Homework 6: Assigned in Laboratory 10, Due at Start of Laboratory 12

This is a pledged assignment; you may receive help only from the class teaching assistants, ITC consultants, and the class professors. You may work in groups of up to three for this assignment. Members of a group must come from the same lab section. No code can be shared between groups. *Note, you will be receiving another assignment next week, it will be due the start of Laboratory 13. Therefore, start this assignment now!*

Objective

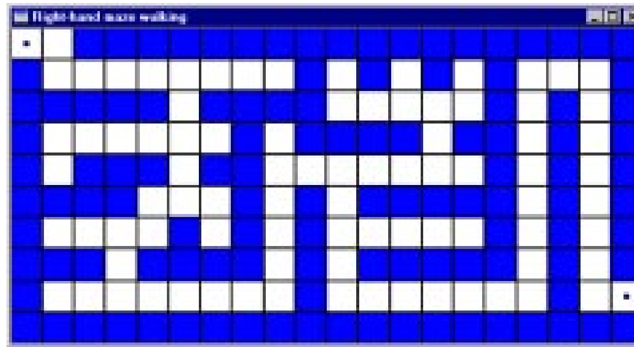
The objective of this assignment is to gain practice with member function development.

Assignment files

A self-extracting archive `hw06.exe` has been created for this assignment. The archive will produce files `maze.ide`, `hw06.cpp`, `location.h`, `wanderer.h`, `goodmaze.exe`, `mydata.dat`, and `maze.lib`. The archive `hw06.exe` can be acquired by accessing the course website `http://www.cs.virginia.edu/cs101`. The project when completed will produce a file `maze.exe` that can be run. You are to submit an electronic and printed copy of `hw06.cpp` before the start of the lab in which it is due. The file `goodmaze.exe` is a compiled working version of the program so that you can see how it is supposed to work. The file `mydata.dat` is an input maze instance. When the maze program prompts for a filename, you can use this file.

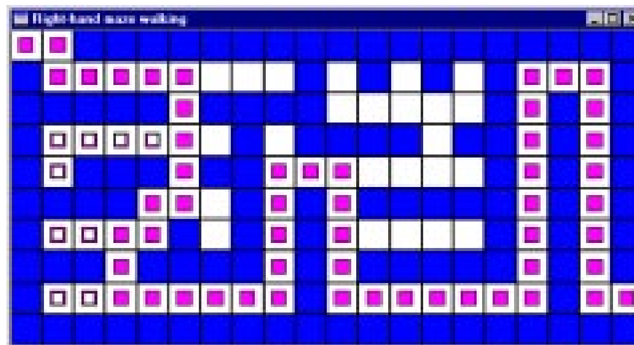
Problem Description

A popular game is traversing a maze by finding a path from the starting point to finishing point. Besides being interesting as a game, the maze traversal problem arises in many important problems such as robot navigation and interconnecting circuit elements on a computer chip. An example maze is given below. In this maze, the starting point is the small square in the upper



left corner. The finishing point is the small square near the lower right corner. The dark areas represent walls and the lighter areas between the walls are corridors. The example maze has no free standing walls, where a free standing wall is one that is not connected to the perimeter. A maze with no free standing walls is called a proper maze. Proper mazes can be solved by using the *right-hand strategy*. The right-hand strategy says to walk along the maze with your right hand constantly sliding along the wall. By doing so, you are guaranteed to reach the finishing point. Note that this strategy does not guarantee your path will be as short as possible. In fact, the strategy may produces a path that doubles back on itself. We consider north to be the direction that heads towards the top of the maze; south to be the direction that heads toward the bottom of the maze; west to be the direction that heads to the left side of the maze; and east to be the direction that heads to the right side of the maze.

A complete traversal of the maze using the right-hand strategy is given in the following figure. After the first fourteen



steps are made the right-hand strategy forces a retracing down this dead-end corridor in the opposite direction. However, when

the entrance to the corridor is reached the second time, the wanderer is facing east, rather than south. Because the wanderer is facing east, the next step with the right-hand strategy is south. Observe in the figure that only the outlines of errant steps are depicted for locations which have been retraced by our wanderer.

