A Guide to Git

Introduction

Disclaimer: This handout has been modified from Harvard's guide to Git

What is Git?

Git is a distributed source code management (SCM) system. Source code management systems, also called "version control systems", are tools that allow groups of programmers to work together on the same projects without getting in each other's way any more than is necessary. They do this in three primary ways: first, they identify a particular collection of files as "the project"; second, they keep track of changes so multiple copies of the project (belonging to multiple programmers) can be kept synchronized; and third, they keep track of history so previous versions of the project can be examined. A good SCM system helps keep a project organized and provides ways for the programmers involved with it to coordinate with one another and do their work in parallel.

The major conceptual difference between a distributed and centralized SCM system is that while a centralized SCM system has a designated master copy of the repository, distributed SCM systems do not. Different copies of the repositories are peers and no one system has a special status, although a group of programmers can decide by convention to treat some copy specially (in 4DM4, you might think of the copy of your code that lives on Github Classroom as "special", since that is what you will use to share code with your partner and to submit assignments).

Why source management?

In most computer science courses, each assignment is a distinct unit: you sit down and code something up, you hand it in, it gets graded, and you immediately forget about it or even throw it away. In this environment, a distributed source management system is not really necessary. (You might still find a SCM system useful to help you keep track of the changes you make in your assignment as you work on it.)

In the real world, however, most programs are large and expensive to develop, so their life-cycles are measured in years or sometimes decades. Over these time scales, and with such large amounts of code, just keeping track of everything becomes a major problem.

Worse, in the real world, programs have users, who are not part of the development team and not (generally) interested in internal details of the program. Usually, someone insists that now and then a new version be made available to the users. The development team has to be able to issue these releases, and then also has to be able, for years afterwards, to handle bug reports and sometimes issue fixes. In this environment it is imperative to be able to go to some official place and get a copy of the precise release you need.

And finally, when you have a number of programmers working on the same program at once, it's essential that some mechanism be put in place to allow them to coordinate their work. Otherwise, each programmer's version slowly diverges from the others, and eventually everyone has a private version different from everyone else's... and none of them work. Once this happens, it takes an immense amount of effort to straighten out the mess.

Source code management (or version control) systems are designed to help programming teams handle these issues.

In 4DM4, the probable lifetime of your project is a few months, not a few years, and most likely no more than two people will be working on it at once. Furthermore, the source you will be working with is several orders of magnitude smaller than a large real-world project. Nonetheless, it is large enough, the time is long enough, and there are enough people involved that failure to use some kind of source management system would be an act of reckless folly.

Why Git?

We require the use of Git in 4DM4 because it is powerful, freely and widely available, widely used, and becoming ever more popular. Many large open-source and proprietary projects are managed using Git.

Remainder of This Document

The rest of this handout is divided into two main sections. The first explains the philosophy of Git, its operating model, and the assumptions behind the way it works. The rest explains, in terms of how one actually uses Git rather than its various commands, a number of basic and not-so-basic Git operations. A small additional section lists the main Git commands.

You do not need to remember everything in this handout. In fact, this handout was written mostly so you *don't* have to remember it all. When you read it the first time,

make sure you have a good mental model of how Git works and how we are using it. Then, use this as a reference to find out how to do particular tasks.

The World According to Git

Distributed Model

The Git model assumes that there is no one central official master copy of anything. Instead, many copies (each a full-fledged *repository*, or "repo") can be made. At the root of your local repository, you can find a directory called `.git`. This directory stores all of the information of your repository, including your project history. If you delete this directory, your repo is gone, although the current version of the files hasn't been touched.

In many projects, one particular repo somewhere is considered "official" for administrative purposes; however, from a technical standpoint this repo is no different from any other. In this class, this repo will live on Github Classroom, so that you and your partner can exchange work with one another (and we may see your work).

When you wish to work on a pre-existing Git-managed project, you *clone* your own private repo of the project. Your private repo is usually called the "local" repo, while any other repo (including the one you cloned) is a "remote" repo.

Since work on a program is an ongoing process, and other people may be working on the same program at the same time, you generally want changes made by other developers to appear in your own private repo. Accomplishing this requires *pulling* the changes that the other developers have made and *merging* them with your local repo. In general, you do not pull directly from the other developer's local repositories. Instead, they *push* their changes to some central repository (e.g., GitHub), from which you can then pull the changes.

