Git

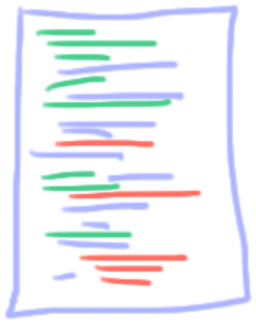
Git is a version control system.

Introduction

So you're working on a problem set. You start with the problem set handout.

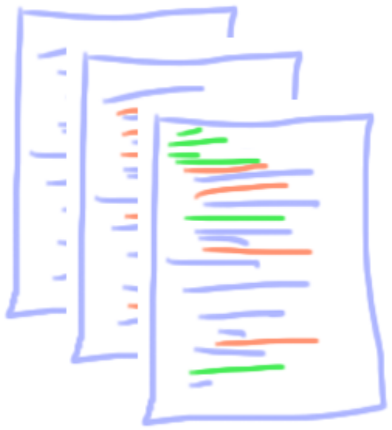
You change it some.

And some more.



But wait a minute: somewhere in this latest version, you made a mistake! Oh no! But there are so many changes—which one is at fault?

If only you could find out *how the file changed*! If only you had a program that remembered **every version** of your file:



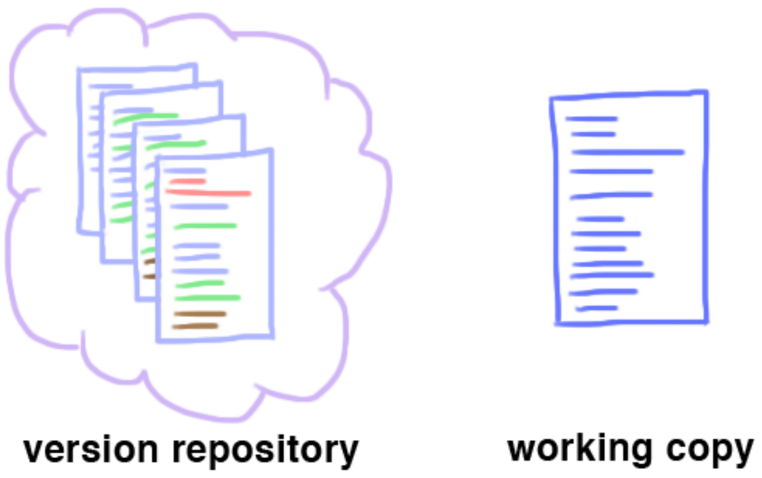
And could highlight differences:

This is what version control systems do. Git is a version control system.

Repositories

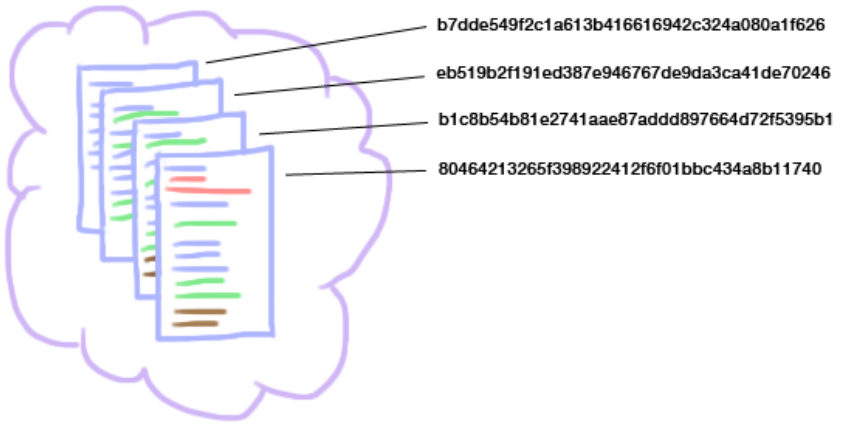
Git works in units called *repositories*. A git repository contains two main pieces: the *version repository* and the *working copy*.

The working copy consists of normal files arranged in a directory structure. The version repository stores previous versions of these files and directories. We'll draw them like this:

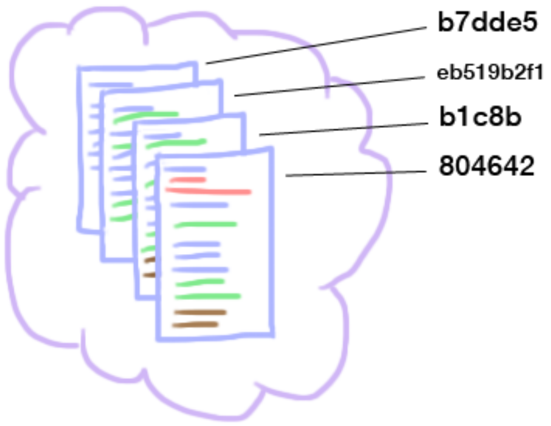


Remember, though, that a working copy usually contains many files, not just one, and each version in the repository is a snapshot of many files' states, not just one file's state.

