# Git Tutorial

## Beginer

Git is a source code version control system. Such a system is most useful when you work in a team, but even when you're working alone, it's a very useful tool to keep track of the changes you have made to your code.

### A. Con□gure your git environment?

Set git user name and email:

```
$ git config --global user.name "Your Full Name"
$ git config --global user.email you@somewhere.com
```

Git stores this information in the **~/.gitconfig** □le.

You may also want to set the EDITOR environment variable to vim or emacs; this controls which editor you use to enter log messages..

### B. Creating a project

Let's create a new directory, ~/tmp/test1, for our rst git project.

```
//~/your_user/Documents
$ cd
$ mkdir tmp
$ cd tmp
$ mkdir test1
$ cd test1
```

Put the directory under git revision control:

```
$ git init
```

If you type ll (I'll assume that ll is an alias for ls -alF), you will see that there is a .git directory. e git repository for the current directory is stashed in the .git directory. Let's start our programming project. Write hello.c with your editor:

```
#include <stdio.h>
int main()
{
printf("%s\n", "hello world");
return 0;
}
```

Compile and run it:

```
$ gcc hello.c
$ ./a.out
```

Let's see what git thinks about what we're doing:

```
$ git status
```

e git status command reports that hello.c and a.out are "Untracked." We can have git track hello.c by adding it to the "staging" area (more on this later):

```
$ git add hello.c
```

Run git status again. It now reports that hello.c is "a new le to be committed." Let's commit it:

```
$ git commit
```

Git opens up your editor for you to type a commit message. A commit message should succinctly describe what you're committing in the rst line. If you have more to say, follow the rst line with a blank line, and then with a more through multi-line description. For now, type in the following one-line commit message, save, and exit the editor.

```
Added hello-world program.
```

Run git status again. It now reports that only a.out is untracked. It has no mention of hello.c. When git says nothing about a le, it means that it is being tracked, and that it has not changed since it has been last committed.

We have successfully put our rst coding project under git revision control.

### C. Modifying Files

Modify hello.c to print "bye world" instead, and run git status. It reports that the file is "Changed but not updated." This means that the File has been modified since the last commit, but it is still not ready to be committed because it has not been moved into the

staging area. In git, a file must go to the staging area before it can be committed.

Before we move it to the staging area, let's see what we changed in the le:

```
$ git diff
```

Or, if your terminal supports color,

```
$ git diff --color
```

The output should tell you that you took out the "hello world" line, and added a "bye world" line, like this:

```
-printf("%s\n", "hello world");
+printf("%s\n", "bye world");
```

We move the file to the staging area with git add command;

```
$ git add hello.c
```

In git, "add" means this: move the change you made to the staging area. The change could be a modification to a tracked file, or it could be a creation of a brand new file. This is a point of confusion for those of you who are familiar with other version control systems such as Subversion.

At this point, git diff will report no change. Our change—from hello to bye—has been moved into staging already. So this means that git diff reports the di erence between the staging area and the working copy of the file.

To see the difference between the last commit and the staging area, add --cached option:

```
$ git diff --cached
```

Let's commit our change. If your commit message is a one-liner, you can skip the editor by giving the message directly as part of the git commit command:

```
$ git commit -m "changed hello to bye"
```

To see your commit history:

```
$ git log
```

You can add a brief summary of what was done at each commit:

```
$ git log --stat --summary
```

Or you can see the full diff at each commit:

```
$ git log -p
```

And in color:

```
$ git log -p --color
```

### D. Flexibility

One of Git's key design objectives is flexibility. Git is flexible in several respects: in support for various kinds of nonlinear development workflows, in its efficiency in both small and large projects and in its compatibility with many existing systems and protocols.

Git has been designed to support branching and tagging as first-class citizens (unlike SVN) and operations that affect branches and tags (such as merging or reverting) are also stored as part of the change history. Not all version control systems feature this level of tracking.

### E. Git tracked, the modiffied, and the staged

A file in a directory under git revision control is either tracked or untracked. A tracked file can be unmodiffied, modi ed but unstaged, or modiffied and staged. Confused? Let's try again.

There are four possibilities for a file in a git-controlled directory:

1. Untracked

   Object files and executable files that can be rebuilt are usually not tracked.

2. Tracked, unmodiffied

   The file is in the git repository, and it has not been modiffied since the last commit. git status says nothing about the file.