The act of requesting and receiving history from another repo is called "pulling" and is done with "git pull". When you clone a Git repository, your new repo contains a link back to the parent repository. If you pull without specifying a repository from which to pull, this parent is the repo from which you will pull changes (this default can be changed). You can also pull new changes from any other repo of the same project that you have access to; this allows, for example, collaborating with someone else on changes that are not yet ready for prime time.

Push is the inverse of pull: it sends your changes to another repository. As with pull, the default location to which a push applies is the parent repository, but you can also

specify any repo to which you have sufficient access. The only difference between "pull" and "push" is in which direction data flows.

Note, however, that changes do not propagate to all repos automatically. You must explicitly pull changes to get them into your local repo when they become available elsewhere (unless someone has permission to push changes to your repo, but this is an unlikely setup for this course). And you must explicitly push changes if you want them to appear somewhere else (e.g., at Github Classroom, so your partner can then pull). You and your partner should discuss what your group policy will be for pushing and pulling. (Pushing changes that don't compile or will break your partner's repository will probably not be conducive to group harmony.)

Working Copy of the Project

Your repo has two pieces: the history (in the `.git` directory) and the actual files you are currently working with. If you happen to be using your history to look at a previous version of the code, your working copy of the files will not be the most recent version represented in the history.

As you make changes to your files, you must *commit* (think "save") them to your history. Given just a .git directory, one can recover the most recent version of files that has been committed to that repository. `Push` and `pull` exchange histories (commits), and pulling will automatically update your working copy of the files -- generally, you will be forced to commit before you are allowed to push or pull. Changes that have not yet been committed do not exist as far as Git (and your history) is concerned.

Accessing Repositories

Git provides a variety of ways for accessing repositories/clones that might be anywhere on the Internet. A repository name can be the name of a (local) directory; however, it can also be a URL. Repositories can be accessed over HTTP (`http://...`), HTTP with SSL (`https://...`), via `ssh`: `ssh://machinename/path`, and via the git protocol: `git@github.com:somerepo.git`.

You can have as many cloned repos of your own as you want. Often you will have only one. However, it often makes sense to have a separate clone on every machine or cluster you work on; e.g. in 4DM4 you might keep clones on a desktop you have, on the lab machine, and also on your laptop. This allows you to work locally in various contexts, and the clones can be synced up easily. When you are done working on a Science Center machine, you can commit and push your changes to Github Classroom, and then pull those changes when you get on your laptop. There are also

circumstances in which it is convenient to have several or even many clones on the same machine, and sometimes you will make temporary ones.

Any clone can itself be cloned, and sometimes you will use this ability on your own private clones. Just remember that each clone works independently, and that changes can be pulled or pushed from any clone to any other clone according to whatever structure you wish. (If you set up a complicated structure, though, it is usually a good idea to also make a `README` file documenting that structure.) Some examples of working setups with multiple local clones are given later.

Changesets and Merging

Each batch of changes someone commits, simply called a *commit* (or sometimes *changeset*), is treated as a new version of the project. Commits are identified by applying a cryptographic hash to the changes; this gives a long hexadecimal code number that globally identifies the commit.

Each commit is based on some specific earlier version. Commits are viewed as "simultaneous" from Git's perspective if they are based on the same previous version of the code. This can easily happen for two remote repos -- for example, your partner and you are independently working on different features, starting from the same code base -- or even locally, if you happen to be working on a separate *branch* of code. Such simultaneous commits create separate *heads* of the codebase (there are multiple "most recent" versions of the code). Even more changes can be committed on top of any or all of these heads, These new versions of the codebase can diverge arbitrarily if desired; however, generally that is not desired (e.g., you and your partner want to combine the features you each made), so multiple heads are combined by *merging*. (An alternative to merging is rebasing, which many prefer, but we won't go into here.)

Merges of commits that change disjoint sets of files are easy. Merges of commits that make small unrelated changes in the same file will also go through automatically. However, overlapping edits result in *merge conflicts* which need to be resolved manually. If two or more people have made different sets of sweeping changes to the same file at once, this editing can become a nightmare. For this reason, while in the distributed model anyone can commit anything anytime, it is always a good idea to coordinate with other programmers on the project before embarking on major or intrusive rewrites.

