CSE 142 Summer 2001

2-D Arrays – an Example
[Corrected Version – Same Great Example,
Way Fewer Bugs]

Introduction

- Review:
  - 2-D arrays
- Today:
  - A game application

Representing a Chess Board

- A 2-D array is a natural representation for many games
- Example: chess (simplified – all pieces the same color)

```
class Chess {
    // pieces that can be stored on the chessboard
    public static final int EMPTY = 0;      // empty square
    public static final int PAWN = 1;       // pawn
    …
    public static final int QUEEN = 100;    // queen
    // size of chessboard
    public static final int SIZE = 8;
    // chessboard
    private int[ ][ ] board = new int[SIZE][SIZE];      // omit constructor to
                                                        // save space…
```


Chess Board Representation

- Picture:

Chess Board Processing

- A queen on a chessboard can capture any piece in any direction, provided there are no other pieces between the queen and the other piece
- Question: Given a location on the chessboard, is it threatened by a queen?
- How would we attack this problem?

Search Right

```
// return true if there is a queen to the right of board[row][col] that threatens
// that square
public boolean threatenedRight(int row, int col) {
    // search to the right until we find a non-blank square or run off the edge




    // if we found a queen, return true, else return false

}
```

Aside – Short Circuit Evaluation

- Way back we mentioned that && and || only evaluate their right arguments if necessary
- We're relying on this in the condition on the while statement

```
while (c < Chess.SIZE && board[r][c] == Chess.EMPTY) {
    c++
}
```

- Referencing board[r][c] will produce an index out of bounds error if c>= Chess.SIZE, so order of test is important

Bug: while (board[r][c] == Chess.EMPTY && r < Chess.SIZE) …

Search Up

```
// return true if there is a queen above board[row][col] that threatens
// that square
public boolean threatenedUp(int row, int col) {
    // search up until we find a non-blank square or run off the edge




    // if we found a queen, return true, else return false


}
```


Search Left

```
// return true if there is a queen to the left of board[row][col] that threatens
// that square
public boolean threatenedLeft(int row, int col) {
    // search to the left until we find a non-blank square or run off the edge
    int r = row; int c = col-1;
    while (c >= 0 && board[r][c] == Chess.EMPTY) {
            c—
    }

    // if we found a queen, return true, else return false
    return (c >= 0 && board[r][c] == Chess.QUEEN);
}
```


Search Down to the Right

```
// return true if there is a queen down to the right of board[row][col] that threatens
// that square
public boolean threatenedDownRight(int row, int col) {
    // search down right until we find a non-blank square or run off the edge
    int r = row+1; int c = col+1;
    while (c < Chess.SIZE && r < Chess.SIZE && board[r][c] == Chess.EMPTY) {
            r++; c++;
    }

    // if we found a queen, return true, else return false
    return (c < Chess.SIZE && r < Chess.SIZE && board[r][c]==Chess.QUEEN);
}
```


Department of Redundancy Department

- Lots of common structure in these searches
- What's different?
- Can we factor out the differences and write one set of code?

Direction Abstraction

- The differences is that each of the searches has a different +1, 0, -1 increment for the row and column numbers, and corresponding bounds test
- We really want to perform one search, parameterized by direction

```
for (int direction = 0; direction < 7; direction++) {
    int r = …; int c = …;
    while (the next square in the current direction on the board and empty) {
       increment r and c in that direction;
    }
    if (r and c are on the board and board[r][c] is a queen) {
            return "this square is threatened"
    }
}
```

Deltas

- We can store the different increments in a table. Definitions:
  - deltaR[direction] is the row increment in that direction
  - deltaC[direction] is the column increment in that direction
- In Java (using new syntax to initialize the arrays all at once)

```
int[ ] deltaR = {-1, -1,  0, +1, +1, +1,  0, -1};
int[ ] deltaC = { 0, +1, +1, +1,  0, -1, -1, -1};
```


Bounds Function

- This is worth isolating in a separate function so it can be reused

```
// return true if row, col is on the board, otherwise return false
boolean inBounds(int row, int col) {
    return (row >= 0 && row < Chess.SIZE && col >= 0 && col < Chess.SIZE);
}
```


Parameterized Search

```
// return true if there is a queen that threatens board[row][col]
public boolean threatened(int row, int col) {
    for (int d = 0; d < 8; d++ {
        if (threatenedInDirection(d, row, col)) {
            return true;
        }
    // no threats
    return false;
}
```


Parameterized Search

```
// return true if there is a queen that threatens board[row][col] in direction d
public boolean threatenedInDirection(int d, int row, int col) {
    // calculate row, column of neighboring square in direction d
    int r = row + deltaR[d];
    int c = col + deltaC[d];
    // search in direction d
    while (inBounds(r, c) && board[r][c] == Chess.EMPTY) {
        r = r + deltaR[d];
        c = c + deltaC[d];
    }
    return (inBounds(r, c) && board[r][c] == Chess.QUEEN);
}
```

- Notice that we are relying on the short-circuit evaluation of && in an essential way