Matthew Manzi
COURSE
PROFESSOR
17 April 2018

Tic Tac Sweeper

This project combines features of the traditional games Tic Tac Toe and Minesweeper. In Tic Tac Sweeper, players may choose from boards of size three spaces tall by three spaces wide, four by four, or five by five. Each player is then asked to place a mine that will be hidden on the board. If a player lands on a mine that is not theirs, they lose and the game ends. In order to win, a player must have three, four, or five (respective to board size) of their pieces in a row. A tie occurs when the board is filled and neither winning states have been encountered.

Starting with the professor's original Tic Tac Toe program, I studied and reverse-engineered it, paying close attention to the logic for checking wins and ties. From there, I began my own version of the program which initially consisted of molding the provided code into my own interpretation, updating variable names and adding comments where helpful.

Once I had reshaped the code into a preferred format and confirmed that it worked as before, I began the extrapolation process. I decided the most efficient way to support 4x4 and 5x5 board sizes, with respect to development time, was to add two more versions of most of the data and code. Although heavier on memory usage, having versioned code proved easier to debug and adapt, as well as quick to write and maneuver.

The most intensive part of this adaptation was creating the array of numbers used to check wins (aptly named "G.o.D.", a.k.a. "Grid of Death", as the 5x5 version totaled 400 bytes) and the subroutines that uses these arrays. I employed the use of newer 64-bit registers (i.e. r8 through r15) and learned to pay careful attention to operand sizes, which demonstrated why it is always a good idea to use explicit sizes.

With the addition of more positions to play on larger game boards, the issue of two-digit input became apparent. Now, the game must support input of one- and two-digit numbers without crashing. Because the logical limit of input to these boards is capped at 24 and `buffer` was already four bytes, it made sense to evaluate numbers up to 999 (three bytes for ASCII digits, one for a return/newline character). Although the game does not crash for input greater than 999, it may yield unexpected results. However, for all input in the range [0,999], a subroutine is used to count the number of bytes up to the character `0xA`, compute the decimal result by looping over each byte and converting its ASCII value to its intended decimal value with respect to the base 10 number system, then store it in a register for future use. Because this subroutine was designed independent of board size, it is used for all numerical input, including mine placements.

The latter feature was implemented last, taking advantage of most of the constructs already in place. To keep track of mine positions and states, extra memory was reserved for each player's mine offset (relative to a linear board), whether or not a mine had exploded, and if mines should be printed. One benefit of storing mine offsets separate from the board as raw numbers is that there are less conversions required to check for a detonation. Additionally, the new subroutines for printing and checking mines are also size-independent, reaping similar benefits as previously mentioned.

With all of the new features, debug printing deserved a small makeover, too. It still provides two linearized versions of the game board (in ASCII and hexadecimal) and now includes support for mine location and status.

All in all, this project was quite the feat. Although I don't believe anyone may ever desire to play a game of Tic Tac Sweeper written in assembly, it gave me thorough exposure to and practice with user input validation, subroutines, conditional jumps, and data manipulation in a low-level language, along with a certain respect for those that frequently write code like this.