Assignment 1 – CS Attitudes Survey

The Department of Computer Science (CSC) at the University of Victoria decided to better understand the problem of students' attitudes towards Computer Science through an opinion survey. In a pilot study, researchers from this department invited SENG 265 students to create a software system to allow them to conduct surveys with university students. To do so, they propose to reuse the questionnaire from an existing study, ***Examining Science and Engineering Students' Attitudes Towards Computer Science***[1].

Their idea is to do a pilot survey with adult students from some different fields so that, in the future, they can expand the poll to the rest of the university. Before answering the survey, Students are asked three demographic attributes: field of study (Engineering, Health, Arts), born in Canada (yes, no), and date of birth (*yyyy-mm-dd*).

In this questionnaire, there are 38 statements divided into five categories: Confidence (8 items), Interest (10 items), Gender (10 items), Usefulness (6 items) and Professional (4 items). Each item is answered in a Likert scale with six levels of agreement to the statements: fully disagree, disagree, partially disagree, partially agree, agree, and fully agree. From each answer, researchers generate five scores, one for each category, converting the answers to a scale from 1 to 6, and averaging each participant's answers in each category.

Researchers asked the software developers to generate some statistics as output:

1. The relative frequency (percentage) of each level of agreement for each question;
2. The five scores for all the interviewed students;
3. The five average scores per respondent.

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 1 by copying the files provided in **`/home/rbittencourt/seng265/a1`** into your **`a1`** directory inside the working copy of your local repository. Do not put your code files in subdirectories of **`a1`**, put them in **`a1`** 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 1, 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. **However, sharing of code solutions is strictly forbidden without the express written permission of the course instructor (Roberto).** 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. (Code-similarity analysis tools will be used to examine submitted work.) 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.

[1] Hoegh, A., & Moskal, B. M. (2009, October). Examining science and engineering students' attitudes toward computer science. In 2009 39th IEEE Frontiers in Education Conference (pp. 1-6). IEEE.

Description of the task

The ***Attitudes Towards Computer Science*** questionnaire has the following questions.

Category/Question	**Order**
Confidence construct (C):	
C1. I am comfortable with learning computing concepts.	Direct
C2. I have little self-confidence when it comes to computing courses.	Reverse
C3. I do not think that I can learn to understand computing concepts.	Reverse
C4. I can learn to understand computing concepts.	Direct
C6. I can achieve good grades (C or better) in computing courses.	Direct
C7. I am confident that I can solve problems by using computer applications.	Direct
C9. I am not comfortable with learning computing concepts.	Reverse
C10. I doubt that I can solve problems by using computer applications.	Reverse
Interest construct (I):	
I1. I would not take additional computer science courses if I were given the opportunity.	Reverse
I2. I think computer science is boring.	Reverse
I3. I hope that my future career will require the use of computer science concepts.	Direct
I4. The challenge of solving problems using computer science does not appeal to me.	Reverse
I5. I like to use computer science to solve problems.	Direct
I6. I do not like using computer science to solve problems.	Reverse
I7. The challenge of solving problems using computer science appeals to me.	Direct
I8. I hope that I can find a career that does not require the use of computer science concepts.	Reverse
I9. I think computer science is interesting.	Direct
I10. I would voluntarily take additional computer science courses if I were given the opportunity.	Direct
Gender construct (G):	
G1. I doubt that a woman could excel in computing courses.	Reverse
G2. Men are more capable than women at solving computing problems.	Reverse
G3. Computing is an appropriate subject for both men and women to study.	Direct
G8. Women and men can both excel in careers that involve computing.	Direct
G10. It is not appropriate for women to study computing.	Reverse
G11. Men produce higher quality work in computing than women.	Reverse
G12. Men are more likely to excel in careers that involve computing than women are.	Reverse
G13. Women produce the same quality work in computing as men.	Direct
G14. Men and women are equally capable of solving computing problems.	Direct
G15. Men and women can both excel in computing courses.	Direct
Usefulness construct (U):	
U1. Developing computing skills will not play a role in helping me achieve my career goals.	Reverse
U2. Knowledge of computing will allow me to secure a good job.	Direct
U4. My career goals do not require that I learn computing skills.	Reverse
U5. Developing computing skills will be important to my career goals.	Direct
U6. Knowledge of computing skills will not help me secure a good job.	Reverse
U8. I expect that learning to use computing skills will help me achieve my career goals.	Direct
Professional construct (P):	
P2. A student who performs well in computer science will probably not have a life outside of computers.	Reverse
P4. A student who performs well in computer science is likely to have a life outside of computers.	Direct
P6. Students who are skilled at computer science are less popular than other students.	Reverse
P8. Students who are skilled at computer science are just as popular as other students.	Direct

Each of the five scores is computed according to the phrasing of their component questions (direct or reverse). Direct questions convert the ordered Likert scale responses (fully disagree, disagree, partially disagree, partially agree, agree, and fully agree) to 1, 2, 3, 4, 5 and 6 respectively. Reverse questions convert them to 6, 5, 4, 3, 2, 1 respectively.

```
C = ( C1 + C2 + C3 + C4 + C6 + C7 + C9 + C10 ) / 8

I = ( I1 + I2 + I3 + I4 + I5 + I6 + I7 + I8 + I9 + I10 ) / 10

G = ( G1 + G2 + G3 + G8 + G10 + G11 + G12 + G13 + G14 + G15 ) / 10

U = ( U1 + U2 + U4 + U5 + U6 + U8 ) / 6

P = ( P2 + P4 + P6 + P8 ) / 4
```

Someone else was hired to produce a user interface for the survey. You just need to know that survey questions, respondents' demographic data and respondents' answers to the survey questions are recorded in a text file following a simple format. You should use this file to read all the input data to your program. In this text file, 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 also develop the system source code in C. Your program will be run from the Unix command line. Input is expected from **`stdin`**, and output is expected at **`stdout`**.

