Version Control—3 January 2015

Ben Klemens

[This is the revision control chapter of *21st Century C*[1], by me, published by O'Reilly Media. I had to sign over all rights to the book—three times over, for some reason I'm still not clear on. But I was clear throughout the contract negotiations of both first and second editions that I retain the right to publish my writing on this blog, and that I retain the movie rights. The great majority of the content in the book is available via the tip-a-day series from this post (entry #049) et seq, or the chapter-long post on parallell processing in C (entry #175).]

[The chapter on revision control gets especially positive reviews. One person even offered to translate it into Portugese; I had to refer him to O'Reilly and I don't know what happened after that. It's in the book because I think it'd be hard to be writing production C code in the present day without knowing how to pull code from a git repository. But in the other direction, this tutorial is not really C-specific at all.]

[So, here it is, with some revisions, in a free-as-in-beer format, to help those of you who are not yet habitual revision control users to become so. If you like this chapter, maybe let the book buying public know by saying something nice on Goodreads[2] or Amazon[3]. And if you think it'll make a good movie, give me a call.]

This chapter is about revision control systems (RCSes), which maintain snapshots of the many different versions of a project as it develops, such as the stages in the development of a book, a tortured love letter, or a program.

Using an RCS has changed how I work. To explain it with a metaphor, think of writing as rock climbing. If you're not a rock climber yourself, you might picture a solid rock wall and the intimidating and life-threatening task of getting to the top. But in the modern day, the process is much more incremental. Attached to a rope, you climb a few meters, and then clip the rope to the wall using specialized equipment (cams, pins, carabiners, and so on). Now, if you fall, your rope will catch at the last carabiner, which is reasonably safe. While on the wall, your focus is not reaching the top, but the much more reachable problem of finding where you can clip your next carabiner.

Coming back to writing with an RCS, a day's work is no longer a featureless slog toward the summit, but a sequence of small steps. What one feature could I add? What one problem could I fix? Once a step is made and you are sure that your code base is in a safe and clean state, commit a revision, and if your next step turns out disastrously, you

[1] `http://www.amazon.com/exec/obidos/redirect?link_code=ur2&camp=1789&tag=caltechdivini-20&creative=9325&path=tg/detail/-/1491903899/qid=1120157199/sr=8-1/ref=pd_bbs_ur_1`

[2] `https://www.goodreads.com/book/show/22800552-21st-century-c`

[3] `http://www.amazon.com/exec/obidos/redirect?link_code=ur2&camp=1789&tag=caltechdivini-20&creative=9325&path=tg/detail/-/1491903899/qid=1120157199/sr=8-1/ref=pd_bbs_ur_1`

can fall back to the revision you just committed instead of starting from the beginning.

But structuring the writing process and allowing us to mark safe points is just the beginning:

- Our filesystem now has a time dimension. We can query the RCS's repository of file information to see what a file looked like last week and how it changed from then to now. Even without the other powers, I have found that this alone makes me a more confident writer.
- We can keep track of multiple versions of a project, such as my copy and my coauthor's copy. Even within my own work, I may want one version of a project (a *branch*) with an experimental feature, which should be kept segregated from the stable version that needs to be able to run without surprises.
- GitHub[4] has about 218,000 projects that self-report as being primarily in C as of this writing, and there are more C projects in other, smaller RCS repository hosts, such as the GNU's Savannah. Even if you aren't going to modify the code, cloning these repositories is a quick way to get the program or library onto your hard drive for your own use. When your own project is ready for public use (or before then), you can make the repository public as another means of distribution.
- Now that you and I both have versions of the same project, and both have equal ability to hack our versions of the code base, revision control gives us the power to merge together our multiple threads as easily as possible.

This chapter will cover Git, which is a distributed revision control system, meaning that any given copy of the project works as a standalone repository of the project and its history. There are others, with Mercurial and Bazaar the other front-runners in the category. There is largely a one-to-one mapping among the features of these systems, and what major differences had existed have merged over the years, so you should be able to pick the others up immediately after reading this chapter.

1 Changes via diff

The most rudimentary means of revision control is via `diff` and `patch`, which are POSIX-standard and therefore most certainly on your system. You probably have two files on your drive somewhere that are reasonably similar; if not, grab any text file, change a few lines, and save the modified version with a new name. Try:

```
diff f1.c f2.c
```

and you will get a listing, a little more machine-readable than human-readable, that shows the lines that have changed between the two files. Piping output to a text file via `diff f1.c f2.c > diffs` and then opening `diffs` in your text editor may give you a colorized version that is easier to follow. You will see some lines giving the name of the file and location within the file, perhaps a few lines of context that did not change between the two files, and lines beginning with + and − showing the lines

[4] `http://github.com`

that got added and removed. Run `diff` with the `-u` flag to get a few lines of context around the additions and subtractions.

