COMP2401

Assignment #1: Ghostly Data Entry

Getting Started with C

Goals

In this assignment you will be writing a program in C using the Ubuntu Linux environment which is run on the course virtual machine (VM). You will get started with arrays and basic code elements in C.

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

- **Write** functional C code at a second-year level on the course VM,
  - Basic user input and output using `printf()`, `scanf()`, and arrays
  - Writing modular, documented functions in a consistent way,
- **Utilize** *nix-style workflows, including the use of virtual machines and the Linux operating system,

Preparation

You can find helpful information for this assignment in the following places:

- **Tutorial 1:** Using a VM and Linux to write programs,
- **Chapter 1:** Systems Programming and C Basics. Chapter 2.5 might be helpful, but goes more in-depth than needed for right now.

Assignment Context (Optional Read)

Each assignment will include optional context to motivate the content of the assignment.

You took the job! Excellent. Welcome to the Carleton University Ghost Hunters Society (aka **CUGHS**)! I know that we are quite dated at this point, umm… our tech, I mean, but this is all about having fun, and it is still fun! See, we go to farm houses, schools, campsites, and use our array of sensors to detect if "ghosts" are in rooms. We've… never seen one directly, but we are sure with your help, we can keep developing our systems to make sure everyone can have a fun time searching together!

Your first job: A skill test. We use something called an "EMF Reader" - it reads electromagnetic fields from things like lights, cameras, wiring in the walls, and, oh! of course, *ghostly entities*. It gives each room of the building a Room ID, and reports back a reading between 0.0 and 5.0 for each room. Can you help us do better than writing things down in a small notebook? We really appreciate your help!

1. Overview

There will be extra callouts throughout that provide some extra information beyond the score of the assignment. They are clearly distinct so that you can choose to overlook them in your review.

In this assignment, you will be creating a simple command line interface in C which prompts the user for pairwise data which represents data collected during a ghost hunt, using the `scanf()` function. It prints this data in a nice way using `printf()` and formatting specifiers. You will also need to use a prescribed algorithm to sort the data. Chapter 1 of the course notes describes what you need to know to use `scanf()` to accept two entries on the command line and how to print data in a nice ways, as well as how to translate your understanding from languages like Java over to C.

1.1. Data

🏁 **Goal**

In every assignment - in every programming experience - it is vital that you take time to understand the **data abstraction**. What data is being represented? What are the boundaries of it? How is it being used? This helps you to understand the context around your problem solving, but during this course we will also learn the limitations of different data types and how to pick the best ones for the job.

A single log entry is represented by these numerical fields:

- **Room ID:** Integer value within the range of 13300000 and 13379999 inclusive,
- **Electromagnetic Field Reading (EMF):** A floating-point value between 0.00 and 5.00 inclusive.

The **collection** of log entries are stored in two **synchronized arrays**. That is to say, if we have two arrays `a[]` and `b[]` we might say we have a “log entry in index 0”, meaning `a[0]` and `b[0]` all represent different fields of the same log entry.

🔥 **Tip**

Later in the course, we’ll learn about **structures**, which are stored very similar to arrays, but are better suited to representing collections of fields that belong together. For this assignment, however, we will be using synchronized arrays. The **Engagement Task Bonus** in Section 3.6 will ask you to optionally refactor your code to use structures for engagement marks.

1.2. Control Flow

With each assignment and project, at or shortly after release, there will be a **video posted alongside the assignment** that walks through the execution of a successful program, notes possible edge cases you might be wondering about, and may also mention a few helpful tips for completing the assignment.

A thorough understanding of the control flow and the data are vital before progressing with any assignment so that you are not trying to follow the instructions without any context so that you can make informed decisions about how to approach your code.

Note: In the assignments, error codes, capacities, and more are usually provided as defined values in the provided code. Make sure to review the provided code to use the appropriate definitions in your code to represent these if it is otherwise not specified.

