Spike: 5
Title: Game Graphs from Data

Author: Ryan Chessum, 102564760

Goals / deliverables:

- Code, see /11 – Spike - Game Graphs from Data/Task 11/
- Spike Report

Technologies, Tools, and Resources used:

List of information needed by someone trying to reproduce this work

- Visual Studio 2019
- C plus plus reference (https://www.cplusplus.com/reference/)
- Zorkish game specifications
- Class and header files from previous spike

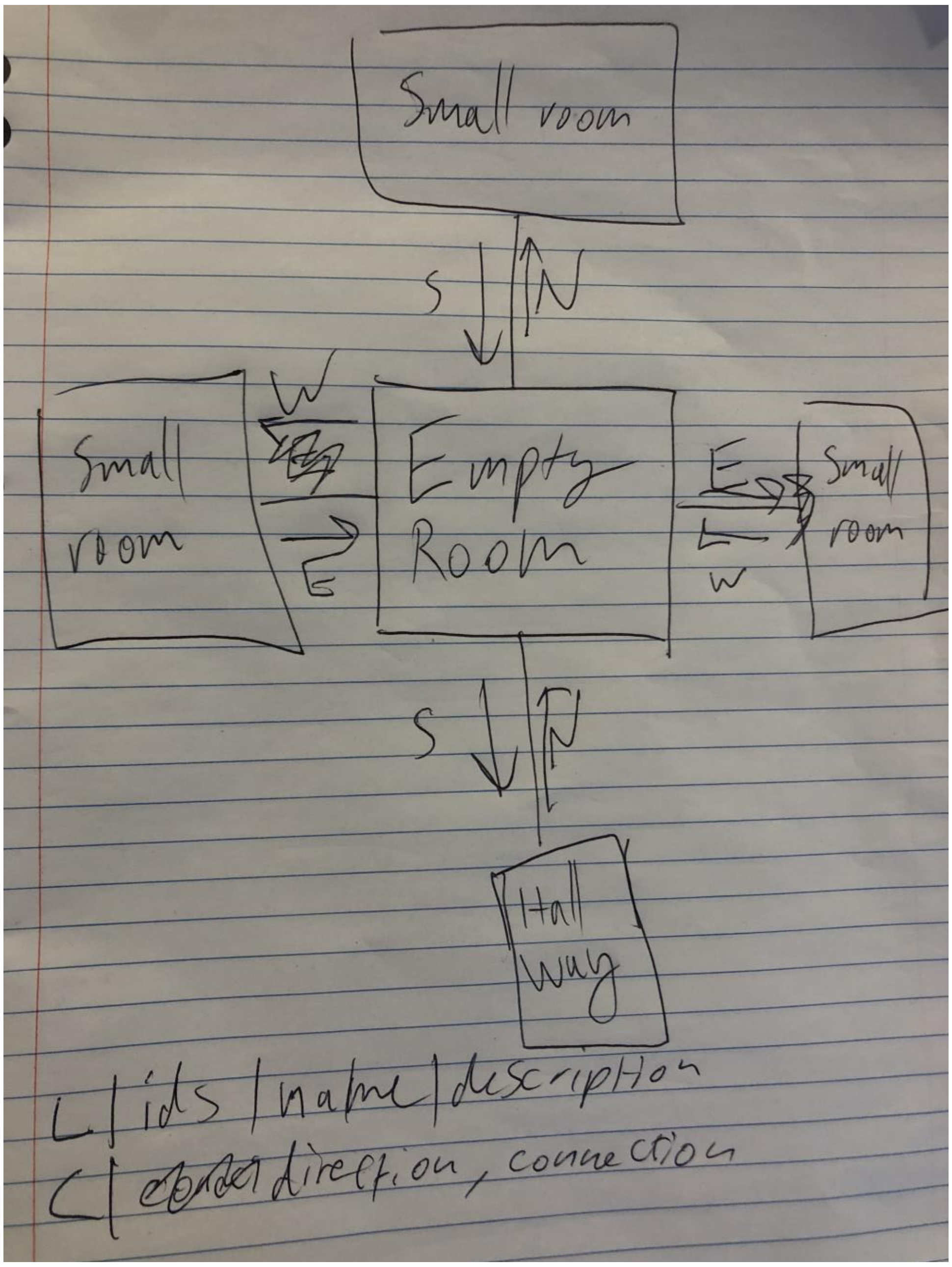
Tasks undertaken:

- Download and install Visual Studio
- Create a new C++ project
- Import Identifiable object, game object and player class code
- Create location class
- Create functions to read in world graph data from a file
- Create a test world file and add locations to it

