

Lab Exercise: Git, GitHub, and C

January 23, 2025

1 Why Version Control?

Version control systems like **Git** allow you to:

- Track all changes to your code
- Allow you to go back to any previous version
- Enable multiple people to work on different features simultaneously (branches)
- Keep a complete history of who changed what and why (commit messages)

GitHub:

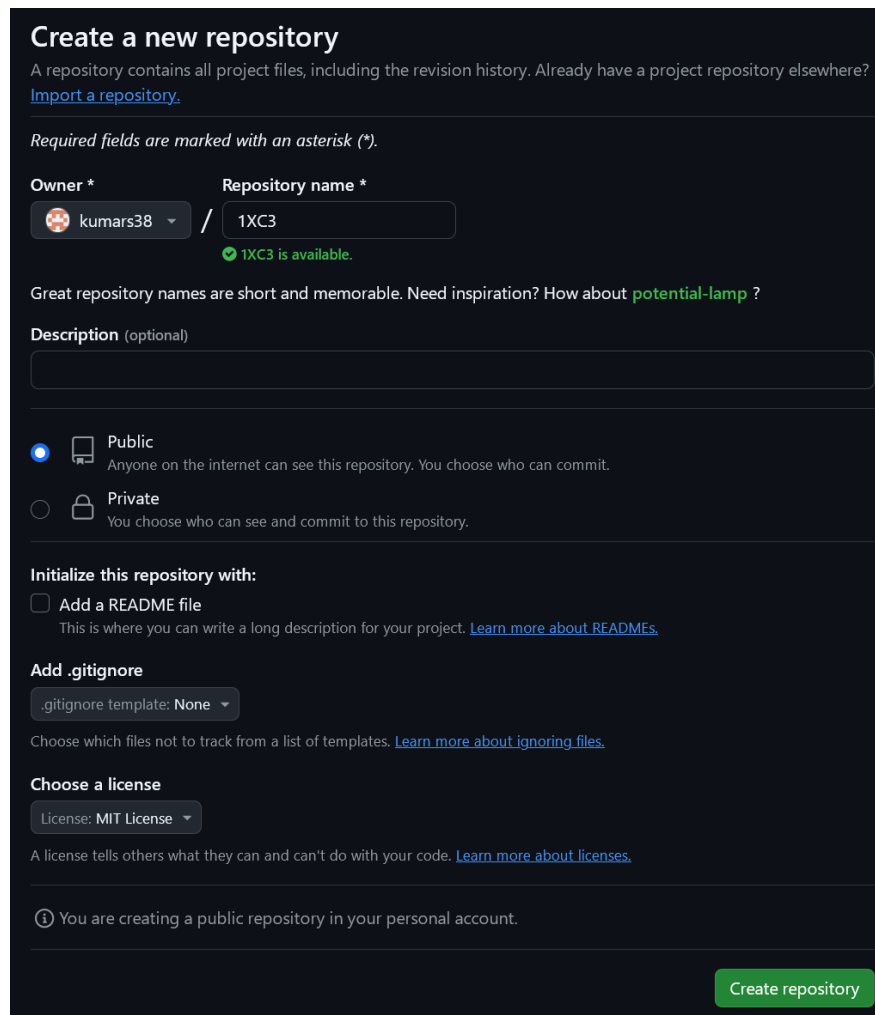
- Provides a central location + backup for your code
- Tools for code review and issue tracking
- Makes it easy to share and work on code with other people
- Show personal projects to employers

Git Quick Reference Guide

2 Exercise Steps

2.1 Part 1: Setting Up The Repository

1. Create a new repository on GitHub named “1XC3”. We will add the README and .gitignore files later ourselves, so leave them unselected. It is good practice to license your code (ex. MIT License).



The screenshot shows the GitHub 'Create a new repository' page. At the top, it says 'Create a new repository' and 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'. Below this, it states 'Required fields are marked with an asterisk (*)'. The 'Owner' field is set to 'kumars38' and the 'Repository name' field is set to '1XC3'. A green checkmark indicates '1XC3 is available.'. There is a link for 'potential-lamp ?'. The 'Description' field is optional and empty. Under 'Visibility', 'Public' is selected with a radio button. Below this, 'Initialize this repository with:' has 'Add a README file' unselected. The 'Add .gitignore' section shows '.gitignore template: None'. The 'Choose a license' section shows 'License: MIT License'. At the bottom, a note says 'You are creating a public repository in your personal account.' and a green 'Create repository' button is on the right.

