UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS110 - Program design: Introduction

Assignment 3 Specification

Linked Lists and Recursion

Release Date: 24-10-2022 at 06:00

Due Date: 13-11-2022 at 23:59

Contents

1 General instructions

- This practical should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.

- Read the entire assignment thoroughly before you start coding.
- **To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at http://www.ais.up.ac.za/plagiarism/index.htm.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will not be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements **Please ensure you use C++98**

2 Overview

This assignment deals with some more advanced types of linked lists. While you will create lists and queues here, you will also create something very similar to a 2D array as a double linked list.
Ensure that you are familiar with linked lists and their fundamentals:
You will be doing a lot more than just your average singly-linked traversal!
Note that this assignment is quite long, so you **will** need to start early. Also try not to lose hope if you get stuck; you've got the spec, the lecture slides, the textbook and, if all else fails, Discord tickets.

Scenario: You've spent almost a full year working at a video game company. So far they haven't let you do anything too interesting, but the higher-ups have finally decided that you get to do something exciting:
Create a small-scale, proof of concept puzzle survival game that runs in the terminal, using C++. It isn't too fancy and it is only the foundation of the game, since it's meant to only be a proof of concept: there is a map in the game containing floors, walls, doors, an exit and of course, a playable character. The goal of the game is to reach the end of the level (the exit). The full version of the game would have items and enemies with paths that they follow around the map to make it an actual puzzle. The company decided not to lay this version on you due to time constraints :)
The 'foundation' version you need to create has the following features:

- The player can move in 8 directions
- There are objects that populate the map. These are
  - Walls
  - Floors
  - Doors
  - Lamps
  - A Player
  - An Exit
- Messages are displayed conveying what is happening in the game. They are put in a message queue, and gradually messages are removed from the queue as they 'age'
- Lamps which provide a light source. Without these light sources, everything would be dark. The lamps use a watered-down version of recursive ray tracing to light up the environment (more on that in the relevant section).
- As soon as a player reaches the exit, the game ends.

3 Your Task

Your task is to implement this first part of the game, step by step. Try not to be intimidated, all aspects of the game will use elements of things that you know. That said, there are a lot of moving parts and things that can go wrong. **Be sure to test thoroughly!** Don't expect everything to work just because you get the expected output!

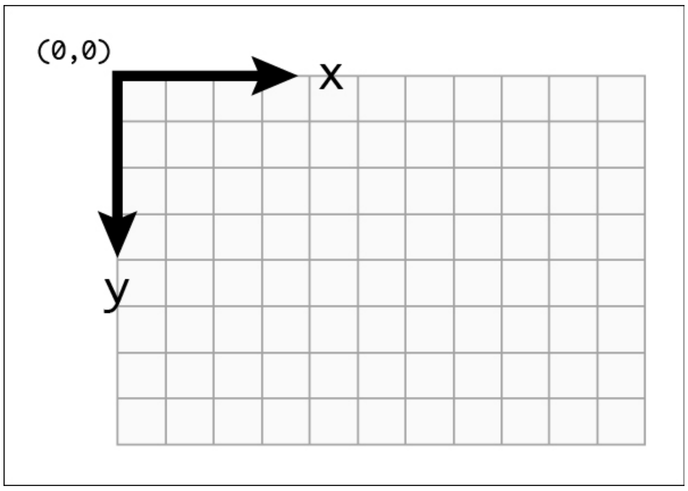
You have been provided with all the necessary files, in which you must complete the practical. The header files are incomplete, and you will need to add the classes to them yourself. All the includes in the header files should be left **as-is!** The header files on fitchfork will have the same includes. Also be sure to double check spelling and capitalisation of class and member names.

4 Part 1

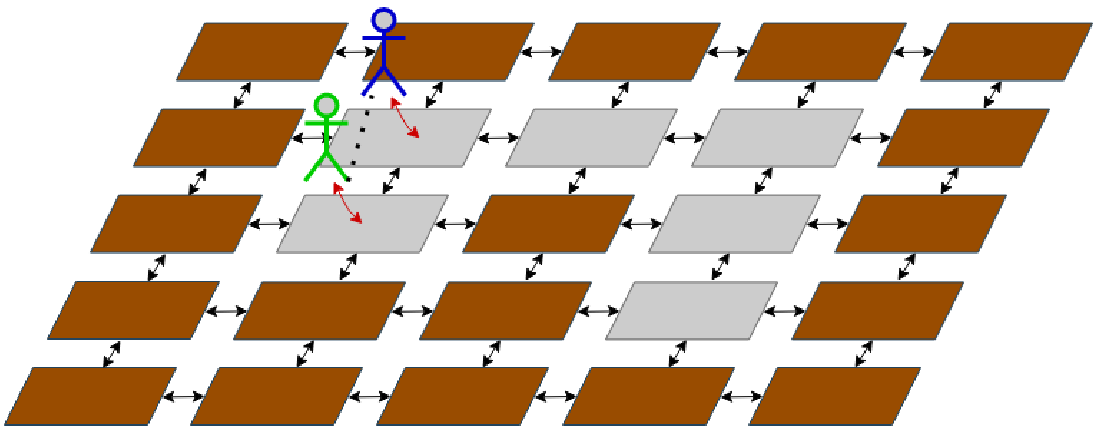
To start off, in this task you will lay the groundwork for the entire game with the creation of the map and object lists. When you are finished, the player will be able to move across the map, continuously changing links along the way, and always appear on top of other objects.

Every object that goes on the map, inherits from a base *Object* class. All the objects need to be in an object list that a game instance can keep track of. So, an *Object* is a **node in a linked list**. Some of the objects, including players and potentially e.g enemies later on, will need to move around. In order to facilitate that, an object needs some way to represent and change its position.

Generally, this can be done with just two numbers, for x and y position. Then, all objects could be kept in a normal single-linked list and updated however a game wants to, searching through the list and finding the object it wants. However, for the lighting system in a later task, this sort of individual linear search is very inefficient. An object needs easier access to the objects that are around it in the game space (to spread light rays to them).
For that reason, each stationary object present in the map is actually part of at least two linked lists: it is part of both a row and a column in the map space. In addition, it will need to be doubly-linked for traversal in both directions along an axis.
The position (0, 0) is at the top left of the map. So, incrementing the y-position moves an object down, and incrementing the x-position moves an object right. If this seems strange, keep in mind that were the map a 2d-array, x and y positions would correspond to indices in the array. This is also the same coordinate system used by virtually all modern systems and their displays.

