APS 105 — Computer Fundamentals

Lab #6: Tic-tac-toe
Winter 2019

Important note: You must use the `submit` command to electronically submit your solution by the end of your lab session.

In this lab, you will write an application that allows two users to play Tic-tac-toe against each other. By the end of this lab, you should be able to:

- Accept and validate user input from the keyboard.
- Declare and define multiple functions.
- Declare, initialize, and modify a one-dimensional array.
- Pass around an array to different functions.

1 Summary of the Game

Tic-tac-toe is a two-player game where each player takes turns marking spaces in a 3x3 grid. In this case, Player A marks their space with 'A' and Player B marks their space with 'B'. A player wins when they manage to place three marks in a straight line horizontally, vertically, or diagonally. It is possible that the game leads to a draw, where there are no places on the 3x3 grid left to mark and neither player has three marks in a straight line.

2 The Game Loop

Game programming involves looping until a game over condition is met. In Tic-tac-toe, the game over condition occurs when one of the two players has won or there is a draw. During the game loop, the program will:

1. Display the state of the game.
2. Accept input from the user.
3. Update the state of the game based on user input.
4. Check the game over condition.

We will use integers to represent the state of the game. We will use functions to help with steps 1, 2, and 4. You can create additional functions if you would like to.

2.1 Representing the Game State

The game board is a 3x3 grid, which will be represented as a one-dimensional integer array with 9 elements. Each element of the array represents a position on the game board. Each element is initialized to a number that identifies its position. Specifically, index 0 is initialized to 1, index 1 is initialized to 2, and so on.

Player A and Player B are each represented by a unique integer. These integers should not be the same as the positions above (i.e., should not be an integer between 1 and 9).

2.2 Display the Game Board

You will write a function that displays the game board. The function will output the game board to the terminal (i.e., via `printf`). The function will accept, as input, an array that represents the game board, a number that represents Player A, and a number that represents Player B. Specifically, the function's signature is:

```
void printBoard(int board[], int playerA, int playerB);
```

The `printBoard` function prints a new line, then the game board as a 3x3 grid, then another new line. An example of an empty game board where neither player has played yet is below. Spaces are annotated as **[space]** and new lines are annoted as **[newline]**, for your convenience.

```
[newline]
[space]1[space]2[space]3[newline]
[space]4[space]5[space]6[newline]
[space]7[space]8[space]9[newline]
[newline]
```

If Player A has marked position 5 and Player B has yet to take their turn, then we would replace 5 with 'A' for Player A:

```
[newline]
[space]1[space]2[space]3[newline]
[space]4[space]A[space]6[newline]
[space]7[space]8[space]9[newline]
[newline]
```

2.3 Accept User Input

You will write a function that accepts user input. The function will read an integer from the keyboard (i.e., using `scanf`) and validate it. You can assume that the keyboard input will always be an integer that can be parsed by `scanf`. An integer is valid if:

1. It is within the range of positions on the game board (i.e., between 1 and 9).
2. The position on the game board has not already been marked by one of the players.

If the input is valid, then the function will `return` the *array index* that has been selected by the user. If the input is invalid, then the function will output an appropriate message and ask the user to try again. The function's signature is:

```
int requestValidInput(int board[], int playerA, int playerB);
```

If the input is not within the range of positions on the game board, the function should ask the user to try again:

```
Invalid input, please try again.[newline]
[wait for new input]
```

If the input is within the range of positions on the game board, but that position has already been marked, the function should ask the user to try again:

```
That position has already been played, please try again.[newline]
[wait for new input]
```

The `requestValidInput` function should continue to loop until valid input is given.

2.4 Update Game State

You should update the state of the game based on the valid input you've received from the user. This involves updating the game board (i.e., the integer array). Once updated, you should also switch to the next player's turn. You do not need to do this step in a function – it can be done directly in the game loop.

2.5 Check for Game Over

There are two ways the game can finish. Either one of the players has won or the game has ended in a draw. We will separate these into two functions. The easiest way to solve this is through brute force (it is only a 3x3 grid).

To check if a player has won, write a function that will accept, as input, an array that represents the game board, a number that represents Player A, and a number that represents Player B. The function will `return` zero if neither player has won. Otherwise, it will `return` the number that represents the player who won the game. The function's signature is:

```
int checkForWinner(int board[], int playerA, int playerB);
```

You should also check if the game has ended in a draw. One option is to create a function, as is the theme of this lab. For example, the function could check to see if there are any positions on the board that are unmarked. If so, then it will `return true`. Otherwise, all positions have been marked and the function will `return false`. There are other ways to approach this, so we leave the checking for a tie up to you.

