Assignment 4

Overview

The program in Assignment 3 has some flaws: the complexity of the main function, the restrictions imposed by the representation of the `World`, the poor handling of items, and the absence of obstacles for the player to overcome.

The `main` function can be simplified by refactoring some of the code into an additional software layer immediately below it. This new layer will manage the components of the game. By refactoring the program in this fashion, the `main` function can be entirely concerned with the handling of input, which will make it easier to read.

The representation of the `World` can be made more versatile by using a list of nodes instead of a two-dimensional (2D) array. The information for a node will tell which other nodes it is connected to and its description. With this format, a long straight path can be represented by a single link instead of a line of nodes, which will make the game more enjoyable for the player. It will also be possible to have different descriptions on different death nodes.

The other flaws will be handled in later assignments. Items will be loaded from a file in Assignment 5 and obstacles will be added in Assignment 6.

The purpose of this assignment is to ensure that you understand how to refactor a program and how to use composition to break a program into distinct layers. For Part A, you will implement an `ItemManager` class to manage the Items. For Part B, you will implement a `Game` class to manage some of the functionality previously contained in the `main` function. For Part C, you will adapt your `main` function to use the new `Game` class. For Parts D to F, you will change the representation of the world so that the `World` class stores a one-dimensional (1D) array of nodes rather than a 2D array of descriptions.

The New Format for the World

Starting in Part D, you will convert the world to be represented as a 1D array of nodes. Each node stores links to which other node (if any) is to the north, south, east, and west. In consequence, the world will now be a **directed graph** (a set of nodes and arrows (named directed arcs) between them). In practice, if going north from node A leads to node B, going south from node B will generally lead to node A. However, exceptions can be handled smoothly, which can lead to interesting features in the game.

The first line in the file is an integer indicting the number of nodes to load. The second line has four integers, the (1) index of the start node, (2) the index of the victory node, (3) the index of the start game message, and (4) the index of the end game message. (The first two will be used as indexes in the node array and the other two will be used as indexes in the description array.) After that, there is one line for each node, starting with node 0. Note: the node indexes do not appear in the node lines; your program has to keep track that the first node line describes node 0, the second node line describes node 1, etc.

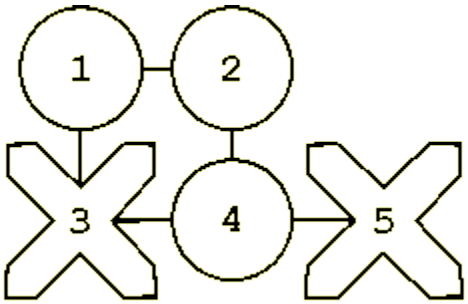
Each node line starts with an `'N'`, `'D'`, or `'O'` character, followed by the description index, followed by the node indexes of the nodes to the north, south, east, and west. If the character was a `'D'`, this is a **death node**; otherwise it is not. If the character was `'O'`, this is an **obstacle node**. In the input line for an obstacle node, there are a `char` and two more numbers at the end of the line, which will be used

in Assignment 6. For now, just read them in and ignore them. Note that the first node is a placeholder, because `0` is still used as the destination for inaccessible links.

A data file for a simple world in the new format is as follows:

```
6
2     1     1     2
N     0     0     0     0     0
O     3     0     3     2     0     a     5     6
N     4     0     4     0     1
D     6     1     0     4     0
N     5     2     0     5     3
D     6     0     0     0     4
```

The above world can be visualized as shown in the following map:



In the map, the X nodes are death nodes (`'D'`) and the circular nodes are the traversable nodes. Node `1` corresponds to the obstructed (`'O'`) node and nodes `2` and `4` correspond to the regular (`'N'`) nodes. We will handle obstructed nodes in Assignment 6. For now, your program should treat them as ordinary nodes and it should ignore the last three values on the line. (Your program must still read past these values in some fashion, either by putting them in dummy variables or using `getline` or `.ignore()`.)

Note that node `3` (the first death node) uses the same description (`6`) as node `5` (the other death node). The player starts at node `2` and needs to move to node `1` to win.

Requirements

Copy the code and data files of your Assignment 3. Do not just modify Assignment 3.

