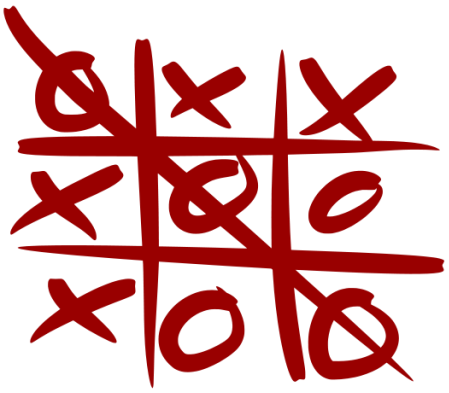
gTic-Tac-Toe



Project Description

Almost everyone learned to play the game of tic-tac-toe when we were growing up (or "noughts and crosses" if you grew up in an English neighborhood.) For this project, we will write a computer program to play the game of tic-tac-toe against an opponent. For a complete description of the game, including strategy and tactics, see https://en.wikipedia.org/wiki/Tic-tac-toe.

We will be using a fairly simple method to represent the current state of the game. We first assign each position on the game board a specific number. The following table defines the mapping from each square on the board to a position number:

0	1	2
3	4	5
6	7	8

Each square on the board can contain one of three values… either blank (not yet chosen), filled in with an X, or filled in with an O. We arbitrarily assign the "value" of a square to the digit "0" for blank, "1" for X, or "2" for O. Then, we represent the entire board as a nine-digit base 3 number, where each digit represents a specific square on the board, and the value of the digit represents what is inside that square. Thus, we can represent any tic-tac-toe state as a number between 0 and 3^9-1. To demonstrate our encoding of a tic-tac-toe board, consider the following example:

		O
X	X	
O		X

Maps to:

Position	8	7	6	5	4	3	2	1	0
Contents	X		O		X	X	O		
Value	1	0	2	0	1	1	2	0	0
	3^8	3^7	3^6	3^5	3^4	3^3	3^2	3^1	3^0
Dec. Value	6,561		729x2 1458		81	27	9x2 18		

$= 102011200_3$
$= 8{,}145_{10}$

Using this encoding scheme, it's fairly simple to show that we can represent each successive move in the game as a single number. For instance, given the above state, suppose the O player chooses to block the threat of 3 X's in the horizontal middle row. He would do so by putting an O at position 5. The resulting state would be calculated by adding $2x3^5$ to the previous state, or 8,145 + (2x243) = 8,631.

The way your tic-tac-toe playing program should work is as follows…

- When your program is invoked, it will have a single argument: either an "X" or an "O" to indicate whether you should play as X or as O in the game.
- You always start with a blank board… state=0 using our encoding from above.
- Players will take turns filling in single squares on the board.
- We will adopt the convention that the X player always takes the first turn. Therefore, if you are the X player, you may choose any position on the board and fill it with an X. After you have done so, calculate the new state of the board, and print the value of that state to standard output.
- When it is your opponents turn, your opponent will enter the position of his choice as a number between 0 and 8 using standard input. You read your opponent's choice, update the state to represent your opponent's choice, and print the updated state to standard output. If your opponent makes an invalid choice (for instance, a number not between 0 and 8, or the number of a position that already contains either an X or an O), then you may declare a foul, and you automatically win the game.
- If either player gets three X's or O's in a row, either horizontally, vertically, or on a diagonal, then that player wins the game.
- If no player gets three X's or O's in a row, and all spaces on the board are used, then the result is a tie game.
- We will use the convention that state=3^9 represents a tie game, $3^9 + 1$ represents a win for X, and $3^9 + 2$ represents a win for O. The ending state, either a tie, or a win for X, or a win for Y, should also be printed to standard output.
- If you print an invalid state (for instance, a number greater than $3^9 + 2$, or a state that contains more than one move from the previous state), then you automatically lose the game.

This sounds really complicated, but most of the code to implement this infrastructure will be provided for you. You need to concentrate on is the part where you are provided with a state (input as a number) and need to determine where to put your next marker.

After the project due date, we will grade your program using several criteria. First, you program needs to compile with no error or warning messages (even with the -Wall flag turned on!) Secondly, your program will be run against a very simple strategy to ensure it runs correctly, produces valid output states, and reads your opponent's choices. Finally, your program will be run against other students' programs, chosen at random. The more your program wins, the higher grade you will get.

