

Git Tutorial - Part 2

Using Git

Getting Ready?

If your running Linux, make sure you have git installed. Open a command window, you now have everything you need .

If your running windows, open shell window for git using "Git BASH", you can right click in a directory and use "Git BASH Here" if its installed (this only gives you a shell already in the correct directory).

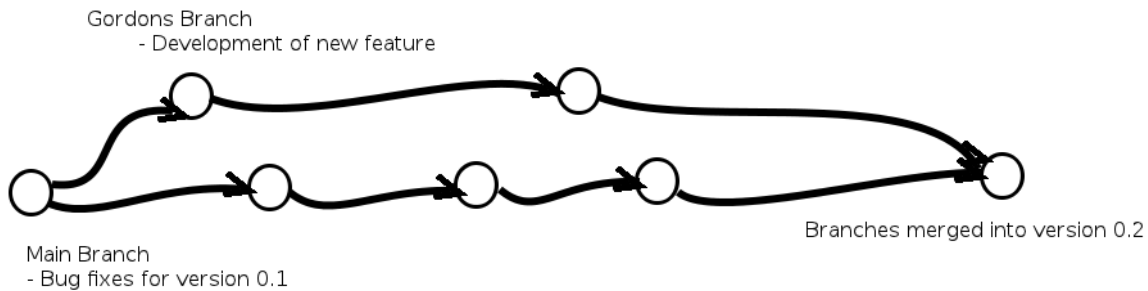
For those of you who have forgotton how the shell works since last week, please see the breif cheat sheet below and link.

External tutorial: http://linuxcommand.org/learning_the_shell.php/learning_the_shell.php

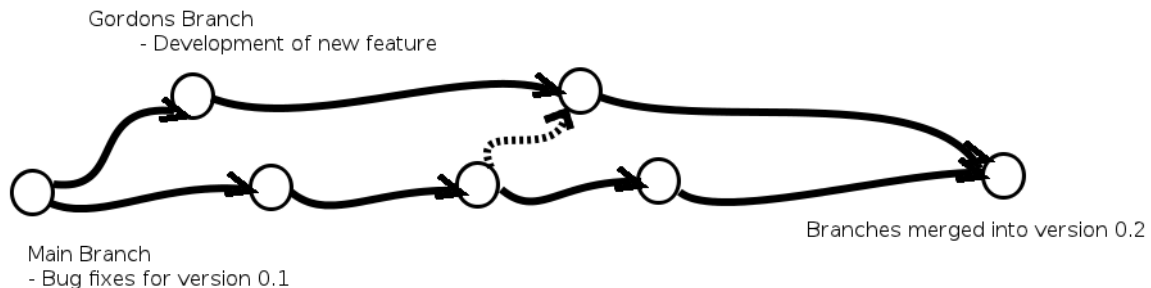
Command	Description
pwd	Print Working Directory – Tells you where you are in the directory structure of the file system. e.g. /home/sc-gj/Documents/Notes
ls	List – Provides a list of files in the current directory, options can be added using – e.g. -l (long format) -h (human readable) -a (all files, inlcuding hidden) and can be combined -alh
man	Manual page – Can be used to display a manual page for any bash command e.g. man ls (this is not included in MINGW).
mkdir <name>	Make Directory – creates a directory with the specified name in the current directory. Can create a directory anywhere in the filesystem (provided parent directory exists) using relative or full path instead of name.
rm	Remove – Can be used to remove files or directories. For directories use -r to recurse though child directories and -f to force (avoid yes / no question for each file). These can be combined, -rf is a very common combination. Be careful however, once they are gone – THEY ARE GONE.
cp <source> <destination>	Copy – a file or directory to another location, to copy and directory and its content use -R
mv <source> <destination>	Moves / Renames – moves or renames a file
cat <filename>	Concatenate to screen buffer – essentially show file content

Branches

To understand branching in Git, we first need to understand what a branch is. The best way to starting thinking about branches is that each branch is as a side project to the main development effort. So for example, Mike, Andy, Gordon and myself are working on a project. Mike, Andy and myself are fixed bugs for the version 0.1 in the main branch, while Gordon is working on a new feature in a branch. His code is not part of the main development though he may update his working copy with our changes. At some point the feature will be finished and we'll merge his branch with the working branch perhaps for version 0.2. I've tried to show this diagrammatically below, time goes from left to right.



This kind of main trunk and branches structure is common to many version control systems including CVS. Git takes this a step further in that everytime you clone a repository you get what is essentially a branch which you then work on until you merge that branch. The above example has one minor flaw which I should point out, Gordon is working alone on his branch for far to long, chances are his version will be very different from the main branch and the final merge a difficult one. The solution is to regullally update the branch and split the big merge into smaller ones (this may actually avoid some problems before they occur).



From last week you will remember that git stores data as a series of snapshots (rather than changes sets or detlas). When Git commits it stores an object that contains a pointer to the snapshot of the content you staged, along with meta-data (time, date, author, comments etc.) and zero or more pointers to the direct parents of this commit.

- In the case of the first commit – Zero parents
- For the 'normal' commits we looked at last week - one parent

- Where a merge has taken place and two branches modified separately have been changed - two parents.

A useful visualisation of this process is provided by the [Pro Git book I'll start by going through their example, we'll then look at one of our own.](#)

Assume we have a directory containing 3 files which and create a git project for them using `git init` (as we did last week). We stage all of them each is given a checksum (SHA-1 hash) and stores that version of the file in the repository (Git calls these blobs) it also adds the checksum to the staging area.

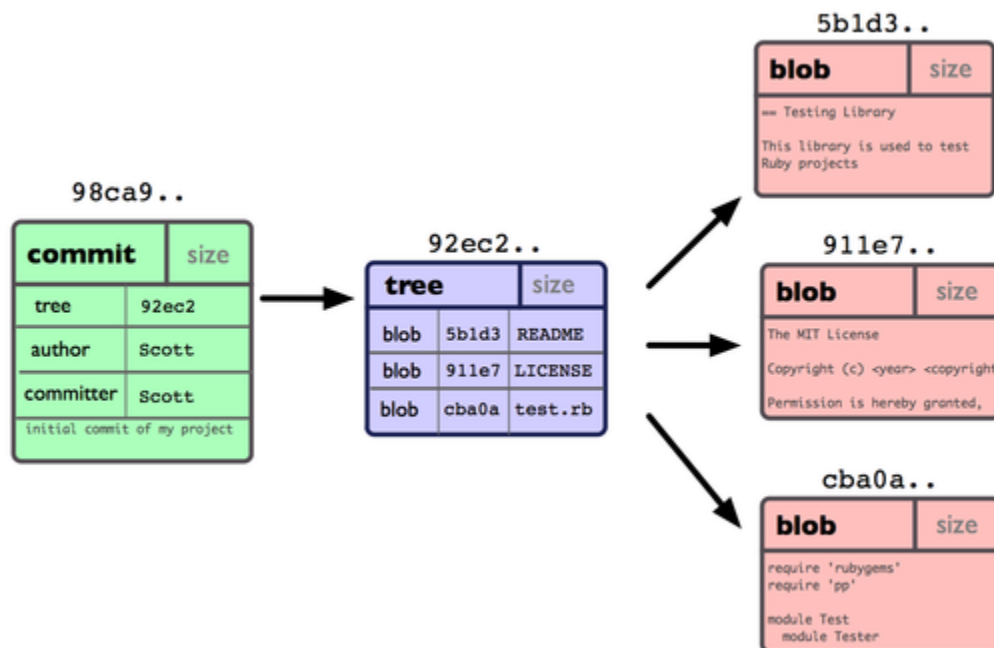
When we commit (`git commit`) git checksums the directory (and any subdirectories) and stores these tree objects in the git repository. It then creates a commit object that has the metadata and a pointer to the root project so it can re-create the snapshot as required. The git repository now has 5 objects.

