ICS 45C Spring 2019
Project #2: *Letter Never Sent*

Due date and time: *Monday, May 6, 11:59pm*

Introduction

Over my years of teaching, I've become quite fond of a grading scale that I jokingly refer to as the "Whatever I Want" scale. Of course, it's not quite as open-ended as my glib joke about it; it's a fair scale in the sense that any two students who perform at a particular level will receive the same grade — though it is neither a straight scale nor a curve, requiring me instead to decide, at the end of the quarter, where to set *cutpoints* that divide the students who receive one grade from the students who receive another. This is the grading scale that I'll employ at the end of the quarter in this course.

This project asks you to write a utility program that is given input via the standard input (i.e., **std::cin**) describing the set of graded artifacts for a course, the students enrolled in the course, and their scores on the various artifacts. Additionally, it is given a sequence of *cutpoint sets* describing the cutpoints (i.e., the number of total points necessary) to achieve various grade levels. The program's output is twofold: the *total score* received by each student in the course and, for each cutpoint set, final course grades for each student. The idea is for a tool like this to be used by a person to look at a few possible outcomes, to determine whether the outcomes are fair. The program makes no determination about that fairness, though; it provides the raw materials for a person to analyze.

Not surprisingly, you will need to write the program in C++, though there are some fairly heavy restrictions on what parts of C++ you're permitted to use, as we're still early in the process of learning it and I'd like you to gain some experience with some of the lower-level abstractions (most notably arrays, **new**, **delete**, and **delete[ ]**) before we layer less error-prone abstractions on top of them. Our goal in the early part of this course is to develop an understanding of what is happening behind the scenes in a C++ program, and it's impossible to develop this understanding if you sit at too high of a level of abstraction.

Getting started

As with the previous project, you'll need to complete some chores before you can start working on this project. *Be sure you read these instructions and follow them completely before proceeding.*

Refreshing your ICS 45C VM environment

Even if you previously downloaded your ICS 45C VM, you may need to refresh its environment before proceeding with this project, so that you have a copy of the **project2** project template that you'll need for this project.

Log into your VM and issue the command **ics45c version** to see what version of the ICS 45C environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version with a

timestamp older than the one listed below, you'll want to refresh your environment by running the command **ics45c refresh** to download the latest one before you proceed with this project.

```
2019-04-17 15:24:26
Project #2 template added
```

If you're unable to get outgoing network access to work on the ICS 45C VM — something that afflicts a handful of students each quarter — then the **ics45c refresh** command won't work, but an alternative approach is to download the latest environment from the link below, then to upload the file on to your ICS 45C VM using SCP. (See the Project #0 write-up for more details on using SCP.) Once the file is on your VM, you can run the command **ics45c refresh_local NAME_OF_ENVIRONMENT_FILE**, replacing **NAME_OF_ENVIRONMENT_FILE** with the name of the file you uploaded; note that you'd need to be in the same directory where the file is when you run the command.

- ics45c-2019spring-environment.tar.gz

Creating your project directory on your ICS 45C VM

A project template has been created specifically for this project. It includes a **gather** script for preparing your files for submission when you're finished, support for running the memory-checking tool Valgrind (Memcheck) to test the memory usage of your proram, as well as a sample input for your program (the same input that is specified in the project write-up), to prevent you from having to type it repeatedly.

Decide on a name for your project directory, then issue the command **ics45c start *PROJECT_NAME* project2** to create your new project directory using the **project2** template.

*Do not use the **basic** or **project1** templates for this project! You may find that they are missing some new tools that you'll be needing in order to complete your work!*

The input

The program's input is to be read from the standard input (i.e., **std::cin**). It is separated into four sections:

- A description of a course's *graded artifacts*, defining both a *number of points possible* and a *relative weight* for each.
- A description of the students in the course, listing a *student ID*, a *grade option*, and a *name* for each.
- A list of the raw scores received on each graded artifact by students in the course.
- A list of *cutpoint sets*, describing the total score required to receive an A, B, C, or D, respectively. (This project assumes that there are no +/- modifiers on grades, and that there are no other kinds of grades such as Incomplete, though we will support Pass/NotPass grades; see below.)

Each section is described in more detail below.

Graded artifacts

The section describing graded artifacts begins with a positive integer, alone on a line, specifying the number of graded artifacts. This is followed, on another line, by a sequence of positive integers separated by spaces, indicating the total points possible on each graded artifact. Finally, on one more line, there will be a sequence of positive integers separated by spaces, indicating the relative weight of each graded artifact. An example of this section would be:

```
7
15 15 15 15 15 50 50
12 12 12 12 12 15 25
```

This example describes seven graded artifacts, the first five having 15 points possible and relative weights of 12, the sixth having 50 points possible and a relative weight of 15, and the last having 50 points possible and a relative weight of 25. Note that, in this example, the relative weights add up to 100; in general, however, this will not always be the case.

Students

The next section of input describes the students enrolled in the course. It begins with a positive integer, alone on a line, specifying the number of students enrolled. Given that positive integer *n*, there will be *n* additional lines, each consisting of a non-negative integer *student ID*, followed by a space, followed by the student's *grade option* (G for "letter grade" or P for "Pass/NotPass"), followed by another space, followed by the student's name. Every character on the line after the grade option and space is considered to be part of the student's name. (It's important to note that the space following the grade option is *not* part of a student's name.) An example of this section would be:

```
5
123 G Alex Thornton
234 G Boo Thornton
345 G Jane Student
456 P Joe Student
567 G Too-Many Courses
```

Student IDs do not necessarily have to be three digits, and they do not necessarily all have to be the same number of digits.

Raw scores

The next section of input describes the raw scores earned by students on each graded artifact. The section begins with a positive integer, alone on a line, specifying the number of students for which raw scores are listed. Given that positive integer *n*, there will be *n* additional lines, each consisting of a sequence of non-negative integers separated by spaces, the first of which is a student ID, and the rest of which are raw scores for each graded artifact. If there are *m* graded artifacts, you can assume each of these lines will contain *m* + 1 integers (one student ID, followed by *m* raw scores), and that the scores correspond, in order, to the graded artifacts listed in the first section. Example:

```
5
345 14 14 14 14 14 45 45
123 13 10 8 14 12 50 37
456 12 9 15 13 11 38 26
234 15 15 15 15 15 50 50
567 8 4 0 10 0 24 12
```

It is possible for a student to be listed in the previous section but *not* to be listed in this section. In that case, assume that the student's raw scores are all 0. It is also possible for a student to be listed in this section who does not appear in the previous section; in that case, ignore the student's raw scores, as the student is assumed to have dropped the course.

It is also possible for a raw score to be higher than the number of points possible on a graded artifact. This is to be interpreted as extra credit, and fits into the formula below as-is.

Cutpoint sets

The last section of input is the *cutpoint sets*. This section begins with a positive integer, alone on a line, specifying the number of cutpoint sets. Given that positive integer *n*, the next *n* lines will consist of four non-negative numeric values (possibly including a decimal point and additional digits) that specify, respectively, the total points required for an A, B, C, or D in the course. Example:

```
3
90.0 80.0 70.0 60.0
85.0 75.0 65.0 55.0
80.0 70.0 60.0 50.0
```

Note that these are not percentages, necessarily; they indicate a total number of points necessary — this is described in more detail later in this write-up.

You may assume that each of the cutpoint values can safely be read into a variable of type **double**, and that the cutpoint values on each line will be non-ascending (e.g., the total points required for a higher grade like A will never be less than the total points required for a lower grade like B or C).

A complete example input

A complete example input for the program follows. It should be possible to copy and paste this into a shell window, or to save this into a file and use redirection to send the file's contents into your program as input. Additionally, this exact input file is provided in the **project2** project template, so you should find a copy of it in your project directory in a directory called **inputs**. (You might want to create other test inputs and place them in that directory, as well. You absolutely should be testing your program with inputs other than the provided one, as we will be when we grade it. However, we won't be providing additional test scenarios. It's up to developers of programs to develop their own tests; that's part of the job.)

```
7
15 15 15 15 15 50 50
12 12 12 12 12 15 25
5
123 G Alex Thornton
234 G Boo Thornton
345 G Jane Student
456 P Joe Student
567 G Too-Many Courses
5
345 14 14 14 14 14 45 45
123 13 10 8 14 12 50 37
456 12 9 15 13 11 38 26
234 15 15 15 15 15 50 50
567 8 4 0 10 0 24 12
3
90.0 80.0 70.0 60.0
85.0 75.0 65.0 55.0
80.0 70.0 60.0 50.0
```

Note that there is nothing that explicitly separates one section of the input from the next, though you will always be able to tell from context (e.g., the number of graded artifacts, the number of students, etc.) where one section ends and the next begins.

Other requirements about the input

The program must not print prompts (e.g., "Enter the number of students") or any other output *except* for what is specified in the section titled *The output* below.