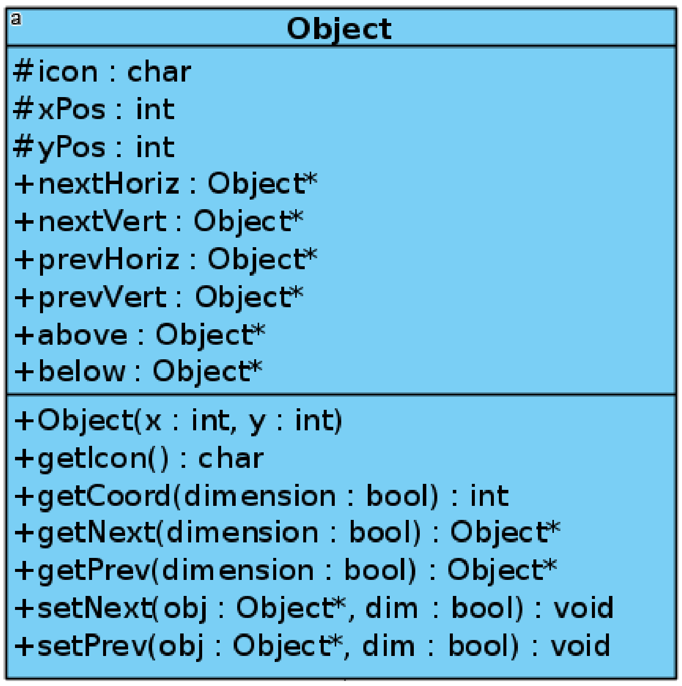


Furthermore: Movable objects can be on top of other objects. Notably, a character needs to be able to move across a floor. So, a floor object and the player will be occupying the same space at the same time. To solve this, each object has a third set of links, to link to objects on top of and beneath them. As a player moves across the floor, it removes itself from the top of the "third dimension list" of one floor object, and adds itself to that list of the floor object it moves to. See below for a visual: the brown tiles are walls, and the grey tiles are floors. The character (green) moves one space up (blue). The red arrows represent the above/below links. The player moves from *above* the first floor tile to *above* the second floor tile. This is a representation of the map in the main file for Part 1, by the way.



Say you have an object that represents something that is on top of a floor (and can be removed later, revealing the floor beneath it). Say also that a player can stand on this item. In that case, the player needs to be at the top of the *above* list, obscuring the item. For that reason, whenever a map needs to find the icon to display at that given location, the icon of the object at the very top needs to be displayed.

4.1 Object



```
a                 Object
#icon : char
#xPos : int
#yPos : int
+nextHoriz : Object*
+nextVert : Object*
+prevHoriz : Object*
+prevVert : Object*
+above : Object*
+below : Object*
+Object(x : int, y : int)
+getIcon() : char
+getCoord(dimension : bool) : int
+getNext(dimension : bool) : Object*
+getPrev(dimension : bool) : Object*
+setNext(obj : Object*, dim : bool) : void
+setPrev(obj : Object*, dim : bool) : void
```

Variables:

- xPos and yPos : int
  The position of an object, with (0, 0) being the top left-hand corner of the map.
- icon : char
  This is the one-character icon that gets displayed on the map to represent an object.
- nextHoriz : Object*
  A link to the object to the right of the current object.
- prevHoriz : Object*
  A link to the object to the left of the current object.
- nextVert : Object*
  A link to the object down from the current object.
- prevVert : Object*
  A link to the object vertically above the current object (above meaning with one less y-value).
- above : Object*
  A link to the object on top of the current object. Meaning two objects occupy the same x and y values.
- below : Object*
  A link to the object below the current object, when two objects occupy the same x and y values.

Functions:

- **Object(x : int, y : int)**
  A constructor which takes in the x and y positions of the object. The constructor should also set the icon to '?'. The reason is that *icon* is a protected member, so it an be accessed by child classes who will set their own icons. If an object is instantiated as-is (not a derived object), a question mark is displayed. Remember to initialise links to null
- **getIcon() : char**
  A getter for the icon. If there is an object above the current object, logically it would be obscured. In that case, the icon displayed should be the icon of the above object. This continues, until the very top object is displayed, so that visually (from top down) only the object at the top of the above chain is visible.
- **getCoord(dimension : bool) : int**
  Instead of having separate getters and setters for each individual position and link, there is a single function per set of links, with a *bool* value passed in. *False* indicates a horizontal request (in this case, x) and *True* indicates a vertical request (in this case, y).
- **getNext(dimension : bool) : Object***
  A getter for the *next* links. Again, *False* indicates horizontal (in this case, *nextHoriz*) and *True* indicates vertial (in this case, *nextVert*).

- **getPrev(dimension : bool) : Object***
  The same as above, except for the *previous* links

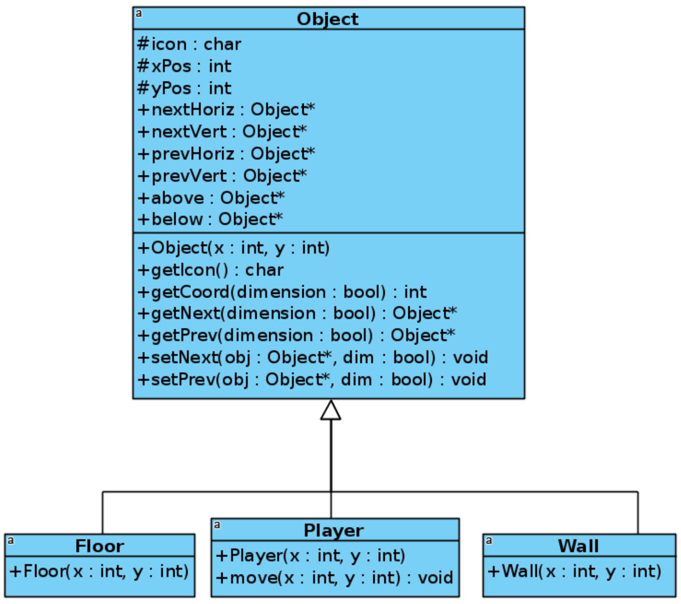
- **setNext and setPrev** follow the same convension, except that they set links.

The following classes need to be created that inherit from Object: Player, Floor and Wall.

