1. Course Introduction and Scope

Lecturer: Paul Eggert

1.1 What is Software Construction?

- CS 35L focuses on software construction: the practical art and science of building software that works, not just toy programs.
- Prerequisite: CS 31 (basic algorithms, data structures, C++). CS31 focused on sequential programs on a single core; this is now only a small part of modern software development.
- Modern software construction involves gluing together code, often written by others, and building on the shoulders of giants.
- The field of software construction is rapidly evolving, especially with the rise of machine learning and AI assistants.

Comparison of Key Terms

Term	Focus/Definition
Programming	Writing code to solve specific problems, often in isolation
Software Construction	Building robust, maintainable, and scalable software using best practices
Software Engineering	Systematic, disciplined approach to the design, development, and maintenance

1.2 The Changing Landscape of Software Development

- The "half-life" of CS knowledge used to be ~7 years, then stabilized at ~15 years, but is now shrinking again due to rapid advances (e.g., machine learning).
- You must expect to learn new technologies throughout your career.
- AI tools (e.g., ChatGPT, Copilot, Cloud Code) can help, but are not yet reliable enough for unsupervised use (current success rate ~75%).
- You must still know how to code and verify results manually; AI is a tool, not a replacement.

Examples of Rapidly Changing Technologies

Old Technology	New Technology/Trend
CVS, Subversion	Git, GitHub, GitLab
Make	CMake, Bazel, npm, Gradle
Perl, Shell scripts	Python, Node.js, Go
Manual deployment	Docker, Kubernetes, CI/CD
On-prem servers	Cloud (AWS, GCP, Azure)

1.3 Course Structure and Goals

- **Group Project:** Teams of ~5 will design and build a client-server application (e.g., a groundskeeping scheduler for UCLA).
- **Individual Assignments:** Six solo assignments covering a range of technologies and skills.
- **Technologies:** Node.js and React are recommended, but not required. Other technologies include Bash, Emacs, Make, Git, Python.
- **Learning Objective:** Rapidly learn and apply new software technologies; focus on practical, hands-on skills.

1.4 Core Topics Overview

1.4.1 File Systems

- Data and programs are stored in files, organized into directories.
- Linux and the POSIX model are used as the case study.
- Key concepts: file abstractions, storage structure, permissions, metadata.

Concept	Definition
File	Named data container in a filesystem
Directory	File that lists other files
POSIX	Portable Operating System Interface (standard)
Permission	Access rights (read, write, execute)
Metadata	Data about files (owner, size, timestamps)

Example: File Permissions

Symbol	Meaning
r	Read
w	Write
x	Execute

Sample Directory Tree

```
/home/user/
  |-- notes.txt
  |-- code/
      |-- main.c
      |-- utils.c
  |-- docs/
      |-- manual.md
```

1.4.2 Scripting

- Scripting: writing programs (scripts) to automate tasks, often with interpreted or command-based languages.
- Languages: Bash (shell scripting), Emacs Lisp, Python.
- Tools: Emacs (text editor/IDE), interactive shell operations.

Tool	Use Case
Bash	Automating build tasks, scripting tools
Emacs Lisp	Writing scripts in Emacs
Python	General-purpose scripting

Example: Bash Script

```
#!/bin/bash
for file in *.txt; do
  echo "Processing $file"
done
```

Example: Python Script

```
import os
for filename in os.listdir('.'):
    if filename.endswith('.txt'):
        print(f"Processing {filename}")
```

Language	Strengths	Weaknesses
Bash	Native to Unix, simple	Hard for complex logic
Python	Powerful, readable	Slower for shell tasks
Emacs Lisp	Extends Emacs	Niche, steep learning

1.4.3 Build and Distribution

- Building: converting source into executable programs.
- Distribution: installing programs for use on other systems.
- Tools: make, npm (for JavaScript projects).
- Challenges: consistency across systems, managing dependencies.

Build Process Flow

```
[Source Code] -> [Build Tool: make/npm] -> [Executable/Package] -> [Distribution/Install]
```

Build Tool	Language/Platform	Key Features
make	C/C++	Dependency tracking
npm	JavaScript/Node	Package management
CMake	C/C++	Cross-platform builds
Gradle	Java/Kotlin	Flexible scripting

1.4.4 Version Control

- Track and manage changes to code; support collaboration.
- Tool: Git (underlying technology for GitHub).
- Concepts: commit, branch, merge, tag.

