

---

**gist** / ˈdʒɪst/ *n* [Lat. *gistis*, It. *gista* to lie, ultim. It. *jacere* — more at ADJACENT] (ca. 1711) **1** : the ground of a legal action **2** : the main point or part : ESSENCE <the ~ of an argument>  
**1** **git** /'ɡɪt/ *n* [var. of *get*, term of abuse, fr. <sup>2</sup>*get*] (1929) *Brit* : a foolish or worthless person  
**2** **git** *dial* var of GET  
**git-go** var of GET-GO  
**git·tern** /'ɡɪ-tərn/ *n* [ME *giterne*, fr. MF *guiterne*, modif. of OSp *guitarra* guitar] (14c) : a medieval guitar

---

# An (maybe not so) gentle introduction to git

Luca Formaggia

2017 V2

# What is a revision control system (RCS)

Software needs to be **maintained**, **debugged**, **improved**... while keeping track of the changes and being able to recover old revisions if needed.

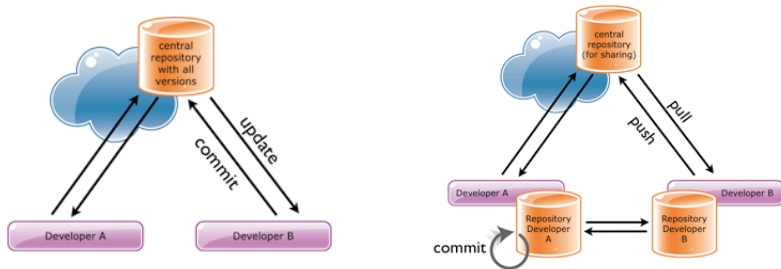
Those task can be done automatically using a **Revision Control System**.

In addition, one may want to keep a copy of the code (and of all its changes) in a remote repository, for safety and for sharing with others.

Most common RCS: CVS, RCS, git, mercurial (Hg)....

---

# Centralised vs Distributed RCS



Centralised RCS compares with a distributed RCS as an **hub** compares with a **peer-to-peer** model.

# The philosophy of a distributed RCS like git

---

- ▶ Each user has a **local copy of the repository**: you have full control of the history of the changes even if you are not connected to the web;
- ▶ **Most commands operate on the local copy (minimize communications)**;
- ▶ You may have multiple remote repositories: better distributed development.

## The (almost\_ only git commands that communicate with a remote repository

---

Even if you do not yet know what they do, here there are the main three commands used by git to operate on a remote repository:

push, fetch, pull

All other commands operate only on the local repository, which resides on your PC!

# First thing to do

---

If you have never used git on a computer you should first let git know who you are (so that people can blame you for your wrong deeds or cherish you for your programming skill!)

```
$> git config --global user.name "Micky Mouse"
```

```
$> git config --global user.email  
"micky.mouse@gmail.com"
```

git config tells git to change some git configuration. global makes it to change some global parameter, i.e. a parameter that applies to all yours git working directories in that computer.

---

## Other useful global setting

---

If you want to set for git a different editor than the system default one

```
$> git config --global core.editor emacs
```

This command will set the default editor for git to emacs.

# Need help?

---

Git has an integrated help. If you need help for the command command just do

```
$> git help command
```

Git uses a lot of terms that may confuse you at the beginning (and not only at the beginning...). A useful command is

```
$> git help glossary
```

You find more on the [Git Book \(git-scm.com/book\)](https://git-scm.com/book)



# More help?

---

Other useful commands

```
$> git help tutorial
```

A tutorial

```
$> git help tutorial-2
```

The second part of the tutorial.

---

Git is a very complete (and complex) revision system ,  
but in fact you will normally need just a few commands.

# Useful stuff

---

On the BEEP site and on Piazza you have some useful material, among which

- ▶ DZone.git.pdf. Technically a cheat sheet, but in fact a short summary of git, very nice, by [refcards.com](#).
- ▶ The funny, but also plenty of advice, zine by Katie Saylor-Miller and Julia Evans, "[Oh shit, git! Recipes for getting out of a git mess](#)". Not for the absolute novice, but rather clear. If you like it and you wish to make Katie and Julia happy go to their site [ohshitgit.com](#) and donate a few dollars (the site is available also translated in Italian). The language is very very colloquial, if you prefer less swarful language go to the edulcorated version in [dangitgit.com/en](#).