We have developed a solution for simulating a wanderer traversing through a maze. To complete the solution you will implement two classes and some auxiliary operators. In particular, you will develop a class Location that represents the integer row and column coordinates of a location along with auxiliary overloadings of the equality operator ==, inequality operator !=, and the stream operators << and >>. The definition of class Location and prototypes of the auxiliary operators are given below. The definition and prototypes are from the file location.h.

```
class Location {
    public:
        // Constructor
        Location(int r = 0, int c = 0);
        // Inspectors
        int GetRow() const;
        int GetColumn() const;
        // Mutators
        void SetLocation(const Location &p);
        void SetLocation(int r, int c);
        void SetRow(int r);
        void SetColumn(int c);
    private:
        // Data members
        int Row;
        int Column;
};
// auxiliary operators
bool operator==(const Location &p, const Location &q);
bool operator!=(const Location &p, const Location &q);
ostream& operator<<(ostream &sout, const Location &p);
istream& operator>>(istream &sin, Location &p);
```

As part of our solution for this problem we also need a class Wanderer that can represent a object that can step through the maze. The Wanderer class requires the use of enumerated types Direction and Status that are already defined for you.

```
enum Direction = { North, East, South, West } ;
enum Status = { Free, Obstacle, Start, Finish, OutOfBounds };
```

Status is used in the labeling of locations with respect to the maze, i.e., whether a location is available for traversing, an obstacle/wall, the starting point, the ending point, or not part of the maze.

The complete definition of the Wanderer class is given below. The definition shows that Wanderer supports information hiding by making all of the data members private. The member functions are all public, because each one is useful to clients. The definitions can be found in the file wanderer.h.

```
class Wanderer {
   public:
      // Constructor
      Wanderer(const Location &p = Location(0,0),
       const Direction &d = East);
      // Inspectors
      Direction GetDirection() const;
      Location GetLocation() const;
      // Facilitators for looking at surroundings
      void LookNorth(const Status &s);
      void LookEast(const Status &s);
      void LookSouth(const Status &s);
      void LookWest(const Status &s);
      // Direction mutator
      void SetDirection(const Direction &d);
      // Movement mutators
      void MoveNorth();
      void MoveEast();
      void MoveSouth();
      void MoveWest();
```

```
      private:
         // Data members
         Direction CurrDirection;
         Location CurrPos;
         bool OkNorth;
         bool OkEast;
         bool OkSouth;
         bool OkWest;
   };
```

Your implementation of the `Location` and `Wanderer` member functions and the `Location` auxiliary operators will go in `hw06.cpp`.

Location member function specification

- `Location(int r = 0, int c = 0)`: a constructor that builds a location at row `r` and column `c`.
- `GetRow()`: an inspector that returns the row coordinate of the location.
- `GetColumn()`: an inspector that returns the column coordinate of the location.
- `SetRow(int r)`: a mutator that sets the row coordinate of the location to `r`.
- `SetColumn(int c)`: a mutator that sets the column coordinate of the location to `c`.
- `SetLocation(int r, int c)`: a mutator that sets the row and column coordinates to `r` and `c` respectively.
- `SetLocation(const Location &p)`: a mutator that sets the row and columns to match the row and column coordinates of `p`.
- `operator==(const Location &p, const Location &q)`: logical auxiliary operator that returns true if locations `p` and `q` represent the exact same coordinate; otherwise, the operator returns false.
- `operator!=(const Location &p, const Location &q)`: logical auxiliary operator that returns true if locations `p` and `q` represent different coordinates; otherwise, the operator returns false.
- `operator<<(ostream &sout, const Location &p)`: inserts to stream `sout` the location coordinate in the form `(r, c)`, where `r` is the row coordinate of `r` and `c` is the column coordinate of `p`. Stream `sout` is also the return value.
- `operator>>(istream &sin, Location &p)`: extracts the next values `r` and `c` from the stream `sout` and uses them to set `p`'s row and column coordinates.

