Git Tutorial

DR. YIHSIANG LIOW (JANUARY 30, 2025)

Contents

1 Introduction

1.1 What is version control software, git, github?

"Git is a distributed revision control system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Every Git working directory is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Git's current software maintenance is overseen by Junio Hamano. Git is free software distributed under the terms of the GNU General Public License version 2."

– Wikipedia

Basically git can keep track of files in your project directory. You can think of that as a backup copy. But it's more than that because git remembers the history of changes to your files. So for instance you can say "OK ... I took a wrong turn. I would like to replace my code with the version 2 hours ago."

The bank that contains the backup copies of your files is called the repository or repo for short. Of course you also have your own project directory containing your project files. Your project directory is called the working directory. Basically you need to know how to add/delete/modify your files in the bank/repository and how to copy the files in your bank/repository back to your project directory (in case you screwed up.)

Such a software is called a version control software.

Github (http://www.github.com) is a website that allows you to store your repositories online. You can access github through your git software running in your laptop. In other words if you use github to store your project, you project appears in (at least) three places: your project directory, a repo on your laptop, and a repo in github. There are two benefits of using github: it's a backup (do you really trust your laptop's HD? or your own backups?) and github allows you to work collaboratively with others.

1.2 Goal

The purpose of this document is to give you the most useful and commands in their simplest form. Git (or any version control system) is a complex system. Instead of telling you everything about git in a top-down fashion, I'll tell you the story of git from three points of view.

- Part 1: I'll talk about using git with the repo on your local machine in a simple setting.
- Part 2: I'll talk about using git and github.
- Part 3: In Part 3, I'll talk about branches.

1.3 Pre-requisites

1. You're using a Fedora machine. This set of notes has been tested on Fedora 31.
2. You have gone over `unix1.pdf` on basic linux commands
3. You know how to write and compile C++ program. I'll be using C++ source files as examples.
4. You have gone over my `make.pdf` on makefiles. However I'll only be using very simple makefiles.

Some text files (example: makefile) will require you to insert tab characters. This is how you do it with emacs. Here's an example of a `makefile`:

```
# File: makefile
helloworld.exe: helloworld.cpp
        g++ helloworld.cpp -o helloworld.exe

clean:
        rm helloworld.exe

run:
        ./helloworld.exe
```

The identation must be a tab character. In emacs, to insert a tab character do `C-q [tab]`.

To create a blank file do

```
[student@localhost git] touch a.txt
```

To create a text file with some text do

```
[student@localhost git] echo 'hello world' > a.txt
```

1.4 How to use this doc

As in all my other docs, this is written to be a tutorial. You learn by doing and not just reading.

2 Part 1: Git

debug: installati

2.1 Installation

In all the following, I'm working in a directory called `git`. We'll be creating several repositories inside `git`. You might want to do the same too.

Short version: Install git-core on your system. Go to next section.

Longer version: Read on. Open a bash terminal shell and become root (by doing `su`) and install git by doing

```
[root@localhost ~] dnf -y install git-core
```

Logout of root (by doing `exit`). (You should know by now that you should get out of root as soon as possible after system administrative tasks, right?)

debug: config.te

2.2 Basic configuration

Add your name, email address, favorite editor to git (replace my info with yours):

```
[student@localhost git] git config --global user.name "Yihsiang Liow"
[student@localhost git] git config --global user.email "yliow@ccis.edu"
[student@localhost git] git config --global core.editor "emacs"
```

(When git wants a "commit message" (see later) from you, it will open an editor using the editor you specified above.

Exercise 2.1. By the way, the configuration data is stored in `~/.gitconfig`. Go ahead and take a look at it.

OK ... let's start using git ...

2.3 Create a Repository

Create a directory for our experiment. I'll be calling my directory `testgit`.

```
[student@localhost git] rm -rf testgit
[student@localhost git] mkdir testgit
[student@localhost git] cd testgit
```

In `testgit` create a git repository:

```
[student@localhost testgit] git init
Initialized empty Git repository in home/student/testgit/.git/
```

Now, files/directories created in `testgit` can be stored in this git repository.

You will see a `.git` directory in your `testgit` directory:

```
[student@localhost testgit] ls -la
total 28
drwxrwxrwx. 1 root root     0 Jan 30 02:22 .
drwxrwxrwx. 1 root root 28672 Jan 30 02:22 ..
drwxrwxrwx. 1 root root     0 Jan 30 02:22 .git
```

I'll give you 10 seconds to peek around in the `.git` directory, but you had better not change anything in it.

```
[student@localhost testgit] ls -la .git
total 10
drwxrwxrwx. 1 root root 4096 Jan 30 02:22 .
drwxrwxrwx. 1 root root    0 Jan 30 02:22 ..
drwxrwxrwx. 1 root root    0 Jan 30 02:22 branches
-rwxrwxrwx. 1 root root  130 Jan 30 02:22 config
-rwxrwxrwx. 1 root root   73 Jan 30 02:22 description
-rwxrwxrwx. 1 root root   23 Jan 30 02:22 HEAD
drwxrwxrwx. 1 root root 4096 Jan 30 02:22 hooks
drwxrwxrwx. 1 root root    0 Jan 30 02:22 info
drwxrwxrwx. 1 root root    0 Jan 30 02:22 objects
drwxrwxrwx. 1 root root    0 Jan 30 02:22 refs
```

Exercise 2.2. What happens when you init a repository twice? If you have a helloworld.cpp in this directory, will re-initializing the git repository for this directory wipe out that file?

2.4 Adding a file

In `testgit`, create a hello world program, say we call it `helloworld.cpp`:

```
// File: helloworld.cpp
#include <iostream>

int main()
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

Here's a `makefile` you can use:

```
# File: makefile
helloworld.exe: helloworld.cpp
        g++ helloworld.cpp -o helloworld.exe

clean:
        rm helloworld.exe

run:
        ./helloworld.exe
```

This is what we now have in our `testgit` directory:

```
[student@localhost testgit] ls -la
total 33
drwxrwxrwx. 1 root root     0 Jan 30 02:22 .
drwxrwxrwx. 1 root root 28672 Jan 30 02:22 ..
drwxrwxrwx. 1 root root  4096 Jan 30 02:22 .git
-rwxrwxrwx. 1 root root   118 Jan 30 02:22 helloworld.cpp
-rwxrwxrwx. 1 root root   136 Jan 30 02:22 makefile
```

Now we add `helloworld.cpp` to our git repository.

First we do this:

```
[student@localhost testgit] git add helloworld.cpp
```

This tells git to remember to process `helloworld.cpp`. This is called staging `helloworld.cpp` or we say that `helloworld.cpp` is placed in the staging area.

Then we do this:

```
[student@localhost testgit] git commit -m "initial"
[master (root-commit) 8e6e77f] initial
 1 file changed, 8 insertions(+)
 create mode 100644 helloworld.cpp
```

The string `"initial"` is just a comment/documentation for this commit. You can use whatever string you like except that it cannot be empty. (If you enter an empty string, git will prompt you to enter a commit message using an editor. If you did not configure your editor, git will use vim. In vim, type `i`, enter a commit message, press the `[esc]` key, the `:` (the colon), the `w`, and then the `q`. Yes, that's pretty bizarre. I won't explain. If you want to learn more about vim, search the web for vim tutorials.)

Note also that next to `master (root-commit)` is a bunch of 7 random-looking characters:

`8e6e77f`

That's the id of the commit. We'll be doing a lot of commits. Each commit is identified by the commit id.

A commit id is actually pretty long; only 7 characters of the id are shown above. When referring to the id, you usually only need to specify the first 5-7 characters of the id. (The id is a SHA1 hash.)

debug: tag.tex

2.5 Tagging a commit

It's kind of annoying to use the id since it's not readable and easy to mis-type. So I'm going to use something more readable. This is called a tag. The general syntax to assign a commit a tag is like this:

```
git tag [tagname] [id]
```

If you don't specify the id, git assumes that you're tagging the last commit.

I'm going to tag our first commit with a tag name of `v-1`. In general, I'm going to tag with each commit with `v-` followed by the commit message. (v = version.)

Do this to tag the last commit:

```
[student@localhost testgit] git tag v-1
```

If you don't mind typing in the commit id, then you don't have to tag. But in this tutorial, I'll be mostly using tags.

2.6 Status

Now go ahead and do this:

```
[student@localhost testgit] git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        makefile
nothing added to commit but untracked files present (use "git add" to track)
```

AHA! This tells us that `makefile` has not been added and committed to the repository. We have forgotten about the makefile.

Instead of adding one file at a time, you can do this to add everything in the current directory:

```
[student@localhost testgit] git add .
```

. is the current directory. This will recursively add all changes of this directory.

Now when we check the status again:

```
[student@localhost testgit] git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   makefile
```

As you can see, git now knows that there's a new file called `makefile`.

But remember this does not mean that the file is actually in git yet because you need to do git commit:

```
[student@localhost testgit] git commit -m "makefile"
[master e15eb3c] makefile
 1 file changed, 9 insertions(+)
 create mode 100644 makefile
[student@localhost testgit] git tag v-2
```

Exercise 2.3. Check the status on your own. Read the output message.

At this point, `helloworld.cpp` and `makefile` are in our git repository.

Exercise 2.4. We have created a directory, initialized it with git, created some files, then put the files into a git repo. But what if you *already* have a project directory and you want to put that into a git repo? Do this in 5 minutes:

1. Create another directory and go into it.
2. Create a hello world program with makefile.
3. Do "`git init`".
4. Do "`git status`".
5. Do "`git add .`" on *both* files.
6. Do "`git commit`".

Exercise 2.5. When you have time (after you're done with this tutorial), you can look over all your work in your laptop and see which one you want to version with git.

debug: list.tex

2.7 List repository files

The git status basically does a simple comparison between your project files against what's kept in the git repository. If you want to see the names of all the files in the repository, do this:

