Leo Venn STUDENTID:3430125

Hello junior developer,

The code you have written up contains errors and bad practices. But fear not! I have written this letter to address this and help you become a better programmer. I have not listed all the issues but have compiled at least one example of every type that I found. Near the end I have compiled issues in a more general sense, such as under GETTING USER INPUT, I have addressed multiple issues that fall into multiple categories.

SYNTAX/VISUAL FORMAT:

I will not address every syntax issue you have. But errors like this:

```
if (ss[i]->firstName > ss[j]->firstName)
```

Where you have ">" instead of "->" are littered throughout the code. Make sure when you write your code you constantly run it for debugging purposes and catch syntax errors.

Also, you should try to tidy your code so its easy to read. For example, in this struct the asterisk are in different places.

```
struct S{
    char *firstName;
    char* lastName;
    int phone;
    char* emialAddress;
};
```

I have fixed all these syntax/visual format issues.

IRRELEVANT LIBRARIES / REPEATED CODE

It is bad practice to include libraries that you are not using. This of course generalizes further to unused declared variables and functions. Also, there is obviously no use in repeating code that has already served its purpose. In the example below you have a repeated include statement and you do not actually use the math library.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
```

I have removed the redundant code.

ILLOGICAL VARIABLES/DATA FORMAT

In your struct to store client records you use an int to represent phone numbers. However, it would be better to use a string to do so as phone numbers can start with 0 which is removed when passed in phone number is stored as an int.

I have changed phone to type char*

```
struct S{
    char *firstName;
    char* lastName;
    int phone;
    char* emialAddress;
};
```

```
typedef struct ClientRecord {
  char *firstName;
  char *lastName;
  char *phone;
  char *emailAddress;
}
ClientRecord;
```

NAMES/NAMING CONVENTIONS

Ideally you should name your struct using typedef so it can be referenced by a variable name (shown in images in previous section). Also, it is good practice to name your functions something descriptive. E.g. sfn could be instead search_first_name. This also applies to variables and as you will see in later screenshots I have changed pretty much all your variable names although it should be obvious what variables these changes were associated with.

I have made these changes to your code

STATIC VARIABLES AND IMPROPER USE OF THEM

Although static variables are fine. You should only use them in cases where they are necessary. These variables:

```
static int i, j;
static int count;
```

Are used in your search and find functions. Although count is used fine. The i,j variables used in your for loops for these functions can be defined in the functions to keep things simpler and maintain a more logical scope. Also, there is a logic error in which you do not set these variables to 0 in some functions such as your fln function.

```
int fln(struct S** ss, char* s){
    while(++i < count){
        if(ss[i]->lastName == s)
            return 1;
    }
    return 0;
}
```

In this case. “I” could be any value depending on previous functions that incremented it. So, there is a serious logic error.

I have fixed this by defining i and j in the functions. I have also used count as a variable that is passed to these functions so the code has no static variables.

LOGIC/DESIGN ERRORS

There are numerous of these littered throughout. In say the sfn function:

```
void sfn(struct S** ss){
    for(i = 0; i < count; i++)
        for (j = 0; j < count; j++)
            if (ss[i]->firstName > ss[j]->firstName)
                ss[i] = ss[j];
                ss[j] = ss[i];
}
```

You have set the value at ss[i] to the value at ss[j] but then you set ss[j] to the value at ss[i] which is the same as itself. To swap variable you need a temp variable to hold the variable as shown below. Also, I have compiled all of your search functions into a single function to reduce visual space and passed to it the search option to decide what to sort. Depending on design this isn't always the way to go but it's theway I have done it. I also modified your sort function to be more efficient but that's not mandatory.

```
/*Sorts client records based off of user input*/
void sort(ClientRecord * record, size_t n, char option) {
  ClientRecord temp;
  size_t i = 0;
  size_t j = 0;

  for (i = 0; i < n; i++) {
    for (j = 0; j < n - 1 - j; j++) {
      if (option == 'f') {
        if (strcmp(record[j].firstName, record[j + 1].firstName) > 0) {
          temp = record[j];
          record[j] = record[j + 1];
          record[j + 1] = temp;
        }
      } else if (option == 'l') {
        if (strcmp(record[j].lastName, record[j + 1].lastName) > 0) {
          temp = record[j];
          record[j] = record[j + 1];
          record[j + 1] = temp;
        }
      } else if (option == 'p') {
        if (strcmp(record[j].phone, record[j + 1].phone) > 0) {
          temp = record[j];
          record[j] = record[j + 1];
          record[j + 1] = temp;
        }
      } else if (option == 'e') {
        if (strcmp(record[j].emailAddress, record[j + 1].emailAddress) > 0) {
          temp = record[j];
          record[j] = record[j + 1];
          record[j + 1] = temp;
        }
      }
    }
  }