3. Tracked, modiffied, but unstaged

   You modiffied the file, but didn't git add the file. The change has not been staged, so it's not ready for commit yet.

4. Tracked, modiffied, and staged

   You modiffied the file, and did git add the file. The change has been moved to the staging area. It is ready for commit.

The staging area is also called the "index."

## F. Other useful git commands

Here are some more git commands that you will find useful.

To rename a tracked file:

```
$ git mv old-filename new-filename
```

To remove a tracked file from the repository:

```
$ git rm filename
```

The mv or rm actions are automatically staged for you, but you still need to git commit your actions. Sometimes you make some changes to a file, but regret it, and want to go back to the version last committed. If the file has not been staged yet, you can do:

```
$ git checkout -- filename
```

If the file has been staged, you must first unstage it:

```
$ git reset HEAD filename
```

There are two ways to display a manual page for a git command. For example, for the git status command, you can type one of the following two commands:

```
$ git help status
$ man git-status
```

Lastly, git grep searches for specified patterns in all files in the repository. To see all places you called printf():

```
$ git grep printf
```

## G. Cloning Project

You created a brand new project in the test1 directory, added a file, and modified the file. But more often than not, a programmer starts with an existing code base. When the code base is under git version control, you can clone the whole repository.

Let's move up one directory, clone test1 into test2, and cd into the test2 directory:

```
$ cd ..
$ git clone test1 test2
$ cd test2
```

Type `ll` to see that your hello.c file is cloned here. Moreover, if you run git log, you

will see that the whole commit history is replicated here. git clone not only copies the latest version of the files, but also copies the entire repository, including the entire commit history. After cloning, the two repositories are indistinguishable.

Let's make some changes—and let's be bad. Edit hello.c to replace printf with **printf%^&**, save and commit:

```
$ vim hello.c
$ git add hello.c
$ git commit -m "hello world modification - work in progress"
```

Now run git log to see your recent commit carrying on the commit history that was cloned. If you want to see only the commits after cloning:

```
$ git log origin..
```

Of course you can add -p and --color to see the full diff in color:

```
$ git log -p –color origin..
```

Let's make one more modiffication. Fix the printf, and perhaps change the "**bye world**" to "**rock my world**" while we're there.

```
$ vim hello.c
$ git add hello.c
$ git commit -m "fixed typo & now prints rock my world"
```

Run git log -p --color origin.. again to see the two commits you have made a□er cloning.

## H. Adding a directory into your repository

Enter the original **test1** directory, create the solution subdirectory, and add two files to it:

```
$ cd ../test1
$ mkdir solution
$ cd solution
$ cp ../hello.c .
$ echo 'hello:' > Makefile
```

Type **ll** to see that two files—**Makefile** and **hello.c**—have been created in the solutions directory. (**hello.c** was copied from the parent directory, and **Makefile** was created directly on the command line using the echo command.

Now, move out of the solution directory, and **`git add & git commit`** the solution directory:

```
$ cd ..
$ git add solution
$ git commit -m "added solution"
```

Note that git add solution stages all files in the directory

## I. Pulling changes from a remote repository

When others are working on the same project, you will often want to get updated files from a remote repository. You do that by "pulling" the changes in my repository into your repository. Let's pull the changes we just made in test1 into test2:

```
$ cd ../test2/
$ git pull
```

The git pull command looks at the original repository that you cloned from, fetches all the changes made since the cloning, and merges the changes into the current repository. You now have the solution right in your repository.

## J. Working with remote repositories: sharing

See what your remote is:

```
$ git remote # what's our remote
$ git remote -v # some more info
```

If you can add remote repositories like gitlab, githab using this command to establishing a remote connection with github or gitlab and calling it origin (a common name used or you can used any name), with the address we are connecting to.

```
$ git remote add origin [urlLink]
```

To sets the target branch you can uses thin command, you can set any branch of remote repositories:

```
$ git branch -M main
```

If you want to push all our contents from our local repository to the remote repository or cloud. Use this command;

```
$ git origin main
```

is used to fetch and download content from a remote repository and immediately

update the local repository to match that content

```
$ git pull
```

# Reference


- http://www.cs.columbia.edu/~sedwards/classes/2013/4840/git-tutorial.pdf
- https://web.stanford.edu/class/cs124/lec/git_tutorial.pdf
- https://docs.gitlab.com/ee/tutorials/learn_git.html
- https://git-scm.com/docs
- https://git-scm.com/docs/gitworkflows