The program should follow the following control flow upon execution:

1. The user enters zero or many entries:
   a. Each entry is a Log ID and a floating point value for EMF reading
   b. The system checks to see if each value was valid, and if it was invalid, prompts the user to retry
      • A valid entry is just an entry where all values are within the expected ranges
   c. The user enters entries until either they type `-1 -1` or until the arrays are full
   d. If the arrays are full and the user tries to add a new entry, print an error message and quit the program without completing; this may not be the most graceful behaviour, but it is what is requested
   e. Example of two entries being added:

```
> 13300083 4.1
> 13300104 1.5
> -1 -1
```

2. The system prints out all entries in a nicely formatted table and then the total number of entries,
   a. The print should be in a nice table formatted according to the guidelines in Section 3.2
3. The system sorts the data in descending order of EMF reading and prints the list again

1.3. Restrictions

For full marks, you must adhere to the following restrictions and the requirements described throughout. In general, try to only use techniques described so far in class or in the recommended/required readings.

1. You must accept all of the fields for an entry with one call of `scanf()` per entry on a single line,
2. You must use `printf()` specifiers to create a nice looking table when printing entries according to the guidelines in Section 3.2,
3. Implement the sorting functionality as required,
4. In general for all assignments, you may not modify code except where stated. E.g., do not modify the return values/parameters of functions, the values defined in `#define`, or provided functions you are not meant to modify or you may face major deductions.

2. Reading and Running the Code

- **Extract:** Extract and review the files in `a1-posted.tar`. The tutorials and overview video provide some information on extracting `tar` files. You can also run `man tar` in the terminal to view the "manpages" (manual pages) for `tar` to learn how.
- **Comments:** Assignment 1 has many comments to explain some C code. You may remove these if you wish.
- **Files:** This code is broken into two files, `main.c` and `readings.c`.
  - Usually, we try to keep `main.c` very minimal, only running the main function. Most of our automated grading tests will test the functions **independently**, with out own `main.c`, and so they cannot be in the same file as a `main()` function.
  - We will learn a better way to build these later, but for now you can use the command `gcc -Wall main.c readings.c -o a1`. Using `-Wall` is required, which turns on a few more helpful and important warnings.
    – See Appendix Appendix A: for more information about what this command means and why it isn't great.

Tip

You can use `man` to get the documentation for just about any C function and command in the terminal. They are important to learn to read, but can be tricky when getting started. An open-source, community effort to write simplified manpages can be found at https://tldr.sh but is **unofficial** and you should be careful when executing commands from online.

3. Instructions

3.1. Getting Entries

Feature 1 Description: I want to be able to enter data entries on the command line to be used later.

Feature 1 Requirements: Feature is implemented in `get_entries()`.

1.1. Entries are entered one line at a time, with ID and EMF separated by a space (e.g., `13300000 3.4`),

1.2. User has a nice prompt telling them what to input,

1.3. Stops accepting user input when the user types `-1 -1` or the arrays reach the capacity defined by a provided `#define` value,

Feature 1 Implementation: These are implementation requirements for relevant function, in no order.

1.4. In `get_entries()`:

- 1.4.1. Returns the number of elements in the arrays entered by the user,
- 1.4.2. Must accept both ID and EMF with a single call to `scanf()`; this can be called repeatedly in a loop, but accept both in a single format string,
- 1.4.3. Arrays must remain synchronized; i.e., the third element in the EMF Readings array is the reading for the third room in the Room ID array.

1.5. In `main()`:

- 1.5.1. Create two arrays of the predefined capacity to store EMF Readings and Room IDs and an integer to store the resulting size,
- 1.5.2. Call `get_entries()` to get user's input.

3.2. Printing Data

Feature 2 Description: I want to see the data that was entered printed to the screen in a clean way.

Feature 2 Requirements: Updates `main()` to print the data using `print_entries()` after `get_entries()` returns,

2.6. Printed data is formatted as follows using `printf()` print formatting:

- 2.6.1. Two columns with the titles "Room ID" and "EMF", about 10 spaces and 5 spaces wide,
- 2.6.2. Each room ID and EMF on a single line aligned as columns,
- 2.6.3. EMF has exactly 2 numbers after the decimal place,
- 2.6.4. Print the total number of entries in a user-friendly way.

Stop! Test! Submit! This is a good time to stop, test your code, and submit partial progress.

3.3. Validating Data

Feature 3 Description: I want to make sure the EMF and Room ID data I enter is valid.

Feature 3 Requirements: Validation behaviour implemented in `invalid_room()` and `invalid_reading()` and `get_entries()` is updated to validate inputs.