Given two directories holding two versions of your project, `v1` and `v2`, generate a single diff file in the unified diff format for the entire directories via the recursive (`-r`) option:

diff −ur v1 v > diff−v1v2

The `patch` command reads diff files and executes the changes listed there. If you and a friend both have `v1` of the project, you could send `diff-v1v2` to your friend, and she could run:

patch < diff−v1v2

to apply all of your changes to her copy of `v1`.

Or, if you have no friends, you can run `diff` from time to time on your own code and thus keep a record of the changes you have made over time. If you find that you have inserted a bug in your code, the diffs are the first place to look for hints about what you touched that you shouldn't have. If that isn't enough, and you already deleted `v1`, you could run the patch in reverse from the `v2` directory, `patch -R < diff-v1v2`, reverting version 2 back to version 1. If you were at version 4, you could even conceivably run a sequence of diffs to move further back in time:

cd v4
patch −R < diff−v3v4
patch −R < diff−v2v3
patch −R < diff−v1v2

I say *conceivably* because maintaining a sequence of diffs like this is tedious and error-prone. Thus, the revision control system, which will make and track the diffs for you.

2 Git's Objects

Git is a C program like any other, and is based on a small set of objects. The key object is the commit object, which is akin to a unified diff file. Given a previous commit object and some changes from that baseline, a new commit object encapsulates the information. It gets some support from the *index*, which is a list of the changes registered since the last commit object, the primary use of which will be in generating the next commit object.

The commit objects link together to form a tree much like any other tree. Each commit object will have (at least) one parent commit object. Stepping up and down the tree is akin to using `patch` and `patch -R` to step among versions.

The repository itself is not formally a single object in the Git source code, but I think of it as an object, because the usual operations one would define, such as new, copy, and free, apply to the entire repository. Get a new repository in the directory you are working in via:

git init

OK, you now have a revision control system in place. You might not see it, because Git stores all its files in a directory named `.git`, where the dot means that all the usual utilities like `ls` will take it to be hidden. You can look for it via, e.g., `ls -a` or via a show hidden files option in your favorite file manager.

Alternatively, copy a repository via `git clone`. This is how you would get a project from Savannah or Github. To get the source code for Git using `git`:

git clone https*://github.com/gitster/git.git*

The reader may also be interested in cloning the repository with the examples for this book:

git clone https*://github.com/b−k/21st−Century−Examples.git*

If you want to test something on a repository in `~/myrepo` and are worried that you might break something, go to a temp directory (say `mkdir ~/tmp; cd ~/tmp`), clone your repository with `git clone ~/myrepo`, and experiment away. Deleting the clone when done (`rm -rf ~/tmp/myrepo`) has no effect on the original.

Given that all the data about a repository is in the `.git` subdirectory of your project directory, the analog to freeing a repository is simple:

rm −rf .git

Having the whole repository so self-contained means that you can make spare copies to shunt between home and work, copy everything to a temp directory for a quick experiment, and so on, without much hassle.

We're almost ready to generate some commit objects, but because they summarize diffs since the starting point or a prior commit, we're going to have to have on hand some diffs to commit. The index (Git source: `struct index_state`) is a list of changes that are to be bundled into the next commit. It exists because we don't actually want every change in the project directory to be recorded. For example, `gnomes.c` and `gnomes.h` will beget `gnomes.o` and the executable `gnomes`. Your RCS should track `gnomes.c` and `gnomes.h` and let the others regenerate as needed. So the key operation with the index is adding elements to its list of changes. Use:

git add gnomes.c gnomes.h

to add these files to the index. Other typical changes to the list of files tracked also need to be recorded in the index:

git add newfile
git rm oldfile
git mv flie file

Changes you made to files that are already tracked by Git are not automatically added to the index, which might be a surprise to users of other RCSes (but see below). Add each individually via `git add changedfile`, or use:

git add −u

to add to the index changes to all the files Git already tracks.

At some point you have enough changes listed in the index that they should be recorded as a commit object in the repository. Generate a new commit object via:

git commit −a −m ”here is an initial commit.”

