Linux Mastery for C++ Developers

A Comprehensive Roadmap with Commands, Tools, and Real-World Examples

by **CompilerSutra**

Version 3.1 | November 2, 2025

"The Linux philosophy is 'Laugh in the face of danger'. Oops. Wrong One. 'Do it yourself'. Yes, that's it."
— *Linus Torvalds*

Contents

1. Introduction

1.1. Why Linux Matters for C++ Developers

Linux is the standard development and deployment environment for many C++ projects, especially in systems programming, networking, embedded systems, and high-performance computing. Being comfortable with Linux significantly improves debugging capabilities, automation workflows, and overall productivity.

> 🎓 **Best Practice:**
> Master Linux to become a better C++ developer. Most production C++ code runs on Linux servers, and understanding the environment helps in debugging complex issues.

1.2. How to Use This Roadmap

This document is structured progressively from beginner to expert level. Each section contains practical commands, real-world examples, and industry best practices. Use the appendix as a quick reference cheat-sheet and add your personal notes where needed. Read, practice, and modify the examples — the real learning happens in the terminal.

1.3. Real-world Use Cases

- **Production Debugging**: Diagnosing crashes using gdb and core dumps
- **Performance Optimization**: Profiling code with `perf` and `valgrind`
- **Build Systems**: Managing complex projects with CMake
- **Interview Preparation**: Common Linux questions in C++ interviews
- **DevOps Integration**: CI/CD pipelines and containerization

🚀 **Best Practice:** Maintain a daily practice log of commands. Add frequently used commands to your shell configuration files as aliases for increased productivity.

2. Linux Basics (Beginner Level)

2.1. Filesystem Hierarchy Standard

Understanding the Linux filesystem structure is crucial:

```
/ # Root directory
/bin # Essential user binaries
/etc # Configuration files
/home # User home directories
/usr # User programs and data
/var # Variable data (logs, etc.)
/tmp # Temporary files
/opt # Optional software packages
/dev # Device files
/proc # Kernel and process info (virtual FS)
```

2.2. File Operations Mastery

```
# Create nested directories
mkdir -p project/{src,include,build,test}

# Copy with progress and preserve attributes
rsync -av --progress source/ destination/

# Find and delete specific files
find . -name "*.o" -type f -delete

# Create symbolic links
ln -s /path/to/original link_name

# Archive and compress
tar -czvf project.tar.gz project/
tar -xzf project.tar.gz -C /tmp
```

> 🎓 **Best Practice:**
> Use `rsync` instead of `cp` for large directories as it's more reliable and shows progress. Use `-dry-run` first to verify operations.

2.3. Advanced Text Editing

```
# View multiple files
less file1.txt file2.txt

# Follow growing file (like logs)
tail -f /var/log/syslog

# View with line numbers
cat -n source.cpp

# Real-time log monitoring
tail -f application.log | ccze # if ccze is installed for coloring
tail -f application.log | grep -i error
```

2.4. Permission Deep Dive

```
# Numeric permissions explained
chmod 755 script.sh # rwxr-xr-x
chmod 644 file.txt # rw-r--r--
chmod 600 secret.key # rw-------

# Change ownership recursively
chown -R user:group project/

# Set sticky bit on directories
chmod +t shared_folder/

# Setuid and setgid (use with caution)
chmod u+s /usr/bin/somebinary
chmod g+s somedir/
```

Exercise: Create a project structure with proper permissions and practice file operations for 30 minutes daily. Use `getfacl` / `setfacl` to experiment with ACLs.

3. Intermediate Linux Commands

3.1. Advanced File Searching

```
# Find C++ files modified in last 7 days
find . -name "*.cpp" -mtime -7

# Find files larger than 100MB
find / -type f -size +100M 2>/dev/null

# Find and execute commands on results
find . -name "*.tmp" -exec rm {} \;

# Search in source code excluding build directories
grep -R --exclude-dir=build "TODO" .

# Use ripgrep (rg)  faster, respects .gitignore
rg "TODO" src/ --hidden
```

3.2. System Performance Monitoring

```
# Real-time system monitoring
htop
glances # Comprehensive system monitor

# I/O statistics
iostat -x 1

# Memory usage details
cat /proc/meminfo
free -h

# Process tree view
pstree -p

# System resource summary
vmstat 1 5
```

3.3. Network Diagnostics

```
# Detailed network interface info
ip addr show
ip route show

# Continuous ping with timestamp
ping -D 8.8.8.8

# Check open ports and listening processes
ss -tulpn
netstat -tulpn

# Download with progress
wget --progress=bar http://example.com/file
curl -O http://example.com/file

