

Spike: 1**Title:** Gridworld

Author: Ryan Chessum, 102564760

Goals / deliverables:

- Code, see /03 – spike - Gridworld/GridWorld/
- Simple plan/design outline for the program
- 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/>)
- Gridworld game specifications

Tasks undertaken:

- Download and install Visual Studio
- Create a new C++ project
- Design a basic design for the code
- Create a basic game loop structure
- Create a world class that uses a 2d array for a map
- Create a player class that can navigate the 2d array
- Create a game based on the gridworld specifications using the class and gameloop we created

What we found out:

Most games utilise a basic game loop architecture. While running, the game will 1.) get an input from the player, 2.) update the game and process the players input, and 3.) Render the updated game.

To demonstrate this concept, I made a version of the game gridworld for the gridworld specifications on canvas

I made a simple plan before I wrote the code. The game loop will be in the main function. It will do all 3 functions, input, update and render. The game itself will be made up of a few basic objects. There will be a player object to keep track of and update the players current position in the game and a world object to hold the data of the world and give it to us when we need it.

Gridworld Plan

Input()

- Get and store player input

Update()

- Check if input is valid
- Update the players position

Render()

- Display to player the new movement options or try again if movement was invalid

Player

- Coordinates [coordinate]
- + GetCoordinates() [int]
- + MoveNorth()
- + MoveSouth()
- + MoveEast()
- + MoveWest()

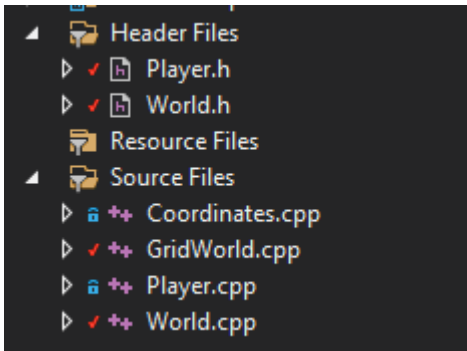
World

- Map[8][9] [string]
- + GetPoint(Coordinate) [string]
- + GetMons(Coordinate) [string]

Next, I began writing the code. The final version I ended up with was a bit different from the plan but ultimately follows the general idea laid out.

To start off with I made the functions for the game loop and created the classes. In C++ classes are (or can be) split into two files. The header files which contain the declaration for the members of the class and then the

definition of each function in the .cpp file. This is so that the function declarations can be included at the top of other files by including the .h file without having the definition meaning the compiler knows what members a class has. I also made a struct for coordinates made up of two integers for x and y. This made it easy to pass both coordinates at the same time to functions and is also a great way of organising our data.



Next, I started to write the code for the player class. It has a member variable to store the coordinates and functions to get its current coordinates and move in either of the 4 directions.

