Project 1: The Rat Pack

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Use ADT Queue to control breadth-first search
- Use bitwise operations to extract individual bits from words
- Set up a general maze solving system
- Write applications involving multiple interacting classes and objects

Operational Objectives: Design and implement a simulation that receives as input a rectangular 2-dimensional maze along with "begin" and "goal" locations in the maze and produces a shortest path through the maze from "begin" to "goal" using breadth-first search.

Deliverables: Six (6) files: `maze.h`, `maze.cpp`, `ratpack.cpp`, `makefile`, `mymaze`, and `log.txt`.

Maze Concepts

1. A *maze* is a rectangular array of square cells such that:
   a. The exterior perimeter of the rectangle is solid (impenetrable).
   b. Walls between cells may be designated as extant, meaning a barrier is in place, or non-extant, meaning passage between cells at that location is not restricted.
   c. Cells are numbered consequtively from left to right and top to bottom, beginning with cell number 0 in the upper left corner.
2. A *maze problem* consists of:
   a. A maze
   b. A designated "start" cell
   c. A designated "goal" cell
3. A *path* in a maze is a sequence of cell numbers such that adjacent cells in the sequence share a common face with no wall.
4. A *solution* to a maze problem is a path in the maze from the start cell to the goal cell.

Maze File Syntax

A maze file is designed to describe a maze problem as defined above. The file consists of unsigned integers that may be in any format in the file but are typically arranged to mimic the actual arrangement of cells in the maze problem. Your program should read maze data from an external file, assuming the following file format:

```
numrows numcols
(first row of codes)
(second row of codes)
...
(last row of codes)
start goal
[optional documentation or other data may be placed here]
```

We will refer to such a file as a *maze file*. (Note that the actual format is optional, for human readability. The file is syntactically correct as long as there are the correct number of (uncorrupted) unsigned integer entries.)

Maze File Semantics

A maze file is interpreted as follows:

1. `numrows` and `numcols` are the number of rows and columns, respectively, in the maze.

2. Each entry in a row of codes (after `numrows` and `numcols` but before `start` and `goal`) is a decimal integer code in the range [0..15]. The code is interpreted as giving the "walls" in the cell. We will refer to this as a *walls code*. A specific walls code is interpreted by looking at the binary representation of the code, with 1's bit = North wall, 2's bit = East wall, 4's bit = South wall, and 8's bit = West wall. (Interpret "up" as North when drawing pictures.) A bit value of `1` means the wall exists. A bit value of `0` means the wall does not exist. For example walls code 13 means a cell with North, South, and West walls but no East wall, because $(13)_{10} = 8 + 4 + 1 = (1101)_2$ (2's bit is unset).

3. Cells are numbered consequtively from left to right and top to bottom, beginning with cell 0 at the upper left. (Thus, a 4x6 maze has cells 0,1,2,...,23.) We refer to these as *cell numbers*. `start` and `goal` are cell numbers (NOT walls codes). They inform where the maze starting location is and where the goal lies.
4. The optional data following `goal` is treated as file documentation (i.e., ignored).

Procedural Requirements

1. The official development/testing/assessment environment is the environment on the `linprog` machines.
2. Work within your subdirectory `cop4530/proj1`. The usual COP 4530 rules apply, as explained in Chapter 1 of the Lecture Notes.
3. From the course library directory `LIB`, copy the following files into your project directory:

```
LIB/proj1/proj1submit.sh   # submit script
LIB/proj1/maze.startHere   # start for maze.cpp
LIB/proj1/ratpack.cpp      # maze client
LIB/proj1/mazetest.cpp     # maze client
LIB/proj1/maze*             # example maze files
LIB/proj1/badmaze*          # more maze files
```

4. Begin by copying `maze.startHere` to `maze.h`. Edit `maze.h` appropriately, at the very least by personalizing the file header.
5. Create the files `makefile`, `maze.cpp`, and `mymaze` adhering to the requirements and specifications below. NOTE: a makefile is distributed. It should not need modification.
6. Turn in files `maze.h`, `maze.cpp`, `ratpack.cpp`, `makefile`, `mymaze`, and `log.txt` using the submit script `submitscripts/proj1submit.sh`.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

Technical Requirements and Specifications

I. **Solution design.** Your solution should be based on the framework distributed in the file `proj1/maze.startHere`. This framework consists of two classes, `Maze` and `Cell`.