4.1.1 Dimension

In this section there is reference to a coordinate 'dimension'. Here is some additional explanation for it. To make the linking of objects in the lists easier, which links the list should use is specified. For instance, should the list store an item by using its horizontal or vertical links? It may want to use the vertical links because the list represents a column. On the other hand, it may just be a general-purpose list, in which the items don't use their 2D map links. It can then use those links, but it needs to use either vertical or horizontal links consistently. To solve this, a boolean is used to represent a dimension. *False* represents the horizontal dimension. To return this coordinate, that would mean the x-coordinate. In a list, that means linking on the *nextHoriz* and *prevHoriz* links. *True* represents the vertical dimension. To return this coordinate, that would mean the y-coordinate. In a list, that means linking on the *nextVert* and *prevVert* links.

4.2 The Objects


Object
#icon : char
#xPos : int
#yPos : int
+nextHoriz : Object*
+nextVert : Object*
+prevHoriz : Object*
+prevVert : Object*
+above : Object*
+below : Object*
+Object(x : int, y : int)
+getIcon() : char
+getCoord(dimension : bool) : int
+getNext(dimension : bool) : Object*
+getPrev(dimension : bool) : Object*
+setNext(obj : Object*, dim : bool) : void
+setPrev(obj : Object*, dim : bool) : void
Floor
+Floor(x : int, y : int)
Player
+Player(x : int, y : int)
+move(x : int, y : int) : void
Wall
+Wall(x : int, y : int)

Each object simply takes in the position it should be placed as input parameter to its constructor, and also sets its icon. In addition, *Player* has a move method. It would be best to leave the move method implementation for after you have the map set up, so its description is after that section.

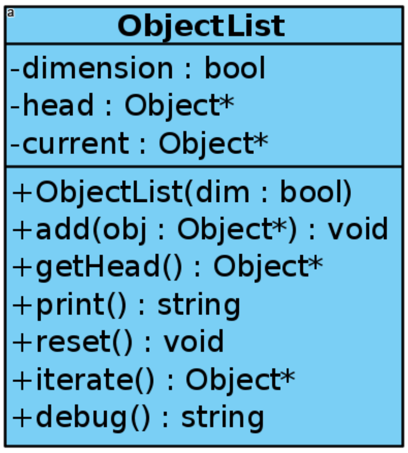
- **Player** uses the icon '&'

- **Wall** uses the icon '#'

- **Floor** uses the icon '.'

4.3 ObjectList

This is a list that holds several objects, chained along a given dimension (e.g a row would be an ObjectList, chained along the horizontal links of object). A list allows easier iteration of objects, and these lists are also used for other things later on (like storing a list of lamps for lighting).



Variables:

- dimension : bool
  Which dimension the list uses when chaining Objects

- head : Object*
  Head of the list

- current : Object*
  The current object in an iteration cycle

Functions:

- **ObjectList(dim : bool)**
  Constructor which initialises everything as it should be (to an empty list), and sets the list dimension.

- **No destructor**

- **add(obj : Object*) : void**
  Add an object to the list. Remember that these lists are used to store objects along a dimension. They also need to be inserted in the correct spot along that dimension, that is to say in ascending order to fit with the 2D map you will be creating.
  Example: Say you have three object, and the list stores items along the horizontal (0/false) dimension. If the objects have positions (2, 5), (9, 2) and (6, 0), then they will be stored in the order (2, 5), (6, 0), (9, 2). If an object is added with the same dimension value as an existing object, it is added **in front of** that object.

- **getHead() : Object***
  A getter for the head member

- **print() : string**
  Return a string concatenating the icons of every item (no newline at the end). Essentially this will be used later to print out e.g a row; iterate from start to finish, and return a string containing all the icons in order.

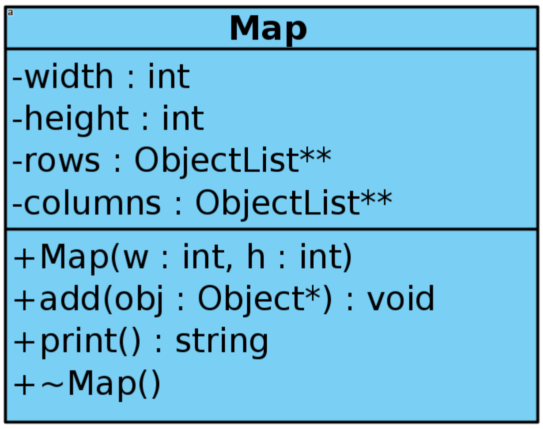
- **iterate() : Object***
  This is make iterating over all the items more convenient for anything using the list. Instead of them needing to keep track of which dimension the list uses, they simply call *iterate* and the next object along the correct dimension is returned. Should iteration be over (end of the list), then *null* is returned. Keep track of where iteration is by using the *current* member variable.

- **reset() : void**
  This resets iteration, so that the *current* member points at the head

- **debug() : string**
  This is used to make sure your links are all correct. The start of the string is the text "Forward:\n". For each item in the list, add the following followed by a newline (for each object): "*<icon>* at (*xPos*,*yPos*) Top:*Y or N*". *Top* is true (Y) if an object does not have anything above it (if *above* is null). After the entire list is iterated through, the text "Back:\n" is added, and the entire process is repeated with the list going in reverse, thereby testing the double-linked list. See example output for further clarification.

4.4 Map

One level up from the *ObjectList* is the *Map*. This has two arrays of *ObjectList*s, one for rows and one for columns.



Variables:

- width/height : int
  The height and width of the map

- rows : ObjectList**
  This is an array of type *ObjectList** and of size *height*

- columns : ObjectList**
  This is an array of type *ObjectList** and of size *width*

Functions:

- **Map(w : int, h : int)**
  A constructor which takes in width and height, and creates the required rows and columns, filled with empty lists using the correct dimension.

- **~Map()**
  Destructor, deletes all objects on the map. You can use a list's **getHead** to get the start of each e.g row, then delete every row. Remember to then **not** delete the columns as well.

- **add(obj : Object*) void**
  Add an object to the map. In order to have all the links set up correctly, you have objectlists for both the rows and columns. So, when an item is added, you need to add it into both the correct row, **and** the correct column. That way all the links are updated correctly by the lists, and you end up with each object in two lists making up a map. From there, an object can directly reference the objects around it.