3.7. `invalid_room()` returns `C_OK` if the Room ID is (13300000 <= Room ID <= 13379999), otherwise it returns the appropriate predefined error value,

3.8. `invalid_reading()` returns `C_OK` if the EMF Reading is (0.00 <= EMF <= 5.00), otherwise it returns the appropriate predefined error value,

3.9. Update `get_entries()` so that the user receives an informative error message when at least one of the inputs are invalid, and prompts them to enter another value. If both are invalid, showing only one as invalid is acceptable.

⚠ **Warning**

For this assignment, adding an error message into `get_entries()` is okay, but in general it is not always the best practice, and for most assignments in the future the functions will be broken up so that `main()` handles most of the error handling as part of its duty of coordinating the main control flow.

3.4. Sorting Data

Feature 4 Description: I want the data to be sorted by EMF reading so that the rooms with the highest EMF appear first.

Feature 4 Requirements: Sorting behaviour is implemented in `find_max_index()` and `sort_entries()`.

4.10. `find_max_index()` takes in an array of EMF values and the number of valid elements,
  - 4.10.1. Returns the index of the highest value in the array, or an appropriate error flag.

4.11. `sort_entries()` takes in two arrays (Room IDs and EMF reading values) and the number of entries,
  - 4.11.1. Both arrays are reordered so that the entries are sorted in descending order of EMF (highest EMF at the front, lowest at the end),
  - 4.11.2. Arrays remain synchronized; the EMF reading for a room always stays paired with its corresponding Room ID,
  - 4.11.3. The function does not use any C standard sorting functions.

Feature 4 Implementation: These are implementation requirements for relevant functions, in no order.

4.12. In `find_max_index()`:
  - 4.12.1. Loops through the EMF array and identifies the index of the highest EMF value,
  - 4.12.2. Returns that index to the caller or an appropriate error code if one cannot be found.

4.13. In `sort_entries()`:
  - 4.13.1. Creates **deep copies** of the arrays before sorting that are used to find the next highest index,
  - 4.13.2. Uses a loop over each position in the output arrays:
    - 4.13.2.a. Calls `find_max_index()` on the copied EMF array to locate the current highest value,
    - 4.13.2.b. Places the found EMF value and its associated Room ID into the correct position in the **original** arrays,
    - 4.13.2.c. Marks the used EMF value in the local copy with a sentinel value (e.g., −1) so it is skipped in future iterations.
  - 4.13.3. Returns a success flag if sorting completes successfully, or an error flag otherwise.

3.5. Finishing Up

Make sure to test your code and ensure it is all working correctly. Review the video posted alongside the submission to make sure it follows the correct I/O and control flow.

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

3.6. 1% Optional Engagement Task Bonus

i 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.

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

We can structure our code quite a bit better than we are doing here. When we are working with collections of data, we are frequently working with either sequential data or associative data; that is, data that has an order or data that maps values to named keys.

Arrays are how we handle sequential data, but in C, we can use **structs** to create **structured data** that associates keys to values. While you might be tempted to think of them like classes in other languages, they are quite a bit simpler. Really, they are represented in essentially the same way as arrays in machine code, but the keys used in the source code get replaced with offsets from the start of the “array” data that is our struct. By defining a **struct** type, you’re really saying, “This field name is x bytes from the start of the struct’s data.”

The EMF and Room ID data are **much** better suited to be represented by a struct to keep the two fields grouped together.

Additionally, it is easy to make mistakes when copy-pasting important data across source files, like the `#define` lines or forward-declarations for functions. Copy-pasting makes it difficult to make changes, as you have to change it in many locations.

Challenge: There are two components to this challenge.

6.14. Create a new `struct` type that represents a single Entry, which has a Room ID and an EMF reading,
  6.14.1. Replace your multiple arrays with a single array of this struct type and modify your code to work with the new structures rather than the old arrays.

6.15. Create a new header file `readings.h` which contains the `#define` values and the forward declarations for all of the functions in `readings.c`, and include this header file in both `readings.c` and `main.c`.

3.7. Reflection

You do not need to submit the following reflection, though you may choose to include responses in your README. They will not be assessed, but it might be helpful to consider these questions to get the most out of this assignment.

7.16. Why is it so important to send the size of an array into functions when we pass an array?

7.17. Why might we need to specify the exact size of arrays in our source code and not have them automatically expand as needed like Python lists?

