CS 35L

Week 10

TA: Tomer Weiss
March-07-2016

goo.gl/7klZ6L

Slides

Announcements

- Student presentations today:
  - HBM Memory
  - Why Sarcasm is Such a Problem in AI
  - Computers Can Tell If You're Bored

  web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html

- Submit report here
  - For reference on presentation, grading, please refer to this rubric.
- Assignment 10 is due this Friday, 11:55pm

Announcements - Please Fill teaching evaluation survey

Students have until 8:00 AM Saturday, March 12 to log into MyUCLA to complete evaluations for your courses listed below:

COM SCI 35L section 6

You have the option to use class time for your students to complete their evaluations with smartphones, iPads, and tablets. Please note that not all classrooms are WI-FI enabled.

For more details about this process, please contact your departmental Evaluation Coordinator or visit our website at:

http://www.oid.ucla.edu/assessment/eip

Thank you!
Evaluation of Instruction Program
eip@oid.ucla.edu

Final Review

Week 10

Notes about these review slides:

- Not comprehensive
  - Meant to give you a review of some of the concepts we covered in the course
- Conceptual understanding of concepts is more important than memorization
  - The final is open book/note, if you need something specific you'll have it in front of you!
    - no electronic devices though
- Questions are posed throughout the review slides
  - Strive to be able to confidently answer all these questions

Final Information

- Tuesday, March 15, 2016, 11:30am-2:30pm
- Boelter 3760 (our regular room)
- Open book and open note
  - No calculators, smartphones, smartwatches, etc.
- 50% of course grade (from syllabus)

Week 1

GNU/Linux

- Open-source operating system
  - **Kernel**: core of operating system
    - Allocates time and memory to programs
    - Handles file system and communication between software and hardware
  - **Shell**: interface between user and kernel
    - Interprets commands user types in
    - Takes necessary action to cause commands to be carried out
  - **Programs**

Files and Processes

- Everything is either a **process** or a **file**:
  - **Process**: an executing program identified by PID
  - **File**: collection of data
    - A document
    - Text of program written in high-level language
    - Executable
    - Directory
    - Devices

The Basics: Shell

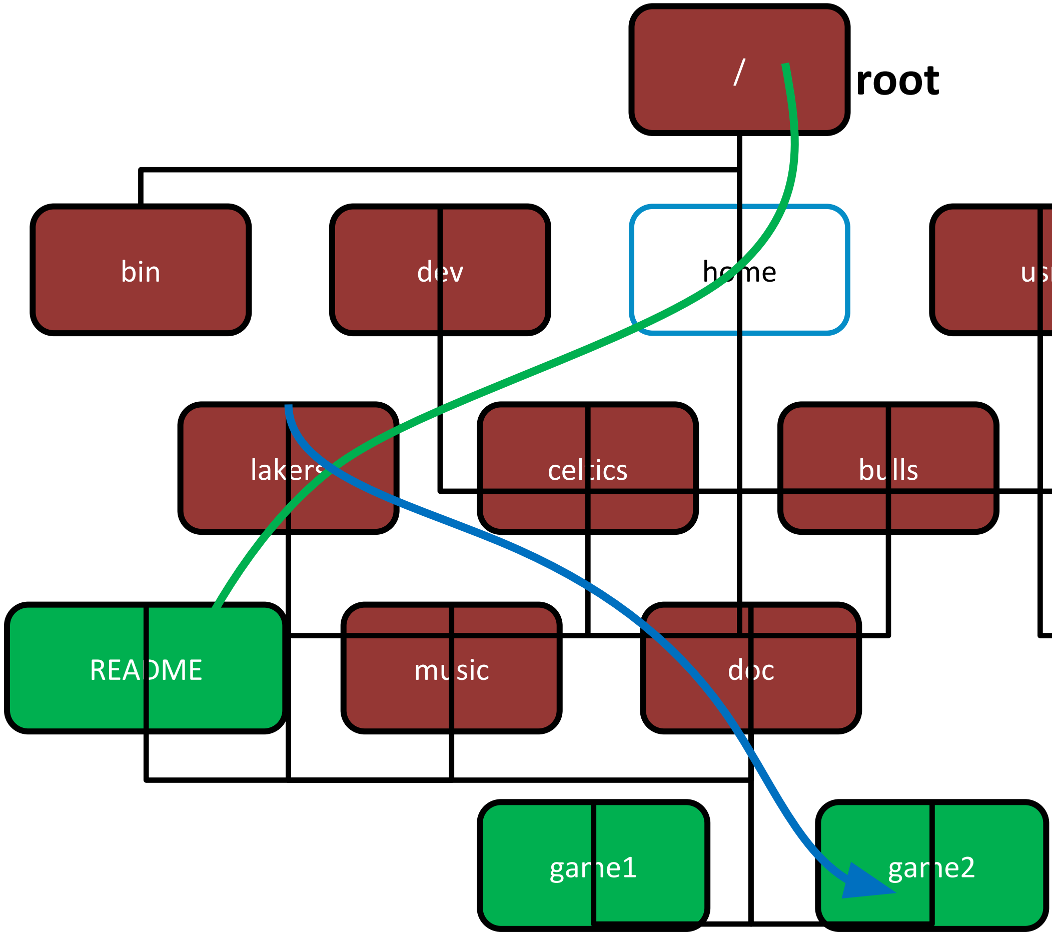
Some of the CLI utilities from you should be familiar with:

- `pwd`
- `cd`
- `mv`
- `cp`
- `rm`
- `mkdir`
- `rmdir`
- `ls`
- `ln`
- `touch`
- `find`
- `which`
- `man`
- `ps`
- `kill`
- `diff`
- `wget`
- `tr`
- `wc`
- `grep`
- `and others...`

The Basics: Shell

- How do I find where files are on the system?
- How do I find out what options are available for a particular utility?
- When is a file a file and when is it a process?
- What types of links are there?

