Git Workshop

User Settings

- Before being able to commit files to any of the git repositories locally git requires two pieces of information, your name and email address.
- This is used to identify you as the person who commited the code.

```
git config --global user.name "Your Name"
git config --global user.email "your.name@email.server"
```

SSH Settings

- You can connect to remote repositories via HTTPS or SSH.
- SSH is more practical to avoid entering username/password.

1: Check for an existing SSH key

```
cd ~/.ssh
ls -a
```

If you see the file `id_rsa.pub` then you already have an SSH key and can skip to Section 3.

2: Create a new SSH key

We will be using the `ssh-keygen` tool to create a new SSH key with the RSA cryptosystem, your email is also required in this step.

```
ssh-keygen -t rsa -C "your.name@email.server"
```

- You can leave default location and no passphrase.
- You should now have two new files in your `~/.ssh` directory: your private key `id_rsa` which must be kept safe, and public key `id_rsa.pub` which can be provided to a Git server to enable SSH authentication.

3: Adding SSH key to you Git server

- This step depends on who your chosen Git hosting provider is.
- For sites such as GitHub and Bitbucket and self hosted installations of GitLab has a settings section which allows you to add your public key to the server quickly and easily.

Mergetool

- Solving merge conflicts can be a difficult and tedious task which can be made much easier with a visual merge tool such as meld.
- `meld` is available for Windows as well as Unix based systems.
- In order to setup Git to take advantage of an installed merge tool on Windows we need to perform the following configuration.
- Take care to replace the path to your merge tool with the correct path to the exectuable.

```
git config --global mergetool.meld.path "path to meld.exe"
git config --global merge.tool meld
```

Help

Autocorrect

It is a common problem to type a Git command incorrectly and recieve a message such as the following.

```
git: 'stauts' is not a git command. See 'git --help'.

Did you mean this?
        status
```

If this is a common problem for you then it is possible to enable a helper setting which will make an attempt to guess which command you were attempting to use.

```
git config --global help.autocorrect 1
```

Cross Platform Issues

Line Endings (Optional)

- Windows and Unix like systems use different characters to define the end of a line, Windows uses CRLF and Unix like systems use LR.
- The following setting tells Git to automatically deal with line endings globally.

```
git config --global core.autocrlf true
```

- If you checkout a commit before setting this option, you may be asked to commit changes to files you have not changed.
- In this situation the easiest approach is to remove the files which were checked out with the wrong line endings, then perform a hard reset on the repository.

Useful Aliases (Optional)

- When working on projects with many branches it can be confusing to determine the order of commits and which branches were merges and where. With the following alias it is possible show this information on the command line.
- First we need to create an alias called `tree` (you can choose other name).

```
git config --global alias.tree
"log --graph --abbrev-commit --decorate
--date=relative
--format=format:'%C(bold blue)%h%C(reset) -
%C(bold green)(%ar)%C(reset) %C(white)%s%C(reset)
%C(dim white)- %an%C(reset)%C(bold yellow)%d%C(reset)' --all"
```

- Once this is done we can use the alias with the command `git tree` .

Creating a New Project

- To start using Git first we need to be in a repository.
- We will initialize a new one, to do this we will need to run `git init`.
- To get started run the following commands:

```
mkdir tutorial
cd tutorial
git init
```

Copying Projects

- Many open source projects already use git (github).
- To work on any of them you need to clone the project, which creates a copy of the repository.
- You can do this with `git clone <url>`.

Checking the Status

- Now that we have initialized the repository, we can check the status of the repository.
- To see the state of the working directory or staging area just run `git status`.

```
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" ...)
```

Adding Content & Committing

- Now that we can get `git` to tell us the status of the repository, lets create some content for git to keep track of.

```
echo "foo" > foo.txt
echo "bar" > bar.txt
```

- By adding files to the working directory of the repository we have changes its status.
- Lets check how the change the status of the repository, run `git status` again

Adding Content & Committing (cont.)

```
On branch master

Initial commit

Untracked files:
(use "git add <file>..." to include in what will be committed

    bar.txt
    foo.txt

nothing added to commit but untracked files present...
```

Adding Content & Committing (cont.)

- Adding these files changed the working directory of our repository, we can use the `git add <file>` command to add these changes to the staging area.
- This is an area which tracks which changes to the working directory will be in the next commit.
- The staging area doesn't change the repository as these changes won't actually be recorded until you commit them.

Adding Content & Committing (cont.)

- To add your changes run `git add .`, this command adds all changes to files in the repository (but not deleted files).
- Now check the result by running `git status` again. You should see the following output:

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   bar.txt
    new file:   foo.txt
```

- `git add` can also be used to add directories in the same manner as files, it is simple `git add <directory>`.

Adding Content & Committing (cont.)

- Now that you have some changes in the staging area, we can go ahead and create your first commit:
  - `git commit -m "Initial commit"`.
- The `-m` tells `git commit` that you are giving it a message on the command line.
- Now our changes that were in the staging area have been stored as a commit.
- If you run `git status` again you will see the following message.

```
On branch master
nothing to commit, working directory clean
```

Adding Content & Committing (cont.)

- If you make a mistake when writing a commit message use `git commit --amend`, this will allow you to edit the commit message.

Viewing History

- All version control systems including Git show a projects commit history.
- The `git log` command displays the commited snapshots, showing your commit history.
- If you run `git log` you should see similar output:

```
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 15:38:15 2014 +0100

    Initial commit
```

Viewing History (cont.)

- Let's create some more commits so we can see what `git log` looks like when populated with more than one commit.

```
echo "foobar" > foobar.txt
echo "foo is not bar" > foo.txt
echo "bar is not foo" > bar.txt
```

Viewing History (cont.)

- Instead of using `git add .`, use `git add -p` (interactive mode)
- You should be prompted with output similar to this:

```
diff --git a/bar.txt b/bar.txt
index 5716ca5..02626e7 100644
--- a/bar.txt
+++ b/bar.txt
@@ -1 +1,2 @@
 bar
+ is not foo
Stage this hunk [y,n,q,a,d,/,e,?]?
```

Viewing History (cont.)

- Enter `y` and then `git commit -m "Add changes to foo & bar"`.
- Finally do the following:

```
git add foobar.txt
git commit -m "Add foobar.txt"
git log
```

Viewing History (cont.)

You should see similar output:

```
commit bff897c9d3d27cfee7451693550038 8b417f1c11
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:50:14 2014 +0100

    Add foobar.txt

commit 0c5275028bd818395de035aa1733a3f2889532e5
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:45:13 2014 +0100

    Add changes to foo & bar

commit 36a24afa1548fb767439f36e3e8bee8458f571ee
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:23:27 2014 +0100

    Initial commit
```

Viewing History (cont.)

- This output is fairly simple to understand, every commit has a 40 character SHA1 hash, which is a checksum for the commit and a unique ID. - Running `git log --decorate` will show some extra output.

```
commit bff897c9d3d27cfee7451693550388b417f1c11 (HEAD, master
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:50:14 2014 +0100

    Add foobar.txt
```

Viewing History (cont.)

- This extra output `(HEAD, master)` indicates that the latest commit `bff897c` is the current commit and that it is the tip of the master branch.
- You can use the `HEAD` reference and branch names with Git commands that require a commit id.

Viewing History (cont.)

- Let's try some of the following commands, see the different output they produce:

```
git log -n <limit>
git log --oneline
git log --stat
git log -p
git log --author"<pattern>"
git log --grep="<pattern>"
git log <since>..<until>
git log <file>
git log --graph --decorate --oneline
git log --graph --stat -p --decorate
```

Viewing History (cont.)

- Using `git log <since>..<until>` command allows you to seen between the specified range.
- For example `git log HEAD~2..HEAD`:

```
commit bff897c9d3d27cfee7451693550038 8b417f1c11
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:50:14 2014 +0100

    Add foobar.txt

commit 0c5275028bd818395de035aa1733a3f2889532e5
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sat May 10 22:45:13 2014 +0100

    Add changes to foo & bar
```

Viewing History (cont.)

- The `~` character is used for referencing the relative parent of the specified commit.
- HEAD~1 being the commit previous to HEAD and bff897c~1 being 0c52750, the commit before bff897c.
- For more information about `git log` visit here

Removing, Reverting, Resetting & Cleaning

- Now that we have some history in the repository, let's use it for what it is meant to do, store file versions safely.
- Run the following command `cat foo.txt` you should see:

```
foo is not bar
```

- If you wanted to use the original file, or just inspect what it contained you can run `git checkout HEAD~2` `foo.txt` is now in original state.
- Running `cat foo.txt` should yeild the original contents:

```
foo
```

- `git status` will show that `foo.txt` has been modified compared to HEAD (the current commit).

Resetting

- To redo the changes made to `foo.txt` there are several options:

```
git checkout foo.txt
git reset HEAD foo.txt
git reset --hard HEAD
```

Resetting (cont.)

- Try a command out then run `git status` to confirm it has been restored to the latest version (the second command will not full restore the file, the third command will).
- Use `git checkout HEAD~2` to set `foo.txt` back to the first version and try the next command.
- If `foo.txt` has been restored to the latest version you should see the following:

```
On branch master
nothing to commit, working directory clean
```

Resetting (cont.)

- The first command in the list, creates a `foo.txt` from the snapshot at HEAD, replacing the contents of the file.
- The second command resets any changes that were in the staging area for that file, all the changes are still present in the file.
- The last command will restore all tracked files in the staging area and the working directory to those in the HEAD commit.

Removing Changes & Files

- When making changes to your files, occasionally you might want to remove the the change in the case where you added a mistake.
- Run the following commands:

```
echo "foobar is neither foo nor bar" > foobar.txt
git add foobar.txt
git status
```

Removing Changes & Files (cont.)

- Now the change to `foobar.txt` has been added to the staging area, check with `git status` to confirm:

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   foobar.txt
```

Removing Changes & Files (cont.)

- Now to remove this change to `foobar.txt` from the staging area, you can use `git reset HEAD <file>` to remove changes to files from the staging area.
- To remove the change to `foobar.txt` run `git reset HEAD foobar.txt`.

```
echo "foobar is neither foo nor bar" > foobar.txt
git add foobar.txt
git commit -m "Add changes to foobar"
```

Removing Changes & Files (cont.)

- Say removing `foobar.txt` was a mistake, we can reset the index of the repository, essentially rewinding history. Let's try it out:

```
git rm foobar.txt
git commit -m "Remove foobar"
```

- We can run `git reset HEAD~1`, this will reset the index to the previous commit.
- **WARNING:** Only do this on commits to your local repository. DO NOT do this when a commit has come from a remote repository, it LITTERALLY rewrites history.

Reverting

- The correct way to undo a commit is to use `git revert`. This command creates a commit which will undo the changes of a specified commit.
- Let's try it `git revert HEAD`, reverting HEAD (the lastest commit) should result in the following `git log`:

```
commit 5b4c1b41f75673407c6965466b9e7dbe96d074d4
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sun May 11 09:28:22 2014 +0100

    Revert "removed foobar"

    This reverts commit d3840ad88286808062169c1a201316a00d.

commit d3840ad88286808062169c1a201316a00d85f880
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sun May 11 03:04:09 2014 +0100

    removed foobar
```

Reverting (cont.)

- We can see better what has happened by using `git log -p --oneline HEAD~2..HEAD`

```
5b4c1b4 Revert "removed foobar"
diff --git a/foobar.txt b/foobar.txt
new file mode 100644
index 0000000..4768904
--- /dev/null
+++ b/foobar.txt
@@ -0,0 +1 @@
+foobar is neither foo nor bar

d3840ad removed foobar
diff --git a/foobar.txt b/foobar.txt
deleted file mode 100644
index 4768904..0000000
--- a/foobar.txt
+++ /dev/null
@@ -1 +0,0 @@
-foobar is neither foo nor bar
```

Reverting (cont.)

- We can see in these diffs that `git revert HEAD` has created a commit that does the exact opposite of the previous commit, it has added `foobar.txt` back.
- This is the correct way to undo commits if they have been pushed to public projects.

Bare Repositories

- Remote repositories are normally stored as bare repositories. Bare repositories store all the information from inside the `.git` directory.

```
.
├── branches
├── config
├── description
├── HEAD
├── hooks
├── info
├── objects
├── packed-refs
└── refs

5 directories, 4 files
```

- `git clone https://github.com/jpmaldonado/git-tutorial-basic`

Bare Repositories (cont.)

- For the rest of the tutorial we will create a bare repository (to simulate what would happen with a remote one).

```
git clone --bare git-tutorial-basic
```

- This should produce the following output:

```
Cloning into bare repository 'git-tutorial-basic.git'...
done.
```

Pushing & Pulling

- Let's go back to our repository and get it working with our bare repository.
- All of the following commands will work the same even if the repository was an actual remote repository.
- The only differene would be the URL to access it.

```
cd git-tutorial-basic
git remote set-url origin ../git-tutorial-basic.git
```

Pushing & Pulling (cont.)