(If you are using Visual Studio, you must start by creating a new project for Assignment 4. Do NOT copy the whole folder including the `.sln` file or massive confusion will result!)

Part A: The `ItemManager` Class [35% test program]

Create a class named `ItemManager` to keep track of the items in the game. Much of the functionality you will need for the `ItemManager` class is already in your `main` function.

By the end of Part A, your `ItemManager` class will have `public` member functions with the following prototypes:

- **`ItemManager (const string& game_name);`**
- **`unsigned int getCount () const;`**

- `int getScore () const;`
- `void printAtLocation (const Location& location) const;`
- `void printInventory () const;`
- `bool isInInventory (char id) const;`
- `void reset ();`
- `bool take  (char id, const Location& player_location);`
- `bool leave (char id, const Location& player_location);`

Note: There is no default constructor.

The `ItemManager` class will also have `private` member functions with the following prototypes:

- `unsigned int find (char id) const;`
- `bool isInvariantTrue () const;`

Perform the following steps:

1. Create a class named `ItemManager` that contains an array of `ITEM_COUNT Item`s as a member variable, i.e., the type of the array should be `Item` and there should be `ITEM_COUNT` entries in the array. The `ItemManager` class should have its own header and source files (you should now be able to set these up yourself and put the right parts in each file). Move the `ITEM_COUNT` constant from the `main.cpp` file to the header file; place it before the class statement for the `ItemManager` class.
    - **Reminder:** You will need to use a `#include` statement to include `Item.h`.
2. Add a constructor that takes a `string` representing the game name as a parameter. It should initialize the same `Item`s as in `main.cpp` from previous assignments. We will not use the game name in this assignment, but we will need it in Assignment 5. The items should be specified in some order that is not sorted by ID character, such as the order the items were listed in Assignment 3.
    - **Note:** The `Item`s need to be unsorted because we will sort them in Assignment 5. If they were already sorted, testing our sort command would be difficult.
    - **Reminder:** Every constructor must set the value of every member variable. In this case, that means initializing each of the `ITEM_COUNT Item`s.
    - **Hint:** If you are using the solution from the website, you want the items in the `initAllItemsGhostwood` function, not the `initAllItemsBlizzard` function.
3. Add a `getCount` function that returns the number of items in the `ItemManager`. In Assignment 4, this function should just return `ITEM_COUNT`.
4. Add a `getScore` function that calculates and returns the player's current score.
    - **Hint:** You have code to do this in `main.cpp`.
    - **Reminder:** The player's current score is the sum of the scores (also named points) for the individual items.

5. Add a `printAtLocation` function to print all the `Item`s at a location provided as a parameter. Each `Item` should be printed on its own line.

    - **Hint:** You have code to do this in `main.cpp`, perhaps in an existing function.

6. Add a `printInventory` function that prints the `Item`s in the player's inventory. Each `Item` should be printed on its own line.

7. Add a `reset` function that resets all the `Item`s.

    - **Hint:** Look in `Item.h` to find a suitable function and then call it for each item in the item array.

8. Add a `private` helper function named `find` that takes an item id as a parameter. If there is such an item, it should return that item's index. Otherwise, it should return `NO_SUCH_ITEM`, a constant with value `999999999`. (This constant can be declared in the `ItemManager.h` or the `ItemManager.cpp` file.) For now, just use a linear search.

    - **Note:** We will use a binary search in Assignment 5.

9. Add an `isInInventory` function that takes an item id as a parameter and returns a value indicating whether that item is in the player inventory. It should call the `find` function to find the correct item in the array. If the item is not in the array, it should return `false`. Otherwise, it should return whether the item with that index is in the player inventory.

10. Add a `take` function that takes an item id and the player location as parameters and attempts to take that item. If the player successfully took the item, return `true`. If the player failed to take it, print `"Invalid item."` (with a newline) and return `false`.

    - **Reminder:** The player can take an item if it is at the same location and not otherwise.

    - **Hint:** There is code to take an item in `main.cpp`. Move the part of the code that searches through the item array to ItemManager.cpp. Do <u>not</u> move the parts of the code that get the user input and ensure it is not an empty string (`""`).

11. Add a `leave` function similar to the `take` function. The player can only leave items that are in the inventory. If the player cannot leave the item, print `"Invalid item."` (with a newline).

