# Introduction to Version Control

- Source-code control: motivation
- Basic Concepts: Repository
- Basic Concepts: Working Copy
- Intro to git
- Some git workflow
- The "commit" concept
- Pushing, pulling, fetching and merging
- Conflict resolution

Basic Concepts: Repository

- A repository remembers every **committed** change to every **controlled** file
  - Even remembers additions and deletions to directory trees
- Clients reading from the repository normally sees latest version of file structure
  - Clients, however, can also choose to view previous states of the file structure
- Examples of previous states:
  - "What did this directory contain last Wednesday?"
  - "Who was the last person to change this file?"
  - "What changes did Pat make to this file?"
  - "Give me the version for release 3 of the code."

What we <u>do not</u> expect/want…
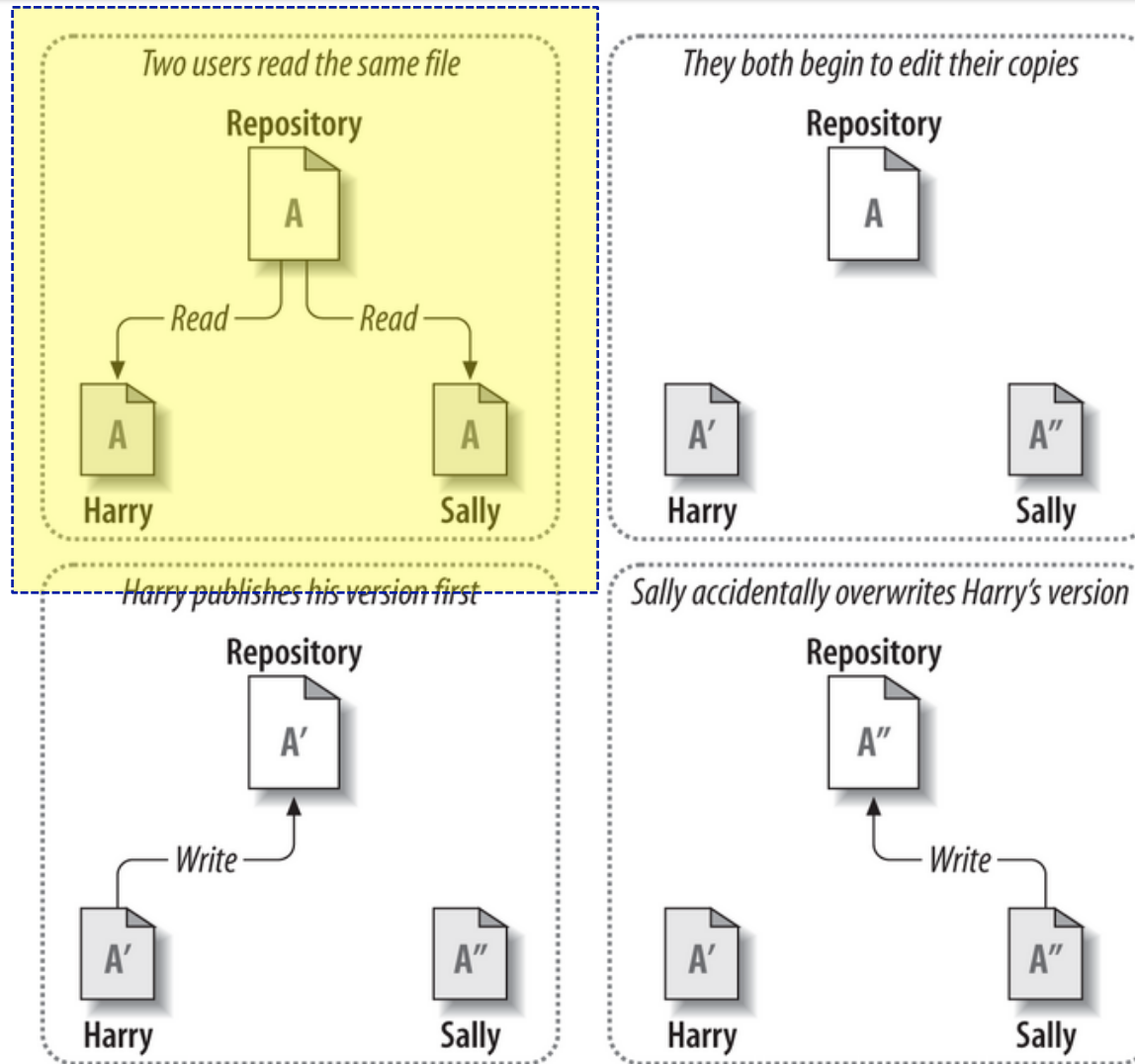
One approach around problem…
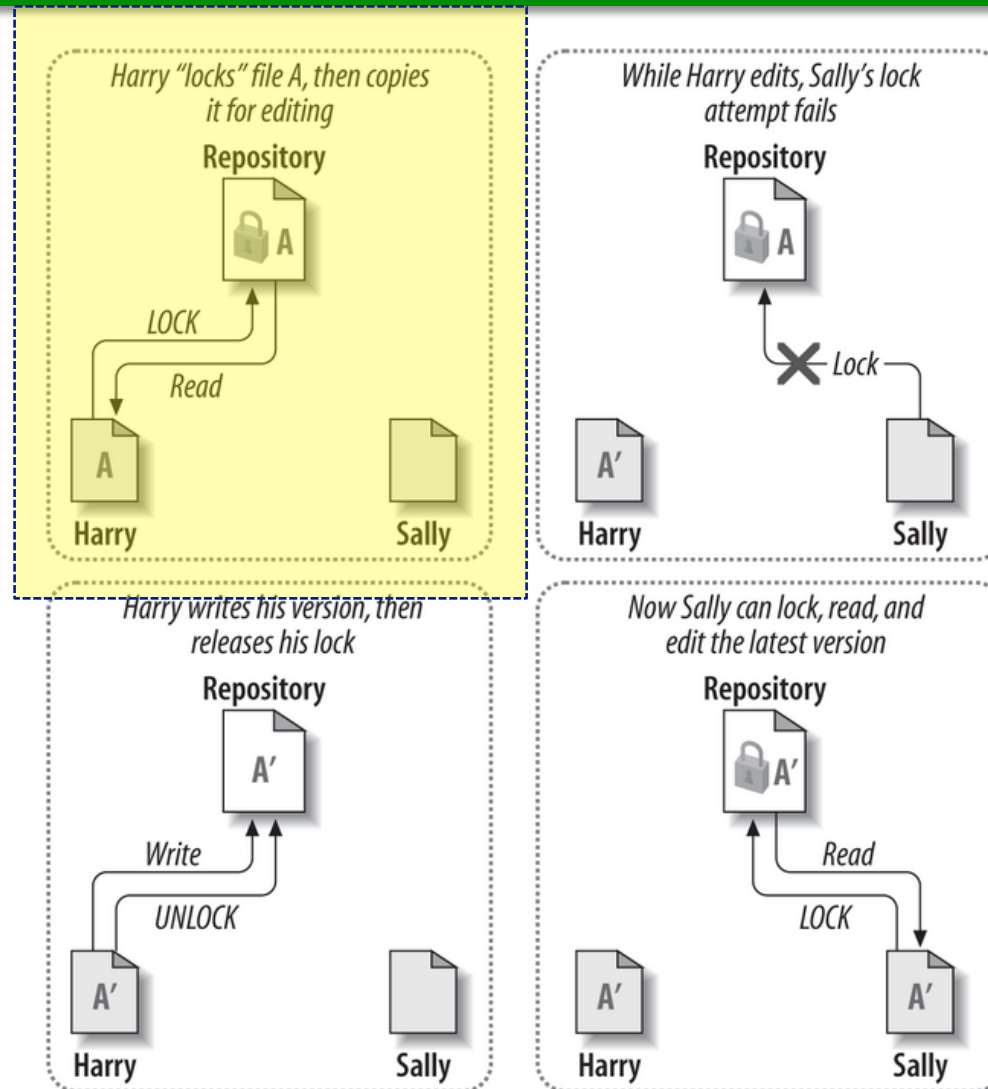
Some downsides to "locking"

