the scenario (0, 1 or multiple obstacles)
- It will perform one of `check`, `map` or `path` (with valid parameters)
- It will enter `exit` to exit the program

The marking for the test case will depend on the type of action performed

- For `check` your program will receive full marks if it correctly displays all of the safe directions and none of the unsafe directions, and no marks otherwise
- For `map` your program will receive full marks if the entire map is displayed correctly and no marks otherwise
- For `path`:
    - If your program successfully outputs a route and the agent can follow it to get to the objective without being blocked by an obstacle, you will receive **full marks** for that test case
    - If your program successfully outputs a route and the agent can follow it and get close to the objective without being blocked by an obstacle (in other words, walking into a fence or guard, or being spotted by a sensor or camera), you will receive marks for that test case based on how close the agent got: (d - r) ÷ d marks, where r is the Manhattan distance away from the objective that the agent got to and d is the Manhattan distance between the agent's starting position and the objective. If the result is negative (that is, the agent wound up farther from the objective than the starting position was), no marks will be awarded
    - If your program successfully outputs a route but it leads to the agent being obstructed/spotted by an obstacle (walking into a fence or being spotted by a camera, for example), you will receive no marks for that test case
    - If your program does not successfully output a route (or we cannot detect the route you output, possibly due to writing the wrong output), you will receive no marks for that test case
    - If your program writes multiple consecutive path steps in the same direction (e.g. `Head east for 1 klick.` / `Head east for 1 klick.` instead of combining them into a single `Head east for 2 klicks.`) your program will receive **half the marks it would have otherwise received** for that particular test case. The same will happen if `Head <direction> for 0 klicks.` appears in your output.
    - **Due to the scoring methodology, it is most important to handle the obstacles correctly and ensure that the agent does not stray into them. Even if you are unable to always find a path to the objective, if you direct the agent to get as close as possible to the objective without running into an obstacle and then stop, you should still get some marks.**

The remaining 20 test cases are designed to test your user interface. These will look at the messages you print out to the screen and will ensure that you print out the correct text and the correct error messages in response to invalid input (and ensuring that these error messages do not appear in the case of valid input.) These 20 test cases will also include, for example, a `path` test where there is no valid path, and one where the agent is already at the objective.

It is important that your program produce output **exactly** as specified in the problem description, user interface and obstacles list above. Variations in the output formatting (i.e. the messages your program prints out) may cause the autograder to misinterpret your input and cause you to fail test cases that you would have otherwise passed.

- With check, for example, it's vital that your program print out the correct line of text - if there are safe directions, you must print out `Agent, your location is compromised. Abort mission.` exactly, because the autograder will find that line and look at the lines following it to find the safe directions.
- With map you must make sure that your code is printing out `Here is a map of obstacles in the selected region:` on the line directly before the map appears, because this is used to find your map in the output.
- With `path`, the most important line for those tests is `The following path will take you to the objective:`, because the lines directly after that one appears are assumed to make up the path. (It is therefore also essential that it be on a line by itself)

In general, however, just use the text given in this specification exactly and your program should have no issues with the autograder. One last thing is to make sure you use **Console.ReadLine()** for reading all text - other Console functions for reading keys may not work with the autograder, even though they might seem to be fine when run on your machine. Every input is on a line by itself and terminated by pressing the enter key.

One last note: While we do not vary the test cases from run to run, if we discover during manual inspection of your program that you have **hard-coded** in our test cases and results (that is, instead of implementing the obstacle logic, you have built your program to specifically recognise our test cases and produce the output we expect) you will receive a score of 0 for functionality, as this is not be considered useful work.

Marking criterion

7: High Distinction (21.25-25 marks)	Submitted program implements the specification without hard-coding results and successfully passes 85/100 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 21.25 or greater.)
6: Distinction (18.75-21.24 marks)	Submitted program implements the specification without hard-coding results and successfully passes 75/100 tests on Gradescope (or partially passes enough tests to

	achieve a functionality score of 18.75 or greater.)
5: Credit (16.25-18.74 marks)	Submitted program implements the specification without hard-coding results and successfully passes 65/100 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 16.25 or greater.)
4: Pass (12.5-16.24 marks)	Submitted program implements the specification without hard-coding results and successfully passes 50/100 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 12.5 or greater.)
3: Marginal fail (10-12.49 marks)	Submitted program implements the specification without hard-coding results and successfully passes 40/100 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 10 or greater.)
2: Fail (6.25-9.99 marks)	Submitted program implements the specification without hard-coding results and successfully passes 25/100 tests on Gradescope (or partially passes enough tests to achieve a functionality score of 6.25 or greater.)
1: Low Fail (0-6.24 marks)	Submitted program passes fewer than 25 tests on Gradescope (or, through partially passing some tests, receives a functionality score of less than 6.25), or does not compile/run, or uses hard-coded results to attempt to subvert the assessment process

Object-oriented design and implementation: 6 marks

Your object-oriented design and implementation will be checked by looking at your class inheritance hierarchy and examining source code to check your use of polymorphism and encapsulation, as well as the private/protected/internal/public state of your methods, fields and properties.

