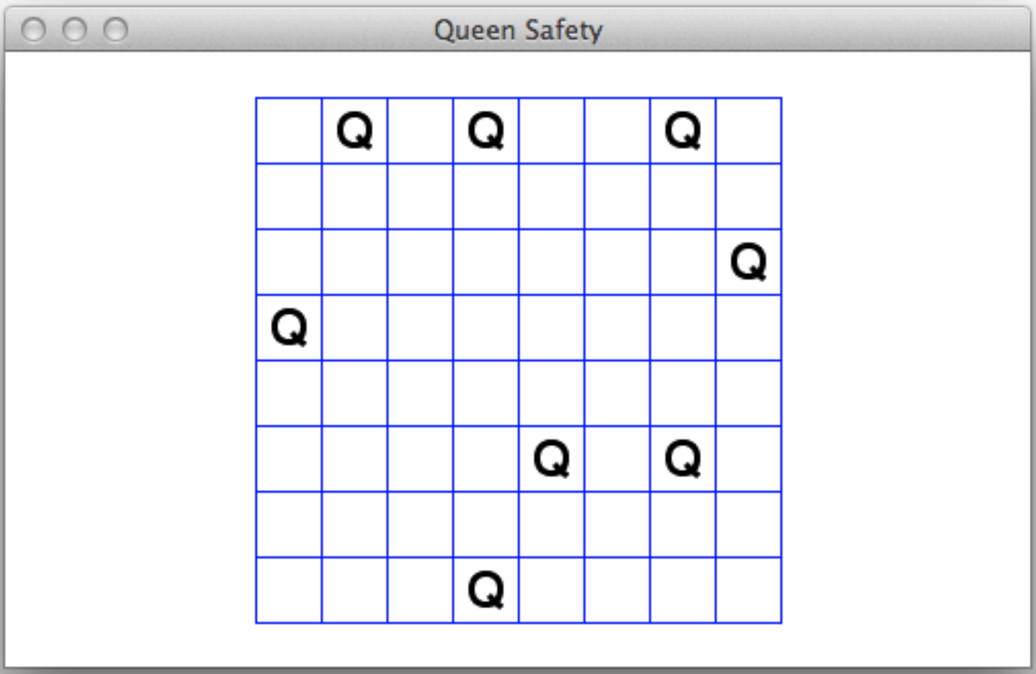
Grids and Queen Safety

Handout written by Jerry Cain.

Today's larger example demonstrates the use of a grid—that is, a single declaration of a **`Grid<bool>`**—to maintain information about a chessboard and queen placement. All algorithmically relevant code is presented in this handout.