The `-m` flag attaches a message to the revision, which you'll read when you run `git log` later on. If you omit the message, then Git will start the text editor specified in the environment variable `EDITOR` so you can enter it (the default editor is typically vi; export that variable in your shell's startup script, e.g., `.bashrc` or `.zshrc`, if you want something different).

The `-a` flag tells Git that there are good odds that I forgot to run `git add -u`, so please run it just before committing. In practice, this means that you never have to run `git add -u` explicitly, as long as you always remember the `-a` flag in `git commit -a`.

A warning: It is easy to find Git experts who are concerned with generating a coherent, clean narrative from their commits. Instead of commit messages like “added an index object, plus some bug fixes along the way,” an expert Git author would create two commits, one with the message “added an index object” and one with “bug fixes.” These authors have such control because nothing is added to the index by default, so they can add only enough to express one precise change in the code, write the index to a commit object, then add a new set of items to a clean index to generate the next commit object. I found one blogger who took several pages to describe his commit routine: “For the most complicated cases, I will print out the diffs, read them over, and mark them up in six colors of highlighter...” However, until you become a Git expert, this will be much more control over the index than you really need or want. That is, not using `-a` with `git commit` is an advanced use that many people never bother with. In a perfect world, the `-a` would be the default, but it isn't, so don't forget it.

Calling `git commit -a` writes a new commit object to the repository based on all the changes the index was able to track, and clears the index. Having saved your work, you can now continue to add more. Further—and this is the real, major benefit of revision control so far—you can delete whatever you want, confident that it can be recovered if you need it back. Don't clutter up the code with large blocks of commented-out obsolete routines—delete!

A useful tip: After you commit, you will almost certainly slap your forehead and realize something you forgot. Instead of performing another commit, you can run `git commit --amend -a` to redo your last commit.

An aside: Diff/Snapshot Duality Physicists sometimes prefer to think of light as a wave and sometimes as a particle; similarly, a commit object is sometimes best thought of as a complete snapshot of the project at a moment in time and sometimes as a diff from its parent. From either perspective, it includes a record of the author, the name of the object (as we'll see later), the message you attached via the `-m` flag, and (unless it is the initial commit) a pointer to the parent commit object(s).

Internally, is a commit a diff or a snapshot? It could be either or both. There was once a time when Git always stored a snapshot, unless you ran `git gc` (garbage collect) to compress the set of snapshots into a set of deltas (aka diffs). Users complained about having to remember to run `git gc`, so it now runs automatically after certain commands, meaning that Git is probably (but by no means always) storing diffs. [end aside]

Having generated a commit object, your interactions with it will mostly consist of looking at its contents. You'll use `git diff` to see the diffs that are the core of the commit object and `git log` to see the metadata.

The key metadata is the name of the object, which is assigned via an unpleasant but sensible naming convention: the SHA1 hash, a 40-digit hexadecimal number that can be assigned to an object, in a manner that lets us assume that no two objects will have the same hash, and that the same object will have the same name in every copy of the repository. When you commit your files, you'll see the first few digits of the hash on the screen, and you can run `git log` to see the list of commit objects in the history of the current commit object, listed by their hash and the human-language message you wrote when you did the commit (and see `git help log` for the other available metadata). Fortunately, you need only as much of the hash as will uniquely identify your commit. So if you look at the log and decide that you want to check out revision number fe9c49cddac5150dc974de1f7248a1c5e3b33e89, you can do so with:

```
git checkout fe9c4
```

This does the sort of time-travel via diffs that `patch` almost provided, rewinding to the state of the project at commit `fe9c4`.

Because a given commit only has pointers to its parents, not its children, when you check `git log` after checking out an old commit, you will see the trace of objects that led up to this commit, but not later commits. The rarely used `git reflog` will show you the full list of commit objects the repository knows about, but the easier means of jumping back to the most current version of the project is via a *tag*, a human-friendly name that you won't have to look up in the log. Tags are maintained as separate objects in the repository and hold a pointer to a commit object being tagged. The most frequently used tag is `master`, which refers to the last commit object on the master branch (which, because we haven't covered branching yet, is probably the only branch you have). Thus, to return from back in time to the latest state, use:

```
git checkout master
```

Getting back to `git diff`, it shows what changes you have made since the last committed revision. The output is what would be written to the next commit object via `git commit -a`. As with the output from the plain `diff` program, `git diff > diffs` will write to a file that may be more legible in your colorized text editor.

Without arguments, `git diff` shows the diff between the index and what is in the project directory; if you haven't added anything to the index yet, this will be every change since the last commit. With one commit object name, `git diff` shows the sequence of changes between that commit and what is in the project directory. With two names, it shows the sequence of changes from one commit to the other:

```
git diff #Show the diffs between the working directory and the index.
git diff --staged #Show the diffs between the index and the previous commit.
git diff 234e2a #Show the diffs between the working directory and the given commit object.
git diff 234e2a 8b90ac #Show the changes from one commit object to another.
```

A useful tip: There are a few naming conveniences to save you some hexadecimal. The name `HEAD` refers to the last checked-out commit. This is usually the tip of a branch; when it isn't, git error messages will refer to this as a "detached `HEAD`."

Append `~1` to a name to refer to the named commit's parent, `~2` to refer to its grandparent, and so on. Thus, all of the following are valid:

git diff HEAD~4 #Compare the working directory to four commits ago.
git checkout master~1 #Check out the predecessor to the head of the master branch.
git checkout master~ #Shorthand **for** the same.
git diff b0897~ b8097 #See what changed in commit b8097.

At this point, you know how to:

- Save frequent incremental revisions of your project.
- Get a log of your committed revisions.
- Find out what you changed or added recently.
- Check out earlier versions so that you can recover earlier work if needed.

Having a backup system organized enough that you can delete code with confidence and recover as needed will already make you a better writer.

2.1 The Stash

Commit objects are the reference points from which most Git activity occurs. For example, Git prefers to apply patches relative to a commit, and you can jump to any commit, but if you jump away from a working directory that does not match a commit you have no way to jump back. When there are uncommitted changes in the current working directory, Git will warn you that you are not at a commit and will typically refuse to perform the operation you asked it to do. One way to go back to a commit would be to write down all the work you had done since the last commit, revert your project to the last commit, execute the operation, then redo the saved work after you are finished jumping or patching.

Thus we employ the *stash*, a special commit object mostly equivalent to what you would get from `git commit -a`, but with a few special features, such as retaining all the untracked junk in your working directory. Here is the typical procedure:

git stash # Code is now as it was at last checkin.
git checkout fe9c4

Look around here.

git checkout {\em \verb@master@} # Or whatever commit you had started with
Code is now as it was at last checkin, so replay stashed diffs with:
git stash pop

Another sometimes-appropriate alternative for checking out given changes in your working directory is `git reset --hard`, which takes the working directory back to the state it was in when you last checked out. The command sounds severe because it is: you are about to throw away all work you have done since the last checkout.

3 Trees and Their Branches

There is one tree in a repository, which got generated when the first author of a new repository ran `git init`. You are probably familiar with tree data structures, consisting of a set of nodes, where each node has links to some number of children and a link to a parent (and in exotic trees like Git's, possibly several parents).

Indeed, all commit objects but the initial one have a parent, and the object records the diffs between itself and the parent commit. The terminal node in the sequence, the tip of the branch, is tagged with a branch name. For our purposes, there is a one-to-one correspondence between branch tips and the series of diffs that led to that branch. The one-to-one correspondence means we can interchangeably refer to branches and the commit object at the tip of the branch. Thus, if the tip of the `master` branch is commit `234a3d`, then `git checkout master` and `git checkout 234a3d` are entirely equivalent (until a new commit gets written, and that takes on the `master` label). It also means that the list of commit objects on a branch can be rederived at any time by starting at the commit at the named tip and tracing back to the origin of the tree.

The typical custom is to keep the master branch fully functional at all times. When you want to add a new feature or try a new thread of inquiry, create a new branch for it. When the branch is fully functioning, you will be able to merge the new feature back into the master using the methods to follow.

There are two ways to create a new branch splitting off from the present state of your project:

```
git branch newleaf # Create a new branch...
git checkout newleaf # then check out the branch you just created.
    # Or execute both steps at once with the equivalent:
git checkout -b newleaf
```

Having created the new branch, switch between the tips of the two branches via `git checkout master` and `git checkout {\em newleaf}`.

What branch are you on right now? Find out with:

```
git branch
```

which will list all branches and put a * by the one that is currently active.

