```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file main.cpp
* @assignment 8: Breadth-First Search
*/

/**
* This program reads a maze from STDIN (see assignment '21-6' for more info).
* The shortest path from the start to the exit of the maze is found and sent to
* STDOUT. This program was tested with mazes of varying sizes that were visually
* inspected with Bergamini's maze viewer. The program was also checked with
* Valgrind for memory leaks. No errors were found and the program appears to be
* working to all specs.
*/

#include <iostream>
#include <sstream>
#include <vector>

#include "maze.h"

int main() {
   // Load input
   std::string buffer;
   std::vector<int> walls;
   while (std::cin >> buffer) {
      for (size_t i = 0; i < buffer.size(); i++) {
         int wallValue = std::stoi(buffer.substr(i, 1), nullptr, 16);
         walls.push_back(wallValue);
      }
   }

   // Generate path
   Maze maze(walls);
   maze.BreadthFirstSearch();

   // Output path
   std::cout << maze;
}
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file maze.cpp
* @assignment 8: Breadth-First Search
*/

#include "maze.h"

Maze::Maze(const std::vector<int>& walls)
   : m_walls(walls), m_numWalls(walls.size()),
   m_sideLength(std::sqrt(walls.size())), m_exit(walls.size() - 1) {}

void Maze::BreadthFirstSearch() {
   // Initialize vertex info
   m_candidate.assign(m_numWalls, true);
   m_parent.assign(m_numWalls, m_null);

   // Initalize maze exit and place into queue
   m_candidate[m_exit] = false;
   m_activeCandidates.push(m_exit);

   // Breadth-First Search Algorithim
   while (!m_activeCandidates.empty()) {
      auto vertex = m_activeCandidates.front();
      m_activeCandidates.pop();
      findAdjacents(vertex);
      for (auto adjacent : m_adjacents) {
         if (m_candidate[adjacent]) {
            m_candidate[adjacent] = false;
            m_parent[adjacent] = vertex;
            m_activeCandidates.push(adjacent);
         }
      }
   }
}

void Maze::findAdjacents(int vertex) {
   m_adjacents.clear(); // Clears past adjacency candidates

   if (m_connectsLeft(vertex)) { // Path to left adjacent tile
      m_adjacents.push_back(m_toLeft(vertex));
   }
   if (m_connectsRight(vertex)) { // Path to right adjacent tile
      m_adjacents.push_back(m_toRight(vertex));
   }
   if (m_connectsUp(vertex)) { // Path to adjacent tile above
      m_adjacents.push_back(m_toUp(vertex));
   }
   if (m_connectsDown(vertex)) { // Path to adjacent tile below
      m_adjacents.push_back(m_toDown(vertex));
   }
}

std::ostream& operator<<(std::ostream& os, Maze& maze) {
   auto node = maze.m_entry; // Starting at entry of maze
   while (node != maze.m_null) {
      int y = node / maze.m_sideLength;
      int x = node - (y * maze.m_sideLength);
      os << "(" << x << ", " << y << ")" << std::endl;
      node = maze.m_parent[node];
   }
   return os;
}
```



```
VERSION = -std=c++20

p8: main.o maze.o
	g++ $(VERSION) -o p8 main.o maze.o

main.o: main.cpp
	g++ $(VERSION) -c main.cpp

maze.o: maze.cpp maze.h
	g++ $(VERSION) -c maze.cpp

clean:
	rm -f p8 *.o *~
```



```
bool Maze::m_connectsLeft(int vertex) {
   // Ensures adjacent tile is in bounds
   if (vertex % m_sideLength == 0) {
      return false;
   }

   // Ensures tile has path left
   if ( ( m_walls[vertex] & m_leftWall ) == m_leftWall) {
      return false;
   }

   return true;
}

bool Maze::m_connectsRight(int vertex) {
   // Ensures adjacent tile is in bounds
   if ( (vertex + 1) % m_sideLength == 0) {
      return false;
   }

   // Ensures tile has path right
   if ( ( m_walls[vertex] & m_rightWall ) == m_rightWall) {
      return false;
   }

   return true;
}
bool Maze::m_connectsDown(int vertex) {
   // Ensures adjacent tile is in bounds
   if (vertex + m_sideLength >= m_numWalls) {
      return false;
   }

   // Ensures tile has path down
   if ( ( m_walls[vertex] & m_downWall ) == m_downWall) {
      return false;
   }

   return true;
}

bool Maze::m_connectsUp(int vertex) {
   // Ensures adjacent tile is in bounds
   if (vertex - m_sideLength < 0) {
      return false;
   }

   // Ensures path up
   if ( ( m_walls[vertex] & m_upWall ) == m_upWall) {
      return false;
   }

   return true;
}
```

```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file maze.h
* @assignment 8: Breadth-First Search
*/

#pragma once

#include <vector>
#include <ostream>
#include <queue>
#include <cmath>

/**
* 'Maze' reads in a maze and uses a breadth-first search algorithim to solve the
* shortest path from start to finish; the interface allows the user to view the
* solution.
