

CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: prabhakarsucharitha@gmail.com

Outline

- Software version management
- Git
- Lab Assignment

References:

<https://git-scm.com/>



Software development process

- Involves making a lot of changes to code
 - New features added
 - Bugs fixed
 - Performance enhancements
- Software team has many people working on the same/different parts of code
- Many versions of software released
 - Ubuntu 10, Ubuntu 12, etc
 - Need to be able to fix bugs for Ubuntu 10 for customers using it, even though you have shipped Ubuntu 12.

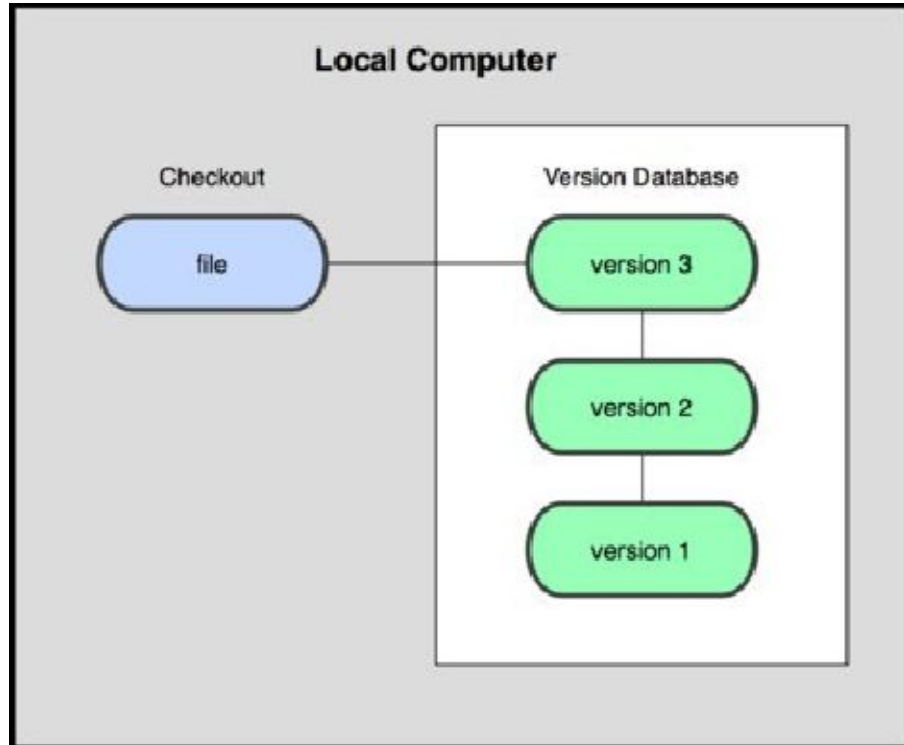


Version control

- Track changes to code and other files related to the software
 - What new files were added?
 - What changes made to files?
 - Which version had what changes?
 - Which user made the changes?
- Track entire history of the software
- Version control software
 - Examples: GIT, Subversion, Perforce



