COMP2401 A/B/C

Assignment #2: Structured Device Information

Structs and Strings

Goals

In this assignment you will be working with structures, unions, pointers to structures, and string manipulation functions. You have been provided with a header file containing the structure and union definitions and placeholder (stub) functions to implement your code in. You may find quickly that the code can be cleaned up by planning out some helper functions to keep each function simple, single-purpose, and easy to read.

Restrictions

Make sure to review the course outline’s collaboration and AI rules for information about acceptable collaboration and appropriate use of AI before proceeding.

Note: You must be prepared to explain your design decisions and each line of code regardless of your collaboration or use of external resources.

Do not modify the names of functions, definitions, or anything else that breaks the signature / contract of the provided code. You **are** permitted to create helper functions and additional helper definitions. These functions must work as specified for the autograder to correctly compile.

Learning Outcomes

- **Apply** correct string library functions in a safe manner to manipulate strings in C.
- **Implement** a C program that uses structures as an aggregate data type, including nested structures.
- **Implement** a C program that uses pointers to structures, and the corresponding arrow operator.
- **Implement** a C program that uses a union to store different types of data in the same structure field.
- **Implement** a program that uses pointers to avoid creating unnecessary copies of data.

Preparation

Where to find more help for the content of this assignment:

- **Chapter 2.3-2.6:** Compound data types, arrays, strings, structures, unions.
- Appendix A for information about structs, Appendix B for information about strings, and Appendix C for information about `fgets()`

Assignment Context (Optional Read)

This is great! You’re really starting to help out a lot around here. We know we don’t usually run into any real ghosts, but it’s really fun going out and exploring with a group of friends. We’re actually able to afford picking up some new equipment, so we were thinking you might be able to upgrade our data collection software to support all kinds of new types of data!

An old volunteer started writing out the data structures to interface with our devices, but didn’t get around to coding the rest. We’ll be tracking readings across three different kinds of devices and would love a simple menu system to add readings and print them out.

You’ve got this! Thank you for all the help, the club really appreciates it all of your hard work.

1. Overview

You are given a complete `defs.h` and a starter `main.c` with `print_menu(...);` you'll implement the functions declared in `defs.h` and the menu handlers in `main.c`. You have also been provided with `loader.o` which contains helpful functions for loading test data and testing the sorting order.

- **Definitions**:
  - `C_ERR_*`: Error codes to return from functions, `C_ERR_OK` means no error occurred,
  - `TYPE_*`: Device types, used in the `Reading` structure,
  - `MAX_ARR`, `MAX_STR`: Maximum sizes for arrays and strings,
- **Data Model**: All of the `LogEntry` and `Room` structures should only be created once and are stored in the collections that are declared in `main.c`.
  - `ReadingValue`: Represents a union of possible device reading values,
  - `Reading`: Represents a single reading from a device, including its type and value,
  - `LogEntry`: Represents a single reading taken in a specific room at a specific time,
  - `EntryCollection`: Represents a collection of all log entries,
    - This collection is kept in sorted order: first by room name alphabetically in ascending order (i.e., A comes before B), then by device type in ascending order (i.e., 1 < 2), then by timestamp in ascending order (i.e., earlier timestamps come before later timestamps).
  - `Room`: Represents a room where readings are taken points to all entries logged in that room,
  - `RoomCollection`: Represents a collection of all rooms, which is not required to be sorted.
- **Functions to Implement**: It is highly recommended that you consider the work that each function will need to do and break some of these down into smaller helper functions to keep each function simple and easy to read. Overly complex functions that would be best broken down into smaller pieces may lose marks for design during TA review.
  - **manager.c:** `rooms_add()`, `rooms_find()`, `room_print()`, `entries_create()`, `entry_print()`, `entry_cmp()`,
  - **main.c:** `main()`, and likely several helper functions for handling the menu options.
  - **loader.o:** No implementations, but provides some test data and test functions to help verify correctness.

1.1. Control Flow

Watch the Sample I/O video on Brightspace for more information.