- When cloning the repository Git will also set a remote connection called origin
- This command tells Git to change the URL of the remote connection origin to `../git-tutorial-basic.git` we need to do this so we can use the new bare repository as a remote connection.

```
git push -u origin master
```

Pushing & Pulling (cont.)

- This should produce similar output:

```
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 359 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ../git-tutorial-basic.git
   7a2bf50..cd10233  master -> master
```

Pushing & Pulling (cont.)

- `git push` send your commits to the remote repository.
- To get changes from the remote server you can use `git fetch` and getting the changes and applying them to the working directory you can use `git pull`

Branches

- Git's more popular feature is its support of light weight branching.
- A branch is an abstract representation of an independent line of development.
- It allows you to make changes to your code, as if you were being given new working directory,staging area and project history.
- However you can instantly switch to others which may have other development in them.

Branches (cont.)

- So far we have been working exclusively one `master`, the default branch name.
- To create and switch to a new branch there are two methods:

```
git branch <name>
git checkout <name>
```

or

```
git checkout -b <name>
```

Branches (cont.)

- Both of these create a new branch and switch the working directory to it.
- Creating branches for specific tasks such as bug fixes, adding features and refactoring code are common practise.
- It allows several lines of development to be carried out in parrallel without conflict.

Branches (cont.)

- Once a line of development is complete, it should be merged into the appropriate branch and then deleted.
- This can be done using `git branch -d <name>` if a branch has not been merged but you still wish to delete it use `git branch -D <name>`
- Switching between branch is as simple as calling `git checkout <name>` this will switch HEAD to the relevant snapshot.

Ignore file

- When working on projects it is common that extra files will be generated by the development process, whether these are related to the build system, temporary files generated by your application running, or just something that you like having around, it is useful to be able to ignore these files.
- The `git` method for not adding files to the repository easily it the `.gitignore` file.
- This file lives in the root of your repository but is not provided by default, creating one is easy.

```
touch .gitignore
```

Ignore file (cont.)

- To ignore a specific file you simply add the file name to `.gitignore`, however it is usually easier to ignore directories or types of files.
- The `*` signifies a wildcard, meaning it will match any string.

```
# Ignore a directory
images
# Ignore all .txt files
*.txt
# Ignore all .txt files in images
images/*.txt
```

- And that's it, but remember that files which have already been commited to the repository will not be ignored.

Stashing

- The stash is a temporary place where local changes can stored whilst other operations are performed on the repository, such as pulling the lastest commits or changing branch before a commit.
- To stash changes is simple.

```
git stash
```

Stashing (cont.)

- You should see the following if the stash was successful.

```
Saved working directory and index state WIP on master:
```

- Otherwise if you have not made any changes you will see the following, please make a change to a file in order to continue.

```
No local changes to save
```

Stashing (cont.)

- Now that we have stashed some changes we can do whichever task for which the stash was required. It is possible to list all the current stashes.

```
git stash list
```

Which will show output in the following form.

```
stash@{0}: WIP on master: <commit> <message>
... possibly more stashes ...
```

Stashing (cont.)

- Stashes are stored in a stack, so the latest stash will have the `stash@{0}` identifier, any older stashes will have the form `stash@{n}` where `n` is the position in the stack.
- When the time comes when you want to unstash the changes there are two options, apply the changes or pop the changes which does an apply then a drop. We will start with apply.

```
git stash apply stash@{0}
```

Stashing (cont.)

- A successful apply will show the output of a call to `git status` and your changes will be applied to the repository. If we call `git stash list` you will see that your stash is still avaialble.
- The alternative is to use `git stash pop` which will apply the changes then drop the stash from the stash list.
- First we need to perform a hard reset to avoid a merge conflict.

```
git reset --hard HEAD
git stash pop
git stash list
```

Stashing (cont.)

- If all goes well the stash should be applied to the working directory and the displayed list will no longer show the stash.
- Lets add the stash once again, then explore how to view the contents of a stash.

```
git stash
git stash list
```

Stashing (cont.)

