Spike: 6
Title: Command pattern

Author: Ryan Chessum, 102564760

Goals / deliverables:

- Code, see /12 – Spike – Command pattern/Task 12/
- Spike Report
- UML Diagram

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
- Code from previous spikes

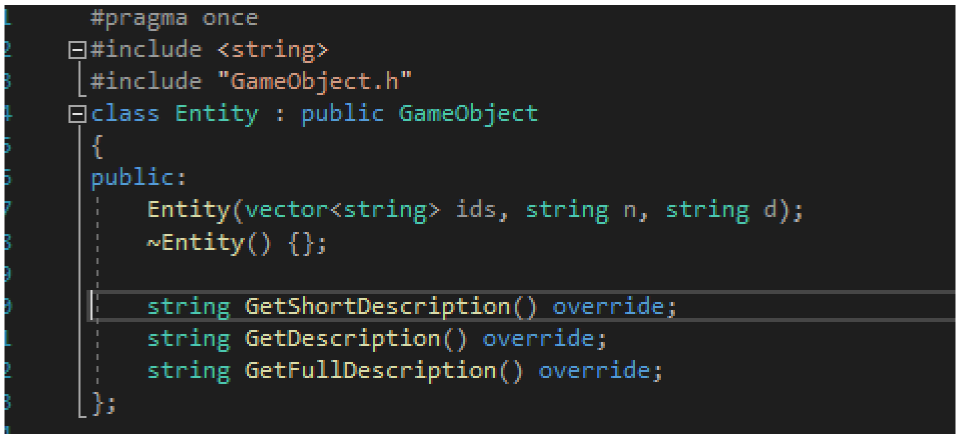
Tasks undertaken:

- Download and install Visual Studio
- Create a new C++ project
- Import code from previous spikes
- Implement game into PlayAdventure state
- Add Entities
- Make entities loaded from file
- Create Command class
- Create Command Processor class
- Create Child command classes
- Implement command processor into game

What we found out:

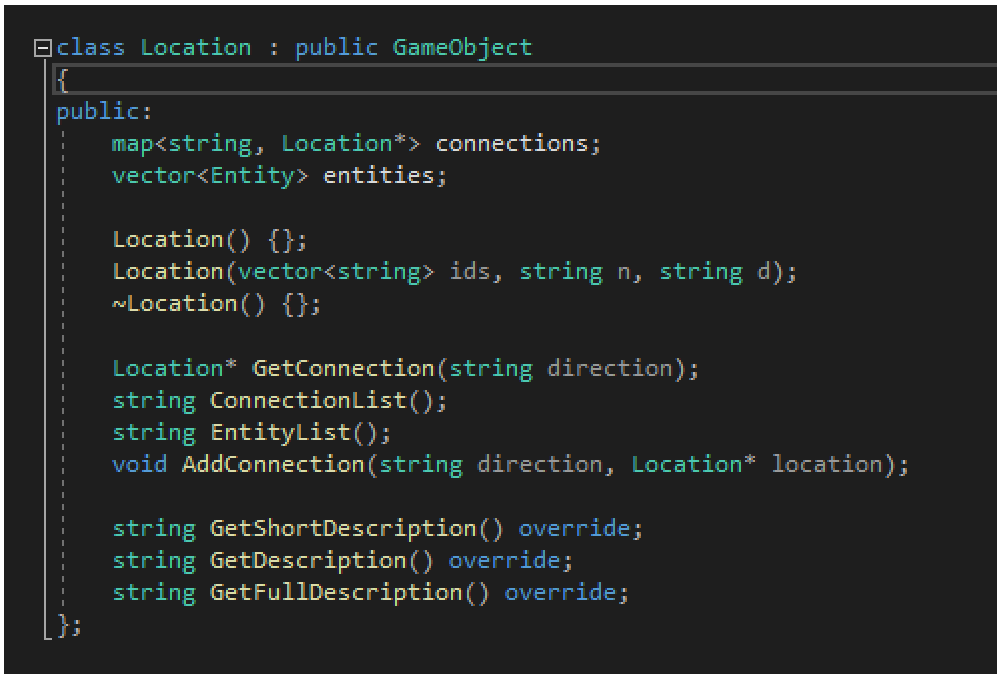
In order to process many different types of commands without making a giant if else block, we can make a command processor class that finds the command we are trying to use and executes the corresponding code.

To show this, I needed to create a proper implementation of Zorkish that had a graph world with entities. The first thing I did was import the files for the classes we needed including the state related classes and classes that inherited from the identifiable object class like locations and the player.



```
#pragma once
#include <string>
#include "GameObject.h"
class Entity : public GameObject
{
public:
    Entity(vector<string> ids, string n, string d);
    ~Entity() {};

    string GetShortDescription() override;
    string GetDescription() override;
    string GetFullDescription() override;
};
```

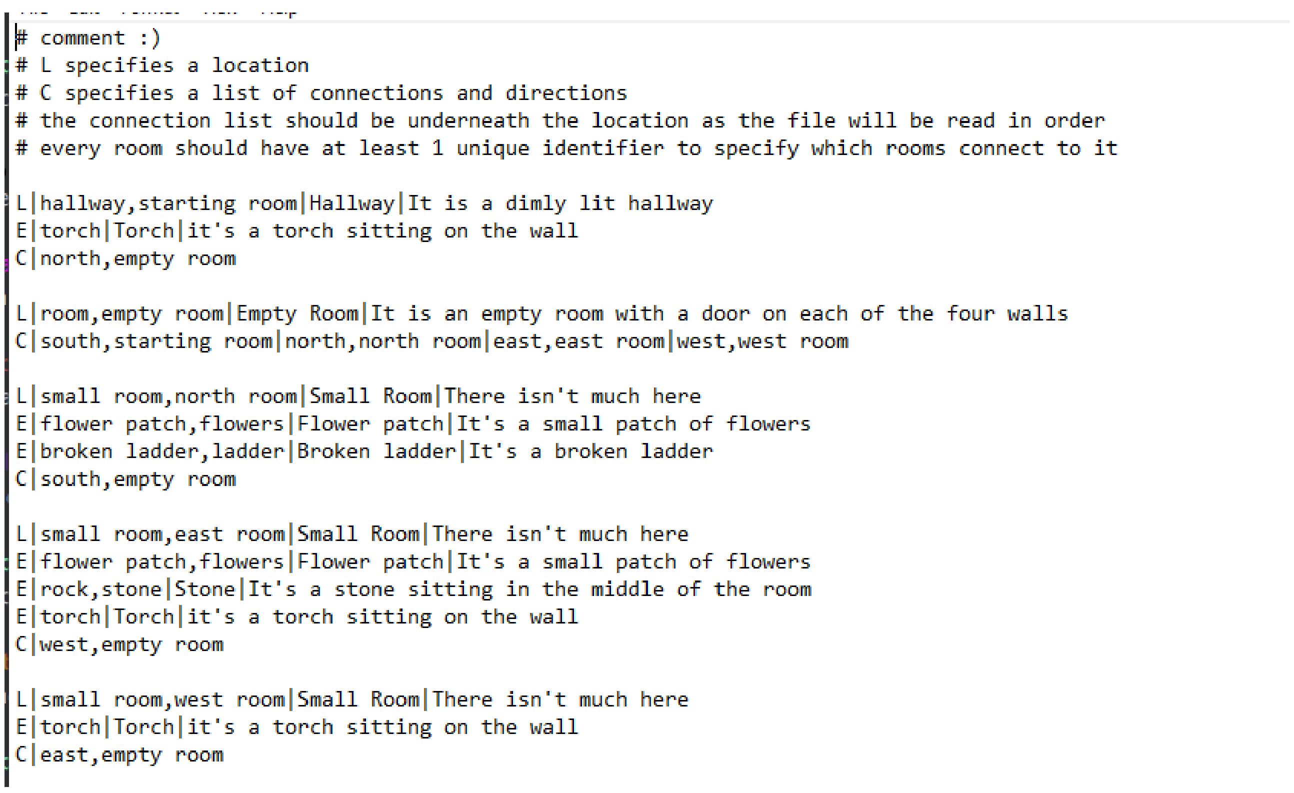
First, I made a basic class to use for entities. It's like the item class. I then added a vector of entities in the location class so that locations can store entities within them and a function to list all the entities held in the vector as a string.



```
class Location : public GameObject
{
public:
    map<string, Location*> connections;
    vector<Entity> entities;

    Location() {};
    Location(vector<string> ids, string n, string d);
    ~Location() {};

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

    string GetShortDescription() override;
    string GetDescription() override;
    string GetFullDescription() override;
};
```

