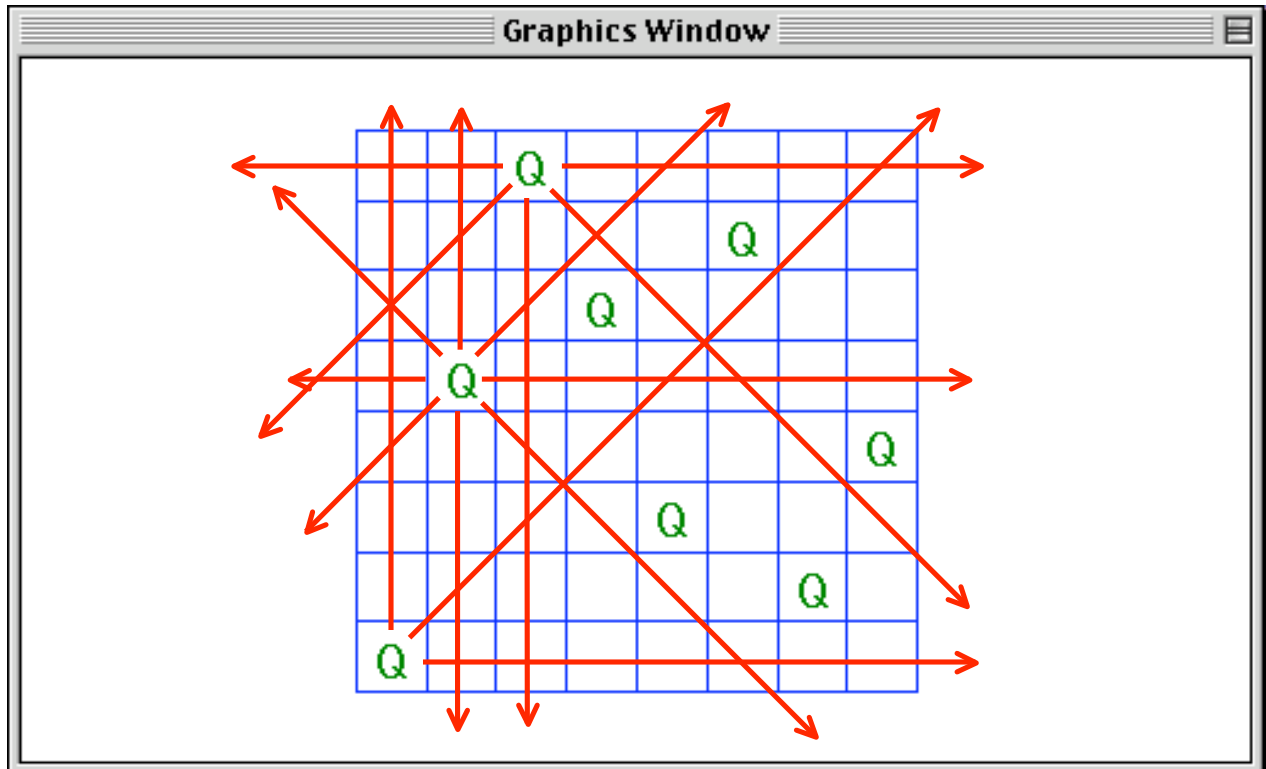


## Two-Dimensional Grids and Queen Safety

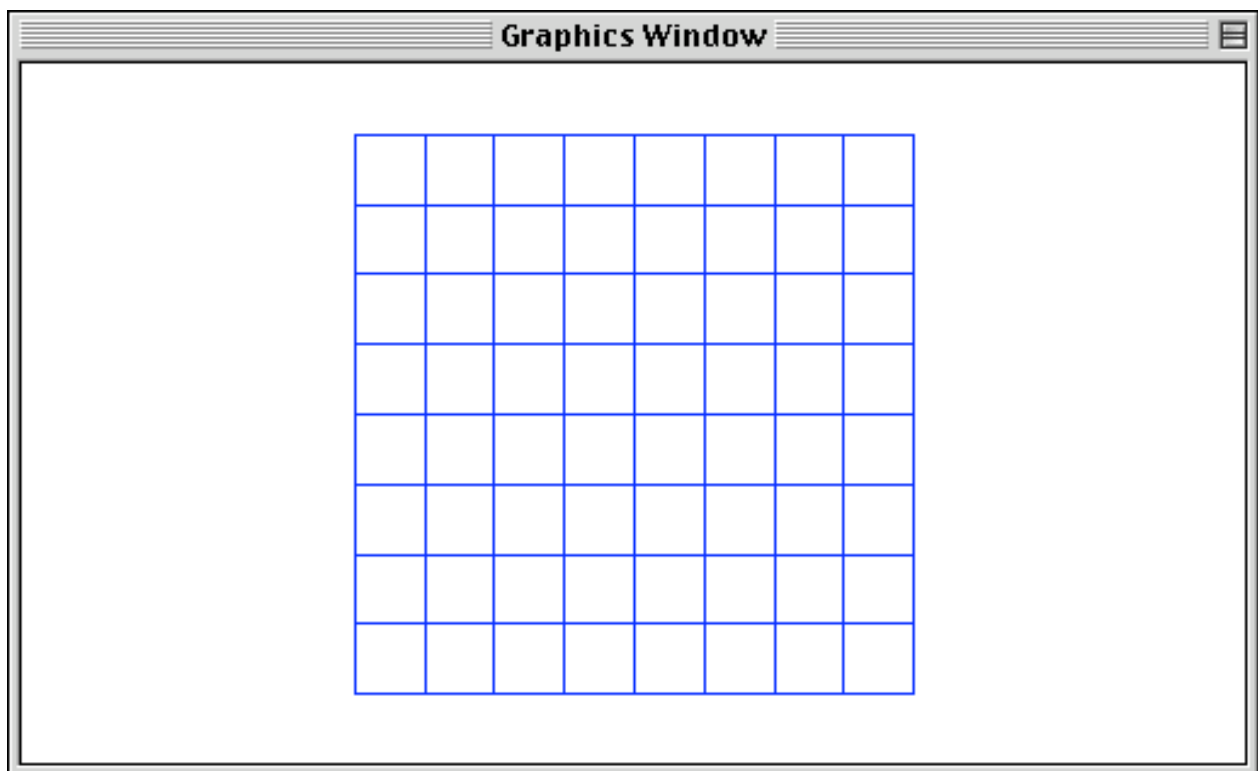
Today's larger example demonstrates the use of a grid of Boolean values—that is, a single declaration of a `Grid<bool>`—to maintain information on a chessboard and the placement of queens. The code relevant to the demonstration is presented here in this handout.



We want to use a two-dimensional `Grid` template to maintain information for all of the squares of a chessboard. Because our particular example only deals with the queen piece, we only need to store a `true` or `false` for each board location—`true` means a queen is present, `false` means there's no queen. Ultimately, we are going to be interested in looking at an arbitrary position of the board and determine whether or not any of the queens placed elsewhere can attack that position.

The following function `InitQueensBoard` 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 index. After this function gets called, the state of the board is very easily represented.

```
static const int kNumRows = 8;
static const int kNumColumns = 8;
void InitQueensBoard(Grid<bool>& board)
{
    for (int row = 0; row < kNumRows; row++) {
        for (int col = 0; col < kNumColumns; col++) {
            board[row][col] = false;
        }
    }
}
```