- To view the contents of a stash we can use `git stash show stash@{0}` to see the insertions and deletions, however it is usually more useful to see the changes in patch form.

```
git stash show -p stash@{0}
```

- Finally if we decide we no longer require the changes containted within it. The stash can be dropped as follows.

```
git stash drop stash@{0}
git stash list
```

Merging

- The majoritory of the time `git` can automatically merge files without problems, this gives learning how to deal with merge conflicts a lower priority especially when working on your own.
- However when a merge conflict does arise it can be a slow and painful experience to fix it properly.
- This section relies on having a visual merge tool setup on your system, we will be using `meld` which you should have setup if you followed the Mergetool section above.

Merging (cont.)

- First of all lets setup a situation where a merge conflict will occur, for this we need to have a file commited in out repository.

```
echo "Some text" > conflict.txt
git add conflict.txt
git commit -m "Add conflict.txt"
```

Merging (cont.)

- Now that we have our file in the repository tree we require an additional branch, switch to this branch and make a change to contents of `conflict.txt`. Once this is done we can commit these changes and change back to the master branch.

```
git branch conflict
git checkout conflict
echo "Some changed text" > conflict.txt
git commit -am "Changed conflicts.txt"
git checkout master
```

Merging (cont.)

- Back on the master branch now and the final change to `conflict.txt` before we can see how to solve a merge conflict.

```
echo "Another change" > conflict.txt
git commit -am "The conflicting change"
```

Merging (cont.)

- We are now in a situation where a merge conflict will occur when we attempt to merge the `conflict` branch into master as follows.

```
git merge conflict
```

- This will output the following message

```
Auto-merging conflict.txt
CONFLICT (content): Merge conflict in conflict.txt
Automatic merge failed; fix conflicts and then commit.
```

Merging (cont.)

- This is usually a good time to view the status of the repository in order to see the extent of the merge conflict.

```
git status
```

Merging (cont.)

- Which should look like this.

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      conflict.txt

no changes added to commit (use "git add" and/or "git commit"
```

Merging (cont.)

- So now `git` has told us that we need to resolve the merge conflict in the file `conflict.txt` we do this by invoking the merge tool we set up earlier.