```

```
# HTTP quick checks
curl -I https://example.com
curl -sS https://example.com | head -n 20

# DNS troubleshooting
dig example.com +short
nslookup example.com
```

3.4. Package Management Deep Dive

```
# Ubuntu/Debian advanced usage
sudo apt update && sudo apt upgrade
sudo apt install build-essential cmake git gdb valgrind

sudo apt search "c++ library"
sudo apt show package-name
sudo apt remove --purge package-name

# Snap packages
sudo snap install clion --classic

# Flatpak applications
flatpak install flathub com.jetbrains.CLion
```

🚀 **Best Practice:** Regularly update your system and development tools. Use version pinning for critical production dependencies. Use containers to isolate toolchains when necessary.

4. Advanced System Administration

4.1. Systemd Service Management

```
# Create user service
mkdir -p ~/.config/systemd/user/
# Edit ~/.config/systemd/user/myservice.service with content like:
# [Unit]
# Description=My user service
# [Service]
# ExecStart=/home/user/bin/run_task.sh
# [Install]
# WantedBy=default.target

# Reload and enable
systemctl --user daemon-reload
systemctl --user enable myservice
systemctl --user start myservice

# View service logs
journalctl --user-unit=myservice -f
```

4.2. Kernel and System Optimization

```
# View kernel parameters
sysctl -a | grep tcp

# Temporary modification
sudo sysctl -w net.ipv4.tcp_fin_timeout=30

# Permanent changes in /etc/sysctl.conf
echo "net.ipv4.tcp_fin_timeout=30" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p

# CPU governor
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
sudo cpupower frequency-set -g performance
```

4.3. Advanced Security Features

```
# SELinux context
ls -Z /usr/bin/ssh
ps -Z

# AppArmor status
sudo aa-status

# Firewall management (UFW)
sudo ufw enable
sudo ufw allow 22/tcp
sudo ufw allow 8080/tcp
sudo ufw status verbose
```

4.4. Containerization with Podman/Docker

```
# Development container
docker run -it --rm -v $(pwd):/workspace \
  -w /workspace gcc:latest bash

# Multi-stage build for C++
# Dockerfile example:
FROM gcc:latest as builder
WORKDIR /app
COPY . .
RUN g++ -O2 -std=c++17 -o myapp main.cpp

FROM ubuntu:latest
COPY --from=builder /app/myapp /usr/local/bin/myapp
CMD ["/usr/local/bin/myapp"]
```

Exercise: Create a systemd unit that runs a small HTTP server and logs to journalctl. Test enabling, starting, stopping, and viewing logs.

5. Shell Scripting Excellence

5.1. Advanced Bash Features

```
#!/usr/bin/env bash
set -euo pipefail # Exit on error, undefined vars, pipe failures
IFS=$'\n\t' # Better word splitting

# Color output
RED='\033[0;31m'
GREEN='\033[0;32m'
NC='\033[0m' # No Color

log_success() { echo -e "${GREEN}[SUCCESS]${NC} $*"; }
log_error() { echo -e "${RED}[ERROR]${NC} $*" >&2; }

# Function with parameters
build_project() {
    local build_type=${1:-Release}
    log_success "Building in $build_type mode"
    cmake -B build -DCMAKE_BUILD_TYPE=$build_type
    cmake --build build -j$(nproc)
}
```

5.2. Error Handling and Debugging

```
# Advanced error handling
handle_error() {
    local exit_code=$?
    log_error "Command failed with exit code $exit_code"
    # Add diagnostics: print last 20 lines of relevant log
    journalctl -n 20 --no-pager
    exit $exit_code
}

trap handle_error ERR

# Debug mode
if [[ "${DEBUG:-false}" == "true" ]]; then
    set -x
fi
```

5.3. Configuration Management

```
# Sample .bashrc additions for C++ developers
export CC=/usr/bin/gcc
export CXX=/usr/bin/g++

# Development tools
export CMAKE_GENERATOR="Ninja"
export CTEST_OUTPUT_ON_FAILURE=1

# Python virtual environment
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
```

```
12
13 # Custom functions
14 ccache_enable() { export PATH="/usr/lib/ccache:$PATH"; }
15 docker_clean() {
16     docker system prune -af
17     docker volume prune -f
18 }
```

Best Practice: Place frequently used functions and aliases in a separate `/.bashrc.d/dev.sh` and source them from `/.bashrc` to keep files modular.

