Project 1: recproc

Project 1: specification

Released:	8/23
Due:	9/7

GitHub Invitation Link (https://classroom.github.com/a/-qdMtgqw)

Accept the GitHub invitation, wait until you get an email saying the import is complete, and then clone the git repository to your local machine.

The goal of this assignment is to refresh your programming skills and to help you recall material from ECE 2514/3514. You can also use this assignment to gauge how well prepared you are for the rest of 3574. If you cannot complete this assignment, you may consider drop.

Description

Write a program called **recproc** that can process records in a simple text-based database. The database is maintained in a Linux text file. Each *record* is stored as one line of text, and each record consists of *fields* that are separated by delimiter characters. For this assignment, use the colon character (':') as the delimiter. The following is an example database consisting of 3 records, and each record consists of 6 fields:

```
Jones:John P.:123 Pheasant Lane:Fredericksburg:VA:22401
Henry:Patrick:456 Sparrow St.:Richmond:VA:23221
Adams:Abigail:789 Robin Rd.:Boston:MA:02134
```

Your program will be invoked from the command line in your Virtual Machine. During typical usage, your program will read or modify one record. Assume that the records are implicitly numbered beginning at 0 in the file. Within each record, also assume that fields are numbered left-to-right beginning at 0. For example, assuming the database shown above, you could print record 0 using the following command:

```
% recproc -r 0
Jones:John P.:123 Pheasant Lane:Fredericksburg:VA:22401
```

Similarly, with the following command you could print field 2 from record 1:

```
% recproc -r 1 -f 2
456 Sparrow St.
```

For the case of an invalid record number or field number, your program should send a diagnostic message to std::cerr similar to the following.

```
% recproc -r 1 -f 6
Error: invalid field (exceed)
% recproc -r 3
Error: invalid record number <3> to file named <database.txt>
```

Your program should be able to add a new record to the end of the file using -a, as shown below. Your program should be able to delete a record using –d followed by a record number.

```
% recproc -a "AAA:BBB:CCC:DDD:EEE:FFF"
Info: append <AAA:BBB:CCC:DDD:EEE:FFF> to file named <database.txt>
% recproc -d 1
Info: delete record number <1> to file named <database.txt>
```

After these two operations, the original database file will have been changed to have following contents:

```
Jones:John P.:123 Pheasant Lane:Fredericksburg:VA:22401
Adams:Abigail:789 Robin Rd.:Boston:MA:02134
AAA:BBB:CCC:DDD:EEE:FFF
```

Your program should also be able to modify individual fields by supplying –m along with a new value that replaces the previous value. Here are some examples, for the database shown in the previous paragraph:

```
% recproc -r 1 -f 2 -m "789 Peacock Road"
Info: modify record number <1> of field index <2> with <789 Peacock Road> to <database.txt>
% recproc -r 0 -f 4 -m MD
Info: modify record number <0> of field index <4> with <MD> to <database.txt>
```

After these two operations, the database file will have been changed to have following contents:

```
Jones:John P.:123 Pheasant Lane:Fredericksburg:MD:22401
Adams:Abigail:789 Peacock Road:Boston:MA:02134
AAA:BBB:CCC:DDD:EEE:FFF
```

To summarize, your program should accept the following combinations of command-line arguments (CLA):

% recproc [-i filename] -a string	(append a record)
% recproc [-i filename] -d int	(delete a record)
% recproc [-i filename] -r int	(print an entire record)

% recproc [-i filename] -r int -f int (print a field)

% recproc [-i filename] -r int -f int -m string (modify an existing field)

In this summary, int refers to an integer value that is used as an index, and string refers either to a new field value or a new record value.

The brackets “[ ]” indicate CLAs that are optional. If -i is present, then your program should use the specified filename as the database file. If -i is not present, then your program should use a default file that is named database.txt. If the database file does not exist when the –a argument is present, then your program should create the file and place one record in it. If the database file does not exist when –d or –r is present, then your program should print a helpful error message and return EXIT_FAILURE.

More implementation details

In normal operation, your program should return EXIT_SUCCESS. In case of error, such as a file cannot be opened or an invalid index is specified, then your program should take no action except to print a helpful error message and return EXIT_FAILURE. (You are not required to throw exceptions in this assignment.)

Your program should not place upper limits on the number of records or fields.

Your program should assume that every line of text in the database file represents one record. (Notice that blank lines are not allowed.)

You may assume that the delimiter character (':') will never be present within a field.

For a given database file, all records must have the same number of fields. Empty fields are permitted. You may assume that the minimum number of fields per record is 2. (I.e., at least 1 delimiter character will be present on each line of text.)

If your program needs to know the number of records in the file, or the number of fields per record, then your program must determine these values by examining the database file.

The starter code will contain an example database.txt file, which is in Unix-style text format. Each line of text ends with a line-feed (LF) character only. No special characters, other than the final LF, indicate the end of the file. (For comparison, Windows-format text files provide an additional carriage-return (CR) character at the end of each line.) The autograder will test your code using Unix-style files only.

Building and testing the starter code in the reference environment

Prerequisites:

- You have completed `Exercise 01: Setup` and have a working installation of VirtualBox and Vagrant
- You have read through the **Vagrant Getting Started (https://www.vagrantup.com/docs/getting-started/)** guide.

Steps:

1. Create a working directory for your project somewhere on your computer. Open your command line terminal and make a working directory. Then change to that directory.
2. Clone the recproc project after accepting the GitHub invitation **at this link (https://classroom.github.com/a/-qdMtgqw)**

   ```
   % git clone https://github.com/VTECE/ece3574-f123-proj1-USER.git
   ```

   where **USER** is your GitHub username. You may have to enter your GitHub username and password.
3. Change to this repository directory. You should see several files, including recproc.cpp. This is the only file that you should need to modify for this assignment. Run vagrant to setup the virtual machine.

   ```
   % vagrant up
   ```

   This will take a several minutes to complete.
4. After step 3 completes. Halt the VM using:

   ```
   % vagrant halt
   ```

   This is how you stop the reference environment, but leave it ready to start up again.
5. Restart the VM using step 3 again. You should now see a graphical window.
6. Open a command line in the VM window by right clicking and selecting Terminal emulator. If successful, you are now using a command-line interface inside the Ubuntu Linux virtual machine.
7. From inside the VM, see what directory you are in

   ```
   % pwd
   ```

   You should be in /home/vagrant. This is the home directory for the default user setup by Vagrant. Feel free to just remove the default directories created using `rm -r *`.
8. List the files in the host operating system shared with the Virtual Machine.

```
% ls /vagrant
```

You should see the files that are present from the cloned repository on your host machine. Typically, this is where your source code will be.

9. Still in the VM, build the starter code using cmake.

```
% cmake /vagrant
```

Then

```
% cmake --build .
```

or just

```
% make
```

Now you should be able to see the executable file recproc in /vagrant.

> To configure and run the build in strict mode (increased warnings, warnings become errors)
>
> ```
> % cmake -DSTRICT=True /vagrant
> % make clean; make
> ```

10. Run the recproc executable

```
% ./recproc
```

11. When finished with this session, log out of VM.

```
% logout
```

12. Now back on your host command line, you can halt the VM.

```
% vagrant halt
```

13. When you no longer need the VM or want to recreate it from scratch, you can destroy it.

```
% vagrant destroy
```

The systems prompts you for confirmation.

14. Now, use git to commit the source file you changed to the local repository.

```
 % git add reproc.cpp
 % git commit -m "Implement an argument parser"
```

15. Finally, use git push to synchronize the repository with your remote repository on GitHub

```
% git push
```

You may have to enter your GitHub username and password.

Submission

To submit your milestone:

1. Tag the git commit that you wish to be considered for grading as "final".

   ```
   % git tag final
   ```

2. Push this change to GitHub

   ```
   % git push origin final
   ```

3. Do a final test of this code version at the Inginious autograder.

   (Autograder details are not provided here)

If your “final” version needs to be improved, simply repeat the above 3 steps with “final2”, “final3”, etc., as the new tags.

Notice that you can also double-check your submission by re-cloning your repository into a new directory on your local machine. Then you can verify that all of the files that you submitted are what you intended. Failure to complete these steps by the due date will result in a failed submission.

Grading

- 5 points: Code compiles in the reference environment
- 35 points: Code correctness based on the instructor’s tests (proportional)
- 5 points: No memory leaks (checked using valgrind in the reference environment)
- 5 points: Good coding style
- 5 points: Good development practices

  Total: 55 points

Grading Notes:

- If your code does not compile/build in the reference environment, then the grade for this assignment will be 0 points.
- Correctness is determined by the proportion of instructor tests that pass (5 points each).

- Code quality means your code compiles with no warnings at a high-warning level (STRICT in the reference environment).
- Good development practices is assessed by looking for regular, incremental, and well-packaged commits. For grading purpose, the (human) grader will look for at least 5 meaningful commits. If a function has more than 50 lines, split into two functions.

Last update on Sept. 3, 2023