```
git mergetool conflict.txt
```

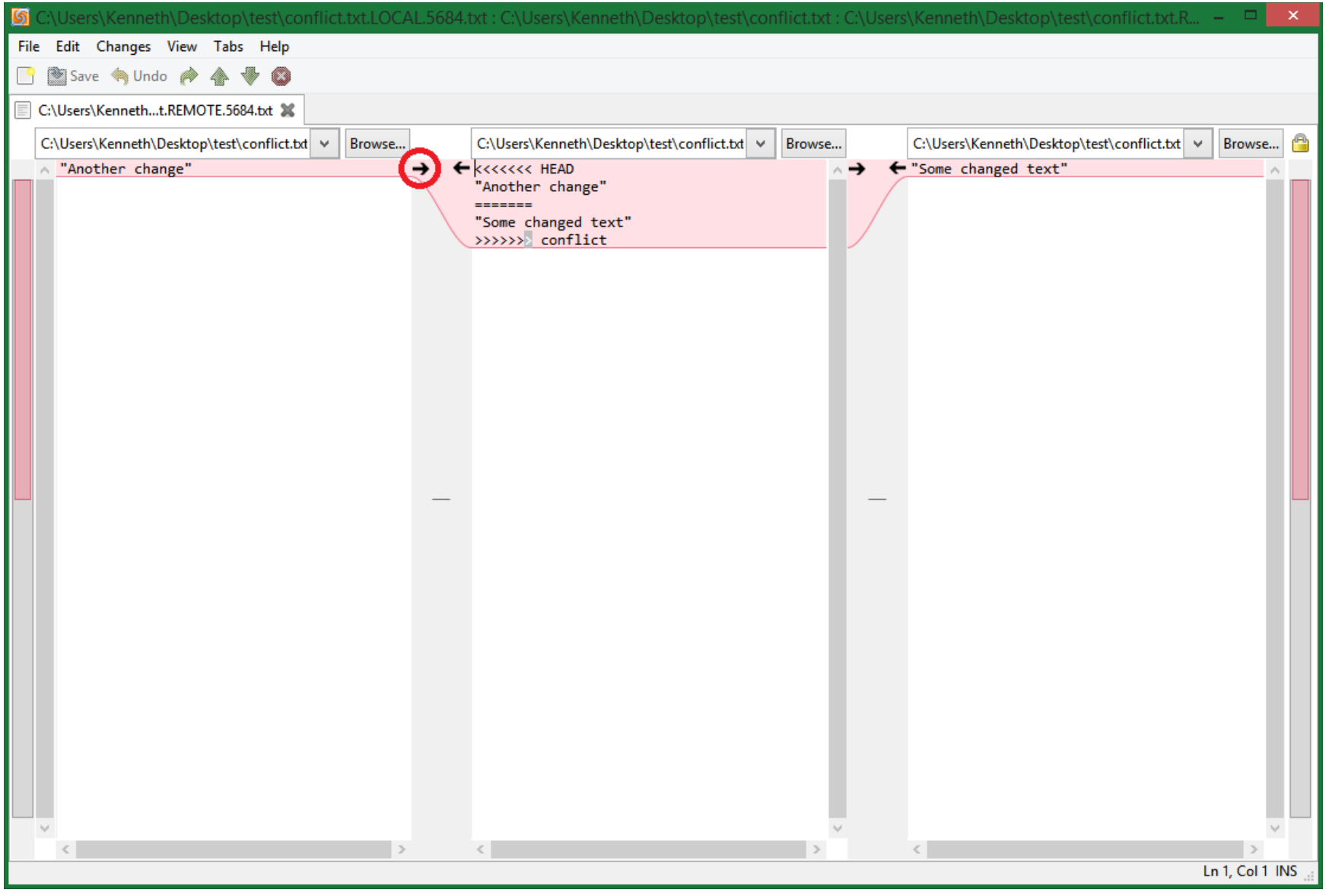
Merging (cont.)

- Will prompt you with the following, just hit enter and a `meld` window will appear.

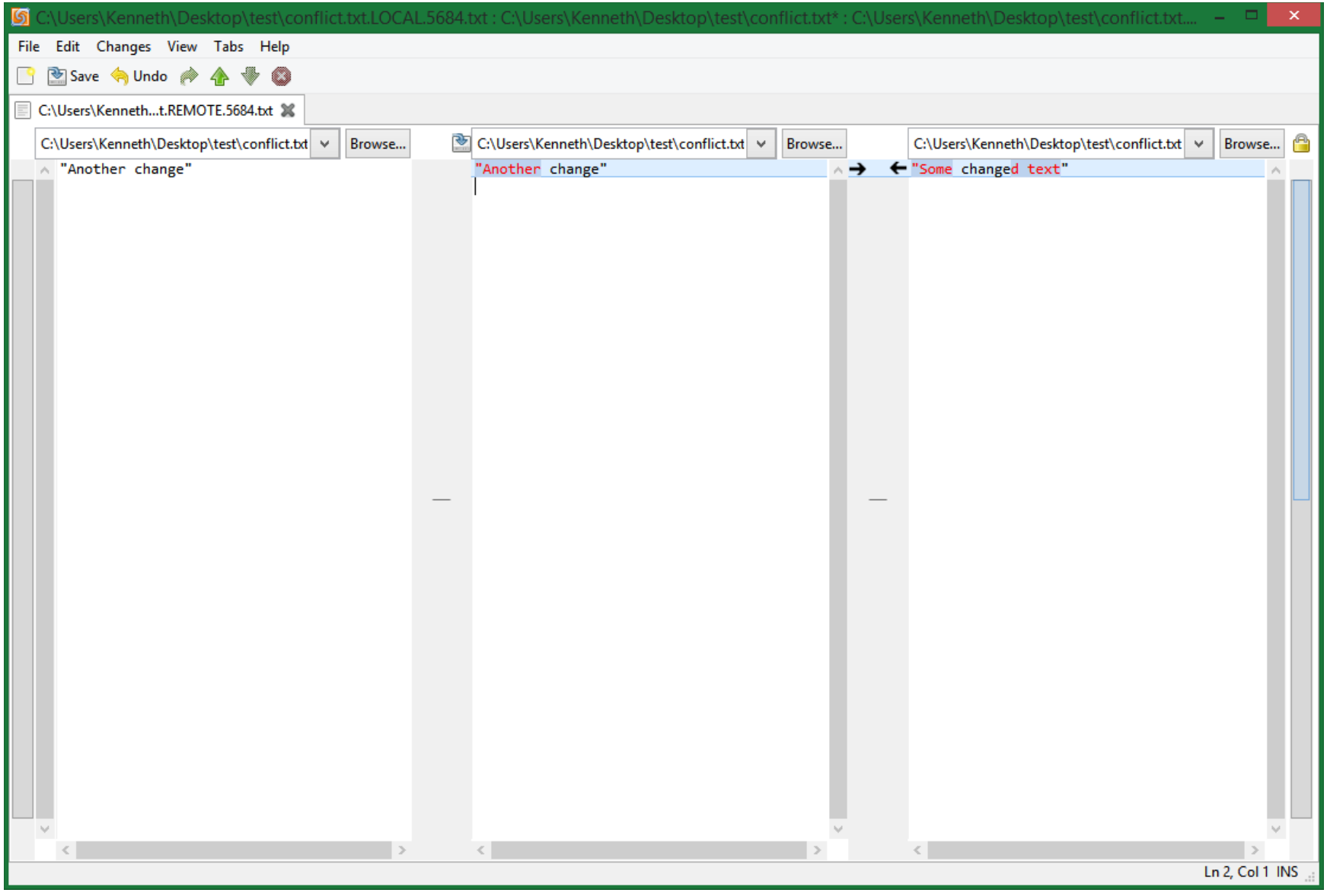
```
Merging:
conflict.txt