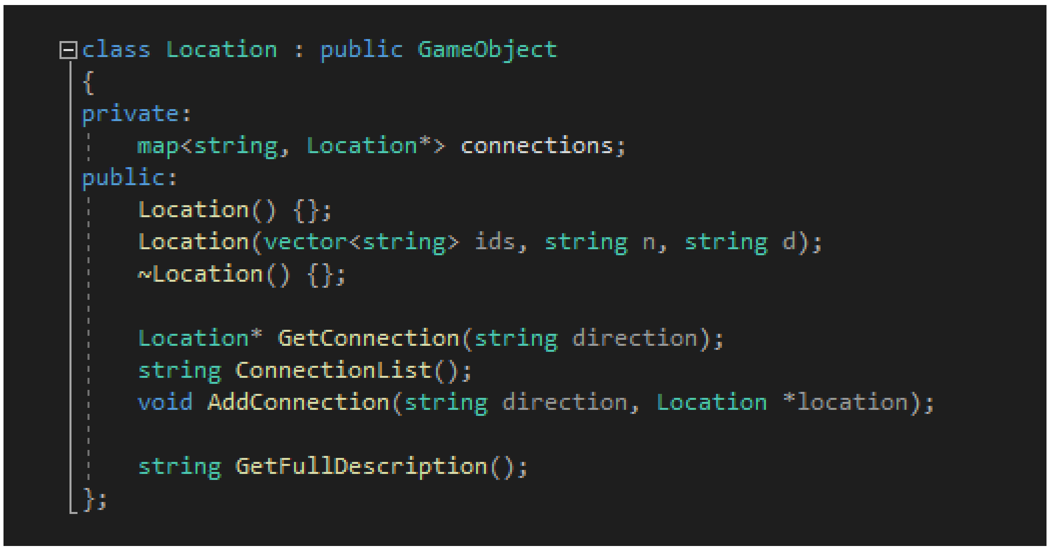
What we found out:

A graph is a group of nodes or points connected by vertices. If we look at the Zorkish specifications, there can be any number of locations connected in any direction. At first this may seem difficult to implement but if we approach it as a graph it’s not too bad.

We can implement the locations by having each location in the graph be a node. Then each location node can contain a list of every connection which would be the vertex connecting each location in the graph together.
Here’s how I implemented the location class. Like the items from the previous spike, it inherits from the gameObject class. This lets us get name/description functions and ids.


Small room
S
N
W
Small room
E
Empty Room
E
W
Small room
S
N
Hall way
L | ids | name | description
C | ~~order~~ direction, connection

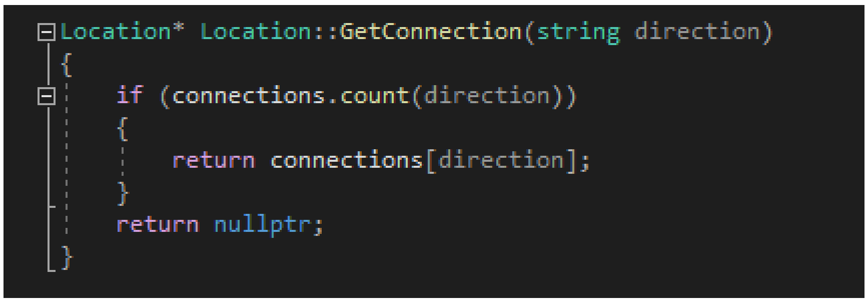
To implement the location graph, each location is going to use a map to store its connections. This means we can pair each connected location with a direction. Up, down, left, right, north, south, east, west, north-east, a magical portal. Whatever we set the key string as will be the direction the player can move to travel to the paired location.



```cpp
class Location : public GameObject
{
private:
    map<string, Location*> connections;
public:
    Location() {};
    Location(vector<string> ids, string n, string d);
    ~Location() {};

    Location* GetConnection(string direction);
    string ConnectionList();
    void AddConnection(string direction, Location *location);

    string GetFullDescription();
};
```

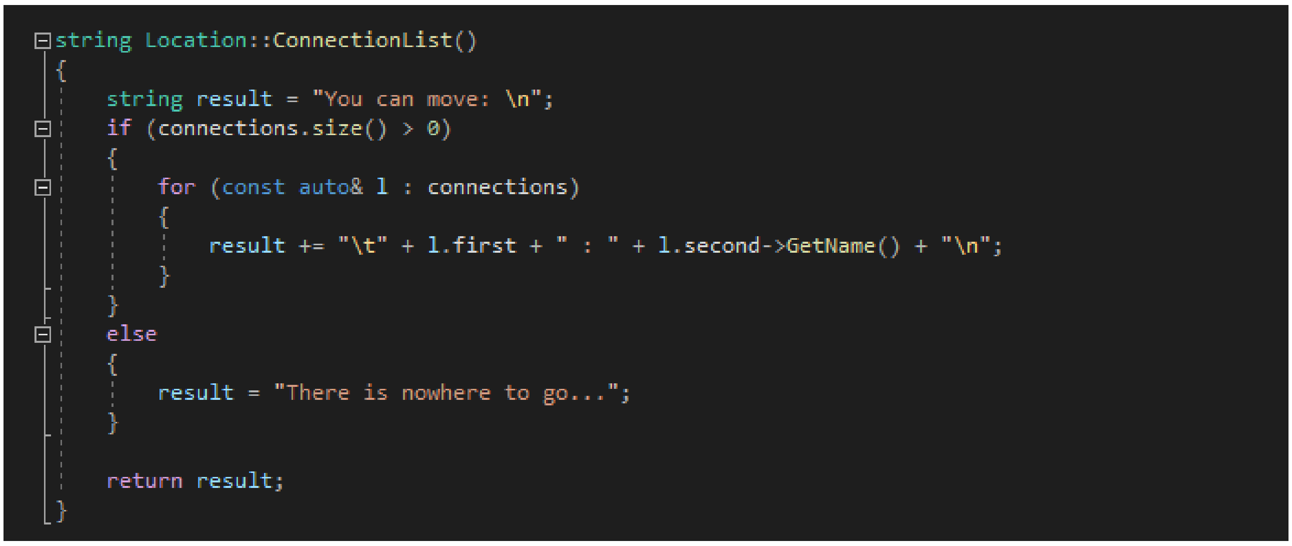
Then there are functions to help us traverse and manage the graph.