- **print() : string**
  This prints the map, row by row, and adds a newline after each row. It starts from the top row, working its way down. This method is used to actually print the map when running the game.

4.5 Player Movement

Before proceeding, you need to implement the *move* function for the player. It takes in two parameters: xdifference and ydifference. That is the difference by which a player must move along each dimension. The value are either -1, 0 or 1. That means the player can only move one tile away in any direction, at a time.

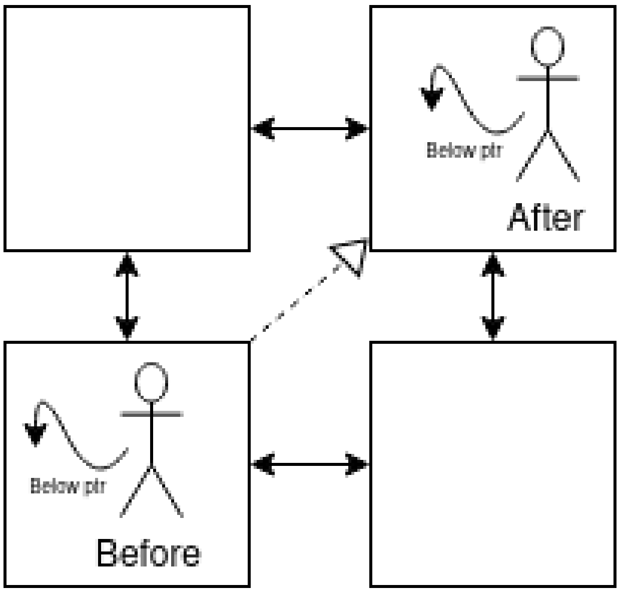
- **xDiff = 1, yDiff = 0** is a move to the right
- **xDiff = -1, yDiff = 0** is a move to the left
- **xDiff = 0, yDiff = -1** is a move up
- **xDiff = -1, yDiff = 1** is a move to the left and one down (diagonally)
- etc.

By using this method, we are effectively simulating an 8-directional control, centred around the number 5 in a numpad. Valid inputs are 1, 2, 3, 4, 6, 7, 8, 9. Anything else is ignored for now.

The actual mappings should be obvious, but they are explicitly stated in the *Game* subsection should you need clarification.

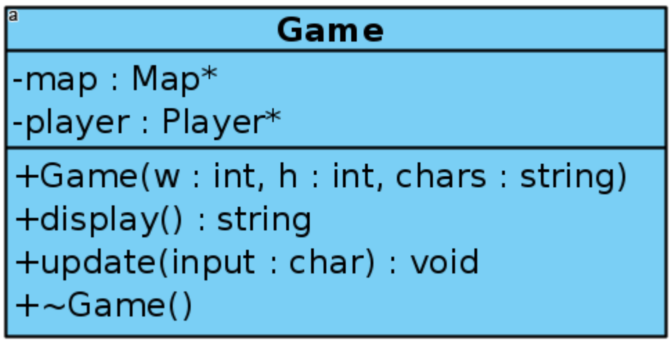
When a player moves, it needs to calculate where it will end up. By using the *below* link of the player, it has access to the floor object beneath it. You *need to make sure* that you use the lowest object possible for this. From there, use the horiz/vert links to find the object the player will end up. Make sure that you find this object *before* you start 'moving' the player (e.g for a diagonal move). When placing the player above its new floor, again make sure that it goes at the very top (for display purposes).

In the example below, a player moves diagonally up and to the right (xdiff: 1, ydiff: -1). After the move, all the below and above pointers of involved objects need to reflect the result. Remember to set the above of the old below object to null, since it no longer has anything above it. Also, make sure that you do, in fact, do this to the correct (highest) object in that position.



4.6 Game

This is the final component for this part. The Game class serves as an entry point into everything else from a main function.



Game

-map : Map*
-player : Player*

+Game(w : int, h : int, chars : string)
+display() : string
+update(input : char) : void
+~Game()

Variables:

- map : Map*
  A pointer to a map instance

- player : Player*
  A pointer to the player of the game (there is only one)

Functions:

- **Game(w : int, h : int, chars : string)**
  The constructor for a game instance. The third parameter is a text representation of a map. You need to iterate through the entire string, character by character, and determine
  1. what object is at that position
  2. what the x and y values are for that position.

  You can assume that the map will always be a perfect rectangle, so you can use the width and height values to keep track of your current 'position' in the iteration. Then for each character, add a corresponding Object to the map.
  Important: When a player is created, it assumed that there is a floor object beneath it. The floor is added to the map, but the player **is not,** and is just assigned to the player member of the game for future use. The *above/below* links still need to be set. **This technique is only used for the player, and one other object, the lamp (in Part 2)**

- **~Game()**
  Destructor, deletes the map and player

- **display() : string**
  This displays the entire game. For now, this only returns the map display (*print()*).

- **update(input: char) : void**
  This performs an update on the game. At the moment, what that entails is moving the player (if there is one) depending on the input. Any undefined input advances the game, but does not do anything else. The input system was explained before, but here it is explicitly what each input does to make a player move:

  - **1**
    Diagonally down/left
    xDiff = -1, yDiff = 1
  - **2**
    Down
    xDiff = 0, yDiff = 1
  - **3**
    Diagonally down/right
    xDiff = 1, yDiff = 1
  - **4**
    Left
    xDiff = -1, yDiff = 0
  - **6**
    Right
    xDiff = 1, yDiff = 0

- **7**
  Diagonally up/left
  xDiff = -1, yDiff = -1
- **8**
  Up
  xDiff = 0, yDiff = -1
- **9**
  Diagonally up/right
  xDiff = 1, yDiff = -1

The provided main has a test you can run to make sure the map is created and displayed correctly. When you are done with that, there is a second map, which contains a character for a *Player*. When adding a player to the game, assume that there is a floor beneath the player. The player itself is not added to the map, but is updated separately by the game. Remember that the player will still be visible, due to how an object's *getIcon()* works....
Since there is no checking being done to ensure a player does not walk into walls, you can do so. In fact, use it to test that your player can walk on top of other things properly, and remains visible. Note that you can walk outside the map, and that will cause a segfault. You can leave that as-is for now, it need not be prevented yet.

5 Part 2

For this next part, you will add the message queue, the exit and the lighting system. This will require the creation of new classes, and the modification of existing ones.

