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

```
+---------+         +---------+         +---------+
| Client  |         | Server  |         | Client  |
|   A     | <-----> |  State  | <-----> |   B     |
+---------+         +---------+         +---------+
      \                                   /
       \_________________________________/
           (No direct client-to-client)
```

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.

Emacs, Shells, and Linux File Systems

1. Overview

In modern software development, users often deal with a triad of interrelated systems: scripting environments (such as shells), Integrated Development Environments (IDEs) like Emacs, and file systems (e.g., Linux file systems). This lecture takes a holistic, realistic approach to learning these tools simultaneously to mirror real-world development environments. Instead of isolating topics, systems are introduced in the overlapping, interdependent way in which they were historically developed and are actually used.

> **Realism in Learning:** The lecture intentionally blends topics, reflecting how technologies were co-developed and are used together in practice. This mirrors the real-world experience of being "thrown into" multiple new systems at once, as in industry.

---

2. Emacs as an IDE

History and Purpose

- The instructor originally used Vim/VI, but switched to Emacs for its programmability and extensibility.
- Emacs embeds a programming language (Emacs Lisp), enabling rapid development of IDE-like tools and customizations.
- Emacs is programmable, extensible, and scriptable, making it suitable for heavy development work and building custom environments.

Emacs Features

Emacs is known for:

- Deep programmability
- Visual frame and window system
- Modality through Modes ("modeful" editor)
- Internal shell invocation
- Editing many kinds of resources (files, directories, buffers)

Buffers vs. Files

- **Buffer:** A temporary, in-memory representation of data (text, code, etc.). Fast, but not persistent.
- **File:** Stored on disk, persistent across system reboots.

Key distinction:

- Opening a file loads it into a buffer.
- Changes are made to the buffer, not directly to the file.
- Save operation (`Ctrl-X Ctrl-S`) writes buffer content to the file.

Use case:

```
Ctrl-X Ctrl-F hello.txt       # Open or create file in buffer
Editing in buffer             # Does not affect file
Ctrl-X Ctrl-S                 # Save buffer to file
```

Buffer Management

- Emacs can have many buffers open at once (files, directories, shell sessions, etc.).
- Switch buffers: `Ctrl-X b` (prompt for buffer name), `Ctrl-X Ctrl-B` (list all buffers).
- Kill buffer: `Ctrl-X k` (prompt for buffer to close).

ASCII Diagram: Buffers and Files

```
[File on Disk] <---save/load---> [Buffer in RAM] <---view---> [Window in Frame]
```

The Problem of Persistence

- Applications should survive power outages (persistence), be fast, and be understandable.
- Naive persistent variables (e.g., writing to storage on every change) are too slow due to I/O latency (milliseconds vs. nanoseconds).

Example:

```
persistent int numberOfStudents;
numberOfStudents++;
```

This would require saving to durable storage at every write, which is slow.

Emacs' Approach to Persistence

Emacs balances speed and persistence by:

1. Keeping data in buffers (fast, in-memory)
2. Periodically auto-saving to temp files (e.g., `#hello.txt#`)
3. Creating symbolic links for lock/metadata (e.g., `.hello.txt`)
4. Only writing to the actual file on explicit save (`Ctrl-X Ctrl-S`)

Auto-save, Backup, and Lock Files:

- `#filename#`: Auto-save file, periodically updated (for crash recovery).
- `filename~`: Backup file, created on first save (previous version, for undoing mistakes).
- `.filename`: Symbolic link containing metadata (user, host, Emacs PID, system boot time) for lock detection.

Crash Recovery Example:
If Emacs crashes, on restart it will detect `#filename#` and offer to recover unsaved changes. If a lock file (`.filename`) exists but no Emacs process is editing, it warns of a stale lock.

Table: Emacs Save/Backup/Lock Files

File Type	Example	Purpose
Auto-save	`#foo.txt#`	Crash recovery (periodic save)
Backup	`foo.txt~`	Previous version (undo)
Lock (symlink)	`.foo.txt`	Prevent concurrent edits

Emacs Modes

- **Major Modes:** Define the main editing mode for a buffer (e.g., Text, Python, Dired, Shell). Only one major mode per buffer.
- **Minor Modes:** Add optional features (e.g., line numbers, auto-fill, flyspell). Multiple minor modes can be active.
- Mode line (bottom of window) shows active modes.

Table: Major vs. Minor Modes

Type	Example	Description
Major Mode	`python-mode`	Main editing mode (syntax, keys)
Minor Mode	`flyspell-mode`	Optional feature (spellcheck)

Windows and Frames

- **Window:** A viewport into a buffer within a frame (Emacs terminology).
- **Frame:** What most OSes call a "window"; can contain multiple Emacs windows.
- Multiple windows can show different buffers side-by-side.

Command	Action
Ctrl+X 4 D	Open Dired in new window
Ctrl+X O	Move between windows
Ctrl+X 1	Focus single window

Dired Mode (Directory Editor)

- Dired is a mode for editing and navigating directories.
- Shows a textual view of directory contents.
- Limited editing (e.g., flagging for deletion, renaming).
- Changes are driven back to the filesystem.

Key	Action
d	Mark file for deletion

Key	Action
x	Execute marked delete actions
u	Unmark

- Advanced features:
  - Batch rename: `R` (rename multiple files)
  - Mark files by regex: `% m` (mark by pattern)
  - Run shell commands on files: `!` (run command on marked files)
  - Copy (`C`), move (`R`), create symlinks (`S`), compress (`Z`), etc.

Example: Batch Rename

```
Mark files with `m` or `% m`, then press `R` to rename all at once.
```

Shell Access Inside Emacs

- `M-x shell`: Opens a shell inside Emacs.
- Multiple shells: `M-x shell` creates a new shell buffer each time.
- Shell buffers can be renamed for organization (`M-x rename-buffer`).

Emacs Lisp Example

Emacs is programmable via Emacs Lisp. Example: define a command to insert the current date.

```
(defun insert-date ()
  "Insert the current date at point."
  (interactive)
  (insert (format-time-string "%Y-%m-%d")))
;; Bind to key: Ctrl-c d
(global-set-key (kbd "C-c d") 'insert-date)
```

Emacs Save Files and Metadata

- Emacs creates temporary metadata files for buffers:
  1. `#filename#`: Regular auto-save file (periodically updated)
  2. `.filename~`: Backup file (not discussed in detail)
  3. `.filename.swp`: May appear in other editors (not Emacs)
  4. `.filename` symlink: Contains editor and session metadata (username, hostname, Emacs PID, system boot timestamp)
- Symbolic link is used as a lock and to warn about concurrent edits (dangling symlink if no process is editing).
- If Emacs crashes, these files help recover work and warn about stale locks.

Example:

```
ls -l
#hello.txt#    # Auto-save buffer
.hello.txt     # Symbolic link containing editing info
hello.txt      # Main file
```

---

3. Linux File System Concepts

Files and File Types

- Files are objects storing a finite sequence of bytes.
- Associated with metadata (permissions, size, timestamps).
- Common types:

Symbol	Type
-	Regular file
d	Directory
l	Symbolic link

Symbol	Type
b	Block device
c	Char device
p	FIFO/pipe
s	Socket

File Metadata (shown with `ls -l`)

Example:

```
-rwxr-xr-- 1 egert egert 1234 Apr 5 10:00 hello.txt
```

Component	Meaning
-rwxr-xr--	Permissions (user/group/other)
1	Link count
egert egert	Owner and Group
1234	Size in bytes
Apr 5 10:00	Last modified time
hello.txt	File name

File Permissions

- 9-bit mask: 3 groups (user/owner, group, others), each with read (r), write (w), execute (x).
- Use `id` command to check current user and group.
- Each file has an owner and a group field.

Section	Symbol	Octal	Role
User	rwx	7	Owner permissions
Group	r-x	5	Group permissions
Other	r--	4	Others' permissions

Common Permission Formats:

Symbolic	Octal	Meaning
rwxr--r--	744	Owner can read/write/execute
rw-r--r--	644	Owner can read/write, others read
rwxrwxr-x	775	Group shares full rights

Special Permission Bits

- Linux includes 12 permission bits: 9 common bits (3x rwx for user/group/others), 3 special bits:

Name	Symbol	Effect
setuid	s	Run file as file's owner
setgid	s	Run as file's group
sticky	t	Applies to directories — restricts deletion

Example:

```
-rwsr-xr-- 1 root root 1234 su       # setuid (run as root)
drwxrwxrwt 10 root root 4096 /tmp   # sticky bit (shared dir)
```

- The sticky bit (t) on directories means only the file's owner can delete/rename files within.
- setuid (s) means the program runs as the file's owner (e.g., `su` runs as root).

- setgid (s) means the program runs as the file's group.

Changing Permissions

- `chmod` changes file permissions.
- Syntax:

```
chmod 644 file.txt
chmod +x script.sh
```

- Only the file's owner (or root) can change permissions.

Symbolic and Hard Links

- **Symbolic Links (symlinks):**
  - Point to a path (text string)
  - Created with: `ln -s target linkname`
  - Can be "dangling" (point to non-existent file)
  - Used by Emacs for lock/metadata files
  - Symlinks are like pointers (textual), followed by splicing the path
- **Hard Links:**
  - Share inode (data) and increase link count
  - Created with: `ln file1 file2`
  - Both names refer to the same data (same inode)
  - Cannot hard-link directories (except for `.` and `..` by convention)

Hard Link Example:

```
touch foo
ln foo bar        # bar is another name for foo
```

Inode Confirmation with `ls -i`:

```
123456 foo
123456 bar       # same inode = same file
```

- Use `rm` to remove a name; file persists until all links are removed.
- Link count > 1 means hard link(s) exist.

Table: Hard vs. Symbolic Links

Feature	Hard Link	Symbolic Link
Points to	Inode (data)	Path (name)
Link count	Increases	No effect
Cross-filesystem	No	Yes
Can link dirs	No (except . ..)	Yes (with caution)
Survives target	Yes	No (dangling link)
`ls -l` shows	Same inode	Arrow (`->`)

File Deletion with Hard Links

- Deleting a file name with `rm` only removes that name.
- File data is deleted **only** when all hard links are removed (link count = 0).

Example:

```
touch foo
ln foo bar
rm foo      # bar still exists, data not deleted
rm bar      # now data is deleted
```

Directories

- Directories are files that map names to inodes.
- Always contain `.` (self) and `..` (parent).
- Directories start with link count = 2:
  - One for itself (`.`)
  - One from parent (`dirname`)
- Each subdirectory adds 1 to parent's link count (from its `..`).
  - `..` from `/` (root) points to itself.
  - Root has link count matching number of subdirectories + 1.

```
mkdir dir
ls -ld dir        # shows link count 2
mkdir dir/subdir
ls -ld dir        # link count now 3
```

ASCII Diagram: Directory Link Count Changes

```
mkdir dir
ls -ld dir   # link count = 2 (dir itself + parent)

mkdir dir/subdir
ls -ld dir   # link count = 3 (dir itself + parent + subdir)

# General rule:
# link count = 2 + number of subdirectories
```

Tree-Structured File System

- Root directory: `/`
- Files organized in hierarchy beneath root
- Paths:
  - Absolute: `/home/user/file.txt`
  - Relative: `./file.txt` or `../file.txt`

ASCII Diagram:

```
/
├── home/
│   └── egert/
│       ├── foo
│       └── bar
├── usr/
│   └── bin/
└── tmp/
```

Inode Numbers

- Every file has a unique inode number.
- Use `ls -i` to view.
- Link count: Number of names pointing to an inode.
- Inode numbers are like addresses for files, but you cannot access a file by inode alone—names are required.

---

4. Shell Commands and Utilities

These utilities interact with the file system and processes.

Basic Shell Overview

- Shells provide process management, file commands, and scripting features.
- Emacs can embed a shell as a buffer (`M-x shell`).
- Shells can run scripts: a file with commands, executed with `sh script.sh` or by making it executable (`chmod +x script.sh`).
- Scripts can use variables, loops, and conditionals.

Example: Simple Shell Script

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

Control Characters

Key	Meaning
Ctrl+D	End-of-file/input (EOF)
Ctrl+C	Interrupt (stop process)
Ctrl+G	Emacs interrupt (abort command)
Ctrl+@ / Ctrl+Space	Set mark for copy
Meta+W	Copy region (kill-ring)
Ctrl+Y	Paste ("yank")

Redirection and Pipes

- `>`: Redirect output to file (`ls > out.txt`)
- `>>`: Append output to file (`echo hi >> out.txt`)
- `<`: Read input from file (`cat < in.txt`)
- `|`: Pipe output to another command (`ls | grep foo`)

Example: Combine Commands

```
cat file.txt | grep hello | wc -l > count.txt
```

Process Management

- Every process has a parent (except `init`/`systemd`).
- `ps -ef` shows parent PID (PPID).
- `kill <PID>` sends a signal to a process (default: SIGTERM).
- `kill -9 <PID>` forcefully kills (SIGKILL).
- `jobs`, `bg`, `fg` manage background/foreground jobs in shell.

ASCII Diagram: Process Tree

```
1 (init)
 |
 +-- 22588 (emacs)
      |
      +-- 22601 (bash)
            |
            +-- 22610 (cat)
```

cat

- `cat file`: Outputs content of a file.
- `cat`: With no args, reads from keyboard and echoes input.
- Ctrl+D at start of line signals EOF.
- `cat file1 file2`: Concatenates and prints both files.

ps

- Shows process status
- `ps -ef`: Lists all processes with full info
- Process ID (PID) is used in Emacs symbolic links for lock/metadata

Sample Output:

```
UID   PID   PPID  CMD
egert 22588 1     emacs
```

chmod

- Changes file permissions
- Syntax:

```
chmod 644 file.txt
chmod +x script.sh
```

ln

- Creates links:
  - `ln file1 file2`: Hard link
  - `ln -s path link`: Symbolic link
- Cannot hard-link directories (except for `.` and `..` by convention)
- Emacs uses symbolic links to indicate lock/status metadata for edited files

touch, rm, and truncate

- `touch`: Creates empty file or updates timestamp
- `rm`: Removes file name (not file unless final link)
- `truncate`: Sets file size without writing content

Example:

```
truncate -s 1T bigfile   # Create 1 terabyte sparse file
ls -lh bigfile           # Shows size
cat bigfile              # Outputs null bytes (if not crashed)
```

- Modern filesystems may use sparse files, compression, or encryption, so file size may not reflect actual disk usage.

---

5. Summary

This lecture introduced three core interconnected systems in a Unix/Linux environment: scripting via shell, file systems, and editing environments like Emacs. Emphasis was placed on realism through simultaneous learning and usage reflective of real-world software engineering scenarios. Emacs was covered in depth with its concept of buffers, windows, modes, saving mechanics, and integration with directories and the shell. File system fundamentals such as inode numbers, link counts, file permissions, symbolic vs hard links, and hierarchical structure were explored. Command-line utilities like `cat`, `ls`, `chmod`, `ps`, and `ln` helped demonstrate these principles. The engineer's need to balance persistence, performance, and understandability in system architecture was repeatedly highlighted.

File Persistence in Emacs

ASCII Flowchart: Emacs File Save Lifecycle

```
[Buffer in RAM]
     |
     v
[Auto-save file (#filename#)]
     |
     v
[Lock file (.filename symlink)]
     |
     v
[Manual Save (Ctrl-X Ctrl-S)]
     |
     v
[File on Disk]
```

Windows and Frames

ASCII Diagram: Emacs Frame, Windows, and Buffers

```
+-----------------------------+   Frame (OS window)
| +-----------+ +-----------+ |
```

```
| |  Window   | |  Window   | |
| |  (view 1) | |  (view 2) | |
| +-----------+ +-----------+ |
+-----------------------------+
      |              |
      v              v
   Buffer A       Buffer B
```

Directories

ASCII Diagram: Directory Link Count Changes

```
mkdir dir
ls -ld dir   # link count = 2 (dir itself + parent)

mkdir dir/subdir
ls -ld dir   # link count = 3 (dir itself + parent + subdir)

# General rule:
# link count = 2 + number of subdirectories
```

---

6. Additional Resources

- Emacs Documentation
- Linux File System Basics
- Shell Scripting Tutorial

---

7. Conclusion

This lecture provided a comprehensive overview of Emacs, Linux file systems, and shell scripting. It highlighted the importance of understanding file systems and shell commands in a Unix/Linux environment. The engineer's need to balance persistence, performance, and understandability in system architecture was emphasized throughout the lecture.

Filesystems, Bash, and Scripting

1. File Systems and Links

1.1. File System Overview

The Unix file system is a hierarchical tree, with the root directory `/` at its base. Each entry (file, directory, or link) is uniquely identified by an **inode number**.

- **Root Directory (`/`)**: The entry point to the file system.
- **Subdirectories**: Standard directories include `/bin`, `/usr`, `/home`, etc.
- **Types of Entries**:
  - Regular files (e.g., `.gitignore`)
  - Directories
  - Symbolic links (symlinks)

Table: Common Unix Directories

Directory	Purpose
`/bin`	Essential user binaries
`/usr`	User programs, libraries
`/home`	User home directories
`/etc`	System configuration files
`/tmp`	Temporary files
`/var`	Variable data (logs, spool)
`/dev`	Device files
`/proc`	Process and kernel info (virtual)

Inodes:

- Each file or directory is represented by an inode, which stores metadata (owner, permissions, timestamps, file size, pointers to data blocks) but not the filename.
- Directories are special files that map filenames to inode numbers.

ASCII Diagram of a Simple Filesystem:

```
/
├── bin
│   └── sh
├── usr
│   └── bin
├── home
│   └── user
│       └── .gitignore
```

Each file or directory is identified by its inode number. You can view inode numbers with `ls -i` or `ls -li`.

Example:

```
ls -li /bin/sh
1234567 lrwxrwxrwx 1 root root 4 Jan 1 00:00 /bin/sh -> bash
```

Where `1234567` is the inode number.

1.2. Hard Links

- **Definition**: Multiple directory entries (names) that point to the same inode (file content).
- **Key Facts**:
  - Hard links share the same inode.
  - Cannot create hard links to directories (to avoid cycles and filesystem corruption).
  - Changes via one hard link are visible through all others.

- Removing one hard link does not delete the file until all links are removed (link count reaches zero).
- Hard links cannot span filesystems (must be on the same device).

Diagram: Hard Links

```
Directory A:           Directory B:
file1  ----+           file2  ----+
           |                      |
        [inode 97311: file data]  |
           +----------------------+
```

Example:

```
ln file1 file2
```

Name	Inode
file1	97311
file2	97311

Why no hard links to directories?

- Allowing hard links to directories could create cycles, making the filesystem a graph instead of a tree, which breaks tools like `fsck` and makes traversals ambiguous.

Effect on Deletion:

- The file's data is only deleted when the last hard link is removed (link count = 0).
- Disk usage is not freed until all hard links are gone.

1.3. Symbolic (Soft) Links

- **Definition**: A symlink is a special file containing a path (string) to another file or directory.
- **Key Facts**:
  - Symlinks have their own inode and metadata.
  - Can point to non-existent targets (dangling symlinks).
  - Can be relative or absolute.
  - Symlinks to directories are allowed.
  - Symlinks can cross filesystem boundaries.

Command:

```
ln -s target symlink
```

Type	Starts with	Interpreted From
Relative	No `/`	Directory containing link
Absolute	`/`	Root `/`

Examples:

```
ln -s /bin/sh mysh   # Absolute
ln -s ../bin/sh relsh  # Relative
```

Table: Hard Links vs Symbolic Links

Feature	Hard Link	Symbolic Link
Inode	Same as target	Own inode
Can cross FS?	No	Yes
Can link dirs?	No	Yes
Dangling?	No	Yes

Feature	Hard Link	Symbolic Link
Link count	Increases target	Own link count
Affected by mv?	No (same inode)	Yes (may dangle)

ASCII Diagram: Hard vs Symbolic Links

```
# Hard Link
/home/user/file1  ----+         (inode 97311)
/home/user/file2  ----+

# Symbolic Link
/home/user/file1  (inode 97311)
/home/user/link1  (inode 97312, contents: 'file1')
```

Why Two Kinds of Links?

- Hard links: For multiple names to the same file (safe renaming, backup, etc.).
- Symbolic links: For linking to directories, creating links to files that may not exist yet, and for more flexible naming (including dangling links).

Restrictions

- **No hard links to directories** (to prevent cycles and confusion in the filesystem).
- **Symlinks can point to directories or non-existent files**.

Symlink and File Operations:

- `rm symlink` removes the link, not the target.
- `rm file` removes the file (if no other hard links).
- `mv symlink newname` renames the link, not the target.
- `cp symlink newcopy` (with `-P` or `-L`) can copy the link or the target, depending on options.

---

2. Pathname Resolution

2.1. Absolute vs. Relative Paths

- **Absolute**: Starts with `/`, resolution begins at root.
- **Relative**: Does not start with `/`, resolution begins at current working directory (CWD).
- Each process has its own CWD, which can be changed with `cd`.

Example:

```
cd dirname
```

2.2. Name Resolution Algorithm (namei)

- Start at `/` if absolute, else at CWD.
- For each path component:
  - If it's a directory, move into it.
  - If it's a symlink, substitute the symlink's contents into the path:
    - If the symlink is **relative**, interpret from the directory containing the link.
    - If **absolute**, restart from root.
- Continue until the final component is resolved to an inode.
- If a symlink is encountered, the system increments a symlink traversal counter; if the limit (e.g., 20) is exceeded, resolution fails with an error.

ASCII Flowchart: Pathname Resolution

```
START
  |
  v
Is path absolute?
  |--Yes--> Start at /
  |--No-->  Start at CWD
  |
  v
```

```
For each component:
  |
  v
Is component a symlink?
  |--No--> Move into directory
  |--Yes-->
      Is symlink absolute?
        |--Yes--> Restart from /
        |--No-->  Substitute relative path
  |
  v
Repeat until end or symlink limit reached
```

Example: Complex Path with Symlinks
Suppose:

- `/home/user/docs` is a symlink to `/mnt/storage/docs`
- `/mnt/storage/docs/report` is a symlink to `../archive/report`

Resolving `/home/user/docs/report`:

1. `/home/user` (directory)
2. `docs` (symlink) → `/mnt/storage/docs`
3. `/mnt/storage/docs` (directory)
4. `report` (symlink) → `../archive/report`
5. `/mnt/storage/archive/report` (final target)

2.3. Symlink Loops and Dangling Links

- **Dangling symlink**: Points to a non-existent target.

  ```
  ln -s nowhere foo
  cat foo  # No such file or directory
  ```

- **Symlink loop**: Two or more symlinks refer to each other, causing infinite resolution.

  ```
  ln -s B A
  ln -s A B
  cat A  # Fails due to loop
  ```

- **System limit**: To prevent infinite loops, the system limits symlink traversals (e.g., 20 in Linux). Exceeding this gives an error: `Too many levels of symbolic links`.

2.4. Hard Links to Symlinks

- You can create hard links to a symlink file itself (not to its target).
- All hard links to a symlink share the same inode and link count.
- Removing one hard link does not delete the symlink if others remain.

2.5. Relative Symlinks and Directory Context

- Relative symlinks are interpreted from the directory containing the symlink, not necessarily from where you access it.
- This can cause a symlink to work in one context but not another.

Example:

```
mkdir e
ln foo e/.baz
```

- `e/.baz` may resolve correctly if accessed from `e/`, but not from the parent directory.

---

3. The Shell

3.1. Purpose and Philosophy

- The shell is a simple scripting language for launching/configuring other programs.

- Its main job: set up the environment (I/O, variables, etc.) and run programs.
- Not intended for complex computation (use Python/C for that).
- **Types of shells:** Common ones include `sh`, `bash`, `zsh`, `csh`, `ksh`. Each has its own features and syntax quirks.

Table: Shell Scripting vs. Other Scripting Languages

Feature	Shell Script	Python	Perl
Syntax	Minimal	Rich	Rich
System control	Excellent	Good	Good
Text processing	Basic	Excellent	Excellent
Portability	High	Medium	Medium
Computation	Weak	Strong	Strong

3.2. Command Execution and Standard I/O

- **Redirection**:
  - `>`: Redirect stdout
  - `<`: Redirect stdin
  - `2>`: Redirect stderr
- **Pipes**: `|` connects stdout of one command to stdin of the next.

File Descriptors:

Name	Number	Default
stdin	0	Keyboard
stdout	1	Screen
stderr	2	Screen

Diagram: Pipe Connecting Processes

```
[cmd1] --stdout--> [PIPE] --stdin--> [cmd2]
```

Example:

```
cat file > output.txt 2> errors.log
cmd1 | cmd2 | cmd3
```

3.3. Argument Passing

- Arguments are passed as strings to the program's `main` function.
- `argv[0]` is the command name, `argv[1]` is the first argument, etc.

3.4. Tokenization and Quoting

- The shell splits input into tokens (words) using whitespace and special characters.
- **Special characters**:
  - Whitespace (space, tab, newline)
  - `<`, `>`, `|`, `&`, `;`, `#`, `$`, `` ` ``, `(`, `)`
- **Quoting**:
  - `\char`: Escape a single character
  - `'text'`: Single quotes, literal string (no expansion)
  - `"text"`: Double quotes, allows variable/command expansion

Table: Quoting Effects

Syntax	Variable Expansion	Command Substitution	Field Splitting
'text'	No	No	No
"text"	Yes	Yes	No
unquoted	Yes	Yes	Yes

Examples:

```
echo 'This is $HOME'  # Literal
```

```
echo "This is $HOME"  # Expands $HOME
```

- Newlines can be included in double or single quoted strings.
- **Null bytes**: The shell does not allow embedded null bytes in variables or strings.

Subtle Quoting Points

- To include a single quote inside a single-quoted string, close the quote, escape the quote, and reopen:

  ```
  echo 'Three o'\''clock'
  ```

- Quoting affects argument splitting: `cat $v` vs `cat "$v"` (unquoted may split on spaces, quoted preserves as one argument).

---

4. Reserved Words and Control Structures

4.1. Reserved Words

- Only reserved at the start of a command.
- Examples: `if`, `then`, `else`, `fi`, `for`, `in`, `do`, `done`, `while`, `until`, `case`, `esac`, `{`, `}`, `!`

4.2. Conditionals and Loops

Table: Shell Control Structures

Structure	Syntax Example
if	if ...; then ...; else ...; fi
while	while ...; do ...; done
for	for x in ...; do ...; done
case	case x in ... esac
group	{ ...; }
negate	! command

If Statement

```
if cat file; then
  echo "Success"
else
  echo "Failure"
fi
```

- Exit status: 0 = success, nonzero = failure (opposite of C/C++ convention).

While Loop

```
while cat file; do
  sleep 1
done
```

For Loop

```
for i in a b c; do
  echo $i
```

```
done
```

- The control variable takes each value in the list.
- Commonly used with expansions (e.g., `for f in *; do ...`).

Case Statement

```
case $x in
  *.c) gcc $x;;
  *.cpp) g++ $x;;
  *) echo "Unknown type";;
esac
```

- Patterns can be arbitrary (not just file extensions).
- If no pattern matches, exit status is 0 (success).

Grouping and Negation

- `{ ... }`: Group commands as a single command (semicolon required before `}` if on same line).
- `!`: Negate the exit status of a command.

---

5. Variables and Parameters

5.1. Special Variables

- `$0`: Script/command name
- `$1`, `$2`, ...: Positional arguments
- `$#`: Number of arguments
- `$*`: All arguments as a single word
- `$@`: All arguments as separate words

Usage	Result
"$*"	All args as a single string
"$@"	All args, each as a word

- Quoting matters: `"$@"` preserves argument boundaries, `"$*"` joins them into one string.

5.2. Default and Conditional Substitution

- `${VAR:-default}`: Use default if VAR is unset
- `${VAR:=default}`: Set VAR to default if unset
- `${VAR:+alt}`: Use alt if VAR is set
- `${VAR:?error}`: Print error if not set

Examples:

```
name=${USER:-guest}   # Use $USER if set, else 'guest'
file=${1:?Must provide a filename}
```

5.3. Unsetting and Exporting

- `unset x`: Unset variable x
- `export VAR=value`: Make variable available to subcommands (environment variable)

5.4. The Environment

- The shell maintains a set of environment variables (name-value pairs) inherited by child processes.
- Use `export` to add variables to the environment.
- **Shell variables** are local to the shell; **environment variables** are inherited by child processes.

5.5. Special Built-in Variables

- `$$`: PID of the current shell
- `$!`: PID of the last background command

6. Redirection, Pipes, and Concurrency

6.1. Standard File Descriptors

Descriptor	Number	Default
stdin	0	Keyboard
stdout	1	Screen
stderr	2	Screen

6.2. Redirection

- `2>file`: Redirect stderr to file
- `1>&2`: Redirect stdout to stderr
- `command &`: Run in background

6.3. Advanced Redirection and Pipes

- You can redirect different file descriptors in a pipeline:

```
A 2>err1 | B 2>err2 | C
```

- To merge stderr and stdout:

```
command 2>&1
```

- To split output to multiple places, use `tee`:

```
cmd | tee file | another_cmd
```

6.4. Multiple Writers/Readers and Race Conditions

- If multiple processes read from or write to the same pipe, the data is split in a race condition (nondeterministic).
- Example:

```
cmd1 | tee output.txt | cmd2
```

- Avoid multiple writers/readers unless you want nondeterministic behavior.

Diagram: Multiple Writers/Readers

```
[Writer1]---+
            |---> [PIPE] ---> [Reader1]
[Writer2]---+                [Reader2]
```

- If a process closes its end of a pipe, the other end receives EOF (readers) or SIGPIPE (writers).

6.5. Pipes as Bounded Buffers

- Pipes are implemented as bounded buffers in the kernel (size varies by OS).
- If the buffer is full, the writer blocks; if empty, the reader blocks.
- No overflow: the OS suspends the process if it tries to write more than the buffer can hold.

ASCII Diagram: Pipe

```
[Program A] --writes--> [PIPE (buffer)] --reads--> [Program B]
```

7. Shell Expansion Rules

Order of processing:

1. Tilde expansion (`~`, `~user`)
2. Parameter expansion (`$VAR`)
3. Command substitution (`$(cmd)` or `` `cmd` ``)
4. Arithmetic expansion (`$((expr))`)
5. Field splitting (on whitespace)
6. Pathname expansion (globbing, e.g., `*.c`)
7. Redirection

Table: Shell Expansion Steps and Examples

Step	Example	Result/Effect
Tilde expansion	`echo ~`	`/home/user`
Parameter expansion	`echo $USER`	`dsun`
Command substitution	`echo $(date)`	`Mon Jun 3 ...`
Arithmetic expansion	`echo $((1+2))`	`3`
Field splitting	`set -- a b c`	3 args: a, b, c
Pathname expansion	`ls *.c`	`foo.c bar.c`
Redirection	`echo hi > out.txt`	Output to file

Example:

```
echo $((1 + 2)) # => 3
ls *.c          # Lists all .c files
```

- If no files match a glob pattern, the pattern remains unexpanded (unless `nullglob` is set in some shells).

---

8. Summary

This lecture covered the structure and function of POSIX file systems, focusing on file naming, hard and symbolic links, and the resolution of file names. It explained the shell's role as a scripting and coordination tool, including command execution, redirection, quoting, reserved words, variables, environment, process management, pipes, and the order of shell expansions. Special attention was paid to subtle points like symlink loops, quoting nuances, and race conditions in pipes.

Pattern Matching, Regular Expressions, and Emacs

1. Globbing (Shell Pattern Matching)

Globbing is a simple, fast pattern-matching syntax used by shells (e.g., bash) for matching file names. It is related to, but simpler and more limited than, regular expressions. Globbing is used in commands like `ls`, `echo`, `rm`, and in shell control structures like `case`.

- **Globbing matches only filename components** (parts of a path between slashes). Globs never match `/`.
- **Dot files** (files starting with `.`) are not matched by `*` or `?` unless the pattern itself starts with a dot.
- If a glob pattern matches nothing, the shell usually passes the pattern unchanged to the command.

Globbing vs Regular Expressions

Feature	Globbing (Shell)	Regular Expressions (Regex)
Used for	Filenames	Text/string matching
Wildcards	`*`, `?`, `[abc]`	`.`, `*`, `+`, `?`, `[]`, `()`
Matches `/`?	Never	Can match `/` (unless told not to)
Syntax	Simple	More powerful, complex
Negation	`[!abc]`	`[^abc]`
Quoting needed	To prevent expansion	To prevent shell expansion

Special Characters and Syntax

Symbol	Meaning
`*`	Matches any sequence of zero or more characters
`?`	Matches exactly one character
`[abc]`	Matches any one character in the set (here: 'a', 'b', or 'c')
`[a-z]`	Matches any character in the range (here: 'a' to 'z')
`[!abc]`	Matches any character NOT in the set (negation)

- Use `-` at the end of a bracket expression to match a literal `-`.
- To match a literal `]`, put it first in the set: `[]abc]` matches `]`, `a`, `b`, or `c`.
- To match a literal `!`, put it anywhere but first.

Globbing and Directory Structure

- Globs never match `/` (directory separator).
- Patterns like `*/foo*` match `foo*` in direct subdirectories only.
- To match files two levels down: `*/*/foo*`.

Dot Files and Globbing

- `*` and `?` do **not** match files starting with `.` unless the pattern itself starts with a dot.
- Example: `echo *` will not show `.hidden` files.
- To match hidden files: use a pattern like `.??*` (matches hidden files with at least three characters).

