Fall 2021 CS 31

Programming Assignment 3
Grid Game

Time due: 11:00 PM Wednesday, October 27

Introduction

You and hundreds of other people are stuck on an island where you must compete in deadly games. In the current game, you are strapped into a robot car that will navigate a maze to reach a goal. You must provide the car with a navigation plan beforehand and can not change it once your ride begins. If the car reaches the goal after obeying the instructions in the plan within a certain time, you survive; if it crashes into a wall, …

Fortunately, you have come across a computer with a C++ compiler installed, so you can write a program to help you verify that your proposed plan is properly formed and will safely get you to the goal.

The maze is represented by a rectangular grid of up to 25 rows and up to 30 columns. Each cell of the grid is either empty or occupied by a wall. Here is a small example:

```
  1234
1 ...*
2 .*..
3 .*..
```

In this example, there are three rows and four columns. There are walls at (1,4), (2,2), and (3,2). (The notation *(r,c)* means the cell at row r, column c.) If the car is at an empty cell, it can move only to an empty cell adjacent to it in one of the four cardinal directions (north, east, south, and west). Following usual map conventions, *north* means in the direction of decreasing row numbers, *east* is in the direction of increasing column numbers, etc. In our model, we assume that there are walls all around the grid, so that a car at (2,4) is blocked from moving east. In the example above, a car at the empty cell (1,2) could move west to (1,1) or east to (1,3); it could not move south to (2,2), because there's a wall there, nor could it move north, since the assumed wall around the grid blocks it.

Let's define a *turn letter* as by one of the two letters R or L, in either upper or lower case, and a *direction letter* as one of the four letters N, E, W, or S, in either upper or lower case. We define a *plan portion* as zero, one, or two digits, followed by a turn letter. Examples are L and 2R and 13L. A *plan* is a sequence of zero or more plan portions (not separated or surrounded by spaces, commas, or anything else; every character in a non-empty plan must be part of a portion). An example of a plan is LL2R2r2L1R. Given a start position and a start direction the car is facing, an *obeyable plan* is a plan that the car could obey from start to end without starting on or trying to move to a cell containing a wall. By *obeying* a plan, we mean interpreting the plan portions, in order, as instructions to move the specified number of steps forward in the direction the car is facing, then rotating the car 90 degrees as instructed by the turn letter: right/clockwise for R, left/counterclockwise for L. A turn letter preceded by no digits means to not move the car any steps, but simply rotate it as instructed. As for a plan portion with two digits, the plan portion `13R`, for example, means to move 13 steps forward and then rotate right.)

As an example, in the grid shown above, if the start position is (3,1) and the car s facing south, then LL2R02LR0r2L1R is an obeyable plan: The car is directed to turn left (so it now faces east), turn left (so it now faces north), move forward 2 steps and turn right (so it's now at (1,1) facing east), move forward 2 steps and turn left (so it's now at (1,3) facing north), turn right (so it now faces east), move 0 steps and turn right (so it's still at (1,3), but now facing south), move forward 2 steps and turn left (so it's now at (3,3) facing east), and move forward 1 step and turn right (so it's now at (3,4) facing south. This plan is obeyable, since it never directs a car into a wall grid point or off the grid. (An obeyable plan that for the same starting conditions would end up at the same place without making any unnecessary turns would be LL2R2r2L1R.) From the same starting position and direction, none of these are obeyable plans: 1R (goes off the grid), LL3R (goes off the grid), L3R (moves into a wall), LL2R3r2L (moves into a wall). Starting from (2,2) there are no obeyable plans, since (2,2) itself is a wall. The string R0FL is not an obeyable plan because it is not even a plan: it doesn't fit the definition of a plan.

Your task

For this project, you will implement the following three functions, using the exact function names, parameter types, and return types shown in this specification. (The parameter *names* may be different if you wish.)

```
bool hasCorrectForm(string plan)
```

This function returns true if its parameter is a plan (i.e., it meets the definition above). Here are two strings that are plans: 5rL00L0R09R7L and 42l (the last character of the latter is a lower case L). Here are four strings that are not plans: L2, 1Rx2L, 144R, 1R+2L. N144, and w2+n3. Notice that a string may be a plan without being obeyable in a particular grid. (This is analogous to the sentence "Travel east on Wilshire Blvd. until you come to the first signal past Sunset Blvd." being syntactically correct English, but not obeyable in Los Angeles, since Wilshire never intersects Sunset.)

```
int determineSafeDistance(int r, int c, char dir, int maxSteps)
```