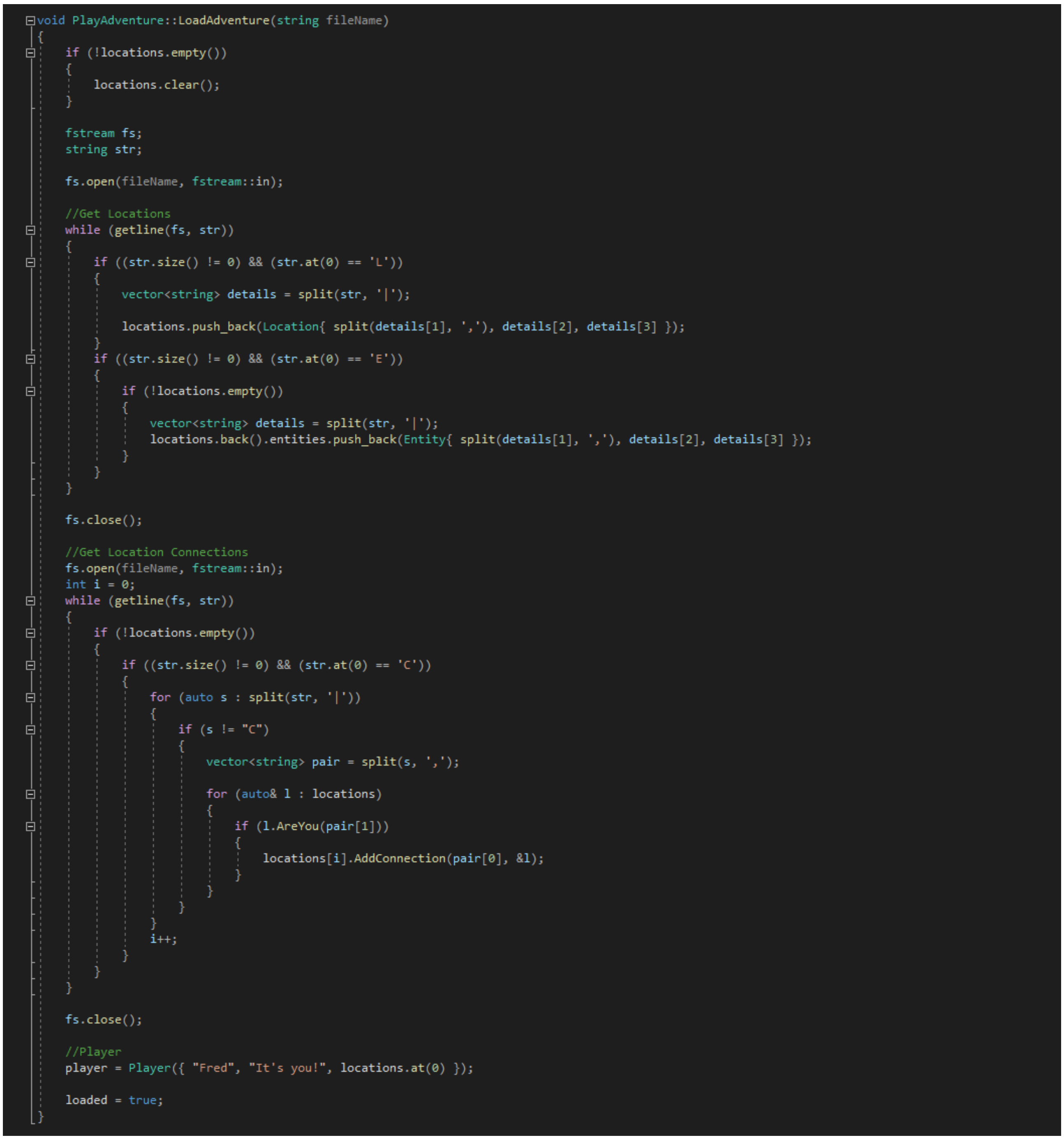
After we had that I copied the adventure loading code from the previous spike into a new function the PlayAdventure state and called it LoadAdventure, it takes the filename as a parameter.

PlayAdventure also stores a Boolean variable to keep track if an adventure is loaded. If one is loaded, then the Zorkish game will run otherwise the program will run the menu to select an adventure. When an adventure is selected it loads an adventure and then sets the Boolean variable to true. It turns back to false when you quit the game.



```
# 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
E|torch|Torch|it's a torch sitting on the wall
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
E|flower patch,flowers|Flower patch|It's a small patch of flowers
E|broken ladder,ladder|Broken ladder|It's a broken ladder
C|south,empty room

L|small room,east room|Small Room|There isn't much here
E|flower patch,flowers|Flower patch|It's a small patch of flowers
E|rock,stone|Stone|It's a stone sitting in the middle of the room
E|torch|Torch|it's a torch sitting on the wall
C|west,empty room

L|small room,west room|Small Room|There isn't much here
E|torch|Torch|it's a torch sitting on the wall
C|east,empty room
```

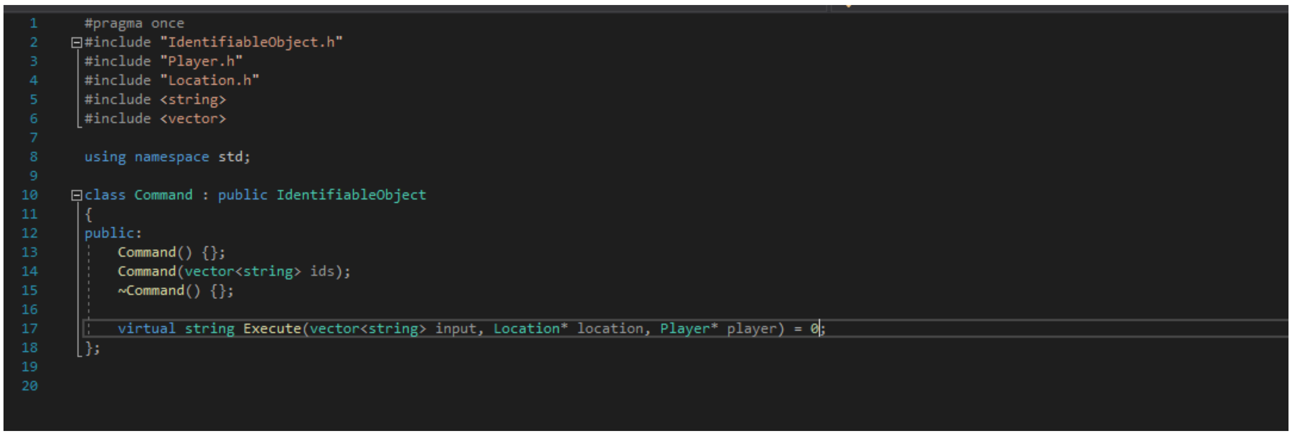
Then I added some entities to the adventure file. I signify them with an E and are loaded in the same way as a location as their constructors take the same parameters. When the load function loads in an entity it will add it to the last location that it loaded. So, like the connections entities you want to have in a location should be placed underneath the desired location.



```cpp
void PlayAdventure::LoadAdventure(string fileName)
{
    if (!locations.empty())
    {
        locations.clear();
    }

    fstream fs;
    string str;

    fs.open(fileName, fstream::in);

    //Get Locations
    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] });
        }
        if ((str.size() != 0) && (str.at(0) == 'E'))
        {
            if (!locations.empty())
            {
                vector<string> details = split(str, '|');
                locations.back().entities.push_back(Entity{ split(details[1], ','), details[2], details[3] });
            }
        }
    }

    fs.close();

    //Get Location Connections
    fs.open(fileName, fstream::in);
    int i = 0;
    while (getline(fs, str))
    {
        if (!locations.empty())
        {
            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);
                            }
                        }
                    }
                }
                i++;
            }
        }
    }

    fs.close();

    //Player
    player = Player({ "Fred", "It's you!", locations.at(0) });

    loaded = true;
}
```