Globbing Negation

- `[!abc]*` matches files not starting with `a`, `b`, or `c`.
- `[!abc.]*` matches files not starting with `a`, `b`, `c`, or `.`.

Globbing Examples

Pattern	Description	Matches
`*`	All files except hidden ones	`foo`, `bar.txt`
`?.txt`	One character followed by `.txt`	`a.txt`
`[abc]*`	Files starting with `a`, `b`, or `c`	`apple`, `cat`
`[!abc]*`	Files not starting with `a`, `b`, or `c`	`dog`

Pattern	Description	Matches
`*.bash`	Files ending with `.bash`	`script.bash`
`.??*`	Hidden files with name length ≥ 3	`.xrc`

Example Directory

Suppose directory contains: `foo`, `bar`, `.hidden`, `a.txt`, `a-b.txt`

Command	Matches
`echo *`	foo, bar, a.txt, a-b.txt
`echo .*`	.hidden, .., .
`echo ?.txt`	None (needs 1-char name)
`echo [a-c]*`	bar
`echo [!abc]*`	Anything not starting a/b/c

Globbing and Slashes

- Patterns never match `/`.
- A trailing slash in a pattern (e.g., `foo/`) requires the match to be a directory.

Globbing and Quoting

- If you want to prevent the shell from expanding a glob, quote it: `'[a-z]*'`.
- Quoting is also necessary to prevent the shell from interpreting special characters in regex patterns.

Why "glob"?

- The original function implementing this was called `glob`.

Escaping Special Characters in Globs

- To match a literal `*`, `?`, or `[`, you can escape them with a backslash (e.g., `\*`, `\?`, `\[`).
- Example: `echo file\*` matches the file named `file*` (not all files starting with `file`).

Shell Options Affecting Globbing

- `shopt -s dotglob`: Makes `*` and `?` match dotfiles.
- `shopt -s nullglob`: Expands globs to nothing if no match (instead of passing the pattern unchanged).

Globbing Expansion Flowchart

```
+-----------------------------+
| User types command with glob|
+-------------+---------------+
              |
              v
+-----------------------------+
| Shell scans for matches     |
+-------------+---------------+
              |
      +-------+-------+
      |               |
      v               v
+---------+     +----------------+
| Matches |     | No matches     |
+----+----+     +-------+--------+
     |                 |
     v                 v
Command runs     Pattern passed as-is
with matches     to command (unless nullglob)
```

---

2. Shell I/O Redirection

Shells use file descriptors for I/O:

Descriptor	Description
0	Standard Input
1	Standard Output
2	Standard Error

Redirection Syntax

Syntax	Meaning
`>`	Redirect stdout to file (overwrite)
`>>`	Redirect stdout to file (append)
`2>&1`	Redirect stderr to where stdout points
`3< file`	Open file on descriptor 3 for reading
`3> file`	Open file on descriptor 3 for writing
`3<> file`	Open file for reading and writing on descriptor 3
`3>&1 1>file`	Save current stdout on fd 3, then redirect stdout

Here Documents

A "here document" feeds a block of text as stdin to a command:

```
cat <<EOF
line one
line two
EOF
```

- Variable expansion occurs unless the delimiter is quoted: `<< 'EOF'` prevents expansion.

File Descriptor Redirection Diagram

```
[stdin 0] <---+         +---> [stdout 1] ---> file or terminal
              |         |
              |         +---> [stderr 2] ---> file or terminal
              |
         [other fds: 3,4,...]
```

Common Redirection Idioms

Idiom	Meaning
`command > file`	stdout to file (overwrite)
`command >> file`	stdout to file (append)
`command 2> file`	stderr to file
`command > out 2>&1`	stdout and stderr to same file
`command < file`	stdin from file
`command 2>&1 >file`	stderr to old stdout, then stdout to file

Using Custom File Descriptors in Scripts

```
exec 3>log.txt   # Open log.txt for writing on fd 3
echo "log entry" >&3
exec 3>&-        # Close fd 3
```

---

3. Shell Scripting: Exit, Return, Functions, Scripts, Aliases

Exit and Return

Command	Effect
`exit`	Terminates shell or script
`exit 1`	Exits with status code 1
`return`	Exits from a shell function only

- `exit` ends the shell; `return` only exits a function.
- Exit status of last command: `$?`

Shell Functions vs. Shell Scripts

Shell Function

```
g() {
  grep "$@"
}
```

- Defined in `.profile` or interactively
- Lightweight, runs in current shell
- Not visible to other programs

Shell Script

File `g`:

```
#!/bin/bash
grep "$@"
```

- Saved in a directory in `$PATH`
- Executed as a new process
- Visible to all programs

Feature	Function (in-shell)	Script (file)
Scope	Local to shell	Global (all programs)
Overhead	Low	High (new process)
Portability	Low	High
Speed	Fast for small	Better for large

Aliases

- For simple substitutions only:

```
alias g='grep'
```

- Not suitable for complex logic; use functions instead.

Sourcing Scripts

- Use `source script.sh` or `. script.sh` to run a script in the current shell (variables/functions persist).

Variable Scope Example

```
myfunc() {
  local x=5
  echo "x in function: $x"
}
x=10
myfunc
# x outside function is still 10
```

Script/Function/Alias Execution Flowchart

```
+-------------------+
| User enters name  |
+---------+---------+
          |
          v
+-------------------+
| Is it an alias?   |--yes--> expand and run
|        |          |
|        no         v
|             Is it a function? |--yes--> run in current shell
|        |          |
|        no         v
|             Is it a script in $PATH? |--yes--> run as new process
|        |          |
|        no         v
|             Command not found
```

---

4. Regular Expressions (Regex)

Regular expressions are a "little language" for string pattern matching, used in tools like `grep`, `sed`, `awk`, Python, etc. There are multiple regex syntaxes (ERE, BRE, Perl, Python, etc.)—be careful to use the right one for your tool.

Extended Regular Expressions (ERE)

- Use with `grep -E` or `egrep`.

Core Syntax and Operators

Operator	Description
`.`	Any character except newline
`*`	0 or more repetitions
`+`	1 or more repetitions
`?`	0 or 1 occurrence (optional)
`|`	Alternation (OR)
`()`	Grouping (changes precedence)
`{n}`	Exactly n occurrences
`{n,m}`	Between n and m repetitions
`^`	Start of line anchor
`$`	End of line anchor
`[]`	Bracket expressions (character classes)

Operator Precedence (Highest to Lowest)

1. `*`, `+`, `?`, `{}`
2. Concatenation
3. `|` (alternation)

Bracket Expressions

- `[abc]`: matches 'a', 'b', or 'c'
- `[a-z]`: matches any lowercase letter
- `[^abc]`: matches any character except 'a', 'b', or 'c'
- `[[:alpha:]]`: matches any alphabetic character (locale-aware)
- To include `-`, put it at the end: `[a-z-]`
- To include `]`, put it first: `[]abc]`
- To include `^`, put it anywhere but first
- Inside brackets, `-` denotes range unless at start or end

- Backslash is NOT special inside brackets

Special bracket tricks

- `[.]`, `[-]`, `[]]` — see above for how to include these literally
- `[[:alnum:]]`, `[[:digit:]]`, `[[:print:]]` — POSIX character classes

Examples

Pattern	Matches Example
`abc`	Only the string 'abc'
`a.b`	'a' followed by any char, then 'b'
`a*`	zero or more 'a's
`a+`	one or more 'a's
`(ab|cd)+`	'ab' or 'cd', repeated
`^xyz$`	entire line must be 'xyz'
`^(.)(.)(.).\3\2\1$`	six-character palindromes

Anchors

- `^` at start: match only at beginning of line
- `$` at end: match only at end of line

Quoting and Escaping

- Backslash escapes special characters: `\*`, `\.`, `\(`, etc.
- To match `\`, often need `\\` (shell and regex both interpret backslash)
- Be careful: the shell may expand or interpret backslashes before passing to grep
- Always quote regex patterns in the shell to avoid globbing: `'[a-z]*'`

Common Pitfalls

- A regex with just a backslash (`\`) is invalid
- Quoting and escaping can be tricky—test your patterns

Bracket Expression Edge Cases

- To include `-` literally, put it at the start or end: `[-a-z]` or `[a-z-]`
- To include `]` literally, put it first: `[]a-z]`
- To include `^` literally, put it anywhere but first
- Ranges like `[z-a]` are invalid
- `[a-]` matches 'a' or '-'
- `[a-z-]` matches 'a' to 'z' and '-'

ASCII Diagram: Bracket Expression Rules

```
[ ^ ]
 |   |
 |   +-- If first after [, means negation
 +------ If not first, just a literal ^

[ - ]
 |   |
 |   +-- If at start or end, literal -
 +------ Else, range

[ ] ]
 |   |
 |   +-- If first after [, literal ]
 +------ Else, ends bracket expression
```

Basic Regular Expressions (BRE)

- Used with `grep` (no `-E`)

- Fewer metacharacters are special: `+`, `?`, `{}` are NOT special
- Use `\(...\)` for grouping
- Use `\{n,m\}` for repetition
- No alternation (`|`) operator

Backreferences (BRE only)

- `\1`, `\2`, ... refer to the nth parenthesized group
- Example: `\(abc\)\1` matches 'abcabc'
- Example: `^\(\.\)\(\.\)\2\1$` matches 4-character palindromes
- Backreferences are slow and non-regular—avoid if possible

Example Table

Pattern	Description
`\(abc\)\1`	Matches 'abcabc'
`\(^a.*b$\)`	Entire line starting with a, ending with b
`^\(.\)\(.\)\2\1$`	Matches 4-character palindromes

Regex Syntax Comparison

Feature	ERE (`grep -E`)	BRE (`grep`)	Perl/Python	Globbing
Grouping	`()`	`\(\)`	`()`	N/A
Alternation	`|`	N/A	`|`	N/A
`+`, `?`, `{}`	Yes	No (use `\{}`)	Yes	N/A
Backrefs	N/A	`\1`, `\2`	`\1`, `\g<1>`	N/A
Wildcard	`.`	`.`	`.`	`*`, `?`
Negation	`[^abc]`	`[^abc]`	`[^abc]`	`[!abc]`

Regex Matching Process Diagram

```
+-------------------+
| Start at position |
+---------+---------+
          |
          v
+-------------------+
| Try to match next |
| regex element     |
+---------+---------+
          |
          v
+-------------------+
| If match, advance |
| else, backtrack   |
+---------+---------+
          |
          v
+-------------------+
| End: full match?  |
+---------+---------+
          |
      yes | no
          v
   Success  Fail
```

Greedy vs Non-Greedy Quantifiers

- **Greedy**: Match as much as possible (default: `*`, `+`, `?`)
- **Non-greedy (lazy)**: Match as little as possible (Perl/Python: `*?`, `+?`, `??`)
- Example (Python):

```python
import re
s = 'aaaab'
print(re.match(r'a+?', s).group())  # 'a' (non-greedy)
print(re.match(r'a+', s).group())   # 'aaaa' (greedy)
```

Multiline Matching Example (Python)

```python
import re
text = """foo\nbar\nbaz"""
print(re.findall(r'^b.*', text, re.MULTILINE))  # ['bar', 'baz']
```

Example: Regex in Python

```python
import re
pattern = r'\b\w{3}\b'
text = 'cat dog bird fish'
print(re.findall(pattern, text))  # ['cat', 'dog']
```

---

5. Emacs: Keyboard-Driven Text Editing

Emacs is a highly extensible, keyboard-driven text editor. Its philosophy is to maximize efficiency by minimizing hand movement from the keyboard. It is modular, with major and minor modes, and features a powerful mini-buffer for commands and scripting.

Core Concepts

- **Buffers**: In-memory views of files, outputs, or shell sessions
- **Windows**: Viewports into buffers (not OS windows)
- **Kill Ring**: Stores multiple cut/copied text entries; cycle with `M-y` after `C-y`

Essential Commands

Command	Action
`C-x C-f`	Open file
`C-x C-s`	Save file
`C-g`	Cancel current command
`C-SPC`/`C-@`	Set mark at cursor (start selection)
`M-w`	Copy region to kill ring
`C-w`	Cut (kill) region
`C-y`	Yank (paste) last kill ring
`M-y`	Cycle through kill ring after yank
`C-x C-x`	Exchange point and mark
`C-x b`	Switch buffer
`C-x C-b`	List all buffers
`C-x o`	Switch windows
`C-x 2`	Split window horizontally
`C-x 3`	Split window vertically
`C-x 0`	Close current window
`C-x 1`	Maximize current window
`C-h k`	Describe key binding
`C-h m`	Describe current mode
`C-h i`	Info documentation browser

- **Meta key (`M-`)**: Usually `Alt` or `Esc`

Mini-buffer Operations

- `M-x`: Run Emacs command by name
- `M-:`: Evaluate Emacs Lisp
- `M-!`: Run shell command
- `M-|`: Run shell command with region as input

Examples

- `M-! date`: Run shell `date` command
- `M-| sort`: Sort selected region

Modes

- **Major modes**: Tailor Emacs for file type or buffer (e.g., Fundamental, Dired)
- **Minor modes**: Add auxiliary features (e.g., line numbers)
- Use `C-h m` to see active modes and key bindings

ASCII Diagram: Emacs Buffer/Window/Region

```
+-------------------------------+
|           Emacs Frame         |
| +-----------+ +-----------+   |
| |  Buffer 1 | |  Buffer 2 |   |
| +-----------+ +-----------+   |
|                               |
| [Kill Ring: C-w, C-y, M-y]    |
+-------------------------------+
```

Emacs Configuration Files

- User configuration is stored in `~/.emacs` or `~/.emacs.d/init.el`.
- You can customize Emacs by adding Emacs Lisp code to these files.

Example: Simple Emacs Lisp Function

```
(defun hello-world ()
  (interactive)
  (message "Hello, world!"))
```

- Run with `M-x hello-world`.

Kill Ring Cycling Diagram

```
[First kill] -> [Second kill] -> [Third kill]
      ^                              |
      |------------------------------|

C-y: yank most recent
M-y: cycle through previous kills
```

Scripting Languages — Syntax, Semantics, Pragmatics, and Emacs Lisp

1. Introduction to Scripting Languages

When discussing programming languages, three major areas are essential:

Category	Description	Scripting Example	Traditional Example
Syntax	Structure or form of code (e.g., semicolons, braces).	Python: whitespace, Shell: no braces	C++: braces, semicolons
Semantics	Meaning or behavior of the program—what the code does.	Python: dynamic typing, Shell: command execution	C++: static typing, compiled execution
Pragmatics	Practical concerns: efficiency, configuration, usability, security, and interoperability. Most emphasized in scripting.	Python: glue code, Shell: automation	C++: performance, static analysis

- Scripting languages focus heavily on pragmatics, not just syntax or semantics.
- Pragmatics includes: how to get things working, configuration, efficiency, security, trust, and integration with other modules.

Pragmatics in Action:

> Suppose you need to automate a nightly backup. In C++, you would need to write, compile, and deploy a program, handling all errors and edge cases. In Bash, a 3-line script using `rsync` and a cron job suffices. The scripting approach wins on pragmatics: speed, integration, and ease of change.

---

2. Overview of Example Scripting Languages

Language	Primary Use	Integration	Performance	Extensibility	Example Code
Shell	OS automation, scripting	Unix tools, files	Low	High	`echo Hello`
Emacs Lisp	Editor extension	Emacs internals	Low-Medium	Very High	`(message "Hi")`
Python	General scripting, glue	C/C++ modules, APIs	Medium	High	`print("Hi")`
JavaScript	Web, extension, general	Browsers, Node.js	Medium-High	High	`console.log("Hi")`

Bytecode and JIT:

- **Bytecode:** An intermediate, platform-independent code executed by a virtual machine (e.g., Python `.pyc`, Java `.class`).
- **JIT (Just-In-Time) Compilation:** Converts bytecode to native machine code at runtime for speed (e.g., JavaScript V8, Java HotSpot).

```
[Source Code]
     |
     v
[Bytecode] --(JIT)--> [Native Machine Code]
```

---

3. Key Themes of Scripting Languages

Ease of Use

- Scripting languages prioritize accessibility and quick onboarding.
- Easy to learn, fast to write short scripts (e.g., "Hello World!").
- Low barrier to entry increases ecosystem participation.

Reliability

- Scripting languages avoid worst-case crashes common in C++ (e.g., null pointer exceptions, subscript errors).
- Instead, they raise exceptions or print errors.
- **Downside:** Permissiveness may allow incorrect programs to execute, decreasing logical reliability.

Language	Crashes on Subscript Error?	Throws Exception?
C++	Yes	No
Python	No	Yes
Shell	No	Typically alerts

Example:

- C++:

```
int arr[2] = {1,2};
int x = arr[5]; // Undefined behavior, may crash
```

- Python:

```
arr = [1,2]
x = arr[5] # Raises IndexError
```

Scalability

- Good for small to medium programs, glue code, and lightweight data transformation.
- Problems with very large codebases due to lack of strict typing and fragmented tooling.
- Large systems often use scripting languages to configure or glue together performant code written in C++ or similar.

Performance

- Generally slower than compiled languages due to runtime checks and interpreted execution.
- Tradeoff: slower performance for higher productivity and reliability.
- JIT compilers and bytecode interpreters (e.g., JavaScript, Emacs Lisp) help mitigate the performance gap.
- Example: C++ is fast because it doesn't do subscript checking; Python checks at runtime and throws exceptions.

Productivity vs. Performance:

```
Productivity
  ^
  |        Python, Shell
  |      /
  |     /
  |    /
  |   /
  |  /      C++
  +--------------------> Performance
```

---

4. Concepts and Philosophical Differences

General vs. Extension Languages

Language Type	Example	Description	Example Code
General Purpose	Python, Java	Use it for almost anything	`print(42)`
Extension	JavaScript, ELisp	Embedded in host applications (e.g., browser, editor) to extend functionality	`(message "Hi")` or `alert('Hi')`

Extension Language Example:

- JavaScript in a browser:

```
document.body.style.background = 'yellow';
```

- Emacs Lisp in Emacs:

```
(set-background-color "yellow")
```

Traditional vs. Scripting

Feature	Scripting Languages	Traditional Languages
Performance	Low	High
Flexibility	High	Medium
Error Handling	Graceful	Harsh (e.g. segfaults)
Compilation	Optional or runtime only	Mandatory
Type System	Dynamic/optional	Static/mandatory
Tooling	Fragmented, lightweight	Mature, integrated
Deployment	Script file, interpreter	Compiled binary

Design Purpose of Languages

Language	Intended Use
JavaScript	Extending browser functionality
Shell	Automated OS-level task and config scripts
Emacs Lisp	Extending the editor's features
SQL	Query and data specification
Rust	High-performance and safe system programming

---

5. Emacs Lisp: An Introduction

Historical Context and Purpose

- Lisp was created for AI in the 1960s, focusing on self-modifying and program-generating code.
- Emacs Lisp is a variant tailored to the GNU Emacs text editor.
- Emacs Lisp is an **extension language**: enhances and customizes Emacs behavior.

Emacs Lisp as an Extension Language

- Deep integration with Emacs internals: buffers, windows, syntax highlighting, etc.
- Provides primitives focused on editing rather than general computing (compare to SAP's payroll-specific extension language).
- Emacs is built on:
  - Low-Level C libraries
  - Lisp Interpreter written in C

Emacs Architecture:

```
+-------------------+
|   Emacs (C Core)  |
+-------------------+
         |
         v
+-------------------+
| Lisp Interpreter  |
+-------------------+
         |
         v
+-------------------+
| User Lisp Code    |
+-------------------+
```

- Code entered in Emacs is interpreted or compiled into bytecode.

Self-Modifying Code Example:

```
(defun add-x (n)
  (defun x () n))
(add-x 42)
(x) ; => 42
```

---

6. Syntax and Semantics in Lisp

Lisp Syntax Characteristics

- Simple, uniform syntax using parenthesized prefix notation.
- All function calls look like: `(function arg1 arg2 ...)`
- No operator precedence, no infix notation.
- Symbols and data use homogenous parenthetical syntax.

Structuring Code as Lists

- Code and data are blurred: both written as lists.
- Example:
  - Data: `'('a b c)`
  - Code: `(+ 34 25)`

Symbols and Special Forms

- **Symbols**: Unique named atomic values (e.g., `nil`, `t`, `my-var`).
- **Special Forms**: Control structures and definition constructs that are not function calls (e.g., `defun`, `let`, `lambda`, `quote`).

Special Form	Purpose	Example
defun	Define a function	`(defun foo (x) x)`
let	Local variable binding	`(let ((x 1)) x)`
lambda	Anonymous function	`(lambda (x) (+ x 1))`
quote	Prevent evaluation	`'(a b c)`
if	Conditional	`(if cond a b)`
progn	Sequence of expressions	`(progn a b c)`

Macro Example

```
(defmacro when (cond &rest body)
  `(if ,cond (progn ,@body)))

(when (> 3 2)
  (message "3 is greater than 2"))
```

Expressions vs. Data

- `'expression` prevents evaluation.
- `(eval expression)` forces evaluation even of quoted data.

Form	Meaning
'abc	Symbol `abc` as data
abc	Evaluate variable `abc`
'(a b c)	List of three symbols a, b, c
(quote (a b c))	Same as above

Interactive Evaluation (Scratch Buffer)

- Evaluated using:
  - `C-J`: Evaluate previous expression and show result below.
  - `C-x C-e`: Evaluate expression at point and show in mini-buffer.

Debugging Basics

- Emacs will drop into the debugger upon errors.
- Key command to exit debugger: `C-]` (Control + closing square bracket)
- Debugger sessions can nest; exit each with `C-]` until you return to the top level.

---

7. Emacs Lisp Data Structures

Numbers

- Integers: Arbitrary precision (bignums)
- Floats: IEEE 64-bit doubles
- Example: `(+ 100000000000000000 1)`
- Integer division truncates; floating-point division if one operand is a float.
- No support for smaller-width floats (e.g., 8-bit) as in some ML applications.

Strings

- Written with double quotes
- Supports newlines: `"Hello\nWorld"`
- Strings can contain any characters, including newlines.
- Functions like `format` allow C-style string formatting.

Symbols

- Named values that point to variables or functions
- Used for identifiers, constants, or stand-alone atoms
- Special symbols:
  - `nil`: false, end of list
  - `t`: true
- Each symbol is unique by name; two symbols with the same name are the same object.

Lists

- Fundamental concept in Lisp: "List Processing"
- Built from pairs (cons cells): each has a value and a pointer to the next cell.
- Built-in functions for lists:
  - `cons`: Construct a new pair
  - `car`: Return first element
  - `cdr`: Return rest of list

List Example

```
(cons 'a (cons 'b nil)) ; => (a b)
```

Improper Lists

- If the last element is not `nil`, it's an improper list.

```
(cons 'a 'b) ; => (a . b)
```

- `(a . b)` is the printed form for an improper list.

Improper List Diagram:

```
+-------+-------+
|  a    |   o---|---> b
+-------+-------+
```

Quoting and Data/Code Duality

- `'abc` yields the symbol `abc` as data.
- `'('a b c)` yields the list `(a b c)` as data.
- `(eval ...)` can turn data into code.

Every Paren Counts

- In Lisp, every parenthesis is meaningful. Extra parentheses change the meaning (e.g., `(n)` is a function call to `n`, not just the value of `n`).

Vectors

- Fixed-size, indexable arrays

- Created via: `(make-vector 5 'x)`
- Less commonly used than lists, but allow random access.

Hash Tables, Character Tables, Markers, Buffers, Frames, Windows, Processes

- **Hash Tables**: Key-value associative maps.
- **Character Tables**: Hash tables with character keys for efficiency.
- **Markers**: Pointers to specific positions in buffers that adapt as buffer changes.
- **Buffers**: Editable text containers (contents of a file, scratch, etc.).
- **Frames/Windows**: Emacs GUI elements.
- **Processes**: Represent running subprocesses (e.g., shells) under Emacs.
- **Fonts, X widgets, Terminals**: Other Emacs object types.

Example: Creating and Using a Marker

```
(setq m (make-marker))
```

- `m` is now a marker object. To make it point to a buffer, pass a buffer argument.

Example: Buffers

```
(current-buffer) ; returns the current buffer object
```

Comparison Table: Lists, Vectors, Hash Tables

Structure	Ordered?	Random Access	Typical Use	Example Creation
List	Yes	No	Sequence, stack, queue	`(list 1 2 3)`
Vector	Yes	Yes	Fixed-size, fast lookup	`(make-vector 3 'x)`
Hash Table	No	Yes (by key)	Key-value mapping	`(make-hash-table)`

---

8. Functional Programming Concepts in Emacs Lisp

Functions

- First-class objects
- Created via `lambda` or `defun`

Using `defun`:

```
(defun add-three (x) (+ x 3))
```

Interactive Commands:

- Use `interactive` keyword to make a function callable via UI/keystrokes.

```
(defun greet-user ()
  (interactive)
  (message "Hello!"))
```

Example:

```
(defun show-buffer-name ()
  (interactive)
  (message (buffer-name)))
```

Local Bindings (`let`)

- Creates local variables bound to evaluated values.

```
(let ((x 3) (y 4)) (+ x y)) ; => 7
```

- Temporarily masks global variables if names overlap.
- `let` can bind multiple variables; each binding is a pair `(name value)`.
- If only a symbol is given, it is bound to `nil`.

let vs. Global Variable Table:

Feature	let (local)	Global Variable
Scope	Expression/block	Everywhere
Lifetime	Temporary	Until unset
Side Effects	None (isolated)	Can affect all code
Example	`(let ((x 1)) x)`	`(setq x 1)`

Lambda Expressions and Closures

- Emacs supports anonymous functions with `lambda`.

```
(lambda (x y) (+ x (* y y)))
```

- Can pass lambdas as arguments or assign to variables.
- **Closures:** Functions that capture their lexical environment.

```
(let ((n 10))
  (lambda (x) (+ x n))) ; Returns a function that adds 10
```

Special Forms

- Not all parenthesized expressions are function calls. Some are special forms (e.g., `lambda`, `let`, `quote`, `defun`).
- Special forms have unique evaluation rules.

---

9. Emacs Lisp Utilities and Commands

Emacs Commands

- Functions callable via UI must be declared interactive:

```
(defun my-command ()
  (interactive)
  (message "Hello"))
```

- Can query keystrokes or input with `interactive` argument.

Accessing Documentation

- `C-h k`: Describe key bindings.
- `C-h f`: Describe function.
- `C-h m`: Describe current mode.
- Mouse over source file link (e.g., window.el) to open implementation in Emacs.

Useful Emacs Lisp Introspection Commands:

Command	Description
`describe-function`	Show documentation for a function
`describe-variable`	Show documentation for a variable
`apropos`	Search for functions/variables by name
`find-function`	Jump to function source code

Command	Description
`eldoc-mode`	Show function signatures in minibuffer

10. Summary

This lecture introduced scripting languages, emphasizing their focus on pragmatics—real-world concerns of integration, performance trade-offs, ease of use, reliability, and extensibility. It differentiated scripting languages from general-purpose and low-level languages, exemplified by Emacs Lisp and Python. The lecture gave an in-depth presentation of Emacs Lisp as an extension language, explaining its evaluation model, key data types (including symbols, lists, strings, functions), and its support for self-modifying code, which allows it to script and modify Emacs in powerful ways. The class also demonstrated fundamental Lisp programming techniques including quoting, evaluation, function definition, lambda expressions, local variable bindings, and debugging tools within Emacs.

Visualizations and Diagrams

List Structure in Lisp (ASCII Diagram)

```
+-------+-------+    +-------+-------+    +-------+-------+
|  a    |   o---|--->|  b    |   o---|--->|  c    |  nil  |
+-------+-------+    +-------+-------+    +-------+-------+
```

Table: Lisp Data Types

Type	Example	Notes
Integer	42, 100000000000000000	Arbitrary precision
Float	3.14, -0.5	IEEE 64-bit double
String	"Hello\nWorld"	Can contain newlines
Symbol	'abc, 'nil, 't	Unique by name
List	'(a b c), (cons 'a nil)	Linked list of cons cells
Vector	(make-vector 5 'x)	Fixed-size, indexable array
Hash Table	(make-hash-table)	Key-value map
Marker	(make-marker)	Buffer position pointer
Buffer	(current-buffer)	Editable text container
Frame/Window	(selected-frame)	GUI elements
Process	(start-process ...)	Running subprocess

Table: Special Forms vs. Function Calls

Syntax Example	Is Function Call?	Notes
(+ 1 2)	Yes	Calls function +
(defun foo ...)	No (special form)	Defines a function
(let ((x 1)) ...)	No (special form)	Local variable binding
(lambda (x) (+ x 1))	No (special form)	Anonymous function
(quote (a b c))	No (special form)	Returns data, not code

Table: Emacs Lisp Evaluation Commands

Command	Effect
C-J	Evaluate previous expression, show result below
C-x C-e	Evaluate expression at point, show in mini-buffer
C-]	Exit debugger
C-h k	Describe key bindings

Command	Effect
C-h f	Describe function
C-h m	Describe current mode

Table: Comparison of Scripting and Traditional Languages

Feature	Scripting Languages	Traditional Languages
Performance	Low	High
Flexibility	High	Medium
Error Handling	Graceful	Harsh (e.g. segfaults)
Compilation	Optional or runtime only	Mandatory

Language Types Summary Diagram

```
[General Purpose]
      |
      v
[Extension Language] <---[Host Application]
      |
      v
[Scripting Language]
```

Macro Expansion and Evaluation Process

```
[Source Code]
     |
     v
[Macro Expansion]
     |
     v
[Evaluation]
     |
     v
[Result]
```

Python Scripting: Introduction & Fundamentals

1. Introduction

1.1 Audience Spectrum

This lecture is designed for a mixed audience: some already experienced with Python, others complete beginners. The approach balances basic teaching with deeper insights, aiming to appeal to both groups.

1.2 Teaching Approach

- Uses live examples to build intuition.
- Encourages Pythonic practices and high-level thinking.
- Highlights common pitfalls, historical motivations, and the philosophy behind Python features.

---

2. Writing and Executing Python

2.1 Interactive Use

Python can be run in various environments:

- Emacs (`Meta-X run-python`)
- Terminal/command line
- Jupyter notebooks
- Any preferred editor or IDE

Table: Python Execution Environments

Environment	How to Start	Features
REPL (python)	`python` in terminal	Interactive, quick testing
Script	`python script.py`	Runs full file, reproducible
Jupyter Notebook	`jupyter notebook`	Rich output, cells, visualization
IDE (e.g., PyCharm)	Open project, run	Debugging, code completion
Emacs	`Meta-X run-python`	Editor integration

2.2 Running Scripts vs. Interactive Mode

- **Script mode:** Good for reproducibility, sharing, and larger programs.
- **Interactive mode:** Good for experimentation, debugging, and learning.

2.3 Example: Parsing Stock Market Data

Given a string:

```
"GOOG,100,153.36"
```

Goal:

- Extract the symbol ("GOOG")
- Convert 100 to int
- Convert 153.36 to float

Target Output:

```
['GOOG', 100, 153.36]
```

Step-by-step:

1. **Split the string** using `.split(',')`:

   ```
   s_line = "GOOG,100,153.36".split(',')  # ['GOOG', '100', '153.36']
   ```

2. **Define types**:

```
types = [str, int, float]
```

3. **Zip types and values**:

```
list(zip(types, s_line))  # [(str, 'GOOG'), (int, '100'), (float, '153.36')]
```

   - Note: `zip` returns an iterator; casting to `list()` is needed to see all elements.
   - Once iterated, a zip object is exhausted.
4. **List comprehension for conversion**:

```
[c(v) for c, v in zip(types, s_line)]  # ['GOOG', 100, 153.36]
```

Pythonic Principle

- Prefer list comprehensions and high-level constructs over manual indexing and C++-style loops.

---

3. Pythonic Programming Philosophy

3.1 Avoid Low-Level Thinking

- Avoid C++-style manual indexing, pointer manipulation, and low-level constructs.
- Python encourages thinking at a higher level of abstraction.

3.2 Embrace High-Level Abstractions

- Use list comprehensions, built-in functions, and idiomatic constructs.
- Prefer `for x in y` over `for i in range(len(y))`.

---

4. Brief History of Python

4.1 Language Evolution

- **Fortran (1950s):** Unforgiving, used for scientific computing, not beginner-friendly.
- **BASIC (1960s):** Simplified Fortran, beginner-friendly, used timesharing.
- **ABC (1980s, Netherlands):**
  - Enforced indentation for block structure.
  - Built-in data structures (no need to implement basic algorithms like heapsort).
  - Focused on scripting and ease of use.
  - Flopped commercially, but ideas lived on.
- **Python:**
  - Designed as a scripting language with built-in data structures and enforced indentation.
  - Aimed to be simple, readable, and educational.
  - Influenced by frustration with poor-quality scripting tools (e.g., Perl).

---

5. Python Syntax and Indentation

5.1 Indentation Rules

- Blocks start with a colon (`:`):

```
if x > 0:
    print("Positive")
```

- All code in a block must be indented equally.
- Mixing tabs and spaces is discouraged; use spaces only.
- Copy-paste errors can cause indentation issues.

- Indentation is enforced by the interpreter; improper indentation leads to syntax errors.

Tabs vs Spaces

- Tabs can be interpreted differently (4 vs 8 spaces); avoid them.
- Use single ASCII spaces for indentation.

Metaprogramming Caveat

- Indentation-based syntax makes it harder to write programs that generate other programs (metaprogramming).