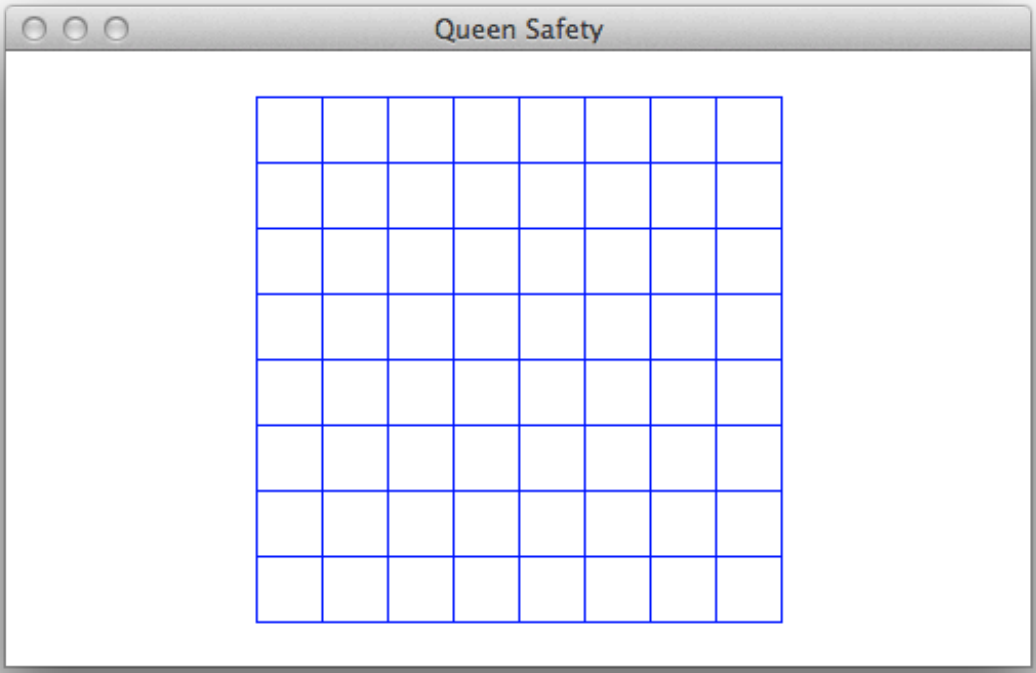


We want to use a two-dimensional **`Grid`** template to maintain information about all the locations on a chessboard. Because our particular example deals with the presence or absence of a queen, we only need to store **`true`** or **`false`** at each location: **`true`** means there's a queen, and **`false`** means there isn't. Ultimately, we are going to be interested in looking at an arbitrary position of the board to determine whether any queens placed elsewhere can attack.

The following function **`clearBoard`** takes a **`Grid<bool>`** by reference and initializes every entry of the grid to be **`false`**. Note the use of a double-**`for`** loop to generate every pair of **`row`** and **`col`** that corresponds to a legal coordinate.

*Over the course of the course, skim Chapters 1 through 4 just enough to know where to look when you need to teach yourself something (strings, files) that you've already seen in other programming languages. Once you've done that, read through Section 5.1 for information on the **`Vector`** and **`Grid`** classes.*

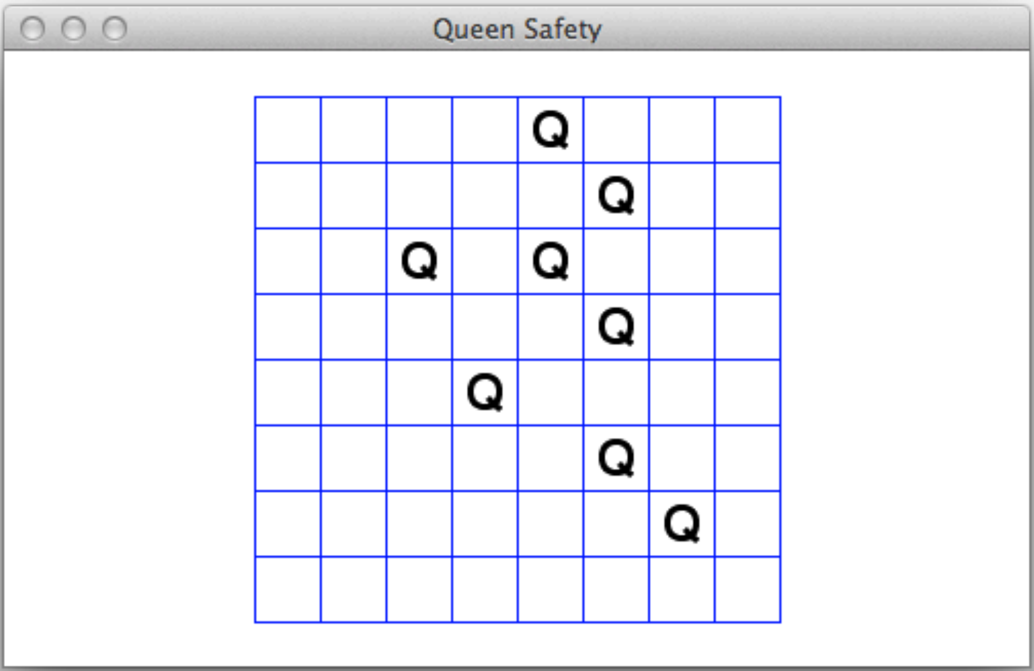
```
static void clearBoard(Grid<bool>& board) {
   for (int row = 0; row < board.numRows(); row++) {
      for (int col = 0; col < board.numCols(); col++) {
         board[row][col] = false;
         drawSquare(row, col, "Blue"); // assume a square is drawn at (row, col)
      }
   }
}
```