Each version in the repository is called a *commit*, and each commit has a unique name. The name is a 40-character hexadecimal string, derived from a SHA-1 cryptographic hash of the commit's contents and its previous history. (There are 2160 possible commit hashes.)


b7dde549f2c1a613b416616942c324a080a1f626
eb519b2f191ed387e946767de9da3ca41de70246
b1c8b54b81e2741aae87addd897664d72f5395b1
80464213265f398922412f6f01bbc434a8b11740

Luckily you can abbreviate these names to unique prefixes. Only 5 or 6 characters are usually required to uniquely identify a commit in the current repository, but you can type as many as you want.



Creating a repository

To create a repository in the current directory—for instance, when you're starting a new project—just run `git init`. This will make a blank repository with no commits.

It's more common, though, to make a copy of an existing repository, such as one on code.seas.harvard.edu. The command for that is `git clone URL`, where `URL` names the "upstream" repository you want to clone. For example:

```
% git clone git@github.com:cs61/cs61-psets.git
```

will create a new repository on your machine, placing it in the `cs61-psets` directory. The repository is initialized with a copy of the commits in `git@github.com:cs61/cs61-psets.git`.

You can also create a local copy of a repository with `git clone OLDDIRECTORY NEWDIRECTORY`.

Committing

Say you make an edit to your working copy.

This edit is *not* part of your version repository yet. Git doesn't automatically track every change.

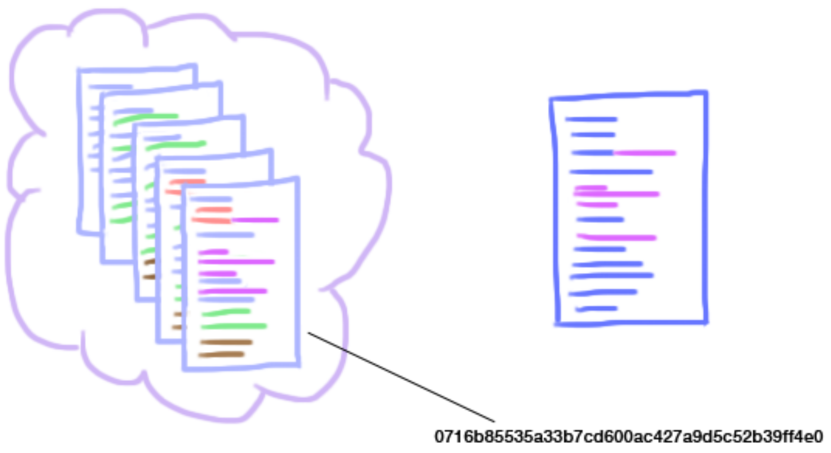
The `git commit -a` command takes a snapshot of your current working copy and saves it in the version repository.

git commit -a

The `-a` means "commit *all* changes in the working copy."

As part of the commit process, you'll be required to enter a *commit message*. This describes the commit. It's best to enter something that will be meaningful to you later, but don't let that stop you from committing early and often.

Once you've entered a commit message, git will create the new commit's hash



and report it to you.

```
% git commit -a
[master 0716b85] More
 1 file changed, 11 insertions(+), 11 deletions(-)
```

Logging

The `git log` command reports all of the commits in your current history. It lists them in reverse order, so the most recent commit comes first.

```
% git log
commit 1c1bbc3eb93e0ea44d138cb35c77f225b6ce7b54
Author: Eddie Kohler <ekohler@gmail.com>
Date:   Thu Sep 20 20:42:04 2012 -0400

    compare.pl: {Spaces}??? -- the ??? can be preceded by any number of spaces.

    Thanks to Kenneth Ho.

commit a20ef24d4824c499db2d2211f87281ce2065446b
Author: Eddie Kohler <ekohler@gmail.com>
Date:   Tue Sep 18 08:52:42 2012 -0400

    Add and update boundary write error tests.

commit 6240b8872becb52531d92ff373f40f4aea291365
Author: Eddie Kohler <ekohler@gmail.com>
Date:   Fri Sep 14 21:23:34 2012 -0400

    Initial commit of problem set 1.
```