3 Blobs – one for each of your 4 files.

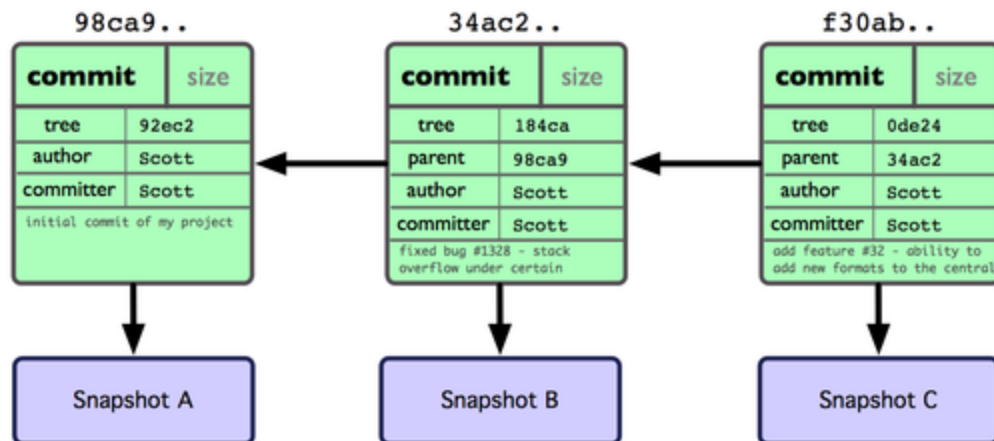
1 Tree – that lists the files stored in each of the blobs

1 Commit – with a pointer to the root tree and all commit meta data

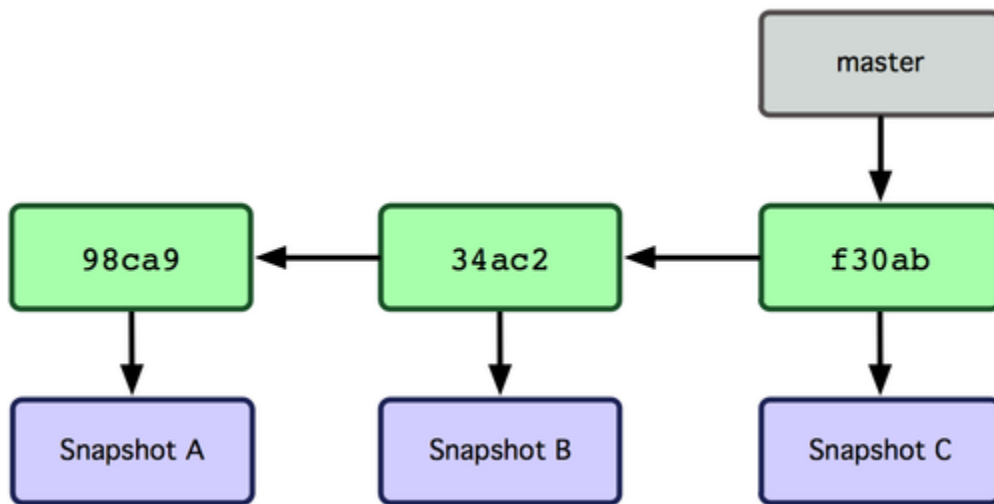
Conceptually ...



For each commit we make Git stores a pointer to the one before it, so after a further two commits our history might look something like this.



A branch in git is simply a lightweight movable pointer to one of these commits, the default being 'master'. As you make commits the master branch moves to the last commit you made.



We need to see this for our selves, so lets quickly make ourselves a project we can play about with locally.

At the command line we make a directory

```
mkdir helloWorld
cd helloWorld
```

Make a quick test file

```
touch main.c
```

```
echo "#include <stdio.h>" >> main.c
```

```
echo " " >> main.c
```

```
echo "int main()" >> main.c
```

```
echo "{" >> main.c
```

```
echo "    printf("Hello world\n");" >> main.c
```

```
echo "}" >> main.c
```

Check its content

```
cat main.c
```

Make this a git repository

```
git init
```

Stage the file

```
git add main.c
```

Commit the file

```
git commit -m "First commit of hello world program"
```

Check the status

```
git status
```

Check the log

```
git log
```

Now lets look at the branches we currently have, using git branch.

```
git branch
```

Adding the -v flag for verbose mode provides us with a partial SHA-1 hash and comment allowing us to confirm that the master branch points to our commit.

```
[sc-gj@sc-miho helloWorld]$ git log
commit afe46890a7213e289783c87af0dd16e031cad2bb
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 13:32:40 2012 +0100

    First commit of hello world program
[sc-gj@sc-miho helloWorld]$ git branch
* master
[sc-gj@sc-miho helloWorld]$ git branch -v
* master afe4689 First commit of hello world program
[sc-gj@sc-miho helloWorld]$ █
```

