Laboratorio di Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/
Code versioning: techniques and tools
Software versions

- All software has multiple versions:
  - Each time you edit a program
  - Versions within a development cycle
    - Test release with debugging code
    - Alpha, beta of final release
  - Variations for different platforms
    - Hardware and software
  - Different releases of a product
Version control

• Version control tracks multiple versions of code.

• In particular, allows:
  • old versions to be recovered
  • multiple versions to exist simultaneously

• Typically multiple users can contribute to software development and version control systems allow them to collaborate:
  • multiple versions of multiple users, merging their contribute
  • tracks who did what
Version control

- Version control tracks multiple versions of code.

In general **version control** (or **revision control**, or **source control**) is about managing multiple versions of documents, programs, web sites. It works best on text documents but can manage also binary files such as images.

- multiple versions of multiple users, merging their contribute

- tracks who did what
Why using version control?

• Because it is useful
  • You will want old/multiple versions
  • Without version control, can’t recreate project history
  • Allows to go back in history, to solve bugs introduced since the last version of the code

• Because everyone does
  • A basic software development tool. Beware of those who do not use it.
  • If you need to share coding responsibilities or maintenance of a codebase with another person, you need version control.
Why using version control?

• For working by yourself:
  • Gives you a “time machine” for going back to earlier versions
  • Gives you great support for different versions of the same project

• For working with others:
  • Greatly simplifies concurrent work, merging changes
A Code Base does not just mean code! It also includes:

- Documentation
- Build Tools (CMake files, Makefiles, etc.)
- Configuration files

All these files may change over time and older versions have to be kept.
A Code Base does not just mean code! It also includes:

- Documentation
- Build Tools (CMake files, Makefiles, etc.)
- Configuration files

All these files may change over time and older versions have to be kept.

Manage these things using a version control system (VCS)

A version control system is a system which allows for the management of a code base.
Types of Version Control Systems

- **Local only** - keeps a local database of changes in your local machine filesystem.

- **Centralized** - (Subversion, CVS), require a connection to a central server and “checkout”

- **Distributed** - (Git, Mercurial) allow for local systems to be “mirrors” of the central repo. You don’t need to be connected to the central server to get work or commits done.
Local only

- IDE like CLion and Eclipse maintain a local history of each file developed within the IDE.
- Pros: you don’t have to do anything. This versioning is automatic.
- Cons: each file has its own history. You do not know which versions of several files was used at a certain moment.
Local only

- IDE like CLion and Eclipse maintain a local history of each file developed within the IDE.
- **Pros:** you don't have to do anything. This versioning is automatic.
- **Cons:** each file has its own history. You do not know which versions of several files was used at a certain moment.
IDE like CLion and Eclipse maintain a local history of each file developed within the IDE.

- **Pros:** You don't have to do anything. This versioning is automatic.
- **Cons:** Each file has its own history. You do not know which versions of several files was used at a certain moment.
Centralized

- Traditional version control system
- Server with database
- Clients have a working version
- Examples
  - CVS
  - Subversion
  - Visual Source Safe
- Challenges
  - Multi-developer conflicts
  - Client/server communication
Distributed

- Authoritative server by convention only
- Every working checkout is a repository
- Get version control even when detached
- Backups are trivial

Examples

- Git
- Mercurial
- BitKeeper
- Bazaar
Overview of the process

• Files are kept in a **repository**

• Repositories can be local or remote to the user

• The user edits a copy called the **working copy**

• Changes are **committed** to the repository when the user is finished making changes

• Other people can then access the repository to get the new code

• Can also be used to manage files when working across multiple computers
Branching

- Branches allows multiple copies of the code base within a single repository.
- Different customers have different requirements
  - Customer A wants features A, B, C
  - Customer B wants features A & C but not B because his computer is old and it slows down too much.
  - Customer C wants only feature A due to costs
- Each customer has their own branch.
- Different versions can easily be maintained
Basic features of a VCS

• Check-in and check-out of items to repository
• Creation of baselines (labels/tags)
  • e.g. “Version 1.0 released!”
• Control and manipulation of branching
  • management of multiple versions
• Overview of version history
  • Allows to see who changed what
Check out / check in

- If you want to make a change the file needs to be **checked out** from the repository.
- When changes are completed the new code is **checked-in**.
- A **commit** consists of a set of checked in files and the diff between the new and parent versions of each file.
- Each check-in is accompanied by a user name and other meta data.
- Check-ins can be exported from the Version Control System the form of a **patch**.
Revision

- Consider
  - Check out a file
  - Edit it
  - Check the file back in
- This creates a new version of the file
- With each revision, system stores
  - The diffs for that version (typically for efficiency, the VCS doesn’t store entire new file, but stores diff with previous version)
  - The new file version number
  - Other metadata
  - Author
  - Time of check in
  - Log file message
Merge

- There are occasions when multiple versions of a file need to be collapsed into a single version.
- E.g. a feature from one branch is required in another, or two developers worked on the same file.
- This process is known as a **merge**.
There are occasions when multiple versions of a file need to be collapsed into a single version. E.g. a feature from one branch is required in another, or two developers worked on the same file. This process is known as a **merge**.
Merging

