School of Computer Science, McGill University

COMP 206: Introduction to Software Systems
Assignment 4: files and pointers

Instructors: Jacob Errington and Joseph Vybihal

Winter 2024

Before you start, please read the following instructions:

- **Individual assignment**
  This is an individual assignment. Collaboration with your peers is permitted (and encouraged!) provided that no code is shared; discuss the ideas.
- **Use Discord and OH for questions**
  If you have questions, check if they were answered in the `#clarifications` or `#q-and-a` channels on Discord. If there is no answer, post your question on the forum. Do not post your code. If your question cannot be answered without sharing significant amounts of code, please use the TA/Instructors office hours. **Do not email the TAs and Instructors with assignment questions.** TAs should only be contacted by emails if you have questions about a grade you received.
- **Late submission policy**
  Late penalty is -10% per day. Maximum of 2 late days are allowed.
- **Must be completed on mimi server**
  You must use `mimi.cs.mcgill.ca` to create the solution to this assignment. An important objective of the course is to make students practice working completely on a remote system. You cannot use your Mac/Windows/Linux terminal/command-line prompt directly without accessing the mimi server. You can access `mimi.cs.mcgill.ca` from your personal computer using **ssh** or **putty** as seen in class and in Lab A. **If we find evidence that you have been doing parts of your assignment work elsewhere than the mimi server, you might lose all of the assignment points.** All of your solutions should be composed of commands that are executable in `mimi.cs.mcgill.ca`.
- **Must be completed with vim**
  A goal of the class is to get you to be comfortable in a command-line text editor. This wouldn't happen just from knowing the vim commands from slides: You become comfortable by using it a lot. Assignments are a good opportunity for that.
- **You must use commands from class**
  In this assignment, anything in the C standard library is allowed. However, the emphasis in the assessment is on clarity and readability of the code. Do not do anything unnecessarily generalized, abstract, or complex, when simple, straightforward code suffices.
- **Your code must run**
  Your code must run as-is and nearly instantly. **TAs WILL NOT modify your code in any way to make it work.**
- **Please read through the entire assignment before you start working on it**
  You can lose up to 3 points for not following the instructions in addition to the points lost per questions.

1 A CSV-backed database written in C

In this assignment, your task will be to implement a command-line program called `igdb` in C for managing a persistent database stored as a CSV file. This is a simplified example of what C is actually used for in practice: writing fast, low-level data management software. The popular, high-performance, open-source relational database management system PostgreSQL is written in C, for instance.[1]

Learning goals assessed:

1. Write robust system interactions in C.
2. Create custom data structures and algorithms in C.
3. Organize code in C for readability.

1.1 What's in the database?

The database in this assignment is for tracking Instagram accounts. One record in the database stores the following information.

- Handle, e.g. `@spottedmcgill`
- Follower count, e.g. `14900`
- Comment, e.g. `a bit cringe tbh`
- Date last modified, e.g. `1710521259`

This record would be represented in CSV form as

```
@spottedmcgill,14900,a bit cringe tbh,1710521259
```

To accommodate the CSV file format, the characters `'\0'` (null) `'\n'` (line feed) and `','` (comma) are forbidden from appearing in the comment field.

1.2 Representing dates and times

You might have noticed that the "date last modified" above doesn't look like a date at all...

Representing dates and times robustly can be a challenge in software engineering. A very common and straightforward solution is to represent an absolute moment in time in so-called UNIX Epoch format. This is the number of seconds elapsed since midnight on 1 January 1970. These are also called (UNIX) timestamps. This format is independent of timezones, which makes it great for storage in a database. User-facing applications convert these timestamps into human-readable "local time" strings, accounting for timezomes, when displaying the time to a user.

Your database will store UNIX timestamps, and your application must present these times in human-readable strings in the local timezone. The C standard library contains functions for working with UNIX timestamps and dates and times in general.

- You will need to `#include <time.h>`
- See `man 2 time` (get current timestamp)
- See `man 3 localtime` (convert timestamp into time structure in current timezone)
- See `man 3 strftime` (format time structure into string)

1.3 Representing database records in C

You must define a `struct Record` to represent a single line (record) from the CSV file (database). It must have one member for each of the fields of the record described above.

- The "handle" and "comment" fields must be character arrays (not pointers) with sizes 32 and 64 respectively.
- The "follower count" and "date last modified" fields must be of type `long unsigned int`

[1]`https://github.com/postgres/postgres` – to learn more about SQL database systems, take COMP 421.

1.4 Representing the database itself in C

You must define a `struct Database` to represent a whole database. This will be an implementation of dynamic arrays, i.e. arrays that can "grow" during runtime, similar to the `ArrayList` class in Java or Python's built-in `list`.

A dynamic array has three attributes: a pointer to an underlying (fixed-size) array, an integer called the capacity, and an integer called the size. The capacity is the length of the underlying array, and the size is the count of elements actually stored inside that array.

When we want to add an item to the end of a dynamic array, we have to check if there's space left in the underlying array by comparing the size and the capacity. If the size is less than the capacity, then we can treat the size as the index at which to write the new item, then increment the size. Else, when the size has reached the capacity, a new underlying array with double the capacity is allocated, the elements from the old array are copied to the new array, and the old array's memory is freed, before proceeding as before.