```
[student@localhost testgit] git ls-files
helloworld.cpp
makefile
```

2.8 Log

To see the history log we do this:

```
[student@localhost testgit] git log
commit e15eb3cc3ed1e640c1f862b4a0f726bb1eccf5c0
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:22:12 2025 -0500
    makefile
commit 8e6e77fe664a9843c1d1c0deb791ba6f87b40dc2
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:22:06 2025 -0500
    initial
```

(The long chains of random characters after the words commit are the ids.)

Notice that tags are not shown. To show the tags, you do:

```
[student@localhost testgit] git log --decorate=full
commit e15eb3cc3ed1e640c1f862b4a0f726bb1eccf5c0 (HEAD -> refs/heads/master, tag:
 refs/tags/v-2)
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:22:12 2025 -0500
    makefile
commit 8e6e77fe664a9843c1d1c0deb791ba6f87b40dc2 (tag: refs/tags/v-1)
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:22:06 2025 -0500
    initial
```

For a summary, do this:

```
[student@localhost testgit] git log --oneline
e15eb3c makefile
8e6e77f initial
```

or this if you want a summary with tags:

```
[student@localhost testgit] git log --oneline --decorate=full
e15eb3c (HEAD -> refs/heads/master, tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

When the log is long, you can use `-1` for the last long entry, `-2` for the last two, etc:

```
[student@localhost testgit] git log -2 --oneline --decorate=full
e15eb3c (HEAD -> refs/heads/master, tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

You can also see the log of a specific file.

```
[student@localhost testgit] git log --oneline --decorate=full makefile
e15eb3c (HEAD -> refs/heads/master, tag: refs/tags/v-2) makefile
```

2.9 Ignore files

At this point, let's compile and run the program:

```
[student@localhost testgit] make
g++ helloworld.cpp -o helloworld.exe
[student@localhost testgit] make run
./helloworld.exe
hello world
```

Of course this means you have an extra file: `helloworld.exe`.

You frequently want to do git status to, for instance, see what files you have created but have forgotten to git-add-commit. In the case of `helloworld.exe`, you probably do not want to git-add-commit this file. But everytime you do git status, you are reminded that you forgot to git-add-commit `helloworld.exe`:

```
[student@localhost testgit] git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        helloworld.exe
nothing added to commit but untracked files present (use "git add" to track)
```

It's kind of annoying. So let's just tell git to ignore those files.

In `testgit` directory, create a file `.gitignore` with these two lines:

```
*.exe
*~
```

(The $*$ is a wildcard character just like in linux commands.)

Now do git status again:

```
[student@localhost testgit] git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
nothing added to commit but untracked files present (use "git add" to track)
```

See the difference?

However git reports on `.gitignore`. Let's add the file:

```
[student@localhost testgit] git add ".gitignore"
```

and commit:

```
[student@localhost testgit] git commit -m ".gitignore"
[master b44803a] .gitignore
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

and tag it:

```
[student@localhost testgit] git tag v-3
```

Now when we do git status we get this:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
```

Exercise 2.6. Create another directory with these files:

```
# File: helloworld.py
def helloworld():
    print("hello world")
```

```
# File: main.py
import helloworld
helloworld.helloworld()
```

```
# File: makefile
run:
        python main.py
```

1. Version these files with git.
2. Run make.
3. Do git status.
4. One file was created when you ran make. That file was automatically generated and need not be stored in git. Make git ignore that file. In fact look at the file extension of that file. Make git ignore files with the file extension.

5. Do git status again and make sure git does not bother you with that auto generated file.

Remove this directory when you are done with this experiment.

2.10 Modifying files

Suppose we modify a file, say `helloworld.cpp`:

```
// File: helloworld.cpp
#include <iostream>

int main()
{
    std::cout << "hello world!!!" << std::endl;
    return 0;
}
```

Let's do a git status:

```
[student@localhost testgit] git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   helloworld.cpp
no changes added to commit (use "git add" and/or "git commit -a")
```

Git status tells us that `helloworld.cpp` is modified. Let's commit the new version (the changed version) of `helloworld.cpp`:

```
[student@localhost testgit] git add helloworld.cpp
[student@localhost testgit] git commit -m "helloworld!!!"
[master 97fe22d] helloworld!!!
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testgit] git tag v-4
```

and do a git status:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
```

and check the log:

```
[student@localhost testgit] git log --oneline --decorate=full
97fe22d (HEAD -> refs/heads/master, tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

What if I change both `helloworld.cpp`:

```
// File: helloworld.cpp
#include <iostream>

int main()
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

and:

```
# File: makefile
helloworld.exe: helloworld.cpp
        g++ helloworld.cpp -o helloworld.exe

clean:
        rm helloworld.exe

run:
        ./helloworld.exe

r:
        ./helloworld.exe
```

You can do

```
[student@localhost testgit] git add .
[student@localhost testgit] git commit -m "helloworld! and add r for makefile"
[master 881fe4d] helloworld! and add r for makefile
 2 files changed, 4 insertions(+), 1 deletion(-)
[student@localhost testgit] git tag v-5
```

Here's the git status:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
```

and git log:

```
[student@localhost testgit] git log --oneline --decorate=full
881fe4d (HEAD -> refs/heads/master, tag: refs/tags/v-5) helloworld! and add r fo
r makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

2.11 Deleting a file

Now suppose I don't want the `makefile`. Here's how to remove it:

```
[student@localhost testgit] ls -la makefile
-rwxrwxrwx. 1 root root 158 Jan 30 02:22 makefile
[student@localhost testgit] git rm makefile
rm 'makefile'
```

Now when we look for makefile, it's not there anymore:

```
[student@localhost testgit] ls -la makefile
ls: cannot access 'makefile': No such file or directory
```

Let's commit the change:

```
[student@localhost testgit] git commit -m"rm makefile"
[master 37e5cb5] rm makefile
 1 file changed, 12 deletions(-)
 delete mode 100644 makefile
[student@localhost testgit] git tag v-8
```

Here's the git status and log:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
[student@localhost testgit] git log --oneline --decorate=full
37e5cb5 (HEAD -> refs/heads/master, tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

Note that git rm removes the file from your directory and records this fact for git commit.

Now what if, instead of git rm, we forgot and did `rm`?

First let's recreate `makefile`:

```
# File: makefile
helloworld.exe: helloworld.cpp
        g++ helloworld.cpp -o helloworld.exe

clean:
        rm helloworld.exe

run:
        ./helloworld.exe

r:
        ./helloworld.exe
```

and commit it:

```
[student@localhost testgit] git add makefile
[student@localhost testgit] git commit -m "makefile"
[master 7edf12b] makefile
 1 file changed, 12 insertions(+)
 create mode 100644 makefile
```

Now go ahead and delete makefile and do git status:

```
[student@localhost testgit] rm makefile
[student@localhost testgit] git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    makefile
no changes added to commit (use "git add" and/or "git commit -a")
```

I see git warning me that the `makefile` has been deleted. Don't panic. You just do git rm and commit:

```
[student@localhost testgit] git rm makefile
rm 'makefile'
[student@localhost testgit] git commit -m "rm makefile again"
[master 8d5b3c3] rm makefile again
 1 file changed, 12 deletions(-)
 delete mode 100644 makefile
```

Here's the status and log:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
[student@localhost testgit] git log --oneline --decorate=full
8d5b3c3 (HEAD -> refs/heads/master) rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

Yet another thing you can do it is to execute

```
[student@localhost testgit] git add makefile
```

This will add `makefile` to be deleted at the next git commit.

So git add does not mean "add the new files". It means record the changes (addition/deletion/mofication to files) for the next commit.

Exercise 2.7. You can commit multiple adds, rms, and modifications.

1. Create another directory and put that in git.
2. Create three files `a.txt`, `b.txt`, `c.txt`.
3. Commit all the three files.
4. Now modify `a.txt`, delete `b.txt`, and add a new file `d.txt` and commit all changes with the least number of git commands.

Answer on next page.

Here's the initial setup. First I create a directory:

```
[student@localhost git] rm -rf test42; mkdir test42
```

Next I create the files and commit them:

```
[student@localhost test42] git init
Initialized empty Git repository in home/student/test42/.git/
[student@localhost test42] touch a.txt; touch b.txt; touch c.txt
[student@localhost test42] git add .
[student@localhost test42] git commit -m "init"
[master (root-commit) b2cad24] init
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt
 create mode 100644 b.txt
 create mode 100644 c.txt
[student@localhost test42] git status
On branch master
nothing to commit, working tree clean
[student@localhost test42] git log --oneline --decorate=full
b2cad24 (HEAD -> refs/heads/master) init
```

Now I make changes and commit:

```
[student@localhost test42] echo "hello world" > a.txt
[student@localhost test42] git rm b.txt
rm 'b.txt'
[student@localhost test42] touch d.txt
[student@localhost test42] git add .
[student@localhost test42] git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   a.txt
        renamed:    b.txt -> d.txt
[student@localhost test42] git commit -m "mod a.txt, rm b.txt, add d.txt"
[master bf4b899] mod a.txt, rm b.txt, add d.txt
 2 files changed, 1 insertion(+)
 rename b.txt => d.txt (100%)
[student@localhost test42] git status
On branch master
nothing to commit, working tree clean
[student@localhost test42] git log --oneline --decorate=full
bf4b899 (HEAD -> refs/heads/master) mod a.txt, rm b.txt, add d.txt
b2cad24 init
```

2.12 Checkout: recovering file from last commit

Here's `helloworld.cpp`:

```
[student@localhost testgit] less helloworld.cpp
// File: helloworld.cpp
#include <iostream>
int main()
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

Suppose you accidentally removed your cpp file:

```
[student@localhost testgit] rm helloworld.cpp
[student@localhost testgit] ls -la helloworld.cpp
ls: cannot access 'helloworld.cpp': No such file or directory
```

YOU SHOULDN'T HAVE DONE THAT!!!

OK. Don't panic. Do this:

```
[student@localhost testgit] git checkout helloworld.cpp
Updated 1 path from the index
```

This recovers `helloworld.cpp` from the last commit state. We quickly verify:

```
[student@localhost testgit] ls -la helloworld.cpp
-rwxrwxrwx. 1 root root 119 Jan 30 02:22 helloworld.cpp
[student@localhost testgit] less helloworld.cpp
// File: helloworld.cpp
#include <iostream>
int main()
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

and check the status:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
```

and log:

```
[student@localhost testgit] git log --oneline --decorate=full
8d5b3c3 (HEAD -> refs/heads/master) rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

Note that

```
[student@localhost testgit] git checkout helloworld.cpp
```

checkouts `helloworld.cpp` from the *last* commit.

