NWEN241: Systems Programming

C Programming Lab

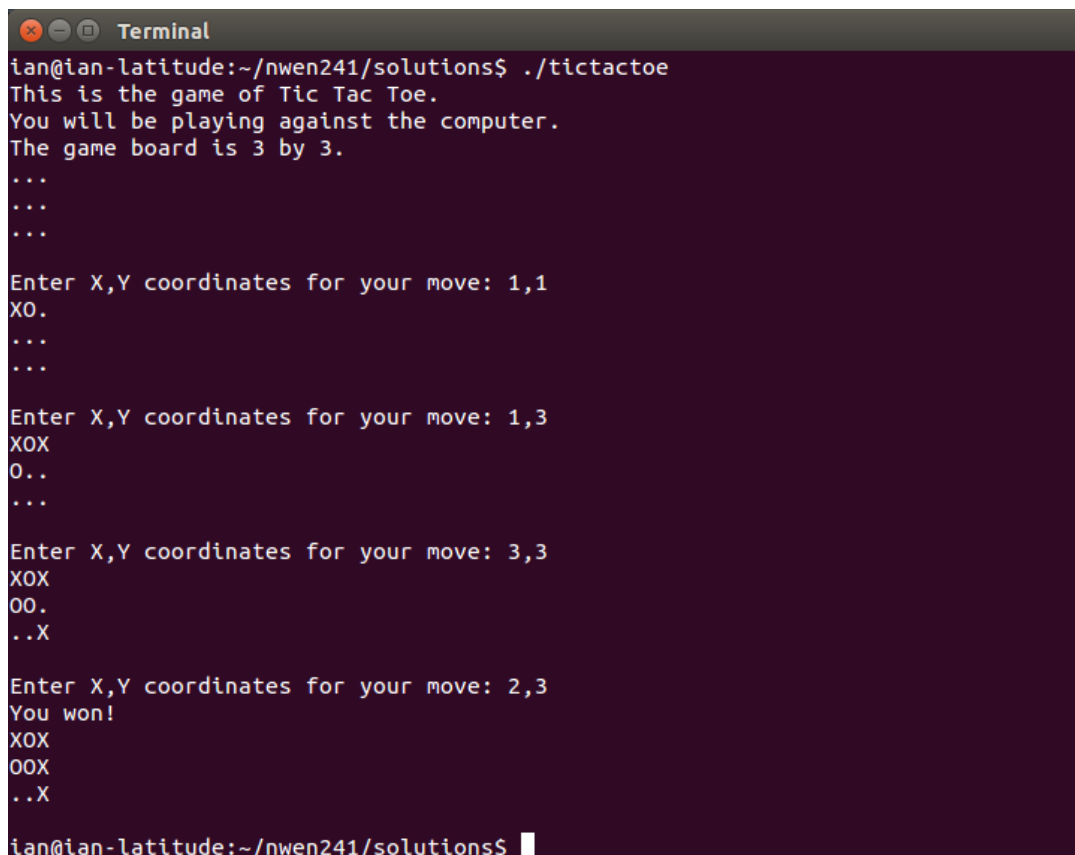## Due: Sunday 7th June @ Midnight (late days still apply)

Tic Tac Toe

The `tictactoe` program is a simple board game written in C. The standard rules have been implemented:

- Each player takes turns placing either a `O` or `X` on the board.
- The game ends when either three-in-a-row is obtained or no more moves are possible.
- A draw is declared if no more moves are possible and neither player has achieved a three-in-a-row.

The two players are a human and computer. The computer does not use any strategy to win and simply chooses the first empty place when making its move. **Note that improving the computer's game playing is not the aim of this lab.**

The screenshot below shows the program being played.

Part 1: Generalising the Game (60%)

The `tictactoe` program is already functioning but it is not general.