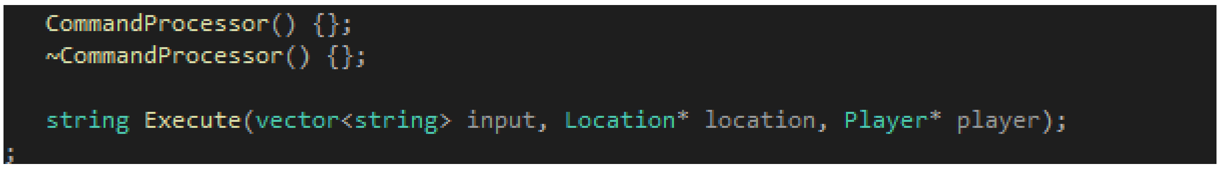
Now we have a basic version of the game where the player can traverse through the locations filled with entities. From here I Implemented the command processor.

First, I made a basic command class which each command will inherit from. It is a child class of identifiable object. This will let us use Ids as the command sting. So, if the command processor finds a matching id at the start of the players input string it can call the execute function of the corresponding command.



```cpp
#pragma once
#include "IdentifiableObject.h"
#include "Player.h"
#include "Location.h"
#include <string>
#include <vector>

using namespace std;

class Command : public IdentifiableObject
{
public:
    Command() {};
    Command(vector<string> ids);
    ~Command() {};

    virtual string Execute(vector<string> input, Location* location, Player* player) = 0;
};
```

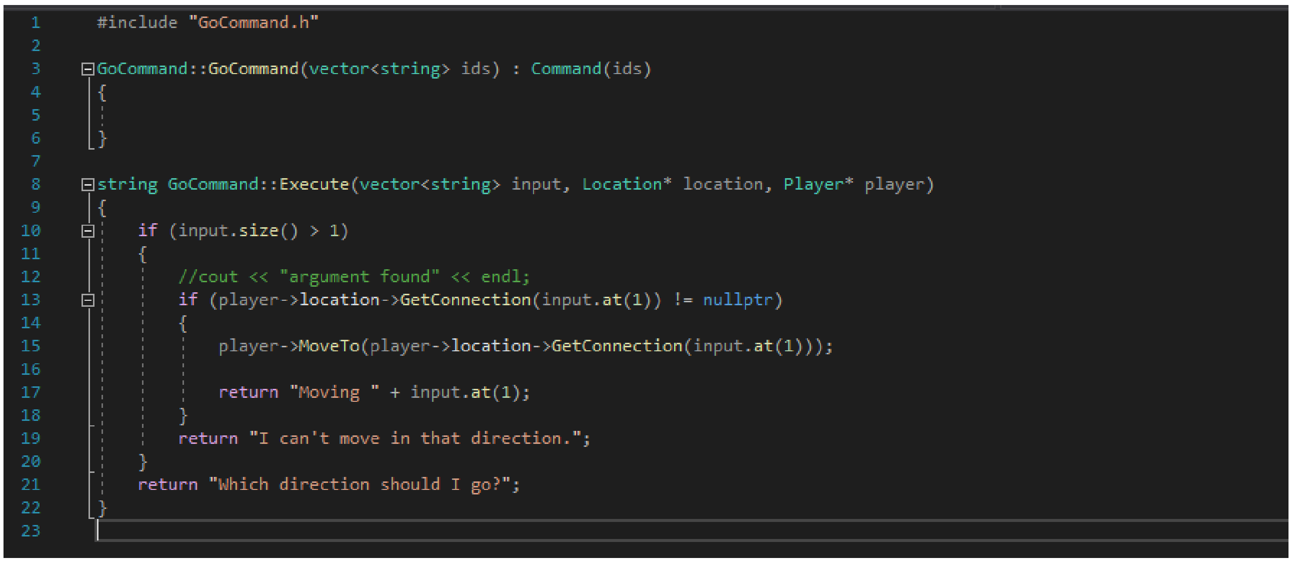
Next, I set up a command processor class. It also has an execute function.



```cpp
    CommandProcessor() {};
    ~CommandProcessor() {};

    string Execute(vector<string> input, Location* location, Player* player);
;
```

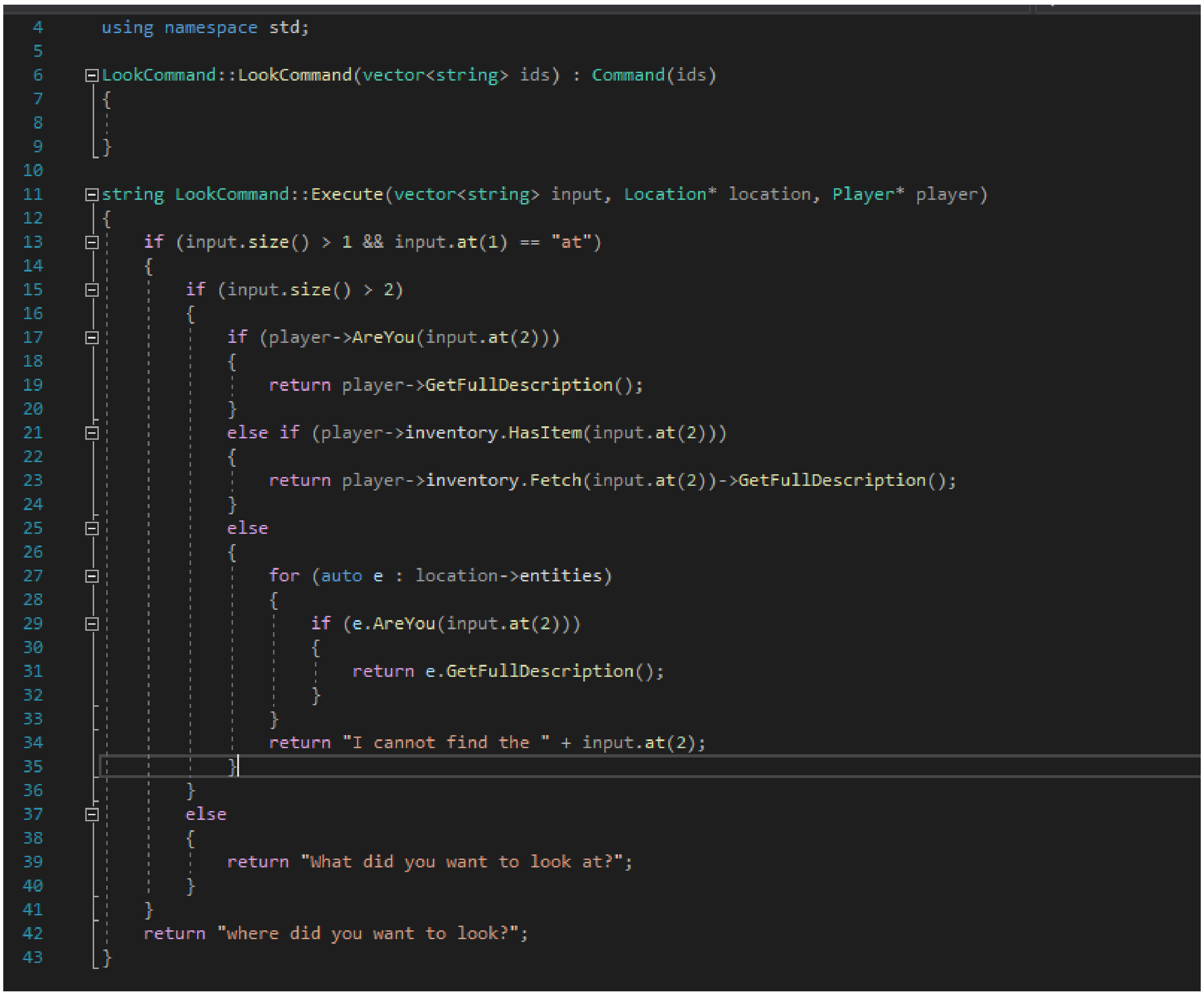
Before I set up the rest I went back and created the command subclasses for all the commands I wanted to add.