---

6. Numerical Types and Operations

6.1 Integers

- Arbitrary precision (limited by available memory).
- Example:

```
x = 10**1000  # Very large integer
```

6.2 Floats

- Limited to ~10^308; overflow produces `inf`.

```
float('inf') > 99999999  # True
```

6.3 Division

- `/` is floating-point division:

```
1 / 2  # 0.5
```

- `//` is integer (floor) division:

```
1 // 2  # 0
```

6.4 Complex Numbers

- Use `j` for the imaginary part:

```
1 + 2j
import cmath
cmath.sqrt(-1)  # returns 1j
```

- `cmath` for complex math, `math` for real math.

---

7. Strings in Python

7.1 Quotes

- Single `'abc'` or double `"abc"` quotes are interchangeable.
- Python displays with single quotes by default.

7.2 Escape Sequences

- `\n` for newline, `\t` for tab, etc.

7.3 Raw Strings

- Prefix with `r` to prevent escape sequence interpretation:

```
r"\n"  # literal backslash and n
```

7.4 Triple Quotes

- Use triple quotes for multi-line strings:

```
'''This is
```

a multiline string'''

```
- Triple double quotes (`"""..."""`) also allowed, but don't mix single and double.

---

## 8. Python Object Model

### 8.1 Every Object Has:
- **Identity:** via `id(obj)` (like a pointer/address, but not directly usable)
- **Type:** via `type(obj)`
- **Value:** the object itself

### 8.2 Immutability
- **Immutable:** Value cannot change (`int`, `str`, `float`, etc.)
- **Mutable:** Value can change (`list`, `dict`, etc.)

#### Example:
```python
a = 12
print(id(a))
a += 1
print(id(a))  # ID has changed
```
```

8.3 Variables vs Objects

- Variables are bindings to objects, not the objects themselves.
- Assignment copies references, not objects.

Example:

```python
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]
```

8.4 Aliasing and Identity

- Assigning one variable to another copies the reference.
- `is` checks identity (like pointer comparison in C++), `==` checks value equality.

Example Table:

Operation	Description
`a is b`	True if `a` and `b` are the same object
`a == b`	True if `a` and `b` have equal values

---

9. Python's Built-In Data Types Overview

9.1 `None`

- Singleton object representing "nothing" (like `null` or `nullptr`).

9.2 Numbers

- `int`, `float`, `complex`

9.3 Sequences

- Common: `str`, `list`, `tuple`
- Indexable, iterable

9.4 Mappings

- Primarily `dict`: key/value store

9.5 Callables

- Functions, methods, classes, lambdas

---

10. Python Sequences

10.1 Indexing

- `s[i]` returns the ith element (0-based)
- Raises `IndexError` if out of range (unlike C++, which is undefined behavior)

10.2 Negative Indexing

- `s[-1]` is the last element, `s[-2]` is the second-to-last, etc.
- Valid for `-len(s) <= i < len(s)`

10.3 Slicing

- `s[i:j]` returns a new sequence from index i up to (but not including) j
- `s[:j]` from start to j-1
- `s[i:]` from i to end
- `s[:]` is a copy of the entire sequence
- Slices are new objects, but elements are references to the same objects
- If `i == j`, returns empty sequence; if `i > j`, raises error

ASCII Diagram: Slicing

```
Sequence:  S = [A, B, C, D, E]
Indexes:    0  1  2  3  4
Negative:  -5 -4 -3 -2 -1

S[1:4] -> [B, C, D]
S[-3:-1] -> [C, D]
```

10.4 Common Sequence Operations

Operation	Description
`len(s)`	Number of elements
`min(s)`	Minimum element
`max(s)`	Maximum element
`list(s)`	Convert sequence to list

- `min`/`max` require comparable elements; error if types are mixed or sequence is empty.
- Sequences can be heterogeneous, but many operations expect homogeneity.

10.5 Strings Are Immutable

- Cannot assign to elements or slices of a string.
- To modify, convert to list, change, then join back to string.

---

11. Mutable Sequences: Lists

11.1 Assign to Element

```
lst[i] = value
```

11.2 Assign to Slice

```
lst[i:j] = [a, b]
```

- Can grow or shrink the list.
- Assignment replaces the slice with the new sequence.

11.3 Delete Items

```
del lst[i]         # Delete index
del lst[i:j]       # Delete slice
```

- Decreases the length of the list.
- `del` only works on mutable sequences.

11.4 List-Specific Methods

Method	Description
`append(v)`	Add element to end
`extend([v])`	Add multiple elements to end
`insert(i, v)`	Insert before index i
`pop()`	Remove and return last item
`pop(i)`	Remove and return item at index i
`count(v)`	Count occurrences
`index(v)`	Index of first matching item
`sort()`	In-place sort

- `append` is amortized O(1) due to over-allocation.
- `extend` is a batch append.
- `insert` and `pop` can operate at any index.
- `sort` is in-place and uses a stable algorithm (Timsort).

ASCII Diagram: List Append and Growth

```
Initial:   [A, B, C]
append(D): [A, B, C, D]

Internal (over-allocated):
[ A | B | C | D |   |   ]
                ^
              (unused slots)
```

Amortized Analysis of Append

- Python lists over-allocate memory (double when full).
- Most appends are O(1); occasional O(n) when resizing.
- Total cost of n appends: O(n).

---

12. Iteration and Comprehensions

12.1 Iterating Over Sequences

- Use `for x in sequence` to iterate over elements.
- Avoid manual indexing unless necessary.

12.2 List Comprehensions

- Concise way to build lists:

```
[f(x) for x in sequence]
```

- More Pythonic than manual loops.

---

13. Summary

This lecture introduced Python from both basic and intermediate perspectives. It covered:

- Python's core types and data structures
- The distinction between mutable and immutable objects
- Data parsing using list comprehensions and functions like `zip()`
- The unique object model of Python (identity, type, value)
- Sequences (strings, lists, tuples): indexing, slicing, mutation
- Efficiency of list operations (especially `append()` and amortized analysis)
- Pythonic idioms and reasoning for clean, efficient code

---

14. Additional Notes and Edge Cases

- **Strings and Lists:**
  - To convert a list of characters back to a string: `''.join(list_of_chars)`
- **Equality:**
  - `==` compares values, `is` compares identities.
- **Slicing with Steps:**
  - `s[i:j:k]` allows stepping; negative steps reverse direction.
- **Error Handling:**
  - Indexing out of range raises `IndexError`.
  - `min`/`max` on empty or non-comparable sequences raises errors.
- **List Growth:**
  - Assigning to `lst[len(lst):]` is equivalent to `append`.
- **Mutable vs Immutable:**
  - Mutating a mutable object does not change its identity.
  - You cannot change the type of an object after creation.

---

15. Visual Summary

Python Object Model (ASCII Diagram)

```
Variable   --->   Object (has id, type, value)
   |
   +--->   [id(obj)]
   +--->   [type(obj)]
   +--->   [value]

Assignment:
  a = obj1
  b = a   # b points to the same object as a

Aliasing:
  a = [1,2,3]
  b = a
  b.append(4)
  # a and b both see [1,2,3,4]
```

List Growth (ASCII Diagram)

```
[ A | B | C ]  --append(D)-->  [ A | B | C | D |   |   ]
```

2.3 Error Handling and Exceptions

Python uses exceptions to handle errors. If an error occurs, Python raises an exception and (unless caught) stops execution.

Example: Handling Exceptions

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ValueError:
    print("Not a valid integer!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Table: Common Python Exceptions

Exception	When It Occurs
`ValueError`	Invalid value (e.g., `int('abc')`)
`TypeError`	Wrong type (e.g., `1 + 'a'`)
`IndexError`	Sequence index out of range
`KeyError`	Dict key not found
`ZeroDivisionError`	Division by zero
`AttributeError`	Attribute not found
`ImportError`	Import fails
`FileNotFoundError`	File does not exist

- Use `raise` to throw exceptions manually.
- Use `finally` for cleanup code that always runs.

2.4 Functions: Definition, Arguments, and Scope

Defining Functions

```
def add(x, y=0):
    return x + y
```

- `x` is a required argument, `y` has a default value.

Argument Types

Type	Example	Description
Positional	`f(1, 2)`	Matched by position
Keyword	`f(x=1, y=2)`	Matched by name
Default	`def f(x, y=0)`	Uses default if not provided
*args	`def f(*args)`	Tuple of extra positional args
**kwargs	`def f(**kwargs)`	Dict of extra keyword args

Example: All Argument Types

```
def demo(a, b=2, *args, **kwargs):
    print(a, b, args, kwargs)
```

Variable Scope Diagram

```
[Global Scope]
    |
    +--[Enclosing Scope]
            |
            +--[Local Scope]
```

- Python uses LEGB rule: Local, Enclosing, Global, Built-in.

2.5 Mutable vs. Immutable Types

Type	Mutable?	Examples
int, float	No	1, 2.5
str, tuple	No	'abc', (1, 2)
list, dict	Yes	[1, 2], {'a': 1}
set	Yes	{1, 2, 3}

Pitfall: Default mutable arguments retain changes between calls.

```
def f(x, lst=[]):
    lst.append(x)
    return lst
f(1)  # [1]
f(2)  # [1, 2]  # Not what you expect!
```

- Use `None` as default and create inside function.

2.6 Advanced List Comprehensions

- With condition:

```
[x for x in range(10) if x % 2 == 0]  # [0, 2, 4, 6, 8]
```

- Nested:

```
[(x, y) for x in range(2) for y in range(2)]  # [(0,0), (0,1), (1,0), (1,1)]
```

Construct	Example	Output
List comprehension	`[x*x for x in range(3)]`	`[0, 1, 4]`
Map	`list(map(lambda x: x*x, range(3)))`	`[0, 1, 4]`
Filter	`list(filter(lambda x: x%2, range(5)))`	`[1, 3]`

- List comprehensions are generally more readable and Pythonic.

2.7 Dictionaries and Sets

Dictionaries (`dict`)

- Key-value store, fast lookup.

```
d = {'a': 1, 'b': 2}
d['c'] = 3
print(d['a'])  # 1
```

Method	Description

Method	Description
`d.keys()`	All keys
`d.values()`	All values
`d.items()`	All (key, value) pairs
`d.get(k, v)`	Get value or default
`d.pop(k)`	Remove and return value

Sets (`set`)

- Unordered, unique elements.

```
s = {1, 2, 3}
s.add(4)
s.remove(2)
```

Method	Description
`add(x)`	Add element
`remove(x)`	Remove element
`union(s2)`	Union with another set
`intersection(s2)`	Intersection with another set

2.8 String Formatting

- Old style: `'Hello %s' % name`
- `str.format()`: `'Hello {}'.format(name)`
- f-strings (Python 3.6+): `f'Hello {name}'`

Example Table

Method	Example	Output
`%`	`'%d %s' % (3, 'hi')`	`'3 hi'`
`.format()`	`'{} {}'.format(3, 'hi')`	`'3 hi'`
f-string	`f'{3} hi'`	`'3 hi'`

2.9 Modules and Imports

- Use `import module` to use standard library or your own code.
- Example:

```
import math
print(math.sqrt(16))  # 4.0
from math import pi
print(pi)
```

- Use `as` to alias: `import numpy as np`

2.10 Iterators and Generators

- Any object with `__iter__()` and `__next__()` is an iterator.
- Use `iter()` to get an iterator, `next()` to get next value.

Example: Manual Iteration

```
lst = [1, 2, 3]
it = iter(lst)
```

```
print(next(it))  # 1
print(next(it))  # 2
```

Generator Functions

- Use `yield` to create a generator.

```
def count_up(n):
    i = 0
    while i < n:
        yield i
        i += 1
for x in count_up(3):
    print(x)  # 0 1 2
```

2.11 Pythonic Best Practices and Anti-Patterns

Practice	Pythonic?	Example/Note
List comprehensions	Yes	`[x*x for x in xs]`
Manual indexing	No	`for i in range(len(xs))`
Use of built-in functions	Yes	`sum(xs)`, `min(xs)`
C-style loops	No	`for (int i=0; i<n; i++)`
Exception handling	Yes	`try/except`
Catch-all except (bare except)	No	`except:` (should specify exception)
Mutable default arguments	No	`def f(x, lst=[])` (use `None` instead)
Use of `is` for equality	No	Use `==` for value equality
Use of `is` for identity	Yes	Use `is` for singleton checks (`None`)
Explicit variable naming	Yes	`total_sum` vs. `ts`
Wildcard imports (`from x import *`)	No	Pollutes namespace, unclear origin

Client-Server Computing, Node.js, and Event-Driven Programming

1. Introduction

This lecture covers foundational topics in distributed computing, focusing on:

- Client-server models
- Introduction to Node.js
- Event-driven programming
- Performance and reliability in networked systems
- Comparative discussion of distributed computing architectures
- Historical context: from circuit switching to packet switching

---

2. Client-Server Architecture

Overview

A client-server model is a fundamental pattern in distributed systems where:

- A **server** hosts resources or services (e.g., databases)
- A **client** requests access to those services

Key Properties

- The application logic is split between client and server
- Runs over a network (often visualized as a line or cloud)
- The server maintains the authoritative state of the system
- The client acts as an interface, with little or no authoritative state

Real-World Example: Course Enrollment

- The server (e.g., registrar's database) stores the true state (who is enrolled)
- The client (e.g., your laptop/browser) only requests changes or queries state
- Only the server can authoritatively say if you are enrolled

Multiple Clients, Server as Mediator

- Many clients can exist simultaneously
- Clients communicate only with the server, not directly with each other
- Example: Two students try to enroll at the same time
    - The server must resolve the conflict (pick a winner, reject both, or—if buggy—accept both)

```
Client1
   \
    \
    [Network] — [Server]
    /
   /
Client2
```

What Can Go Wrong?

- New types of bugs: race conditions, state conflicts, server bottlenecks
- Server is a single point of failure
- Scaling issues as the number of clients grows
- **Network partition:** Clients may be unable to reach the server, causing split-brain or stale state
- **Server crash:** All clients lose access to the service

Failure Scenario Diagram

```
Client1   Client2
   \      /
    [Network]   [Server] (down)
         X
```

Model Comparison Table

Model	Centralized?	Fault Tolerance	Scalability	Consistency	Example Use Case
Client-Server	Yes	Low	Moderate	Strong	Web apps, databases
P2P	No	High	High	Weak/Varies	File sharing, BitTorrent
Master-Worker	Partial	Moderate	High	Strong	ML training, MapReduce

---

3. Event-Driven Programming

Definition

A programming paradigm where code responds to asynchronous events (e.g., web requests, file completions, database responses).

Synchronous vs. Event-Driven

Synchronous Program	Event-Driven Program
Linear execution	Loops + Handlers
Blocking I/O	Non-blocking I/O

The Event Loop

A core concept in environments like Node.js:

```
while (true) {
  event = getNextEvent();
  handleEvent(event);
}
```

Step-by-Step Event Loop (ASCII)

```
[Event Queue] -> [Event Loop] -> [Handler] -> [Done]
     ^                                |
     |________________________________|
```

Real-World Analogy: Restaurant Order System

- The chef (event loop) handles one order (event) at a time, quickly moving to the next as soon as possible. If a dish takes too long, the kitchen (server) gets backed up.

```
[Order Queue] -> [Chef] -> [Serve Dish]
     ^                        |
     |________________________|
```

Constraints of Event Handlers

- Must be **fast** (microseconds to a few milliseconds)
- **Never block** (no synchronous file or network reads/writes)
- Blocking causes delays, queued requests, and unresponsiveness

Bad Example (Blocking):

```
function handleEvent() {
  const result = readBigFileSync(); // BAD: blocks event loop!
}
```

Good Strategy (Non-blocking):

- Split long tasks into short, fast handlers
- Schedule continuation via next event trigger

Example:

```
function doOne(i) {
   doHeavyTask(i);
   scheduleNext(() => doOne(i+1));
}
```

Callback Hell Example

```
readFile('a.txt', (err, dataA) => {
  if (err) throw err;
  readFile('b.txt', (err, dataB) => {
    if (err) throw err;
    readFile('c.txt', (err, dataC) => {
      // ...
    });
  });
});
```

Solution: Promises/Async-Await

```
async function readFiles() {
  const dataA = await readFileAsync('a.txt');
  const dataB = await readFileAsync('b.txt');
  const dataC = await readFileAsync('c.txt');
}
```

Problems in Event-Driven Code

- Higher overhead per action (event dispatch, handler setup)
- Error handling is more complex (no exceptions across events)
- Loss of execution order (events may be handled out of order)
- Debugging and reasoning can be difficult

Multi-threading vs. Event-Driven

Feature	Event-Driven	Multi-threading
Concurrency Model	Single-threaded, non-blocking	Multiple threads, context switching
Shared State	No (single thread = no races)	Yes (needs locks to avoid races)
Performance Potential	Limited CPU parallelism	High parallel performance
Safety	No race conditions	Race conditions possible
Complexity	Moderate	High (synchronization)

Key Benefit of Event-Driven

- No need for locks
- Less prone to race conditions

Limitations

- No true parallelism (one core at a time)
- If an event handler crashes or loops, the whole system halts
- Not suitable for adversarial or untrusted code

Other Languages

- Ruby: EventMachine
- Python: Twisted, asyncio (non-blocking I/O)

Error Handling in Event-Driven Code

- Error handling is more complex than in synchronous code.
- No exception propagation across events: if an event handler fails, you must explicitly record the error state for future handlers to check.
- You cannot rely on try/catch to propagate errors between events.
- Engineering mindset: always ask "what can go wrong?" and plan for error cases.
- In practice, much of robust software engineering is about anticipating and handling errors.

---

4. Web Technologies: HTML, CSS, JavaScript, and the DOM

HTML (HyperText Markup Language)

- Describes the structure and content of web pages (declarative).
- Elements are organized in a tree structure (the DOM: Document Object Model).
- Elements can have attributes (name-value pairs) and can be normal, raw text, or void (self-closing).
- HTML is forgiving: browsers try to render as much as possible even with errors.

DOM Tree Example

```
<html>
  <body>
    <h1>Hello</h1>
    <p>World</p>
  </body>
</html>
```

CSS (Cascading Style Sheets)

- Describes the presentation (style) of HTML elements.
- Styles can be inherited and cascade through the DOM tree.
- Separation of content (HTML) and style (CSS) improves accessibility and maintainability.
- Multiple sources of styles: browser defaults, user preferences, author styles; priority rules determine which wins.

JavaScript

- Imperative programming language for dynamic behavior in web pages.
- Can manipulate the DOM, respond to events, and update content dynamically.
- Can be included inline or as external files.
- Modern web apps often use JavaScript frameworks (e.g., React, which uses JSX as a declarative extension for UI).

Declarative vs. Imperative

- HTML and CSS are declarative: describe what you want, not how to do it.
- JavaScript is imperative: describes how to perform actions.
- Good web design separates content, style, and behavior.

Data Exchange: JSON vs. XML

Feature	JSON	XML
Syntax	Lightweight, concise	Verbose, tag-based
Readability	Easy for humans/machines	Harder for humans
Data Types	Objects, arrays, numbers	Text, attributes, elements
Use Cases	Web APIs, config, storage	Legacy systems, documents
Comments	Not supported	Supported

- JSON (JavaScript Object Notation) is now the most common format for client-server data exchange (tree-structured, easy for JavaScript).
- XML is more verbose and less common in modern web apps.

Security Issues in Web Technologies

- **XSS (Cross-Site Scripting):** Malicious scripts injected into web pages.
- **CORS (Cross-Origin Resource Sharing):** Controls which domains can access resources.
- **CSRF (Cross-Site Request Forgery):** Tricks users into submitting unwanted actions.

---

5. Writing a Basic Node.js Web Server

Example Code

```
const http = require('http');
const ip = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello CS35L World\n');
});

server.listen(port, ip, () => {
  console.log(`Server running at http://${ip}:${port}/`);
});
```

Explanation

Line	Purpose
1	Load the HTTP module
2	Set the local IP address
3	Define the port number
5–9	Set up request handler (callback)
10–12	Start the server and define startup callback

- `require('http')`: Loads the HTTP module (can be named anything, but conventionally matches the module name)
- `ip = '127.0.0.1'`: Localhost IP; server only accessible from the same machine
- `port = 3000`: Port number (no distinction between integers and floats in JS)
- `http.createServer(...)`: Sets up a server with a callback for each request
- Callback parameters: `req` (request object), `res` (response object)
- `res.statusCode = 200`: HTTP status code (200 = OK)
- `res.setHeader('Content-Type', 'text/plain')`: Sets response header
- `res.end('Hello CS35L World\n')`: Ends response, sends data
- `server.listen(...)`: Starts server, with a callback for when it is ready
- Template literals (backticks) allow variable interpolation: `` `Server running at http://${ip}:${port}/` ``

Request Flow Diagram

```
[Client] -> [Request] -> [Node.js Server] -> [Handler] -> [Response] -> [Client]
```

What if the Server Blocks?

- If a synchronous/blocking call is made in the handler, all incoming requests are delayed until it finishes.
- This can make the server unresponsive and is a common pitfall in Node.js.

Demo

- Run: `node app.js`
- Access: `http://localhost:3000`
- Response: "Hello CS35L World"

---

6. Distributed Computing Models

1. Client-Server Model

- Centralized control and data
- Problems: scalability, single point of failure
- Example: Traditional web applications

2. Peer-to-Peer (P2P)

- Each peer has its own data and control

- No centralized server
- Pros: scalable, redundancy
- Cons: complex consistency, data propagation challenges
- Example: BitTorrent, blockchain

3. Primary-Secondary (Master-Worker)

- One primary/master controls computation
- Many secondary/worker nodes perform tasks
- Used in ML training, distributed databases
- Example: MapReduce, parameter servers in ML

Model	Use Case	Pros	Cons	Fault Tolerance	Consistency
Client-Server	Web apps, DBs	Simple	Scalability, SPOF	Low	Strong
P2P	File sharing, crypto	Scalable	Complexity	High	Weak/Varies
Master-Worker	ML, batch jobs	Control	Bottleneck at primary	Moderate	Strong

ASCII Diagrams

Client-Server:

```
Client1   Client2   Client3
   \        |        /
    \       |       /
     [Network]---[Server]
```

Peer-to-Peer:

```
[Peer1]---[Peer2]
   |        |
[Peer3]---[Peer4]
```

Primary-Secondary:

```
         [Primary]
        /   |   \
   [Sec1] [Sec2] [Sec3]
```

---

7. Distributed Systems Challenges

A. Performance Problems

Throughput

- Number of operations per second
- Optimization methods:
  - Handle events out-of-order
  - Prioritize cached contents
  - Balance workload

Latency

- Time between request and response
- Optimization methods:
  - Cache on client (or server)
  - Use faster protocols
  - Reduce payload sizes

Example: Browser Cache

- Caches previously fetched results
- Avoids round trips to server (especially important for distant servers)

B. Correctness Problems

Serialization (Expanded)

- Serialization means the system's behavior can be explained as if all actions happened in some serial order, even if they were actually interleaved or out of order.
- This is not a cure-all: the resulting computation must still be valid (e.g., not exceeding enrollment limits).
- Serialization is a way to justify out-of-order or parallel execution as long as the end result matches some valid serial order.
- If your application is buggy, serialization won't save you; it's a tool for reasoning about correctness, not a fix for logic errors.
- Example: In a bank, if you deposit and then withdraw, the system must not reorder these in a way that causes an overdraft fee if you had enough money.
- For a single client, actions should appear in the order they were issued; for multiple clients, the system can justify the result as if some serial order occurred.

Cache Staleness

- Client caches may become stale
- Solutions:
  1. **Validate cache**: Compare a unique server state ID (e.g., UUID) with the client cache; if mismatched, update
  2. **Tolerate staleness**: If the application can handle slightly outdated data (e.g., video games, weather apps)

Condition	Strategy
Cache critical	Validate with server
Cache negligible (e.g., news)	Ignore freshness slightly

C. CAP Theorem

- States that a distributed system can only guarantee two out of three: **Consistency**, **Availability**, **Partition Tolerance**.
- **Consistency:** Every read receives the most recent write or an error.
- **Availability:** Every request receives a (non-error) response, without guarantee that it contains the most recent write.
- **Partition Tolerance:** The system continues to operate despite network partitions.

Property	Description
Consistency	All nodes see the same data at the same time
Availability	Every request gets a response (no error)
Partition Tolerance	System works even if network is split

CAP Theorem Diagram

```
      Consistency
         /   \
        /     \
       /       \
Availability---Partition Tolerance
```

- In practice, systems must choose which two to prioritize during a partition.

D. Consistency Models

- **Strong Consistency:** All clients see the same data at the same time after a write.
- **Eventual Consistency:** All clients will eventually see the same data, but not immediately.
- **Causal Consistency:** Writes that are causally related are seen by all processes in the same order.

Model	Guarantees	Example Use Case
Strong Consistency	Immediate, global	Banking, transactions
Eventual Consistency	Eventually, not instant	DNS, social feeds
Causal Consistency	Order of related events	Collaborative editing

---

8. Deep Dive: Legacy Network Technology

Circuit Switching Model

- Used in traditional (1960s–1980s) telephone systems
- Mechanism: A **dedicated communication path** is established for each call
- Strengths: Guaranteed bandwidth, efficient for long stable conversations, high reliability
- Problems: Inefficient (wastes bandwidth), poor scalability, prone to failure if any intermediate node fails, no encryption (security by trust), motivated by Cold War defense needs

```
Phone A → Central Office → [Switch Network] → Central Office B → Phone B
```

- If any wire or node fails, the whole circuit is lost
- Inefficient: most of the time, the line is silent but still reserved
- Not mobile or portable
- Security: only as strong as the trust in the phone company
- Motivated the search for more robust, scalable, and survivable systems (e.g., in case of nuclear war)

---

9. Packet Switching (Modern Networks)

Overview

- Backbone of the modern Internet
- Data is split into **packets** (typically ~1000 bytes)
- Each packet routes independently via routers
- Routers make independent decisions for each packet
- If a node fails, packets are rerouted
- No need to reserve a full path; more efficient and resilient

```
[UCLA]--[Router]--[Phoenix]--[Chicago]--[Boston]
   \_________________________/
         (alternate path)
```

Key Terminology

Term	Explanation
Packet	A bundle of data (≤1500 bytes)
Router	Device routing packets
IP	Internet Protocol
Hop	Transition to next router

- Packets may arrive out of order; the receiver must reassemble them
- Routers use heartbeats or other mechanisms to detect failed nodes

Protocol Layers (The Internet Stack)

The modern Internet is built on a layered protocol stack:

```
+---------------------+
| Application Layer   |  (e.g., HTTP, Zoom, Email)
+---------------------+
| Transport Layer     |  (e.g., TCP, UDP, QUIC)
+---------------------+
| Internet Layer      |  (e.g., IP)
+---------------------+
| Link Layer          |  (e.g., Ethernet, Wi-Fi)
+---------------------+
```

- **Link Layer:** Handles direct connections between devices (e.g., Ethernet, Wi-Fi).
- **Internet Layer:** Handles routing of packets across networks (IP addresses, best-effort delivery, packets may be lost, duplicated, or reordered).
- **Transport Layer:** Provides end-to-end communication (TCP: reliable, ordered, error-checked; UDP: fast, unreliable, unordered; QUIC: modern, supports streams, used in HTTP/3).
- **Application Layer:** Protocols for specific applications (web, email, video, etc.).

Motivation for Packet Switching

- Developed for resilience (e.g., Cold War, Paul Baran at RAND): if nodes are destroyed, packets can be rerouted.
- No need to reserve a dedicated path; more efficient and robust than circuit switching.
- Each packet is routed independently; packets may arrive out of order and must be reassembled by the receiver.
- Routers use heartbeats to detect failed nodes and reroute traffic.

Real-World Implications

- TCP provides reliability, ordering, and error checking, but can cause "head-of-line blocking" (all subsequent data waits for a lost packet to be retransmitted).
- UDP is used for applications that can tolerate loss (e.g., video, voice, sensor data).
- HTTP/3 and QUIC are designed to avoid head-of-line blocking and improve performance for real-time applications (e.g., video conferencing in browsers).
- Protocols are layered: each layer builds on the services of the one below.

Circuit vs. Packet Switching Table

Feature	Circuit Switching	Packet Switching
Path	Dedicated, reserved	Dynamic, per-packet
Failure Impact	Call dropped	Packets rerouted
Efficiency	Low (idle time wasted)	High (shared bandwidth)
Scalability	Poor	Excellent
Security	Trust-based	Encryption possible

---

10. Conclusion & Key Takeaways

- Event-driven programming requires a different mindset: focus on performance **and** correctness
- Node.js enables scalable, single-threaded network apps
- Client-server is the dominant model, but alternatives (P2P, master-worker) exist and are important
- Performance tuning (latency, throughput) involves caching, async logic, and distributing load
- Serialization and cache validation are key to correctness
- Circuit switching vs. packet switching: understanding the evolution of network infrastructure
- CAP theorem and consistency models are central to distributed system design

---

11. Security and the Engineering Mindset

- Security is a major concern in distributed systems, but is often omitted in basic discussions. Always consider security implications (e.g., trust, encryption, adversaries).
- Engineering mindset: Always ask "what can go wrong?" and plan for failures, scaling, and adversarial conditions. Design for reliability, correctness, and performance.
- "Problems of success": Plan for scaling up if your system becomes popular.

Common Security Threats

Threat	Description	Mitigation
Man-in-the-middle (MITM)	Attacker intercepts/modifies traffic	Use TLS/SSL encryption
DDoS	Overwhelm server with traffic	Rate limiting, filtering, scaling
XSS	Inject malicious scripts into web pages	Input validation, CSP
CSRF	Trick user into unwanted actions	Tokens, same-site cookies
Data Breach	Unauthorized data access	Access controls, encryption

---

Appendix: HTTP Requests and Status Codes

Code	Meaning
200	OK
404	Not Found
500	Internal Server Error

Internet Technology and Web Services

1. Introduction: The Evolution of Internet and Web Technology

This lecture explores the foundations of Internet technology, focusing on the problems and solutions of packet-switched networking, the layered protocol model, and the development of the World Wide Web. We cover the technical and historical context, including protocol design, the TCP/IP stack, and the evolution of web services such as HTTP, HTTPS, and HTML.

---

2. Fundamental Concepts of Networking

2.1 Circuit Switching vs. Packet Switching

- **Circuit Switching**: Establishes a dedicated path between endpoints (e.g., traditional telephony). Predictable but inefficient for bursty or variable traffic.
- **Packet Switching**: Data is sent in small units (packets) that may take different paths to the destination. Used in the Internet. More efficient and robust, but introduces new challenges.

Analogy:

- Circuit switching is like reserving an entire lane on a highway for one car's trip.
- Packet switching is like sending many cars (packets) that can take any available lane and route to reach the destination.

ASCII Diagram:

```
Circuit Switching:
A =======[Dedicated Path]====== B

Packet Switching:
A ==\    /== Router ==\    /== B
      ==/             ==/
(Packets can take different routes)
```

Feature	Circuit Switching	Packet Switching
Path setup	Required	Not required
Reliability	More predictable	Less predictable
Efficiency	Low (fixed path)	High (dynamic routing)
Fault Tolerance	Low	High
Resource Sharing	Poor	Excellent

2.2 What is a Packet?

- A packet is a small unit of data (typically ~1 KiB) sent over the network.
- **Header**: Contains control information (destination, protocol type, etc.), used by routers and the network.
- **Payload**: Contains application data, used by the recipient application.
- The division between header and payload is sometimes a judgment call.

Example Packet Structure:

```
+-------------------+----------------------+
|      Header       |       Payload        |
+-------------------+----------------------+
| Src IP | Dst IP | ... |  Data (e.g., file, email) |
+-------------------+----------------------+
```

Field	Purpose
Header	Routing, protocol type, TO/FROM
Payload	Email text, file content, etc.

---

3. Protocols and Packet Switching Challenges

3.1 What is a Protocol?

- A protocol is an agreed-upon set of rules for message transmission and processing.
- Analogy: Diplomatic protocols—if you don't follow the rules, you get ignored.
- Protocols specify both message formats (headers, fields) and behaviors (how to respond to messages, what to do on error, etc.).
- Protocols are written as specifications (specs), not code.

Real-World Analogy:

- Like the postal system: envelopes (headers) carry letters (payloads), and everyone agrees on addressing and delivery rules.

3.2 Challenges in Packet Switching

Packet switching introduces several fundamental problems:

1. **Packet Loss**
   - Packets can be silently dropped, usually due to router buffer overflow or network congestion.
   - Loss rates can range from 1% to 50%+ under heavy load.
2. **Packets Received Out of Order**
   - Packets may arrive in a different order than sent, due to multiple network paths.
   - Can be problematic for applications requiring strict order (e.g., file uploads).
3. **Packet Duplication**
   - Rare, but possible due to network misconfigurations (e.g., routers misconfigured as bridges).
4. **Packet Corruption**
   - Bit errors from unreliable hardware or faulty links.
   - Requires checksums or error-detection mechanisms.

---

4. Layered Network Architecture

To manage complexity, protocols are organized into layers, each providing a different abstraction.

Layer	Functionality	Example Protocols
Application	High-level data exchange (e.g., HTTP, FTP, SMTP, HTML)	HTTP, FTP, SMTP, DNS
Transport	Reliable or fast delivery (e.g., TCP, UDP)	TCP, UDP
Internet	Routing across the network (e.g., IP)	IP (IPv4, IPv6), ICMP
Link	Data link between adjacent network nodes (e.g., Ethernet)	Ethernet, Wi-Fi, ARP

- Other models (e.g., OSI) define 7 layers, but the 4-layer model is most common in practice.
- Each layer adds its own header to the data (encapsulation).