It is reasonable for your program to assume that the program's input will always be structured as specified here. It is fine for your program to misbehave or even crash if given input that does conform to these specifications.

It is safe to assume that all integers will be small enough that they will fit into an **unsigned int** or **int** variable (using the ICS 45C VM and the tools we're using on it, the largest value that would fit in both of these kinds of variables is 2,147,483,647).

Calculating the total points

The program's output is largely determined by the *total score* earned by each student in the course. In order to complete the program, you'll need to understand the correct formula to use.

In the example input above, there are seven graded artifacts defined:

1. 15 points possible, with a relative weight of 12
2. 15 points possible, with a relative weight of 12
3. 15 points possible, with a relative weight of 12
4. 15 points possible, with a relative weight of 12
5. 15 points possible, with a relative weight of 12
6. 50 points possible, with a relative weight of 15
7. 50 points possible, with a relative weight of 25

From this example, we can see that simply adding together the raw scores on the various graded artifacts won't work, because, for example, artifact #6 is being graded on a 50-point scale, but is worth only slightly more, overall, than aritfact #5, which is being graded on a 15-point scale. Summing the raw scores would give too much weight to artifact #6. So we'll need to scale each of the raw scores, so that their relative weights are respected, then add the scaled scores together.

The total relative weight of all graded artifacts in this example is 100, which means that the total score for each student will range from 0.0 to 100.0. (It won't always be like this; we don't assume necessarily that all courses have a 100-point scale.) We calculate this by multiplying the *percentage of points earned* on each graded artifact by its relative weight, then summing the results. For example, suppose a student received these scores:

1. 14
2. 13
3. 15
4. 12
5. 10
6. 40

7. 45

The calculations would proceed as follows:

1. (14 / 15) * 12 = 11.2
2. (13 / 15) * 12 = 10.4
3. (15 / 15) * 12 = 12.0
4. (12 / 15) * 12 = 9.6
5. (10 / 15) * 12 = 8.0
6. (40 / 50) * 15 = 12.0
7. (45 / 50) * 25 = 22.5

Summing these together would yield a total of 11.2 + 10.4 + 12.0 + 9.6 + 8.0 + 12.0 + 22.5 = 85.7. So this student's *total score* is 85.7 out of a possible 100.

Scores that include extra credit (i.e., a raw score higher than the number of points possible on a graded artifact) do not need to be treated differently; they should be plugged into the formula the same way as any other.

Negative raw scores are not permitted in the input, as described above, so they don't factor into this formula at all.

Determining final course grades

The final course grade for a student is determined by comparing the *total score* earned by that student to the cutpoints for an A, B, C, or D.

- If a student's total score is greater than or equal to the A cutpoint, the student's final course grade is A
- Otherwise, if a student's total score is greater than or equal to the B cutpoint, the student's final course grade is B
- Otherwise, if a student's total score is greater than or equal to the C cutpoint, the student's final course grade is C
- Otherwise, if a student's total score is greater than or equal to the D cutpoint, the student's final course grade is D
- Otherwise, the student's final course grade is F

Note that students whose grade option is P (Pass/NotPass) are handled a bit differently. In their case, determine what their letter grade would have been (as described above) and then assign them one of these two grades:

- If the student's letter grade would have been A, B, or C, the student's final course grade is P (i.e., passing)
- If the student's letter grade would have been D or F, the student's final course grade is NP (i.e., not passing)

The output

While reading the input, there are specified points at which output will be generated and printed to the standard output (i.e., **std::cout**). These are specified, along with the format of that output, below.

Student roster

After reading the raw scores on all graded artifacts, but before reading the next section of the input, total scores are printed for all students. The format for this output is as follows:

- The words **TOTAL SCORES**, alone on a line
- For each student enrolled in the course, the student ID, followed by a space, followed by the student's name, followed by a space, followed by the student's total score

Example:

```
TOTAL SCORES
123 Alex Thornton 79.1
234 Boo Thornton 100
345 Jane Student 92
456 Joe Student 72.4
567 Too-Many Courses 30.8
```

It is not necessary to format the total score to a particular number of decimal places, though you should not truncate it or round it to an integer. Whatever way the C++ **iostream** library formats a **double** value by default is fine here.

