Alex Lake-Smith 5400306
Etude 13: Bug Squashing

Hi Junior Developer,

I have recently been given your code to review and check for errors and unfortunately I found a few areas of code which needed to be corrected. With a few edits I have tried to help make the program more concise and understandable and will give you a few pointers on how I did this, and some good practices to keep in mind for future. I will further outline some aspects that I think you should work on trying to improve if you want to be offered further development opportunities:

1. Methods and variables with unclear purposes.
2. No comments.
3. Inconsistency of style throughout code.
4. Various syntax errors.
5. Security and Memory issues
6. User Experience
7. Inefficient approach to solving/finding queries.

Below is a further breakdown.

1. Methods and variables with unclear purposes.

Now, I presume that this program is supposed to try and query a certain person in a database and return that person's records if it matches the query entered. When I first got given your code it was a little bit ambiguous and the methods and variables throughout had no clear purpose and had confusing names. I would only advise abbreviations where the purpose is outlined clearly in comments or the people using your code understand what it is supposed to do. I have since renamed your methods so that their purpose is much clearer. I recommend you further change the name of your struct or other variables to something clearer.

2. No Comments

This sort of leads on from my last pointer about being extremely clear whilst also being concise when providing code others may need to use. All of your methods and most of your variables did not have comments so I had to figure out what the parameters and return values were myself. I have since added comments which you can view and learn to style similarly next time you are given a project to code.

3. Inconsistency of style throughout code

I would advise that in future you should stick to one way that works for you when developing code and try to stick to that style throughout the whole program. For example I found that only some of your for/if/while statements would have curly braces and then some of them would not. It is good practice to include these for any statements you may be declaring and also to keep indentation consistent throughout also. You must remember to always close the braces when you are finished writing your conditional statements, otherwise it may lead to many confused senior developers such as myself.

4. Syntax Errors

Your program that was given to me would also not compile correctly without producing errors/warnings. Your *makefile* which was provided was quite misleading and actually redirects any errors that may be caused by the incorrect code and simply bins them so you as the user cannot see them. Instead I would recommend next time trying to compile your code as such:

gcc -Wall -O2 -o clientsearch clientsearch.c

This way you will be provided error messages which you can refer to when trying to fix aspects of code that are not working correctly. One area of interest that I had to fix was some of your scanf() calls which were providing errors that they were expecting different variable/pointer types to what they were actually being passed. I have corrected these syntax issues and a list will be provided below which lists all the errors found.

5. Security and Memory Issues

A professor at University told me once to never trust the input that the user may give your program as they may try to take advantage of it and use it for malicious purposes. To avoid this issue with user input I added a buffer to all expected strings input and would discard that input if it was over the character limit, this way of development we can always trust that we have done our best to protect the program and if the data input the user provides is incorrect, only they are to blame. I also used fscanf() instead of using scanf() to read in the input as it allowed you to set a buffer for the maximum number of characters allowed for a certain input. This would avoid any malicious attempts from users that may be trying to break our code. The danger of using gets() is that you have to know exactly how many characters you will be reading, so that you can make your buffer large enough. This is why I set a limit of 90 characters for each string variable being read in. I also had to fix quite a few memory issues, you were not freeing all allocated memory correctly so I included a free_all function which did that. I also made sure to free any additional allocated memory such as the memory allocated to the query variable *val.* I found a problem in the code which resulted in memory leaks when trying to exit the code by using command: 5. The issue occurred because I would EXIT_SUCCESS from the program but forgot to free memory that was allocated to variables such as val, ss or s. I instead inserted my free_all function and used free before printing “Exiting program” and then terminating the program with EXIT_SUCCESS.

6. User Experience

It was also important that a better user interface was implemented within the program. When I first received your code I was unsure of how to use it as a user, this is why I implemented print statements which instructed the user on their next steps required throughout the program. This way the user was kept informed and understood what it was the program was intended for and what you want them to enter. I have included case statements which control what query is done and have included an exit case which frees all allocated memory, prints an exit message and exits the program.

7. Inefficient approach to solving/finding queries

The purpose of your program was to ‘find people’ in a list of data selected by the user which is read into the program. I liked how you set up your struct and incrementally read in each individual client's data however I was a bit confused with your sorting methods. Currently each sort method runs at $O(n^2)$ as you have two for loops that iterate through the data, simply searching through the list with your individual find functions which I have kept has instead an O(n) worst case time complexity (much better). I think an idea like adding a Binary Search algorithm may be ideal if you have to sort and search large sets of data or information, however you have to search 4 different kinds of attributes for your client which is not ideal as a typical BST algorithm only has one attribute that it has to search for. I instead only used the find methods to iterate through the array of structs and locate the index that the record should be, if it can’t be found I instead return a 0 and print a message saying that this person cannot be found. You also have to think about how you originally wrote your program to respond if a person was found. Originally it only printed a yes/no if the person was located or not. Do you not think it would be more beneficial that if a query was successful that the user would be able to see all the details of the query they wanted to find? I have since implemented a print_client function which does this for you, I would suggest having a look at the code and thinking about how this was done.

I appreciate your efforts and understand that you may have found this first assignment stressful, however I think with further practice, taking in my suggestions and having a look at useful C documentation especially pointers, memory, user input and how to correctly compile your program would serve as a good refresher or learning experience. Attached on the next page is a list of all the errors I have found and have corrected since you sent me your code. Please feel free to contact me if you have any questions or want any further explanation.

All the best,

Alex Lake-Smith

Errors found

- Changed all instances of the *emialAddress* typo to *emailAddress*
- Fixed instances of loop declaration errors where i/j is not set to 0
- Added { / } braces to for/if/while statements that were missing.
- Renamed the file *clientsearch.c* for clarity for user
- Added comments above functions and to variables
- Improved grammar/spelling errors
- Changed method names to be more informative
- Added print statements to direct user on how to query people in the database
- Added a free_all method to free allocated memory given to the struct.
- Corrected memory allocation to use *sizeof*()
- Removed the sorting method and instead used the search methods only, sort was not necessary for this task

- Changed the compile method to allow the user to see if any errors are found with the code
- Changed User Experience to be more clear and instructive.
- Corrected indentation on while/for/if/switch statements
- Added ind variable which keeps track of which index in the struct array to print for client information
- Added a if/while statement while reading in the chosen file. Outputs message if file cannot be found and also reads in all client data while it has not reached the end of reading the file.
- Removed multiple count declarations within the main method
- Replaced *get*() with *scanf*() for user input
- Removed *&* from in front of string struct variables
- Changed phone number to also be *char** instead of an int so that if an 0 is at the front it does not get cut off from output
- Used *strcmp* instead of = when comparing strings in search methods
- Added if statements in the main method to ensure that user input is correct before searching for a client
- Moved memory declarations to be in the correct location of the while loop.
- Added a *found* variable which returns either 1/0 this determines if a client is found and if output is 1 it will print their information
- Freed memory of internal variables such as *val*
- Corrected pointer declarations to be consistent to remove compilation errors and to pass expected values
- Created a README.txt file which can be used for instructions on how to use the program.
- Included character buffer to ensure user input does not exceed buffer value, this is for security purposes.
- Removed static ints i and j and declared them as local variables inside functions
- Added count parameter to all find functions that would keep track of how many clients are in the S* ss.