6. Text Processing and Data Manipulation

6.1. grep, rg, awk, sed

```
# grep examples
grep -RIn "TODO" .
grep -n --line-number "pattern" file

# ripgrep (rg) - fast, respects .gitignore
rg "TODO" --hidden

# awk examples
awk '{print $1, $3}' data.txt
awk -F: '$3 >= 1000 {print $1}' /etc/passwd

# sed examples
sed -n '1,100p' file.txt
sed -i.bak 's/old/new/g' file.txt
```

6.2. cut, sort, uniq, tr, wc

```
# cut
cut -d',' -f2 data.csv

# sort & uniq
sort names.txt | uniq -c | sort -nr

# tr and wc
tr -d '\r' < file.txt > unix_file.txt
wc -l *.cpp
```

6.3. Practical assembly / log parsing

```
# Example: find lines with register writes in assembly and list unique registers
rg "r[0-9]+" asm.log | sed -E 's/.*(r[0-9]+).*/\1/' | sort | uniq -c | sort -nr

# More advanced: extract destination registers from patterns like "add.f r40.x, r1.z;"
rg -o "r[0-9]+\.[xyzw]" asm.log | sed 's/\..$//' | sort | uniq -c
```

🚀 **Project:** Write a small bash + awk pipeline to read a compiler IR dump and summarize instruction counts per opcode, output CSV for visualization.

7. Developer Productivity Tools

7.1. tmux Essentials

```
tmux new -s dev # start session named dev
tmux ls # list sessions
tmux attach -t dev # attach to session
tmux kill-session -t dev

# Split panes: prefix (Ctrl-b) then %
# Create window: prefix c
# Switch windows: prefix n / p or prefix <number>
```

7.2. fzf, ripgrep, bat

```
# fzf interactive file finder
rg --files | fzf --preview 'bat --style=numbers --color=always {}'

# Open file under cursor in vim
vim $(rg --files | fzf)

# bat for nice cat with syntax highlight
bat src/main.cpp
```

7.3. Editor setups: clangd, LSP, ccls

```
# clangd uses compile_commands.json generated by CMake:
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
# In editor, point clangd to build/compile_commands.json
```

🎓 **Best Practice:**
Use `ccache` + `Ninja` to speed development builds and combine with `clangd` for fast incremental indexing.

8. Build Systems & Source Control

8.1. CMake advanced patterns

```
# Modern CMake example
cmake_minimum_required(VERSION 3.20)
project(MyProject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Boost 1.70 REQUIRED COMPONENTS filesystem system)

add_executable(myapp main.cpp)
target_compile_options(myapp PRIVATE -Wall -Wextra -Wpedantic)
target_link_libraries(myapp PRIVATE Boost::filesystem Boost::system)
```

8.2. Toolchain files and cross-compiling

```
# toolchain-arm.cmake (snippet)
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_C_COMPILER arm-linux-gnueabihf-gcc)
set(CMAKE_CXX_COMPILER arm-linux-gnueabihf-g++)
```

8.3. Makefile tips and parallel builds

```
# Example wrapper Makefile for CMake
.PHONY: all clean format

all:
	cmake -S . -B build -G Ninja
	cmake --build build -- -j$(nproc)

clean:
	rm -rf build

format:
	clang-format -i $(shell rg --files-with-matches --glob '!build' -g '*.cpp' -g
	    '*.hpp' -S)
```

8.4. Git best practices

- Commit small, logical units. One change = one commit.
- Use descriptive commit messages (imperative tense).
- Rebase local feature branches before opening PRs to keep history linear.
- Protect main branch with required status checks in CI.

9. CI/CD and Automation

9.1. GitHub Actions: Build, Test, Sanitize

```yaml
name: C++ CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        build_type: [Debug, Release]
    steps:
      - uses: actions/checkout@v3
      - name: Install deps
        run: sudo apt-get update && sudo apt-get install -y build-essential cmake ninja
            -build
      - name: Configure
        run: cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=${{ matrix.build_type }}
      - name: Build
        run: cmake --build build -- -j$(nproc)
      - name: Run tests
        run: ctest --test-dir build --output-on-failure
```

9.2. Automated packaging with CPack

```
# CPack can produce .deb, .rpm, TGZ packages from CMake
include(InstallRequiredSystemLibraries)
set(CPACK_GENERATOR "DEB;TGZ")
set(CPACK_PACKAGE_NAME "myapp")
set(CPACK_PACKAGE_VERSION "1.0.0")
include(CPack)
```

🚀 **Project:** Add a GitHub Action that runs clang-tidy and sanitizers on PRs. Fail the build if severity exceeds policy.

