Assignment 2 – Refactoring and Improving the Survey system

The Department of Computer Science (CSC) at the University of Victoria appreciated the work done by the SENG 265 students in Assignment 1 and decided to extend the partnership. After the pilot project conducted on the survey on attitudes towards Computer Science, some fellow department faculty decided to advise students to produce a more efficient and well-designed system. With the improved system, they plan to later extend it to handle more flexible surveys.

According to Mary Weiss, the department survey coordinator, the largest challenge in the pilot project was the need to work with fixed static memory, the lack of improved data structures and handling all the code in a single file.

Mary also points out that she may want to filter data and present filtered results, for example, for a given undergraduate program, for students either born in Canada or not, for a given age group established by the researcher, or a combination of these three demographic data. Thus, they decided to change the input file format, adding one line for the number of survey responses before the survey responses *per se*. Then, they also added one or more lines after the survey responses to describe one or more filtering options. For the given filtering configuration, the statistics should be computed and written to the output for the selected subset of respondents.

To wrap up, she said, "Look, last time we hired people to do something like that, the folks made very messy software. When we hired another person to maintain the code, it was easier to find a needle in a haystack. When coding the system, don't forget to modularize everything, separate data input handling from statistics calculations, separate calculation for each statistic, separate printing output data. Use different files for different categories of functions. You know, those things to make maintainers' lives easier in the future."

Programming environment

For this assignment, please ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, give yourself one or two days before the due date to iron out any bugs in the C programs you have uploaded to a lab workstation. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Start your Assignment 2 by copying the files provided in **`/home/rbittencourt/seng265/a2`** into your **`a2`** directory inside the working copy of your local repository. Do not put your code files in subdirectories of **`a2`**, put them in **`a2`** itself (this is important for the automated grading scripts).

Commit your code frequently (**`git add`** and **`git commit`**), so you do not lose your work. We will look at the code commits you did in this assignment. **We require at least three different commits** (you should split your work into parts). The final grading will take that into account. In the end, do not forget to **`git push`** into your remote repo.

Finally, when you finish Assignment 2, be sure to send the final version to the remote repo with a git push command.

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact the course instructor as soon as possible. The URLs of significant code fragments you have found online and used in your solution must be cited in comments just before where such code has been used.

Description of the task

Someone else was hired to produce a user interface for the survey. You just need to follow the given requirements (including memory efficiency and software design requirements) and make the tests pass. In doing that, and by using adequate modularization, the interface developer may use your functions to process data and to show it in different interfaces.

You should use the test files to read all the input data to your program. In these text files, lines that start with **#** are comments and should be ignored by your program. The other lines have data whose description is in the comment lines.

You should send your program's output to the standard output (usually the computer screen or console). By using Unix pipes and redirection, one will be able to send your program's output to a file. You will use this strategy yourself to test your program.

You must NOT provide filenames as input parameters to the program, nor hardcode input and output file names. Code that does either one or the other will receive grade zero.

Your program must NOT hardcode questions, question types, answers types and responses. Instead, you should read that information from the `stdin`. Code that does either one or the other will receive grade zero.

Implementing your program

The C program you will develop in this assignment:

1. Starts from either your **`survey.c`** file from Assignment 1 or from the standard answer provided by the instructor (to be sure you will not be late because of problems from the previous assignment). After setting up your environment in your local repo **`a2`** directory, and copying the files given for the Assignment 2, rename your source file to **`dyn_survey.c`** and use it as a start.
2. Adapts the input handling process to the changes in the file format required by the department survey coordinator (see input files). Reads the input file with the test configuration, question descriptions, question types (direct or reverse), answer types, number of responses, responses *per se*, and the filtering options, and stores data appropriately.
3. Splits the functionalities into different source files and controls compilation by using appropriate header files, the **`make`** tool and a given **`Makefile`** file. This latter file describes which files are needed for compilation. That automates the build process by simply running the **`make`** command. The only function inside **`dyn_survey.c`** must be the main function; there must be at least one function in each of the **`input_handling.c`**, **`processing.c`** and **`output.c`** files (use common sense to split your functions from Assignment 1 into the appropriate source files). The main function must call those functions in the required files.
4. Uses dynamic memory appropriately: all array variables must be declared as pointers and must be allocated in dynamic memory with the appropriate size derived from the information given in the input file (you can use the **`emalloc()`** function given). Memory must be freed whenever needed to avoid memory leaks. You can estimate a single maximum size for the variable that reads a line from **`stdin`** to make it simpler.
5. Uses appropriate struct types: a **`Respondent`** (and their demographic data) must be stored in a struct variable; a **`Response`** must also be stored in a struct variable, with both a **`Respondent`** struct and the answers array for that particular respondent as struct members; all struct types must be declared in **`typedef`** statements inside the **`dyn_survey.h`** file.
6. Handles each filtering step by analyzing the dynamic array of **`Response`**s and marking the ones who stay and the ones who are filtered out. You can use whichever strategy you prefer for filtering. One possibility is using a filtering array. Another one is having a filter variable member in the struct.
7. Computes the requested statistics according to the user's filtering options expressed in the input file.
8. Finally, writes the requested results into **`stdout`**.

