

Version Control with `git`

Robert Lupton

26 July 2016

Why Version Control?

Have you ever ...

- 1 Tried to add a feature to a program and broken it so badly you wished you could abandon all your changes?
- 2 Left a program in an unrunnable, or uncomparable state. . . and you needed new results for a conference **now**?
- 3 Tried to work on a program or write a document with one or more other people?

There are solutions. You could:

- 1 Backup frequently, keeping all backups.
- 2 Only ever develop on a copy of your code
- 3 Appoint someone whose job it is to merge all contributions

Version Control Systems (VCSs) offer a far better solution to all these problems. *And* offer many other advantages too.

As a single developer

If you like the backup solution, many versions of `cp` support the `-b` flag¹, so all you need do to set things up is:

```
$ mkdir Backups
```

Then

```
$ cp -b hello.c Backups
hack hack
$ cp -b hello.c Backups
hack hack
$ cp -b hello.c Backups
```

After that marathon session:

```
$ ls -lt Backups/
total 96
-rwxr-xr-x 1 rhl rhl 21797 Sep 21 16:01 hello.c
-rwxr-xr-x 1 rhl rhl 21734 Sep 20 06:12 hello.c.~2~
-rwxr-xr-x 1 rhl rhl 21612 Sep 19 23:57 hello.c.~1~
```

You'll then have to `grep` the file to find the version you had in mind.

If `cp -b` doesn't work you can write a simple script:

```
#!/bin/sh
shopt -u nullglob
for f in "$@"; do
    cp $f Backups/$f~$(date +%F-%T)~
done
```

¹os/x is not one of them

Using git instead

All you need do to set things up is:

```
$ git init  
Initialized empty Git repository in /Users/rhl/TeX/Courses/APC524/.git  
$ git add .
```

Then

```
$ git commit -m "Initial version" hello.c  
hack hack  
$ git commit -m "Made everything global" hello.c  
hack hack  
$ git commit -m "Removed that confusing define" hello.c
```

Those `-m` strings are associated with the *commits* (also called *checkins*), and you can list them with `git log`:

```
$ git log --oneline  
9613186 Removed that confusing define  
8187a17 Made everything global  
140c443 Initial version
```

Thinks: "I could easily add commit messages to my little shell script..." but why should you bother when Linus did the work already?

Details, details, ...

Actually,

```
$ git commit -m "Initial version" hello.c
```

generated a message:

```
Committer: Robert Lupton the Good <rlh@babayaga.astro.princeton.edu>  
Your name and email address were configured automatically based  
on your username and hostname. Please check that they are accurate.  
You can suppress this message by setting them explicitly:
```

```
git config --global user.name "Your Name"  
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with:

```
git commit --amend --reset-author
```

Do what it says, and don't worry about it; you'll never see this message again.

I also have the lines

```
[alias]  
ci = commit  
co = checkout
```

in my **.gitconfig** file:

```
$ git config --global alias.ci commit; git config --global alias.co checkout
```

but I'll try not to inflict them on you.

git clone

I often start projects from scratch with `git init`, but I also often start from someone else's work; in that case the initial command is something like:

```
$ git clone git@github.com:RobertLuptonTheGood/APC524GitLecture
```

rather than

```
$ git init  
$ git add .
```

Even when I'm working alone, I often want to set things up so I can *clone* a remote copy of my project, because I worry about what will happen when I lose my laptop.

There are lots of ways to store git repositories; a convenient way is to use one of the hosting sites such as `bitbucket.com` or `github.com` (I'll use github in this lecture).

Other options include using `ssh`, `httpd`, or the file system.

There are some slides about how to get started with github near the end of this lecture.

git log

You get more information if you don't request `--oneline`:

```
$ git log
```

```
commit 9613186caf30a2694145b298cdc02653b0a90512
```

```
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
```

```
Date:   Mon Sep 24 11:56:15 2012 -0400
```

```
    Removed that confusing define
```

```
commit 8187a17d2785f07bf3332b590a94579ccb0ac7aa
```

```
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
```

```
Date:   Mon Sep 24 11:52:57 2012 -0400
```

```
    Made everything global
```

```
commit 140c4431b48be9884e3586d3f1d421c5b31cd500
```

```
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
```

```
Date:   Mon Sep 24 11:50:38 2012 -0400
```

```
    Initial version
```

That all makes sense, except what are those

```
9613186caf30a2694145b298cdc02653b0a90512
```

```
8187a17d2785f07bf3332b590a94579ccb0ac7aa
```

```
140c4431b48be9884e3586d3f1d421c5b31cd500
```

strings?

Secure Hash Algorithm

It turns out that VCSs such as `git` tend to store *diffs* between files rather than multiple copies of files; `9613186...` is a *SHA-1* of that diff. *SHA* is the 160-bit result of the Secure Hash Algorithm² — for our purposes, it's an almost-certainly unique fingerprint for our set of changes (our "changeset").

²*SHA-0* was withdrawn by the *NSA* for undisclosed reasons, and there are some signs that *SHA-1* isn't quite as strong as the spooks would like. There's also *SHA-2* which has no publicly-known problems.

What have we gained over `cp -b`?

We've associated a message with the saved version; that's useful.
We have used `git clone` as a way to manage off-site backups (we'll see more about that soon); I suppose that that's useful.
We've saved disk space by storing *diffs* rather than copies of files.
At first glance this is an implementation detail, and once upon a time that was true. However `git` doesn't think: "What does my file look like?", it thinks: "What changesets went into the current state of my file?". This'll matter later.
So, what *have* we gained over `cp`?

git diff

We can ask `git` to show us those diffs:

```
$ git diff 140c443..8187a17
diff --git a/hello.c b/hello.c
index ea0dbc7..45bba92 100644
--- a/hello.c
+++ b/hello.c
@@ -2,12 +2,14 @@

#define NITER 10

+int i;                                /* an integer */
+const char *str = "Hello World";
+
+int
main()
{
-   const char *str = "Hello World";
-   printf("What I tell you %d times is true\n", NITER);
-   for (int i = 0; i < NITER; ++i) {
+   for (i = 0; i < NITER; ++i) {
+       printf("%d %s\n", i, str);
+   }
}
```

note that `git` allowed me to abbreviate the SHAs. But even abbreviated and cut-and-pasted, SHAs are a nuisance.

git diff

Fortunately, there's a shortcut; you could have got the same result with `git diff HEAD~..HEAD~` (read: "What's the diff between the version two-commits ago to the one-commit ago?"). Or:

```
$ git diff HEAD~
diff --git a/hello.c b/hello.c
index 45bba92..df38fa3 100644
--- a/hello.c
+++ b/hello.c
-1,15 +1,13
 #include <stdio.h>

-#define NITER 10
-
  int i;                                /* an integer */
  const char *str = "Hello World";

  int
  main()
  {
-   printf("What I tell you %d times is true\n", NITER);
-   for (i = 0; i < NITER; ++i) {
+   printf("What I tell you %d times is true\n", 10);
+   for (i = 0; i < 10; ++i) {
        printf("%d %s\n", i, str);
    }
  }
```

(this is an abbreviation for `git diff HEAD~..HEAD`).

Adding files; git status

We need a Makefile!

