Vector of Vectors in C++

Many problems give input data that can most easily be visualized as a two-dimensional grid of either characters or integers. Note that since a vector of characters is just a string, just a vector of strings is the same as a vector of vector of char. For example, the following might represent the elevation at each grid cell. Treat the top left as (0, 0), the first number as the row index and the second number as the column index:

3 2 5 6
1 6 3 8
3 5 7 10

Reading in a Vector of Vectors

The most natural way to store this in C++ is a vector of vectors. In many problems, the dimensions of a grid of data will be given right before the data. Thus, the data above might be presented as:

```
3    4
3    2    5    6
1    6    3    8
3    5    7    10
```

where the first number represents the number of rows, and the second, the number of columns.

Here is how we can read in this data:

```
int r,c,data;
vector<vector<int>> grid;
cin >> r >> c;
for (int i=0; i<r; i++) {
    vector<int> tmp(c);
    for (int i=0; i<c; i++) {
        cin >> tmp[i];
    }
    grid.push_back(tmp);
}
```

In short, we are creating a vector of vector<int> first. Inside the outer for loop, we create a temporary vector of size c and read in each value into this vector, one by one. Once the vector tmp is complete, we add it to the end of grid.

Accessing/Changing Values in a Vector or Vectors

To access an individual variable store in a vector of vectors, simply use []'s, where the first index specifies which vector, and the second index specifies which element in that vector, and as usual, the indexes are zero-based.

Note that in the example above, grid.size() is 3 while grid[0].size() is 4. For this data, grid[1][3] would be set to 8 (the value in the second row and last column).

If we wanted to add 1 to each location in grid, we could do the following:

```
for (int i=0; i<grid.size(); i++)
    for (int j=0; j<grid[0].size(); j++)
        grid[i][j]++;
```

DR/DC Arrays

In many grid problems, we must "move" from one square to another, and there are some restrictions on how we can move. Almost always, we are given some fixed set of moves. Most commonly, we might be allowed to move up, down, left or right. Thus, if we are currently at location [r][c], we may move to one of the four following locations:

[r-1][c]
[r][c-1]
[r][c+1]
[r+1][c]

We could "spell these out" in four separate statements, but it's much better to harness loops, and store the change in row and change in column (hence the abbreviations dr and dc) in a constant vector, as follows:

```
const vector<int> DR = {-1,0,0,1};
const vector<int> DC = {0,-1,1,0};
```

Notice that (DR[0], DC[0]) represents the first "direction of movement" from the original location, (DR[1], DC[1]) represents the second "direction of movement" from the original location, and so forth.

One thing to also note is that depending on the actual values of r and c, some of these neighboring squares might be off the grid. Thus, with almost all grid programs, it's important to write an inbounds function that takes in the coordinates of a location and returns true if and only if those coordinates are inbounds. Taking all of this into account, some generic code to go through each neighboring square to a location [r][c] might look like this:

```
for (int i=0; i<DR.size(); i++)
    if (inbounds(r+DR[i], c+DC[i])
         // Process grid[r+DR[i]][c+DC[i]]
```

All of this will make more sense once we look at an example.

Kattis Problem: Nine Knights (Vector of Vectors, use of DR/DC arrays)
The link to the problem is here:

https://open.kattis.com/problems/nineknights

Knights have very specific movement (an L shape). Thus, we must first design DR/DC arrays that encapsulate how knights move (2 in one direction, 1 in the other):

```
const vector<int> DR = {-2,-2,-1,-1,1,1,2,2};
const vector<int> DC = {-1,1,-2,2,-2,2,-1,1};
```

In this problem, the grid is always size 5 x 5. Thus, we can create another constant N = 5, which will make checking if a location is inbounds easier. In problems where the # of rows and columns is variable, we can make these global variables. While this is a bad practice in large industrial coding projects, it's common practice in competitive programming to simplify code (reducing the number of parameters passed to functions). Good competitive programmers have their own style rules for which variables are allowed to be global and which ones should not be. Over time, you'll develop rules for yourself that you are comfortable with that minimize the chance of errors.

In this problem, we have to verify two main things:

(a) There are exactly 9 knights on the grid.

(b) No square that contains a knight can attack another square that contains a knight.

Since the grid is static (many grids won't be), it's perfectly fine to make it a global variable. Counting knights can be done while reading in the input. Finally, three functions will be helpful:

(a) inbounds function, as previously mentioned

(b) a function that takes in (r,c) and sees if any of the squares that can be attacked from that square contain a knight.

(c) Wrapper function that calls function (b) from each location with a knight and returns true iff two knights can attack each other.

