This problem is a modification of problem #118, “Mutant Flatworld Explorers”, from the ACM International Collegiate Programming Contest Problem Set Archive: http://acm.uva.es/problemset/.

You are asked to write a program that simulates the use of robots to explore a flat world; however, some components of the program will be specified for you.
Synopsis. Robots are placed one at a time on the world and are given commands for moving around in the world. However, since the world is flat, it is possible that a robot will fall off the edge of the world. When a robot falls off the edge of the world, sensing its demise, it leaves behind a ‘marker’ so that other robots will know that the edge of the world is nearby.

The world
The flat world is to be modeled by a mxn rectangular grid (array) with upper-left corner at the point (0, 0) and the opposite corner at the point (m− 1, n − 1). The first coordinate (x) represents horizontal offset (positive direction is left to right), and the other – vertical (positive direction is top to bottom). The values of m and n are supplied at run-time. We will implement the world grid as an abstract data type (ADT); that is, a data type accessed only through a given set of functions — the actual representation (`struct _Grid`) is hidden from the user of the data type. Specifically, the interface for this ADT is given in the following header file.

```
#ifndef CS170_GRID
#define CS170_GRID
namespace CS170 {
  namespace Grid {
    typedef struct _Grid *Instance;
    Instance Create(int x, int y);
    void Destroy(Instance grid);
    bool Inside(Instance grid, int x, int y);
    void Mark(Instance grid, int x, int y);
    bool Marked(Instance grid, int x, int y);
  }
}
#endif
```

Instance — a pointer to an incomplete structure, or opaque pointer. You will provide the definition of the actual structure in your implementation file. We use an opaque pointer here to programmitically shield the user from the ADT from the implementation details: even the actual structure used!
Create — creates a world grid; returns a valid unmarked world grid instance. The dimensions of the world grid should be mxn, where m and n are values provided as function arguments. The array representing the world grid must be allocated dynamically.
Destroy — destroys a previously created world grid.
Inside — tests for grid containment; precondition: grid is valid; returns: true if (x, y) is inside grid; false otherwise.
Mark — places a marker on the world grid;
Marked — tests for a grid marker; precondition: grid is valid and (x, y) is inside grid; returns: true if point (x, y) is marked; false otherwise.

You may not modify the above interface (you must use the grid.h header file as is), and your implementation must consist of a single file, named grid.cpp. You may only use the cstdlib, cassert, and grid.h header files for this part of the assignment.

Robots

Robots are characterized by their position and orientation within the world. The position is given by the coordinates (x, y) of the robot on the world array (x and y are both integers). The orientation is either N (north), S (south), E (east), or W (west); north is along the y–axis: the direction from (0, 1) to (0, 0), and east is along the x–axis: the direction from (0, 0) to (1, 0). Instructions for the robot are encoded by individual characters: one character for each instruction. Robots respond to three instructions: l (turn left 90), r (turn right 90), and f (move forward one world
array unit); all other instructions (characters) are ignored by the robot. If a robot receives an instruction that would move it over the edge of the world, and if the robot is currently at a position that another robot has 'marked', then the robot will ignore the instruction; otherwise, the robot will fall over the edge and will be lost — but not before leaving its own 'mark' on the world. As with the world grid, we will model the robots as an abstract data type. The interface for the robot ADT is given in the following header file.

```
#ifndef CS170_ROBOT
#define CS170_ROBOT
#include "grid.h"

namespace CS170 {
  namespace Robot {
    typedef struct _Robot *Instance;
    enum Orientation { NORTH,     EAST,     SOUTH,     WEST,  UNDEF };
                       //    0        1         2          3
                       //    so turn left  (orientation+3)%4
                       //       turn right (orientation+1)%4
    Instance Create(int _x, int _y, Orientation orientation, Grid::Instance
_grid);
    void Destroy(Instance robot);
    void Move(Instance robot, char cmd);
    bool QueryPosition(Instance robot, int* x, int* y, Orientation *o);
  }
}
#endif
```

The details of this interface are as follows.

Instance — opaque pointer to a robot instance.

Orientation — an enumerated type for the valid robot orientations.

