Spike: Task 11.P
Title: Game Graphs from Data

Author: Mitchell Wright, 100595153

Goals / deliverables:

To demonstrate an understanding of collection types, their strengths and weaknesses, and apply that knowledge in creating a functional inventory system.

Items created during task:

- Code, see: \11 - Spike – Game Graphs from Data\game_graphs

Technologies, Tools, and Resources used:

- Visual Studio 2022
- SourceTree
- GitHub
- Lecture 3.2 – Data Structures

Tasks undertaken:

- Design the File Format
- Implementing the Location Class
- Implementing Interfaces
- Testing Locations
- Commit to Git

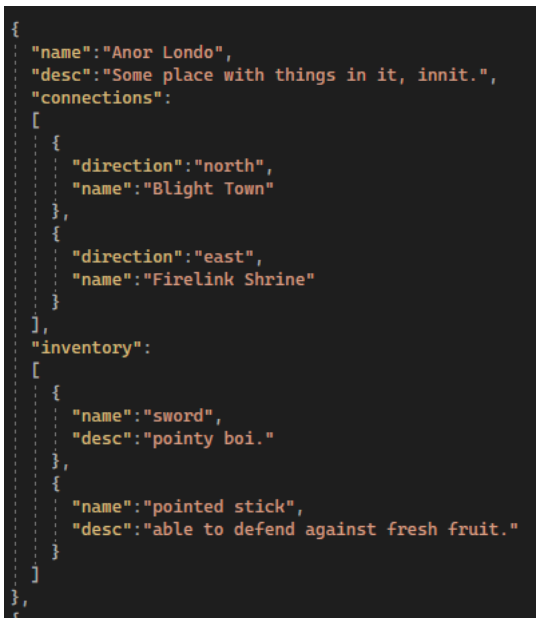
What we found out:

1. Designing the File Format:

I decided to go with JSON for the file format containing the location data. This is mainly due to how simple it was when used in the previous task. The files are broken up as below.

Please note that this is easily able to be added to, with the file currently only containing location data, and not yet containing any data about the specific adventure these locations are part of.

In this current iteration, each location contains a name and description. The connections are pairs of direction and location name. Finally, the inventory is made up of a list of items using name and description pairs (for now).



This is what the file ended up looking like.

2. Implementing the Location Class:

The Location class is primarily made up of a constructor and an interface with the inventory. I was briefly considering storing the connection data within a specific manager, though this idea was dropped as it seemed a bit out of scope for this specific task.

```
class Location
{
private:
    string _name;
    string _desc;

    map<string, string> _connections;//May move to /gamelocation manager at some stage.

    Inventory _inventory;
public:
    Location(json);
    ~Location();

    //Descriptors
    string getName() const;
    string getDesc() const;

    //Connections
    bool findConnection(const string&);
    void showConnections();

    //Inventory interface
    bool findItem(const string&);
    void viewItems();
    bool addItem(Item*);
    bool removeItem(const string&);
    Item* getItem(const string&);
};
```

The constructor is the most interesting part of this class, as much of the rest of it are getters and setters, with some interfaces to the Inventory.

```
Location::Location(json data)
{
    //name
    _name = data["name"];

    //desc
    _desc = data["desc"];

    //connections
    for (json connection : data["connections"])
    {
        _connections.emplace(connection["direction"], connection["name"]);
    }

    _inventory = Inventory();

    //inventory
    for (json item : data["inventory"])
    {
        _inventory.add(new Item(item["name"], item["desc"]));
    }
}
```

While the name and description assignment are pretty simple, the item and connection lists both involve assignment via a loop. These are done by emplacing

the connection pair within a map, using the direction being the key. This however, will mean that “east” can only lead to a single place (Not important right now, but does remove the possibility of some fun stuff later on).

```
void Location::showConnections()
{
    for (auto const& con : _connections)
    {
        cout << con.first << " ";
    }
    cout << endl;
}
```

It should be noted that for now, the interfaces for connections and inventory, the functions directly output text to console, rather than an Ostream object to be output by a different part of the program. This is up for change depending on later tasks.

3. Testing Locations:

The test harness reused a lot of the previous Zorkish tasks. There’s a manager that updates and renders, and it contains a vector of locations created at runtime.

```
int main(int argc, char** argv)
{
    if (argc < 2)
        return 1;

    vector<Location*> locations;

    ifstream input_file(argv[1]);
    json data = json::parse(input_file);
    input_file.close();

    for (json loc : data["locations"])
    {
        locations.push_back(new Location(loc));
    }

    Manager manager = Manager(locations);

    while (manager.running())
    {
        manager.render();
        manager.update();
    }

    return 0;
}
```

The manager is created as above, with render and update being called in a loop in main.

```
void Manager::update()
{
    vector<string> input = processInput();

    if (!input.empty() && input.front() == "quit")
    {
        _running = false;
    }
    else if (input.size() > 1 && input.front() == "go")
    {
        move(_currentLocation->findConnection(input.at(1)));
    }
    else
        cout << "Invalid input." << endl;
}
```

A veeeeeeeeeeeeeery basic command system has been implemented in order to check movement and location linking. Please note the presence of a player class, even though it basically does nothing as of yet.

4. Commit to Git: