Durham University

Version Control: Maintenance

COMP51915 – *Collaborative Software Development*
Michaelmas Term 2024

Christopher Marcotte[1]

[1] For errata and questions please contact christopher.marcotte@durham.ac.uk

Outline

- Branches
- Merging
- Merge Conflicts
- File Management
- Maintenance Summary

Learning Outcomes

- Critical understanding of branch management strategies in version control
- Ability to merge non-conflicting contributions to a repository
- Ability to resolve simple merge conflicts
- Ability to use the helpful GitHub systems, and understanding of best practice

Branches

Branches are divergent development histories of a `git` managed repository.

In `git`, the branching model is very lightweight so things can be done quickly.

Your typical approach in modifying a repository should be
1. *branch*,
2. *edit*, and finally
3. *merge*.

Branching and *merging* are critical tools for maintaining a repository, as they ensure your working directory is clean and separate from `main`.

Creating Branches

A new branch `newbranch` can be made as easily as `git branch newbranch`.

But this only *creates* the new branch – switch using `git checkout newbranch`.

Creating Branches

A new branch `newbranch` can be made as easily as `git branch newbranch`.

But this only *creates* the new branch – switch using `git checkout newbranch`.

From here, we can:

1. edit a file (e.g. with `echo "import numpy as np" >> file2.py`), and
2. add a file (e.g. `touch file3.h`), and
3. stage the changes (with `git add file2.py file3.h`), and even
4. commit changes to `newbranch` (e.g. `git commit -m "commit-ing crimes"`)

Branch Management

Return to the main branch with `git checkout main` and look with `cat file2.py`.

We'll find that `file2.py` is *empty* on `main`, and `file3.h` is *missing* entirely.

[2] Indeed, `git` will not let you switch to a different branch if your directory is modified and unstaged.

Branch Management

Return to the main branch with `git checkout main` and look with `cat file2.py`.

We'll find that `file2.py` is *empty* on `main`, and `file3.h` is *missing* entirely.

Changing branches *modifies* the content in the working directory.[3]

To get the changes we committed in `newbranch` to be reflected in the `main` branch we need to discuss *merging* and the `git merge` command.

[3] Indeed, `git` will not let you switch to a different branch if your directory is modified and unstaged.

Merging

The command to merge content from `newbranch` into the current branch is:

```
git merge newbranch
```

Merging

The command to merge content from `newbranch` into the current branch is:

```
git merge newbranch
```

Because we have no conflicts, `git` merges using *fast-forward*.

Fast-forward requires your current commit (`main`) occur in the history of your target commit (`newbranch`) – and works by moving the pointer for the current commit to the target commit – this is very cheap!

Merging

The command to merge content from `newbranch` into the current branch is:

```
git merge newbranch
```

Because we have no conflicts, `git` merges using *fast-forward*.

Fast-forward requires your current commit (`main`) occur in the history of your target commit (`newbranch`) – and works by moving the pointer for the current commit to the target commit – this is very cheap!

There are other methods `git` will use to manage a more complex merge, but these are selected automatically – *usually*.

Branching Strategies

We mentioned that there are a number of branching strategies used for `git`.

We can not cover all possibilities; some of the more widely used are:

- long-running branches, discriminated by *code stability*
- short-lived branches, discriminated by *feature implementation*

Remote repositories and collaboration give us even more options. We'll briefly discuss these later.

Branching Strategies

We mentioned that there are a number of branching strategies used for `git`.

We can not cover all possibilities; some of the more widely used are:

- long-running branches, discriminated by *code stability*
- short-lived branches, discriminated by *feature implementation*

Remote repositories and collaboration give us even more options. We'll briefly discuss these later.

`git` differs from most VC systems because of the inexpensive branching model – this makes multiple short-lived branches scalable and uniquely `git`.

Deleting Branches

We can delete a branch using `git branch -d newbranch`.

You should typically delete a branch after it's been merged into `main` – the label for both branches will point to the same commit from then on.

Deleting Branches

We can delete a branch using `git branch -d newbranch`.

You should typically delete a branch after it's been merged into `main` – the label for both branches will point to the same commit from then on.

This is part of repository maintenance, though you might have good reason for keeping a branch around.

If you are wondering if you should do something as part of maintenance, ask yourself:

Will doing this change now help avoid confusion later?

Merge Conflicts

Thus far we've seen simple merges. Merge conflicts occur between two branches have modified a file in incompatible ways. Let's create a merge conflict!

Merge Conflicts

Thus far we've seen simple merges. Merge conflicts occur between two branches have modified a file in incompatible ways. Let's create a merge conflict!

1. Create a new branch `git branch newbranch` and
   - modify `file2.py`,
   - then `git add file2.py`,
   - and `git commit`.
2. On branch `main`,
   - edit `file2.py`,
   - then `git add file2.py`
   - and `git commit`.

Creating a Merge Conflict

When we try to `git merge newbranch` we'll get a failure notice:

```
Auto-merging file2.py
CONFLICT (content): Merge conflict in file2.py
Automatic merge failed; fix conflicts and then commit the result.
```

Creating a Merge Conflict

When we try to `git merge newbranch` we'll get a failure notice:

```
Auto-merging file2.py
CONFLICT (content): Merge conflict in file2.py
Automatic merge failed; fix conflicts and then commit the result.
```

The merge has been paused while you fix your repository. Use `git status`:

```
Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file2.py

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

Resolving Merge Conflicts

`git` has helpfully added lines to point out the conflict. Use `cat file2.py`:

```
<<<<<<< HEAD
  return np.sin(x)-x
=======
  return x*x*x
>>>>>>> newbranch
```

You resolve the merge conflict by :

1. modifying the section,
2. staging the file with `git add` – this marks the conflict as resolved, and
3. finishing the commit with `git commit`.

File Management

As you use `git` for actual projects, you'll find the working directory cluttered with all manner of nonsense.

In the case of data, this presents a security issue:

How do we avoid publishing private files in the working directory?

File Management

As you use `git` for actual projects, you'll find the working directory cluttered with all manner of nonsense.

In the case of data, this presents a security issue:

How do we avoid publishing private files in the working directory?

One way is to exclude these files from `git` is including them (by name, or extension) in the `gitignore` file.

I recommend you include extensions foremost in the `gitignore` as that will prevent you from accidentally including a filename you should not.

Maintenance Summary

1. Use branches liberally with `git` – they're cheap and useful
2. Delete branches after merging them, they'll clog up your labels
3. Developing a branch management strategy is a good idea, but you needn't stick to one
4. Merges can be resolved automatically if they're simple enough, but conflicts require manual resolution
5. Use the other tools to keep a clean working directory – i.e. `gitignore`.