What would happen if you were to build a time machine, go back to before you were born, and kill your parents? If we learned anything from science fiction, it's that if we change history, the present doesn't change, but a new alternate history splinters off. So if you check out an old version, make changes, and check in a new commit object with your newly made changes, then you now have a new branch distinct from the master branch. You will find via `git branch` that when the past forks like this, you will be on `(no branch)`. Untagged branches tend to create problems, so if ever you find that you are doing work on `(no branch)`, then run `git branch -m new_branch_name` to name the branch to which you've just splintered.

Sidebar: Visual Aids There are several graphical interfaces to be had, which are especially useful when tracing how branches diverged and merged. Try `gitk` or

`git gui` for Tk-based GUIs, `tig` for a console (curses) based GUI, or `git instaweb` to start a web server that you can interact with in your browser, or ask your package manager or Internet search engine for several more.

3.1 Merging

So far, we have generated new commit objects by starting with a commit object as a starting point and applying a list of diffs from the index. A branch is also a series of diffs, so given an arbitrary commit object and a list of diffs from a branch, we should be able to create a new commit object in which the branch's diffs are applied to the existing commit object. This is a *merge*. To merge all the changes that occurred over the course of `newleaf` back into `master`, switch to `master` and use `git merge`:

```
git checkout master
git merge newleaf
```

For example, you have used a branch off of `master` to develop a new feature, and it finally passes all tests; then applying all the diffs from the development branch to `master` would create a new commit object with the new feature soundly in place.

Let us say that, while working on the new feature, you never checked out `master` and so made no changes to it. Then applying the sequence of diffs from the other branch would simply be a fast replay of all of the changes recorded in each commit object in the branch, which Git calls a *fast-forward*.

But if you made any changes to `master`, then this is no longer a simple question of a fast application of all of the diffs. For example, say that at the point where the branch split off, *gnomes.c* had:

```
short int height_inches;
```

In `master`, you removed the derogatory type:

```
int height_inches;
```

The purpose of `newleaf` was to convert to metric:

```
short int height_cm;
```

At this point, Git is stymied. Knowing how to combine these lines requires knowing what you as a human intended. Git's solution is to modify your text file to include both versions, something like:

```
<<<<<<< HEAD
int height_inches;
=======
short int height_cm;
>>>>>>> 3c3c3c
```

The merge is put on hold, waiting for you to edit the file to express the change you would like to see. In this case, you would probably reduce the five-line chunk Git left in the text file to:

```
int height_cm;
```

Here is the procedure for committing merges in a non-fast-forward, meaning that there have been changes in both branches since they diverged:

- Run `git merge other_branch`.
- In all likelihood, get told that there are conflicts you have to resolve.
- Check the list of unmerged files using `git status`.
- Pick a file to manually check on. Open it in a text editor and find the merge-me marks if it is a content conflict. If it's a filename or file position conflict, move the file into place.
- Run `git add your_now_fixed_file`.
- Repeat steps 3–5 until all unmerged files are checked in.
- Run `git commit` to finalize the merge.

Take comfort in all this manual work. Git is conservative in merging and won't automatically do anything that could, under some storyline, cause you to lose work.

When you are done with the merge, all of the relevant diffs that occurred in the side branch are represented in the final commit object of the merged-to branch, so the custom is to delete the side branch:

```
git delete other_branch
```

The `other_branch` tag is deleted, but the commit objects that led up to it are still in the repository for your reference.

4 The Rebase

Say you have a main branch and split off a testing branch from it on Monday. Then on Tuesday through Thursday, you make extensive changes to both the main and testing branch. On Friday, when you try to merge the test branch back into the main, you have an overwhelming number of little conflicts to resolve.

Let's start the week over. You split the testing branch off from the main branch on Monday, meaning that the last commits on both branches share a common ancestor of Monday's commit on the main branch. On Tuesday, you have a new commit on the main branch; let it be commit `abcd123`. At the end of the day, you replay all the diffs that occurred on the main branch onto the testing branch:

```
git branch testing # get on the testing branch
git rebase abcd123 # or equivalently: git rebase main
```

With the `rebase` command, all the changes made on the main branch since the common ancestor are replayed on the testing branch. You might need to manually merge things, but by only having one day's work to merge, we can hope that the task of merging is more manageable.

Now that all changes up to `abcd123` are present in both branches, it is as if the branches had actually split off from that commit, rather than Monday's commit. This is where the name of the procedure comes from: the testing branch has been rebased to split off from a new point on the main branch.

You also perform rebases at the end of Wednesday, Thursday, and Friday, and each of them is reasonably painless, as the testing branch kept up with the changes on the main branch throughout the week.