Concept	Explanation
Commit	Set of changes saved to the repository
Branch	Separate line of development
Merge	Combining changes from different branches
Tag	Named commit, usually for release versions

Branching and Merging

```
main
  |
  |---- feature1
  |         \
  |----------> merge
  |
```

Command	Description
git init	Initialize a new repo
git add	Stage file for commit
git commit -m "msg"	Commit staged changes
git branch	Create a new branch
git checkout	Switch branches
git merge	Merge branch into current
git log	Show commit history

1.4.5 Low-Level Debugging and Dynamic Linking

- Focus: understanding compiled artifacts and their behavior.
- Tools: GDB (GNU Debugger), dynamic linkers.
- Slightly higher-level than CS33; emphasis on dynamic linking and fallback mechanisms.

Static vs. Dynamic Linking

```
[Program] --(static linking)--> [Executable with all code included]
[Program] --(dynamic linking)--> [Executable + Shared Libraries]
```

Example: Dynamic Linking

- When you run `ldd /usr/bin/less`, you see which shared libraries are loaded at runtime.

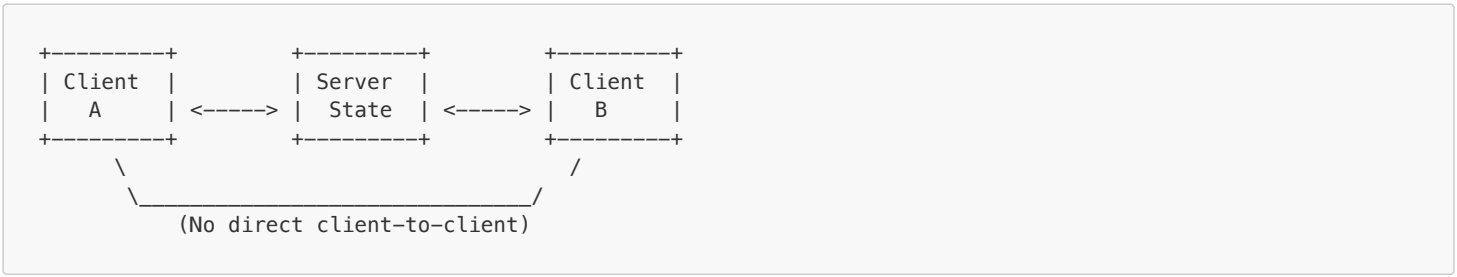
1.4.6 Client-Server Architecture

- Model: server holds shared state (e.g., database, schedules); clients communicate with the server, not directly with each other.
- Example: groundskeeping scheduler—server tracks all schedules, clients update/view schedules.

Client-Server Request/Response Cycle

```
Client ----request----> Server
         <---response---
```

ASCII Diagram: Client-Server Model



1.5 What is NOT Covered in CS35L (vs. CS130)

- CS130 (Software Engineering) covers:
  - Project scheduling
  - Large-scale integration
  - Thorough testing (theory and practice)
  - Project forensics (post-mortem analysis)
  - CI/CD (e.g., Docker, Kubernetes)
  - Large-scale deployment
- CS35L will only briefly touch on:
  - Integration
  - Configuration
  - Data management (schemas, ACID)
  - Security basics
  - Distributed architectures (very briefly)
  - Deployment considerations

Not Covered Fully	Mentioned Briefly
Scheduling Projects	Integration
Thorough Testing	Configuration
Project Forensics	Data Management
CI/CD (e.g., Docker)	Security Basics
Large-Scale Deployment	Networked Systems

1.6 Additional Topics and Student Input

- Brief coverage of:
  - Prompt engineering
  - Data management (schemas, ACID)
  - Security basics
  - Distributed architectures (1/8 of a lecture)
  - Deployment considerations
- Students encouraged to suggest topics for inclusion/exclusion.
- Scope management is a key project skill.

1.7 Mechanics and Logistics

- Course materials on BruinLearn and web.cs.ucla.edu.
- Piazza used for Q&A and participation (worth 1% of grade).
- Assignments due before 23:55 on due date; late penalty grows geometrically (1 day late = -1pt, 2 days = -3pt, 3 days = -9pt, etc.).
- Hard deadline: Friday of 10th week for all submissions.
- Grading breakdown:
  - Assignments: 18%
  - Class participation: 1%
  - Feedback surveys: 0.5%
  - Midterm: 18%
  - Final: 27%

    - Project: ~35%
- Academic integrity: work solo on assignments, do not share code, cite all sources (including AI tools and search queries used).
- Required to submit an appendix with logs of AI/search tool usage and a brief after-action report for each assignment.