# Create/clone a repo

---

You can create your repo locally by creating a new directory, go in the directory and type

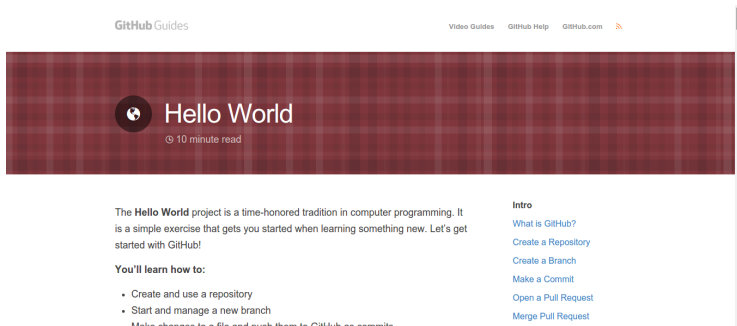
```
$> git init
```

Or you can clone from an existing remote repository

```
$> git clone RometeRepoAddress
```

# Github

**Github** is a web site that allows you to create repositories and share them with others. It provides also a nice web interface and the possibility of forking from other repositories.



The screenshot shows the GitHub Guides interface. At the top, there's a navigation bar with 'GitHub Guides' on the left and links for 'Video Guides', 'GitHub Help', and 'GitHub.com' on the right. Below this is a large red banner with a white GitHub logo and the text 'Hello World' and '10 minute read'. The main content area has a dark background. On the left, it describes the 'Hello World' project as a time-honored tradition in computer programming. On the right, there's a sidebar with the title 'Intro' and a list of links: 'What is GitHub?', 'Create a Repository', 'Create a Branch', 'Make a Commit', 'Open a Pull Request', and 'Merge Pull Request'.

GitHub Guides

Video Guides GitHub Help GitHub.com

## Hello World

10 minute read

The **Hello World** project is a time-honored tradition in computer programming. It is a simple exercise that gets you started when learning something new. Let's get started with GitHub!

**You'll learn how to:**

- Create and use a repository
- Start and manage a new branch
- Make changes to a file and push them to GitHub as commits

**Intro**

- [What is GitHub?](#)
- [Create a Repository](#)
- [Create a Branch](#)
- [Make a Commit](#)
- [Open a Pull Request](#)
- [Merge Pull Request](#)

# Bitbucket

Bitbucket is another git based site that allows you to create repositories and share them with others and have a nice user interface. It supports also private projects (github allow them only in the professional version or if you register with polimi email).

The screenshot shows the Bitbucket web interface. At the top is a dark blue navigation bar with the Bitbucket logo, menu items for Teams, Projects, Repositories, and Snippets, a search bar labeled 'Find a repository...', and a user profile icon. Below this is a light gray 'Dashboard' header with tabs for Overview, Pull requests, Issues, and Snippets. The main content area features a 'Your dashboard, improved' announcement with a laptop icon, explaining re-organization and new filters, with a 'Create a team' button and a link to 'Learn more about projects'. Below the announcement is a filter section with 'FILTER BY:' and dropdowns for 'Owner', 'Project', and 'Watching', alongside a 'Find repositories' search box. A table lists repositories with columns for Repository, Project, and Owner. One repository is shown: 'oseen-stabilization-2013' under the owner 'Andre Massing'. To the right, there are two sections: 'Latest updates' with a link to 'Improved pull request list' and 'Build status' with a link to 'How to integrate'. At the bottom right, a 'Recent activity' section is partially visible.

Bitbucket Teams Projects Repositories Snippets Find a repository...

## Dashboard

Overview Pull requests Issues Snippets

Your dashboard, improved

We've re-organized your dashboard and added smarter filters so you can quickly get to the repositories that matter to you. We've also introduced projects for Bitbucket teams, making it easier to organize repositories. [Learn more about projects](#)

[Create a team](#) [Ok, got it](#)

