CS2604 Fall 2001
PROGRAMMING ASSIGNMENT #4: Maze Generator

Due Wednesday, December 5 @ 11:00 PM for 125 points
Early bonus date: Tuesday, December 4 @ 11:00 PM for 13 point bonus
Late date: Thursday, December 6 at 11:00 PM with a 13 point penalty

Assignment:
The goal of this project is to write a program that will automatically generate and solve mazes. Each time the program is run, it will generate and print a new random maze and solution. You will use the disjoint set (*union-find*) data structure, depth-first search (*DFS*), and breadth-first search (*BFS*). See Sections 6.2 and 11.3 in the textbook.

Generating a Maze:
Conceptually, to generate a maze, first start with a grid of rooms with walls between them. The grid contains *r* rows and *c* columns for a total of *r*c* rooms. For example, Figure 1 is a 4x4 grid of 16 rooms. The missing walls at the top left and bottom right of the maze border indicate the start and finish rooms of the maze.

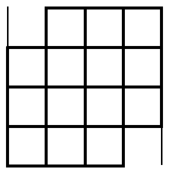
Figure 1

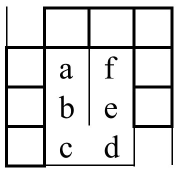

Figure 2

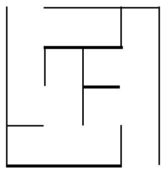
Figure 3

Now, begin removing interior walls to connect adjacent rooms. The difficultly in generating a good maze is in choosing which walls to remove. Walls should be removed to achieve the following maze characteristics:

1. *Randomized*: To generate unpredictable different mazes, walls should be selected randomly as candidates for removal. For randomization, you must use the `rand()` function to generate random numbers, and `srand()` to initially seed the random number generator with a unique value, e.g. `srand((unsigned)time(NULL))`.
2. *Single solution*: There should be only one path between the start room and the finish room. Unnecessarily removing too many walls will make the maze too easy to solve. Therefore, a wall should not be removed if the two rooms on either side of the wall are already connected by some other path. For example, in Figure 2, the wall between a and f should not be removed because walls have previously been removed that create a path between a and f through b,c,d,e. Use the **disjoint set data structure** to determine room connected-ness.
3. *Fully connected*: Enough walls must be removed so that every room is reachable from the start room. There must be no rooms or areas that are completely blocked off from the rest of the maze. Figure 3 shows an example generated maze.

Solving the Maze:
After generating a maze, your program should then solve the maze first using **DFS** and then again using **BFS**. Each search algorithm will begin at the start room and search for the finish room by traversing wall openings. The search should terminate as soon as the finish room is

found. For each search algorithm, you will output the order in which rooms where visited and indicate the shortest solution path from start to finish.

Design:
You will likely represent the maze as a graph data structure, where rooms are nodes and removed walls are edges between nodes. Since the size of the graph is known at startup, a 2 dimensional array-based implementation that mimics the grid structure may work well. To randomly select walls for removal, you will also need to maintain a separate list of walls eligible for removal. As randomly selected walls are removed from the maze or determined to be ineligible for removal (because of rule 2 above), they are eliminated from the wall list. This places a tight upper bound on the number of iterations of the wall-removal loop.

You must use the **disjoint set** data structure for the union and find operations on rooms when generating the maze. It is required that you implement the **weighted union rule** and the **path compression** technique for maximum efficiency. Rooms that are connected by some path are in the same set. The find operation reveals whether two rooms are already connected by some path. When removing a wall, the union operation is used to join the two sets together. The maze generator is done when there is only one set left, indicating that all rooms are connected. The disjoint set data structure enables efficient processing of the union and find operations, so that maze generation is fast.