12. Test your `ItemManager` module with the `TestItemManager4A.cpp` program provided. You will also need the `TestHelper.h` and `TestHelper.cpp` files. Run the resulting program. It should give you full marks.

    - **Hint:** `g++ Location.cpp Item.cpp ItemManager.cpp TestHelper.cpp TestItemManager4A.cpp`

13. Add a class invariant that is checked by a `private` helper function named `isInvariantTrue`. The class invariant requires that all the items are initialized.

    - **Hint:** There is an `Item` function that checks whether an item is initialized.

- **Hint:** The `isInvariantTrue` function should return `true` if everything is good and `false` if even one thing is wrong. Therefore, you should structure it as a series of checks that each return `false` if that check fails. At the end of the function return `true`.

14. Use `asserts` to check that the class invariant at the end of each `public` member function that is not declared as `const` (including the constructor). If the function has a `return` statements, check the invariant immediately before it. If it has more than one `return` statement, check the invariant immediately before each of them.

    - **Reminder:** You will need to `#include <cassert>`.
    - **Reminder:** `assert(isInvariantTrue());`
    - **Reminder:** You do not have to check the class invariant at the end of `const` functions because they cannot change the class state.

15. Use `asserts` to check that the class invariant at the beginning of each `public` member function except the constructor. This includes `const` functions.

    - **Reminder:** Do not check the class invariant at the start of a constructor because the class state will (probably) not be correct yet. The constructor will make it correct.
    - **Warning:** Do not check the invariant in `isInvaraintTrue`. If you do, your program will crash.

Part B: The `Game` Class [10% code, otherwise marked with Part C]

Create a class named `Game` to represent the complete state of the game. Much of the functionality you will need for the `Game` class is already in your `main` function. The `Game` class will also make use of the `ItemManager` class.

By the end of Part B, your `Game` class will have `public` member functions with the following prototypes:

- `Game (const string& game_name);`
- `bool isOver () const;`
- `void printStartMessage () const;`
- `void printEndMessage () const;`
- `void printDescription () const;`
- `void printInventory () const;`
- `void printScore () const;`
- `void reset ();`
- `void moveNorth ();`
- `void moveSouth ();`
- `void moveEast ();`
- `void moveWest ();`
- `void takeItem (char id);`
- `void leaveItem (char id);`

Note: There is no default constructor.

The `Game` class will also have a `private` member function with the following prototype:

- **`bool isInvariantTrue () const;`**

Perform the following steps:

1. Create a `Game` class that contains a `World`, an `ItemManager`, and a player `Location`.
2. Add a `Game` constructor that takes a `string` representing the game name as a parameter. It should initialize the `World` and the `ItemManager` with the game name and set the player `Location` to the start location for the `World`.

    - **Reminder:** The `World` and `ItemManager` classes do not have default constructors. Therefore, you will need to initialize them using an initializer list. The syntax is:

      ```
      MyClass :: MyClass (int param1, float param2)
            : member_variable1(param2),
              member_variable2(arg2, param1),
              member_variable3()
      {
            // code here
      }
      ```

      Note that the list starts with a colon, the terms are separated by commas, and that there are no semicolons. Each term gives the name of a member variable in the class and then, in parentheses, the parameters it needs for one of its constructors. You can pass the parameters from this constructor to the constructors for the member variables.

    - **Hint:** Review the online notes for Section 8-1on Composition, especially the extra file showing the bicycle class.

    - **Reminder:** Every constructor must set the value of every member variable. For the `Game` class, this is the world, item manager, and the player location.

    - **Hint:** The easiest way to initialize the player location is with an assignment statement in the constructor body. The player should start at location `world.getStart()`. We want to initialize the player location last so the world has its values before we try to look at them.

    - **Warning:** You can initialize the player location using an initializer list instead of in the function body. However, if you do, remember that initialization is performed in the order that the member fields are listed in the class definition in the header file. Thus, if you want to initialize the player location based on the world, the world must be listed before the player location in the class definition in the header file.

3. Add an `isOver` function that returns whether the game is over. The game is over if the player is at a death location or at the victory location.
4. Add the `printStartMessage` and `printEndMessage` functions. They should each call the corresponding function in `World`.