First is the go command which works pretty much the same way it did before.



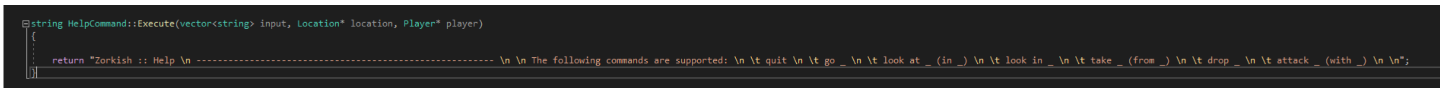
```cpp
#include "GoCommand.h"

GoCommand::GoCommand(vector<string> ids) : Command(ids)
{

}

string GoCommand::Execute(vector<string> input, Location* location, Player* player)
{
    if (input.size() > 1)
    {
        //cout << "argument found" << endl;
        if (player->location->GetConnection(input.at(1)) != nullptr)
        {
            player->MoveTo(player->location->GetConnection(input.at(1)));

            return "Moving " + input.at(1);
        }
        return "I can't move in that direction.";
    }
    return "Which direction should I go?";
}
```

Then I added the look command which checks all of the entities int the location to see if any of their ids match the 3rd input string. It also to see if that string is the player.



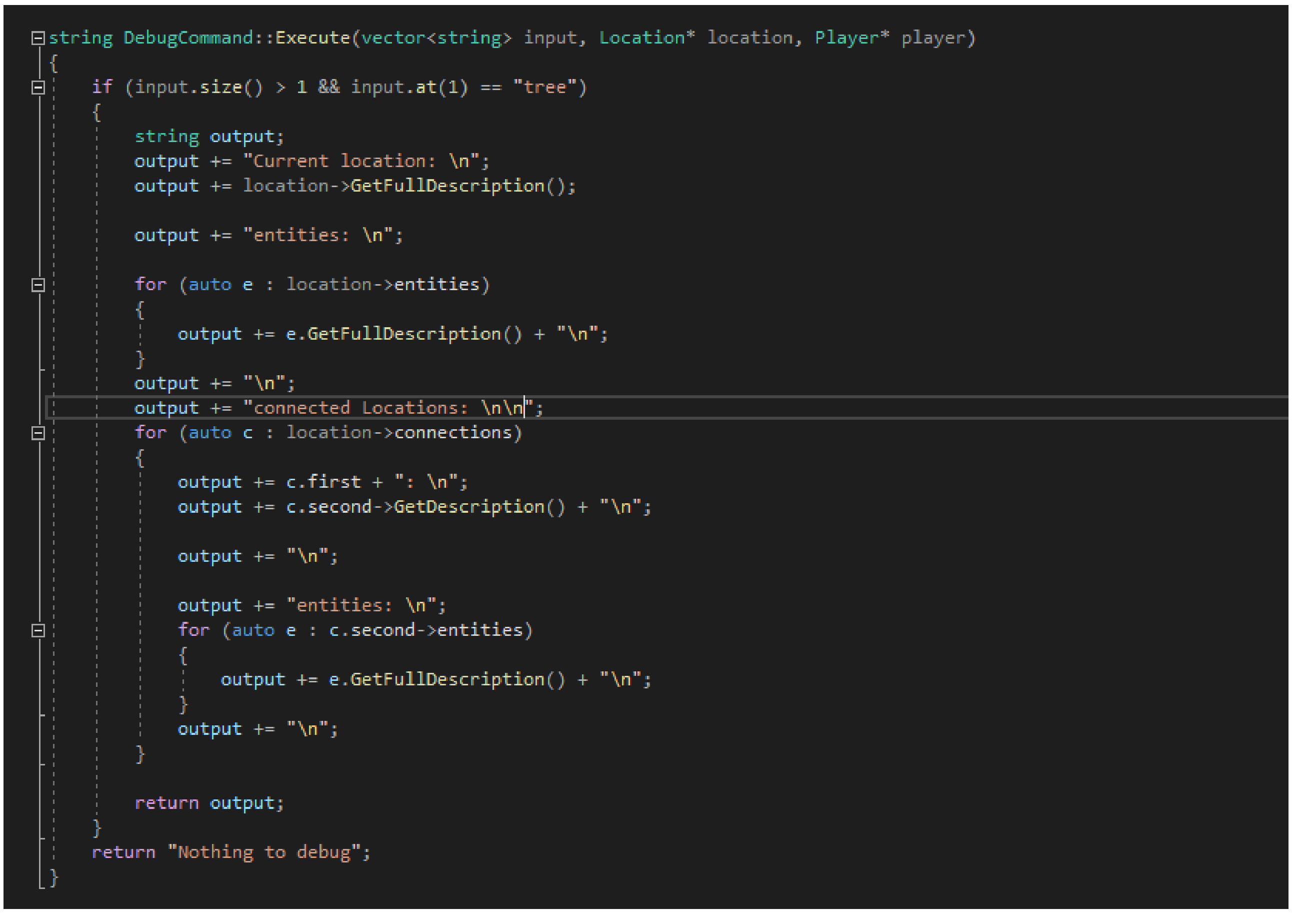
```cpp
using namespace std;

LookCommand::LookCommand(vector<string> ids) : Command(ids)
{

}

string LookCommand::Execute(vector<string> input, Location* location, Player* player)
{
    if (input.size() > 1 && input.at(1) == "at")
    {
        if (input.size() > 2)
        {
            if (player->AreYou(input.at(2)))
            {
                return player->GetFullDescription();
            }
            else if (player->inventory.HasItem(input.at(2)))
            {
                return player->inventory.Fetch(input.at(2))->GetFullDescription();
            }
            else
            {
                for (auto e : location->entities)
                {
                    if (e.AreYou(input.at(2)))
                    {
                        return e.GetFullDescription();
                    }
                }
                return "I cannot find the " + input.at(2);
            }
        }
        else
        {
            return "What did you want to look at?";
        }
    }
    return "where did you want to look?";
}
```

The help command prints the same help message as the one on the main menu.