My favorite way to log commits is `git log -p`. This shows, with each commit, the *difference* between that commit and the previous commit. The difference is called a **patch** (thus the `-p`).

```
% git log -p
commit 1c1bbc3eb93e0ea44d138cb35c77f225b6ce7b54
Author: Eddie Kohler <ekohler@gmail.com>
Date:   Thu Sep 20 20:42:04 2012 -0400

    compare.pl: {Spaces}??? -- the ??? can be preceded by any number of spaces.

    Thanks to Kenneth Ho.

diff --git a/pset1/compare.pl b/pset1/compare.pl
index b9faaed..6e5fbf8 100644
--- a/pset1/compare.pl
+++ b/pset1/compare.pl
@@ -18,6 +18,7 @@ while (defined($_ = <EXPECTED>)) {
                  "r" => "", "match" => []};
        foreach my $x (split(/(\?\?\?|\?\?\{.*?\}(?:=\w+)?\?\?)/)) {
            if ($x eq "???") {
+               $m->{r} =~ s{(\\ )+$}{\\s+};
                $m->{r} .= ".*";
            } elsif ($x =~ /\A\?\?\{(.*)\}=(\w+)\?\?\z/) {
                $m->{r} .= "(" . $1 . ")";

commit a20ef24d4824c499db2d2211f87281ce2065446b
Author: Eddie Kohler <ekohler@gmail.com>
Date:   Tue Sep 18 08:52:42 2012 -0400

    Add and update boundary write error tests.

diff --git a/pset1/test020.c b/pset1/test020.c
index 0f59130..42e0ca5 100644
--- a/pset1/test020.c
+++ b/pset1/test020.c
@@ -2,7 +2,7 @@
 #include <stdio.h>
 #include <assert.h>
 #include <string.h>
-// test020: check for wild writes off the end of the allocated block.
+// test020: check for boundary write errors off the end of an allocated block.

 int main() {
     int *ptr = (int *) malloc(sizeof(int) * 10);
diff --git a/pset1/test027.c b/pset1/test027.c
...
```

It is valuable to learn how to read patches. Compared to the previous commit, a commit deletes the lines indicated by `-` and adds new lines indicated by `+`. Lines that start with a space weren't changed; they're provided for context. (Learn more about diffs)

Branches

Git commits are arranged into lists called *branches*. The most important branch is `master`. This is the default branch; it's created whenever you initialize a new git repository.

Remember what git printed when you committed?

```
% git commit -a
[master 0716b85] More
 1 file changed, 11 insertions(+), 11 deletions(-)
```

Now you know what `master` means. This commit changed the `master` branch.

Git can support many branches, not just `master`. Furthermore, a branch can involve actual branching points, rather than a linear sequence of commits.

A branch name, such as “`master`”, also names a commit, namely the most recent commit on that branch. This most recent commit changes over time as you add new commits. Commit hashes, in contrast, are stable: a commit hash always means the same version.

Diffs

[Main article on diffs][2018/Diff]

Comparing different versions of your code is incredibly valuable, and one of the best reasons to use a version control system. Git presents differences in what’s called the *unified diff* format.

There are a couple main patterns for calling `git diff`.

- Run `git diff` (with no arguments) to see the differences between your current commit (plus the staging area; see below) and your working copy.
- Run `git diff COMMIT` to see the difference between a named commit and your working copy.
- Run `git diff COMMIT1 COMMIT2` to see the differences between the two named commits.

If `git diff` reports too much information, then try `git diff FILENAME` (or `git diff COMMIT1 COMMIT2 FILENAME`), which prints just the differences to FILENAME.

Undoing changes

To undo changes to a file’s working copy, use `git checkout FILENAME`. This permanently throws away your changes and rolls back to the most-recently-committed version of `FILENAME`. You can also roll back to an earlier version: try `git checkout COMMIT FILENAME`.

Adding files

`git commit -a` commits changes to all the files that git knows about. But you may sometimes want to add a *new* file to the repository—to tell git to start tracking another file. This requires the `git add` command:

```
% git add FILENAME
% git commit -a
```

The `git add` command doesn’t actually commit the change. Instead, git adds the file to an invisible *staging area* that holds changes until the *next* commit. That’s why we follow `git add` with `git commit -a`.

Removing and renaming files

To remove a file from the version repository, use `git rm FILENAME`. Like `git add`, this affects the working tree and staging area; a separate `git commit` is required to really change the version repository. To rename a file, use `git mv OLDFILENAME NEWFILENAME` (and then commit).

Status

The `git status` command reports on the current state of your working copy. Here's an example:

```
% git status
# On branch master
# Your branch is ahead of 'origin/master' by 6 commits.
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   pset1/README.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       pset1/test.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

`git status` is a good overview, and its parenthetical comments are helpful too. Its most valuable output is the "`Untracked files`" section. This tells you which files in your working copy have no equivalents in the version repository. Forgetting to `git add` a file is one of the most common git errors. Running `git status` once in a while and checking the untracked files section is good practice for everyone.

Partial commits

Sometimes you don't want to commit your entire working copy. For instance, maybe your `README.txt` is ready to share with your partner, but your `m61.c` is broken and currently crashes all the tests.

Tell `git commit` the files you want to commit, instead of passing `-a`, and it will commit just the named files. For instance:

```
% git commit README.txt
```

Alternately, you can stage files for commit using `git add`. Just as when you add files, this tells git that the *next* commit, whenever it happens, should include the changes currently in `README.txt`:

```
% git add README.txt
```

`git status` will show that `README.txt`'s change has been staged. It also says how you could unstage the file.

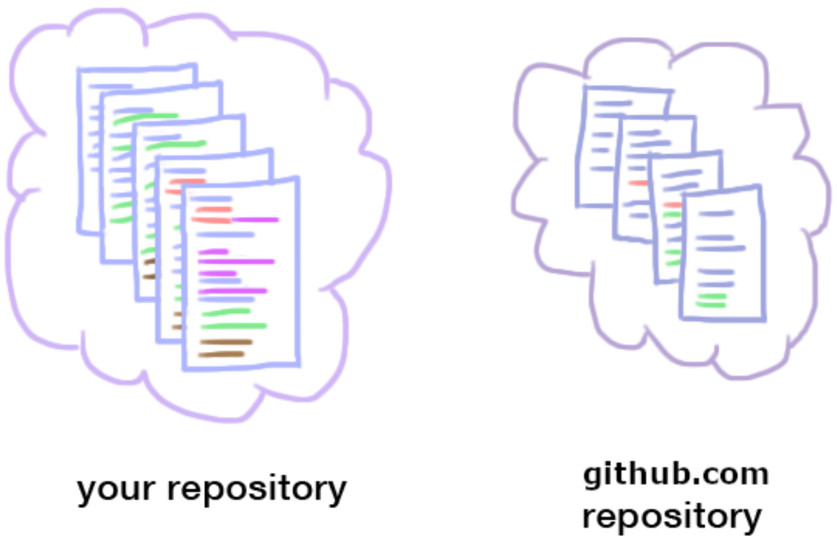
```
% git status
# On branch master
# Your branch is ahead of 'origin/master' by 6 commits.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   pset1/README.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       pset1/test.txt
```

Once you're ready, `git commit` will commit all staged changes.

Finally, the wicked cool `git add -p` interactively prompts you to select *individual modifications* to stage.

Remotes

A git repository can associate with other repositories called *remotes*. Remotes are stored elsewhere—for example, on GitHub, or code.harvard.edu.



For example, in CS 61, you make commits in the repository on your CS 61 VM. But to turn in your code, and to protect your work against VM problems, you save your changes into a git repository hosted on GitHub. Your Appliance repository sees your GitHub repository as a remote.

The `git remote` command lists a repository's remotes. If you used `git clone` to create your repository, you will see at least one remote, called `origin`:

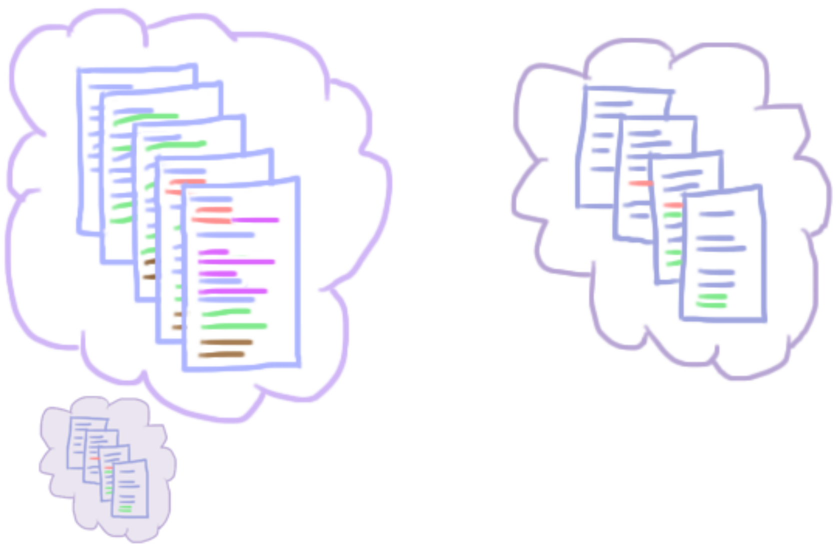
```
% git remote
origin
```

The `origin` remote keeps track of the original source of your repository. In CS 61, this will most likely be a repository hosted on GitHub.

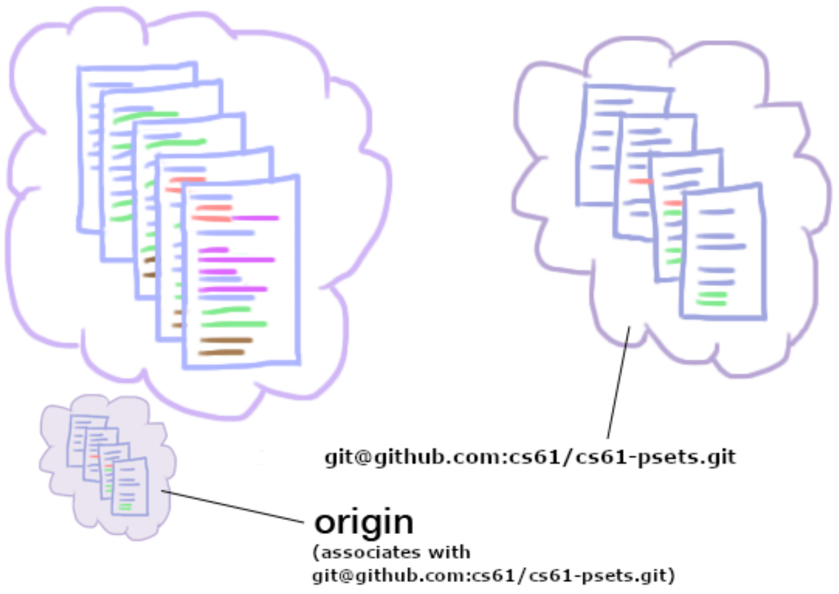
You can add a new remote whenever you like using `git remote add`. For instance, add a remote "`handout`" for CS 61 handout code:

```
% git remote add handout git@github.com:cs61/cs61-psets.git
```

Your git repository maintains a shadow copy of all the data in each remote.



The remote name, such as `origin`, refers to the shadow copy.


git@github.com:cs61/cs61-psets.git
origin
(associates with
git@github.com:cs61/cs61-psets.git)

Since git stores shadow copies of remotes, it can also display their contents. Many of the commands we've seen work well on remotes. For example, say you have a `handout` remote. Then try:

```
git log handout/master
```

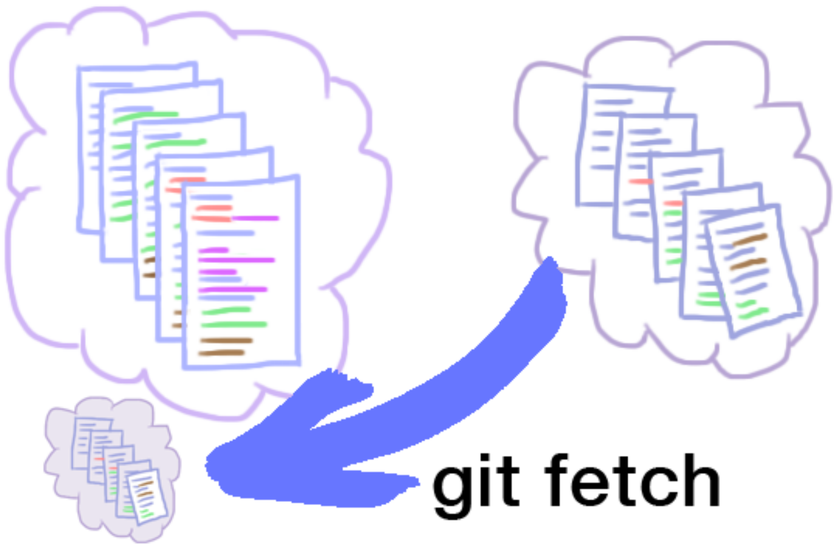
Print log of commits to the `handout` remote's `master` branch.