- Can cause some administration problems
  - User with lock on holidays? sick?
  - Is it appropriate for all changes?
- Awkward in distributed environments
  - Absence of locking is attractive for open-source projects
  - Such projects have developers located around the globe
  - Locked files would be very inconvenient (if not disastrous)
- Locking gives a false sense of security.
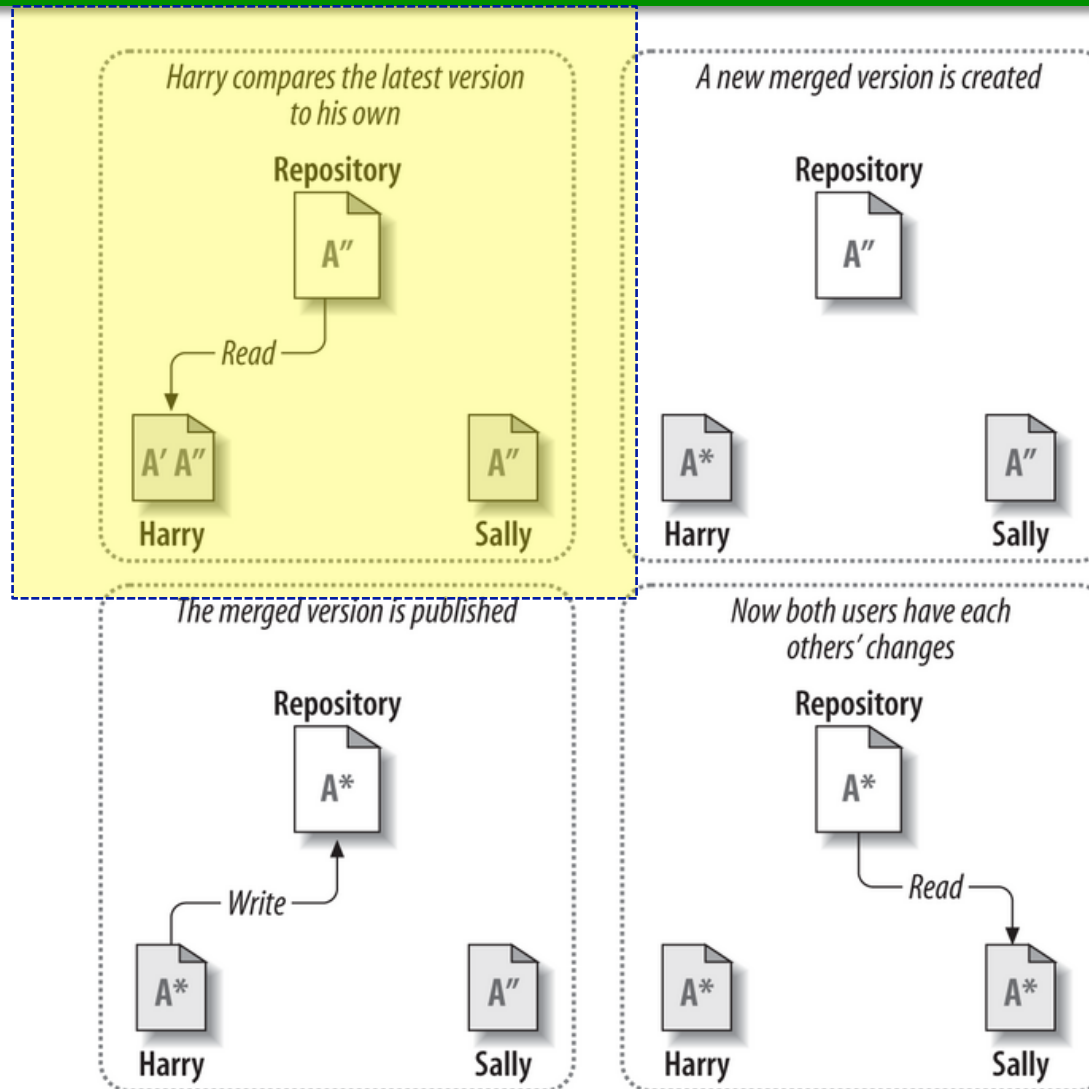- But how do we manage without locking?

