Unix & C Programming Assignment
Tic-tac-toe Game

Reece Jones

Contents 1

[0] Overview

The goal of this project is to produce a basic, but functional, M-N-K tic-tac-toe game, as a command-line application. The game supports user-provided settings for board size and win condition, as well as game action logging. The game is written in C89 and builds using Make.

[1] Usage

[1.1] Command line

The command-line usage of the application is as follows:

```
tictactoe <settings_file_path>
```

where `tictactoe` is the path to the application's executable (typically `./tictactoe` from the same directory as the application), and `<settings_file_path>` is the path to the file containing the settings to use.

[1.2] Settings file

The settings file is an ASCII plaintext file, with one option specified per line. The syntax of a line is as follows:

```
<option>=<value>
```

where `<option>` is a single character (case-insensitive) specifying the option to set, and `<value>` is the desired value of the option. Whitespace before and after `<option>` and `<value>` is ignored. Lines consisting of only whitespace are also ignored.

The following table outlines the allowed options:

Option character	Description	Value type	Required?
M	Width of the game board	Integer >0	Yes
N	Height of the game board	Integer >0	Yes
K	Consecutive tiles to win	Integer >0	Yes

The application loads and validates the settings file immediately upon startup. If there are any invalid values, error message(s) will be shown and the application will abort.

Some values or combinations of values will still allow the application to run, but may affect usability. These will produce warnings:

K=1 – allowed, however the game will always be won on the first move.

K > M and K > N – also allowed, however the game cannot be won and will always end in a draw.

[1.3] Main menu

After loading the settings, the application will enter the main menu, which contains the following options:

Start a new game - starts a new game of tic-tac-toe using the current settings.

Edit settings (*editor mode only*[1]) - allows modification of the application settings.

View current settings – displays the current application settings.

View game logs – displays the game logs from this session.

Save game logs to file (*not in secret mode*[1]) – exports this session's game logs and settings to file.

Exit application: terminates the application.

[1]*See section 2.1 for explanation of the editor and secret modes.*

Each menu option is identified by an integer, which the user enters in order to make a selection.

[1.4] Gameplay

After choosing the "Start a new game" option from the main menu, the application runs a standard game of M-N-K tic-tac-toe. Player "X" always moves first. The player will be prompted for a coordinate in the format

```
<x>,<y>
```

where <x> is the column number and <y> is the row number of the tile to be marked for that turn. `0,0` is the top left corner of the board.
The following is an example rendering of a 5x4 board, with 3 tiles already marked:

```
---------------------
| X |   |   |   |   |
---------------------
|   | O |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   | X |   |
---------------------
```

Once a player gets *K* (the 'K' option from the settings file) tiles in a row (horizontally, vertically, or diagonally), they have won and the game ends. The application then returns to the main menu.

[1.5] Logging

For each game played, the players' actions are logged. These logs can then be displayed or saved by selecting the "View game logs" or "Save game logs to file" options from the main menu.
The game logs are rendered as a sequence of the following format:

```
GAME <N>
   <turns>
```

where <N> is the game number from the start of this session, and <turns> is a sequence of player turns, formatted as follows:

```
Turn: <M>
Player: <P>
Location: <x>,<y>
```

Where <M> is the turn number, <P> is the player whose turn it was (either 'X' or 'O'), and <x>,<y> is the coordinate of the placed tile.
Additionally, when saving the game logs to a file, the application settings are prepended to the output, in the following format:

```
SETTINGS:
   M: <M>
   N: <N>
   K: <K>
```

Where <M>, <N> and <K> are the application settings values.

[2] Build system

The application is built with Make, using the makefile in the project's root directory.

[2.1] Main application build

The Make target `tictactoe` creates the application executable. It can be invoked with

```
make tictactoe
```

which will produce the executable `tictactoe` in the project's root directory.
There are two build options available, which modify the behaviour of the application:

`EDITOR_MODE` – when set, enables the "Edit settings" option in the main menu.
`SECRET_MODE` – when set, disables the "Save game logs to file" option in the main menu.

These are specified by assigning them a value in the Make command, for example:

```
make tictactoe SECRET_MODE=1
```

[2.2] Unit test build

Alongside the main application, there is a separate executable available to build that unit tests code for the main application. It is also built with Make, under the `tictactoe_test` target:

```
make tictactoe_test
```

which will produce the executable `tictactoe_test` in the project's root directory.
Usage of the previously mentioned application build options will not affect this build.
Please see section 4 for more information on the unit tests.

[3] Application design

The application's code is divided into several modules which each serve a logical, individual purpose.
The source code files referenced here are located in the `src/main` directory.

[3.1] Main module (`main.c`)

The main module is the entry point of the application and contains only a small amount of code/logic. It simply validates the command line arguments, invokes the settings module to read and validate the settings, and invokes the main menu (aborting if any errors occur along the way).

[3.2] "board" module (`board.c/.h`)

This module defines the tic-tac-toe game board and associated operations.

The primary object is the `GameBoard` structure, which represents the game board and contains its data. Functions are provided to properly create and destroy this structure, in order to prevent indeterminate values and memory leaks. Other functions provide various operations on the game board, for example accessing specific tiles (with bounds checking), checking if a player has won, displaying the board, etc. Effort is made to encourage checking the state of a `GameBoard` object through these functions rather than accessing its data members directly, in order to prevent bugs (particularly, out-of-bounds memory access).

