**Homework 1**

* Assigned: August 30, 2019 PST
* Due: September 13, 2019 at 23:59 PST
* Directory name in your github repository for this homework (case sensitive): hw1
  + Once you have cloned your https://github.com/usc-csci104-fall2019/hw-username repo, create this hw1 folder underneath it (i.e. hw-username/hw1).
  + If your hw-username repo has not been created yet, please do your work in a separate folder and you can copy over relevant files before submitting.

**A Few Notes on Repositories**

1. Never clone a repository into another. If you have the course material in a folder and you clone your personal repo hw\_username inside it, the outside repository will start tracking the new files and it will be a mess. Whenever you want to clone another repository, you need to do it in the folder above, adjacent to your original repository.
2. If your homework repositories are not available, make sure you've followed the instructions for creating your account. That set of instructions is available at the end of [lab zero](http://bytes.usc.edu/http:/bytes.usc.edu/cs104/labs/lab00.html).

**Skeleton Code**

On many occasions we will want to distribute skeleton code, tests, and other pertinent files. To do this, we have made a separate repository, homework\_resources, under our class GitHub site. You should clone this repository to your laptop and do a git pull regularly to check for updates (yes, even we sometimes make mistakes; when we do, we will fix them as quickly as possible, but you'll only get the fixes when you pull).

git clone git@github.com:usc-csci104-fall2019/homework\_resources.git

Here is a clickable link: <https://github.com/usc-csci104-fall2019/homework_resources>. Again, be sure you don't clone this repository into your hw-username repo, but at some higher-up point like in a csci104 folder on your computer.

**1. Course Policies (10%)**

Carefully study the information on the course site and answer the following questions about course policies. Anywhere from no to all answers may be correct. **Place your answers to this question in a file named** ANSWERS.md **in your homework repository**.

**Part (a)**

Which of the following are acceptable behaviors in solving homework problems and doing projects?

1. Looking up information relevant to the course online.
2. Looking up or asking for sample solutions online.
3. Copying code from my classmates, and then editing it significantly.
4. Asking the course staff for help.
5. Sitting next to my classmate and coding together as a team or with significant conversation about our approach.
6. Sharing my code with a classmate, if he/she promises not to copy it, but to just read over it and learn from it.

**Part (b)**

To dispute a grade on an assigment you should:

1. Email your professor immediately.
2. Fill out the homework regrade form.
3. Post on Piazza if you do not receive any response to your regrade request.
4. Raise an issue within 1 week of receiving the grade.

**Part (c)**

Which of the following are recommended ways of writing code?

1. gedit
2. emacs
3. Eclipse
4. sublime
5. Microsoft Visual Studio
6. notepad

**Part (d)**

What is the late submission policy?

1. Each assignment can be submitted late for 50% credit.
2. Each student has 4 late days of which only two can be used per HW.
3. Students need to get approval before submitting an assignment late.
4. One hour late submission is acceptable for each assignment.
5. There is no late policy.

**Part (e)**

After making a late submission by pushing your code to Github you should:

1. Do nothing. Sit back and enjoy.
2. Complete the online submission form as you would for an on-time submission.
3. Start the next assignment sooner.

**2. Git Quiz (10%)**

**Put your answers to this question in the file named** ANSWERS.md.

**Part (a)**

Which of the following git user interfaces are accepted and supported in this course?

1. Git Bash (Windows)
2. GitHub Desktop Client
3. Terminal (Mac or Linux)
4. Eclipse eGit
5. Tower Git Client

**Part (b)**

When cloning a Git repo, which of the following should you avoid?

1. Cloning into a folder that itself is a git repo.
2. Cloning into a sync'ed folder like Dropbox or Google Drive.
3. Cloning into the Desktop folder of your VM.

**Part (c)**

Provide the appropriate git command to perform the following operations:

1. Stage an untracked file to be committed. The (hypothetical) file is called 'hw1q2b.cpp'.
2. Display the details of the last three commits in the repository.

**Part (d)**

Let's say you staged three files to be committed. Then, you ran git commit. What will git do?

**Part (e)**

What is the full command to re-clone your private CSCI104 repo to your VM? Assume you are in an appropriate folder.