ASCII Diagram: Layered Model

```
+-------------------+
|   Application     |
+-------------------+
|    Transport      |
+-------------------+
|    Internet       |
+-------------------+
|      Link         |
+-------------------+
```

Encapsulation Example:

```
Application Data
   ↓
[Application Header | Data]
   ↓
[Transport Header | Application Header | Data]
   ↓
[Internet Header | Transport Header | Application Header | Data]
   ↓
[Link Header | Internet Header | Transport Header | Application Header | Data]
```

4.1 The Link Layer

- Handles communication over a single, direct link.
- Protocols are hardware-specific (Ethernet, Wi-Fi, USB).
- Only cares about local addresses (e.g., MAC address).

Technology	Description
Ethernet	Wired LAN standard
Wi-Fi	Wireless communication
USB	Peripheral connections

4.2 The Internet Layer

- Main protocol: IP (IPv4 or IPv6).
- Responsible for global addressing and routing.
- Stateless, best-effort delivery (no guarantees).
- Introduces problems like loss, reordering, duplication.
- Introduces IP addresses (32-bit for IPv4, 128-bit for IPv6).

IP Address Breakdown Example:

```
IPv4: 192.168.1.42
Binary: 11000000.10101000.00000001.00101010
         |   192   |   168   |    1    |   42   |
```

- **Subnetting:** Divides address space for efficient routing and security (e.g., 192.168.1.0/24).
- **Private IPs:** Reserved for local networks (e.g., 10.0.0.0/8, 192.168.0.0/16).

4.3 The Transport Layer

- Provides reliable (or fast) delivery over the unreliable Internet layer.
- Introduces data channels (logical streams), reliability, ordering, error correction.
- Protocols: TCP (reliable), UDP (unreliable, fast).
- Solves the three main packet problems (loss, order, duplication).

Ports:

- Allow multiple services to run on one host (e.g., web server on port 80, SSH on port 22).

ASCII Diagram: Ports on a Host

```
[Host]
 |-- Port 22: SSH
 |-- Port 80: HTTP
 |-- Port 443: HTTPS
 |-- Port 25: SMTP
```

4.4 The Application Layer

- Protocols for specific services (HTTP, FTP, SMTP, DNS, etc.).
- Focuses on user/application concerns.

---

5. Internet Protocol (IP)

5.1 IPv4 Overview

- Introduced in 1983 (John Postel, UCLA alum).
- Connectionless: just delivers packets, no guarantee of delivery or order.

5.2 IPv4 Header Fields

Field	Description
Length	Packet size in bytes
Protocol Number	Defines encapsulated protocol (e.g., TCP = 6; UDP = 17)

Field	Description
Source IP	32-bit address of sender
Destination IP	32-bit address of recipient
TTL	Time-to-live (prevents infinite loops; decremented at each router hop)
Checksum	16-bit checksum for packet corruption detection

- IP addresses are written as four decimal numbers (e.g., 192.168.0.1), each representing 8 bits.
- TTL (Time To Live) prevents packets from looping forever.
- Checksum is for accidental corruption, not security.

5.3 IPv4 vs. IPv6

Version	Address Size	Release Year	Supports
IPv4	32-bit	1983	~4.3B addresses
IPv6	128-bit	1998	2^128 addresses

- IPv4: 32-bit address space (~4.3 billion addresses), now exhausted.
- IPv6: 128-bit address space, more header fields, future-proofing.
- IPv6 adoption: ~40% (IPv4 still ~60%) due to inertia and legacy systems.
- IPv6 is required in regions with insufficient IPv4 addresses (e.g., China).

5.4 Checksums and Reliability

- 16-bit checksum for error detection (not cryptographic).
- End-to-End Principle: Each layer should do its own error checking, not trust lower layers.
- Not sufficient for malicious actors, only for accidental errors.

End-to-End Principle Example:

- If you download a file, the application (e.g., browser) should verify the file's integrity (e.g., with a hash), not just trust the network.

---

6. Transport Layer Protocols

6.1 User Datagram Protocol (UDP)

- Designed by David Reed (MIT, 1980s).
- Very thin layer over IP; adds source/destination port numbers.
- Unreliable, unordered, no guarantees—application must handle loss, order, duplication.
- Used for streaming, DNS, telemetry, IoT (where speed > reliability).
- Example: IoT thermometer sends a UDP packet every 10 minutes; if lost, no big deal.

6.2 Transmission Control Protocol (TCP)

- Designed by Vint Cerf (UCLA) and Bob Kahn (Princeton).
- Reliable, connection-oriented, stream-based.
- Guarantees:
  - **Reliability**: Retransmits lost packets.
  - **Ordering**: Reassembles out-of-order packets.
  - **Error Checking**: End-to-end validation.
  - **Flow Control**: Adjusts sending rate to avoid network overload.
- Used for web traffic, file transfers, assignment submissions, etc.

Challenge	TCP Feature That Solves It	Description
Loss	Retransmission	Re-sends lost data
Out-of-Order	Reassembly	Orders packets before delivering to application
Duplication	Sequence numbers, ACK	Identifies and discards redundant data
Overflow	Flow Control	Sender adjusts speed based on network conditions

- Retransmission: Sender keeps packets in RAM until acknowledged.
- Reassembly: Recipient buffers and reorders packets before delivering to application.
- Flow control: Prevents flooding the network.

TCP Three-Way Handshake:

```
Client           Server
  | SYN  ------> |
  |     <------ SYN-ACK
  | ACK  ------> |
Connection established
```

---

7. Protocol Specification: How to Create a Protocol

- Protocols are written as specifications (not code).
- Must define:
  - Packet formats and headers (field order, required fields)
  - Expected behaviors (how to respond to valid/invalid requests)
- Analogy: Like the C++ language standard (C++23, C++26)—if you don't follow the spec, your code won't work.

Example: Simple Protocol Spec

- Message: [4 bytes length][UTF-8 string payload]
- Behavior: If length doesn't match payload, receiver discards message.

---

8. The Web: Fundamentals and Evolution

8.1 Web History and Components

- Invented by Tim Berners-Lee at CERN.
- Two main components:
  1. **HTTP** (Hypertext Transfer Protocol)
  2. **HTML** (Hypertext Markup Language)
- Original goal: Make it easy to share and navigate research papers.
- Relied on simple document formatting (HTML) and a simple application-layer protocol (HTTP over TCP).
- First web server: Berners-Lee's NeXT workstation at CERN ("Do not power off" sign).

8.2 HTTP: Hypertext Transfer Protocol

- Built atop TCP (originally, now also UDP/QUIC for HTTP/3).
- Request-response model: client sends a request, server sends a response.
- Each request/response has a header and a body (data).

Example HTTP/1.0 Request

```
GET / HTTP/1.0
<empty line>
```

Example HTTP/1.1 Response Header

```
HTTP/1.1 200 OK
Date: Tue, 01 Jan 2020 00:00:00 GMT
Server: Apache
Content-Length: 12345
Content-Type: text/html
Connection: close
```

Common HTTP Header Fields

Field	Purpose
Date	Timestamp of response
Server	Web server software (e.g., Apache)
Last-Modified	When resource last changed
E-Tag	Resource identifier/version
Accept-Ranges	Allow clients to request only part of a resource

Field	Purpose
Content-Length	Byte count of body
Connection	Whether connection should be closed after response
Content-Type	Media type (e.g., text/html, image/jpeg)

- E-Tag: Unique identifier for a version of a resource; used for caching and conditional requests.

8.3 HTTP Protocol Evolution

Version	Year	Key Features	Approx Usage
HTTP/1.0	1990s	One request per connection	Legacy
HTTP/1.1	1999	Persistent connections, Host headers, multiple requests per connection	~10%
HTTP/2	2015	Multiplexing, header compression, server push, pipelining, based on TCP	~59%
HTTP/3	2022	Built on UDP/QUIC, avoids TCP head-of-line blocking, better for real-time media	~32%

HTTP/1.1

- Adds persistent connections (multiple requests per TCP connection).
- Host header allows multiple domains per server.
- Improves network efficiency.
- HTTPS (HTTP over TLS) encrypts data for privacy and integrity.
- Tools: GnuTLS CLI, OpenSSL.

HTTP/2

- Launched 2015, ~59% adoption.
- Solves head-of-line blocking by multiplexing multiple streams over a single TCP connection.
- Features:
  - **Header Compression**: Reduces size of repetitive headers.
  - **Server Push**: Server can send multiple resources in response.
  - **Pipelining**: Multiple requests sent without waiting for responses.
  - **Multiplexing**: Interleaved transmission of multiple streams.

HTTP/1.1 vs HTTP/2 Multiplexing:

```
HTTP/1.1 (Pipelining):
Client: |--A--|--B--|--C--|
Server: |--A--|--B--|--C--|
(Responses must be sent in order)

HTTP/2 (Multiplexing):
Client:  |--A--|--B--|--C--|--D--|
Server:  |--B--|--D--|--A--|--C--|
(Responses can be interleaved)
```

HTTP/3

- Developed by Google (Jim Roskind), 2022, ~32% adoption.
- Built atop QUIC (UDP-based), not TCP.
- Motivation: Avoid TCP's head-of-line blocking, improve real-time streaming (e.g., Zoom).
- Features:
  - Streams with tolerable loss (partial loss tolerance).
  - Multiple streams between endpoints.
  - Encryption by default (TLS is integral).
  - Better performance over unreliable networks.

---

9. HTML: HyperText Markup Language

- Based on SGML (Standard Generalized Markup Language).
- HTML: Simplified, adds "hypertext" (links, interactivity).
- Uses angle-bracket tags (e.g., `<p>`, `<a href>`, `<h1>`, `<img src>`).
- Lowercase convention (vs. SGML's uppercase).

- DOM structure: Tree of nested elements.

DOM Tree Example:

```
<html>
  <head>
    <title>
  <body>
    <h1>
    <p>
```

Example HTML

```
<html>
  <head><title>My Web Page</title></head>
  <body>
    <h1>Hello, World!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

---

10. Additional Questions and Examples

1. **Why is pipelining/multiplexing useful?**
   - Reduces latency, especially for asynchronous data.
   - Lets users interact without waiting for all data.
2. **Why can TCP be inefficient for video?**
   - Retransmitted packets introduce delay/stutter; QUIC/UDP solves this.
3. **Why don't all use IPv6?**
   - Legacy hardware/software, institutional inertia, IPv4 still works.
4. **What is head-of-line blocking?**
   - One lost packet blocks all subsequent ones from being processed.

Summary

Layer	Main Protocols	Key Functionality
Application	HTTP, FTP, SMTP, DNS	User-facing services, data formats
Transport	TCP, UDP	Reliable/unreliable delivery, ports
Internet	IP, ICMP	Routing, addressing, fragmentation
Link	Ethernet, Wi-Fi, ARP	Local delivery, MAC addressing

This lecture introduced foundational concepts of Internet technology and packet-switched networking, focusing on the importance of protocol layers in handling reliability, order, and abstraction. It covered the four key network layers (link, internet, transport, application), the problems of packet loss, duplication, and disorder, and how these are abstracted away in the transport layer (TCP). The evolution of the web was discussed, from HTTP and HTML to modern protocols like HTTP/2 and HTTP/3, with their performance optimizations and trade-offs. The lecture also emphasized the importance of protocol specifications and the end-to-end principle in network design.

HTML, the DOM, CSS, JavaScript, JSX, and Python Dictionaries

1. HTML: Structure and Design Principles

1.1 Origins and Goals

- HTML is based on SGML (Standard Generalized Markup Language).
- SGML was designed for **portability** and **separation of content from form** (presentation).
- HTML inherits:
  - **Content**: What the user wants to convey.
  - **Form**: How content is presented (appearance/style).
- HTML is **declarative**: it specifies what should be displayed, not how to do it.

1.1.1 Semantic HTML

- **Semantic tags** (e.g., `<header>`, `<nav>`, `<main>`, `<article>`, `<footer>`, `<section>`, `<aside>`) describe the meaning of content, not just its appearance.
- **Benefits:**
  - **Accessibility:** Screen readers and assistive tech can better interpret the page.
  - **SEO:** Search engines understand page structure, improving ranking.
  - **Maintainability:** Code is easier to read and update.
- **Example:**

```html
<header>
  <h1>My Blog</h1>
</header>
<main>
  <article>
    <h2>Post Title</h2>
    <p>Content...</p>
  </article>
</main>
<footer>Copyright 2024</footer>
```

1.2 Declarative vs. Imperative

- **Declarative** (HTML): Describes what should appear (structure + content).
- **Imperative** (JavaScript): Describes how things should be done (logic + behavior).
- Original design: Prefer HTML/CSS for content/styling, use JavaScript only for behavior.
- In practice, this separation is often blurred.

1.3 Device Independence and Flexibility

- HTML aims for consistent rendering across devices (laptops, phones, etc.).
- Browsers have flexibility in rendering to improve usability and accessibility.

1.4 HTML Syntax and Elements

1.4.1 Elements and the DOM Tree

- An HTML element = a node in the document's tree structure (DOM).
- Syntax:

```html
<tagname attribute="value">Content</tagname>
```

- Browsers keep the DOM tree in memory; JavaScript manipulates it.
- **ASCII Diagram: DOM Tree Example**

```
<html>
  <body>
    <h1>Hello</h1>
    <p>World</p>
  </body>
</html>
```

```
DOM Tree:
html
 └── body
      ├── h1
      │    └── "Hello"
      └── p
           └── "World"
```

1.4.2 Nesting and Valid Syntax

- Tags must be properly nested:
  - Invalid:

    ```
    <a><b>…</a></b>
    ```

  - Valid:

    ```
    <a><b>…</b></a>
    ```

- Browsers are forgiving: they try to render as much as possible even with errors.

1.4.3 Optional and Void Tags

- Closing tags can sometimes be omitted if unambiguous.
- **Void elements**: Standalone, no children (e.g., `<br>`).
  - `<br>` is equivalent to `<br></br>`, but the short form is preferred.
- Some elements (rare) are **raw text**: can only contain text, not other elements.
- **More Examples:**
  - `<input>`, `<hr>`, `<meta>`, `<link>` are all void elements.
  - `<script>` is a raw text element (can only contain text, not HTML elements).
- **Edge Case:**
  - `<li>` closing tag can be omitted if followed by another `<li>` or if it's the last child.

1.4.4 Attributes

- Elements can have attributes: name-value pairs (strings).
- Syntax:

  ```
  <img src="image.jpg" width="500" height="300">
  ```

- Attribute names must be unique per element and appear in the opening tag.
- Browsers want to know attributes early for rendering.
- **Boolean attributes:**
  - Some attributes (e.g., `checked`, `disabled`) can appear without a value: `<input checked>` is valid and equivalent to `<input checked="checked">`.

1.4.5 Types of Elements

Type	Characteristics	Example Tags
Normal	Can contain mixed content/children	`<p>`, `<div>`
Raw Text	Can only contain text (rare)	`<script>`
Void	No content or children	` `, `<img>`

1.4.6 Catalogs and DTDs

- Early HTML used a **Document Type Declaration (DTD)** to specify:
  - Allowed elements, their types (void, raw, normal), allowed attributes, valid nesting.
- DTDs defined what trees were valid.
- Used in HTML 1–4, but couldn't keep up with browser innovation.

1.4.7 HTML5 and Living Standards

- HTML5 (~2008) introduced a "living standard" (constantly updated, no version freezes).
- Maintained at w3.org.
- Allows for rapid evolution and flexibility.

1.4.8 Generous Parsing

- Browsers parse incomplete/incorrect HTML to maximize content rendering.
- Contrast: C++ halts on syntax errors; HTML tries to show as much as possible.

---

2. XML vs HTML

2.1 XML Overview

- **XML** = Extensible Markup Language.
- Stricter than HTML: closing tags are mandatory, less forgiving.
- Used for data exchange, especially where correctness/validation is critical (e.g., government, social security).
- **XHTML**: XML-based HTML.
- XML is a generalization of HTML for tree-structured data.
- **Real-world XML use cases:**
  - **SOAP:** Web services protocol.
  - **RSS/Atom:** News feeds.
  - **SVG:** Vector graphics.
  - **Configuration:** `.plist`, `.pom`, `.xml` config files.

2.1.1 XML Tree Example

```
<book>
  <title>CS35L</title>
  <author>Jane Doe</author>
</book>

Tree:
book
 ├── title: "CS35L"
 └── author: "Jane Doe"
```

2.2 XML vs HTML Syntax

Feature	HTML	XML
Closing Tags	Often optional	Required
Error Tolerance	Very forgiving	Strict
Usage	Web pages (UI)	Structured data, APIs

- XML is used for domain-specific extensions and reliable data governance.

---

3. DOM: Document Object Model

3.1 Overview

- DOM = Tree of HTML (or XML) content as objects in RAM.
- JavaScript (and other languages) can access/manipulate the DOM.
- Nodes: elements, text, attributes, etc.

3.2 DOM Operations

- **Navigation**: Locate elements (e.g., `getElementById`).
- **Traversal**: Visit nodes to search/extract data.
- **Modification**: Change structure, add/remove elements.
- DOM APIs are language-agnostic, but JavaScript is dominant.

3.3 DOM Traversal and Manipulation Example

```
// Find an element by ID and change its text
const el = document.getElementById('greeting');
```

```
el.textContent = 'Hello, world!';

// Add a new element
const p = document.createElement('p');
p.textContent = 'New paragraph.';
document.body.appendChild(p);
```

3.4 DOM Tree Diagram

```
<html>
  └─ body
      ├─ h1
      └─ p
```

---

4. CSS: Cascading Style Sheets

4.1 Motivation

- Separates content (HTML) from presentation (style).
- Early HTML used direct styling (e.g., `<i>`, `<b>`), mixing content and presentation.
- Modern approach: semantic tags (`<em>`, `<strong>`) + CSS for style.
- Accessibility: semantic tags help screen readers, etc.

4.2 CSS Inheritance and Cascading

- Styles apply hierarchically (cascade down the DOM tree).
- Child elements inherit parent styles unless overridden.
- Priority scheme: browser defaults, user overrides, author styles.

Source	Description
Browser	Default styles
User	Custom overrides via browser settings
Author	Defined in the webpage/CSS files

4.2.1 CSS Specificity and Cascade

- When multiple rules apply, the browser uses **specificity** to decide which wins.
- **Specificity Table:**

Selector Type	Specificity Value
Inline style	1000
ID selector (#id)	100
Class (.class)	10
Element (div, p)	1

- The higher the value, the more specific.
- **Example:**

  ```
  p { color: blue; }         /* specificity: 1 */
  .highlight { color: red; } /* specificity: 10 */
  #main { color: green; }    /* specificity: 100 */
  ```

  If a `<p id="main" class="highlight">` exists, its color will be green.

4.2.2 CSS Rule Override Example

```
<style>
  p { color: blue; }
  .important { color: red; }
</style>
<p class="important">This is red.</p>
```

4.3 CSS Syntax and Usage

- Inline style:

```
<span style="font-variant: small-caps;">Styled Text</span>
```

- External stylesheet:

```
<link rel="stylesheet" href="styles.css">
```

- CSS syntax:

```
selector {
  property: value;
}
```

- CSS is more design-focused; collaboration between developers (logic) and designers (presentation).

---

5. JavaScript and JSX

5.1 JavaScript Basics

- Dynamic, imperative language embedded in HTML.
- More dynamic than Python: runtime decisions, harder to predict.
- Used for interactivity and logic in webpages.

5.1.1 JavaScript Event Handling

- JavaScript can respond to user actions (clicks, input, etc.) using event listeners.
- **Example:**

```
<button id="myBtn">Click me</button>
<script>
  document.getElementById('myBtn').addEventListener('click', function() {
    alert('Button clicked!');
  });
</script>
```

5.1.2 JavaScript and the DOM: Event Flow

- **Event bubbling:** Events propagate from the target element up to the root.
- **Event capturing:** Events can be intercepted on the way down.
- **ASCII Diagram: Event Bubbling**

```
[window]
   |
[document]
   |
[body]
   |
[div]
   |
[button] (event target)
```

Event bubbles up from button → div → body → document → window.

5.2 Embedding JavaScript

- **External file**:

```
<script src="hello.js"></script>
```

- **Inline code**:

```
<script>
  // code here
</script>
```

- **Pros of external scripts**:
  - Modularity, code reuse, easier updates, 3rd-party libraries, copyright compliance.
  - Cons: extra HTTP request, more complexity for small scripts.

5.3 DOM Manipulation and JSX

- JavaScript often manipulates the DOM directly (many API calls).
- **JSX**: Shorthand for DOM manipulation, used in React and similar frameworks.
  - JSX is preprocessed into JavaScript function calls.
  - Example:

```
const header = <h1 lang="en">CS35L Homework</h1>;
ReactDOM.render(header, document.getElementById("root"));
```

  - JSX supports embedding JavaScript expressions in curly braces, and deep nesting.
  - You can interpolate variables:

```
const lang = "en";
const course = "CS35L";
const header = <h1 lang={lang}>{course} Homework</h1>;
```

- JSX is not required; you can use plain JavaScript DOM APIs, but JSX is more concise.
- **JSX Transformation Example:**

```
// JSX
const el = <h1>Hello</h1>;
// Compiles to
const el = React.createElement('h1', null, 'Hello');
```

---

6. Client-Server Data Transfer: JSON and XML

6.1 JSON: JavaScript Object Notation

- Lightweight, tree-structured data-interchange format.
- Common in web APIs and async data loading.
- Readable/writable in JavaScript; built-in support.
- **JSON is to JS as XML is to HTML**.

Example JSON:

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" }
      ]
    }
  }
}
```

6.1.1 Parsing JSON Safely

- **Never use `eval` to parse JSON!**
- Use `JSON.parse()` in JavaScript:

```
const jsonString = '{"a": 1, "b": 2}';
const obj = JSON.parse(jsonString); // {a: 1, b: 2}
```

- **Security risk:** `eval` can execute arbitrary code.

6.1.2 Fetching JSON Example

```
fetch('/api/data')
  .then(response => response.json())
  .then(data => {
    console.log(data);
  });
```

6.1.3 Data Flow Diagram: Server to Client

```
[Server] --(JSON/XML)--> [Network] --(parse)--> [Browser/JS]
```

6.2 JSON vs XML: Tree Representation

Feature	JSON	XML
Label Position	On arcs (keys)	On nodes (element names)
Syntax	Curly braces/arrays	Tags with attributes
Parsing	Native in JavaScript	Requires XML parser

6.3 Comparison Table

Feature	JSON	XML
Syntax Simplicity	Simpler	Verbose
Readability	Easier horizontally	Harder
Data Orientation	Keys on arcs	Tags on elements (nodes)
Parsability	Built into JavaScript	Requires external library

Format	Description	Example Usage
XML	Verbose, hierarchical	Government, legacy systems
JSON	Lightweight, JavaScript-native	Modern web applications

---

7. Browser Rendering Pipeline

7.1 Stages

1. Parse HTML → create DOM
2. Apply CSS → compute layout
3. Render → display pixels

7.1.1 Reflow vs. Repaint

- **Reflow:** Changes to layout (e.g., adding/removing elements, changing size) cause the browser to recompute positions and geometry for part or all of the page.
- **Repaint:** Changes only to appearance (e.g., color, visibility) without affecting layout.
- **Reflow is more expensive than repaint.**

Diagram: Reflow vs. Repaint

```
[DOM Change]
   |
   v
[Reflow?] --yes--> [Recompute layout] --+--> [Repaint]
   |                                    |
   no                                   |
   |                                    v
   +-------------------------------> [Repaint only]
```

7.2 Optimization Strategies

- **Lazy rendering**: Start showing content before full parsing.
- **Skip off-screen elements**: Don't render or execute JS for elements not visible yet.
- **Parallel execution**: JavaScript may run while parsing/rendering continues.
- Browsers render partially parsed HTML for responsiveness.
- **Debounce/throttle events:** Limit how often expensive operations run (e.g., on scroll/resize).
- **Minimize layout thrashing:** Batch DOM reads/writes to avoid repeated reflows.

7.3 JavaScript and Rendering

- JavaScript execution may be delayed:
  - If DOM is not visible (e.g., below-the-fold elements).
  - If rendering is incomplete.
- JavaScript and rendering can interleave (race conditions possible if DOM is mutated before render).
- **Self-modifying DOM**: JavaScript can modify the DOM during load.
- **Deferred execution**: JS for hidden elements may not run immediately.

7.4 Performance Tips

- Be careful where `<script>` tags are placed; improper placement affects load speed and UX.
- Test on variable latency networks; fast local networks may hide timing issues.

---

8. Python: Dictionaries and Data Types

8.1 Mapping Types – `dict` (Dictionaries)

- Store key-value pairs.
- Keys must be hashable and immutable; values can be any type.
- Dictionaries are glorified hash tables; keys must be immutable for hashing to work.

8.1.1 Hashability and Dict Keys

- **Hashable:** An object is hashable if it has a fixed hash value for its lifetime and can be compared for equality.
- **Valid keys:** Immutable types (str, int, float, tuple of immutables).
- **Invalid keys:** Mutable types (list, dict, set).
- **Example:**

```python
d = {}
d[(1,2)] = 'ok'   # valid
d[[1,2]] = 'bad'   # TypeError: unhashable type: 'list'
```

8.2 Syntax & Operations

```python
d = {"a": 1, "b": 2}
d["c"] = 3            # Add item
print(d["a"])         # Access value
del d["b"]            # Remove item
```

8.3 Core Dictionary Operations

Method	Description
`len(d)`	Number of key-value pairs

Method	Description
`d.clear()`	Remove all pairs
`d.copy()`	Shallow copy of dictionary
`d.items()`	View object of (key, value) tuples
`d.keys()`	View object of keys
`d.values()`	View object of values

- `.items()`, `.keys()`, `.values()` return **view objects**:
  - Dynamic: reflect changes in the dictionary.
  - Not independently mutable; just a window into the dict.

8.4 Copying vs Assignment

- **Assignment**: `e = d` (both refer to the same dict; changes via one are seen by the other).
- **Copy**: `e = d.copy()` (new dict, but values are shared if mutable).
- **Shallow copy**: Only the dict is copied; mutable values are shared.
- **Deep copy**: Use `copy.deepcopy()` to recursively copy everything.

8.4.1 Shallow vs Deep Copy Table

Copy Type	What is copied?	Mutable values shared?
Assignment	Reference only	Yes
Shallow copy	Dict only	Yes
Deep copy	Dict and all contents	No

8.5 Dictionary Ordering (Python 3.7+)

- Insertion order is preserved.
- Reassigning a key does not affect its position.
- Deleting and re-adding a key puts it at the end.
- Order matters for reproducibility and debugging.

8.6 Iterating Over Dictionaries

```
for key in d:
    print(key, d[key])

for key, value in d.items():
    print(key, value)
```

- You can convert `.items()`, `.keys()`, `.values()` to lists for random access.

8.7 Performance and Theory

- Dictionary operations are **O(1)** (amortized) due to hashing, as long as memory is finite (word size is fixed).
- If memory could grow without bound, hash table operations would be O(log n) due to word size growth (theoretical note).
- Python dictionaries are not simple hash tables; they maintain insertion order for reproducibility.

Lecture 10: Version Control and Backups

1. Introduction: Why Version Control?

- **Version control** is about more than just tracking code changes; it's fundamentally about managing history and enabling recovery from mistakes or failures.
- **Backups** are a general concept: you want to know not only what you have today, but what you had yesterday, last week, last year, etc.
- **Version control** can be seen as a specialized backup system for source code, but backups are needed for many things beyond code.
- **Analogy:** Backups are like a time machine for your entire house; version control is a time machine for your work desk, letting you see every change you made to your projects.

2. Backups vs. Version Control: Roles and Responsibilities

- **Traditional division:**
  - *Developers*: Innovate, create new things, push the envelope.
  - *Operations staff*: Maintain stability, ensure nothing is lost, manage backups.
- **DevOps approach:**
  - Blurs the line between development and operations; everyone is responsible for both innovation and maintenance.
  - DevOps teams manage their own backups and deployments, streamlining processes.
- **Key Difference:**
  - Backups focus on disaster recovery and data retention.
  - Version control focuses on tracking changes, collaboration, and project history.

Summary Table: Backup vs. Version Control

Aspect	Backup System	Version Control System
Primary Purpose	Disaster recovery, data retention	Track changes, enable collaboration
Granularity	Whole files, directories, or disks	Individual changes (commits)
History Depth	Snapshots at intervals	Complete, fine-grained history
Collaboration	Not designed for collaboration	Built-in support (branches, merges)
Recovery Model	Restore to previous state	Revert/checkout any commit
Metadata Tracked	File metadata, timestamps	Author, message, diffs, metadata
Typical Use Cases	System restore, accidental deletion	Software development, document editing

Real-World Scenario: Backup vs. Version Control

Suppose you are working on a research paper. If you accidentally delete the file, a backup system allows you to restore the last saved version. However, if you want to see how your paper evolved, compare different drafts, or collaborate with co-authors, a version control system like Git is essential. Version control lets you track every change, revert to any previous version, and merge contributions from multiple people.

3. Why Do We Need Backups? (Failure Models)

- **Purpose:** To recover from failures (hardware, user error, attacks, bugs, insider threats).
- **Failure model:**
  - You must know what kinds of failures you want to protect against to design a good backup strategy.
  - Without a clear failure model, you risk wasting resources or missing critical risks.

Common Failure Types and Examples

Type	Description	Example
Hardware	Drive fails (flash or disk); e.g., 1–2% annualized failure rate (AFR) for flash, ~2–4% for disks.	Disk crash, SSD wear-out
User Error	Accidental deletion or overwriting of files.	`rm -rf *` in wrong directory
Attacker Action	External or internal attackers delete or modify data.	Ransomware, disgruntled employee
Software Bug	Bugs that corrupt or delete data.	Faulty update script wipes configs
Insider Threat	Trusted users (e.g., with shared accounts) tamper with data.	Admin deletes logs to hide activity

- **AFR (Annualized Failure Rate):**
  - E.g., 1–2% for flash drives in server environments; higher for disks.
  - Large installations (Amazon, Google) see these rates in practice.
  - **Correlated failures:**

        - Multiple drives from the same batch may fail together (manufacturing defect).
        - Power surges, firmware bugs, or environmental hazards can cause simultaneous failures.

4. Simple Backup Models: Mirroring

- **Mirroring:** Keep two identical copies of all data.
    - *Pros:* Simple, protects against single drive failure.
    - *Cons:* Doubles storage cost, assumes independent failures (not always true—bad batches, shared environments).
- **AFR for Mirroring:**
    - Naive: Square the AFR (e.g., 0.01 × 0.01 = 0.0001) for both drives failing in a year (if independent).
    - Realistic: Must consider correlated failures and recovery time (e.g., time to replace a failed drive).
    - Actual risk is lower if failed drives are replaced promptly, but higher if failures are correlated.
    - **Revised AFR Calculation (Considering Recovery Time):**
        - If a drive fails and is replaced within a certain time window (e.g., 10 hours), the probability of data loss is the probability that the second drive fails during that window.
        - The revised formula is:
            - `AFR ≈ AFR_single * (AFR_single * (drive replacement time / 1 year))`
            - For example, if AFR_single = 0.01 (1%), and replacement time is 10 hours:
                - `AFR ≈ 0.01 * (0.01 * (10 / (365*24)))`
        - This is much lower than simply squaring the AFR, provided prompt replacement and independent failures.

Mirroring and Recovery Window (Detailed ASCII Diagram):

```
Time 0:         Drive 1 fails
                |
                v
+---------------+-------------------+-------------------+
| Both drives OK| Only Drive 2 OK   | Both drives failed|
+---------------+-------------------+-------------------+
                |<-- Replacement -->|
                |   Window (e.g. 10h)
+---------------+-------------------+-------------------+
| Data safe     | Data at risk      | Data lost         |
+---------------+-------------------+-------------------+
```

Data is only at risk if the second drive fails during the replacement window after the first failure. If both drives are in the same enclosure and a power surge occurs, both may fail at once—mirroring does not protect against this.

5. Recovery Models: Testing Your Backups

- **Don't just model failure—model recovery!**
    - Many engineers forget to test recovery.
    - Example: Simulate a drive failure by unplugging a device and see if you can recover.
    - *Personal test:* Lose your laptop for a weekend—can you still get your work done?

> **Philosophy:**
> "Don't just think it, do it." Test your recovery model in practice, not just in theory.

Common Mistakes Table:

Mistake	Consequence
Only model failure, not recovery	Backups may be useless in a real emergency
Forget to test restore process	Can't recover when needed
Assume backups work	False sense of security
Delete incrementals blindly	Lose ability to restore full history

Backup Testing Checklist:

- ☐ Can you restore a single file?
- ☐ Can you restore an entire directory?
- ☐ Can you restore to a new machine?
- ☐ Are all permissions, links, and metadata preserved?
- ☐ Are encrypted files still accessible?

6. What to Back Up?

- **Data to back up:**
  - File contents
  - File names
  - Directory structure
  - Permissions, ownership, access rights (as shown by `ls -l`)
  - Timestamps (modification time)
  - Inode/link count (for hard links)
  - Symbolic link contents and file types
  - Backup meta-information (version number, timestamp, creator)
- **Delicate files:**
  - Passwords, encryption keys—handle with care! Don't store in backups unless secured; many breaches have resulted from mishandling these.
- **Be thorough:**
  - If `ls -l` or `ls -li` shows it, you probably need to back it up.
  - Easy to forget things like hard links or symlink contents—be meticulous.