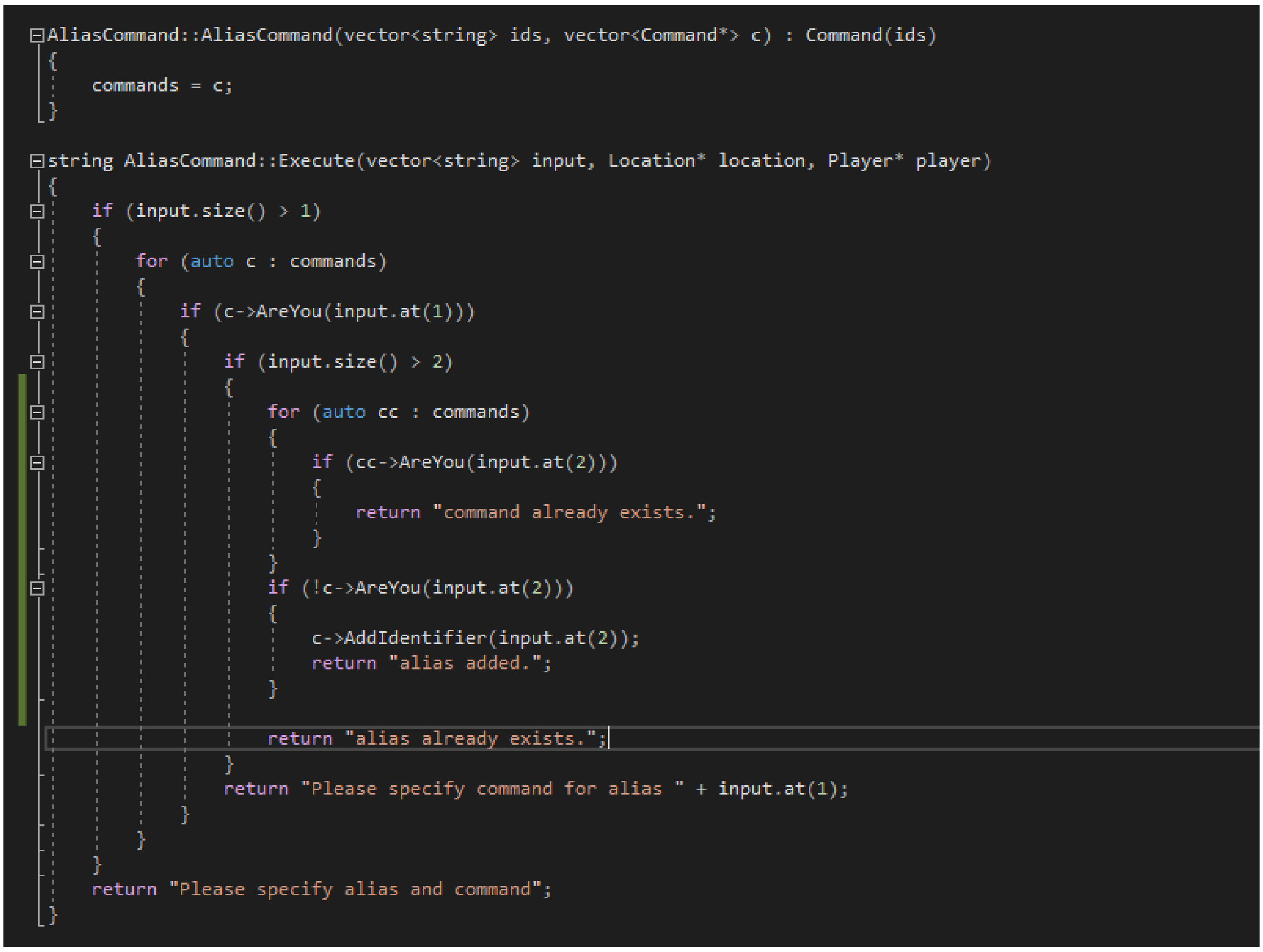
```cpp
string HelpCommand::Execute(vector<string> input, Location* location, Player* player)
{

    return "Zorkish :: Help \n --------------------------------------------------------- \n \n The following commands are supported: \n \t quit \n \t go _ \n \t look at _ (in _) \n \t look in _ \n \t take _ (from _) \n \t drop _ \n \t attack _ (with _) \n \n";
}
```

The debug command is set up for debug related commands. At the moment if the second word is tree it prints off the location details along with the list of entities stored there. As well as each connected location and their entities.



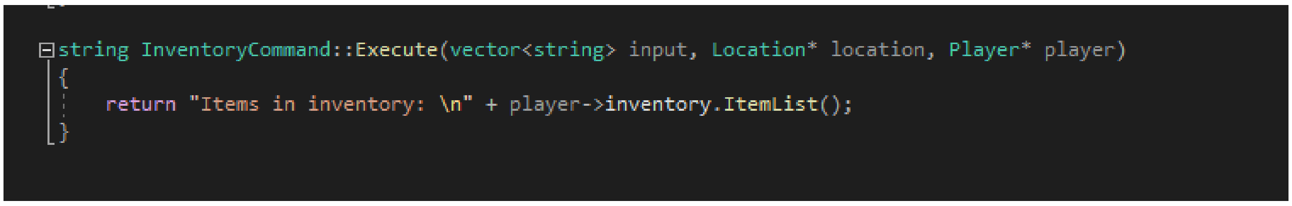
```
string DebugCommand::Execute(vector<string> input, Location* location, Player* player)
{
    if (input.size() > 1 && input.at(1) == "tree")
    {
        string output;
        output += "Current location: \n";
        output += location->GetFullDescription();

        output += "entities: \n";

        for (auto e : location->entities)
        {
            output += e.GetFullDescription() + "\n";
        }
        output += "\n";
        output += "connected Locations: \n\n";
        for (auto c : location->connections)
        {
            output += c.first + ": \n";
            output += c.second->GetDescription() + "\n";

            output += "\n";

            output += "entities: \n";
            for (auto e : c.second->entities)
            {
                output += e.GetFullDescription() + "\n";
            }
            output += "\n";
        }

        return output;
    }
    return "Nothing to debug";
}
```

The alias command adds ids to commands so other strings can be used as command identifiers. To do this it needs to hold a reference to every command in the command processor so it can check the up to date identifiers on all of them. When the command processor is constructed, it will pass a vector of references in so that they can be used in the command. When the command is executed, it will check all the commands to see if it exists then checks the alias you want to add and makes sure it’s not an already existing command. If a command with that string doesn’t exist it will add that identifier to the specified command.



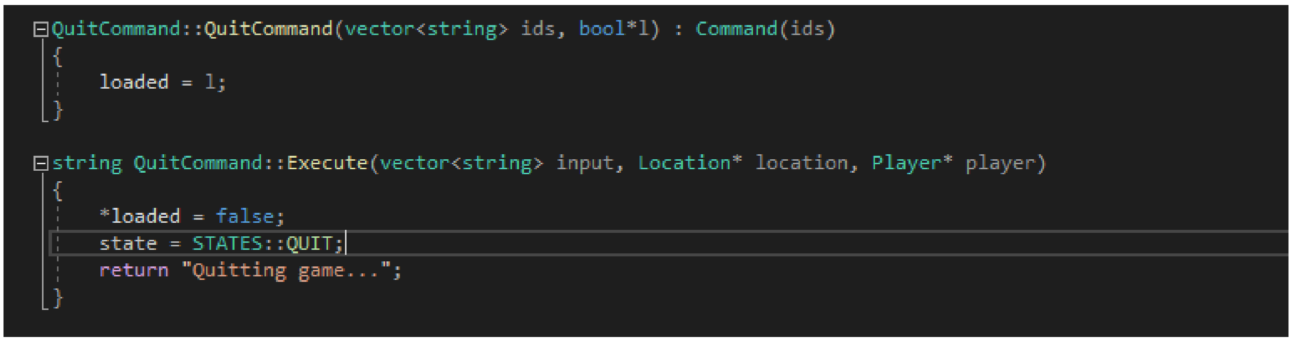
```
AliasCommand::AliasCommand(vector<string> ids, vector<Command*> c) : Command(ids)
{
    commands = c;
}

string AliasCommand::Execute(vector<string> input, Location* location, Player* player)
{
    if (input.size() > 1)
    {
        for (auto c : commands)
        {
            if (c->AreYou(input.at(1)))
            {
                if (input.size() > 2)
                {
                    for (auto cc : commands)
                    {
                        if (cc->AreYou(input.at(2)))
                        {
                            return "command already exists.";
                        }
                    }
                    if (!c->AreYou(input.at(2)))
                    {
                        c->AddIdentifier(input.at(2));
                        return "alias added.";
                    }

                    return "alias already exists.";
                }
                return "Please specify command for alias " + input.at(1);
            }
        }
    }
    return "Please specify alias and command";
}
```

The inventory command simply lists the players inventory.