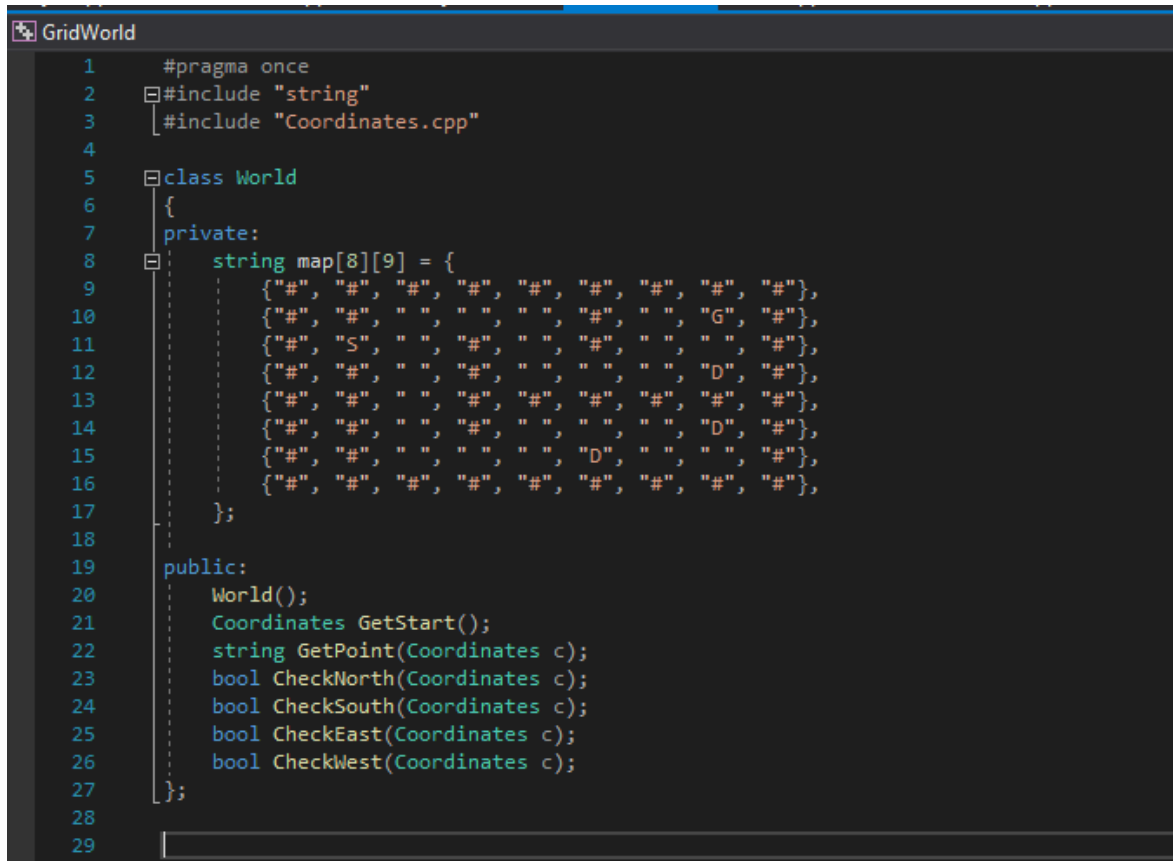
```
1  #pragma once
2  #include "Coordinates.cpp"
3
4  class Player
5  {
6  private:
7      Coordinates coords;
8
9  public:
10     Player(Coordinates startCoords);
11     ~Player() {};
12     Coordinates GetCoords();
13     void MoveNorth();
14     void MoveSouth();
15     void MoveEast();
16     void MoveWest();
17 };
```

The function definitions are very basic. When an instance of the player is created, the starting coordinates are passed in as parameters and set by the constructor. Then the movement functions just add or subtract 1 from the player's position. We won't worry about checking to see if a move is valid as this can be done in the update function.

```
1  #pragma once
2  #include "Player.h"
3  #include <iostream>
4
5  using namespace std;
6
7
8  //constructor
9  Player::Player(Coordinates startCoords)
10 {
11     coords = startCoords;
12 }
13
14 //returns the x coordinate
15 Coordinates Player::GetCoords()
16 {
17     return coords;
18 }
19
20 //moves the player north by updating the y position
21 void Player::MoveNorth()
22 {
23     //cout << coords.y << endl;
24
25     coords.y++;
26
27     //cout << coords.y << endl;
28 }
29
30 //moves the player south by updating the y position
31 void Player::MoveSouth()
32 {
33     coords.y--;
34 }
35
36 // moves the player east by updating the x position
37 void Player::MoveEast()
38 {
39     coords.x++;
40 }
41
42 // moves the player west by updating the x position
43 void Player::MoveWest()
44 {
45     coords.x--;
46 }
47
```

Next, I wrote the code for the World class. The world itself is just a simple 2d array of strings. Initialising a 2d array can be a bit tricky to figure out, adding values will add them to the “second” array as the first one is just an array containing the string arrays. The map I used is copied from the game specifications. To make it a bit easier to understand the coordinates I wanted them to work like traditional coordinates on a cartesian plane with x first and y