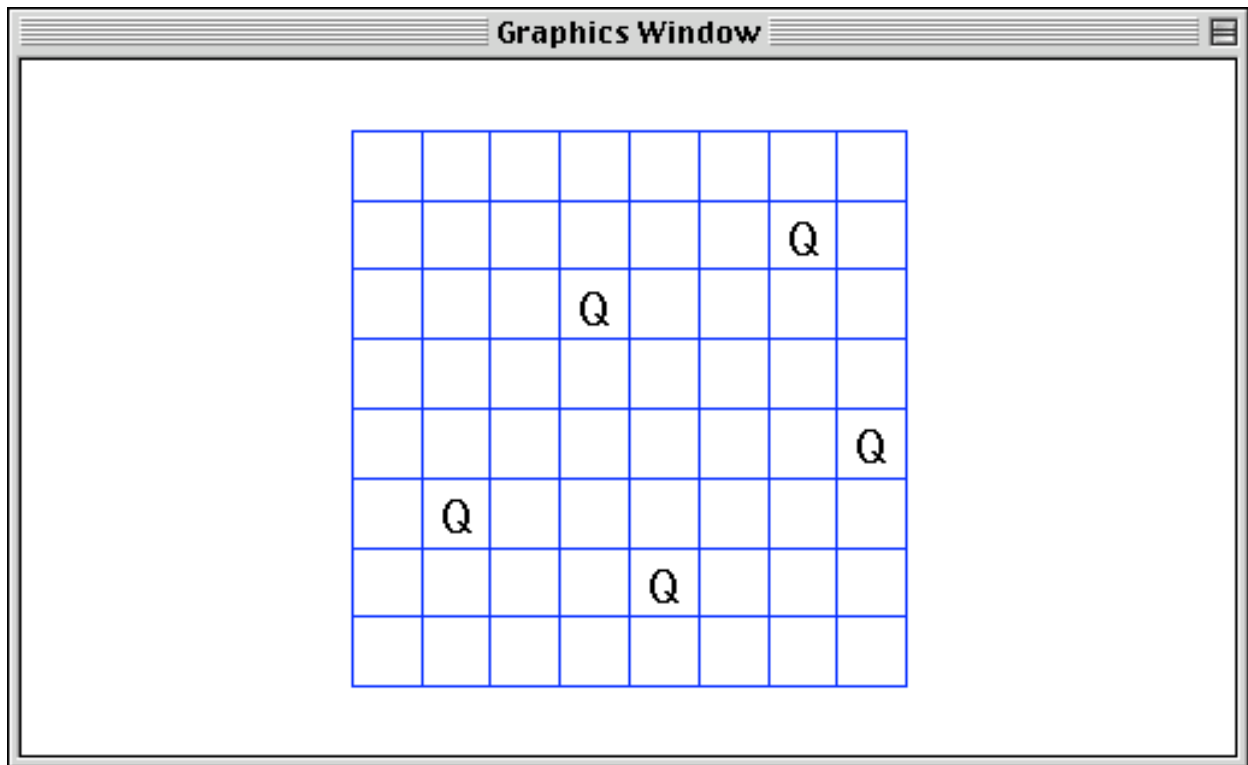
Next, we want a function **PlaceRandomQueens** that places some number of queens on the board at random locations. We enter the function with an idea of exactly how many queens need to be placed, and then repeatedly generate random coordinates on the board and placing queens there. Note the care **PlaceRandomQueens**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

```

void PlaceRandomQueens(Grid<bool>& board, int numQueensToPlace)
{
    Randomize();
    int numQueensPlaced = 0;
    while (numQueensPlaced < numQueensToPlace) {
        int row = RandomInteger(0, kNumRows - 1);
        int col = RandomInteger(0, kNumColumns - 1);
        if (!board[row][col]) {
            board[row][col] = true;
            MarkLocation(Occupied, row, col); // assume it draws a 'Q'
            numQueensPlaced++;
        }
    }
}

```

If we are dealing with an 8 x 8 board, then a call to **PlaceRandomQueens(board, 5)** **might** produce the following configuration:



At this point, we want to determine whether each of the unoccupied squares (those that are blank in the above drawing) can be attacked by a queen from one or more directions. Pretending for the moment that an **IsSafe** predicate function already exists, we can easily label each of the empty locations as safe or not using the following code snippet:

```

void MarkAllSquares(Grid<bool>& board)
{
    for (int row = 0; row < kNumRows; row++) {
        for (int col = 0; col < kNumColumns; col++) {
            if (!board[row][col]) { // is location empty?
                MarkLocation((IsSafe(board, row, col) ? Safe : Unsafe), row, col);
            }
        }
    }
}

```

To determine whether or not a particular location is safe from attack, we need to be able to search in all eight directions to see whether or not a queen is present in one or more of them. At first glance, we might be interested in writing functions like **IsSouthWestSafe**, **IsSouthSafe**, **IsSouthEastSafe**, etc. and then implement a function **IsSafe** to simply 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.
 */