"Copy-modify-merge" solution

**scenario:
conflict is
introduced**
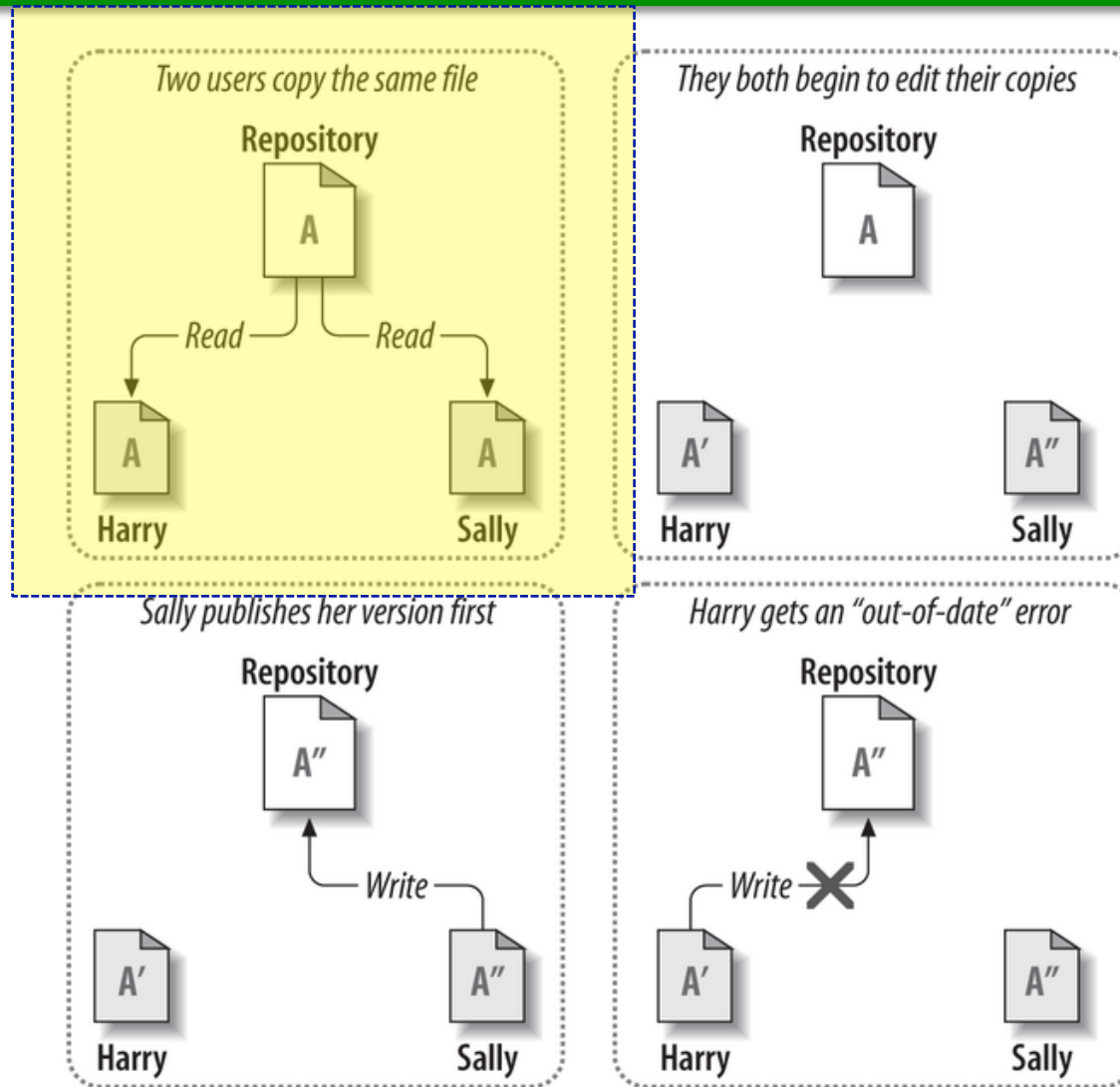
"Copy-modify-merge" solution

scenario: conflict is resolved

Observations

- **Copy-modify-merge** allows users to work in parallel
  - Most concurrent changes do not overlap
  - Consistency amongst files is explicitly managed (i.e., no false sense of security)
- This model assumes files are line-based text files
  - Assumes changes can (usually) be merged

More observations

- For binary files, very difficult (if not impossible) to merge conflicting changes
  - JPEG files
  - Object files and executable images
- However, we may still want to keep such items in the repository
  - Also: there exist "text" versions of some image formats (PNG, SVG)
- git does not support the notion of "release numbering"
  - As there may not be a global shared repo, no global number is possible
  - git instead associates a hash with each commit…
  - … which is actually a SHA-1 checksum of the git object created by the commit)

What is git?

- **A framework for version-control system workflows**
- Tracks changes to files and directories over time
  - These files/directories are usually associated with some software development project
- Resembles some features of a file system yet:
  - remembers changes to all files and directories managed by the repository
  - sometimes behaves like a time machine for files and directories
- Permits concurrent access to a repository over a network
  - This facilitates work on shared projects, herefore also enhances collaboration
  - Some technologists use services such as GitHub as a one-stop data repository

Some git history

- Proprietary distributed VCS (BitKeeper) had been used by the Linux kernel dev team
- When this was no longer available (2005), Linus Torvalds started work on a replacement
- Idea:
  - git is a "lower-level" VCS
  - front-ends can be created to provide different VCS workflows
- GitHub (2008) made git widely known by combining repository hosting with a simple git workflow
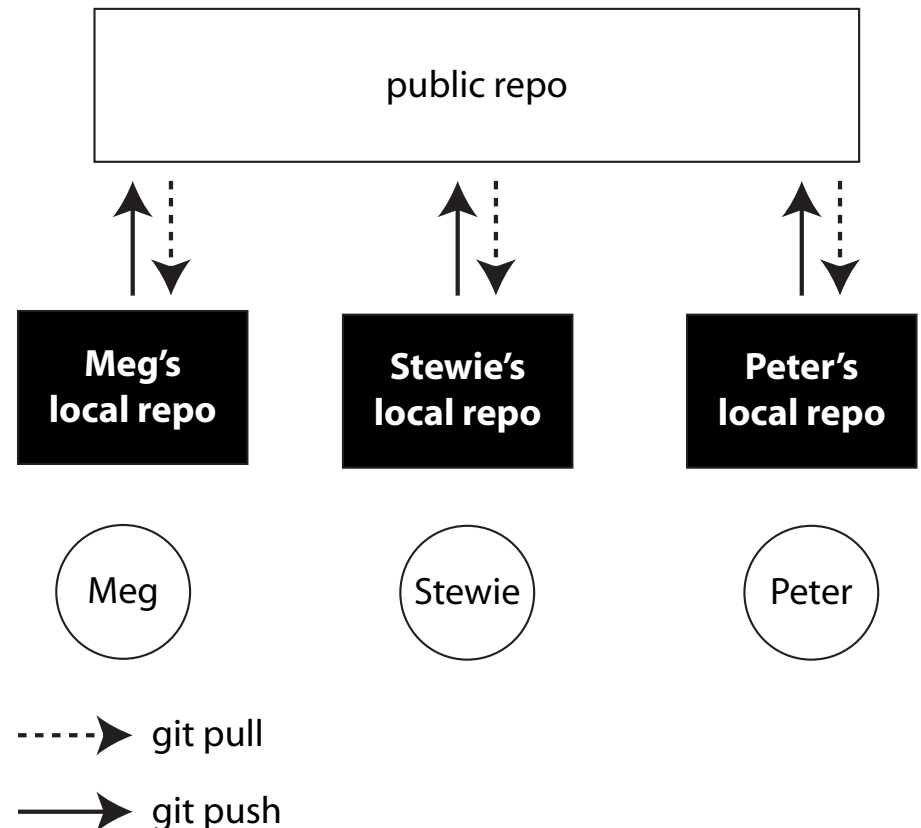
Basic Concepts: Repository

- A VCS **repository** is a store of data
- Repositories may be stored remote **repository servers**
- Data could be visualized as stored as something **resembling** a filesystem tree
- Any number of **clients** can connect to the repository
    - These clients can then read and write files in the repository
- By writing (e.g., committing) files, client makes files available to other clients
- By reading (e.g., updating) files, client is receiving information from other clients