second. So, each tile of the world is added bottom to top, left to right. Initialiser lists using curly brackets can make it a bit easier for us to visualise and program this. There are 5 types of tiles: walls (#), the start (S), the goal (G), death tiles (D) and traversable empty tiles represented by a space.



```
GridWorld
1  #pragma once
2  #include "string"
3  #include "Coordinates.cpp"
4
5  class World
6  {
7  private:
8      string map[8][9] = {
9          {"#", "#", "#", "#", "#", "#", "#", "#", "#"},
10         {"#", "#", " ", " ", " ", " ", "#", " ", "G", "#"},
11         {"#", "S", " ", " ", "#", " ", " ", " ", " ", "#"},
12         {"#", "#", " ", " ", "#", " ", " ", " ", "D", "#"},
13         {"#", "#", " ", " ", "#", "#", "#", "#", "#", "#"},
14         {"#", "#", " ", " ", "#", " ", " ", " ", "D", "#"},
15         {"#", "#", " ", " ", " ", " ", "D", " ", " ", "#"},
16         {"#", "#", "#", "#", "#", "#", "#", "#", "#", "#"},
17     };
18
19 public:
20     World();
21     Coordinates GetStart();
22     string GetPoint(Coordinates c);
23     bool CheckNorth(Coordinates c);
24     bool CheckSouth(Coordinates c);
25     bool CheckEast(Coordinates c);
26     bool CheckWest(Coordinates c);
27 };
28
29
```

For the functions, there is nothing to initialise as the world is hardcoded. There is a function to get what tile is at a coordinate and a function which finds and returns the starting position. Lastly instead of a function to return a list of valid coordinates like in the initial design there is a function to check if moving in a direction is a valid move for each function.

The function for finding the start coordinate has a loop within a loop that checks every coordinate for "S". The GetPoint function returns the string at the coordinate passed in. Each check function looks to see if the coordinate 1 unit in that direction contains a wall. If it does, it returns false.

```
31
32 Coordinates World::GetStart()
33 {
34     //change this later to check the map for the starting point but for now just give the hardcoded start
35     //done now searches
36
37     Coordinates c;
38
39     //default start
40     c.x = 0;
41     c.y = 0;
42
43     Coordinates tc;
44     for (int i = 0; i <= 7; i++)
45     {
46         tc.x = i;
47         for (int j = 0; j <= 8; j++)
48         {
49             tc.y = j;
50             if (GetPoint(tc) == "S")
51             {
52                 c = tc;
53             }
54         }
55     }
56     return c;
57 }
58
59 string World::GetPoint(Coordinates c)
60 {
61     return map[c.x][c.y];
62 }
63
64 bool World::CheckNorth(Coordinates c)
65 {
66     if (map[c.x][c.y + 1] != "#")
67     {
68         return true;
69     }
70     return false;
71 }
72
73 bool World::CheckSouth(Coordinates c)
74 {
75     if (map[c.x][c.y - 1] != "#")
76     {
77         return true;
78     }
79     return false;
80 }
81
82 bool World::CheckEast(Coordinates c)
83 {
84     if (map[c.x + 1][c.y] != "#")
85     {
86         return true;
87     }
88     return false;
89 }
90
91 bool World::CheckWest(Coordinates c)
92 {
93     if (map[c.x - 1][c.y] != "#")
94     {
95         return true;
96     }
97     return false;
98 }
```

The next step was using these classes to create the working program in the main function.

```
136 int main()
137 {
138     bool validAction = new bool(true);
139     string input;
140     World world;
141     Player player(world.GetStart());
142
143     cout << "Welcome to GridWorld: Quantised Excitement. Fate is waiting for You!" << endl;
144     cout << "Valid commands: N, S, E and W for direction. Q to quit the game." << endl << endl;
145
146
147
148     Render(world, player, validAction);
149
150     while (isRunning)
151     {
152         validAction = false;
153         input = Input();
154         Update(input, world, player, validAction);
155         Render(world, player, validAction);
156     };
157
158     return 0;
159 }
160
```

Here is the main function for the program. First the variables for the game are initialised, the game is “rendered” for the player and then the game loop starts. In the game loop, the players input is collected and stored in the input variable, the update function is called where it processes the input made. It either updates the players position if their input was a valid move or does nothing if invalid. Then the render function is called and either the new directions the player can move is shown to them or the game tells them that the input was invalid and tells them to try again.

```
string Input()
{
    string i;
    cin >> i;
    transform(i.begin(), i.end(), i.begin(), ::toupper);
    return i;
}
```