**3. More Git Questions (10%)**

In this problem, we will be working with a fictional sample repository, which we pretend is located at <https://github.com/usc-csci104-fall2019/SampleRepo>. There is no actual repo there, so don't be surprised if you cannot actually clone it or otherwise interact with it. This exercise is designed to measure your understanding of the [file status lifecycle](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository) in git. Please frame your answers in the context of the following lifecycle based on your interaction with the repository as specified below:

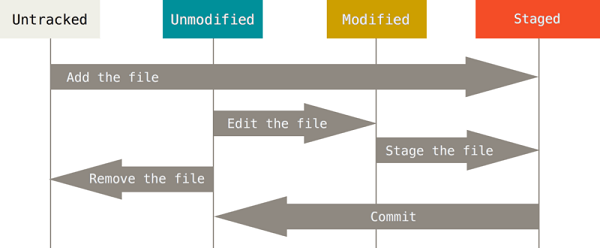


Figure courtesy of the [Pro Git](http://git-scm.com/book) book by Scott Chacon

Parts (a) through (f) should be done in sequence. In other words, when you get to part (f), you should assume that you already executed the earlier commands (a) through (e). You **must** use the terminology specified in the lifecycle shown above, for example the output of git status is not accepted as a valid answer. For the purposes of this question, you can assume you have full access (i.e., read/write) to the repository. **Put your answers to this question in the file named** ANSWERS.md.

**Part (a)**

What is the status of README.md after performing the following operations:

*# Change directory to the home directory*

cd

*# Clone the SampleRepo repository*

git clone git@github.com:usc-csci104-fall2019/SampleRepo.git

*# Change directory into the local copy of SampleRepo*

cd SampleRepo

**Part (b)**

What is the status of README.md and fun\_problem.txt after performing the following operations:

*# Create a new empty file named fun\_problem.txt*

touch fun\_problem.txt

*# List files*

ls

*# Append a line to the end of README.md*

echo "Markdown is easy" >> README.md

**Part (c)**

What is the status of README.md and fun\_problem.txt after performing the following operation:

git add README.md fun\_problem.txt

**Part (d)**

What is the status of README.md and fun\_problem.txt after performing the following operations:

git commit -m "My opinion on markdown"

echo "Markdown is too easy" >> README.md

echo "So far, so good" >> fun\_problem.txt

**Part (e)**

What is the status of README.md and fun\_problem.txt after performing the following operations:

git add README.md

git checkout -- fun\_problem.txt

Also, what are the contents of fun\_problem.txt? Why?

**Part (f)**

What is the status of README.md after performing the following operation:

echo "Fancy git move" >> README.md

Explain why this status was reached.

**Valgrind**

You will lose points if you have memory leaks, so be sure to run valgrind once you think your code is working.

You can run valgrind as follows:

$ valgrind --tool=memcheck --leak-check=yes ./sum 4098

If you have no errors, your leak summary will look like this, though note that the number of bytes and blocks in use at exit might differ:

==3184== HEAP SUMMARY:

==3184== in use at exit: 72,704 bytes in 1 blocks

==3184== total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated

==3184==

==3184== LEAK SUMMARY:

==3184== definitely lost: 0 bytes in 0 blocks

==3184== indirectly lost: 0 bytes in 0 blocks

==3184== possibly lost: 0 bytes in 0 blocks

==3184== still reachable: 0 bytes in 0 blocks

==3184== suppressed: 72,704 bytes in 1 blocks

These 72,704 bytes come from a known error in the g++ and valgrind compatibility, and you should not worry about them.

**Command Line Arguments**

**Hint**: In order to read parameters as command line arguments in C++, you need to use a slightly different syntax for your main function: int main (int argc, char\* argv[]). When your program is called at the command line, argc will then contain the total number of arguments that the program was given, and argv will be an array of strings, the parameters the program was passed. The char\* at argv[0] is always the name of your program, then argv[1] is first argument. The operating system will assign the values of argc and argv, and you can just access them inside your program.