Exercise 2.8. Open `helloworld.cpp`. Change `world` to `columbia` in your code. Save the file. YIKES! WHAT DID YOU DO? YOUR HELLOWORLD PROGRAM WAS CORRECT TO BEGIN WITH! Change your program to what it was ... using git. Test it. □

Exercise 2.9. Do the following experiment:

1. Create a directory and version it.
2. Commit 1: Create file f. Git add-commit f.
3. Commit 2: Remove f and commit.
4. Commit 3: Create file g. Git add-commit g.
5. Do "`checkout git f`". What's the problem?

2.13 Checkout by tag/id

Now suppose I'm not happy with the current version of `helloworld.cpp`:

```
[student@localhost testgit] less helloworld.cpp
// File: helloworld.cpp
#include <iostream>
int main()
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

and want to go back to the version with tag `v-4`.

Do this:

```
[student@localhost testgit] git checkout v-4 helloworld.cpp
Updated 1 path from 88955b7
```

and check:

```
[student@localhost testgit] less helloworld.cpp
// File: helloworld.cpp
#include <iostream>
int main()
{
    std::cout << "hello world!!!" << std::endl;
    return 0;
}
```

If we're happy with this we can commit:

```
[student@localhost testgit] git add .
[student@localhost testgit] git commit -m "revert to cpp of v-4"
[master 38e16f0] revert to cpp of v-4
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testgit] git tag v-6
```

and check the status and log:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
[student@localhost testgit] git log --oneline --decorate=full
38e16f0 (HEAD -> refs/heads/master, tag: refs/tags/v-6) revert to cpp of v-4
8d5b3c3 rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

We can also replace the `helloworld.cpp` with the one in `v-5` using the commit id (you only need the first 5 characters of the SHA1 hash):

```
[student@localhost testgit] git checkout 881fe4d helloworld.cpp
Updated 1 path from 37ea447
[student@localhost testgit] less helloworld.cpp
// File: helloworld.cpp
#include <iostream>
int main()
{
    std::cout << "hello world!" << std::endl;
    return 0;
}
```

We then commit:

```
[student@localhost testgit] git add .
[student@localhost testgit] git commit -m "revert to v-5 of cpp"
[master a22d3be] revert to v-5 of cpp
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testgit] git tag v-7
```

Here's the status and log:

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
[student@localhost testgit] git log --oneline --decorate=full
a22d3be (HEAD -> refs/heads/master, tag: refs/tags/v-7) revert to v-5 of cpp
38e16f0 (tag: refs/tags/v-6) revert to cpp of v-4
8d5b3c3 rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

You can also checkout multiple files using for instance

```
[student@localhost testgit] git checkout v-4 *.cpp makefile
Updated 2 paths from 88955b7
```

2.14 Renaming a file

Suppose now I want to rename `helloworld.cpp` to `helloworld2.cpp`. Do this:

```
[student@localhost testgit] git mv helloworld.cpp helloworld2.cpp
[student@localhost testgit] git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    helloworld.cpp -> helloworld2.cpp
        new file:   makefile
```

Then I commit:

```
[student@localhost testgit] git commit -m "helloworld2.cpp"
[master c0734ca] helloworld2.cpp
 2 files changed, 10 insertions(+), 1 deletion(-)
 rename helloworld.cpp => helloworld2.cpp (59%)
 create mode 100644 makefile
[student@localhost testgit] git tag v-9
```

Here's git status and log

```
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
[student@localhost testgit] git log --oneline --decorate=full
c0734ca (HEAD -> refs/heads/master, tag: refs/tags/v-9) helloworld2.cpp
a22d3be (tag: refs/tags/v-7) revert to v-5 of cpp
38e16f0 (tag: refs/tags/v-6) revert to cpp of v-4
8d5b3c3 rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
97fe22d (tag: refs/tags/v-4) helloworld!!!
b44803a (tag: refs/tags/v-3) .gitignore
e15eb3c (tag: refs/tags/v-2) makefile
8e6e77f (tag: refs/tags/v-1) initial
```

That's it.

Of course it's possible that you did a `mv` instead of `git mv`. What will happen? Let's try it out. First go back to the beginning of this section:

```
[student@localhost testgit] git mv helloworld2.cpp helloworld.cpp
[student@localhost testgit] git commit -m "undo"
[master 9911e6a] undo
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename helloworld2.cpp => helloworld.cpp (100%)
```

Now suppose we forget and do a `mv` (instead of `git mv`):

```
[student@localhost testgit] mv helloworld.cpp helloworld2.cpp
```

When we do a git status, we see a problem:

```
[student@localhost testgit] git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    helloworld.cpp
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        helloworld2.cpp
no changes added to commit (use "git add" and/or "git commit -a")
```

Git detects a new file `helloworld2.cpp` and a missing file `helloworld.cpp`. You can do git add on `helloworld2.cpp` and git rm on `helloworld.cpp`:

```
[student@localhost testgit] git add helloworld2.cpp
[student@localhost testgit] git rm helloworld.cpp
rm 'helloworld.cpp'
[student@localhost testgit] git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    helloworld.cpp -> helloworld2.cpp
[student@localhost testgit] git commit -m "add helloworld2.cpp, rm helloworld.cp
p"
[master fc39ebd] add helloworld2.cpp, rm helloworld.cpp
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename helloworld.cpp => helloworld2.cpp (100%)
[student@localhost testgit] git status
On branch master
nothing to commit, working tree clean
```

Problem solved.

In some cases git will detect a mv so the above won't be necessary. Try this experiment. Let's create a directory

```
[student@localhost git] rm -rf testmv; mkdir testmv
```

go into it, make a git repo

```
[student@localhost testmv] git init
Initialized empty Git repository in home/student/testmv/.git/
```

and create some files and commit them:

```
[student@localhost testmv] echo "hello world" > a.txt
[student@localhost testmv] echo "hello columbia" > b.txt
[student@localhost testmv] ls -la
total 29
drwxrwxrwx. 1 root root     0 Jan 30 02:23 .
drwxrwxrwx. 1 root root 28672 Jan 30 02:23 ..
-rwxrwxrwx. 1 root root    12 Jan 30 02:23 a.txt
-rwxrwxrwx. 1 root root    15 Jan 30 02:23 b.txt
drwxrwxrwx. 1 root root     0 Jan 30 02:23 .git
[student@localhost testmv] git add .
[student@localhost testmv] git commit -m "add a.txt b.txt"
[master (root-commit) 28e968b] add a.txt b.txt
 2 files changed, 2 insertions(+)
 create mode 100644 a.txt
 create mode 100644 b.txt
```

Let's do a mv and see what git says:

```
[student@localhost testmv] mv a.txt c.txt
[student@localhost testmv] git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    a.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        c.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Let's see what git will do:

```
[student@localhost testmv] git add .
[student@localhost testmv] git commit -m "mv a.txt c.txt"
[master 111b1b4] mv a.txt c.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename a.txt => c.txt (100%)
```

Hey! See that? git actually figured out that there should be a move from a.txt to c.txt!

2.15 Directories

Directories are just like regular files. Do this exercise on your own (solution is below) in a directory named `testdir`:

1. Create a directory `ham`.
2. In `ham`, create a text file `eggs.txt`
3. Now add `ham` and `eggs.txt` to your git repository.
4. Do a git status.
5. Now commit.

Recall that you can add multiple files using "`git add .`".

Do a second exercise:

1. Create several nested subdirectories in `ham` and add files to these directories
2. Commit.

Note that if you have creating a new directory with files in that new directory, when you do git status, it will only show you that the new directory

Once you are done with the above two exercises, remember to do the following to remove the above two commits before going on to the next section and remove `ham`:

```
git reset HEAD~2
rm -rf ham
```

(More on git reset in the next section.)

By the way, git does not add empty directories. So you must have a file in a directory.