Repository

Write   Read   Read

Client   Client   Client
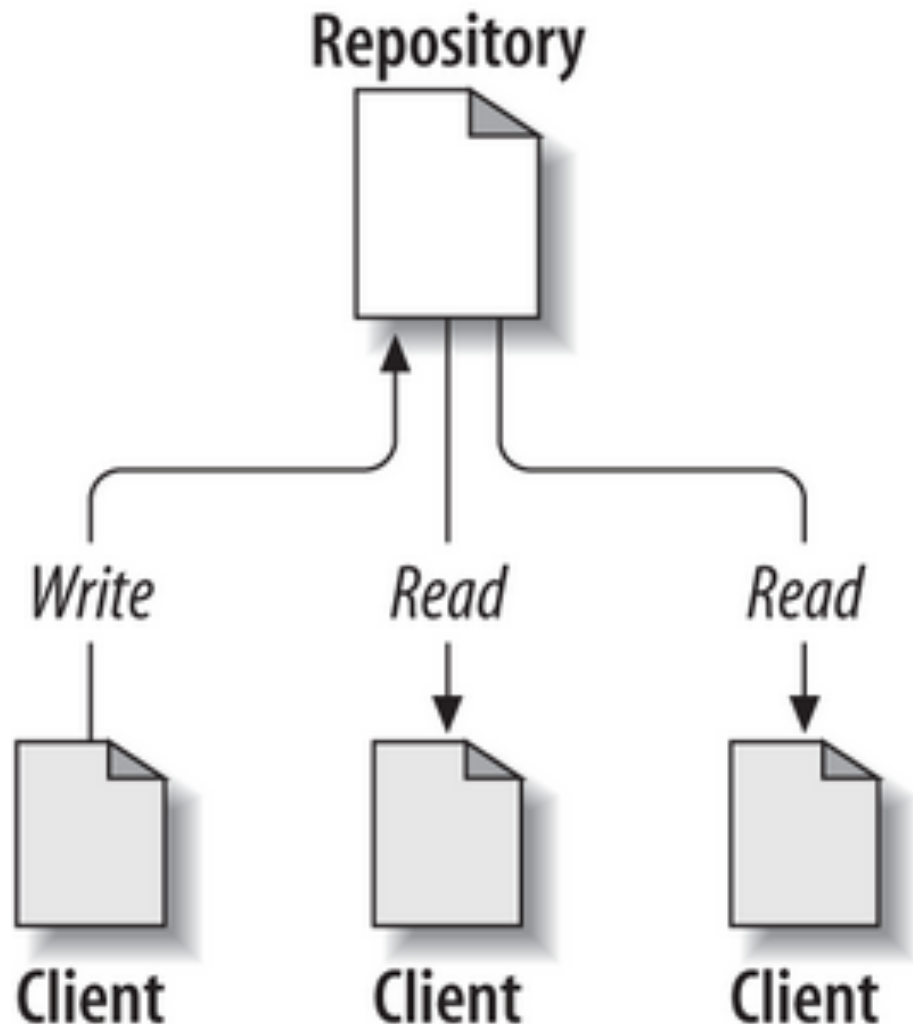
The git twist

- Git combines together the local repo with a working copy
- Git is often used assuming a shared repository….
- In "git-lish":
  - Writing to remote repo == **push**
  - Reading from remote repo == **pull**


```
public repo
```

Meg's local repo   Stewie's local repo   Peter's local repo

Meg   Stewie   Peter

- - - -▶ git pull

——▶ git push

The git twist

- … and git be used in a more distributed manner
- This means:
  - Developers have read/write (i.e., full push/pull) access on their repo
  - They can have read (i.e., pull) access on repos from other team members
- One level of links shown here…
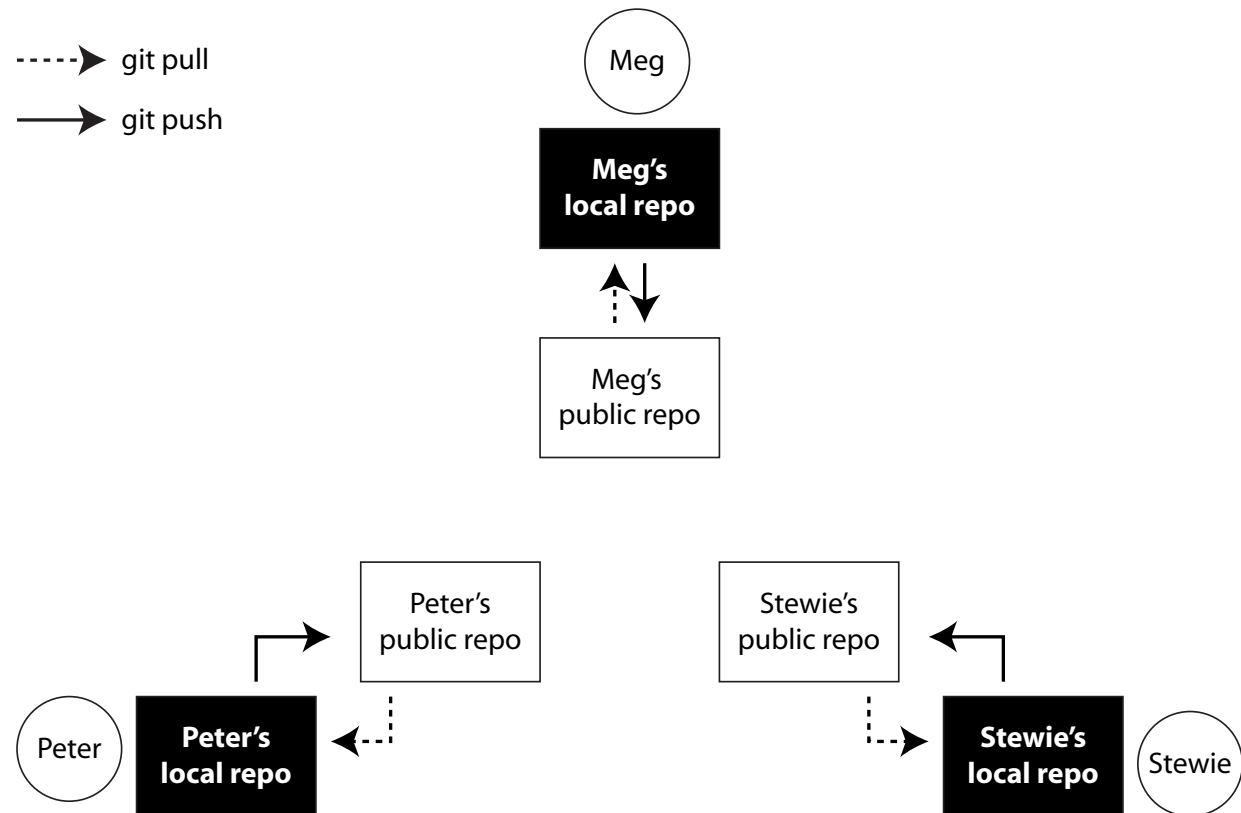
The git twist

- … and an additional level of links is shown here
- With distributed repos, there is no longer necessarily a single repo with all of the code
  - In practice, one team member becomes the "repository of record" (maybe it should be Stewie?
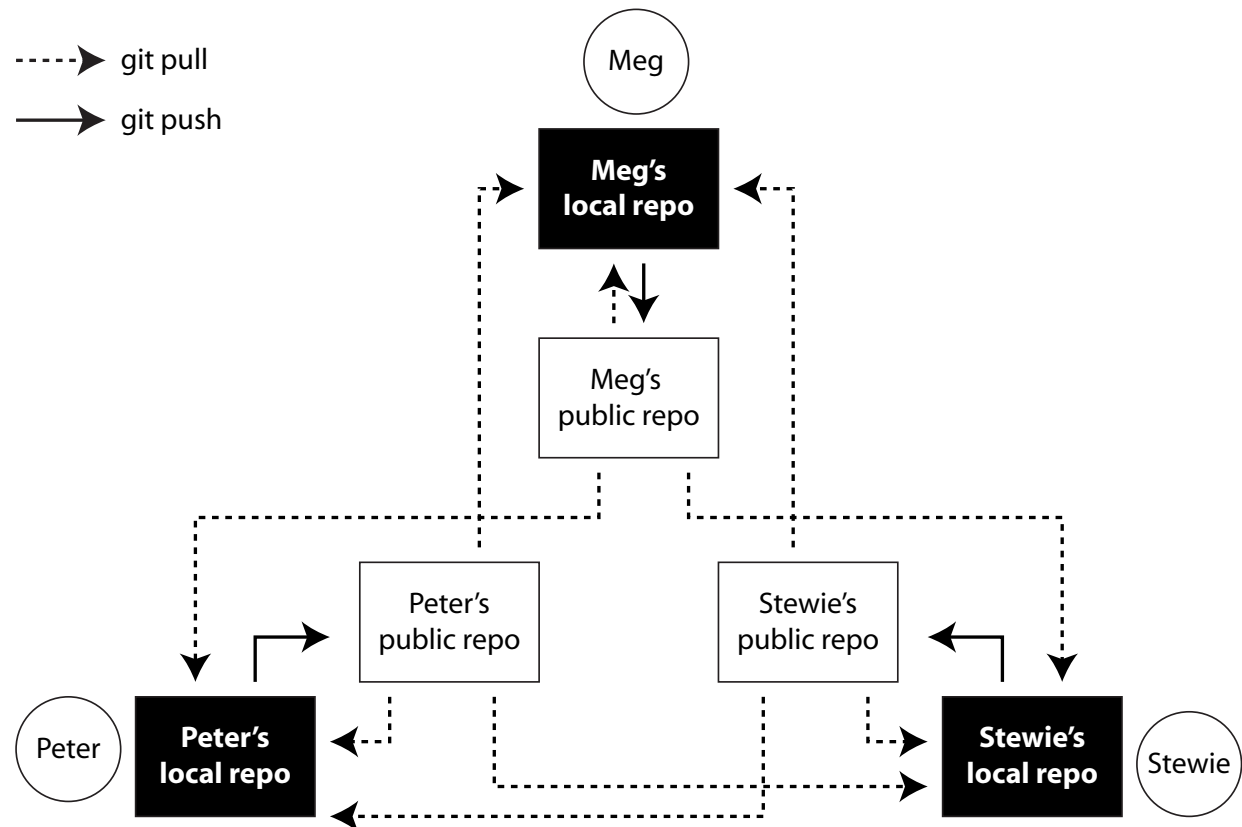
git for UVic's seng265

