

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

# Learn to git cherry-pick a commit with this easy example

*Cameron McKenzie TechTarget 17 May 2018*

7-8 minutes

---

One of the most commonly misunderstood Version control commands is *git cherry-pick*, and that's a real shame because the ability to *git cherry-pick* a commit is one of the most useful skills a developer can employ when trying to isolate a software bug or fix a broken build.

## What is *git cherry-pick*?

According to the official [git](#) documentation, the goal of a cherry-pick is to “*apply the changes introduced by some existing commit.*”

Essentially, a cherry-pick will look at a previous commit in the repository's history and apply the changes that were part of that

earlier commit [to the current working tree](#). The definition is seemingly straight forward, yet in practice there is a great deal of confusion over what exactly happens when someone tries to *git cherry-pick* a commit, or even cherry-pick from another branch. This *git cherry-pick* example will eliminate that confusion.

## **A *git cherry-pick* example**

We will begin this *git cherry-pick* example with a completely clean [repository](#), which means first you will create a new folder, which we will name *git cherry-pick tutorial*, and then issuing a *git init* command [from within it](#).

```
/c/ git cherry-pick tutorial
$ git init
Initialized empty Git repository in C:/ git
cherry-picktutorial/.git/
```

With the [git repository](#) initialized, create five html files, and each time a you create a file, perform a commit. In each commit message, the commit number and the number of files in the working tree will be included as part of the commit message. In a subsequent step, where we cherry-pick a commit, it is one of these commit steps that will be chosen.

Here are the commands to create the five, alphabetically ordered .html files along with the git commands required to add each file independently to the git index and subsequently commit those files to the repository:

```
/c/ git cherry-pick tutorial
$ echo 'alpha' > alpha.html
$ git add . | git commit -m "1st commit: 1 file"

$ echo 'beta' > beta.html
$ git add . | git commit -m "2nd commit: 2 files"

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

$ echo 'whip it' > devo.html
$ git add . | git commit -m "4th commit: 4 files"

$ echo 'Big Lebowski' > eagles.html
$ git add . | git commit -m "5th commit: 5 files"
```

### **A note on the echo command**

The echo command will be used as a file creation shortcut. For those unfamiliar with this shortcut, echoing text and then specifying a file name after the greater-than sign will create a new file with that name, containing the text that was echoed. So the following command will create a new file named *agenda.html* with the word *devops* contained within it:

```
$ echo 'devops' > agenda.html
```

## Delete and commit before the cherry-pick

We will now delete all five recently created files to clear out the working directory. A subsequent commit will be needed here as well.

```
/c/ git cherry-pick tutorial
$ rm *.html
$ git add . | git commit -m "all deleted: 0 files"
```

To recap, five files were created, and each file created has a corresponding commit. And then all the files were deleted and a sixth commit was issued. A chronicling of this commit history can be concisely viewed in the reflog. Take special note of the hexadecimal identifier on the third commit, which we will cherry-pick in a moment:

```
/c/ git cherry-pick tutorial
$ git reflog
189aa32 HEAD@{0}: commit: all deleted: 0 files
e6f1ac7 HEAD@{1}: commit: 5th commit: 5 files
2792e62 HEAD@{2}: commit: 4th commit: 4 files
60699ba HEAD@{3}: commit: 3rd commit: 3 files
4ece4c7 HEAD@{4}: commit: 2nd commit: 2 files
cc6b274 HEAD@{5}: commit: 1st commit: 1 file
```

## What happens when we *git cherry-pick* a commit?

This is where the *git cherry-pick* example starts to get interesting. We need to *git cherry-pick* a commit, so let's choose the third commit where the file named charlie.html was created. But before we do, ask yourself what you believe will happen after the command is issued. When we *git cherry-pick* a commit, will we get all of the files associated with that commit, which would mean alpha.html, beta.html and charlie.html will come back? Or will we get just one file back? Or will the attempt to *git cherry-pick* a commit fail since all of the files associated with the commit have been deleted from our workspace?

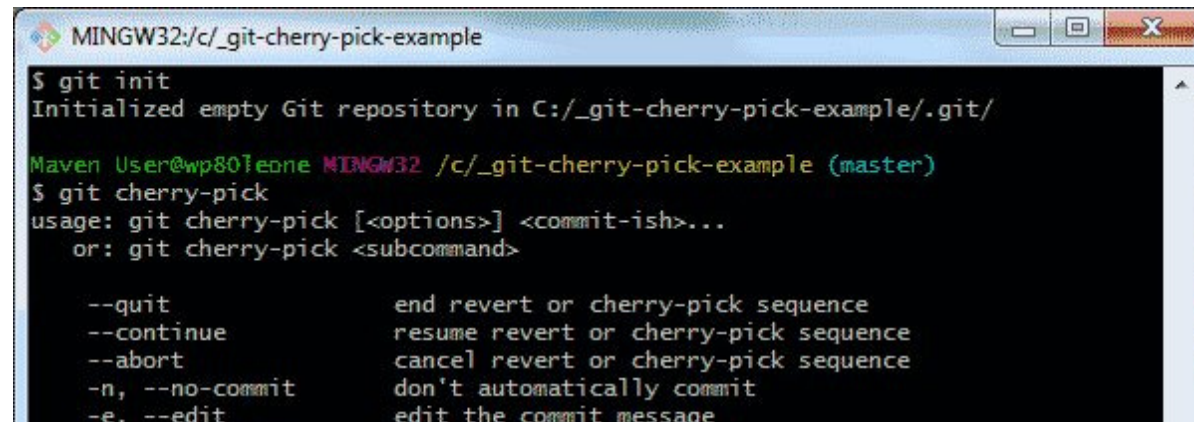
## The git cherry-pick command

Here is the command to *git cherry-pick* commit number 60699ba:

```
/c/ git cherry-pick tutorial (master)
$ git cherry-pick 60699ba
```

```
[master eba7975] 3rd commit: 3 files
1 file changed, 1 insertion(+)
create mode 100644 charlie.html
```