Now that we've planned out each of the pieces of the solution, let's take a look at it:

```
using namespace std;
#include <bits/stdc++.h>

// Useful constants.
const int N = 5;
const vector<int> DR = {-2,-2,-1,-1,1,1,2,2};
const vector<int> DC = {-1,1,-2,2,-2,2,-1,1};

vector<string> grid;
```

```
bool inbounds(int r, int c);
bool canAttack(int r, int c);
bool twoAttack();

int main() {

    string tmp;
    int numK = 0;

    // Read grid, add up knights.
    for (int i=0; i<N; i++) {
        cin >> tmp;
        grid.push_back(tmp);
        for (int j=0; j<N; j++)
            if (grid[i][j] == 'k')
                numK++;
    }

    // Check both conditions...
    if ( (numK == 9) && !twoAttack())
        cout << "valid" << endl;
    else
        cout << "invalid" << endl;

    return 0;
}

bool inbounds(int r, int c) {
    return r>=0 && r<N && c>=0 && c<N;
}

bool canAttack(int r, int c) {

    // We just check each neighboring square.
    for (int i=0; i<DR.size(); i++)
        if (inbounds(r+DR[i],c+DC[i]) && grid[r+DR[i]][c+DC[i]] =='k')
            return true;
    return false;
}

bool twoAttack() {

    // Go to each square.
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {

            // Skip ones without knights.
            if (grid[i][j] != 'k') continue;

            // If this knight can attack another, we're done.
            if (canAttack(i, j)) return true;
        }
    }

    // If we get here, no two knights attack each other.
    return false;
}
```

Notice that by breaking this problem down into separate pieces, the solution itself is rather manageable. Each function does something fairly simple and straightforward. Also, notice that both canAttack and twoAttack have if statements without an else. If some condition is true, we immediately know the answer for the whole data, without looking further. But, if that condition isn't true, we don't necessarily know the answer and have to continue looking. A common error beginning programmers make is to have an else with a return in situations like this where adding the else makes code incorrect. Finally, for logical clarity, it sometimes nice to have a wrapper function call a smaller function, like twoAttack calls canAttack.

Now, let's look at a second problem on a grid of characters involving a crossword puzzle.

Kattis Problem: prva

Here is a link to the problem:

https://open.kattis.com/problems/prva

In this puzzle we must find all words in a crossword type puzzle, two letters or longer. Of all of these words, we must identify the one that comes first alphabetically. A crossword puzzle is a grid where all of the words are either horizontal, from left to right, or vertical, from top to bottom, so there are only two directions of movement possible. (So, instead of having DR/DC arrays of size 8 like the last problem our DR/DC array will only be size 2.)

In order for a square to be the start of a valid word, if the word is horizontal, either it must start on the left most square, OR the square directly to the left must be "crossed-out." Consider the following puzzle:

```
pot#car
e#o#a#a
a#u#r#t
rotate#
```

Here are all of the horizontal words of length 2 or greater:

```
pot
car
rotate
```

Here are all of the vertical words of length 2 or greater:

```
pear
tour
cart
rat
```

Of all of these words, car comes first alphabetically.

To solve this problem, we can simply try each possible starting position and direction. First, we must see if the combination of location and direction is a valid starting location. Then, if it is, we want to create the word formed by going in the appropriate direction until we either (a) hit the end of the grid, OR hit a '#'. For every word formed, we can compare it to the best answer yet and simply always store the first alphabetical choice. Let's take a look at the solution:

```
using namespace std;
#include <bits/stdc++.h>

// Can only move in two ways.
const vector<int> DR = {0,1};
const vector<int> DC = {1,0};

// Global grid
int r, c;
vector<string> grid;

// Function prototypes.
bool inbounds(int myr, int myc);
bool isValidStart(int myr, int myc, int dir);
string formWord(int myr, int myc, int dir);
string solve();

int main() {

    // Read in the grid.
    cin >> r >> c;
    for (int i=0; i<r; i++) {
        string tmp;
        cin >> tmp;
        grid.push_back(tmp);
    }

    // Solve it!
    cout << solve() << endl;

    return 0;
}

// Returns true iff (myr,myc) is in the grid.
bool inbounds(int myr, int myc) {
    return myr>=0 && myr<r && myc>=0 && myc<c;
}

// Returns true iff (myr,myc) in direction dir is a valid starting
// spot. Assumes (myr,myc) is inbounds.
bool isValidStart(int myr, int myc, int dir) {
    if (grid[myr][myc] == '#') return false;
    if (!inbounds(myr-DR[dir], myc-DC[dir])) return true;
    return grid[myr-DR[dir]][myc-DC[dir]] == '#';
}
```

```
string formWord(int myr, int myc, int dir) {
    string res = "";

    // Keep going as long as we're inbounds.
    while (inbounds(myr,myc) && grid[myr][myc] != '#') {

        // Add in this character.
        res += grid[myr][myc];

        // Advance to next square in direction dir.
        myr += DR[dir];
        myc += DC[dir];
    }

    // This is the string formed in this direction...
    return res;
}

// Returns the solution to the problem.
string solve() {

    string res = "";

    // Try all potential locations/directions.
    for (int i=0; i<r; i++) {
        for (int j=0; j<c; j++) {
            for (int k=0; k<DR.size(); k++) {

                // Skip these.
                if (!isValidStart(i,j,k)) continue;

                // A word to consider.
                string tmp = formWord(i,j,k);

                // Skip these.
                if (tmp.size() < 2) continue;

                // Two situations to update our answer.
                if (!res.compare("")) res = tmp;
                else if (tmp.compare(res) < 0) res = tmp;
            }
        }
    }

    return res;
}
```