FILTER BY: Owner Project Watching Find repositories

Repository	Project	Owner
oseen-stabilization-2013		Andre Massing

**Latest updates**

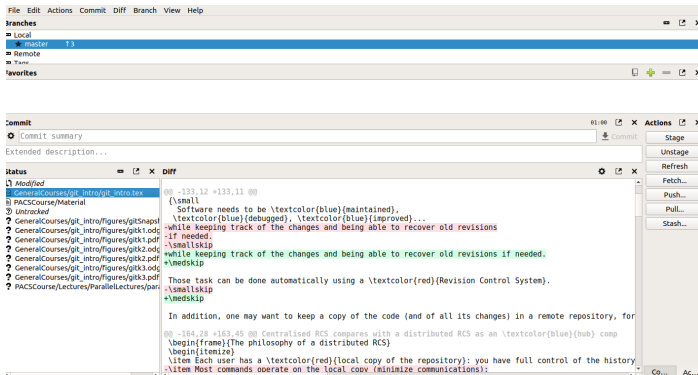
[Improved pull request list](#)  
Try a new pull requests page with improved filtering (status, target branch, ...) and a new look and feel. Enable the beta today!  
[Give feedback](#)

[Build status](#)  
Want to see your CI results directly in Bitbucket? This feature provides a build status on commits, branches, and pull requests, indicating when it's safe to merge changes.  
[How to integrate](#)

**Recent activity**

# Graphical Interfaces

There are plenty of graphical interfaces for git. I sometimes use **git-cola**, but you may find many others on the web. A list may be found **here**.



The directory where you have created the repository and all its subdirectories and files is called the **working area** or **working tree**.

How do you know that a directory is the root of a git repo? Type

```
$> ls -a
```

And you will find the **hidden** directory **.git**. **DO NOT TRY TO REMOVE IT!**.

# What should git keep track of?

---

You will put in the git repo only the main files (sources, headers...), i.e. files that are not created from other files.

you do not put in the repo object files, executables, temporary files etc. etc., i.e. files that are in fact the product of a certain process.

You will see that there is a way to tell git to completely ignore certain files in your working area, so you do not risk of committing them by mistakes.



# A test repo

---

For this lecture I will create a test repo, so that you can experience the use of git directly.

# Create a new file

---

Edit a new file called `file1.cpp`  
show status of the repo:

```
$> git status
```

The output is:

On branch master  
Your branch is up to date with 'origin/master'.

Untracked files:  
(use "git add <file>..." to include in what will be committed)  
`file1.cpp`

nothing added to commit but untracked files present  
(use "git add" to track)

# Add a new file to the git history

---

The file `file1.cpp` is at the moment still **unknown to git** (it's a new file!). It is in the so called **untracked** state. We need to tell git that we want to take track of it.

```
$> git add file1.cpp
```

show again status of the repo

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: file1.cpp

The file is ready to be committed in your (local) git repository!

# The first commit

Now the file `file.cpp` is in the **staging area**, ready to be committed in the repository. Maybe we may want to add other files to the commit, or maybe we just want to add just that file to the repository. In the latter case we just do

```
$> git commit
```

An editor will appear asking for a commit message. The message is compulsory.

We can commit into the repository a newly created file without adding it beforehand, by using

```
$> git commit <file>
```

You can commit by giving a short message directly:

```
$> git commit -m "This is my first file"
```

# The status after the commit

---

```
$> git commit -m "This is my first file"
```

```
[master aba508f] This is my first file  
1 file changed, 5 insertions(+)  
create mode 100644 file1.cpp
```

## And if you do `git status`

```
On branch master  
Your branch is ahead of 'origin/master' by 1 commit.  
(use "git push" to publish your local commits)  
nothing to commit, working tree clean
```

# What does it mean?

---

The first message says that in the **branch** master (the only one we have at the moment) we have created a commit.

A commit is effectively a snapshot of the situation at the moment of the commit, and is identified by an **hash key**, a long hexadecimal number whose first significant digits are shown (aba508f) and are normally enough to **identify the commit uniquely**.

You can at any time retrieve the situation of your file (those under git control) at a given commit using its hash key (we will see that there are also other ways).

The message also says that in that commit you have added a file (the mode indicates just the file permission), which consists of 5 lines.

# What does it mean?

---

The message of `git status` tells you that at this moment the branch you are in (master since is the only one at the moment) has 1 commit more than that stored in the remote repository. Indeed by default git names your remote repository as `origin` (you can change the name with `git remote`) and `origin/master` is branch master in the remote repo.

Moreover you are warned that you can use `push` to send the commit to the remote, there is nothing else to commit and there are no new files in your working tree.

# The remote

---

How can git know that your local repo (more precisely the master branch of your local repo) is one step ahead of the remote? Git has not communicated with GitHub since commit operates only on the local repo. Well, let's try to do

```
$> git branch -a -v
```

You get

```
* master aba508f [ahead 1] This is my first file  
remotes/origin/HEAD -> origin/master  
remotes/origin/master 5fe0589 Initial commit
```

What does it mean?



