Ethan Simmonds (1423402)

DEVELOPER ASSESSMENT

REPORT

Dear Junior Developer,

There is a lot left to be desired with you have you submitted. One common problem in many sections of the program is a lack of coherency in either design or implementation. Their effects are quite present in every respect so I hope you adopt some of the following advice I offer so you will develop better solutions in the future.

Code Style

The programming style you have put into the program makes for very difficult for me to understand, and it feels as if you would be the only one able to accurately tell me what is happening. Incorrect bracket usage coupled with poor naming choices for your functions do not describe their use well enough. My suggestion with you here is to consider the programming layman when writing code. Stick to a style of writing general enough for even hobbyist developers to understand it at first glance. This means conforming to standard naming and style conventions. There are many tried and tested examples of this online, so do please look at them when you have the chance.

There is also a lack of any comments anywhere in the code. While you aren't expected to describe each and every line in the program, some top-level descriptions of what each aspect of the workflow is doing would be nice. This may manifest as brief descriptions at the top of your functions, or perhaps describing the process of the function and what the expected outcome is. I would also suggest for what you do have that you check for spelling errors. There is a particularly egregious example in what you have submitted (starts with *e*).

Ethan Simmonds (1423402)

Resource checking

There are a handful of examples that your resources are being misused. Perhaps you aren't realising it as you're doing it so I will offer the remedies in the following:

One large example is memory usage. Nothing allocated is being adequately deallocated after they are no longer used. In the quest of developing a high-quality solution it is crucial programs start and finish cleanly using as minimal amount of resources as possible. When your objects have finished their use they are still sitting in memory until the program dies. If this occurs many times there will inevitably be a great deal of bloat, which we want to eliminate.

My advice for you is to design a freeing function which you will call to free all memory allocated to your client struct object and all clients within it when you are finished with them. You could do this manually in the main function however it is unclean code, and is prone to failure. In the follow-up submission I have designed this exact freeing function. It is simple, and I hope you incorporate such a feature in the future.

Another large example is your usage of files. You open a file which is never closed after it is used. This large problem can be solved with a simple single line. Read up! I would suggest you look at how you extract each field from the line in the file too. A small hint before you start looking at the follow-up submission is there are too many symbols.

There is a small issue with the amount of compilation issues being returned. When I run your program it is spitting out many warnings, which you have opted to hide via your makefile settings. While your program as it stands does successfully compile, it comes at the cost of being very unsafe. Hiding these issues won't make them go away, and some of them are important to fix lest you want a messy application. I suggest you read up on pointers and how the C language uses them effectively, because there are quite a few warnings for pointer safety.

Ethan Simmonds (1423402)

Implementation process

I suggest you reconsider the implementation of how the program is finding clients. In your submission you have chosen to sort by field before searching, making for a time complexity of $O(n^2)$. This can be reduced to an O(n) problem by eliminating the sort feature altogether. I think it is an unnecessary feature considering the size of your client list (looks like 50 as shown on line 98) and thus is discouraged on the grounds of poor practice to do so. While I have implied they aren't needed anymore your sort functions still have errors in them, which I have fixed in the follow-up solution.

User Interface

The user interface for your program is very threadbare and doesn't fully convey how or when the user is supposed to be interfacing. It is very confusing to operate since it doesn't return any sort of prompt to let me know when it is ready to use. It would be nice to know this - even if it goes straight to an input menu.

On the topic of an input menu, it may be in your best interest to design a list of available commands the user can input in order to use the program. This can go alongside or even be a part of the landing menu I hope you will implement in future projects. This way the user knows what to expect. As for making an input menu, I suggest this is one of the first things you do. If you establish this functionality first then you can expand on offering convenience features for the user at a later stage much more effectively.

Convenience features you might want to consider include partial matching for attributes in the database. Fields such as email addresses have the potential to be quite long, and for the user can be quite irksome to put in especially if a typo is made. Another suggested feature to add is some sort of helper command incase the user feels they are lost or have forgotten how to operate part of the program. Adding some sort of page (in the console) which clarifies the options they have will do favours to perhaps some of the more...forgetful individuals. Lastly, and perhaps it will even help you even as a developer, is to include the expected output for each field. This means for emails you specify before input that the input needs a clear and valid mailbox address and domain. For phone numbers perhaps how to include number prefixes? Just something to think about.

Ethan Simmonds (1423402)

There is very minimal feedback given to the user at all aspects of the program execution. In-fact I thought the program was hanging when I first started it. It took looking at the code before I realised it was expecting input the entire minute I waited for it. Not a very good first impression! I suggest you include some confirmation for every action in the program so that the user is able to navigate the program with ease.

Feedback is a very important part of the user experience and you should follow some sort of convention or template when designing it - even if it is a product of your own. The main practices when telling users about the status of the program is to:

- Distinguish the output from the input. Line breaking is a simple and effective method.
- Map each action with an appropriate response. Does the user want to search phone numbers? It would make logical sense for the program to return some sort of confirmation it is doing exactly this and not something else.
- Technical jargon in the interface must be as minimal as possible.
- When applicable, updating the user when the program is performing tasks.
- When menuing, place required input before any descriptions. (*0 -> exit* is an example)

Also, please make sure the interface presentation is sharp and correct. Watch the formatting of your print statements already in the program.

Attached at the bottom is a document of changes between your submission and my follow-up.

Best Wishes,
Ethan

Ethan Simmonds (1423402)

Changes

- struct *s* renamed to struct *client.*
- References to *emialAddress* changed to *email.*
- Struct field names changed.
- Duplicate inclusion of stdio.h has been removed/fixed.
- Booleans used for search functions.
- Pointer formatting fixed.
- Function header names much more clearer.
- For loop fixes (such as variable *i* missing in sph).
- Static integer variables *i*, *j* no longer global - referenced within functions instead.
- Search functions have their *while* statements turned into *for* statements.
- Search functions now use *strcmp.*
- Function *freeClients* added to deallocate memory from clients + struct.
- Bracketing fixes (mainly in loops).
- *FILE f* changed to *FILE file*. For consistency's sake.
- Input checking from file is fixed.
- File now successfully closes after usage.
- *Math.h* inclusion removed (not needed).
- Most print statements in the main function block are much more descriptive.
- Memory allocation now uses *sizeof(char)* instead of *sizeof(val[0)).*
    - A handful of similar examples were fixed too.
- Program breaks out of execution properly when command is *0* (memory properly freed too).
- Functions which return integers now return booleans (not an issue, but for coherency).