Wanderer member function specification

- `Wanderer(const Location &p = Location(0,0), const Direction &d = East)`: a constructor that builds a wanderer that is at location `p` facing in direction `d`.
- `GetDirection()`: an inspector that returns the direction in which the wanderer is currently facing.
- `SetDirection(const Direction &d)`: a mutator that causes the wanderer to face direction `d`.
- `GetLocation()`: an inspector that returns the location of the wanderer.
- `LookNorth(const Status &s)`: a facilitator that gives the wanderer the message `s` about the status of the location to the immediate north of the wanderer, i.e., the status of the location to the immediate north is `s`.
- `LookSouth(const Status &s)`: a facilitator that gives the wanderer the message `s` about the status of the location to the immediate south of the wanderer.
- `LookEast(const Status &s)`: a facilitator that gives the wanderer the message `s` about the status of the location to the immediate east of the wanderer.
- `LookWest(const Status &s)`: a facilitator that gives the wanderer the message `s` about the status of the location to the immediate west of the wanderer.
- `MoveNorth()`: a mutator that attempts to move the location of the wanderer to the immediate north of the current location. Prior to the invocation of this member, an invocation of `LookNorth()` must have previously been made after the wanderer reached its current location. The move north can only be made if that invocation of `LookNorth()` received neither the message `Obstacle` nor the message `OutOfBounds`. If the move is made, the wanderer is currently facing `North`. If the move cannot be made, there is no change to the wanderer.

  Your wanderer class supports the following member functions that we have defined.
- `MoveSouth()`: a mutator that attempts to move the location of the wanderer to the immediate south of the current location. Prior to the invocation of this member, an invocation of `LookSouth()` must have previously been made after the wanderer reached its current location. The move south can only be made if that invocation of `LookSouth()` received nei-

ther the message Obstacle nor the message OutOfBounds. If the move is made, the wanderer is currently facing South. If the move cannot be made, there is no change to the wanderer.

- MoveEast(): a mutator that attempts to move the location of the wanderer to the immediate east of the current location. Prior to the invocation of this member, an invocation of LookEast() must have previously been made after the wanderer reached its current location. The move east can only be made if that invocation of LookEast() received neither the message Obstacle nor the message OutOfBounds. If the move is made, the wanderer is currently facing East. If the move cannot be made, there is no change to the wanderer.
- MoveWest(): a mutator that attempts to move the location of the wanderer to the immediate west of the current location. Prior to the invocation of this member, an invocation of LookWest() must have previously been made after the wanderer reached its current location. The move west can only be made if that invocation of LookWest() received neither the message Obstacle nor the message OutOfBounds. If the move is made, the wanderer is currently facing West. If the move cannot be made, there is no change to the wanderer.

To implement these methods several data members are necessary. The necessity of two data members is immediately obvious: the current location CurrPos and the current direction in which the wanderer is facing CurrDirection. To guarantee that the various move member functions are correctly implemented, the results of the last invocations of the corresponding look member functions must be known. These results can be maintained using bool objects OkNorth, OkEast, OkSouth, and OkWest. An Ok object has the value true only if the last invocation of the corresponding Look function did not receive either an Obstacle or an OutOfBounds message. As a newly constructed wanderer has not received any Look messages, a wanderer's Ok objects are initialized to false during construction.