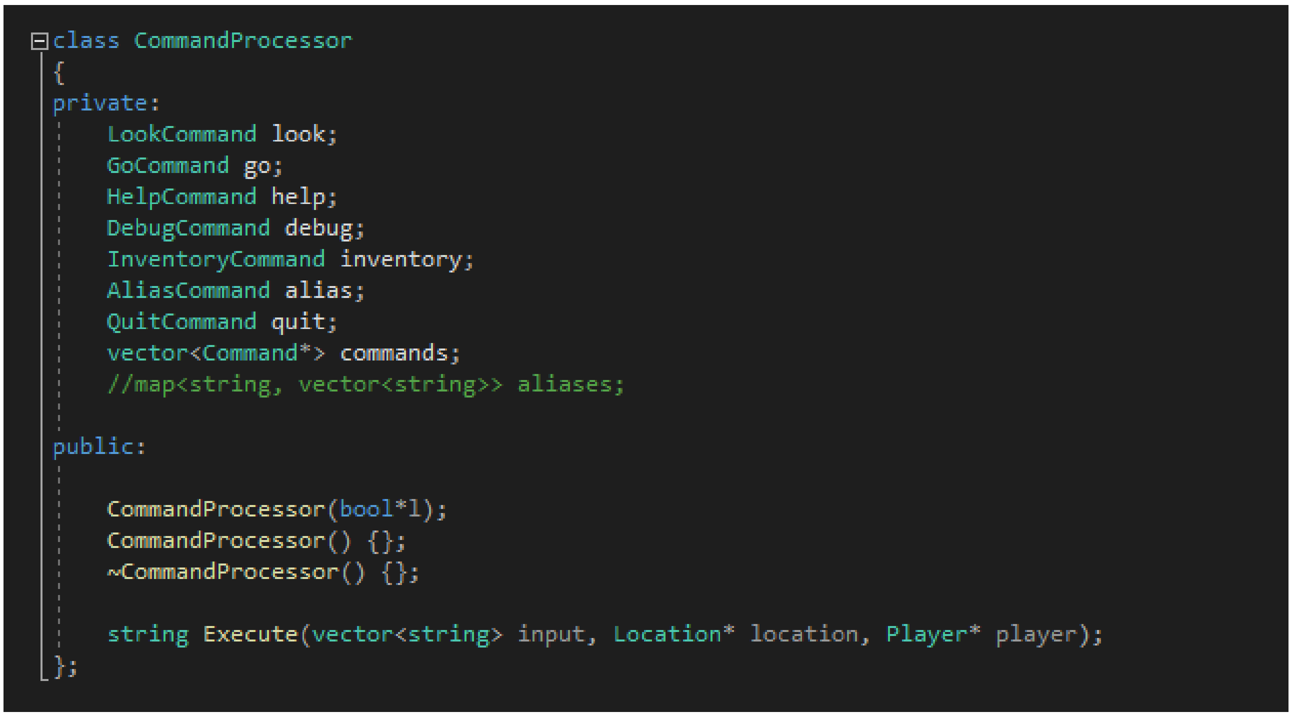
```
string InventoryCommand::Execute(vector<string> input, Location* location, Player* player)
{
    return "Items in inventory: \n" + player->inventory.ItemList();
}
```

Lastly the quit command gets passed in the loaded variable from the constructor. When the player chooses to quit it will change it back to false and set the state to QUIT.



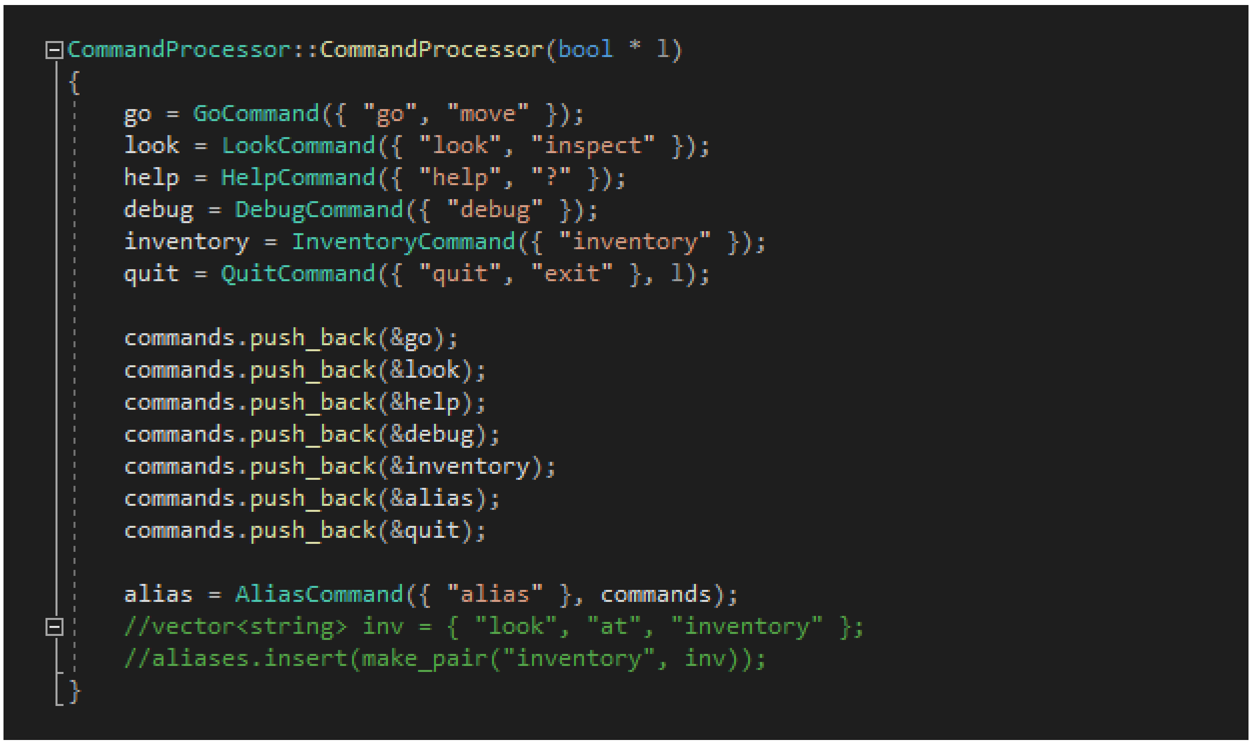
```
QuitCommand::QuitCommand(vector<string> ids, bool*l) : Command(ids)
{
    loaded = l;
}

string QuitCommand::Execute(vector<string> input, Location* location, Player* player)
{
    *loaded = false;
    state = STATES::QUIT;
    return "Quitting game...";
}
```

Now we can go back to the command processor and add a member of each command type as a variable.



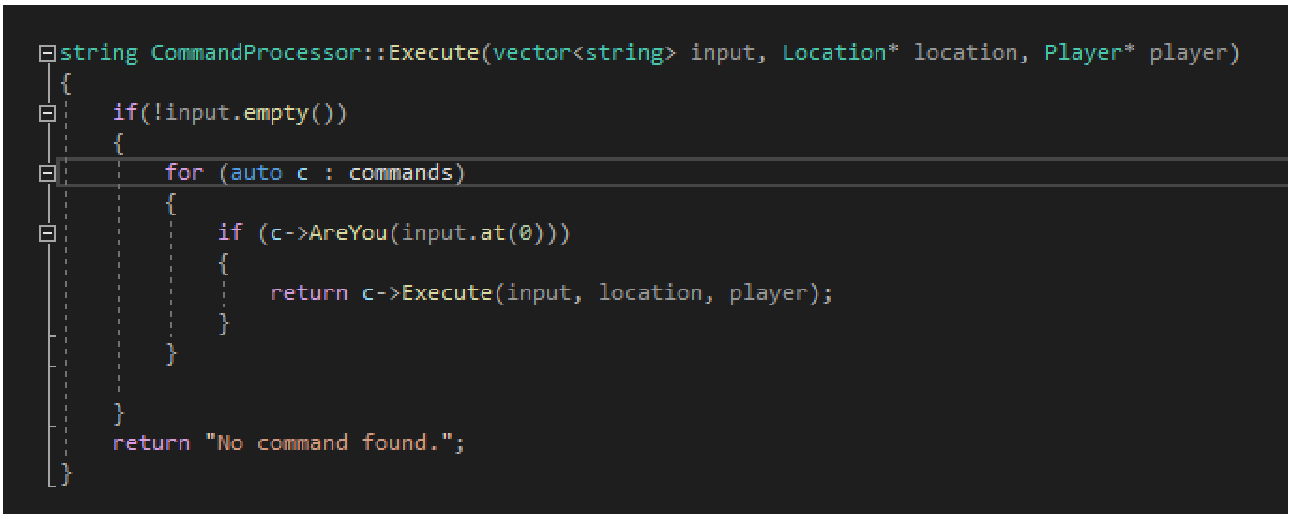
```
class CommandProcessor
{
private:
    LookCommand look;
    GoCommand go;
    HelpCommand help;
    DebugCommand debug;
    InventoryCommand inventory;
    AliasCommand alias;
    QuitCommand quit;
    vector<Command*> commands;
    //map<string, vector<string>> aliases;

public:

    CommandProcessor(bool*l);
    CommandProcessor() {};
    ~CommandProcessor() {};

    string Execute(vector<string> input, Location* location, Player* player);
};
```

When the processor is constructed, it will set all of the ids for each command and add a reference of them to the command vector.