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.

You must NOT hardcode strings with the phrasing of the questions, possible answers (Likert items) and participants' responses. Instead, you should read that information from the `stdin`. Code that hardcodes strings with that information will receive grade zero.

Implementing your program

The C program you will develop in this assignment:

- Starts by reading the input from **`stdin`** with the questions and options, and stores them appropriately;
- Continues reading from **`stdin`** with the data from a sample of respondents.
- Computes the requested statistics according to the user's request expressed in the input file;
- finally, writes the requested results into **`stdout`**.

Assuming your current directory **`a1`** contains your **`survey.c`** source file as well as the compiled executable file (named **`survey`**), and a **`tests`** directory containing the assignment's test files is also in the **`a1`** directory, then the command to run your script will be.

```
% cat tests/in01.txt | ./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 | ./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/a1`** to guide your implementation effort. Inside **`tests`**, there are four input files and four 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. A script with four tests is available there for your final checks before submission.

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. When you are confident that your code is working correctly, 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. Write your program **`survey.c`** program in the **`a1`** directory within your git repository.
    a. Practice with **`stdin`** by reading it line by line with loops and **`fgets()`** and printing the output with **`printf()`** or **`fprintf()`**.
    b. Play with standard input redirection, by redirecting input to read from a file instead of reading from the keyboard.
    c. Learn how to tokenize a line (i.e., split a string line into a group of string tokens) stored in a string using the **`strtok()`** (you may wish to look at the example in the course slides or any good web tutorial that explains how to tokenize a sentence into words using the C language and the **`strtok()`** function).
    d. Learn how to loop over string tokens by printing them on the terminal screen.
    e. Learn how to store questions and answer options in an array of strings.
    f. Learn how to store answer options in an array of strings.
    g. Learn to use **`strcmp()`** or **`strncmp()`** to search for a particular string.
    h. Now search for any of the answer options stored in an array of strings, looping over it.
    i. Convert a respondent's answer using a string array of answer options and the type of question (direct or reverse). Use a function to encapsulate this search.
    j. Play with the standard output, using **`printf()`** to print on your terminal screen a respondent's answers converted into a numerical scale.
    k. Read all the students' answers and store the absolute frequencies of each question-answer pair appropriately (using bidimensional arrays may be a good idea).
    l. Compute the relative frequency of each answer (again, using bidimensional arrays may be a good way of doing this).
    m. Print the relative frequencies (percentages) of each level of agreement for each question.
    n. Format the relative frequencies appropriately to converge with the format in the tests.
    o. Play with standard output redirection, by redirecting **`stdout`** to save data in a file instead of sending them to the console.
    p. Test your program using pipes and the **`diff`** tool with the first test file as explained in the previous section.
    q. You may realize that just printing the words with **`printf()`** to your terminal screen may lead to some disorganized output (spaces missing or in excess). Those issues are related to the limitations of the **`strtok()`** function (particularly when a line ends). You may bring organization to your output by checking the separators you used for splitting the tokens (you may add **`\n`** as a separator together with the standard separator).
    r. Compute the scores for a particular respondent using the given formulas and print them to **`stdout`**. Check whether the formulas are computing each score correctly.
    s. Computer the average scores the total of respondents and print them to **`stdout`**.
    t. Play with standard output redirection, by redirecting **`stdout`** to save data in a file instead of sending them to the console.
    u. Test your program using pipes and the **`diff`** tool with the second test file as explained in the previous section.
    v. Have you thought about modularizing 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`** flag when compiling to ensure your code meets the ISO/IEC 9899:2011 standard for the C language.
3. Keep all of your code in one file for this assignment. In later assignments we will use the separable compilation features of C.
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/a1/tests`** (i.e., on the lab-workstation filesystem, you may also find them on Brightspace, under Assignment 1) 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 1 (one) single C source file named **`survey.c`**, within your **`git`** repository (and within its **`a1`** 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 **`survey.c`** file). 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 hash** until the due date.

Evaluation

Our grading scheme is relatively simple and is out of 100 points. Assignment 1 grading rubric is split into nine parts.

1) Modularization - 10 points - the code should have appropriate modularization, dividing the larger task into simpler tasks (and subtasks, if needed);
2) Documentation - 10 points - code comments (enough comments explaining the hardest parts (loops, for instance), no need to comment each line), function comments (explain function purpose, parameters and return value if existent), adequate indentation;
3) Version control - 10 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 three commits are expected for this assignment);
4) Tests: Part 1 - 20 points - passing test 1 in **`tests`** directory: **`in01.txt`** as input and **`out01.txt`** as the expected output;
5) Tests: Part 2 - 20 points - passing test 2 in **`tests`** directory: **`in02.txt`** as input and **`out02.txt`** as the expected output;
6) Tests: Part 3 - 20 points - passing test 3 in **`tests`** directory: **`in03.txt`** as input and **`out03.txt`** as the expected output.
7) Tests: Part 4 - 10 points - passing test 4 in **`tests`** directory: **`in04.txt`** as input and **`out04.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.