Étude 9: Bug Squashing

Here is the list of changes made to the main.c file that the junior developer created. Each listed fix will include a brief explanation of the problem to be learned from.

- Removed duplicate header inclusion of the <studio.h> header which is redundant and not necessary.

```
5 #include <stdio.h>
```

- Fixed typo in the S struct which just adds more clarity to what the char pointer is pointing at. To make this widespread across the program you can choose to “Rename Symbol” which will rename all instances of “emialAddress” to “emailAddress”.

```
11     char* emialAddress;
```

```
11     char* emailAddress;
```

- Renamed the struct from S to Client just for clarity to what information the struct is storing.

```
7  struct S{
8      char *firstName;
9      char* lastName;
10     int phone;
11     char* emialAddress;
12 };
13
```

```
8  struct Client
9  {
10     char *firstName;
11     char *lastName;
12     char *phoneNumber;
13     char *emailAddress;
14 };
15
```

- Changed phone to phoneNumber just for name clarity and the type from int to char*. The reason is because phone numbers generally use hyphens, possibly parentheses and indication for the country code.

```
10     int phone;
```

```
12     char *phoneNumber;
```

- Renamed count to numberOfClients just for clarity to what was being counted. The variable j was also unused.

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

```
16 static int i, numberOfClients;
```

- Removed unnecessary sort methods since the data we are dealing with is so little. The search algorithm that was already implemented by the junior developer was linear search. Even though it has a O(n) it is fast enough for any array we’d be dealing with regards to the client data. For smaller arrays with only a few dozen or hundred elements the overhead of sorting and implementing more complex search algorithms may outweigh the benefits. Linear search only noticeably slows down when dealing with larger arrays typically in the range of hundreds or thousands of elements.

```
17 void sfn(struct S** ss){
18     for(i = 0; i < count; i++)
19         for (j = 0; j < count; j++)
20             if (ss[i]->firstName > ss[j]->firstName)
21                 ss[i] = ss[j];
22                 ss[j] = ss[i];
```

```
32 void sln(struct S** ss){
33     for(i = 0; i < count; i++)
34         for (j = 0; j < count; j++)
35             if (ss[i]->lastName > ss[j]->lastName)
36                 ss[i] = ss[j];
37                 ss[j] = ss[i];
38 }
48 void sem(struct S** ss){
49     for(i = 0; i < count; i++) {
50         for (j = 0; j < count; j++) {
51             if (ss[i]->emialAddress > ss[j]->emialAddress) {
52                 struct S *s = ss[i];
53                 ss[j] = ss[i];
54                 ss[j] = s;
55             }
68 void sph(struct S** ss){
69     for(; i < count; i++) {
70         for (; j < count; j++) {
71             if (ss[i]->phone > ss[j]->phone) {
72                 struct S *s = ss[i];
73                 ss[i] = ss[j];
74                 ss[j] = s;
75             }
```

The junior developer used bubble sort as well for the sorting algorithm and when combined with linear search is inefficient having a $O(n^2)$. Bubble sort was also not correctly implemented in any of the sort functions.

For example, in the function `sfn` (sorting by first name) the comparison is incorrect and should use strcmp() > 0 rather than just >. This is because > should be used for comparison of numbers while if you want to compare strings you should be using strcmp() instead. In the inner loop the loop condition is $count - i - 1$ instead of $count$ because in each iteration of the outer loop, the largest element gets placed at its correct position at the end of the array. Therefore, in subsequent iterations, we don't need to compare and swap elements that are already in their correct positions. By reducing the range of the inner loop, we can avoid unnecessary comparisons and improve the efficiency of the algorithm. The swap operation ss[i] = ss[j]; ss[j] = ss[i]; is incorrect and will not perform a valid swap since the smaller element that is being swapped down will immediately be overwritten by the larger element. A temporary element needs to be created to store the value of the smaller string that is being swapped.

```
1 void sfn(struct S** ss){
2     for(i = 0; i < count; i++)
3         for (j = 0; j < count; j++)
4             if (ss[i]->firstName > ss[j]->firstName)
5                 ss[i] = ss[j];
6                 ss[j] = ss[i];
7 }
```

```
1  void sfn(struct S** ss) {
2      for (i = 0; i < count; i++) {
3          for (j = 0; j < count - i - 1; j++) {
4              if (strcmp(ss[j]->firstName, ss[j+1]->firstName) > 0) {
5                  char* temp = ss[j];
6                  ss[j] = ss[j+1];
7                  ss[j+1] = temp;
8              }
9          }
10     }
11 }
```

A mixture of these issues is also present in the other sorting algorithms. My recommendation would be using a more efficient sorting algorithm like quicksort in combination with a search algorithm like binary search for maximal efficiency.

- The find functions now searches for all instances of clients with the search parameter instead of just returning the first client found. The while loop has been changed to a for loop just for better clarity to what happens in the loop. There were also two bugs in the declaration of the while loop. Firstly i wasn't properly initialized to 0 anywhere which would have caused an error. Secondly assuming that the junior developer intended i to be 0 at the start of the loop you shouldn't be pre-incrementing

(++i) and instead should be post-incrementing (i++) otherwise the first client will never be searched against or i should be initialized to -1 instead.

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

```
// Find clients by last name
void findLastName(struct Client **clients, char *s)
{
    bool found = false;
    for (i = 0; i < numberOfClients; i++)
    {
        if (strcasecmp(clients[i]->lastName, s) == 0)
        {
            found = true;
            printf("Client found!\n----------\nName: %s %s\nEmail: %s\nPhone Number: %s\n----------\n", clients[i]->firstName, clients[i]->lastName, clients[i]->emailAddress, clients[i]->phoneNumber);
        }
    }
    if(found == false) printf("%s not found.\n", s);
}
```

Just as the sort functions above the comparison should be using a string comparison function rather than just a standard operator. For the search functions strcasecmp() was used as well so the search wouldn't be case sensitive. The function then prints out the client's details if the search parameter matches that of the current's client. The found flag is then set to true and the "not found" text will only be outputted if no matching client was found.

The return type of the function has also been changed from int to a void. In the original file 1 was returned if the matching client was found and 0 otherwise. The function would only search to see if the client was matched but not any of the details of the client.

```
                        printf("looking for email %s\n", val);
                        sem(ss);
                        printf("found it? %d\n", fem(ss, val));
```

Similar changes were made for all the find functions.

- Added a function that displays all the clients that have been saved into the array of clients. This was just for debugging purposes as well as an extra feature to the program. It prints out the same as if a client was found in the find functions but for every client.

```
void displayClients(struct Client **clients, int numberOfClients)
{
    // Iterate over all clients
    int i;
    for (i = 0; i < numberOfClients; i++)
    {
        // Print the client's information
        printf("----------\nName: %s %s\nEmail: %s\nPhone Number: %s\n", clients[i]->firstName, clients[i]->lastName, clients[i]->emailAddress, clients[i]->phoneNumber);
    }
    printf("----------\n");
}
```

- When reading in a file from the command line argument proper error checking to ensure that there was a second command line argument, or the file exists were absent. The program now accepts the program name from the command line argument or will prompt the user for a valid. If the file from command line can't be read in or none is specified, then the user will be prompted for a valid file that can be read in. The benefits of this is error handling, avoiding undefined behavior, program stability and overall user experience.

```
int main(int argc, char **argv)
{
    char fileName[MAX_LENGTH];
    FILE *file = NULL;

    // Use the file name from the command-line argument
    if (argc >= 2)
    {
        file = fopen(argv[1], "r");
        if (file == NULL)
        {
            printf("File not found from command line argument\n");
        }
    }

    bool validFile = (file != NULL);

    // Prompt the user for the file name until a valid file is entered
    while (!validFile)
    {
        printf("Enter the file name for the client data: ");
        scanf("%s", fileName);
        file = fopen(fileName, "r");
        if (file == NULL)
        {
            printf("File not found. Please try again.\n");
        }
        else
        {
            printf("File found.\n");
            validFile = true;
        }
    }
```

- The size allocated for the clients should be a pointer to the struct rather than a pointer to a pointer of the struct. Since ** is equivalent to an array the space allocated should be a large value multiplied by the size of each element that will be stored in the array in this case it would be struct Client *.

```
struct S** ss = (struct S**) malloc(100*sizeof(struct S**));
```

```
struct Client **clients = malloc(100 * sizeof(struct Client *));
```

- The values from the file are now correctly read in.

```
    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;
```

```
    while (fgets(line, sizeof(line), file) != NULL) // Read the file line by line
    {
        // Allocate memory for the client
        clients[numberOfClients] = malloc(sizeof(struct Client));
        clients[numberOfClients]->firstName = malloc(80 * sizeof(char));
        clients[numberOfClients]->lastName = malloc(80 * sizeof(char));
        clients[numberOfClients]->phoneNumber = malloc(80 * sizeof(char));
        clients[numberOfClients]->emailAddress = malloc(80 * sizeof(char));

        // Parse the values from the line using sscanf()
        sscanf(line, "%s %s %s %s", clients[numberOfClients]->firstName, clients[numberOfClients]->lastName, clients[numberOfClients]->phoneNumber, clients[numberOfClients]->emailAddress);

        // Increment the number of clients
        numberOfClients++;
    }
    // Close the file
    fclose(file);
```

The loop was using a fixed iteration with a count of 50 which has been changed to be a while loop which uses fgets to read the file line by line until the end of the file is reached. The function sscanf() is also used instead of fscanf() because the while loop now reads in line by line of the file so sscanf() is used to extract all the data from each line.