Assuming your current directory **`a2`** contains your source files as well as the compiled executable file (named **`dyn_survey`**), and a **`tests`** directory containing the assignment's test files is also in the **`a2`** directory, then the command to run your script will be.

```
% cat tests/in01.txt | ./dyn_survey
```

In the command above, you will pipe the input query text from the **tests/in01.txt** to the **survey** script and the output will appear on the console.

```
% cat tests/in01.txt | ./dyn_survey | diff tests/out01.txt -
```

The **diff** command above allows comparing two files and showing the differences between them. Use **man diff** to learn more about this command. The ending hyphen/dash informs **diff** that it must compare **tests/out01.txt** contents with the text output from your program piped into **diff**'s **stdin**. This is how you should test your code.

The **tests** directory may be retrieved from the lab-workstation filesystem inside **/home/rbittencourt/seng265/a2** to guide your implementation effort. Inside **tests**, there are six input files and six output files: the ones finishing with **01** are related to the simpler tests; the ones finishing with **02** are related to more complex tests and so on. The first three sets of files are similar to the ones in Assignment 1, and are there for checking your code still passes the tests after you refactor your code to improve its efficiency and design as requested. The other three sets of tests relate to filters applied to the data.

Start with simple cases. In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong".

You should commit your code whenever you finish some functional part of your it. This helps you keep track of your changes and return to previous snapshots in case you regret a change. Whenever you feel your code is working correctly (even though not fully solving the assignment), you may push it to the remote repository.

Exercises for this assignment

You may develop your code the way that suits you best. Our suggestions here are more for facilitating your learning than as a requirement for your work. You may not need to do the exercises below if you want to practice deeper problem solving skills. But, in case you get stuck, you may look at them as a reference. On the other hand, if C programming seems difficult to you, you may use them as a script to learn and practice.

1. Copy your previous **survey.c** (or the standard answer provided by the instructor) as **dyn_survey.c** program into the **a2** directory within your git repository.
    a. Practice with **stdin** by reading it line by line with loops and **fgets()** or **getline()** and printing the output with **printf()** or **fprintf()**.
    b. Adapt your input handling to the new input file format and test whether your changed code passes tests 1, 2 and 3.
    c. Split the functionalities of your **dyn_survey.c** into different source files as required in step 3 in the previous section. Keep the appropriate functions in the appropriate source files (use files names and common sense to decide how to split them). Move the function prototypes onto the appropriate header files. It may take a while until you get used with this and your code is running with **make**. Be sure that each of your **.c** files import the appropriate **.h** files to compile without warnings. Once this is working, you will see that your build process will be very simple (just run **make** or **make clean**).
    d. Create a function to count the number of tokens in a line. Use that information to learn the size needed for your arrays. Change your code to allocate arrays as dynamic memory (you may use the supplied **emalloc()** function). Be sure to deallocate memory (using the **free()** function) when the array is no longer needed. Pay attention how to handle bidimensional arrays as dynamic memory. Test whether your changed code still passes tests 1, 2 and 3.
    e. Use **struct** and **typedef** to define a *Respondent* type, as well as a *Response* type. From the number of responses in the file, allocate an array of *Response* structs as dynamic memory to hold the survey responses. Free the array when it is no longer needed. Adapt the rest of your code to handle responses using structs. Test whether your changed code still passes tests 1, 2 and 3.
    f. Implement the filtering options expressed in the next three test cases and test each of them. Do this by analyzing the dynamic array of *Response* structs and marking the ones who stay and the ones

who are filtered out. You can use whichever strategy you prefer for filtering. One possibility is using a filtering array. Another one is having a filter variable member inside the *Response* struct.

g. Use the final test case to test combined filtering options.
h. Have you thought about remodularizing your work during the development? If not, now it would be a good time to separate parts of your code into different functions, in case you want an "A" grade.