Next, we want a function **`placeRandomQueens`** that places some number of queens on the board at random locations. We enter the function with an idea of exactly how many queens need to be placed, and then repeatedly generate random coordinates on the board and place queens there. Note the care **`placeRandomQueens`** takes to assign at most one queen to each location—otherwise, the **`if (!board[row][col])`** wouldn't be necessary.

```
static void placeRandomQueens(Grid<bool>& board, int numQueensToPlace) {
   int numQueensPlaced = 0;
   while (numQueensPlaced < numQueensToPlace) {
      int row = randomInteger(0, board.numRows() - 1);
      int col = randomInteger(0, board.numCols() - 1);
      if (!board[row][col]) {
         board[row][col] = true;
         markLocation("Q", row, col, "Black"); // assume a Q is drawn
         numQueensPlaced++;
      }
   }
}
```

If we are dealing with an 8 x 8 board, then a call to **`placeRandomQueens(board, 8)`** might produce the following:



At this point, we want to determine which of the unoccupied squares can be attacked. Pretending for the moment that some **`isSafe`** predicate function already exists, we can easily label each of the empty locations as safe or not using the following code snippet:

```
static void identifySafeLocations(Grid<bool>& board) {
   for (int row = 0; row < board.numRows(); row++) {
      for (int col = 0; col < board.numCols(); col++) {
         if (!board[row][col]) {
            if (isSafe(board, row, col)) {
               markLocation("S", row, col, "Green");
            } else {
               markLocation("X", row, col, "Red");
            }
         }
      }
   }
}
```

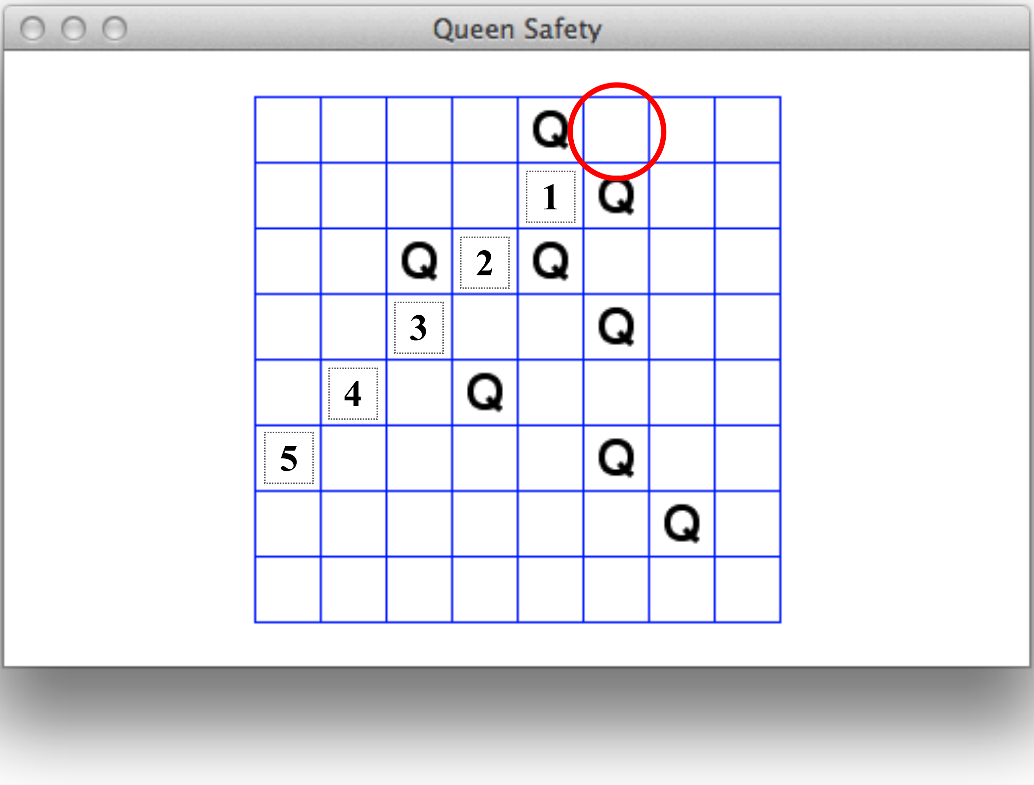
To determine whether or not a particular location is safe from attack, we need to search in all eight major and semimajor compass directions to see if a queen is visible. At first glance, we might be interested in functions like **`isSouthWestSafe`**, **`isSouthSafe`**, **`isSouthEastSafe`**, etc. and then implement **`isSafe`** to return the conjunction of all of them.

```
/**
 * Predicate Function: isSafe
 * --------------------------
 * isSafe returns true if and only if no queen
 * can be seen in any of the eight directions stemming radially
 * outward from the specified row and column.
 */
static bool isSafe(Grid<bool>& board, int row, int col) {
   return (isSouthWestSafe(board, row, col) &&
           isSouthSafe(board, row, col) &&
           isSouthEastSafe(board, row, col) &&
           isWestSafe(board, row, col) &&
           isEastSafe(board, row, col) &&
           isNorthWestSafe(board, row, col) &&
           isNorthSafe(board, row, col) &&
           isNorthEastSafe(board, row, col));
}
```