```
$ emacs Makefile
$ make
cc -o hello -Wall hello.c
```

Here's a new git command:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Makefile
#       hello
nothing added to commit but untracked files present (use "git add" to track)
```

git status is your friend; type it whenever you want to know where you've got to with git.

The first thing to do is to ignore the machine-generated file **hello**:

```
$ echo hello > .gitignore
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .gitignore
#       Makefile
nothing added to commit but untracked files present (use "git add" to track)
```

Adding files; `git add`

Let's do what they say:

```
$ git add .gitignore Makefile
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   Makefile
#
$ git commit -m "Added Makefile; ignored hello"
[master 4882a1f] Added Makefile; ignored hello
 2 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Makefile
$ git status
# On branch master
nothing to commit (working directory clean)
```

Actually, that wasn't a very good way to do things — it's usually better to split separate functionality into separate commits. You can use `git rebase --interactive` to help you fix such misjudgments, but that's a topic a little bit too advanced to fit in these notes. It's also one of the few things I use a gui for. Try `git gui` or `sourcetree`, which is free from Atlassian.

The Repository

What's in my directory?

```
$ ls -A  
.git          .gitignore    Makefile      hello          hello.c
```

I know about the last four of those, but what about **.git**? It's git's secret stash of my project's history, and is called *the repository*. It's just a directory like any other, full of more-or-less obscure files:

```
$ ls .git  
COMMIT_EDITMSG  FETCH_HEAD    ORIG_HEAD     config         hooks  
info            objects  
COMMIT_EDITMSG~ HEAD           branches      description    index  
logs            refs  
$ cat .git/refs/heads/master  
4882a1fe4f5970fdb07998e77e1c7c68a5e6f047  
$ git log --oneline  
4882a1f Added Makefile; ignored hello  
9613186 Removed that confusing define  
8187a17 Made everything global  
140c443 Initial version
```

git add revisited

You say hello; I say goodbye. So I edit **hello.c** and ask git what's going on:

```
$ git diff
diff --git a/hello.c b/hello.c
index df38fa3..182fc70 100644
--- a/hello.c
+++ b/hello.c
-1,7 +1,7
#include <stdio.h>

int i;                                /* an integer */
-const char *str = "Hello World";
+const char *str = "Goodbye Universe";

int
main()
```

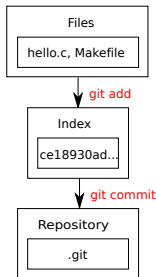
Fine. Let's commit that change:

```
$ git commit -m "Changed sign of greeting"
# On branch master
# 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:   hello.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Nothing happened. And what's that about git add?

The Index

There's another layer, called *the index* (or *staging area* or *cache*)



So to check in my changes I need to first add them to the index, and then commit them to the repository:

```
$ git add hello.c
$ git commit -m "Changed sign of greeting"
[master ce18930] Changed sign of greeting
1 file changed, 1 insertion(+), 1 deletion(-)
```

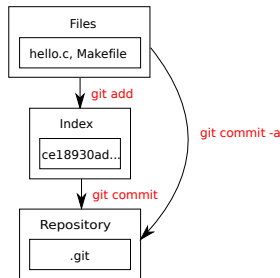
Note that we added the changeset `ce18930`, not the entire file, to the index.

The Index

If you're familiar with `cvs`, `svn`, or `hg` you are probably thinking, "How silly". Maybe you'll be happy to learn that I could have said:

```
$ git commit -a -m "Changed sign of greeting"
```

and skipped the `git add` entirely.



If you specify a filename explicitly you can also skip the add:

```
$ git commit -m "Changed sign of greeting" hello.c
```

The Index

However, you usually have sets of changes to many files that belong together and it makes sense to group them together. What's more, you probably have other changes that you *don't* want to commit (e.g. hacks to disable slow bits of the code that got in the way during testing). The ability to choose what should go into the index, and then to commit the changes all together with a helpful message is very valuable.

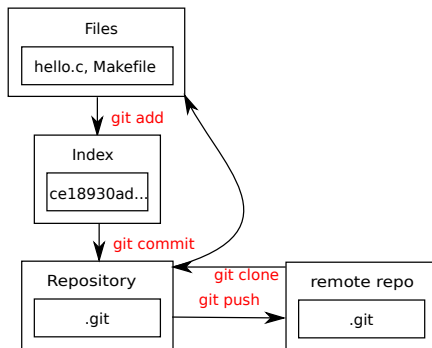
You can control what goes into the index at the single-patch or -line level (`git add -p` or `git gui` or `sourcetree`).

git push

I promised that `git clone` offered protection against disk crashes.
If I say

```
$ git push
```

all my local changes are *pushed back* to the repository whence they were cloned.



Commit Messages

Question: Who needs good commit messages?

Answer: Everyone.

Question: Isn't it more efficient to save time by typing less? *E.g.*

```
$ git ci -a -m "Made misc changes"
```

Answer: No.

Good commit messages start with one beautifully composed line.

And then degenerate into all the embarrassing details about the changeset that only the truly dedicated or desperate readers want or need to know.

Remember that that desperate reader may well be you a month before you're hoping to earn your Nobel prize.

Examining old versions

What was the state of **hello.c** two revisions ago?

```
$ git show HEAD~~ hello.c
```

shows me the changeset, not the file. What you need is

```
$ git show HEAD~~:hello.c  
#include <stdio.h>
```

```
int i;                                /* an integer */  
const char *str = "Hello World";  
  
int  
main()  
{  
    printf("What I tell you %d times is true\n", 10);  
    for (i = 0; i < 10; ++i) {  
        printf("%d %s\n", i, str);  
    }  
}
```

Getting back old versions

I can get back an old version with:

```
$ git checkout 8187a17d hello.c
```

And return to my initial state with

```
$ git reset --hard
```

or

```
$ git checkout HEAD hello.c
```

If instead I say

```
$ git checkout 8187a17d
```

git replies:

```
Note: checking out '8187a17d'.
```

```
You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
```

i.e. "I don't know where to store any changes you might make". The resolution is easy (e.g. `git checkout -b foo`), but I'm not going to explain it now. We'll return to *branches* later.

You can get back to your initial state with:

```
$ git checkout master
```

```
Previous HEAD position was 8187a17... Made everything global
Switched to branch 'master'
```

Naming versions (git tag)

Referring to HEAD~3 isn't very practical, as after I commit a change I have to remember to say HEAD~4 instead.

So the safe thing to do is to refer to 4882a1f and write

```
v1.0 == 4882a1f (version I sent to Obama)
```

on a postit note stuck to my laptop; safe, but inconvenient.

Fortunately, git doesn't insist on SHAs:

```
$ git tag v1.0 4882a1f
$ git show v1.0
commit 4882a1fe4f5970fdb07998e77e1c7c68a5e6f047
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
Date:   Mon Sep 24 15:40:36 2012 -0400
```

```
    Added Makefile; ignored hello
```

```
diff --git a/.gitignore b/.gitignore
...
```

I can use v1.0 wherever I can use a SHA; e.g.

