Programming Assignment 3
Rescue Robots

Time due: 9:00 PM Thursday, October 29

Introduction

You are an intern at Intelligent Tools, a company that manufactures robots that work in structures that are too dangerous for people to enter, such as burning buildings. (Their motto: *Intelligent Tools rush in where humans fear to tread*℠.) While a robot needs to be able to find a route from a start position to a goal, your task is simpler than writing the code that does that: You need to write something that will simply verify that a proposed route indeed works.

The floor plan of a building is represented by a rectangular grid of up to 20 rows and up to 20 columns. Each cell of the grid is either empty or occupied by an obstacle (a wall, a piece of furniture, a pillar — we'll just call them all walls). Here is a small example:

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

In this example, there are three rows and four columns. There are walls (i.e., some kind of obstacle) at (1,4), (2,2), and (3,2). (The notation *(r,c)* means the cell at row r, column c.) Of course, it's understood that there are walls all around the building. If a robot 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 the example above, a robot 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 outer wall of the building blocks it.

Let's define a *route segment* as one of the four letters N, E, S, or W, in either upper or lower case, followed by zero, one, or two digit characters. Examples are N and s2 and e13. A *route* is a sequence of zero or more route segments; an example is N2e1Es2e1. Given a start position, a *navigable route* is one a robot could navigate from start to end without starting on or trying to move to a cell containing a wall. By *navigating* a route, we mean interpreting the route segments, in order, as instructions to move the specified number of steps in the specified direction. (A direction letter followed by no digits mean to move one step in that direction. As for two digits, the route segment `e13`, for example, means to move 13 steps east.)

As an example, in the grid shown above, if the start position is (3,1), then N2eE01n0s2e1 is a navigable route: The robot is directed to go north 2 steps to (1,1), east 1 step to (1,2), east 1 step to (1,3), north 0 steps (so it won't move), south 2 steps to (3,3), and east 1 step to (3,4), without ever trying to move to a wall segment or off the grid. From the same starting position, none of these are navigable routes: W, N3, N2e3. Starting from (2,2) there are no navigable routes, since (2,2) itself is a wall.

Your task

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

```
bool isRouteWellFormed(string route)
```

> This function returns true if its parameter is a syntactically valid route, and false otherwise. A syntactically valid route is a sequence of zero or more route segments (not separated by spaces, commas, or anything else). Here are two syntactically valid routes: N2eE01n0e2e1 and W42. Here are four strings that are not syntactically valid routes: 3sn, e1x, N144, and w2+n3. Notice that a route may be syntactically valid without being navigable in a particular grid.

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

> This function determines the number of steps a robot starting at position (`r`,`c`) could take in the direction indicated by `dir`. In the normal case, when this function is called, (`r`,`c`) is an empty grid position, `dir` is one of the letters N, E, S, or W, in either upper or lower case, and `maxSteps` is the proposed number of steps to take in the indicated direction. In this case, if the robot starting at (`r`,`c`) could indeed take that number of steps in that direction without moving to a cell 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 robot could take (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 an invalid direction character, or if `maxSteps` is negative, the function returns −1.

```
int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps)
```

> This function determines the number of steps a robot starting at position (`sr`,`sc`) takes when following the indicated route, which should lead to the end position (`er`,`ec`). In the normal case, (`sr`,`sc`) and (`er`,`ec`) are empty grid positions and `route` is a syntactically valid navigable route. In this case, the function sets `nsteps` to the number of steps a robot starting at (`sr`,`sc`) takes when navigating the complete route, and returns 0 if the robot ends up at (`er`,`ec`), or 1 otherwise. If (`sr`,`sc`) or (`er`,`ec`) are not valid empty grid positions or if `route` is not syntactically valid, the function returns 2 and leaves `nsteps` unchanged. If (`sr`,`sc`) and (`er`,`ec`) are empty grid positions and `route` is syntactically valid, but the robot could not navigate the complete route 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 robot can take along the route (which might be 0). You 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: `navigateRoute` 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, their use may help you structure your program more readably. Of course, to test all your functions, 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 throroughly 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 `maze.cpp`. The routines we provide are in the files `grid.h` and `grid.cpp`. You will turn in only `maze.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.)

Your functions must not use any global variables whose values may be changed during execution (so 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 could not, for example, assume anything about `c`'s value at the point indicated, or even whether or not the program crashes:

```
int main()
{
    string s = "Hello";
    int n;          // n is uninitialized
    char c = s[n];  // The program might die here; even if it
                    // does not, c's value is undefined
    …
```

You must use your best programming style, including commenting where appropriate. Your program must build successfully. Try to ensure that your functions do something reasonable for at least a few test cases. 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 route: ";
        string route;
        getline(cin, route);
        if (route == "quit")
            break;
        cout << "isRouteWellFormed returns ";
        if (isRouteWellFormed(route))
            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 (isRouteWellFormed("n2e1"))
        cout << "Passed test 1: isRouteWellFormed(\"n2e1\")" << endl;
    if (!isRouteWellFormed("e1x"))
        cout << "Passed test 2: !isRouteWellFormed(\"e1x\")" << endl;
    if (navigateSegment(3, 1, 'N', 2) == 2)
        cout << "Passed test 3: navigateSegment(3, 1, 'N', 2)" << endl;
    int len;
    len = -999;  // so we can detect whether navigateRoute sets len
    if (navigateRoute(3,1, 3,4, "N2eE01n0s2e1", len) == 0  &&  len == 7)
        cout << "Passed test 4: navigateRoute(3,1, 3,4, \"N2eE01n0s2e1", len)" << endl;
    len = -999;  // so we can detect whether navigateRoute sets len
    if (navigateRoute(3,1, 3,4, "e1x", len) == 2  &&  len == -999)
        cout << "Passed test 5: navigateRoute(3,1, 3,4, \"e1x\", 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 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 isRouteWellFormed(string route)
{
    … Your code goes here …
}

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

int navigateRoute(int sr, int sc, int er, int ec, string route, int& nsteps)
{
    … Your code goes here …
}

int main()
```

```
{
    setSize(3,4);
    setWall(1,4);
    setWall(2,2);
    setWall(3,2);
    assert(isRouteWellFormed("n2e1"));
    assert(!isRouteWellFormed("e1x"));
    assert(navigateSegment(3, 1, 'N', 2) == 2);
    int len;
    len = -999;  // so we can detect whether navigateRoute sets len
    assert(navigateRoute(3,1, 3,4, "N2eE01n0s2e1", len) == 0  &&  len == 7);
    len = -999;  // so we can detect whether navigateRoute sets len
    assert(navigateRoute(3,1, 3,4, "e1x", len) == 2  &&  len == -999);
    len = -999;  // so we can detect whether navigateRoute sets len
    assert(navigateRoute(2,4, 1,1, "w3n1", len) == 3  &&  len == 1);
    …
    cout << "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 **maze.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. The file must be a complete C++ program that can be built and run, so it must contain appropriate #include lines, a main routine, and any additional functions you may have chosen to write.

2. A file named **report.doc** or **report.docx** (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 and the expected result of each test. You don't have to include your program's actual results for 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.) 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 28, 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, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." 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.