```

// Analyzer.h

#ifndef ANALYZER_H
#define ANALYZER_H

#include "field.h"

typedef std::set< std::pair< int, int > > MineSet;

// For analyzing Minesweeper playing fields. Utilizes contradiction
// testing to determine the state of unknown squares as either safe
// or a mine.
class Analyzer
{
  protected:
    MSfieldPart1 & field; // Reference to the actual field.
    MSfieldPart1 temp;    // Copy of the field to work on.
  public:
    Analyzer(MSfieldPart1 & field) : field(field), temp(field) {}
    bool IsMine(int x,int y);
    bool IsSafe(int x, int y);
    void UseRule1(int x, int y);
    void UseRule2(int x, int y);
    void UseRule3(int x, int y);
    bool HasContradiction(int x, int y);
    void ProcessField();
    bool CheckField();
};

#endif
```

```

// Analyzer.cpp

#include "analyzer.h"
#include <algorithm>

void GetDifference(MineSet& xset, MineSet& yset, MineSet& diffset);

//-------//
// Tests //
//-------//

// Check to see if this space is a mine using contradiction.
bool Analyzer::IsMine(int x,int y)
{
  // Copy the field for my own use.
   temp = field;

  // Assume the opposite case, the space is safe.
  temp.MarkAsSafe(x, y);

  // Process the field.
  ProcessField();

  // Check for contradictions.
   return CheckField();
}

// Check to see if this space is safe using contradiction.
bool Analyzer::IsSafe(int x, int y)
```

```
{
  // Copy the field for my own use.
  temp = field;

   // Assume the opposite case, the space is a mine.
  temp.MarkAsMine(x, y);

  // Process the field.
  ProcessField();

  // Check for contradictions.
   return CheckField();
}

//-------//
// Rules //
//-------//
void Analyzer::UseRule1(int x, int y)
{
  // If neighboring mines remaining is equal to the number of unknown neighbors..
  int MinesLeft = temp.GetMineCount(x, y) - temp.KnownMines(x, y);
  if (MinesLeft == (int)temp.UnKnownLocations(x, y).size())
  {
    // Then all unknown neighbors are mines.
    MineSet n = temp.UnKnownLocations(x, y);
    MineSet::iterator cur = n.begin();
    MineSet::iterator end = n.end();

    for (; cur != end; ++cur)
    {
      temp.MarkAsMine(cur->first, cur->second);
    }
  }
}

void Analyzer::UseRule2(int x, int y)
{
  // If known mines is equal to neighboring mines..
  if (temp.GetMineCount(x, y) == temp.KnownMines(x, y))
  {
    // Then all unknown neighbors are safe.
    MineSet n = temp.UnKnownLocations(x, y);
    MineSet::iterator cur = n.begin();
    MineSet::iterator end = n.end();

    for (; cur != end; ++cur)
    {
      temp.MarkAsSafe(cur->first, cur->second);
    }
  }
}

void Analyzer::UseRule3(int x, int y)
{
  // Check all squares within two of this square.
  int MinesLeftCurrent, MinesLeftNeighbor;
  int x_end = std::min(x+2, temp.GetMaxX());
  int y_end = std::min(y+2, temp.GetMaxY());
  for (int x_cur = std::max(x-2, 0); x_cur < x_end; ++x_cur)
  {
    for (int y_cur = std::max(y-2, 0); y_cur < y_end; ++y_cur)
    {
      // Skip if this is the cell we're checking.
      if (x_cur == x && y_cur == y) continue;

```

```
96        // Skip if this isn't an open square.
97        if (!temp.IsHasValue(x_cur, y_cur)) continue;
98
99        // Find the difference of the two sets of unknown cells.
100       MineSet diff;
101       MineSet setx = temp.UnKnownLocations(x, y);
102       MineSet sety = temp.UnKnownLocations(x_cur, y_cur);
103       GetDifference(setx, sety, diff);
104
105       // If there are as many unknown cells in the difference of the two sets
106       // as there are mines left for the current cell minus mines left for the
107       // neighbor cell..
108       MinesLeftCurrent = temp.GetMineCount(x, y) - temp.KnownMines(x, y);
109       MinesLeftNeighbor = temp.GetMineCount(x_cur, y_cur) - temp.KnownMines(x_cur, y_cur);
110       if ((MinesLeftCurrent - MinesLeftNeighbor) == (int)diff.size())
111       {
112         // Then all cells in the differences are mines.
113         MineSet::iterator cur = diff.begin();
114         MineSet::iterator end = diff.end();
115
116         for (; cur != end; ++cur)
117         {
118           temp.MarkAsMine(cur->first, cur->second);
119         }
120
121         diff.clear();
122         GetDifference(sety, setx, diff);
123
124         cur = diff.begin();
125         end = diff.end();
126
127         for (; cur != end; ++cur)
128         {
129           temp.MarkAsSafe(cur->first, cur->second);
130         }
131       }
132     }
133   }
134 }
135
136 //-----------//
137 // Analyzer //
138 //-----------//
139 // Check the given cell for the two contradictions.
140 bool Analyzer::HasContradiction(int x, int y)
141 {
142   // First contradiction: Too many known mines.
143   if (temp.KnownMines(x, y) > temp.GetMineCount(x, y))
144   {
145     return true;
146   }
147
148   // Second contradiction: Not enough mines will be found.
149   if (temp.KnownMines(x, y) + (int)temp.UnKnownLocations(x, y).size() < temp.GetMineCount(↩
          x, y))
150   {
151     return true;
152   }
153
154   // No contradiction found.
155   return false;
156 }
157
158 // Processes each space on the field with rules 1-3.
159 void Analyzer::ProcessField()
```

```
{
  // Process each space on the field with rule 1-3.
  int x_end = temp.GetMaxX();
  int y_end = temp.GetMaxY();
  int UnassignedSpaces = x_end * y_end;

  while (UnassignedSpaces > 0)
  {
    for (int x_cur = 0; x_cur < x_end; ++x_cur)
    {
      for (int y_cur = 0; y_cur < y_end; ++y_cur)
      {
        // Skip if this is not an open space.
        if (!temp.IsHasValue(x_cur, y_cur))
        {
          --UnassignedSpaces;
          continue;
        }

        UseRule1(x_cur, y_cur);
        UseRule2(x_cur, y_cur);
        UseRule3(x_cur, y_cur);
      }
    }
  }
}

bool Analyzer::CheckField()
{
  // Check each space for contradictions.
  int x_end = temp.GetMaxX();
  int y_end = temp.GetMaxY();
  for (int x_cur = 0; x_cur < x_end; ++x_cur)
  {
    for (int y_cur = 0; y_cur < y_end; ++y_cur)
    {
      // Skip if this is not an open space.
      if (!temp.IsHasValue(x_cur, y_cur)) continue;

      // If there's a contradiction, return true.
      if (HasContradiction(x_cur, y_cur))
        return true;
    }
  }

  return false;
}

//----------//
// Helpers //
//----------//
// Helper for finding the difference of two sets of unknown cells.
void GetDifference(MineSet& xset, MineSet& yset, MineSet& diffset)
{
  MineSet::iterator cur = xset.begin();
  MineSet::iterator end = xset.end();

  for (; cur != end; ++cur)
  {
    if (yset.find(*cur) == yset.end())
    {
      diffset.insert(*cur);
    }
  }
}
```