Project 0

This project teaches some of the technologies and skills that will be essential to your success in this course. To help you focus on the relevant aspects of these technologies and understand how you will apply these technologies to later projects, this project teaches these technologies through a tutorial that mimics the recommended procedure for working on projects in this course.

Read the explanations and follow the instructions carefully. You will be asked to take screenshots throughout the process and submit your final project when you are done. If you already know how to use a particular technology, still complete all the steps of the tutorial, but you can just skim the explanations you already understand.

Project 0 **1**

- Create a Git Repository on GitHub 3
  - Explanation 3
  - Instructions 3
- Clone your Repository on your Local Machine 4
  - Explanation 4
  - Instructions 5
- Add Files to your Repository 6
  - Explanation 6
  - Instructions 6
- View your Repository in CLion 7
  - Explanation 7
  - Instructions 8
- Debug your Code 9
  - Explanation 9
    - Breakpoints 10
    - Inspecting Variables 10
    - Step Over/Step In/Step Out/Continue 11
    - Stack Trace 11
    - General Debugging Tips 11
  - Instructions 11
- Running the Makefile 12
  - Explanation 12
  - Instructions 12
- Finishing the Project 12

Explanation 12
Instructions 13

Create a Git Repository on GitHub

Explanation

You may have heard about GitHub before. GitHub is a cloud-based platform that allows developers to share code and other files. The files are stored in containers called repositories. GitHub uses Git as the foundation for much of its functionality; however, GitHub and Git are two separate but related technologies.

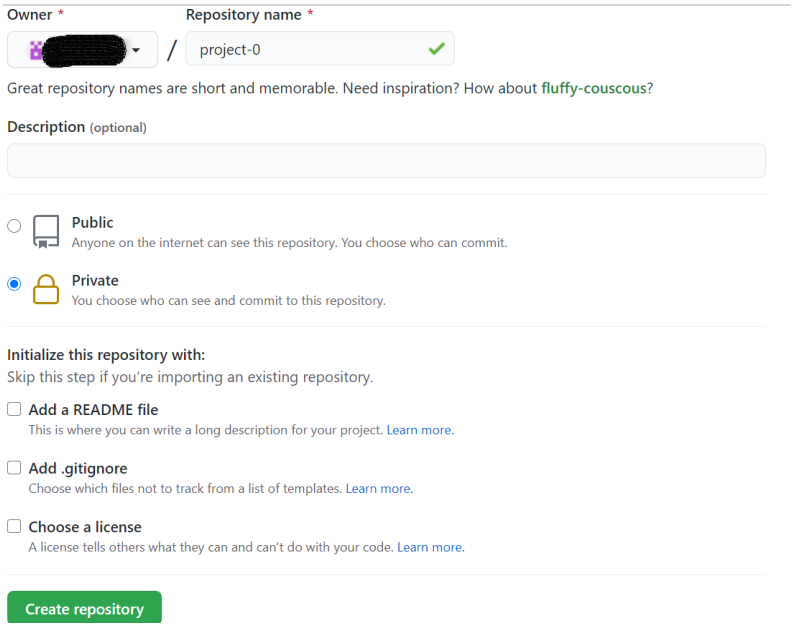
Git is a popular version control system (VCS). Like many version control systems, it keeps track of the changes that are made to your files over time and allows you to easily revert back to a previous version. This functionality is useful for software developers because it allows them to make changes to their code without worrying about breaking what they've already written. If something goes wrong with the new code, they can always go back to the last working version. Software development teams also use Git to facilitate collaboration between developers because Git allows changes made by different individuals to be easily merged and any conflicts resolved.

In this course, we recommend that you create a Git repository as the "home" for your code. Your project files will live in this repository stored in the cloud, independent from any IDE or computer. Git will keep track of the changes you make to your project. This will allow you to feel more confident when you experiment and make changes to your code. Git will also back up your projects so you don't have to worry about losing your files if something goes wrong. Additionally, Git provides an easy way to copy your files onto the CS lab machines.

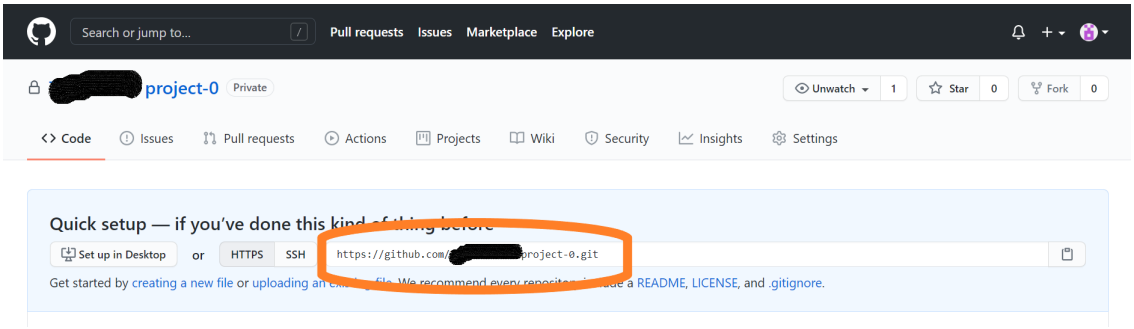
Instructions

1. Navigate to https://github.com. If you don't already have a GitHub account, create one. You can use any email address and username you would like.
2. Create a new private repository by selecting the appropriate button and filling out the requested information.
    a. The button for creating a repository may say something like "Create repository" or "New".
    b. You can choose any name you would like for the repository. For example, "project-0".
    c. You can add a description if you would like.

d. Make the repository private.
e. You do not need to initialize the repository with any of the recommended files.



3. Once you have your repository created, locate the url that identifies your repository. You will need this url for the next steps.
    a. The url should look something like this (without the angle brackets): https://github.com/<your-username>/<your-repository-name>.git



Clone your Repository on your Local Machine

Explanation

Now that you have your Git repository stored on the cloud through GitHub, let's clone (copy) the repository onto your local machine so that you can easily add files and make edits. We will use a terminal and Linux terminal commands to clone the repository to your machine.

Follow the following instructions to download Git and use Git and Linux commands to clone the repository.

Instructions

1. Make sure that you have Git installed. If you don't have Git installed, find the appropriate version here and follow the instructions.
    a. You can check if you have Git already by typing "git" (without the quotes) into a terminal or command line. If your terminal says the command is not recognized, you need to install Git.
2. Make sure you can access a terminal or command line that will recognize Linux commands.
    a. Most Mac computers come with a Terminal application that can do this.
    b. If you have a Windows machine, you may want to use Git Bash. This application should have been installed when you downloaded Git.
3. Open up your terminal.
4. Type "pwd" (without the quotes) to find out what directory you are currently in. This will print out the path to the current directory
5. Type "ls" (l as in the letter, not the number) to find out what folders and files are inside this directory. This will print out a list of all the folders and files.
6. Open up your file manager and see if you can use the information you learned in the previous steps to locate the directory open in the terminal in your file manager. This is to help you understand how the information in the terminal corresponds to the information in the file manager. As you complete the rest of this tutorial by using the terminal, you can follow along in your file manager if you would like.
7. Through the terminal, navigate to the directory where you would like to store your local clone of the Git repository. You may want to create a new directory specifically for cs236 files.
    a. The "cd" command allows you to navigate to a new directory. Use it as follows:
        i. Type "cd <directory-name>" to navigate to a directory inside the current directory. Replace <directory-name> with the name of the directory you want to navigate to. Do not include the angle brackets or the quotes.
        ii. Type "cd .." (without the quotes) to navigate to the parent directory of the current directory.
        iii. You can also provide an *absolute* or *relative* path to any directory to navigate to that directory. An *absolute* path starts with either ~ (to refer to your home directory) or / (to refer to your root directory). Then you list all the directories from the home or root directory to the directory you want to navigate to, separating each directory from the others with a /. A *relative* path means that you list all the directories from your current directory to the one you want to navigate to, again separating each directory from the others with a /.
        iv. Tip: hit the tab key to autocomplete the name of a file or directory.

        v. For examples and more instructions on how to use the cd command, check out this link.
    b. The “mkdir” command allows you to create a new directory. After you have navigated to the directory where you want the new directory to be created, use it as follows:
        i. Type “mkdir <directory-name>” (without angle brackets or quotes) to create a directory. Replace <directory-name> with the name you want the new directory to be called.
8. Once you have navigated to the directory where you want your local clone to be stored, clone the remote repository onto your machine.
    a. First, retrieve the URL that identifies your repository.
    b. Then, in the terminal, make sure you are in the desired directory and type “git clone <repository-url>” (without the quotes or angle brackets). Replace <repository-url> with the url for your repository.
9. Type “ls” (with the letter ‘l’, not the number one) and notice that you now have a new folder. This folder is your local Git repository.
10. **Take a screenshot showing your terminal, the git clone command, the ls command, and the resulting output.**
11. Feel free to experiment further with Linux commands if you would like. This is not required but you may find it helpful. Some other ones that you may want to look into include rm, cp, mv, cat, and nano. You can Google these Linux commands for more information.

Add Files to your Repository

Explanation

Now, you have a repository set up on the cloud and you have also cloned it on your local machine. However, the repository is empty. Let’s add some files to the local repository on your machine and then update the remote repository so it includes those same files.

To complete these instructions, it is important to know about what a Git commit is. A commit is like a snapshot of your files at a particular point in time. When you make a commit, you are saving the state and contents of your files into a snapshot. Git keeps track of your commits and will allow you to look at what the contents of the files were when a commit was made or restore your project files to a previous version that was captured by a commit.

Instructions

1. Download this zip which contains the files that you will be using for this tutorial.
2. Read the README file to get an understanding of what this program is designed to do.

3. Also, notice the .gitignore file. In this file, you can list the names of files or folders that you do not want to be stored in your Git repository; they will remain on your local machine only. Generally, IDE-specific files and executables are put in the .gitignore file. For this project, you don't need to worry about what is in the .gitignore since it is already written for you.
4. Move all the files from the zip into your local Git repository.
    a. You can use your file manager to drag and drop the files into the appropriate location.
    b. You can also use the "mv" command in the terminal. Use this command as follows:
        i. Type "mv <path-to-source> <path-to-destination>", without the quotes or angle brackets. Replace <path-to-source> with the path to the file you want to move. Replace <path-to-destination> with the path to the directory you want to move the file to.
5. In the terminal, navigate inside the Git repository. Type "ls" (without the quotes) to make sure the correct files are inside the repository.
6. Type "git status" (without the quotes) to see useful information about the repository. Notice that the output tells you that you have untracked files.
7. Type "git add ." (without the quotes). This tells Git that you want it to track all the files that are inside the current directory. It takes these files and the changes that have been made to them up to this point and moves them into a staging area. Changes in this staging area will be included in the next commit.
    a. Note that if you make additional changes to these files later that you also want to include in the commit, you will need to run the "git add" command again.
8. Type "git status" (without the quotes) again. Notice that Git now reports that there are changes ready to be committed.
9. Type 'git commit -m "Add initial files"' (without the single quotes, but including the double quotes).
    a. This tells Git to commit all the files and changes that are currently in the staging area. The -m tag tells Git that we want to add the message associated with the commit as a command line argument. "Add initial files" is the message associated with our commit.
        i. This message uses command voice, as if you were telling someone what to do. This is not required, but it is the convention for how commit messages are usually written.
10. Type "git push" (without the quotes). This tells Git to push the commit that we made on our local repository to the remote repository that is stored on the cloud.
11. Open up GitHub in your browser and check that the new files are in your repository.

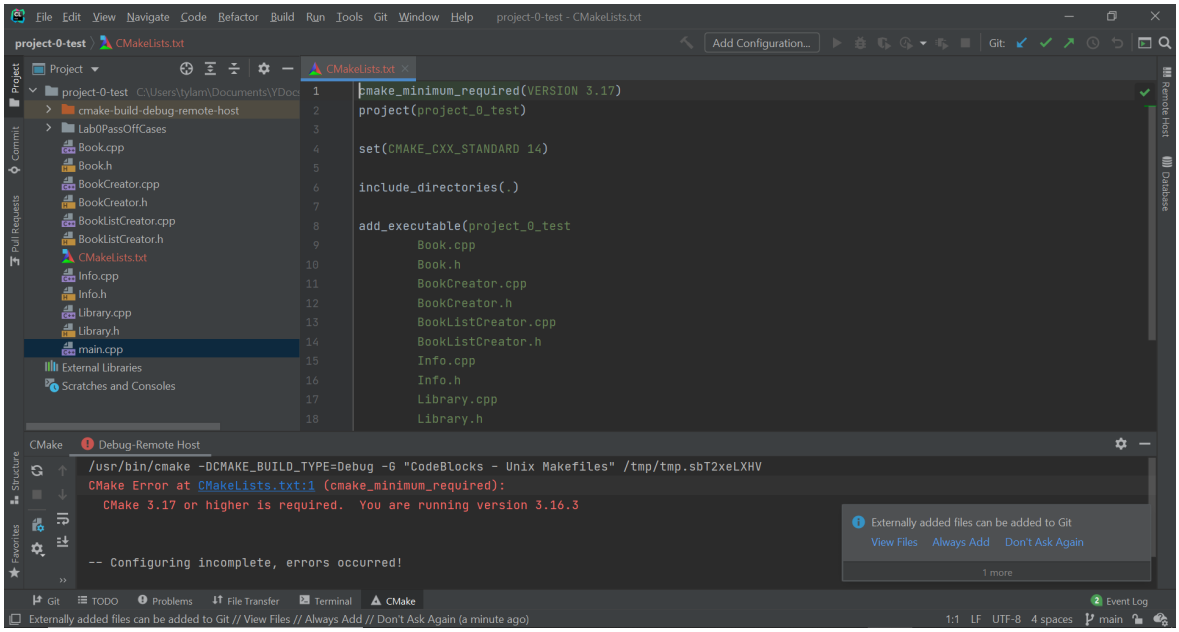
View your Repository in CLion

Explanation

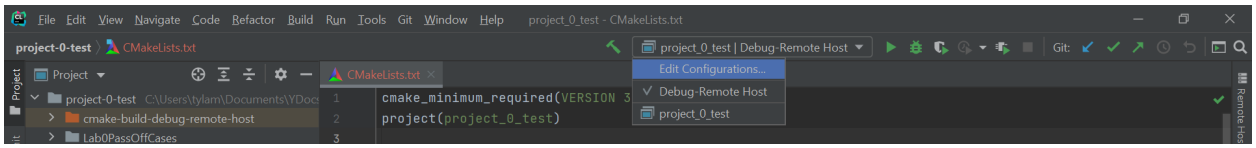
Now that you have some files in your repository, let's open the files up in an IDE so that we can run the code. CLion is the required IDE for this course. This IDE was carefully chosen because of its excellent debugging functionality, ease of use, and ability to connect remotely to the lab machines. In this part of the tutorial, we get CLion set up and we test our project on one of the input files.

Instructions

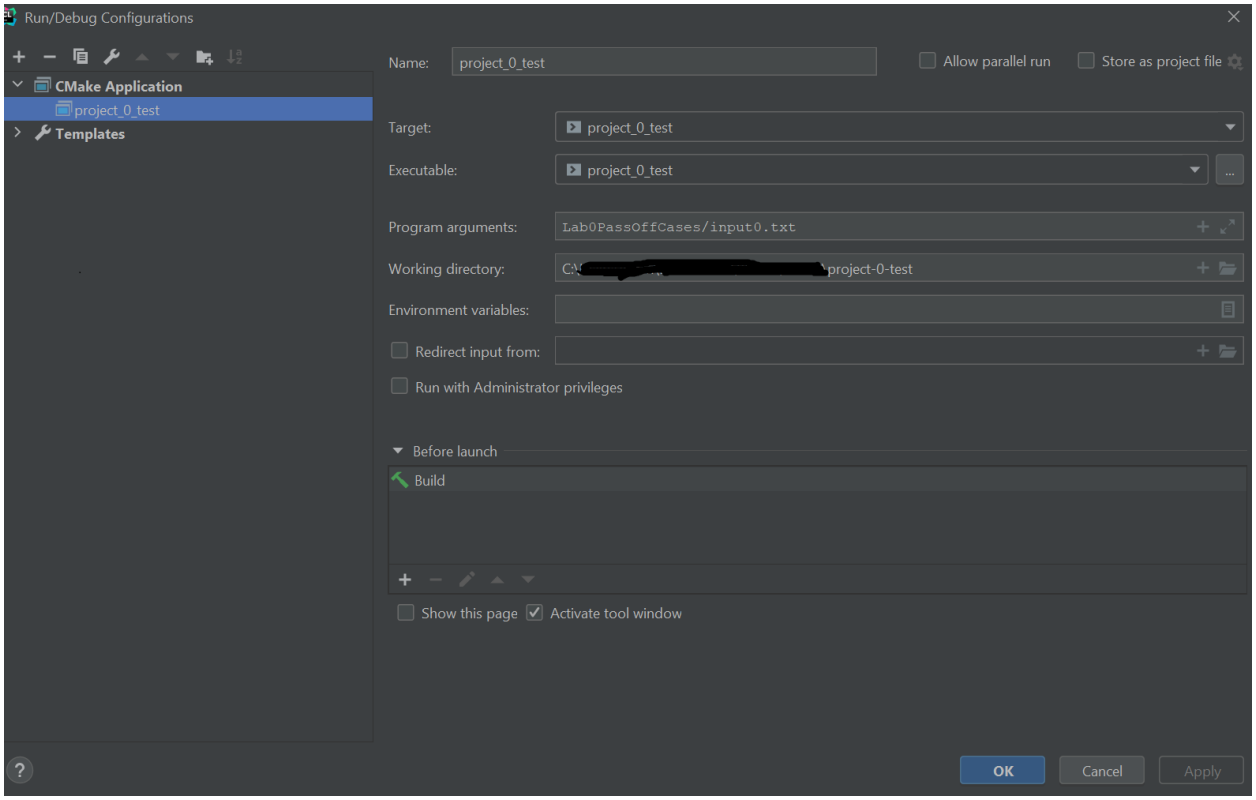
1. Create a JetBrains account with a student license by following these instructions.
2. Install CLion by finding the appropriate download here.
3. Open CLion up and select "Open" to open a project from existing sources.
4. Find and select the directory corresponding to your Git repository.
5. Configure CLion to run on the CS lab machines by following these instructions. (Recommended)
    a. Make sure that you are on campus and/or connected to the VPN.
    b. If you don't yet have a CS account that allows you to access the lab machines, or if you forgot your login information, access this page for the relevant instructions.
6. Open up main.cpp. Notice that you get a message at the top of the file that says "CMake project is not loaded". In order for CLion to properly run your code, it must generate some files specific to the CLion IDE.
7. Click "Create CMakeLists.txt" and make sure that all your .h and .cpp files are selected before clicking the "OK" button. This allows CLion to begin generating the necessary files.
    a. You may get the error shown below. If so, open your CMakeLists.txt and change the first line of text so that the version number is equal to the one that the error message says you are currently running.



8. Now you are ready to run your project! Hit the green play button in the upper right. You will notice that the code prints out a usage message instructing you how to use the program. Note that a command line argument specifying the input file is required. Let's learn how to add a command line argument in CLion.
9. A little to the left of the play button, there is a box with a drop down menu. Expand the menu and click "Edit configurations…"



10. Add the path "Lab0PassOffCases/input0.txt" as a program argument. Also add the current working directory (this is just whatever folder holds your project files). Select "Apply" and "OK".

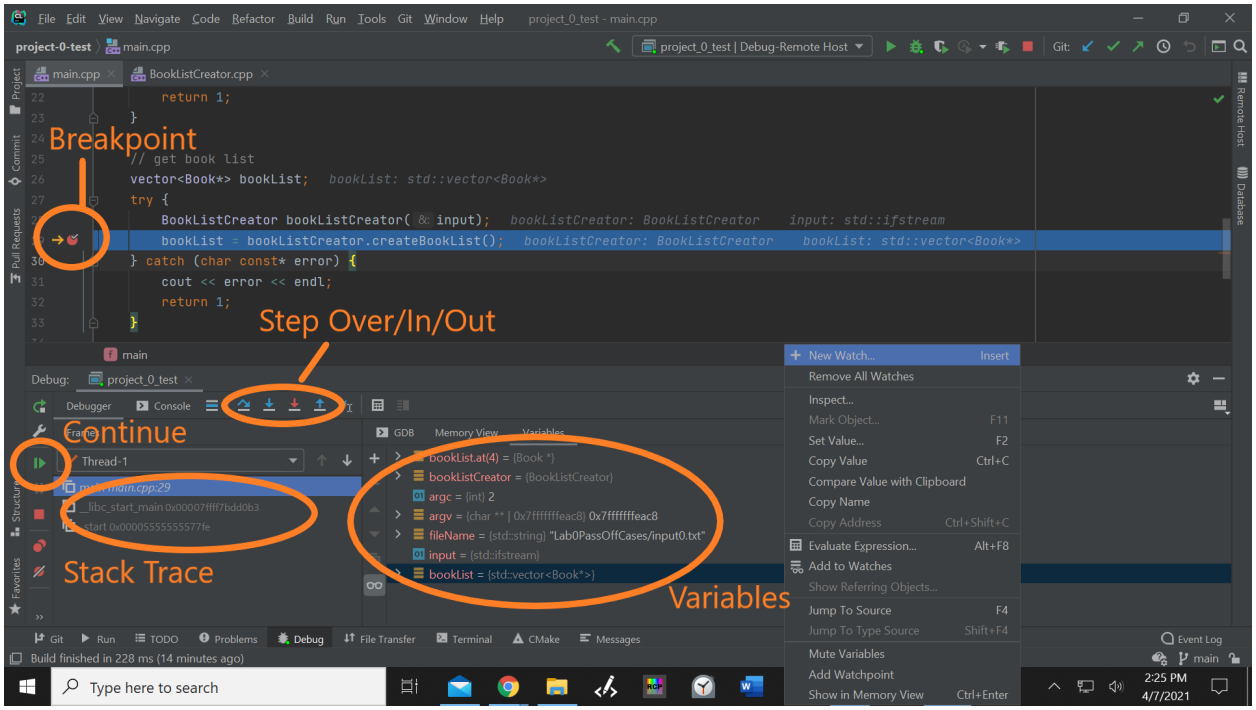


11. Run your program again. It should run now and produce some output. Compare the output to the file answer0.txt inside the Lab0PassOffCases folder. Notice that there are some differences. We will need to fix a few bugs so that the program runs as expected!

Debug your Code

Explanation

CLion's debugging tools can help you find bugs more quickly and easily than you can through code inspection or through print statements. First, learn about these tools by reading the explanations below or finding tutorials and instructions online. Then, apply those tools to help you fix the program so that the test case passes.



Breakpoints

Breakpoints allow you to pause your program at a particular line. You can then inspect the values of variables at that point in the code. To set a breakpoint in CLion, click the right of the line number that you would like to set the breakpoint on. Then, you can hit the green bug button (next to the green play button) to start debugging the program. The program should pause at the breakpoint.

Conditional breakpoints make it so that the program only pauses when a certain condition is true. This is useful, for example, if you know you have a problem occurring on the 100th iteration of a for loop. You can set a breakpoint such that it is only hit when your index variable is equal to 99. To set a conditional breakpoint, right click on a breakpoint and enter the condition in the box labelled “Condition:”.

Inspecting Variables

While the program is paused, you can inspect the values of the variables at that point in time. In the “Variables” window, there is a list of variables. If the variable is a list or object, you can click on the arrow to the left of the variable to see more information.

Custom watch variables let you designate a particular variable that you would like to keep an eye on so that you can see its value more easily. This is especially useful if the variable you want to look at is nested deeply inside other objects or lists. You can create a watch variable by right

clicking the “Variables” window and selecting “+ New Watch”. You can then enter the name/expression representing the variable you want to keep an eye on.

Step Over/Step In/Step Out/Continue

When you “step in” to a line of code, if there are any function calls then the program will pause again at the first line of the function. When you “step over” a line of code, it executes the line of code and pauses again after it has finished executing it. When you step out of a function, the program finishes executing the function and then pauses right after it returns from the function. When you resume or continue the program, the program runs until it hits the next breakpoint or until it finishes executing.

Stack Trace

The stack trace allows you to see the current stack of function calls at that point in the program. This is especially useful is your program is throwing an exception. Simply hit the debug button (you don’t even need to put breakpoints) and your program will pause where the exception was thrown. You can then examine the stack trace to find out what line of code caused the exception.

General Debugging Tips

One of the hardest parts of debugging can be determining where a problem occurred. Try these steps to help you diagnose the issues:

1. Start with the outermost function where you think the problem could be occurring. Make predictions (based on how you think your code should work) about what the values of variables should be after each line of code.
2. Set a breakpoint at the beginning of the function and begin stepping through. Step over each line rather than stepping in. After each line is executed, check the actual result against your predictions.
3. When you find a line of code where the result does not match your prediction, then you know that that line is where the problem is. If that line is a function call, then you should focus on debugging that function. Go back to step 1 and repeat the process.

As you become accustomed to this process, you might start to build up an intuition about the different places in your code that might cause a particular bug. Then, instead of going line by line as recommended in step 1 and 2, you can just skip to the parts of your code that you think could have caused the problem and check to see if they are giving you the desired outputs.

Instructions

1. Debug the code using the guidelines provided. **As you are debugging, take a screenshot showing a breakpoint that you set and the variables that you are examining.**
2. You should only have to fix 1 bug in order for your output (when using input1.txt as the input file) to match answer1.txt. **When you find the bug, make a comment in your code next to where the bug was**.

3. Be sure to run the add, commit, and push Git commands you learned previously to save and push your changes to the remote repository.

Running the Makefile

Explanation

We have provided a makefile that allows you to easily compile your code and then run all the tests to ensure they pass. This is much easier than testing each test case by hand. In this section, you will be accessing the lab machines remotely through the terminal and then running the makefile.

Instructions

1. Open up a terminal and access the CS lab machines remotely using ssh.
    a. Type "ssh <your-net-id>@schizo.cs.byu.edu", without the quotes or angle brackets. Replace <your-net-id> with your net ID.
    b. When asked for your password, provide your CS lab machine password.
2. Navigate to the directory where you would like to put your project files.
3. Clone your Git repository by typing "git clone <repository-url>" (without the quotes or angle brackets). Replace <repository-url> with the url for your repository. This copies all your project files to the lab machines.
4. Navigate inside the folder that has your project code inside. Rename the makefile so that it is called "makefile" rather than "makefile_0".
    a. You can do this rename by using the "mv" command that is usually used for moving files. Type "mv makefile_0 makefile" (without the quotes).
5. Type "make compile" to compile your code.
6. Type "make" to run all the tests. If a test works, then the only output will be "Running input <input-number>". If it fails, there will be additional output. Note that some of the tests are failing. Make note of which tests are failing.

Finishing the Project

Explanation

Running the makefile has revealed that our code still has a few more bugs. However, debugging through the terminal is not as easy or as nice as using an IDE. We will find the remaining bugs through CLion.

Instructions

1. Go back to CLion and change the command line argument so it is the path to one of the failing tests.
2. Debug the failing test cases using the same methods described previously. There should be 2 more bugs to fix. **For each bug, remember to put a comment in the code next to where you found the bug.**
3. When you think you have fixed all the bugs, remember to run the add, commit, and push Git commands to save and push your changes to the remote repository.
4. To make sure that all the tests are now passing, ssh back into the lab machines and navigate to the directory with your project code. Type “git pull” to pull the most recent changes from the remote repository.
5. Run the makefile again (you may have to rename it again). **If all the tests pass, take a screenshot of the terminal.** Otherwise, keep debugging.
6. When you are finished, submit your screenshots and a zip folder with your project code to Learning Suite.