**4. File Reverser (15%)**

Write a program that reads in a text file and outputs the characters of that file in reverse order to the terminal. To achieve this, you will need to use dynamic memory allocation. The program should receive the filename as a command line parameter. An empty file\_reverse.cpp file is provided in the file\_reverse directory of the skeleton code for you to work in.

For the text file, the first line will consist only of an integer representing the number of characters in the file; let's call it n. This will be followed by n characters on the second line. The following is a sample input and output of the program.

Input text file example.txt:

44

The quick brown fox jumps over the lazy dog.

Compiling the program on the command line:

g++ -g file\_reverse.cpp -o file\_reverse

Running the program and the output:

$ ./file\_reverse example.txt

.god yzal eht revo spmuj xof nworb kciuq ehT

Another input text file example:

27

Able was I, ere I saw Elba.

The output:

.ablE was I ere ,I saw elbA

**5. The Rani (55%)**

**Please put all solution code in the file named** the\_rani.cpp. **In the future you will be able to separate your code into separate files, since you will be required to submit a makefile.**

The Rani is planning to run her latest round of experiments to build genetically modified mutant creatures. In order to do so, she will start with a population of experimental subjects taken from Miasimia Goria, and subject different sets of them to different experiments. She needs to keep track of which subjects are in which experiment, and which experiments they have been subjected to in the past, so that when she finally finds a suitable mutant, she can remember what experiments she performed to get it.

You will write a program that helps her with keeping track of her experiments - while she is a world-class mad scientist, her coding skills were not enough to get her into USC Computer Science. You will maintain a 2-d array of strings subject\_history[][], where subject\_history[i] stores all of the subjects currently involved in experiment i, and subject\_history[i][j] contains a string listing all of the experiments that this particular subject has been a part of.

You will be given her experimental logbook, which contains a sequence of the following operations, one per line. Errors and how to handle them are discussed below.

* START n: starts a new set of experiments, with n subjects involved. These n subjects (numbered 0 to n - 1) will be added to experiment 0. We will call this "experiment 0", but it's really just her subject pool; see below.
* ADD: adds a new experiment, giving it the next higher number. If she has 3 experiments going, they are numbered 1-3 (remember that 0 is the subject pool), so ADD would add experiment 4 next.
* MOVE x y n m: moves the subjects numbered n to m (inclusive) from experiment x to experiment y. What this means is that the subjects with these numbers are removed from experiment x, and all higher-numbered subjects have their numbers shifted down, so they now start being numbered at x. In other words, you always keep subjects' numbering consecutive. When they are added to experiment y, they are added at the next highest numbers. So if she adds 5 more subjects to an experiment having 7 subjects already, they are numbered 7 to 11, since the previous 7 were numbered 0 to 6. It is perfectly fine to move subjects from an experiment to the same experiment. Notice that if you do so, the internal numbering of the subjects in the experiment will likely change. Also, the experiment should be listed multiple times (as often as the subject was in it).
* QUERY x n: returns the sequence of experiments that the current nth subject in experiment x was exposed to, in order. If the subject was part of experiments 0 (subject pool), 1, 6, and is currently number 5 in experiment 2, then QUERY 2 5 would output 1 6 2. Don't output the subject pool, even if the subject is later returned to it.

Your program will take two strings as input: the first specifies an input file, and the second specifies an output file. The input file could contain many different slates of experiments. Any time you encounter the line START n for some parameter n, the previous experiment is completely discarded, and a new slate of experiments is started from scratch. Once the last line in the input file has been processed, the program should correctly deallocate all memory and terminate.

We are providing you with some skeleton code to get you started in the homework\_resources repo (see the top of this assignment for information on that repo). In your solution, you **cannot** use vector, deque, or list data types from the STL. Instead you will be dynamically allocating and deallocating all of your memory via arrays of strings. In addition, you should be using good programming style by breaking your code up into smaller class functions (consider carefully what functions should be private and public). Outside of these requirements, you are free to make any changes you like to the skeleton code; it is merely given to you as a suggestion. The most important part of this problem is to make sure that your solution does not leak memory. You will lose significant points (more than half) for serious memory leaks.