Absolute Path vs. Relative Path

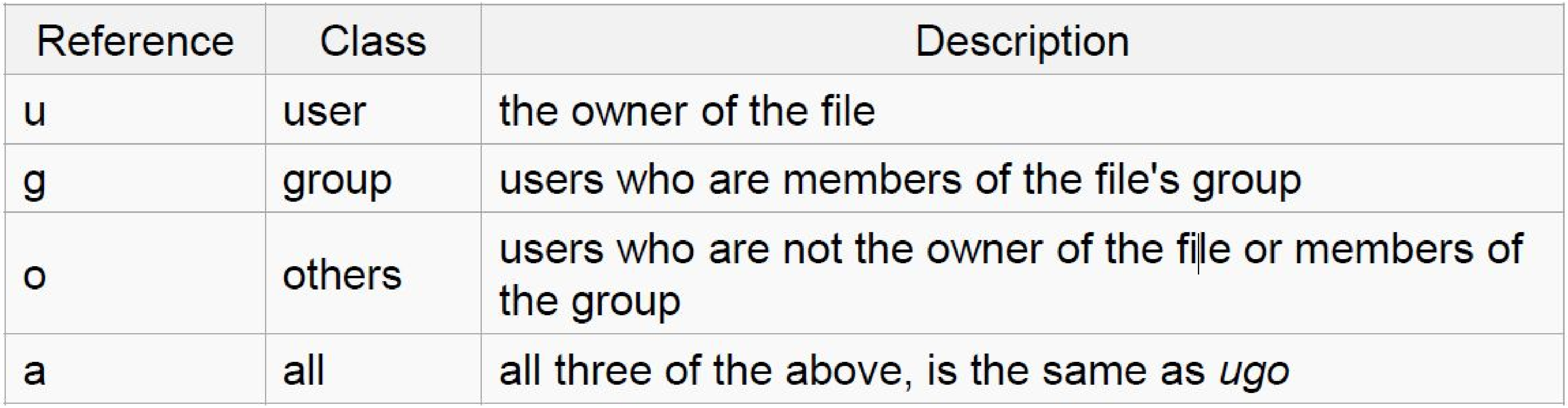


Current directory: home

What are the differences between absolute and relative paths?

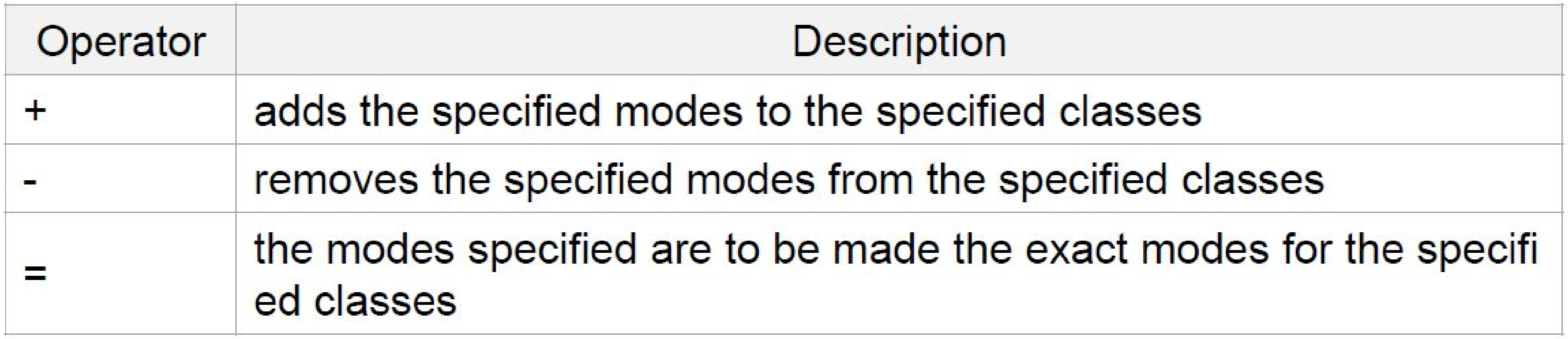
Linux File Permissions

- chmod
  - read (r), write (w), executable (x)
  - User, group, others
- Why do we have permissions at all?

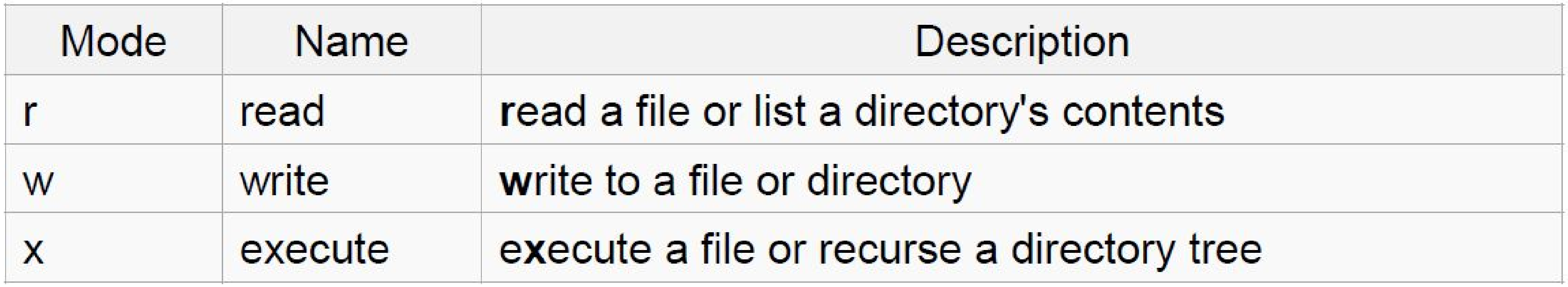


Reference	Class	Description
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are not the owner of the file or members of the group
a	all	all three of the above, is the same as *ugo*

The Basics: chmod (symbolic)

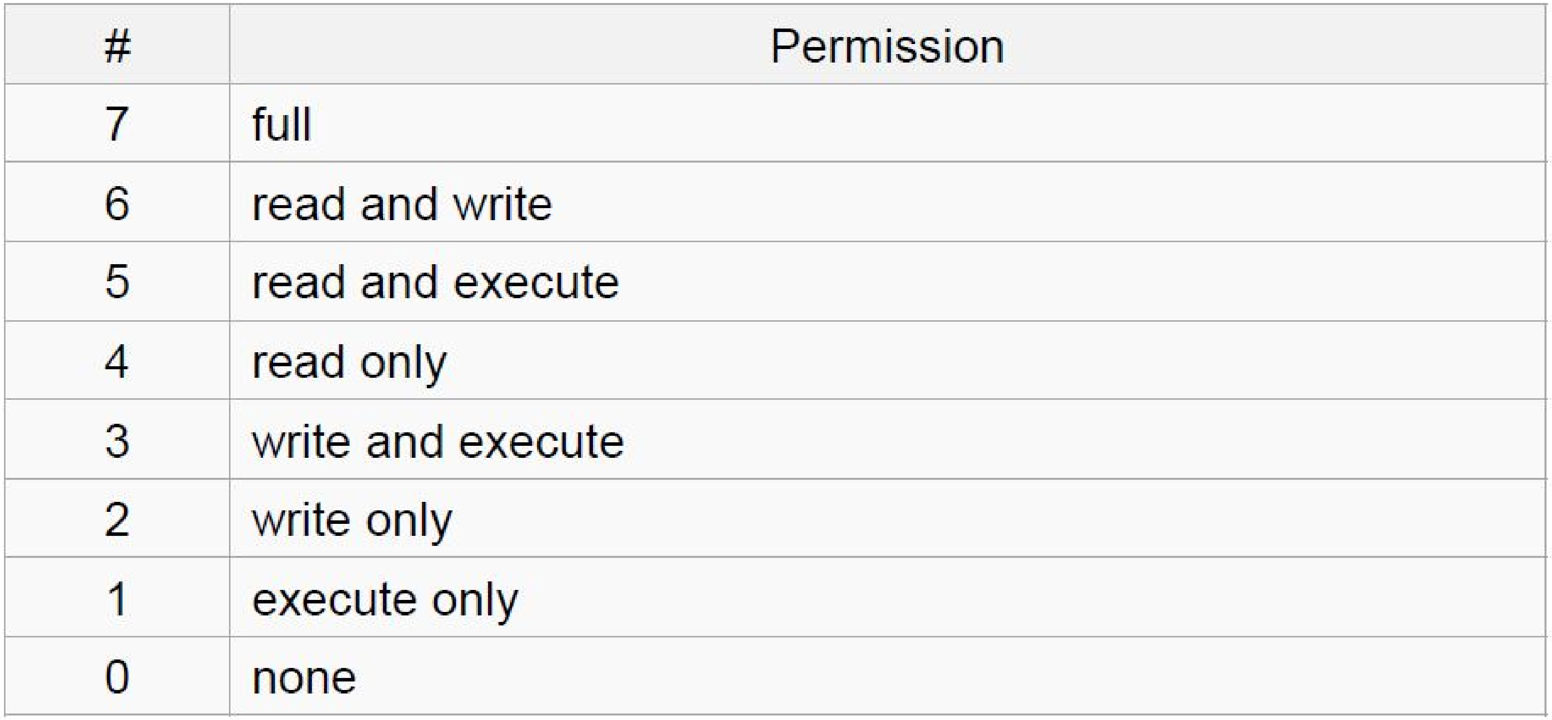


Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes



Mode	Name	Description
r	read	**r**ead a file or list a directory's contents
w	write	**w**rite to a file or directory
x	execute	e**x**ecute a file or recurse a directory tree

The Basics: chmod (numeric)



#	Permission
7	full
6	read and write
5	read and execute
4	read only
3	write and execute
2	write only
1	execute only
0	none

- Usage

– chmod ["references"]["operator"]["modes"] "file1" ...

Example: **chmod** ug+rw mydir, **chmod** a-w myfile,

Example: **chmod** ug=rx mydir, **chmod** 664 myfile

Week 2

Locale

A locale

- Set of parameters that define a user’s cultural preferences
  - Language
  - Country
  - Other area-specific things
- What else does the locale affect?

`locale` command

prints information about the current locale environment to standard output

Environment Variables

- Variables that can be accessed from any child process
- Why do we have these at all? What functions do they serve?

Common ones:

- **HOME**: path to user’s home directory
- **PATH**: list of directories to search in for command to execute
- Change value:

  ```
  export VARIABLE=…
  ```

Locale Settings Can Affect Program Behavior!!

Default sort order for the `sort` command depends:

- LC_COLLATE=‘C’: sorting is in ASCII order
- LC_COLLATE=‘en_US’: sorting is case insensitive except when the two strings are otherwise equal and one has an uppercase letter earlier than the other.

Other locales have other sort orders!

Compiled vs. Interpreted

Compiled languages

- Programs are translated from their original source code into machine code that is executed by hardware
- Efficient and fast
- Require recompiling
- Work at low level, dealing with bytes, integers, floating points, etc.
- Ex: C/C++
- When would I want to use a compiled language?

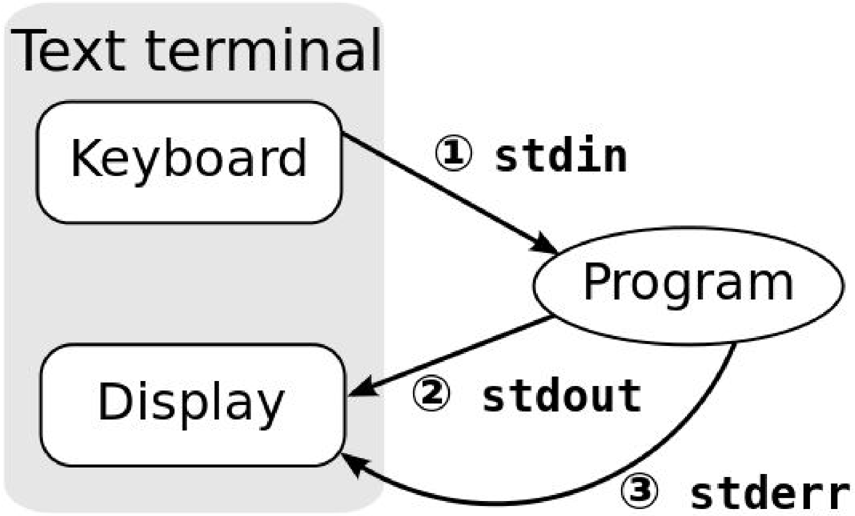
Interpreted languages

- Interpreter program (the shell) reads commands, carries out actions commanded as it goes
- Much slower execution
- Portable
- High-level, easier to learn
- Ex: PHP, Ruby, bash
- When would I want to use an interpreted language?

Why do we have the notion of compiled and interpreted languages?
Why not just have one type of language?

Standard Streams

- Every program has these 3 streams to interact with the world
  - stdin (0): contains data going into a program
  - stdout (1): where a program writes its output data
  - stderr (2): where a program writes its error msgs



Redirection and Pipelines

- *program < file* redirects *file to programs’s stdin*:

```
cat <file
```

- *program > file* redirects *program*'s stdout to *file2*:

```
cat <file >file2
```

- *program 2> file* redirects *program*'s stderr to *file2*:

```
cat <file 2>file2
```

- *program >> file* **appends** program’s stdout to *file*
- *program1 | program2* assigns stdout of *program1* as the stdin of *program2; text 'flows' through the pipeline*

```
cat <file | sort >file2
```

Why would we want to redirect I/O? What are some examples of use cases for I/O redirection? How do we implement this in C?

Regular Expressions

- Notation that lets you search for text with a particular pattern:
  - For example: starts with the letter a, ends with three uppercase letters, etc.
- Why do these exist? Why not just program our own text searching? Are the expressions the same across languages? Platforms?
- What's the difference between a basic and an extended regular expression? When would I use either?
- How do I write a regular expression to accomplish x?

- http://regexpal.com/ to test your regex expressions
- Simple regex tutorial http://www.icewarp.com/support/online_help/203030104.htm

4 Basic Concepts

- Quantification
  - How many times of previous expression?
  - Most common quantifiers: ?(0 or 1), *(0 or more), +(1 or more)
- Grouping
  - Which subset of previous expression?
  - Grouping operator: ()
- Alternation
  - Which choices?
  - Operators: [] and |
    - Hello|World   [A B C]
- Anchors
  - Where?
  - Characters: ^ (beginning) and $ (end)
- How do I use a combination of the above to accomplish tasks?

Regular Expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(…\) and \{…\}.
.	Both	Match any single character except NUL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, .* means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont’d)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont’d)

\n	BRE	Replay the nth subpattern enclosed in \\(and \\) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{*n*,*m*}	ERE	Just like the BRE \\{*n*,*m*\\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
\{*n*\}	Exactly n occurrences of the preceding regular expression
\{*n*,\}	At least n occurrences of the preceding regular expression
\{*n*,*m*\}	Between n and m occurrences of the preceding regular expression

Examples

Expression	**Matches**
tolstoy	The seven letters tolstoy, anywhere on a line
^tolstoy	The seven letters tolstoy, at the beginning of a line
tolstoy$	The seven letters tolstoy, at the end of a line
^tolstoy$	A line containing exactly the seven letters tolstoy, and nothing else
[Tt]olstoy	Either the seven letters Tolstoy, or the seven letters tolstoy, anywhere on a line
tol.toy	The three letters tol, any character, and the three letters toy, anywhere on a line
tol.*toy	The three letters tol, any sequence of zero or more characters, and the three letters toy, anywhere on a line (e.g., toltoy, tolstoy, tolWHOtoy, and so on)

Text Processing Tools

- You should be familiar with:
  - `wc`: outputs a one-line report of lines, words, and bytes
  - `head`: extract top of files
  - `tail`: extracts bottom of files
  - `tr`: translate or delete characters
  - `grep`: print lines matching a pattern
  - `sort`: sort lines of text files
  - `sed`: filtering and transforming text
- What are the differences between tr, sed, and grep? When would I use each one?
- How can I combine and use these tools together?

`sort`, `comm`, and `tr`

`sort`: sorts **lines** of **text** files

- Usage: sort [OPTION]…[FILE]…
- Sort order depends on locale
- C locale: ASCII sorting

`comm`: compare two **sorted** files **line by line**

- Usage: comm [OPTION]…FILE1 FILE2
- Comparison depends on locale

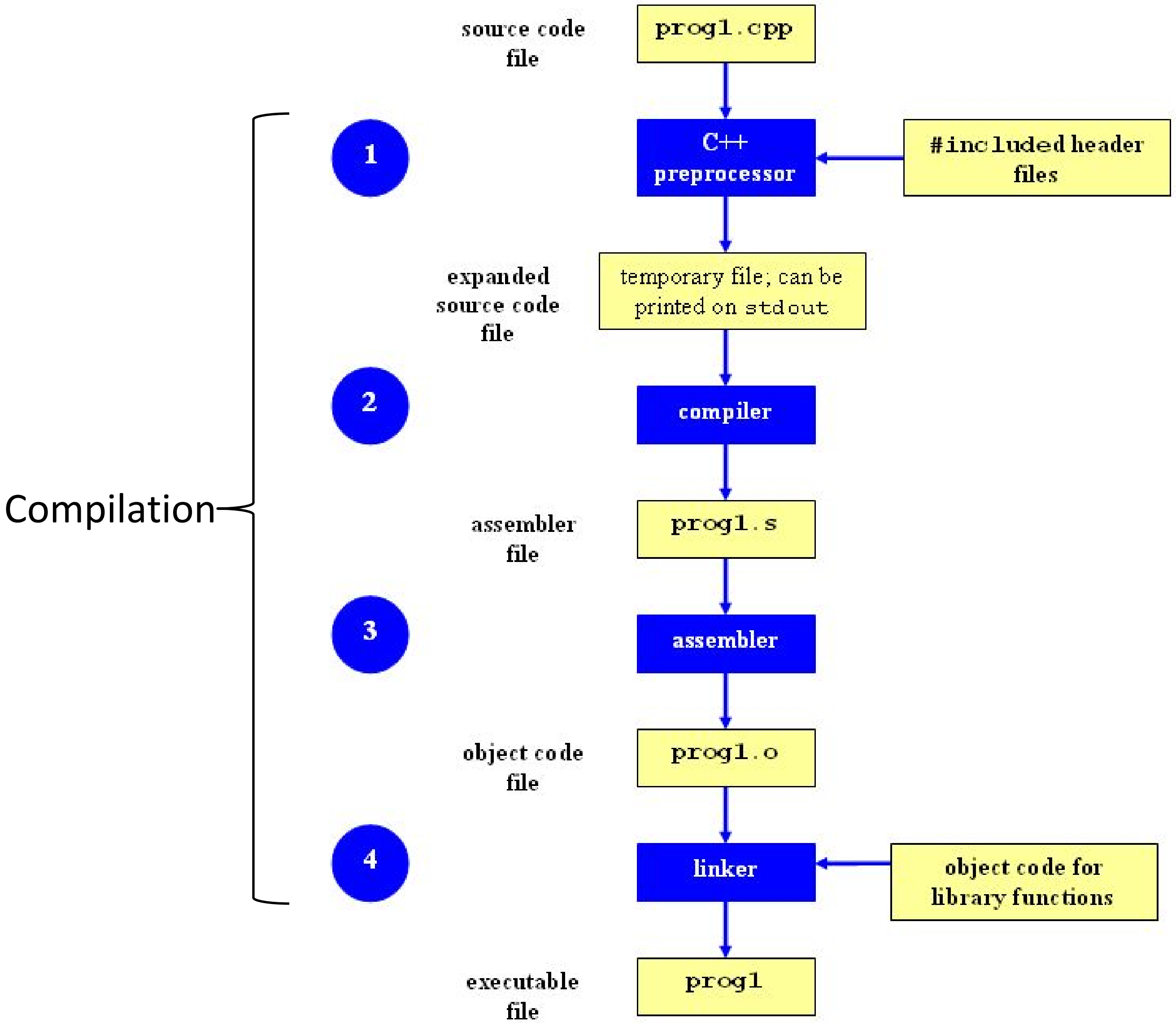
`tr`: translate **or** delete characters

- Usage: tr [OPTION]…SET1 [SET2]

You've implemented comm and tr by hand, do you remember how you did that?

Week 3

Compilation Process



source code file

prog1.cpp

1

C++ preprocessor

#included header files

expanded source code file

temporary file; can be printed on stdout

2

compiler

assembler file

prog1.s

3

assembler

object code file

prog1.o

4

linker

object code for library functions

executable file

prog1

Compilation

Compilation Process

- Why do we have this process?
- What are the different components of the process?
  - “I just typed gcc to compile my programs… does that mean gcc has all of the components within it?”
- Why can’t I execute individual object code files?
- What are the differences between open source and closed source software? When would I want to use one or the other?

Make

- Utility for managing large software projects
- Compiles files and keeps them up-to-date
- Efficient Compilation (only files that need to be recompiled)
- Why do we have make at all?
  - why don’t we just run ‘gcc …’ from the terminal