This function determines the number of steps a car starting at position (`r`,`c`) could travel in the direction indicated by `dir`. In the normal case, when this function is called, (`r`,`c`) is an empty grid position, `dir` is a direction letter, and `maxSteps` is the proposed number of steps to travel in the indicated direction. In this case, if the car starting at (`r`,`c`) could indeed travel that number of steps in that direction without moving to a grid point containing a wall or running off the edge of the grid, then the function returns that number of steps; otherwise, the function returns the maximum number of valid steps in that direction the car could travel (which will be less than the value of `maxSteps`, and might even be zero). If (`r`,`c`) is not a valid empty grid position, or if `dir` is not a direction letter, or if `maxSteps` is negative, the function returns −1.

```
int obeyPlan(int sr, int sc, int er, int ec, char dir, string plan, int& nsteps)
```

This function determines the number of steps a car starting at position `sr`,`sc`) and initially facing in the direction indicated by `dir` travels when obeying the indicated plan, which should lead to the end position (`er`,`ec`). ~~In the normal case, (`sr`,`sc`) and (`er`,`ec`) are empty grid positions, `dir` is a direction letter, and `plan` is an obeyable plan.~~ In this case, the function sets `nsteps` to the number of steps a car starting at (`sr`,`sc`) initially facing in direction `dir` travels when obeying the complete plan, and returns 0 if the car ends up at (`er`,`ec`), or 1 otherwise. ~~If (`sr`,`sc`) or (`er`,`ec`) are not valid empty grid positions or if `dir` is not a direction letter or if the `plan` string is not a plan, the function returns 2 and leaves `nsteps` unchanged~~. If (`sr`,`sc`) and (`er`,`ec`) are empty grid positions and `dir` is a direction letter and the `plan` string is a plan, but the car could not obey the complete plan without moving to a cell containing a wall or running off the edge of the grid, then the function returns 3 and sets `nsteps` to the maximum number of steps that the car can travel (which might be 0) obeying the plan until it tries to move to a cell containing a wall or off the grid (which might be 0). ~~The function must *not* assume that `nsteps` has any particular value at the time this function is entered.~~

These are the only three functions you are required to write. (Hint: `obeyPlan` may well call the other two functions.) Your solution may use functions in addition to these three if you wish. While we won't test those additional functions separately, using them may help you structure your program more readably.

Of course, to test the functions you write, you'll want to write a main routine that creates a grid and calls your functions. During the course of developing your solution, you might change that main routine many times. As long as your main routine compiles correctly when you turn in your solution, it doesn't matter what it does, since we will rename it to something harmless and never call it (because we will supply our own main routine to thoroughly test your functions).

To help you write your functions, we have provided `getRows`, `getCols`, and `isWall` routines that you may call to determine properties of the grid. They're described in the Project 3 Grid Library writeup, along with `setSize` and `setWall` routines you'll want to use in your main routine to set up a test grid, and `draw` routines you may want to use to help you visualize what's going on. Make sure you never call any of these routines with invalid arguments, because if you do, it will abruptly terminate your program after writing an error message.

All the code you write will be in the file `gridgame.cpp`. The routines we provide are in the files `grid.h` and `grid.cpp`. You will turn in only `gridgame.cpp`, so don't make any changes to `grid.h` or `grid.cpp`, since we will never see any such changes. (We'll use our own versions when testing your code.)

Programming Guidelines

The functions your write must not use any global variables whose values may be changed during execution. Global *constants* are allowed.

When you turn in your solution, none of the three required functions, nor any functions you write that they call, may read any input from `cin` or write any output to `cout`. (Of course, during development, you may have them write whatever you like (perhaps by calling `draw`) to help you debug.) If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish. (Note that the `draw` functions write to `cerr`, not `cout`.)

The correctness of your program must not depend on undefined program behavior. Your program must never access out of range positions in a string. Your program must not, for example, assume anything about `n`'s value at the point indicated, or even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    int n;              // n is uninitialized
    s.at(5*n/n) = '!';  // undefined behavior!
    …
```

Be sure that your program builds successfully, and try to ensure that your functions do something reasonable for at least a few test cases under both g31 and either Visual C++ or clang++. That way, you can get some partial credit for a solution that does not meet the entire specification.

There are a number of ways you might write your main routine to test your functions. One way is to interactively accept test strings:

```
int main()
{
    setSize(3,4);
    setWall(1,4);
    setWall(2,2);
    setWall(3,2);
    for (;;)
    {
        cout << "Enter plan: ";
        string p;
        getline(cin, p);
        if (p == "quit")
            break;
        cout << "hasCorrectForm returns ";
        if (hasCorrectForm(p))
            cout << "true";
        else
```

