

SECTION 1: CODE REASONING + VERSION CONTROL

CSE 331 – Summer 2018

slides borrowed and adapted from Alex Mariakis and CSE 390a, CSE 331 lecture slides, and Justin Bare and Deric Pang Section 1 slides.

OUTLINE

- **Introductions**
- **Code Reasoning**
 - Forward Reasoning
 - Backward Reasoning
 - Weaker vs. Stronger statements
- **Version control**

REASONING ABOUT CODE

- **Two purposes**
 - *Prove* our code is correct
 - Understand *why* code is correct
- **Forward reasoning: determine what follows from initial conditions**
- **Backward reasoning: determine sufficient conditions to obtain a certain result**

TERMINOLOGY

- The **program state** is the values of all the (relevant) variables
- An **assertion** is a logical formula referring to the program state (e.g., contents of variables) at a given point
- An assertion **holds** for a program state if the formula is true when those values are substituted for the variables

TERMINOLOGY

- An assertion before the code is a **precondition** - these represent assumptions about when that code is used
- An assertion after the code is a **postcondition** - these represent what we want the code to accomplish

FORWARD REASONING

- Given: Precondition
- Finds: postcondition for given precondition.
 - Aka Finds program state after executing code, when using given assumptions of program state before execution.

FORWARD REASONING

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
//
```

```
x = x + y
```

```
//
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```

FORWARD REASONING

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
//
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```


FORWARD REASONING

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
// {x >= 16, y = 16}
```

```
x = sqrt(x)
```

```
//
```

```
y = y - x
```

```
//
```

FORWARD REASONING

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
// {x >= 16, y = 16}
```

```
x = sqrt(x)
```

```
// {x >= 4, y = 16}
```

```
y = y - x
```

```
//
```

FORWARD REASONING

```
// {x >= 0, y >= 0}
```

```
y = 16;
```

```
// {x >= 0, y = 16}
```

```
x = x + y
```

```
// {x >= 16, y = 16}
```

```
x = sqrt(x)
```

```
// {x >= 4, y = 16}
```

```
y = y - x
```

```
// {x >= 4, y <= 12}
```

FORWARD REASONING

```
// {true}
if (x>0) {
    //
    abs = x
    //
}
else {
    //
    abs = -x
    //
}
//
//
```

FORWARD REASONING

```
// {true}
if (x>0) {
    // {x > 0}
    abs = x
    //
}
else {
    // {x <= 0}
    abs = -x
    //
}
//
//
```

FORWARD REASONING

```
// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
}
else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
}
//
//
```

FORWARD REASONING

```
// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
}
else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
}
// {x > 0, abs = x OR x <= 0, abs = -x}
//
```

FORWARD REASONING

```
// {true}
if (x>0) {
    // {x > 0}
    abs = x
    // {x > 0, abs = x}
}
else {
    // {x <= 0}
    abs = -x
    // {x <= 0, abs = -x}
}
// {x > 0, abs = x OR x <= 0, abs = -x}
// {abs = |x|}
```


BACKWARD REASONING

- Given: Postcondition
- Finds: The weakest precondition for given postcondition.

BACKWARD REASONING

- Given: Postcondition
- Finds: The weakest precondition for given postcondition.
- So, finds most general assumption code will use to get given postcondition.

BACKWARD REASONING

//

$$a = x + b;$$

//

$$c = 2b - 4$$

//

$$x = a + c$$

// {x > 0}

BACKWARD REASONING

//

$$a = x + b;$$

//

$$c = 2b - 4$$

$$// \{a + c > 0\}$$

$$x = a + c$$

$$// \{x > 0\}$$

BACKWARD REASONING

//

$$a = x + b;$$

// { $a + 2b - 4 > 0$ }

$$c = 2b - 4$$

// { $a + c > 0$ }

$$x = a + c$$

// { $x > 0$ }

BACKWARD REASONING

```
// Backward reasoning is used to determine the
// weakest precondition
// {x + 3b - 4 > 0}
a = x + b;
// {a + 2b - 4 > 0}
c = 2b - 4
// {a + c > 0}
x = a + c
// {x > 0}
```

ASIDE: WEAKEST PRECONDITION?

