

C_LINGO

Roxanne van der Pol

1688823

23/09/2020



Welcome to C_lingo!

play

manual

about

Table of contents

1. What is it
2. Version control
3. Tools used and tested environments
4. Back-end
 - a. Lingo_words
 - i. Build instructions
 - ii. File IO
 - b. Lingo_server
 - i. Build instructions
 - ii. Important variables
 - iii. Initializing the socket and buffers
 - iv. Tour through the Lingo struct
 - v. Server calls
 1. GET
 2. POST
 - vi. Error handling
5. Front-end
 - a. Javascript
 - i. Global variables
 - ii. Functions
6. Sources

What is it

C_Lingo is a Back end / Front end combination that allows you to play a variation of the popular game Hangman, called Lingo, in your web browser. The back end is entirely written in the programming language C, while the functionality of the front-end is written in Javascript.

When starting the game, you will be given an unknown word that is five letters long, of which only the first letter is revealed. The goal is to guess the complete word. Per word you get five tries. When attempting to guess it, you lose a try and hints will be displayed: Yellow means that the letter is present in the word, just not in the right position; red means that the letter was guessed correctly, and blue means the letter doesn't occur at all. When guessing the word correctly, you are given a new word - this one being six letters long. When guessing correct again, a seven-letter long word is given, after which the cycle repeats: 5 -> 6 -> 7 -> repeat. When guessing a word correctly, you get points: $10 * (5 - \text{guesses_remaining})$, each win, which will count towards your final score.

w	o	r	t	h
w	o	r	d	s
w	r	o	n	g

```
--- HIGH SCORES ---
itworks..... : 00000050
c..... : 00000100
a..... : 00000050
b..... : 00000050
helloworld..... : 00000050
helloworld..... : 00000030
aa..... : 00000020
bb..... : 00000020
g..... : 00000020
h..... : 00000010
```

approximately one minute to do so, after which the game will deallocate its resources.

Per try you are given about ten seconds to make a guess. If you do not make a guess within that time you will lose the game. If your guess does not start with the first given hint letter you will lose the game. If you run out of tries you will lose the game. When you lose, your final score is given, and you are given the opportunity to enter your name into the high score if your score is higher than zero. You have

- when using the default settings -

Version control

Different iterations of the C_Lingo code are kept in the C_Lingo Github repository: https://github.com/RoxanneMango/C_Lingo. The repository has a developer branch and a master branch. The most stable releases can be found on the master branch, while the developers branch may have more implemented features and bugs. Use at your own risk.

Tools used and tested environments

The following tools were used:

1. GCC version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
 - a. Target: x86_64-linux-gnu
 - b. -> To compile the back-end C code
2. GNU Make 4.1
 - a. Built for x86_64-pc-linux-gnu
 - b. -> Automate the compiling process
3. Notepad++ v7.8.7
 - a. Build for 32-bit Windows 10
 - b. -> Write all the code
4. git version 2.17.1
 - a. Linux version
 - b. -> Version control
5. git version 2.26.2.windows.1
 - a. Windows version
 - b. -> Version control

The C_Lingo back end code is written for a Linux environment and offers no innate support for Windows or Mac environments. This is because of the `<arpa/inet.h>` header which the server uses to create its sockets, and the `<pthread.h>` header, which is used for multithreading purposes.

The front-end code is written to be platform independent; it was tested using the Google Chrome web browser: Version 84.0.4147.135 (Official Build) (64-bit). The back-end code was written and tested on a Debian based x86_64 GNU/Linux sub-system, which was a part of a x86_64 Windows 10 environment. There have been reports of issues when visiting the website with the Internet Explorer browser. It does not appear to work on mobile devices.

Back-end

The C_Lingo back-end code can be divided into two parts: The lingo_words command line application code, and the lingo server code. The lingo server code is dependant on the lingo_words application for its supply of words while the lingo_words application can be used as a stand-alone application. There is some debate on the internet whether Javascript is to be counted as back-end or front-end. In this case, it will be counted as part of the front-end.

Lingo_words

Lingo_words is a Command Line Interface (CLI) application written in C. It is used for gathering words with five, six, and seven letters, and saves them in their respective .txt files.