Marking criterion

7: High Distinction (6 marks)	Leverages advanced object-oriented techniques like polymorphism and inheritance to elegantly solve the task AND meets the requirements for achievement levels 4-6 as stated below
6: Distinction (5 marks)	Non-const fields are always private and all external access to them is controlled through public methods and/or properties which enforce appropriate domain restrictions AND the submission meets the requirements for achievement levels 4-6 as stated below
5: Credit (4 marks)	Features a high quality, thoughtful object-oriented design with appropriate classes, with minimal to no use of non-const public fields AND meets the requirements for achievement level 4 as stated below
4: Pass (3 marks)	The program is implemented using classes with appropriate subdivision of responsibilities and some kind of object-oriented design (classes have appropriate fields and methods)
3: Marginal Fail (2 marks)	The program is implemented using classes but with little use of object-oriented design or programming (e.g. mostly static methods), OR too many methods/fields are unnecessarily public/internal
2: Fail (1 mark)	Very limited use of classes; no object oriented design.
1: Low Fail (0 marks)	Not an object-oriented design (e.g. all static methods or only one class) OR too little code submitted to evaluate the appropriateness of the design

Cohesion: 5 marks

Your project's cohesion will be checked by examining class interdependencies and looking at the methods/fields/properties present in each class. Good cohesion means that each class is responsible for a single unified task and all methods/fields/properties support that task.

Marking criterion

7: High Distinction (5 marks)	Classes are highly cohesive (responsible for a particular task and all methods/properties/fields contribute to this task)
6: Distinction (4 marks)	Classes are mostly cohesive, potentially with some responsibilities creeping into other classes
4-5: Pass (3 marks)	Cohesion is at an acceptable level, but with some mixing of responsibilities

3: Marginal fail (2 marks)	Classes have poor cohesion and responsibilities are mixed
2: Fail (1 marks)	Classes have very poor cohesion and responsibilities appear to be allocated haphazardly
1: Low Fail (0 marks)	Classes have extremely poor cohesion or this is a single-class solution

Abstraction: 5 marks

Your use of abstraction is marked by analysing your source code. Under this criterion, the following will be penalised:

- Overly long or complex methods that should be broken up into simpler methods
- Repetitive coding constructs that should be a loop instead
- Sections of code or methods that are overly similar to other sections of code/methods and should be abstracted out into a common method
- Use of magic numbers (value literals other than -1, 0 or 1) in your code (these should be replaced by declaring a `const` field with a descriptive name containing the value)

In general, we are looking for anything other than code clarity and your OOP design (as those are marked in their own criteria) that makes your code more difficult to maintain.

Marking criterion

7: High Distinction (5 marks)	The submission contains no code that could be improved by abstracting out to an additional method AND the submission meets the requirements for achievement levels 4-6 as stated below
6: Distinction (4 marks)	No use of magic numbers and no repetitive coding constructs that should be a loop instead AND the program is broken up into appropriately-sized methods, with nearly all methods being under 50 lines long AND the submission meets the requirements for achievement levels 4-5 as stated below
4-5: Pass (3 marks)	The program features no or only minor instances of repetition
3: Marginal Fail (2 marks)	The program features long methods with large amounts of repetition
2: Fail (1 mark)	Much of your program's logic is enclosed within a small number of methods, with minimal use of helper methods sharing the workload
1: Low Fail (0 marks)	Everything is in main() OR too little code submitted to evaluate your use of abstraction

Code clarity and comments: 5 marks

Code clarity is marked by looking at how your choice of identifiers (names of classes, methods, properties, fields, local variables) and your program's flow (use of loops / branching) affects the comprehensibility and maintainability of your code. For comments, we are looking for two different kinds: **top-level XML tag comments** **(https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags)** (///) and body-level comments (usually single-line // comments). If your code has a 'goto' in it, unless you have an extremely good reason the maximum mark you can get for this section is 1.5/5.

Marking criterion

7: High Distinction (5 marks)	All public classes/methods/properties feature C# XML documentation comments at the top level explicitly defining the code's external interface AND the submission meets the requirements for achievement levels 4-6 as stated below
6: Distinction (4 marks)	All or nearly all public classes, methods and properties feature useful and informative comments at the top level AND the submission meets the requirements for achievement levels 4-5 as stated below
4-5: Pass (3 marks)	Many public classes feature useful and informative comments at the top level (outside the code body)
3: Marginal Fail (2 marks)	Complex or confusing sections of code feature inline comments describing the programmer's intention. Comments are not unnecessarily frequent. Identifiers are well chosen and the program flow is logical

2: Fail (1 marks)	Useful comments are sparse OR comments are far too unnecessarily frequent to the point that they interfere with the reading of the code OR identifiers are poorly chosen OR the program flow is confusing
1: Low Fail (0 marks)	No useful comments OR too little code submitted to evaluate the quality of the code

Exception handling: 4 marks

Your use of exception handling (try / catch blocks and throw statements) will be marked by looking at your code (examining areas). If unhandled exceptions terminate the program at all during our testing (either automated or otherwise) the maximum mark you can get for this section is 1.5/5 - **unless** that unhandled exception is a `NotImplementedException`.

