ADS Coursework

Kieran Burns
40272382@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1 Introduction

The aim of this coursework was to make a C application to allow users to play a game of Tic-Tac-Toe (Naughts and Crosses) from the Command Line. The features I implemented were; a helpful menu system to explain to users how to operate the application, a name input system to help with player identification, the 2 player Tic-Tac-Toe game with "Win" and "Draw" conditions, an "Undo" function to allow players to revoke actions made and a system to log any match started, with a way to view/replay/continue any previously started match.

2 Design

Prior to starting development of the project, I drafted a short text document breaking down the tasks of the project and taking note of the various ways they could be done. Due to my slight unfamiliarity with the C programming language, I did have a little difficulty initially deciding on which methods would likely be the most time efficient and simplest to implement, as my time allocated to the development of the project was far more limited than I would have otherwise liked.

The first task from the broken down list was a Main Menu screen that would serve as a "central hub" for each of the features of the program which I intended on implementing. Each action taken from this menu was intended to span from, and lead back to the menu. I started this by creating a function to display a large amount of formatted text. This was done with a series of "printf" commands, containing all of the desired formatted content as it was the fastest and most simple way of displaying the static text.
I then used the "scanf" command to take in user input to then be validated and return the next action to be taken by the main menu. The first of the implemented actions was the "play" action, used to begin a game. The final result of the main menu screen can be seen in **figure 1**.

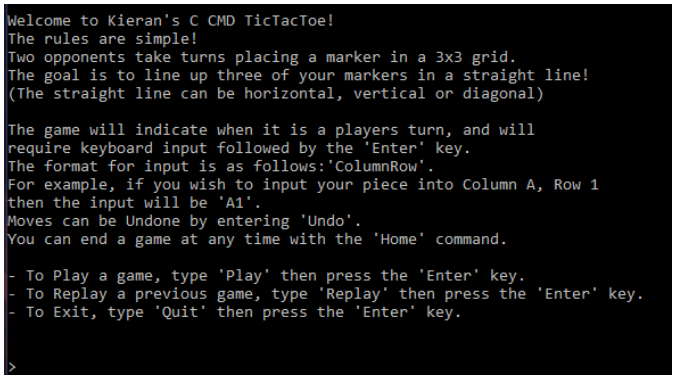

figure 1 - Main Menu

The next task was to create a username input screen. This screen would be the first thing that is accessed when the user submits the "play" command to the main menu. I decided the input would ask for player 1's name, take in the input, ask for player 2's name, take in the input, then have a confirmation for both names. This would allow both users to ensure their names were correctly input. This can be seen in **figures 2 - 4**.