```
$ git log v1.0~..v1.0
commit 4882a1fe4f5970fdb07998e77e1c7c68a5e6f047
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
Date:   Mon Sep 24 15:40:36 2012 -0400
```

```
    Added Makefile; ignored hello
```

Naming versions (git tag)

I still need that sticky note for the message, but:

```
$ git tag -f -m "Version I sent to Obama" v1.0 4882a1f
Updated tag 'v1.0' (was 4882a1f)
$ git show v1.0
tag v1.0
Tagger: Robert Lupton the Good <rlh@astro.princeton.edu>
Date:   Thu Sep 27 15:14:24 2012 -0400
```

```
Version I sent to Obama
```

```
commit 4882a1fe4f5970fdb07998e77e1c7c68a5e6f047
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
Date:   Mon Sep 24 15:40:36 2012 -0400
```

```
    Added Makefile; ignored hello
```

```
diff --git a/.gitignore b/.gitignore
...

```

Actually, I could have created an *annotated tag* in the first place with

```
$ git tag -a -m "Version I sent to Obama" v1.0 4882a1f
```

and dispensed with the note. This also works better with `git describe`.

Naming versions (git tag)

If I worry about you masquerading as me, I can cryptographically sign the tag:

```
$ git tag -s -m "Version I should have sent to Putin" v1.1 ce18930
```

```
You need a passphrase to unlock the secret key for
user: "Robert Lupton the Good <rlh@astro.princeton.edu>"
2048-bit RSA key, ID 318B6ABA, created 2012-02-02 (main key ID C630EBCB)
```

(that message seems to mean, "Everything shipshape, sir"); use `git tag -v v1.1` to check the signature.

I can list all my tags with:

```
$ git tag -n
v1.0          Version I sent to Obama
v1.1          Version I should have sent to Putin
```

Once you've acquired lots of tags, it can be useful to look at a subset:

```
$ git tag -n --list 'v*0'
v1.0          Version I sent to Obama
```

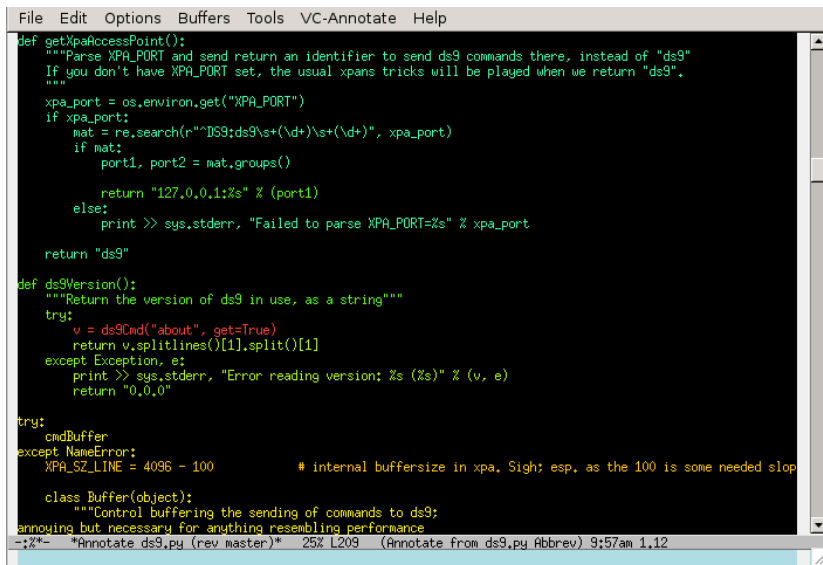
git blame

Sometimes you're reading code and ask yourself why, *why*, **why** did they do that?

```
$ git blame -b -e --date=short hello.c | cat
      (<rh!> 2012-09-24 1) #include <stdio.h>
      (<rh!> 2012-09-24 2)
8187a17d (<rh!> 2012-09-24 3) int i;                                /* an integer */
ce18930a (<rh!> 2012-09-24 4) const char *str = "Goodbye Universe";
8187a17d (<rh!> 2012-09-24 5)
      (<rh!> 2012-09-24 6) int
      (<rh!> 2012-09-24 7) main()
      (<rh!> 2012-09-24 8) {
9613186c (<rh!> 2012-09-24 9)     printf("What I tell you %d times is true\n", 10);
9613186c (<rh!> 2012-09-24 10)    for (i = 0; i < 10; ++i) {
      (<rh!> 2012-09-24 11)        printf("%d %s\n", i, str);
      (<rh!> 2012-09-24 12)    }
      (<rh!> 2012-09-24 13) }

$ git log --oneline 9613186c~..9613186c
9613186 Removed that confusing define
```

git blame + emacs (vc-annotate)



```
File Edit Options Buffers Tools VC-Annotate Help

def getXpaAccessPoint():
    """Parse XPA_PORT and send return an identifier to send ds9 commands there, instead of "ds9"
    If you don't have XPA_PORT set, the usual xpans tricks will be played when we return "ds9".
    """
    xpa_port = os.environ.get("XPA_PORT")
    if xpa_port:
        mat = re.search(r"^DS9:ds9\s+(\d+)\s+(\d+)", xpa_port)
        if mat:
            port1, port2 = mat.groups()
            return "127.0.0.1:%s" % (port1)
        else:
            print >> sys.stderr, "Failed to parse XPA_PORT=%s" % xpa_port
    return "ds9"

def ds9Version():
    """Return the version of ds9 in use, as a string"""
    try:
        v = ds9Cmd("about", get=True)
        return v.splitlines()[1].split()[1]
    except Exception, e:
        print >> sys.stderr, "Error reading version: %s (%s)" % (v, e)
        return "0.0.0"

try:
    cmdBuffer
except NameError:
    XPA_SZ_LINE = 4096 - 100          # internal buffersize in xpa. Sigh; esp. as the 100 is some needed slop

    class Buffer(object):
        """Control buffering the sending of commands to ds9;
        annoying but necessary for anything resembling performance

-:~*- *Annotate ds9.py (rev master)* 25% L209 (Annotate from ds9.py Abbrev) 9:57am 1,12
```

git blame + emacs (vc-annotate)