- Each student has their own repository on git.seng.uvic.ca/seng265
- Each student can both push and pull (read and write) on their **remote**
- Students cannot push or pull other student's repos
- Teaching team able to pull from (but not push to!) student repos
- Note that students could have multiple local copies of the same repo. Achtung!
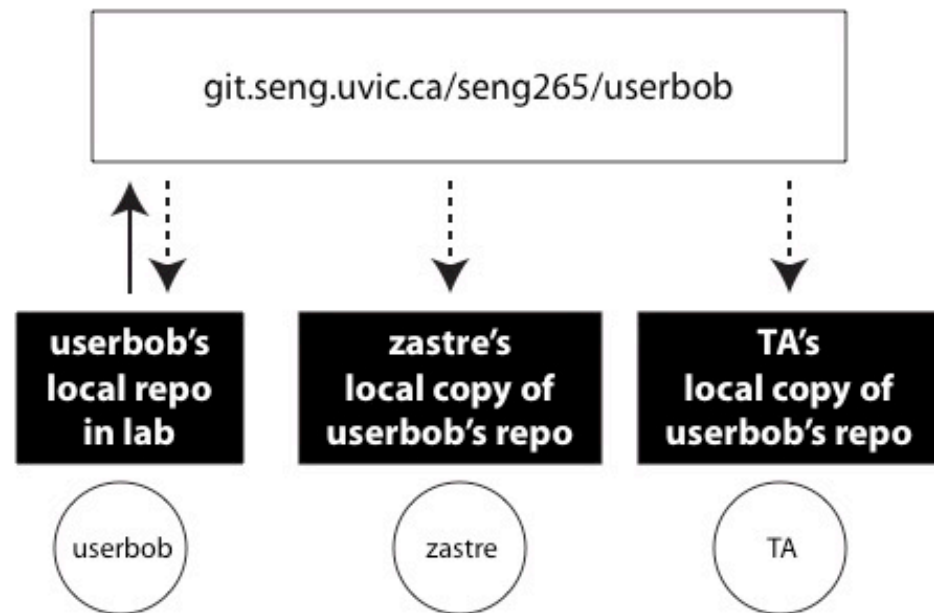
Basic general git cycle (assuming shared repo)

1. Either:
   - perform a **clone** to make a working copy of some remote repo's branch, or
   - perform a **init** plus other actions to connect this new local repo with a (possibly empty) remote repo branch
2. In your working copy of the project (which is a directory):
   - edit files or
   - create files the project or
   - do both
3. If needed, update our local copy with a **pull** (and perhaps also a **merge**)
   - this picks up changes made by team members / project participants since your last update
4. build / run / test / view / render / read / <fill-in-verb> your work
5. **add** the name of changed files that are ready to be committed in your local repo ("staging")
6. **commit** your changes to your local repo
7. if changes are not yet ready to be sent on the remote repo, go to 2
8. **push** committed changes to the remote repo
9. go to step 2

Basic Concepts: Working Copy

- A **working copy** is an ordinary directory on your local system
- These are the files you edit
- When changes are in at a suitable stage, add & commit your changes in your local repo
  - In practice, this usually means metadata files in the .git directory within the working copy are added or changed (.git directory not shown)
- You may even decide to push committed changes to the remote repo
- Challenges:
  - Files already changed? must merge!
  - May need to maintain .gitignore
  - Can get confusing with branches
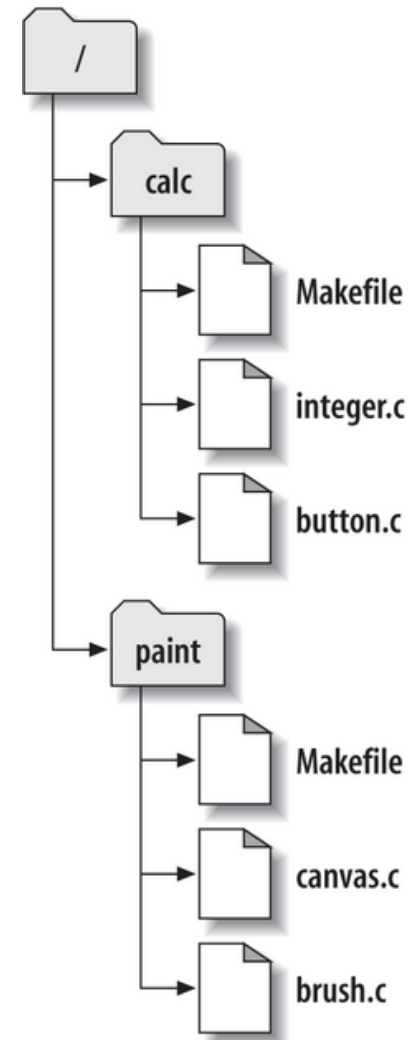
Basic Concepts: Working Copy

- Obtaining a working copy means either **cloning** an existing repository or performing **init** in some existing directory
  - This is normally done **only once** per working copy
- Repository access methods differ:
  - direct access via local disk (file:///) **(ugh!)**
  - via ssh:// (**we'll use ssh in the labs and for assignments**)
  - http or https (as used by GitHub and GitLab)
  - original access method for working copy / local repo is stored as metadata in a .git directory in that working copy)

Basic Concept: Working Copy

- Example: get working copy of the "calc" project from git.example.com

```
$ git clone ssh://stewie@git.example.com/repos/calc
Cloning into 'calc'...
<password for stewie>
remote: Counting objects: 37, done.
remote: Compressing objects: 100% (31/31), done.
remote: Total 37 (delta 5), reused 0 (delta 0)
Receiving objects: 100% (37/37), done.
Resolving deltas: 100% (5/5), done.
Checking connectivity... done.

$ cd calc
$ git ls-files
Makefile
button.c
integer.c

$
```