2. Clone the repository to your local machine:

```
1 # Navigate to folder which will contain your repository folder
2 # I'll use my home directory (home/username) denoted by ~
3 cd ~
4
5 # Clone your repository
6 # ex. git clone https://github.com/kumars38/1XC3
7 git clone <your-repository-url>
8
9 # Move into your repository folder
10 cd 1XC3
11
12 # Confirm you are in the git repo
13 # Should see something like 'On branch main, Your branch is up to date'
14 git status
```

3. Create a project folder structure. Separating your repo contents into “src”, “assets”, “test”, “doc” (documentation), etc. is a common practice in software development.

```
1 # Create a folder for your source code
2 mkdir src
3
```

2.2 Part 2: Creating a C Program

1. Creating empty program:

```
1 # Move into the src directory
2 cd src
3
4 # Create a new, empty file
5 # equivalent to '> hello.c'
6 touch hello.c
7
```

2. Add this sample code to hello.c (Hint: use “nano hello.c” in the terminal):

```
1 #include <stdio.h>
2 int main() {
3     printf("Hello, World!\n");
4     return 0;
5 }
6
```

3. Compile and run your program:

```
1 # Compile the program
2 gcc hello.c -o hello
3
4 # Run the program
5 ./hello
6
```

2.3 Part 3: Git Continued

1. Check the status of your repository. It should tell you there are untracked files. This makes sense because we created a new directory (src), which contains hello.c

```
1 # Let's go back to the 1XC3 directory
2 cd ..
3 # Check status
4 git status
5
```

2. Add your C file to Git. “Adding” a file tells Git that you want this file to appear in your GitHub repo and want to track all changes to the file in the future.

```
1 # Add the C program
2 git add src/hello.c
3
```

3. Create and add a **README** file. This is a special file that shows its contents whenever someone visits the GitHub repository. So you can add a description of your project, authors, steps for usage/installation, etc.

```

1 echo "# COMPSCI 1XC3: Comp. Sci. Practice and Experience" > README.md
2 # We also want to add this README file to Git
3 git add README.md
4

```

4. Commit your changes. Committing is like creating a save point that you can come back to. Changes to files that have been tracked (added) get saved along with a message of what was changed. The message is useful for yourself and other people if working on a software team, as they can see the changes at a glance.

```

1 git commit -m "Initial commit"
2

```

5. After adding and committing, the final step is to push your changes to GitHub. You may need to set the upstream branch for the first push.

```

1 # For every push after this on same branch, you can just use 'git push'
2 git push -u origin main
3

```

6. If you check your git status, it will tell you that "src/hello" is untracked. In general, we exclude executable files like this from our GitHub repo. To do this, we can create a **.gitignore** file, which can be updated when necessary.

```

1 git status
2 # Create a .gitignore file which ignores a file called 'hello' in the src
  folder.
3 echo "src/hello" > .gitignore
4
5 # Following procedure from earlier
6 git add .gitignore
7 git commit -m "Created .gitignore file"
8 git push
9
10 # Check status again, no longer complains about src/hello.
11 git status
12

```

7. Refresh your GitHub repository in the browser and confirm that changes are present. You should be able to see all of the commits so far in the **main** branch.

2.4 Part 4: Making Changes with Branches

1. So far, we directly made changes and pushed to the main branch. In practice, this is not how we use git. Instead, you want to work on a feature in a separate branch, then merge the branch when changes are ready. This avoids conflicts between multiple developers, leads to cleaner code, and in case of new changes breaking the code, they can be easily reverted.

2. Create a new branch for adding features:

```

1 # Create and switch to a new branch called 'feature-scan-name'
2 git checkout -b feature-scan-name
3

```

3. Modify hello.c to ask for the user's name.

```

1  #include <stdio.h>
2  int main() {
3      char name[50];
4      printf("Please enter your name: ");
5      scanf("%s", name);
6      printf("Hello, %s!\n", name);
7      return 0;
8  }
9

```

Aside: do you understand what `char name[50]`, `scanf`, and `%s` are doing in the code?