```

File Edit Options Buffers Tools Log-View Help

ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 196) """Parse XPA
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 197) If you don't
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 198) """
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 199) xpa_port = 0
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 200) if xpa_port:
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 201)     mat = re
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 202)     if mat:
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 203)         port
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 204)
a2be114f python/lsst/afw/display/ds9.py (rhl 2010-01-18 03:10:25 +0000 205)     retu
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 206)     else:
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 207)         prin
6ca80c53 python/lsst/afw/display/ds9.py (price 2010-11-04 21:37:08 +0000 208)
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 209)     return "ds9"
ae6e4fbe python/lsst/afw/display/ds9.py (rhl 2009-03-25 19:14:09 +0000 210)
7fd5d77 python/lsst/afw/display/ds9.py (rhl 2010-01-08 13:37:07 +0000 211) def ds9Version()
7fd5d77 python/lsst/afw/display/ds9.py (rhl 2010-01-08 13:37:07 +0000 212)     """Return th
6edafcf python/lsst/afw/display/ds9.py (rhl 2010-03-20 18:52:05 +0000 213)     try:
11050ed0 python/lsst/afw/display/ds9.py (Robert Lupton the Good 2012-08-01 22:09:08 +0900 214)         v = ds9C
77940d02 python/lsst/afw/display/ds9.py (rhl 2010-08-18 20:54:52 +0000 215)         return v
6edafcf python/lsst/afw/display/ds9.py (rhl 2010-03-20 18:52:05 +0000 216)     except Excep
77940d02 python/lsst/afw/display/ds9.py (rhl 2010-08-18 20:54:52 +0000 217)         print >>
6edafcf python/lsst/afw/display/ds9.py (rhl 2010-03-20 18:52:05 +0000 218)         return ">
7fd5d77 python/lsst/afw/display/ds9.py (rhl 2010-01-08 13:37:07 +0000 219)
9f42c862 python/lsst/afw/display/ds9.py (rhl 2011-02-22 09:55:11 +0000 220)     try:
9f42c862 python/lsst/afw/display/ds9.py (rhl 2011-02-22 09:55:11 +0000 221)         cmdBuffer
9f42c862 python/lsst/afw/display/ds9.py (rhl 2011-02-22 09:55:11 +0000 222)     except NameError

-:~*- *Annotate ds9.py (rev master)* 25% L214 (Annotate from ds9.py Abbrev) 9:58am 1.29
Commit 11050ed01ddb361401c048eccc2d4741947ea
Author: Robert Lupton the Good <rhl@astro.princeton.edu>
Date: Wed Aug 1 22:09:08 2012 +0900

    Added get argument to ds9Cmd; used it to implement getMaskTransparency

-:~*- *vc-change-log* All L1 (Git-Log-View from *Annotate ds9.py (rev master)* Abbrev) 9:58am 1.29
  
```

How do I know that my code will compile?

"Robert, when discussing the index, you told us to pick-and-choose the changes that actually get committed to the repository. Doesn't that mean that I can't be sure that my code will work when I go back to that version later?"

Good question. Fortunately, there's a good answer. Enter `git stash`:

```
$ git status
# On branch master
# 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:   hello.c
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git stash
Saved working directory and index state WIP on master: cel8930 Changed sign of greeti
HEAD is now at cel8930 Changed sign of greeting
$ git status
# On branch master
nothing to commit (working directory clean)
$ make
cc -o hello -Wall hello.c
$
```

git stash

OK, it compiled. Let's get that change back...

```
$ git stash pop
# On branch master
# 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:   hello.c
#
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (d767b49c389d80cca437d2b18dd25d41da70c0cb)
$ make
cc -o hello -Wall hello.c
hello.c:11:32: error: expected ';' after expression
    printf("%d %s\n", i, str)
                               ^
                               ;
1 error generated.
make: *** [hello] Error 1
```

In this case, I think we'll forget that that proto-version ever existed:

```
$ git reset --hard
HEAD is now at ce18930 Changed sign of greeting
```

git won't actually forget for around a month. For example:

```
$ git fsck --no-reflog | awk '/dangling commit/ {print $3}' | xargs git show
...
$ git stash apply d767b49c389d80cca437d2b18dd25d41da70c0cb
```

git branch

Inspired by PITP you start to wonder if all your software is quite as well written as you once believed. You'd like to experiment, starting with the initial version of the code; what should you do?

Rather than making a copy of one of your backups, you say:

```
$ git branch refactor 140c443
$ git checkout refactor
Switched to branch 'refactor'
$ ls -A
.git                hello.c
$ head -7 hello.c
#include <stdio.h>

#define NITER 10

int
main()
{
```

Miraculous! That's the initial version (as `git log` would have told you if you'd asked). An alternative way to create the branch would have been:

```
$ git checkout -b refactor 140c443
```

git branch v. git tag

How's that different from a tag?

- A tag is a label for the SHA of a particular changeset
- A branch starts out as a label for the SHA of a particular changeset, but when you commit a change the label moves.

Remember, both tags and branches are labels for changesets, not files. When you want to talk about files, you're talking about a set of changesets, starting at the tag/branch and stretching back in time.

git cherry-pick

My branch starts further back in pre-history than the Makefile. What should I do?

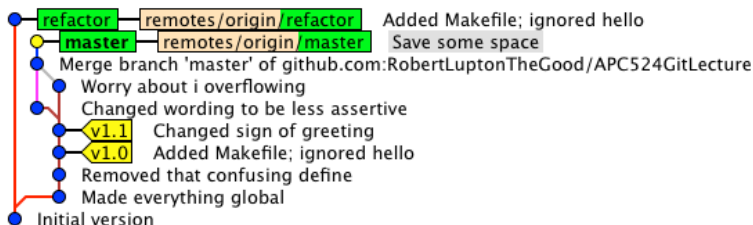
```
$ git log --grep=Makefile master
commit 4882a1fe4f5970fdb07998e77e1c7c68a5e6f047
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
Date:   Mon Sep 24 15:40:36 2012 -0400

    Added Makefile; ignored hello

$ git cherry-pick 4882a1f
[refactor bbbeld7] Added Makefile; ignored hello
 2 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Makefile
$ ls -A
.git                .gitignore         Makefile           hello.c
$ git status
# On branch refactor
nothing to commit (working directory clean)
```

The Philosophy of Git

There's a tool `gitk` that comes with `git` and I can ask it to show me my repository (and select `View -> All Branches` from the menu):



(there are other available tools, e.g. `sourcetree`).

The yellow labels are tags. Each blue or yellow circle is a changeset with associated SHA — my entire project is a DAG of changesets. I never ask for a version of a file; I ask `git` to apply the changesets that would generate that version.

Sharing Repositories

We learnt to clone a repository:

```
$ git clone git@github.com:RobertLuptonTheGood/APC524GitLecture
```

Question: What happens if someone else issues the same command on *their* laptop?

Answer: They get their own copy of my code.

That's OK. But what happens if

- they make changes and `git push` the results;
- I make different changes, and `git push` the results

Which version of the modified file should git accept?

Actually, that's the wrong question. git doesn't think in files, it thinks in changesets. A file in your directory is just the result of applying a set of changesets — so you meant to ask, Which changesets should be applied? Answer: all of them.

git push (Developer A)

Developer A finds a potential bug, and fixes it:

```
$ emacs hello.c
$ git diff
diff --git a/hello.c b/hello.c
index 182fc70..1ad9c72 100644
--- a/hello.c
+++ b/hello.c
-1,6 +1,6
#include <stdio.h>

-int i;                                /* an integer */
+unsigned int i;                       /* an unsigned integer */
const char *str = "Goodbye Universe";

int
$ git commit -m "Worried about i overflowing" hello.c
[master c61f64c] Worried about i overflowing
1 file changed, 1 insertion(+), 1 deletion(-)

$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 317 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:RobertLuptonTheGood/APC524GitLecture
ce18930..c61f64c master -> master
```

If you want to push your tags too, say `git push --tags`

git push (Developer B)

OK, that was easy. What has Developer B been up to?

```
$ git status
# On branch master
# 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:   hello.c
#
no changes added to commit (use "git add" and/or "git commit -a")
$ git diff
diff --git a/hello.c b/hello.c
index 182fc70..b98fela 100644
--- a/hello.c
+++ b/hello.c
@@ -6,7 +6,7 @@ const char *str = "Goodbye Universe";
 int
main()
{
-   printf("What I tell you %d times is true\n", 10);
+   printf("What I mention %d times is true\n", 10);
   for (i = 0; i < 10; ++i) {
       printf("%d %s\n", i, str);
   }
}
$ git commit -m "Changed wording to be less assertive" hello.c
[master 858d33e] Changed wording to be less assertive
1 file changed, 1 insertion(+), 1 deletion(-)
```

git push (Developer B)

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
$ git log --oneline
858d33e Changed wording to be less assertive
ce18930 Changed sign of greeting
4882a1f Added Makefile; ignored hello
9613186 Removed that confusing define
8187a17 Made everything global
140c443 Initial version

$ git push
To git@github.com:RobertLuptonTheGood/APC524GitLecture.git
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:RobertLuptonTheGood/APC524GitLecture.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

Hmmm. Note that the complaints aren't about *files*, they're about *refs* and *updates* — that is, changesets.

git pull (Developer B)

We'll do what they say:

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2)
Unpacking objects: 100% (3/3), done.
From github.com:RobertLuptonTheGood/APC524GitLecture
   ce18930..c61f64c  master       -> origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
  hello.c |      2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
nothing to commit (working directory clean)
```

No complaints, so let's try again:

```
$ git push
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 648 bytes, done.
Total 6 (delta 4), reused 0 (delta 0)
To gitgithub.com:RobertLuptonTheGood/APC524GitLecture.git:c61f64c..da65de9  master
-> master
```

Hurrah!

Developer A and Developer B

What does Developer B have?

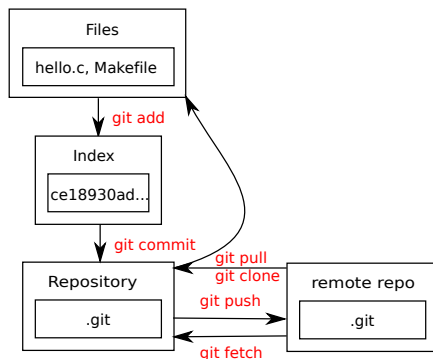
```
$ git log --oneline
da65de9 Merged branch 'master' of github.com:RobertLuptonTheGood/APC524GitLecture
c61f64c Worried about i overflowing
858d33e Changed wording to be less assertive
ce18930 Changed sign of greeting
4882a1f Added Makefile; ignored hello
9613186 Removed that confusing define
8187a17 Made everything global
140c443 Initial version
```

Switching back to Developer A:

```
$ git status
# On branch master
nothing to commit (working directory clean)
$ git pull
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 4), reused 6 (delta 4)
Unpacking objects: 100% (6/6), done.
From github.com:RobertLuptonTheGood/APC524GitLecture
   c61f64c..da65de9  master    -> origin/master
Updating c61f64c..da65de9
Fast-forward
 hello.c |    2 +
 1 file changed, 1 insertion(+), 1 deletion(-)
```

The output from `git log` is now identical to Developer B's.

git push and git fetch



i.e. git fetch just updates my repository, while git pull first fetches and then updates my working files.

git remote

There are commands to tell you about your repositories

```
$ git remote show origin
* remote origin
Fetch URL: git@github.com:PrincetonUniversity/APC524
Push URL: git@github.com:PrincetonUniversity/APC524
HEAD branch: master
Remote branches:
  master          tracked
  no-class-registry tracked
  registry        tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (fast-forwardable)
```

I have an git alias to make this sort of thing easier:

```
[alias]
  rem = !"git remote -v ; echo '    current branch:' \
    $(git for-each-ref --format='%(upstream:short)' $(git symbolic-ref -q HEAD))"
  rso = remote show origin

$ git rem
origin git@github.com:PrincetonUniversity/APC524 (fetch)
origin git@github.com:PrincetonUniversity/APC524 (push)
  current branch: origin/master
```

git fetch

Why is `git fetch` interesting?

```
$ git fetch
$ git diff origin/master
diff --git a/hello.c b/hello.c
index 8be616d..228d88e 100644
--- a/hello.c
+++ b/hello.c
@@ -1,6 +1,6 @@
    #include <stdio.h>

    unsigned short int i;                /* counter */
+   unsigned int i;                      /* an unsigned integer */
    const char *str = "Goodbye Universe";

    int
```

In other words, "what's the difference between the master branch at origin and HEAD?" /i.e. what would be pushed if I typed `git push`. The origin is my remote repository:

```
$ git remote -v
origin  git@github.com:RobertLuptonTheGood/APC524GitLecture (fetch)
origin  git@github.com:RobertLuptonTheGood/APC524GitLecture (push)
```

If I just wanted the log messages, I'd say

```
$ git log origin/master..HEAD
```

git merge

I can merge those changes into my working copy with:

```
$ git merge
fatal: No commit specified and merge.defaultToUpstream not set.
```