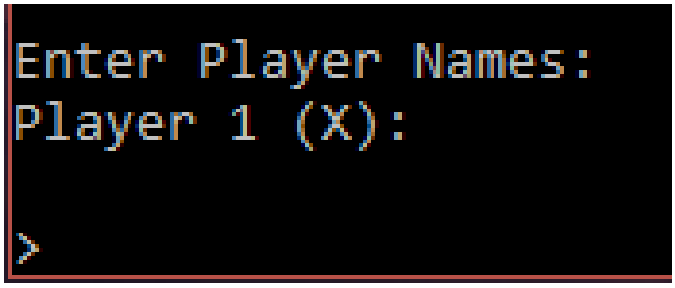

figure 2 - Player 1 input

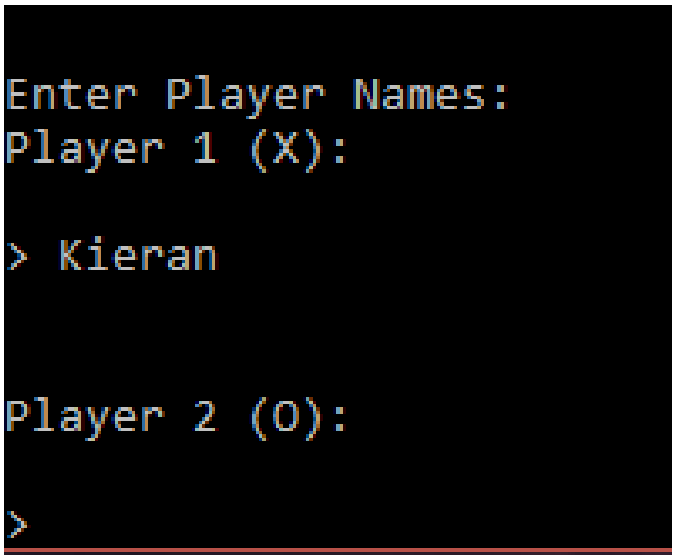

figure 3 - Player 2 input

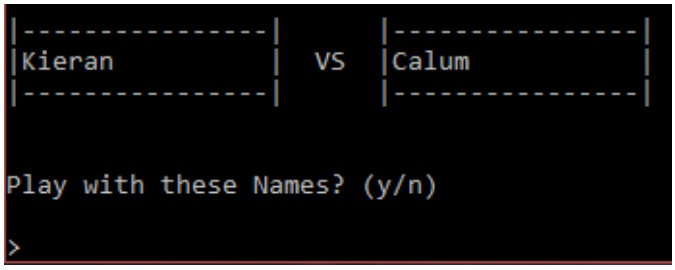

figure 4 - Player Confirm

I also enforced a "less than sixteen" character rule to allow for convenient formatting and light memory usage for the usernames, which were both stored in their own separate character arrays for ease of access for later in the program. The character rule validation can be seen in **figure 5**.

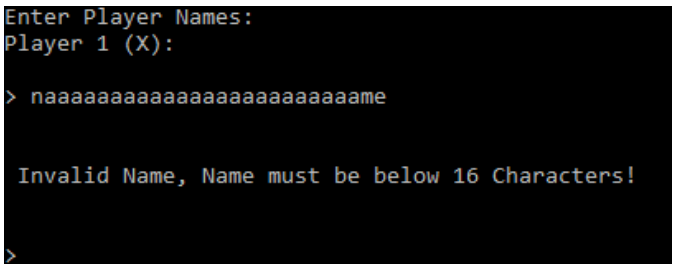

figure 5 - Length Validation

The next task was implementing the actual gameplay loop. In regards to turn order, for the sake of helping ease the work needed within the time constraints, I decided to implement the rule that Player 1 is always represented by X, and X always goes first. With the implementation of the game loop I had to first consider the game board, and how it would be stored. The board is a three by three grid, meaning nine values would need to be stored, this lead to the consideration of using a standard single-dimension array of size nine, and the consideration of using a two-dimensional array of a three by three size to more accurately represent the shape of the game board, making it easier to keep track of which values are going to which board positions. I decided to use the two dimensional array due to it's convenience. I also considered the use of a linked list to store the state of the entire board on each turn, however this seemed much less efficient then simply storing each turns action, so I decided to avoid that idea.

After deciding how to store the game board, I then began to consider the display aspect, I ended up using a series of specifically formatted "printf" statements to create a board shape. I then filled each spot in the board array with a "." character. This would be used to represent an open space on the board. The board would be displayed on screen, followed by a prompt asking the player who's turn it is to make their action for the turn. The board on the first turn of the game can be seen in **figure 6**.

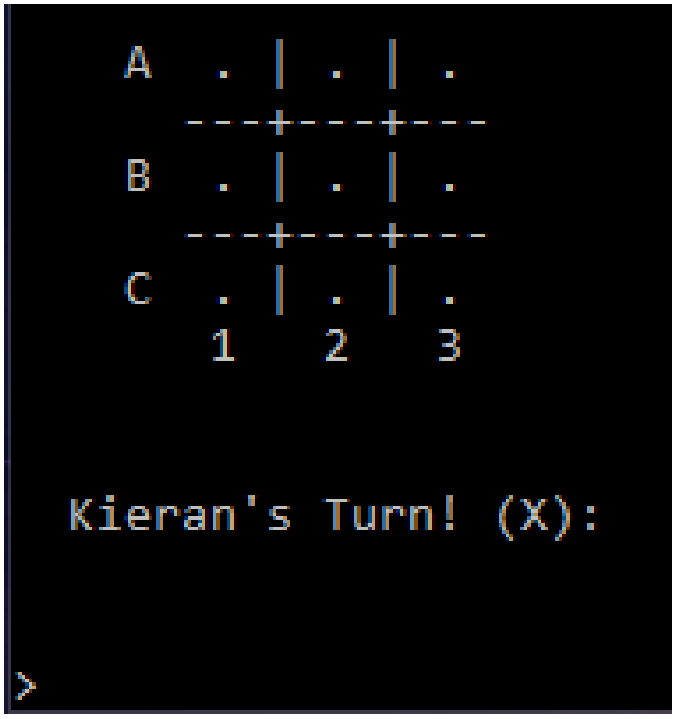

figure 6 - Turn 1 (Before action)

A "scanf" statement would take in the users input, where it would be validated. If the user's input was valid and in the right format, then the action would be taken and turn would change. This can be seen in **figure 7**.