GetConnection retrieves a location based on the string passed in by using the string parameter as a key.



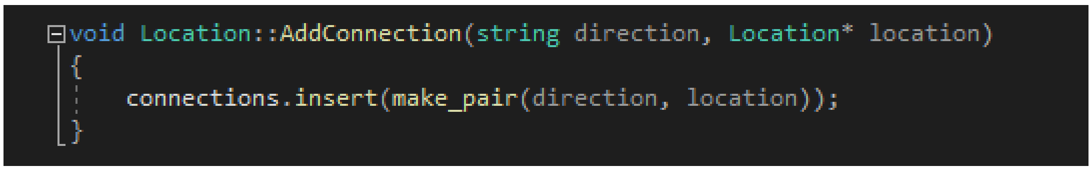
```cpp
Location* Location::GetConnection(string direction)
{
    if (connections.count(direction))
    {
        return connections[direction];
    }
    return nullptr;
}
```

ConnectionList simply returns a string listing each connection that the location has along with its direction.



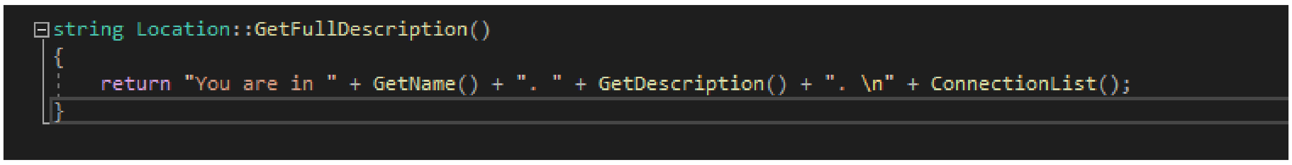
```cpp
string Location::ConnectionList()
{
    string result = "You can move: \n";
    if (connections.size() > 0)
    {
        for (const auto& l : connections)
        {
            result += "\t" + l.first + " : " + l.second->GetName() + "\n";
        }
    }
    else
    {
        result = "There is nowhere to go...";
    }

    return result;
}
```

Finally AddConnection takes in a location pointer and a direction as parameters and adds them as a pair to the connection map.



```
void Location::AddConnection(string direction, Location* location)
{
    connections.insert(make_pair(direction, location));
}
```

I also overrode the full description to have the list of connections with the location name and description.

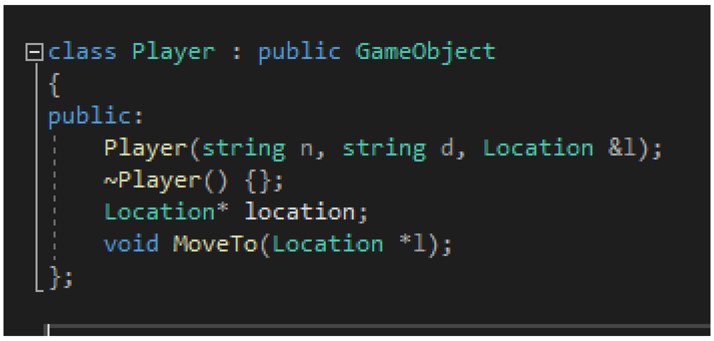


```
string Location::GetFullDescription()
{
    return "You are in " + GetName() + ". " + GetDescription() + ". \n" + ConnectionList();
}
```

Next, we need a way to traverse the graph. For this we are going to use the player. So, should each location keep track of the players position, or should the player keep track of which location it is currently in?

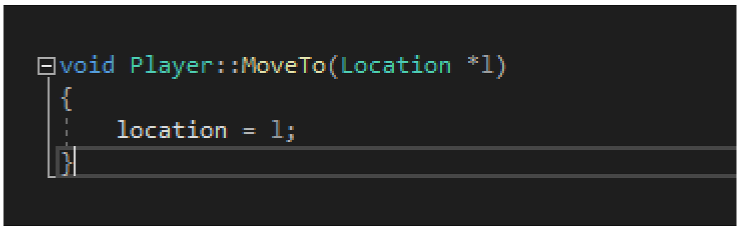
You could really do either if you tried, but having the player keep track of its location is the better option in this case. There is one player, many locations and the player can only be in 1 location at a time. We can simply add a variable that points to a location to the player and that can give us our current location. Then if we want to move the player, we just change the pointer to look at the next location. This is much easier to implement than having to test each location to make sure the player is in one place at a time, removing the reference to the player then adding it to another location,etc.

So, we can add a pointer variable to the player which points to the current location.



```
class Player : public GameObject
{
public:
    Player(string n, string d, Location &l);
    ~Player() {};
    Location* location;
    void MoveTo(Location *l);
};
```