Input:
The program should be named 'maze' and should accept the number of rows r and columns c of the maze as command-line arguments. If no command line arguments are given, it should default to 20 rows by 20 columns. The following invocation would create a maze that is 10 rows by 20 columns:

```
% maze 10 20
```

Output:
The program should print to standard-out the maze, then the DFS solution, then the BFS solution. The maze is printed in ASCII using the vertical bar '|' and dash '-' characters to represent walls, '+' for corners, and space character for rooms and removed walls. The start and finish rooms should have exterior openings as shown.

For the DFS and BFS solutions, the maze should be output twice for each algorithm. The first maze output for each algorithm should show the order that the rooms were visited by the algorithm. The maze should be printed exactly as above except that rooms should be printed with the low-order digit of the visitation order number. The start room is '0'. Unvisited rooms should remain a space character. The second maze output for each algorithm should show the shortest solution path, using hash '#' character for rooms and wall openings on the solution path.

You will need to view the output in a fixed-width font. The program should output "<pre>" as the first line. This will enable you to view your mazes correctly in a web browser by directing standard output to a file on the command line:

```
% maze 10 20 > mymaze.html
```

Following is sample output for the maze in Figure 3:

```
<pre>
+ +-+-+-+
|     | |
+ +-+-+ +
| | |   |
+ + +-+ +
|   |   |
+-+ + + +
|     | |
+-+-+-+ +

DFS:
+ +-+-+-+
|0 1 2| |
+ +-+-+ +
|3| |   |
+ + +-+ +
|4 5|8 9|
+-+ + + +
|  6 7|0|
+-+-+-+ +

+ +-+-+-+
|#    | |
+#+-+-+ +
|#| |   |
+#+ +-+ +
|###|###|
+-+#+#+#+
|  ###|#|
+-+-+-+ +

BFS:
+ +-+-+-+
|0 1 3| |
+ +-+-+ +
|2|7|   |
+ + +-+ +
|4 5|0 1|
+-+ + + +
|9 6 8|2|
+-+-+-+ +

+ +-+-+-+
|#    | |
+#+-+-+ +
|#| |   |
+#+ +-+ +
|###|###|
+-+#+#+#+
|  ###|#|
+-+-+-+ +
```

Programming Standards:

You must conform to good programming/documentation standards, as described in the Elements of Programming Style. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation, unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point. You may not use code from STL, MFC, or a similar library in your program.

Testing:

Sample data files will be posted to the website to help you test your program. This are not the data files that will be used in grading your program. While the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

You will submit this project electronically. In particular, you will create a zip'ed archive file containing the following items (and nothing else):

- all source code files necessary to build an executable
- either the project workspace files (.dsw and .dsp) for Visual C++ users, or a makefile for g++ users.

Windows users should be sure to use a modern zip tool which preserves long file names. A suitable freeware command-line zip tool will be posted on the course website. UNIX users should submit a gzip'ed and tar'ed file.

Once you have assembled the archive file for submission, for your own protection, please move it to a location other than your development directory, unzip the contents, build an executable, and test that executable on at least one input file. Failure to do this may result in delayed evaluation of your program, and a loss of points.

You will submit your project to the automated Curator server. The instructions and necessary software are available at: `http://ei.cs.vt.edu/~eags/CuratorGuides.html`. If you make multiple submissions, only your last submission will be evaluated.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement in the header comment preceding the function main() in your program. The text of the pledge will also be posted online.

```
//  On my honor:
//
//  - I have not used C++ language code obtained from another student,
//    or any other unauthorized source, either modified or unmodified.
//
//  - All C++ language code and documentation used in my program
//    is either my original work, or was derived, by me, from the source
//    code published in the textbook for this course.
//
//  - I have not discussed coding details about this project with anyone
//    other than my instructor, ACM/UPE tutors or the GTAs assigned to this
//    course. I understand that I may discuss the concepts of this program
//    with other students, and that another student may help me debug my
//    program so long as neither of us writes anything during the discussion
//    or modifies any computer file during the discussion.  I have violated
//    neither the spirit nor letter of this restriction.
//
```

Programs that do not contain this pledge will not be graded.