The memory location has also been fixed it should be allocating memory based on the actual data type char rather than the member of the struct. The same s pointer is also reused in each iteration of the loop meaning that all the elements in ss array will point to the same element. Instead, a new instance of the struct should be malloced for each element in the array i.e., clients[numberOfClients] = malloc(sizeof(struct Client)). For each element in the clients array each member of that element will be malloced (80 * sizeof(char)). The numberOfClients is then iterated.

It's also important to close the file that is being read in after all the data is extracted to release system resources, flush buffered data, prevent data corruption, release file locks, and ensure code portability.

- Added a short print statement just to confirm to the user the number of clients that were loaded from the file.

```
// Display the number of clients loaded from the file
printf("Welcome! %d clients loaded from the file.\n", numberOfClients);
```

- The menu has been reimplemented to provide a more user-friendly and structured approach to interact with the program, select tasks, and perform operations on the client data.

```

int command = 10;
        while(command != 0){
                char* val = malloc(100*sizeof(val[0]));


                gets(buffer);
                command = atoi(buffer);
                gets(buffer);
                strcpy(val, buffer);
                switch(command){
                        case 1:
                        printf("looking for email %s\n", val);
                        sem(ss);
                        printf("found it? %d\n", fem(ss, val));
                        break;
                        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;
                        }
                }
        }
```

```
// Display the menu and prompt the user for a command
char input;
char searchParameter[MAX_LENGTH];
do{
printf("This program provides the following functions::\n 1. Search by first name\n 2. Search by last name\n 3. Search by phone number\n 4. Search by email\n 5. Display all clients\n q. Quit\nSelect task : ");
        scanf(" %c", &input);
        switch (input)
        {
        case '1':
            printf("Enter the first name: ");
            scanf("%s", searchParameter);
            findFirstName(clients, searchParameter);
            break;
        case '2':
            printf("Enter the last name: ");
            scanf("%s", searchParameter);
            findLastName(clients, searchParameter);
            break;
        case '3':
            printf("Enter the phone number: ");
            scanf("%s", searchParameter);
            findPhoneNumber(clients, searchParameter);
            break;
        case '4':
            printf("Enter the email address: ");
            scanf("%s", searchParameter);
            findEmailAddress(clients, searchParameter);
            break;
        case '5':
            printf("Displaying all clients:\n");
            displayClients(clients, numberOfClients);
            break;
        case 'q':
            printf("Quitting...\n");
            break;
        default:
            printf("Invalid command. Try again.\n");
            break;
        }
    } while (input != 'q');
```

The while loop has now been replaced with a do-while loop just to make that the loop is at least executed once which is more intuitive because the user should see the menu at least once. A clear menu is displayed telling the user exactly how to use all the commands.

The program now also uses a simple scanf() to get the user input rather than reading input from the user using the gets() function, storing it in the buffer array, converting the input stored in buffer to an integer using atoi() and then assigning the result to the command variable. This can be done

much more easily by using different chars as the different cases which saves the conversion that occurs. Once in a specific case the program will then prompt the user for the specific search parameter to search the clients for. This is more intuitive as the users clearly know what they're inputting a value for the command versus the search parameter. The function scanf() uses appropriate format specifiers to read user inputs and considers the maximum length of the input string (e.g., searchParameter) to avoid buffer overflow issues. The max length of which was defined at the top of the file.

```
6 #define MAX_LENGTH 100
```

The default case for the switch-case now prints out an error message telling the user that an invalid command was used just for better user-experience.

The termination input is now q instead of 0 just because q corresponds better to quit e.g., like in Vim.

- The allocated space is now freed this was causing most of the segmentation in the code before. It loops through the array of clients freeing all the members of the struct as well as the struct itself and finally the array itself.

```
1    //  Iterate over all clients annd free the malloc-ed memory space.
2    for (i = 0; i < numberOfClients; i++)
3    {
4        // Free the memory allocated for the client
5        free(clients[i]->firstName);
6        free(clients[i]->lastName);
7        free(clients[i]->emailAddress);
8        free(clients[i]->phoneNumber);
9        free(clients[i]);
10   }
11   // Free the memory allocated for the clients array
12   free(clients);
```

- Return 0 at the end of the program. This is just a convention that indicates successful execution of the program. By convention, a return value of 0 signifies that the program terminated without any errors or issues.

```
203     return 0;
```

- Comments were also added throughout just to add clarity for any developers down the line who might use the code.