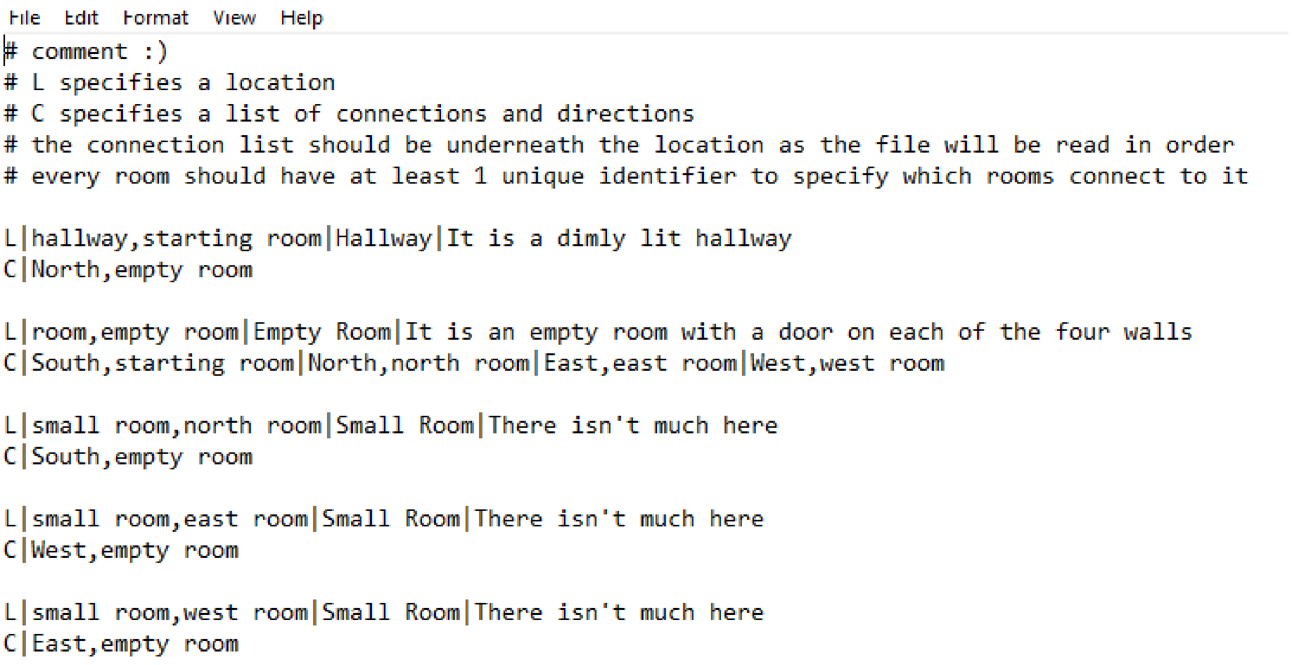
The we also have a function which moves the player to the location at a pointer passed in.



```
void Player::MoveTo(Location *l)
{
    location = l;
}
```

Next, I worked on making a playable game world using these classes. To create the location graph, the program reads location and connection data from a graph and adds the locations to a vector. The vector isn't used to access any locations after all the data has been loaded, it's just to store the locations and is used in initialisation.

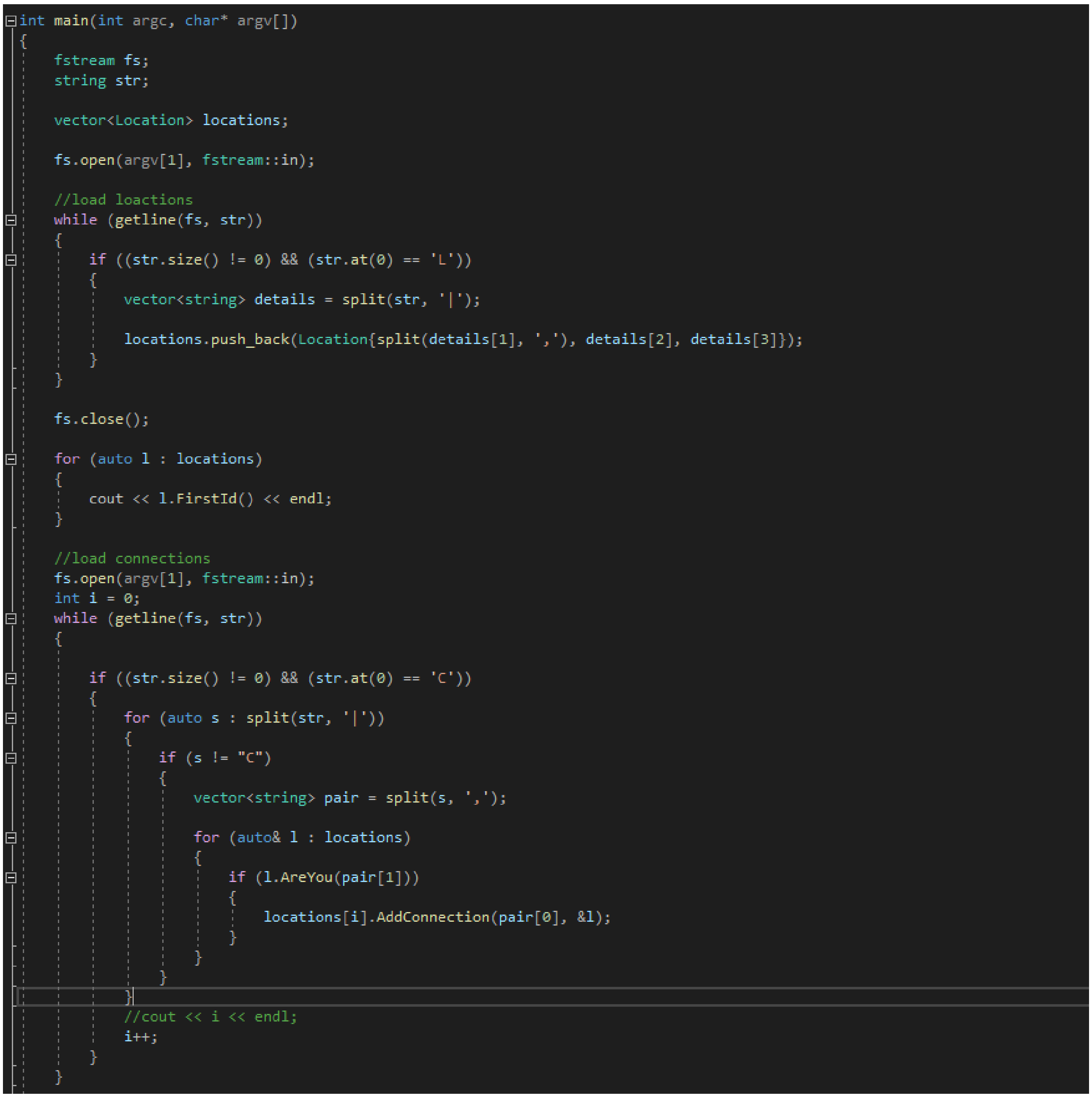
This is the format I used to load in the locations.



```
# comment :)
# L specifies a location
# C specifies a list of connections and directions
# the connection list should be underneath the location as the file will be read in order
# every room should have at least 1 unique identifier to specify which rooms connect to it

L|hallway,starting room|Hallway|It is a dimly lit hallway
C|North,empty room

L|room,empty room|Empty Room|It is an empty room with a door on each of the four walls
C|South,starting room|North,north room|East,east room|West,west room

L|small room,north room|Small Room|There isn't much here
C|South,empty room

L|small room,east room|Small Room|There isn't much here
C|West,empty room

L|small room,west room|Small Room|There isn't much here
C|East,empty room
```