1. The user is presented with a menu of options in a loop until exiting,
2. The user can load sample data,
3. The user can add new rooms or new entries,
4. The user can print out the entries as a list of entries or grouped by room,
   a. Entries print in nice columns with all necessary information
5. (OPTIONAL) The user can run the loader tester to verify that sorting is correct,

2. Reading and Running the Code

- **Extract:** review the starter files.
- **Files to submit:** A single `.tar` file containing: your completed `main.c`, `manager.c`, the provided `defs.h` with any necessary modifications (if any), `README.md`, and the provided `loader.o`.
- **Build:** `gcc -Wall main.c manager.c loader.o -o a2`
  - You may optionally include a `Makefile` if you know how to work with these, but it is not required and must be original work for this class.

3. Instructions

⚠️ **Warning**

- Do not modify the names or signatures of any functions
- All functions should return `C_ERR_OK` or the appropriate error codes or error state / value as specified in the provided documentation
- You may create helper functions as needed within the provided functions, but they must be in the files which require them (i.e., no new `.c` files, anything needed in `manager.c` should be in `manager.c`)

3.1. Rooms

Feature 1 Description: I want to add rooms by name with no duplicates.

Feature 1 Requirements: Implement `rooms_add(RoomCollection*, const char*)` and `rooms_find(RoomCollection*, const char*)`.

- `rooms_add` appends a room if it does not already exist; returns `C_ERR_DUPLICATE` if it does, `C_ERR_FULL_ARRAY` if at capacity, `C_ERR_OK` otherwise.
- `rooms_find` returns a pointer to the existing room or `NULL`.
- The user can add rooms with spaces in their name via the menu (update `main.c`)

3.2. Logs & Sorted Insertion

Feature 2 Description: I want adding a log to automatically keep collections in order.

Feature 2 Requirements:

- Implement `entry_cmp(const LogEntry*, const LogEntry*)` with the rule **Room ascending, Type ascending, Timestamp ascending**; return `<0` if `a` is before `b`, `0` if equal, otherwise `>0`.
- Implement `entries_create(EntryCollection*, Room*, int type, ReadingValue value, int timestamp)`:
  1. Validate `type` is one of `TYPE_TEMP | TYPE_DB | TYPE_MOTION`; otherwise return `C_ERR_INVALID`.
  2. Check capacity for **both** the "global entries" (main-allocated) array and the room's pointer list; return `C_ERR_FULL_ARRAY` if full.
  3. **Insert by value** into `EntryCollection` at the correct position (shift right, using `entry_cmp`).
     a. **Important:** Inserting by value **moves** existing entries during the shift (you copy `src` into `dst`).
     b. Any room that pointed to `src` must now point to `dst`. During the shift loop, after `*dst = *src;`, retarget the **one pointer** in `dst->room->entries[]` from `src` to `dst`.
  4. **Insert a pointer in sorted order** to the newly placed entry into `room->entries[]` (use the same `entry_cmp` rule).
  5. **Requirement**: every entry appears **exactly once** in the global array and **exactly once** in its room's pointer list. The `loader_test_rooms()` function can help to verify this, along with manual review.

💡 **Idea**

You should notice that retargeting the room's pointer during shifts is required in this design because entries are stored **by value** in the global array. Consider alternative designs (e.g., arrays of **pointers** globally, or rooms storing **indices** instead of pointers) and what their trade-offs would be—this will connect directly to our discussion of dynamic memory later in the course.

3.3. Printing

Feature 3 Description: I want to be able to print my entries in a nice, clean tabular format - either as a big list of sorted entries or grouped by room.

Feature 3 Requirements: Exact printing format is not required, but should provide clean, titled columns similar to that in the sample video.