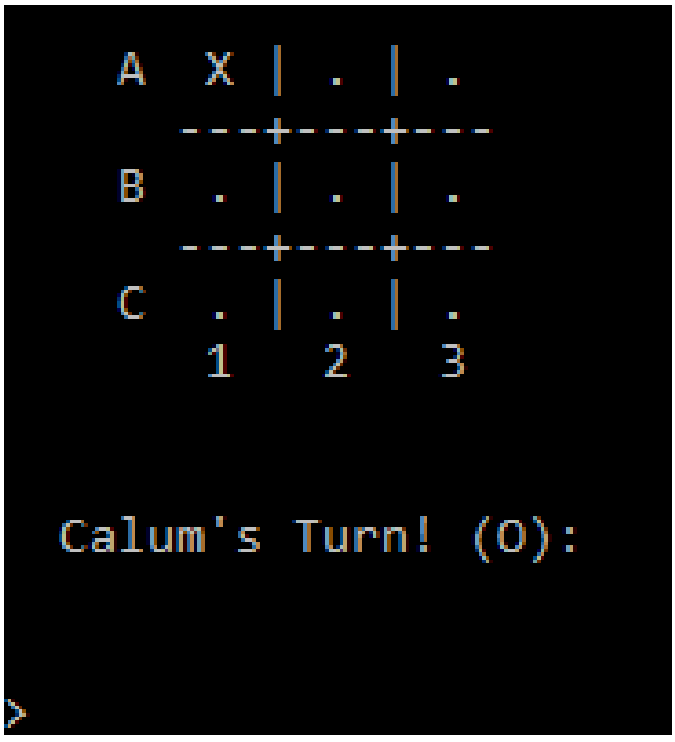

figure 7 - Turn 2 (After Taking Turn 1 Action "A1")

If a user inputs an invalidly formatted command, then the program will return an error and make them re-enter their input. This can be seen in **figure 8**.

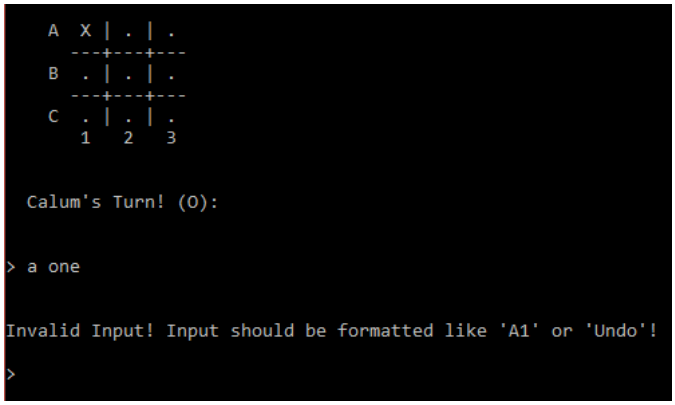

figure 8 - Turn 2 (Invalid Input)

If a space is already taken and a user tries to place their marker there, they will get a different error message, as can be seen in **figure 9**.

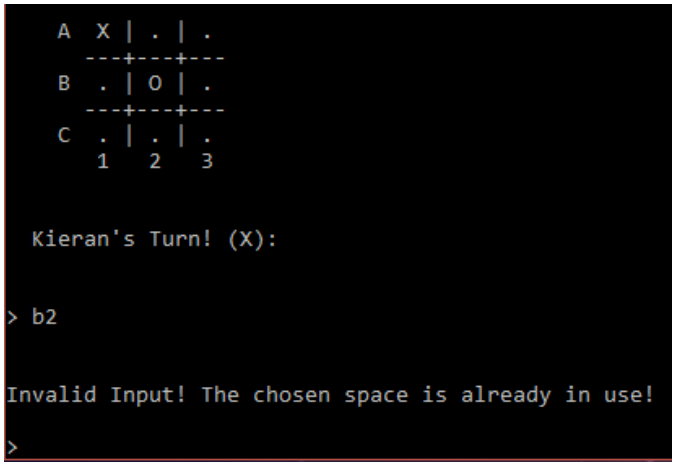

figure 9 - Turn 3 (Trying to take already held space)

At the end of each player's turn, the program checks a series of logic statements to see if a winner has been decided, which can be see in **figure 10**, or if the game has run out of playable spaces and must be called a draw, which can be seen in **figure 11**.