}
```

In your find functions you also have logic errors. The > operator does not compare strings alphabetically in these cases and you need to use strcmp(). As in when you find a record, you immediately return without looking further to find more matches. You also don't return what is found, just that you found it. It would be better to find a match you display that match to the user, such as that clients' details. I have changed your find functions and put them into one function as with the sort functions. They also now print out the found records.

```
int fln(struct S** ss, char* s){
    while(++i < count){
        if(ss[i]->lastName == s)
            return 1;
    }
    return 0;
}
```

```
/*Finds records based off of client input*/
void find(ClientRecord * record, size_t record_size, char * search_parameter, char option) {

  int matchCount = 0;
  size_t i = 0;
  if (option == 'f') {

    for (i = 0; i < record_size; i++) {
      if (strcmp(search_parameter, record[i].firstName) == 0) {

        printf("[%s; %s; %s; %s] \n", record[i].firstName, record[i].lastName, record[i].phone, record[i].emailAddress);
        matchCount++;
      }
    }

    if (matchCount != 0) {
      printf("%d entries found for first_name: %s \n", matchCount, search_parameter);
    } else {
      printf("No Entries found for first name: %s \n", search_parameter);
    }
  }
```

MEMORY ALLOCATIONS

Even though you have used malloc correctly, you should have your malloc functions contained within a function that you can check for errors when memory allocating as this is very common. Also, you have not created re-allocation functions which means that you could run out of space to write to given a malloc size that was too small. You should dynamically update the size of your mallocs with re – allocs to address this issue. I have added in these functions.

```
/*Allocate memory. */
void * emalloc(size_t s) {
  void * result = malloc(s);
  if (NULL == result) {
    printf("Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }
  return result;
}

/*Re-allocate memory. */
void * erealloc(void * a, size_t s) {
  void * result = realloc(a, s);
  if (NULL == result) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
  }
  return result;
}
```

READING FROM DATABASE

So, a couple of issues here. First you are only read in 50 records. Given a database with more records, you will miss out on them, and given less you will create unnecessary, empty records (you have also set your ss malloc to be of size 100 so we knew we can store at least 100). It's better to just use a while loop to iterate over all records. This is pretty easy as we know the input is always valid. Also, you malloc size 80, but do not set a limit on the size of a piece of data that can be read in can take. A first name greater than 80 characters will pose serious issues so please be very wary of reading in data and mallocs as they pose big memory issues. Therefore, make sure to set a limit on the size of what you can read in. 80 should be big enough so I have used that for my first and last name in fscanf. (note: you also do not malloc for phone number and the mallocs you do have all use firstname as sizeof)

```
for(i = 0; i < 50; i++){

    s->firstName = (char*) malloc(80 * sizeof(s->firstName[0]));
    s->lastName = (char*) malloc(80 * sizeof(s->firstName[0]));
    s->emialAddress = (char*) malloc(80 * sizeof(s->firstName[0]));

    fscanf(f, "%s %s %d %s", &s->firstName, &s->lastName, &s->phone, &s->
emialAddress);

    ss[count] = s;
    count += 1;
{
```

```
do{
  client.firstName = emalloc( 80 * sizeof( * client.firstName));

  client.lastName = emalloc(80 * sizeof( * client.lastName));

  client.phone = emalloc(20 * sizeof( * client.phone));

  client.emailAddress = emalloc(50 * sizeof( * client.emailAddress));

  clientRecArray[recordCount++] = client;
  if (recordCount == recordCapacity) {
    recordCapacity += recordCapacity;
    clientRecArray = erealloc(clientRecArray, recordCapacity * sizeof clientRecArray[0]);

  }


} while (4==fscanf(file_database, "%80s %80s %20s %50s", client.firstName, client.lastName, client.phone, client.emailAddress  ));
```

We might think to dynamically update the size. But this is actually a bad idea as users can just enter insanely long names and we will store them, taking up memory. It's better to just set an appropriate limit size. Also, instead of setting a set size of 100 records, we can dynamically reallocate the client record array to fit all records like so.