Working on the Project

You have been provided with the basic infrastructure for the C code to play tic-tac-toe. On the class web page, there is a file called proj1.tar.gz that you can download to your own UNIX directory. The command:

```
tar -xvzf proj1.tar.gz
```

will first create a sub-directory of your current directory called "proj1", and then populate that sub-directory with the contents of the proj1.tar.gz file. The proj1 sub-directory will contain the following files:

- ttt.c – C source code that contains several functions to implement the tic-tac-toe game playing infrastructure. The functions provided are as follows:
  - main : You should not have to modify the code in the main function. This function:
    - reads the parameters to find out whether you are playing X or O and sets "me" and "them" to indicate so.
    - initializes the state (the "state" variable) to zero (empty board),
    - initializes the player taking the current turn (the "turn" variable) to X,
    - then loops until a player wins or it's a tie. In the loop:
      - if it's my turn, main invokes the "myTurn" function. If it's my opponents turn, main invokes the "theirTurn" function. Both of these functions return an updated state.
      - prints out the updated state to standard output,
      - invokes the "checkWin" function to see if we have reached a winning state.
      - If not, switches the current turn from X to O (or O to X) and goes around the loop again.
    - When the loop completes, the main function prints out the final state to standard output, and
    - writes a message to standard error indicating who won, and
    - returns (exits the program)
  - myTurn : This is the function you need to concentrate on. It is currently an empty function. It takes two arguments: the current state, and a "me" variable to indicate whether you are playing X (me==1) or O (me==2). You need to modify this function so that it figures out where to put your next marker (either X or O, depending on the "me" variable.) Once you have determined a position, you may want to invoke the "addState" function to calculate the new state that results when you put your marker in the position you selected, and return the result – the updated state.
  - theirTurn : You should not have to modify the code in the theirTurn function. This function:
    - prints the current board, based on the input state variable, to standard error.
    - Prints a prompt to standard error asking the user to enter the position of his next choice
    - Reads a single number from standard input, and saves that number in the "pos" variable.

        - Checks to make sure that the "pos" variable contains a valid position – one that does not already contain an X or O. If not, returns the new state of "I win"… in other words, X wins if me is 1, or O wins if me is 2.
        - Invokes addState with the user's chosen position to update the state variable, and returns that updated value.
    - checkWin : You should not have to modify the code in the checkWin function. This function:
        - Checks to see if the current state contains three X's or three O's in a row, either horizontally, vertically, or on a diagonal. If so, returns the state indicating X wins or O wins.
        - Checks to see if there are any empty squares in the current state. If not, returns the state indicating that there is a tie.
        - Otherwise, just returns the input state.
    - addState : You should not have to modify the code in the addState function. This function takes three parameters: the current state, a position index, and an indicator of what should go in that position, an X or an O. The function checks to make sure the specified position is empty in the current state, and then returns an updated state to indicate the state with the specified marker put into the square at the specified position.
    - getState : You should not have to modify the code in the getState function. This function takes to parameters; the current state, and the position you are interested in. The function then returns a 0, 1, or 2 to indicate that in the specified state, the specified positon is empty, contains an X, or contains an O, respectively.
    - fprintState : You should not have to modify the code in the fprintState function. This function takes the current state as a parameter, and prints out the board that corresponds to that state to standard error.
    - p2c : You should not have to modify the code in the p2c function. The p2c function takes two arguments; the current state, and a position, and returns the character that represents what is the square for the specified state and position, either a "_" for a blank, an "X" for X,and an "O" for O.
- Makefile – a make file that contains several targets, as follows:
    - ttt : The name of the tic/tac/toe executable file, built from the ttt.c file using the gcc compiler
    - test : A pseudo-target to invoke the ttt executable file with an X parameter, and then with an O parameter.
    - clean : A pseudo-target to remove the ttt executable file
    - submit : A pseudo-target to create the tar file to submit on blackboard. NOTE: If you do not use "make submit" to create the tar file, you will get points deducted for not following directions. Furthermore, please run "make submit" on an LDAP machine. We depend on using the LDAP userid as a part of the tar file name we create. This is done automatically on an LDAP machine, but will be incorrect if

you run "make submit" in other environments! If you submit a tar file created somewhere other than on an LDAP machine, you will get points deducted for not following directions.

When you first untar the file, cd to the proj1 directory, and try building and running ttt. Note that as delivered, the "myTurn" function simply returns it's input state. That means that when your program has a turn, it doesn't do anything (which makes it pretty easy to beat.)

Your job for this project is to re-write the myTurn function and to test and debug that function. You may add functions to ttt.c, or modify the functions that are already available. Use the internet to look up anything you are not sure about. If you still can't figure it out, feel free to ask the TA's or the professor about how the existing infrastructure works, or about how to resolve a specific problem.