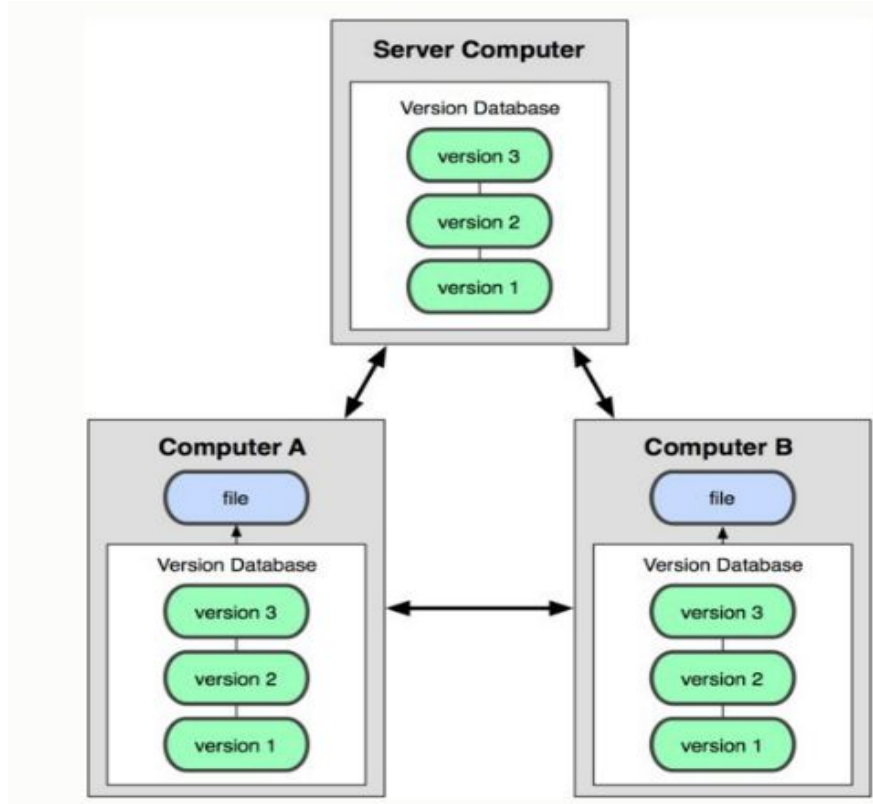
Local VCS



- Organize different versions as folders on the local machine
- No server involved
- Other users should copy it via disk/network

Image Source: git-scm.com

Distributed VCS



- Version history is replicated at every user's machine
- Users have version control all the time
- Changes can be communicated between users
- Git is distributed

Technical Terms

- Repository
 - Files and folder related to the software code
 - Full History of the software
- Working copy
 - Copy of software's files in the repository
- Check out
 - To create a working copy of the repository
- Check in/ Commit
 - Write the changes made in the working copy to the repository
 - Commits are recorded by the VCS



Source Control with GIT

Git States

Local Operations

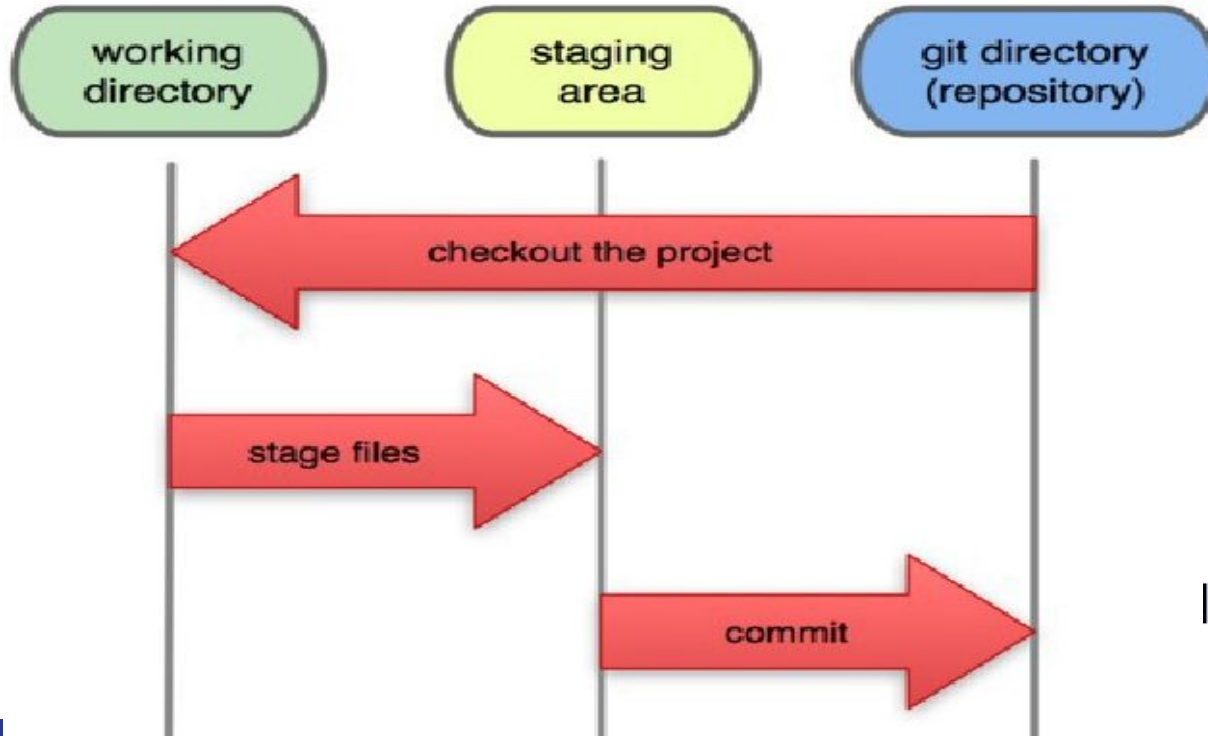


Image Source: git-scm.com

First Git Repository

- `mkdir gitroot`
- `cd gitroot`
- `git init`
 - creates an empty git repo (.git directory with all necessary subdirectories)
- `echo "Hello World" > hello.txt`
- `git status`
- `git add .`
 - Adds content to the index
 - Must be run prior to a commit
- `git commit -m 'Check in number one'`



Git commands

- `echo "I love Git" >> hello.txt`
- `git status`
 - Shows list of modified files
- `git diff`
 - Shows changes we made compared to index
- `git add hello.txt`
- `git diff HEAD`
 - Now we can see changes in working version
- `git commit -m "Second commit"`



Git commands

- Create Repo
 - `git init` (Create a new repo)
 - `git clone` (Create copy of existing repo)
- Branching
 - `git checkout <tag/commit> -b <new_branch_name>` (creates a new branch)
- Commits
 - `git add` (Stage modified/new files)
 - `git commit` (check-in the changes to the repository)



Git commands

- Getting info
 - git status (Shows modified files, new files, etc)
 - git diff (compares working copy with staged files)
 - git log (Shows history of commits)
 - git show (Show a certain object in the repository)
- Getting help
 - git help



Git Merge

A---B---C topic
/
D---E---F---G master

A---B---C topic
/ \
D---E---F---G---H master

Join two or more development histories

Merging hotfix branch into master

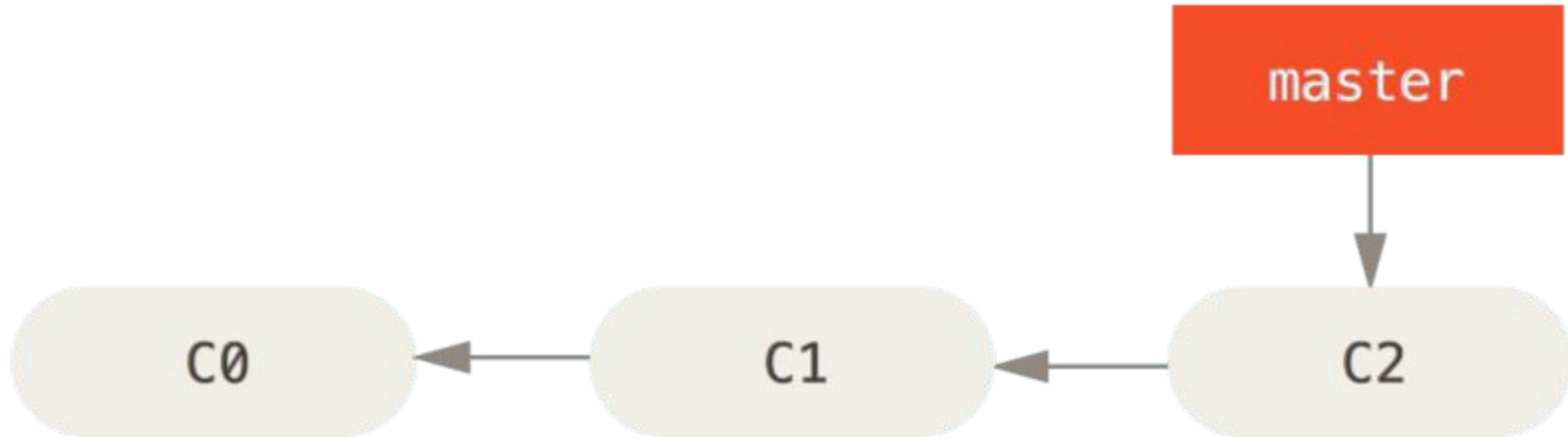
- git checkout master
- git merge topic
- Git tries to merge automatically