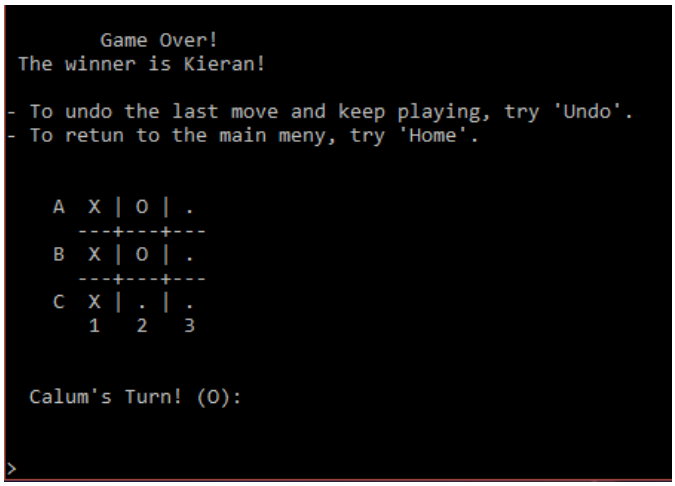

figure 10 - Turn 6 (Player 1 wins)

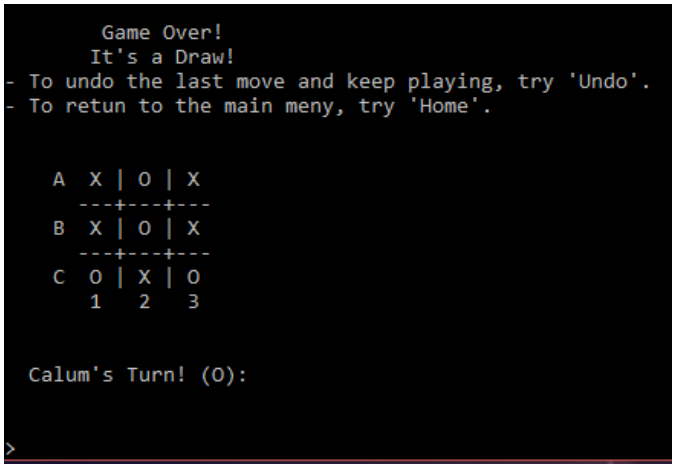

figure 11 - Turn 10 (Game is a draw)

After implementing the game logic, an input for "home" was added, which escapes from the game loop and returns the player to the main menu.

The next feature I intended to implement was the redo feature, however, given the data types used up to this point, I wasn't entirely sure on how I wanted to actually implement the feature, so I decided to put it on hold temporarily, and work on the logging and replay feature next. In regards to the logging side of the feature, I decided to simply use ".txt" files to contain logged games, this started with the creation of a "hub" file, to store the locations of each created log file, as each game would have it's own file. If a version of the hub file did not exist in the same directory as the game ".exe" file, one would be created. All log files would be saved to a folder named "replays" by default.

The actual logging to file format was quite simple, each item logged would have it's own line. Both Player 1 and Player 2's names would be logged as soon as they are confirmed, along with the creation of the log file. Whenever a valid move is made (for example, a1), the move would be added to the log file. The "Home" command and the later implemented "Undo" command would not be logged, neither would any invalid input.

For the replay feature, off the back of the logging, a second path from the main menu was created when the "replay" command is entered. This command leads to a sub-menu containing all logged games for the user to choose from. This menu can be seen in **figure 12**.