Your program should be robust to many types of errors, but there are some that would be very cumbersome for you to handle. For that reason, we provide you here with a list that hopefully covers all/most of the errors that could come up. If you have questions about errors not covered here, post (publicly) on Piazza. If multiple errors from the list below apply, only output the first message. For instance, if a command has a wrong number of parameters, and is given a string instead of an integer, just output that it has the wrong number.

**Handling Runtime Errors**

Whenever you diagnose an error in a line, you should write Error on line <number>: <message> to the output file where <number> is the line number of the incorrect command you found followed by the error-specific <message> on the next line. Once an errors is caught and the corresponding message is written to output, the program should **continue**. Below, we will list which types of errors in lines you need to catch, and which ones we promise will not happen.

* If the first line is not a START command, you should write an error with message "no subjects yet".
* We might give you command names that do not exist, such as OUTPUT. These non-existent commands could have any number of parameters. In that case, write an error with message "command does not exist".
* Commands might have too few parameters. For instance, we might give you QUERY 3 or MOVE 1 4. In all of these cases, your error message should be "too few arguments".
* Some parameters which are supposed to be integers might be garbage (strings) or doubles. In that case, output an error message of "expected integer argument". See below for hints on how to handle this.
* Some numbers may be out of the reasonable range. For instance, a number may be negative. Or you are asked to move subjects from an experiment that does not exist, or to an experiment that does not exist. Or the range of subjects you are supposed to move (or the subject you are queried about) is outside of the number of subjects for the experiment. For instance, you might be given MOVE 3 5 7 10, but experiment 3 only has 9 subjects (so there are no subjects 9 and 10). In that case, you should output an error with message "argument out of range".
* For the MOVE command, if the first number n is larger than the second number m, write an error with message "invalid range of subjects to move".

When a command contains multiple errors (i.e. QUERY -1 5), you only need to write out the first error. The order that errors should be check is the order they are listed above. Error messages should be printed without the quote around them: when we say message "no subjects yet", your output file should contain Error on line <number>: no subjects yet).

**Errors You Don't Need to Handle**

* If commands have too many parameters, such as QUERY 3 5 2 or ADD hello-world, then ignore the extra parameters: interpret these as QUERY 3 5 and ADD.
* We promise that any integer we do give you will always fit into the int data type. They will not be larger than 2147483648.
* We also promise you that the number of experimental subjects will be small enough that this number of subjects and all their history will fit comfortably into main memory on any normal computer.
* As mentioned above, moving a subject from an experiment to the same experiment is not an error.

**Hints on Handling Parameters of Incorrect Types**

C++ provides several useful functions for turning strings into numbers (integers or doubles). Check out C++ documentation to learn about them and save yourself a lot of work. One method would put the string into a stringstream and extract numbers from it. Another would just use conversion functions explicitly. Conveniently, these methods also let you know when they failed, which indicates that the input was not a number. Unfortunately, even functions that are supposed to extract an integer will not fail when the string is actually a double. When you try to extract an integer from the string "3.14159", you will simply get 3.

To deal with this, we offer three potential approaches (there may be more). The first is to extract a double, then check if the double has zero for its fractional value (there is a function to get the fractional part of a double). If so, convert it to an integer; otherwise, treat it as an error. Another is to think about which symbols are not supposed to occur in an integer number. Then check out functions like find\_first\_not\_of() and some of its relatives to see how you might easily detect such incorrect symbols. A final option might be to peek into the stringstream you're working with to check whether the next character is allowed after reading an integer.

**A Note on Exceptions**

You are encouraged (but not required) to familiarize yourself with exceptions, and use them in your error-checking process. There is a catch clause supplied in TheRani::main which can be modified to output correctly formatted errors. If you use exceptions, other exception classes you might use from <stdexcept> include invalid\_argument and runtime\_error. While it's good practice to use a semantically appropriate error type to differentiate in the catch block, for the purpose of this exercise any exception may be thrown as long as the message is correct. This is because the existing try block has a blanket catch(exception& e), which will catch any of these exceptions and print just the message.

For some very basic testing of your code (obviously, our real test cases will be much more difficult), we are providing you with an easy input and the corresponding output. The files are linked below, and are also in the homework resources repo. We provide you with two versions of the input: one that will work for your program, and another that has some comments added so that you can see what the input means: