This lecture demonstrates an application of stacks: implementing backtracking to solve the N-Queens problem. The presentation includes a demonstration program which you can run at a couple points during the presentation. The demonstration requires EGA or VGA graphics on a PC.

The best time for this lecture is after the students have read Chapter 7 on stacks. If the students want additional information about the N-queens problem, you can direct them to Programming Project 9 in Chapter 7.
We'll start with a description of a problem which involves a bunch of queens from a chess game, and a chess board.
Some of you may have seen this problem before. The goal is to place all the queens on the board so that none of the queens are attacking each other.
If you play chess, then you know that this forbids two queens from being in the same row...
The N-Queens Problem

Two queens are not allowed in the same row, or in the same column...

...or in the same column...
As a quick survey, how many of you think that a solution will be possible? In any case, we shall find out, because we will write a program to try to find a solution.

As an aside, if the program does discover a solution, we can easily check that the solution is correct. But suppose the program tells us that there is no solution. In that case, there are actually two possibilities to keep in mind:

1. Maybe the problem has no solution.
2. Maybe the problem does have a solution, and the program has a bug!

Moral of the story: Always create an independent test to increase the confidence in the correctness of your programs.
The program that we write will actually permit a varying number of
queens. The number of queens must always equal the size of the chess
board. For example, if I have six queens, then the board will be a six by
six chess board.
At this point, I can give you a demonstration of the program at work. The demonstration uses graphics to display the progress of the program as it searches for a solution.

During the demonstration, a student can provide the value of N. With N less than 4, the program is rather boring. But N=4 provides some interest. N=10 takes a few minutes, but it is interesting to watch and the students can try to figure out the algorithm used by the program.
I want to show you the algorithm that the program uses. The technique is called \textit{backtracking}. The key feature is that a \textit{stack} is used to keep track of each placement of a queen.
How the program works

Each time the program decides to place a queen on the board, the position of the new queen is stored in a record which is placed in the stack.

For example, when we place the first queen in the first column of the first row, we record this placement by pushing a record onto the stack. This record contains both the row and column number of the newly-placed queen.
In addition to the stack, we also keep track of one other item: an integer which tells us how many rows currently have a queen placed.
When we successfully place a queen in one row, we move to the next row. We always start by trying to place the queen in the first column...

How the program works

When we successfully place a queen in one row, we move to the next row. We always start by trying to place the queen in the first column of the new row.
How the program works

...if there is a conflict with another queen, then we shift the new queen to the next column.

But each new placement must be checked for potential conflicts with the previous queen. If there is a conflict, then the newly-placed queen is shifted rightward.
Sometimes another conflict will occur, and the newly-placed queen must continue shifting rightward.
How the program works

When there are no conflicts, we stop and add one to the value of filled.

When the new queen reaches a spot with no conflicts, then the algorithm can move on. In order to move on, we add one to the value of filled...
How the program works

Let's look at the third row. The first position we try has a conflict...

...and place a new queen in the first column of the next row.
In this example, there is a conflict with the placement of the new queen, so we move her rightward to the second column.
Another conflict arises, so we move rightward to the third column.

How the program works

...and we shift to the third column.
Yet another conflict arises...

Another conflict arises, so we move rightward to the third column.
Yet another conflict arises, so we move to the fourth column. The key idea is that each time we try a particular location for the new queen, we need to check whether the new location causes conflicts with our previous queens. If so, then we move the new queen to the next possible location.
Sometimes we run out of possible locations for the new queens. This is where backtracking comes into play.
To backtrack, we throw out the new queen altogether, popping the stack, reducing filled by 1, and returning to the previous row. At the previous row, we continue shifting the queen rightward.
Notice that we continue the previous row from the spot where we left off. The queen shifts from column 3 to column 4. We don't return her back to column 1.

It is the use of the stack that lets us easily continue where we left off. The position of this previous queen is recorded in the stack, so we can just move the queen rightward one more position.
The new position for row 2 has no conflicts, so we can increase filled by 1, and move to row 3.

The new position for row 2 has no conflicts, so we can increase filled by 1, and move again to row 3.
At the new row, we again start at the first column. So the general rules are:

When the algorithm moves forward, it always starts with the first column.

But when the algorithm backtracks, it continues wherever it left off.
Here's the pseudocode for implementing the backtrack algorithm. The stack is initialized as an empty stack, and then we place the first queen.

After the initialization, we enter a loop with three possible actions at each iteration. We'll look at each action in detail...
The nicest possibility is when none of the queens have any conflicts. In this case, we can increase filled by 1.

If filled is now N, then we are done!

But if filled is still less than N, then we can move to the next row and place a queen in the first column. When this new queen is placed, we'll record its position in the stack.

Another aside: How do you suppose the program "checks for conflicts"? Hint: It helps if the stack is implemented in a way that permits the program to peek inside and see all of the recorded positions. This "peek inside" operation is often implemented with a stack, although the ability to actually change entries is limited to the usual pushing and popping.
The second possibility is that a conflict arises, and the new queen has room to move rightward. In this case, we just move the new queen to the right.
The last possibility is that a conflict exists, but the new queen has run out of room. In this case we backtrack:

Pop the stack,
Reduce filled by 1.

We must keep doing these two steps until we find a row where the queen can be shifted rightward. In other words, until we find a row where the queen is not already at the end.

At that point, we shift the queen rightward, and continue the loop.

But there is one potential pitfall here!
Pseudocode for N-Queens

repeat these steps
    □ if there are no conflicts with the queens...
    □ else if there is a conflict and there is room to shift the current queen rightward...
    □ else if there is a conflict and there is no room to shift the current queen rightward...
Backtrack!
Keep popping the stack, and reducing filled by 1, until you reach a row where the queen can be shifted rightward. Shift this queen right.

The potential pitfall: Maybe the stack becomes empty during this popping. What would that indicate?

Answer: It means that we backtracked right back to the beginning, and ran out of possible places to place the first queen. In that case, the problem has no solution.
Just for fun, we can run the demonstration program again now. See if you can follow the backtracking in action.
Stacks have many applications. The application which we have shown is called **backtracking**. The key to backtracking: Each choice is recorded in a stack. When you run out of choices for the current decision, you pop the stack, and continue trying different choices for the previous decision.

A quick summary . . .
Feel free to send your ideas to:

Michael Main
main@colorado.edu