Solution to above two exercises:

First I create the directories:

```
[student@localhost git] rm -rf testdir; mkdir testdir; mkdir testdir/ham
```

Then in `testdir` I do

```
[student@localhost testdir] git init
Initialized empty Git repository in home/student/testdir/.git/
[student@localhost testdir] echo "secret 1" > ham/eggs.txt
[student@localhost testdir] git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ham/
nothing added to commit but untracked files present (use "git add" to track)
```

Note that only the new directory is reported. The new file in the new directory is not reported. Now do:

```
[student@localhost testdir] git add ham ham/eggs.txt
[student@localhost testdir] git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   ham/eggs.txt
[student@localhost testdir] git commit -m "ham ham/eggs.txt"
[master (root-commit) 1f759bf] ham ham/eggs.txt
 1 file changed, 1 insertion(+)
 create mode 100644 ham/eggs.txt
```

```
[student@localhost testdir] mkdir ham/ham2; echo "secret 2" > ham/ham2/eggs2.txt
[student@localhost testdir] mkdir ham/ham3; echo "secret 3" > ham/ham3/eggs3.txt
[student@localhost testdir] mkdir ham/ham2/ham4
[student@localhost testdir] echo "secret 4" > ham/ham2/ham4/ham4.txt
[student@localhost testdir] git add .; git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   ham/ham2/eggs2.txt
        new file:   ham/ham2/ham4/ham4.txt
        new file:   ham/ham3/eggs3.txt
[student@localhost testdir] git commit -m "ham directory"
[master 3224c2c] ham directory
 3 files changed, 3 insertions(+)
 create mode 100644 ham/ham2/eggs2.txt
 create mode 100644 ham/ham2/ham4/ham4.txt
 create mode 100644 ham/ham3/eggs3.txt
```

2.16 Diff

I've already mentioned the diff utility in `unix1.pdf`. Here's an example: Save the following to two files names a.cpp and b.cpp:

```
// a.cpp
#include <ioseam>
int main()
{
    int x = 0;
    std::cout << x + 0.5 << '\n';
    return 0;
}
```

```
// b.cpp
#include <iostream>
int main
{
    double x = 0;
    std::cout << x + 0.5 << '\n';
    return 0;
}
```

Now in your bash shell run

```
[student@localhost git] diff a.cpp b.cpp
1,3c1,3
< // a.cpp
< #include <ioseam>
< int main()
---
> // b.cpp
> #include <iostream>
> int main
5c5
<     int x = 0;
---
>     double x = 0;
```

and you'll see what I mean. To understand the output of diff, you can google for an

introduction to diff although you can probably guess most of it.

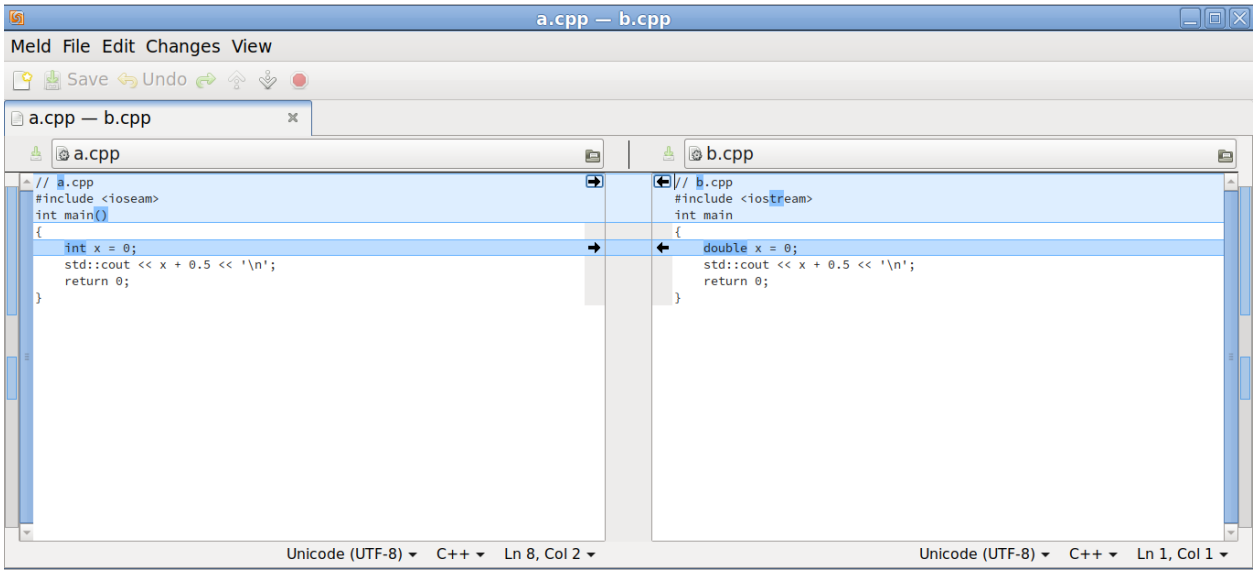
Other famous programs to check the difference between two files include

1. diff
2. kdiff3
3. meld

etc. Programmers when they want to compare between two files f1 and f2 will say they want to "diff f1 and f2". There are also many "visual" diff utilities. One of them is meld. Go ahead and run

```
[student@localhost git] meld a.cpp b.cpp
```

and you'll see this:



and use it to synch up the two files so that `a.cpp` and `b.cpp` are the same and valid C++ programs. After this experiment, delete `a.cpp` and `b.cpp`.

(Meld should be installed in our fedora virtual machine. If meld is not installed, as root, do `dnf install -y meld`.)

Suppose you just made a change a file and you were thinking to yourself that maybe the change was not a great idea. Now you don't want to checkout the last commit because if you do that it might wipe out made many other modifications which you don't want to lose. So you just want to peek at the previous state of the file from the last commit and make a comparison and make selective rollback.

Let's change `helloworld2.cpp` to this:

```
#include <iostream>

int main()
{
    std::cout << "Hello World\n";
    std::cout << "Goodbye World\n";

    return 0;
}
```

Now try this:

```
[student@localhost testgit] git diff helloworld2.cpp
diff --git a/helloworld2.cpp b/helloworld2.cpp
index 49e773f..9f6bc33 100644
--- a/helloworld2.cpp
+++ b/helloworld2.cpp
@@ -1,8 +1,9 @@
-// File: helloworld.cpp
 #include <iostream>

 int main()
 {
-    std::cout << "hello world!!!" << std::endl;
+    std::cout << "Hello World\n";
+    std::cout << "Goodbye World\n";
+
     return 0;
 }
\ No newline at end of file
```

Even better, you might want to use a *visual* diff to compare your working code with committed code. I'll show you how to use meld with git. First modify your `~/.gitconfig` by adding the following to the file:

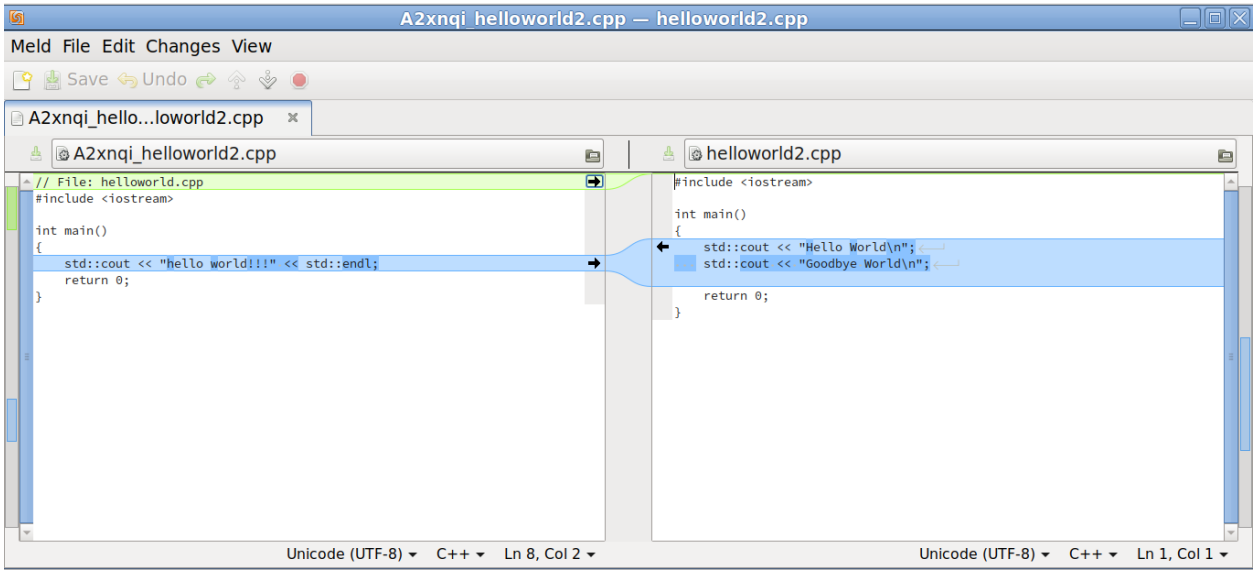
```
[diff]
        tool = meld
[difftool]
        prompt = false
[difftool "meld"]
        cmd = meld "$LOCAL" "$REMOTE"
[merge]
        tool = meld
```

```
[mergetool "meld"]
        cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

Now do

```
[student@localhost testgit] git difftool helloworld2.cpp
```

In meld, you'll see two windows.



On the right is your `helloworld2.cpp` in your current work directory and on the left is the file in the last commit corresponding to a previous version of `helloworld2.cpp`. You can merge the contents from the committed version to your working copy.

If you want to diff with a commit, you need to know the commit id or tag (if there's one). The command is

```
[student@localhost testgit] git difftool [commit id or tag] helloworld2.cpp
```

Note that because of the way git stores your files, if a file is not changed, it's not stored in that commit you are making. You might need to go to an earlier commit for the old version you are looking for.

Exercise 2.10. Create a directory for this exercise. And go into that directory.

1. Version the directory with git.
2. Commit 1: Add a hello world C++ program and a makefile and commit the files.

3. Commit 2: Make any change you like in your C++ source file and commit it.
4. Commit 3: “Accidentally” copy your makefile to your C++ source file. Commit it.
5. Go a diff of your makefile wth Commit 3. Any diff?
6. Go a diff of your C++ source file wth Commit 3. Any diff?
7. Going through all the commits (commit 3, 2, 1) until you arrive at the first commit. Copy the contents of the first hello world to the hello world in your working directory, test it, and commit it.

2.17 History of a file

In the last two subsection, I talked about recovered an earlier file after you made some modifications. You can look at the log and do a diff of a file in a commit with the corresponding file in your working directory.

There are two problems with that:

1. What about a deleted file? If you have deleted a file in your working directory, you can't diff that file!
2. If you have 1000 commits, it will take you a long time to find that particular file you are interested in!

You can actually filter your git log by filename. Let's try that.

Note that we have deleted out makefile. Let's look for it:

```
[student@localhost testgit] git log --oneline --decorate=full -- makefile
c0734ca (tag: refs/tags/v-9) helloworld2.cpp
8d5b3c3 rm makefile again
7edf12b makefile
37e5cb5 (tag: refs/tags/v-8) rm makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
e15eb3c (tag: refs/tags/v-2) makefile
```

This shows you commits that involve `makefile`. You can then look at the commit ids (or tags) and find your file for instance by doing checkout or a clone of the relevant commit. The relevant commit ids are:

```
7edf12b makefile
881fe4d (tag: refs/tags/v-5) helloworld! and add r for makefile
e15eb3c (tag: refs/tags/v-2) makefile
```

Let's do a checkout of the most recent and relevant commit id:

```
[student@localhost testgit] git checkout 7edf12b -- makefile
[student@localhost testgit] less makefile
# File: makefile
helloworld.exe: helloworld.cpp
        g++ helloworld.cpp -o helloworld.exe
clean:
        rm helloworld.exe
run:
        ./helloworld.exe
r:
        ./helloworld.exe
```

There you go.

2.18 Clone

In previous sections, we have been working with a single repository. In the section on checkout, we checkout a file from an earlier commit into our working directory. In the section on diff, we look at the diff of a file in our current directory with a committed file.

Now we are doing to work with two copies of the same git repo directory (not an individual file) in two different directories.

First let's create an empty directory called `testgit2`:

```
[student@localhost git] rm -rf testgit2; mkdir testgit2
```

Now we make a clone of `testgit` into `testgit2`:

```
[student@localhost git] git clone --no-hardlink testgit testgit2
Cloning into 'testgit2'...
done.
```

Let's compare their logs. Here is `testgit`'s log:

```
[student@localhost testgit] git log -3 --oneline --decorate=full
fc39ebd (HEAD -> refs/heads/master) add helloworld2.cpp, rm helloworld.cpp
9911e6a undo
c0734ca (tag: refs/tags/v-9) helloworld2.cpp
```

and here's the log for `testgit2`:

```
[student@localhost testgit2] git log -3 --oneline --decorate=full
fc39ebd (HEAD -> refs/heads/master, refs/remotes/origin/master, refs/remotes/ori
gin/HEAD) add helloworld2.cpp, rm helloworld.cpp
9911e6a undo
c0734ca (tag: refs/tags/v-9) helloworld2.cpp
```

More or less the same. (The log for `testgit2` has a bit more info after the HEAD.) Note that we are not copying files. The `testgit2` is actually a directory that is versioned. (It's actually also tied to `testgit`. I'll talk about that later.)

You can easily look through the various versions of your code in `testgit2`, checking out files from different versions without disrupting your `testgit`. For instance in the previous

section I mentioned you can do a diff to find a previous file that you want. Frequently making a clone and playing around with the clone to find a file is easier than to do git diff.

In fact later we'll see that a clone is useful not just for finding a file. It can be very useful for experimenting with your code and when you have finish the experiment in `testgit2`, you can combine your new code back into `testgit`. I'll talk about this later in Part 3. For now, think of clone as another way to look for a previous file (besides diff). After you found what you are looking for, you can merge the contents found to the relevant file in `testgit`. You can then delete `testgit2`.

Exercise 2.11. Create a git repo.

1. Commit 1: Add three files f, g, h and commit.
2. Commit 2: Make some modifications to f, add a new file i, delete g, and commit. Take note of the contents of f that you will be recovering later.
3. Commit 3: Make some modifications to f,h,i, add some new files, delete some files, and commit. Take note of the contents of h at this point.
4. Commit 4: Make some modifications, add some new files, delete some files including f, and commit.
5. Clone your git repo to another directory. In your clone, go through the commits backward in time until you find the the file f and h you wanted. Copy your f and h to your original repo and commit. Delete your clone.

Of course recovering the old f and h might require recovering or modifying other files related to f and h.

debug: summary

2.19 Summary

Installation

```
dnf -y install git-core meld
```

~/.gitconfig:

```
[user]
        email = jdoe@gmail.com
        name = John Doe
[core]
        editor = emacs
[diff]
        tool = meld
[difftool]
        prompt = false
[difftool "meld"]
        cmd = meld "$LOCAL" "$REMOTE"
[merge]
        tool = meld
[mergetool "meld"]
        cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

git command	
`git init`	add git repo to current directory
`git status`	compare working directory and repo
`git add .`	stage all changes from . recursively
`git add [file]`	stage [file] if it was added, deleted, or modified
`git rm [file]`	stage [file] and delete it
`git mv [file1] [file2]`	mv [file1] [file2] and stage it
`git commit -m 'a msg'`	commit changes recorded in staging area
`git tag 'a tag'`	add tag to last commit
`git log`	history of commits
`git ls-files`	ls on repo
`git checkout [file]`	copy [file] from last commit to working directory
`git checkout [id] [file]`	copy [file] of commit with commit [id] to working directory
`git checkout [tag] [file]`	copy [file] of commit with tag [tag] to working directory
`git diff [file]`	diff [file] of last commit with working copy
`git difftool [file]`	visual diff last commit of [file] with working copy
`git difftool [id] [file]`	visual diff [file] of commit [id] with working copy

`git difftool [tag] [file]`	visual diff commit [tag] of [file] with working copy
`git clone --no-hardlink [dir1] [dir2]`	clone [dir1] to [dir2]

3 Part 2: Github

3.1 Github

Now that you know the basics of git, it's time to work with a remote git repo.

In part 1, our git repository is in our local machine. In part 1, you have

1. a local working directory
2. a local git repo

You can work with a git repository that is remote. For this part, I'll be using github as our remote git repository. So for this part you will be working with

1. a local working directory
2. a local git repo
3. a remote git repo (at github)

The benefits of using a remote git repo are:

1. Your laptop might crash and you might lose you work unless you do backups regularly. Github will most likely (hopefuly) have better backups than what you are capable of.
2. It's easier to share your repository with others. For instance you might have an open source project and you want to share the code with others. Or maybe a hiring manager wants to see your work.
3. Besides sharing your repository with others, github also makes it easier for you to work with others collaboratively so that others can contribute code to your open source project.

3.2 Create github user and token

1. Go to http://github.com and create a user.
2. Login to github. Click on the top right button and pull down options and then click on Settings. On left sidebar click on Developer settings. On the left sidebar click on Personal access token. Click on Generate new token. Remember the token. The token is like your password when you use your command line git commands to connect to github.

3.3 Store user info

Remember earlier you entered your info and your favorite editor into `.gitconfig` by doing:

```
git config --global user.name "Yihsiang Liow"
git config --global user.email "yliow@ccis.edu"
git config --global core.editor emacs
```

Whenever you connect to github, you will be asked your username and token. If you don't mind that, go to next subsection. Otherwise continue reading.

In your bash shell install the following (as root)

```
dnf install git-credential-libsecret
```

Next add the following to the file `~/.gitconfig`:

```
[credential]
        helper = /usr/libexec/git-core/git-credential-libsecret
```

(The indentation is a tab character.) You will be asked your token once. After entering your token, it will be remembered.

3.4 Begin versioning a project directory on github

Suppose you already have a project directory and you want to create a git repo for that project at github. Do the following:

1. In your bash shell, goto your project directory. Suppose the directory name is `xyz`.
2. Open browser and login to github's website.
3. On the left of browser, look for "Repositories" and click on "New".
4. In browser, enter "Repository name" (example: `xyz`), select private or public, click on "Create repository".
5. In browser, look for "... or create a new repository on the command line" and copy the commands. It will look something like this:

```
echo "# xyz" >> README.md
git init
... [snip] ...
git push -u origin main
```

   In your bash shell, paste the commands you have just copied. You'll see something like this (for password, enter your token and not your user password):

```
[student@localhost xyz]$ echo "# xyz" >> README.md
[student@localhost xyz]$ git init
Initialized empty Git repository in /home/student/shares/yliow/Documents/work/projects/xyz/.git/
... [snip] ...
[student@localhost xyz]$ git push -u origin main
Username for 'https://github.com': yliow@ccis.edu
Password for 'https://yliow@ccis.edu@github.com':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 210 bytes | 52.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/yliow/xyz.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

6. Go to your browser and check that your files for `xyz` are in the `xyz` repo in your github account.

3.5 Download (clone) a github repo

Suppose you want to work on a git repo. (This can be a project that belongs to you, but you have deleted the project from your laptop. Or it can be a git repo that belongs to someone else and you have access to it.)

1. In your bash shell, execute (replacing `yliow` with the correct username and `xyz` with the repo name):

   ```
   git clone https://github.com/yliow/xyz
   ```

   and you'll see something like this:

   ```
   Cloning into 'xyz'...
   warning: redirecting to https://github.com/yliow/xyz/
   ... snip ...
   Unpacking objects: 100% (3/3), 190 bytes | 7.00 KiB/s, done.
   ```

2. Check that you now have a directory named `xyz` holding the repo.

3.6 Pull and push

When someone has modified the remote git repo, you can update your local copy with the changes by doing

```
git pull
```

If you have committed changes to your local repo and you want to update the remote git repo, you do

```
git push
```

That's it. With the git commands from Part 1, you now know how to work with a remote git repo:

1. You write code in your local working directory.
2. You commit your local working directory changes to your local git repo.
3. You synchronize your local git repo with the remote git repo using git pull and git push.

3.7 Conflicts

If when you do `git pull` and get an error message such as this:

```
[student@localhost xyz]$ git pull
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

it means that your version of `main.cpp` cannot be merged the version in github, i.e., most likely someone else has made changes to it after your last git pull. If you open `main.cpp`, you will see something like the following somewhere in your `main.cpp`:

```
<<<<<<< HEAD
    THIS IS STUFF THAT
    YOU WROTE
=======
    THIS IS STUFF PULLED
    DOWN FROM GITHUB
>>>>>>> 3f5c9b931d7138e697701ce4686d1ce2a88c36f0
```

You have to manually merge the above maybe to this:

```
    THIS IS STUFF THAT
    YOU WROTE
    THIS IS STUFF PULLED
    DOWN FROM GUTHUB
```

(or maybe you need to delete some of your stuff or delete some stuff from the downloaded content). After saving the changes, you can try to do git add-commit-push.

