Bug Squashing

Hi Junior Developer, I have had a look at your program and wrote this manifest to help you identify some ways you can improve your programming in the future. Unfortunately the program has some significant issues, so I have interpreted the flow of your program and started from square one to demonstrate some best practices in C. The new code is attached.

Here are some general comments about your code.

Makefile

- Your makefile command doesn't have any warning flags enabled, flags such as -W -Wall and -pedantic can give you useful warnings to help ensure your code is well written.
- Your build target isn't very descriptive, give your executable a semantic name.

Main.c #includes and #defines

- It looks like you aren't using any functions from the <math.h> header, so this one isn't necessary and you don't need to include the -lm flag in your makefile.
- Your struct "S" looks like it is defining a client. It would be wise to give this struct a more semantic name like "Client" to avoid confusion when you are programming. Furthermore you can typedef your struct to a pointer so that you can simply declare variables as "Client clnt;" instead of "struct Client *clnt;".
- Strings are the appropriate data type for storing phone numbers, unfortunately integers will strip any leading zeros.
- Pointers (* and &) bind to the right token, so when you are creating pointers it is wise to group them with the right token instead of the left (e.g. "char* lastName;" should be written as "char *lastName;").
- There are a few spelling mistakes with your variable names (i.e. "emialAddress"). It would be wise to make sure your spelling is correct to avoid headaches in the future.
- To avoid potential side effects in your program, it is wise not to declare iterators and sizes in the global scope (i.e. i, j, and count). Keep these variables local and pass them as parameters if necessary.

Search and sort algorithms

- Unless you have only one command to execute and can fit it on one line, it is generally preferred to use curly braces instead of indentation to denote blocks. This improves the readability and maintainability of the code.

- It looks like you are using selection sort and linear search for your algorithms. The implementation for these aren't quite correct - you especially need to make sure that you start "i" at 0 in your search algorithm. What would be better is to use more efficient algorithms such as merge sort and binary search. These will scale much better with larger numbers of clients.
- You have a lot of code duplication between the sort and search algorithms for each attribute. You can condense these down into two functions by passing the required attribute as a parameter.

Main() function

- You are redeclaring "i" and "count" in your main function. These are different variables to the ones you declared in the global scope - watch out for that.
- You seem to be using a lot of "magic numbers". These are numbers that are seemingly pulled out of thin air. It is best to #define these at the top of your file. This will reduce headaches in the future if you need to change their value.
- Once again, make sure you use semantic variable names. "ss" could be called "list_of_clients" or something similar.
- When you use the malloc() function, you should always check to see if it returns a NULL pointer, otherwise it will likely break your program if the malloc() fails.
- It looks like you've got some indentation issues at your first for loop. This considerably changes the flow of the program and you need to be careful with where you blocks begin and end.
- You need to malloc your "s" pointer inside the for loop, otherwise it will not "create" a new client and will just be pointing at one client.
- fscanf() and gets() are generally considered dangerous and deprecated functions. It is preferred to use fgets() and sscanf(), however I have opted to use fgets() and strtok().
- Once more, semantic names are preferred for "val" and "buffer".
- Your prompt messages aren't very descriptive. Perhaps it would be useful to print out the details of any clients matched? Furthermore, a help command that displays the program's commands would be very useful.
- Your prompt commands aren't very intuitive. Instead of using numbers, you could use "f" for first name, "l" for last name, etcetera.
- With C programs, any memory you malloc() absolutely must be free()'d after you have finished with it. Programs like valgrind can help you ensure you don't have any memory leaks.

Here are some extra comments about my code which illustrate some of the points above, and introduce a couple more things to think about.

- emalloc() is a wrapper function for malloc() that includes error checking - this avoids code duplication in other functions because malloc() is very commonly used and needs to be checked for a NULL pointer every time it is used.
- The print_help_message() function prints a useful prompt showing you how to use the program.
- With the print_client() function, we can print useful information about matches we find.
- get_attribute() returns a specified attribute of a client. This is how we avoid the code duplication born of sorting and searching between different attributes.
- We are using merge sort and binary search - both very efficient algorithms.
- In the query() function, you can see that we are printing out all of the matches we can find, and not just the single match returned by binsearch().
- In main(), all of our variables have descriptive, albeit long names. This is almost always preferred to short, meaningless names.
- We are performing error checking on our fopen(). This is essentially a malloc() and needs to be released at the end of the program with fclose().
- For arrays of unknown size, we first ascertain the size of the array (line 180), then we malloc() the array (line 181), then we can populate the array (while loop beginning at line 184).
- For our main control loop (starting at line 209), we added a prompt: “>>> ”, improved the quality of our info, error, and help messages, and changed the program’s commands to be more semantic and memorable.
- At the end of our program, we are free()ing all memory malloc()’d and returning a SUCCESS.

A quality-of-life feature you should consider including, as I have demonstrated in my code, is partial and case-insensitive string matching, for example typing “f b” would list all entries with a first name that begins with “b” or “B”. This is achieved with the strncasecmp() function of the <strings.h> header.

Unless your workplace prescribes it, the syntactic style you choose to code in doesn’t really matter, however when you pick a style, you should always ensure that you are consistent.

Best of luck