```
git diff handout/master master
```

Compare the most recent handout code (`handout/master`) with your most recent commit (`master`).

Shadow copies get stale as remote repositories change.

The `git fetch` command updates your repository's shadow copy from the actual remote repository. `git fetch REMOTENAME` updates your shadow copy of the `REMOTENAME` remote; `git fetch --all` updates all remotes.



`git fetch` **does not change your version repository.** It only changes shadow copies. Your working copy, and your repository's version history, remain unchanged.

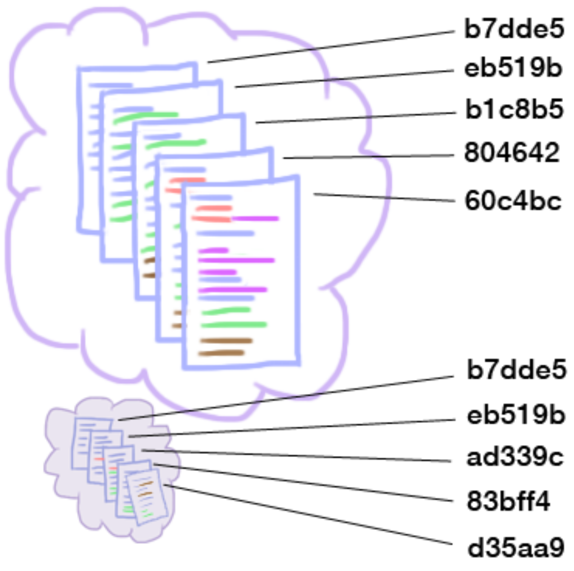
Merges

A *merge* joins development streams together. For instance:

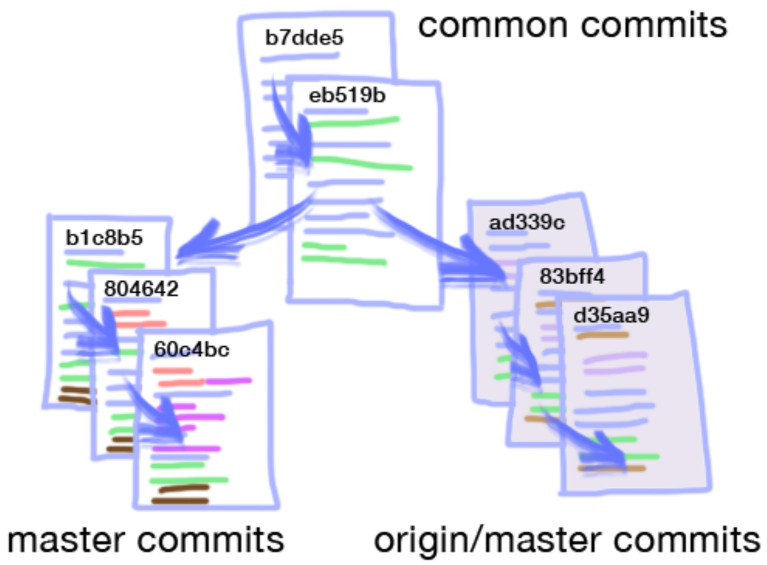
- You and a partner split up work on a problem set, and commit your work to different repositories. A merge can combine your changes.
- The instructors make changes to a problem set, and you have already begun work. A merge will combine your changes with the instructors' updates.

Let’s take an example: `git merge origin/master`. This merges the current branch with `origin/master`, which is the current value of the `master` branch in the shadow copy of the `origin` remote.

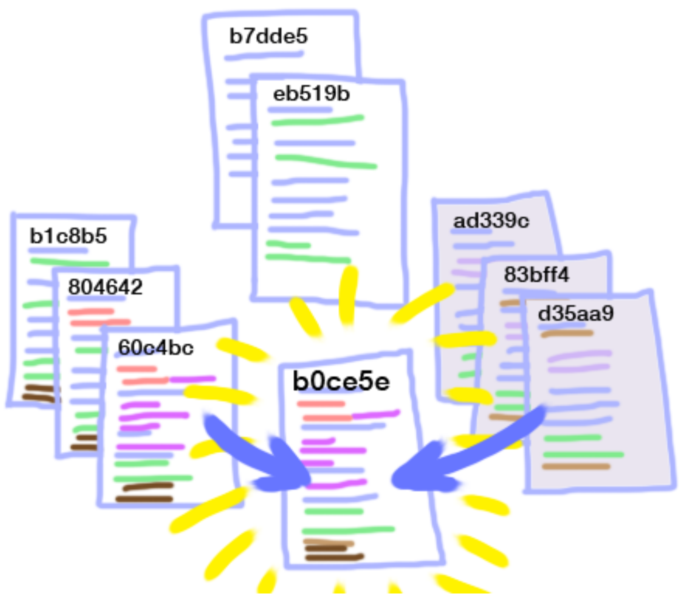
First, git checks where the two repositories diverged. Let’s say the commits look like this:



The common commits are b7dde5 and eb519b. After that, the histories diverge. Git thinks of these histories as a branched graph.



The `git merge origin/master` command combines the branches into a new commit that covers *both* histories.



If you look at `git log` after merging, you can see both histories:

```
commit b0ce5ed01c688921ed0eae69b6bd571619935fad
Merge: 60c4bc9 d35aa90
Author: Eddie Kohler <kohler@seas.harvard.edu>
Date:   Thu Oct 4 14:25:55 2012 -0400

    Merge branch 'master' of git@code.seas.harvard.edu:cs61/cs61-psets.git
```

`git merge` is a form of `git commit`; it will allow you to edit the merge message if you like.

Conflicts

In an ideal world, all merges would complete automatically. But in the real world, sometimes the merged branches edit exactly the same portion of the code. Git is not smart enough to figure out which edits to use, so it stops and reports a *conflict*. You must edit the files to *resolve* the conflict by hand.

Here's an example failed merge:

```
% git merge origin/master
Auto-merging pset1/test004.c
CONFLICT (content): Merge conflict in pset1/test004.c
Auto-merging pset1/test003.c
CONFLICT (content): Merge conflict in pset1/test003.c
Automatic merge failed; fix conflicts and then commit the result.
```

ALL CAPS ARE SCARY, WHAT SHOULD WE DO? Well, maybe you don't want to do a merge after all! Type `git merge --abort` and git will restore the pre-merge version. But probably you want to resolve the conflict. To do this, you need to know how git shows conflicts. Here's the conflicted state of `pset1/test003.cc`:

```
#include "m61.hh"
#include <stdio.h>
// test003: active allocation counts.

int main() {
    void *ptrs[10];
    for (int i = 0; i < 10; ++i)
<<<<<<< HEAD
        ptrs[i] = malloc(i | 1);
=======
        ptrs[i] = malloc(i + 1);
>>>>>>> origin/master
    for (int i = 0; i < 5; ++i)
        free(ptrs[i]);
    m61_printstatistics();
}