Basic Concepts: Commit

- Suppose you wish to make a change to **button.c**
  - You edit the file using your normal workflow
  - Time and date on edited file will be more recent than time and date of file in repository
  - Changes are published by committing your changed file to the repository
- We first stage our changes (**add**)…
- … and then make a "permanent" record of the change (**commit**)

```
$ pwd
calc

$ git add button.c
$ git commit -m "Fixed the geometry of button for v3 of library"
[master 12788ce] Fixed the geometry of button for v3 of library
 1 file changed, 1 deletion(-)
$
```

Basic Concepts: Commit

- Note that commits always to our local repo
  - i.e., take place within our working copy
- We can have many such commits as we work through sets of changes
- **To have such changes available to others, we must push them to a remote repository**
- Observations:
  - Commits can be quite frequent (if needed by our workflow)
  - **Pushes are much rarer** (perhaps reflecting that our work has reach some suitable state and is ready for others to pull).
  - Example: writing each assignment will require several commits
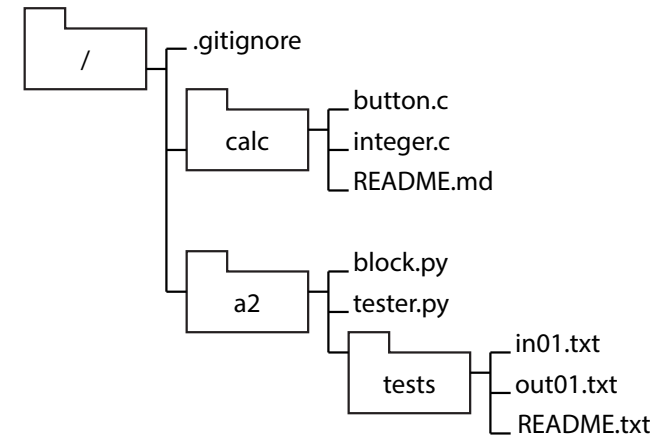 - Example: submitting an assignment will require a push

Basic Concepts: Commit

- Each commit results **in a new snapshot of the code** in our working copy
- Snapshots are kept in chronological order
  - "git log" produces a list of the snapshots
  - Default log output order is reverse chronological (i.e., most recent commit/snapshot is listed first)
- "git status" reports the relationship amongst files in our working directory with what within git's local repository
  - More precisely, "status" tells us what has changed in our working directory…
  - … and therefore what may need to be "add"ed and "commit"ed to the local repository
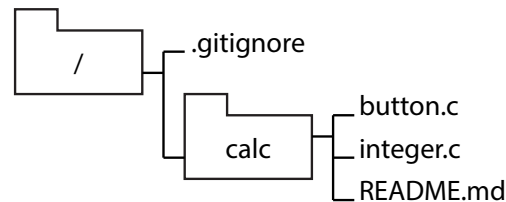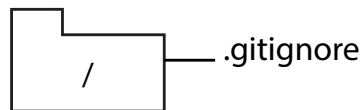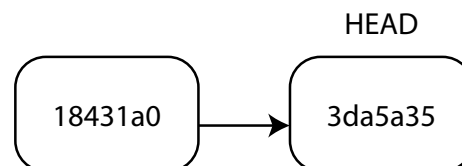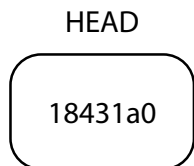
(steps lead to third commit)

```
$ pwd
/home/stewie/project

$ git add *  # Let git know what files are to be committed; recursive on directories

$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

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

    new file:   a2/block.py
    new file:   a2/tester.py
    new file:   a2/tests/README.txt
    new file:   a2/tests/in01.txt
    new file:   a2/tests/out01.txt

$ git commit -m "Now the A#2 directory (a2) is in place"  # Note message
[master 85687aa] Now the A#2 directory (a2) is in place
 5 files changed, 6 insertions(+)
 create mode 100644 a2/block.py
 create mode 100644 a2/tester.py
 create mode 100644 a2/tests/README.txt
 create mode 100644 a2/tests/in01.txt
 create mode 100644 a2/tests/out01.txt
```

Basic Concepts: Commit

```
$ git log
commit 85687aa056e299897153a3125c1826f64581bdc5
Author: Stewie Griffin <stewie@uvic.ca>
Date:   Thu Sep 15 10:05:54 2016 -0700

    Now the A#2 directory (a2) is in place

commit 3da5a353956c320fbe8e585cd692b173e44b06c1
Author: Stewie Griffin <stewie@uvic.ca>
Date:   Thu Sep 15 10:01:57 2016 -0700

    Added calc and some files

commit 18431a0b85f0645c98e4cceb311074594a19a38d
Author: Stewie Griffin <stewie@uvic.ca>
Date:   Thu Sep 15 09:38:34 2016 -0700

    Initial commit (just .gitignore for now)
$
```

HEAD

18431a0 → 3da5a35 → 85687aa

Basic Concepts: Commit

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean

$ git push
<verbiage>
<password for stewie>
Counting objects: 13, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (12/12), 1.11 KiB | 0 bytes/s, done.
Total 12 (delta 0), reused 0 (delta 0)
To ssh://stewie@git.example.com/project
   18431a0..85687aa  master -> master