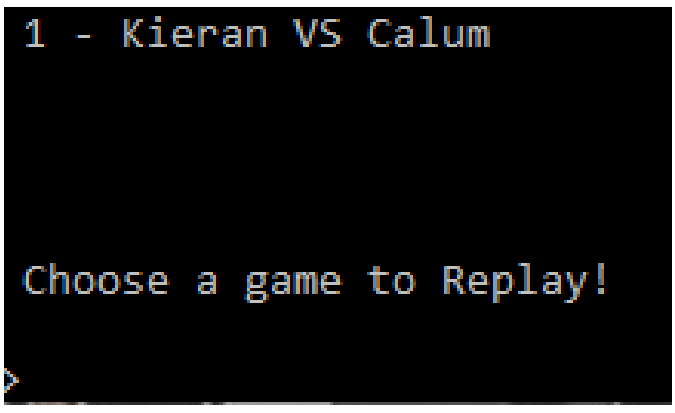

figure 12 - Replay menu.

The content of the replay menu depends on how many files are linked to by the hub file.

Once a file is selected from the replay menu, the file is loaded in using the main game loop. This load however, skips over the name input section and instead sets the names to lines 1 and 2 of the file consecutively, then sets the game board based on the rest of the lines in the file. If no other lines exist after the names, a blank game board will be loaded. Any changes made to the game board during the replay function will be saved, as the function also plays the part of a "resume game" option. From here, the game can continue as a normal game would, replay and play intersect at this point, with the same conditional checks and save features.

At this point in development, I was beginning to run out of time to add new features. I decided to make a simplistic undo system using the save file by removing the last logged move from the board, then removing that line from the log file. This would inevitably return the game board to it's state during the previous player's turn, as if no move was made. I would have liked to use a Linked list to store each game move to be cycled back and forth through, to allow implementation of both undo and redo, however, this was pushed aside due to the time constraints. To allow the file side of the undo feature to work, a temporary file would need to be created, filled with all of the contents of the current log file, then both files would need to be closed, the original log would be deleted, the temporary file renamed to the log file, then lastly the new log file opened. The process sounds far more complicated than it actually is to program, and with the addition of an integer file line counter, this was a very simple feature to implement. An example of the undo feature can be seen in **figures 13 & 14**.

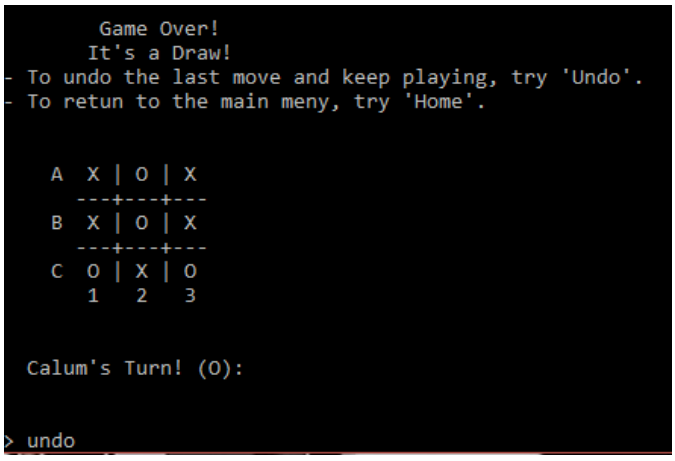

figure 13 - Turn 10 (Before Undo)

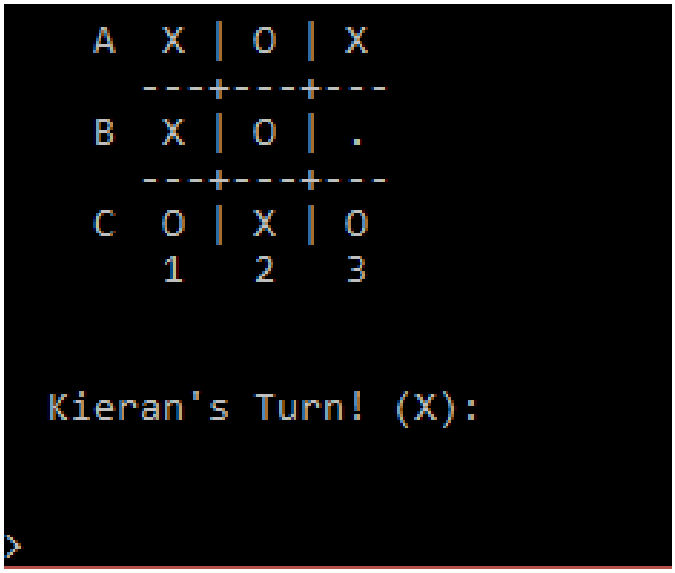

figure 14 - Turn 11 (After Undo)

Undo can be performed up to as many times as there are taken spaces on the board. If the undo action is attempted, but there are no more taken tiles, an input error message is displayed and the player's turn continues. This can be seen in **figure 15**.