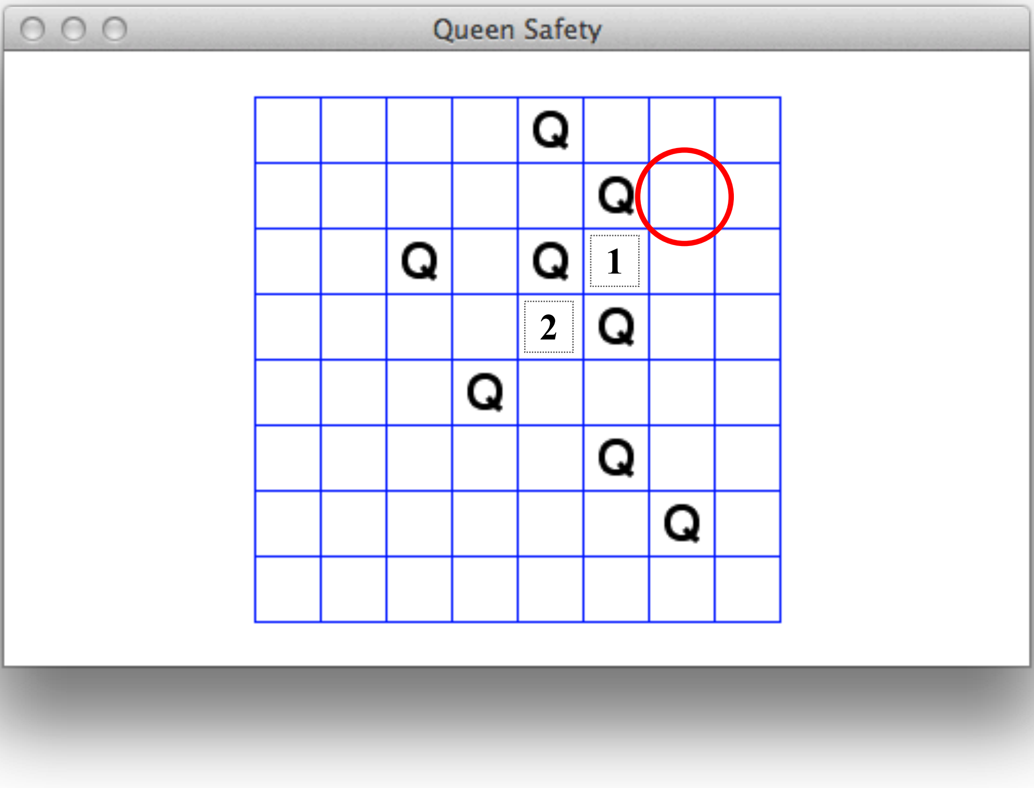
However, doing so requires the implementation of eight distinct helper functions, all of which are basically the same code block with a few trivial differences. Just compare two of them.

```
/**
 * Function: isSouthWestSafe
 * -------------------------
 * isSouthWestSafe decides whether or not any danger can be seen
 * by looking from the specified (row, col) coordinate in the
 * southwest direction.  Assuming the origin (0,0) coincides
 * with the upper left corner of the board, isSouthWestSafe must examine
 * the coordinates (row + 1, col - 1), (row + 2, col - 2), etc, in that
 * order, until either the edge of the board or a queen is encountered.
 */
static bool isSouthWestSafe(Grid<bool>& board, int row, int col) {
   row++;
   col--;
   while (board.inBounds(row, col) && !board[row][col]) {
      row++;
      col--;
   }

   return !board.inBounds(row, col);
}

/**
 * Function: isSouthSafe
 * ---------------------
 * isSouthSafe decides whether or not any danger can be seen
 * by looking from the specified (row, col) coordinate in the
 * southern direction.  Assuming the origin (0,0) coincides
 * with the upper left corner of the board, isSouthSafe must examine
 * the coordinates (row + 1, col), (row + 2, col), etc, in that
 * order, until either the edge of the board or a queen is encountered.
 */
static bool isSouthSafe(Grid<bool>& board, int row, int col) {
   row++;
   while (board.inBounds(row, col) && !board[row][col]) {
      row++;
   }

   return !board.inBounds(row, col);
}
```