   A. Interface (`Maze` public member functions)

      1. `bool Initialize(char*)`
      2. `bool Consistent()`
      3. `void Solve()`
      4. `void ShowMaze(std::ostream&)`

   B. Internal representation

      1. A `Maze` object can hold one internal maze representation, read from a maze file. Class `Maze` is designed to be a singleton: only one `Maze` object exists in a given execution environment. Use of a copy constructor and assignment operator by client is prohibited. `Maze` data is as follows:

```
private: // variables
  unsigned int                                   numrows_, numcols_;
  Cell *                                         start_;
  Cell *                                         goal_;
  fsu::Vector < Cell >                           cellVector_; // cell inventory
  fsu::Queue  < Cell* , fsu::Deque < Cell* > >   conQueue_;   // control queue
```

      2. `numrows_` and `numcols_` store the dimensions of the maze.
      3. `cellVector_` is used to store actual `Cell` objects that make up a maze. `Cell` objects should not be copied.
      4. `start_` and `goal_` point to the start and goal cells (in the cell inventory).

5. conQueue_ is the principal control queue for the maze solver algorithm. Note that conQueue_ stores pointers to Cell objects, which reside in the cell inventory.

6. *No data members may be added to class Maze*.

7. Class Cell is a subsidiary class for Maze, tightly coupled by granting friend status to class Maze. A Cell object represents one square cell in a 2-dimensional maze, used by Maze to represent rectangular mazes of square cells. The constructors and assignment operator of Cell are public, to facilitate storing Cell objects in container classes. All other methods and data are private, hence accessible only by the friend class Maze. Cell data is as follows:

```
private:  // variables
  unsigned int         id_;
  bool                 visited_;
  Cell *               searchParent_;
  fsu::List < Cell* >  neighborList_;
```

8. id_ is the name (cell number) of the cell

9. neighborList_ is a list of (pointers to) cells that are adjacent to this cell in the maze

10. visited_ is a flag indicating whether a cell has been visited

11. searchParent_ is a pointer to the predecessor cell computed by BFS

12. *No data members may be added to class Cell*

C. Internal functionality

1. bool Maze::Initialize(char*)

   Reads data from a maze file and sets up the internal representation of the maze problem. The complete internal representation of a maze problem is established, including the maze dimensions, cell inventory (with neighborlists for each cell), and start/goal. Checks the maze file syntax while reading. Returns true when read is successful. The following should be checked as the file is read:

   i. walls codes in range [0..15]
   ii. start, goal in range of cells in maze
   iii. uncorrupted input of the correct number of numbers

   The outer maze boundary should be represented as solid, independent of walls codes for the boundary cells. (If a walls code for a boundary cell fails to specify a wall at a boundary face, optionally report this fact, but continue building the representation.)

2. bool Maze::Consistent() const

   Checks the internal representation of a maze (previously established by Initialize()), detecting and reporting situations where the representation is not self-consistent. Returns true when no inconsistencies are found. The following logic errors should be caught ([R] = required, [O] = optional):

   i. disagreement by adjacent cells on their common wall [R]
   ii. tunnelling (where a magic door opens from one cell to another non-adjacent cell) [O]
   iii. missing boundary wall [O]

   Note that tunnelling is impossible to represent in a maze file, however, it can result from incorrect internal handling - see badrat.x for an example. Note also that missing boundary walls should be corrected by Initialize(), but checking again here will catch errors in the implementation of Initialize().

3. void Maze::Solve(fsu::TList<unsigned int>& solution)

   Uses breadth-first-search (facilitated by conQueue) to find a solution to the internally represented maze problem. Once the problem is solved, the solution is placed in the list solution passed in by the client program.

4. `void Maze::ShowMaze(std::ostream&) const`
   Produces a graphic of the maze using the internal representation.

