Etude #13 – Bug Squashing

Letter to junior developer, author of main.c and the makefile

Hi, I'm writing to inform you that there were bugs in both the makefile and main.c, regarding syntax, semantics, readability, and memory leakages. I am writing to you to inform you what bugs have been made, how I've updated some of these bugs and some tips on what to do next time. The revised code is attached, and I have written the comments and suggestions under headings below. If you have any questions, please don't hesitate to email me.

Makefile
First, the make file is throwing compiler errors off to the 'void' with > dev/null. This should be avoided, because it will not allow us to locate the errors in the code when compiling. When the code is ready to deploy, compiling with the -O2 flag will help for optimizing, and at the moment the -Wall flag returns some recommended compiler warnings, so I've added this to the make file. There is no need for the -lm in the make file. Also, the executable file should be named more descriptively next time, so other programmers can see what the purpose of the program is. I would perhaps suggest naming the program something like client_search or person_search, but the final name is down to your preference.

Main.c – Syntax Errors
It should be noted that there are two include statements for the system input/output (<stdio.h>) module, and the <math.h> header is not necessary. Only one is required, so I have removed the second include statement. These sorts of bugs are easy to let pass, so it is important to read through your code multiple times. The email address property of the struct person is spelt wrong, not that it will affect the compilation but makes the code more unreadable and confusing, so I've corrected the spelling, as this sort of thing can cause headaches later. Phone can be changed to be a string (which is a char pointer in C), because we may have characters within a phone number, and comparing phone numbers based on integer conversion unfortunately doesn't work (integers will strip any leading zeroes). When declaring pointer variables, the asterisk should be directly to the left of the char variables – firstName, lastName, emailAddress and phone. The search functions are ambiguously named, as well as the sorting functions, and so they have been changed. It is important to name the functions descriptively for readability purposes. The sorting functions are comparing strings with > and < but remember a string is char pointer so the C compiler will not be able to use such operators on strings, so I've added a call to strcmp to compare the strings instead. The file opening was moved to the top of the main function also because C forbids mixing declarations and code, and the same with other variables declared later. The functions such as ffn, fln, fem, fph all have while loops with the variable 'I'. The variable I is static, which is technically OK, but it may have changed to any number over the course of the program, so I've set I equal to 0 at the beginning of each of these functions. It might pay to only have these variables declared in the local scope of each function as local variables to avoid errors from having changed the content of 'I' somewhere else. For example, you might loop through an array starting from the variable I, thinking it is zero, but it was changed in another function and is now equal to the length of the array. The loop will never get through even the beginning of the array now. I also noted above that these functions were renamed for readability purposes. Lastly regarding these functions, you need to use curly braces for if statements and while loops that have more than one line. It is generally best practice to stick to curly braces in general instead of indentations for readability purposes.

You should return an integer at the end of the main function. An exit-code should be returned, a descriptive constant such as EXIT_SUCCESS or EXIT_FAILURE is better because it signals obviously to the programmer and other people reading the code, what exit-code is returned. The fscanf function is not secure and is for formatted input only. Instead, I've changed this to use getline for getting a line strtok for breaking the line up into first name, last name, email and phone number. This data is then entered into the people struct. In the switch statement I've changed the command to a char type instead of an integer, which makes it easier to read in commands and compare in the switch statement, although integer would work just as well, albeit requiring more complex functions such as atoi(). The file also must always be closed with fclose. You can read up about these functions in books such as Brian Kernighan's C Programming, or online at various websites. They are built into the standard libraries, so you don't need to use any third-party library.

Main.c – Efficiency
Outside of general syntax errors, the searching and sorting methods will not scale as good as other algorithms could allow. Better algorithms would include merge sort and binary search. Instead of duplicating the sort and search functions, why not create a general search function and a general sorting function? I will leave this up to you to think about.

Main.c - Readability
Readability is very important, for example it took me over an hour to figure out what the program was supposed to do. For this reason, I've updated the struct s to be called person, because the struct has properties that you would expect a person to have. I've also taken the liberty of renaming the ss array of person's to 'people'. In the switch statement, spelling mistakes have been corrected. There are two "looking for email" printf statements. The second one has been changed to "looking for phone number". It's very important to add comments to your code, especially code that is obscure. It might also be best to define important numbers such as the size of input and expected variables, at the top of the file, instead of having so-called 'magic numbers'.

Main.c – User Experience
In terms of user experience, the user will not know what 0 and 1 means after a search has been performed, so I've introduced a function that takes the integer as input and returns a "Yes" for 1 and a "No" for 0. Also, it isn't clear that the user of the program will know what commands what do, so I've added an extra command that prints to the user all 4 commands This is command 5, and it says to enter 5 when the user is first prompted. What happens if the user doesn't enter a file? This could be a major issue, so you need to check the file pointer for NULL and if it points to NULL, print to the stderr, and exit the program. This is all done in the main function.

Sanity Check
A sanity check is a basic test to quickly evaluate whether the input has been entered or checking that the data entered is smaller than a certain length, to ensure we don't run into buffer overflow problems. Because I have changed the function call to getline() when the file is being read, the entire line cannot be over the length of the second parameter. I've used a #define statement to set the buffer length to 80, which allows for consistency around the entire program. I would suggest making this more accurate and ensuring that each string token does not exceed 80 characters, otherwise they will not fit within the allocated memory. For the command, I use fgets with a size of buffer which is set at 80 characters (BUFF_SIZE which is defined at the top of the code), so the command and the search string cannot exceed 80 characters. This is a good sanity check, because none of our data will be greater than 80 characters.

Main.c – Memory Issues
There is no memory being freed at the end of the program which signals memory leakages. Memory leakages are caused by memory being allocated but not freed later in the program (before it is lost). I have added in some free statements at the end to illustrate how freeing works. The individual persons are freed first, then the pointer to the array of people.

Any calls to malloc could possibly return NULL if there is no more memory to allocate. This would mean that the program will break unexpectedly. We don't want this to happen of course, so I suggest creating a function called emalloc that you could call each time you wish to allocate memory to a pointer variable. The function will exit the program with an error message if a call to malloc fails, otherwise will simply return the pointer to the caller.

For lack of time, I was unable to spend any more time on memory leakages, however I would recommend using valgrind to detect any last leakages. I must stress to be careful to always free any memory that was allocated using malloc.

Summary of Suggestions

- Name the executable file more descriptively.
- Use more efficient searching and sorting functions – merge sort and binary search for example. This will make the program scale better with largers sets of data.
- Create an emalloc function to error check for malloc calls easily.
- Create a readme file so other programmers or users of the program can quickly edit or use the program without having to understand all of the code.
- Write more comments for other programmers (and the future you) to look back on and be able to understand what the code is doing at each step. I've started you off with some basic comments.
- Use valgrind to fix memory leakages.
- Also, think about using typedef to reduce the complexity of the struct declarations. This could also make the code more readable.

Good luck!