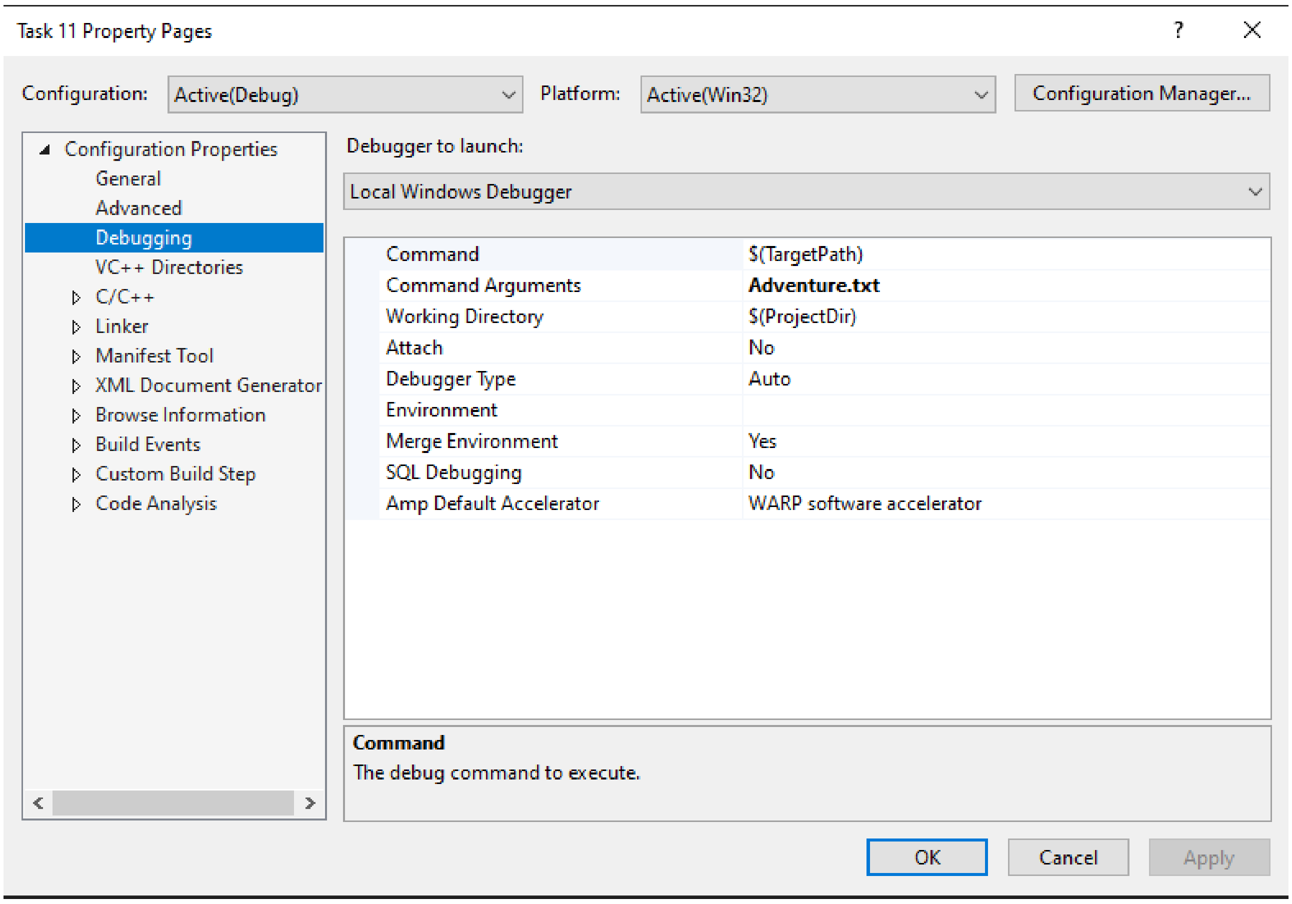
L signifies a location and C signifies the connections to other location. The file will be read twice. First it will go through and find all the locations. Then it will go through again and add all the connections to each location. The connections need to be underneath the location they are for. The reason for this is because all the locations need to be added first before we can add connections between them. So if the connections are underneath they should be read in at the same position in the vector. Every location Should have at least one connection, if not those locations can just go at the very end of the file.