You must implement the following functions for handling the dynamic array.

```c
typedef struct Database { /* fill this in */ } Database;

Database db_create();
// ^ The database must have initial size 0 and capacity 4.

void db_append(Database * db, Record const * item);
// ^ Copies the record pointed to by item to the end of the database.
// This is where you need to implement the resizing logic described above.

Record * db_index(Database * db, int index);
// ^ Returns a pointer to the item in the database at the given index.
// You need to check the bounds of the index.

Record * db_lookup(Database * db, char const * handle);
// ^ Returns a pointer to the first item in the database whose handle
// field equals the given value.

void db_free(Database * db);
// ^ Releases the memory held by the underlying array.
// After calling this, the database can no longer be used.
```

To implement these functions, you will need to use some of the standard library functions for memory management: check out `malloc`, `calloc`, `realloc`, and `free`.

As usual, you can check the documentation for these with, e.g. `man 3 malloc`.

1.5 Reading and writing CSV files

When the database application starts, it will load the file `database.csv` into memory, populating a `Database` object. When the application runs, the user will be able to command it to write the database. You will implement the following functions that accomplish reading and writing the whole database.

```c
void db_load_csv(Database * db, char const * path);
// ^ Appends the records read from the file at `path`
// into the already initialized database `db`.
// (The given database does not have to be empty.)

void db_write_csv(Database * db, char const * path);
// ^ Overwrites the file located at `path` with the
// contents of the database, represented in CSV format.
```

Of course, reading the database can be decomposed into two subproblems: parsing a single line of the CSV file into a `Record` structure versus looping over all the lines in the file to populate the whole database. Your code must reflect this separation of concerns by implementing and using the following function.

```
Record parse_record(char const * line);
// ^ Parses a single line of CSV data into one Record
```

You must use the following standard library functions to implement the functions in this section.

- `fopen` to open files for reading or writing. You must call `fclose` on these when you finish reading or writing.
- `getline` reads a line from a stream. You must be careful to properly free any buffers that this function might allocate for you.
- `fprintf` writes formatted data to a given stream.
- `strtok` is used to break up a string according to a delimiter, such as `','`.

As always, check the documentation in the man pages, e.g. `man 3 getline`.

1.6 User interactions with the database

The user manipulates the data in the database via interactive prompt. The below example session illustrates how this should work. Each line beginning with `>` is a prompt, and my input is given after it.

```
jerrin@teach-node-04:~$ ./igdb
Loaded 5 records.
> list
HANDLE           | FOLLOWERS | LAST MODIFIED    | COMMENT
-----------------|-----------|------------------|--------------------------------
@spottedmcgill   | 14900     | 2024-03-15 17:03 | a bit cringe tbh
@foo             | 420       | 2024-03-13 13:37 | foo is foo
@bar             | 8008135   | 2024-03-14 20:03 | go to bar
@foobar          | 8008555   | 2024-03-15 17:01 | foo + bar = foobar
@quux            | 1234567   | 2024-03-16 16:06 | how to pronounce quux?
> add @zzzz 98765
Comment> sleeping in is great zzzz
> list
HANDLE           | FOLLOWERS | LAST MODIFIED    | COMMENT
-----------------|-----------|------------------|--------------------------------
@spottedmcgill   | 14900     | 2024-03-15 17:03 | a bit cringe tbh
@foo             | 420       | 2024-03-13 13:37 | foo is foo
@bar             | 8008135   | 2024-03-14 20:03 | go to bar
@foobar          | 8008555   | 2024-03-15 17:01 | foo + bar = foobar
@quux            | 1234567   | 2024-03-15 17:03 | how to pronounce quux?
@zzzz            | 98765     | 2024-03-15 17:08 | sleeping in is great zzzz
> update @bar 0
Comment> !!!account deleted!!!
> add
Error: usage: add HANDLE FOLLOWERS
> add @looooooooooooooooooooooooooong.name 12345
Error: handle too long.
> add @short.name 12345
Comment> ummm, this is a comment
Error: comment cannot contain commas.
> add @short.name lmao
Error: follower count must be an integer
> list
HANDLE           | FOLLOWERS | LAST MODIFIED    | COMMENT
-----------------|-----------|------------------|--------------------------------
@spottedmcgill   | 14900     | 2024-03-15 17:03 | a bit cringe tbh
@foo             | 420       | 2024-03-13 13:37 | foo is foo
@bar             | 0         | 2024-03-15 17:09 | !!!account deleted!!!
@foobar          | 8008555   | 2024-03-15 17:01 | foo + bar = foobar
```

```
@quux           | 1234567   | 2024-03-15 17:03 | how to pronounce quux?
@zzzz           | 98765     | 2024-03-15 17:08 | sleeping in is great zzzz
> update @wow 12345
Error: no entry with handle @wow
> add @spottedmcgill 12345
Error: handle @spottedmcgill already exists.
> exit
Error: you did not save your changes. Use `exit fr` to force exiting anyway.
> save
Wrote 6 records.
> exit
jerrin@teach-node-04:~$
```

Here is a detailed description of the commands that your interactive mode must support.