We would also like to remove the reliance on global variables used to track the state of the board (**board**) and whether the game is complete or not (**winner**).
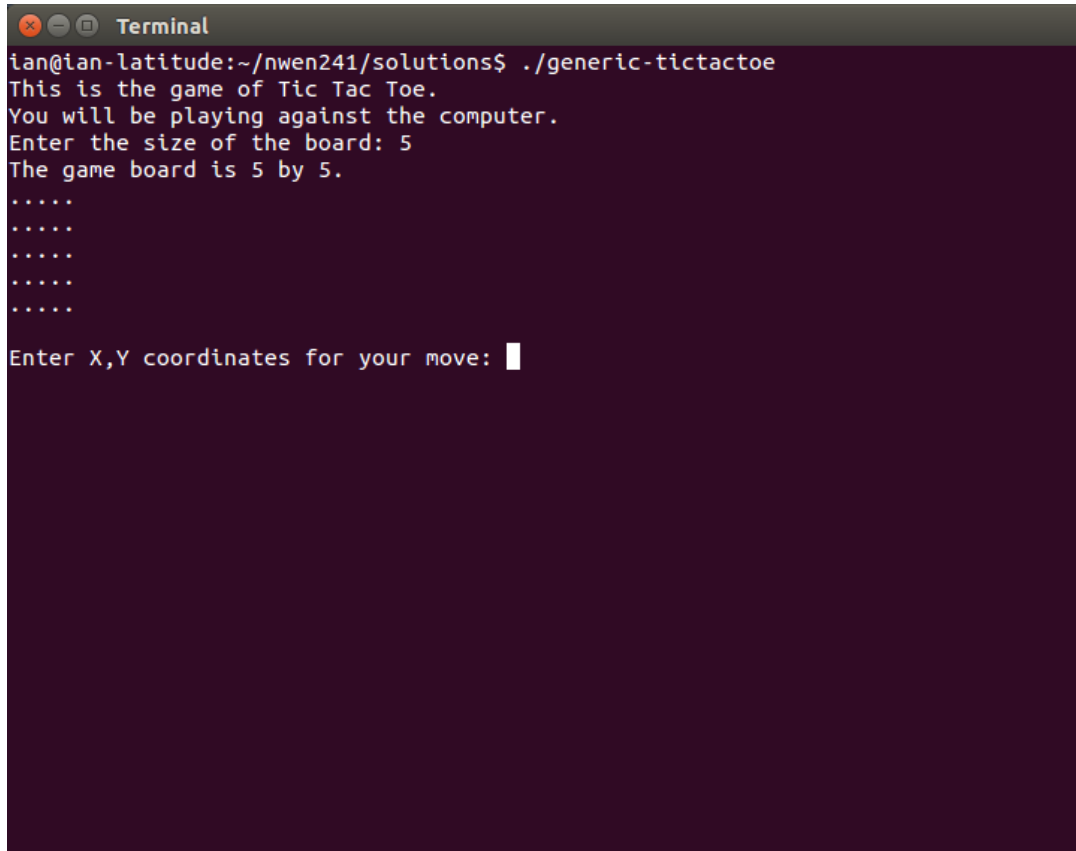
The code for the program is here:

- tictactoe.c

- tictactoe.h

Note that the `.c` program includes the `.h` when compiled (this header file contains some useful pre-defined constants).

Write a version of the program called **generic-tictactoe**.

As shown below, when you run the game you should be able to allow the game to played on any size of grid that is specified at runtime. For example:

You should declare a `struct` that represents all the state associated with a game:

```
typedef struct TicTacToe
{
     int size; // this is the size of the game board
     int **board // this is the game board
     int winner; // who won
} TicTacToe;
```

The pointer `**board` should be assigned at runtime to a 2D array (an array of array of ints) using **malloc**. You must do this in such a way that you can refer to any element of the board using the array notation, for example `board[0][1]`.

For a hint on how to do this checkout this useful resource:
http://web.cs.swarthmore.edu/~newhall/unixhelp/C_arrays.html#dynamic2D

When the program ends you must make sure that you use **free** to allow the OS to deallocate the memory.

You can check for memory leaks using valgrind which is installed on the ECS workstations. A short tutorial on using is here:

http://www.cprogramming.com/debugging/valgrind.html

However, the main use case is to run it with **memcheck** as shown below:

```
int check(TicTacToe* game);
```

```
void init_game(TicTacToe* game, int size);
void free_game(TicTacToe *game);
int player_move(TicTacToe* game);
void computer_move(TicTacToe* game);
void print_game(TicTacToe game);
char tokenstr(int token);
void print_result(TicTacToe game);
```

All of these functions (with the exception of **tokenstr**) will need to be modified to work correctly with a variably-sized board.

In particular, `check()` should check for N tokens in a row, column or diagonally instead of checking for a fixed number. For example, for a 5x5 board the winning condition is met by five tokens in a row rather than simply three as is the current default.

Part 2: Experiment with UNIX Pipes (5%)

The aim of this part is to implement a client-server version of the tictactoe program using UNIX named pipes as a communication mechanism.