Note that we aren't just looking for try / catch - swallowing exceptions that affect the running of the program and not taking the appropriate steps is is **not** considered exception handling (so you cannot pass this criterion simply by wrapping your entire program in a try/catch.) Instead, we consider the appropriateness of the exception handling given the situation.

Also note that `NotImplementedException` does not count as exception handling either (though, at the same time, we will not penalise the use of `NotImplementedException` if your program terminates with it during our testing. For example, if you did not manage to implement the path command, you can throw a `NotImplementedException` if the user enters this. You will not get any marks for the path test cases in our testing, but we will not penalise your exception handling grade for doing this.)

Marking criterion

6-7: (High) Distinction (4 marks)	Exception handling is used to enforce the external interfaces of classes AND the submission meets the requirements for achievement levels 4-5 as stated below
5: Credit (3 marks)	Exception handling is used to prevent undesired/unexpected behaviour and/or results AND the submission meets the requirements for achievement level 4 as stated below
4: Pass (2 marks)	Exception handling is used to handle exceptional events at the appropriate level as per the requirements around invalid data in the assignment specification
2-3: Fail (1 mark)	Only marginal/incidental exception handling is present
1: Low Fail (0 marks)	No use of exception handling (no throw and/or no try/catch)

What to submit

For this assignment, you are required to submit the source files that make up your project to Gradescope. As with Assignment 1, do not submit a solution (.sln) file or C# project (.csproj) file - we will provide our own .csproj file (which will build your code using .NET 8.0), and you are more likely to run into problems if you attempt to submit one. See **Submitting to Gradescope** (https://canvas.qut.edu.au/courses/16661/pages/submitting-to-gradescope) for general information about submitting Gradescope assignments - the output format is quite different here, but the process of uploading your source files is the same.

You get unlimited submissions and submissions will give you useful feedback - in addition, if you have a submission in Gradescope by the end of Week 9 the teaching team will have a look at it and give you some early feedback on your progress. You are encouraged to take advantage of this opportunity.

Feedback

You will receive some automated feedback on the functionality portion of your grade soon after submitting. You can then use this to improve your program and resubmit. After the due date, once marking has been completed you will receive complete feedback and the rest of your marks.

If you have submitted to Gradescope by the end of Week 9, you will receive some early feedback on your progress by the end of Week 11. We will not mark your entire assignment at this time, but we will have a look and tell you whether you're on the right track or not.

Tips / Suggested order of implementation

When approaching a large project like this, it can be difficult to tell where to begin. Here are our suggestions as to how to best utilise your time working on this assignment:

1. Start by implementing the introduction and command listing (as it is simply a series of Console.WriteLine() calls) and, at a minimum, make your program exit when `exit` is entered. The way the test cases are designed, every test will send `exit` to exit the program as the final line of input. This means that, even if you haven't implemented anything else, your submission will at least get through every test case without timing out. You can then slowly add additional functionality to your program as you go, and submit to Gradescope to keep track of your progress. Just displaying the command listing and 'Enter command:' lines correctly will get you one passing test case, for a total of 0.25/25 marks. Add another check to bring out the 'Invalid command:' message when an unsupported command is entered to pass another test case, and implement 'help' as well to pass a further test case, bringing you up to 0.75/25.
2. Implement the first obstacle (the guard) and the first non-obstacle menu item (the direction checker). If you've done this successfully, including checking for invalid parameters, you should now pass an additional 9 test cases (for a total of 3/25)
3. Continue implementing obstacles in order of difficulty: fence, then sensor, then camera. As you complete each one you will be passing more and more test cases, and when you finish, you should have a score of 11.25/25 (or close). If you get stuck on this, take a break and move to step 4 for now, which should help you make a bit more progress and pass a few more test cases. You can then come back to step 3 later.
4. Implement the map. If you have a good OOP solution to handling obstacles, drawing the map should be easy, and completing this will get you an additional 4 marks, for a total of 15.75/25 marks if all the previous test cases passed too.
5. Work towards implementing pathfinding. As you improve your pathfinding approach, your test cases passed should steadily increase. Once you have it working perfectly, you should be getting close to 25/25 for functionality.
6. Now that you have all the functionality working, revisit your source code with an eye on the criteria sheet - get rid of any stray warnings about nullable types, turn any magic numbers in your code into consts, implement exception handling if you aren't already using it, make sure you have XML tag comments on all your public classes and make sure your classes are cohesive.

Declaration:

On submission, you are declaring that, unless otherwise acknowledged, this submission is wholly yours and it has not been used and already submitted. You understand that this work may be submitted for plagiarism checking and consent to this taking place.

(https://canvas.qut.edu.au/courses/16661/modules/items/1598130)

(https://canvas.qut.edu.au/courses/16661/modules/ite

You are unable to submit to this assignment as your enrolment in this course has been concluded.