Simple if its a forward merge

Otherwise, you have to manually resolve conflicts

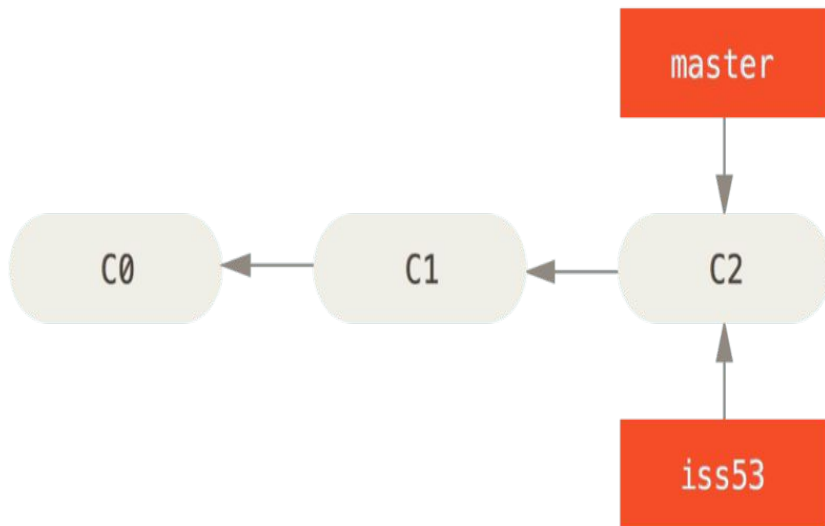
Image Source: git-scm.com

Branching and merging



A simple commit history

Branching and merging

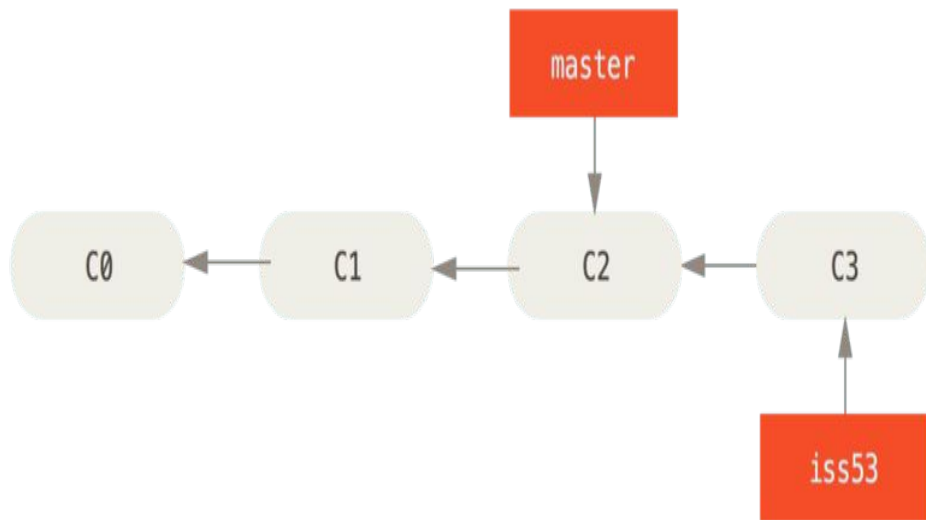


`$ git checkout -b iss53`
Switched to a new branch "iss53"

This is shorthand for:

`$ git branch iss53`
`$ git checkout iss53`

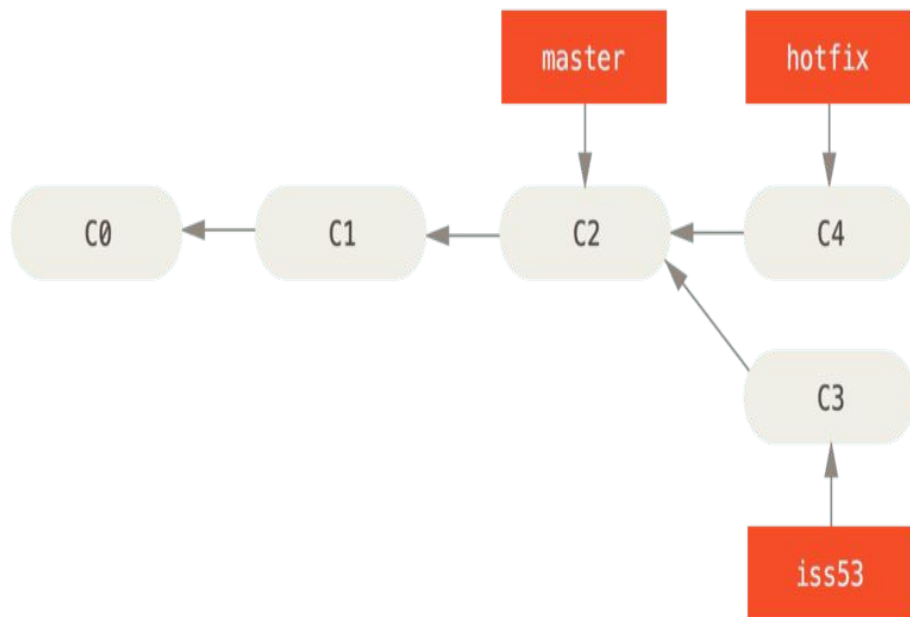
Branching and merging



`vim index.html`

`git commit -a -m 'added a new footer [issue 53]'`

Branching and merging



```
$ git checkout -b hotfix
```

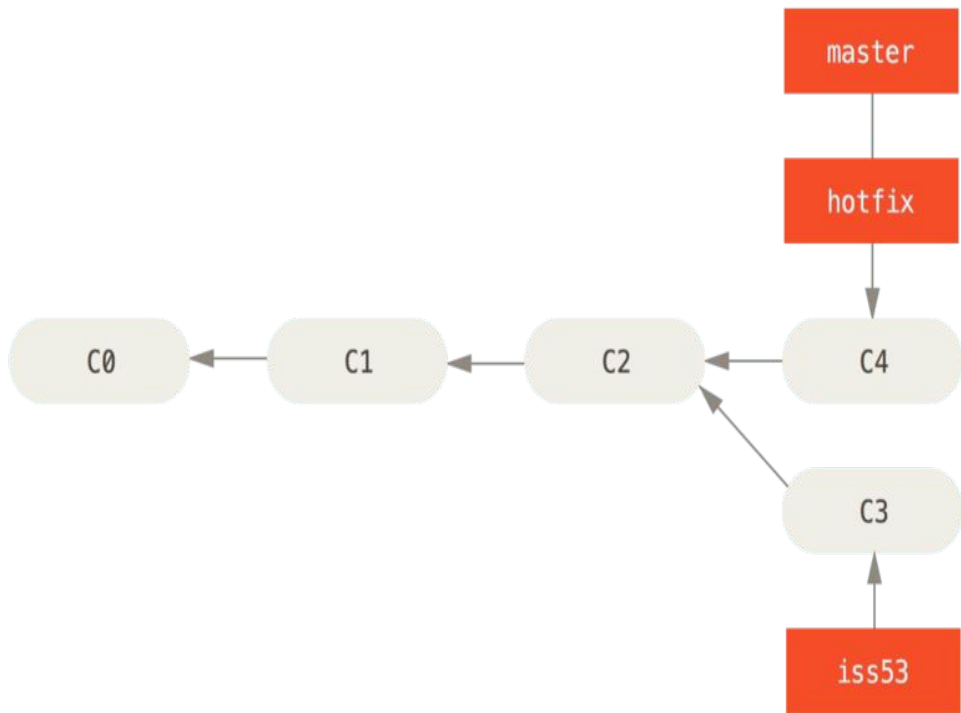
Switched to a new branch 'hotfix'

```
$ vim index.html
```

```
$ git commit -a -m 'fixed the broken email  
address'
```

```
[hotfix 1fb7853] fixed the broken email address  
1 file changed, 2 insertions(+)
```