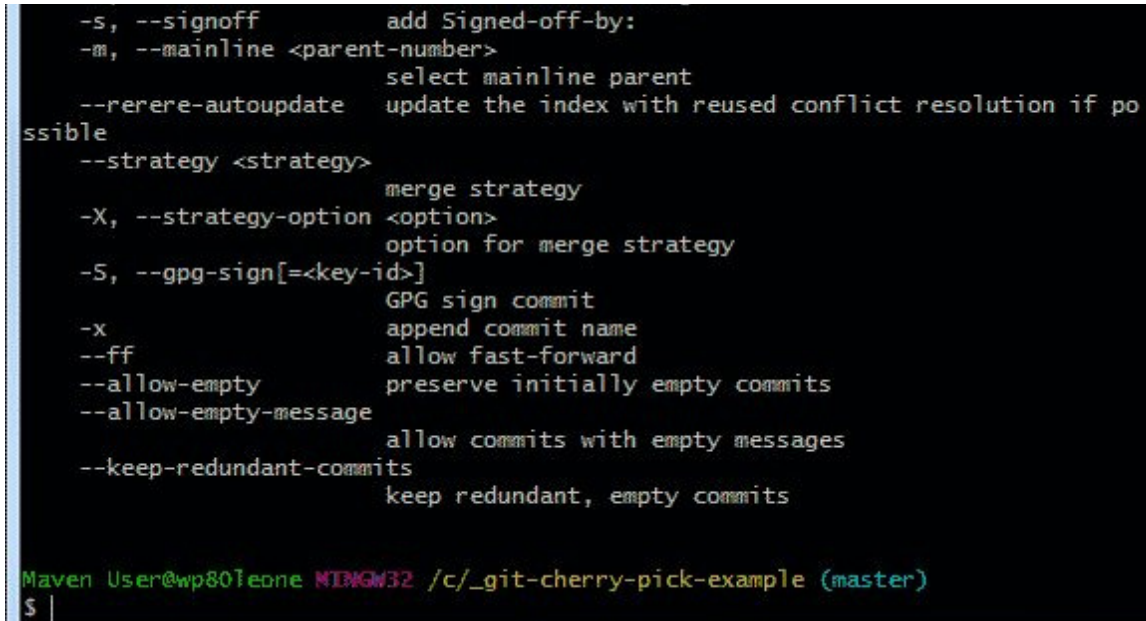
As you can see, only one file was added to the working directory, namely charlie.html. The files that were added to the repository in prior commits were not added, which tends to be the expectation of many users. Many users believe that when you *git cherry-pick* a commit, all of the files that are part of that branch, at that time of the commit, will be brought into the working directory. This is obviously not the case. When you *git cherry-pick* a commit, only the change associated with that commit is re-applied to the working tree.



```
MINGW32:/c/_git-cherry-pick-example
$ git init
Initialized empty Git repository in C:/_git-cherry-pick-example/.git/

Maven User@wp801edne MINGW32 /c/_git-cherry-pick-example (master)
$ git cherry-pick
usage: git cherry-pick [<options>] <commit-ish>...
       or: git cherry-pick <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
```

A screenshot of a terminal window with a black background and white text. It lists various options for the 'git cherry-pick' command, such as '--signoff', '--mainline', '--strategy', and '--allow-empty'. At the bottom, the terminal shows the user 'Maven User' at a prompt in a directory named '/c/\_git-cherry-pick-example' on the 'master' branch.

```
-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
-x                append commit name
--ff              allow fast-forward
--allow-empty     preserve initially empty commits
--allow-empty-message
                    allow commits with empty messages
--keep-redundant-commits
                    keep redundant, empty commits

Maven User@wp801edne MINGW32 /c/_git-cherry-pick-example (master)
$ |
```

Here are the various options available when issuing a git cherry-pick for a commit.

### A hidden *git cherry-pick* commit

It should also be noted that it's not just the working tree that is updated. When you *git-cherry-pick* a commit, a completely new commit is created on the branch, as the following reflog command indicates:

```
/c/ git cherry-pick tutorial (master)
$ git reflog
eba7975 HEAD@{0}: cherry-pick: 3rd commit: 3 files
```

```
189aa32 HEAD@{1}: commit: all deleted: 0 files
e6f1ac7 HEAD@{2}: commit: 5th commit: 5 files
2792e62 HEAD@{3}: commit: 4th commit: 4 files
60699ba HEAD@{4}: commit: 3rd commit: 3 files
4ece4c7 HEAD@{5}: commit: 2nd commit: 2 files
cc6b274 HEAD@{6}: commit (initial): 1st commit: 1
file
```

When a developer encounters a problem in the repository, the ability to *git cherry-pick* a commit can be extremely helpful in fixing bugs and resolving [problems in GitHub](#), which is why understanding how the command works and the impact it will have on the current development branch is pivotal. Hopefully, with this *git cherry-pick* example under your belt, you will have the confidence needed to use the command to resolve problems in your own development environment.

---

## Become a Git power user

Want to become a Git power user? Take a look at the following Git articles and tutorials

- How to do a [Git clean up](#) of branches and commits



- Learn to [rebase onto master](#) and [rebase from the master](#) branch
- [Squash all Git commits](#) on a branch down to one
- [Shelve](#) your changes with [Git stash pop and apply](#)
- Easily explain the [Git vs GitHub difference](#)
- Add a [shallow git clone](#) of [depth 1](#) do your Jenkins jobs
- Set up a [local Git server](#) with a [bare Git repo](#)