File System Metadata Table:

Metadata	Why It Matters
Permissions	Security, access control
Ownership	Determines file access
Timestamps	Auditing, synchronization
Hard links	Prevents data loss, preserves structure
Symlinks	Maintains references, avoids breakage
File types	Correct restoration

7. Backup Strategies: Periodic, Incremental, and Rotational

- **Full backup:** Snapshot of the entire system at a point in time.
- **Incremental backup:** Only changed files since the last backup.
  - *Pros:* Saves space and time.
  - *Cons:* Restoring requires all incrementals to be intact; more fragile.
- **Rotational/Consolidation:**
  - Keep frequent (e.g., hourly) backups for recent data, less frequent (e.g., daily/weekly) for older data.
  - Combine adjacent incrementals to save space.
  - *Caution:* Don't delete incrementals blindly—combine them to avoid losing the ability to restore.

Backup Rotation/Retention Flow:

```
+-------------------+
|  Recent Backups   |
| (hourly, daily)   |
+-------------------+
          |
          v
+-------------------+
|  Older Backups    |
| (weekly, monthly) |
+-------------------+
          |
          v
+-------------------+
|   Oldest Backups  |
| (archived, tape)  |
+-------------------+
```

As time passes, frequent backups are consolidated or deleted, retaining only less frequent snapshots for older data.

Scenario:

- You have hourly backups for the last 24 hours, daily for the last month, and monthly for the last year. If you need a file from 6 months ago, you restore from the monthly backup.

8. Efficient Backup Techniques

- **Differencing (diff/patch):**
  - Store only the changes (diffs) between file versions, not the whole file.
  - Example:

    ```
    diff -u old_file new_file > changes.diff
    patch old_file < changes.diff
    ```

  - *Pros:* Saves space, especially for large files with small changes.
  - *Cons:* All diffs must be intact to restore; more fragile.
- **Data grooming:**
  - Don't back up files you can easily recreate (e.g., `.o` files, build artifacts).
  - Use scripts to exclude such files before backup.
  - *Note:* Data grooming often requires user input or customization to avoid missing important files.
- **Deduplication:**
  - Avoid storing identical data blocks multiple times.
  - Block-level deduplication: Only unique blocks are stored; identical blocks are referenced by pointers.
  - *Copy-on-write (COW):* New blocks are only created when data is changed.
  - *Caveat:* If a shared block is lost, all referencing files are affected. Not a substitute for true redundancy—always maintain physically separate copies for critical data.
- **Compression:**
  - Use lossless compression (e.g., gzip) to reduce backup size.
  - *Tradeoff:* Slower backup/restore, but often negligible with modern CPUs.
- **Encryption:**
  - Encrypt backups to protect against unauthorized access (especially if stored offsite or in the cloud).
- **Staging (multi-tier backups):**
  - Use multiple backup layers: e.g., flash → disk → tape.
  - Tape is slowest/cheapest, disk is mid-tier, flash is fastest/most expensive.
- **Multiplexing:**
  - Use a single backup device for multiple data sources to save on hardware costs.
- **Checksums:**
  - Compute and store checksums (e.g., SHA256) for backup data.
  - Later, verify backups by comparing checksums—cheap and effective for integrity checking.
- **Format conversion:**
  - Use lossless compression (e.g., gzip) to reduce backup size.
  - *Note:* Format conversion (e.g., CRLF vs. LF) is often a pain, but Git can handle it automatically for you, saving you from manual headaches.

Technique Comparison Table:

Technique	Space Savings	Speed Impact	Risk/Tradeoff
Deduplication	High	Low	Shared block loss affects many files
Compression	Medium	Low/Medium	Slower restore, but usually minor
Encryption	None	Low/Medium	Key management is critical

9. Caching and Backup Reliability

- **Cache vs. backup:**

- Caches are small/fast, backups are big/slow.
- Don't rely on caches as backups; treat them as primary data, not backup copies.

- **Incremental/diff risk:**
  - If any incremental or diff is lost/corrupted, you may lose the ability to restore.
  - Reliability of storage is critical for incremental/diff-based backups.

  Scenario:

  - Your browser cache lets you quickly reload a page, but if your hard drive fails, the cache is lost. Only a backup can restore your data.

10. From Backups to Version Control: Developer Needs

- **Backups alone are not enough for software development.**
- **Version control systems (VCS) provide:**
  - Efficient lookups of old versions (must be fast, not just possible).
  - File rename tracking: Recognize when a file is renamed, not just deleted/created.
  - Metadata about development history (e.g., tags, commit messages, authorship).
  - Atomic commits: Group multiple changes into a single, all-or-nothing operation. Prevents inconsistent intermediate states.
  - Hooks: Custom scripts (pre-commit, post-commit, etc.) to enforce policies or automate tasks.
  - Format conversion: Handle line ending differences (e.g., CRLF vs. LF) between platforms.
  - Signed commits: Cryptographically verify authorship and integrity of changes.

  Backup vs. Version Control Table:

Feature	Backup System	Version Control System
Fast lookup of history	No	Yes
Track renames/moves	No	Yes
Metadata (commits, etc)	No	Yes
Atomic changes	No	Yes
Collaboration tools	No	Yes
Signed/authenticated	No	Yes

11. Git: The Most Popular Version Control System

- **Two key concepts in Git:**
  1. **Object database:**
     - Stores the complete history of your project as a database of objects (blobs, trees, commits, tags).
     - Persistent, shared with other developers.
  2. **Index (cache):**
     - Stores your planned changes (the "staging area").
     - Private to you, not shared until committed.
     - Allows you to prepare and review changes before committing to history.

- **Workflow:**
  - You can spend time perfecting your index (staging area), then commit to the object database for a clean, understandable history.
  - Clean histories are easier for collaborators to review and accept.

- **Git clone:**
  - Copies the object database, creates a trivial index, and sets up working files.
  - Local clones are fast and use hard links for deduplication.

  Git Data Flow (Detailed):

```
+----------------+      git add      +-------------+      git commit      +-------------------+
| Working Files  | ---------------->  |   Index     |  ----------------->  |  Object Database  |
| (your edits)   |                   | (Staging)   |                      | (History)         |
+----------------+                   +-------------+                      +-------------------+
       |                                   |                                     |
       |  git checkout                     |  git reset                          |
       +-----------------------------------+-------------------------------------+
```

You edit files, stage changes in the index, and commit them to the object database (history). You can also reset or checkout to move between these states.

12. Real-World Example: CSNet Snapshots

- **Snapshot directories:**
  - Many systems (e.g., CSNet) provide snapshot directories with hourly, daily, and weekly backups.
  - Implemented using block-level deduplication and snapshotting.
  - Users can recover files from up to two weeks ago, with all metadata preserved.

  Scenario:

  - You accidentally delete a file. You navigate to the snapshot directory, find yesterday's version, and restore it with all permissions and links intact.

13. Summary: Best Practices

- Always have a clear failure and recovery model.
- Test your backups—don't just assume they work.
- Be thorough in what you back up (metadata, links, permissions, etc.).
- Use efficient, cost-effective strategies (incrementals, deduplication, compression, encryption, checksums).
- For software development, use a version control system that supports atomic commits, metadata, hooks, and efficient history access.
- Understand the distinction between backups and version control, and use both appropriately.

Best Practices Checklist:

- [ ] Have a documented failure model
- [ ] Regularly test recovery
- [ ] Back up all necessary metadata
- [ ] Use efficient backup strategies
- [ ] Use version control for code and documents
- [ ] Secure sensitive data in backups
- [ ] Periodically review and update backup/versioning procedures

Lecture 11: Version Control Continued

1. Introduction

This lecture continues the exploration of Git as a case study in version control, focusing on the workflow, commit lifecycle, file states, and practical commands. Emphasis is placed on understanding the relationship between the working directory, the index (staging area), and the object database (history), as well as the importance of commit messages, commit IDs, and repository configuration.

---

2. Git's Three States: Working Directory, Index, and Object Database

- **Working Directory (Working Files):**
  - The files you actively edit. These are visible to compilers, test programs, and other tools.
  - Changes here are not yet tracked by Git until staged.
- **Index (Cache/Staging Area/Planned Future):**
  - Where you stage files for the next commit using `git add`.
  - Represents your "planned future"—what you intend to commit next.
  - You can stage multiple changes before committing.
- **Object Database (History):**
  - Contains all committed snapshots (commits) of your project.
  - Commits are immutable; you cannot change history, only add new commits.

ASCII Diagram: Git State Transitions

```
[Working Directory] --git add--> [Index (Staging Area)] --git commit--> [Object Database (History)]
      ^                            |                                  |
      |                            |                                  |
      +-----------git checkout------+                                  |
      |                                                               |
      +----------------------------git reset-------------------------+
```

Table: Comparing Git States

State	Description	Typical Commands
Working Directory	Files you edit directly	`git status`, edit files
Index (Staging)	Planned changes for next commit	`git add`, `git reset`
Object Database	All committed snapshots (history)	`git commit`, `git log`

Example: File Lifecycle

Suppose you have a file `hello.txt`:

```
echo "Hello" > hello.txt
git status
# hello.txt is untracked

git add hello.txt
git status
# hello.txt is staged

git commit -m "Add hello.txt"
git status
# Working directory clean
```

State	File Status	Command to Move to Next State
Working Directory	Untracked/Modified	`git add <file>`
Index (Staging Area)	Staged	`git commit`

State	File Status	Command to Move to Next State
Object Database	Committed	(N/A, file is now in history)

3. Comparing File States with `git diff`

Command	Compares	Description
`git diff`	Working directory vs. staged (index)	Shows changes not yet staged
`git diff --cached`	Staged (index) vs. last commit (HEAD)	Shows what is staged but not yet committed
`git diff HEAD`	Working directory vs. last commit	Shows all changes since last commit
`git diff <commit>`	Working directory vs. specific commit	Compare with any previous commit

- `HEAD` is a symbolic name for the most recent commit.
- Use `git log` to view commit history and obtain commit hashes.

Example: `git diff` Variants

Suppose you edit `hello.txt` after committing:

```
echo "World" >> hello.txt
git add hello.txt
echo "!" >> hello.txt
```

Command	What it shows	Example Output (abbreviated)
`git diff`	Changes in working directory not staged	Shows addition of "!"
`git diff --cached`	Changes staged for commit	Shows addition of "World"
`git diff HEAD`	All changes since last commit	Shows both "World" and "!"

4. Staging and Committing Changes

- **Staging:**
  - `git add <filename>` stages a file for the next commit.
  - You can stage multiple files before committing.
- **Committing:**
  - `git commit` opens an editor for a commit message.
  - The commit message should explain *why* the change was made, not just *what* changed.
  - Good commit messages are crucial for future understanding and collaboration.
  - Some projects enforce commit message formats using hooks (e.g., Diffutils).
- **Hooks and Commit Rejection:**
  - Projects may use hooks to enforce commit message standards.
  - If your message doesn't match, the commit may be rejected.

Example Workflow

```
# Edit file
echo "Edit" >> file.txt

# Stage file
git add file.txt

# Commit staged changes
git commit -m "Describe the edit"
```

Good Commit Message Example

```
Add user authentication to login page

- Implemented password hashing
- Added login error messages
- Updated user model
```

---

5. Commit IDs and SHA-1 Checksums

- **Commit IDs:**
  - Each commit is identified by a SHA-1 hash (160 bits, 40 hex characters).
  - The hash is computed from the commit's contents and metadata (author, date, message, parent commit(s)).
  - Changing any part of a commit results in a new hash.
- **Immutability:**
  - Commits are immutable; you cannot change a commit in place.
  - If you change a commit, a new commit with a new hash is created.
- **Collision Probability:**
  - The chance of two different commits having the same SHA-1 hash is astronomically low.
- **Why Not Sequential IDs?**
  - Sequential IDs require a central server and are problematic for distributed systems.
  - SHA-1 allows decentralized, parallel development without conflicts.

Real-World Analogy: SHA-1 Commit IDs

Think of a commit as a unique fingerprint for a snapshot of your project. If you change even a single character in a file, the fingerprint (SHA-1 hash) changes completely. This ensures that every version is uniquely identified and tamper-evident, much like a tamper-proof seal on a document.

How a Commit Hash is Computed

A commit hash is generated from:

- The contents of the commit (tree, parent, author, date, message)
- Example (simplified):

```
commit 1a2b3c4d...
tree 9e8f7g6h...
parent 0z9y8x7w...
author Alice <alice@example.com> 1712345678 -0700
committer Alice <alice@example.com> 1712345678 -0700

Add feature X
```

Changing any part (even a space in the message) changes the hash.

---

6. Amending Commits

- `git commit --amend`:
  - Allows you to replace the most recent commit with a new one (e.g., to fix mistakes or update the message).
  - The amended commit gets a new SHA-1 hash.
  - Use with caution; beginners are encouraged to let history accumulate naturally.

Example: Amending a Commit

```
git commit -m "Initial commit"
# Realize you forgot a file
git add forgotten.txt
git commit --amend
```

Before:

```
* 1234abcd Initial commit
```

After:

```
* 5678efgh Initial commit (now includes forgotten.txt)
```

Warning: Never amend commits that have been pushed/shared with others.

---

7. Convenience Features and Shortcuts

- **Committing All Changes:**
  - `git commit -a -m "message"` stages and commits all tracked, modified files with a single command.
  - Not recommended for beginners; be explicit about what you stage.
- **Viewing Tracked Files:**
  - `git ls-files` lists all files tracked by Git in the current commit.
- **Searching Content:**
  - `git grep <pattern>` searches for a pattern in all tracked files.
  - Equivalent to `git ls-files | xargs grep <pattern>` but more efficient.

---

8. `git status` and Untracked Files

- `git status` shows:
  - Modified files
  - Staged files
  - Untracked files (not in Git or `.gitignore`)
- **Untracked Files:**
  - Git uses heuristics and `.gitignore` to decide which files to show as untracked.

---

9. Ignoring Files with `.gitignore`

- **Purpose:**
  - `.gitignore` lists patterns for files Git should ignore (e.g., build artifacts, temporary files).
- **Pattern Examples:**

Pattern	Meaning
`*.o`	Ignore all object files in any directory
`/main.mk`	Ignore only at repository root
`build/`	Ignore the entire `build` directory
`/gnu-lib-test/`	Ignore the `gnu-lib-test` directory at root

- **Multiple `.gitignore` Files:**
  - You can have `.gitignore` in any directory; Git checks the closest one first.
- **Versioning:**
  - `.gitignore` is usually versioned and shared with collaborators.

Example `.gitignore` File

```
# Ignore all .log files
*.log

# Ignore build/ directory everywhere
```

```
build/

# Ignore only at root
/.env

# Ignore node_modules in any subdirectory
node_modules/
```

Pattern	Matches Example	Does Not Match Example
`*.log`	`debug.log`, `foo/bar.log`	`foo/bar.txt`
`build/`	`build/`, `foo/build/`	`builder/`
`/.env`	`.env` at root	`foo/.env`

---

10. Git Repository Configuration

- **`.git/config`:**
  - Stores repository-specific settings (core options, remotes, branches, submodules).
  - Not under version control.
- **Viewing and Editing:**
  - `git config -l` lists all configuration settings.
  - `git config <key> <value>` sets a configuration value.
  - You can edit `.git/config` directly, but back it up first.

---

11. Commit Reference Syntax and Arithmetic

- **Shorthand Notation:**

Syntax	Meaning
`HEAD`	Latest commit on current branch
`HEAD^`	First parent (one commit back)
`HEAD^^`	Two commits back
`HEAD~3`	Three commits back
`commit^2`	Second parent (for merge commits)
`HEAD^!`	Difference between current commit and its parent

- **Tags:**
  - Tags can be used as human-readable names for commits.
  - Use `--` to separate commit references from file names to avoid ambiguity.

---

12. `git blame`

- **Purpose:**
  - Shows the last commit that modified each line of a file, along with author and date.
- **Usage:**
  - `git blame <filename>`
- **Cautions:**
  - Not always reliable for finding the true origin of a bug (may only show formatting changes).
  - Should not be used to assign blame in a punitive sense; can create a toxic environment.
  - Use to trace history, not to punish.
- **Tracing Bugs:**
  1. Use `git blame` to find the commit that last changed a line.
  2. Use `git diff <commit>^!` to see what that commit changed.

---

13. Summary Table: Key Git Commands

Command	Description
`git add <file>`	Stage file for commit
`git commit`	Commit staged changes
`git commit --amend`	Amend the last commit
`git status`	Show working directory status
`git diff`	Show unstaged changes
`git diff --cached`	Show staged changes
`git log`	Show commit history
`git ls-files`	List tracked files
`git grep <pattern>`	Search tracked files for a pattern
`git blame <file>`	Show last commit for each line in a file
`git config -l`	List configuration settings

---

14. ASCII Diagram: Git Commit Lifecycle

```
[Working Directory]
      |
      v   git add
   [Index / Staging Area]
      |
      v   git commit
   [Object Database / History]
```

---

15. Best Practices and Warnings

- Write meaningful commit messages explaining *why* changes were made.
- Don't obsess over perfect history; it's okay to make mistakes and fix them in later commits.
- Use `.gitignore` to avoid tracking unnecessary files.
- Be careful when editing `.git/config` directly.
- Use `git blame` for understanding history, not for assigning fault.
- Prefer explicit staging (`git add`) over `git commit -a` for clarity.

Checklist: Git Best Practices

Do's	Don'ts
Write clear, descriptive commit messages	Use vague messages like "fix"
Stage files explicitly with `git add`	Rely on `git commit -a` for everything
Use `.gitignore` for unneeded files	Commit build artifacts
Rebase only local, unpublished branches	Rebase shared/public branches
Use `git status` and `git log` frequently	Edit `.git/config` without backup
Resolve merge conflicts carefully	Use `git blame` to assign fault

---

16. Branching

A **branch** is a lightweight, movable pointer to a commit. Branches help isolate development work and enable parallel feature development, bug fixes, and experimentation.

16.1 Common Use Cases

Type	Purpose
Maintenance	Stable bugfix-only version
Feature	Work on specific new features

Type	Purpose
Experimental/Fork	Alternative implementations or separate project directions

16.2 Managing Branches

```
git branch                # list branches
git branch newbranch      # create a new branch
git checkout newbranch    # switch to a branch
git checkout -b newbranch commit  # create and switch
git branch -d name        # delete a branch (warns if commit is unmerged)
git branch -D name        # force delete
git branch -m old new     # rename branch
```

Example: Branch Workflow

```
git branch feature-x         # Create branch
git checkout feature-x       # Switch to branch
# ... make changes ...
git commit -am "Work on feature X"
git checkout master
git merge feature-x          # Merge changes into master
git branch -d feature-x      # Delete branch
```

Table: Branch Commands

Command	Description
`git branch`	List branches
`git branch <name>`	Create new branch
`git checkout <name>`	Switch to branch
`git checkout -b <name>`	Create and switch to new branch
`git branch -d <name>`	Delete branch (safe)
`git branch -D <name>`	Delete branch (force)
`git branch -m <old> <new>`	Rename branch

16.3 Visual Representation

```
            A---B---C (master)
                     \
                      D---E---F (feature)
```

- `master` and `feature` point to different commits.
- Use `git checkout` to switch between them.
- Commits form a Directed Acyclic Graph (DAG), where arrows point backwards to parent commits.

---

17. Merging

Merging combines content from two different branches into a single commit.

```
git merge branchname
```

17.1 Merge Commit

- When merging master with another branch, Git auto-generates a new commit containing both histories.
- If there's no conflict:
  - Git auto-merges changes.
  - Creates a new commit with lineage from both parents.

17.2 Merge Conflicts

- Occur when the same lines are changed in both branches.
- Git inserts markers like:

```
<<<<<<< HEAD
old version
=======
new version from other branch
>>>>>>> branchname
```

- You must manually resolve conflicts and commit the result.

Example: Merge Conflict and Resolution

Suppose both `master` and `feature` modify the same line in `file.txt`.

```
git checkout master
echo "Line from master" > file.txt
git commit -am "Master change"

git checkout feature
echo "Line from feature" > file.txt
git commit -am "Feature change"

git checkout master
git merge feature
```

Result:

```
<<<<<<< HEAD
Line from master
=======
Line from feature
>>>>>>> feature
```

Resolution:
Edit the file to resolve the conflict, then:

```
git add file.txt
git commit
```

17.3 Merge Algorithms

- Git identifies the "closest common ancestor" (e.g., commit A).
- Computes changes from A to B (`diff B A`) and A to C (`diff C A`).
- If non-overlapping, Git applies changes cleanly.
- Overlapping changes cause a conflict requiring manual intervention.

---

18. Rebasing

Rebasing re-applies commits from one branch to another base commit.

```
git rebase branchname
```

- Instead of creating a merge commit, rebasing rewrites history to make it appear as if your changes were applied on top of the target branch.

18.1 Visual Comparison

Merge:

```
    A---B---C---M (merged)
         \     /
          D---E
```

Rebase:

```
    A---B---C---D'---E' (rebased)
         \
          D---E
```

- Rebase creates new commits (`D'`, `E'`) with same changes, but different parent.
- Resulting history is "clean", linear.

Table: Merge vs. Rebase

Feature	Merge	Rebase
History	Preserves all branches and merges	Linearizes history
Commit Hashes	Unchanged for existing commits	New hashes for rebased commits
Use Case	Collaborative, public branches	Local, private branches
Conflict Resolution	May need to resolve once	May need to resolve at each commit

18.2 When to Use

- Clean, linear history after local development before pushing.
- Avoid publish-rebase: rewriting shared history causes complications.

---

19. Best Practices

- **Commit Messages:**
  - Describe reasoning, not just the change.
  - Avoid generic messages like "xyz."
- **Avoid Perfectionism:**
  - Don't rely heavily on `--amend` early on.
  - Mistakes are natural and traceable; history should reflect development flow.
- **Merging vs. Rebasing:**
  - Merging preserves actual development paths.
  - Rebasing simplifies history—useful for polishing before sharing.

---

20. Additional ASCII Diagram: Git Workflow

```
[Working Directory]
      |
      v   git add
   [Index / Staging Area]
      |
      v   git commit
   [Object Database / History]
      |
      v   git branch
   [Branch Pointer]
      |
      v   git merge / git rebase
   [Integrated History]
```

Lecture 12: Git Internals and Advanced Features

Git as an Object-Oriented Database

- At its core, Git manages your project history as an object-oriented database.
- Every file, directory, and commit is stored as an object, each with a unique SHA-1 checksum as its identifier.
- This object database persists on disk, ensuring your history is safe even if you lose power or close your editor.
- The structure is similar to object-oriented programming, but the objects are stored in files on disk, not in RAM.
- This design enables Git to efficiently store, retrieve, and share complete project histories, and is a key reason for its power and flexibility.

ASCII Diagram: .git Directory Structure

```
.git/
├── branches/         # (Obsolete) Old branch storage
├── config            # Project-specific Git configuration
├── description       # (Obsolete) Short repo description
├── head              # Points to current branch/commit
├── hooks/            # Scripts for Git events (pre-commit, etc.)
├── info/
│   └── exclude       # Ignore rules not under version control
├── index             # Staging area (binary file)
├── logs/             # History of branch/HEAD moves
├── objects/          # All Git objects (blobs, trees, commits)
├── refs/             # Human-readable names (branches, tags)
```

Table: Git Object Types

Type	Description	Example Content
blob	File content	Source code, text files
tree	Directory structure (points to blobs/trees)	List of files, subdirectories
commit	Snapshot metadata + root tree reference	Author, message, parent, tree

Real-World Scenario: Power of Git's Object Database

Suppose you are collaborating on a software project. You make a series of changes, commit them, and later realize a bug was introduced. With Git's object database, you can use `git bisect` to efficiently find the exact commit that introduced the bug, or use `git checkout` to revert to any previous state. The object database ensures every version is stored efficiently and can be shared or restored at any time, enabling robust collaboration and recovery workflows.

---

1. Rebasing in Git

1.1 What is Rebasing?

- Rebasing is the process of reapplying a series of commits on top of another base tip.
- Used to clean up, reorder, or edit commits before publishing to a shared repository.
- Especially useful for unpublished local commits.

1.2 Interactive Rebase (`git rebase -i`)

- Allows you to edit commit history interactively.
- Command:

  ```
  git rebase -i <commit-id>
  ```

  - `<commit-id>`: The commit before the first one you want to rebase.
- In the editor, you can:
  - Change the order of commits
  - Delete commits
  - Edit commits (with `edit`)
  - Drop commits
- Example edit script:

```
pick abcd123 Add logging
pick efgh456 Fix bug
pick xyzw789 Update Makefile
```

Can be reordered or transformed into:

```
pick efgh456 Fix bug
edit abcd123 Add logging
drop xyzw789 Update Makefile
```

- To continue after resolving conflicts or editing:

```
git rebase --continue
```

- If conflicts arise:
  - Fix the conflict
  - `git add` to stage resolved files
  - Continue with `git rebase --continue`
  - Use `git status` to see the current state

1.3 Merge Conflicts During Rebase

- Changing the order of commits may lead to merge conflicts.
- Resolve them manually, stage, and continue the rebase process.
- The original history is preserved until you finish and force-update the branch.

---

2. Remote Repositories and Distributed Git

2.1 Git as a Distributed VCS

- Git is fundamentally distributed: there is no required central authority.
- Multiple repositories can exist (e.g., in different locations), each with equal status.
- In practice, some repositories (like Linus Torvalds' for Linux) are treated as authoritative, but Git itself does not enforce this.
- Each repository can know about "remotes" (other repositories it can fetch from or push to).

2.2 Viewing and Managing Remotes

- `git remote`: List all remotes
- `git remote -v`: List remotes with fetch and push URLs
- By convention, the default remote after cloning is called `origin`.
- You can add or change remotes:

```
git remote add <name> <url>
git remote set-url <name> <new-url>
```

2.3 Fetch vs Pull

Command	Behavior
`git fetch`	Fetch changes from remote without merging
`git pull`	Fetch + Merge (potentially causing merge conflicts)

- `git fetch` updates your repository's view of the remote branches (e.g., `origin/main`), but does not change your local branches.
- `git pull` is equivalent to `git fetch` followed by `git merge` (or `git rebase`, depending on config).
- Some developers prefer to fetch and rebase manually to avoid unnecessary merge commits.
- Tracking branches (created with `git branch --track <branch> origin/<branch>`) help keep local and remote branches in sync.

Example: Keeping Main in Sync

- Some developers always keep their `main` branch identical to `origin/main` and do development in feature branches, rebasing as needed.

---

3. Stashing

3.1 What is Stashing?

- Stashing temporarily saves work-in-progress changes when you're not ready to commit them.
- Useful if you need to switch context (e.g., fix a bug on another branch) without committing incomplete work.
- Commands:
  - `git stash push`: Save current uncommitted changes
  - `git stash apply`: Reapply the last saved stashed changes
  - `git stash list`: View stashed entries
  - `git stash show`: View contents of the top stash
  - `git stash drop`: Remove a stash entry
  - `git stash pop`: Apply and remove the top stash
- Alternative manual stashing:

```
git diff > patch.diff
git checkout branch
# Later
patch -p1 < patch.diff
```

- Stashing is preferred in workflows where every commit must build and pass tests.

---

4. Bisecting: Finding Bug Introductions

4.1 Git Bisect

- Used to find the commit that introduced a bug between two known good/bad states.
- Uses binary search ($\log_2 N$ complexity) to minimize the number of test runs.
- Commands:

```
git bisect start
git bisect bad           # current or specified bad commit
git bisect good <commit> # known good commit
git bisect run <script>  # automates bisection
```

- Example:

```
git bisect start
git bisect bad
git bisect good v23
git bisect run make check
```

- If a commit can't be tested (e.g., doesn't build):

```
git bisect skip <commit>
```

- Some projects enforce "never break the build" to make bisecting reliable.

---

5. Git Internals: The `.git/` Directory

5.1 Directory Structure

Path	Description
`branches/`	(Obsolete) Old branch storage; kept for backward compatibility
`config`	Project-specific Git configuration
`description`	(Obsolete) Short description of the repository
`head`	Points to the current branch or commit
`hooks/`	Scripts triggered by Git events (e.g., `pre-commit`)

Path	Description
`info/exclude`	Ignore rules not under version control
`index`	Staging area between working directory and the repository
`logs/`	History of branch/HEAD move events
`objects/`	Storage for all Git objects (blobs, trees, commits)
`refs/`	Human-readable names (branches, tags) mapped to SHA1

- Many files and directories are for backward compatibility or are rarely used (e.g., `branches/`, `description`).
- `hooks/` contains sample scripts; to activate, rename and customize as needed.
- `index` is a binary file used for efficient staging.
- `info/exclude` is like `.gitignore` but not under version control.
- `logs/` records the history of branch pointers (e.g., for recovering from mistakes).

5.2 Object Database

- Git stores all content as objects in `.git/objects/`.
- Objects are named by their SHA1 checksum, split into a two-character directory and a 38-character filename for efficiency.
- Early versions used flat storage, but this was slow for large repos.
- Objects are compressed binary files.
- There are also packed objects (for efficiency in large repos), but these are more advanced.

Object Types

Type	Description
blob	File content
tree	Directory structure (refers to blobs/trees)
commit	Snapshot metadata + reference to root tree

Viewing and Creating Objects

- Get type:

```
git cat-file -t <sha>
```

- Get content:

```
git cat-file -p <sha>
```

- Create object:

```
echo "hello" | git hash-object -w --stdin
```

- Example:

```
echo "Arma virumque cano" > aeneid.txt
git hash-object -w aeneid.txt
```

Trees

- Trees represent directories; entries contain:
  - Name (filename or subdirectory)
  - Type (`blob` or `tree`)
  - Mode (file permissions, e.g., 100644, 040000)
  - SHA1 (points to blob or subtree)
- Example:

```
git update-index --add --cacheinfo 100644 <blob-sha> filename.txt
git write-tree
```

Commits

- Commits link a tree and metadata about the snapshot.
- Contents:

```
tree <tree-sha>
parent <parent-commit-sha>
author <name> <email> <timestamp>
committer <name> <email> <timestamp>

<commit message>
```

- Create manually:

```
git commit-tree <tree-sha> -m "message"
```

---

6. Porcelain vs Plumbing

Category	Description	Examples
Porcelain	User-facing commands for day-to-day use	`git add`, `git commit`, `git push`
Plumbing	Low-level commands for scripting or development	`git hash-object`, `git cat-file`, `git write-tree`

- Porcelain commands are for everyday use; plumbing commands are for scripting, debugging, or extending Git.
- Example plumbing commands:
  - `git hash-object`: Create a new object from a file or input
  - `git cat-file`: Show type or contents of an object
  - `git write-tree`: Create a tree object from the index
  - `git commit-tree`: Create a commit object from a tree

---

7. Summary

This lecture provided an in-depth exploration of advanced Git features and internals. Key topics included interactive rebasing, distributed nature of Git, remotes and synchronization, stashing, bisecting, and a tour through the internal structure and types within the `.git/` directory. It introduced Git's core concept of representing file changes as a graph of SHA1-addressed objects, and distinguished between user-facing "porcelain" commands and low-level "plumbing" commands that interact directly with Git's object database.

Appendix: Useful Visualizations

Distributed Git Repositories (Example Topology)

```
Repo1 <----> Repo2 <----> Repo3
   ^                      |
   |                      v
  Repo0 <-------------- Repo4
```

- Each repo can fetch/push from/to others; no single "master" required.

.git Directory Structure (Simplified)

```
.git/
├── branches/      # (obsolete)
├── config         # repo config
├── description    # (obsolete)
├── head           # current branch pointer
├── hooks/         # event scripts
├── info/exclude   # ignore rules
```

```
├── index          # staging area
├── logs/          # branch/HEAD history
├── objects/       # all objects (blobs, trees, commits)
├── refs/          # branch/tag names
```

Git Object Relationships

```
[commit]
   |
   v
 [tree]---->[tree]---->[blob]
   |          |          |
   v          v          v
 [blob]    [blob]     [blob]
```

- Commits point to trees (directories), which point to blobs (files) and other trees (subdirectories).

Lecture 13: Git Objects, Compression Algorithms, and Low-Level C Concepts

1. Git Internal Object Representation

1.1 Git Directory Structure

- Git repositories contain a `.git/` subdirectory.
- `.git/objects/` stores Git objects as files, named by SHA-1 hashes.
- The `.git/objects` directory contains subdirectories named by the first two characters of the SHA-1 hash, each containing files named by the remaining 38 characters.

ASCII Diagram: Git Object Storage

```
.git/
  objects/
    6d/
      cb09e...   # File containing the object data
    7a/
      1f2e3...
    ...
```

1.2 Git Object Types and File Naming

- Git objects include:
  - `blob`: file data
  - `tree`: directory structure
  - `commit`: snapshot and metadata
  - `tag`: annotated reference to a commit
- Each object is named by a 40-character SHA-1 hexadecimal string.
  - First 2 characters: directory name.
  - Remaining 38 characters: file name inside that directory.
- Example:

  ```
  SHA-1: 6dcb09e... (40 hex digits)
  Directory: 6d/
  File: cb09e...
  ```

1.3 Blob Format

- Git objects include: `blob`, `tree`, and `commit`. Focus here: `blob`.
- Format: `blob <size><null byte><binary contents>`
  - ASCII "blob", then a space.
  - Decimal count of number of bytes (e.g., `196`).
  - Null byte (`\0`), ASCII value 0.
  - Then `<size>` bytes of content.
- **Why include size in header?**
  - Blob format is self-describing, not dependent on external bookkeeping.
  - Useful for packed structures.
- **Why use decimal for length?**
  - Makes blobs more readable for debugging.
  - Not worth the complexity to use binary for a few bytes of savings.