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 minor differences. Just compare two of them (I omit the other six, because once you understand two, you really understand all eight.)

```

/**
 * Function: IsSouthWestSafe
 * -----
 * IsSouthWestSafe decides whether or not any danger can be seen
 * by looking from the specified (row, col) coordinate in the
 * southwestern direction. Assuming that the origin (0,0) coincides
 * with the lower 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.
 * Notice that we keep decrementing row and col by one every time we wish
 * to move one location further in the southwest direction.
 */

bool IsSouthWestSafe(Grid<bool>& board, int row, int col)
{
    row--;
    col--;
    while (OnBoard(row, col) && !board[row][col]) {
        row--;
        col--;
    }

    return !OnBoard(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 that the origin (0,0) coincides
 * with the lower 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.
 * Notice that we keep decrementing row by one every time we wish
 * to move one location further in the southern direction.
 */

bool IsSouthSafe(Grid<bool>& board, int row, int col)
{
    row--;
    while (OnBoard(row, col) && !board[row][col]) {
        row--;
    }

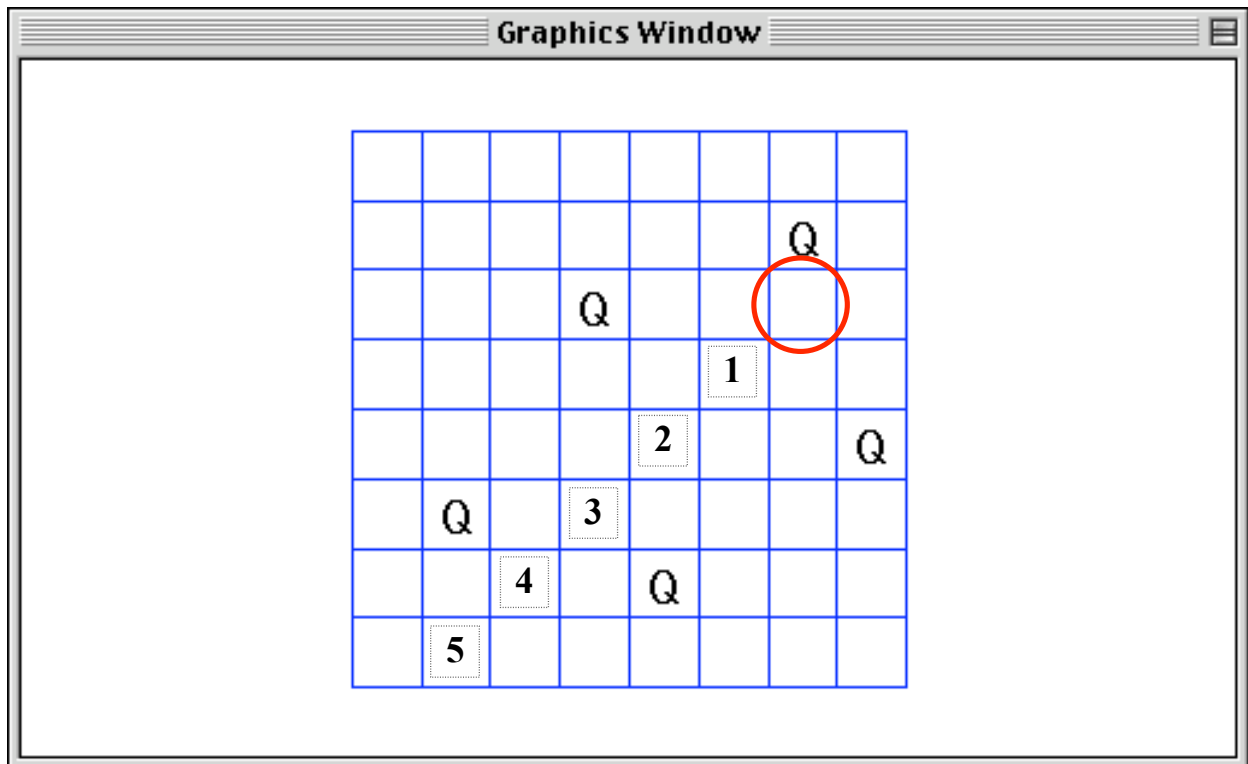
    return !OnBoard(row, col);
}

/**
 * Function: OnBoard
 * -----
 * OnBoard is a predicate function that returns true if and only if the
 * specified coordinate indexes a legal entry of our Grid<bool>.
 */

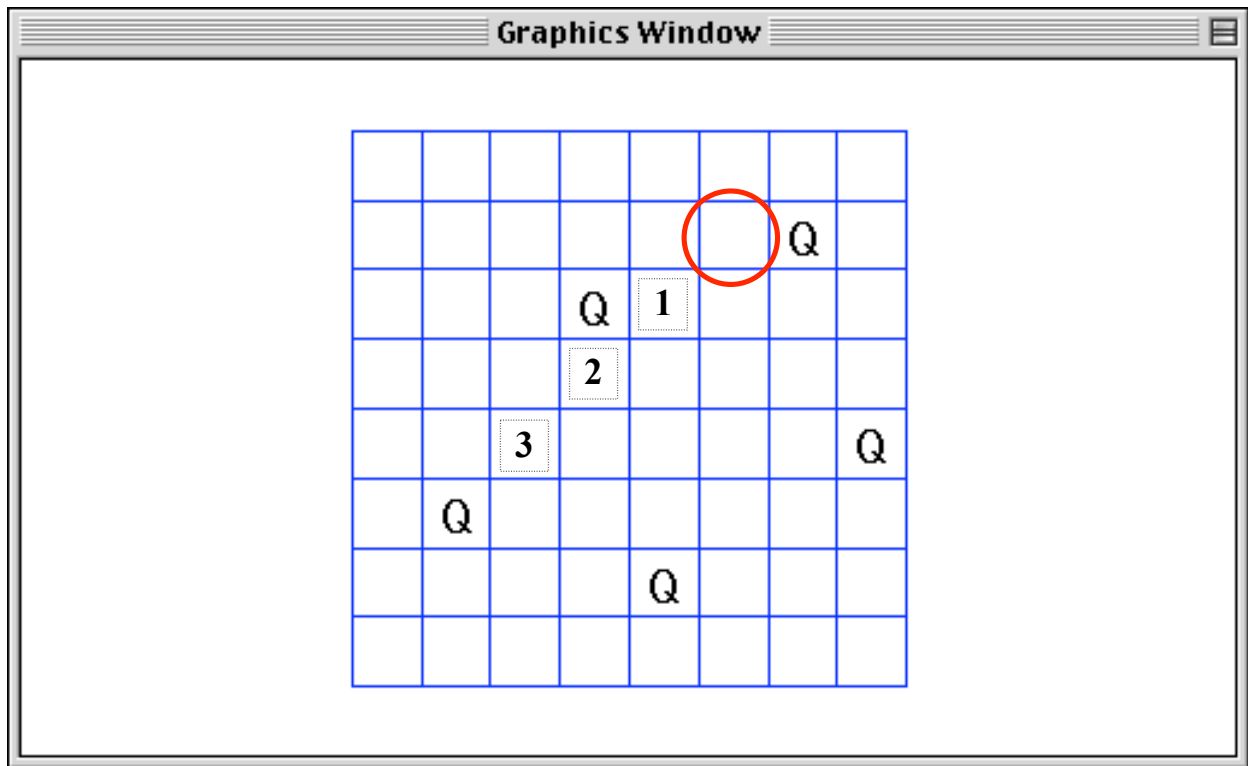
bool OnBoard(int row, int col)
{
    return (row >= 0 && row < kNumRows &&
            col >= 0 && col < kNumColumns);
}