Rebases are often cast as an advanced use of Git, because other systems that aren't as capable with diff application don't have this technique. But in practice rebasing and merging are about on equal footing: both apply diffs from another branch to produce a commit, and the only question is whether you are tying together the ends of two branches (in which case, merge) or want both branches to continue their separate lives for a while longer (in which case, rebase). The typical usage is to rebase the diffs from the master into the side branch, and merge the diffs from the side branch into the master, so there is a symmetry between the two in practice. And as noted, letting diffs pile up on multiple branches can make the final merge a pain, so it is good form to rebase reasonably often.

5 Remote Repositories

Everything to this point has been occurring within one tree. If you cloned a repository from elsewhere, then at the moment of cloning, you and the origin both have identical trees with identical commit objects. However, you and your colleagues will continue working, so you will all be adding new and different commit objects.

Your repository has a list of *remotes*, which are pointers to other repositories related to this one elsewhere in the world. If you got your repository via `git clone`, then the repository from which you cloned is named `origin` as far as the new repository is concerned. In the typical case, this is the only remote you will ever use.

When you first clone and run `git branch`, you'll see one lonely branch, regardless of how many branches the origin repository had. But run `git branch -a` to see all the branches that Git knows about, and you will see those in the remote as well as the local ones. If you cloned a repository from Github, et al, you can use this to check whether other authors had pushed other branches to the central repository.

Those copies of the branches in your local repository are as of the first time you pulled. Next week, to update those remote branches with the information from the origin repository, run `git fetch`.

Now that you have up-to-date copies of the remote branches in your repository, you could merge one with the local branch you are working on using the full name of the remote branch, for example, `git merge remotes/origin/master`.

Instead of the two-step `git fetch; git merge remotes/origin/master`, you can update the branch via

```
git pull origin master
```

which fetches the remote changes and merges them into your current repository all at once.

The converse is `push`, which you'll use to update the remote repository with your last commit (not the state of your index or working directory). If you are working on a branch named `bbranch` and want to push to the remote with the same name, use:

```
git push origin bbranch
```

There are good odds that when you push your changes, applying the diffs from your branch to the remote branch will not be a fast-forward (if it is, then your colleagues haven't been doing any work). Resolving a non-fast-forward merge typically requires human intervention, and there is probably not a human at the remote. Thus, Git will allow only fast-forward pushes. How can you guarantee that your push is a fast-forward?

- Run `git pull origin bbranch` to get the changes made since your last pull.
- Merge as seen earlier, wherein you as a human resolve those changes a computer cannot.
- Run `git commit -a -m "dealt with merges"`.
- Run `git push origin bbranch`, because now Git only has to apply a single diff, which can be done automatically.

To this point, I have assumed that you are on a local branch with the same name as the remote branch (probably `master` on both sides). If you are crossing names, give a colon-separated pair of `source:destination` branch names.

git fetch origin new_changes:master #Merge remote new_changes into local master
git push origin my_fixes:version2 #Merge the local branch into a differently named remote.
git push origin :prune_me #Delete a remote branch.
git fetch origin new_changes: #Pull to no branch; create a commit named FETCH_HEAD.

None of these operations change your current branch, but some create a new branch that you can switch to via the usual `git checkout`.

Sidebar: The Central Repository Despite all the discussion of decentralization, the easiest setup for sharing is still to have a central repository that everybody clones, meaning that everybody has the same origin repository. This is how downloading from Github and Savannah typically works. When setting up a repository for this sort of thing, use `git init --bare`, which means that nobody can actually do work in that directory, and users will have to clone to do anything at all. There are also some permissions flags that come in handy, such as `--shared=group` to allow all members of a POSIX group to read and write to the repository.

You can't push to a branch in a nonbare remote repository that the repository owner has checked out; doing so will cause chaos. If this happens, ask your colleague to `git branch` to a different branch, then push while the target branch is in the background.

Or, your colleague can set up a public bare repository and a private working repository. You push to the public repository, and your colleague pulls the changes to his or her working repository when convenient. [end sidebar]

The structure of a Git repository is not especially complex: there are commit objects representing the changes since the parent commit object, organized into a tree, with an index gathering together the changes to be made in the next commit. But with these elements, you can organize multiple versions of your work, confidently delete things, create experimental branches and merge them back to the main thread when

they pass all their tests, and merge your colleagues' work with your own. From there, `git help` and your favorite Internet search engine will teach you many more tricks and ways to do these things more smoothly.