Notice the key use of the DR/DC arrays in both the isValidStart and formWord functions. By subtracting those values, we naturally see “the square before”. By incrementing by those values, we naturally move to the next square we need to get to to form our word. Without the DR/DC arrays, much of the code doubles in length and requires if statements where it’s easy to introduce bugs into the code.

Kattis Problem: Counting Stars (Putting it all Together)

This final problem will put together the ideas we've already seen in this lecture along with the idea of recursion from a previous lecture to implement an algorithm called, "floodfill." Here is a link to the problem:

https://open.kattis.com/problems/countingstars

Just like the previous question, we can loop through each square on the grid. If it's a star square ('-'), then we must find and mark the whole star that is connected to this point and add 1 to a total count.

How do we mark all the squares in a connected star? We can utilize the floodfill algorithm (which is a specific type of depth first search, a concept you'll likely see in the future). The idea behind a floodfill is as follows:

1. Mark the square you are at.
2. Look at each neighboring square. If any are unmarked, recursively fill those squares also.

The reason this is called a floodfill is because in the old Paint programs, whenever a user marked a closed region, and then clicked inside of it with the Paint Bucket tool with a selected color, all the pixels within that closed region would turn to that color. In essence, the water started at that spot, and filled in all directions until it hit a barrier. In this question, the '-' is where water stars, and the '#' and out of bounds are the boundaries that stop the fill. We want to count the number of regions to be filled.

In the solution presented here, the '*' character will be used to fill stars. This way, once a star is seen, all of its grid squares will no longer appear to be new stars. Thus, over the course of the algorithm, this grid will change so that all regions with '-' will change to be marked with '*'. A different common technique is to not change the grid, but keep a separate 2D vector of boolean to mark which grid squares have been previously visited.

Here is the solution, pay special attention to the floodfill function fillStar:

```
using namespace std;
#include <bits/stdc++.h>

// Stars are connected in 4 directions.
const vector<int> DR = {-1,0,0,1};
const vector<int> DC = {0,-1,1,0};
const char FILL = '*';
const char STAR = '-';

// Global grid
int r, c;
vector<string> grid;

// Function prototypes.
bool inbounds(int myr, int myc);
int solve();
void fillStar(int myr, int myc);

int main() {

    // Process cases.
    int cnt = 1;
    while (cin >> r) {

        // Gotta clear the grid...
        cin >> c;
        grid.clear();

        // Read it.
        for (int i=0; i<r; i++) {
            string tmp;
            cin >> tmp;
            grid.push_back(tmp);
        }

        // Solve it!
        cout << "Case " << cnt++ << ": " << solve() << endl;
    }

    return 0;
}

// Returns true iff (myr,myc) is in the grid.
bool inbounds(int myr, int myc) {
    return myr>=0 && myr<r && myc>=0 && myc<c;
}
```

```
// Solves the given input case.
int solve() {
    int res = 0;

    // Go to each square.
    for (int i=0; i<r; i++) {
        for (int j=0; j<c; j++) {

            // If necessary, fill this star.
            if (grid[i][j] == STAR) {
                res++;
                fillStar(i, j);
            }
        }
    }

    // Ta da!
    return res;
}

// Fill the star at (myr, myc).
void fillStar(int myr, int myc) {

    // Fill this square.
    grid[myr][myc] = FILL;

    // Try all directions.
    for (int i=0; i<DR.size(); i++) {

        // Skip stuff out of bounds and previously filled.
        if (!inbounds(myr+DR[i],myc+DC[i])) continue;
        if (grid[myr+DR[i]][myc+DC[i]] != STAR) continue;

        // Recursively fill this star.
        fillStar(myr+DR[i],myc+DC[i]);
    }
}
```

Notice that the fill function itself is quite short. The structure is generally similar in most applications. First fill/mark the square. Next, try all adjacent squares. Skip ones that are out of bounds, previously visited, or invalid. Then, if a neighbor is previously unvisited and valid, recursively call the fill function on that square.

The rest of the pieces of this solution should remind you of the previous problems in this section. Many of the patterns in these problems can be found in other competitive programming problems.