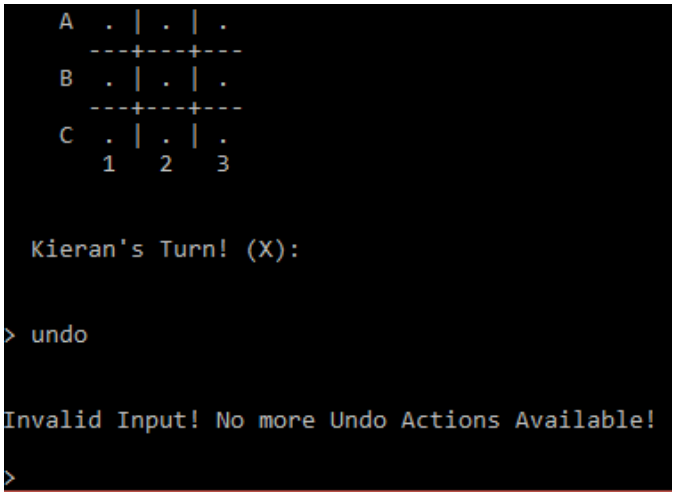

figure 15 - Invalid Undo

3 Enhancements

Due to the time constraints placed upon the project, there are various features I would have liked to have added or changed.

The first addition I would have liked to add was a "Coin-flip" type feature to randomly decide which player went first. This would have been done by generating a random number between 1 and 100 (alternately 1 and 2), using the "rand" function in C, then having all numbers within a certain part of the range represent each player(for example 1-50 = X, 51-100 = O).

Another addition I would have liked to add was the redo feature. Due to the way the undo feature was implemented, redo would need significant additional work to

be added. If I were to do so, I would have to save each position effected by undo in some sort of buffer (likely a doubly linked list), to then be either called if redo is called to "Undo the Undo", or cleared if a move other than undo is called, clearing the redo path.

The last of the major additions I would have liked to add is some sort of player 2 CPU. This could be selected by entering CPU in player 2's name slot during the name input screen. If selected then my initial idea would be to have the CPU play their turn by randomly selecting an open tile with the "rand" function. Some form of basic adversarial AI could also be used here, however this would be far more difficult to implement while still keeping the game fair, as a well made adversary could easily make all of the "best" moves, and be too difficult/impossible to beat, where every game would end in a loss for the player or end in a draw.

One major alteration that I would have liked to implement if I had the time would be a much neater program structure. Looking at the code in retrospect, there are many parts where sections of code could have been made in to separate functions to avoid using the same small blocks of code two or three times. This could also help break down the "game" function, which ended up being much longer than anticipated and left the function somewhat messy and hard to read at times.

4 Critical Evaluation

I believe the final product is a sufficiently well produced piece of work. It uses as many techniques as I know to save time and effort where possible. However due to this approach, the code is notably less versatile than it could have been. The program as a whole could have been more versatile if more time was sunk in to making each of the elements of the code less distinctly linked. This versatility issue is most notable in the case of the undo feature, which used direct communication with a file, rather than first using a linked list as a buffer, which could have been used to easily implement the desired redo feature. Outside of the issues that can be found within the code cleanliness and use of less versatile data structures and functions, I am happy with all of the other implementations within the code.

5 Personal Evaluation

During the process of doing this coursework, I learned a lot about working under tight time constraints, and the sacrifices and corner cutting that needs to be made to still produce a "final product" with said constraints in place. I also learned more about using the C language, especially on the file I/O side of development, which despite not being particularly complicated, was still unfamiliar to me at the start of the project.

References

docs.microsoft.com :-
Help finding more information on any/all Error codes encountered.

www.programmingsimplified.com :-
Additional Background info on File I/O in C, especially for use in Renaming and Deleting Files.

www.tutorialspoint.com :-
Help in finding the "C version" of commands which I wanted to use but was unsure of from the C Standard Library.