```

To illustrate, consider the call `IsSouthWestSafe(board, 5, 6)`, 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 a location storing a queen. For this call, we expect the algorithm to return `true`.



However, a call to `IsSouthWestSafe(board, 6, 5)` would return `false`, because the `(2, 1)` 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 eight are really doing the same thing (you've seen two, but you can imagine that the other six look pretty much the same.) 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** and **col** are each decremented with each move

**South:** **row** gets decremented with each move, but **col** remains the same

**SouthEast:** **row** gets decremented and **col** gets incremented with each move

**West:** **row** remains the same, but **col** gets decremented with each move

**East:** **row** remains the same, but **col** gets incremented with each move

**NorthWest:** **row** gets incremented and **col** gets decremented with each move

**North:** **row** gets incremented with each move, but **col** remains the same

**NorthEast:** **row** and **col** are each incremented with each move

We should generalize our notions of eight different functions to a single function, using the manner in which coordinates get updated to specify the direction. 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 that the origin (0,0) coincides with the lower 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.
 */

bool IsDirectionSafe(Grid<bool>& board, int row, int col,
                    int dRow, int dCol)
{
    if ((dRow == 0) && (dCol == 0)) return true; // protect against bad call

    row += dRow;
    col += dRow;
    while (OnBoard(row, col) && !board[row][col]) {
        row += dRow;
        col += dRow;
    }

    return !OnBoard(row, col);
}

```

Need to search southwest 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.

```

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;
}

```