Location implementation notes

- The Location constructor should use one of the overloaded member functions SetLocation() to accomplish its task.
- To accomplish their tasks, both inspectors GetRow() and GetColumn() simply return the value of the data member of the invoking object that maintains the characteristic of interest. For GetRow(), the interesting data member is Row; for GetColumn(), the interesting data member is Column. The mutator SetRow() needs only to update data member Row of the invoking object with the value of its parameter r to accomplish its task. Similarly, the mutator SetColumn() needs only to update data member Column of the invoking object with the value of its parameter c to accomplish its task.
- There are two overloaded Location mutators SetLocation(). One version of SetLocation() uses its integer parameters r and c to set the invoking object's data members Row and Column respectively. The other version of SetLocation() uses the data members of its Location parameter p to set the invoking object's data members Row and Column.
- The Location auxiliary operators == and != work as expected. The == operator examines its parameters p and q and determines whether the row values are the same and the column values are the same. If the parameters represent the same coordinate, the operator returns true, otherwise the operator returns false. The != operator examines its parameters p and q and determines whether they represent different coordinates. If the parameters represent different coordinates, the operator returns false, otherwise the operator returns true. The != operator can be implemented by taking the complement of the equality operation.
- The Location auxiliary stream operators << and >> have straightforward implementations. The insertion operator << inserts a left parenthesis, the row coordinate value of its Location parameter p, a comma and a space, the column coordinate of the of its Location parameter p, and a right parenthesis. The extraction operator >> first extracts two integer values r and c. The values are used to set p's row and column coordinates. Examine the file mydata.dat to see a sample input file of Location input values.

Wanderer implementation notes

- The Wanderer constructor expects two parameters—a location p and a direction d. Location p is used to set data member CurrPos which represents the current position of the wanderer. Direction d is used by Wanderer mutator SetDirection() to set the current direction of the wanderer. The data members OkNorth, OkEast, OkSouth, and OkWest represent whether it is safe for the wanderer to proceed in a particular direction. Initially there is no knowledge of the surroundings, so these members are set to false.
- To accomplish their tasks, both inspectors GetLocation() and GetDirection() simply return the value of the data member that maintains the characteristic of interest. For GetLocation(), the interesting data member is CurrPos; for GetDirection(), the interesting data member is CurrDirection. Similarly, mutator SetDirection() needs only to update data member CurrDirection with the value of its parameter d to accomplish its task.
- Each of the four Look member functions expect a Status value s as a parameter. The value s is a message to the wanderer regarding the nature of the adjacent location in question. For example, when function LookNorth() is invoked, it is given the status of the location to the immediate north of the wanderer's current position. If the status of that location is

neither `Obstacle` nor `OutOfBounds`, then the wanderer is currently permitted to step north. When this is the case, object `OkNorth` should be set true. If instead, the status of that location is either `Obstacle` or `OutOfBounds` then the wanderer is not currently permitted to step north. For this case, `OkNorth` should be set false. The other Look functions have the same expectation and can have comparable function bodies.

You are only required to define the `MoveNorth()` member function. We have implemented the other Move functions. With respect to the `Wanderer` Move functions, our implementation only defines the `MoveNorth()` member function. The other Move functions are left to the exercises. The `MoveNorth()` specification requires for a northern move that a `LookNorth()` invocation must have previously been made after the wanderer reached its current location. This is the case if `OkNorth` is true. If `OkNorth` is true, the wanderer is moved within the same column up one row. If instead `OkNorth` is false, then the wanderer does not move. If the wanderer is to be moved, then the following process will do it correctly: determine the current row position `r` of the wanderer; set the current row position of the wanderer to `r-1`; set the wanderer's current direction to north; and set all of the Ok objects to false so that the wanderer cannot move until it inspects its new surroundings.

General Notes

- The second maze snapshot on Page 1 was produced from a run of `goodmaze.exe`. The input/behavior of that run is shown below.

- When working on a project, the project file must be opened *before* the source file(s) are opened. Therefore, when working on this assignment, open `maze.ide` *before* opening `hw06.cpp`. To open an `ide` file, select the item `Open Project` under Borland's `Project` menu.
- You should run your program from a command window. The name of the program will be `maze.exe`. It will be placed by the project in `c:\localdata`.
- The project file for this assignment was developed for use in one of the on-grounds ITC computer laboratories such as the Catlin Laboratory (The Stacks). In particular, as required by the Borland compiler, the project file specifies the locations of local and standard libraries. Our specification is their location on the ITC servers. If you decide to work on this assignment elsewhere, please refer to the handout available in the documents section of the class web site.
- The program should follow course programming style guidelines. The guideline is available on the CS 101 website.
- A general grading criteria for this assignment will be posted prior to next week's lab.