2.6 Getting Started

Consider the pseudo code in Algorithm 1 to help you get started. When printing whoever's turn it is, use (where appropriate):

- It is Player A's turn.\n
- It is Player B's turn.\n

When printing the result of the game, use (where appropriate):

- Player A wins!\n
- Player B wins!\n
- It's a draw!\n

Algorithm 1 Pseudocode for Tic-tac-toe

```
Require: Game board is initialized.
Require: $currentPlayer \leftarrow PlayerOne$

  $gameOver \leftarrow No$
  while $gameOver \neq Yes$ do
    print The current state of the Tic-tac-to Board:\n
    $printBoard(...)$

    print Whoever's turn it is.
    print Please enter a valid position to play.\n
    $input = requestValidInput(...)$

    Update game board.
    Update $currentPlayer$.

    $winner = checkForWinner(...)$
    if a winner exists then
      $gameOver \leftarrow Yes$
    else if $checkForStalemate(...)$ then
      $gameOver \leftarrow Yes$
    end if
  end while

  print Whoever won the game, or if it was a draw.
  $printBoard(...)$
```

3 Example Game Output

The following example shows how a game is played to completion. Spaces for the game board are shown explicitly as ␣.

```
The current state of the Tic-tac-toe Board:

␣1␣2␣3
␣4␣5␣6
␣7␣8␣9

It is Player A's turn.
Please enter a valid position to play.
1
The current state of the Tic-tac-toe Board:

␣A␣2␣3
␣4␣5␣6
␣7␣8␣9

It is Player B's turn.
Please enter a valid position to play.
```

```
1
That position has already been played, please try again.
10
Invalid input, please try again.
3
The current state of the Tic-tac-toe Board:

␣A␣2␣B
␣4␣5␣6
␣7␣8␣9

It is Player A's turn.
Please enter a valid position to play.
4
The current state of the Tic-tac-toe Board:

␣A␣2␣B
␣A␣5␣6
␣7␣8␣9

It is Player B's turn.
Please enter a valid position to play.
6
The current state of the Tic-tac-toe Board:

␣A␣2␣B
␣A␣5␣B
␣7␣8␣9

It is Player A's turn.
Please enter a valid position to play.
7
Player A wins!

␣A␣2␣B
␣A␣5␣B
␣A␣8␣9
```

4 Submission Instructions

In a file called `Lab6.c`, write your solution to the problem. There are ten (10) marks available in this lab.

4.1 TA Grading (4 marks)

Your solution will be marked by a Teaching Assistant during your scheduled lab period for programming style and your understanding of the code. You can discuss programming style with your TA and on Piazza. Here are some quick guidelines:

- Good choices for variable names that indicate their purpose.
- A consistent naming convention. Use camelCase for normal variable and function names.
- Comments that explain code that is difficult to understand.
- Proper indentation.
- Appropriate white space between lines for better readability.
- A limited number of, or ideally zero, global variables.

4.2 Automated Grading (6 marks)

Your solution must be submitted electronically by the end of your scheduled lab period. Submission of the lab requires you to use a terminal. In the terminal, you must:

1. Go to the directory that contains `Lab6.c` (i.e., use the cd command in the terminal).
2. Type in the following command: `/share/copy/aps105s/lab6/submit`

This command will run an exercise program that will check to make sure everything looks okay. If it finds a problem, it will ask you if you are sure that you want to submit.

Note that you may submit your work as many times as you want prior to the deadline; only the most recent submission is marked. You can also run the exerciser on your own with the following command:

```
/share/copy/aps105s/lab6/exercise
```

Finally, you can also check to see if what you think you have submitted is actually there, for peace of mind, using the following command:

```
/share/copy/aps105s/lab6/viewsubmitted
```

You must submit your lab by the end of your assigned lab period. Late submissions will not be accepted, and you will receive a grade of zero.

4.3 Obtaining Your Automated Grade

After all lab sections have finished, a short time later, you will be able to run the auto-marker to determine the auto-marked fraction of your grade on the code you have submitted. To do so run the following command:

```
/share/copy/aps105s/lab6/marker
```

This command will compile and run your code and test it with all the test cases used to determine the auto-mark grade. You will be able to see those test casesâĂŹ output and what went right or wrong.

5 Challenge Mode

If you have completed the lab and submitted your code with plenty of time to spare, here are some options to increase the difficulty. **Challenge Mode is not graded and will not provide any bonus marks.** Make sure you don't submit challenge mode code, as it will not conform to the auto-marker. We recommend creating a copy of the lab code before starting Challenge Mode. Or if you are using git (a popular version control tool) create a new branch.

1. Allow the game to be played multiple times, and keep track of the number of Player A wins, Player B wins, and draws. Display the scoreboard when the user decides to exit the game.
2. Instead of Player B being controlled via the keyboard, consider writing code that makes decisions based on how Player A has played (i.e., artificial intelligence).
3. Instead of using a hard-coded 3x3 grid, consider allowing the user to enter how large of a Tic-tac-toe grid they'd like. This will make your `checkForWinner` function more complicated. Some research online shows that this is a popular interview question.