5. Add a function named `printDescription` that prints the description for the player's current location and then prints the descriptions of any items at that node.
    - **Hint:** You can code this function by calling one function from one class and then another function from another class.
6. Add the `printInventory` function. It should call the corresponding function in `ItemManager`.
7. Add the `printScore` function. It should print a sentence saying the player's total score.
    - **Hint:** Call a function in the `ItemManager` class.
8. Add a `moveNorth` function that attempts to move the player north. If the player succeeds at moving to the node that is to the north of the player's current position, then print the description for that node and any items there. If the player cannot move that way, print an appropriate message. Also add the `moveSouth`, `moveEast`, and `moveWest` functions.
    - **Hint:** Use a function to print the descriptions of the node and the items there.
9. Add `takeItem` and `leaveItem` functions. They should call the corresponding functions in `ItemManager` using the player location as a parameter.
10. Add a `reset` function to the `Game` class. It should move the player back to the start location and reset all the items. The `World` never changes, so it does not need to be reset.
    - **Hint:** Use a function from the `ItemManager` class to reset the items.
11. Add a class invariant checked by a `private` helper function named `isInvariantTrue`. The class invariant requires that the player location is valid according to the `isValid` function of the `World` class, which can be called using the `World` field of the `Game` class.
12. Use `assert`s to check that the class invariant is true at the end of each `public` member function that is not declared as `const` (including the constructor).
13. Use `assert`s to check that the class invariant is true at the beginning of each `public` member function except the constructor.

Part C: Update the `main` Function [30% = 10% stability + 20% test scripts]

Adapt your `main` function to use the new `Game` class. Overall, you will be removing a lot of code from the `main` function.

Perform the following steps:

1. Replace the world, player location, and item variables with a single `Game` variable.
    - **Hint:** You will need to invoke a constructor for the `Game` variable.
    - **Hint:** Call the `Game` functions from Part B.

- **Note:** If you are using the provided solution, also remove the helper functions in `main.cpp`. They are no longer needed.

2. Replace the various code that uses the world, player location, and items with calls to `Game` functions. The replacement function calls will often be considerably shorter. The game should end if the `Game::isOver` function returns `true` or if the player enters and confirms a `'q'` (quit) command. The `main` function should handle asking the user which item to take or to leave.

    - **Hint:** If the player always dies at game start, you probably didn't initialize the player variable in the game constructor.

    - **Hint:** When taking and leaving items, leave the code that asks which item to take in the main function. Then, after checking that the input was not an empty string, pass the first character in the string into the appropriate function.

3. Add an `'r'` command to restart the game. It should require a confirmation, just like the quit command. If the player does indeed want to restart the game, then print the current score followed by a blank line, reset the game, and then print the new current description.

4. Test your program with the five test cases provided: `testcase4A.txt`, `testcase4B.txt`, `testcase4C.txt`, `testcase4D.txt`, and `testcase4E.txt`.

    - **Hint:** `g++ Location.cpp World.cpp Item.cpp ItemManager.cpp Game.cpp main.cpp`

    - **Note:** These same test cases will be used after Part F. All the marks for them are here.

    - **Note:** If you get a message saying `Error while reading ghostwood_text.txt`, then you should copy/paste a fresh copy of the `ghostwood_text.txt` file from the Assignment website. Somehow you have the wrong file type (Windows versus replit).

Copy all your files to a folder named Assignment 4A before continuing. If you cannot make your program work after completing Parts D, E, and F, you should also hand in the Assignment 4A copy. Now continue with Assignment 4.

Part D: Change the `Location` Change [5% test program]

Change the `Location` type to store a single node index instead of a row and column. By the end of Part D, your `Location` class will have `public` member functions with the following prototypes:

- `Location ();`
- ~~`Location (int row1, int column1);`~~
- **`Location (unsigned int index);`**
- `bool operator== (const Location& other) const;`
- **`bool isInaccessible () const;`**

It will also still have an associated non-member function with the following prototype:

- `ostream& operator<< (ostream& out,`
  `const Location& location);`

Perform the following steps:

1. Change the `Location` class to store a single `unsigned int` representing a node index. Remove the row and column member variables.
2. Change the initializing constructor to take a node index as a parameter instead of a row and column.
   - **Reminder:** Every constructor must set the value of every member variable.
   - **Note:** C++ is surprisingly generous in calling single-argument constructors to automatically convert values to the correct type. In short, you can use the argument almost anywhere you need the constructed type. All of the following are legal:
     ```
     Location location(32);
     Location location = 32;  // calls constructor
     functionWithALocationParam(7);  // calls constructor
     ```
3. Add an `isInaccessible` member function. It should return `true` if the node index is `0` and `false` if it is any other value.
4. Update the default constructor. It should create an inaccessible location.
5. Update the stream insertion operator (`operator<<`) to print the node index. If the location represents an accessible node, it should print:
   `(node = 4)`
   where `4` is the node index. If the location represents an inaccessible node, the function should print:
   `(node = INACCESSIBLE)`
   - **Note:** You must match this format exactly for the test program to give you the marks.
6. Test your `Location` module with the `TestLocation4.cpp` program provided. You will also need the `TestHelper` module. The test program should give you full marks.
   - **Hint:** `g++ Location.cpp TestHelper.cpp TestLocation4.cpp`
7. In your `ItemManager` class, update the `Locations` for the items. For the `ghostwood` map, the new item locations are:
   - Scarab beetle `'s'` is at location `3`
   - Candlestick `'c'` is at location `7`
   - Key `'k'` is at location `13`
   - Tarantula `'t'` is at location `19`
   - Book `'b'` is at location `22`
   - Moth `'m'` is at location `37`

- Pendant `'p'` is at location `52`
- Dagger `'d'` is at location `53`
- Ring `'r'` is at location `58`

8. Re-test your `ItemManager` module (from Part A) with the `TestItemManager4D.cpp` program provided. It is the same as `TestLocation4A`, except that it uses the updated `Location` type. It should give you the same mark as before.
    - **Hint:** `g++ Location.cpp Item.cpp ItemManager.cpp TestHelper.cpp TestItemManager4D.cpp`

Part E: The `Node` Class [8% test program]

Implement a class named `Node` to represent a single node in the world. By the end of Part E, your `Node` class will have public member functions with the following prototypes:

- `Node ();`
- ```
  Node (unsigned int description1,
        unsigned int north1,
        unsigned int south1,
        unsigned int east1,
        unsigned int west1,
        bool is_death1);
  ```
- `void debugPrint () const;`
- `unsigned int getDescription () const;`
- `Location getNorth () const;`
- `Location getSouth () const;`
- `Location getEast  () const;`
- `Location getWest  () const;`
- `bool isDeath () const;`

Note: There is no class invariant.

Perform the following steps:

1. Create a class named `Node` that contains a description number, the index of the node in each direction, and a `bool` representing whether this is a death node.
    - **Hint:** You can represent the node indexes as `unsigned int`s.
2. Add a default constructor to initialize the descriptions and node indexes to `0`. Make sure that the new `Node` is not a death node.
    - **Reminder:** Every constructor must set the value of every member variable.
3. Add an initializing constructor that initializes the fields to the specified values.
  ```

4. Add a `debugPrint` function that prints the values for the `Node` in the same format as the node data file, ending with a newline. As the first character, print a `'D'` for death nodes and an `'N'` for all other nodes.

    - **Note:** Print the indexes of the nodes in each direction as integers, not in the format used for `Location`s.

    - **Reminder:** In this assignment, the `Node` data structure stores nothing about obstructed nodes. So they will print like ordinary nodes (with an `'N'`) with no extra code needed.

5. Add a `getDescription` function that returns the index of the node description.

6. Add an `isDeath` function that uses information from the `Node`.

7. Add a `getNorth` function that returns a `Location` containing the index of the node to the north. Add similar `getSouth`, `getEast`, and `getWest` functions.

8. Test your `ItemManager` module with the `TestNode4.cpp` program provided. You will also need the `TestHelper` module. The test program should give you full marks.

    - **Hint:** `g++ Location.cpp Node.cpp TestHelper.cpp TestNode4.cpp`

Part F: Change the `World` Representation [12% test program]

Change the `World` class to represent the nodes as a 1D array of `Node` objects. The interface for the `World` class **does not change**. It should still have `public` member functions with the following prototypes:

```
World (const string& game_name);
void debugPrint () const;
bool isValid (const Location& location) const;
bool isDeath   (const Location& location) const;
bool isVictory (const Location& location) const;
bool canGoNorth (const Location& location) const;
bool canGoSouth (const Location& location) const;
bool canGoEast  (const Location& location) const;
bool canGoWest  (const Location& location) const;
Location getNorth (const Location& location) const;
Location getSouth (const Location& location) const;
Location getEast  (const Location& location) const;
Location getWest  (const Location& location) const;
Location getStart () const;
void printStartMessage () const;
void printEndMessage () const;
void printDescription (const Location& location) const;
```

The `World` class will also still have the following `private` member functions:

```
void loadNodes (const string& filename);
```

- `void loadDescriptions (const string& filename);`
- `bool isInvariantValue () const;`

Perform the following steps:

1. Replace the 2D array of node values with a 1D array of `Node`s and an `unsigned int` representing the node count. The array should have a size of `MAX_NODE_COUNT`, a new constant with a value of `1000`. Also add two fields to represent node indexes for the start and victory nodes, and two fields to represent the description indexes of the start game and end game messages.
    - Hint: The `World` class should already have an `unsigned int` that stores the number of description messages. Make sure your `loadDescriptions` function uses this variable instead of declaring a new variable so that you can keep track of how many descriptions have been loaded.
2. Remove the `NodeValue` type and the `ROW_COUNT`, `COLUMN_COUNT`, `INACCESSIBLE`, `START_MESSAGE`, `END_MESSAGE`, `DEATH_NODE`, `START_NODE`, and `VICTORY_NODE` constants.
3. Change the `NO_SUCH_VALUE` constant to be a `Location` with node index `-1`.
    - **Hint:** Declare the constant in `Node.h` before the `Node` class.
4. Change the `loadNodes` function to load data in the new format. The filename will be `*_nodes.txt` instead of `*_grid.txt`; for example, `ghostwood_nodes.txt`.
    - **Hint:** Examine the ghostwood_nodes.txt file to understand the format of the data that needs to be read. Or see the explanation at the beginning of this assignment.
        - Read the node count into the member variable using the insertion operator (i.e., >>).
        - Read the four values (the starting node number, the victory node number, the number of the message for starting the game, and the number of the message for ending the game) into the appropriate member variables.
        - Use a FOR loop to read all the nodes (the number of nodes is given by the node count variable). For each iteration, read one node, which is complicated:
            - First read the type of the node by reading one letter into a char variable (the char will be 'N', 'D', or 'O').
            - Check the value of the char to determine whether or not this node is a death node (the char 'D' denotes a death node).
            - Then read the other values from the node line, beginning with the description index, the index of the node to the north of this node, the index of the node to the south of this node, etc.
            - Then if this node is a death node, read in three more values into dummy variables.

- **Hint:** Look up the syntax for the constructor for a `Node`. You can set individual nodes in your array using a constructor and an assignment operator. For example:
  ```
  nodes[n] = Node(1, 2, 3, 4, 5, false);
  ```
- **Hint:** Make sure you read the values into variables of the correct type. The node type is a `char`. The description number and the directions are `int`s (or `unsigned int`s). If this is an obstructed node, the first extra value is a `char` and the other two are `int`s (or `unsigned int`s). If you use the wrong types, the file will probably not load correctly.

5. Update the `debugPrint` function. It should start by printing the number of nodes, the start node, and the victory node. Then is should call the `debugPrint` function for each `Node`. Then it should print the descriptions using the same format as in Assignment 3.
6. Update the `isValid` function to check if the node index for the `Location` parameter is strictly less than the node count.
7. Update the `printStartMessage` and `printStartMessage` functions to use the member variables instead of the (removed) constants.
8. Update the `printDescription` function to get the description index from the `Node` before printing the corresponding description.
9. Update the `canGoNorth` function to return a value based on the `Location` to the north. If the `Location` is inaccessible, the function should return `false`. Otherwise, it should return `true`. Also update the `canGoSouth`, `canGoEast`, and `canGoWest` functions.