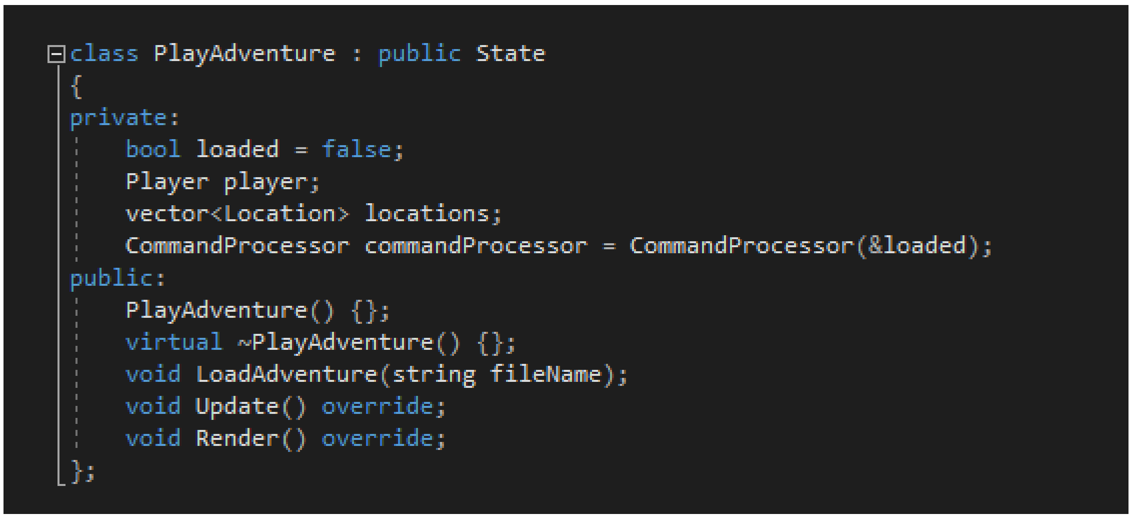
```
CommandProcessor::CommandProcessor(bool * l)
{
    go = GoCommand({ "go", "move" });
    look = LookCommand({ "look", "inspect" });
    help = HelpCommand({ "help", "?" });
    debug = DebugCommand({ "debug" });
    inventory = InventoryCommand({ "inventory" });
    quit = QuitCommand({ "quit", "exit" }, l);

    commands.push_back(&go);
    commands.push_back(&look);
    commands.push_back(&help);
    commands.push_back(&debug);
    commands.push_back(&inventory);
    commands.push_back(&alias);
    commands.push_back(&quit);

    alias = AliasCommand({ "alias" }, commands);
    //vector<string> inv = { "look", "at", "inventory" };
    //aliases.insert(make_pair("inventory", inv));
}
```

When a command is executed, the processor checks the first input string to see if it is a command. If so, it will pass the input string vector, location and player pointers into that command's execute function, executing the command.



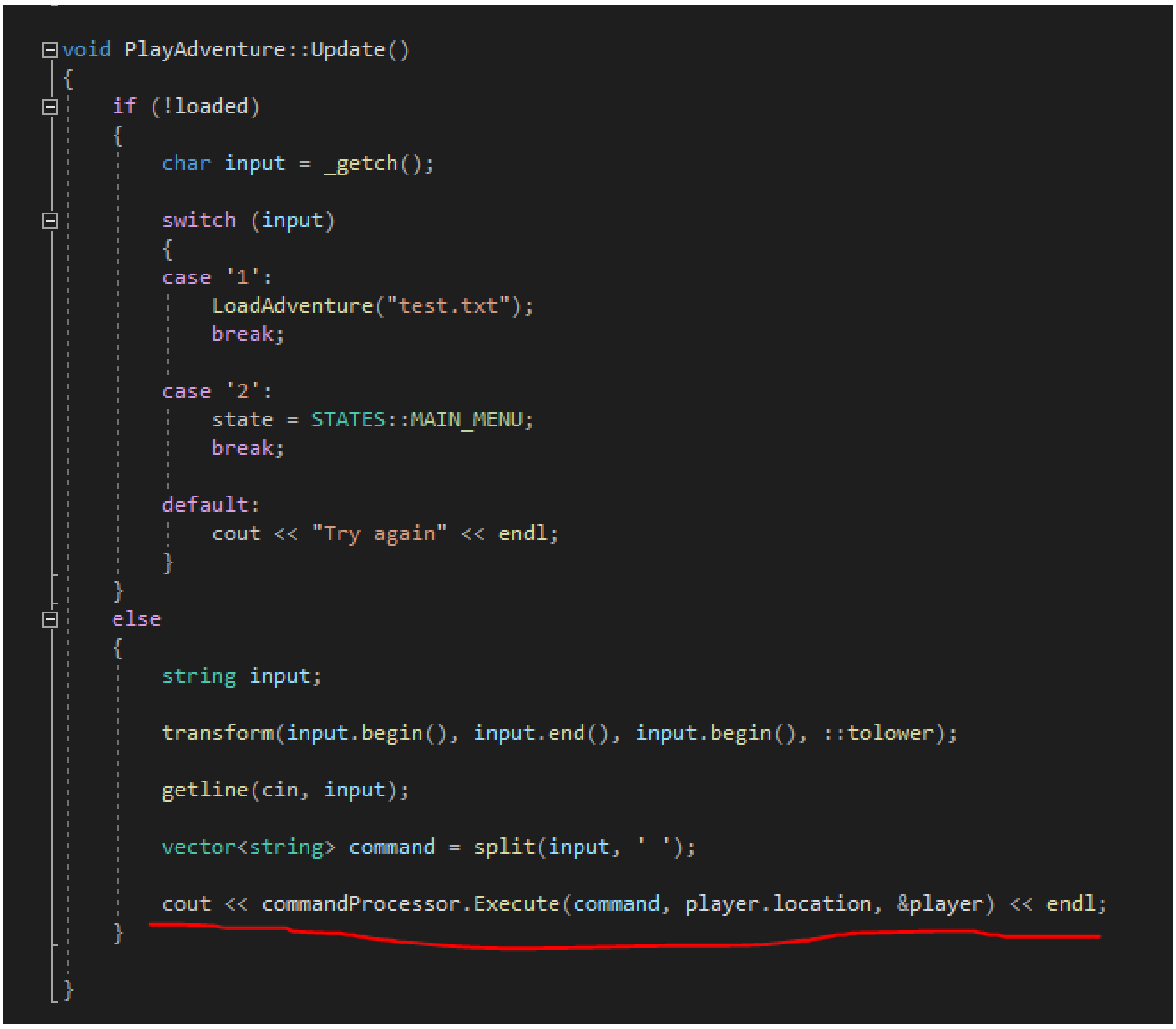
```cpp
string CommandProcessor::Execute(vector<string> input, Location* location, Player* player)
{
    if(!input.empty())
    {
        for (auto c : commands)
        {
            if (c->AreYou(input.at(0)))
            {
                return c->Execute(input, location, player);
            }
        }

    }
    return "No command found.";
}
```

Then we need to use this in the update function. I added the Command processor as a member of the play adventure state.