- What is weakest precondition?
- Well, precondition is just a statement, so...Better ask what makes a statement weaker vs. Stronger?

WEAKER VS. STRONGER

- **Weaker statements = more general**
- **Stronger statements = more specific aka more informational**
- **Stronger statements are more restrictive**
 - Ex: $x = 16$ is stronger than $x > 0$
 - Ex: “Alex is an awesome TA” is stronger than “Alex is a TA”
- **If A implies B, A is stronger and B is weaker.**
- **If B implies A, B is stronger and A is weaker.**
- **If neither, then A and B not comparable.**

HOARE TRIPLES

- **Hoare triples are just an extension of logical implication**
 - Hoare triple: $\{P\} S \{Q\}$
 - P = precondition
 - S = single line of code
 - Q = postcondition
- **A Hoare triple can be valid or invalid**
 - Valid if for all states for which P holds, executing S always produces a state for which Q holds
 - Invalid otherwise

HOARE TRIPLE

EXAMPLE #1

- $\{x \neq 0\} y = x * x; \{y > 0\}$
- Is this valid?

HOARE TRIPLE

EXAMPLE #1

- $\{x \neq 0\} \ y = x * x; \{y > 0\}$
- Is this valid?
 - Yes

HOARE TRIPLE

EXAMPLE #2

- Is $\{\text{false}\} S \{Q\}$ a valid Hoare triple?

HOARE TRIPLE

EXAMPLE #2

- Is $\{\text{false}\} S \{Q\}$ a valid Hoare triple?
 - Yes. Because P is false, there are no conditions when P holds
 - Therefore, for all states where P holds (i.e. none) executing S will produce a state in which Q holds

HOARE TRIPLE

EXAMPLE #3

- Is $\{P\} S \{true\}$ a valid Hoare triple?

HOARE TRIPLE

EXAMPLE #3

- Is $\{P\} S \{\text{true}\}$ a valid Hoare triple?
 - Yes. Any state for which P holds that is followed by the execution of S will produce some state
 - For any state, true always holds (i.e. true is true)