- `entry_print(...)` prints **one line** with columns: **Room**, **Timestamp**, **Type label** (`TEMP`, `DB`, `MOTION`), and **Value**:
    1. TEMP: print with two decimals and unit, e.g., `23.50°C`
    2. DB: integer with `dB`, e.g., `42 dB`
    3. MOTION: array of values `0` or `1`, e.g., `[1,0,1]`
- `room_print(...)` prints a header `Room: NAME (entries=N)` and then each entry via `entry_print(...)`. You may include a small header row for the columns.
- The user should be able to select either printing option from the menu (update `main.c`).

3.4. Loader & Order Test

Feature 4 Description: I want to load sample data and verify my ordering when I select these options in the main menu.

Feature 4 Requirements:
- **Load sample:** call `load_sample()` to pre-populate the collections in `main()` with sample data.
- **Optional:** call `loader_test_order(...)` to verify the sorting order of the entries collection.
- **Optional:** call `loader_test_rooms(...)` to verify that all rooms have correct, unique entries, in the correctly sorted order.

Note: These tests may not catch all errors in your code and you will still need to manually review the outputs for correctness.

3.5. 1% Optional Engagement Task Bonus

Info

About Engagement Bonuses: This challenge is optional, and can be submitted to the Engagement Task forum to receive marks toward your Engagement Task mark any time before the end of class. You are encouraged to either read-ahead to complete it around the same time as you complete this assignment or to revisit it once we are talking about the relevant material in class to use as practice. The exact requirements are much less strict on these bonuses and completion in the spirit of the bonus will be marked SAT.

Warning

You cannot submit this version of the code to the regular assignment submission slot because it will break our autograding tools. Please review the submission requirements below.

Soon we will be learning about dynamic memory allocation. This is a powerful tool, but it can be tricky to get right. For this bonus, you will get a little bit more practice with pointers by implementing two new pieces of functionality: deletion of entries and filtering entries based on criteria.

Challenge: Implement two new functions in `manager.c` and connect them to new menu entries in `main.c`.

1. `entries_delete(EntryCollection*, Room*, int timestamp)`: Deletes the entry in the specified room with the specified timestamp. If no such entry exists, return `C_ERR_NOT_FOUND`. Otherwise, remove the entry from both the global collection and the room's pointer list, shifting elements as necessary to maintain order. Return `C_ERR_OK` on success.
2. `entries_filter_by_type(const EntryCollection*, const int *type, const Room *room, EntryCollection* out)`: Filters the entries in the specified room by the specified types and stores them in the `out` collection.
   - If `type` is not `NULL`, then only the type specified by it should be included. If `type` is `NULL`, then all types should be included.
   - If `room` is not `NULL`, then only entries from that room should be included. If `room` is `NULL`, then entries from all rooms should be included.

This should help you get practice thinking about how pointers to the same data need to be updated when the data that they point to is removed or changed.

3.6. Finishing Up

Make sure to test your code and ensure it is all working correctly.

- Test all menu paths: duplicates, capacity limits, invalid types, printing clarity.
- No function should return `C_ERR_NOT_IMPLEMENTED` in your final submission.
- Keep prints tidy and consistent

For all assignments and projects going forward, you must include a **README** file and make sure your code is clean, consistent, and documented.

README: The README file can be called `README` or `README.txt` or `README.md`. The purpose of this file is to act as a kind of starting-point for your project. It should include the following:

- Your name and student number,
- A description of what the program is and does,
- Instructions for compiling and executing the program via command line,
- Any additional citations or sources when using valid external sources for information.

Make sure to submit your code according to the instructions in Section 4

4. Packaging and Submission

For this assignment, you will submit a single archive file, saved as a `.tar` compressed in Linux using a command similar to `tar -cvf a2.tar <file1> <file2>`, which includes **all files needed to run your code** and your README file, which have been updated to include the necessary information. It **must** have the `.tar` file extension and be a valid tar file.

5. Grading

Grading information is a general reference, and small changes might be made as unexpected cases necessitate, but will be visible on the assignment rubric provided as feedback.