//! malloc count: active          5   total         10   fail        ???
//! malloc size:  active        ???   total        ???   fail        ???
```

The `<<<<<<<`, `=======` and `>>>>>>>` lines are *conflict markers*. Git adds conflict markers around the conflicting edits. The text between `<<<<<<<` and `=======` represents your edits—the ones in `HEAD` (git's name for the current branch). The text between `=======` and `>>>>>>>` represents the other branch's edits—here, the ones in `origin/master`.

To resolve a conflict, just choose which edit to keep and delete the other edit. For example, to keep your edit, remove the markers and the `origin/master` code, to produce:

```
... for (int i = 0; i < 10; ++i)
        ptrs[i] = malloc(i | 1);
    for (int i = 0; i < 5; ++i)...
```

You might also want to create a new version that combines the edits.

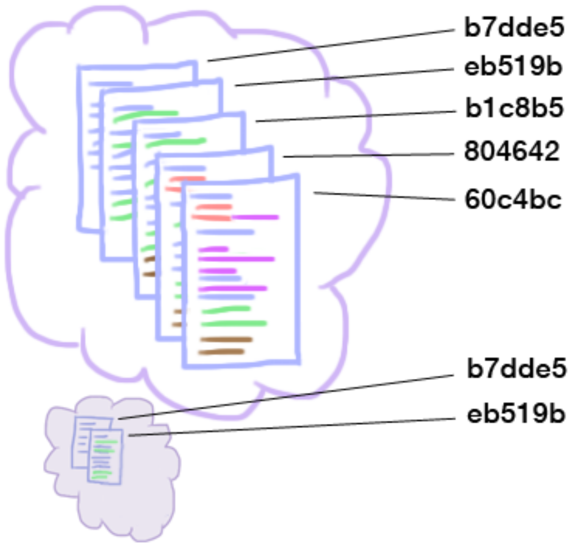
```
... for (int i = 0; i < 10; ++i)
        ptrs[i] = malloc((i | 1) + 1);
    for (int i = 0; i < 5; ++i)...
```

Once you've edited all the conflicts (use `git status` to check your work), commit the result with `git commit -a`. (Just like the merge suggested!) The commit message is pre-loaded with merge information.

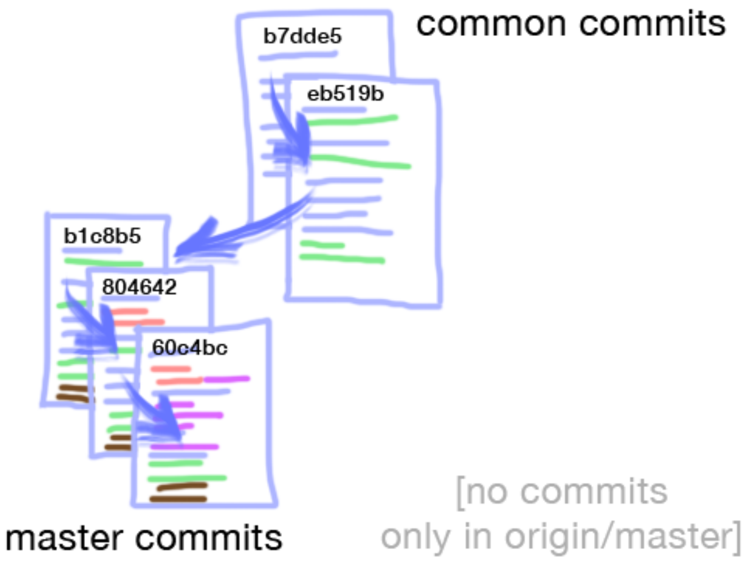
Conflict resolution can get quite hairy. Luckily, there are lots of visual tools available for editing conflicts; try `git mergetool`. We also find that `git rebase` is better at handling certain types of conflicts, but a description of rebasing will have to wait.

Fast forwards

Merging has a best case: one of the branches is a subset of the other. For example, consider `git merge origin/master` with these commits:

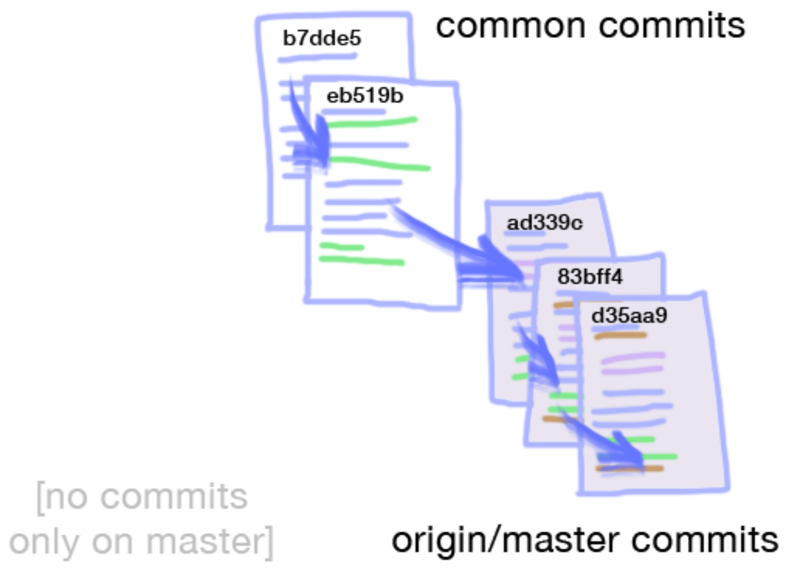


The `origin/master` history is a subset of the `master` history:

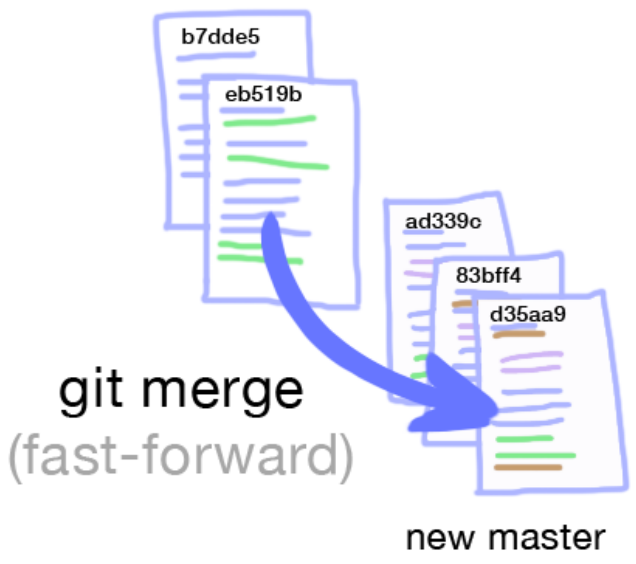


There's nothing for the merge to do, so it will report "`Already up-to-date.`"