Standard Input, Standard Output, and Standard Error

We have not yet talked about the "standard" input and output (IO) streams in C, but this project makes use of those streams. Most of the interaction with these standard IO streams has been provided to you in the infrastructure, but it's worth describing them in slightly more detail. For more detailed information, see https://en.wikipedia.org/wiki/C_file_input/output, or look at The C Programming Language (Kernighan and Ritchie) chapter 7

In UNIX and in C, every program has one input IO connection called "standard input", and two output connections called "standard output" and "standard error". In C, IO (including file IO) is handled by a concept called a "stream". In C, the three standard IO connections are all streams; the standard input stream, standard output stream, and standard error stream.

Normally, C connects the standard input stream up to your keyboard. With that connection, when your program requests input from standard input, then your terminal opens up (the cursor blinks), and the program waits for you to type something on the keyboard. Whatever you type doesn't get to your program until you hit the "Enter" key. When hit the "Enter" key, what you typed goes into the standard input stream, and is available to be consumed by your program. If you type "Ctrl-D", that sends an "End of File" signal to your program. (Note that a "Ctrl-C" key sends an immediate "kill" signal to your program. You can use this if your program ends up in an endless loop.)

The standard output and standard error streams are normally connected to your terminal. If you write something to the standard output stream, it will show up on your terminal. Same with the standard error stream. Why are there two streams? Because UNIX provides the capability of writing information that may be consumed by a program to standard output, and the information that should be consumed by a human can get written to standard error.

The tic-tac-toe program in ttt.c uses the standard stream capability so that we can automate playing against your program. In the automation mode, we override the default connection of standard input by connecting it to a program, and override the connection of standard output by connecting it to a program as well. However, we discard the standard error stream.

We will talk more about standard IO and C streams later in the semester.

Academic Honesty

You may look on the web for ideas and concepts that go in to making your own implementation of this project. If you do so, please include a comment in your code. e.g.

```
/* Concept from: https://xyz.code.org/tictactoe */
```

It is not very likely that you will be able to copy and paste code from the internet into your code… any code from the internet probably will not be compatible with your infrastructure. Usually, it's much more effective, and definitely a much better learning experience if you write your own code from scratch.

Feel free to discuss this project with other students. However, ***DO NOT COPY CODE!*** Your code will be compared with all other students' code. The compare tool looks at the semantics of the code – not just the bytes, so it can detect copied code even if you add comments, change variable names, etc. If your code compares too closely with any other student's code, then both the copier and the student who wrote the code that got copied will get a zero grade on this project. If you use the same concepts, but write your own implementation of the code, there will be enough of a difference in the result so that you will not be accused of cheating.

Submitting your Code

When you have finished coding and testing for this project, if you did not develop your code on an LDAP machine, please copy your code to an LDAP machine, and make sure it compiles and tests correctly in the LDAP environment. Then, on an LDAP machine, run:

```
make submit
```

in your project directory. This will collect your source code and put it in a tar file called "<userid>_proj1.tar.gz", where "<userid>" is your LDAP user ID. (Note that your userid is encoded in tar file itself as well as in the file name, so just renaming the tar file is not good enough. You MUST run make submit on an LDAP machine.) Then upload the <userid>_proj1.tar.gz file onto blackboard in the Project 1 submission area. Note that you may submit as many times as you want. The TA's will disregard all but the latest submission.

Project 1 Grading

Project 1 is worth a total of 100 points. After the due date, your <userid>_proj1.tar.gz file will be downloaded onto an LDAP machine, untarred, and compiled using a Makefile similar to the one you have been using. You will start with 70 points. If you failed to follow directions, and upload a tar other than one created by running make submit on an LDAP machine, you will get a 10 point deduction. If your code gets a compiler error, the TA or professor will try to fix your code. If there is a simple fix, we will make that fix, subtract 20 points, and continue testing. If there is no simple fix, you will get a grade of 25 for this project. Once you code compiles, 10 points will be subtracted for each type of compiler warning in your code. Then, you program

will play automatically against a dumb (basically random) strategy. If your code produces the correct states and correct results against the basic strategy, whether you win, lose, or tie, you will get 10 points, and you will get entered in the "tournament". All entries in the "tournament" will be played against 10 other tournament entries, selected at random. For each win against a random player, you will get +2 points. For each loss, you will get -2 points. For each tie, you will get +1 point. This can be summarized as follows:

Situation	Grade
No Submission (>10 days late)	0
Unfixable Compiler Error	25
Fixable Compiler Error	Base = 50
No Compiler Error	Base = 70
Incorrect Submission (bad tar file)	Base = Base - 10
Late Submission	Base = Base – 10 x (# days late)
Compiler Warning	Base = Base – 10 x (types of warnings)
Problem running against simple strategy	Base
Tournament Player (successful w/ simple strategy)	Base + 10 + (#ties) + 2 x (#wins)

Note: This is the first of four projects, and your lowest project grade will get dropped. So don't despair if you don't do well on this project… that means you will just have to work harder on projects 2, 3, and 4.