The autograder **does not** use your `main.c` file, instead grading each function individually against a series of tests, and will provide some general information about the tests that fail in the feedback. If we can not compile using our own `main.c` file, you will receive a **zero** for the automated tests. Changes to described function names, data types, etc. can all lead to the autograder failing.

For TA review, the TA will compile and execute your full code (i.e., run your `main.c` file) according to the instructions in your README file and attempt a basic execution to evaluate the main control flow, error message quality, and print quality.

After receiving feedback, you will have **one week** to revise and re-submit your code to the revised submission portal on Brightspace. You will then have one week from receiving that grade to dispute your mark. Review the course outline for information on grade dispute and second chance submission policies.

5.1. (50%) Automated Grading, Per-Function Execution

No marks are awarded if the autograder cannot untar your work or the code cannot compile. Test your submission early to avoid issues at the deadline. Always download your submission and test it in an empty directory before the deadline. A full and up-to-date rubric will be provided with your feedback. Some samples of the tests that will be performed have been included in the provided object file and you are encouraged to consider writing tests like these before starting future assignments.

1. `find_room()` returns the correct pointers
2. `entry_cmp()` correctly compares entries
3. Entries are inserted in the correct order
4. Rooms are added correctly with no duplicates
5. Entries are correctly added into rooms in the correct order
6. Functions return appropriate error codes in different circumstances

5.2. Manual Review

The manual review will be based on a simple rubric which observes the following:

1. **Execution:** Main execution control flow works correctly as specified
2. **Packaging:** All required files are included and contain the correct information
3. **Design:** Code is well-structured, with consistent naming and no unnecessary duplication
4. **Approach:** Algorithms and conventions required from the submission are followed
5. **I/O:** Input and output handling is correct and user-friendly
6. Deductions may be applied for, at minimum:
    a. Using global variables (variables defined outside of a function)
    b. Using techniques that are not described in the class or course notes which bypass the assignment requirements

Appendix A: Structures and Pointers

This section serves as a small review for a few important parts of working with structures and pointers.

1. Recall that a pointer is a variable which stores a memory address,
   a. On a 64-bit machine, a pointer will be 8 bytes,
   b. The `*` operator can be read as "the value at", or "the value pointed to by", or the "dereference" operator,
   c. `int *x` states that "The value pointed to by the 8-byte address value `x` is a 4 byte integer."
   d. `x` itself will still have an address on the stack, as it is a variable.
   e. The `&` operator is the "address-of" operator. `&x` will give me the address of `x`; not the value that `x` stores,
   f. Thus `int *x = 5;` is incorrect, as `x` expected an address. `int y = 10; int z = &y;` is incorrect, because `&y` returns the address of `y` while `z` has reserved memory for an integer, not an address to an integer. `int *x = y;` is incorrect, because `y` is a four byte integer and `x` expects to **point** to an integer, thus its value is an address. `int *x = &y;` is correct, because `&` returns the address of `y`, which is an address which **points to** an integer.
2. A structure is a block of memory where the name of a field represents a kind of offset from the start of that block.
   a. The memory for a structure is allocated wherever the structure is declared, exactly what happens when we create an `int` or `float`.
   b. A `struct` passed into a function will copy all of the data of that `struct` onto the Stack Frame, exactly as if we passed an `int` or `float`… but with much more data.
3. We access fields of a structure with `.`, e.g., `student.name`, or `student.personal_info.age`, if `personal_info` itself is a `struct`.
   a. If we are working with a **pointer to a struct**, and not a `struct` itself, e.g., `struct Student *stu;`, then the data stored in `stu` is an **address**, **NOT** a structure. Thus, `stu.name` does not make sense, as an address has no field called `name`, it is an address.
   b. Thus like any other pointer, we have to dereference it to get its value. We *can* do this with `(*stu).name`, meaning, "(The value at the address pointed to by `stu`), get the data at the index `name`", but this is error prone and annoying to write, so we can and should use the equivalent operator `stu->name`, which reads in exactly the same way.
   c. **NOTE:** The `->` operator only implies that the left operand is a pointer, and does not care about the right operand. The only consideration with whether the `->` operator should be used or not is whether the left operand is a pointer and should be dereferenced before accessing fields.