3.8 Experiment on pull conflict

Let's do an experiment. Open two bash shells.

- In the first shell, pretend you are John Doe.
- In the second shell, pretend you are Jane Smith.

STEP 1: JOHN DOE (FIRST SHELL). In the first shell, John Doe (you) creates a directory `testgit3` and in that directory create a file named `main.txt`:

```
john doe: this is line 1
```

Go to your github account and create a repo named testgit3. Copy and paste the commands from github to your bash shell. Your directory `testgit3` is now in github. Do git add-commit-push your contents to github:

```
[student@localhost testgit3]$ git add .
[student@localhost testgit3]$ git commit -m 'v-1'
[student@localhost testgit3]$ git push
```

(Outputs of git not shown.) Your testgit3 project is now versioned in your github repo testgit3.

STEP 2: JANE SMITH (SECOND SHELL). Now in the second shell, Jane Smith (you), creates a directory `testgit3-another` and go into that directory. In this second bash shell, clone the testgit3

```
[student@localhost testgit3-another]$ git clone http://github.com/yliow/testgit3
[student@localhost testgit3-another]$ ls
testgit3
[student@localhost testgit3]$ cd testgit3/
[student@localhost testgit3]$ ls
main.txt  README.md
```

(Git outputs not shown.) Now Jane Smith has the same testgit3 repo as John Doe.

You now have two copies of the testgit3 repo at github:

- John Doe: In directory `testgit3`
- Jane Smith: In directory `testgit3-another/testgit3`

Check that the `main.txt` are the same for both directories.

STEP 3: JANE SMITH (SECOND SHELL). In the second bash shell, Jane Smith modifies `testgit3-another/testgit3/main.txt` to get this:

```
john doe: this is line 1
jane smith: this is line 2
```

and push the changes to the testgit3 in github by doing git add-commit-push:

```
[student@localhost testgit3]$ git add .
[student@localhost testgit3]$ git commit -m 'v-2'
[student@localhost testgit3]$ git push
```

John Doe (first bash shell) is now going to update this copy:

```
[student@localhost testgit3]$ git pull
```

and his main.txt looks exactly like the one updated by Jane Smith.

STEP 4: JOHN DOE (FIRST SHELL). Now John Doe updates main.txt like this:

```
john doe: this is line 1 ... new stuff
jane smith: this is line 2
```

and pushes his contents to github using git add-commit-push.

STEP 5: JANE SMITH (SECOND SHELL). Jane Smith updates her main.txt to

```
john doe: this is line 1
jane smith: this is line 2 ... brand new stuff
```

Now when Jane Smith tries to do git add-commit-push, she gets an error:

```
[student@localhost testgit3]$ git add .
[student@localhost testgit3]$ git commit -m 'v-3'
[main 69e6ce7] v-3
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testgit3]$ git push
warning: redirecting to https://github.com/yliow/testgit3/
To http://github.com/yliow/testgit3
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'http://github.com/yliow/testgit3'
hint: Updates were rejected because the remote contains work that you do
```

```
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

She realized John Doe has changed the main.txt file. She does a git pull:

```
[student@localhost testgit3]$ git pull
warning: redirecting to https://github.com/yliow/testgit3/
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 298 bytes | 9.00 KiB/s, done.
From http://github.com/yliow/testgit3
   afe11be..d4fb960  main       -> origin/main
Auto-merging main.txt
CONFLICT (content): Merge conflict in main.txt
Automatic merge failed; fix conflicts and then commit the result.
```

She sees there's a conflict in main.txt, opens main.txt, and sees this:

```
<<<<<<< HEAD
john doe: this is line 1
jane smith: this is line 2 ... brand new stuff
=======
john doe: this is line 1 ... new stuff
jane smith: this is line 2
>>>>>>> d4fb960b979b779d04febf9ae6f8ebf810e0d25f
```

She carefully merge the two changes by hand to get this:

```
john doe: this is line 1 ... new stuff
jane smith: this is line 2 ... brand new stuff
```

She then do git add-commit-push:

```
[student@localhost testgit3]$ git add .
[student@localhost testgit3]$ git commit -m 'v-4'
[main d60c34b] v-4
[student@localhost testgit3]$ git push
```

This time there's no error.

STEP 6: JOHN DOE (FIRST SHELL). On John Doe's end, when he git-pulls, he gets an update and when he open main.txt, he has the same file as the one Jane Smith git-pushed.

3.9 HEAD

(In the example below, I'm going to use a local repo. But the idea applies to remote repo at github too.)

Recall that we can refer to a commit by the commit id or tag. For instance

```
git checkout [commit id or tag] [file]
git diff [commit id or tag] [file]
git difftool [commit id or tag] [file]
```

And if the commit id or tag is not specified, then the last commit is used.

There's another way to refer to commits.

The last commit is called `HEAD`. The one before `HEAD` is `HEAD~1`. And the one before `HEAD~1` is `HEAD~2`. And the one before `HEAD~2` is `HEAD~3`. Etc.

Let's do an experiment in directory `testhead`:

```
[student@localhost projects] rm -rf testhead; mkdir testhead
```

We'll create 4 commits of a text file `a.txt`:

```
[student@localhost testhead] git init
Initialized empty Git repository in /home/student/shares/yliow/Documents/work/pr
ojects/testhead/.git/
[student@localhost testhead] echo "1" > a.txt; git add .; git commit -m "1"
[master (root-commit) e7e81d9] 1
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
[student@localhost testhead] echo "2" > a.txt; git add .; git commit -m "2"
[master c5c9d57] 2
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testhead] echo "3" > a.txt; git add .; git commit -m "3"
[master cbb0cf8] 3
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testhead] echo "4" > a.txt; git add .; git commit -m "4"
[master 513107e] 4
 1 file changed, 1 insertion(+), 1 deletion(-)
```

At the first commit, `a.txt` contains `"1"`. At the second commit, `a.txt` contains `"2"`. Etc.

Now let's checkout all the 4 versions:

```
[student@localhost testhead] git checkout HEAD a.txt
Updated 0 paths from 9d52192
[student@localhost testhead] less a.txt
4
[student@localhost testhead] git checkout HEAD~1 a.txt
Updated 1 path from 8aac067
[student@localhost testhead] less a.txt
3
[student@localhost testhead] git checkout HEAD~2 a.txt
Updated 1 path from 7b1ecdb
[student@localhost testhead] less a.txt
2
[student@localhost testhead] git checkout HEAD~3 a.txt
Updated 1 path from 02bdfa7
[student@localhost testhead] less a.txt
1
```

See that?

So in this context, the `HEAD` makes it easy to refer to “the last commit”, “the previous to last commit”, etc.

3.10 Workflow 1

Suppose you are working on github repo `xyz`.

1. Decide what you want to implement.
2. If you don't have a download (clone) of `xyz`, do git clone.
3. Loop:
   a) git pull
   b) resolve conflicts
   c) git add/rm/mv and git commit
   d) write code and test

   You want to pull and commit as frequently as you can.
4. When you are done with what you want to implement:
   a) git pull
   b) resolve conflicts
   c) git add/rm/mv and git commit
   d) git push

Exercise 3.1. Write a C++ program that plays tic-tac-toe, but do it pretending you are John Doe and Jane Smith. Every time you (Joe Doe or Jane Smith) have something that compiles and run, commit. Alternate between John Doe and Jane Smith on the following steps. Feel free to create some conflicts.

1. The program prints the game title. Test. If it works – commit.
2. The program prints the game title using a function. Test. If it works – commit.
3. The program initializes a blank board and prints the board. Test. If it works – commit.
4. Continuing, the program initializes a blank board using a function and prints the board. Test. If it works – commit.
5. The program initializes a blank board using a function and prints the board using a function. Test. If it works – commit.
6. Continuing, the program gets a move from X, assumes it is correct, and makes the move for X and prints the board. Test. If it works – commit.
7. Continuing, the program continually gets a move from X until a correct move is made within the boundary of the board, makes the move for X and prints the board. Test. If it works – commit.
8. The program continually gets a move from X until a correct move is made within the boundary of the board and the square taken is available, makes the move for X and prints the board. Test. If it works – commit.
9. Etc.

3.11 Private repo – invite collaborators

If you have created a private repo, you can invite people to be collaborators. Here's what to do:

1. Go to repo on github.
2. Click on Settings (near top-right).
3. Click on Manage access (left).
4. Click on Invite a collaborator (middle).
5. Enter collaborator username or email.
6. Tell the collaborator that you have added him/her to your repo.

debug: summary

3.12 Summary 2

Installation

```
dnf -y install git-core git-credential-libsecret meld
```

~/.gitconfig:

```
[user]
        email = jdoe@gmail.com
        name = John Doe
[credential]
        helper = /usr/libexec/git-core/git-credential-libsecret
[core]
        editor = emacs
[diff]
        tool = meld
[difftool]
        prompt = false
[difftool "meld"]
        cmd = meld "$LOCAL" "$REMOTE"
[merge]
        tool = meld
[mergetool "meld"]
        cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

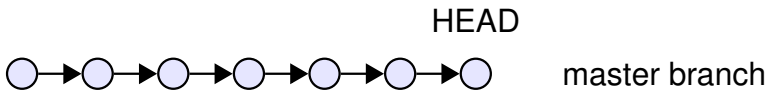
git command	
`git init`	add git repo to current directory
`git status`	compare working directory and repo
`git add [file]`	stage [file] because it was added, deleted, or modified
`git rm [file]`	stage [file] and delete it
`git mv [file1] [file2]`	mv [file1] [file2] and stage it
`git commit -m á msg´`	commit changes recorded in staging area
`git tag á tag´`	add tag to last commit
`git log`	history of commits
`git ls-files`	ls on repo
`git checkout [file]`	copy latest version of [file] to working directory
`git checkout [id] [file]`	copy [file] of version with commit [id] to working directory
`git checkout [tag] [file]`	copy [file] of version with tag [tag] to working directory
`git diff HEAD [file]`	diff last commit of [file] with working copy
`git difftool HEAD [file]`	visual diff last commit of [file] with working copy
`git difftool [id] [file]`	visual diff commit [id] of [file] with working copy
`git difftool [tag] [file]`	visual diff commit [tag] of [file] with working copy
`git clone --no-hardlink [dir1] [dir2]`	clone [dir1] to [dir2]

4 Part 3: Branching

4.1 Branches

(In the example below, I'm going to use a local repo. But the idea applies to remote repo at github too.)

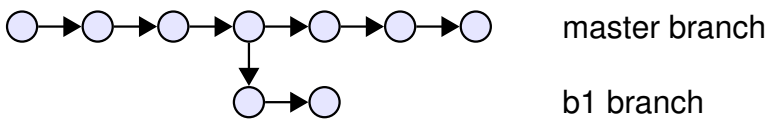
Here's a picture of all your commits for a project:



You can think of a branch as a sequence of commits. There's only one branch now – the master branch. The default branch is either master or main (see below).

Recall from an earlier section that the last commit on the master branch is the `HEAD`. (If you like you can think of `HEAD` as a pointer that points to a commit. You can make `HEAD` point to other commits. I'll talk about that later.)

You can actually create branches from your master branch. Here's an example where there's a branch called `b1`:



You can create branches out of any branch.

What's the point of branches?

Suppose you are working on a project. On Monday, everything in your code is working fine, although it's only half done. You go to your project directory and start work on the next feature. If the code you need to write is not too difficult and especially if this is your own personal project, it's OK to stick to the master branch of commits.

However when you are on a team project, it's possible that that John is working on a part of the project and might be temporarily inserting bugs into a commit in the master branch. And you might be doing that too. So you might want to create a branch and John might want to create his branch so that both of you can work individually on your branch. And when you are done with the new feature implementation on your branch, you put all the new stuff back into the master branch. Even if it's a personal project, you might want to use branches too.

```
[student@localhost projects] rm -rf testbranch; mkdir testbranch
```

We'll create 4 commits of a text file `a.txt`:

```
[student@localhost testbranch] git init
Initialized empty Git repository in /home/student/shares/yliow/Documents/work/pr
ojects/testbranch/.git/
[student@localhost testbranch] echo "1" > a.txt; git add .; git commit -m "1"
[master (root-commit) 6de0f66] 1
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
[student@localhost testbranch] echo "2" > a.txt; git add .; git commit -m "2"
[master 37f903a] 2
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "3" > a.txt; git add .; git commit -m "3"
[master 94c58ac] 3
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "4" > a.txt; git add .; git commit -m "4"
[master 472d633] 4
 1 file changed, 1 insertion(+), 1 deletion(-)
```

At the first commit, `a.txt` contains `"1"`. At the second commit, `a.txt` contains `"2"`. Etc. If I do a git status, I get:

```
[student@localhost testbranch] git status
On branch master
nothing to commit, working tree clean
```

Notice that the name of this branch is `master`. See that?

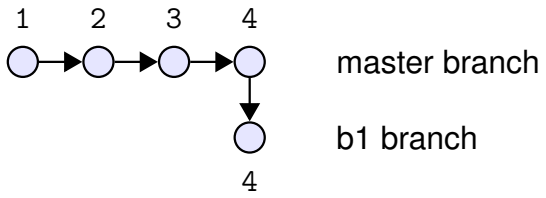
Now I'm going to create a branch called `b1`:

```
[student@localhost testbranch] git checkout -b b1
Switched to a new branch 'b1'
```

Now I'm working on branch `b1`. Wait ... notice that I did not specify which commit (of the four) that I want to branch out from. So what is the `a.txt` now? Let's see:

```
[student@localhost testbranch] less a.txt
4
```

It's the last commit of the master branch. (Duh). Here's a picture:



(Actually when you create a new branch, there is no new commit. So the `4` that is below is actually the same as the `4` that is above.) In the picture, the text near a commit (i.e., the node in the picture) is the content of the file `a.txt` in that commit.

Let's take a look at all branches:

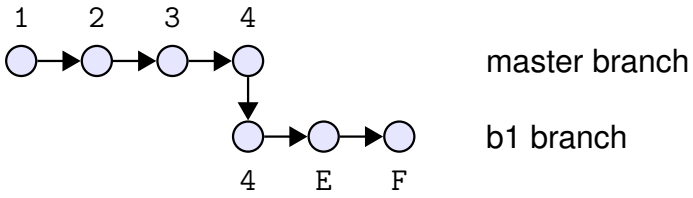
```
[student@localhost testbranch] git branch -a
* b1
  master
```

See that? The `*` indicates you are working on `b1`.

Now I'm going to make two commits in `b1`:

```
[student@localhost testbranch] echo "E" > a.txt; git add .; git commit -m "E"
[b1 de43268] E
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "F" > a.txt; git add .; git commit -m "F"
[b1 23064b2] F
 1 file changed, 1 insertion(+), 1 deletion(-)
```

So the picture is now



Of course it's not surprising that you can switch between branches. Here's going back to `master` and verifying `a.txt` is `"4"`:

```
[student@localhost testbranch] git checkout master
Switched to branch 'master'
[student@localhost testbranch] less a.txt
4
```

And then to `b1`:

```
[student@localhost testbranch] git checkout b1
Switched to branch 'b1'
```

If I go to `b1` again, I get

```
[student@localhost testbranch] git checkout b1
Already on 'b1'
```

Now I'm going to merge the last commit of `b1` back into the last commit of `master`. First go to `master` branch and then execute a git merge with `b1`:

```
[student@localhost testbranch] git checkout master
Switched to branch 'master'
[student@localhost testbranch] git merge b1
Updating 472d633..23064b2
Fast-forward
 a.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Let's check:

```
[student@localhost testbranch] git status
On branch master
nothing to commit, working tree clean
[student@localhost testbranch] less a.txt
F
```

OK. We are now in the master branch and `a.txt` has content `"F"`. Here are all the branches again:

```
[student@localhost testbranch] git branch -a
  b1
* master
```

`b1` is still around. Assuming you don't need `b1` anymore, do this:

```
[student@localhost testbranch] git branch -d b1
Deleted branch b1 (was 23064b2).
[student@localhost testbranch] git branch -a
* master
```

Now the merge is not just merging the file(s) of `b1` into `master`. It actually merges the commits of `b1` into `master` as well:

```
[student@localhost testbranch] git log
commit 23064b21a4c699ccf6f51bcdb60ba28dfa0fa754
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:24:00 2025 -0500
    F
commit de43268308153a2429d24caac7d2743c2b50ebfa
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:24:00 2025 -0500
    E
commit 472d63389177ac383c66ba545024e40fc41bed66
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:23:54 2025 -0500
    4
commit 94c58acb5501b373cd0d6f6f776825901d750677
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:23:54 2025 -0500
    3
commit 37f903a4873255ddcab5a5766b996143e1d0a454
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:23:54 2025 -0500
    2
commit 6de0f66e7974ff662b03f2244dd6aa3c680212ad
Author: Yihsiang Liow <yliow@ccis.edu>
Date:   Thu Jan 30 02:23:54 2025 -0500
    1
```

See that?

4.2 Branch merge conflict

(In the example below, I'm going to use a local repo. But the idea applies to remote repo at github too.)

Of course there are times when it's impossible to merge. Try the following. I clear my test directory:

```
[student@localhost projects] rm -rf testbranch; mkdir testbranch
```

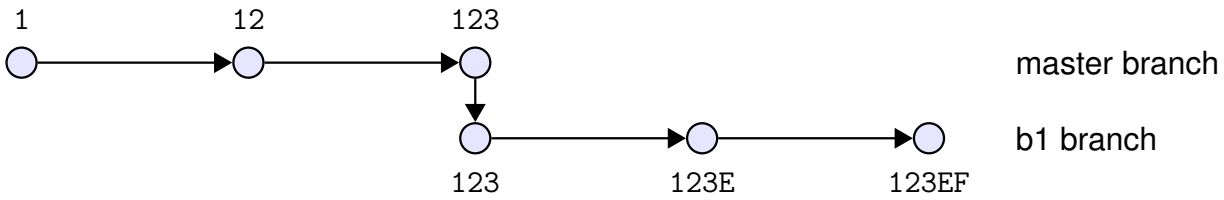
and then create a repo with `a.txt` with several edits and commits:

```
[student@localhost testbranch] git init
Initialized empty Git repository in /home/student/shares/yliow/Documents/work/pr
ojects/testbranch/.git/
[student@localhost testbranch] echo "1" > a.txt
[student@localhost testbranch] git add .; git commit -m "1"
[master (root-commit) ebc0bca] 1
 1 file changed, 1 insertion(+)
 create mode 100644 a.txt
[student@localhost testbranch] echo "12" > a.txt
[student@localhost testbranch] git add .; git commit -m "12"
[master 11c37cf] 12
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "123" > a.txt
[student@localhost testbranch] git add .; git commit -m "123"
[master 57504d2] 123
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now I create a branch and make some changes and commits:

```
[student@localhost testbranch] git checkout -b b1
Switched to a new branch 'b1'
[student@localhost testbranch] echo "123E" > a.txt
[student@localhost testbranch] git add .; git commit -m "123E"
[b1 0670bc3] 123E
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "123EF" > a.txt
[student@localhost testbranch] git add .; git commit -m "123EF"
[b1 7017ce0] 123EF
 1 file changed, 1 insertion(+), 1 deletion(-)
```

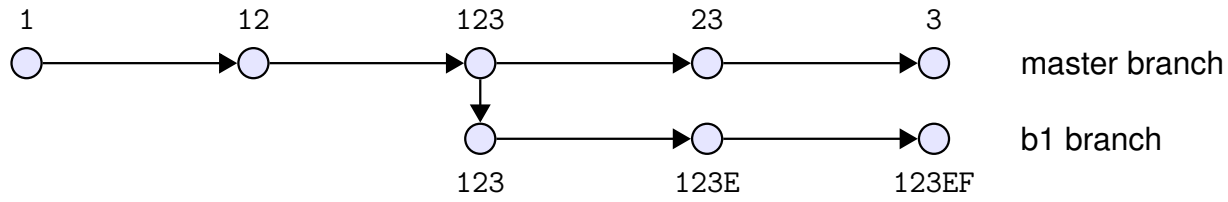
At this point we have this picture:



Now I'm going back to `master` and add more commits:

```
[student@localhost testbranch] git checkout master
Switched to branch 'master'
[student@localhost testbranch] echo "23" > a.txt
[student@localhost testbranch] git add .; git commit -m "23"
[master e2be059] 23
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@localhost testbranch] echo "3" > a.txt
[student@localhost testbranch] git add .; git commit -m "3"
[master 8e1586a] 3
 1 file changed, 1 insertion(+), 1 deletion(-)
```

We now have this picture:



Now when you try to merge:

```
[student@localhost testbranch] git checkout master
Already on 'master'
[student@localhost testbranch] git merge b1
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

```
[student@localhost testbranch] git checkout master
a.txt: needs merge
error: you need to resolve your current index first
[student@localhost testbranch] git merge b1
error: Merging is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
```

you'll get a conflict just like in subsection 3.7 (page 57). So you should know what to do. Here's `a.txt`:

```
<<<<<<< HEAD
3
```

```
=======
123EF
>>>>>>> b1
```

HEAD is of course pointing to the last commit of a.txt in master, and it has content "3" (as shown above). The last commit of b1 for a.txt, which has content "123EF" (as shown above). Of course this conflict won't occur if the master branch did not create conflicts while b1 is changing. So you have to figure out what you want.

Most likely the intention is to have this a.txt:

```
3EF
```

Edit a.txt accordingly:

```
[student@localhost testbranch] echo "3EF" > a.txt
```

When you execute git status you'll see some instructions/help:

```
[student@localhost testbranch] git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   a.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Do a git commit:

```
[student@localhost testbranch] git add .; git commit -m "merge b1"
[master 06d5779] merge b1
```

and delete branch b1:

```
[student@localhost testbranch] git branch -d b1
Deleted branch b1 (was 7017ce0).
```

That's it.

The difference between this scenario and the scenario in the previous section is that for this section the master/main has new commits after the branch was created. The previous merge (no new commits in master/main) is called fast-forward merge.

Exercise 4.1. Redo the C++ program that plays tic-tac-toe (from earlier) using branches.

Exercise 4.2. In a project that involves multiple programmers, why is fast-forward frequently impossible if programmers do this: "branch, complete code in branch, merge branch back to master".

Exercise 4.3. Can you do it the other way around and merge master/main into a branch?

Exercise 4.4. You can *test* if there's going to be a merge conflict. Try "`git merge --no-commit b1`".

Exercise 4.5. You are working on a program with Tom and Sue. There will be lots of tiny features to commit. What do you think is a good workflow that involves branches? (Of course you all want to minimize merge conflicts.)

4.3 Workflow 2

Suppose you are working on github repo `xyz`.

1. Decide what you want to implement.
2. If you don't have a download (clone) of `xyz`, do git clone.
3. a) git pull
   b) resolve conflicts
   c) git add/rm/mv and git commit
   d) create a branch
      i. git pull
      ii. resolve conflicts
      iii. git add/rm/mv and git commit
      iv. write code and test on the branch

   You want to pull and commit as frequently as you can.
4. When you are done with what you want to implement:
   a) switch to master/main, merge branch back to master/main, resolve conflicts if necessary
   b) git pull
   c) resolve conflicts
   d) git add/rm/mv and git commit
   e) git push

4.4 Summary 3

Installation

```
dnf -y install git-core git-credential-libsecret meld
```

~/.gitconfig:

```
[user]
        email = jdoe@gmail.com
        name = John Doe
[credential]
        helper = /usr/libexec/git-core/git-credential-libsecret
[core]
        editor = emacs
[diff]
        tool = meld
[difftool]
        prompt = false
[difftool "meld"]
        cmd = meld "$LOCAL" "$REMOTE"
[merge]
        tool = meld
[mergetool "meld"]
        cmd = meld "$LOCAL" "$MERGED" "$REMOTE" --output "$MERGED"
```

git command	
`git init`	add git repo to current directory
`git status`	compare working directory and repo
`git add [file]`	stage [file] because it was added, deleted, or modified
`git rm [file]`	stage [file] and delete it
`git mv [file1] [file2]`	mv [file1] [file2] and stage it
`git commit -m á msg´`	commit changes recorded in staging area
`git tag á tag´`	add tag to last commit
`git log`	history of commits
`git ls-files`	ls on repo
`git checkout [file]`	copy latest version of [file] to working directory
`git checkout [id] [file]`	copy [file] of version with commit [id] to working directory
`git checkout [tag] [file]`	copy [file] of version with tag [tag] to working directory
`git checkout -b [branch]`	create a new branch named [branch]
`git checkout [branch]`	switch to branch [branch]
`git diff HEAD [file]`	diff last commit of [file] with working copy
`git difftool HEAD [file]`	visual diff last commit of [file] with working copy
`git difftool [id] [file]`	visual diff commit [id] of [file] with working copy
`git difftool [tag] [file]`	visual diff commit [tag] of [file] with working copy
`git clone --no-hardlink [dir1] [dir2]`	clone [dir1] to [dir2]
`git branch -a`	list all branches
`git branch -d [branch]`	remove branch [branch]
`git merge [branch]`	merge [branch] to current branch

5 Miscellaneous tasks

Generally speaking there are so many possible scenarios in maintaining a repository that it's impossible to write a document that covers all of them. The following are some scenarios. You can of course use google if your specific scenario is not listed below.

Q: How do I recover a deleted file if I don't know when in the history of the repo it was deleted?
A: Suppose the file is `a/b/c/d/helloworld.txt`. Do this

```
git rev-list -n 1 HEAD -- a/b/c/d/helloworld.txt
```

and look for the last commit of `a/b/c/d/helloworld.txt` which must have cause it to be deleted. You'll need the SHA of that last commit.

```
git checkout [SHA of last commit]^ -- <file_path>
```

The ^ refers to the previous to the last commit.

Q: How do I completely clear all history of a repo and start all over again?
A: In most cases you do want to keep all the history of a project. However in some cases you are very certain that the the history is redundant. In that case, especially when the repo is huge because of age of the project, you might want to clear the history. Do the following:

```
rm -rf .git
git init
git add .
git commit -m "Initial commit"
git remote add origin <github-url>
git push -u --force origin master
```

Q: How do I completely remove a file and its history?
A: If you are certain that a file is redundant and you want to remove it (for instance the file is a huge image/video file), you do the following:

```
git clone REPO_LOCATION
git remote rm origin
git filter-branch \
  --index-filter 'git rm --cached --ignore-unmatch FILENAME' HEAD \
  git reflog expire --expire=now --all && git gc --prune=now --aggressive
rm -rf .git
git init
git add .
git commit -m "Initial commit"
git remote add origin <github-url>
git push -u --force origin master
```

Replace the centralized repo (aka REPO LOCATION) and any copies (cloud or otherwise) others might be using. Everyone should re-clone the newly replaced repo.

Q: How do I remove a huge file that was wrongly committed? I get this error message:

```
remote: error: File [...] is [...] MB; this exceeds GitHub's file size
limit of 100.00 MB
```

A:

```
git filter-branch --tree-filter 'rm -rf [path/to/your/file]' HEAD
git push
```

Q: git keeps saying a file is too large to commit, but the file was already deleted.
A: The file is still in history. Do git status. You might get

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Look at the number 2. Your number might be different. Now go back by 2 commits (or whatever is your number).

```
git reset HEAD~2
```

Now commit whatever you have:

```
git add .
git commit -m 'message'
git push
```

You can also try

```
git filter-branch --force --index-filter \
'git rm --cached --ignore-unmatch [path to large file]' \
--prune-empty --tag-name-filter cat -- --all
```

6 GUI clients

6.1 git-cola

- installation: `dnf -y install git-cola`
- command to run: `git-cola`
- windows? yes
- need git installation: yes

6.2 git-desktop

- installation:
- command to run:
- windows? yes
- need git installation: no