OOP - Maze Project - 50 points

Problem/Goal: Write a C++ console maze game similar to the example provided. See Figure 1.
http://www.harding.edu/dsteil/345/Assignments/maze.exe

http://www.harding.edu/dsteil/345/Assignments/mazeStudent.exe

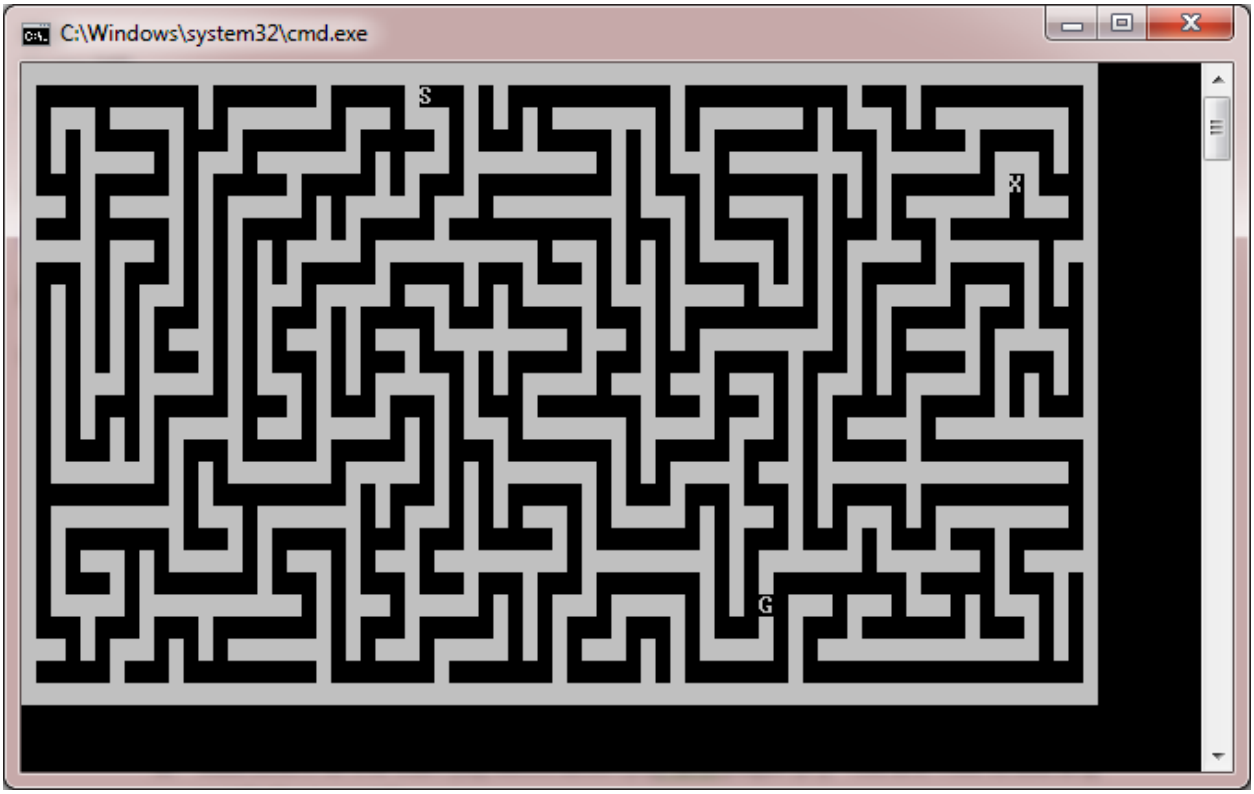

Figure 1

Requirements

Your program must:

1. Generate a new random solvable maze each time the program is run. I used an algorithm represented in the flowchart in Figure 2. You may use any other algorithm you would like. The problem with is algorithm is that the "wrong" paths are often easy to see because they are so short. (25 points)
2. Mark the starting point of the maze. (3 points)
3. Select a destination within the maze that can be navigated to, and mark it. (3 points)
4. Allow the user to use the arrow keys to navigate through the maze. Move a character to indicate where the player current is. The player should not be able to move through walls or go out of bounds. (8 points)
    a. I used the GetAsyncKeyState(i) function from the windows.h library to read the arrow keys from the keyboard . The method I used eats up a processor while the program is running (I am not a fan of that).