# The local remote repo

---

`git branch -a -v` tells git to show all (-a) **branches** and be a bit verbose (-v). The asterisk indicates the current branch (the only one we have so far!), but we have also other stuff.

The branches that begin with `remote/` are **local copies of the branch in the remote repository, made at the time of the last pull, or fetch, or clone.**

`origin` is the name given by git to the remote repository (you can change it and you can have more than one remote!). It refers to our GitHub repo.

Let's at the moment forget about the HEAD stuff. You may see that git is saying: the local copy of the remote repo is commit `5fe0589`, committed with message `Initial commit`.

My master branch is at commit `aba508f` ahead of 1 commit with respect to `5fe0589`.

# What should I do now?

Either continue working in your working tree, or you may decide that it is time to send your modifications to the remote repository. To do that:

```
$> git push
```

or, if you want to be more precise,

```
$> git push origin master
```

which tells git to push on the remote called origin the branch master.

Since master is the current branch, and origin the only remote repo we have, we can use the first, simplified, version of the command.

This command communicates with the remote, i.e. with GitHub, so it will not work if you are not connected to internet

# Your repo and the remote are now synchronized

---

The command push gives some messages that just tell that your master branch has been sent to the remote, and git status now give simply

```
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

and `git branch -a -v`

```
* master                aba508f This is my first file
remotes/origin/HEAD      -> origin/master
remotes/origin/master    aba508f This is my first file
```

All set! Git has also adjourned the local copy of the remote to match the one actually in GitHub.

# Adding more files and directories

---

Now I create a directory `src` where I put some other files `file2.cpp` and `file3.cpp`. I do

```
$> git add src
```

Adding a directory means adding **all files in the directory**.  
Doing `git status` gives indeed:

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: src/file2.cpp

new file: src/file3.cpp

As expected!

# Moving file

But now I realize that also file1.cpp should be in src! The simplest thing to do is to use

```
$> git mv file1.cpp src
```

git status now says

On branch master

Your branch is up to date with 'origin/master'.

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

renamed: file1.cpp -> src/file1.cpp

new file: src/file2.cpp

new file: src/file3.cpp

Great! I am ready for the next commit!

```
$> git commit -m"Added a few more files"
```

# Changing a file

You now work on file3.cpp to better an algorithm of correcting a bug. After the changes git status gives

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

modified: src/file3.cpp

no changes added to commit (use "git add" and/or "git commit -a")

As you see, besides the other information, it marks file3.cpp as modified. If you want to register the modifications in git you can type

```
$> git add src/file3.cpp
$> git commit -m"corrected a bug"
```

or, with a single command

```
$> git commit -a -m"corrected a bug"
```

Here, -a means "add all modified file before commit".

## A note

---

You can launch git commands from any directory of the working tree! Non necessarily from the root.

## Resume: The possible state of a file in the working directory

---

- ▶ **untracked** The file is not under git control. Maybe it is just a temporary file. Or **you want to put it under control using `git add`**.
- ▶ **modified** The file has been modified since the last **commit** in the repository. May be you want to add it to the staging area with `git add`.
- ▶ **staged** (in the index). Ready to be committed in the repository with a `git commit`



## Resume: Add and Commit

---

stage files for a commit

```
$> git add <edited_files>
```

create a new commit in the repository

```
$> git commit  
...  
<vim editor will come up,  
write a description for the commit>
```

or

```
$> git commit -m "message"
```

The last command writes the commit message directly.

# What to put in a commit message

---

A first line with a brief description (it is the one that is shown by some git commands)

Possibly followed by an empty line and a detailed description (possibly wrap lines to 72 characters)

Paragraphs are separated by empty lines.

- you may use lists using the - character

---

If you use the option `-m` you can only set the brief description for the commit (often it is enough!).

# Add, Commit

---

To recall, the basic operations are

- ▶ **git add file(s)** Add the file(s) to the staging area ready for commit. On a new file, it adds also the file to the git history.
- ▶ **git commit** Stores the staging area into the git local repository creating a new “commit”.

---

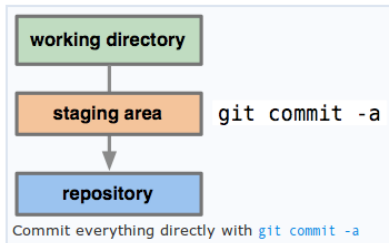
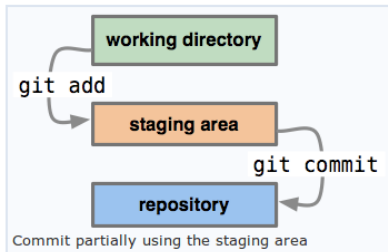
A useful shortcut:

```
$> git commit -a
```

commits all modified files (but **not** untracked ones)

---

# A graphical view



## Other useful commands

---

To show the differences with the last commit (you may specify a different commit and/or some specific files)

```
$> git diff <branch> <files>
```

To remove a file from the working directory and stages it for removal in the next commit (the file must be under git control)

```
$> git rm files
```

To change the name of a file under git control (stages the change for the next commit)

```
$> git mv oldname newname
```

# Adding whole directories

---

If you have a directory with new (or modified) files doing

```
$> git add DirectoryName
```

will add to the staging area all new (or modified) files in that directory. Remember that git operates on files, so trying to add an empty directory just does nothing!

## Resume: What is a commit?

---

A commit is a **snapshot of your git working tree at the moment of the commit.**

It is identified by a hash key, generated from the content of the files in the working tree and the hash key of the previous commit(s).

**Any part of the hash key (as long as unique in the repo)** can be used to address a commit. A commit may also have symbolic names called **tags**.

```
$> git show
```

shows the most recent commit (on the current branch).

# Special symbolic names for some commits

Shorthand	Definition
HEAD	Last commit
HEAD^	One commit ago
HEAD^^	Two commits ago
HEAD~1	One commit ago
HEAD~3	Three commits ago

These shorthands can be used in all git commands that refer to a commit. **HEAD may be thought as a pointer to the last commit.**

```
$> git log HEAD~3..HEAD  
$> git checkout HEAD~2
```

The first command prints a brief log describing the last 4 commits. **The second positions your working area to two commits ago.**



## A note

---

```
$> git checkout HEAD~2
```

positions the working area two commits ago, but does not cancel the other commits! You can have a look and operate on the working area but you **cannot commit** on the same branch!.

Indeed, you can commit only if you are at the **HEAD** of a branch (master at the moment). Git says that you are in **detached head state**. To go back to the HEAD

```
$> git checkout HEAD
```

or

```
$> git checkout master
```

## Some advice

---

- ▶ Do atomic commits: each commit should be related to a single “logical change” in your code: a bug fix, a new feature... **Avoid monster commits with a lot of files.**
  - ▶ Write significant commit messages. Messages like “this is a commit” are useless! A good message helps you to remember what you have done (and others to understand your work).
  - ▶ Git won't allow commits with empty messages.
-

# Communicate with a remote repository

---

To communicate with a remote repository we have basically three commands: **push**, **fetch** and **pull**.

We see now the simplest use of them. We still assume we have a single remote repository (called origin) and a single branch called master (we will talk later about branches). For generality in the following we indicate with remote-name the name of the remote.

# Push to the remote

---

If you want to send (**push** in the git jargon) your last commits to the remote, and so make it visible to your collaborators (or have a backup copy) You just have to do:

```
git push <remote-name> <branch/file>
```

It pushes all the commits for branch branch (the current branch by default) onto the remote remote-name (origin by default).

## Fetch from the remote

Now you want to get from the remote changes made by your collaborators:

```
$> git fetch <remote-name>
```

Fetches all the commits from the remote repository and store them **locally** in

remotes/<remote-name>/<branch>

You can then examine them (read only!) by doing, for instance,

```
$> git checkout remotes/origin/new-branch
```

You can finally merge the commits into your master (or any other branch)

```
$> git checkout master
```

```
$> git merge remotes/<remotename>/<branch>
```

## Pull from the remote

---

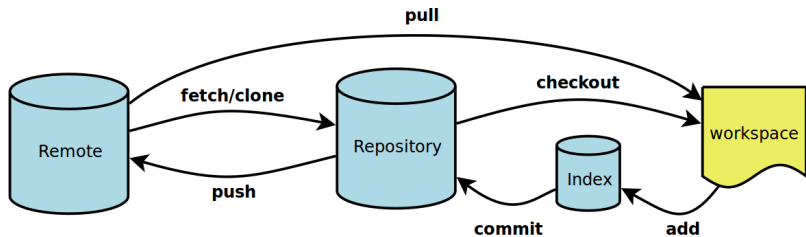
If you trust the work from your collaborators you do fetch and merge at the same time (don't worry, you can always go back!).