1. Start with a file, e.g. v.1.5
2. Bob makes changes A to v.1.5
3. Alice makes changes B to v.1.5
4. Assume Alice checks in first
5. Current revision is v.1.6 = apply(B, v.1.5)
6. Now Bob checks in
7. System notices that Bob checked out v.1.5, but current version is v.1.6
8. Bob has not made his changes in the current version!
9. The system complains
10. Bob is told to update his local copy of the code
11. Bob does an update
12. This applies Alice’s changes B to Bob’s code
13. Two possible outcomes of an update:
   - Success
   - Conflicts
Merge success

- Assume that:
  - apply(A, apply(B, v.1.5)) = apply(B, apply(A, v.1.5))
- Then then order of changes didn’t matter
- Same result whether Bob or Alice checks in first
- The version control system is happy with this
- Bob can now check in his changes
  - Because apply(B, apply(A, v.1.6)) = apply(B, v.1.6)
Merge conflict

- Assume

- \( \text{apply}(A, \text{apply}(B, 1.5)) \neq \text{apply}(B, \text{apply}(A, 1.6)) \)

- There is a conflict
  - The order of the changes matters
  - Version control will complain
  - Arise when two programmers edit the same piece of code
  - One change overwrites another
Merge conflict

- System cannot apply changes when there are conflicts:
  - Final result is not unique
  - Depends on order in which changes are applied

- Version control shows conflicts on update

- Conflicts must be resolved by hand
Conflicts

• Conflict detection is based on “nearness” of changes
  • Changes to the same line will conflict
  • Changes to different lines will likely not conflict
• Note: Lack of conflicts does not mean Alice’s and Bob’s changes work together
Merging conflicts

- Merging is syntactic
- Semantic errors may not create conflicts
  - But the code is still wrong
- You are lucky if the code doesn’t compile
  - Worse if it does . . .
Problem example

- The Linux kernel runs on different processors (ARM, x86, MIPS). These can require significant differences in low level parts of the code base.
- Many different modules
- Old versions are required for legacy systems
- Because it is open source, any one can download and suggest changes.
Git
History

• Came out of Linux development community

• Linus Torvalds, 2005

• Initial goals:
  • Speed
  • Support for non-linear development (thousands of parallel branches)
  • Fully distributed
  • Able to handle large projects like Linux efficiently
Features

- It is distributed
- Everyone has the complete history
- Everything is done offline
- No central authority
- Changes can be shared even without a server
- Snapshot storage instead of diff
Features

- It is distributed
- Everyone has the complete history
- Everything is done offline
- No central authority
- Changes can be shared even without a server
- Snapshot storage instead of diff
Code base

Contains:
• directories
• files
Repository

- Contains
  - files
  - commits
  - ancestry relationships
  - records history of changes
Ancestry relationships

- form a directed acyclic graph (DAG)
Ancestry graph features

- **HEAD**
  - is current checkout
  - usually points to a branch
Git component

- Index
- “staging area”
- what is to be committed
Getting started

- Three areas of Git
- The HEAD
  - last commit snapshot, next parent
- Index
  - Proposed next commit snapshot
- Working directory
  - Sandbox

Local Operations
- checkout the project
- stage files
- commit

Unmodified/modified Files
Staged Files
Committed Files

Index  HEAD
Basic workflow

• Init a repo(sitory): init to start a new project or clone an existing project
  • will create a “.git” directory. This is your local repo.

• Edit files

• Stage the changes (add files to repo)

• Review your changes

• Commit the changes
Init a repo(sitory): init to start a new project or clone an existing project

- will create a “.git” directory. This is your local repo.

Edit files

Stage the changes (add files to repo)

Review your changes

Commit the changes

You can work as much as you like in your working directory, but the repository isn’t updated until you commit something
What not to track

• It’s important to tell Git what files you do not want to track

• Temp files, executable files, etc. do not need version control (and can cause major issues when merging!)

• We add the filenames to the special file .gitignore. We store this file in the repository
Getting started: edit file

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
  - Commit the changes
Getting started: stage

- A basic workflow
  - Edit files
  - **Stage the changes**
  - Review your changes
  - Commit the changes

```git add``
Getting started: review

- A basic workflow
- Edit files
- Stage the changes
- Review your changes
- Commit the changes

```
$ git add hello.txt

$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   hello.txt
```
Getting started: commit

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
- **Commit the changes**