A named pipe is like a file that supports multiple readers and writers. Unlike a file it doesn't actually store anything on the file system, instead everything is held in memory. This makes communication very fast and so pipes are often used by system programs to communicate with one another. For example, I'm working a postgrad student who uses pipes to connect network daemons that he is developing.

A short tutorial on named pipes is here:

http://www.cs.fredonia.edu/zubairi/s2k2/csit431/more_pipes.html

The interface for using pipes is the same as for files (which you cover next week!). The syntax is pretty similar to that used with Python and the general idea is that you can **write** and **read** from a file using a syntax very similar to **printf** and **scanf**.

One thing to keep in mind is that pipes are unidirectional.

In the code below we use a pipe to communicate between the programs **writer** and **reader**.

- reader.c
- writer.c

Download these and compile them:

```
gcc reader.c -o reader
gcc writer.c -o writer
```

When executing these you should have two terminal windows open.

In one window you should execute **reader** (it will block waiting for **writer** to create the pipe and write to it) and in the other you should execute **writer**.

Terminal one

```
./reader
```

Terminal two

```
./writer
```

All being well you should see **Received: Hi** in the **reader** window.

The example shows a string being sent, sending integers is easy as well. For example,

```
int i = 10;
write(fd, &i, sizeof(i)); \* send the int *\
read(fd, &i, sizeof(i)); \* read the int *\
```

Once you have the basic code working you should extend the code so that the **writer** program sends **Hi** and waits for the **reader** program to send **Hello** back before exiting. The **writer** should print what is received to the console.

This is achieved by extending the **writer** to create a pipe that can be used by the **reader** program to write **Hello**.

Note that you should make sure you use **unlink** to remove the pipes when done.

Part 3: Use Unix Pipes to Create a Client–Server Solution (25%)

The aim here is to separate out the game code into a client **t3client** that will act as the interface to the server **t3server** that will implement the game logic and track the state of the game.

We have provided the skeleton for both programs:

- t3server.c
- t3client.c

In both cases the **main** is partially complete and you must complete the rest of the functions.

The flow of control is similar to that of the generic solution you completed above for the server, the main difference is that you pass in file descriptors that allow the server to read commands from it's pipe and send results back to the client via the client's pipe.

For the client it is a little different, the flow should be something like this:
- Send the server the size of the game board.
- Loop until the server tells us the game is over:
    - Get the player's move and send to server (repeat until a valid move has been made).
    - Wait until the server tells us if the move led to the human winning, the computer has made a winning move or we have a draw.

The client communicates with the server by writing to a pipe called **serverpipe** and reading from a pipe called **clientpipe**.

The server communicates with the client by writing to a pipe called **clientpipe** and reading from a pipe called **serverpipe**.

Why two pipes? Because pipes are unidirectional. This is a bit different from files and from network sockets (you use sockets in NWEN243).

Submission

Your program code should be submitted electronically via the **online submission system**, linked from the course homepage. You should follow the following guidelines:

- Your submission is packaged as a zip file, including the source code.
- The names of provided variables, functions etc. remain unchanged.
- The code submitted runs correctly on the ECS machines so the tutors can check your implementation.

You should include a **README** with any helpful information running your code and this README should specify which parts are implemented correctly.

Assessment

This lab will be marked as a letter grade (A+ ... E), based primarily on correctness (90%) and style (10%).

Correctness will be assessed as follows:
- Part 1 (60%) - how well does your generic solution work as specified by the requirements.

- Part 2 (5%) - did you implement two way communication as specified by the requirements.
- Part 3 (25%) - how well did you implement a client-server solution as specified by the requirements.

In general if the code works correctly you automatically receive full marks, if it fails we will provide partial credit.

The qualitative marks for style are given or the following points:

- Commenting of the body of functions.
- Consistency of indenting.
- Understandability (efficient code is good but it must also be readable, comments help!).

Extension: Create a Multiplayer Solution

This is extension work that won't count towards your final grade but is a natural extension of this project.

At present your solution can only allow one client to play with the server at a time (actually you can start multiple clients but they are all playing the same game).

The aim here is to allow multiple concurrent games to be played at the same time.

To solve this you would need to extend your program as follows:

- Allow clients to specify a private pipe to use for communication with the server (you can assume the user can provide a unique name).
- Keep track of each game in a dynamic data structure with appropriate use of `malloc()` and `free()`.
- Modify the communication between the clients and server to include the client name so its possible to track which move belong to which game!
- Never shut down the server to allow clients to join at any time.