If a merge fails, files that require editing will have blocks in them that look like this:

```
int foo(void) {
<<<<<<< local
    bar();
```

```
=======
    baz();
>>>>>>> other
}
```

This means that you changed `foo` to call `bar`, but the other version you merged with changed it to call `baz`. To fix this you must pick one or the other (or edit into some other form entirely) and remove the markings. Git remembers which files need hand-merging, so you need to tell it that you fixed ("resolved") each broken file before it will let you commit the merge. Note however that there's no way it can cross-check what you did; it's up to you to do the right thing and not mark files resolved until they really are. `git mergetool` can be helpful if there are a lot of merge conflicts.

Because merging two commits is itself a change, you must commit it for it to officially exist. If there are no conflicts, this commit can happen automatically. If there are conflicts, you need to commit after fixing the conflicts. If you forget to commit the merge, or you forget to do a merge at all, or someone has committed and pushed other changes upstream that you haven't pulled yet, pushing your changes will fail.

Some notes on merges:

1. Even when there are conflicts, the conflict blocks do not necessarily reflect all the changes associated with the merge. Some may have merged successfully. Occasionally, they may have merged "successfully" but be wrong. If in doubt, look at diffs.
2. Also, sometimes the conflict block delimiters don't contain everything that may be involved in resolving a conflict correctly.
3. While the merge system is reasonably robust, once in a while it makes a mistake, particularly if some but not all of the changes merged. It's prudent to look at diffs after an automatic merge, just in case. Git has a fairly sophisticated merge algorithm, but it's still only an algorithm and has no brain.
4. Merging is painful. Merging a big change is a lot more painful than merging the same amount of change a bit at a time. Push and pull early and often. Commit early and often.

If you are planning to make huge changes to a file, like reordering all the functions or moving large blocks of code into if clauses (which changes the indent, making Git think everything changed), it's a good idea to coordinate manually with anyone else who might have pending changes to the file.

Log Messages

When you commit changes to a Git repository, Git gives you the opportunity to provide a message explaining the change. These messages get saved in the project history and can be reviewed later using `git log`. This can be quite useful when trying to reconstruct the thinking that led to some piece of code you wrote months previously.

These messages can also be logged to some central point or mailed out to the people working on the project. It is possible to set up your Git repository to mail commit messages to you and your partner. (See below.) While the volume of mail thus generated can be irritating, there is no better way to stay in touch with what's going on.

The commit message should thus describe (briefly) what you did and why. The first line should be a short summary; this is all `git log` prints unless you give it the `-v` option. There is no need to report the exact changes, as they can be retrieved using `git diff`.

When to Commit and Push

The general rule for commits is that any change should be committed as soon as you're reasonably certain that it's correct and appropriate in the long term. This is also the general rule for pushing: as soon as you're reasonably certain that a change is correct and appropriate in the long term, you should push it out so it's available to the people you're working with... subject to the proviso that committing and pushing many small changes in quick succession tends to annoy people.

Because in Git (and other distributed SCM systems) commit is local and separated from push, you can commit as much as you want whenever you want without interfering with anyone else's work, and wait to push until you're ready to inflict your changes on your partner. This means that sometimes it makes sense to commit changes that you know *aren't* correct, just to checkpoint them, or so you can push your current code to your laptop so you can work remotely. However, such incorrect or broken changes are generally best done on a separate *branch* instead of on the "master" branch. At the very least, the commit message should contain something like "WIP" (work in progress), so that your partner knows things will be broken if he or she tries to use that version of the code.

Remember that your partner will not see anything you do not push to Github Classroom, and you can't push newer commits before older ones. When developing a big new feature, it often makes sense to do so in a branch, so that if a bug in the "stable" version of the code arises, you can go back to the master branch, make the

quick fix and push it, and then merge the new change back in with your "new feature" branch.

Ideally you and your partner should keep track of which tests you expect to work at any particular time, and before committing and pushing out new code check to make sure that they all still do work.

In most cases, one should try to avoid pushing out changes that cause the program to stop working properly (or, even, stop compiling at all.) This rule can sometimes be profitably bent when you know your partner will not be affected by the errors introduced.

Tags

While Git uses global version numbers, these versions are hash codes, not numbers, and are annoying to work with. Git supports a concept known as a *tag*, which is a symbolic name (such as `asst4-debugged`) that you attach to a particular commit of your project. You can then refer to that version of those files with the tag name.

A tag identifies a single consistent version of an entire project. For instance, the directions for each assignment (other than Assignment 0) tell you to create one tag before starting and another tag after you are done. These tags then identify the versions of all your files that were current before and after you did the assignment. This lets you, for example, ask Git to show you all the changes in the entire system between those two points.

See below for specific directions for manipulating tags with Git.

Branches

Sometimes you might have more than one "line of development" in your program. For instance, when you ship release 1.0 to customers, you might have one team working on release 2.0, and another team making minor bug fixes to the release 1.0 code for release 1.01.

In this case, most changes made for release 2.0 should not be incorporated into release 1.01, and while many fixes made for release 1.01 should be incorporated into release 2.0, some probably should not be.

This sort of situation is handled using "branches". Each branch is a (mostly) separate line of development, diverging from some common ancestor version. (This divergence is where the term "branch" arose.)

In Git every new commit is potentially its own branch, since you are creating a version of the code that may differ from your partners. These lightweight branches are generally resolved quickly by merging. However, note that for safe git usage you *must* explicitly create a branch every time you diverge from a single line of development; otherwise git will print a scary message about detaching your head, and then any changes you commit will be treated as garbage and deleted.

Use Git Effectively

Git (or any version control system) is a tool, not a panacea. It helps you organize and maintain a project, but it does not do it by itself. It requires that you use it in a manner that makes it useful.

In order for the system to be useful for keeping track of what's really part of the project and what isn't, you have to actively maintain the set of files Git knows about. Do not add or commit temporary files, editor backups, object files, and the like to the Git project history. If you have files that are complicating your development process that you do not want to commit, add them to the `.gitignore` file at the top level of the tree. (This is discussed in more detail below.) Do remove files you are not using any more. (You can still get them back later, because they are part of the project history and removing them is just a change that Git tracks.)

In order for the version history to be useful, you have to add tags at important points in development, like releases. You also must write at least minimally useful commit messages so you can look at them later and be reminded of the circumstances.

In order for the merging features to be useful, you have to avoid making sweeping changes without warning your partner, you have to pull regularly, you have to commit and push regularly but not insanely often, and you have to take the trouble to merge correctly by hand when conflicts occur.

If you do not do these things, you will eventually end up in a hole, and Git will not save you from yourself.

How do I...

The previous section explained Git concepts in general terms. In this section we explain how to do various useful things.

How do I set up my username?

You should set up a username and email in Git, so that your commits will show that you authored them, and how to contact you if necessary. To do this, you can run:

```
git config --global user.name "Mohamed Hassan"
git config --global user.email "Mohamed.hassan@mcmaster.ca"
```

Caution: the name and email is attached to every commit and can't readily be changed or removed afterwards, so don't use an email address that you want to keep protected from spammers. For 4DM4 purposes using your full name is certainly sufficient.

How do I make a new repository?

Go to the root of the directory you want to make a repository out of, and type

```
git init
```

This will create a `.git` directory that will keep track of your commits.

How do I make a new project in a repository?

You do not - in almost all cases if you have a separate project you want a separate repository for it.

How do I clone an existing repository?

With `git clone`, like:

```
git clone git@github.com:somerepo.git
```

How do I add stuff to a repository?

Create some files and/or directories and run `git add` on them:

```
git add newfile
```

If you run

```
git add –all
```

it will add all the new files it can find. Do not forget to commit; adding only tells Git to keep track of new files.

How do I commit?

Simply run:

```
git commit
```

You will be prompted to enter your commit message. If you want to do that in one step, you can run

```
git commit -m "Your commit message"
```

, but this can be difficult if you want longer commit messages. The files committed are only those *explicitly* added to the commit.

You can use:

`git commit -a` (or `git commit -am "Your message"`)

to just add all files that have changed that Git knows about to the commit. However, if you have created a new file, then you must explicitly `git add` it in order for it to be included in a commit -- `git commit -a` will not know to commit a new file's changes. Do not forget to push after committing if you want your partner to see your code!

How do I see what has been changed since the last commit?

