HOMEWORK #8:

Heist!

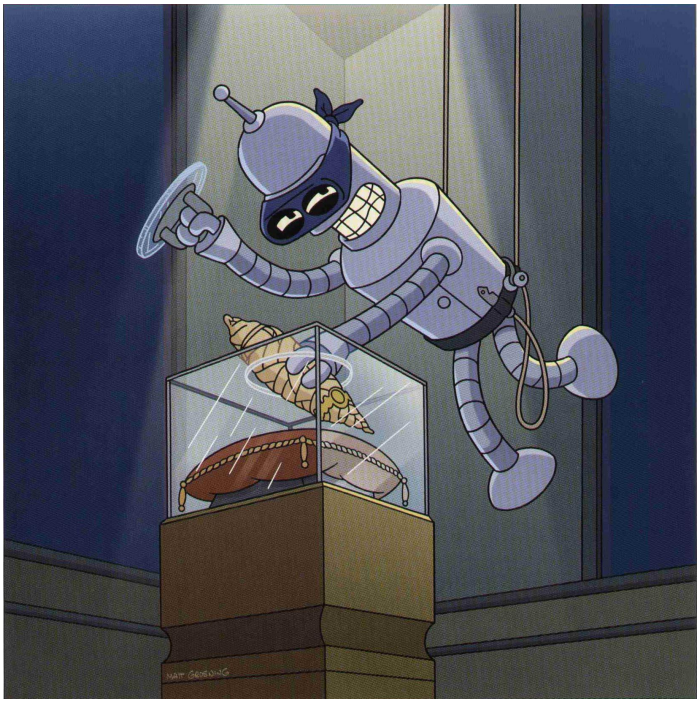
Due Date: Wednesday, April the 13th, 11:59:59pm

For this assignment, submit as many files as you need, but let your '`main()`' function be in a file called : **'`heist.cpp`'**. Remember to put your name and section at the top of all files. Your simulation program should expect all input to come from '`cin`', and all your output should be to '`cout`'.

Problem:

Bender is trapped in the middle of a heist! Suppose the bank Bender is robbing can be mapped as a 2 dimensional grid. Some cells are occupied by walls, so Bender cannot enter them. There is also traps in some of the cells that will do terrible things to Bender if disturbed.

Your job is to write a program that finds, for every map, a path to the exit.



Smooth Criminal.

Input:

The input will consist of a sequence of maps. Each map input starts with the number of columns and the number of rows of the map. In a map, a '`#`' character denotes a wall. A character '`T`' denotes a trap. A ' ' (blank space) character denotes a clear section. '`B`' marks Bender's starting point and '`E`' marks the exit. The input is finished when a map of size `0` by `0` is indicated.

Output:

Output each map with a path from the "Start point" to the "Exit" . Mark the path using cookie crumbles (character '`.`'). Follow the format as in the sample output.

Details:

- Use Recursive Backtracking.
- Bender can move in any of the cardinal directions (North, East, West and South). No diagonal moves are possible.
- Each map will have only one path from Start to Exit, but loops are possible.

Sample

Input	Output
11 4 ########### # #T# # #B# #E# ########### 16 10 ################# # # # T# # # #### ## ## # # ####### # ###### #E # # #T#B # ### ### # # ## # #T# # # ## ####### # # # ################# 0 0	Map : 0 ########### #...#T#...# #B#.....#E# ########### Map : 1 ################# #... # # T# #.#.#### ## ## #.#......####### #.######.#E.. # #.#T#B.#.###.### #.# ##.#.....#T# #.# ##.####### # #...... # #################

Implementation Guidelines:

- Build your own simple test cases.
- Print plenty of status messages to track the progress of your algorithm.
- Start with the following Recursive Backtracking algorithm, and refine it into your implementation. (Or you may use the stack data structure as we discussed in the class).

Recursive Backtracking Algorithm Sketch:

```
SolveMaze(int row, int col)
{

      if(row==goal_row &&col==goal_col) return true;

      if(isSafe(row,col))

      {

            add_to_solution(row,col);



            //travel to 4 directions

            if(SolveMaze(row-1,col)) return true;

            if(SolveMaze(row,col+1)) return true;

            if(SolveMaze(row+1,col)) return true;

            if(SolveMaze(row,col-1)) return true;



            remove_from_solution(row,col);

            return false;

      }

      return false;
}
```

Reading Lines with White-Space

In this assignment, you are required to read lines with white spaces. You may attempt to use something like:

```
cin >> maze[i][j];
```

But that would **NOT** work, as the extraction operator '>>' **ignores** white spaces.

You will therefore be forced to use one of the following library functions:

1. string function `getline()` to read string objects.
2. stream function `getline()` to read "null terminated character arrays".
3. stream function `get()` to read character by character.

Code Sample #1: Reading Strings objects:

```
// Maze is an array of strings
string* maze;

// Readin size of maze
cin >> cs >> rs;
cout << cs << " " << rs << endl;
cin.ignore();              // to move read head to next line

// Allocate Maze Array
maze = new string[rs];

// Read Maze
// each row is a string;
for(int k=0; k < rs; k++){
  getline(cin, maze[k]);
}

// Print Maze Array
for(int k=0; k < rs; k++){
  cout << maze[k] << endl;
}

// De-allocate Maze Array
delete [] maze;
```

Code Sample #2: Reading "Null Terminated Character Arrays":

.

```
// Maze is a 2D array of characters
char** maze;

// Readin size of Maze
cin >> cs >> rs;
cout << cs << " " << rs << endl;
cin.ignore();    // to move read head to next line

// Allocate Maze Array
// Notice that an EXTRA cell is added to the columns
// to account for NULL termination
maze = new char*[rs];
for(int k=0; k < rs; k++){
  maze[k] = new char[cs+1];
}

// Read Maze Array
// Notice that we are reading each line as
// a "NULL Terminated Character Array"
for(int k=0; k < rs; k++){
  cin.getline(maze[k], cs+1);
}

// Print Maze Array
for(int k=0; k < rs; k++){
  cout << maze[k] << endl;
}

// De-allocate Maze Array
for(int k=0; k < rs; k++){
  delete [] maze[k];
}
delete [] maze;
```

.

Code Sample #3: Reading Character by Character:

.

```
// Maze is a 2D array of characters
char** maze;// Readin size of Maze

// Readin size of Maze
cin >> cs >> rs;
cout << cs << " " << rs << endl;
```

```
  cin.ignore();    // to move read head to next line

  // Allocate Maze Array
  maze = new char*[rs];
  for(int k=0; k < rs; k++){
    maze[k] = new char[cs];
  }

  // Read Maza Array
  // Notice that we are reading *Character by Character*
  // and after every row we need to read an extra character
  // to account for the 'end-of-line' character
  char dummy;
  for(int k=0; k < rs; k++){
    for(int j=0; j < cs; j++){
      cin.get(maze[k][j]);
    }
    cin.get(dummy);   // read end-of-line
  }

  // Print Maze Array
  for(int k=0; k < rs; k++){
    for(int j=0; j < cs; j++){
      cout << maze[k][j];
    }
    cout << endl;   // read end-of-line
  }

  // De-allocate Maze Array
  for(int k=0; k < rs; k++){
    delete [] maze[k];
  }
  delete [] maze;
```

.

END.