The system Git config file is found in the mingw32\etc folder of the Git installation. The global Git configuration file is found in the root of the user's local profile or home directory (C:\Users\git-user\). The local Git config file is stored inside the .git directory of the repository in which you are working.

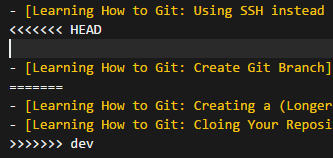
--------------------------------------------------------------------------------------

GIT RM?

If you just use rm, you will need to follow it up with git add <fileRemoved>. git rm does this in one step.

You can also use git rm --cached which will remove the file from the index (staging it for deletion on the next commit), but keep your copy in the local file system. -------------------------------------------------------------------------------- merge conflict:

Sometimes, when we try to merge two branches we are going to face a git conflict. Conflict is the state where certain line(s) in a certain file have different changes in the commits in different branch and git can’t decide which one is the right changes (also if both changes are correct and used). So, we are going to check the file that has the conflicted copy which in this case README.md.



The conflicted file

Most probably your conflicted file(s) will have marks such as the above picture. The marks divide the file state between the destination branch and the branch that being merged to. In this case, the HEAD one will be the destination branch which is the master branch, where the other part will be the dev branch as indicated at the end of the mark.

To resolve the conflict, all we need to do is to edit as we need it. So, the one that resolving the conflict should know which line of code to keep and which line of code to discard. In my example, we will take all of the code (nothing discarded).

After we’ve done with resolving the conflict, run a git commit once and we can commit the merge process as what we called the merge commit.

Creating a merge commit

We’re done! If then we check the log of the git .

**REBASE VS MERGE**

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

Now, let’s say that the new commits in master are relevant to the feature that you’re working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

The Merge Option

The easiest option is to merge the master branch into the feature branch using something like the following:

git checkout feature

git merge master

Or, you can condense this to a one-liner:

git merge feature master

This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:

Merging is nice because it’s a *non-destructive* operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

The Rebase Option

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

git checkout feature

git rebase master

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing *re-writes*the project history by creating brand new commits for each commit in the original branch.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.

But, there are two trade-offs for this pristine commit history: safety and traceability. If you don’t follow the [Golden Rule of Rebasing](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing), re-writing project history can be potentially catastrophic for your collaboration workflow. And, less importantly, rebasing loses the context provided by a merge commit—you can’t see when upstream changes were incorporated into the feature.

Hard and soft reset

When you modify a file in your repository, the change is initially unstaged. In order to commit it, you must stage it—that is, add it to the index—using git add. When you make a commit, the changes that are committed are those that have been added to the index.

git reset changes, at minimum, where the current branch (HEAD) is pointing. The difference between --mixed and --soft is whether or not your index is also modified. So, if we're on branch master with this series of commits:

- A - B - C (master)

HEADpoints to C and the index matches C.

When we run git reset --soft B, master (and thus HEAD) now points to B, but the index still has the changes from C; git status will show them as staged. So if we run git commit at this point, we'll get a new commit with the same changes as C.

Okay, so starting from here again:

- A - B - C (master)

Now let's do git reset --mixed B. (Note: --mixed is the default option). Once again, master and HEAD point to B, but this time the index is also modified to match B. If we run git commit at this point, nothing will happen since the index matches HEAD. We still have the changes in the working directory, but since they're not in the index, git status shows them as unstaged. To commit them, you would git add and then commit as usual.

And finally, --hard is the same as --mixed (it changes your HEAD and index), except that --hardalso modifies your working directory. If we're at C and run git reset --hard B, then the changes added in C, as well as any uncommitted changes you have, will be removed, and the files in your working copy will match commit B. Since you can permanently lose changes this way, you should always run git status before doing a hard reset to make sure your working directory is clean or that you're okay with losing your uncommitted changes.

Important:

git revert --no-commit 0766c053..HEAD

git commit

This will revert everything from the HEAD back to the commit hash, meaning it will recreate that commit state in the working tree as if every commit since had been walked back. You can then commit the current tree, and it will create a brand new commit essentially equivalent to the commit you "reverted" to.

* git revert is used to undo a previous commit. In git, you can't alter or erase an earlier commit. (Actually you can, but it can cause problems.) So instead of editing the earlier commit, revert introduces a new commit that reverses an earlier one.
* git reset is used to undo changes in your working directory that haven't been comitted yet.
* git checkout is used to copy a file from some other commit to your current working tree. It doesn't automatically commit the file.

**Reset -** On the commit-level, resetting is a way to move the tip of a branch to a different commit. This can be used to remove commits from the current branch.

**Revert -** Reverting undoes a commit by creating a new commit. This is a safe way to undo changes, as it has no chance of re-writing the commit history. Contrast this with git reset, which does alter the existing commit history. For this reason, git revert should be used to undo changes on a public branch, and git reset should be reserved for undoing changes on a private branch.