```
$> git pull <remote-name> <branch>
```

Is equivalent to `git fetch` followed by `git merge`. If you omit the branch and remote name you will pull all branches in the repository corresponding to your local branches.

It is the most used command when you want to retrieve commits from a remote repository.

# add, commit, push pull...



# When things goes wrong

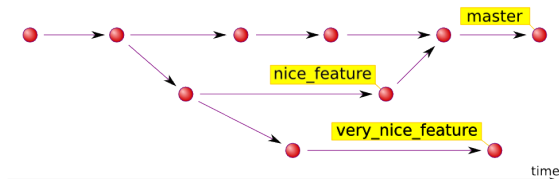
---

What happens if both yourself and your collaborators have worked on the same file, making two conflicting changes?

Well we have indeed a **conflict**, and the merge fails. We will deal it with it in a moment. But first it is useful to introduce branches, since merging is also related to them.



# Branches



**Branching is one of the most important feature of git.**

When you create a repo you create also the master branch. But often you want to create a branch to work on a specific topic (or just to experiment something) without affecting the master branch.

# Branching- create

---

create a new branch

```
$> git branch <shiny_new_branch>
```

Go to that branch.

```
$> git checkout <shiny_new_branch>
```

Or create and position yourself in a new branch with a single command

```
$> git checkout -b <shiny_new_branch>
```

---

# Get other branches

---

list local branches

```
$> git branch  
  b1  
* b2  
  master
```

list remote branches (i.e. branches fetched from remote)

```
$> git branch -r
```

list all branches (local and remote)

```
$> git branch -a
```

## Branching - merge

---

Often when you have finished working on a branch and you are happy of your work you want to merge the work on the master branch (or any other branch).

---

To merge a branch into master:

```
$> git checkout master
$> git merge <branch_name>
Updating e0f73f9..cd928c7
Fast-forward
```

# Branching - merge

---

Often when you have finished working on a branch and you are happy of your work you want to merge the work on the master branch (or any other branch).

To merge a branch into master:

```
$> git checkout master  
$> git merge <branch_name>  
Updating e0f73f9..cd928c7  
Fast-forward
```

Everything was fine, but sometimes conflicts may arise!

```
CONFLICT (content): Merge conflict in file.cpp  
Automatic merge failed; fix conflicts and then commit  
the result.
```

# Why conflicts?

---

Git tries to do the best to merge the work you have done in the branch, but maybe in the meantime you or one of your collaborators have modified the same lines of the same file on master (or the branch to which you are merging).

In that case you have a conflict, git stops the merge and it is up to you to sort out the mess.  
If you want to cancel the merge and go back to the situation just before:

```
$> git merge --abort
```

# Branching - solve conflicts 1

---

Manual solution. In the file with the conflict you have

```
<<<<<< HEAD
...portion in the HEAD commit
=====
...portion in the branch you are merging
>>>>>> branch_name
```

Use your preferred editor and modify the file.  
Eventually you should eliminate the <<<< and >>>>  
lines.

# Branching - solve conflicts 2

---

## Using a gui

---

```
$> git mergetool --tool=<tool>  
merge tool candidates:  meld opendiff kdiff3 tkdiff  
xxdiff tortoisemerge gvimdiff diffuze ecmerge p4merge  
araxis emerge vimdiff
```

You may use one of the tools indicated (you must have it installed). Some, like meld, are rather sophisticated and provide a nice graphical interface.



## Branching - solve conflicts 3

---

After you have fixed the conflicting files, you have to stage them

```
$> git add file.cpp
```

A commit is needed when all the conflicts are solved!

```
$> git commit
```

# Branching - delete

delete a branch

```
$> git checkout master  
$> git branch -d <branch_to_delete>
```

will cause an error if the branch was not merged with master! It's a safety feature!  
To delete it without merging

```
$> git branch -D <branch_to_delete>
```

To delete a remote branch

```
$> git push origin :<branch_to_delete>
```

the syntax comes from the generic generic push command

```
git push <remotename> <localbranch_name>:<remotebranch_name>
```

so we push an empty branch into the remote branch to be deleted

# Branching - stash

Checkout with hanging modifications is forbidden!