1.8 Project and Assignment Details

- **Project:**
    - Group of ~5, build a client-server app (e.g., groundskeeping scheduler).
    - Must specify and understand the problem, write an informal specification, and implement using chosen technology (Node/React suggested, but not required).
    - Focus on learning new technology quickly and building a usable, demo-quality app.
    - **Best Practices:**
        - Communicate frequently with your team (use version control for all code).
        - Write clear documentation and keep your README up to date.
        - Use branches for features/bugfixes and merge regularly to avoid conflicts.
        - Test your code before merging to main.
        - Keep your project scope realistic and well-defined.
- **Assignments:**
    - Solo, spec-driven, submitted on BruinLearn.
    - Test cases provided; robust code required (no arbitrary limits like 255-byte strings).
    - Stick to provided code style in skeletons.
    - **Tips:**
        - Read the assignment spec carefully before starting.
        - Start early to allow time for debugging and resubmission.
        - Use the auto-grader feedback to improve your solution.
        - Log your use of AI/search tools as required.

1.9 Resources and Best Practices

- Written reports and oral presentations required for the project.
- Proper citation of sources is expected (use DOIs for scholarly papers).
- Review templates and rubrics for technical reports and presentations.
- Recommended reading: David Patterson's "How to Give a Bad Talk" (do the opposite!).

1.10 Introduction to Linux, Shell, and Emacs

- Linux servers (CSNET) run Red Hat Enterprise Linux 9.5 on Intel Xeon hardware.
- The shell (bash) is a command-line interpreter (CLI) that launches programs and scripts.
- The shell is a thin layer around the OS, designed to be transparent and simple.
- Programs (executables) are static files; processes are running instances of programs.
- Multiple processes can run from the same executable file.
- The shell itself is a program (e.g., bash, tcsh, zsh).
- The shell acts as a user interface to the OS, allowing users to:
    - Start and manage processes
    - Redirect input/output
    - Chain commands together (pipes)
    - Automate tasks with scripts

Common Shell Commands

Command	Purpose
ls	List files in a directory
cd	Change current directory
pwd	Print working directory
cp	Copy files or directories
mv	Move/rename files or directories
rm	Remove files or directories
cat	Concatenate and display file contents
grep	Search for patterns in files
chmod	Change file permissions

Command	Purpose
ps	Show running processes
kill	Terminate a process

- Process tree: systemd/init → shell (bash) → user commands (e.g., less, emacs).

ASCII Diagram: Process Tree

```
[systemd/init]
     |
   [bash]
     |
  [less] or [emacs]
```

Relationship: Applications, Libraries, System Calls, Kernel, Hardware

```
[Application]
     |
     v
[Library: libc, etc.]
     |
     v
[System Calls]
     |
     v
[Kernel]
     |
     v
[Hardware]
```

Example: File Read in C

```c
#include <stdio.h>
int main() {
    FILE *f = fopen("file.txt", "r"); // library call (libc)
    char buf[100];
    fread(buf, 1, 100, f);            // library call (libc)
    fclose(f);
    return 0;
}
```

- The `fopen`, `fread`, and `fclose` functions are library calls. Under the hood, they use system calls like `open`, `read`, and `close` to interact with the kernel.

Tracing Program Behavior

- `strace <command>`: Shows all system calls made by a program (e.g., file access, process creation).
- `ltrace <command>`: Shows library calls (e.g., `printf`, `malloc`).

Tool	What it Traces	Example Use Case
strace	System calls	Debug file/network access
ltrace	Library calls	Debug C library usage

- Emacs: powerful, extensible text editor (originally "editor macros").
- Emacs commands: ordinary characters insert text; control/meta characters invoke commands.
- Control characters: ASCII 0-31 (e.g., Ctrl-A = 1, Ctrl-X Ctrl-C = exit Emacs).
- Meta key: sets the top bit of the character (e.g., Meta-X for command mode).
- Emacs help: Ctrl-H for help, Ctrl-H K for help on a key sequence.
- ASCII character set: 0-127, divided into four 32-character regions (printable, control, etc.).
- Null byte (0) is a valid ASCII character (Ctrl-@).

ASCII Table Regions

Range	Description

Range	Description
0-31	Control characters
32-63	Printable (punct.)
64-95	Printable (letters)
96-127	Printable/special

Example Commands

Command	Action
ssh username@lnxsrv11.cs.ucla.edu	Log into CSNet
uname -r	Show kernel version
cat /proc/cpuinfo	View CPU info
lsb_release -a	Show system info
ldd /usr/bin/less	List libraries used by 'less'

1.11 Course Philosophy and Advice

- Focus on learning by doing; avoid inefficient learning (e.g., banging your head against the wall).
- Use office hours, discussion, and peer support.
- Assignments are designed to be challenging but fair; auto-grader allows resubmission.
- Don't cheat yourself by copying code—learn the material for your own benefit.
- If you use AI or search tools, log your prompts/queries and results, and reflect on their usefulness.
- **Practical Tips:**
  - When learning a new tool, always figure out how to exit it first (e.g., `Ctrl-X Ctrl-C` for Emacs, `:q` for Vim).
  - Keep a personal log of errors and solutions for future reference.
  - Practice using the shell and Emacs regularly to build muscle memory.
  - When stuck, break the problem into smaller parts and test incrementally.
  - Ask for help early—don't wait until the last minute.