Build instructions

1. Clone the C_Lingo repository in a directory of your choice
`git clone https://github.com/RoxanneMango/C_Lingo`
2. Change directory to the lingoWords directory
`Cd lingoWords`
3. Invoke the Makefile
`make`
4. The makefile will compile the code and start the application for you.

The makefile will produce a binary called 'main', which can be evoked from the command line. When calling lingo_words, you need to give it the path of the file you want it to search through.

File IO

There are three globally #defined variables in the `lingo_words.h` header file: `FIVE_LETTERS`, `SIX_LETTERS` and `SEVEN_LETTERS`. These variables hold the paths of the three files to which the lingo_words application writes its words to; one for five lettered words, one for six lettered words, and one for words that are seven letters long. By default, these paths are:

```
#define FIVE_LETTERS "five_letter_words.txt"

#define SIX_LETTERS "six_letter_words.txt"

#define SEVEN_LETTERS "seven_letter_words.txt"
```

Before words are written to a file, the file is opened in write mode to clear its contents, before continuing in append mode!

Lingo server

The lingo server code comprises two mayor components: The server code and the lingo game code; both are written in C.

Build instructions

1. Clone the C_Lingo repository in a directory of your choice
`git clone https://github.com/RoxanneMango/C_Lingo`
2. Evoke the Makefile
`make`
3. The makefile will compile the code and start the application for you.

Important variables

There are a couple of globally #defined variables that are of special importance.

server.h holds the variables that determine the server's IP address, port number, and the maximum size of the buffers used for sending and receiving data to and from the client. The MAXLINE variable must be at least as big as the number of characters of the largest file you plan to send. If it is smaller, the file will not be sent in its entirety.

```
#define SERVER_IP "192.168.1.9"

#define SERVER_PORT 44900

#define MAXLINE 10000
```

lingo.h houses the variables used for the lingo game: The number of seconds a player has to guess word, the amount of seconds between a loss and the deallocation of the resources of the server, and the three paths used when opening files for retrieving words for the

player to guess. Trying to open files with incorrect paths will - in some cases - result in segmentation faults.

```
#define GUESS_TIME 15

#define NAME_TIME 60

#define FIVE_LETTER_WORD_FILE "lingoWords/five_letter_words.txt"

#define SIX_LETTER_WORD_FILE "lingoWords/six_letter_words.txt"

#define SEVEN_LETTER_WORD_FILE "lingoWords/seven_letter_words.txt"
```

`high_scores.h` contains the variable for the path of the high score file, to which the high scores are written and read from:

```
#define HIGH_SCORE "high_scores.txt"
```

It is not advised to change the values of any other #defined variables besides those listed above.

Initializing the socket and buffers

The `<arpa/inet.h>` library is used for handling network connectivity. The main function in `main.c` calls the `create_socket()` function defined in `server.c`, which starts by creating a socket. The socket is bound using the ip address and port defined in `server.h` and will be set to listen for any incoming connections.

Three buffers are created: *buff*, *recvline* and *message*. *recvline* is used in a read-only fashion for all incoming data, while *buff* is used for all outgoing data. *message* is used as a workbench: It is passed to other functions that put data in it. Eventually *message* is appended to *buff* as the body of the reply message to a client, while *buff* itself starts with the message header:

```
"HTTP/1.0 200 OK\r\n\r\n"
```

This header is the bare minimum needed for sending a valid TCP response, hence why it is hardcoded the way it is in *buff*.

All string buffers are of the `(char *)` or `(char[])` datatype because they are used for sending characters. All, except the *recvline* buffer because this buffer is passed to the `<arpa/inet.h>` `read()` function, which expects an `(uint8_t *)` - unsigned integer of 8 bits long; the unsigned counterpart of the `(char *)`. This buffer is cast to a `(char *)` when being passed to other functions.

All buffers are zeroed out after each use. This is to prevent undefined behaviour from left over data, and segmentation faults from stack smashing.