These commits are also easy to merge:



Here, the `master` history is a subset of the `origin/master` history. A `git merge origin/master` command will *fast-forward* the `master` branch to the latest version on `origin/master`, without creating a new commit:



A fast-forward can never create a conflict.

Pulls

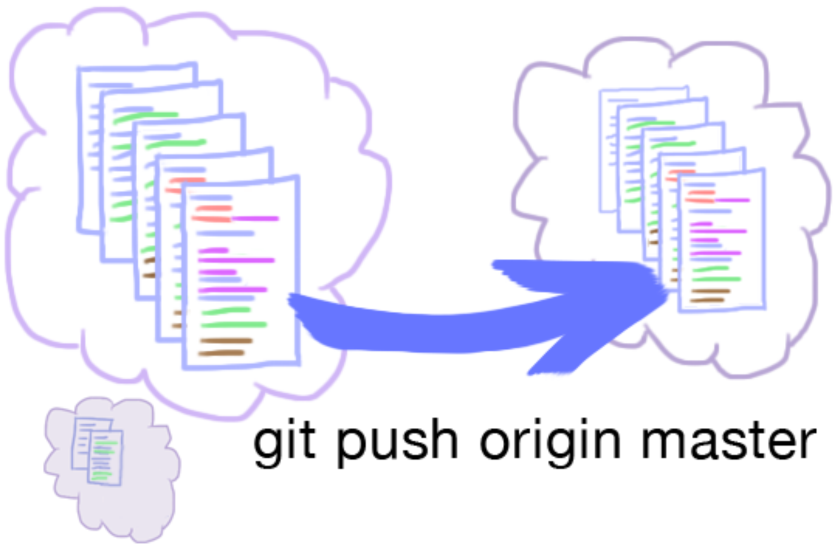
The convenience command `git pull` combines `git fetch` and `git merge`. The command `git pull ORIGIN BRANCH` means almost the same thing as `git fetch ORIGIN; git merge ORIGIN/BRANCH`.

Pushes

`git push` is the inverse of `git fetch`. Where `git fetch` updates a shadow repository from the remote, `git push` takes your *local* version and copies it to a *remote* repository. You use this to save your local changes in a more permanent way (for example, on code.seas.harvard.edu).

The command `git push REMOTE BRANCH` will take your local `BRANCH` and push a copy of it onto `REMOTE`'s `BRANCH`. For example, `git push origin master` pushes your local `master` branch to the `origin` remote's `master` branch. For instance, from this state:

`git push origin master` would do this:



Now `origin/master` is the same commit as the local `master`.

Normally `git push` will reject every attempt to push that is not a fast-forward. If it complains, you need to merge first with the remote branch, for instance by `git pull origin master`. Push again after completing the merge.

Pushes don't consider the working copy at all, they only copy committed versions.

More

This should offer you the conceptual tools and enough basic commands to get a lot done. For more on git, check out:

- The git home page
- Git documentation—especially the book
- Git tutorial videos!
- A git cheatsheet
- Commands I'd like to describe in this document: `git branch`, `git stash`, `git rebase`, `git rebase -i`, `git grep`, `git reset`, `git tag`, `git show`, `git commit --amend`.
- Other cool commands: `git bisect`.