Each line is also split up into each part that needs to be used by where there is a '|'. The locations have their id's separated by a comma, then their name and lastly their description. The connections have each connection split up by ‘|’ and the direction location pair are split up by commas. Each room must have at least one unique id so that it can be used when specifying connections.



```
int main(int argc, char* argv[])
{
    fstream fs;
    string str;

    vector<Location> locations;

    fs.open(argv[1], fstream::in);

    //load loactions
    while (getline(fs, str))
    {
        if ((str.size() != 0) && (str.at(0) == 'L'))
        {
            vector<string> details = split(str, '|');

            locations.push_back(Location{split(details[1], ','), details[2], details[3]});
        }
    }

    fs.close();

    for (auto l : locations)
    {
        cout << l.FirstId() << endl;
    }

    //load connections
    fs.open(argv[1], fstream::in);
    int i = 0;
    while (getline(fs, str))
    {

        if ((str.size() != 0) && (str.at(0) == 'C'))
        {
            for (auto s : split(str, '|'))
            {
                if (s != "C")
                {
                    vector<string> pair = split(s, ',');

                    for (auto& l : locations)
                    {
                        if (l.AreYou(pair[1]))
                        {
                            locations[i].AddConnection(pair[0], &l);
                        }
                    }
                }
            }
            //cout << i << endl;
            i++;
        }
    }
```

When opening the file I am using argv[1]. This is because I am passing in the filename as a command line argument. To do this in visual studio go to properties -> Configuration properties -> Debugging and put any values you want as arguments. Argv[0] will be the filename of the program and the rest will be the values you set here.



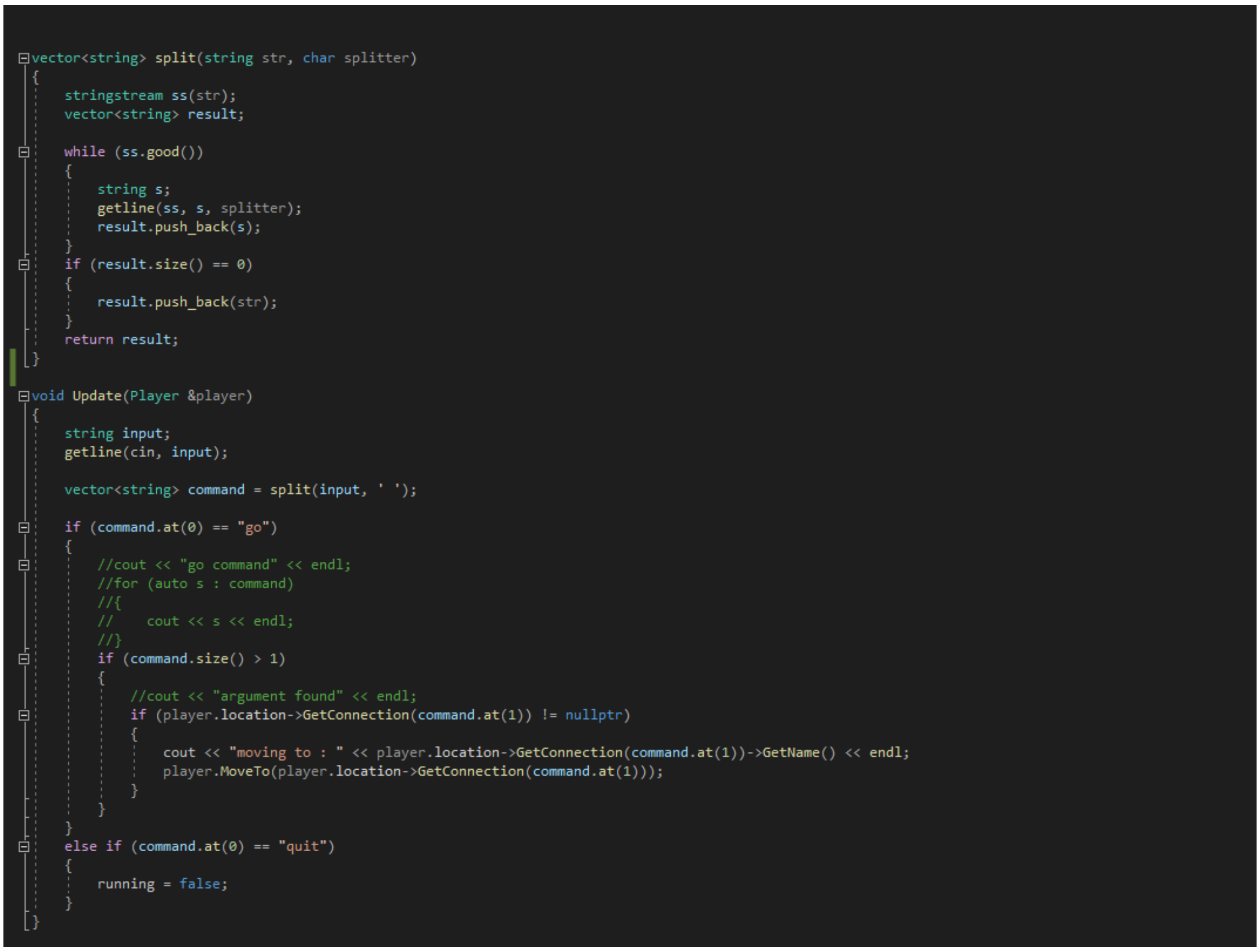
Debugger to launch:

Local Windows Debugger

Command	$(TargetPath)
Command Arguments	**Adventure.txt**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command
The debug command to execute.

