Bug Squashing - Etude 9

List of changes

Basic changes

1. Header files
    a. There was a double-up of `#include <stdio.h>` so the double-up was removed. The `#include <math.h> was also removed since our changes to the program did not require it anymore.

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

    b. We defined a constant MAX_LENGTH of 100 because we were working under the assumption that the input size would not exceed over 100 characters. This was used many times throughout the program to maintain consistency and quality assurance.

```
#define MAX_LENGTH 100
```

2. Quality of life changes; fixed typos, renamed variables to make code easier to read and understand and changed data types of some variables.
    a. Struct S was changed to Struct Client
    b. “emialAddress” fixed to “emailAddress”
    c. int phone variable changed to char* phoneNumber for clarity and also since phone numbers can often include non-number symbols (e.g ‘+’ or ‘-’).
    d. In function variables, struct S** ss changed to struct S** clients.
    e. Added comments to whole program for better code readability.
    f. Added better print statements to guide the user of the program with more clarity.
    g. Function names changed for more clarity (sem -> sortEmail etc)

```
struct S{

    char *firstName;
    char* lastName;
    int phone;
    char* emialAddress;
};

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

```
8
9  struct Client
10 {
11     char *firstName;
12     char *lastName;
13     char *phoneNumber;
14     char *emailAddress;
15 };
16
17 static int i, totalClients;
18
```

3. Comparators
    a. In the sorting functions, the junior developer compared strings incorrectly. They used the '>' operator rather than the `strcmp` function.

4. Memory
    a. In the junior developer's program, the same memory address 's' was being assigned to each element of 'ss' resulting in the data being overwritten in each iteration of the loop. To fix this, this memory allocation was moved into the loop so a new struct instance is created for each iteration.
    b. The memory allocation for ss (clients) was incorrect. ss is an array of pointers to struct S but the junior developer's implementation was allocation memory for ss as an array of pointers of pointers.
    c. At the end of the program, we iterate over all of the clients and free up any malloced memory space to fix the segmentation faults in the junior developer's implementation.

```
//  Iterate over all clients annd free the malloc-ed memory space.
for (i = 0; i < totalClients; i++)
{
    // Free the memory allocated for the client
    free(clients[i]->firstName);
    free(clients[i]->lastName);
    free(clients[i]->emailAddress);
    free(clients[i]->phoneNumber);
    free(clients[i]);
}
// Free the memory allocated for the clients array
free(clients);
```

5. Switch Cases
    a. In the Switch statements, a `break;` was missing after each case meaning all cases were being executed.

Reading in a .txt file

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

```
    FILE *f = fopen(argv[1], "r");
```

```
    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->phon
e, &s->emialAddress);

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

1. When reading in the .txt file from the command line, the junior developer's program does not even check to see whether there is a .txt file input when the program is run. This was changed so there is now a check to see whether there are two or more arguments (if there are, we will use the command line argument to open the .txt file). Alternatively, if there is no input file in the

command line, we prompt the user to input a valid .txt file name in a loop until the user input is a valid .txt file.

2. In the junior developer's implementation of the program, they used a "for-loop" to read in lines of the .txt file. This implementation meant that the program could only read in .txt files of length <= 50 (and even if the .txt file was less than 50, it meant the program would continue looping beyond the size of the input file). Therefore, this was changed to a while loop that would continue looping until we had iterated through until the end of the file.

```
    char fileName[MAX_LENGTH];
    FILE *file = NULL;

    // If there are two or more arguments, 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;
        }
    }
```

