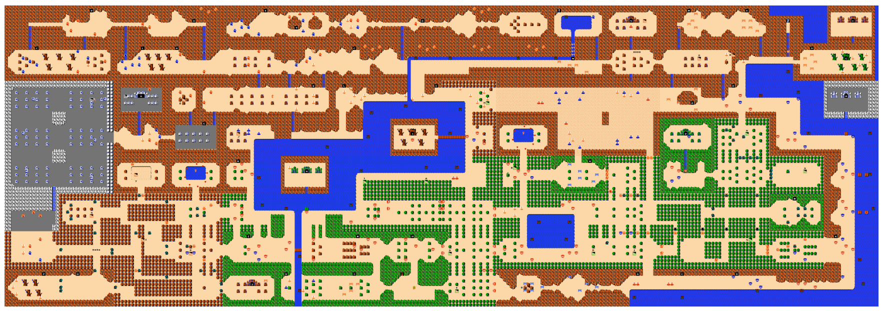
CS 410p/510: Advanced C++
Automated TASing

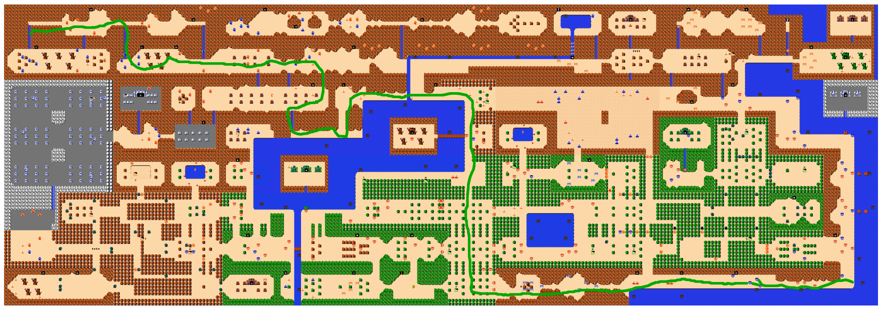
Due: 4/29/20

The purpose of this assignment is to practice using C++'s standard data structures in the STL. The STL includes vectors, linked lists (list), BSTs (set, map), hash sets (unordered_set), hash maps (unordered_map) stacks, queues, and priority queues. If you need a data structure, there's a good chance it's in the STL.

There is a small group of people who try to beat video games in the shortest amount of time possible. They are called speed runners. This actually takes more planning than you would expect. One of the main problems speed runners face is how to route a game, or what path should they take to beat the game in the fastest time possible. As an example let's look at ”The Legend of Zelda”.



By modern standards this game has a pretty simple map. Now, was we all know, the goal of Zelda is to get from the top left corner of the map to the bottom right corner.



There are many ways to do this. So, we'll start by just trying to beat the game, and work our way up to beating the game efficiently.

Preliminaries: the interface

Since I don't want to have to work with bitmaps after assignment 1, we're going to have a must simpler representation of our game. I've made a Maze class that will generate a random maze for you to solve. The maze for Zelda is given below.

```
___________________________________
| _ _ _ |______ ______| | | _ | | |
|___|______  _ _|______________ __|
| _ | | |  ____ | __| _ ____| | |_|
| _ __ _ _|_______| |__ | | | |__ |
| __| | | |_| ____| ___ | |__ __| |
| | _ _ _ | _ __|__ _ _ _ _ _ __| |
|__ __| _____ _ _ _____ _________ |
|_________|_______________________|
```

So, the problem we are trying to solve is get from square (0,0) (the top left) to square (7,16) (the bottom right).

I've given you a few tools. First, There is `path.h` This file defines a few types. A `point` is a pair of integers. You can get the integers from `point p` with `p.first` and `p.second`. A `path` is a `list<point>`.

I've also defined four integer constants `UP, DOWN, LEFT, RIGHT`. These are arbitrary constants to represent directions. I've also given you some convenience functions. `opposite` returns the opposite direction. `moveIn` returns difference between the point, and moving in that direction. So for example: `moveIn(UP)` will return (-1,0), because you are moving one point up. `point operator+` will add two points, so `p + moveIn(UP)` will be the point above `p`. `direction` return the direction from `p1` to `p2`. If `p1` and `p2` aren't adjacent, then it returns the constant `FAILED` You don't have to use any of these functions, but they might be helpful.

I've also included a Maze class

```
class Maze
{
public:
    int rows() const;
    int columns() const;

    bool can_go(int dir, int r, int c) const;

    bool can_go_up(int r, int c) const;
    bool can_go_down(int r, int c) const;
    bool can_go_left(int r, int c) const;
    bool can_go_right(int r, int c) const;

    int cost(int r, int c, int dir) const;
};
```

I don't think there's a lot that's surprising here. You can ask if you can move in any direction from a specific point. You can also use `can_go` to ask using the direction constants. You can also get the cost from moving from point (r,c) in the direction dir.

Solving the maze

In order to solve this maze we need an algorithm. We'll start of simply. First you need to find any valid path from (0,0) with (rows-1,columns-1). I'm going to recommend a backtracking algorithm. It is one of the most fundamental algorithms in Computer Science.

The idea is simple. I'll try to move down. If I can't move down, I'll try to move left. If I can't move left, I'll try to move up. If I can't move up, I'll try to move right. If I can't move in any of those directions, then I'll move back to the room I came from. That is, I'll backtrack to the previous room.

We need a couple of things to make this work. First we need to keep track the room we've come from. There's two ways we can do this. We can make this a recursive function, or we could use a stack. We also need to keep track of the rooms we've seen. If we don't, then we might make an infinite loop. You should never enter a room you've already been in.

Finally, you need a way to keep track of the path. There are several ways to do this. Well... Good luck.