5.1 Messages

The idea is a queue, that responds when the game updates. A message (node in the queue) is essentially only a vehicle for some text that stays in the terminal for a few steps of the game, and then disappears. The lifetime of a message is determined by how many steps of the game occurs. We will assume that a step occurs every time all messages are printed. Once a timeout is reached, the message is dequeued from the message list, and the next message needs to be timed out (from the start). If two messages are added during a single 'step' of the game, one message will need to time out before the next one starts.

5.1.1 Message

Variables:

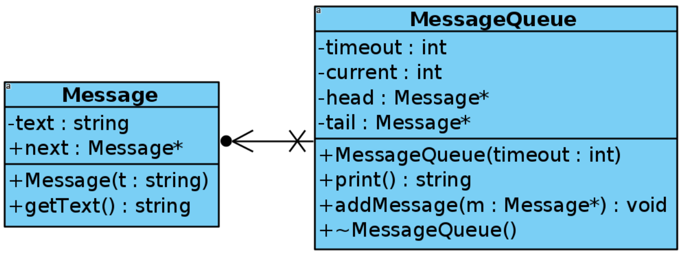


Figure 1: The arrow means "uses"

- **text : string**
  The text that the message contains

- **next : Message***
  The next message in the queue

Functions:

- **Message(string : t)**
  A constructor that takes in the message to store. Also sets next link to null

- **getText() : string**
  Returns the message text

5.1.2 MessageQueue

Variables:

- **timeout : int**
  The timeout that each message lasts

- **current : int**
  The current timer of the timeout

- **head : Message***
  The head pointer

- **tail : Message***
  The tail pointer

Functions:

- **MessageQueue(timeout : int)**
  A constructor that takes in the message timeout. Also initialises links appropriately, and set *current* to *timeout*, to be decremented each time print is called.

- **~MessageQueue()**
  Deletes any remaining messages in the queue

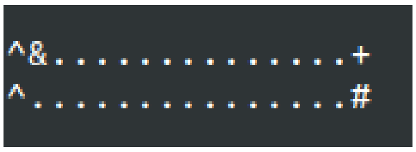
- **addMessage(m : Message*) : void**
  Adds a message to the back of the queue

- **print() : string**
  Returns the current messages in the queue. Each message terminated by a newline (\n). It then updates the *current* timer. When the timer runs out, the message at the head of the queue needs to be removed and deleted, and the timer reset.

5.2 Lighting

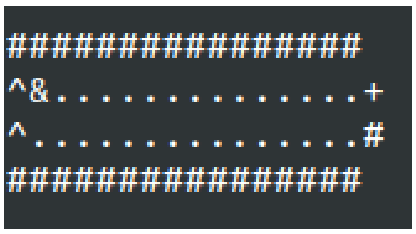
This is the reason for the list setup in Part 1. The lighting in this game is very basic, and only works in 4 directions (up, down, left, right). Basically there is a recursive algorithm that sends a light ray a single step ahead in any given direction. At that step, the light ray is processed, and may either be spread further along the same direction, or stopped (for instance if the light ray hits a wall).
Some features of this system:

1. A light source throws light of a given intensity in 4 directions
2. The next item in a direction is selected using an object's horiz and vert pointers
3. An object which is lit up displays normally. Otherwise, it is blank (represented with a space character)
4. A directional light ray is propagated recursively. When an object is hit by a ray it lights up, and calls the same lighting algorithm on the next object that lies in that direction. The difference is that the light level has now reduced by 1.
5. The algorithm should only be called on floor-level objects. Remember that the icon for an 'above' item is returned by the floor-level object when the map is displayed. Return a blank space if an object is unlit.
6. To keep things simpler, different levels of light have no visual distinction: when light level reaches 1, the object is lit up normally, but the light stops propagating.
7. Solid things like walls and closed doors do not let light through. They simply become visible, and do not propagate light rays.
8. All environmental light sources are kept track of directly by the map, so that it does not need to update each object in the map until it finds a light source.
9. If you implement it like discussed, you may notice something with regards to how the lighting works: unless a light ray directly hits a wall from the opposite side, it remains darkened. That is, if a light ray travels along next to a wall, it stays unlit.



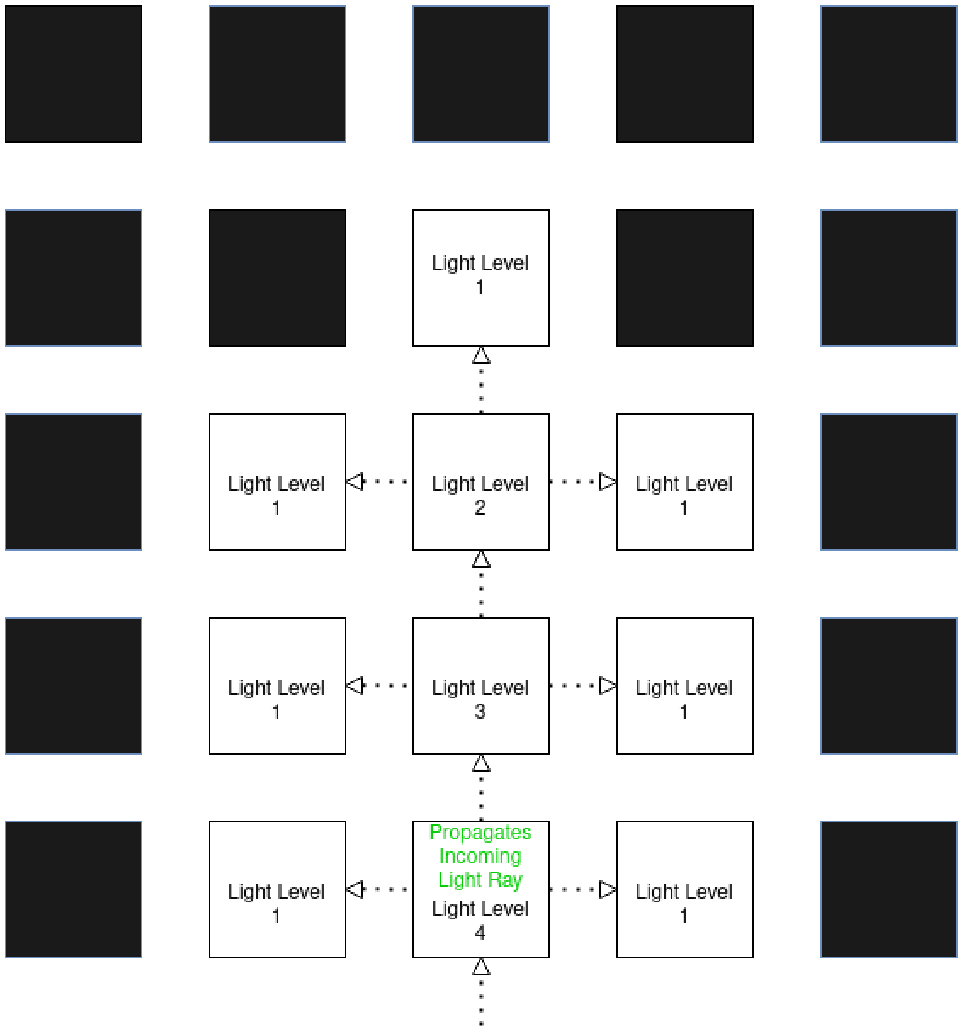
```
^&..............+
^...............#
```