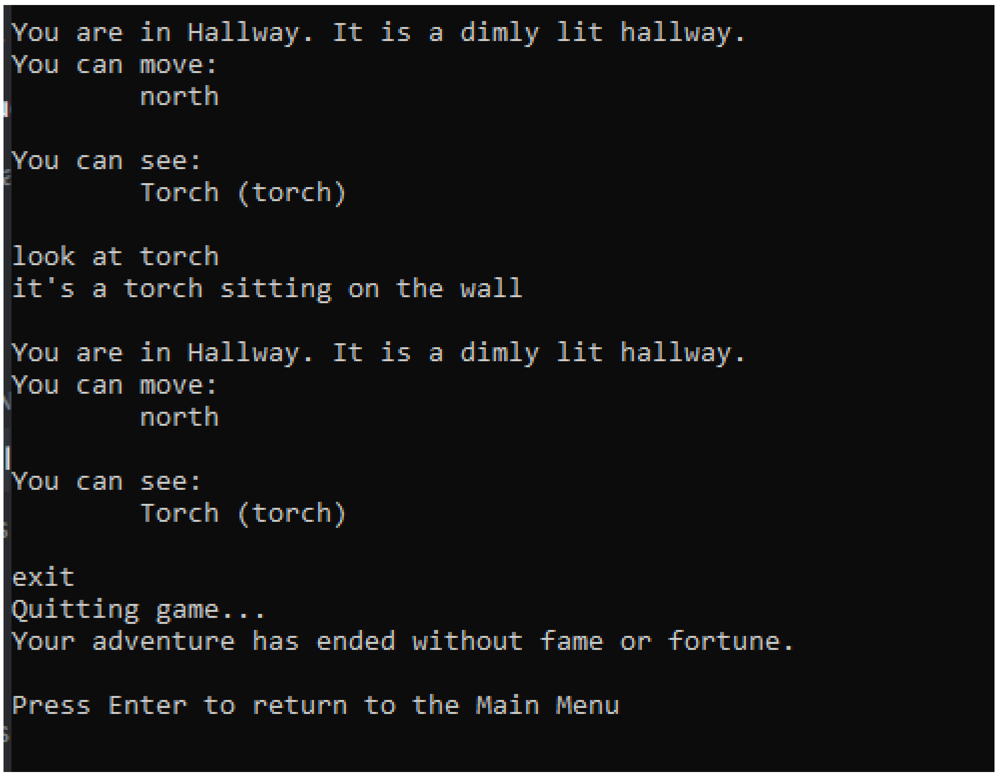
```cpp
class PlayAdventure : public State
{
private:
    bool loaded = false;
    Player player;
    vector<Location> locations;
    CommandProcessor commandProcessor = CommandProcessor(&loaded);
public:
    PlayAdventure() {};
    virtual ~PlayAdventure() {};
    void LoadAdventure(string fileName);
    void Update() override;
    void Render() override;
};
```

Then I called the execute function in update.

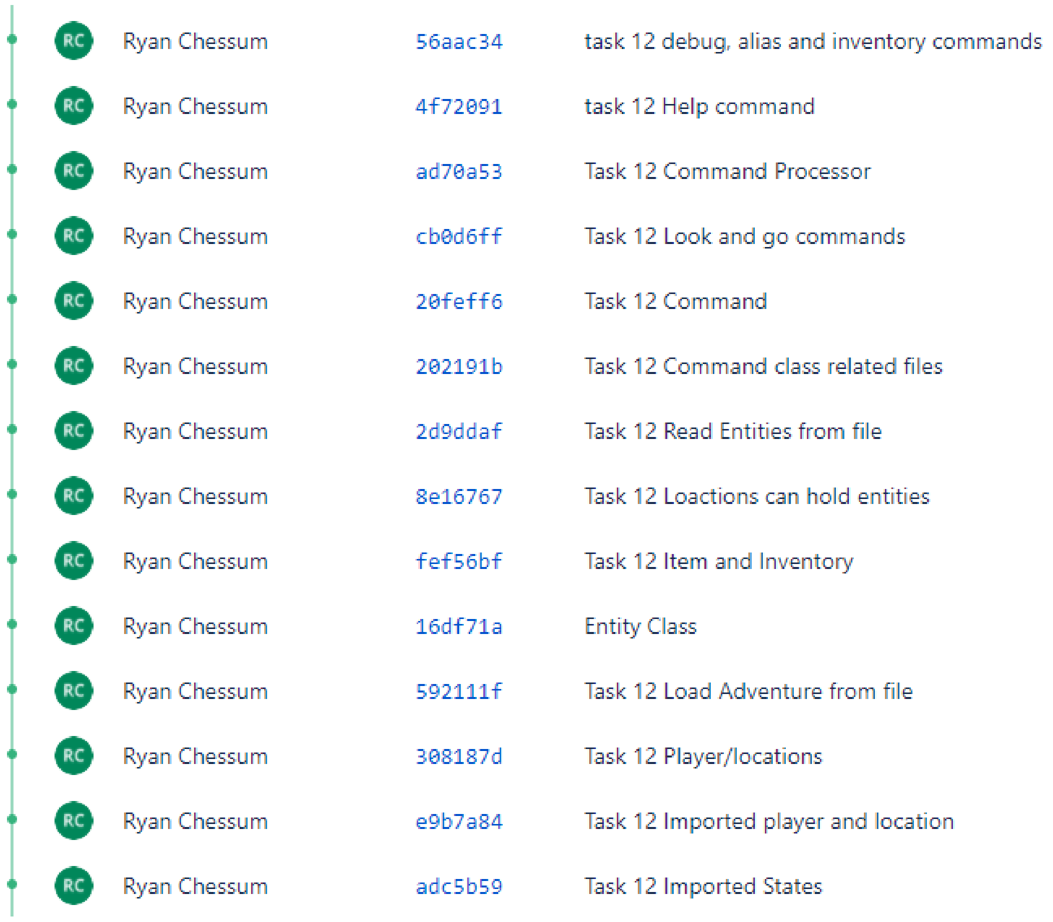


```cpp
void PlayAdventure::Update()
{
    if (!loaded)
    {
        char input = _getch();

        switch (input)
        {
        case '1':
            LoadAdventure("test.txt");
            break;

        case '2':
            state = STATES::MAIN_MENU;
            break;

        default:
            cout << "Try again" << endl;
        }
    }
    else
    {
        string input;

        transform(input.begin(), input.end(), input.begin(), ::tolower);

        getline(cin, input);

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

        cout << commandProcessor.Execute(command, player.location, &player) << endl;
    }

}
```

Now if you try playing the game you should be able to use all the commands we added.



```
You are in Hallway. It is a dimly lit hallway.
You can move:
        north

You can see:
        Torch (torch)

look at torch
it's a torch sitting on the wall

You are in Hallway. It is a dimly lit hallway.
You can move:
        north

You can see:
        Torch (torch)

exit
Quitting game...
Your adventure has ended without fame or fortune.

Press Enter to return to the Main Menu
```



RC	Ryan Chessum	56aac34	task 12 debug, alias and inventory commands
RC	Ryan Chessum	4f72091	task 12 Help command
RC	Ryan Chessum	ad70a53	Task 12 Command Processor
RC	Ryan Chessum	cb0d6ff	Task 12 Look and go commands
RC	Ryan Chessum	20feff6	Task 12 Command
RC	Ryan Chessum	202191b	Task 12 Command class related files
RC	Ryan Chessum	2d9ddaf	Task 12 Read Entities from file
RC	Ryan Chessum	8e16767	Task 12 Loactions can hold entities
RC	Ryan Chessum	fef56bf	Task 12 Item and Inventory
RC	Ryan Chessum	16df71a	Entity Class
RC	Ryan Chessum	592111f	Task 12 Load Adventure from file
RC	Ryan Chessum	308187d	Task 12 Player/locations
RC	Ryan Chessum	e9b7a84	Task 12 Imported player and location
RC	Ryan Chessum	adc5b59	Task 12 Imported States