1.12 First Steps in Linux and Shell

- Logging into CSNET, using SSH.
- Basic shell commands (e.g., `cat /proc/cpuinfo`, `ldd /usr/bin/less`).
- Understanding the relationship between hardware, kernel, libraries, and applications.
- System calls: special function calls that drop into the kernel (e.g., open, read, write, close). These are the only way user programs interact with the kernel and, by extension, the hardware.
- Libraries: C library (libc), C++ standard library, crypto libraries, etc. Libraries provide reusable code and often wrap system calls with higher-level abstractions.
- Applications use libraries, which in turn use system calls.
- Executable files vs. running processes.

Relationship: Applications, Libraries, System Calls, Kernel, Hardware

```
[Application]
     |
     v
[Library: libc, etc.]
     |
     v
[System Calls]
     |
     v
[Kernel]
     |
     v
[Hardware]
```

Example: File Read in C

```c
#include <stdio.h>
int main() {
    FILE *f = fopen("file.txt", "r"); // library call (libc)
    char buf[100];
    fread(buf, 1, 100, f);            // library call (libc)
    fclose(f);
    return 0;
}
```

- The `fopen`, `fread`, and `fclose` functions are library calls. Under the hood, they use system calls like `open`, `read`, and `close` to interact with the kernel.

Tracing Program Behavior

- `strace <command>`: Shows all system calls made by a program (e.g., file access, process creation).
- `ltrace <command>`: Shows library calls (e.g., `printf`, `malloc`).

Tool	What it Traces	Example Use Case
strace	System calls	Debug file/network access
ltrace	Library calls	Debug C library usage

Example Commands

Command	Action
ssh username@lnxsrv11.cs.ucla.edu	Log into CSNet
uname -r	Show kernel version
cat /proc/cpuinfo	View CPU info
lsb_release -a	Show system info
ldd /usr/bin/less	List libraries used by 'less'

1.13 Emacs and ASCII Details

- Emacs: text editor, also used for scripting (Emacs Lisp).
- ASCII: 7-bit character set (0-127), printable and control characters.
- Control characters: generated by clearing the second bit (e.g., Ctrl-A = 1).
- Meta key: sets the top bit (not standard ASCII, but used in Emacs).
- Emacs help system: Ctrl-H for help, Ctrl-H K for key sequence help.
- Exiting Emacs: Ctrl-X Ctrl-C or `M-x save-buffers-kill-terminal`.

More Emacs Shortcuts

Key Sequence	Operation
Ctrl-X Ctrl-C	Exit Emacs
Ctrl-H	Open help menu
Ctrl-H K	Show function of a keystroke
Meta-X	Run command by name
Ctrl-G	Cancel current command (keyboard-quit)
Ctrl-X Ctrl-S	Save current buffer
Ctrl-X Ctrl-F	Open file
Ctrl-X B	Switch buffer
Ctrl-S	Incremental search forward
Ctrl-R	Incremental search backward
Ctrl-A	Move to beginning of line
Ctrl-E	Move to end of line
Ctrl-K	Kill (cut) to end of line

Key Sequence	Operation
Ctrl-Y	Yank (paste)
Ctrl-/	Undo

1.14 Next Steps

- Next lecture: deeper dive into file systems and shell scripting.
- Continue learning by hands-on practice in Linux and Emacs.

1.15 Instructor Anecdotes, Warnings, and Subtle Points

- **AI Verification Anecdote:**
  - The instructor shared a story about using Gemini (an AI assistant) to answer a Python question for an exam. The AI gave a plausible but incorrect answer, which was only revealed by running the code. This underscores the importance of always verifying AI-generated code and not trusting it blindly.
  - AI tools can be helpful, but their answers may look correct while being wrong. Always test and verify.
- **AI and Job Security:**
  - Current AI assistants are not reliable enough to automate yourself out of a job. Their error rate is too high for production use. You must know how to do things by hand and critically evaluate AI output.
- **Programs vs. Processes:**
  - A program (executable) is a static file; a process is a running instance of a program. Multiple processes can run from the same executable. This distinction is fundamental in Linux/Unix systems.
- **Learning to Exit Programs:**
  - The most important thing to learn about any new program is how to exit it. For Emacs, this is `Ctrl-X Ctrl-C`. This is a practical tip for all new tools and environments.
- **Course Philosophy:**
  - The course is designed to be fun and practical, focusing on building real, usable software and learning by doing. Students are encouraged to ask questions, use resources, and avoid inefficient solo struggles.