Appendix B: Strings and Arrays

1. The name of an array is synonymous with the location to the first element in memory; it is like a `const*`, but does not need to reserve space to store the address. Because it reserves no space on the stack to hold the address, the address is just referred to in the instructions directly, and thus an array does not support re-assignment.
2. A regular pointer is just a variable holding an address, and can thus be re-assigned; arrays are special in this way.
3. When using string literals (strings in source code wrapped in `""`), we have two behaviours depending on the context they are used:
    a. If you are initializing an array of characters, e.g., `char my_str[32] = "Hello World"`, the bytes for each character plus the null terminator (`'\0'`) will be copied into the array starting at the address `my_str`.
    b. Anywhere else, `"Hello World"` will allocated these bytes plus the null terminator to the text / code segment, the read only memory, and will return the address to the text segment where these are stored. This is a `const char*`, because it points to a character which is constant; cannot be modified.
4. A string **must** be null terminated; that is, within the allocated space, there must be a `'\0'` character after the text. This must be accounted for in the memory allocated. For example, in `char my_str[32]`, can can only really hold 31 visible characters at most, because at the very least, `my_str[31]` must be `\0`, but the `\0` should come at the end of your text. E.g., `char my_str[32] = {'H', 'e', 'l', 'l', 'o', '\0'};`
5. You can and should `#include <string.h>` when working with strings and use the string functions to work with them. The most common string functions are:
    a. `strncpy(src, dest, n)`: Copy at most `n` characters from `dest` into `src`, stopping at the null terminator; note, this does not guarantee null termination, and it is common to include a null terminator after copying (e.g., `strncpy(a, b, MAX_STR-1); a[MAX_STR-1] = '\0';`),
    b. `strncmp(a, b, n)`: Compare at most `n` characters from `a` and `b` and return a negative value if `a<b`, positive if `a>b` or zero if `a==b`
    c. `strnlen(a, n)`: Return the number of characters in `a` until the `'\0'` character is found, or `n` if one is not found within `n` characters.

Always remember to consider where the string data is allocated when working with them. Attempting to modify a string in the text segment will usually lead to a segmentation fault error. Attempting to assign a pointer to an array will usually result in a compiler error due to an attempt to modify constant memory.

Appendix C: String Input

While we can read basic strings with `scanf("%s", str);`, this can be a big problem in several ways:

1. It does not check the size of the array, so if the user enters more characters than the array can hold, it will overflow into adjacent memory, leading to undefined behaviour,
2. It might leave additional characters in the “input buffer”, which can cause problems for future input operations.
3. It stops reading at the first whitespace, so multi-word strings are not possible,

Let’s address each of these issues one at a time.

Appendix C: Safer Input

If you choose to use `scanf()` to get a string, you can and should specify a maximum width to read. For example, if you have `char str[32];`, you can use `scanf("%31s", str);` to read at most 31 characters, leaving space for the null terminator. This will prevent buffer overflows, but does not solve the other issues.

Appendix C: Input Buffer

The input buffer is a region of memory where input data is stored before being processed by input functions like `scanf()` or `fgets()`. When you enter data, it is stored in this buffer until the program reads it. If there are leftover characters in the buffer after an input operation, they can interfere with subsequent input operations.

A common way to clear the input buffer is to read and discard characters until a newline (`'\n'`) is encountered. After using `scanf()`, you can clear the buffer like this: `while (getchar() != '\n');`. This loop will keep reading characters until it finds a newline, which effectively clears the buffer.

Appendix C: Multi-Word Strings

`scanf()` stops reading at the first whitespace, so it is not suitable for reading multi-word strings. Instead, you can use `fgets()`, which reads an entire line of input, including spaces, until a newline or the end of the buffer is reached.