We perform a simple modification and re-commit the file, using a combination of git log and git branch we can see that the master branch now points to the most recent version.

```
1 files changed, 3 insertions(+), 0 deletions(-)
[sc-gj@sc-miho helloWorld]$ git log
commit f5d77a2401e32e8dc638da8c0a5f62990344a89c
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 14:52:22 2012 +0100

    Added comment

commit afe46890a7213e289783c87af0dd16e031cad2bb
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 13:32:40 2012 +0100

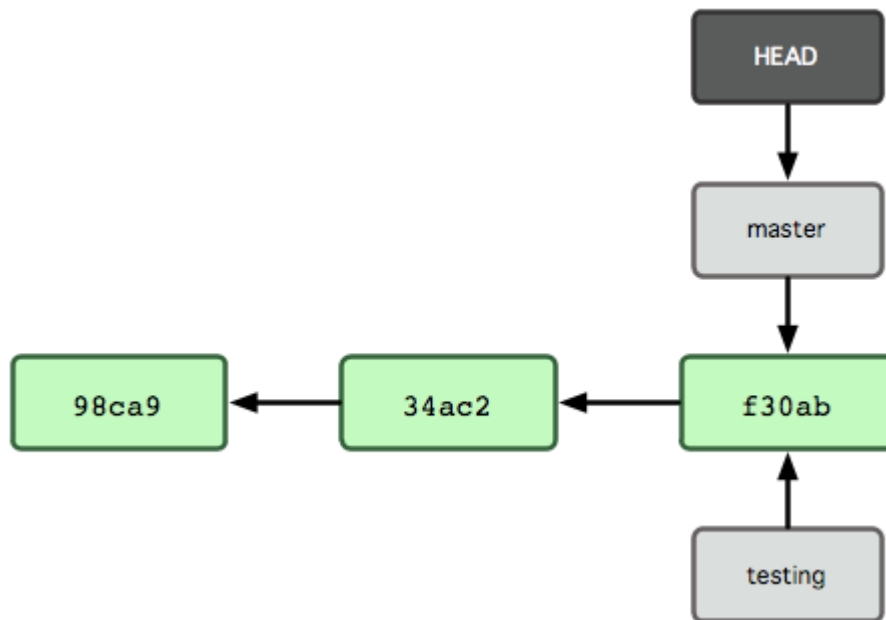
    First commit of hello world program
[sc-gj@sc-miho helloWorld]$ git branch -v
* master f5d77a2 Added comment
[sc-gj@sc-miho helloWorld]$ █
```

Creating a Branch

Now to create another branch, I'll call it testing.

git branch testing

This creates another branch at the same pointer as the commit we're currently on (in this case its the same as pointer to by master, but need not be). A question which arises is how does Git know what branch your currently in? It keeps a special pionter knoww as HEAD, this is unlike the head found in other VCS systems such as CVS (here HEAD represents the most recent version in the centralised repository).



Next we need to know how to move between branches, this is atchieved using git checkout, be warned if you have any uncommitted changes in yuor current branch they will be lost. Lets move to the testing branch and make a change, observing the result using git log and git branch.

```

[sc-gj@sc-miho helloWorld]$ git branch testing
[sc-gj@sc-miho helloWorld]$ git branch
* master
  testing
[sc-gj@sc-miho helloWorld]$ git checkout testing
Switched to branch 'testing'
[sc-gj@sc-miho helloWorld]$ git branch
  master
* testing
[sc-gj@sc-miho helloWorld]$ vi main.c
[sc-gj@sc-miho helloWorld]$ git add main.c
[sc-gj@sc-miho helloWorld]$ git commit -m "Bug fix to printf"
[testing 44b0ee4] Bug fix to printf
 1 files changed, 1 insertions(+), 1 deletions(-)
[sc-gj@sc-miho helloWorld]$ git branch -v
  master f5d77a2 Added comment
* testing 44b0ee4 Bug fix to printf
[sc-gj@sc-miho helloWorld]$ git log
commit 44b0ee47f2a8ae055196fe5ef0c1b53f2683743d
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 15:17:53 2012 +0100

```

Bug fix to printf

```

commit f5d77a2401e32e8dc638da8c0a5f62990344a89c
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 14:52:22 2012 +0100

```

Added comment

```

commit afe46890a7213e289783c87af0dd16e031cad2bb
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 13:32:40 2012 +0100

```

First commit of hello world program

```

[sc-gj@sc-miho helloWorld]$ █

```

The testing branch has now moved one commit beyond the master branch, we can see this from the SHA-1 has numbers and comments. We can move to the master branch at any point and potentially take this in a different direction.

Next we demonstrate divergence in the two branches, i.e. the branches will contain the same files but with different changes. Clearly when this happens merging the branches back together is going to be more difficult. This is not always the case, one branch may have new files or simply changes to other areas of the same files. I'm going to orchestrate a more difficult merge with 'conflicts' for later illustration purposes.

Move back to the master branch (N.B. If we do this without first committing, we'll lose our changes so use `git status` to make sure there are no unstored changes).

```
git checkout master
```

Using `vi`, I make some changes to the `println` in the `main.c` file, as I've updated the same line elsewhere this should cause a conflict. That's not important at present, but will be of interest in the next section. Finally I commit the changes and look use `git branch` and `git log` to examine the master branch.

```
git add main.c  
git commit -m "Changes to printf in the master branch"
```

