A Taste of Artificial Intelligence:

Why AI in this course?
(1) AI and robotics are closely connected.
(2) Many programming techniques were developed first in AI (including many of the ideas of OOP, because AI systems have to represent knowledge, which is often expressed in terms of classes). AI researchers have often been forced to develop new programming techniques. As a consequence, many AI programming ideas (like OOP) have been exported to other application domains.

Definition: AI is the art of getting computers to do things that if they were done by humans would be considered intelligent.

(Another definition: Getting a computer to pass the "Turing Test," which means that it responds indistinguishably from a human in an A-B test. This is a very broad and stringent notion of intelligence.)

Notice that these definitions focus on intellectual activities primarily. Communicating in language. AI researchers have focused on intellectual activities (solving logic problems, proving mathematical theorems, problem solving, language understanding, playing board games such as checkers, chess, and go).

This is sometimes called GOFAI = Good Old-Fashioned AI.

This type of intelligence is all in the head. There has been growing recognition since about 1980 that these intellectual capacities are built on a necessary foundation of lower-level sensory-motor activities. This is called embodied AI (and, when applied to humans, embodied cognitive science or embodied psychology). This is part of the reason for working with robots.

In this chapter, however, we are looking at GOFAI, especially board games.

In programming a board game:
(1) Get it to play legally (according to the rules).
(2) Get it to play well (this is where the intelligence comes in).

How do we program the computer to play Tic-Tac-Toe?

Relevant external objects and internal objects ("objects of thought") need to be represented by some data structures. "Pick the right data structure and the algorithm will design itself" applies here. It will be a lot easier to program if we pick the right data structures.

How do we represent the board? It could be a 3x3 matrix, but in this case it's easier to use a 9-element vector (indexed, as usual, 0 to 8).

```
typedef vector<char> Board;
```

This tells the reader that some vectors of chars are conceptually Boards.

How do we tell if someone has a win? This is another clever data structure choice. We will have a matrix (2D array) of all the legal winning positions. If I have the same piece (X or O) in the same row, column, or diagonal, I have a win. So keep a data structure containing the indices of the rows, columns, and diagonals (8x3 matrix):

0 1 2
3 4 5
6 7 8

0 3 6

1 4 7

2 5 8

0 4 8

2 4 6

We are going to store this information in a C-array called WINS:

```
// These square triples represent wins (three in a row).
int WINS[][3] = {{0, 1, 2},{3, 4, 5},{6, 7, 8},      // the rows
    {0, 3, 6},{1, 4, 7},{2, 5, 8},                   // the columns
    {0, 4, 8},{2, 4, 6}};                            // diagonals
```

Therefor WINS[k] = the k-th way of winning. For example WINS[0] = {0, 1, 2}; WINS[4] = {1, 4, 7}, etc.

Therefore, to win the first way (the same piece across the top row), I need the same piece in board positions
WINS[0][0] = 0, WINS[0][1] = 1, WINS[0][2] = 1.
To win in the 4-th way, I need the same piece in board positions
WINS[4][0] = 1, WINS[4][1] = 4, WINS[4][2] = 7.

The piece (X, O, or blank) in position WINS[4][0] is
board[WINS[4][0]]
To see if we have won in the 4-th way, we have to see if the same piece (X or O) is in:
board[WINS[4][0]] == board[WINS[4][1]] == board[WINS[4][2]]

Key Techniques (in AI):

(1) Generate and Test: Generate possible moves and test to see which is best.

(2) Heuristics: Rules of thumb that don't always work (but they are "the way to bet"). In this program openWins, which guesses that if there more possible ways to win, then it's probably a better move.

(3) Lookahead: Look forward several moves in the game. We usually talk of "plies" (one ply is a pair of moves by the player and their opponent).

To do the look ahead, you have to generate possible moves at each stage (a

generate-and-test process). When you can't look any further ahead (too many possibilities), you apply heuristics.

The problem is combinatorial explosion. The number of possibilities to be evaluated increases exponentially with the amount you look ahead.

Interestingly, human chess masters don't work this way. They focus on only a few possible moves at each stage, but analyze them very deeply.