Solving the maze efficiently

The next step is to solve the maze efficiently, by getting the shortest path. There is also a pretty simple algorithm for this, but we need a little bit of math to justify it. If p_1 is a path from a to b and p_2 is a path from b to c, then $p_1 + p_2$ is a path from a to c.

theorem: if $p_1 + p_2$ is the shortest path from a to c, then p_1 is the shortest path from a to b, and p_2 is the shortest path form b to c.
Proof:
Suppose that this isn't true, so there is a path q from a to b that is shorter than p_1. Then $q + p_2$ is a shorter path from a to c, which is a contradiction. The case for p_2 is similar.

This seems like an obvious theorem, but it's pretty powerful for us. It tells us that the shortest path from the start to the end path will always be made up of shortest paths to intermediate rooms. So, as long as we only consider shortest paths, we'll be fine. This leads to the Breadth First Search algorithm.

First I'll look at all of my neighbors (rooms I can get to in one step). If I haven't seen them before, I add them to a queue. If I have seen them, then there is a shorter path to them, so I ignore them. Then I look at the next room in the queue.

If you look at this algorithm, it kind of looks like water spilling out into all of the rooms.

Step 1:

```
___________________________________
|X_ _ _ |______ ______| | | _ | | |
|___|______  _ _|______________ __|
| _ | | |  ____ | __| _ ____| | |_|
| _ __ _ _|_______| |__ | | | |__ |
| __| | | |_| ____| ___ | |__ __| |
| | _ _ _ | _ __|__ _ _ _ _ _ __| |
|__ __| _____ _ _ _____ _________ |
|_________|_______________________|
```

Step 2:

```
___________________________________
|+_+_ _ |______ ______| | | _ | | |
|±__|______  _ _|______________ __|
| _ | | |  ____ | __| _ ____| | |_|
| _ __ _ _|_______| |__ | | | |__ |
| __| | | |_| ____| ___ | |__ __| |
| | _ _ _ | _ __|__ _ _ _ _ _ __| |
|__ __| _____ _ _ _____ _________ |
|_________|_______________________|
```

Step 3:

```
___________________________________
|+_+_+_ |______ ______| | | _ | | |
|±_±|______  _ _|______________ __|
| _ | | | _____ | __| _ ____| | |_|
| _ ______|_______| |__ | | | |__ |
| __| | | |_| ____| ___ | |__ __| |
| | _ _ _ | _ __|__ _ _ _ _ _ __| |
|__ __| _____ _ _ _____ _________ |
|_________|_______________________|
```

Step 4:

```
___________________________________
|+_+_+_+|______ ______| | | _ | | |
|±_±|±_____  _ _|______________ __|
| _ | | | _____ | __| _ ____| | |_|
| _ ______|_______| |__ | | | |__ |
| __| | | |_| ____| ___ | |__ __| |
| | _ _ _ | _ __|__ _ _ _ _ _ __| |
|__ __| _____ _ _ _____ _________ |
|_________|_______________________|
```

Solving a weighted maze

This is good, but it can often take a variable amount of time to go between rooms in a game. Taking the freeway might mean that you drive more miles, but you can drive faster on the freeway, so you might get home sooner then if you took back streets.

We want to solve the weighted maze shortest path problem. Instead of adding weights to the edges of the maze. I'm going to give each room its own height. All of the height will be between 0 and 9. Some rooms might have a height of 9, while others might have a height of 4. So, the cost from going from the 9 room to the 4 room is 5. In general the cost from room `p1` to room `p2` is `abs(p1.height() - p2.height())`. Fortunately you don't need to worry about this. The method `Maze::cost(row, col, dir)` will get the cost from room (row,col) moving in direction dir.

Now we need to find the path with the lowest cost. Since this isn't necessarily the shortest path, we can use our Breadth First Search Algorithm. Fortunately we can solve this problem with math again. If p_1 is a path from a to b with total cost c_1, and p_2 is a path from b to c with total cost c_2, then $p_1 + p_2$ is a path from a to c with total cost $c_1 + c_2$.

theorem2: if $p_1 + p_2$ is the path of least cost from a to c, then p_1 is the path of least cost from a to b, and p_2 is the path of lest cost form b to c.
Proof:
Suppose that this isn't true, so there is a path q from a to b that is with cost $c_q < c_1$. Then $q + p_2$ has cost $c_q + c_2 < c_1 + c_2$. This is a contradiction. The case for p_2 is similar.

This suggests a very similar algorithm to Breadth First Search. In fact this algorithm is called Dijkstra's algorithm, named after its discoverer Edsger w. Algorithm. In our last algorithm we wanted to process each room in the order we discovered it. This guaranteed that the length of the path to each room was as short as possible. However, this time we want to make sure the *cost* of each path is as short as possible. We want to give higher priority to rooms with a smaller total cost.

Advanced: Actually solving games

It turns out the goal of Zelda isn't actually to get from the top left corner to the bottom right corner. The goal is actually to visit the 9 dungeons. You can visit them in any order, but you have to visit all 9. We want to be able to do this as well, but I'll simplify it for our maze. You will start in the room (rows/2, columns/2), visit all four corners, and return to the starting room. From the last part, we already have an algorithm for going from one room to another in the shortest time. We'll start by constructing a matrix of the cost of going from one corner to any other. As an example with the start S, and corners 1, 2, 3, 4:

```
  S 1 2 3 4
S 0 9 7 3 9
1 9 0 3 2 8
2 7 3 0 9 4
3 3 2 9 0 2
4 9 8 4 2 0
```

Now, we can calculate the total cost of any path that goes from S to all 4 corners, and back to S. We need to find the shortest one. The STL's Algorithms library might be helpful here.