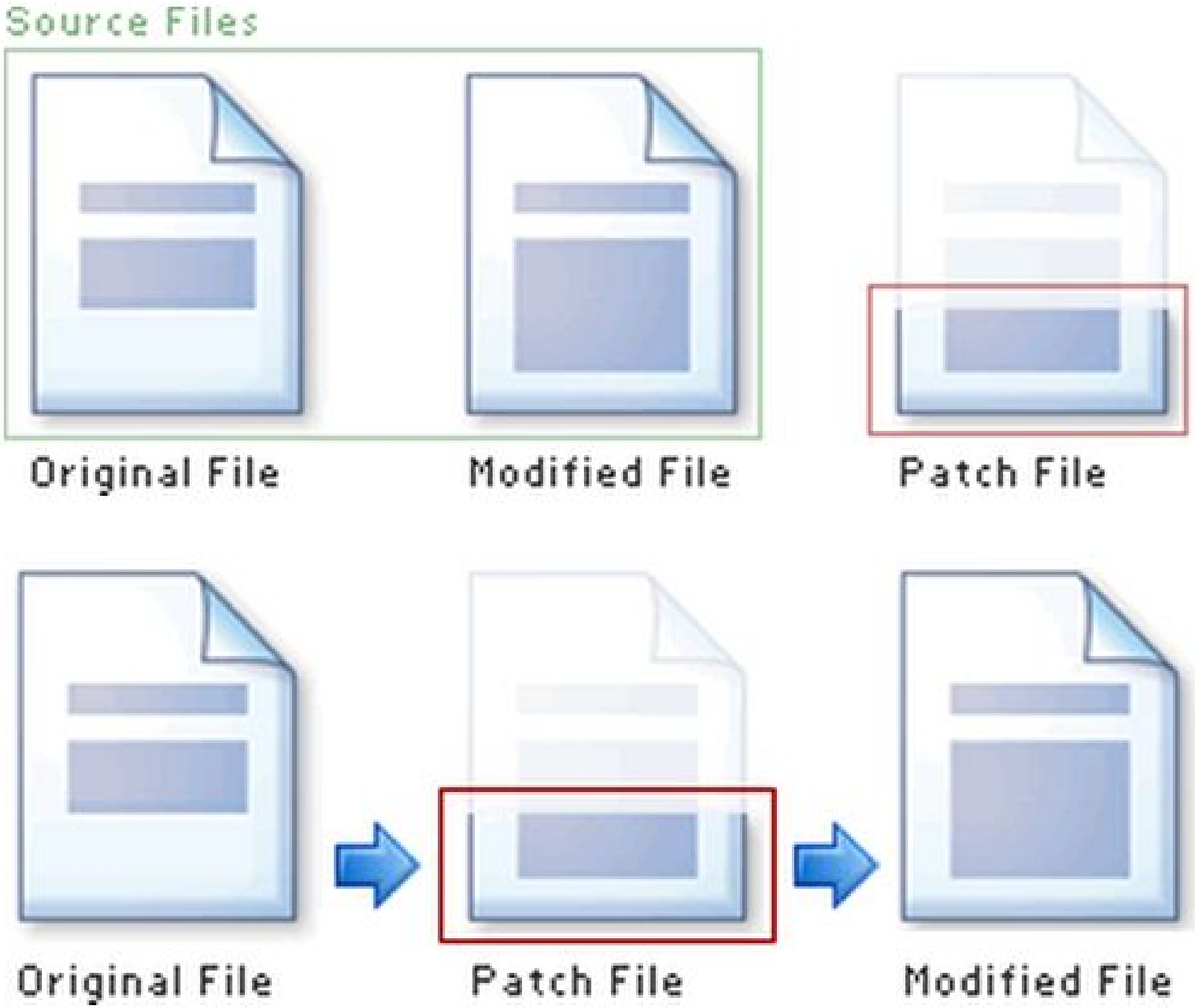
Build Process

- **configure**
  - Script that checks details about the machine before installation
    - Dependency between packages
  - Creates ‘Makefile’
- **make**
  - Requires ‘Makefile’ to run
  - Compiles all the program code and creates executables in current temporary directory
- **make install**
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - executables are copied into the final directories (system directories)

Patching

- A patch is a piece of software designed to fix problems with or update a computer program
- It’s a diff file that includes the changes made to a file
- A person who has the original (buggy) file can use the patch command with the diff file to add the changes to their original file
- Why not just change the original source code to fix it? Why do we have patches?

Applying a Patch



diff Unified Format

- diff –u original_file modified_file

- --- path/to/original_file
- +++ path/to/modified_file

- @@ -l,s +l,s @@
  - @@: beginning of a hunk
  - l: beginning line number
  - s: number of lines the change hunk applies to for each file
  - A line with a:
    - - sign was deleted from the original
    - + sign was added to the original
    - stayed the same

What is Python?

- Not just a scripting language
- Object-Oriented language
  - Classes
  - Member functions
- Compiled and interpreted
  - Python code is compiled to bytecode
  - Bytecode interpreted by Python interpreter
- Not as fast as C but easy to learn, read and use
- Why is python powerful? Why is it popular?
- You should know how to write basic programs in python

Comm.py

- Support all options for `comm`
  - -1, -2, -3 and combinations
  - Extra option –u for comparing unsorted files
- Support all type of arguments
  - File names and – for stdin
- Be familiar with how the linux `comm` utility works
- You should be able to run the comm utility by hand

Week 4

Software development process

- Involves making a lot of changes to code
  - New features added Bugs
  - fixed
  - Performance enhancements

- Software team has many people working on the same/different parts of code

- Many versions of software released
  - Ubuntu 10, Ubuntu 12, etc
  - Need to be able to fix bugs for Ubuntu 10 for customers using it, even though you have shipped Ubuntu 12.

Why do we have all of this?

Source/Version Control

- Track changes to code and other files related to the software
  - What new files were added? What
  - changes made to files?
  - Which version had what changes?
  - Which user made the changes?
- Track entire history of the software
- Version control software
  - GIT, Subversion, Perforce

This seems complicated. Why bother with source control?
What are the strengths and weaknesses of source control?
When would I want to use it? How do I use it?

Terms used

- **Repository**
  - Files and folder related to the software code
  - Full History of the software
- **Working copy**
  - Copy of software's files in the repository
- **Check-out**
  - To create a working copy of the repository
- **Check-in / Commit**
  - Write the changes made in the working copy to the repository
  - Commits are recorded by the VCS

Terms used

- Head
  - Refers to a commit object
  - There can be many heads in a repository
- HEAD
  - Refers to the currently active head
- Detached HEAD
  - If a commit is not pointed to by a branch
  - This is okay if you want to just take a look at the code and if you don't commit any new changes
  - If the new commits have to be preserved then a new branch has to be created
    - git checkout v3.0 -b BranchVersion3.1
- Branch
  - Refers to a head and its entire set of ancestor commits
- Master
  - Default branch

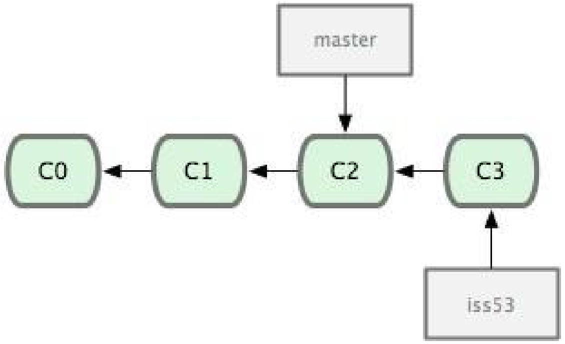


Image Source: git-scm.com

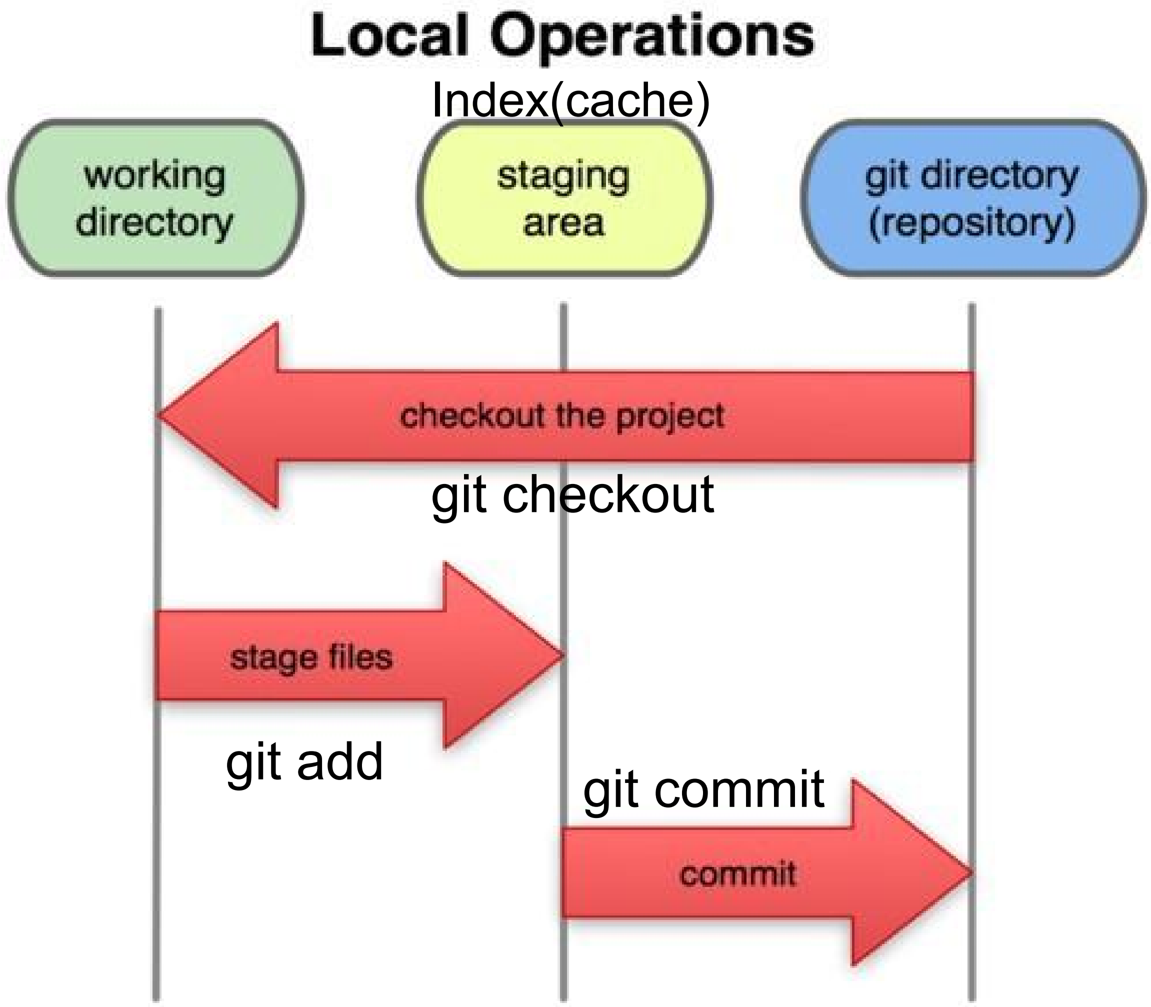
What Is a Branch?

- A pointer to one of the commits in the repo (head) + all ancestor commits
- When you first create a repo, are there any branches?
  - Default branch named ‘master’
- The default master branch
  - points to last commit made
  - moves forward automatically, every time you commit

Questions

- What is the difference between a working copy and the repository?
- What is a commit? What should be in a commit? How many files should commits contain?
- Why bother having branches at all? Why can’t we just all work on the same single master branch?
- What happens when we perform a merge? How does it work?

Git States



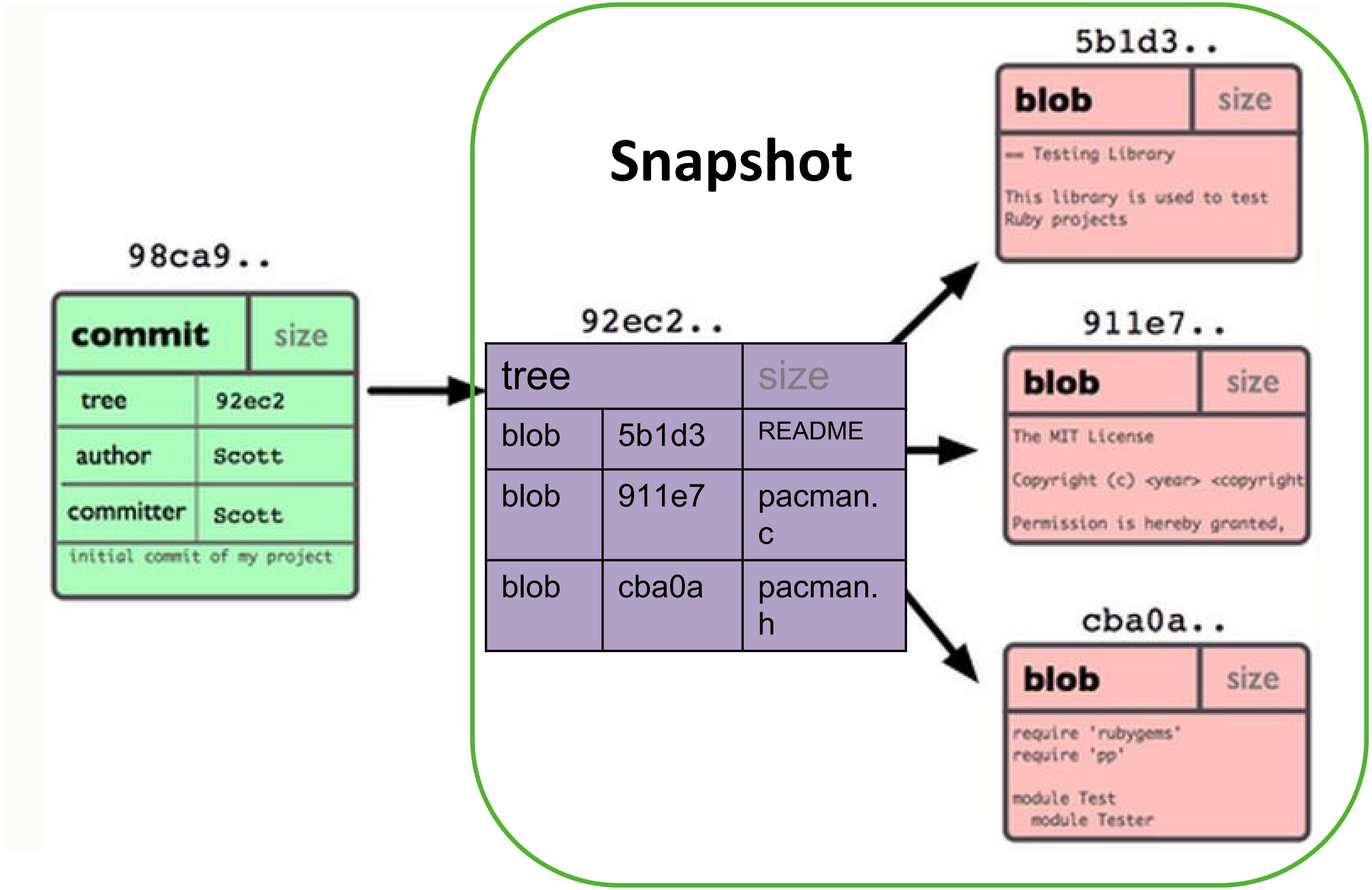
Index(cache)

git checkout

git add

git commit

Git Repo Structure



Snapshot

98ca9..