This screenshot and the one above show that both branches have a common ancestor.

```
[sc-gj@sc-miho helloWorld]$ git branch -v
  master f5d77a2 Added comment
* testing 44b0ee4 Bug fix to printf
[sc-gj@sc-miho helloWorld]$ git log
commit 44b0ee47f2a8ae055196fe5ef0c1b53f2683743d
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 15:17:53 2012 +0100

    Bug fix to printf

commit f5d77a2401e32e8dc638da8c0a5f62990344a89c
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 14:52:22 2012 +0100

    Added comment

commit afe46890a7213e289783c87af0dd16e031cad2bb
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 13:32:40 2012 +0100

    First commit of hello world program
[sc-gj@sc-miho helloWorld]$ git checkout master
Switched to branch 'master'
[sc-gj@sc-miho helloWorld]$ vi main.c
[sc-gj@sc-miho helloWorld]$ git add main.c
[sc-gj@sc-miho helloWorld]$ git commit -m "Modifed printf in the master branch"
[master 067e099] Modifed printf in the master branch
 1 files changed, 1 insertions(+), 1 deletions(-)
[sc-gj@sc-miho helloWorld]$ git branch -v
* master 067e099 Modifed printf in the master branch
  testing 44b0ee4 Bug fix to printf
[sc-gj@sc-miho helloWorld]$ git log
commit 067e09942817a346aac13d7e01334ea3bd67dde2
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 15:57:16 2012 +0100

    Modifed printf in the master branch

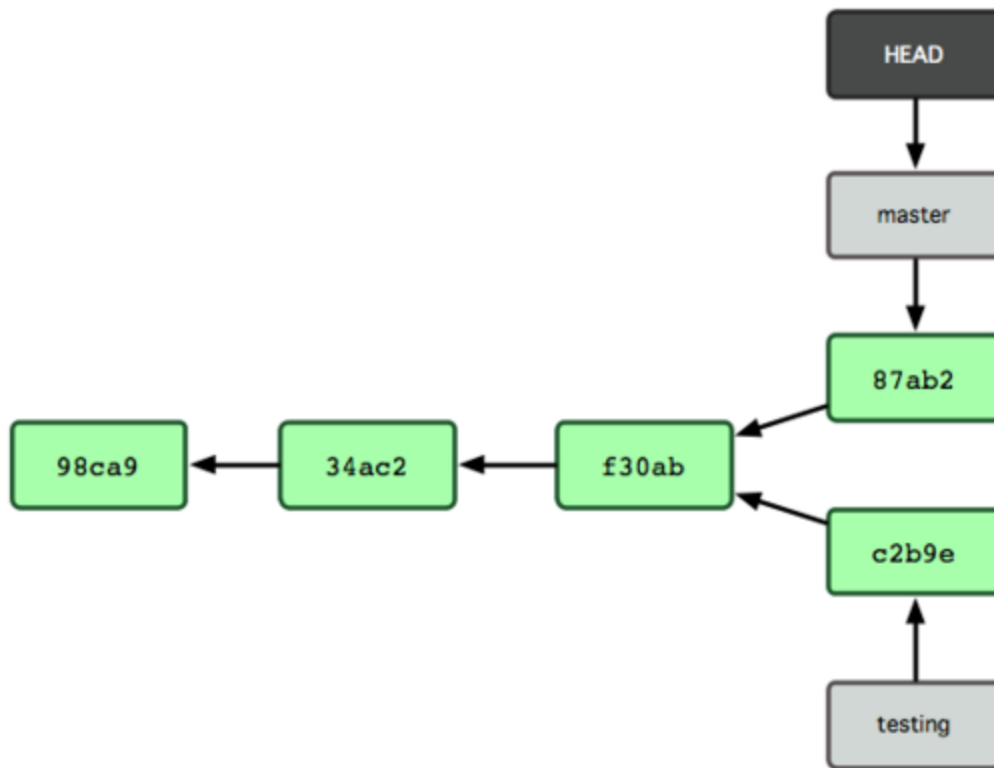
commit f5d77a2401e32e8dc638da8c0a5f62990344a89c
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 14:52:22 2012 +0100

    Added comment

commit afe46890a7213e289783c87af0dd16e031cad2bb
Author: sc-gj <glenn.l.jenkins@smu.ac.uk>
Date: Tue Aug 28 13:32:40 2012 +0100

    First commit of hello world program
[sc-gj@sc-miho helloWorld]$ █
```

Visually we end up with something like this.



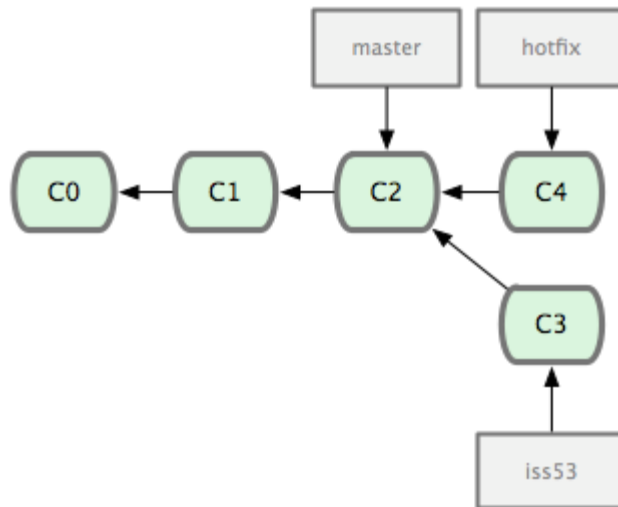
Final Notes on Branching

There are a few points we need to note in order to appreciate the power of Git as a version control solution. The branches are cheap to create and destroy (in terms of memory/processing required) as each is a 41byte file (40 characters of the SHA-1 checksum and an EOF). In other VCS tools branches are copies of the whole repository and creating one can take minutes or even hours. Also because each time we merge we record the parents finding the merge base (place at which the branches differ) is easy to do.

With this in mind the philosophy of Git is to branch often, to anyone familiar with CVS this is a horrific thought as CVS branch / merge operations can be painful.

Merging

When merging together two branches there are two possible scenarios. Firstly, where the branch you are merging is directly upstream of the current branch. In the following example hotfix is directly upstream of master. In this case the merge is pretty simple and git simply moves the pointer forward, in this case there is no divergent work to merge (i.e. we're not trying to merge the end of two branches).



When this sort of merge is executed, the output shows the term "Fast forward" to describe the movement of the pointer. Lets work though an example of this.

Make sure we're in the master branch.

```
git branch
```

If we're not move to the master branch (make sure we don't have anything to commit first using git status).

```
git status
git checkout master
```

Create a new branch.

```
Git branch hotfix
git checkout hotfix
vi main.cpp
```

Make some changes (I've chosen to change the comment).

```
git add main.c
git commit -m "Changes made to comment"
```

Now return to the master and merge in these changes.

```
git checkout master
git merge hotfix
```