Ideally, we would want to to look more like this, just to make this method easier on the eyes:



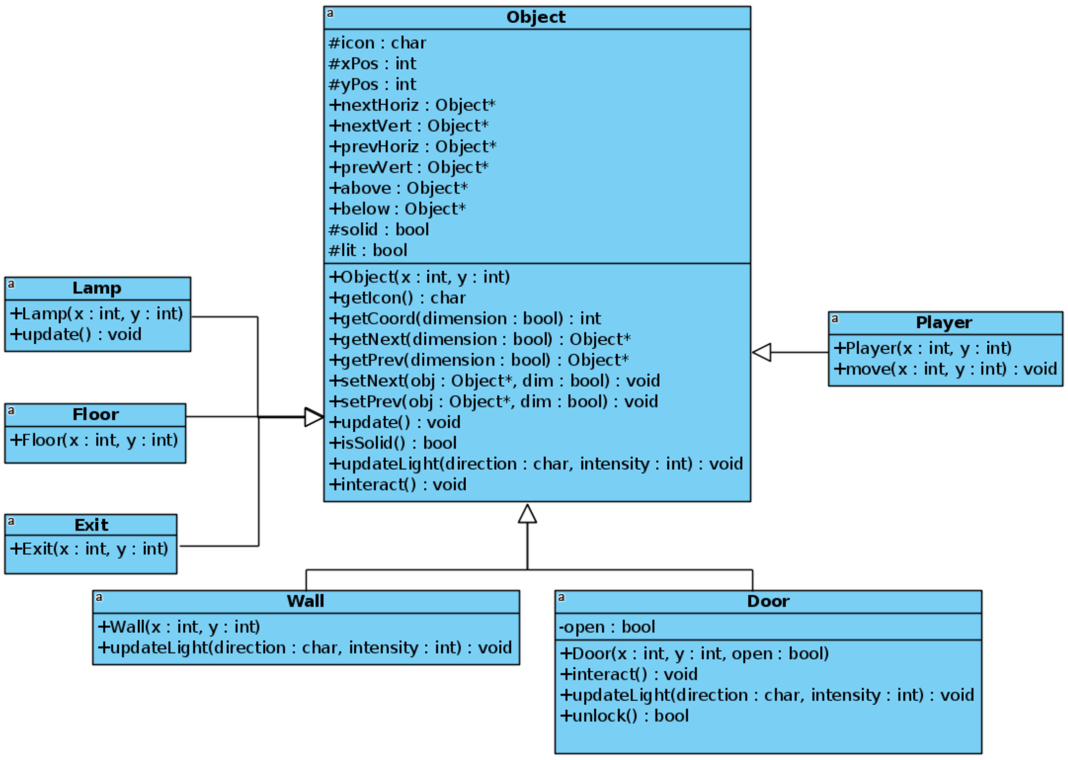
```
################
^&..............+
^...............#
################
```

Therefore, in addition to propagating the light in the same direction it was received, the light also needs to be 'diffused' to the two adjacent sides, for exactly one tile. Basically, the objects on either side need to have the algorithm called on them, with an intensity of 1. Only they will be illuminated, and won't propagate the effect further. See below for a visual (black squares represent potentially unlit tiles)

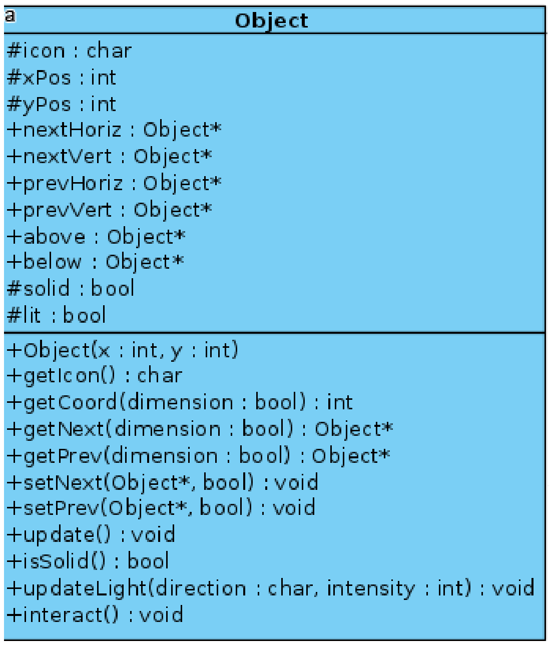


5.3 Updates to Classes

Here is updated UML for existing and new classes related to *Object*. All objects inherit **publicly** from *Object*


Object
#icon : char
#xPos : int
#yPos : int
+nextHoriz : Object*
+nextVert : Object*
+prevHoriz : Object*
+prevVert : Object*
+above : Object*
+below : Object*
#solid : bool
#lit : bool
+Object(x : int, y : int)
+getIcon() : char
+getCoord(dimension : bool) : int
+getNext(dimension : bool) : Object*
+getPrev(dimension : bool) : Object*
+setNext(obj : Object*, dim : bool) : void
+setPrev(obj : Object*, dim : bool) : void
+update() : void
+isSolid() : bool
+updateLight(direction : char, intensity : int) : void
+interact() : void
Lamp
+Lamp(x : int, y : int)
+update() : void
Floor
+Floor(x : int, y : int)
Exit
+Exit(x : int, y : int)
Player
+Player(x : int, y : int)
+move(x : int, y : int) : void
Wall
+Wall(x : int, y : int)
+updateLight(direction : char, intensity : int) : void
Door
-open : bool
+Door(x : int, y : int, open : bool)
+interact() : void
+updateLight(direction : char, intensity : int) : void
+unlock() : bool

Additions and changes required are specified below. Please note that virtual methods are not indicated in the UML, see the method description for whether or not a function is virtual.