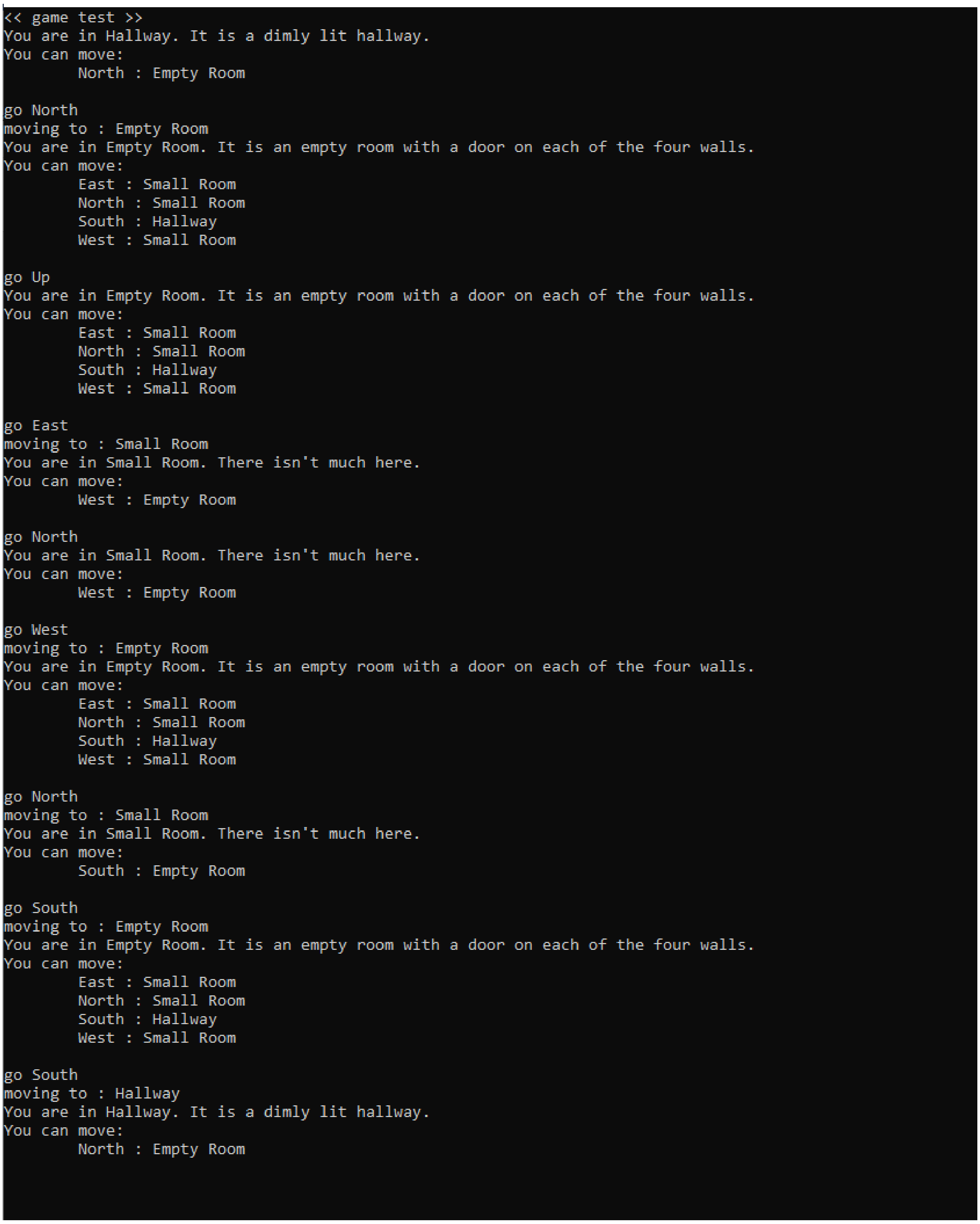
Now we need to add commands to let the player go where they want to go in the graph.