We should see something like this.

```

[sc-gj@sc-miho helloWorld]$ git branch
* master
  testing
[sc-gj@sc-miho helloWorld]$ git status
# On branch master
nothing to commit (working directory clean)
[sc-gj@sc-miho helloWorld]$ git branch hotfix
[sc-gj@sc-miho helloWorld]$ git checkout hotfix
Switched to branch 'hotfix'
[sc-gj@sc-miho helloWorld]$ ls
main.c
[sc-gj@sc-miho helloWorld]$ vi main.c
[sc-gj@sc-miho helloWorld]$ git add main.c
[sc-gj@sc-miho helloWorld]$ git commit -m "Changes made to comment"
[hotfix 91e09c9] Changes made to comment
 1 files changed, 2 insertions(+), 2 deletions(-)
[sc-gj@sc-miho helloWorld]$ git checkout master
Switched to branch 'master'
[sc-gj@sc-miho helloWorld]$ git merge hotfix
Updating 067e099..91e09c9
Fast-forward
  main.c |    4 ++--
  1 files changed, 2 insertions(+), 2 deletions(-)
[sc-gj@sc-miho helloWorld]$ █

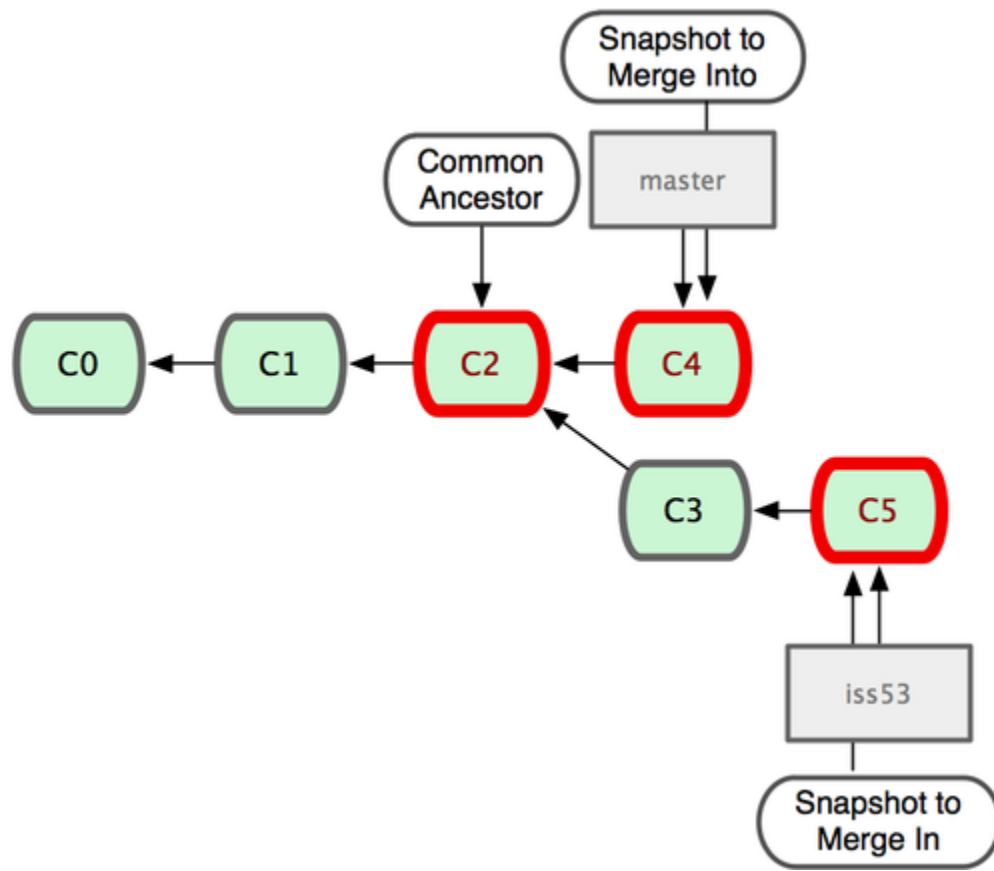
```

We no longer need the hotfix branch, so we delete the branch using git branch and the -d option.

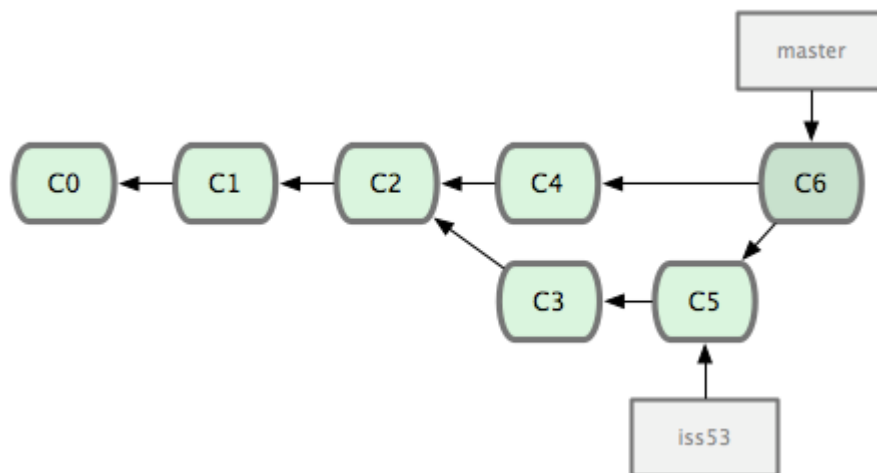
```
git branch -d hotfix
```

Merging is not always as stright forward in the second senario we have two branches each with changes i.e. our master and testing branches from earlier. In this case git performs a three way merge between the tips of the branches and the common ansestor. Unlike other VCS git figures out the most appropriate common ansestor (working this out manually one of the difficult parts of using CVS). Having performed the merge it creates a new snapshot of the result and creates a new commit point (known as the merge commit,) special in that it has more than one parent.

This is illistrated by the following:



With merging leads to:



IMPORTANT: I should stress at this point that when we branch/merge the changes in the branch we merge may not be our own. Consider for example if we had checked main.c out from a central server and so had another developer. When we come to push our local commits we find the other developer has already committed a set of changes and we have to merge these into our local version before we update the upstream version on the server. This is something we'll look at a little later.

Merging with conflicts

Simple merges, especially on single developer projects rarely cause problems. There are also few problems during the early stages of development where bottom up development is being used. Simply because developers are working on separate classes or packages. If Git cannot merge the branches it halts part way and provides information on the files it cannot merge. Git will merge any file where different lines have been changed but not the same line. Remember that even if the merge works the result may need work before it will build!

Git modifies the files in your working copy to contain both the version you checked out and that you tried to merge with, marked as HEAD and with the branch name respectively.

```
<<<<<< HEAD:filename
```

Marks the start of the original block while

```
=====
```

marks the end of the original and the start of the version from the branch we're merging in.

If this was merged with a branch called fred, the section from the merged branch would look like this.

```
>>>>> fred:filename
```

We can verify which of our files need to be merged using git status (we'll look at an example later).

I'm only going to cover manual merging of conflicts in these notes, though there are a large number of merge tools available. Git allows the configuration of a default merge tool which can be launched using 'git mergetool'. Manually fixing the conflicts involves removing which even of the conflicting lines we don't want (or creating a version based on a combination of the two). We then add this file to the staging area using 'git add' then commit.

The default commit message includes some information on the branches being merged and the conflicts which this merge created. Clearly we should aim to augment this to provide information as to exactly what happened.

Time for an example, earlier I created a testing branch and worked on this branch in addition to the master. Here I intentionally modified the same lines in both branches which will result in a conflict when I try to merge them.

First, let's check what branch we're in (I think we should be in the master, but best to check).

```
git branch
```

If you're not in the master, move to the master branch using 'git checkout master' once you're there try to merge in the testing branch.

```
git merge testing
```

The result is a 'Conflict' error message, which is what we want.

```
[sc-gj@sc-miho helloWorld]$ git branch
hotfix
* master
testing
[sc-gj@sc-miho helloWorld]$ git branch -d hotfix
Deleted branch hotfix (was 91e09c9).
[sc-gj@sc-miho helloWorld]$ git branch
* master
testing
[sc-gj@sc-miho helloWorld]$ git merge testing
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.
[sc-gj@sc-miho helloWorld]$ █
```

****MISSING****

I should have called [git status](#) here to show that this reports the file which have not been merged please try this yourselves and contrast the output with that after the file has been changed.

Opening the file with vi we see the expected mark up tags as described above.

```
int main()
{
<<<<<<< HEAD
    printf("Goodbye cruel world\n");
=====
    printf("Hello world\n"); //Made a potentially conflicting change
>>>>>>> testing
}
```

I want to keep the version from the head, so I delete the lower printf and the merge augmentations (i.e. <<<<<<< HEAD, ===== and >>>>>>> testing). Then save and close the file.

