Git

While it may not seem important in the early stages of programming, revision control—using software to track multiple versions of your code, and let multiple developers collaborate easily—is a crucial part of professional development (or even larger-scale academic development). There are many different popular revision control systems, but most of them share some common principles. We are going to focus on git, which is currently the most popular tool, and is quite featureful. We strongly recommend that you learn revision control early, and make it part of your standard practice for software development (and most other things that you do). We are going to introduce some high-level features, and motivations for using revision control here, but refer you to the git book for details: https://git-scm.com/book/en/v2 .

Past versions

The basic principle underlying most revision control system is that you commit your work into the revision control system, and it then keeps a snapshot of that version of your work. When you commit, you write a log entry, which describes what you have done. Later, you can review the log, and return to an older version of your work if you need to. For example, if you decide to rewrite a piece of code to improve it, but find out that you have instead, broken your program, you can return to a prior working version.

A “novice tools” approach to this problem would be to keep copies of your work, and manage them by hand. You might think “well I could just copy all my code before I start, then use that copy later if I need to.” If you are consciously aware that you are about to undertake a risky code rewrite, you might take this course of action. However, what if you start modifying your code, and only later realize the trouble? If you use a revision control system, you should commit regularly, and thus will have good revisions in the past. Furthermore, if you make regular copies manually, you will find it hard to manage them all by hand.

Revision systems such as git also have excellent tools for working with past versions. For example, git has a command called bisect which lets you search for where in the past you broke something. Suppose that I know that a feature was working 6 months ago, however, today, I ran some tests and realized the feature was broken. I would like to go back and find exactly which version broke the feature, see what changed in that version, and then correct the code. The bisect command asks git to help me search for the first broken version. In particular, bisect binary searches through the revisions (thus the name), to find where the problem occured. You can either guide the search manually by telling git whether each revision that it visits is good or bad, or you can have git perform the search fully automatically by providing a shell script that determines if the revision is correct or broken.

Revision control is not limited to code, and, in fact, is incredibly useful for any large, collaborative project (or even for just managing your own data on a small scale). To provide a concrete example of bisect, we use git to revision control all of the materials for the book this course is based on—LATEX source, code examples, animations, and video recordings. One of the animations got inadvertently deleted, and we needed to restore it from the most recent version. Writing a shell script to test if that file existed, then running git bisect found the revision where the file was deleted in a matter of seconds, and let us restore it from the previous revision (and be sure we had the most up-to-date version!).

Collaboration

When you work on a software project with hundreds of source files and dozens of developers, how do you keep everyone up to date on the latest source? E-mailing files back and forth would be a nightmare. Even if you only have two people working on a project, managing changes between the developers is a first-order concern.

Revision control systems such as git support notions of pushing—sending your changes to another repository (located on another computer)—and pulling changes—getting the most recent version from another repository. If you try to push your own changes, but are not up to date with the other repository, git will require you to pull first, so that you cannot unexpectedly overwrite someone else's changes. When you pull, git will take care to make sure your changes are integrated with the remote changes.

If you pull and have not made any changes to your own repository (since the last time you pushed to that repository), git will simply update you to the most recent version of the repository. If you have made changes, git will try to merge your changes with the remote changes. If you have changed different files, or different parts of the same file, git will typically handle the merge automatically. However, if git cannot merge automatically, it will indicate what problems need your attention, and require you to fix them before you push your changes back.

Multiple versions of the present

Revision control tools, such as git, not only let you return to past versions, but also let you keep multiple different "present" versions, via a feature called branches. To see why you might want multiple "present" versions, imagine that you are developing a large software project, and have decided to begin adding a new, complex feature. Adding this feature will take you and your team of five developers four weeks to complete. One week into the effort, a critical problem is found in the currently released version of your software, which must be fixed as soon as possible.

Such a situation is a great use of branches. The development team might maintain one branch, production which contains code that is ready for release. The only changes that may be made to the production branch are those that are ready for deployment. Another development branch can be used to work on active development—adding new features, testing them, etc. In the situation described above, the development branch is where the team would be adding the new complex feature.

How do these branches help? The developer assigned to fix the bug in the production code can checkout that branch (working on that version of the “present”), while the other developers can continue working on the development branch. In fact, this developer could (and should) make a new branch (let us call it bugfix) to work on the bug fix. The developer can then commit changes to bugfix, and other developers can switch to this branch if they need to work on that fix as well. When the bug fix is complete, it can be merged back into the production branch, incorporating the changes into the production software.

At the same time, development of new features continues on the development branch. When the bug fix is completed, it can also be merged into the development branch. Likewise, when the development of new features are completed, and ready for release, they can be merged back into the production branch. Whenever these merges happen, git will perform similar actions as in the case of merging in changes pulled from another repository—either handling it automatically if it can, or requiring the developer to figure out how to resolve conflicts.

Note that this is a bit of a small-scale motivational example. If we were really developing a large piece of software, we would almost certainly want more branches than those described above. We may have different developers working on different features in parallel, and want to manage combination of those changes with branches. We might also want branches for different stages of testing, or other purposes.

Read more!

Having, hopefully, shown you some of the great benefits of revision control, we recommend further reading from the online book about git: http://git-scm.com/book/en/v2/

In our opinion, reading (trying out, and understanding) the first two chapter is a “must” even for those who are only casually interested in learning to program. If you are interested in serious software development, you should master chapters 3–6 quickly, and then continue to hone your git skills as you develop your other programming skills. Over time, you should learn about the remaining material, and work to use git fluently.