```
$> git checkout master  
error: Your local changes to the following files  
would be overwritten by checkout: ...
```

if you are not ready to commit, you can stash the modifications

```
$> git stash save
```

and bring them back later

```
$> git stash { pop | apply }
```

remember to clear the stash (in particular if you use apply, which does not delete the stash).

```
$> git stash list  
$> git stash clear
```

# Merging and rebasing

---

There are in fact two ways in git to integrate work from one branch into another: merge and rebase. Rebasing however is a more advanced feature. We will skip it in this lecture. You find the description in the git book, or other resources on the web.

## Relation between merging and pulling/fetching

---

We have postponed the issue of conflict after explaining branches because in fact you have in your local repository some special branches (at least one: `remotes/origin/master!`), called remote branches, which contains the snapshot of the corresponding branch in the remote repository at the moment of the last pull/fetch command.

So, with `fetch` you retrieve the latest remote branches from the remote repo, and you can merge them later. With `pull` you fetch and merge them in one go. Of course, you may have conflicts, exactly as you may have them when merging local branches.

# Fixing commits: amend reset and revert

Sometimes you want to correct your **last commit**, for instance to add some additional stuff, without creating a new commit

```
$> <do your additional work/add/rm>  
$> git commit --amend
```

**Never amend a commit that has been already pushed to a remote repo: use git revert**

# Eliminating commits: Reset

`git reset` sets your HEAD to a specified commit. So it can be used to “delete commits”

```
$> git reset <file>
```

Removes the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes. [It is the opposite of `git add`.](#)

```
$> git reset
```

This unstages all files without overwriting any changes, giving you the opportunity to add again the files to stage, from scratch.

```
$> git reset <commit>
```

Move the current branch backward to `<commit>`, but leave the working directory alone: all changes made since `<commit>` will still reside in the working directory (but not staged!)

# Hard Reset

```
$> git reset --hard
```

In addition to unstaging changes, the `--hard` flag tells Git to **overwrite all changes in the working directory** too. Put another way: **this obliterates all uncommitted changes**, so you go back to the situation at your last commit.

```
$> git reset --hard <commit>
```

Move the current branch tip backward to `<commit>` and reset both the staging area and the working directory to match. You are back at the situation at commit `<commit>`.

```
$> git reset [--<commit>] --hard <files>
```

resets all `<files>` to their state at commit.



## A warning

---

Reset may change the history. Do not use it to “delete” commits that have already been pushed to a remote repository.

Use revert in this case!.

---

# Revert

---

git revert reverts commits by **playing them backwards**. So it does not “delete” them. It is history safe.

```
$> git revert HEAD~3
```

reverts the changes specified by the fourth last commit in HEAD and creates a new commit with the reverted changes.

revert may be used to revert the action of a set of past commits (advanced usage).

---

# Collaboration - remote

---

Share a new branch with a remote called `repo_name`

```
$> git push <repo_name> <branch_name>  
$> git pull <repo_name> <branch_name>
```

To access a new branch in the repo

```
$> git checkout --track -b <branch_local_name> \  
> <repo_name>/<branch_name>
```

# Get info

---

```
$> git log
```

Shows commit logs.

```
$> git shortlog
```

Summarizes log output, showing commit description from each contributor

```
$> git show <object>
```

Show some properties of the object (blob, commit...).

Use a graphical interface like gitk...

## Some advanced features: submodules

---

A git repository may contain another git repository (and this can be done recursively!).

It is useful, for instance, if you want to keep in your repository a project that is maintained by others, so that you can use it and update it if some useful changes happen upstream.

We do not explain here submodules, it will take too long. Even if I am using them in the repo with the examples for the PACS course. There is a video on the PACS youtube channel on them and I will give instructions if necessary.

## Common issues - II

**Q** I want to discard my edit and get the latest version of the file!

**A** Simple

```
$> git checkout -- file
```

**Q** I want to get the file from another branch or another commit!

**A** It's the same command as before:

```
$> git checkout <branch|commit> -- file
```

## Common issues - III

---

**Q** Get a commit from another branch

**A** cherry-pick it to the correct one

```
$> git checkout correct_branch  
$> git cherry-pick commit_hash
```

**Q** cannot get the branch of the local remote

**A** Try with a fetch

```
$> git fetch remote_name  
$> git remote show remote_name
```

## Common issues - IV

Q the merge has gone bananas!

A give up the merge and try again

```
$> git merge <branch_to_be_merged>  
$> #!&#%$&^  
$> git reset --hard <original_branch>
```

or, more simply