- `list` – print out the whole database formatted as a table. For the widths of the columns, you can choose a fixed width, but make sure that the output always looks like a decent table. For instance, if you choose a fixed width of 20 characters for the handle, but the handle is 25 characters long, you should truncate the remaining extra characters from the handle to make it fit in the column. The same applies to the comment. This can be accomplished using the "width" and "precision" modifiers for the `%s` directive of `printf`.
- `add HANDLE FOLLOWERS` – adds a new entry to the end of the database. That handle must not already exist in the database. A follow-up prompt is generated to ask the user for the value of the comment field.
- `update HANDLE FOLLOWERS` – updates an existing entry for the given handle. An entry for that handle must be present. A follow-up prompt is generated to ask the user for the value of the comment field.
- `save` – writes the database out to the file `database.csv`
- `exit` – quits the program, but warns the user about unsaved changes if any.

As in the previous assignment, your implementation must be robust in the face of user input. Therefore, for things such as integers written by the user in prompts, you will need to use `strtol` as in the previous assignment to ensure that valid input is given.

However, you may assume that the CSV file is well-formed, so you do not need to validate, for instance, that the value stored in the follower count column contains only digits.

2 What to hand in

This assignment comes with starter code.

You should modify the files `database.c`, `database.h`, and `igdb.c` from the starter code, providing the implementations for all the functions described above.

The starter code contains a Makefile, which you can run via the command `make`. This will compile each C file separately and then link them together into an executable `igdb`.

On MyCourses, you must submit a zip file containing `database.c`, `database.h`, `igdb.c`, and the Makefile.

3 Evaluation rubric

The notation "LG" followed by a number refers to an assessed learning goal from the previous section. Each learning goal is associated in this table with the weight it holds in the overall assessment of your learning in this assignment.

If your code does not compile or run at all, it will be given a grade of zero!

The categories A, B, C in the below table do not necessarily match the standard interpretations of those letters in McGill's grading scheme. Instead, they should be understood as "exceeds expectations," (A) "meets expectations," (B) "does not meet expectations," (C) and "not assessable" (N).

	LG1 (50%)	LG2 (30%)	LG3 (20%)
A	All user input is validated correctly and validation logic is separate from business logic. All allocated memory blocks and file handles are freed at appropriate times, and explicitly before program termination. Care is taken to avoid buffer overruns and the program does not cause undefined behaviour, nor does it enter an invalid state. Program output is well-formatted and readable under all typically expected circumstances. Program output reflects the database contents.[2]	Loop conditions clearly reflect the structure of the data. Array offset calculations for indexing are straightforward. Algorithms handle all edge cases properly. The use of pointers is very clear, making appropriate use of dereference syntax.	Code structure follows the outline of the assignment. Additional functions are added if appropriate to further isolate behaviours. Code is well signposted with comments that explain the why, not the how. Code redundancy is minimal. Nesting of control flow structures is generally limited to two or three deep. Code is well-formatted with proper indentation and judicious use of blank lines to improve readability.
B	Most user input is validated. Buffer overruns are possible but unlikely. Input validation logic and business logic are largely separate, but sometimes intermixed. The program does not cause undefined behaviour. The program could enter an invalid state under particularly exceptional circumstances. Program output is generally well-formatted and readable under most circumstances. Program output reflects the database contents.	Most loops are expressed with clear conditions. Control flow is easy to follow and not deeply nested. Some functions do too much. Array offset calculations are at times hard to follow. Algorithms leave some edge cases unhandled. The use of pointers is clear.	Code comments explain the how, but not the why. Code is decomposed into functions, but their purpose is sometimes unclear or ambiguous. Code is redundant, but not excessively. Control flow nesting is deep.
C	User input is generally not validated. Error messages do not necessarily reflect the problem. Input validation logic is haphazard and generally intermixed with business logic. The handling of buffers is improper and overruns are easy to arrange by the user. Simple user errors can cause the program to enter an invalid state. Program output is not well formatted or does not reflect the database contents.	Loop conditions and control flow are confusing or unclear. Pointer dereferences and address-taking are complicated. Array offset calculations are overall confusing. Algorithms do not handle edge cases.	Code comments add no value or are largely missing. The use of functions is limited, or the present functions have unclear purposes. Code is very redundant, with significant pieces appearing copy-pasted.

[2]The program would be in an invalid state if, for instance, after the user writes a bogus command in the interactive mode, the command parser is "off" and always fails even on subsequent valid commands.

N	If the program runs, failure scenarios are not checked. Error messages are not generated. The program generally assumes correctness of inputs. The program is easily made to crash or overrun buffers by the user, even on valid inputs. Program output is largely incorrect or exceedingly difficult to read.	The use of pointers, loops, and arrays is generally incorrect. Loop conditions are outright wrong. Indexing logic is convoluted. Algorithms do not handle edge cases and produce incorrect results on typical inputs, too.	The code lacks comments. It is not decomposed into functions. Code is very redundant and highly nested.