Run `git status`. This will show the status of the whole working tree by default, or you can run it on individual files or subdirectories. This will show which files have been modified; it will also show files that exist, but Git does not know about (haven't been `git add`ed). In general, this case should be rectified; files that exist but are not tracked by Git should either be added or explicitly ignored. Checking the status is useful before committing, when you first sit down to work to remind you where you were, and generally at any other time too.

How do I ignore stray files?

In most projects, compiling causes build products (`.o` files, for example) to appear in the tree. These will then show up under `Untracked files:` when you run `git status`, which is untidy and gets in the way of seeing real status information. To ignore files, create a file `.gitignore` in the top level directory of the project and add to it the path matching the pathnames of the files you want Git to ignore. For example, `build/*.o$` ignores all `.o` files in the build directory.

How do I remove files?

When you wish to remove a file or directory from the tree, run:

```
git rm
```

on it. This will both delete the file itself and record the deletion in your history. Commit at a suitable point afterwards.

It's usually a good idea to compile the project after removing but before committing, just to make sure you aren't breaking things.

Remember that files that have been deleted are still kept in the project history. They'll disappear from people's working trees by default, but you can still look at them, and you can bring them back again later if needed.

How do I add and remove directories?

You do not. Git doesn't track directories as such; directories are created if there are files to put in them and are removed automatically when the last file is removed. This makes dealing with directories almost entirely transparent.

How do I rename things?

Run:

```
git mv
```

Like `git rm` this both moves the file and records the move in Git, and you should commit afterwards. Note that version history is preserved across the rename. You can also rename whole directories full of files with a single `git mv`

How do I pull new changes?

Use `git pull`. You can specify a repository to pull from:

```
 git pull git@github.com:somerepo.git
```

If you do not, Git will pull from the repository you cloned from, if any. You can change this and add abbreviations for other repositories you commonly pull from (or push to). (See `git remote`)

How do I update my working files after pulling?

Updating working files happens automatically with a `git pull`. It's possible to pull changes and update working files in to separate steps, but `git pull` is generally easier.

How do I push out my changes after committing?

Use:

```
git push
```

this is exactly the same as `git pull` except that it sends commits in the other direction.

How do I create a tag?

```
 git tag mytagname
```

The commit tagged will be the version that your working tree is based on unless you supply options to name some other version. (And of course, uncommitted changes cannot be tagged.)

How do I make diffs?

Use:

```
git diff
```

Specify the files or directory trees you wish to compare; if you do not specify anything, by default the whole project is diffed.

By default, your working tree is diffed against the version in the project history to which it was last updated. You can diff against a specific version or tag by using the `git diff 0da94be 59ff30c` (the last two arguments are the beginnings of the hashes of the relevant commits, which you can see using `git log`).

How do I find out where a particular line of code appeared?

The `git annotate` command prints each line of the file with a prefix containing the hash of the commit in which the line appeared. This number can be fed to `git log` for more information.

How do I look at the project history?

With `git log`. By default, it prints the short summary for every version in the project. To get the complete commit messages (and also the complete list of filenames modified, which can sometimes be large) use `-v`. You can look at specific revisions by supplying the commit hash. You can look at the diff (patch) for each revision using `-p`.

How do I back out a bad commit?

It is late at night and you foolishly/accidentally commit some immensely stupid change that breaks everything. (We've all been there; if you have not yet, you will eventually.)

All is not lost. Part of the role of Git is to keep track of old versions; you can extract the old version and re-commit it. Suppose you can determine that version `aa4387471ea8` was the last "good" version of the code. You can return your code to that state using the following command:

```
git checkout aa4387471ea8
```

This updates your working files to the state associated with version `aa4387471ea8`. Now, `git commit` will update the history.

If you did something else stupid, like committing with the wrong commit message, and you have not pushed the resulting commits yet, you can go back to an earlier commit using

```
git reset 0d1d7fc32
```

Note that you can only do this once, and it cannot itself be undone. Additionally, you should not use reset to undo a commit that has already been pushed.

How do I move one of my repositories/clones?

Just move it. Nothing in a Git repository cares where the directory tree it lives in is.

For more information

To get a list of Git commands, you can type `git help`; you can get the options for each command with `git help <command>`, and there is also help available on certain other topics such as date and time strings. There are also man pages. The Git documentation can be dense, but there is plenty of help available on the web, and the staff is always willing to help!