```
                cout << "false";
            cout << endl;
        }
    }
```

While this is flexible, you run the risk of not being able to reproduce all your test cases if you make a change to your code and want to test that you didn't break anything that used to work.

Another way is to hard-code various tests and report which ones the program passes:

```
int main()
{
    setSize(3,4);
    setWall(1,4);
    setWall(2,2);
    setWall(3,2);
    if (hasCorrectForm("2R1r"))
        cout << "Passed test 1: hasCorrectForm(\"2R1r\")" << endl;
    if (!hasCorrectForm("1Lx"))
        cout << "Passed test 2: !hasCorrectForm(\"1Lx\")" << endl;
    if (determineSafeDistance(3, 1, 'N', 2) == 2)
        cout << "Passed test 3: determineSafeDistance(3, 1, 'N', 2)" << endl;
    int len;
    len = -999;  // so we can detect whether obeyPlan sets len
    if (obeyPlan(3,1, 3,4, 'S', "LL2R2r2L1R", len) == 0  &&  len == 7)
        cout << "Passed test 4: obeyPlan(3,1, 3,4, 'N', \"LL2R2r2L1R\", len)" << endl;
    len = -999;  // so we can detect whether obeyPlan sets len
    if (obeyPlan(3,1, 3,4, 'N', "1Lx", len) == 2  &&  len == -999)
        cout << "Passed test 5: obeyPlan(3,1, 3,4, 'N', \"1Lx\", len)" << endl;
    len = -999;  // so we can detect whether obeyPlan sets len
    if (obeyPlan(2,4, 1,1, 'w', "3R1L", len) == 3  &&  len == 1)
        cout << "Passed test 6: obeyPlan(2,4, 1,1, 'w', \"3R1L\", len)" << endl;
    …
```

This can get rather tedious. Fortunately, the library has a facility to make this easier: `assert`. If you #include the header `<cassert>`, you can call `assert` in the following manner:

```
assert(some boolean expression);
```

During execution, if the expression is true, nothing happens and execution continues normally; if it is false, a diagnostic message is written to `cerr` telling you the text and location of the failed assertion, and the program is terminated. Using `assert`, we can write the tests above more easily:

```
#include <iostream>
#include <cassert>
using namespace std;

bool hasCorrectForm(string plan)
{
    … Your code goes here …
}

int determineSafeDistance(int r, int c, char dir, int maxSteps)
{
    … Your code goes here …
}

int obeyPlan(int sr, int sc, int er, int ec, char dir, string plan, int& nsteps)
{
    … Your code goes here …
}

int main()
```

```
        {
            setSize(3,4);
            setWall(1,4);
            setWall(2,2);
            setWall(3,2);
            assert(hasCorrectForm("2R1r"));
            assert(!hasCorrectForm("1Lx"));
            assert(determineSafeDistance(3, 1, 'N', 2) == 2);
            int len;
            len = -999;  // so we can detect whether obeyPlan sets len
            assert(obeyPlan(3,1, 3,4, 'S', "LL2R2r2L1R", len) == 0  &&  len == 7);
            len = -999;  // so we can detect whether obeyPlan sets len
            assert(obeyPlan(3,1, 3,4, 'N', "1Lx", len) == 2  &&  len == -999);
            len = -999;  // so we can detect whether obeyPlan sets len
            assert(obeyPlan(2,4, 1,1, 'w', "3R1L", len) == 3  &&  len == 1);
            …
            cerr << "All tests succeeded" << endl;
        }
```

The reason for writing one line of output at the end is to ensure that you can distinguish the situation of all tests succeeding from the case where one function you're testing silently crashes the program.

What to turn in

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **gridgame.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code. A project that contains your `gridgame.cpp`, your `grid.cpp` and our `grid.h` must be able to be built and run, so `gridgame.cpp` must contain appropriate #include lines, a main routine, and any additional functions you may have chosen to write.

2. A file named **report.docx** or **report.doc** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains:
   a. A brief description of notable obstacles you overcame.
   b. A description of the design of your program. You should use pseudocode in this description where it clarifies the presentation.
   c. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests, but you must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) If you use the `assert` style above for writing your test code, you can copy those `asserts`, along with a very brief comment about what each is testing for. Notice that most of this portion of your report can be written just after reading the requirements in this specification, before you even start designing your program.

Your zip file must *not* contain the `grid.h` and `grid.cpp` files that we supply. When we test your program, we will use our own versions of these files that are specially instrumented for automated testing.

By October 26, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in. There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.