    b. I used `setConsoleCursorPosition` from the windows.h library to change where I would write characters at specific coordinates
5. Notify the user when they have completed the maze. (3 points)
6. Have at least one of the following options: (8 points)
    a. Add colors to the start (S), destination (X), and player (P). You are free to use any symbols you want.
    b. Keep a low time list. Allow the user to enter their name if they have one of the 10 best times. You will most likely want to save this to a text file.
    c. Have a ghost or ghosts that randomly move around the maze. If you touch them you lose. You may allow ghosts to move through walls. This option will require threading.
    d. Add an options menu to set things such as the maze size and START, DESTINATION, and PLAYER symbols.
    e. Drop bread crumbs. Pick up the breadcrumbs and do not drop more if you are backtracking. In this way, when you are done only the solution is marked.
    f. Add portals (special walls) that will move the player somewhere else on the board (may be a random location).

Design requirements:

1. Organize the vast majority of your maze logic within a class named Maze.
2. Organize the operations you perform on the maze in concise methods.
3. Use the STL stack in the maze generation. See example STL stack code at http://www.dreamincode.net/forums/topic/57497-stl-stack/.

Your syntax must: (Penalty points will be deducted for breaking the following syntax rules.)

1. Use descriptive variable and function names. No abbreviations.
2. Have appropriate classes, properties and methods.
3. Use constants where appropriate.
4. Avoid duplicate code.
5. Avoid long, cumbersome, functions.
6. Avoid comments that repeat the code. (No flower boxes before functions.)
7. Always use {} with conditional statements even when they are not required.
8. Always free memory that has been dynamically allocated.

What to Turn in:

Submit on Easel a single zip file containing your visual studio solution folder. This should contain a solution file, project file, source code files, and possibly a couple of other files**. Delete all debug, release, object and bin folders along with any file with the extension .sdf before zipping the folder. Your zip file should be ~25KB or less.** 5 points will be deducted for not following these instructions.