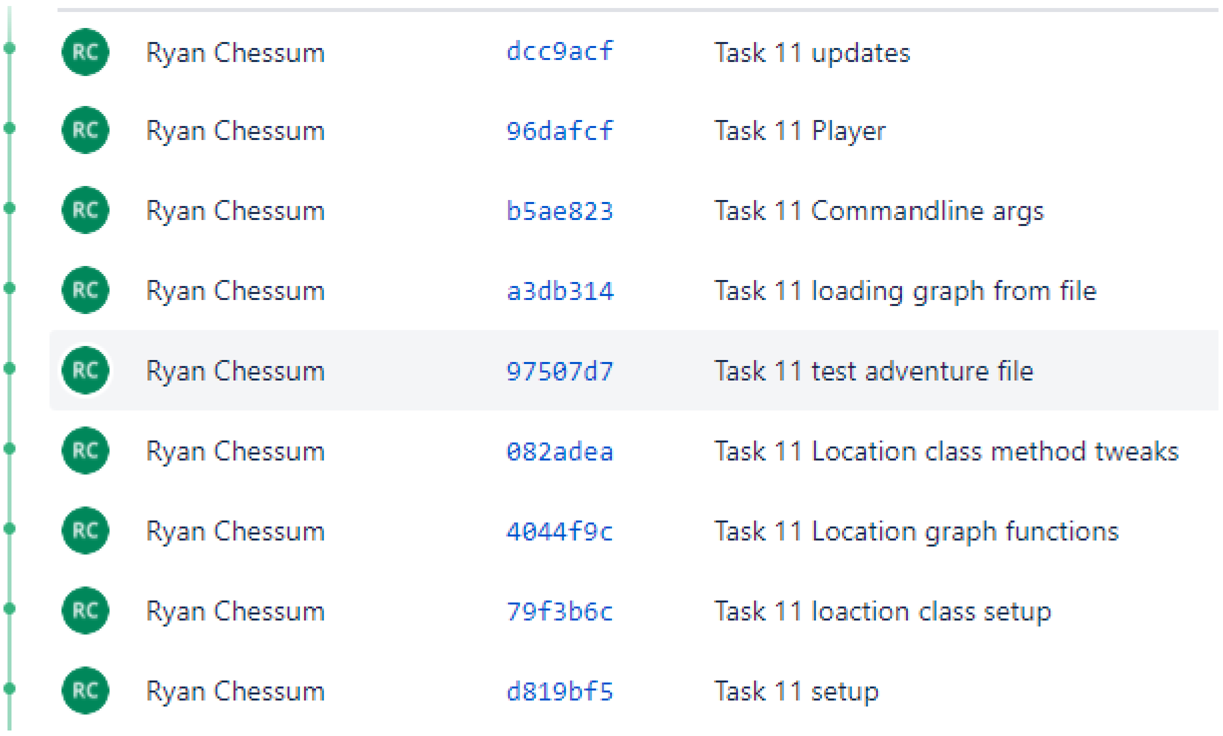
```cpp
vector<string> split(string str, char splitter)
{
    stringstream ss(str);
    vector<string> result;

    while (ss.good())
    {
        string s;
        getline(ss, s, splitter);
        result.push_back(s);
    }
    if (result.size() == 0)
    {
        result.push_back(str);
    }
    return result;
}

void Update(Player &player)
{
    string input;
    getline(cin, input);

    vector<string> command = split(input, ' ');

    if (command.at(0) == "go")
    {
        //cout << "go command" << endl;
        //for (auto s : command)
        //{
        //    cout << s << endl;
        //}
        if (command.size() > 1)
        {
            //cout << "argument found" << endl;
            if (player.location->GetConnection(command.at(1)) != nullptr)
            {
                cout << "moving to : " << player.location->GetConnection(command.at(1))->GetName() << endl;
                player.MoveTo(player.location->GetConnection(command.at(1)));
            }
        }
    }
    else if (command.at(0) == "quit")
    {
        running = false;
    }
}
```

We can make a simple command processor in the update function. We can get the players input and split it into each word. If the first word is go then the program tries to get a connected location to the players current connection with the second word. If there is something there, then the players location is updated to the new one.

After implementing everything the program should run like this.



```
<< game test >>
You are in Hallway. It is a dimly lit hallway.
You can move:
        North : Empty Room

go North
moving to : Empty Room
You are in Empty Room. It is an empty room with a door on each of the four walls.
You can move:
        East : Small Room
        North : Small Room
        South : Hallway
        West : Small Room

go Up
You are in Empty Room. It is an empty room with a door on each of the four walls.
You can move:
        East : Small Room
        North : Small Room
        South : Hallway
        West : Small Room

go East
moving to : Small Room
You are in Small Room. There isn't much here.
You can move:
        West : Empty Room

go North
You are in Small Room. There isn't much here.
You can move:
        West : Empty Room

go West
moving to : Empty Room
You are in Empty Room. It is an empty room with a door on each of the four walls.
You can move:
        East : Small Room
        North : Small Room
        South : Hallway
        West : Small Room

go North
moving to : Small Room
You are in Small Room. There isn't much here.
You can move:
        South : Empty Room

go South
moving to : Empty Room
You are in Empty Room. It is an empty room with a door on each of the four walls.
You can move:
        East : Small Room
        North : Small Room
        South : Hallway
        West : Small Room

go South
moving to : Hallway
You are in Hallway. It is a dimly lit hallway.
You can move:
        North : Empty Room
```



RC	Ryan Chessum	dcc9acf	Task 11 updates
RC	Ryan Chessum	96dafcf	Task 11 Player
RC	Ryan Chessum	b5ae823	Task 11 Commandline args
RC	Ryan Chessum	a3db314	Task 11 loading graph from file
RC	Ryan Chessum	97507d7	Task 11 test adventure file
RC	Ryan Chessum	082adea	Task 11 Location class method tweaks
RC	Ryan Chessum	4044f9c	Task 11 Location graph functions
RC	Ryan Chessum	79f3b6c	Task 11 loaction class setup
RC	Ryan Chessum	d819bf5	Task 11 setup