7.18. What considerations did you need to make when writing code in C that you didn’t necessarily have to make in a language like Python?

4. Packaging and Submission

For this assignment, you will submit three files to Brightspace: `main.c`, `readings.c`, and `README.md` which have been updated to include the necessary information (as well as any AI transcripts not included in your README if AI was used). **Note:** In later submissions, you will be required to submit a single file archive file containing all of your work, but to focus on C syntax in Assignment 1, you may submit three separate files. `readings.h` is also permitted if you complete the Engagement Task bonus, but you will still need to submit the updated files to the Private Engagement Task Submissions forum to be credited.

Alternatively, you may submit the files packaged into a `.tar` file, such as `a1_<studentid>.tar`, but make sure to verify that the `tar` file can be extracted with the correct files by downloading our submission and testing it. Future submissions may **require** a tar file be submitted, so it is good to get practice with this.

Grading information is on the next page

5. Grading

Info

Note: Assignments will be graded by TAs using a simpler rubric than the final project so that they have more time to provide feedback. The submissions will all be assessed partly using some autograding and then by TAs on a simpler and more general rubric. There may be some subjectivity in the rubric given to TAs, but we aim for consistency, and you should take advantage of the **Second Chance Policy** described in the course outline to improve your mark before any grade disputes are sent.

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.

- `get_entries()` correctly accepts two numbers on one line and loops until reaching `-1 -1` and stores them.
- `invalid_reading()` and `invalid_room()` both return the correct values.
- `find_max_index()` correctly returns the max index.
- `sort_entries()` correctly sorts the test data.
- All functions return the appropriately pre-defined error codes in easily detectable error states.

5.2. Manual Review

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

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

Appendix A: About Warnings, Building, and GCC

The command for this assignment will resemble `gcc -Wall main.c readings.c -o a1`. Chapter 1 describes this a little bit and we will review this in class, but let’s break down what is happening.

`gcc` is the “GNU Compiler Collection” (formerly GNU C Compiler), a collection of compiler tools for various programming languages. You can read more about it here. The job of a compiler is to translate translation units (C source files) into Object Code, and then `gcc` uses other tools to create a final executable.

Building a Program

Each `.c` file gets compiled into a single object file. This object file contains the machine code instructions for the file as well as placeholders for where any referenced functionality is. For example, `main.c` does not have the instructions for `get_entries()`, but it does need to execute them, so when compiled to `main.o` it cannot execute, but once it knows about `get_entries()` it can fill in the placeholder.

Once all of the files are compiled into object files, they get **linked** together – all of the placeholders get filled in if necessary.

By default, `gcc` will put the object files in a temporary directory and remove them, alongside any other steps in the build process. `gcc main.c readings.c -o a1` states that the two C files should be compiled into two object files, linked, and result in the `-o`utput file named `a1` which can be executed.

Warnings

Warnings are **essential** to coding in C. **Errors** only occur when it is impossible to compile the program. **Warnings** indicate areas where there *may* be a problem. By default, it will only warn for the most egregious issues. `-Wall` is almost always used because it includes many very helpful warnings without too many warnings that are likely non-issues. While individual warnings can be toggled on or off, `-Wall` enabled all of the most avoidable problems, but some can just warn because you aren’t yet done your code. A few include:

- Warnings about unused variables, unused functions, or using a variable that hasn’t been initialized to a value
- Warnings that a branch in your code may leave a variable uninitialized
- Warnings that you are comparing possibly incompatible data types or formatting prints/inputs with the wrong data types

You shouldn’t ignore any basic warnings, and most of the warnings in `-Wall` can be avoided using better coding practices that make your code run more consistently.

We can choose to suppress specific warnings to de-clutter the warnings while you work. For example, if you have placeholder functions and do not want warnings about unused functions or variables, you can add the flags `-Wno-unused-variable -Wno-unused-function` to your `gcc` call.

It is important not to ignore warnings, but to understand what they mean and try to understand how you can get around it. Sometimes in this class we will do things in non-standard ways to experiment and learn and allowable warnings will be noted in assignments accordingly, but in general if you compile frequently and keep warning free, you will run into fewer issues down the line by catching them early.

Appendix B: 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 C: 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 D: Changelog

- 2025-09-05: Updated wording to say `sort_entries()` rather than `sort_readings()` to match the provided definitions.