

[theserverside.com](https://www.theserverside.com)

How to revert a git commit: A simple undo changes example

By Cameron McKenzie, TechTarget Published: 21 Jun 2018

6-8 minutes

[Problem solve](#) Get help with specific problems with your technologies, process and projects.

The best way to undo previous changes to the source code repository is to learn how to revert a git commit.

-
-

Two of my favorite source code control commands for working with previous commits are git cherry-pick and git revert. The [git cherry-](#)

[pick](#) command is great when you need to pull in a bug fix from [another active branch](#). And the ability to revert a Git commit is great way to back out of a bad contribution to the code base. But as useful as these commands are, I rarely see devs practice them on active projects. Team leads often go to exorbitant lengths to undo a problem in their source code repository. Really, they could just git revert a troublesome commit instead. Here, we'll provide a simple git revert example to show how the command works and how you can use it to revert a specific commit.

How to revert a git commit

In the name of simplicity, this git revert example will start off with a completely clean repository. That means issuing a quick [git init](#) command.

```
/c/ git revert changes example
$ git init
Initialized empty Git repository in C:/ git revert
changes example/.git/
```

Now that we've initialized the [Git](#) repository, create five HTML files, and perform a git commit with each one. To keep track of the commit history, each commit will include the commit number along

with the count of the number of files in the working tree. Now, we can git revert a previous commit and examine the state in which it leaves our [development environments](#).

```
/c/ git revert changes example
$ echo 'alice' > alpha.html
$ git add . && git commit -m "1st git commit: 1
file"
$ echo 'becky' > beta.html
$ git add . && git commit -m "2nd git commit: 2
files"

$ echo 'callie' > charlie.html
$ git add . && git commit -m "3rd git commit: 3
files"

$ echo 'diana' > delta.html
$ git add . && git commit -m "4th git commit: 4
files"

$ echo 'ellen' > edison.html
$ git add . && git commit -m "5th git commit: 5
files"
```

With this git revert example, you'll learn how to undo a previous

source code repository update.

To recap this git revert example, we have created five HTML files and executed a commit for each one. We can [look at the history](#) if we invoke the git reflog command:

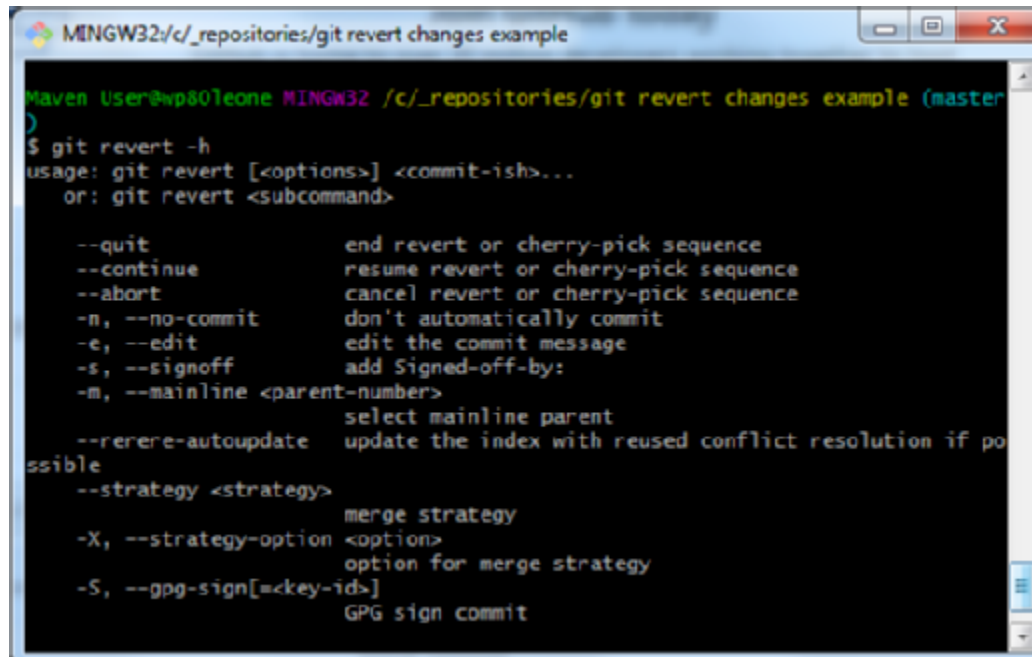
```
/c/ git revert changes example
$ git reflog
(HEAD -> master)
d846aa8 HEAD@{0}: commit: 5th git commit: 5 files
0c59891 HEAD@{1}: commit: 4th git commit: 4 files
4945db2 HEAD@{2}: commit: 3rd git commit: 3 files
defc4eb HEAD@{3}: commit: 2nd git commit: 2 files
2938ee3 HEAD@{4}: commit: 1st git commit: 1 file
```

What happens when we git revert a commit?

At this point, we have five files in the working directory. So, what exactly happens when we git revert a commit? If we performed a git revert on the third git commit -- the one with *charlie.html* -- in what type of state would git leave our development environment? When developers git revert a commit for the first time, they usually guess the command will do one of three things:

1. The git revert will leave two files on the file system -- *alpha.html* and

- beta.html* -- and will roll back to the state prior to the third commit.
2. The git revert will leave three files on the file system -- *alpha.html*, *beta.html* and *charlie.html* -- and will roll back to the state at which the third commit occurred.
 3. The git revert will leave us with four files and remove only *charlie.html*, the only file associated exclusively with the third commit.



```
MINGW32:/c/_repositories/git revert changes example
Maven User@wp80leone MINGW32 /c/_repositories/git revert changes example (master)
$ git revert -h
usage: git revert [<options>] <commit-ish>...
       or: git revert <subcommand>

    --quit                end revert or cherry-pick sequence
    --continue            resume revert or cherry-pick sequence
    --abort               cancel revert or cherry-pick sequence
    -n, --no-commit       don't automatically commit
    -e, --edit             edit the commit message
    -s, --signoff         add Signed-off-by:
    -m, --mainline <parent-number>
                          select mainline parent
    --rerere-autoupdate   update the index with reused conflict resolution if possible
    --strategy <strategy>
                          merge strategy
    -X, --strategy-option <option>
                          option for merge strategy
    -S, --pgp-sign[=<key-id>]
                          GPG sign commit
```

Here's the command you'll use to git revert a commit.

So, which is it? We can figure it out if we issue the git revert on the third commit.

```
/c/ git revert changes example  
$ git revert 4945db2
```

A simple directory listing shows us that four files remain, with the file named *charlie.html* removed.

```
/c/ git revert changes example  
$ ls  
alpha.html  beta.html  delta.html  edison.html
```

How to git revert a previous commit

As you can see from this git revert example, when you git revert a previous commit, the command only removes the changes associated with that previous commit. [Prior changes](#) and those made after that previous commit remain. When you git revert a previous commit, the only things plucked out of your development environment are the changes explicitly associated with the reverted commit. In this case, the command only removes the third commit we issued on the source code repository. Again, only that third commit falls victim to the git revert command, as it removes *charlie.html* from the working tree.

And that's it. That's what happens when you git revert a commit. It's a great way to remove a bug that may have been introduced into

the Version control system at some point in the past or back out of a feature enhancement that wasn't well-received by the client. It's definitely a [feature of Git](#) that can save the [DevOps team](#) a lot of time when they need to roll back a small, historic change without wiping out all of the good changes and updates they have subsequently committed.