Normal merge conflict for 'conflict.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

Merging (cont.)

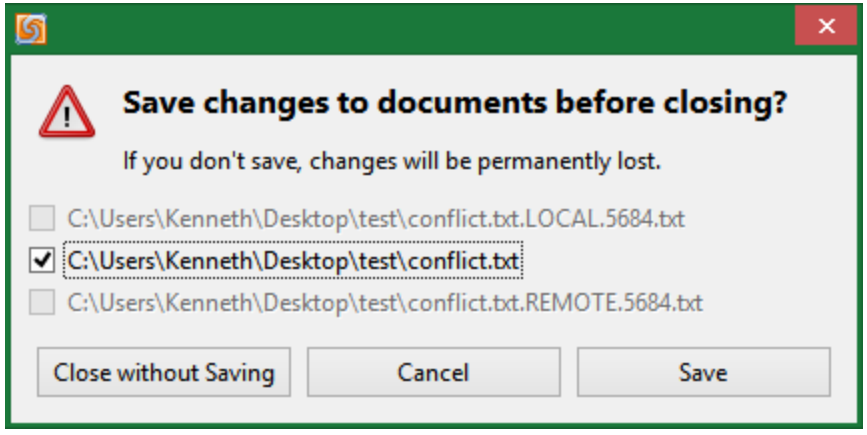


Merging (cont.)



Merging (cont.)

- Finally we can exit `meld` and save the `conflict.txt` file.



Merging (cont.)

- We have successfully resolve the merge conflict, the final task is to commit the changes.

```
git commit
```

The commit message will look like this, there is no need to change it.

```
Merge branch 'conflict'

Conflicts:
    conflict.txt
```

- And you are done, merge conflicts are no longer your enemy.

Submodules

- The majority of projects are built upon existing libraries, distributing these projects can prove difficult because these external dependencies must be available on the target system which makes building a problem.
- One approach to solving this problem is to supply the source code for the libraries you rely on with your own project.

Submodules (cont.)

- However copying the source code into your repository is hard to update and can lead to files being accidentally changed.
- Submodules are the git approach to this problem, a submodule is an existing git repository that you reference in your own git repository.
- To add a submodule to your repository you simply need to the URL of the repository you want to use. For this example we will create a new repository and add the GitHub GLFW repository as a submodule.

Setup

```
mkdir glfw-example
cd glfw-example
git init
```

- It is good practice, but not essential, to keep your submodule in a directory within your repository. Here we will store GLFW in the `external` repository although it is also common to use `ThirdParty` or similar.

```
mkdir external
```

Adding a submodule

- Now we will add our submodule

```
git submodule add \
https://github.com/glfw/glfw.git external/glfw
```

which results in the following output

```
Cloning into 'external/glfw'...
remote: Reusing existing pack: 13159, done.
remote: Total 13159 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (13159/13159), 6.41 MiB | 1.41 MiB/s.
Resolving deltas: 100% (8082/8082), done.
Checking connectivity... done.
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings...
```

Adding a submodule (cont.)

- Now we can see that the GLFW library source code has been added to the `external/glfw` directory, but we are not finished. If we check the status of our repository we will see the following output.

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitmodules
        new file:   external/glfw
```

Adding a submodule (cont.)

- There are two entries to be added, the first is the file `.gitmodules` which contains a list of all the submodules which have been added to the project. The second is the submodule its self. Lets commit our changes.