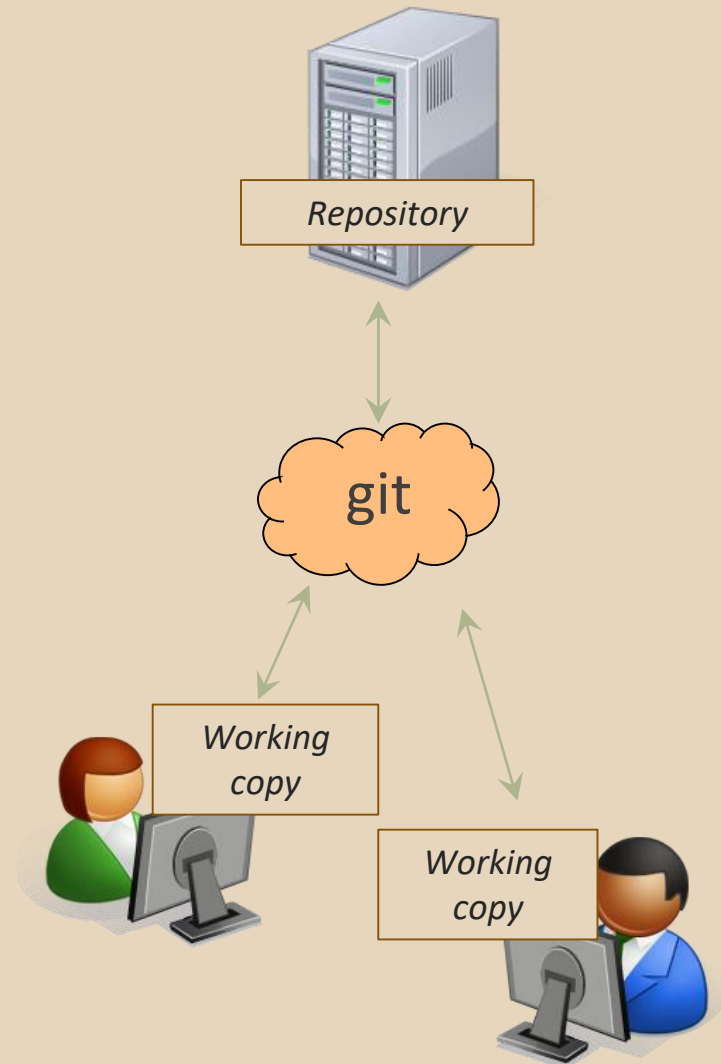
VERSION CONTROL

WHAT IS VERSION CONTROL?

- Also known as source control/revision control
- System for tracking changes to code
 - Software for developing software
- Essential for managing projects
 - See a history of changes
 - Revert back to an older version
 - Merge changes from multiple sources
- We'll be talking about git/GitLab, but there are alternatives
 - Subversion, Mercurial, CVS
 - Email, Dropbox, USB sticks (don't even think of doing this)

VERSION CONTROL ORGANIZATION

- A *repository* stores the master copy of the project
 - Someone creates the repo for a new project
 - Then nobody touches this copy directly
 - Lives on a server everyone can access
- Each person *clones* her own *working copy*
 - Makes a local copy of the repo
 - You'll always work off of this copy
 - The version control system syncs the repo and working copy (with your help)



REPOSITORY

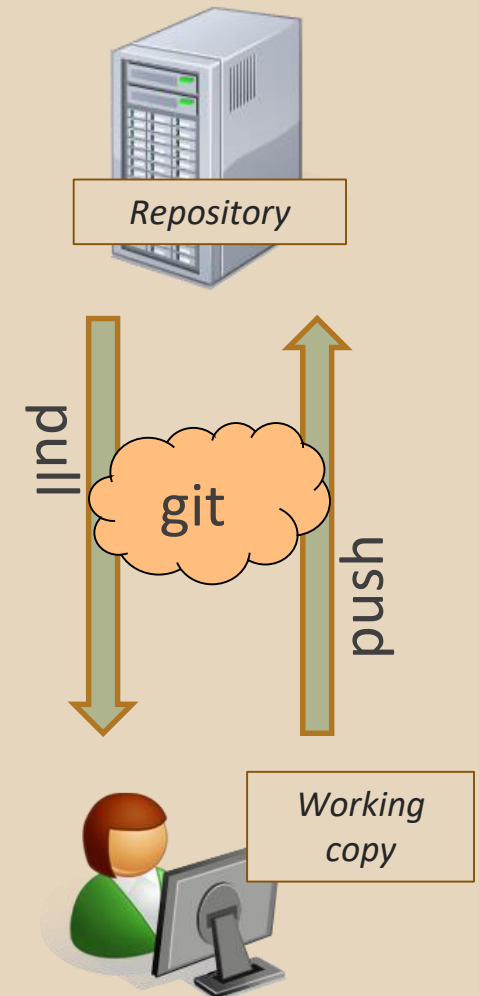
- Can create the repository anywhere
 - Can be on the same computer that you're going to work on, which might be ok for a personal project where you just want rollback protection
- But, usually you want the repository to be robust:
 - On a computer that's up and running 24/7
 - Everyone always has access to the project
 - On a computer that has a redundant file system
 - No more worries about that hard disk crash wiping away your project!
- We'll use CSE GitLab – very similar to GitHub but tied to CSE accounts and authentication

VERSION CONTROL

COMMON ACTIONS

Most common commands:

- **commit / push**
 - integrate changes *from* your working copy *into* the repository
- **pull**
 - integrate changes *into* your working copy *from* the repository