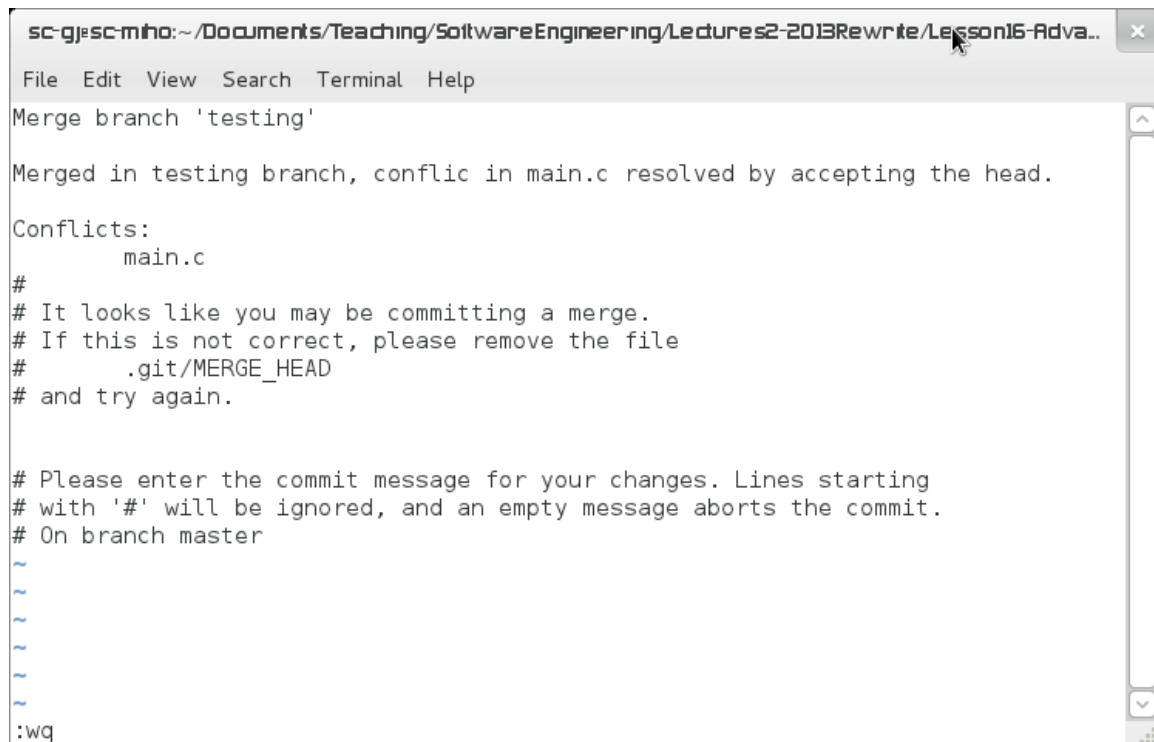
Using `git status` we can see that the files state has changed to modified, prior to this it would indicate that the file was unmerged.

```
[sc-gj@sc-miho helloWorld]$ vi main.c
[sc-gj@sc-miho helloWorld]$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      main.c
#
no changes added to commit (use "git add" and/or "git commit -a")
[sc-gj@sc-miho helloWorld]$ git add main.c
```

Next we add the file/s (remember there may be many files which cause conflicts,) and use `git commit` to commit them. We'll not use `-m` this time as `git` produces a reasonably useful default message which its best to make use of.

```
git add main.c
git commit
```

This opens an editor window (in most cases this will be `vi`), enter a comment (i to move in to insert mode, enter the text, escape to move into command mode :wq to write then quit).

A screenshot of a terminal window titled "sc-gj@sc-miho: ~/Documents/Teaching/SoftwareEngineering/Ledures2-2013Rewrite/Lesson16-Adva...". The terminal shows the output of a git merge command. It starts with "Merge branch 'testing'", followed by "Merged in testing branch, conflic in main.c resolved by accepting the head." Then it shows "Conflicts:" with "main.c" listed below it. There are several lines of instructions: "# It looks like you may be committing a merge.", "# If this is not correct, please remove the file", "# .git/MERGE_HEAD", and "# and try again." Below these, it says "# Please enter the commit message for your changes. Lines starting", "# with '#' will be ignored, and an empty message aborts the commit.", and "# On branch master". There are four tilde (~) characters representing empty lines for the commit message. At the bottom, the text ":wq" is visible, indicating the user has written and quit the editor.

Done! The merge is successful!

Merging is an important process, when working with remote repositories and multiple users we often need to merge our branch (origin – the version of the remote repository we checked out) with the remote's master branch. To integrate our changes into the repository. If no one else has modified the files then it's simply a fast forward (as seen above) otherwise it's a three-way merge and the potential exists for conflicts. However despite the addition of remote repositories and other users the process is exactly the same.

Working with Shared Remotes

To make this as 'real world' as possible I'm going to work with a remote repository on the server shared between a number of users. This is essentially how your group project is going to work, I'll start by going through exactly how to setup a shared project.

I'll then setup a scenario in which

1. Two developers (A and B) clone/pull the same remote version, each has a branch.
2. Developer A pushes their modified local repository to the server, moving on the master branch
3. Developer B tries to push their changes and fails (has to merge their changes into the current master branch)

Creating a Shared Repository

Sharing Git repositories is a bit of a minefield in terms of the initial setup as there are a number of ways it can be achieved, depending on the size of the group, access controls required and whether the group has ssh access to the host machine.

As for the single user repository we created last week there is some setup required on the server-side. So we use ssh to connect to the server.

ssh username@sc-largo.sid.sihe.ac.uk

This takes us to our home directory /home/username we need to traverse to the /home/git directory where there are some folders one for each of the SEP groups. If there are not folders tell me and I'll make some :-D

```
cd /home/git  
ls
```

Change directory to your group's folder

```
cd /home/git/yourgroup
```

We can now create a bare repository i.e. one with no working directory of files, only a database and snapshots. This is very similar to what we did previously when creating our own private remote repositories, however this time we add an additional flag to indicate that the directory is shared. The sets various flags on the files so that the group owner is the same as the directory in which you create the project, allowing those in the same group (your SEP group) to access the files via Git.

```
git init --bare --shared projectName.git
```

In this case the .git at the end of the project name is significant (do not leave it out).

Users can now check out the project from other machines on the network (unfortunately not via the internet due to security restrictions at SMU). However the local nature of Git allows students to check out the latest version of the group projects and work on them at home before merging them back in, so this shouldn't be a problem.