Example: Creating and Inspecting a Blob

```
echo "hello world" > file.txt
git hash-object file.txt  # Creates a blob and prints its SHA-1
SHA=($(git hash-object -w file.txt))
# Inspect the blob:
git cat-file -p $SHA
```

1.4 Compression in Git

- Git compresses objects using Zlib (same as Gzip).
- When examining files directly, decompression is required.

---

2. Limits and Theory of Data Compression

2.1 Fundamental Limits

- No "universal" compression algorithm can always reduce file size for all inputs.
- **Information-theoretic argument (Pigeonhole Principle):**
  - If input has 2^N possible values, you need N bits to represent all possibilities.
  - If you try to compress every possible input, some must get larger (since there aren't enough shorter outputs for all inputs).
- **Compression only works when input has redundancy or patterns.**
- **Compression behavior:**
  - Highly redundant text: major size reduction.
  - Random bytes: may get larger due to headers.

Example: Why Random Data Can't Be Compressed

Suppose you have all possible 8-bit values (0-255). Any mapping to fewer than 8 bits would require collisions (loss of information), so true random data cannot be compressed.

---

3. Gzip/Zlib Compression Algorithms

- Gzip uses two main ideas:
  1. **Huffman Coding** (1950s)
  2. **Dictionary Compression** (1970s)
- These are often combined (as in DEFLATE, used by gzip/zlib).

---

4. Huffman Coding

4.1 Overview

- Assigns variable-length bit strings to symbols based on frequency.
- Frequent symbols get shorter codes.
- Optimal for symbol-by-symbol compression with known probabilities.

4.2 Example: Symbol Probabilities

Character	Frequency
' ' (space)	0.2
e	0.06
t	less than e
z/q	very rare

4.3 Encoding Table Example

Symbol	Encoding
space	00
e	01
t	10
q	11100101

- **Prefix-free property:** No code is a prefix of another.

4.4 Huffman Tree Construction

- Start with all symbols as leaves, ordered by frequency.
- Merge two least frequent symbols at each step.
- Repeat until one node (the root) remains.
- Path from root to leaf = codeword for symbol.
- For 256 leaves (bytes): 255 internal nodes.

Step-by-Step Example: Huffman Tree

Suppose we have symbols A (5), B (9), C (12), D (13), E (16), F (45):

```
Step 1: Combine A(5) + B(9) = 14
Step 2: Combine C(12) + D(13) = 25
Step 3: Combine 14 + E(16) = 30
Step 4: Combine 25 + F(45) = 70
Step 5: Combine 30 + 70 = 100
```

ASCII Diagram:

```
        [100]
       /    \
    [30]    [70]
   /   \    /  \
[14] [16] [25] [45]
/  \      /  \
A   B    C    D
```

4.5 Optimality & Limitations

- **Pros:** Minimizes total bits if symbol probabilities are known and fixed.
- **Cons:**
  - Sender and receiver must share the encoding table.
  - Table size adds overhead.
  - Inflexible if input corpus changes (e.g., language switch mid-text).

---

5. Adaptive Huffman Coding

5.1 Overview

- Sender and receiver start with a trivial, balanced Huffman tree.
- Tree is updated dynamically as input is processed.
- No need to transmit the full table upfront.
- Rarely used in practice due to speed and memory tradeoffs; more common in academic settings.

5.2 Process

1. Start with uniform tree (equal frequency for all symbols).
2. Transmit first few symbols in full (e.g., 8-bit bytes).
3. After each symbol:
   - Update tree.
   - Both sender and receiver maintain identical updated trees.

Example: Tree Evolution

```
Initial tree: all symbols equal frequency
After 'A': 'A' node frequency increases
After 'B': 'B' node frequency increases
Tree structure changes as frequencies change
```

5.3 Advantages

- Only one pass over input required.
- Tree adapts to input statistics over time.
- Handles changing symbol distributions (e.g., English to Portuguese).

5.4 Trade-Offs

- Inefficient at the start (no compression until tree adapts).
- Slightly suboptimal compared to full-table Huffman.
- Sender/receiver must synchronize tree updates exactly.
- Requires extra RAM for tree (but small for byte-oriented data).

---

6. Dictionary Compression

6.1 Basic Idea

- Replace frequent sequences of bytes (e.g., words) with short codes.
- Instead of encoding each byte, encode entire sequences.

6.2 Dictionary Setup

- Build a dictionary mapping sequences to indexes.

```
Index | Value
------|------------------
1     | "the "
2     | "and"
3     | "understandable"
```

- Data stream encodes sequences as index references (e.g., "understandable" → index 3, 16-bit code).

6.3 Issues & Considerations

1. **Dictionary size:**
   - Full English dictionary > 100,000 entries.
   - If too large, can't fit in 16 bits.
2. **Non-word characters:**
   - Dictionary includes all common byte sequences, not just words.
3. **Language switching:**
   - Static dictionary may not compress well if input language changes.

---

7. Adaptive Dictionary Compression (Sliding Window)

7.1 Problem with Static Dictionaries

- Non-adaptive method requires two passes: build dictionary, then encode.
- Not ideal for large files or streaming data.

7.2 Adaptive Approach

- Uses a **sliding window** dictionary (e.g., 64KiB of recent data).
- Sender and receiver both maintain a history of previous N bytes.
- Dictionary dynamically reflects recent input.
- Format for reference:

```
{Offset (16 bits), Length (8 bits)}
```

ASCII Diagram: Sliding Window Compression

```
Input Stream:  ... | W | I | N | D | O | W |  ...
                 <------ Sliding Window ------>
Current Position:   ^
```

The sliding window moves forward as new data is processed, always containing the most recent N bytes.

Step-by-Step Example: Adaptive vs. Static Dictionary

Suppose the input is: `the cat and the dog`

- **Static Dictionary:**
  - Pre-built dictionary: {1: "the", 2: "cat", 3: "dog"}
  - Encoded as: [1] [2] [and] [1] [3]
- **Adaptive (Sliding Window):**
  - Start with empty window.
  - As "the" is seen, it's added to the window.
  - When "the" appears again, it's referenced by offset/length in the window.

- "cat" and "dog" are also added as they appear.

Example: LZ77 (Sliding Window) Encoding

Suppose the input is: `abcabcabc`

- First "abc": output as raw bytes
- Second "abc": reference offset=3, length=3
- Third "abc": reference offset=3, length=3

Table: Static vs. Adaptive Dictionary Compression

Feature	Static Dictionary	Adaptive (Sliding Window)
Dictionary	Pre-built, fixed	Built dynamically as input read
Passes Needed	Two (build, then encode)	One (encode as you go)
Handles New Data	Poorly (if not in dict)	Well (adapts to new patterns)
Memory Usage	Fixed (dict size)	Sliding window size
Use Case	Known, repetitive data	Streaming, unknown data

7.3 Compression Technique

1. Start with empty dictionary.
2. At each input point:
    - Find longest match in dictionary.
    - Emit (offset, length).
3. If no match:
    - Emit raw byte.

7.4 Search Optimization

- Naive search: $O(n^2)$.
- Efficient implementations use tries or suffix trees ($O(n \log n)$).

7.5 Limitations

- If a matching sequence is not in the window, must send as raw byte.
- Random data compresses poorly.
- Sensitive to data loss—losing sync causes downstream errors.
- More RAM can improve compression, but at a cost.

---

8. C vs. C++: What C Leaves Out

8.1 Simplified View: C = C++ - High-Level Features

Feature	C++	C
Classes & Objects	Yes	No
Polymorphism	Yes	No
Encapsulation	Yes	Limited
Inheritance	Yes	No
Abstract Data Types	Yes	Less supported
Static Members	Yes	Not in structs
Namespaces	Yes	No
Function Overloading	Yes	No
Exceptions	Yes	No (discouraged)
New/Delete	Yes	No (use malloc/free)
`cin`/`cout`	Yes	No (use `stdio.h`)

- In C, an "object" is just a piece of memory with a type.
- You still have `struct`, but not `class`.

- No built-in memory allocation operators (`new`, `delete`); use `malloc`/`free`.
- No namespaces; must manage names manually.
- No function overloading; use unique names.
- Exception handling exists but is discouraged.

8.2 Low-Level Control in C

- C provides direct access to memory via pointers and pointer arithmetic.
- Manual memory management: allocate with `malloc`, free with `free`.
- No automatic garbage collection.
- Allows direct manipulation of hardware and system resources.

Table: Memory Management in C vs. C++

Feature	C++ (Modern)	C
Manual Allocation	`new`/`delete`	`malloc`/`free`
Automatic (RAII)	Yes (constructors)	No
Smart Pointers	Yes (`std::unique_ptr`, etc.)	No
Garbage Collection	No (but can use libs)	No

---

9. C Compilation Process

9.1 Steps

Step	Input	Output	GCC Flag
Preprocessing	.c	.i	`-E`
Compilation	.i	.s	`-S`
Assembly	.s	.o	`-c`
Linking	.o	(executable)	-o (with `gcc`)

ASCII Flowchart: C Compilation Pipeline

```
[Source.c] --(Preprocessing: -E)--> [Source.i] --(Compilation: -S)--> [Source.s] --(Assembly: -c)-->
[Source.o] --(Linking: -o)--> [Executable]
```

9.2 Example

```c
#define INTMAX 2147483647
int main() { return INTMAX; }
```

- Preprocessing expands macros:

```c
int main() { return 2147483647; }
```

- Compilation to assembly:

```
main:
    movl $2147483647, %eax
    ret
```

- Linking resolves library calls (e.g., `abs`).

---

10. Process Management & Developer Tools

10.1 Process Management Commands

Command	Purpose
`ps`	Show running processes
`ps -ef`	All processes (full view)
`ps -ejH`	Show hierarchy of processes
`kill <PID>`	Send default signal (TERM)
`kill -9 <PID>`	Force kill using SIGKILL signal
`kill -STOP`	Pause process
`kill -CONT`	Resume process

Table: Common Signals

Signal	Number	Description
SIGTERM	15	Graceful termination
SIGKILL	9	Forceful kill (cannot be caught)
SIGSTOP	19	Pause process
SIGCONT	18	Resume process

10.2 Debugging Tools

- `time <command>`: Shows real/user/sys time for a command.
- `strace <command>`: Traces system calls (kernel interactions).
- `ltrace <command>`: Traces library calls (e.g., `printf`, `malloc`).
- `valgrind <command>`: Memory error detection (invalid access, leaks, uninitialized use). Slower but thorough.

Example: Using Valgrind

Suppose you have a buggy C program `buggy.c`:

```c
int main() {
    int *p = malloc(sizeof(int));
    return *p; // uninitialized read
}
```

Compile and run with valgrind:

```
gcc buggy.c -o buggy
valgrind ./buggy
```

Valgrind output will report use of uninitialized value.

Summary

This lecture explored Git's internal object model, especially blob storage and format, and explained how Git compresses data using Gzip/Zlib. It covered the theory and practical limits of compression, the two main strategies in Gzip (Huffman coding and dictionary compression), and both static and adaptive versions of these algorithms. The lecture then transitioned to low-level C programming, contrasting it with C++ by highlighting omitted features for greater control and performance. It included a walk-through of the C compilation pipeline and the tools developers use to observe and manage processes, debug, and track system and memory behavior. These concepts are essential for understanding system-level programming and preparing for assignments that work closely with system internals.

Lecture 14: Low-Level Programming

1. Introduction and Context

- Focus: Low-level software development, primarily in C and C++.
- These languages are considered "low-level" compared to Python, JavaScript, etc.
- Most modern software is not written in C/C++, but these languages remain crucial for performance-critical and system-level work.
- Tools for low-level development (debugging, performance, security) are the most mature due to decades of evolution.
- Many techniques discussed here are not yet available in higher-level languages, but may propagate over time.
- **Low-level languages** provide direct access to memory and hardware, with minimal abstraction. **High-level languages** abstract away hardware details for ease of use and safety.

Feature	Low-Level (C/C++)	High-Level (Python, JS)
Memory Management	Manual (malloc/free)	Automatic (GC, no pointers)
Hardware Access	Direct (pointers, I/O)	Indirect/abstracted
Performance	High, predictable	Lower, less predictable
Safety	Low (UB, buffer overflows)	High (bounds checks, types)
Use Cases	OS, drivers, embedded	Web, scripting, data science

- **Assembly** is even lower-level than C/C++, providing a 1:1 mapping to machine instructions. C/C++ is often called "portable assembly."

2. Attitude Toward Debugging

- **Avoid using a debugger as a first resort.**
  - Using a debugger often means you've already failed to prevent a bug.
  - Debugging is human-intensive and inefficient compared to preventive techniques.
- **Alternative:** Use compiler features and static/dynamic analysis to catch bugs early.
- This attitude is especially helpful at the low level, but is good practice at any level.

Approach	Description	Example Tool/Method
Preventive	Catch bugs before they happen	static_assert, -Wall
Reactive	Find/fix bugs after they occur	gdb, printf, valgrind

Example:

- *Static analysis*: Compiler warns about uninitialized variable (`-Wall`).
- *Debugger*: You run the program, it crashes, and you use gdb to find the bug.

3. Compilers as Debugging and Security Tools

3.1 GCC and Clang

- **GCC** and **Clang** are the two major free compilers for C/C++.
- They are largely compatible, but some options differ in spelling or behavior.
- Most techniques discussed apply to both, but examples use GCC syntax.

3.2 Security-Related Compiler Options

3.2.1 `_FORTIFY_SOURCE`

- **Usage:** `-D_FORTIFY_SOURCE=2 -O2`
- **Purpose:** Replaces standard functions (e.g., `memcpy`) with safer, bounds-checked versions.
- **How it works:**
  - Compiler injects extra runtime checks for buffer overflows and similar bugs.
  - Example: `memcpy(dest, src, size)` will check that `dest` and `src` are valid and within bounds if possible.
- **Limitations:**
  - If the compiler cannot determine object sizes, it falls back to the standard (unchecked) version.
  - Not a guarantee of safety—just a partial defense.

Example:

```
char buf[8];
memcpy(buf, src, 16); // _FORTIFY_SOURCE detects overflow at runtime if possible
```

3.2.2 `-fstack-protector`

- **Purpose:** Defends against stack buffer overflows by inserting a "canary" value in the stack frame.
- **How it works:**
  - Canary is a random value placed before the return address.
  - On function return, the canary is checked. If it has changed, the program aborts.
  - Analogy: Like a canary in a coal mine—detects danger early.

```
|----------------------|  <-- High Address
|   Local Variables    |
|----------------------|
|      Canary Value    |  <--- Inserted by compiler
|----------------------|
|   Return Address     |
|----------------------|  <-- Low Address
```

The canary sits between local variables and the return address. If a buffer overflow overwrites the canary, the program aborts.

Step-by-Step: Buffer Overflow Detection with Canary

1. Function is called; stack frame is set up with local variables, canary, and return address.
2. A buffer overflow (e.g., writing past the end of an array) overwrites the canary value.
3. Before returning, the function checks the canary.
4. If the canary is changed, the program aborts immediately, preventing exploitation.

Real-World Analogy:
Imagine a sealed envelope with a tamper-evident sticker (the canary). If someone tries to open the envelope (overflow the buffer), the sticker is broken. When you receive the envelope, you check the sticker; if it's broken, you know not to trust the contents.

3.2.3 Control Flow Enforcement Technology (CET) / Shadow Stack

- **Usage:** `-fcf-protection`
- **Purpose:** Hardware-enforced control flow integrity using a shadow stack.
- **How it works:**
  - The shadow stack stores only return addresses, is inaccessible to user code, and is protected by the OS.
  - On function call: return address is pushed to both the normal and shadow stack.
  - On return: both are popped and compared. If they differ, the program traps/crashes.

```
Normal Stack (accessible):
|----------------------|  <-- High Address
|   Local Variables    |
|----------------------|
|   Return Address     |  <--- Target for attacks
|----------------------|  <-- Low Address

Shadow Stack (protected):
|----------------------|  <-- High Address
|   Return Address     |  <--- Only return addresses, protected
|----------------------|  <-- Low Address
```

The shadow stack is protected and only manipulated by special instructions. On return, both stacks are checked for agreement.

Step-by-Step: Shadow Stack Operation

1. Function call: Return address is pushed to both stacks.
2. Function executes.
3. Function return: Return address is popped from both stacks and compared.
4. If mismatch, program aborts (attack detected).

Requirements:

- Supported on recent Intel/AMD CPUs (CET-capable).
- Requires OS and compiler support; not enabled by default everywhere.

Real-World Analogy:
Think of the shadow stack as a secure logbook kept in a locked safe. Every time you leave a room (function call), you write down your destination (return

address) in both your personal notebook (normal stack) and the logbook (shadow stack). When you return, both records must match. If someone tampers with your notebook but can't access the logbook, the mismatch is detected and an alarm sounds (program aborts).

Security Feature	Compiler Flag/Option	What It Protects Against
Buffer Overflow	`_FORTIFY_SOURCE`, `-fstack-protector`	Overwriting memory, stack attacks
Return Address Tampering	`-fcf-protection`	Control flow hijacking
Use of Dangerous APIs	Warnings, static analysis	Unsafe functions (gets, strcpy)

4. Performance Improvement via Compiler Features

4.1 Optimization Levels

Flag	Description	Speed	Size	Debuggability
-O0	No optimization (default)	Slow	Large	Best
-O1	Basic optimizations	Med	Med	Good
-O2	Moderate, safe optimizations (most common)	Fast	Med	OK
-O3	Aggressive optimizations; may be buggy	Fastest	Large	Poor
-Os	Optimize for size instead of speed	Fast	Small	OK
-Og	Optimize for debugging (intended, but often still confusing)	Med	Med	Good

- **Caveats:**
  - Higher levels (especially `-O3`, `-Ofast`) may introduce compiler bugs or make debugging harder.
  - Optimizations may reorder instructions, making debugging confusing.
  - `-Og` is intended for debuggability, but may still cause confusion.

Example of Optimization Bug:

```
volatile int x = 0;
if (x == 0) {
    // ...
}
// With -O3, compiler may optimize away code if it thinks x can't change.
```

4.2 Build Scripts and Makefiles

- Build systems (e.g., Makefiles) control how programs are compiled and linked.
- Makefiles specify dependencies and recipes for building each part of a program.
- C/C++ build systems are more mature and sophisticated than those for higher-level languages.

Simple Example Makefile:

```
all: main

main: main.o util.o
    gcc -o main main.o util.o

main.o: main.c util.h
    gcc -c main.c

util.o: util.c util.h
    gcc -c util.c

clean:
    rm -f *.o main
```

This Makefile builds an executable from two source files, tracking dependencies and providing a clean target.

Build Process with Makefile

```
[Source Files]
     |
     v
```

```
 [Makefile]
     |
     v
[Compiler (gcc/clang)]
     |
     v
[Object Files (.o)]
     |
     v
   [Linker]
     |
     v
[Executable]
```

The Makefile orchestrates the build process, specifying how to compile and link source files into an executable.

4.3 Link Time Optimization (LTO)

- **Usage:** `-flto`
- **How it works:**
  - Object files include extra information (intermediate representation of the source code).
  - At link time, the compiler can optimize across all modules, enabling inlining and other whole-program optimizations.
- **Advantages:**
  - Can produce smaller, faster code.
- **Disadvantages:**
  - Slower link times.
  - Higher risk of compiler bugs, especially for large programs.

Diagram: Normal vs. LTO Build

```
Normal Build:
[Source1.c] -> [Obj1.o] --\
[Source2.c] -> [Obj2.o] ---+--> [Linker] -> [Executable]

LTO Build:
[Source1.c] -> [Obj1.o (IR)] --\
[Source2.c] -> [Obj2.o (IR)] ---+--> [Linker (with IR)] -> [Optimized Executable]
```

4.4 Profile-Guided Optimization (PGO)

- **Usage:** `-fprofile-generate` and `-fprofile-use`
- **How it works:**
  1. Compile with `-fprofile-generate`.
  2. Run the program to collect profiling data.
  3. Recompile with `-fprofile-use` to optimize based on real usage.
- **Caveats:**
  - Test runs may not match production behavior, so optimizations may not always be ideal.

Example:

```
gcc -fprofile-generate -o myprog myprog.c
./myprog   # Run to collect data
gcc -fprofile-use -o myprog-opt myprog.c
```

4.5 Optimization Hints

4.5.1 `__builtin_unreachable()`

- **Purpose:** Tells the compiler that a code path cannot be reached.
- **Example:**

  ```
  if (a < 0) __builtin_unreachable();
  x = a / 1024; // Compiler can optimize to x = a >> 10;
  ```

- **Difference from `abort()`:**

- `abort()` always crashes; `__builtin_unreachable()` allows the compiler to optimize more aggressively.
- In debug mode, you can use `abort()`; in release mode, switch to `__builtin_unreachable()` for performance.

- **Caution:** Only use when you are certain the code is unreachable; otherwise, you risk undefined behavior.

4.5.2 Function Attributes: `cold` and `hot`

- **Usage:** `__attribute__((cold))` or `__attribute__((hot))`
- **Purpose:**
  - `cold`: Function is rarely called (e.g., error handling).
  - `hot`: Function is frequently called (e.g., main loop).
- **Effect:**
  - Compiler may place cold code away from hot code to improve cache usage.
  - Can improve performance by keeping hot code in the instruction cache.
- **Example:**

```
__attribute__((cold)) void error_log();
__attribute__((hot)) int main_loop();
```

4.5.3 Profiling and Coverage

- **Compiler can use profiling data to automatically determine hot/cold code.**
- **Caveat:** Profiling assumes test runs match production runs, which may not always be true.

5. Static and Dynamic Code Checking

5.1 Static Checking

- **Static checking:** Compiler checks code at compile time, before running the program.
- **Example:**

```
#include <assert.h>
static_assert(INT_MAX < UINT_MAX, "Int must be smaller than UInt");
```

- **Benefits:**
  - Catches bugs before the program runs.
  - No runtime overhead.
- **Limitations:**
  - Only works for properties that can be checked at compile time.

Analysis Type	When It Runs	Catches Bugs Before Run?	Overhead
Static	Compile time	Yes	None
Dynamic	Runtime	No (only if executed)	Yes

5.1.1 Compiler Warnings

- **Enable with:** `-Wall` (and other `-W` flags)
- **Common warnings:**

Flag	Description
`-Wall`	Enable commonly helpful warnings
`-Wparentheses`	Warn about ambiguous or confusing expressions
`-Waddress`	Warn about suspicious pointer comparisons
`-Wstrict-aliasing`	Warn about type-punning across incompatible pointers
`-Wmaybe-uninitialized`	Warn if a variable might be used without initialization
`-Wtype-limits`	Warn about always-true/false comparisons due to types

- **Examples:**
  - `if (a + b << c)` // May be ambiguous; parentheses may be needed.
  - `if (p == "abc")` // Comparing pointer to string literal's address.
  - `int v; if (n < 0) v = 3; if (n < -10) return v;` // v may be uninitialized.

5.1.2 Controversial Warnings

- Some warnings (e.g., `-Wstrict-aliasing`, `-Wtype-limits`) are controversial:
  - May produce false positives or be inappropriate for certain codebases (e.g., Linux kernel).
  - Developers may disable them for portability or practicality.

5.1.3 Interprocedural Analysis: `-fanalyzer`

- **Purpose:** Performs static analysis across function boundaries.
- **Pros:** Can find bugs that single-function analysis misses.
- **Cons:** Slow, may produce many false positives, not always practical for large codebases.

5.2 Function Attributes for Checking and Optimization

Attribute	Description	Example Usage
`noreturn`	Function does not return (e.g., `exit()`)	`void fatal() __attribute__((noreturn));`
`const`/`unsequenced`	No side-effects; return depends only on arguments	`int square(int) __attribute__((const));`
`pure`/`reproducible`	No side-effects; may read global state	`int get_val() __attribute__((pure));`
`hot`/`cold`	Indicate call frequency for optimization	`void err() __attribute__((cold));`

- **Benefits:**
  - Help compiler optimize and check code correctness.
  - Example: `noreturn` allows compiler to warn about unreachable code after `exit()`.

5.3 Dynamic Checking

- **Dynamic checking:** Compiler inserts runtime checks to catch errors as the program runs.
- **Trade-offs:**
  - Slows down execution.
  - Only catches bugs if the problematic code path is executed during testing.

Common Dynamic Tools:

Tool	What It Detects
AddressSanitizer	Buffer overflows, use-after-free
Valgrind	Memory leaks, invalid accesses
UndefinedBehaviorSanitizer	Undefined behavior

5.4 Safe Integer Arithmetic (C23)

- **Header:** `<stdckdint.h>`
- **Functions:** `ckd_add`, `ckd_sub`, etc.
- **Usage:**

```c
#include <stdckdint.h>
int r;
if (ckd_add(&r, a, b)) {
  // Overflow occurred
}
```

- **Purpose:** Reliably detect integer overflow, which is a common attack vector.
- **Available as of C23.**

Operation	Traditional C (Risk)	C23 Checked (Safe)
Addition	`int c = a + b;`	`ckd_add(&c, a, b);`
Subtraction	`int c = a - b;`	`ckd_sub(&c, a, b);`
Multiplication	`int c = a * b;`	`ckd_mul(&c, a, b);`

6. Summary Table: Compiler Tools for Reliability and Performance

Use Case	Recommended Tool/Option

Use Case	Recommended Tool/Option
Undefined Behavior	`-fsanitize`, `-Wall`, `static_assert`
Stack Protection	`-fstack-protector`, Shadow Stacks (CET)
Buffer Overflow	`_FORTIFY_SOURCE`, `-fsanitize=address`
Performance	`-O2`, `-O3`, PGO, `-flto`
Integer Overflow	`<stdckdint.h>`
Optimization hints	`__builtin_unreachable()`, `__attribute__((cold))`

7. Final Notes

- Low-level programming requires careful attention to security, performance, and correctness.
- Modern compilers provide a wealth of options and features to help developers write safer, faster, and more reliable code.
- Avoid relying solely on debuggers; leverage compiler checks, static/dynamic analysis, and modern language features.
- Stay up to date with new standards (e.g., C23) and hardware features (e.g., CET).

Summary Table: Key Compiler Security Features

Feature/Flag	Purpose	How It Protects
`_FORTIFY_SOURCE`	Bounds-checks standard functions	Detects buffer overflows at runtime
`-fstack-protector`	Inserts canary for stack protection	Detects stack buffer overflows
`-fcf-protection`	Enables shadow stack (CET)	Prevents return address tampering
`-flto`	Link Time Optimization	Enables cross-module optimization
`-fprofile-generate`/`-fprofile-use`	Profile-Guided Optimization	Optimizes based on real usage
`-Wall`	Enables common warnings	Catches likely bugs at compile time

Lecture 15: Debugging Without a Debugger & GDB Deep Dive

1. Debugging Without a Debugger

1.1 Static Checking

- **Static checking** is performed at compile-time using tools like `gcc` and `clang`.
- **Other static tools:**
  - **Linters:** e.g., `cpplint`, `clang-tidy` for style and bug patterns.
  - **Static analyzers:** e.g., Coverity, Clang Static Analyzer, Infer (Facebook), which can find deep bugs across function boundaries.
  - **Formal verification:** Mathematical proofs of correctness (rare in practice, used in safety-critical systems).
- **Limitations:**
  - Cannot catch all bugs (e.g., runtime-specific issues).
  - Theoretical limits (e.g., Halting Problem) prevent perfect static analysis.
  - May produce false positives/negatives.

Tool/Method	What It Checks	Example Tool
Compiler Warnings	Syntax, types, some logic	`-Wall`, `-Wextra`
Linter	Style, simple bug patterns	`cpplint`, `pylint`
Static Analyzer	Deep logic, interprocedural	Clang Static Analyzer, Coverity
Formal Verification	Mathematical correctness	Frama-C, SPARK

1.2 Dynamic Checking

- **Dynamic checking** occurs during program execution.
- Catches bugs missed by static checking, but only for the specific run/test case.
- **Manual dynamic checks:**
  - Add assertions to validate state at runtime.
  - Example: Array bounds checking

    ```
    if (!(0 <= i && i < n)) error();
    a[i];
    ```

  - Example: Integer overflow (incorrect check)

    ```
    if (j * k > INT_MAX) error(); // This does NOT work in C/C++
    ```

    - In C/C++, signed integer overflow is undefined behavior. The implementation can do anything, so the check above is unreliable.
  - **Correct overflow check (C23/C++26):**

    ```
    #include <stdckdint.h>
    if (__builtin_mul_overflow(j, k, &i)) error();
    ```

    - `__builtin_mul_overflow` returns true if overflow occurred.
    - C++26 will have similar utilities.
  - **Downsides:**
    - Tedious and error-prone to add checks everywhere.
    - Easy to make mistakes in the checks themselves.

1.2.1 GCC Sanitizer Flags

- Compiler flags that insert runtime checks for various classes of bugs:

Flag	Purpose	Captures
`-fsanitize=undefined`	Catch undefined behavior (e.g., overflow)	Integer overflow, divide by 0
`-fsanitize=address`	Catch memory/address issues	Buffer overflows, bad pointers
`-fsanitize=thread`	Detect race conditions	Multi-thread concurrency bugs

Flag	Purpose	Captures
`-fsanitize=leak`	Find memory leaks	Leaked malloc() allocations

- **Limitations:**
  - Some flags cannot be used together (e.g., `undefined` vs `address`).
  - Not all undefined behaviors are caught (e.g., some pointer aliasing bugs).
  - These flags slow down execution.
  - Only effective if the code path is executed during testing.

1.2.2 Valgrind

- Tool for dynamic analysis at the binary level (no recompilation needed).
- Detects memory errors, leaks, and some undefined behaviors.
- **Advantages:**
  - Can be used on production binaries.
  - No need for special compilation flags.
- **Disadvantages:**
  - Much slower than sanitizer flags (interprets each instruction).
  - Lacks source code context, so may miss semantic errors.
  - Cannot catch all concurrency bugs or logic errors.

Tool	Source Required	Speed Impact	Coverage	Usability in Production
Sanitizers	Yes	Moderate	Targeted checks	No
Valgrind	No	High	Broader, shallower	Yes
Fuzzers	Yes	High	Randomized input paths	No

1.2.3 Compiler Flag: `-fwrapv`

- Forces signed integer overflow to wrap around (modulo 2^n).
- Makes code more predictable, but disables some optimizations (e.g., loop unrolling).
- **Not the default** because it can slow down code and prevent optimizations.

Example: Loop Unrolling

```
for (int i = 0; i < n; i++) {
  f(i);
}
```

- If `n` is `INT_MAX`, `i++` might overflow.
- With `-fwrapv`, the compiler cannot assume overflow doesn't happen, so it can't unroll the loop.

Table: Static vs. Dynamic Checking

Aspect	Static Checking	Dynamic Checking
When Performed	Compile-time	Run-time
Tools	Compiler, linters, analyzers	Assertions, sanitizers, Valgrind
Bugs Detected	Syntax, type, some logic	Memory, concurrency, runtime
Overhead	None at runtime	Slows down execution
Limitations	Can't catch runtime-specific	Only checks tested code paths

More Dynamic Tools

- **Fuzzers:** e.g., AFL, libFuzzer. Generate random/semi-random inputs to find crashes and bugs.
- **Dynamic analyzers:** e.g., Dr. Memory, ThreadSanitizer (for race conditions).

Assertions: Static vs. Runtime

- **Runtime assertions:** Checked during execution. Can be compiled out with `-DNDEBUG`.
- **Static assertions:** Checked at compile time (e.g., `static_assert` in C11/C++11).
- **Pitfall:** Assertions with side effects can change program behavior if compiled out.

```
// BAD: Side effect in assertion
assert(x++ > 0); // x is incremented only if assertions are enabled!
```

Assertion Type	When Checked	Can Be Compiled Out?	Example
Runtime	Run-time	Yes (`-DNDEBUG`)	`assert(x > 0);`
Static	Compile-time	No	`static_assert(...);`

Practical Debugging Workflow

Suppose a program crashes with a segmentation fault. A robust workflow:

```
[Start]
  |
  v
[Reproduce Failure]
  |
  v
[Static Checking (compiler, linter)]
  |
  v
[Dynamic Checking (sanitizer, Valgrind, fuzzer)]
  |
  v
[Add Assertions/Logging]
  |
  v
[Trace Error (print/debugger)]
  |
  v
[Fix Bug]
  |
  v
[Retest All Cases]
  |
  v
[End]
```

- **Decision Tree Example:**
  - If bug is a crash: Use sanitizer/Valgrind first.
  - If bug is a logic error: Add assertions and logging.
  - If bug is intermittent: Try ThreadSanitizer or fuzzing.

Debugging Tool Comparison Table

Tool/Method	Detects	Overhead	Source Needed	Best For
Compiler Warnings	Syntax, types	None	Yes	Early bug detection
Linter	Style, simple bugs	None	Yes	Code quality
Static Analyzer	Deep logic	Low	Yes	Complex bugs
Sanitizer	Memory, UB, races	Med	Yes	Memory/concurrency bugs
Valgrind	Memory, leaks	High	No	Production binaries
Fuzzer	Crashes, edge cases	High	Yes	Unusual input bugs

2. Portability Checking

- Ensures code runs correctly on different platforms (OS, architecture, browser, etc.).
- **Examples:**
  - Cross-browser JavaScript testing.
  - Cross-platform compilation (32-bit vs 64-bit).
  - Testing with different OS/browser/plugin combinations.
- **Techniques:**
  - Build and run on multiple environments.
  - Use compiler flags like `-m32` to generate 32-bit binaries.
  - **Feature testing:** Use tools like Autoconf, CMake to check for features, not just OS.

Flag	Purpose
`-m32`	Force 32-bit compilation on GCC
default	Typically compiles to 64-bit binary

- **Note:** Portability checking can be expensive due to the combinatorial explosion of possible environments.
- **Best Practice:** Prefer feature checks over OS checks. Example:

```
#ifdef HAVE_RENAMEAT2
  // Use renameat2
#else
  // Fallback
#endif
```

Approach	Pros	Cons
OS-based checks	Simple, direct	Brittle, not future-proof
Feature checks	Robust, portable	More setup, needs tooling

3. Test Cases

- **Purpose:** Not to prove code works, but to find bugs.
- **Mindset:** "If my test cases didn't find a bug, I failed."
- **Types of tests:**
  - **Unit tests:** Test individual functions/components.
  - **Integration tests:** Test interactions between components.
  - **Regression tests:** Ensure old bugs stay fixed.
  - **Fuzz tests:** Randomized input to find edge cases.
- **Test infrastructure:**
  - Automate test execution (shell scripts, `make check`, CI systems).
  - Run tests in parallel (`make -j N`).
  - Separate quick/cheap tests (run frequently) from heavy/expensive tests (run less often).
- **Tools:**
  - Scripts (e.g., `run_tests.sh`)
  - Makefiles with `check` targets
  - GitHub Actions/CI
- **Test case generation:**
  - LLMs (Large Language Models) are effective for generating test cases (Meta found 40–50% LLM contribution is optimal).
- **Randomness testing:**
  - Impossible to mathematically prove randomness.
  - Can test statistical properties (bit balance, lack of patterns, distribution coverage).
  - Example of a bad random generator that passes naive tests:

    ```
    return UINT64_MAX / 3; // Alternating bits
    ```

- **Test Coverage:**
  - **Statement coverage:** Every line executed.
  - **Branch coverage:** Every branch taken.
  - **Path coverage:** Every possible path (usually infeasible for large programs).

Test Type	Scope	Detects
Unit	Function	Local logic errors
Integration	Subsystem	Interface bugs
Regression	Whole system	Recurring bugs
Fuzz	Whole system	Edge cases, crashes

4. Defensive Programming

- **Goals:**
  - Prevent bugs before they occur.
  - Minimize the impact of bugs that do occur.

    - Detect bugs early and reliably.
- **Techniques:**
    - Static checking (compiler warnings)
    - Dynamic checking (sanitizers, Valgrind)
    - Test case design for failure
    - Defensive coding (asserts, bounds checks)
    - **Input validation:** Always check user input for validity.
    - **Fail-fast:** Abort early on error to avoid propagating bad state.
    - **Error handling patterns:** Return error codes, use exceptions, or error objects as appropriate.

Defensive Technique	Example/Pattern	Benefit
Input validation	`if (!valid(x)) return -1;`	Prevents bad data
Assert invariants	`assert(ptr != NULL);`	Catches bugs early
Fail-fast	`exit(1)` on error	Avoids cascading failures
Error codes	`return -1;`	Explicit error propagation
Exception handling	`try { ... } catch { ... }`	Structured error management

5. Terminology: Error, Fault, Failure

Term	Description	Real-World Analogy
Error	Developer's mistake (mental/conceptual)	Forgetting to check tire pressure
Fault	Error reflected in the code (latent bug)	Flat tire in the car
Failure	Fault triggered during execution (observable bug)	Car accident

- **Debugging process:**
    - Start with symptoms (failure), trace back to fault (code), then to error (developer's mistake).

```
[Error: Developer's Mistake]
           |
           v
   [Fault: Bug in Code]
           |
           v
[Failure: Runtime Symptom]
           |
           v
[Debugging: Trace Back]
           |
           v
[Fix: Correct the Error]
```

Step	What You See	What It Means
Failure	Crash, wrong output	Something went wrong
Fault	Bug in code	Root cause in implementation
Error	Mental slip	Design/logic misunderstanding

6. Debugging Best Practices

6.1 Steps

1. **Reproduce/Stabilize the failure**
    - Make the bug consistent and repeatable.
    - May require disabling features like ASLR (Address Space Layout Randomization).
2. **Locate the fault**
    - Use backwards reasoning from symptoms to code.
    - Use debugger features to narrow down the cause.
3. **Minimize the test case**
    - Reduce input to the smallest case that still triggers the bug.
4. **Form a hypothesis**
    - Based on evidence, guess the root cause.

5. **Test the hypothesis**
    - Change code or add diagnostics to confirm/refute.
6. **Fix and retest**
    - Correct the bug and rerun all tests to ensure the issue is resolved.

```
[Observe Failure]
      |
      v
[Reproduce & Stabilize]
      |
      v
[Minimize Test Case]
      |
      v
[Form Hypothesis]
      |
      v
[Test Hypothesis]
      |
      v
[Fix & Retest]
```

6.2 Anti-patterns

- Randomly modifying code lines hoping it works (futile, non-scalable).
- Avoid using GDB as a crutch; it's for reasoning, not fixing.
- Ignoring compiler warnings.
- Not automating tests (manual testing is error-prone).
- Failing to minimize test cases (harder to debug).

Debugging Checklist

- ☐ Can you reproduce the bug reliably?
- ☐ Have you checked compiler warnings and static analysis?
- ☐ Have you run dynamic tools (sanitizer, Valgrind, fuzzer)?
- ☐ Have you minimized the test case?
- ☐ Do you understand the code path leading to the bug?
- ☐ Have you confirmed the fix with all relevant tests?

7. GDB: Debugger Deep Dive

7.1 GDB's Role

- GDB is a "Program Execution Explorer."
- GDB controls the debugged process by communicating with the OS kernel.
- Can start a program, attach to a running process, or modify program state.

7.2 Key GDB Commands

Command	Purpose
`run` / `r`	Start the program within GDB
`quit` / `q`	Exit GDB
`attach PID`	Attach to an already running process
`detach`	Detach from the debugged process

Setup Commands

Command	Description
`set cwd /path`	Set working directory for debugged proc
`set env VAR value`	Set environment variable
`set disable-randomization off`	Enable ASLR

- **ASLR (Address Space Layout Randomization):**
    - Randomizes memory layout to prevent exploits.

- Hurts reproducibility for debugging.
- GDB disables ASLR by default for reproducibility.

7.3 Breakpoints

Command	Description
`break <loc>` / `b`	Set breakpoint at function or line
`info break` / `ib`	List all breakpoints
`delete <num>` / `d`	Remove specified breakpoint
`cond <num> <expr>`	Set condition for breakpoint (advanced)

- **Implementation:**
  - GDB replaces the instruction at the breakpoint with a trap instruction.
  - When the program hits the trap, it stops and GDB regains control.
  - **Hardware breakpoints:** Use CPU support, limited in number (e.g., 4 on x86-64).
  - **Software breakpoints:** Unlimited, but slower (require code modification).

7.4 Control Commands

Command	Action
`continue` / `c`	Resume execution after break
`step` / `s`	Step into the next line of source code (includes functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns

- **Note:** Stepping can be confusing with optimized code; use `-O0` and `-g3` for best results.

7.5 Advanced Commands

Command	Purpose
`reverse-continue` / `rc`	Execute backwards to previous state (requires special setup, slows down)
`watch <expr>`	Set a watchpoint to break when expression value changes
`checkpoint` / `restart`	Save and reload program state (manual reverse execution)
`print <expr>` / `p`	Evaluate and print a variable or expression
`define <macro>`	Define a custom GDB macro (automation)

- **Reverse execution:**
  - `rc` requires GDB to keep snapshots of program state, which is slow and memory-intensive.
  - `checkpoint`/`restart` is a manual, more efficient alternative.
- **Watchpoints:**
  - Hardware support is limited (e.g., x86-64 supports 4 hardware watchpoints).
  - Software watchpoints are much slower.
- **Conditional breakpoints:**
  - Only break when a condition is true (e.g., `b foo if x == 42`).
- **Macros and scripting:**
  - Automate repetitive tasks with GDB's macro language or Python scripting.
- **Remote debugging:**
  - Debug programs running on another machine (e.g., embedded systems) via network or serial port.

7.6 Print Command Usage

- Print variables, expressions, or call functions:

```
p a + b
p my_struct.member
p my_function()
p exit(1)  // Dangerous: will cause program to terminate
```

7.7 Common GDB Pitfalls and Solutions

Pitfall	Solution/Workaround
Optimized code hard to debug	Compile with `-O0 -g3`
Variables "optimized out"	Use less optimization, check symbol table
Stepping skips lines	Use `stepi` for instruction-level control
Breakpoints not hit	Check for inlined/optimized code
Watchpoints not triggering	Use hardware watchpoints, minimize scope

---

Summary:

- Use static and dynamic tools together for best coverage.
- Defensive programming and rigorous testing are key to robust code.
- Debugging is a process: observe, reproduce, minimize, hypothesize, test, fix, and retest.
- GDB is powerful, but should be used thoughtfully—not as a crutch.
- Always automate and document your debugging and testing processes.

Lecture 16: Vibe Programming and AI-Driven Software Development

1. Introduction: Vibe Programming Overview

- Guest lecture by Prof. Kerry Nachenberg, with a demonstration-based format and live coding.
- Focus: Vibe Programming—AI-assisted software development, current capabilities, and workflows of AI agents in software construction.

---

2. What is Vibe Programming?

2.1 Definition and Origin

- Term coined by Andrej Karpathy (AI expert).
- Karpathy's description:

  > "Fully give into the vibes, embrace exponentials, and forget that the code even exists."

- ChatGPT-3 definition:

  > "An informal AI-assisted approach where you prototype by chatting with a model, iterating code, tweaking prompts, and following intuition instead of detailed upfront specs."

- Aspirational: True "vibe" programming is not fully realized yet; debugging and requirements are still necessary.

Key Distinction:

- Vibe Programming is not about abandoning rigor or requirements—it's about leveraging AI to accelerate prototyping, iteration, and learning, while still applying critical thinking and validation.

2.2 "Embrace Exponentials"

- AI capabilities are improving at an exponential rate.
- Example: LLMs can now complete hour-long human tasks with much greater success than before; progress doubles roughly every 7 months.
- **Implication:** The tools and workflows you use today may be outdated in a year—adaptability is crucial.

---

3. The Spectrum of Vibe Programming

Phase	Description	Example Use Case
LLM Assistance	Use AI for small tasks (e.g., generating functions or boilerplate code).	"Write a Python function to parse CSV files."
Collaboration	Human and AI co-develop; AI handles sections, humans debug and integrate.	"Build a web app skeleton, I'll add business logic."
Full AI Generation	AI builds complete applications without human-written code (e.g., Pac-Man).	"Generate a playable Pac-Man clone from scratch."

- **Real-world adoption:** Y Combinator reports 20% of teams use AI for most of their product, with 95% of code written by AI in some startups.
- **Most effective today:** Collaboration, not full AI generation.

ASCII Diagram: Spectrum of Vibe Programming

```
[LLM Assistance] <--> [Collaboration] <--> [Full AI Generation]
    ^                    ^                        ^
    |                    |                        |
  Human-driven      Human+AI team           AI-driven, human reviews
```

---

4. Tools of Vibe Programming

4.1 Cursor (Modified VS Code)

- IDE based on VS Code, optimized for AI-assistance.
- Supports multi-file projects, shell access, testing integration.
- Eliminates manual file management (no more copy-pasting from ChatGPT).

4.2 Other Tools

- Claude Code

- Codex (OpenAI)
- Google's Gemini

4.3 Tool Selection Guidance

Tool	Best For	Limitations
ChatGPT	One-off scripts, simple files	No multi-file/project support
Cursor	Multi-file, iterative projects	Requires setup, learning curve
Claude/Gemini	Large codebases, code review, explanations	May lack IDE integration

Agentic vs. Non-Agentic Tools:

- **Agentic tools** (e.g., Cursor) can read/write files, run shell commands, and interact with your environment.
- **Non-agentic tools** (e.g., ChatGPT web) only generate text/code and require manual copy-paste.

---

5. Understanding Agentic AI

5.1 What is an Agent?

- A software system that couples:
  - An LLM (e.g., GPT-4, Claude)
  - A Tool/API Layer (e.g., filesystem, shell, calendar, process execution)
- Agents interpret LLM output (often JSON commands) and execute actions in the real world.

Example: Agentic vs. Non-Agentic Workflow

```
Non-Agentic:
[User] <-> [LLM] (text only)

Agentic:
[User] <-> [Agent (LLM + Tools)] <-> [Filesystem/Shell/External APIs]
```

5.2 Components in Agentic Framework

1. **Client (IDE):** Cursor (VS Code fork); interfaces with local files/shell.
2. **Backend Server:** Hosted by Cursor to enrich/route prompts and manage command schemas.
3. **LLM:** External model (OpenAI, Google, Anthropic) used to generate commands and logic.

5.3 Agent Commands (Examples)

Action	Command Format (JSON-like)
Google Search	{ "command": "search", "query": "what is pathfinding?" }
Read From File	{ "command": "read_file", "path": "src/main.ts" }
Write File	{ "command": "write_file", "path": "game.ts", "data": "..."}
Execute Shell Command	{ "command": "shell", "cmd": "npm start"}
Display Message	{ "command": "message", "text": "Task completed successfully!"}

- Agentic frameworks use a protocol (e.g., MCP) to standardize these commands.

---

6. Agentic Workflow Architecture

6.1 Flow of Execution

1. User issues a command (e.g., "Fix bug in foo").
2. Sent to backend server, which appends available command schemas.
3. Prompt sent to LLM.
4. LLM returns a command (e.g., read a file).
5. Backend interprets and executes the command via client.
6. Feedback (e.g., file contents) appended to prompt.
7. Repeat until desired output is attained.

Step-by-Step Example:

- User: "Add a login page."
- Agent reads `app.tsx`.
- Agent writes new `Login.tsx` file.
- Agent updates `routes.tsx`.
- Agent runs `npm install react-router-dom`.
- Agent notifies user: "Login page added."

6.2 Prompt Management

- Every interaction builds on prior ones; context accumulates.
- Accumulated context can confuse the model ("crud accretes").
- Solution: Re-initialize sessions regularly to avoid confusion.

Improved ASCII Diagram: Agentic Workflow (with context accumulation)

```
+--------+      +-------------+      +--------------+      +-----+
|  User  | ---> | Cursor IDE  | ---> | Cursor Server| ---> | LLM |
+--------+      +-------------+      +--------------+      +-----+
     ^                 |                    |                |
     |                 |<-------------------|<---------------|
     |                 |   (context, files, |   (commands,   |
     |                 |    feedback, etc.) |    responses)  |
     |                 +------------------------------------+
     |                        (context accumulates)
     +-------------------------------------------------------+
```

---

7. Key Concepts Before Starting Vibe Programming

7.1 Coding Rules Configuration

- Specify constraints and style guidelines for the LLM.
- Examples:
  - Reuse/refactor existing code when possible.
  - Do not introduce new frameworks/technologies without confirmation.
  - Make minimal required changes.
- These rules are included in the prompt for every request.

7.2 Requirements Gathering

- Essential for successful Vibe Programming, especially for complex projects.

Table: Requirements Gathering Components

Component	Description
Functional Req.	What the software must do (UI, game physics, etc.)
User Stories	How users interact with the system
Framework/API Info	Preferred tech stack
UI Mockups	Visual layouts, wireframes, Figma diagrams
Business Rules	Backend logic and constraints
Testing Requirements	How the code is tested (unit tests, etc.)
Performance/Security	Runtime and safety expectations
Acceptance Criteria	Success conditions for tasks/project

Step-by-Step Example: Requirements to PRD

1. **Gather Requirements:**
   - Interview stakeholders, collect user stories, define business rules.
2. **Draft PRD:**
   - Write a Product Requirements Document with purpose, features, scope, stretch goals.
3. **Review and Refine:**
   - Share PRD with team, iterate based on feedback.
4. **Break Down Tasks:**

- Create a dependency-aware task list from PRD.

Example PRD Outline:

```
Product: 3-Player Pong Game
Purpose: Enable 3 users to play Pong simultaneously.
Features:
  - Three paddles, one for each player
  - Ball physics and collision
  - Score tracking
  - Stretch: Power-ups, AI opponent
Acceptance Criteria:
  - Game runs in browser
  - All paddles respond to input
  - Game ends when a player reaches 10 points
```

---

8. Demonstration Project: 3-Player Pong Game in TypeScript

8.1 Process Overview

1. **PRD Generation (Product Requirements Document):**
   - Formal structure: purpose, features, scope, stretch goals.
   - Can be generated by GPT using a task description.
2. **Dependency-Aware Task List:**
   - Created from PRD; includes ordered and split sub-tasks.
   - Example tasks: setup project, create main loop, paddle class, ball class, input handling, collision detection.
3. **Implementation using Cursor:**
   - Tasks executed in order via automated prompting.
   - Human triggers one task at a time; Cursor performs file creation, code generation, package installation, UI/gameplay updates.

8.2 Implementation Details

- Each task is executed in a fresh chat session to avoid context confusion.
- If a task goes "off the rails" (e.g., major bugs, wrong language/framework), revert to last checkpoint and try again.
- Validate each step by running the code and checking results.
- Use diagnostics/logging and model switching for debugging.

ASCII Diagram: Task Execution Loop

```
[Start Task] -> [Fresh Chat] -> [Model Executes Task] -> [Validate Output]
      ^                                                  |
      |                                                  v
[If Off Rails: Revert] <------------------------ [If Bug: Debug/Retry]
```

---

9. Debugging with Vibe Programming

9.1 Bugs and Failures

- "Off-rails": Non-functional or highly illogical generations.
- Partial bugs: Small issues amidst working components.

9.2 Debugging Procedure

1. Start a clean chat session.
2. Clearly describe the issue.
3. Ask for multiple root cause suggestions (not just one).
4. Add diagnostics (e.g., logging) as suggested.
5. Run the program and gather output/logs.
6. Request a targeted fix based on new information.
7. If the fix fails, revert to checkpoint and/or try a different model.

Table: Debugging Workflow

Step	Action	Purpose

Step	Action	Purpose
1	Start clean chat session	Avoid context confusion
2	Describe the issue	Give the model clear context
3	Ask for multiple root causes	Get a range of hypotheses
4	Add diagnostics/logging	Gather more information
5	Run and collect output	Provide evidence for next step
6	Request targeted fix	Focus on the actual problem
7	Revert/try different model if needed	Avoid compounding errors

Debugging Example:

- Issue: Game crashes when ball hits paddle edge.
- Steps:
    1. Start new chat, describe bug.
    2. Ask: "What are 3 possible causes?"
    3. Add logging to collision function.
    4. Run and collect logs.
    5. Ask for targeted fix based on logs.

9.3 Debugging Strategies

- Avoid "fix on top of fix"; revert instead.
- Use diagnostics to iteratively narrow down the issue.
- Try different LLMs (Claude, Gemini, etc.) for fresh perspectives.
- Use revert/restore for checkpointing.

---

10. Pro Tips for Effective Vibe Programming

Tip	Explanation
Use Correct Language and Platform	AI performs best in familiar environments (e.g., TypeScript for web, Python for ML).
Add Tests to Requirements	Guides AI to generate better, verifiable code.
Avoid Fixes on Fixes	Causes accumulation of errors; revert instead.
Paste API Docs	Enables LLM to better call external services reliably.
Ask for Refactoring	AI can improve and clean up code it has previously written.

- Choose the right language/platform for the task (e.g., TypeScript for games, Python for ML).
- Add testing requirements to ensure code is verifiable and self-debugging.
- Paste API documentation directly into the project for better integration.
- Ask for code refactoring to improve maintainability and reduce duplication.

---

11. Future of AI in Software Development

11.1 Job Market Perspective

- AI can't fully replace human intuition, system design, or novel reasoning (yet).
- Developer roles may shift from writing to orchestrating and reviewing code.
- Increasing importance of product thinking: defining requirements clearly.
- Collaboration, not handoff: AI is a tool, not a replacement.
- **Q&A Insight:** The future of software jobs is not "no code" or "all code"—it's a blend. Not using AI is a disadvantage, but over-reliance leads to poor code quality. Human review and understanding remain essential.
- **Domain expertise** is critical for security and correctness. Use AI for productivity, but always review and understand the output, especially for security-sensitive or complex systems.

11.2 Educational Use

- Ideal for learning/understanding concepts and debugging unfamiliar code.
- Over-dependence can hinder foundational coding skill development.
- Use AI to explain code and concepts, not just to generate code.
- **Advice:** Use AI to help understand, not just to code. For learning, generate code with AI, then ask for line-by-line explanations to build intuition.

- **Warning:** Studies show that using AI for code completion can hurt learning and intuition if overused. Use it to supplement, not replace, your own problem-solving.

Learning Strategies Table:

Strategy	Benefit	Risk if Overused
Generate code, then explain	Deepens understanding, builds intuition	May skip critical thinking
Use AI for debugging unfamiliar	Accelerates learning, exposes new patterns	May not learn root cause analysis
Ask for refactoring suggestions	Improves code quality, exposes best practices	May not learn to refactor solo
Rely only on completions	Fast, but shallow learning	Weakens problem-solving skills

---

12. Empirical Data on AI in Software Development (Dr. Eggert's Segment)

12.1 JetBrains 2024 Developer Survey

- 63% of developers save less than 4 hours/week using AI tools (10% productivity boost for a 40-hour week).
- Main benefits: less time searching, faster coding, easier repetitive tasks.

12.2 Meta / Satya Nadella Observations

- Microsoft: 20–30% of repo code is AI-generated; Meta aims for >50%.
- Meta's internal study: AI involvement from 10% to 80% of code; lowest bug rates at 40–50% AI involvement.
- AI excels at:
  - Boilerplate/framework code
  - Large-scale refactoring
  - Migration scaffolds
  - Test case generation
- AI struggles with:
  - Complex logic paths
  - Context-heavy features
  - Real systems thinking/architecture
  - Stateful or edge-case-heavy code

Table: AI Strengths and Weaknesses

Strengths	Weaknesses
Boilerplate/framework code	Complex logic paths
Large-scale refactoring	Context-heavy features
Migration scaffolds	Real systems thinking/architecture
Test case generation	Stateful/edge-case-heavy code

ASCII Diagram: AI Involvement vs. Bugs

```
AI Involvement (%)
|
|   *         *
|  * *       * *
| *   *     *   *
|*     *   *     *
+----------------------
 0   20   40   60   80
(Bug rate lowest at 40-50% AI involvement)
```

12.3 Key Concerns

Concern	Explanation	Actionable Advice
Privacy & Data Leakage	Risk of sending proprietary code to external models.	Use private/incognito modes, avoid sensitive data.
IP Ownership	Who owns AI-generated code?	Check company policy, monitor legal developments.
Maintenance Complexity	Large, AI-written codebases may be harder to maintain.	Enforce code review, require documentation.

Concern	Explanation	Actionable Advice
Security Vulnerabilities	Risk of inserting exploitable code (e.g., SQL injection).	Use static analysis, review with a second model.
Data Poisoning Risk	Adversarial inputs training the AI to produce backdoors.	Use trusted sources, review generated code.

- **Additional issues:**
  - Leakage of private information (e.g., medical/student data, company secrets, student grades). Use private/incognito modes if available.
  - Unclear IP/copyright status of AI-generated code. Legal landscape is evolving; be cautious with proprietary or commercial projects.
  - Maintenance and operational costs may increase with AI-generated code due to bloat or lack of documentation.
  - Security: AI may generate insecure code, especially if trained on bad examples or adversarial code. Use a second model to review for vulnerabilities.
  - **Data poisoning:** Malicious actors could publish insecure code to "poison" AI training data, leading to vulnerabilities in generated code.

---

13. Summary

This lecture explored the evolving paradigm of Vibe Programming—a method of AI-assisted software development that leverages large language models and integrated agentic systems to generate and manage code. The concept, introduced by Andrej Karpathy, is built on a spectrum from simple LLM assistance to full AI-driven codebases. The lecture emphasized the importance of clear requirements, tooling like Cursor for agentic workflows, and formalized processes such as PRD and dependency-aware task lists.

It provided a deep dive into the architecture of agentic AI, demonstrating how JSON-based command exchanges between user, server, and model guide automated code generation and problem-solving within development environments. Debugging strategies, coding rules, and requirement gathering were also covered, showing how developers can maximize AI's utility while avoiding pitfalls like hallucinations, model confusion, and inappropriate code rewrites.

The session concluded with insights into the current and future job market implications, security risks, and empirical observations on AI effectiveness, suggesting that while AI tools are improving developer productivity, human oversight and understanding remain indispensable for building robust and secure systems.

Final Takeaways:

- Use AI as a collaborator, not a replacement.
- Always review, test, and understand AI-generated code, especially for security and correctness.
- Clear requirements and domain expertise are more important than ever.
- The legal, educational, and security landscape is rapidly evolving—stay informed and cautious.

Introduction to GDB, Python Classes, and Packaging

1. Debugging in GDB

1.1 Should We Demo Debugging Tools?

- Discussion on whether live demonstrations of debugging tools (like GDB) should be included in lectures.
- **Pros of demos:**
  - Shows real-time problem-solving and how functions behave in practice.
- **Cons:**
  - Demos take more time and might cover fewer topics.
  - Risk of being dull if not executed well.

1.2 GDB Essentials

1.2.1 Common GDB Commands

Command	Description
`run` / `r`	Start the program within GDB
`quit` / `q`	Exit GDB
`break <loc>` / `b`	Set breakpoint at function or line
`info break` / `ib`	List all breakpoints
`delete <num>` / `d`	Remove specified breakpoint
`continue` / `c`	Resume execution after break
`step` / `s`	Step into the next line (enters functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns
`print <expr>` / `p`	Evaluate and print a variable or expression
`watch <expr>`	Break when expression value changes

1.2.2 Breakpoints: Types and Management

- **Regular breakpoints:** Pause execution at a specific line or function.
- **Conditional breakpoints:** Only trigger if a condition is true.

```
b sqrt
condition 27 x < 0
```

- **Hardware breakpoints:** Use CPU support for efficient watchpoints (limited number).
- **Pitfalls:**
  - Breakpoints may not trigger if code is optimized/inlined.
  - Use `info break` to check all breakpoints.
  - Remove with `delete <num>`.

1.2.3 Stepping and Execution Control

Command	Action
`step` / `s`	Step into the next line (enters functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns

- **Note:** Stepping can be confusing with optimized code; compile with `-O0 -g3` for best results.

1.2.4 Stack-Related Commands

Command	Full Form	Description
`i f`	`info frame`	Shows information about the current function stack frame.
`bt` or `i s`	`backtrace` or `info stack`	Lists call stack trace, showing how program arrived at current function.

- Stack frames record what function called what.
- **Machine-level optimizations** (e.g., tail-call optimization) may cause missing frames in the trace. For example, if a function's last action is to call another, the intermediate frame may be omitted.
- Local variables may be optimized out and hence unavailable in GDB.
- Backtrace only records enough information for the system to continue execution, not every detail of the call history (e.g., loops are not shown).

1.2.5 Expression Evaluation

- `print expr`: Standard way to evaluate an expression.
- `display expr`: Continuously print value after each step.
- **Caution:** Calling functions from GDB can have side effects and may alter program state.
- GDB allows tooling with Python to customize how structures (like graphs) are displayed.
- You can write Python code to help GDB print complex data structures in a readable way.

1.2.6 Register Inspection

```
i r
p $xmm0
```

- `info registers`: Shows machine register values.
- `print $register`: Print a specific register value.
- Useful for performance checks and low-level debugging.
- Registers are fast-access storage; inspecting them can reveal performance-critical code paths.
- **Example:**

  ```
  (gdb) info registers
  (gdb) print $rax
  (gdb) print $xmm0
  ```

1.3 Remote and Cross-Platform Debugging

1.3.1 Debugging Different Architectures

```
target architecture-name
```

- Allows debugging code meant for different architectures (e.g., ARM, x86-64).
- GDB can debug programs running on a different architecture than the host.

1.3.2 Remote Debugging

- GDB can debug programs running on remote machines via serial ports or network (TCP/IP).
- Useful for embedded systems or IoT devices lacking GDB capabilities natively.
- Requires a copy of the executable on both the host and target machines.
- Configuration may involve specifying architecture and connection details (e.g., SSH, serial port).
- **Typical workflow:**
  1. On target: `gdbserver :1234 ./myprog`
  2. On host: `gdb ./myprog`, then `target remote <ip>:1234`
- **Security:** Only use on trusted networks or with SSH tunneling.

```
[Host GDB] --network--> [gdbserver on Target]
      |                         |
   (local exe)           (running exe)
```

1.4 GDB Macros and Extensions

1.4.1 GDB Macros

```
define printlook
  print *(long*)$arg0
end
```

- Custom debugging commands.
- `$arg1`, `$arg2`, etc., can be used for parameter substitution.
- GDB has its own macro language for automating repetitive tasks.
- **When to use:** For simple automation and repetitive command sequences.

1.4.2 Python Extensions in GDB

- GDB supports scripting in Python for advanced automation and pretty-printing.
- **Example:**

```
# .gdbinit or loaded via 'source'
python
class MyPrinter:
    def __init__(self, val):
        self.val = val
    def to_string(self):
        return f"MyStruct: {self.val['field']}"
def register_printers(objfile):
    gdb.pretty_printers.append(MyPrinter)
end
```

- **When to use:** For complex data visualization, integration, or automation.
- **Comparison:** Use macros for simple tasks, Python for complex logic.

1.5 Alternative Debugging Strategies

1.5.1 Comparison Table: Assertions, Exceptions, Logging, Tracing

Strategy	Purpose	Example	Pros	Cons
Assertion	Catch bugs early	`assert(x >= 0);`	Fast, simple, runtime check	Can be compiled out, no recovery
Exception	Handle errors	`try: ... except ...`	Structured, recoverable	Can obscure flow, overhead
Logging	Record events	`print()`, logs	Persistent, post-mortem	Can be noisy, performance
Tracing	External monitoring	`strace`, `ltrace`	No code change, system-wide	Limited detail, setup needed

1.5.2 Debugging Process Diagram

```
[Observe Failure]
      |
      v
[Reproduce & Stabilize]
      |
      v
[Minimize Test Case]
      |
      v
[Form Hypothesis]
      |
      v
[Test Hypothesis]
      |
      v
[Fix & Retest]
```

1.5.3 Assertions

C-style:

```
assert(x >= 0);
```

- If `x < 0`, the program prints error and aborts.
- Can be compiled away using `-DNDEBUG`.
- **Restriction:** Assertions should be free of side effects; otherwise, program behavior may differ when assertions are compiled out.
- Difference from `unreachable`: Assertions are runtime checks; `unreachable` is a message to the compiler about code paths.

1.5.4 Exception Handling

Python-style:

```
try:
    do_something()
except ZeroDivisionError:
    handle_issue()
```

Alternative C-style:

```
if (do_something() < 0) {
    handle_issue();
}
```

Feature	Try-Catch	Manual Checking
Clearer mainline code	Yes	No
Easier to track flow	No	Yes

- **Try-catch**: Mainline code is clearer, but error handling can be less explicit.
- **Manual checking**: Error handling is explicit, but mainline code is cluttered.
- Both have pros and cons; know when to use each.

1.5.5 Logging and Tracing

- Print/log statements to track execution.
- **Logs:** Explicit in code (e.g., print statements, server logs). Levels: DEBUG, INFO, WARNING, ERROR, CRITICAL.
- **Traces:** Generated by external tools (e.g., `strace` for Linux, which logs system calls).
- Useful for debugging production systems where attaching a debugger is impractical.
- **Best practices:** Use log levels, avoid logging sensitive data, rotate logs.

1.5.6 Checkpoint-Restart

- Save application state at intervals.
- On crash, reload checkpoint to reproduce or recover state.
- Useful for reproducing bugs in long-running programs.
- **Tools:** `checkpoint`/`restart` in GDB, application-level checkpointing libraries.
- **Limitations:** May not capture all state (e.g., open sockets, external resources).

1.5.7 Barricades

- Architecture for legacy modernization.
- Build a clean subsystem separated by a "barricade".
- Only allow access via validated interfaces.
- Barricade checks all data crossing from messy to clean code, ensuring reliability.

Versioned cleanup pattern:

1. Start with this structure:

```
[Messy Code] --barricade--> [Clean Code]
```

2. Gradually clean messy code and push barricade outward.

```
+----------------+    Barricade    +----------------+
|  Messy Legacy  | --------------> |    Clean New   |
|     Code       | <-------------- |     Code       |
+----------------+   (interface)   +----------------+
```

- Only validated data/operations cross the barricade.
- Over time, the barricade moves as more code is cleaned up.

1.5.8 Stronger Isolation Mechanisms

- **Interpreters:** Execute untrusted code through a carefully written interpreter. Slow but safe. Used in Chrome, Emacs, GDB, etc.
- **Virtual Machines and Containers:**
  - Virtual machines emulate entire systems, providing strong isolation.
  - Containers share host OS but isolate user space. Lighter, but less isolated than VMs.
  - Trade-off: Security vs. performance.