The Input function simply uses `std::cin` to get the players input. Then makes it uppercase so we don't have to check for case and returns the value.


```
void Update(string input, World world, Player& player, bool& validAction)
{
    switch (input[0])
    {
    case 'N' :
        if (world.CheckNorth(player.GetCoords()))
        {
            //cout << "you moved north" << endl;
            player.MoveNorth();
            validAction = true;
        }

        break;

    case 'S' :
        if (world.CheckSouth(player.GetCoords()))
        {
            //cout << "you moved south" << endl;
            player.MoveSouth();
            validAction = true;
        }

        break;

    case 'E' :
        if (world.CheckEast(player.GetCoords()))
        {
            //cout << "you moved east" << endl;
            player.MoveEast();
            validAction = true;
        }

        break;

    case 'W' :
        if (world.CheckWest(player.GetCoords()))
        {
            //cout << "you moved west" << endl;
            player.MoveWest();
            validAction = true;
        }

        break;

    case 'Q' :
        isRunning = false;
        validAction = true;
        break;

    default :
        validAction = false;
    }
}
```


The update function takes in the input variable as well as our player and world instances. There is also a Boolean variable which we use to keep track of whether an input was valid or not. Note that the player and Boolean are passed in as constant references with an & symbol in the parameters. This means we can update the variables in main without having to return them. Since the player can only move in cardinal directions, we only really need to check the first letter of the string they enter. This does mean that other strings starting with the same letters can be used but that doesn't matter for the purposes of this program. Since the first letter of a string is just a char we can use a switch statement to check the input. Using the member functions of the class we can check if the player can move in that direction. If so, the player's position is updated. If Q is pressed the isRunning Boolean is set to false and if no valid action is found the default case sets validation to false.

```
void Render(World world, Player player, bool validAction)
{
    // cout << player.GetCoords().x << player.GetCoords().y << endl;
    if (isRunning)
    {
        if (validAction == false)
        {
            cout << "Invalid command, please try again." << endl;
        }

        if (world.GetPoint(player.GetCoords()) == "G")
        {
            cout << "Wow - you've discovered a large chest filled with GOLD coins!" << endl;
            cout << "YOU WIN!!" << endl;

            isRunning = false;

            return;
        }
        else if (world.GetPoint(player.GetCoords()) == "D")
        {
            cout << "Arrrrgh... you've fallen down a pit." << endl;
            cout << "YOU HAVE DIED!" << endl;

            isRunning = false;

            return;
        }

        string moves = "You can move ";

        if (world.CheckNorth(player.GetCoords()))
        {
            moves += "N, ";
        }

        if (world.CheckSouth(player.GetCoords()))
        {
            moves += "S, ";
        }

        if (world.CheckEast(player.GetCoords()))
        {
            moves += "E, ";
        }

        if (world.CheckWest(player.GetCoords()))
        {
            moves += "W, ";
        }

        moves = moves.substr(0, moves.length() - 2);
        cout << moves << ":>";
    }
    else
    {
        cout << "Closing game...";
    }
}
```

Finally, the render function checks the players position to determine what it should display. If the action they entered previously was invalid, a message telling the player is printed to screen. Then if the player position is on the goal or death tile a message is printed, the isRunning variable is set to false and return breaks the function, closing the program. If the player is just on a regular tile, then it puts together a string showing where the player can move and shows it on screen.

```
Microsoft Visual Studio Debug Console
Welcome to GridWorld: Quantised Excitement. Fate is waiting for You!
Valid commands: N, S, E and W for direction. Q to quit the game.

You can move N: >n
You can move S, E, W: >w
You can move N, E: >n
You can move N, S: >n
You can move S, E: >q
Closing game...
C:\Users\ryanc\OneDrive\Documents\GitHub\GamesProgramming\cos30031-102564760\03 - Spike - Gridworld\GridWorld\Debug\GridWorld.exe (process 23756) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

If you run this everything should work as intended.

RC	Ryan Chessum	1f98065	Finished Render Function
RC	Ryan Chessum	b5e8d37	Update Function
RC	Ryan Chessum	78a6fc7	Finished input function
RC	Ryan Chessum	6688fc6	Input, update and render function parameters
RC	Ryan Chessum	dadd3f7	Header files fix
RC	Ryan Chessum	1c73e3a	Player and world datatype corrections + init
RC	Ryan Chessum	2066d52	World class
RC	Ryan Chessum	495d638	i fogor
RC	Ryan Chessum	5c0f007	Added coordinates struct
RC	Ryan Chessum	84dd847	set up player class functions
RC	Ryan Chessum	6f51aab	Lab 2 Complete and gridworld files setup