Tour through the Lingo struct

The Lingo struct in `lingo.h` is an absolute unit. I will go over each variable; why it is needed and what it is used for.

```
char * word;  
char * hints;  
char ** guesses;
```

word contains the word that the player needs to guess. After the player guesses incorrectly, the word is appended to *guesses*. Each correctly guessed letter is added to *hints*. All three of these arrays are allocated using the `calloc()` function for null initialization, after which they are populated with more meaningful values. *guesses* contains all the guesses the player made. *hints* contains all the letters the player guessed correctly.

```
int index;  
int hintSize;  
int wordSize;  
int numberOfGuesses;  
int guessesRemaining;
```

All these variables are used in conjunction with the above three char arrays. *index* is used to determine the player's position within the 2-dimensional Lingo game grid *guesses*. *hintSize* is used in combination with *hints* to display the correctly guessed letters up to the point the player has guessed within the lingo game grid. *wordSize* is the length of the word that the player needs to guess. *numberOfGuesses* is basically always initialized to 5, and it will always be 5, but to avoid magic numbers it was kept as a variable, even though it could be hardcoded. *guessesRemaining* is the amount of tries the player has left until they lose; it is lowered by one every time a player makes a guess.

```
int score;  
char * name;
```

These two variables are used to keep track of the score of a player. *score* holds the score which is calculated as $(10 * \text{lingo->guessesRemaining})$ and is added to itself every time a player guesses a word correctly.

```
time_t startTime;  
time_t endTime;  
int guessTime;  
int lostCountDown;
```

To keep track of the game time, there are the *startTime* and the *endTime* variables. If *startTime* is larger than *endTime*, it means the player took too long to make a guess, and they will lose the game. The *guessTime* is the number of seconds a player has left to guess. At the start of the game, and after each guess, *startTime* is set to the current time and *endTime* is set to *startTime* + *guessTime*. The *lostCountDown* variable is the number of seconds a player has left to enter their name for the high score list, and it is lowered by 1 every second.

```
pthread_t thread;
```

```
bool mutex_free;
```

The lingo game runs separately from the server code; it runs on its own thread, called *thread*. This was necessary to keep track of the game time without a steady supply of incoming data from a client: The server code only completes a loop if there is incoming data, otherwise it will stall to keep listening for any incoming connections. While the server code stalls, the lingo code keeps doing its checks and will update its flags and other variables accordingly. A mutex was introduced to block the access of lingo data when *thread* was performing operations with it: *mutex_free*.

```
bool guessed;
```

```
bool isRunning;
```

```
bool isLost;
```

```
bool isWon;
```

```
bool isWonAck;
```

```
bool killSignal;
```

These are all the flags the game uses to prevent you from getting automatic win/lose conditions, segfaults and infinite wait conditions due to the asynchronous nature of multithreading. *guessed* is initially initialized to false and is set to true through a server call from the client when a guess is made. The *lingo_game()* function running on *thread* will set it back to false after it has updated the necessary values.

When a player guesses a word correctly, the *isWon* flag is set to true. To allow the front-end enough time to register the win and retrieve the necessary data, the *isWonAck* flag was created. It is initially set to false, and is set to true via a server call, after which *isWon* and *isWonAck* are both set to false again by *lingo_game()*.

When a player loses the game, the *isLost* flag is set to true. It works the same way as the *isWon* flag, although, due to the longer grace period granted by *lostCountDown* and the more permanent nature of losing as opposed to winning – where you would continue playing – there was no need for a ‘*isLoseAck*’ flag. The *isLost* flag eventually triggers the *killSignal* flag when *lostCountDown* gets down to zero, or if a server call is made to set *killSignal* to true. When *killSignal* is set to true, the deallocating process

will begin, after which *isRunning* – which is always initialized to true on startup – is set to false, and the game halts.

Server calls

The server will respond to only two types of packages: GETs and POSTs. While it does recognize the other variants as well – PUT, PATCH, DELETE – it does not do anything with them, because there are no server calls associated with those types. It is all very hard coded, and while it may be bothersome to add another code entry for each new GET request for a new web page, it does make it impossible to ask for things you are not supposed to get, because the server will ignore it completely.