Branching and merging



\$git checkout master

\$ git merge hotfix

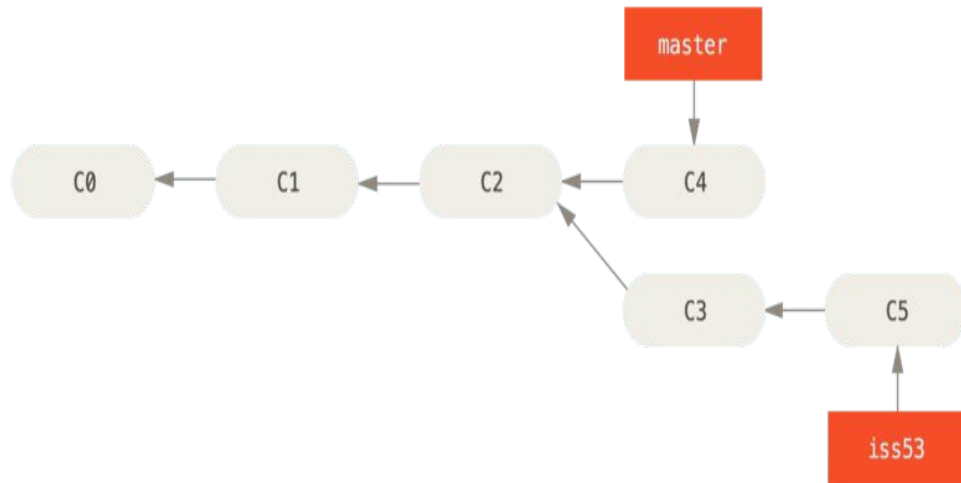
Updating f42c576..3a0874c

Fast-forward

index.html | 2 ++

1 file changed, 2 insertions(+)

Branching and merging



```
git branch -d hotfix
```

```
$ git checkout iss53
```

Switched to branch "iss53"

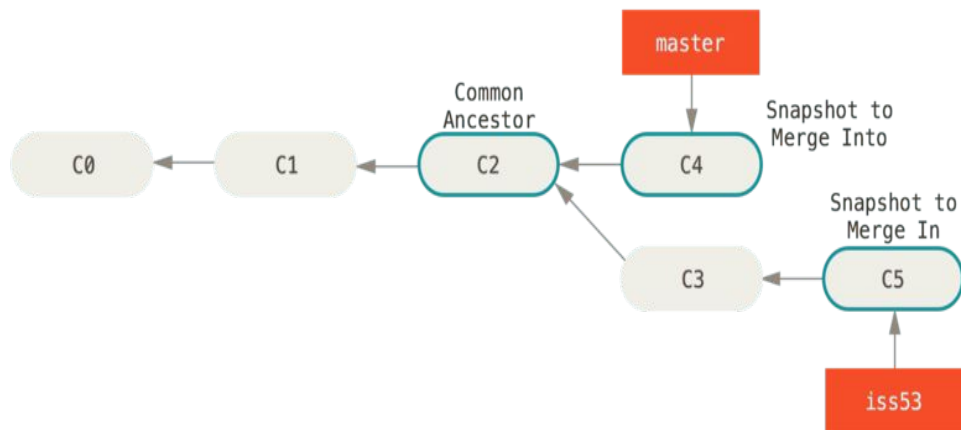
```
$ vim index.html
```

```
$ git commit -a -m 'finished the new footer  
[issue 53]'
```

```
[iss53 ad82d7a] finished the new footer [issue  
53]
```

```
1 file changed, 1 insertion(+)
```

Branching and merging



git checkout master

Switched to branch 'master'

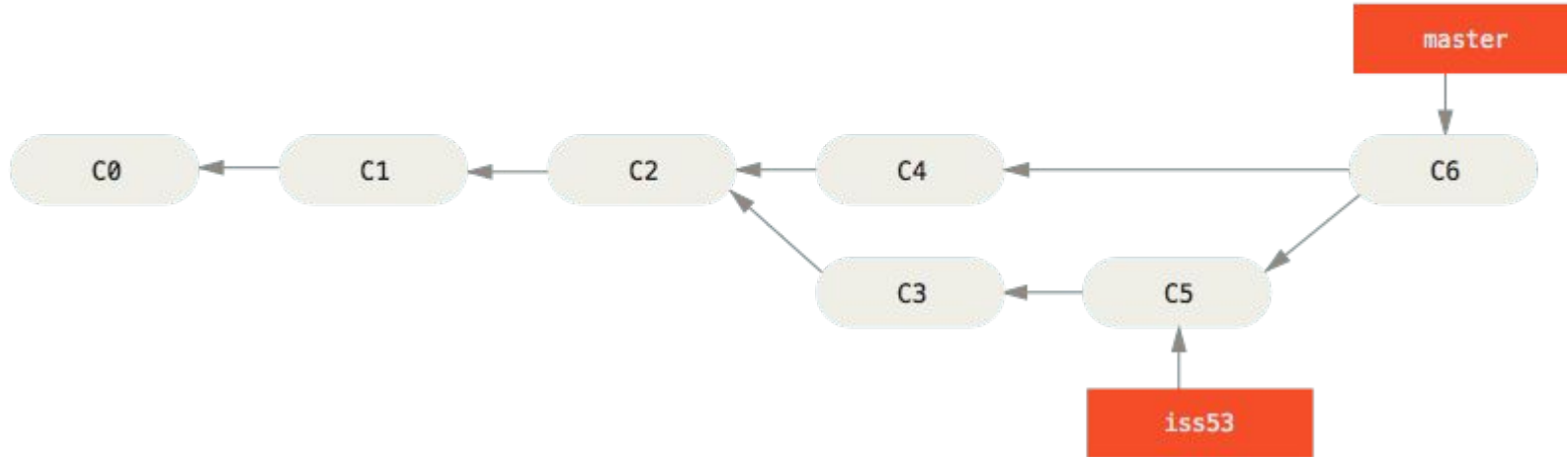
\$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