5. `void Cell::AddNeighbor (Cell * N)`
   Adds a (pointer to) a new neighbor cell of this cell.

6. `Cell* Cell::GetNextNeighbor () const`
   Returns (pointer to) this cell's next unvisited neighbor. Returns 0 if this has no unvisited neighbors.

7. `bool Cell::IsNeighbor (const Cell * N) const`
   Returns true if `N` is a neighbor of this cell.

8. Various data accessors and manipulators
   Recommended to be used for safe programming practice when implementing `Maze` methods.

II. **Required elements.**

1. No data members may be added to class `Maze` or class `Cell`.
2. Walls codes may not be stored in Maze or Cell objects. Walls codes may only be stored in local variables of method `Initialize()`.
3. Do not use recursion.
4. Private methods may be added to `Maze` as desired, but no methods should be added to `Cell`.
5. A minimum functioning solution consists of methods `Initialize()` and `Solve()` operating correctly on correct maze files. (This is the 80 point level.)
6. An advanced solution also implements method `Consistent()` and operates correctly on both correct maze files and improperly configured maze files. (This is the full 100 point level.)
7. Up to 20 **extra credit** project points may be earned for implementation of method `ShowMaze()`. *Note:* To get the full 20 points extra credit for `ShowMaze()`, you have to do something more creative than is illustrated in `ratpack.x` and `mazetest.x` [option 4]. If you do that exactly, you can get 10 extra credit points. A 20 point solution is illustrated in `mazetest.x` [option 5].
8. One maze file `mymaze` of your own creation is required. (That is, it should not be based on any distributed files and should not be obtained from another source). Your `mymaze` should be a (correctly configured) representation of a maze problem (1) with at least 5 rows and 6 columns, (2) with more than one solution, and (3) such that all solutions are at least 20 steps in length.
9. The client program distributed as `proj1/ratpack.cpp` should be modified *only by un-commenting the functionality you are implementing* and submitted as part of the project. Note that this is how you signal which functionalities you are submitting.
10. Review of deliverables:
    `maze.h` and `maze.cpp` define and implement classes `Maze` and `Cell`
    `ratpack.cpp` showcases the functionality you implement for `Maze`
    `makefile` creates an executable called `ratpack.x` using the above three files
    `mymaze` is a maze file of your own creation
    `log.txt` work and testing log

Hints:

- Begin the project by creating a few maze files. Recommended practice: put an ascii graphic of the maze in the maze file, following the maze file data. Use `LIB/area51/ratpack.x` or `LIB/area51/mazetest.x` to check out your maze files.
- Recommended order of implementation of methods:

  1. `Cell::` member functions
  2. `Maze::Initialize()`

3. `Maze::Consistent()`
4. `Maze::Solve()`
5. `Maze::ShowMaze()`

Comment out all of the calls to `Maze::` methods in `mazetest.cpp`, then uncomment them one at a time to test as you implement methods. This way you will have a debugged internal representation before you try to implement the BFS solving algorithm.

- Think of the cell vector as a master inventory of cells. All other references to cells point into this inventory (rather than making copies of cells). "Point" with *pointers*, not indices.

- Assuming that `number, row, col` are `unsigned int`, the number of a cell is obtainable from its `row, col` indices as
  ```
      number = row * numcols + col;
  ```
  Similarly, if you know the cell number then you can find it's `row, col` indices from the cell number as
  ```
      row = number / numcols; col = number % numcols;
  ```
  Now ask yourself: what are the numbers of the neighbors to the North, East, South, and West?

- A `walls` code of a cell is easily converted to see which neighbor cells are accessible. For example, to see whether `cellVector[number]` has an accessible neighbor to the West, do something like

  ```
      if ((0 < col) && ((walls & 0x08) == 0))
      {//add the West neighbor to the neighbor list}
  ```

- Sample executables `ratpack.x` and `mazetest.x` will be available in area51.

- You may also enjoy the defective version area51/badpack.x.

- *Dont forget:* You will be graded on what you submit that works. If your submission doesn't compile, or if you don't compile a particular functionality, you will be graded accordingly.