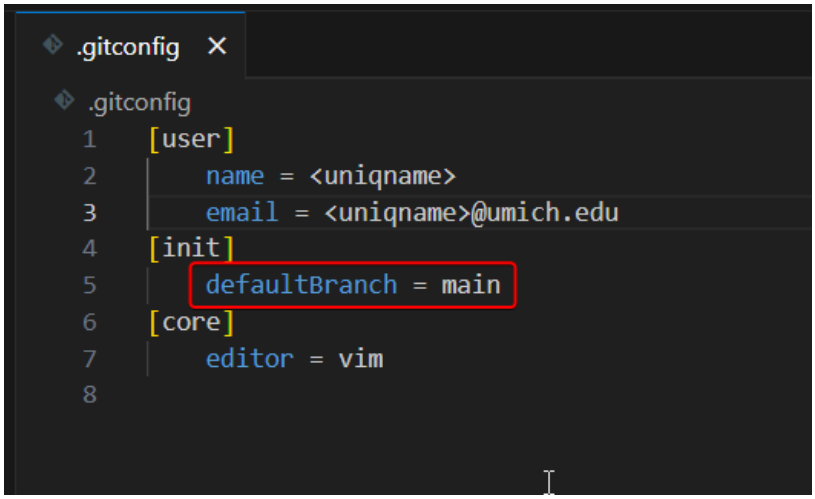
Basic terminology	
Git	According to Google: “a distributed version control system that tracks changes in any set of computer files, usually used for coordinating work among programmers who are collaboratively developing source code during software development”.
Repository	According to Google: “a central storage location for managing and tracking changes in files and directories”. A local repository is one stored on your computer’s disk. A remote repository is one hosted online, usually by a service such as github.com.
GitHub	According to Google: “a developer platform that allows developers to create, store, manage and share their code. It uses Git software, providing the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.” GitHub is not the same thing as Git.
Commit	According to Google: “the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project.” Commit messages should have a title in imperative mood whose first letter is capitalized, and a body containing more details about what was modified in the commit.
Commit history	A log/record of all your previous commits. All the commits in your current branch can be viewed by running **`$ git log`**. You can see a log of all recent Git actions by running **`$ git reflog`**. You can view a specific commit by running **`$ git show <commit_hash>`** (if no commit hash is provided, it shows the latest commit by default). Scroll up or down using the arrow keys, and run **`: q`** to exit the pager view.
File states	All files in a Git repository are in one of these four states: ● Untracked- the file is not being version-controlled by Git. ● Modified- the file is being tracked and was changed since the previous commit, but not staged. ● Staged- the file is being tracked and was changed since the previous commit, and is denoted as being part of the next commit. This is done by running **`$ git add <file_name>`**; it will be one of the files that are committed if **`$ git commit`** is run. Google the `-u` and `-A` flags as they’re particularly useful. ● Committed/clean- the file is being tracked but hasn’t been changed since the previous commit; it’s up-to-date.
How Git represents your commit history	Your commit history is essentially represented as a linked list of commits, where each node is a commit, and each commit has a pointer to the commit before it.
`HEAD`	A pointer to where you are in your commit history. This is typically the latest commit of the current branch you’re in.
Branch	A new/separate version of the main repository. More information about what branches are can be found in the link.

Text enclosed in **<angled brackets>** are templates and not literal.

<table>
<tr><td colspan="2">Creating and using SSH keys for both school (or work) and personal accounts- necessary for easily pushing commits from a local repository to a remote repository.</td></tr>
<tr><td>Initial setup</td><td>Set your global default email for Git to your school email by running
<code>$ git config --global user.email "<uniqname>@umich.edu"</code></td></tr>
<tr><td>Creating the SSH keys</td><td>1. Run <code>$ ssh-keygen -t ed25519 -C "<uniqname>@umich.edu"</code>
2. When prompted to enter which file to save the key, just press enter to save the key to the default location of <code>~/.ssh/id_ed25519</code>
You can also just press enter and make your passphrase empty when prompted.
3. Run <code>$ ssh-keygen -t ed25519 -C "<personal>@gmail.com"</code>
4. When prompted to enter which file to save the key, enter <code>~/.ssh/personalKey</code> (you may need to create the directory and file before doing this)
5. Create/modify <code>~/.ssh/config</code> to have the following contents:
<pre>Host github.com
    HostName github.com
    IdentityFile ~/.ssh/id_ed25519

Host gh_personal
    HostName github.com
    IdentityFile ~/.ssh/personalKey</pre></td></tr>
<tr><td rowspan="2">Adding the SSH public keys to GitHub</td><td>1. Run <code>$ cat ~/.ssh/id_ed25519.pub</code> and copy the resulting output to your clipboard
2. On the school GitHub account, click your profile picture > settings > SSH and GPG keys > New SSH key
3. In the “Title” field, choose an appropriate title (for example, “<uniqname> SSH”)
4. In the “Key” field, paste the public key you copied earlier
5. Click “Add SSH key”</td></tr>
<tr><td>1. Run <code>$ cat ~/.ssh/personalKey.pub</code> and copy the resulting output to your clipboard
2. On the personal GitHub account, click your profile picture > settings > SSH and GPG keys > New SSH key
3. In the “Title” field, choose an appropriate title (for example, “<personal> SSH”)
4. In the “Key” field, paste the public key you copied earlier
5. Click “Add SSH key”</td></tr>
</table>

<table>
<tr><td colspan="2">Using Git- commands you'll run with every project. Must have your SSH keys set up to do this. I also changed the default name of the master branch to 'main'.</td></tr>
<tr><td>Setting your default branch name to 'main'</td><td>Add the following code to ~/.gitconfig, as shown below:
[init]
defaultBranch = main
 
</td></tr>
<tr><td>Every time you create a new project</td><td>For your school account:
1. Run $ git init
2. (Optional) Run $ git config user.email and make sure that the output is <uniqname>@umich.edu. If it's not, run $ git config user.email "<uniqname>@umich.edu"
3. Create a new remote repository at https://github.com/new on your school GitHub account. Make it private, do not add a README file, and do not add a .gitignore
4. Run $ git remote add origin git@github.com:<school_GitHub_account_username>/<repository_name>.git
5. Make a README.md file formatted like this:
<Project name>
<Project description>
6. Make a .gitignore file. You may want it to include env/, node_modules/, *.txt, __pycache__/, .vscode/, etc.
7. Run $ git status to make sure you've ignored everything you should.
8. Run $ git add -A
9. Run $ git commit and title the commit "Initial commit"
10. Run $ git push -u origin --all</td></tr>
</table>

Note: when git clone'ing repository from this account, remember to change "github.com" to "gh_personal"	For your personal account: 1. Run **`$ git init`** 2. Run **`$ git config user.email "<personal>@gmail.com"`** 3. Create a new remote repository at https://github.com/new on your personal GitHub account. Make it private, do not add a README file, and do not add a .gitignore 4. Run **`$ git remote add origin git@gh_personal:<personal_GitHub account_username>/<repository_name>.git`** 5. Make a README.md file formatted like this: **`# <Project name>`** **`<Project description>`** 6. Make a .gitignore file. You may want it to include **`env/`**, **`node_modules/`**, **`*.txt`**, **`__pycache__/`**, **`.vscode/`**, etc. 7. Run **`$ git status`** to make sure you've ignored everything you should. 8. Run **`$ git add -A`** 9. Run **`$ git commit`** and title the commit "Initial commit" 10. Run **`$ git push -u origin --all`**
Daily workflow	`$ git status` (check if any files aren't saved yet) `$ git pull` (retrieve changes from the remote repository) `$ git add -A` (track all files, and stage all modified files) **`$ git add -u`** (stage all tracked, modified files) **`$ git commit`** (commit changes to the local repository) **`$ git push`** (push changes from the local repository to the remote repository)

Working with branches- see here on how you can pretty-print a graphical representation of your git history	
`$ git branch`	List the names of all branches in the current repository
`$ git branch <branch>`	Create a new branch named **`<branch>`**. **`<branch>`** will point to the same commit as **`HEAD`**.
`$ git switch <branch>`	Change the branch you're working in to **`<branch>`**. Doing this will change the state of the files on your disk to match the state that they're in for that particular branch.
`$ git stash`	Push the current state of the files on your disk to the top of the stash (a stack data structure), and change the state of the files on your disk to match the state they were in in the latest commit.
`$ git stash pop`	Pop the most recently added state from the top of the stash, and change the state of the files on your disk to match the popped state. Note that this is equivalent to running **`$ git stash apply`** and then **`$ git stash drop`**.
`$ git stash list`	Show the states currently on the stash
`$ git merge <branch>`	Merge **`<branch>`** into the branch you're currently in. This can be done in multiple ways: ● Fast forward- the current branch's pointer is moved up to point to the same commit as **`<branch>`**. ● Merge commit- you'll be prompted to create a new commit with multiple parents- the last commit in each of the branches you're merging. Your current branch will point to the new commit, while **`<branch>`** will keep pointing to where it was.

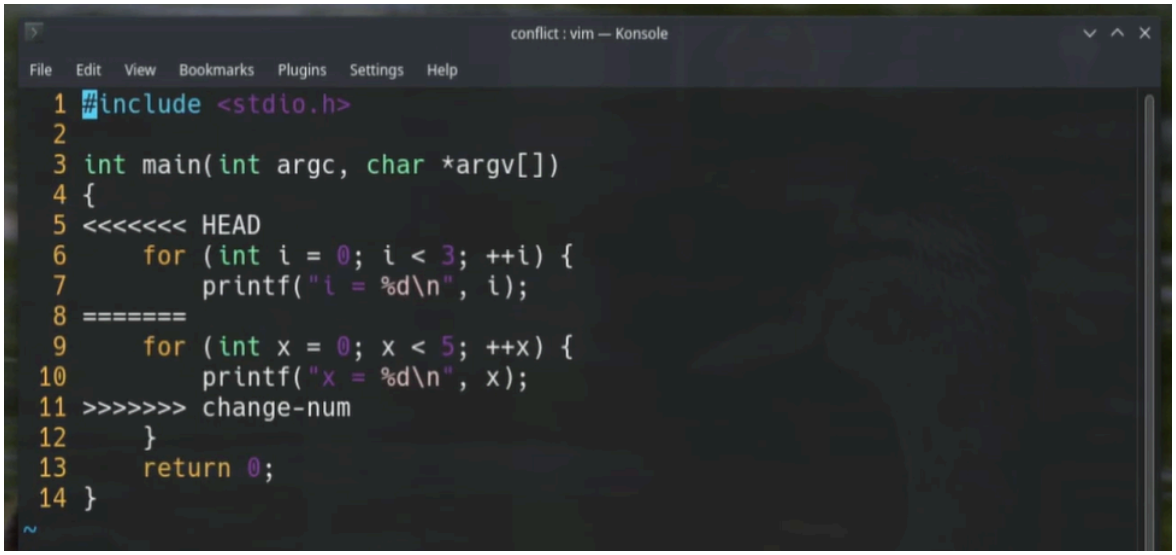
Working with remote repositories	
$ git clone <remote_repo_url>	Download the remote repository at **<remote_repo_url>** to your device.
$ git remote -v	List the remote repository associated with this local repository if there is one.
$ git remote add <remote_repo_name> <remote_repo_url>	Add a remote repository to be associated with this local repository. **<remote_repo_name>** is commonly chosen to be “origin”, but it doesn’t have to be. **<remote_repo_url>** can be the HTTPS or SSH url of the remote repository you’re adding. This must be done before you can set up any of your local branches to track any of the remote branches in this remote repository.
$ git remote remove <remote_repo_name>	Removes the remote repository named **<remote_repo_name>** associated with this local repository from being the upstream repository.
$ git branch --set-upstream-to=<remote_repo_name>/<remote_branch_name>	Set up your current local branch to track the remote branch **<remote_branch_name>**. Tracking means that your local branch will pull commits from and push commits to **<remote_branch_name>**. Note that **<remote_branch_name>** must exist in the remote repository.
$ git pull	Retrieve changes from the remote branch that your current local branch is set up to track, then merge them into your local branch. Note that this is equivalent to running **$ git fetch** and then **$ git merge**, so it can create a merge commit.
$ git pull --rebase	Retrieve changes from the remote branch that your current local branch is set up to track, set your local branch to match the remote branch, and then add the commits that are on your local branch but not on your remote branch on top of that. This does not create a merge commit; your commit history stays linear.
$ git reset --hard <remote_repo_name>/<remote_branch_name>	If you run this after running **$ git pull** it will make your local repository match the state of the remote repository exactly (though it will keep untracked files unchanged). Be VERY careful when running this, because you may not be able to undo the effects.
$ git push	Push changes from your current local branch to the remote branch that your current local branch is set up to track.
$ git push -u <remote_repo_name> <remote_branch_name>	Set up your current local branch to track the remote branch **<remote_branch_name>**, and push the changes from your current local branch to the remote branch **<remote_branch_name>**. **<remote_branch_name>** will be created in the remote repository if it does not already exist.
$ git push --force	Force the remote branch to exactly match your local branch. Be careful with this, especially if you’re working with others on the branch.

<table>
<tr><th colspan="2">Workflows (how to work with a remote repository that multiple people are adding to)</th></tr>
<tr><td>Everyone is pushing to/pulling from one main branch on the remote repository</td><td>To push your changes to the remote repository:
Run <code>$ git add -u</code> to stage your modified files
Run <code>$ git commit</code> to commit your staged files to your local repository
Retrieve the changes to the remote branch by running <code>$ git pull --rebase</code>
Run <code>$ git push</code> to push the commits on your local repository to the remote repository.</td></tr>
<tr><td rowspan="3">You make a new branch for every feature</td><td>To push your changes to the remote repository:
If you aren’t already in your feature branch, switch to the feature branch by running <code>$ git switch <feature_branch_name></code>
Run <code>$ git add -u</code> to stage your modified files
Run <code>$ git commit</code> to commit your staged files to your local repository
Switch to the main branch by running <code>$ git switch <main_branch_name></code>
Retrieve the changes to the remote branch by running <code>$ git pull</code>
You have two options here:</td></tr>
<tr><td>Nonlinear history with merge commits:<ol start="6">
Merge your main and feature branches by running <code>$ git merge <feature_branch_name></code>
Run <code>$ git push</code> to push the commits on your local repository to the remote repository.</td></tr>
<tr><td>Linear history:<ol start="6">
Switch to the feature branch by running <code>$ git switch <feature_branch_name></code>
Rebase your feature branch commits on top of your main branch commits by running <code>$ git rebase <main_branch_name></code>
Switch to the main branch by running <code>$ git switch <main_branch_name></code>
Fast-forward merge your main branch’s pointer to point to the same commit as your feature branch’s pointer by running <code>$ git merge</code>
Run <code>$ git push</code> to push the commits on your local repository to the remote repository.</td></tr>
</table>

Fixing mistakes	
$ git restore <file>	Restore **<file>** to the state that it was in in the latest commit
$ git reset <file>	Unstage **<file>**
$ git reset HEAD .	Unstage all staged files
$ git commit --amend	Modify the latest commit to include all the files that are currently staged (note: this technically creates a new commit and moves **HEAD** to point there instead of where it's currently pointing)
$ git reset HEAD~<number>	Move **HEAD** back **<number>** commits- if **<number>** is 1, it moves **HEAD** to point at the previous commit. This is a mixed reset, so it does not change the state of the files on your disk and keeps changes unstaged.
$ git reset --soft HEAD~<number>	Move **HEAD** back **<number>** commits. This does not change the state of the files on your disk and keeps changes staged.
$ git reset --hard HEAD~<number>	Move **HEAD** back **<number>** commits. This changes the state of the files on your disk to match their state in the commit you moved to.
$ git rebase -i <base tip hash>	A command that allows you to change your commit history in various ways. Work with all the commits after, but not including, **<base tip hash>**. You'll enter into an interactive text editor to modify a file whose first lines are formatted **<command> <commit hash> <commit title>**. By default, **<command>** is "pick" for each line, but can be changed to be any of the following depending on what you want to do: ● **p**, **pick**- use this commit ● **r**, **reword**- use this commit, but modify the commit message ● **e**, **edit**- once you exit the interactive text editor, you can add stuff to the commit you're editing, then confirm your changes by staging your modifications then running **$ git commit --amend**. Once you're done with all that, run **$ git rebase --continue** to finish the rebasing operation. ● **s**, **squash**- squash this commit into the first commit before it that is picked. You'll be prompted to change the commit message if you want. ● **f**, **fixup**- like squash, but discards the commit messages of the commits that you fixed up, keeping only the commit message of the picked commit that you fixed up into. For example, you could run **$ git rebase -i HEAD~2** and change the second commit's command from **p** to **f** to combine the two latest commits into just one commit. You can also re-order commits by changing the order in which they appear in the interactive text editor. Keep in mind that modifying a commit or its commit message technically creates a new commit, so the modified commit, as well as all commits after it, will have different commit hashes than they did before the rebase.

Dealing with merge conflicts

Merge conflicts occur when you try to merge branches that made modifications to the same line of code. Git will automatically insert text into the file in question to denote how the branches differ as shown below.



To address this, modify the file(s) where the merge conflict is happening. You'll need to remove the extra stuff that Git added like `<<<<<<< HEAD`, `=======`, and `>>>>>>> change-num` in this example. Once you've modified the file to be in the state you want it to be in, resolve the merge conflict by staging and committing the file.

Running code in CAEN

The autograder runs in CAEN, so it is a good idea to run your code in CAEN's environment to ensure your code works in the same environment as the autograder. If your code is in a remote repository, you can pull the code from the remote repository into CAEN.

First, make sure you are on the University of Michigan's wifi network. If you're not, connect to the University of Michigan's VPN. Then to connect to CAEN, run

```
$ ssh <uniqname>@login.engin.umich.edu
```

For the initial setup, create your SSH key(s) in CAEN using the instructions above. (You can also follow the instructions to use Git inside a VM). Then, you can clone your remote repository in CAEN and run your code. Example:

```
$ git clone git@github.com:<uniqname>/EECS281_Project1.git
```

To get your most recently pushed code in CAEN, you can run

```
$ git pull
```

To close your SSH connection to CAEN, run

```
$ exit
```