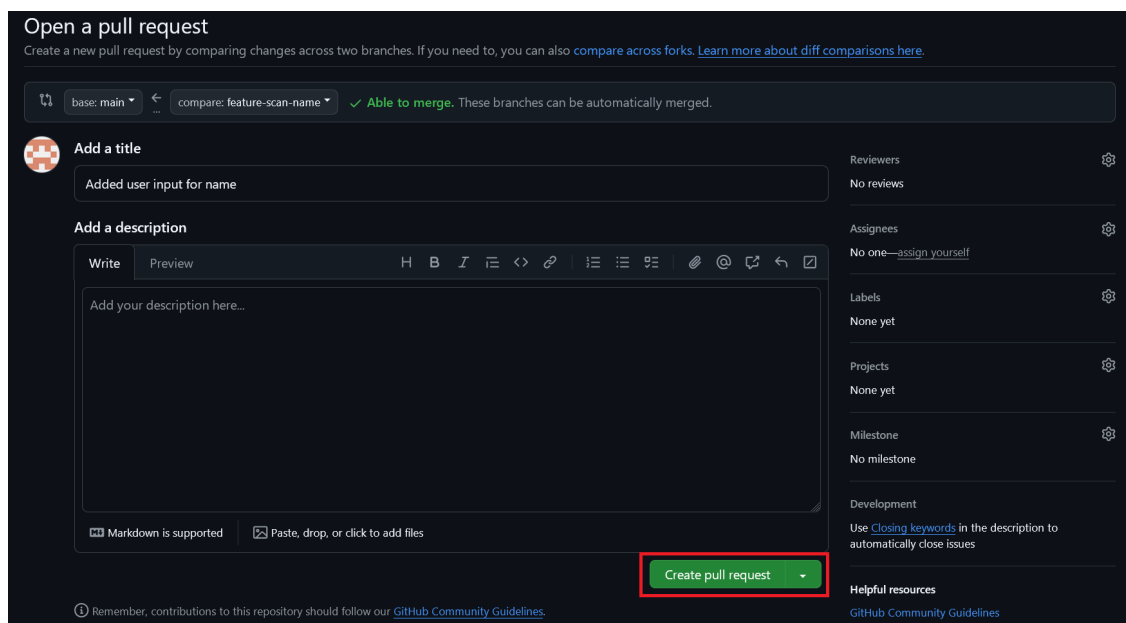
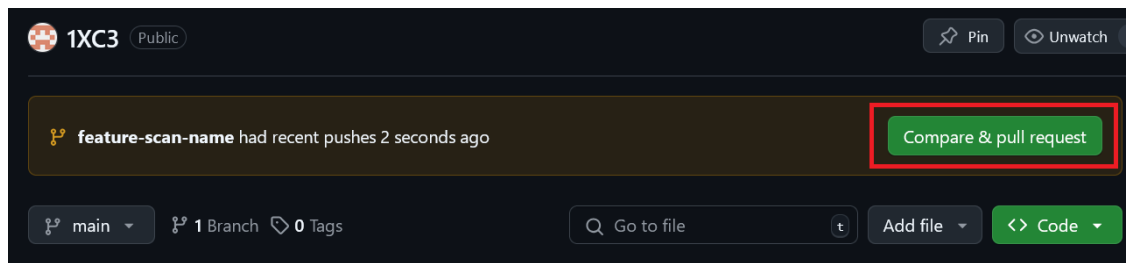
4. Commit and push your changes:

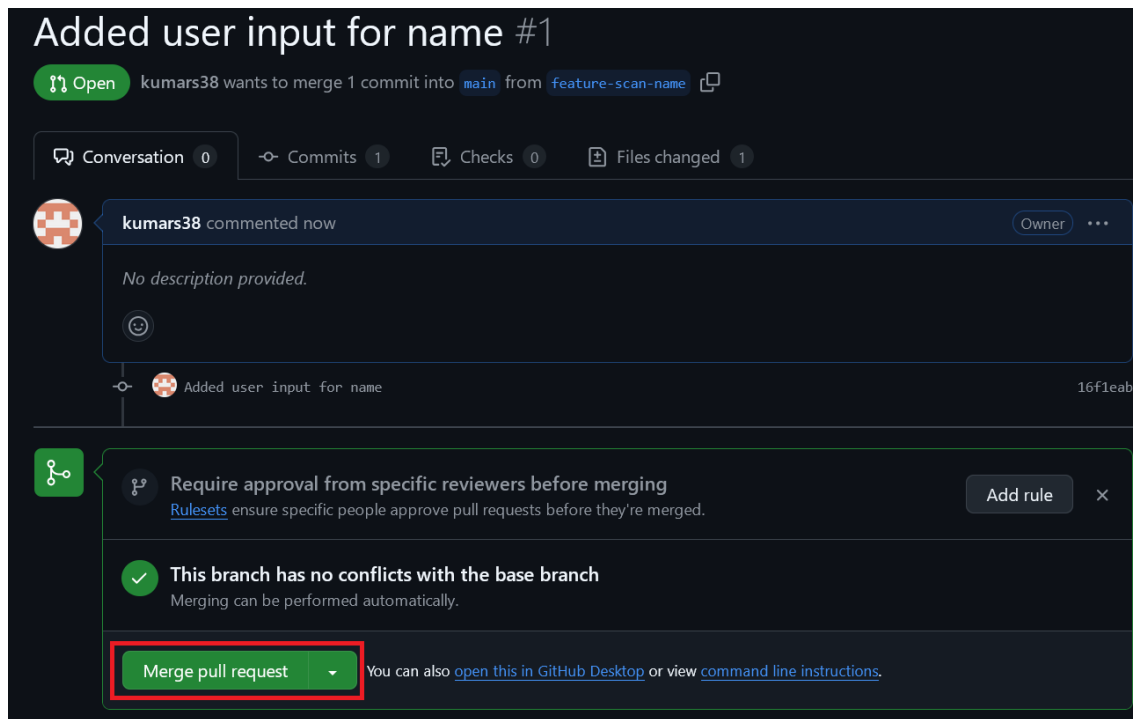
```

1  # Recall, we added this file earlier.
2  git add src/hello.c
3  git commit -m "Added user input for name"
4
5  # Create remote branch, syncs with our local branch
6  git push -u origin feature-scan-name
7