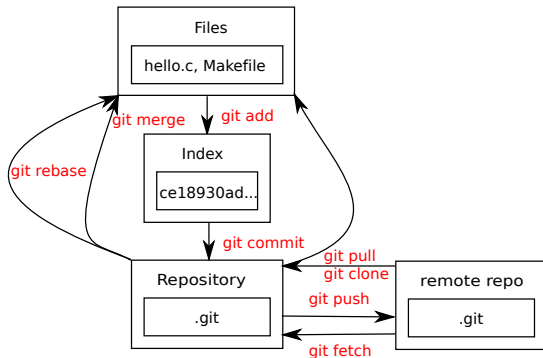
Oh no I can't. I should have typed

```
git merge origin/master
```

or I can set that configuration option:

```
$ git config merge.defaultToUpstream true
$ git merge
Updating da65de9..e07f6c6
Fast-forward
 hello.c |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

git merge and git rebase



I'll come back to rebase later.

git push for branches

I can make my branch visible to my colleagues with

```
$ git push -u origin refactor
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 392 bytes, done.
Total 4 (delta 0), reused 2 (delta 0)
To git@github.com:RobertLuptonTheGood/APC524GitLecture
 * [new branch]      refactor -> refactor
Branch refactor set up to track remote branch refactor from origin.
```

Now, your developers can say:

```
git checkout refactor
```

N.b. older versions of git required them to say:

```
git checkout -t origin/refactor
```

but you should be OK with the simpler version.

The great `git push` gotcha

Until moderately recently (git 1.7.10?), `git pull` and `git push` were asymmetric:

- `git pull` pulled only the current branch
- `git push` pushed *all* branches.

This is very confusing when pushing a branch you're not working on fails — the symptoms are that you can't push your work as the other branch (which you've forgotten all about) needs to be pulled before *it* can be pushed.

You can change this behaviour with:

```
$ git config --global push.default = upstream
```

i.e. only push my current branch, mirroring `pull`'s behavior.

For git versions before 1.7.6, you needed to set `push.default = tracking` instead. `upstream` became the default behaviour of `git push` sometime in 2012.

When `git merge` fails

As soon as your fellow developers start making changes, you just know that they'll make overlapping changes. In the dark ages, we dealt with this by forbidding more than one person to work simultaneously on the same file. But not only was waiting for permission to work unproductive, it didn't solve the problem — in general, you'd need to lock all files that `#include` the file you're working on. So we declared open-season on editing; the corollary is that there *will* be conflicts.

Conflicting changes

Developer A has no problems:

```
$ emacs hello.c
$ git diff
diff --git a/hello.c b/hello.c
index 8be616d..d336329 100644
--- a/hello.c
+++ b/hello.c
@@ -10,4 +10,6 @@ main()
     for (i = 0; i < 10; ++i) {
         printf("%d %s\n", i, str);
     }
+
+    return 0;
+}
$ git commit -m "Returned success code to the shell" hello.c
[master 19d6562] Returned success code to the shell
 1 file changed, 2 insertions(+)
$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 320 bytes, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:RobertLuptonTheGood/APC524GitLecture.git
e07f6c6..19d6562 master -> master
```

Conflicting changes

Developer B's happy too:

```
$ emacs hello.c
$ git diff
diff --git a/hello.c b/hello.c
index 8be616d..caff08b 100644
--- a/hello.c
+++ b/hello.c
@@ -10,4 +10,6 @@ main()
     for (i = 0; i < 10; ++i) {
         printf("%d %s\n", i, str);
     }
+
+   exit(0);
+ }
$ git commit -m "Returned success code to the shell" hello.c
[master d52182f] Returned success code to the shell
1 file changed, 2 insertions(+)
```

At first.

```
$ git push
To git@github.com:RobertLuptonTheGood/APC524GitLecture
! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@github.com:RobertLuptonTheGood/APC524GitLecture'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

Conflicting changes

If developer B is cautious, she can figure out what's going on:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2)
Unpacking objects: 100% (3/3), done.
From github.com:RobertLuptonTheGood/APC524GitLecture
   e07f6c6..19d6562  master    -> origin/master
$ git diff origin/master..HEAD
diff --git a/hello.c b/hello.c
index d336329..caff08b 100644
--- a/hello.c
+++ b/hello.c
@@ -11,5 +11,5 @@ main()
     printf("%d %s\n", i, str);
 }

-    return 0;
+    exit(0);
 }
```

The two of them have made the same change too different ways.
No merge tool can sort that out automatically.

git merge

What *does* git do?

```
$ git pull
Auto-merging hello.c
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit the result.

$ cat hello.c
#include <stdio.h>

unsigned short int i;                /* counter */
const char *str = "Goodbye Universe";

int
main()
{
    printf("What I mention %d times is probably true\n", 10);
    for (i = 0; i < 10; ++i) {
        printf("%d %s\n", i, str);
    }

    <<<<< HEAD
    exit(0);
    =====
    return 0;
    >>>>> 19d656221992b96eb3a05927572765908a963e74
}
```

That's helpful; git marked the conflict and left it up to the humans.

Resolving conflicts

Developer B knows how to resolve that:

```
$ emacs hello.c
$ git add hello.c
$ git commit -m "Kept my version of the change"
[master 70f0225] Kept my version of the change
```

And now:

```
$ git push
Counting objects: 8, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 458 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
To git@github.com:RobertLuptonTheGood/APC524GitLecture
19d6562..70f0225 master -> master
```

Resolving conflicts

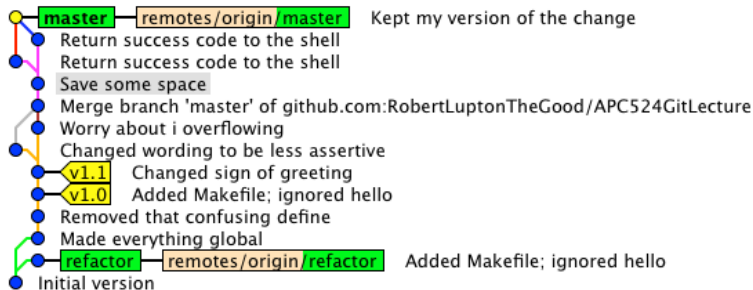
What about Developer A; does he have to resolve the conflict too?