Create — create a robot; precondition: orientation is valid and grid is valid; returns: a valid robot instance (note that it may not be on grid).

Destroy — destroy a previously created robot;

Move — give robot a single command; precondition: robot is valid; postcondition: robot is moved to new position (possibly off the world grid). If robot has moved off the grid, the grid will be marked at point of departure; if

- cmd is invalid (not one of 'f', 'l', or 'r'),
- robot is off grid,
- robot is on a marked point and commanded to move off the grid,

cmd is ignored.

Note: even if robot has moved off the world grid, it must still be destroyed explicitly.

Note: when robot goes off the grid, it will not move again, and it's position remain the same.

QueryPosition — get the position and orientation of a robot - values of x, y, o are filled in; returns: true if robot is on the grid, false if it is lost.

Again, you may not modify this interface. Your implementation of the robot ADT must be in a single file name robot.cpp. You may only use the cassert, grid.h, and robot.h header files for this part of the assignment.

Program (this is already implemented in driver.cpp)
You are to write a program that makes use of the Grid and Robot ADTs. Specifically, your program will allow the user to run one or more robot missions. For each robot mission, the user should be prompted for the initial position and orientation of the robot, and for a sequence of instructions. A typical mission will look something like this:

```
create a robot? y
enter robot position: 2 4
enter robot orientation: w
enter robot instructions: flrfflrrffllr
```

The program should then report the final position and orientation of the robot; if the robot has fallen off the edge of the world, its last known position and orientation (just before it falls off the edge of the world) should be reported, along with the message **lost**.. The user should then be prompted if he/she wants to put forth a new robot.
For example:

```
robot final position: -1 2 w lost
```

and proceed to the beginning of the main loop:

```
create a robot? y
```

If the user responds with 'y', another robot mission should be started; otherwise, the program should terminate. Each robot must be created dynamically before its mission and destroyed after completing its mission.

To simplify the assignment you may assume that input is always legal, that is

- opening (create robot question) is either 'y' or 'n' character
- position is always 2 integers (which may be off the grid though),
- orientation is one of the 4 allowed characters: 'w', 'e', 'n', 's'.
- instruction line is new-line terminated. Instruction line may be **any** line of characters (including spaces, up until the newline character is encountered) and should be accepted as a valid command string, even in it contains invalid commands (these commands are passed to Robot::Move which will just ignored them).

Your output messages must match the above examples exactly. For this part of the assignment, you may include only the iostream, cstdlib, ctime, cassert, grid.h, and robot.h header files, as well as any header files that you create. In addition to program correctness, you will be graded on your coding style.

A typical run:

```
create a robot? y
enter robot position: 2 2
enter robot orientation: n
enter robot instructions: fff
robot final position 2 -1 n lost
create a robot? n
```

another run

```
create a robot? y
enter robot position: 2 2
enter robot orientation: n
enter robot instructions: fff
robot final position 2 -1 n lost
create a robot? y
enter robot position: 2 2
enter robot orientation: n
enter robot instructions: fff
robot final position 2 0 n
create a robot? n
```

In the second run the second robot did not fall off the grid, since the square was marked by the first one.

Running from the command line (using Cygwin, MS prompt doesn't require "./"):

```
./a.out <plan1.txt >out_plan1.txt
```

input from `plan1.txt`, redirect output to `out_plan1.txt`. Both files are available in project folder.

Content of `plan1.txt`:

```
y 2 2 n fff
y 2 2 n fff
y 3 3 w rrrfff
n
```

Content of `out_plan1.txt` (wrapped):

```
create a robot? enter robot position: enter robot orientation: enter robot
instructions: robot final position 2 -1 n lost
create a robot? enter robot position: enter robot orientation: enter robot
instructions: robot final position 2 0 n
create a robot? enter robot position: enter robot orientation: enter robot
instructions: robot final position 3 5 s lost
create a robot?
```

Notice that input data is not echoed as it was with console/keyboard input.

There is another pair available `plan2.txt` and `out_plan2.txt`.

To submit:

Electronically (through my web-site)

- `robot.cpp`
- `grid.cpp`

Hardcopies stapled together (see formatting instructions in the checklist):

- checklist