Object
#icon : char
#xPos : int
#yPos : int
+nextHoriz : Object*
+nextVert : Object*
+prevHoriz : Object*
+prevVert : Object*
+above : Object*
+below : Object*
#solid : bool
#lit : bool
+Object(x : int, y : int)
+getIcon() : char
+getCoord(dimension : bool) : int
+getNext(dimension : bool) : Object*
+getPrev(dimension : bool) : Object*
+setNext(Object*, bool) : void
+setPrev(Object*, bool) : void
+update() : void
+isSolid() : bool
+updateLight(direction : char, intensity : int) : void
+interact() : void

Variables:

- **lit : bool**
  This indicates whether or not the object is lit (if it can be seen, has light shining on it). When it is constructed, an object is lit by default.

- **solid : bool**
  This indicates whether or not an object is solid. If it is solid, a player cannot walk through it (e.g a wall). Otherwise a player can stand on it (e.g floor).

Functions:

- **update() : void**
  Virual (but not pure virtual) method. This is the method called when an object needs to be updated, and its behaviour is that it sets *lit* to false. Any deriving class has the option to override this function.

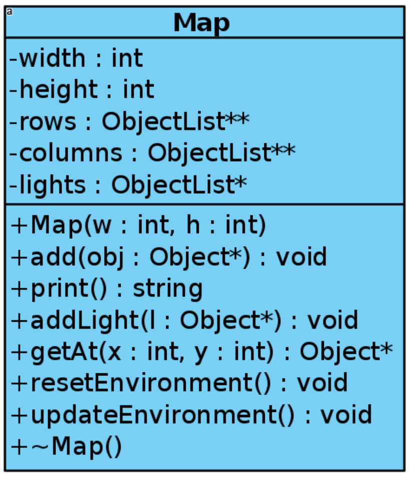
- **getIcon() : char**
  If an object is not lit, a space character needs to be returned instead.

- **isSolid() : bool**
  A getter for *solid*. If the object is not solid, it must still register as solid if the object on top of it (via *above*) is solid. This is for collision detection.

- **interact() : void**
  Virtual (not pure virtual). This is called whenever the player wants to interact with something. By default, this does nothing, and a child class overrides this method if it can be interacted with

- **updateLight(direction : char, intensity : int) : void**
  Virtual (not pure virtual) method. This is the recursive ray-tracing algorithm, or at least part of it. When an incoming right ray 'hits' this object, the *lit* member is set to true and the method is called on the next object to propagate the light in a specific direction. The parameters passed in are the direction (up ('u'), down ('d'), left ('l'), right ('r')) and the light intensity. When a light ray is propagated, *updateLight(...)* is called on the next object in that direction. The default behaviour is to treat the object as one that light can pass through (note that this is irrespective of whether or not an object is solid. For instance a hypothetical window is solid, but lets light pass through). Any class that does not work that way, overrides this method. When the light level passed in reaches 1, then *lit* is set to true, but no other objects around it are lit further. Remember that the surrounding tiles need to have the algorithm called on them as well, with a light level of 1 (as discussed in the *lighting* section)

5.4 Map



Map
-width : int
-height : int
-rows : ObjectList**
-columns : ObjectList**
-lights : ObjectList*

+Map(w : int, h : int)
+add(obj : Object*) : void
+print() : string
+addLight(l : Object*) : void
+getAt(x : int, y : int) : Object*
+resetEnvironment() : void
+updateEnvironment() : void
+~Map()

Map needs the following additions: **Variables:**

- **lights : ObjectList***
  A list of all the stationary light sources on the map, for more convenient access

Functions:

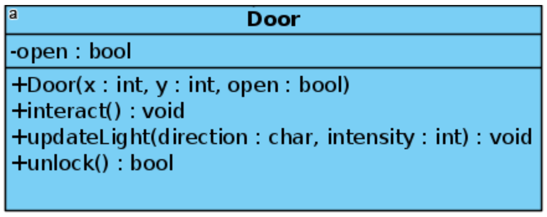
- **getAt(x : int, y : int) : Object***
  This is mostly a multi-purpose helper, should you ever want the floor-level object on the map, at point (x, y). If nothing is at that point for whatever reason, return null.
- **addLight(light : Object*) : void**
  When a light source object is created, it is created as an object that exists above a floor tile. The caller should add the light to the map's light list so that the lights get updated. Therefore this method adds an object to the light list, which is an object list. An objectlist links objects along one of the axes, you can choose whichever you want (because it has the iterate method, which will keep track of which direction it's linked in.). The links of the lights are otherwise unused, since they are above floor tiles, like the player object, which also has its next/prev pointers constantly null.
- **resetEnvironment() : void**
  This method calls the update method on every floor-level item on the map. So, iterate through either the rows or columns, and update each object in that row/column. For most objects this will set them to 'unlit'
- **updateEnvironment() : void**
  After the environment has been reset, the lighting needs to be re-calculated since something may have changed. Thus what this method does, is call *update()* on each light source in the *lights* list.

5.5 Exit

When a player reaches (and is on top of) an exit, the game ends. This will be implemented in the *Game* class. *Exit* actually does not have any interesting features. A reference to the exit is kept by the game, and upon each update the game checks if the player and the exit are on the same spot (more on that later). The icon for an exit is '@', and it is not solid (otherwise it would be useless, but funny when someone tries to leave).

5.6 Door

The door is an object that can be interacted with, either opened or closed. When open, its icon is '=', and when closed its icon is '+'. Whenever it is either opened or closed, the icon (and other variables) need to be updated.


a
Door
-open : bool
+Door(x : int, y : int, open : bool)
+interact() : void
+updateLight(direction : char, intensity : int) : void
+unlock() : bool

Variables:

- **open : bool**
  Whether or not a door is open. When a door is open, it is not solid, and when a door is not open, it *is* solid since a player cannot walk through it.

Functions:

- **Door(x : int, y : int, open : bool)**
  Constructor for the door class. Sets all members appropriately, including *icon* and *solid*.