`fgets()` is actually intended to be used to read strings from files, but it works just as well for standard input. The function signature is: `char *fgets(char *str, int n, FILE *stream);`, where `str` is the array to store the string, `n` is the maximum number of characters to read (including the null terminator), and `stream` is the input stream (use `stdin` for standard input).

Here is an example of using `fgets()` to read a string, with some error handling removed for clarity:

```c
char str[32];
printf("Enter a string: ");
fgets(str, sizeof(str), stdin);
// Usually, we then remove the newline character, because fgets() includes it if there's space
str[strnlen(str, sizeof(str)) - 1] = '\0'; // Replace newline with null
```

Appendix D: Code Style guide

We follow good design principles to be consistent, make it easier to read and understand our work, to make it easier to modify and update our work, and to simplify debugging.

```c
int main(void) {

}

// or

int main(void)
{

}
```

Use a consistent brace style.

```c
// Print each odd value in the array
for (i=0; i<size; i++) {
  if (arr[i] % 2 != 0) {
    printf("%d", arr[i]);
  }
}
```

Use comments to explain behaviour that isn’t obvious at-a-glance.

```c
int i; // Not good

int main(void) {
  for (i=0; i<42; i++) {
    // ...
  }
}
```

Do not use global variables (like `i`) unless instructed to.

```c
// Bad example...
for (i=0; i<MAX_ELEMENTS; i++)
  if (i % 2 == 0)
    printf("At least I indented...");
  if (i % 3 == 0) printf("A lot wrong here.");

```

Generally avoid loops and conditions without parentheses (It can be tough to debug!)

```c
/*
    Removes the id and reading for any readings strictly under `minimum` from the `ids` and `readings` arrays.

    in/out ids[]:       Array of IDs associated with EMF readings in `readings`
    in/out readings[]:  Array of EMF readings to check against the minimum
    in     size:        Number of elements in `ids` and `readings`
    Returns:
        - Updated number of elements in the arrays
*/
int sort_entries(int ids[], float readings[], int size) {
```

Include function documentation in comments which explains the purpose of the function, what the parameters do, and what it can return.

- Input Parameters: The data will be read by the function.
- Output Parameters: The data will be modified by the function.
- Input/Output Parameters: The data will be both read and modified.

Appendix E: Example README

README files are very common for code projects. They typically include similar information. On website like GitHub, **Markdown** is commonly used to format these files. Markdown is a format to make text files easily rendered with styles while still being readable. It is not strictly necessary to use Markdown, but it is a helpful tool to learn! There are often small variations in the syntax, but generally these are good to use:

~~~
# This is a Header

## This is a Second-Level Header, like "Section 2.1"

*This text is italic*, **This text is bold**,
_This is usually italic as well_. This is just regular text.

[This is a Link](https://brightspace.carleton.ca)

This `is a line of code` inside a paragraph.

* This is
* A bullet-point list

```c
int main(void) {
  // This is a block of code with a language specified
}
```
~~~

For this course, you can format your README however you like, but make sure to include your Student ID, your name, a description of the program, any citations required, and instructions for compiling and running the assignment. Consider something like this (but with proper details of course):

```
# Assignment 1, Data Entry
This program has a few features and they do things that a student would describe more specifically that this vague sentence is. Maybe I even take a moment to remind myself of what the learning goals were so when I read this code in 5 years I understand where I was!

## Building and Running
1. Open a terminal and navigate to the folder containing the program's files,
2. Type the command `whatever compiles this`,
3. You should see something specific when that happens to know it worked!
4. From that directory, you can run `this command` to execute the project.
5. If you want to test it, you can use `this other command`!
6. Maybe some Usage instructions to teach someone how to interact with your program once it is running

## Credits
- Developed individually by Me (133700042)
- Performed Q&A with GPT-4 via ChatGPT (September 37th version) to learn about `scanf()`, transcript of chat attached as `CHATGPT-SCANF.txt`

```

Appendix F: Changelog

1. 2025-09-22: Updated to add new tests in the loader object file, clarifying that you must loop and update pointers in each room when shifting entries in the global array.