GET

The server can GET the content of the following files, as part of `page_requests.c`:

```
/ (also gives index.html)
/index.html
/manual.html
/lingo_setup.html [DEPRECATED]
/lingo.html
/about.html
/style.css [NO_FORMATTING]
/lingo.js
```

The server will respond to the following GET calls, as part of `lingo.c`:

```
/lingo_board
```

Returns a string of size $((wordSize * 2) * numberOfGuesses)$ where each letter in *guesses* is paired with a flag signalling whether that letter is incorrect (!), correct but wrongly placed (?), or correct (=). Each individual word in the *guesses* matrix is placed in between braces (). Any not-yet initialized letters are set to underscores (!_). Example:

```
Word = hello;
```

```
Guess = alloh;
```

```
Response = (!a?l=1?o!h);
```

`/lingo_is_running`

Returns a string: "1" if *isRunning* is true and *killSignal* is false, otherwise "0".

`/lingo_is_lost`

Returns a string: "1" if *isLost* is true, otherwise "0".

`/lingo_score`

Returns *score* in string format.

`/lingo_is_won`

Returns a string: "1" if *isWon* is true, otherwise "0".

`/lingo_guesses_remaining`

Returns *guessesRemaining* in string format. However, if *isRunning* is false, it will return "0" regardless what value *guessesRemaining* has.

`/lingo_time_remaining`

Returns $(endTime - time(0))$ in string format. However, if *isRunning* is false, it will return "0" regardless what outcome $(endTime - time(0))$ has. $Time(0)$ is the current time.

`/lingo_size`

Returns *wordSize* in string format, unless *isRunning* is false, at which it will return "0" regardless what value *wordSize* has.

POST

A struct called *Param* was created – with an instance called *param* – for POST packages. It has a key and a value variable. When the server gets a POST package, the incoming data is sanitized and *param* is populated. Then it is forwarded to the *lingo_input()* function in *lingo.c*, where the {key:value} pairs are evaluated for the following combinations:

- `lingo_game`
 - `start`
 - call the *start_lingo()* function if *isRunning* is false, to start a (new) lingo game.
 - `quit`
 - set *killSignal* to true if *isRunning* is true and *killSignal* is false, to start the deallocation process of the lingo game.

- `lingo_add_name`
 - `{value}`
 - If `isRunning` is true, and `killSignal` is false, and `isLost` is true, and `{value}` is not empty, set `name` to `{value}`.
- `lingo_is_won_ack`
 - `ack`
 - if `isRunning` is true, and `isWon` is true, set `isWonAck` to true.
- `lingo_size`
 - `five`
 - if `isRunning` is true, set `wordSize` to 5.
 - `six`
 - else if `isRunning` is true, set `wordSize` to 6.
 - `seven`
 - else if `isRunning` is true, set `wordSize` to 7.
- `lingo_guess`
 - `{value}`
 - Set `guesses[index]` to `{value}` if `isRunning` is true, and both `killSignal` and `isLost` are false, and `{value}` passes the `isValid()` check from `param.c`.
 - Set the letter of `hints` to the letter of `word`, where the letter of `{value}` corresponds to the letter of `word`, if it passes the same flag checks as mentioned above.
 - Set `guessed` to true if it passes the same flag checks as mentioned above.

Error handling

Error handling is done through a patchwork of checks and evaluations. If an error is severe enough it is passed to the `err_n_die()` function in the `error_handling.c` file. This will print the error code alongside any custom message added when making the function call. `err_n_die()` will cause the program to exit with a return value of 1. Severe errors are those that hinder the integrity of the program in terms of memory and performance, such that it can no longer progress. `err_n_die()` is used mostly during network initialization.

There is a known risk that a segmentation fault will occur when trying to read from a file that does not exist. This will be patched with the appropriate checks in a future update.

Front-end

The C_Lingo front-end was made to resemble a combination between an arcade game and a terminal. Black background, big buttons, faded colours. All the pages have an unnamed `<div>` that serves as a container. In that `<div>` there is the `<h1>` title, a “word_container” class `<div>` and a “menu” class `<div>`. The reason all the CSS is constantly repeated at the top of every html page, is because I couldn’t get the server to transmit the CSS-HTML pairs in order, so the HTML file would arrive before the CSS file, and it would not have any effect whatsoever.