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

$
```

commit != push

- git is different from many other kinds of VCSes
  - commits are made to the local repo, **not the remote**
  - making these changes available to others means transferring data from the local repo to the remote repo (i.e., a push)
- git separates **the tracking of file/directory changes** from **their storage on remote servers/repositories**
  - this is very different from Subversion, Perforce, etc.
  - can seem a bit confusing at first sight

commit != push
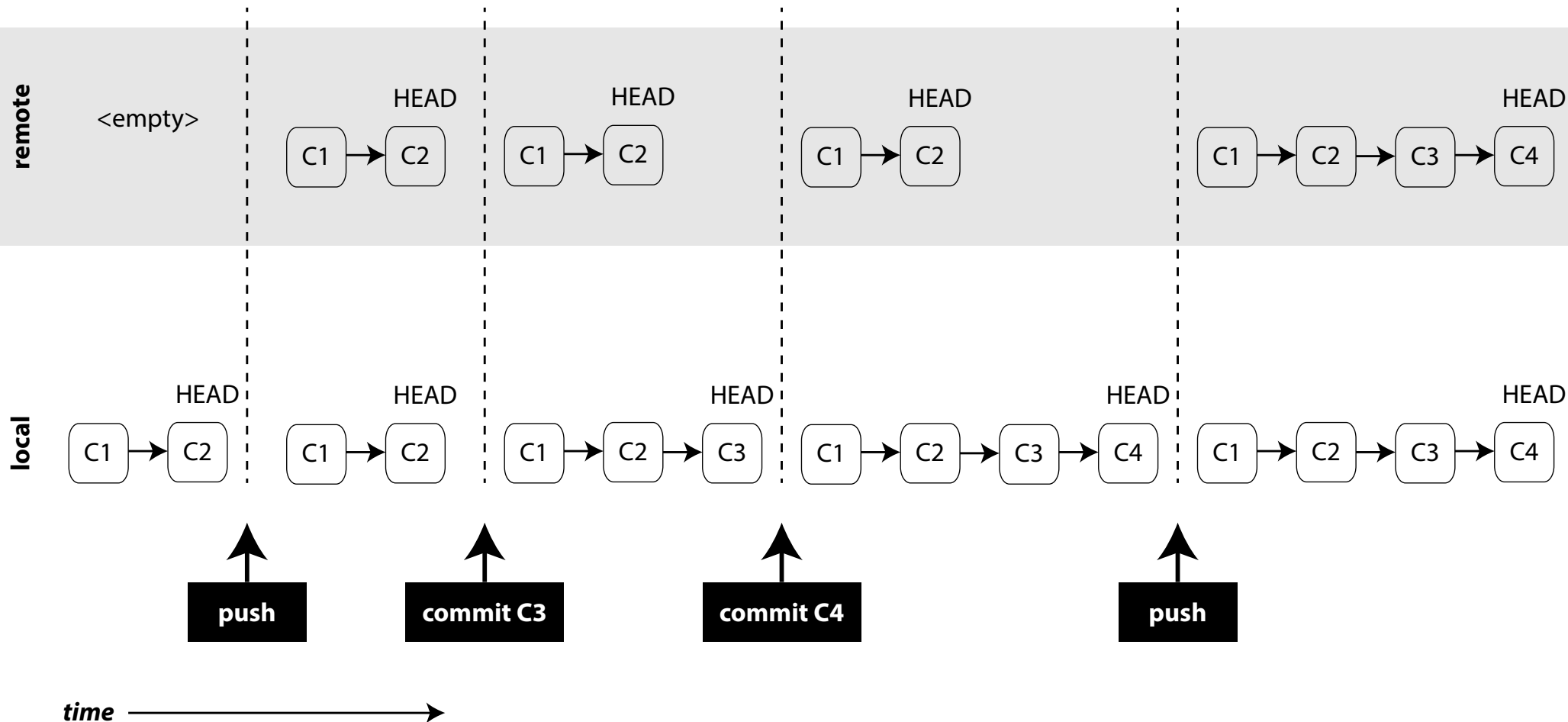
Basic Concepts: Update

- What if Meg starts working on the project after someone else's commit?
  - Assume she made a working copy of Stewie's repo some time ago
  - (Also assume here she has read & write privileges on Stewie's remote.)
  - Let's also assume she was not working on button.c in calc
- She can ask git to bring her working copy "up to date"
  - git will only update files for which there are changes on the remote
  - Principle: Make sure to update often if working with a group on a project that uses a repository!

```
$ pwd
/home/meg/calc

$ git pull
<password for meg>
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
<... snip ...>
Fast-forward
 calc/button.c | 1 +
 1 file changed, 1 insertion(+)

$
```

Basic Concepts: Update

- "git pull" is actually two commands together:
  - "git fetch" followed by "git merge"

```
$ pwd
/home/meg/calc

$ git fetch
<password for meg>
remote: Counting objects: 7, done.
<snip>

$ git log --name-status
commit eb6c8a6ffb2e2a70a89e4a89db8d62b5a22eccd4
Author: Stewie Griffin <stewie@uvic.ca>
Date:    Thu Sep 15 10:27:23 2016 -0700
      Added headers so buttons can be beveled


M       calc/button.c
```

Basic Concepts: Update

```
$ pwd
/home/meg/calc

$ git merge origin/master
Updating 85687aa..eb6c8a6
Fast-forward
 calc/button.c | 3 +++
 1 file changed, 3 insertions(+)
$
```

- Therefore "git pull" is the same as the following two commands in succession
  - git fetch
  - git merge origin/master
- We used "git log --name-status" to obtain the names of files that are different from our working copy and the remote repo

What is with "origin"? "master"?

- A working copy / local repo may be associated with:
  - No remote repo, or
  - **One remote repo** (UVic SENG 265) or
  - Several remote repos.
- A working copy / local repo may have:
  - No code branches tracked by git, or
  - **One code branch** (UVic SENG 265) tracked by git, or
  - Several code branches tracked by git.
- By git convention, the default remote repo is named **origin**.
- By convention, the main branch of code development is named **master** (i.e., it is the "master" or "main branch" of code development)

What is with "origin"? "master"?

```
$ pwd
/home/stewie/calc

$ git remote -v
origin  ssh://stewie@git.example.com/repo/calc (fetch)
origin  ssh://stewie@git.example.com/repo/calc (push)

$ git branch -v
* master b5b22e2 Bevels now in place
```

- When using "git pull" and "git push" for repos already cloned:
  - origin & master are **usually** default values
  - "git pull" == "git pull origin master"
  - "git push" == "git push origin master"
- Note: We'll discuss branching workflows later in the term

What is with "origin"? "master"?

HEAD

```
┌──────────┐      ┌──────────┐      ┌──────────┐
│ 18431a0  │ ───► │ 3da5a35  │ ───► │ 85687aa  │
└──────────┘      └──────────┘      └──────────┘
```

- This diagram we saw earlier of our commits / snapshot is an example of a **branch**
- The git convention is that every repo has **at least one branch** which is the main branch
  - Usually referred to as the **master** branch.
- Above is shown master branch from slide 24 (i.e., project with "calc", "a2" directories and their subdirectories/files).

Adding gotcha

- This is a bit more subtle in git
- **add** results in a file or directory being staged for commit
- When **commit** is performed, changes to staged files are stored into the local repo
- Note, however:
  - Every file or directory in the project that is to be tracked by git needs to be added at least once in the project's lifetime
  - Also: add gives us fine-grained control as to what needs to be in a commit's snapshot
  - Sometimes, though, we just want all of the changed files to be staged and committed without having to use add
  - **git commit -a -m "message"**

Never store generated files in the repository.

For example, if your project includes C source code, you would store the .c and .h files. <u>**You would not store the .o or executables.**</u>

If your project includes Java source code, you would store the .java file, but <u>**you wouldn't store the generated .class or .jar files.**</u>

Use .gitignore

- This text file needs to be committed to the project
- Normally stored in the top-level of the working directory
- Each line in the file is used as a pattern
- Possible entries:
  - *.pyc
  - *.o
  - DS_STORE
  - *.class

What about "conflicts"?

- Suppose you've fetched changes and merged them into your master branch
- Normally we see a clean report of git's work
  - Downloading objects from remote
  - Merging changes into our working copy (i.e., new files, removed files, edits to existing files)

```
$ git pull
<password>
remote: Counting objects: 19, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 18 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (18/18), done.
From ssh://git.example.com/repo/calc
   b5b22e2..4955e13  master     -> origin/master
Updating b5b22e2..4955e13
Fast-forward
 calc/README.md      | 4 ++++
 calc/main_init.py   | 3 +++
 calc/guiframe.py    | 1 +
 3 files changed, 8 insertions(+)
 create mode 100644 calc/README.md
 create mode 100644 calc/main_init.py
 create mode 100644 calc/guiframe.py
```

What about "conflicts"?

- However, sometimes we don't see a clean report

What about "conflicts"?

- However, sometimes we don't see a clean report
- Scenario:
  - Meg has pulled the remote. Both she and Stevie have the first version of button.c shown here.

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

What about "conflicts"?

- However, sometimes we don't see a clean report
- Scenario:
  - Meg has pulled the remote. Both she and Stevie have the first version of button.c shown here.
  - Stevie makes a change to calc/button.c, **but doesn't commit**

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>

/* I, Stevie, am the smartest Griffin */
int _init_button (int id, int skin,
    char *label) { /* A stub, I say! */ }
```

What about "conflicts"?