[3.3] "common" module (`common.c/.h`)

Contains miscellaneous, generic utilities required in the main application. The purpose of its contents is straightforward and are sufficiently explained in the in-code documentation.

[3.4] "interface" module (`interface.c/.h`)

Handles the user interface, user interaction, and associated high-level logic.
It provides only one public interface, the `mainMenu()` function, which handles the application's main menu. The workings of the main menu are kept internal and private since they are not required anywhere else in the codebase.

[3.5] "linked list" module (`linked_list.c/.h`)

This module provides an implementation of a generic linked list data structure. It was copied almost exactly from my submission for practical worksheet 7. The linked list is required for the storage of the game logs (see section 3.6).

The implementation in this module is a typical linked list and should be self-explanatory. The only noteworthy point is that care was taken to be unambiguous about how the generic data is handled in the linked list with respect to deallocation. I have generally taken the approach that the linked list should not "own" the data that is stored in it – therefore it was important to remember that the calling code *must* handle the data allocation and deallocation

correctly. None of the remove methods free the data, except for `listFreeAndRemoveAll()`, which is provided for convenience (and has a verbose name so it is obvious it does free the data).

[3.6] "logging" module (`log.c/.h`)

Handles the logging of game actions, and the handling of those logs.

I have taken the approach that the module is stateful and internally stores and handles the game logs. They are stored statically, that is, they persist in the logging module for the duration of the program instance. A purposely limited public interface is exposed which provides simplified, high-level operations.
Perhaps this is not the best approach for a larger and more complex application, but in this case it simplifies the usage significantly. Additionally, making the log representation private to this module reduces the chance of bugs, particularly since the logging process involves a considerable amount of dynamic memory allocation.

The game logs are stored in a linked list of `GameLog` objects, inside each there is a linked list of `PlayerTurn` objects. Using arrays was not an option, since it cannot be known in advance how many games will be played and logged, nor how many turns a game will last for.

The module provides the `writeGameLogs()` function to write the logs in textual form to a stream. This function simply traverses the game logs using the linked list's traversal function, which calls back a function for each `GameLog` object, inside which the linked list of `PlayerTurn` objects is similarly traversed and rendered.
Since the `writeGameLogs()` function accepts any stream, it may be used to render the game logs to standard output or to file.

The assignment specification wishes for me to discuss complications that arose while implementing the logging functionality. However, the process was mostly straightforward and unexceptional.

[3.7] "settings" module (`settings.c/.h`)

Handles reading, validating, and other handling of the application settings.

Settings are stored in the `Settings` structure.
The `readSettings()` function reads settings from a file (with large amounts of error checking and handling, of course).
The `writeSettings()` function renders settings to a stream in textual form.

The only noteworthy thing in this module is the settings validation functions, of which there are four. The reason there are so many is that each individual setting must be able to be validated separately for the "Edit settings" main menu option, as well as validated all together. The combinations of error states, warnings, and the requirement that messages are displayed to the user, unfortunately, in C, necessitated multiple functions for such a "simple" task.

[4] Unit tests

The project includes a suite of unit tests for the main application's code. Each module in the main application has its own unit test module, except for the main module (tested manually) and the "interface" module (also tested manually). Each public function in a module is tested.

To ensure maximum test coverage, it's recommended to run the unit test executable with a utility such as Valgrind.

The source code files referenced here are located in the `src/tests` directory.

[4.1] Test main module (`main.c/.h`)

This is the entry point of the unit tests. Its only purpose is to invoke each unit test module to allow them to run their tests.

[4.2] “test common” module (`common.c/.h`)

Contains general utilities for the unit tests. The purpose of its contents is straightforward and are sufficiently explained in the in-code documentation.

[4.3] “board” module test (`board_test.c/.h`)

Runs unit tests for the “board” module of the main application.

Testing of the basic board properties and cell access has been automated to cover as many test cases as possible (the number of tests generated can be adjusted in the test source file).
Particular attention was paid to testing the win checking function, which has somewhat tricky code for checking of the diagonals and could harbour many bugs.

[4.4] “common” module test (`common_test.c/.h`)

Runs unit tests for the “common” module of the main application.

Particular care was taken while testing the `readUntil()` function, since I/O in C often has quirks and edge cases. A variety of test data was used, which can be found in the `test_data/readUntil` directory.

[4.5] “linked list” module test (`linked_list_test.c/.h`)

Runs unit tests for the “linked list” module of the main application. The test code was copied almost exactly from my submission for practical worksheet 7.

All the unit test are automated and run a very large amount of test cases (which can be adjusted in the test source file) in order to test as thoroughly as possible.

[4.6] “logging” module test (`log_test.c/.h`)

Runs unit tests for the “logging” module of the main application.

The module has little to test. The primary cause for concern is memory leaks due to the nested linked lists of the game logs, so it is highly recommended to use a utility such as Valgrind here.

[4.7] “settings” module test (`settings_test.c./h`)

Runs unit tests for the “settings” module of the main application.

This module contains many test cases for the very large amount of code paths and error conditions present in reading and validating the application settings.
Particular care was taken testing the `readSettings()` function, where I have tried to cover as many combinations of valid and invalid data as possible. Test data for this function is located in the `test_data/settings` directory.