---

2. Python Programming Constructs

2.1 Functions and Lambdas

```
f = lambda x, y: x + y + 1
g = f
```

- Functions are first-class callables.
- Can be assigned to other variables, passed as arguments, or returned from functions.
- Lambda expressions create anonymous functions.
- **Example:**

```
def apply_func(f, x):
    return f(x)
print(apply_func(lambda y: y * 2, 5))  # 10
```

2.2 Python Classes and Inheritance

2.2.1 Classes as Objects

- In Python, classes are regular objects.
- Can be dynamically created and assigned.
- Each class has a `__dict__` that stores its attributes (a dictionary mapping names to values).
- `__dict__` is a reserved name; don't modify unless you know what you're doing.
- **Dynamic class creation example:**

```
MyClass = type('MyClass', (object,), {'x': 42})
obj = MyClass()
print(obj.x)  # 42
```

2.2.2 Inheritance Model and Method Resolution Order (MRO)

```
class C(A, B):
    ...
```

- Multiple inheritance allowed.
- Python uses **depth-first left-to-right** search for method resolution, avoiding re-visits.
- **MRO Example:**

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
```

```
print(D.__mro__)
# (<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>)
```

- **Comparison Table:**
Language	MRO Strategy
Python	Depth-first, left-to-right
C++	Depth-first, ambiguous
Java	Single inheritance only
JavaScript	Prototype chain

2.3 Namespaces and Scoping

- Namespace: Collection mapping names to objects (essentially a `dict`).
- Each class, module, and function defines its own namespace.
- Namespaces prevent name collisions and organize code.
- **Diagram:**

```
[Global Namespace]
      |
      +-- [Module Namespace]
              |
              +-- [Class Namespace]
                      |
                      +-- [Instance Namespace]
```

---

3. Python Modules and Import System

3.1 Basic Module Mechanics

```
# foo.py
a = 19
def f(x): return x + 1
class C: ...
```

- Modules are Python files (`.py`) that define variables, functions, and classes.
- `import foo` creates a new namespace, executes `foo.py` in that namespace, and binds `foo` in the importer's namespace.
- Use `foo.a` to access `a` from the module.
- **Pitfalls:**
  - Circular imports (A imports B, B imports A) can cause issues; Python tries to avoid re-importing modules.
  - Name collisions: If you already have a variable named `foo`, importing a module with the same name will overwrite it.

3.1.1 Selective Import

```
from foo import f
```

- Imports only `f` from `foo` into the current namespace.
- `from foo import *` imports all names, but can cause name collisions and is discouraged unless necessary.

3.1.2 Import Styles Table

Style	Example	Pros	Cons
`import foo`	`import math`	Namespace isolation	Verbose
`from foo import f`	`from math import sin`	Direct access, less typing	Name collisions possible
`from foo import *`	`from math import *`	Quick, all names imported	Collisions, unclear origin

3.1.3 Circular Imports: Detection and Solutions

- **Detection:** Import errors, partially initialized modules, or `AttributeError` at import time.
- **Solutions:**

- Refactor shared code into a third module.
- Use local imports inside functions.
- Avoid top-level code that triggers imports.

3.1.4 Inspecting Modules

- `dir(foo)` lists all names in the module.
- The `builtins` module contains all built-in functions and names.

3.1.5 Running Modules as Scripts

- Running `python foo.py` creates a new namespace and sets `__name__` to `__main__`.
- Common idiom:

```
if __name__ == "__main__":
    # test code or main application
```

- Allows modules to be used both as importable libraries and as standalone scripts.

3.1.6 Import System Search Path Diagram

```
[Current Directory] -> [PYTHONPATH dirs] -> [Standard Library] -> [Site-packages]
```

3.2 Packages and Hierarchical Organization

- Packages are directories containing an `__init__.py` file.
- Packages allow hierarchical organization of modules (tree structure).
- Dots in import names correspond to directory structure (e.g., `import a.b.c` looks for `a/b/c.py`).
- `__init__.py` can perform package-level initialization.
- Relative imports (e.g., `from . import foo`) allow navigation within the package hierarchy.
- **init.py roles:**
  - Marks a directory as a package.
  - Can execute initialization code or expose selected submodules.
  - Pitfall: Missing `__init__.py` in older Python (❤️.3) breaks imports.

ASCII Diagram: Python Package/Module Hierarchy

```
my_app/
├── __init__.py
├── utils/
│   ├── __init__.py
│   └── math_helpers.py
└── models/
    ├── __init__.py
    └── user.py
```

- Each directory with `__init__.py` is a package.
- Modules are `.py` files within packages.

3.2.1 Comparison Table: Packages in Python vs. Other Languages

Language	Package Structure	Initialization File	Import Syntax
Python	Directory tree	`__init__.py`	`import a.b.c`
Java	Directory tree	none	`import a.b.C`
Node.js	Directory tree + package.json	none	`require('a/b/c')`

3.2.2 Multiple Libraries and PYTHONPATH

- Python searches for modules using the `PYTHONPATH` environment variable (colon-separated list of directories).
- The `import` statement searches these directories in order.

3.2.3 Package Managers and Virtual Environments

- **pip**: Python's package manager for installing, upgrading, and removing packages.
  - `pip install numpy` installs the NumPy library.
  - `pip uninstall numpy` removes it.
  - `pip list` shows installed packages.
  - `pip list --outdated` shows outdated packages.
  - `pip list --format=json` outputs in JSON format.
- **Dependencies:** pip installs dependencies automatically, but this can sometimes pull in unwanted packages.
- **Extending Python:** Most extensions are done via libraries, but you can also modify the Python interpreter itself (written in C) for deeper changes. Such changes are proposed via Python Enhancement Proposals (PEPs).
- **Virtual environments:**
  - `venv` (standard library, Python 3.3+): `python -m venv venv`
  - `virtualenv` (older, more features): `virtualenv venv`
  - `pipenv`, `conda` (alternative tools): manage dependencies and environments.
  - **Best practice:** Always use a virtual environment for projects to avoid dependency conflicts.

3.2.4 Comparison to Other Ecosystems

- Node.js uses `npm` for package management; similar issues and workflows as Python's `pip`.

---

4. Build Tools and Packaging for Compiled Languages (Preview)

- Building and packaging for compiled languages (C, C++, Rust) is more complex than for interpreted languages.
- Executables are platform-specific; even machines with the same architecture may have subtle differences.
- The build/distribution/install process involves multiple roles:
  - **Developers**: Write the code.
  - **Builders**: Compile the code.
  - **Distributors**: Ship executables to users.
  - **Installers**: Install executables on end-user machines.
- Simple scripts can automate builds, but more robust tools are needed for complex projects.
- **Build automation tools:**
  - `make`, `CMake`, `Bazel`, `Ninja` for C/C++/Rust.
  - Handle dependencies, platform differences, and incremental builds.
- **Comparison Table: Interpreted vs. Compiled Build/Distribution**
Aspect	Interpreted (Python)	Compiled (C/C++/Rust)
Portability	High (source code)	Low (binary format)
Build Step	None (run directly)	Required (compile/link)
Distribution	Source or bytecode	Platform-specific binaries
Dependency Mgmt	pip, venv, etc.	System pkg mgr, static/dyn link

ASCII Diagram: Build Process Stakeholder Flow

```
[Developer]
     |
     v
[Builder]
     |
     v
[Distributor]
     |
     v
[Installer]
     |
     v
[End User]
```

- Each role may be automated or manual.
- Artifacts flow from source code to executable to installed application.
- **Build pipeline diagram:**

```
[Source Code] -> [Build Tool] -> [Binary/Package] -> [Distribution] -> [Installation] -> [User Execution]
```

Lecture 18: Building Software, Build Automation, and Databases

1. The Painstaking Process of Building Software

- Building and delivering software is a painstaking, error-prone process.
- Historically, all steps were done by hand: writing code, compiling, linking, running.
- In early CS courses (CS31/32), students manage these steps manually.
- **Theme:** Whenever a task is boring or repetitive, automate it! If you don't, someone else will.

---

2. Automating the Build Process

2.1 Naive Automation with Scripts

- Developers write shell or Python scripts to automate builds.
- Example shell script:

```
#!/bin/sh
gcc -c foo1.c
gcc -c foo2.c
gcc -c foo3.c
echo $1 > foo4.c
gcc -c foo4.c
gcc foo1.o foo2.o foo3.o foo4.o -o myprogram
```

- **Problems:**
  - Inefficient: recompiles everything, even if only one file changed.
  - Not scalable for large projects.
  - No partial builds; always does full rebuild.

2.2 The Need for Smarter Automation

- Ideal: Only rebuild what's necessary (incremental builds).
- Desire for a tool that can "partially execute" a build script, skipping unnecessary steps.

Example: Incremental Build with `make`

Suppose you change only `foo2.c`:

1. `make` checks timestamps:
   - `foo2.c` is newer than `foo2.o` → rebuild `foo2.o`.
   - `foo1.o`, `foo3.o`, `foo4.o` are up-to-date → not rebuilt.
   - `myprogram` is now older than `foo2.o` → relink all `.o` files.
2. Only the necessary steps are performed, saving time.

Step-by-step:

```
$ touch foo2.c           # Simulate editing foo2.c
$ make
  gcc -c foo2.c         # Only foo2.o is rebuilt
  gcc foo1.o foo2.o foo3.o foo4.o -o myprogram  # Relink
```

---

3. `make`: Smarter Build Automation

3.1 What is `make`?

- A tool that automates builds, only rebuilding what's needed.
- Input: a `Makefile` describing targets, dependencies, and build commands.
- Uses file timestamps to determine what needs rebuilding.

3.2 Makefile Structure

- Rules:

```
target: dependencies
    command-to-build-target
```

- Example:

```
myprogram: foo1.o foo2.o foo3.o foo4.o
    gcc foo1.o foo2.o foo3.o foo4.o -o myprogram

foo1.o: foo1.c
    gcc -c foo1.c

foo2.o: foo2.c bar.h
    gcc -c foo2.c

foo3.o: foo3.c
    gcc -c foo3.c

foo4.c:
    echo $1 > foo4.c

foo4.o: foo4.c
    gcc -c foo4.c
```

3.3 How `make` Works

- Compares timestamps: if a target is older than any dependency, it rebuilds.
- Skips steps if target is up-to-date.
- **Dependency Graph:** Internally, `make` builds a graph of targets and dependencies.
- Can detect cycles (e.g., A depends on B, B on A) and will error out.

Example: Make Dependency Graph (ASCII Diagram)

```
foo1.c   foo2.c   foo3.c   foo4.c
   |        |        |        |
   v        v        v        v
foo1.o   foo2.o   foo3.o   foo4.o
   \       |        /        /
     \     |      /        /
        v  v  v  v
        myprogram
```

3.4 Common Pitfalls

- **Forgetting dependencies:** If a file includes a header (e.g., `bar.h`) but the Makefile omits it, changes to the header won't trigger rebuilds.
- **Over-specifying dependencies:** Adds unnecessary rebuilds but ensures correctness.
- **Solution:** Use compilers (GCC/Clang) to auto-generate dependencies (e.g., `-MMD` flag).

Example: Missing Dependency

```
foo2.o: foo2.c
    gcc -c foo2.c
```

If `foo2.c` includes `bar.h` and `bar.h` changes, `foo2.o` will NOT be rebuilt! Fix:

```
foo2.o: foo2.c bar.h
    gcc -c foo2.c
```

Example: Dependency Cycle (Error)

```
A: B
B: A
```

This creates a cycle. `make` will report an error:

```
make: Circular A <- B dependency dropped.
```

Table: Makefile Pitfalls and Solutions

Pitfall	Consequence	Solution
Missing dependency	Outdated builds, bugs	List all headers as dependencies
Over-specifying	Unnecessary rebuilds	Use auto-generated dependencies
Cyclic dependencies	Build fails	Refactor dependency graph

3.5 Clock Skew

- Distributed builds can suffer from inconsistent clocks (clock skew).
- Can cause unnecessary or missed builds.
- **Alternative:** Use file checksums instead of timestamps (more management overhead).

3.6 Parallel Builds

- `make -j` runs independent build steps in parallel.
- Great for large projects and multi-core machines (e.g., Linux kernel builds).

3.7 Scaling Build Systems

- **Recursive make:** Each subdirectory has its own Makefile; top-level Makefile runs `cd subdir && make`.
  - Harder to parallelize.
- **Single top-level Makefile with includes:**
  - Centralized control, better parallelism.
  - Uses `include` directives to pull in sub-makefiles.

3.8 Modern Build Systems

- **CMake:** Generates platform-specific Makefiles or Ninja files; supports complex projects and cross-platform builds.
- **Ninja:** Fast, minimal build system; often used as a backend for CMake.
- **Bazel:** Scalable, hermetic builds; used by Google and others for large codebases.
- **Meson:** Modern, fast, user-friendly; generates Ninja files.

Tool	Strengths	Typical Use Case
Make	Simple, ubiquitous	Small/medium C/C++ projects
CMake	Cross-platform, feature-rich	Large, portable projects
Ninja	Extremely fast, simple syntax	Backend for CMake/Meson
Bazel	Scalable, reproducible, language-agnostic	Monorepos, CI/CD
Meson	Modern, easy to use, fast	Modern C/C++/Python builds

---

4. Portability and System Configuration

4.1 The Problem

- Not all platforms support the same system calls (e.g., `renameat2()` on Linux, not Windows).

4.2 Conditional Compilation

- Use C preprocessor to select implementations:

```c
#ifdef __linux__
    // Use renameat2
#else
    // Use fallback logic
#endif
```

- **Problem:** OS-based checks can break if OSes add/remove features.

4.3 Feature Testing (Preferred)

- Instead of checking OS, check if the feature exists.
- Write a test program to probe for the feature during build.
- Generate a config header (e.g., `config.h`):

```
#define HAS_RENAMEAT2 1
```

4.4 Build-Time Configuration Tools

Tool	Purpose
`configure`	Probes system, generates `config.h`
`autoconf`	Generates the `configure` script
`automake`	Generates portable `Makefile`s

- These tools automate and layer the build configuration process.

ASCII Diagram: Autotools Configuration Flow

```
[autoconf] ---> [configure script]
      |                |
      v                v
[configure.ac]     [User runs ./configure]
                        |
                        v
                [config.h, Makefile]
```

---

5. Packaging in Higher-Level Languages

5.1 Python Example

- Uses `pyproject.toml` for metadata and build configuration.

```toml
[build-system]
requires = ["setuptools >= 77.0"]
build-backend = "setuptools.build_meta"

[project]
name = "myproject"
version = "1.0.0"
authors = [{ name = "Your Name", email = "you@example.com" }]
license = "GPL-3.0"
```

- **Warning:** Faulty configuration files can break installs and compatibility.

Table: Python Packaging Tools

Tool	Purpose
pip	Install Python packages
setuptools	Build and distribute packages
wheel	Binary package format
virtualenv	Isolated Python environments
pyproject.toml	Modern build/config metadata

Example: setup.py

```
from setuptools import setup
setup(
```

```
    name='myproject',
    version='1.0.0',
    author='Your Name',
    packages=['myproject'],
    install_requires=[],
)
```

---

6. Databases: Why Not Just Use File Systems?

6.1 Motivation

- File systems are unstructured, lack consistency guarantees, and are slow for search.
- Databases provide:
  - Structured access
  - Consistency
  - Efficient search
  - Concurrency-safe access

6.2 Relational Databases

- **Data Model:** Tables (relations) with rows and columns.
- **Columns:** Typed (e.g., `INT`, `VARCHAR(255)`).
- **Rows:** Each is a record/entity. Usually unique (set semantics), but some DBs allow duplicates (multiset).
- **Keys:**
  - **Primary key:** Uniquely identifies a row (e.g., student ID).
  - **Foreign key:** References a key in another table; enforces referential integrity.
- **NULL:** Some DBs allow missing values (NULL), leading to three-valued logic (TRUE, FALSE, UNKNOWN).

6.3 Relational Algebra

- **Selection:**

  ```
  SELECT * FROM student WHERE major = 'CS';
  -- Returns only rows where major is CS
  ```

- **Projection:**

  ```
  SELECT DISTINCT major FROM student;
  -- Returns unique majors only
  ```

- **Join:**

  ```
  SELECT student.id, student.major, class.class
  FROM student JOIN class ON student.id = class.student_id;
  ```

Expanded Examples

- **Selection:**

  ```
  SELECT * FROM student WHERE major = 'CS';
  -- Returns only rows where major is CS
  ```

- **Projection:**

  ```
  SELECT DISTINCT major FROM student;
  -- Returns unique majors only
  ```

- **Join:**

```
SELECT student.id, student.major, class.class
FROM student JOIN class ON student.id = class.student_id;
```

Operation	SQL Example	Result Description
Selection	SELECT * FROM student WHERE major = 'CS';	Filters rows by predicate
Projection	SELECT DISTINCT major FROM student;	Selects columns, removes dups
Join	SELECT ... FROM A JOIN B ON A.x = B.y;	Combines rows on matching columns
Union	SELECT ... FROM A UNION SELECT ... FROM B;	Combines rows from both tables
Intersection	SELECT ... FROM A INTERSECT SELECT ... FROM B;	Rows common to both tables
Cartesian	SELECT * FROM A, B;	All combinations of rows

6.4 SQL: Structured Query Language

- **Schema definition:**

```
CREATE TABLE student (
    id INT PRIMARY KEY,
    family_name VARCHAR(255),
    given_names VARCHAR(255),
    major VARCHAR(64)
);

CREATE TABLE class (
    student_id INT,
    letter_grade VARCHAR(2),
    FOREIGN KEY (student_id) REFERENCES student(id)
);
```

- **Insert:**

```
INSERT INTO student VALUES (1, 'Abdul-Jabbar', 'Kareem', 'Sociology');
INSERT INTO student VALUES (2, 'Coppola', 'Francis', 'Film Studies');
```

- **Query:**

```
SELECT id, major FROM student ORDER BY major ASC;
```

---

7. Alternative Database Models

7.1 Entity-Relationship (ER) Model

- Focuses on entities (objects), attributes (properties), and relationships (references).
- Can be implemented atop relational DBs or from scratch.

7.2 Object-Oriented Databases

- Data as objects with attributes, methods, and class hierarchies.
- Integrates code (methods) with data.

7.3 NoSQL

- Rejects rigid table format; allows flexible, dynamic schemas.
- **Key-value stores:** Like hash tables, but persistent.
- **Wide-column:** Columns can differ between rows.
- **Document stores:** Store arbitrary documents (JSON, XML, etc.).
- **Graph databases:** Entities as nodes, relationships as edges; supports cycles and arbitrary graphs.
- **Trade-off:** More flexibility, but may lack strong guarantees and efficient search.

Table: NoSQL Database Types

Type	Example Systems	Data Model	Strengths	Weaknesses
Key-Value	Redis, DynamoDB	Hash table	Fast, simple	No structure, limited query
Wide-Column	Cassandra, HBase	Table, flexible	Scalable, flexible	Complex, eventual consistency
Document	MongoDB, CouchDB	JSON/BSON/XML	Flexible, nested data	Weaker consistency, joins hard
Graph	Neo4j, ArangoDB	Nodes/edges	Rich relationships	Niche, complex queries

Example: Document Store (MongoDB)

```
{
  "_id": 1,
  "name": "Alice",
  "courses": ["CS101", "CS102"]
}
```

---

8. Database Reliability: ACID Properties

Property	Description
Atomicity	Transactions are all-or-nothing
Consistency	DB always satisfies constraints/invariants
Isolation	Concurrent operations don't interfere (as if run sequentially)
Durability	Changes persist across failures

- **Example:**

  ```
  BEGIN TRANSACTION;
  INSERT INTO student VALUES (...);
  UPDATE classes SET grade='A' WHERE ...;
  COMMIT;
  ```

- Without atomicity, partial changes may persist if system crashes.
- Without isolation, concurrent users may see inconsistent data.

Example: Transaction Failure (Atomicity Violation)

Suppose a bank transfer:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
-- System crashes here!
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

If atomicity is not enforced, $100 is lost! With atomicity, neither change is visible after crash.

ASCII Diagram: Transaction Timeline

```
[Start Transaction]
      |
      v
[Step 1: Deduct $100 from A]
      |
      v
[System Crash]
      |
      v
[Recovery: Rollback]
      |
      v
[No changes applied]
```

Example: Isolation Violation

If two users transfer from the same account at the same time, without isolation, both may see the same initial balance and overdraw the account. With isolation, transactions appear to run one after another.

8.1 Compromising ACID

- Some cloud/NoSQL DBs sacrifice consistency or isolation for scalability (eventual consistency).
- Developers must balance speed, complexity, and correctness.

8.2 Practical Tools

- **Firebase:** Popular NoSQL backend; not fully ACID-compliant, emphasizes client sync.
- **Oracle, MySQL, PostgreSQL:** Relational, support full ACID.

---

9. Summary

- Software build automation evolved from manual steps to sophisticated tools (`make`, `autoconf`, `automake`).
- Build systems must handle partial builds, parallelism, and portability.
- Databases provide structured, reliable, and efficient data management.
- Relational model, SQL, and ACID properties are foundational.
- Alternative models (ER, object, NoSQL, graph, document) offer flexibility for modern needs.
- Understanding trade-offs is key to building robust, scalable systems.

Lecture 19: Software Security

1. Introduction: Why Security Matters

- Security must be considered from the very beginning of software design—not as an afterthought.
- Integrate security into initial specs, requirements, and all stages of development.
- Clients/customers assume developers are building secure systems, even if they don't specify requirements.
- Ignoring security leads to wasted time, insecure architectures, and potentially catastrophic vulnerabilities.
- Security is a major field in practical computing (jobs, internships, real-world impact), second only to AI in scope.
- Even if you don't specialize in security, you must understand the basics to avoid critical mistakes and communicate with specialists.
- **Regulatory/Compliance:** Many industries require security by law (e.g., GDPR for privacy in Europe, HIPAA for healthcare in the US).
- **Real-World Example:**
  - In 2017, the Equifax breach exposed sensitive data of 147 million people due to an unpatched vulnerability, leading to billions in damages and loss of trust.

---

2. Step Zero: Security Mindset

Before you write code, develop two models:

2.1 Security Model

- What are you defending? ("Crown jewels"/assets)
- Model the application and its valuable data/resources.
- **CIA Triad:**
  - **Confidentiality (Privacy):** Prevent unauthorized data access/leakage.
  - **Integrity:** Prevent unauthorized modification/tampering.
  - **Availability (Service):** Ensure systems are usable and accessible.

```
+-------------------+
|   Application     |
|-------------------|
| Confidential Data |
+-------------------+
      |   |   |
      v   v   v
 C (no leaks)  I (no tampering)  A (always up)
```

Example Security Model: Online Banking

- Assets: Account balances, transaction history, personal info
- Confidentiality: Only account owner and authorized staff can view balances
- Integrity: Only valid transactions can modify balances
- Availability: Users must be able to access accounts 24/7

2.2 Threat Model

- Who might attack the system, and how?
- Prioritize defense strategies based on realistic threats.
- Models are never perfect—improve them continuously.

Threat Modeling Frameworks

Framework	Focus	Example Use
STRIDE	Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege	Web apps, APIs
DREAD	Damage, Reproducibility, Exploitability, Affected Users, Discoverability	Risk ranking
PASTA	Process for Attack Simulation and Threat Analysis	Enterprise systems

Checklist for Security Modeling

Category	Purpose	Example
Assets	What needs protection	Database, config files, credentials
Vulnerabilities	Weaknesses/channels for attack	Open ports, input flaws, misconfig

Category	Purpose	Example
Threats	Potential attackers/scenarios	Ex-employees, bots, phishing, insiders

3. Key Security Functions

3.1 Authentication

- Verifies user identity.
- Examples: Passwords, two-factor authentication (2FA), USB keys, biometrics (retinal scan, fingerprint).
- Multi-factor authentication (MFA):
  - Something you know (password)
  - Something you have (device, app, USB key)
  - Something you are (biometric)
- **Password Best Practices:**
  - Use long, random passwords (consider passphrases)
  - Never reuse passwords across sites
  - Use a password manager
  - Avoid common pitfalls: security questions, SMS-based 2FA (can be intercepted)

Method Type	Description	Example
Something you know	Knowledge-based	Password, PIN
Something you have	Possession-based	USB key, Duo app, TOTP
Something you are	Biometric	Fingerprint, retina

3.2 Authorization

- Dictates what authenticated users can do.
- Example: Instructor can modify grades in their own classes, not others.
- Uses Access Control Lists (ACLs):

User	Read	Write	Execute
Eggert	Yes	Yes	Yes
Milstein	Yes	No	No
Frank	No	No	No

- **Role-Based Access Control (RBAC):**
  - Users are assigned roles (e.g., admin, user, guest), and roles have permissions.
  - Easier to manage than per-user ACLs in large systems.

Role	Permissions
Admin	Read, Write, Execute
User	Read, Write
Guest	Read

- **Principle of Least Privilege:**
  - Users/processes should have the minimum access necessary to perform their tasks.
- **Fail-Safe Defaults:**
  - Deny access by default; only allow what is explicitly permitted.

3.3 Integrity Mechanisms

- Detect and recover from unauthorized changes.
- Examples: Checksums, secure backups, digital signatures.

Mechanism	Description
Checksums	Detect tampering by comparing stored values
Backups	Restore data after compromise/loss
Digital Sig.	Verify authenticity and integrity

3.4 Auditing

- Logs user actions to detect/recover from intrusions.
- Logs must be secure, complete, and regularly reviewed.
- **What to Log:**
  - Logins (success/failure), privilege changes, data access, configuration changes, errors.
  - Protect logs from tampering (write-once, append-only storage).

3.5 Supporting Principles

Principle	Description
Correctness	Security must not break core functionality
Efficiency	Security must not overly degrade performance
Usability	Security should not make the system unusable

---

4. Threat Modeling and Classification

- Classify threats to prioritize and determine effective defenses.

4.1 Network Attacks

Threat	Description	Example
Phishing	Tricking users to click links or enter credentials	Fake bank email asks for password
Drive-by Downloads	Malware triggered by visiting malicious pages	Malicious ad installs spyware
Denial-of-Service	Overloading servers to make them unavailable	Botnet floods login page
Buffer Overruns	Overflowing memory buffers to hijack control flow	See code below
Cross-Site Scripting	Malicious JavaScript in user's browser (e.g., bank theft)	Attacker injects <script> tag
Prototype Pollution	Attacking object prototypes in JavaScript	Overwriting **proto** in JS

Buffer Overrun Example (C):

```
char buf[8];
gets(buf); // No bounds checking! Attacker can overwrite memory.
```

4.2 Device Attacks

Attack	Description	Example
Bad USB	USB device boots/installs malware, hijacks boot process	Malicious USB stick
Insider Attack	Authorized users misuse privileges	Employee steals data
Supply Chain	Compromise occurs in hardware/software before delivery	Pre-installed malware on device

4.3 Social Engineering

- Convincing users to reveal info or grant access (e.g., fake repairman, lost ID).
- Hard to defend; requires robust integrity and auditing.
- **Example:** Attacker calls pretending to be IT and asks for your password.

4.4 Physical Security

- Physical access can defeat most software security.
- Examples: Stolen laptops, unlocked server rooms, dumpster diving for sensitive documents.

---

5. OWASP Top 10 Application Security Risks (2021)

#	Vulnerability	Description/Examples
1	Broken Access Control	URL/JWT/cookie manipulation, insecure direct object references
2	Cryptographic Failures	HTTP over HTTPS, weak crypto, improper certificate validation
3	Injection Attacks	SQL/NoSQL/command injection via untrusted input

#	Vulnerability	Description/Examples
4	Insecure Design	No threat modeling, poor design practices
5	Security Misconfiguration	Default passwords, open ports, unnecessary services
6	Vulnerable/Outdated Components	Using libraries/OS with known exploits
7	Identification/Authentication Failures	Weak passwords, no rate limiting, brute force attacks
8	Software/Data Integrity Failures	Poor update mechanisms, unverified sources, supply chain attacks
9	Security Logging/Monitoring Failures	Lack of logging, logs not reviewed, filled logs
10	Server-Side Request Forgery (SSRF)	Tricking server to access internal/private network addresses

Examples:

- **1. Broken Access Control:** User changes their user ID in the URL to access another user's data.
- **2. Cryptographic Failures:** Site uses HTTP instead of HTTPS, exposing passwords in transit.
- **3. Injection Attacks:** Attacker enters `'; DROP TABLE users;--` in a login form.
- **4. Insecure Design:** No input validation, no threat modeling, no secure defaults.
- **5. Security Misconfiguration:** Admin interface left open to the internet with default password.
- **6. Vulnerable/Outdated Components:** Using an old version of OpenSSL with known bugs.
- **7. Identification/Authentication Failures:** No account lockout after repeated failed logins.
- **8. Software/Data Integrity Failures:** Application updates from untrusted sources.
- **9. Security Logging/Monitoring Failures:** No alert when admin logs in from a new country.
- **10. SSRF:** Attacker tricks server into fetching internal metadata from AWS.

Clarification:

- **Broken Access Control** is about users being able to access things they shouldn't (authorization failure).
- **Identification/Authentication Failures** are about not being able to reliably tell who a user is (authentication failure).

---

6. Security Testing: Philosophy and Strategies

6.1 Testing Philosophy

Traditional Testing	Security Testing
Inputs: typical users	Inputs: malicious attackers
Failures: random	Failures: deliberate, targeted
Bugs: accidental	Bugs: systematically exploited

6.2 Strategies

- **Static Analysis:** Analyze code without running it (find buffer overflows, races, etc.).
- **Dynamic Analysis:** Analyze code while running (finds runtime issues, e.g., memory leaks, race conditions).
- **Penetration Testing:** Hire trusted "black hats" to simulate real-world attacks.
- **Fuzz Testing:** Automatically generate random/malformed inputs to find crashes and vulnerabilities.

Testing Type	Description	Example Tool
Static Analysis	Examines code without running it	Coverity
Dynamic Analysis	Examines code during execution	Valgrind
Penetration Test	Simulated attack by security experts	Metasploit
Fuzz Testing	Randomized input to find bugs	AFL, libFuzzer

6.3 Side-Channel and Timing Attacks

- Attackers infer internal state/data by measuring timing (e.g., cache timing, Spectre, Meltdown).
- Apple restricts high-res timers to inhibit these attacks; Linux allows nanosecond timing (riskier).
- **Example: Timing Attack on Password Check**

```
// Vulnerable password check
for (int i = 0; i < N; i++) {
    if (input[i] != secret[i]) return 0;
}
```

```
return 1;
// Leaks how many bytes are correct via timing
```

- **Countermeasures:**
  - Use constant-time comparison functions
  - Limit timing information available to untrusted code

6.4 Subtle Abstraction Violations

- Example: Partial string comparison leaks password byte-by-byte via timing.
- Advanced: Manipulate memory layout, exploit page boundaries, cache access patterns.

```
[User Input] ---> [Password Check]
                |
                +---> [Timing Difference] ---> [Attacker infers secret]
```

---

7. Trusting Trust: Ken Thompson's Attack

- Described in the Turing Award lecture "Reflections on Trusting Trust."
- Modify the C compiler to insert a backdoor when compiling login.c:

```
if (strcmp(name, "ken") == 0) return true;
```

- Then, modify the compiler to insert this backdoor into any future compiler it compiles.
- Result: Even if you inspect the source code for login.c and cc.c, the executables will still regenerate the backdoor.

```
[Source: login.c] --(buggy cc)--> [login (backdoor)]
[Source: cc.c]    --(buggy cc)--> [cc (backdoor)]
[Source: new cc.c] --(backdoored cc)--> [new cc (backdoor)]
```

- **Modern Implications:**
  - Supply chain attacks in open source: malicious code injected into dependencies, compilers, or build tools.
  - Example: SolarWinds attack (2020) where attackers compromised the build system to insert backdoors.
- **Defense:**
  - Define a **Trusted Computing Base (TCB):** Minimal set of components (compiler, OS, hardware) that must be trusted.
  - Software reproducibility and rigorous review help, but trust must start somewhere.
  - Use reproducible builds and independent verification.

---

8. Summary and Takeaways

- Security must be integrated from the start—never "added later."
- Develop both a security model (what to protect) and a threat model (who/what to defend against).
- The CIA Triad (Confidentiality, Integrity, Availability) is foundational.
- Key functions: authentication, authorization, integrity, auditing, correctness, efficiency.
- Threats include network, device, social engineering, and insider attacks.
- OWASP Top 10 highlights common vulnerabilities—know and avoid them.
- Security testing is adversarial: expect intelligent, targeted attacks.
- Advanced attacks (timing, side-channel, supply chain, trusting trust) require deep awareness.
- Always think like a defender—and sometimes like an attacker—to build robust, secure software.

Checklist for Secure Software Development:

- [ ] Define assets and security requirements
- [ ] Build a security and threat model
- [ ] Apply the principle of least privilege
- [ ] Use secure defaults (fail-safe)
- [ ] Validate and sanitize all inputs
- [ ] Use strong authentication and authorization
- [ ] Encrypt sensitive data in transit and at rest
- [ ] Keep software and dependencies up to date

- ☐ Log and monitor security-relevant events
- ☐ Regularly test for vulnerabilities (static, dynamic, fuzz, pen testing)
- ☐ Plan for incident response and recovery