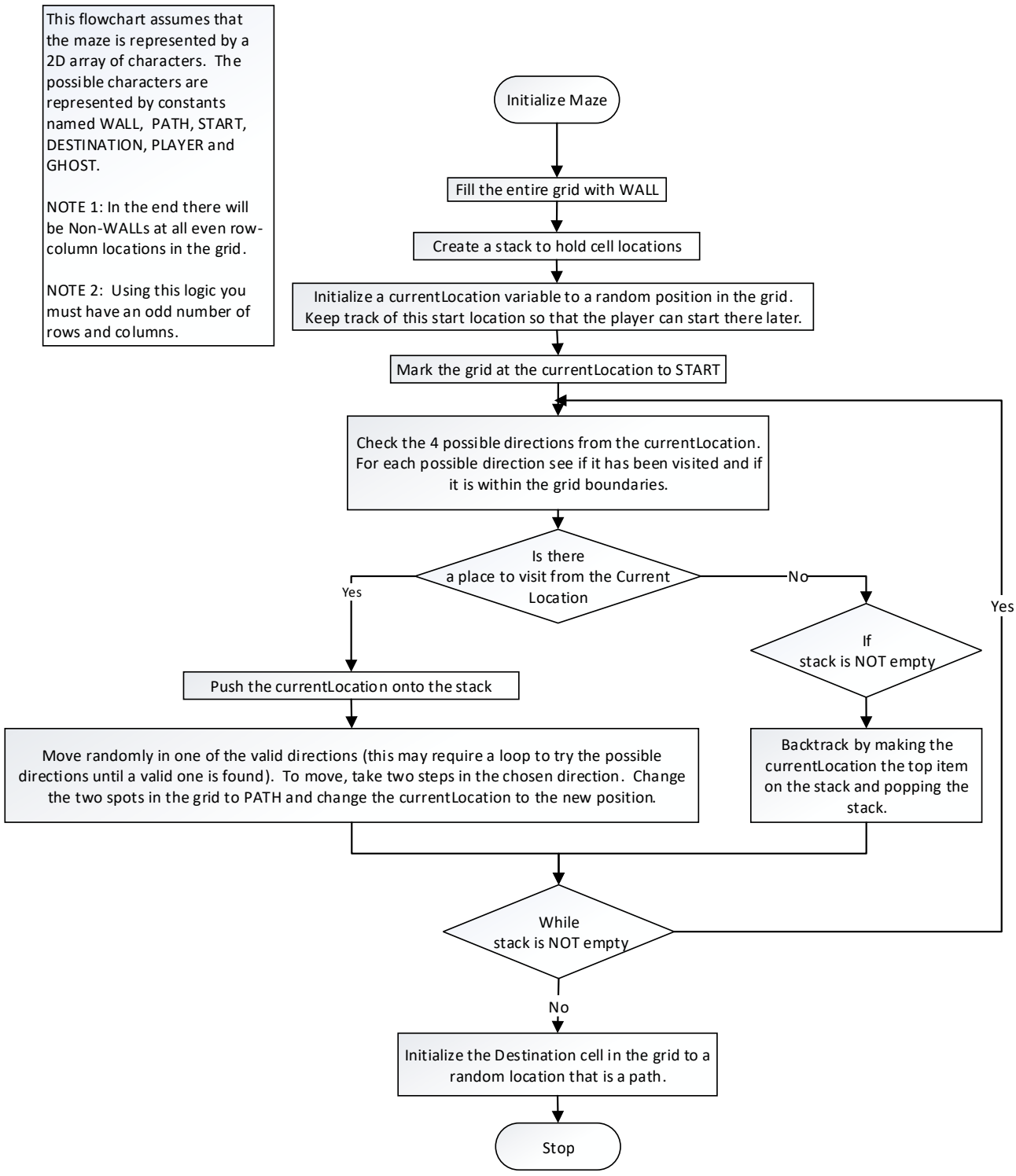
This flowchart assumes that the maze is represented by a 2D array of characters. The possible characters are represented by constants named WALL, PATH, START, DESTINATION, PLAYER and GHOST.
NOTE 1: In the end there will be Non-WALLs at all even row-column locations in the grid.
NOTE 2: Using this logic you must have an odd number of rows and columns.
Initialize Maze
Fill the entire grid with WALL
Create a stack to hold cell locations
Initialize a currentLocation variable to a random position in the grid. Keep track of this start location so that the player can start there later.
Mark the grid at the currentLocation to START
Check the 4 possible directions from the currentLocation. For each possible direction see if it has been visited and if it is within the grid boundaries.
Is there a place to visit from the Current Location
Yes
No
Push the currentLocation onto the stack
Move randomly in one of the valid directions (this may require a loop to try the possible directions until a valid one is found). To move, take two steps in the chosen direction. Change the two spots in the grid to PATH and change the currentLocation to the new position.
If stack is NOT empty
Backtrack by making the currentLocation the top item on the stack and popping the stack.
Yes
While stack is NOT empty
No
Initialize the Destination cell in the grid to a random location that is a path.
Stop

Figure 2

```
#include<iostream>
#include<windows.h>
using namespace std;

int getKey();
const char WALL = (char)219;

void main(){
	int row = 0;
	int column = 0;

	int key = getKey();

	COORD newCoord = { column, row };
	SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), newCoord);
	cout << WALL;

	while (key != VK_ESCAPE)    {
		if (key == VK_LEFT && column > 0){
			column--;
		}
		else if (key == VK_RIGHT){
			column++;
		}
		else if (key == VK_UP && row > 0){
			row--;
		}
		else if (key == VK_DOWN){
			row++;
		}
		COORD newCoord = { column, row };
		SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), newCoord);
		cout << WALL;

		key = getKey();
	}
}

int getKey(){
	int result = 0;
	while (result == 0)
	{
		short MAX_SHORT = 0x7FFF; //111111111111111
		if (GetAsyncKeyState(VK_LEFT) & MAX_SHORT){
			result = VK_LEFT;
		}
		else if (GetAsyncKeyState(VK_UP) & MAX_SHORT){
			result = VK_UP;
		}
		else if (GetAsyncKeyState(VK_RIGHT) & MAX_SHORT){
			result = VK_RIGHT;
		}
		else if (GetAsyncKeyState(VK_DOWN) & MAX_SHORT){
			result = VK_DOWN;
		}
	}
	return result;
}
```

Figure 3