```
git commit -m "Add glfw submodule"
```

Removing a submodule

- Removing a submodule from a repository can be tricky, so here is the process. First we need to edit the `.gitmodules` file and remove the following entry.

```
[submodule "external/glfw"]
    path = external/glfw
    url = https://github.com/glfw/glfw.git
```

And then stage `.gitmodules`

```
git add .gitmodules
```

Adding a submodule

Then we need to edit the file `.git/config` removing the following entry.

```
[submodule "external/glfw"]
    url = https://github.com/glfw/glfw.git
```

Adding a submodule

- Now we need to remove the submodule from the repository tree, and also remove the module reference.

```
git rm --cached external/glfw
rm -rf .git/modules/external/glfw
```

- Now we can commit the changes made to remove the submodule

```
git commit -m "Remove submodule external/glfw"
```

Adding a submodule

- Finally we can now remove the physical files from the file system and view the state of the repository.

```
rm -rf external/glfw
git status
```

Which results in the following

```
On branch master
nothing to commit, working directory clean
```

Cloning a Repo with Submodules

To clone a repo and check out all of its submodules then run

```
git clone --recursive <url>
```

You can specify the branch also by running

```
git clone --recursive -b <branch> <url>
```

Rebasing

- Git rebasing is a very powerful tool.
- It can be be used to rewrite your repositories history, for better or worse.
- `git rebase` does this by copying commits from a branch and copying them to new base commit.

Rebasing (cont.)

Let's `git rebase <base>` to understand how it works.

```
cd ../
git clone https://github.com/jpmaldonado/git-rebasing.git
cd git-rebasing
```

Rebasing (cont.)

- If you run `git log` you will see that this repository has an extra commit on the master branch compare to the last repository

```
commit f172eb1d904be0621d4152f2bdde0b5d89433152
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Mon May 12 07:48:38 2014 +0100

    Add barfoo.txt

commit 7a2bf50939c737a2bb5fa47d2a2763b692adda79
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sun May 11 09:48:50 2014 +0100

    Remove foobar
```

Rebasing (cont.)

- By running `git branch -v` we can see there is more than one branch.

```
  bring-back-foobar b24dcb0 Revert "Remove foobar"
* master            f172eb1 Add barfoo.txt
```

Rebasing (cont.)

- It appears both have different tip commits. `git log bring-back-foobar` reveals
  that the history is also different between these branchs:

```
commit b24dcb0b0da31ba7cfe84dde0298b35f0e0213d4
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Mon May 12 08:02:25 2014 +0100

    Revert "Remove foobar"

    This reverts commit 7a2bf50939c737a2bb5fa47d2a2763b692a.

commit 7a2bf50939c737a2bb5fa47d2a2763b692adda79
Author: Sean Jones <neuralsandwich@gmail.com>
Date:   Sun May 11 09:48:50 2014 +0100

    Remove foobar
```

Rebasing (cont.)

- If we wanted to merge `bring-back-foobar` into master, we would end up with a non-linear history. `git log --graph --oneline`

```
*   fcfb589 Merge branch 'bring-back-foobar'
|\
| * b24dcb0 Revert "Remove foobar"
* | f172eb1 Add barfoo.txt
|/
* 7a2bf50 Remove foobar
* fff6840 Add changes to foobar
* 2042f6c Add foobar.txt
* 9085b36 Add changes to foo & bar
* 2f7907c Initial commit
```

Rebasing (cont.)

- While this in itself is not a bad thing, for the trivial change that this branch brings, it doesn't need to introduce a mess into the history. Let's fix this so when we do merge, we can get a nice linear history.

```
git co bring-back-foobar
git rebase master
```

Rebasing (cont.)

- Now if we look at the history of this branch we will see a nice linear series of commits. `git log --graph --oneline`

```
* da75a4c Revert "Remove foobar"
* f172eb1 Add barfoo.txt
* 7a2bf50 Remove foobar
* fff6840 Add changes to foobar
* 2042f6c Add foobar.txt
* 9085b36 Add changes to foo & bar
* 2f7907c Initial commit
```

Rebasing (cont.)

- Now if we merge bring-back-foobar into master, it will fast-forward and create the nice linear history we can see above.

Exercise

- Git workflow workshop for two

References

- Oh Shit, Git!?!
- Software Carpentry Tutorial
- LearnGitBranching Tutorial