We can now log out of ssh using exit.

```
exit
```

Clone the repository

Back on our working machine (rather than the server,) we clone using the URL of the repository including the username we are using to access the server via ssh. This is identical to our the previous tutorial when working with private remote repository.

```
git clone username@sc-largo.sid.sihe.ac.uk:/home/git/groupName/projectName.git
```

The first time we do this git will notify us we've checked out a remote repository, we know this, its not an error, so ignore it.

Merging with Remotes

Now lets return to the scenario above, user A (gjenkins) and user B (hpotter) have both cloned a repository at some point (this could be the same empty project or one which has been built up over a period of months/years).

User A makes some changes, commits them and pushes them to the remote. For demonstrative purposes I've interrogated git to find out the current branch and available remotes before doing this.

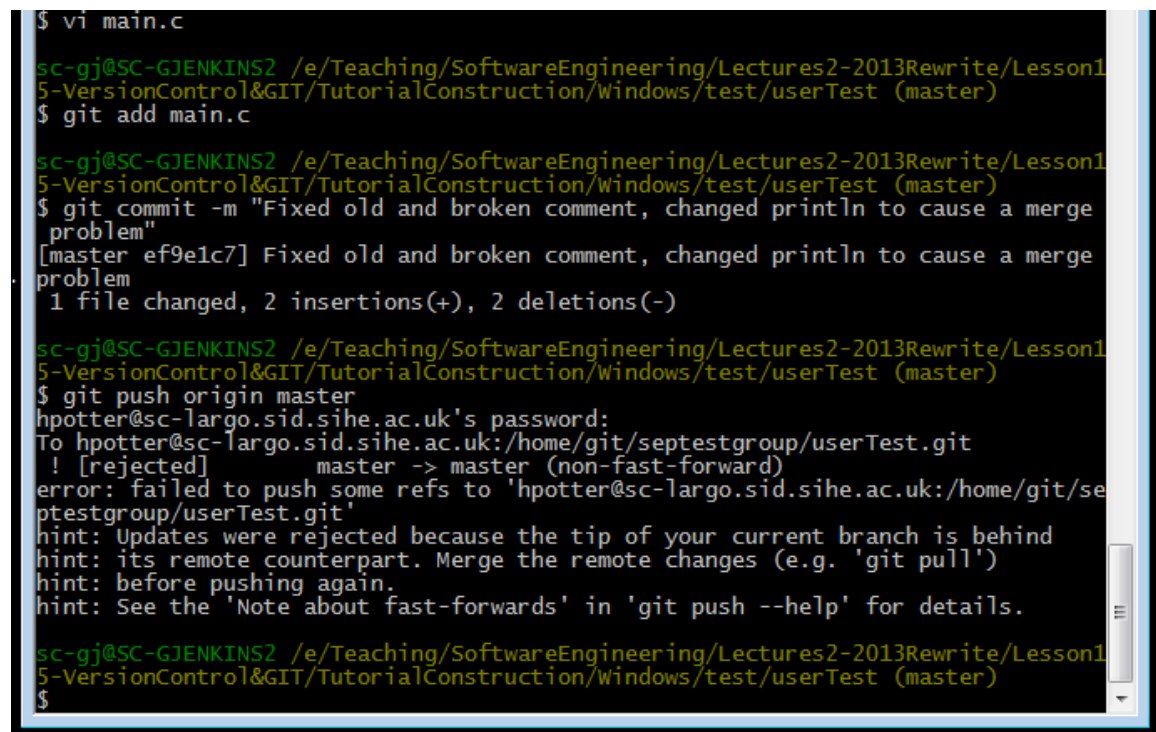
```

[sc-gj@sc-miho userTest]$ vi main.c
[sc-gj@sc-miho userTest]$ git add main.c
[sc-gj@sc-miho userTest]$ git commit -m "modification to message"
[master d1b2df0] modification to message
 1 files changed, 1 insertions(+), 1 deletions(-)
[sc-gj@sc-miho userTest]$ git remote
origin
[sc-gj@sc-miho userTest]$ git branch
* master
[sc-gj@sc-miho userTest]$ git push origin master
gjenkins@sc-largo.sid.sihe.ac.uk's password:
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 334 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To gjenkins@sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest.git
    ad1622a..d1b2df0  master -> master
[sc-gj@sc-miho userTest]$ █

```

Note here the name of the remote and the branch we're pushing (origin and master). This proceeds with no problems, it's essentially a fast forward.

Now, user B (hpotter) cloned (or pulled) a copy of the repository at the same time as user A (gjenkins) and is unaware that their original is now outdated. Clearly if Git simply allowed user B to push the changes then A's changes would be lost. What happens?



```

$ vi main.c

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$ git add main.c

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$ git commit -m "Fixed old and broken comment, changed println to cause a merge
problem"
[master ef9e1c7] Fixed old and broken comment, changed println to cause a merge
problem
 1 file changed, 2 insertions(+), 2 deletions(-)

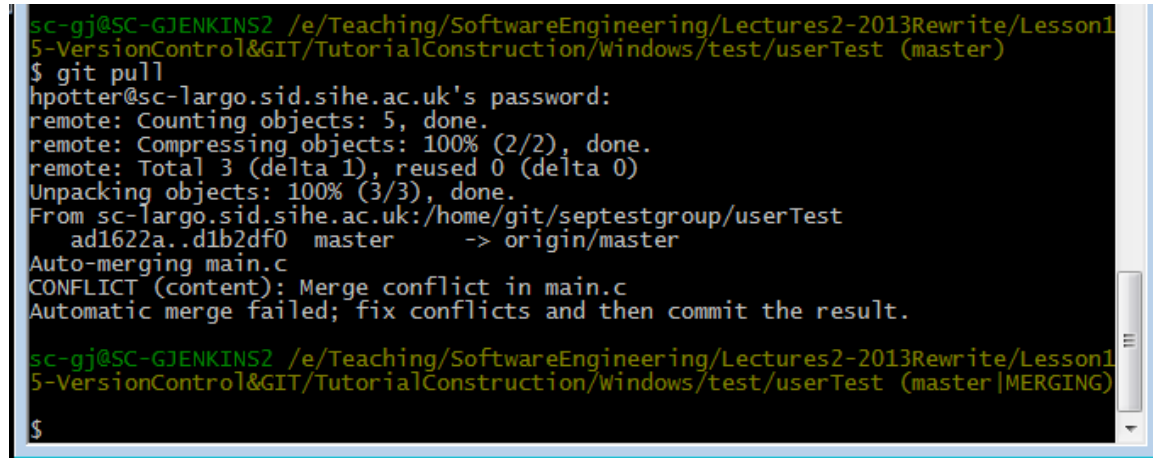
sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$ git push origin master
hpotter@sc-largo.sid.sihe.ac.uk's password:
To hpotter@sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'hpotter@sc-largo.sid.sihe.ac.uk:/home/git/se
ptestgroup/userTest.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$

```

The error message here is pretty self explanatory, Git tried to add this commit to the chain and found that our branch is behind that on the remote! Essentially we need to fetch the remote version and merge it with our own. Git allows these operations to be performed separately, or in one go using 'git pull'.

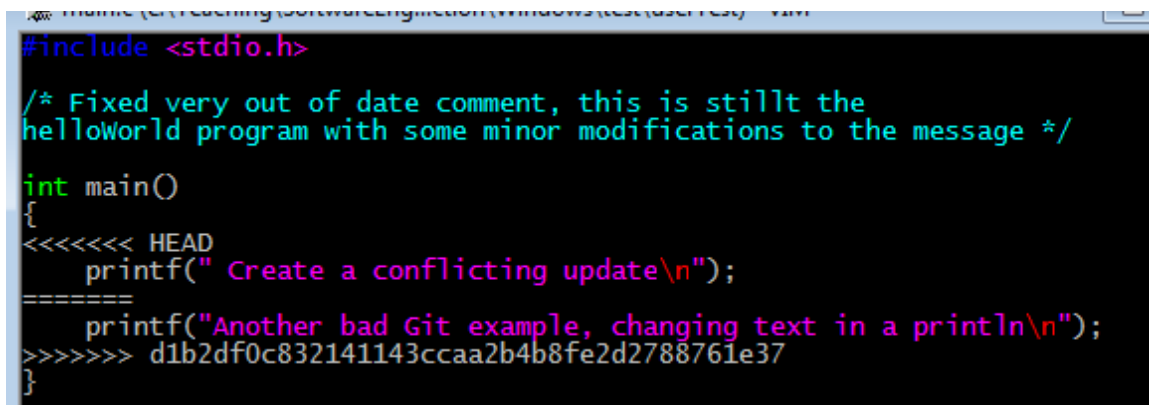
So we need to perform a git pull.



```
sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$ git pull
hpotter@sc-largo.sid.sihe.ac.uk's password:
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest
   ad1622a..d1b2df0  master    -> origin/master
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master|MERGING)
$
```

This message should seem familiar, fetching and merging results in a conflict (n.b. 'git pull' is essentially a 'git fetch' followed by a 'git merge'). This is the same as our earlier merge conflict example and the solution again is to open the file (with vi or other editor) and make the required changes – removing tags and ensuring the results is the result of merging the two files together.



```
#include <stdio.h>

/* Fixed very out of date comment, this is stillt the
helloWorld program with some minor modifications to the message */

int main()
{
<<<<<<< HEAD
    printf(" Create a conflicting update\n");
=====
    printf("Another bad Git example, changing text in a println\n");
>>>>>>> d1b2df0c832141143ccaa2b4b8fe2d2788761e37
}
```

This is actually quite interesting, I made two changes to one of the files (i.e. I changed the comment at the top). As this did not change in both files Git has merged this change in without registering the conflict, however where the println had been changed in two files we have a conflict. Note that rather than the branch name we get the SHA-1 hash of the commit.

Once the changes are complete we can stage and commit the resulting file/s (main.c in this case) and push the new updated version to the remote.

```
$ vi main.c

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master|MERGING)

$ git add main.c

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master|MERGING)

$ git commit
[master 6fcdc68] Merge branch 'master' of sc-largo.sid.sihe.ac.uk:/home/git/sept
estgroup/userTest

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
$ git push origin master
hpotter@sc-largo.sid.sihe.ac.uk's password:
Permission denied, please try again.
hpotter@sc-largo.sid.sihe.ac.uk's password:
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 854 bytes, done.
Total 6 (delta 1), reused 0 (delta 0)
To hpotter@sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest.git
    d1b2df0..6fcdc68  master -> master

sc-gj@SC-GJENKINS2 /e/Teaching/SoftwareEngineering/Lectures2-2013Rewrite/Lesson1
5-VersionControl&GIT/TutorialConstruction/Windows/test/userTest (master)
```

The server copy is now up-to-date, we've resolved the changes and are effectively sharing a single code base with another user. You can try this yourself with other members of the group, create a test project (or more than one) between you and experiment with making changes to a single file in parallel.

Final observation, looking at the log for the repository. We can see what was changed and by who, this gives us an insight into what was done by who. This can be used to provide data on the productivity of developers or by unscrupulous lecturers to mark group assignments.

I've used

```
git log > log.txt
```

as this is too much info to screen shot. What we should not do is that there is more than one author committing to the repository.

```
commit 6fcdc6876c8ed633002880c958fbff57a5263d84
```

```
Merge: ef9e1c7 d1b2df0
```

```
Author: Harry Potter <hpotter@linux.smu>
```

Date: Wed Aug 29 16:31:15 2012 +0100

Merge branch 'master' of sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest

Merged my old branch into the latest version from the remote, to push to

the server.

Conflicts:

main.c

commit ef9e1c7855072bc1ad17543979ecd87c9953a412

Author: Harry Potter <hpotter@linux.smu>

Date: Wed Aug 29 15:34:27 2012 +0100

Fixed old and broken comment, changed println to cause a merge problem

commit d1b2df0c832141143ccaa2b4b8fe2d2788761e37

Author: sc-gj <glenn.l.jenkins@smu.ac.uk>

Date: Wed Aug 29 15:25:25 2012 +0100

modification to message

commit ad1622af5a7a1b807e1f295263f73e94cd217e21

Merge: 74d1341 e74d79d

Author: sc-gj <glenn.l.jenkins@smu.ac.uk>

Date: Wed Aug 29 14:44:18 2012 +0100

Merge branch 'master' of sc-largo.sid.sihe.ac.uk:/home/git/septestgroup/userTest

Merged changes into updated master.

Conflicts:

main.c

commit 74d13416003adf2173bef3a7fc147c774b656700

Author: sc-gj <glenn.l.jenkins@smu.ac.uk>

Date: Wed Aug 29 14:42:45 2012 +0100

Modifiaiton to output message

commit e74d79df5dae5000dfdd6d58d14547074d5b8a81

Author: Harry Potter <hpotter@linux.smu>

Date: Wed Aug 29 14:40:59 2012 +0100

modificaiton to print message

commit 96bb5e7deade99038ba87ca42b2f233125ba92ab

Author: sc-gj <glenn.l.jenkins@smu.ac.uk>

Date: Wed Aug 29 14:01:14 2012 +0100

First commit - user test, checking group members can share repos

Working with Remote Branches

Probably not something you'll be doing with your group assignment but should you need to its covered well here - <http://git-scm.com/book/en/Git-Branching-Remote-Branches>
Essentially all that changes is rather than pushing the master, we push the branch name.