To illustrate, consider the call **`isSouthWestSafe(board, 0, 5)`**, where the distribution of **`true`** and **`false`** in the **`board`** array corresponds to the placement of queens in the graphic presented below. Notice that the specified location is circled, and that each of the locations positioned southwest of it are labeled by the order **`isSouthWestSafe`** would visit them. The **`isSouthWestSafe`** function would examine all squares along that direction until it hits the edge of the board, or until it references one with a queen. For this call, we expect the algorithm to return **`true`**.



However, a call to **`isSouthWestSafe(board, 1, 6)`** would return **`false`**, because the **`(4, 3)`** location of the board is occupied by a queen. Check out the following graphic to see:



My major problem with this implementation is that all eight directional checks currently have their own function, but all are really doing the same thing. The only difference between each is the manner in which we update the **`row`** and **`col`** variables to move in a particular direction:

`SouthWest`:	**`row`** is incremented, and **`col`** is decremented with each move
`South`:	**`row`** is incremented with each move, and **`col`** remains the same
`SouthEast`:	**`row`** and **`col`** are incremented with each move
`West`:	**`row`** remains the same, but **`col`** is decremented with each move
`East`:	**`row`** remains the same, but **`col`** is incremented with each move
`NorthWest`:	**`row`** and **`col`** are decremented with each move
`North`:	**`row`** is decremented with each move, and **`col`** remains the same
`NorthEast`:	**`row`** is decremented, and **`col`** gets incremented with each move

We should work to unify the implementations of all eight predicate functions into a single one. That way we refine and debug all logic in a single place instead of many.

Check this out:

```
/**
 * Function: isDirectionSafe
 * -------------------------
 * isDirectionSafe decides whether or not any danger can be seen
 * by looking from the specified (row, col) coordinate in a particular
 * direction--specified by arguments four and five in the form of exactly
 * what values should be added to (row, col) to move in a specified
 * direction.
 *
 * Assuming the origin (0,0) overlays the upper left corner of
 * then board, isDirectionSafe must examine the coordinates
 * (row + drow, col + dcol), (row + 2 * drow, col + 2 * dcol), etc,
 * in that order, until either the edge of the board or a queen
 * is encountered.
 */
static bool isDirectionSafe(Grid<bool>& board, int row, int col,
                            int drow, int dcol) {
   if (drow == 0 && dcol == 0) return true;
   row += drow;
   col += dcol;
   while (board.inBounds(row, col) && !board[row][col]) {
      row += drow;
      col += dcol;
   }

   return !board.inBounds(row, col);
}
```

Need to search northwest from index **`(6, 5)`**?

Call **`isDirectionSafe(board, 6, 5, -1, -1)`**.

Need to search south from index **`(6, 5)`**?

Call **`isDirectionSafe(board, 6, 5, 1, 0)`**.

Need to search east from the origin?

Call **`isDirectionSafe(board, 0, 0, 0, 1)`**.

This allows us to unify common functionality to a single helper function, not eight of them, and it also makes the implementation of the **`isSafe`** function (the one that checks all eight directions, not just one) more compact.

```
static bool isSafe(Grid<bool>& board, int row, int col) {
   for (int drow  = -1; drow  <= 1; drow++) {
      for (int dcol = -1; dcol <= 1; dcol++) {
         if (!isDirectionSafe(board, row, col, drow, dcol))
            return false;
      }
   }
   return true;
}
```