commit	size
tree	92ec2
author	Scott
committer	Scott

initial commit of my project

92ec2..

tree		size
blob	5b1d3	README
blob	911e7	pacman. c
blob	cba0a	pacman. h

5b1d3..

blob	size

```
== Testing Library

This library is used to test
Ruby projects
```

911e7..

blob	size

```
The MIT License

Copyright (c) <year> <copyright

Permission is hereby granted,
```

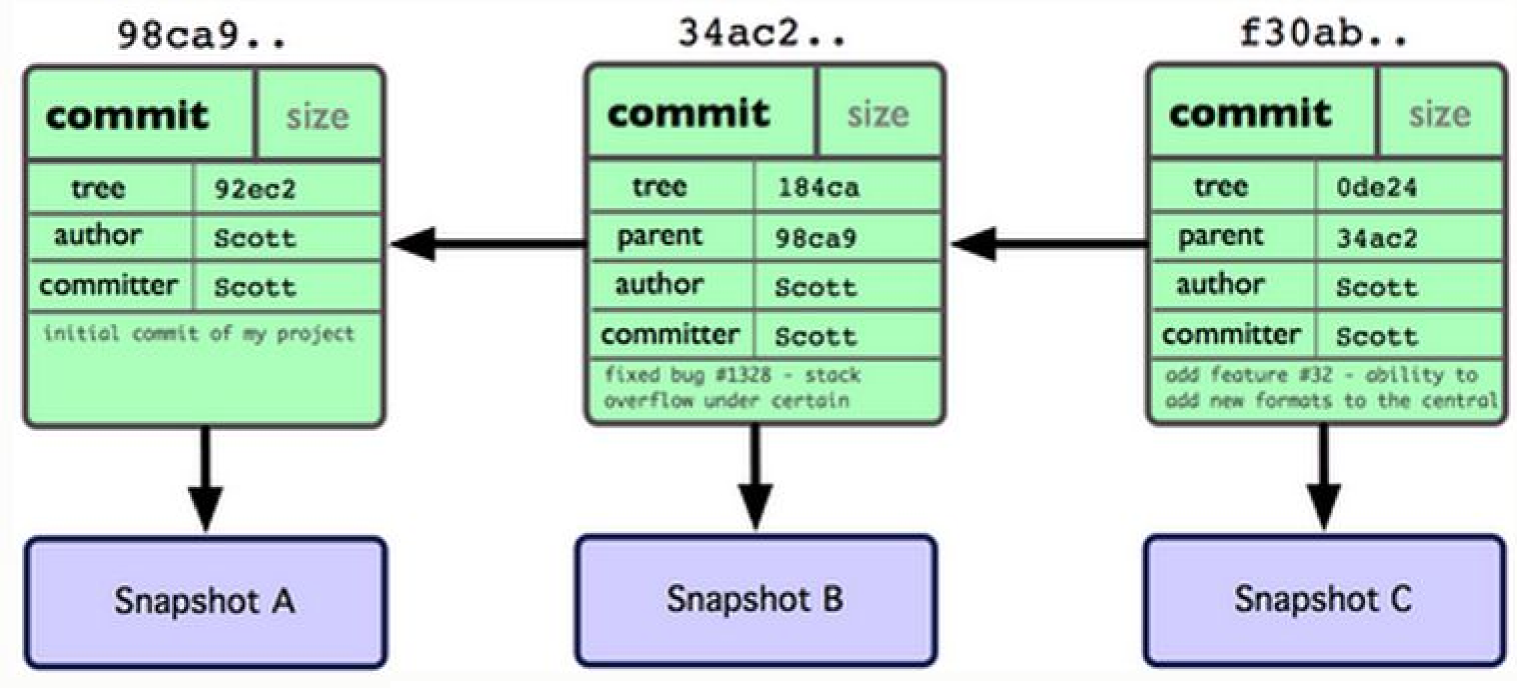
cba0a..

blob	size

```
require 'rubygems'
require 'pp'

module Test
  module Tester
```

After 2 More Commits…



Git commands

- Repository creation
  - $ git init (Start a new repository)
  - $ git clone (Create a copy of an exisiting repository)
- Branching
  - $ git checkout <tag/commit> -b <new_branch_name> (creates a new branch)
- Commits
  - $ git add (Stage modified/new files)
  - $ git commit (check-in the changes to the repository)
- Getting info
  - $ git status (Shows modified files, new files, etc)
  - $ git diff (compares working copy with staged files)
  - $ git log (Shows history of commits)
  - $ git show (Show a certain object in the repository)
- Getting help
  - $ git help

You should be familiar with how these commands work and when to use them.

More Git Commands

- Reverting
  - $ git checkout HEAD main.cpp
    - Gets the HEAD revision for the working copy
  - $ git checkout -- main.cpp
    - Reverts changes in the working directory
  - $ git revert
    - Reverting commits (this creates new commits)
- Cleaning up untracked files
  - $ git clean
- Tagging
  - Human readable pointers to specific commits
  - $ git tag -a v1.0 –m 'Version 1.0'
    - This will name the HEAD commit as v1.0

You should be familiar with how these commands
work and when to use them.

Sample Final Review

Week 10

Sample Final

Available here: goo.gl/kSdbH2