   - **Hint:** There is a function in the `Location` class that will tell you whether a location is inaccessible.
10. Update the `getNorth` function to get its value from the `Node`. It should still have the same `assert` checks. Also update `getSouth`, `getEast`, and `getWest`.
11. Update the `isDeath` function to return whether the `Node` specified by the `Location` is a death node.
12. Update the `isVictory` function to check if the `Location` is the victory node.
    - **Hint:** You have a member variable that will be helpful.
13. Update the `getStart` function to return the start node. It no longer has to search the nodes.
    - **Note:** There are no longer any circumstances in which this function returns `NO_SUCH_VALUE`. This is not a problem.
14. Update the class invariant to match the changes to the node and description formats. In addition to the previous checks, the class invariant should now also require that
    - The node count is less than or equal to `MAX_NODE_COUNT`
    - The indexes of the start and victory nodes are strictly less than the node count

- The indexes of the game start and end messages are strictly less than the description count

15. Test your `World` module with the `TestWorld4.cpp` program provided. You will also need the `TestHelper` module. The test program should give you full marks.

    - **Hint:** `g++ Location.cpp Node.cpp World.cpp TestHelper.cpp TestWorld4.cpp`

    - **Hint:** If everything is going wrong, check the output from `debugPrint`. It should be the similar to the contents of the `ghostwood.txt` file, except that the `'O'` nodes turned into `'N'` nodes. If it isn't, something is wrong with your `worldLoadNodes` function. (Or you could have a mistake in `debugPrint`.)

16. Recompile your game and re-run the test cases used in Part C. They should still work.

    - **Hint:** `g++ Location.cpp Node.cp World.cpp Item.cpp ItemManager.cpp Game.cpp main.cpp`

    - **Note:** If you get a message saying `Error while reading ghostwood_text.txt` file, then you should copy/paste a fresh copy of the ghostwood_text.txt file from the Assignment website. Somehow you have the wrong file type (e.g., you have a Windows version but you are running replit.)

Formatting [−10% if not done]

1. Neatly indent your program using a consistent indentation scheme.

2. Put spaces around your arithmetic operators:
   `x = x + 3;`

3. Use symbolic constants, such as `MAX_DESCRIPTION_COUNT`, when appropriate.

4. Include a comment at the top of `main.cpp` that states your name and student number.

5. Format your program so that it is easily readable. Things that make a program hard to read include:

    - **Very many blank lines**. If more than half your lines are blank, you probably have too many. The correct use of blank lies is to separate logically distinct sections of your program.
    - **Multiple commands on the same line.** In general, don't do this. You can do it if it makes the program clearer than if the same commands were on separate lines.
    - **Uninformative variable names.** For a local variable that is only used for a few lines, it doesn't really matter. But a variable that is used over a larger area (including all global and member variables) should have a name that documents its purpose. Similarly, parameters should have self-documenting names because the function will be called from elsewhere in the program.
    - **No parameter names in function prototypes**. Function parameters should have the same name in the prototype as in the implementation. This makes calling the function much less confusing.

Submission

- Submit a complete copy of your source code. You should have the following files with exactly these names:
    1. **Game.h**
    2. **Game.cpp**
    3. Item.h
    4. Item.cpp
    5. **ItemManager.h**
    6. **ItemManager.cpp**
    7. **Location.h**
    8. **Location.cpp**
    9. **main.cpp**
    10. **Node.h**
    11. **Node.cpp**
    12. **World.h**
    13. **World.cpp**
    - **Note:** A Visual Studio `.sln` file does NOT contain the source code; it is just a text file. You do not need to submit it. Make sure you submit the `.cpp` files and `.h` files.
    - **Note:** You do not need to submit the test programs or data files. The marker has those already.
- If you had problems completing Parts D, E, and F, also submit a separate folder containing your copy of Assignment4A, with the following files:
    1. **Game.h**
    2. **Game.cpp**
    3. Item.h
    4. Item.cpp
    5. **ItemManager.h**
    6. **ItemManager.cpp**
    7. Location.h
    8. Location.cpp
    9. **main.cpp**
    10. World.h
    11. World.cpp
- If possible, convert all your files to a single archive (`.zip` file) before handing them in
- Do NOT submit a compiled version
- Do NOT submit intermediate files, such as:
    - `*.o` files
    - `Debug` folder
    - `Release` folder
    - `ipch` folder
    - `*.ncb`, `*.sdf`, or `*.db` files
- Do NOT submit a screenshot