Course grades

After reading *each* cutpoint set, but before reading the next part of the input, final course grades for that cutpoint set are printed. For the purposes of this output, cutpoint sets are numbered consecutively starting from 1. The format of this output is as follows:

- The words **CUTPOINT SET *n***, alone on a line, where *n* is replaced by the cutpoint set number (1 for the first cutpoint set, 2 for the second, and so on)
- For each student enrolled in the course, the student ID, followed by a space, followed by the student's name, followed by a space, followed by the student's grade

Example:

```
CUTPOINT SET 1
123 Alex Thornton C
234 Boo Thornton A
345 Jane Student A
456 Joe Student P
567 Too-Many Courses F
```

Output timing

Do not store all of the output in memory and print it to the standard output only at the end of the program. Instead, you will be required to write output to the standard output at the points in time specified above.

Output formatting

Your output should be formatted exactly as specified above, including spacing, capitalization, spelling, and so on. Note that we will be testing your programs in an automated fashion, so we will need these details to match the requirements precisely in order for our tests not to fail.

A complete example of the exact proper output for the provided sample input file will appear in your project directory in a directory called **outputs**, in which you'll find a file called **sample.out**.

One more note about output format: As in previous projects, I've indented the example inputs and outputs in this project write-up to set them apart from the text that surrounds them, but neither inputs nor outputs should be indented (e.g., the first line of the output should be **TOTAL SCORES**, alone on a line and *not* indented). The only way your output is allowed to differ from ours is in the number of digits after decimal places in students' total scores, which is fine, so long as all of your total scores are within 0.1% of what we expect in every case (which is more than enough of a buffer to account for inaccuracies introduced by the use of floating-point types like **float** or **double**).

A complete example of program execution

The following is a complete example of program execution from the Linux shell, demonstrating how input and output are interleaved. Input is shown in a regular font weight; output is shown in **bold**.

```
7
15 15 15 15 15 50 50
12 12 12 12 12 15 25
5
123 G Alex Thornton
234 G Boo Thornton
345 G Jane Student
456 P Joe Student
567 G Too-Many Courses
5
345 14 14 14 14 14 45 45
123 13 10 8 14 12 50 37
456 12 9 15 13 11 38 26
234 15 15 15 15 15 50 50
567 8 4 0 10 0 24 12
```
TOTAL SCORES
123 Alex Thornton 79.1
234 Boo Thornton 100
345 Jane Student 92
456 Joe Student 72.4
567 Too-Many Courses 30.8
```
3
90.0 80.0 70.0 60.0
```
CUTPOINT SET 1
123 Alex Thornton C
234 Boo Thornton A
345 Jane Student A
456 Joe Student P
567 Too-Many Courses F
```
85.0 75.0 65.0 55.0
```
CUTPOINT SET 2
123 Alex Thornton B
234 Boo Thornton A

```
345 Jane Student A
456 Joe Student P
567 Too-Many Courses F
80.0 70.0 60.0 50.0
CUTPOINT SET 3
123 Alex Thornton B
234 Boo Thornton A
345 Jane Student A
456 Joe Student P
567 Too-Many Courses F
```

Some rules and limitations

Because we're beginning our exploration of C++ by building our knowledge from some of its lowest-level abstractions, there are some fairly strict limitations on what features of the language you'll be permitted to use in this project. Those limitations will lift quickly as we move forward this quarter, so don't be concerned that you'll always be put into a tiny box, but this project has some very particular goals. If you have prior experience with C++, you may find that there are other techniques that you'd prefer to use — and it's quite possible that you're right that they'd be better techniques, in general — but you'll need to stick with what's allowed this time, because there are particular learning objectives here.

Here are the rules and limitations governing your work on this project.

- Other than **iostream**, **sstream**, and **string** — which you'll need for processing input and generating output — you are not permitted to include any C or C++ standard library headers in your program. Aside from standard input/output (using, for example, **std::cin**, **std::cout**, and **std::endl**), the **std::string** class, and stringstreams (**std::istringstream** and **std::ostringstream**), the standard library is off-limits (e.g., you cannot use collections like **std::vector**).
- Use arrays to store and manipulate data within your program. Note, too, that you won't know the correct sizes for these arrays at compile time (because the sizes are specified in the program's input), so you will find that you need to use dynamically-allocated arrays extensively.
- You are permitted to use **struct**s (in the C sense, i.e., implicitly public member variables with *no member functions, constructors, destructors, etc.*), but **class**es are off-limits until the next project, as our exploration of object-oriented techniques still lies ahead of us.
- Even though this is what you might call a "batch mode" program (i.e., it reads input, processes it, generates output, and then ends), you are still required to explicitly delete any memory that you dynamically allocate using **new** before the program ends. This is a habit that is vital to build in C++, so this will always be a requirement in this course. (While the symptoms of leaking memory are often invisible, we'll use tools in this course to make them visible; more about that below.)