- **interact() : void**
  Opens or closes the door. If it is open, close it and **throw** a string "You closed a door". If it is closed, open it and throw a string "You opened a door". This string will be caught by a higher class. Be sure to throw a *string*, and not *const char**.

- **updateLight(direction : char, intensity : int)**
  The corresponding function in the base *Object* class was declared virtual, so as a derived class *Door* can override it. Door can either propagate light like the base object class, or prevent light from continuing like a wall. This depends on whether or not it is closed.
  Hint: Remember that you can explicitly call the version of a function belonging to a parent class, using ::

5.7 Wall and Floor

Floor has minimal change, other than that the constructor now needs to set *solid* to false. Wall needs to be solid, and override the *updateLight(...)* method to only light itself and then not propagate a light ray any further.

5.8 Lamp

The lamp class represents an object that is a light source. The icon for the lamp is '^' (a circumflex), and a lamp is solid. Its two functions are:

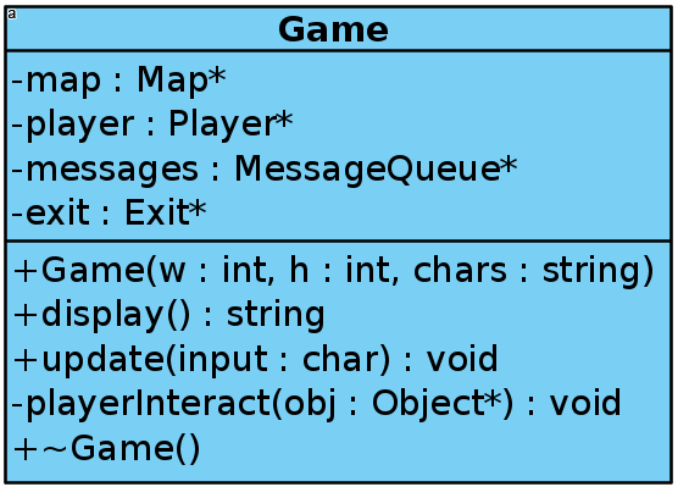
- **Lamp(x : int, y : int)**
  The constructor, setting position and icon.

- **update() : void**
  An override of the **Object** update. When a lamp is updated, it sends out light rays in the four

directions around it. For this specific light source, it should light up 20 tiles in all directions. That is, it calls the adjacent tiles' *updateLight* with a starting light intensity of 20. Also note that a lamp is not technically a floor-level tile. So, here 'adjacent tile' means an adjacent tile of the floor beneath it.

5.9 Game

While the *Game* class has some new variables and functions, it also requires some updates to existing methods


a
Game
-map : Map*
-player : Player*
-messages : MessageQueue*
-exit : Exit*
+Game(w : int, h : int, chars : string)
+display() : string
+update(input : char) : void
-playerInteract(obj : Object*) : void
+~Game()

Variables:

- **messages : MessageQueue***
  A pointer to the message queue the game will use
- **exit : Exit***
  A reference to the exit of the level. There is only one at most.

Functions:

- **Game(w : int, h : int, chars : string)**
  The constructor needs to be updated to add the new items in the game to the map, should they be encountered. An exit has its own member in a game instance, and lights need to be added differently (with floors beneath them). Also create doors correctly, based on whether or not they are open/closed/locked. Other changes include needing to create a message queue that makes messages persist for **4 steps** of the game. At the end of the constructor, reset and update the map once to initialise the lighting for the first display.
- **update() : void**
  Firstly, if there is no player, a message needs to be added to the message queue "Missing player", and obviously the player does not get updated (the rest of the game does though). Secondly, now whenever a player is about to take a step in one of 8 directions, the game needs to check that the object at the target spot is not solid. If it is solid, a message should be added to the

message queue, "Walked into something". If the player attempts to walk out of bounds, the message "Out of bounds" needs to be added, and the player prevented from moving. After the player is moved, the map needs to have its environment reset. Then, the map environment needs to be updated. Finally, if the player is standing on the exit, **throw** the string "You reached the exit!". This is just to mark the end of the game, and will be **caught** in the main function.

  In addition to this process, a new user input needs to be defined: 'e' for interact. When a user presses e, all the objects around it need to be interacted with. Specifically only the object at the top, bottom and to its sides need to be interacted with, not any object diagonally around it. To do this, find the relevant pointers (remember to go to *floor-level first*), and call *playerInteract(Object*)* on that object.

- **display() : string**
  Display now needs to output first all the messages in the game, then the game map. The first row of the map should be directly after the last message, with only a newline separating them (not a blank line in-between).

- **playerInteract(obj : Object)**
  This is a helper function, used by **Game::update()**. It takes in an object, then finds the object that is at the top of the 'above' chain in its position. It then calls *interact()* on that object, and catches any string messages it may throw as a result. If a string is caught, it needs to be added as a new message to the game message queue. Also remember to add a null check, in case the passed-in object is not valid.

6 Some tips

Really try to test your code well before moving on to the next part. Not only to get marks for that part on fitchfork, but because even fitchfork cannot catch all your bugs. When you are fully confident that your code works, you can move on. By the end, problems that have been there from the start become **IMMENSELY** difficult to find, especially when linked lists are involved.

Regarding the second main:
There are several maps available in the second main. You are supposed to uncomment the map string and map dimensions for the one you want to use and test. Have every other map commented out, only the one you want to use should be uncommented. Then, compare the output of that map with the given output at the bottom of the main, in the given scenario. Since moving the character around is tedious, the output for every single game update was not given, and you need to compare 'snapshot' situations with the given output.

7 Uploading

Upload the following files in an archive:

- **Part 1**
  1. Game.h, Game.cpp
  2. Object.h, Object.cpp
  3. Wall.h, Wall.cpp
  4. Floor.h, Floor.cpp
  5. Map.h, Map.cpp
  6. Player.h, Player.cpp
  7. ObjectList.h, ObjectList.cpp
- **Part 2**
  1. Game.h, Game.cpp
  2. Object.h, Object.cpp
  3. Wall.h, Wall.cpp
  4. Floor.h, Floor.cpp
  5. Map.h, Map.cpp
  6. Player.h, Player.cpp
  7. ObjectList.h, ObjectList.cpp
  8. Door.h, Door.cpp
  9. Message.h, Message.cpp
  10. MessageQueue.h, MessageQueue.cpp
  11. Lamp.h, Lamp.cpp