```

- Go to GitHub and create a pull request. Imagine you're on a project with multiple collaborators. You can tell them to review your pull request (PR), so they can easily see your changes on the feature branch. Then, if it looks good, they can approve and merge your changes into a central branch (ex. main). In this exercise, you can just merge the code yourself.





6. Merge your changes on GitHub, following above. Once changes are merged, you can safely delete the branch called “feature-scan-name” from the repo through the GitHub interface.

7. Update your local main branch:

```
1 # Should see the new changes after this
2 git checkout main
3 git pull
4
```

8. Try compiling and executing the modified hello.c file, to ensure it is working.

```
1 gcc src/hello.c -o src/hello
2 src/hello
3
```

3 Working with Merge Conflicts

Merge conflicts occur when Git can’t automatically merge changes from different branches. Let’s go through an example:

1. Create two branches from main:

```
1 cd src
2 git checkout main
3 # Good practice to always pull remote changes before making modifications
4 git pull
5 git checkout -b feature1
6 # Modify line 4 of hello.c (using editor like nano), asking for their
7 # first name instead of just name. ex:
8 # printf("Please enter your first name: ");
9
# As usual: add, commit, and push the changes
```

```

10  git add hello.c
11  git commit -m "Ask user for first name"
12  git push -u origin feature1
13
14  # Switching back to the main branch, the modification does not exist here.
15  git checkout main
16  git pull
17  git checkout -b feature2
18  # Modify line 4 of hello.c, asking for their last name ex:
19  # printf("Please enter your last name: ");
20  git add hello.c
21  git commit -m "Ask user for last name"
22  git push -u origin feature2
23

```

2. Briefly confirm the changes in GitHub. You should see feature1 and feature2 branches have both appeared with distinct changes to hello.c

3. Now try to merge both into main:

```

1  git checkout main
2  git pull
3  git merge feature1
4  git merge feature2 # This will cause a conflict because the commit from
   this branch occurs on the same line
5

```

4. Resolve the conflict:

- Open the conflicted file (hello.c)
- Look for conflict markers (<<<<<<, =====, >>>>>>)
- Choose which changes to keep (let's choose to keep '... enter your first name: ')
- Remove the conflict markers
- Save the file

5. Complete the merge. You will always have to 'git add' files that were involved in merge conflicts, confirming the final changes you want to keep. Following the usual process, we then commit and push the changes.

```

1  git add hello.c
2  git commit -m "Resolved merge conflict between feature1 and feature2"
3  git push
4

```

6. With the conflict resolved, confirm that hello.c is working with the changes from feature1 (ask for **first** name)

```

1  gcc hello.c -o hello
2  ./hello
3

```

That's it for the exercise.

4 More Useful Git Commands

These are just for your reference, you don't have to follow anything step-by-step here.

4.1 Undoing Changes

- Undo staged changes (undo git add):

```
1 git restore --staged <file>
2
```

- Discard local changes to file:

```
1 git restore <file>
2
```

- Reset all local changes back to the last **local** commit:

```
1 git reset --hard HEAD
2
```

- Reset all local changes back to the last **remote** (i.e. GitHub) commit:

```
1 git reset --hard origin/<branch-name>
2
```

- Force push changes (*):

```
1 git push --force-with-lease # Safer force push
2
```

4.2 Viewing History

- View commit history:

```
1 git log --oneline --graph # Compact view with branch structure
2
```

- See changes:

```
1 git diff # Changes in working directory
2 git diff --staged # Staged changes
3 git show # Last commit changes
4
```

4.3 Working with Branches

- List branches:

```
1 git branch -a # All branches (local and remote)
2
```

- Delete branches:

```
1 git branch -d <branch> # Safe delete
2 git branch -D <branch> # Force delete (*)
3
```

4.4 Stashing Changes

- Save changes temporarily:

```
1 git stash
2 git stash pop # Apply and remove stash
3 git stash apply # Apply but keep stash
4 git stash list # Show stashed changes
5
```


5 Test your Knowledge

Answer the following questions:

1. What does the command `git status` tell you?
2. Why do we create branches instead of working directly on main?
3. What is the difference between `git add` and `git commit`?
4. What happens when you run `git pull`?
5. Why do we need to compile C programs?

Explain what each of these commands does:

- `pwd`
- `ls -l`
- `cd ..`
- `mkdir test`
- `rm -r test`