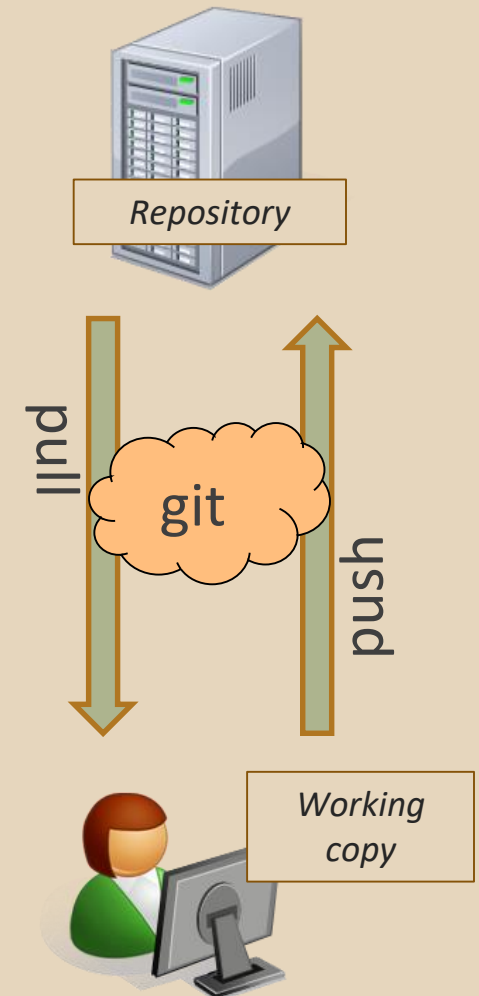


VERSION CONTROL

UPDATING FILES

In a bit more detail:

- You make some local changes, test them, etc., then...
- `git add` – tell git which changed files you want to save in repo
- `git commit` – save all files you've "add"ed in the local repo copy as an identifiable update
- `git push` – synchronize with the GitLab repo by pushing local committed changes

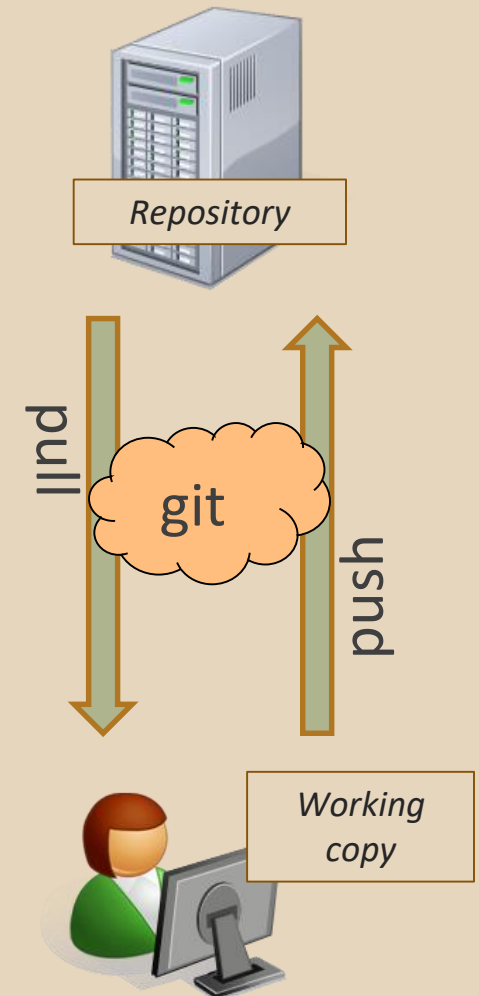


VERSION CONTROL

COMMON ACTIONS (CONT.)

Other common commands:

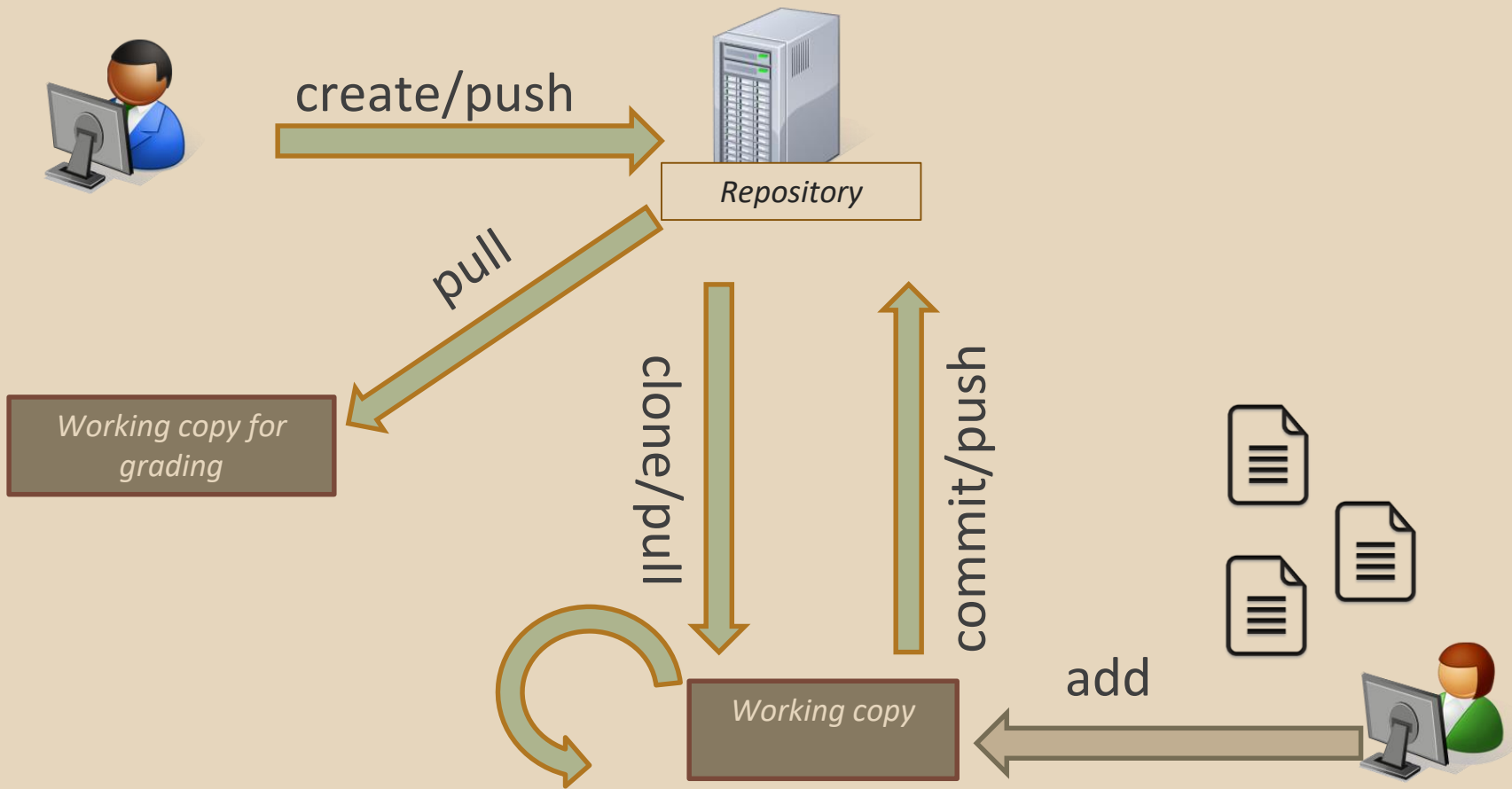
- **add, rm**
 - add or delete a file in the working copy
 - just putting a new file in your working copy does not add it to the repo!
 - still need to commit to make permanent



THIS QUARTER

- We distribute starter code by adding it to your GitLab **repo**. You retrieve it with **git clone** the first time then **git pull** for later assignments
- You will write **code** using Eclipse
- You turn in your files by **adding** them to the repo, **committing** your changes, and eventually **pushing** accumulated changes to GitLab
- You “turn in” an assignment by **tagging** your repo and pushing the tag to GitLab
- You will **validate** your homework by **SSHing** onto attu, cloning your repo, and running an Ant build file

331 VERSION CONTROL



AVOIDING GIT PROBLEMS

- For the projects in this class, you should never have to merge
 - Except when the staff pushes out a new assignment
- Rules of thumb for working in multiple places:
 - Each time before you start working on your assignment, git pull to get the latest code
 - Each time after you are done working for a while, git add/commit/push in order to update the repository with the latest code