```
git merge --abort
```

Q i want a file from that branch!

A get it with

```
$> git checkout <branch> <files>
```



# Common issues - V

## Q git push refuses to work

```
remote: error: refusing to update checked out branch: refs/heads/<branch_name>
remote: error: By default, updating the current branch in a non-bare repository
remote: error: is denied, because it will make the index and work tree
inconsistent
remote: error: with what you pushed, and will require 'git reset --hard' to
match
remote: error: the work tree to HEAD.
remote: error:
remote: error: You can set 'receive.denyCurrentBranch' configuration variable to
remote: error: 'ignore' or 'warn' in the remote repository to allow pushing into
remote: error: its current branch; however, this is not recommended unless you
remote: error: arranged to update its work tree to match what you pushed in some
remote: error: other way.
remote: error:
remote: error: To squelch this message and still keep the default behaviour, set
remote: error: 'receive.denyCurrentBranch' configuration variable to 'refuse'.
To <repo>
! [remote rejected] <branch_name> -> <branch_name> (branch is currently checked
out)
```

## A use a bare repo

```
$> git clone --bare <repo> <repo>.git
```

## Tips & tricks - I

---

- ▶ `gitk` is your friend

## Tips & tricks - I

---

- ▶ `gitk` is your friend
- ▶ do small, atomic commits

## Tips & tricks - I

---

- ▶ `gitk` is your friend
- ▶ do small, atomic commits
- ▶ read git output

# Tips & tricks - I

- ▶ `gitk` is your friend
- ▶ do small, atomic commits
- ▶ read git output
- ▶ branch in `PS1`: add to `.bashrc`

```
function GitBranch {  
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"  
    -e "s/^\*\s//" )"  
  test -n "$_branch" && echo -e "@$_branch"  
}  
PS1='\u@\h:\w$(GitBranch)> '
```

# Tips & tricks - I

- ▶ `gitk` is your friend
- ▶ do small, atomic commits
- ▶ read git output
- ▶ branch in `PS1`: add to `.bashrc`

```
function GitBranch {  
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"  
    -e "s/^\*\s//" )"  
  test -n "$_branch" && echo -e "@$_branch"  
}  
PS1='\u@\h:\w$(GitBranch)> '
```

- ▶ watch out for non history-safe commands after pushing! (`cherry-pick`, `rebase`, ...)

# Tips & tricks - I

- ▶ `gitk` is your friend
- ▶ do small, atomic commits
- ▶ read git output
- ▶ branch in `PS1`: add to `.bashrc`

```
function GitBranch {  
  _branch="$(git branch 2>/dev/null | sed -e "/^\s/d"  
    -e "s/^\*\s//" )"  
  test -n "$_branch" && echo -e "@$_branch"  
}  
PS1='\u@\h:\w$(GitBranch)> '
```

- ▶ watch out for non history-safe commands after pushing! (`cherry-pick`, `rebase`, ...)
- ▶ everything else (and more...) on git

<http://book.git-scm.com/index.html>

## Tips & tricks - II

---

- ▶ bash autocompletion

[http://git.kernel.org/?p=git/git.git;a=blob\\_plain;f=contrib/completion/git-completion.bash;hb=HEAD](http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD)



## Tips & tricks - II

---

- ▶ bash autocompletion

[http://git.kernel.org/?p=git/git.git;a=blob\\_plain;f=contrib/completion/git-completion.bash;hb=HEAD](http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD)

- ▶ colored output

```
$> git config --global color.ui true
```

## Tips & tricks - II

- ▶ bash autocompletion

[http://git.kernel.org/?p=git/git.git;a=blob\\_plain;f=contrib/completion/git-completion.bash;hb=HEAD](http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD)

- ▶ colored output

```
$> git config --global color.ui true
```

- ▶ no empty push

```
$> git config --global push.default nothing
```

## Tips & tricks - II

- ▶ bash autocompletion

[http://git.kernel.org/?p=git/git.git;a=blob\\_plain;f=contrib/completion/git-completion.bash;hb=HEAD](http://git.kernel.org/?p=git/git.git;a=blob_plain;f=contrib/completion/git-completion.bash;hb=HEAD)

- ▶ colored output

```
$> git config --global color.ui true
```

- ▶ no empty push

```
$> git config --global push.default nothing
```

- ▶ creating archives with the content of a commit

```
$> git archive <commit_or_branch> | gzip > \  
> <archive_name>.tgz
```