From a design perspective, you are again required to break large functions into smaller ones (where each has a single, straightforward responsibility), and to arrange your functions into multiple source files (where each source file encompasses a set of strongly related functions). Any source file that needs to "export" things (e.g., functions) to other source files should also include its own separate header file.

Redirection (or, How to avoid re-typing the input every time you run your program)

When you use the **run** script to execute your program — and, generally, by default in Linux — the standard input and standard output are connected to the shell window where you executed the program. The program's input is typed into that shell window; the program's output is written to that shell window.

However, you can also use a technique called *redirection* to connect the standard input and/or standard output to other devices — most importantly for us, to files! — instead of the default. For example, the contents of an existing file may be redirected into the standard input, so that your program will read its input from the file instead of from the shell window; similarly, the standard output can be redirected into a file, meaning that all of the output printed to **std::cout** will be saved into that file, rather than being displayed in the shell window.

The typical mechanism for redirection is to use the < and > operators on the command line, like this:

```
SomeProgramYouWantToRun <FileContainingStdInput.txt >FileToStoreStdOutput.txt
```

You don't have to use both; you can use only < (in which case the standard input is read from a file, but the standard output is written to the shell window) or only > (in which case the standard input is read from the shell window, but the standard output is written to a file). It's up to you.

Using redirection to avoid re-typing test input

This can be a handy technique to use in your testing, to avoid the problem of having to re-type the program's input every time you run it. If you used the **project2** project template — and you should have! — when creating your project, you should see a directory called **inputs** in your project directory; in the **inputs** directory, you should see a file **sample.in**, which contains the sample input from this project write-up. If you want to run your program using the same input, issue this command (from your project directory):

```
./run <inputs/sample.in
```

The effect is that the contents of the **sample.in** file in the **inputs** directory will be redirected into your program's standard input, in lieu of you having to type it. You might also want to write other test files for yourself in that same **inputs** directory, then use a similar technique to redirect them into the standard input of your program. You can avoid a lot of re-typing this way!

A note about output timing

Note that the redirection technique will not give you an accurate view of whether your program is printing the right output at the right times, as specified in the requirements above, but it will tell you whether your program is generating the correct answer. You can enter the input manually sometimes when you want to test the timing of the output.

Using Valgrind (Memcheck) to check your memory usage

The ICS 45C VM includes a program called Valgrind, which is used for detecting a variety of difficult-to-find problems in C and C++ programs. It consists of a set of *tools*, each of which detects a certain kind of problem. For our work here, we'll be interested in a tool in Valgrind called Memcheck, which watches a program while it runs and carefully tracks the way it uses memory, looking for a variety of mistakes that might otherwise be silent bugs and makes them visible.

A detailed look at Memcheck can be found in the Illuminating the Dark Corners notes; I suggest that you take

a look at those and use Memcheck to work through issues in this project (and subsequent ones) while you work.

Running your program using Memcheck

The usual **run** script, which you use to run your program after building it, has an option that lets you run the program using Memcheck. If you run your program with this command:

```
./run --memcheck
```

then Memcheck will observe your program as it runs and report its findings. You'd type in the input as usual and would see the output appear as usual, but Memcheck's preamble will be displayed when your program starts, its final report will be displayed when your program ends, and you may see additional warnings while the program runs.

Note, too, that you can combine this with redirection:

```
./run --memcheck <inputs/sample.in
```

if you want to run the program with Memcheck but avoid typing the input by hand.

We will be running your program using Memcheck as part of the grading process, and expect it to generate a clean report (i.e., no memory leaks and no memory-related issues while the program runs, other than the things that we intentionally suppressed) to receive full credit, so you'd be well-served to use Memcheck periodically as you work, and to fix issues as they arise.

Deliverables

After using the **gather** script in your project directory to gather up your C++ source and header files into a single **project2.tar.gz** file, submit that file (and only that file!) to Checkmate.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally. (It's not a bad idea to look at the contents of your tarball on your host operating system before submitting it.)

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course which is described in the section titled *Late work* at this link.

Clarifications about Memcheck and supppression of reporting added by Alex Thornton, Fall 2016.
Additional clarifications and requirement tweaks by Alex Thornton, Fall 2015.
Some tweaks to requirements and some additional clarifications by Alex Thornton, Fall 2014.
A few clarifications added by Alex Thornton, Winter 2014.
Requirements tweaked and reorganized for ICS 45C VM by Alex Thornton, Fall 2013.
Originally written by Alex Thornton, Fall 2012.