*/
class Maze {

public:

   /**
   * 'Maze' constuctor.
   * @param 'walls' vector containing wall values for all tiles
   */
   Maze(const std::vector<int>& walls);

   /**
   * Outputs 'Maze' solution.
   * @param 'os' ostream reference to insert maze solution into
   * @param 'maze' maze containing solution
   */
   friend std::ostream& operator<<(std::ostream& os, Maze& maze);

   /**
   * Executes breadth first search algorithim.
   */
   void BreadthFirstSearch();

private:

   /**
   * Finds adjacent tiles that have a path between them and places them into
   * 'm_adjacents'.
   * @param 'vertex' index of tile
   */
   void findAdjacents(int vertex);

   /**
   * Determines if there is a path to tile directly to the right.
   * @param 'vertex' index of tile
   * @return true if path to tile directly right, else false
   */
   bool m_connectsRight(int vertex);

   /**
   * Determines if there is a path to tile directly above.
   * @param 'vertex' index of tile
   * @return true if path to tile directly above, else false
   */
   bool m_connectsUp(int vertex);

   /**
```



```
   * Determines if there is a path to tile directly below.
   * @param 'vertex' index of tile
   * @return true if path to tile directly below, else false
   */
   bool m_connectsDown(int vertex);

   /**
   * Determines if there is a path to tile directly left
   * @param 'vertex' index od tile
   * @return true if path to tile directly below, else false
   */
   bool m_connectsLeft(int vertex);

   /**
   * Calculates index of tile directly to right.
   */
   int m_toRight(int vertex) { return vertex + 1; }

   /**
   * Calculates index of tile directly to left.
   */
   int m_toLeft(int vertex) { return vertex - 1; }

   /**
   * Calculates index of tile directly above.
   */
   int m_toUp(int vertex) { return vertex - m_sideLength; }

   /**
   * Calculates index of tile directly below.
   */
   int m_toDown(int vertex) { return vertex + m_sideLength; }

   std::vector<int> m_walls; // The values of all walls

   std::vector<int> m_adjacents; // Vector of adjacents walls (see findAdjacents)

   std::vector<bool> m_candidate; // Whether a tile has been discovered or not

   std::vector<int> m_parent; // The parent index of a tile

   std::queue<int> m_activeCandidates; // Tiles who will be explored with BFS

   const int m_null = -1; // Flag value for 'm_parent' when no parent exists

   const int m_exit; // Index of tile that exits maze

   const int m_entry = 0; // Index of tile that enters maze

   const int m_rightWall = 0x1; // Value of only a right wall

   const int m_leftWall = 0x4; // Value of only a left wall

   const int m_upWall = 0x8; // Value of only a wall above

   const int m_downWall = 0x2; // Value of only a wall below

   int m_numWalls; // The total number of tiles (entries in maze matrice)

   int m_sideLength; // Side length of maze matrice
};
```