```
clientRecArray[recordCount++] = client;
if (recordCount == recordCapacity) {
  recordCapacity += recordCapacity;
  clientRecArray = erealloc(clientRecArray, recordCapacity * sizeof clientRecArray[0]);
```

GETTING USER INPUT

There are quite a few errors in this section as well as some design philosophies I'd like to mention I won't go over every change I've made, I'll just point out the more important issues and the general approach I've taken towards getting and processing user input.

First, the biggest issue. Memory leakage. This issue here is you create a buffer of size 10, assuming the user will enter a command that can be stored within this safely, then you use the gets function to get the user input and store it in this buffer. However, the BIG issue here is gets does not check the buffer size and so can easily take a user input longer than the specified size and write into memory it shouldn't. Obviously, this can be a massive issue and so you should be very careful about allocating memory and what functions you use. I have changed this by using fgets which allows you to choose a input stream and sets a limit on the size of the input that can be entered by the user to prevent writing over memory we want kept. I set this input size to a max of MAX_MEM, a variable that have declared, and read into into a buffer of the same size. This means any input longer than this buffer is cut off and we don't write over memory.

```
while(command != 0){
        char* val = malloc(100*sizeof(val[0]));
        gets(buffer);
        command = atoi(buffer);
        gets(buffer);
        strcpy(val, buffer);
```

```
char inputLine[MAX_MEM];

fgets(inputLine, MAX_MEM, stdin);
char * token = strtok(inputLine, " ");
strcpy(searchParameter, token);
token = strtok(NULL, " ");
searchParameter[strcspn(searchParameter, "\n")] = 0;
commandAmount = 1;
while( token != NULL ) {
  strcpy(searchClientRecord, token);
  searchClientRecord[strcspn(searchClientRecord, "\n")] = 0;
  token = strtok(NULL, " ");
  commandAmount ++;
}
```

I have also made it so the search parameter and value to search for are given in one input line and split using strtok, this creates an easier user experience instead of having to input multiple times to complete one search. I keep track of the number of arguments a user has entered and in the image below I do appropriate checks for valid and invalid cases with appropriate error messages.

```
if (commandAmount == 2) {
    if(strcmp(searchParameter, "f")*strcmp(searchParameter, "l")*strcmp(searchParameter, "e")*strcmp(searchParameter, "p") == 0){
        sort( & clientRecArray[0], recordCount, searchParameter);
        find( & clientRecArray[0], recordCount, searchClientRecord, searchParameter);



    }else{

        printf("Invalid arguements: Please see command format below (f for first name search, l for last, p for phone number, e for email)");
    }

}else if (commandAmount == 1 && strcmp(searchParameter, "q") == 0){
    break;
}else if (commandAmount == -1){
    break;
}else{
    printf("Invalid arguements: Please see command format below (f for first name search, l for last, p for phone number, e for email)");
}
```

In your code you have made other assumptions, such as the user will input a number in as their first command which you can then use atoi on, similarly you have assumed the user will also enter correct data for the search value. This is obviously a big oversight. When in doubt, always assume the user is trying to break your code. You should have checks at each step to ensure the user has inputted correct values and print error messages that tell them what they should do when they haven't.

Make sure you are always performing sanity checks on user input and keeping in mind how a user could break your code and addressing it. The last thing you want is to have your code broken/exploited.

FREEING MEMORY/MEMORY LEAKAGE

Another big issue in your code is that you do not free memory. Take for Example:

```
for(i = 0; i < 50; i++){

    s->firstName = (char*) malloc(80 * sizeof(s->firstName[0]));
    s->lastName = (char*) malloc(80 * sizeof(s->firstName[0]));
    s->emialAddress = (char*) malloc(80 * sizeof(s->firstName[0]));

    fscanf(f, "%s %s %d %s", &s->firstName, &s->lastName, &s->phone, &s->
emialAddress);

    ss[count] = s;
    count += 1;
{
```

You malloc 50 times for each data field and then store it in ss. This in itself is fine, given that all of these mallocs are appropriately freed afterwards. If not, then over time your system can get run out of memory as more and more space inside it is reserved for these mallocs. This is a big oversight and can be exploited by users. MAKE SURE you free all mallocs at the end of the program. You also have this problem too when getting user input:

```
while(command != 0){
        char* val = malloc(100*sizeof(val[0]));
        gets(buffer);
        command = atoi(buffer);
        gets(buffer);
        strcpy(val, buffer);
```

The char* val is malloc'd multiple times but this time the error is even more egregious, in this case a reference to the previous malloc is not stored as it is with the records in ss, therefore we couldn't even free the memory if we wanted too! This memory leakage is once again a very big issue and takes up memory. I have changed the code to t free the mallocs (names have been changed). Also, since you have set the for loop to run 50 times you are creating potential unnecessary mallocs that just take up system space. Only malloc something when you need too and make sure to free it afterwards

```
/*free memory*/
  for (i = 0; i < recordCount; i++) {
    free(clientRecArray[i].firstName);
    free(clientRecArray[i].lastName);
    free(clientRecArray[i].phone);
    free(clientRecArray[i].emailAddress);
  }
  free(clientRecArray);
```

USER INPUT/EXPERIENCE/INTERFACE

User experience is important and covers many issues so should be utmost importance.

Firstly, you expect the user to enter a database file, but the user has no way of knowing this. You should tell the user that a file is expected through some error message and perform the appropriate checks to make sure the file exists.

```
/*Open database*/
FILE * file_database = fopen(argv[1], "r");


/*Check if arguenments given is 2*/
if (argc != 2) {
  printf("Invalid arguements. Make sure you execute the program in this format: %s <database_file>\n", argv[0]);
  exit(0);
}

/*File not found*/
if (!file_database) {
  perror("Error: ");
  exit(EXIT_FAILURE);
}
```

Secondly, as I have already addressed before. Its better to have the search query in one line rather than 2 inputs and make sure to print appropriate error messages when the input is invalid, so the user knows how to use your program changed the program so when it starts it always prints “Command Format…” and when an invalid input is given it looks like this.

```
Command Format: f <query>, l <query>, p <query>, e <query>, q (quit application)
Invalid arguements: Please see command format below (f for first name search, l for last, p for phone number, e for email)

Command Format: f <query>, l <query>, p <query>, e <query>, q (quit application)
Thank you for using my programm.
```

Thirdly. Your commands used in your switch statement are not intuitive.

```
case 2:
printf("looking for firstname %s\n", val);
sfn(ss);
printf("found it? %d\n", ffn(ss, val));
break;
case 3:
printf("looking for lasname %s\n", val);
sln(ss);
printf("found it? %d\n", fln(ss, val));
break;
case 4:
printf("looking for email %s\n", val);
sph(ss);
printf("found it? %d\n", fph(ss, atoi(val)));
default:
break;
```

They are simply numbers, and the user has no idea they even need to enter numbers. As shown in the image above this one, I used letters to represent search commands and told the user what they represented as well as how an entire command is formatted. It is very important you do this otherwise no one will want (or even be able to) use your program.

I used the letter commands to determine what to do as shown in the below image:

```
if(strcmp(searchParameter, "f")*strcmp(searchParameter, "l")*strcmp(searchParameter, "e")*strcmp(searchParameter, "p") == 0){
    sort( & clientRecArray[0], recordCount, searchParameter);
    find( & clientRecArray[0], recordCount, searchClientRecord, searchParameter);


}else if (commandAmount == 1 && strcmp(searchParameter, "q") == 0){
```

Fourth. You have it so that you are always:

```
printf("found it? %d\n", ffn(ss, val));
```

Even though your sfn, sln… functions could return 0 and not have found anything. Once again this is just confusing for the user.

MAKING YOUR CODE ACCESSIBLE

You do not have any comments in your code, nor do you have a readme explaining how your code works. These are important for other developers such as myself when coming in and trying to expand upon code you have written. Make sure you comment your code clearly and communicate to future developers your design.

COMPILING

Finally, your makefile does not require -lm and redirecting to /dev/null is generally bad practice.

```
main: main.c
	gcc –o main main.c –lm 2> /dev/null
```

```
main: main.c
	gcc -Wall -o main main.c
```

I hope that this letter has been helpful. There are other issues but they are small and fall under one of the categories I have listed here. The most important thing to take away from this is make sure to

1) Address memory leakage
2) Free malloc memory
3) Make your code accessible to users and developers

That's all. Good luck with your future coding!