```
$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 3), reused 4 (delta 3)
Unpacking objects: 100% (4/4), done.
From github.com:RobertLuptonTheGood/APC524GitLecture
   19d6562..70f0225  master       -> origin/master
Updating 19d6562..70f0225
Fast-forward
  hello.c |    2 +-
  1 file changed, 1 insertion(+), 1 deletion(-)
$ git status
# On branch master
nothing to commit (working directory clean)
```

And there was much rejoicing.

State of repository

After all these operations, what does git's DAG look like?



Note that the non-overlapping changesets *Changed wording to be less assertive* and *Worried about i overflowing* have resulted in a diverge-and-merge which doesn't really tell us anything about the history of the project.

git merge v. git rebase

Developers A and B are both working on the refactor branch.

Developer A:

```
$ git checkout refactor
$ git add hello.c
$ git commit -m "Set the number of iterations on the command line"
$ git push
```

Developer B makes a different change:

```
$ git diff
diff --git a/hello.c b/hello.c
index ea0dbc7..0d1c76d 100644
--- a/hello.c
+++ b/hello.c
-10,4 +10,6  main()
    for (int i = 0; i < NITER; ++i) {
        printf("%d %s\n", i, str);
    }
+
+   return 0;
+ }
$ git commit -m "Set proper return code" hello.c
```


git merge v. git rebase

B would like to avoid that extra merge, so:

```
$ git pull --rebase
```

```
From github.com:RobertLuptonTheGood/APC524GitLecture
```

```
bbbeld7..696a322 refactor -> origin/refactor
```

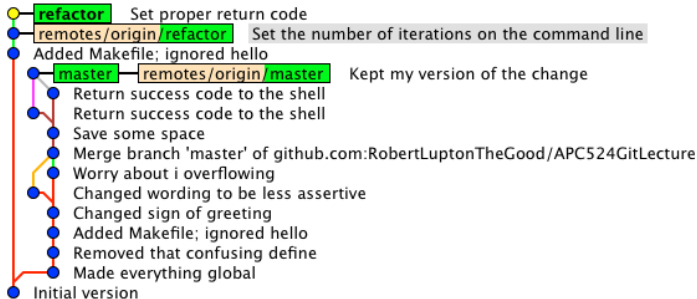
```
First, rewinding head to replay your work on top of it...
```

```
Applying: Set proper return code
```

```
Using index info to reconstruct a base tree...
```

```
Falling back to patching base and 3-way merge...
```

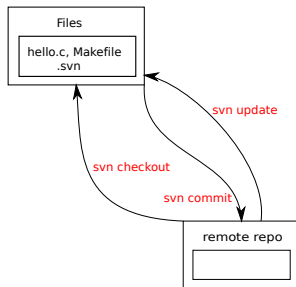
```
Auto-merging hello.c
```



Description	Graph
origin/refactor refactor Set proper return code	
Set the number of iterations on the command line	
origin/master origin/HEAD master Kept my version of the change	
Return success code to the shell	

git merge v. svn update

Some of you may be yearning for svn's simple view of the world:



Maybe you'll feel better if I remind you that that innocuous

```
$ svn update
```

corresponds to

```
$ git stash; git pull --rebase; git stash pop
```

There's a reason why seasoned svn hands often tar up their repositories before major updates, or (horrors) merges.

Other `git` commands

There are many other `git` commands, and options to the commands you've seen, which you'll see when you start googling for help. You'll see `git bisect` when we talk about debugging.

Binary File Support

`git` doesn't handle large binary files well. In fact, a common error is to check in a binary or shared object file by accident; you remove it immediately, but your repository remains bloated. The problem is that `git` remembers that you once added that file, so it needs to keep a copy in case you want it back.

The official solution is to use `git filter-branch` (you'll probably need to consult [stackoverflow](#)). The recommended solution is to use Roberto Tyley's `bfg` instead.

git-lfs

If you really did mean to add those files (maybe they're images used in unittests), a good option is `git lfs`. The trick is that the contents of the file are replaced with some sort of text pointer to an external 'cloud' location (e.g. `github`).

```
$ brew install git-lfs
#
$ git lfs install
#
$ git lfs track "*.fits"
$ git commit -m "Use lfs for FITS files" .gitattributes
# "There is no step three. Just commit and push to GitHub as you normally would"
$ git add M31.fits
$ git commit -m "Add image of comet"
$ git push origin master
```

The downside is:

Every user and organization on GitHub.com with Git LFS enabled will begin with 1 GB of free file storage and a monthly bandwidth quota of 1 GB. You can buy more data packs (50GB + 50GB/month) for 5\$/month.

Other sites (e.g. `bitbucket`, `gitlab`) are gearing up to support `git lfs`; e.g. it's free for up to 1Gb on `bitbucket` during their beta

Tracking down where a bug was introduced

We all know how frustrating it is to sit at a desk thinking, "I know this used to work. . . ". I haven't had a chance to teach you about unit testing, so I don't know when it slipped in. How do I go about finding a bug in a large code?

It turns out that I broke the python lecture when I fixed some problems after I gave it. How should I find what went wrong?

```
$ git log --oneline HEAD~5..HEAD
96b7e75 Renamed L03-C to L-C
801b2c1 Added xkcd about pointers
c11d1fa Fixes post-lecture
98688b7 Started to update debuggingII
029216b It's more than a week now
```

In this case, it was pretty clear which commit caused the problem, but let's be formal and use `git bisect`:

```
$ git bisect start
$ git bisect bad
$ git bisect good 029216b
Bisecting: 1 revision left to test after this (roughly 1 step)
[c11d1fa8affe900779a800d0c3a7065c17436204] Fixes post-lecture
```

git bisect continued

```
$ make
```

```
...
```

```
! Package Listings Error: File 'src/example_matplotlib.py(.makeArtist.snip)' not found
```

```
...
```

```
make: *** [L-python.pdf] Error 1
```

```
$ git bisect bad
```

```
Bisecting: 0 revisions left to test after this (roughly 0 steps)
```

```
[98688b7cd519144c3ac3035c6ed20c89ed7a5d68] Started to update debuggingII
```

```
$ make
```

```
$ git bisect good
```

```
c11d1fa8affe900779a800d0c3a7065c17436204 is the first bad commit
```

```
commit c11d1fa8affe900779a800d0c3a7065c17436204
```

```
Author: Robert Lupton the Good <rlh@astro.princeton.edu>
```

```
Date: Thu Oct 11 15:36:16 2012 -0400
```

```
Fixes post-lecture
```

```
:100644 100644 583c08ff68f62e91daa3100f103c2f9060268419 176cbbf4d8ccad5b7bc2afe8ea678
```

```
.gitignore
```

```
:100644 100644 2d721db7f9612ac1529fd2819e0ce72cea8ad891 780c8032df7f5542287cbb53a5c49
```

```
L-python.org
```

```
:040000 040000 728ff8e40915f78a4192cd89fbb939198eeefa5a 11aeebd7880eb20c489c736882b2f
```

```
src
```

git leaves us at this commit (as `git status` will tell you). `git bisect reset` will get you back to where you started.

git bisect with a script

Back to the `size` bug.

If you have a command that reveals the problem you can make `git` work harder.

If we have a script **myTest**:

```
#!/bin/sh
make bug1 || exit 125 # skip this revision
valgrind bug1 2>&1 |
perl -ne 'if(/definitely lost:\s*(\d+)/ && $1 > 0) {
    warn "Leaked $1 bytes\n"; exit 1;
}'
exit 0
```

we can say:

```
$ git bisect start
$ git bisect bad
$ git bisect good 9a45fc0
$ git bisect run myTest
```

which will print the first bad commit's SHA. As before, use `git bisect reset` to return to your initial state.

rcs or cvs or svn or hg or git or bzt?

There are lots of source code managers to choose from. On unix the original options were `sccs` and `rsc`, but both were essentially superseded 20 years ago by `cvs`. `svn` was meant to be a better `cvs`, with an 1.0 release in 2004. Starting around the same time three other options appeared, `bzt`, `git`, and `hg`.

Which should you use?

- Use `cvs` if you have legacy code in a `cvs` repository. But you might want to migrate to `svn`
- Use `svn` if you have legacy code in an `svn` repository, or you know `cvs` and want to learn as little as possible.
- Otherwise use `git` (or `hg` (Mercurial) if you insist). I think that `bzt` is slowly dying.

rcs or cvs or svn or hg or git or bazaar?

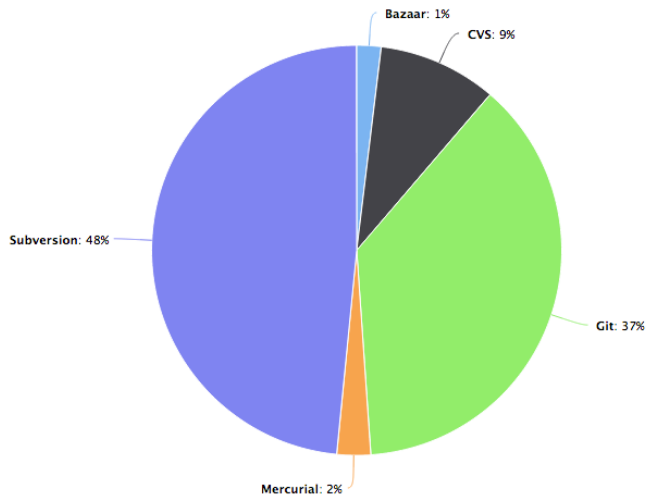
The oracle reported:

Wiki:git

The Eclipse Foundation reported in its annual community survey that as of May 2014, Git is now the most widely used source code management tool, with 42.9% of professional software developers reporting that they use Git as their primary source control system compared with 36.3% in 2013, 32% in 2012; or for Git responses excluding use of GitHub: 33.3% in 2014, 30.3% in 2013, 27.6% in 2012 and 12.8% in 2011. Open source directory Black Duck Open Hub reports a similar uptake among open source projects

Black Duck (née Ohloh) says that the git fraction's 39% in 2016

rcs or cvs or svn or hg or git or bazaar?



2014



Should you use svn?

Email on the gdb mailing list

From: Jim Blandy <jimb@red-bean.com>

Date: December 20, 2010 10:46:18 pm EST

Subject: Re: time to be serious about dropping CVS

As one of the original designers of SVN, I really recommend switching to either git or Mercurial. It takes some getting used to, but any GDB hacker can handle that challenge. Once you switch, you will love the speed so much you'll cry when you have to use CVS (or SVN).

They moved to git.

Sharing using github

I already have a github account (RobertLuptonTheGood) so I connected and created a new repository called APC524GitLecture. Then all I needed to do to share the test repository that this lecture's built around was:

```
$ git remote add origin git@github.com:RobertLuptonTheGood/APC524GitLecture.git
$ git push -u origin master
Counting objects: 16, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (16/16), 1.53 KiB, done.
Total 16 (delta 5), reused 0 (delta 0)
To git@github.com:RobertLuptonTheGood/APC524GitLecture.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

And that's all that there is to it; there's a copy of my repository at

<https://github.com/RobertLuptonTheGood/APC524GitLecture>

Furthermore, anyone can say:

```
$ git clone git@github.com:RobertLuptonTheGood/APC524GitLecture.git
```

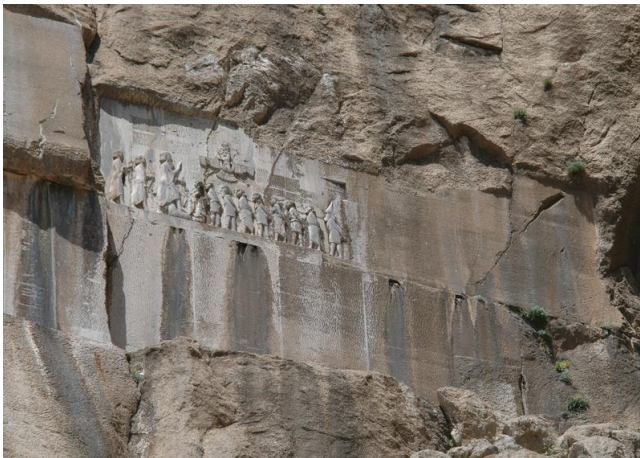
and get their own *clone* on their own machine.

All these clones are equivalent — anyone can look at the history, commit changes, add files, or anything else we've learned to do.

Private github repos

You may not like the idea that just *anyone* can read your carefully written code on github. Fear not; it's also possible to have private repositories if you pay. Alternatively, `bitbucket` offers free hosting for five or fewer participants and `github` for education (see e.g. `github private repos` article). It's also perfectly possible to run your own git server — if you're interested google `gitolite` and maybe `gitlab`.

Behistun Inscriptions



The Behistun Inscription, written in cuneiform in three languages: Old Persian, Elamite, and Babylonian.

git for English and American speakers

git init	Start a new project
git clone url	Get a copy of someone's repository
git add file	Prepare to save changes to file
git commit [-a] -m ...	Save those [all] changes (locally)
git push	Tell a remote site about my changes
git pull [--rebase]	Synchronize my local copy with a remote
git fetch	Synchronize my local repo with a remote
git rebase -i	Rewrite history
git status	What's up? (locally)
git log [--oneline]	What have I been doing?
git blame [FILE]	Who did what to FILE?
git checkout -b ABC	Make a branch called ABC
git push -u REMOTE ABC	Push branch ABC to a REMOTE
git tag -a XYZ	Make a tag named XYZ
git checkout FILE [REV]	Reset a FILE to REV
git rm FILE	Remove a FILE
git mv FILE NEW	Rename FILE to NEW
git show rev:FILE	Show an old version of FILE

svn for git users

git clone url
git pull --rebase
git status
git -f checkout path
git add file
git rm file
git mv file
git commit -a
git log
git show rev:fileName
git show rev:dirName
git show rev
git branch B
git merge --no-commit B
git tag -a T

svn checkout url
svn update
svn status
svn revert path
svn add file
svn rm file
svn mv file
svn commit
svn log
svn cat url
svn list url
svn log -rrev url
svn copy ^/svn/trunk ^/svn/branches/B
svn merge -r 20:HEAD ^/svn/branches/B
svn copy ^/svn/trunk ^/svn/tags/T