3. The file is closed once all the data has been extracted to allow for releasing system resources, flushing buffered data, preventing data corruption, releasing file locks, and ensuring code portability. This ensures that the file is properly managed and all associated resources are freed, promoting efficient and reliable execution of the code.

```
    char line[MAX_LENGTH];
    struct Client **clients = malloc(100 * sizeof(struct Client *));
    // totalClients = 0;
    while (fgets(line, sizeof(line), file) != NULL) // Read the file l
ine by line
    {
        // Allocate memory for the client
        clients[totalClients] = malloc(sizeof(struct Client));
        clients[totalClients]->firstName = malloc(80 * sizeof(char));
        clients[totalClients]->lastName = malloc(80 * sizeof(char));
        clients[totalClients]->phoneNumber = malloc(80 * sizeof(cha
r));
        clients[totalClients]->emailAddress = malloc(80 * sizeof(cha
r));

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

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

4. In the new implementation, `sscanf` is used rather than `fscanf` as `sscanf` allows for more flexibility in parsing the input data as we are not reading directly from the file itself but rather the formatted string (line) from `fgets` which is used to read the file line by line.

Switch Statement

1. In the switch statement, more print statements have been added to further clarify the functionalities of the program and to prompt the user for user input.

2. The sorting functions have been removed from all of the cases as they were actually adding no real functionality to the program. Instead, these sorting functions have been extracted and implemented into our custom displayClients function which will be touched upon further in the report.
3. Have changed the '0' switch case to 'e' for a better user experience (e for exiting the program).
4. There is now a default case in which a print statement tells the user that an invalid command was used.

Filter functions

1. As previously mentioned, all of the filter functions have been changed to use `strcmp` to compare the user input with the client variable correctly.
2. All of the filter functions have also been changed from returning an int (1 for if client is found or 0 if not) to now being a void function. Instead, if a client is found, the client details are printed out in the table format and if the client is not found, a print message stating the client was not found will be printed instead.
3. To further improve the filter functions, the function has been changed to implement a for loop which will iterate through the entire client array and print out the details of any clients who fulfil the filter criteria.

Display Clients

1. A new function that I have added to the program is the ability to display all clients. This function displays all of the clients in the system in a table. The table has 3 columns - name, email and phone number. In the name column, both the firstName and lastName variables are combined.
2. The reason for adding this function was so the user could clearly see all of the clients that have been imported into the system from the .txt file. It is also important as it allows the previously

Default sorting order:

Name	Email	Phone Number
John Doe	john.doe@example.com	123456789
Jane Smith	jane.smith@example.com	987654321
Michael Johnson	michael.johnson@example.com	555555555
Emily Davis	emily.davis@example.com	999888777
David Anderson	david.anderson@example.com	111222333
Sarah Wilson	sarah.wilson@example.com	444444444
Robert Taylor	robert.taylor@example.com	666666666
Jessica Clark	jessica.clark@example.com	777777777
Christopher Ma...	christopher.martinez@example.com	222333444
Olivia Lewis	olivia.lewis@example.com	888888888
Matthew Lee	matthew.lee@example.com	123123123
Ava Rodriguez	ava.rodriguez@example.com	987987987
Daniel Walker	daniel.walker@example.com	456456456
Sophia Hall	sophia.hall@example.com	789789789
James Wright	james.wright@example.com	321321321
Mia Green	mia.green@example.com	654654654
Benjamin Hill	benjamin.hill@example.com	159159159
Evelyn Adams	evelyn.adams@example.com	753753753
Andrew Turner	andrew.turner@example.com	258258258
Charlotte Collins	charlotte.collins@example.com	852852852
William Hughes	william.hughes@example.com	963963963
Amelia Carter	amelia.carter@example.com	741741741
Joseph Hernandez	joseph.hernandez@example.com	369369369
Harper Foster	harper.foster@example.com	147147147
Henry Gray	henry.gray@example.com	753951753
Ella Bennett	ella.bennett@example.com	852963852
Samuel Flores	samuel.flores@example.com	369258147
Victoria Reed	victoria.reed@example.com	741852963
Grace Watson	grace.watson@example.com	159357852

implemented sorting functions to actually serve a purpose in the new program. Now, when the user is wanting to display all of the clients, they have options to sort by firstName, lastName, email and phone number.

```
// Check the specified sorting order
if (strcasecmp(s, "firstname") == 0)
{
    printf("Here are all clients displayed in order of first name:\n");
    sortFirstName(clients, totalClients); // Sort clients by first name
}
else if (strcasecmp(s, "lastname") == 0)
{
    printf("Here are all clients displayed in order of last name:\n");
    sortLastName(clients, totalClients); // Sort clients by last name
}
else if (strcasecmp(s, "email") == 0)
{
    printf("Here are all clients displayed in order of email address:\n");
    sortEmail(clients, totalClients); // Sort clients by email address
}
else if (strcasecmp(s, "phone") == 0)
{
    printf("Here are all clients displayed in order of phone number:\n");
    sortPhone(clients, totalClients); // Sort clients by phone number
}
else
{
    printf("Default sorting order:\n");
}
```

Sorting Functions

1. In the junior developer's attempt to implement sorting functions in the program, they used a bubble sort algorithm which has a O(n^2). Not only is the implementation slow but it was also incorrect as they used "<" and ">" operators for comparing the strings to one another which does not work. In order to compare strings correctly, they needed to use the `strcmp` function.

2. To improve the sorting function, the bubble sort algorithm was replaced with quick sort. In the worst case, both bubble sort and quick sort actually have the same O(n^2), however in the average and best case scenarios, quick sort is significantly faster than bubble sort by having a time complexity of (nlogn). A quick sort algorithm works by recursively dividing the array into smaller sub-arrays based on a pivot point. This approach helps us increase the efficiency of our program especially when dealing with much larger client data sets. Also, the correct comparator method `strcmp` is also used. I implemented the quick sort algorithm for sorting by firstName, lastName, email and phone number. These are the functions that are called depending on the user input when they are prompted for how they want to view the client data.

Conclusion

Overall, the junior developer had some great ideas but there were clearly flaws in their understanding of C especially in memory management. The junior developer also lacks some attention to detail with repetitive blocks of code and typing errors. The junior developer could really benefit from taking the time to clearly lay out the program and assigning meaningful variable/function names in their program.