Branching and Merging



Branching and Merging

```
$ git merge iss53
```

```
Auto-merging index.html
```

```
CONFLICT (content): Merge conflict in index.html
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

- Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `git status`:

```
$ git status
```

```
On branch master
```


```
You have unmerged paths.
```

```
(fix conflicts and run "git commit")
```

```
Unmerged paths:
```

```
(use "git add <file>..." to mark resolution)
```

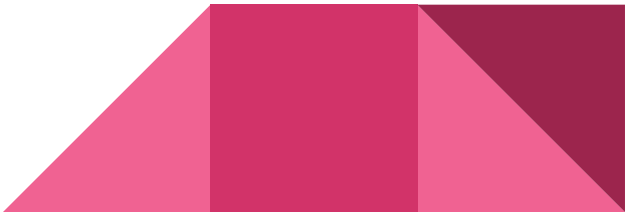
```
both modified:   index.html
```



Branching and Merging

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```



Branching and Merging

This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =====), while the version in your iss53 branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

This resolution has a little of each section, and the <<<<<<, =====, and >>>>>> lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git.



Branching and Merging

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run `git status` again to verify that all conflicts have been resolved:

```
$ git status
```

```
On branch master
```


```
All conflicts fixed but you are still merging.
```

```
(use "git commit" to conclude merge)
```

```
Changes to be committed:
```

```
    modified:   index.html
```

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit.



Branching and Merging

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
Merge branch 'iss53'
```

```
Conflicts:
```

```
    index.html
```

```
# It looks like you may be committing a merge.
```

```
# If this is not correct, please remove the file
```

```
#      .git/MERGE_HEAD
```

```
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

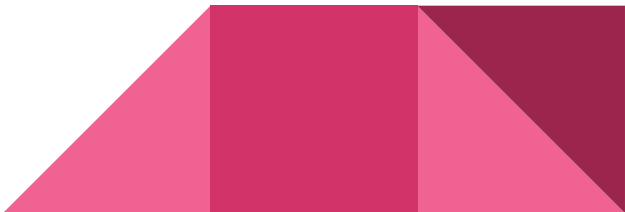
```
# On branch master
```

```
# All conflicts fixed but you are still merging.
```

```
#
```

```
# Changes to be committed:
```

```
#      modified:   index.html
```



More Commands

- Reverting
 - `git checkout HEAD main.cpp`
 - Gets the HEAD revision for the working copy
 - `git checkout -- main.cpp`
 - Reverts changes in the working directory
 - `git revert`
 - Reverting commits (this creates new commits)
- Cleaning up untracked files
 - `git clean`
- Tagging
 - Human readable pointers to specific commits
 - `git tag -a v1.0 -m 'Version 1.0'`
 - This will name the HEAD commit as v1.0



Lab Assignment

Setup

- Installing Git
 - Ubuntu: `sudo apt-get install git`
 - SEASnet: Git is installed in `/usr/local/cs/bin`
- Add it to PATH variable or use whole path
 - `export PATH=/usr/local/cs/bin:$PATH`
- Make a directory 'gitroot' and get a copy of the Diffutils Git repository
 - `mkdir gitroot`
 - `cd gitroot`
 - `git clone git://git.savannah.gnu.org/diffutils.git`
- Use `man git` to find commands



Hints

- Backporting
 - Apply a patch to a previous version
- Fix an issue with the diff diagnostic
- Hints
 - `git clone`
 - `git log`
 - `git tag`
 - `git show <hash_value>`
 - `git checkout v3.0 -b <BranchName>`