Javascript

`lingo.js` is by far the biggest file out of all the files that need to be send over a socket. This is largely due to the many synchronization checks and measures that had to be taken due to multithreading and asynchronous function calling, and the amount of formatting that had to be done to make it look sort of decent.

Global variables

The global variables listed in `lingo.js` mirror those found in the Lingo struct in `lingo.h`, to prevent unnecessary data transmission, and are initialized as followed:

```
let i = 0;
let tries = 0;
let interval = 0;
let isRunning = true;
let isLost = 0;
let isWon = false;
let score = 0;
let isReady = true;
```

Functions

There are a total of 20 functions in `lingo.js`:

`lingo_init()`

Populates the global variables with actual values retrieved from the server.

`startLingo()`

Set the `lingo->isRunning` flag to true.

`stopLingo()`

Set the `lingo->killSignal` flag to true.

`setLingoSize(data)`

Set `lingo->wordSize` to {data}.

`lingo_isWonAck()`

Set `lingo->isWonAck` to true.

`lingo_addName(data)`

Set `lingo->name` to {data}.

`post(data)`

Make a POST request to /lingo with payload {data}.

`get_lingoSize()`

Set the `maxLength` of input to `lingo->wordSize`.

`get_lingoScore()`

Set global variable `score` to `lingo->score`.

`get_lingoIsLost()`

Set global variable `isLost` to `lingo->isLost`.

`get_lingoBoard()`

Fetch the `lingo->guesses` and build the lingo board.

`get_lingoTime()`

Set global variable `i` to $(\text{lingo->endTime} - \text{time}(\theta))$, where $\text{time}(\theta)$ is the current server time.

`get_lingo_high_scores()`

Fetch the list of highscores and display them.

`get_lingoIsRunning()`

Set global variable `isRunning` to `lingo->isRunning`

`get_lingoGuessesRemaining()`

Set global variable `tries` to `lingo->guessesRemaining`

`get_lingoIsWon()`

Set global variable *isWon* to *Lingo->isWon*.

`get_lingoStats()`

Call `get_lingoTime()` if *i* ≤ 0 to check whether the player has run out of time. If so, retrieve the *Lingo->isLost* status. If it is false, display the player's current score, their number of remaining ties, and their remaining guess time. If it is true, but the game is still running, check whether their score is 0. If it is 0, display the high score, otherwise give them the chance to enter their own name for the high score. If the game is no longer running, display that the game has stopped running.

`submitHighScore(event)`

Submit input taken from input field if it is longer than 0 characters.

`submitForm(event)`

Submit a word guess if it is the same length as the *maxLength* of the input field.

`Timer()`

This is an interval function. The interval time is set to 1000 milliseconds. Check whether *isWon* is true, or *isReady* is false, or *isLost* is true. If *isWon* is true, set *isReady* to false, and call `lingo_isWonAck()`. If *isReady* is false, call `get_lingoIsWon`, otherwise call `get_lingoStats()`. If *isReady* and *isWon* is false, call `lingo_init()`. If *isLost* is true, clear the interval.

Sources

This video playlist helped a lot when trying to figure out socket programming:

https://www.youtube.com/playlist?list=PL9IEJIKnBJjH_zM5LnovnoaKlXML5qh17

I did not really know how Lingo worked. These sources helped me better understand it:

https://www.kijk.nl/programmas/lingo/bwoSKfcmjbG/seizoen/122816040319/afleveringen/video/empty_episode-lingo-s2-e84-2020-06-16/zrtSydwcSxQ

[https://en.wikipedia.org/wiki/Lingo_\(Dutch_game_show\)](https://en.wikipedia.org/wiki/Lingo_(Dutch_game_show))

This website was helpful when writing the `lingo_restart()` function in `lingo.c`:

<https://en.cppreference.com/w/c/memory/realloc>

The website `tutorialspoint.com` has a lot of useful short function explanations and examples. This particular example helped me with sanitizing my inputs:

https://www.tutorialspoint.com/c_standard_library/c_function_tolower.htm

This link reminded me of the existence of the `atoi()` function, to convert strings to integers in C:

https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rtrf/itoi.htm

I got my `word_list.txt` contents from this website:

<https://www.ef.com/ca/english-resources/english-vocabulary/top-1000-words/>