- However, sometimes we don't see a clean report
- Scenario:
  - Meg has pulled the remote. Both she and Stevie have the first version of button.c shown here.
  - Stevie makes a change to calc/button.c, **but doesn't commit**
  - Meg makes a change to calc/button.c, **then commits and pushes it**.

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>

/* I, Stevie, am the smartest Griffin */
int _init_button (int id, int skin,
    char *label) { /* A stub, I say! */ }
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>

void _init_button (int code, int look,
    char *title) { /* For you, Connie! */ }
```

What about "conflicts"?

- However, sometimes we don't see a clean report
- Scenario:
  - Meg has pulled the remote. Both she and Stevie have the first version of button.c shown here.
  - Stevie makes a change to calc/button.c, **but doesn't commit**
  - Meg makes a change to calc/button.c, **then commits and pushes it**.
  - Afterwards Stevie tries to commit and then push

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>

/* I, Stevie, am the smartest Griffin */
int _init_button (int id, int skin,
    char *label) { /* A stub, I say! */ }
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>

void _init_button (int code, int look,
    char *title) { /* For you, Connie! */ }
```

What about "conflicts"?

```
meg$ git add button.c
meg$ git commit -m 'That will be a swell looking button!'
[master bda4b22] That will be a swell looking button!
 1 file changed, 4 insertions(+)
meg$ git push
<password>
<... snip ...>
Total 4 (delta 1), reused 0 (delta 0)
To ssh://stewie@git.example.com/repo/calc
   15c486a..bda4b22  master -> master
```

```
stewie$ $ git commit -a -m "A sterling job, I say, with _init_button."
[master 6708583] A sterling job, I say, with _init_button.
 1 file changed, 4 insertions(+)
stewie$ git push
<password>
<... snip ...>
error: failed to push some refs to 'ssh://stewie@git.example.com/repo/calc'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
stewie$ curses!
-bash: curses!: command not found
```

What about "conflicts"?

```
stewie$ $ git pull
<password>
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From ssh://git.example.com/repo/calc
   15c486a..bda4b22  master      -> origin/master
Auto-merging calc/button.c
CONFLICT (content): Merge conflict in calc/button.c
Automatic merge failed; fix conflicts and then commit the result.

stewie$ git status
-On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")

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

        both modified:    button.c

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

Resolving a merge conflict (what Stewie sees)

```
/* button.c */

#include <stdio.h>
#include <gui.h>

<<<<<<< HEAD
/* I, Stevie, am the smartest Griffin */
int _init_button (int id, int skin,
    char *label) { /* A stub, I say! */ }
=======
void _init_button (int code, int look,
    char *title) { /* For you, Connie! */ }

>>>>>>> bda4b226dc91226a0ab310ad4a7feef2c069b4b4
```

- git indicates the conflict with a bit of markup
  - "<<<<<<< HEAD" to "=======": Stevie's original code
  - "=======" to ">>>>>>> bda4b22…": What Meg pushed

Resolving a merge conflict

- Before Stewie can successfully commit and push his work to the remote, he must resolve the conflict
- There are two parts to this:
  - The **human part** (harder)
  - The **technical part** (easy)
- The human part is to decide what should code in button.c should be kept
- The technical part is editing the file in a way reflecting that decision, then commiting (and possibly pushing)

Resolving a merge conflict

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

before

```
<<<<<<< HEAD
/* I, Stevie, am the smartest Griffin */
int _init_button (int id, int skin,
    char *label) { /* A stub, I say! */ }
=======
void _init_button (int code, int look,
    char *title) { /* For you, Connie! */ }

>>>>>>> bda4b226dc91226a0ab310ad4a7feef2c069b4b4
```

```
/* button.c */

#include <stdio.h>
#include <gui.h>
```

after

```
int _init_button (int id, int skin,
    char *title) { /* This stub is to be completed */ }
```

University of Victoria
Department of Computer Science

What about "conflicts"?

```
stewie$ $ vim calc/button.c
<actions to edit the file>

stewie$ git commit -a -m "button.c now back on track"
[master 8d0c566] button.c now back on track

stewie$ git push
<password>
<... snip ...>
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 1.04 KiB | 0 bytes/s, done.
Total 8 (delta 2), reused 0 (delta 0)
To ssh://stewie@git.example.com/repo/calc
   bda4b22..8d0c566  master -> master
```

```
meg$ git pull
<password>
<... snip ...>
Unpacking objects: 100% (8/8), done.
From ssh://git.example.com/repo/calc
   bda4b22..8d0c566  master     -> origin/master
Updating bda4b22..8d0c566
Fast-forward
 calc/button.c | 5 ++---
 1 file changed, 2 insertions(+), 3 deletions(-)
```

Using git

- **You need a git client**

- In this course we use a command-line client
  - `git` (usually `/usr/bin/git`)
  - provide `git` with commands and arguments

- Note other client possibilities:
  - Git functionality might be built into IDE
  - Git functionality built into tool
  - Some web interfaces for using git (especially needed for GitHub)

Using git

- **For UVic SENG265, you need a repository**
- **Need access to that repository**
- UVic Software Engineering hosts a repository for you:
  - this is on a per-course basis
  - `ssh://yourlogin@git.seng.uvic.ca/seng265/yourlogin`
  - replace "`yourlogin`" with your Netlink ID
- Other possibilities you might encounter:
  - `gibhub`-like services
  - BitKeeper
  - administering your own server (careful!!)

(a wee word…)

- For this course, use the seng265 repository
  - Do not use github!
- Each of your assignments and labs …
  - … will be subdirectories within your respository
  - … which are accessible to the lab instructors and administrators for help when debugging (but accessible to no one else!)
  - … and can be accessed remotely by you

git commands

- Note:
  - We've already referred to git "commands"
  - Yet `git` itself **is a UNIX command**
- A `git` command is how we specify an action from the `git` client
- Syntax
  - `git` command [option] [arguments]
- The number of git commands and options is very large
  - We'll be focusing on a much smaller subset of these.
  - (Beware of Google and StackOverflow as answers there can lead you astray…)

Previously seen

Git command	argument
$ git clone	ssh://stewie@git.example.com/repo/calc
$ git add	button.c
$ git fetch	

Git command   **argument**

A few more examples

Git command	options & arguments
$ git remote	add origin https://git.420.com/fubar
$ git commit	-a -m "Fixed typo in label"
$ git diff	--name-status HEAD

Git command   **options & arguments**

Some useful commands

- `clone` — make a local copy of a remote repo
- **add** — stage files/directories so they'll be ready to be committed
- **commit** — store into local repo a snapshot of working-copy changes
- **status** — list working copy files/dirs differing from local repo
- **log** — output commit messages with their snapshot SHA-1 hashes
- `diff` — show differences of working-dir contents with local repo
- `pull` — fetch and merge into local repo any remote repo changes
- **push** — transfer local repo snapshots to remote repo
- `fetch` — download data for remote changes to local repo (but no more!)
- `merge` — combine new remote changes with files in local repo
- **rm** — remove file from working copy / working tree
- `init` — convert working directory into git local repo (caution!)

If you need help…

- For a specific command:
  - `git <command> --help`

  - Provides list of arguments and options

- For info on repository access
  - speak to the provider, or

  - read the provided documentation

  - follow the instructions given in the lab and on the assignments!

A few further topics for later in term

- branch creation, selection and merging
- more complicated merge-conflict resolution
- tags
- multiple remotes
- diffs and blame (and logs in general)
- patches
- setting up repos (init) and setting up git servers