```
git commit
```
Getting started

- A basic workflow
  - Edit files
  - Stage the changes
  - Review your changes
  - Commit the changes
File life lifecycle

Files outside Git
Commits and graphs

- A commit is when you tell git that a change (or addition) you have made is ready to be included in the project.
- When you commit your change to git, it creates a commit object, that represents the complete state of the project, including all the files in the project.
- The very first commit object has no “parents”.
- Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object.
  - Hence, most commit objects have a single parent.
  - You can also merge two commit objects to form a new one, in this case the new commit object has two parents.
  - Hence, commit objects form a directed graph.
- Git is all about using and manipulating this graph.
Commits and graphs

• A head is a reference to a commit object
• The “current head” is called HEAD (all caps)
• Usually, you will take HEAD (the current commit object), make some changes to it, and commit the changes, creating a new current commit object
• This results in a linear graph: A → B → C → ... → HEAD
Good practice

• In git, “Commits are cheap.” Do them often.

• When you commit, you must provide a one-line message stating what you have done

  • Terrible message: “Fixed a bunch of things”

  • Better message: “Corrected the calculation of median scores”

• Commit messages can be very helpful, to yourself as well as to your team members
Branching and merging

- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch
Branching and merging

- Branch annotates which commit we are working on

- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch
Branching and merging

- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch

Diagram:

- A → B → C
- D → E
- Merge to master branch
- Bug 123
Branching and merging

- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch
Branching and merging

- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch
Retrieve old commit

• Use **checkout** to select a committed version of the project or to branches

• allows to go back in time, e.g. to see when a bug was introduced

• we can also just evaluate the difference between current ad older versions of code base
Working with remote

- **Add** and **Commit** your changes to your **local** repo
- **Pull from remote** repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
- **Push** your changes to the **remote** repo
- **Fetch** to retrieve from remote without merging with current code.
Working with remote

- **Add** and **Commit** your changes to your **local** repo
- **Pull from remote** repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
- **Push** your changes **to the remote** repo

Good practice: Pull then Push

Push will update the remote server.
If you are out of date, Git will reject that push.
Git at a glance
Git and CLion
Check git install

- OSX command line development tools include git. Linux and Windows require to install it.
Start using git on a project

- Enable VCS integration
Start using git on a project

- Enable VCS integration
Start using git on a project

- Enable VCS integration
Start using git on a project

- Enable VCS integration
• Stage files adding them to git versioning. Use project view or Version Control tab that shows also invisible files like those of the CLion project.
Commit

- Commit whole directory or single files
Commit

- Commit whole directory or single files
Commit whole directory or single files
Ignore files

- Add to `.gitignore` with `Ignore`
Branch

- Use the contextual menu to add new branches, or to checkout them.
- The same menu can be used to merge the current branch with one of the list. The same applies for comparison.
• CLion can use Github as remote server, and allows also to create an account from the options.
Use scenarios
Scenario 1: bug fix

First public release of the hot new product
Scenario 1: bug fix

First public release of the hot new product

Time

Releases

1.0 → 1.3
Scenario 1: bug fix

First public release of the hot new product

Internal development continues, progressing to version 1.3
Scenario 1: bug fix

First public release of the hot new product

Internal development continues, progressing to version 1.3

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.
First public release of the hot new product

Internal development continues, progressing to version 1.3

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

Note that there are now two lines of development beginning at 1.0. This is branching.
Scenario 1: bug fix

1. A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

Note that there are now two lines of development beginning at 1.0. This is branching.
Note that there are now two lines of development beginning at 1.0. This is branching.

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.
Note that there are now two lines of development beginning at 1.0. This is branching.

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.
Scenario 1: bug fix

1.0

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

The bug fix should also be applied to the main code line so that the next product release has the fix.

Note that there are now two lines of development beginning at 1.0. This is branching.

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.
Scenario 1: bug fix

1.0

Time

Releases

1.3

bugfix

The bug fix should also be applied to the main code line so that the next product release has the fix.

Note that two separate lines of development come back together in 1.4.
This is merging.

Note that there are now two lines of development beginning at 1.0.
This is branching.
Scenario 2: normal dev.

You are in the middle of a project with three developers named a, b, and c.
Scenario 2: normal dev.

You are in the middle of a project with three developers named a, b, and c.
Scenario 2: normal dev.

You are in the middle of a project with three developers named a, b, and c.

At the beginning of the day everyone checks out a copy of the code.
A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.
**Scenario 2: normal dev.**

You are in the middle of a project with three developers named a, b, and c.

The local versions isolate the developers from each other’s possibly unstable changes. Each builds on 1.5, the most recent stable version.

A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.
You are in the middle of a project with three developers named a, b, and c.

At the end of the day everyone checks in their tested modifications. A check in is a kind of merge where local versions are copied back into the version control system.
Scenario 3: debugging

A software system is developed through several revisions.
Scenario 3: debugging

A software system is developed through several revisions.

In 1.7 you suddenly discover a bug has crept into the system. When was it introduced?

With version control you can check out old versions of the system and see which revision introduced the bug.
You are building software on top of a third-party library, for which you have source.

You begin implementation of your software, including modifications to the library.
Scenario 4: external libraries

You are building software on top of a third-party library, for which you have source.

You begin implementation of your software, including modifications to the library.
Scenario 4: external libraries

You are building software on top of a third-party library, for which you have source. You begin implementation of your software, including modifications to the library.

A new version of the library is released. Logically this is a branch: library development has proceeded independently of your own development.
You merge the new library into the main code line, thereby applying your modifications to the new library version.
Reading material

• E. Sink, “Version Control by Example” - cap. 2, 4, 8
These slides are based on the material of:

- Prof. Aiken,
- Dr. N. Benatar,
- Prof. R. Anderson, Univ. Washington
- P. Chen, Stanford