2. Use the **`-std=c11`** , **`-Wall`** and **`-O0`** flags when compiling to ensure your code meets the standard used for the C language. Actually, using **`make`** and the given **`Makefile`** will assure those flags are used.
3. You must put different function groups into different C files according to their functionality. Keep the main function in **`dyn_survey.c`** for this assignment. Use the **`make`** tool and the given **`Makefile`** to manage dependencies in your system project.
4. Commit your code frequently (**`git add`** and **`git commit`**), so you do not lose your work. For instance, you may combine some of the exercises above or each partial solution to a separate test case into a commit, and describe it with a commit message.
5. When you are done with your commits, do not forget to **`git push`** them into your repo (you can also do this immediately after **`git add`** and **`git commit`**, if you prefer). Of course, our final grading of the code functionalities will not analyze any initial commits and pushes, just the final push.
6. Use the test files in **`/home/rbittencourt/seng265/a2/tests`** (i.e., on the lab-workstation filesystem) to guide your implementation effort. Start with simple cases. In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong".

What you must submit

- You must submit all the C source files (**`.c`**) and header files (**`.h`**) as requested by the given **`Makefile`** file within your **`git`** repository (and within its **`a2`** subdirectory) containing a solution to this assignment. Ensure your work is committed to your local repository and pushed to the remote before the due date/time. (You may keep extra files used during development within the repository, there is no problem doing that. But notice that the graders will only analyze your **`.c`** and **`.h`** files). Do not forget to send your final commit to the remote repo with **`git push`**.
- On Brightspace, you must send your Netlink ID and your final commit code (hash) until the due date.

Evaluation

Our grading scheme is out of 100 points. Assignment 2 grading rubric is split into 14 parts.

1) No use of global variables - 5 points - the code should not have global variables; all variables must be declared inside the main function or inside the other functions;
2) Appropriate use of **`struct`** types - 5 points - a **`Respondent`** must be stored in a struct variable; a **`Response`** must also be stored in a struct variable, with both a **`Respondent`** and the array of that particular respondent's answers as struct members; all struct types must be declared in **`typedef`** statements inside the **`dyn_survey.h`** file.
3) Appropriate use of dynamic memory - 5 points – all array variables must be declared as pointers and must be allocated in dynamic memory with the appropriate size derived from the information given in the input file. Memory must be freed when no longer used to avoid memory leaks.
4) Code modularization - 5 points - the code should have appropriate modularization, dividing the larger task into simpler tasks/functions (and subtasks, if needed).
5) File modularization - 5 points - the only function inside **`dyn_survey.c`** must be the main function; there must be at least one function in each of the **`input_handling.c`**, **`processing.c`** and **`output.c`** files, appropriate associated header must handle function prototypes according to the given **`Makefile`** file description; and the main function must call those functions in the required files (of course, there can be auxiliary functions called from the same file).
6) Documentation - 5 points - code comments (enough comments explaining the hardest parts such as loops, for instance, no need to comment each line), function comments (explain function purpose, parameters and return value if existent), adequate indentation;

7) Compilation standards - 5 points - your code compiles with no warnings when we run **`make`** from the configuration present in the given **`Makefile`**.
8) Version control - 5 points - Appropriate committing practices will be evaluated, i.e., your work cannot be pushed in just one single commit, evolution of your code must happen gradually (at least five commits are expected for this assignment);
9) Tests: Part 1 - 5 points - passing test 1 in **`tests`** directory: **`in01.txt`** as input and **`out01.txt`** as the expected output;
10) Tests: Part 2 - 5 points - passing test 2 in **`tests`** directory: **`in02.txt`** as input and **`out02.txt`** as the expected output;
11) Tests: Part 3 - 5 points - passing test 3 in **`tests`** directory: **`in03.txt`** as input and **`out03.txt`** as the expected output.
12) Tests: Part 4 - 15 points - passing test 4 in **`tests`** directory: **`in04.txt`** as input and **`out04.txt`** as the expected output;
13) Tests: Part 5 - 15 points - passing test 5 in **`tests`** directory: **`in05.txt`** as input and **`out05.txt`** as the expected output;
14) Tests: Part 6 - 15 points - passing test 6 in **`tests`** directory: **`in06.txt`** as input and **`out06.txt`** as the expected output.

We will only assess your final submission sent up to the due date (previous submissions will be ignored). On the other hand, late submissions after the due date will not be assessed.