10. Debugging & Profiling

10.1. GDB essentials and advanced usage

```
# compile with debug info
g++ -g -O0 main.cpp -o main

# start gdb
gdb ./main
(gdb) break main
(gdb) run
(gdb) backtrace full
(gdb) info locals
(gdb) print variable_name

# TUI mode
gdb -tui ./main
# Reverse debugging (when enabled by gdb)
(gdb) record
(gdb) reverse-step
```

10.2. Core dump debugging

```
# enable core dumps
ulimit -c unlimited
# run program -> on crash core.<pid>
gdb ./myprog core.<pid>
(gdb) bt full
(gdb) info registers
```

10.3. Valgrind and Sanitizers

```
# Valgrind memcheck
valgrind --leak-check=full ./app

# Massif for heap profiling
valgrind --tool=massif ./app
ms_print massif.out.####

# AddressSanitizer
g++ -fsanitize=address -fno-omit-frame-pointer -g main.cpp -o main

# UndefinedBehaviorSanitizer
g++ -fsanitize=undefined -g main.cpp -o main
```

10.4. perf: CPU profiling

```
# lightweight stat
perf stat ./app

# record call graph
perf record -g ./app
perf report -g graph
```

```
# annotate hot functions
perf annotate --stdio
```

> **Tip:**
> Use ‘perf‘ for production-like sampling profiling and ‘valgrind‘/sanitizers for debug builds. Sampling profiler (perf) avoids huge overhead of instrumentation.

11. System Call and Library Tracing

11.1. strace and ltrace

```
# trace system calls
strace -f -o strace.out ./app

# summary of syscall counts and times
strace -c ./app

# library calls (function-level calls)
ltrace ./app
```

11.2. ldd, ltrace, and dependency investigation

```
ldd ./myprog # list dynamic dependencies
readelf -d myprog | less # dynamic section info
objdump -p myprog | grep RPATH -A 5
```

Exercise: Reproduce a missing symbol error by modifying $\mathrm{LD}_{L}IBRARY_{P}ATHandobservethegdb/lddiagnostics.Documentthestepsandfix.$

12. System Binary Inspection

12.1. ELF, symbols, and objdump

```
readelf -h myprog # ELF header
readelf -s myprog | less # symbol table
nm -C myprog # demangled symbols
objdump -d myprog | less # disassembly
```

12.2. Static vs Dynamic linking

- Static linking produces a larger binary but is self-contained.
- Dynamic linking uses shared objects (.so) and reduces binary size, but requires the correct runtime environment.

12.3. Strip and size

```
# Strip debug/symbols to reduce size
strip --strip-all myprog
# View binary size
size myprog
file myprog
```

13. Containers, Sandboxing, and Reproducible Builds

13.1. Developer containers

```
# build a dev image
docker build -t myapp-dev -f Dockerfile.dev .

# run interactive
docker run --rm -it -v "$(pwd):/workspace" -w /workspace myapp-dev bash
```

13.2. Reproducible builds and deterministic timestamps

- Use fixed base images and pinned versions.
- Avoid embedding timestamps in artifacts, or set SOURCE$_{D}ATE_{E}POCH$.
- Validate by comparing checksums between builds.

13.3. Podman vs Docker

- Podman is daemonless and rootless-capable — suitable for CI on some infra.
- Most Docker files work with Podman after minor tweaks.

14. Filesystems, Storage and Snapshots

14.1. Common filesystems

- ext4 — stable default for many distros.
- xfs — high-performance, good for large files.
- btrfs — snapshots and subvolumes.
- zfs — robust data integrity (license/caveats on Linux).

14.2. LVM basics

```
# create physical volume, volume group, logical volume
pvcreate /dev/sdb1
vgcreate vgdata /dev/sdb1
lvcreate -n lvdata -L 100G vgdata
mkfs.xfs /dev/vgdata/lvdata
mount /dev/vgdata/lvdata /data
```

14.3. Snapshots

```
# LVM snapshot
lvcreate --size 1G --snapshot --name snap1 /dev/vgdata/lvdata
# Btrfs snapshot
btrfs subvolume snapshot /mnt/data /mnt/data_snapshot
```

🚀 **Best Practice:** Test snapshot restore procedures and ensure backups are automated. Snapshots are not backups — combine with external backups.

15. Packaging and Distribution

15.1. Creating .deb packages with CPack

```
# In CMakeLists.txt
include(InstallRequiredSystemLibraries)
set(CPACK_GENERATOR "DEB")
set(CPACK_PACKAGE_NAME "myapp")
set(CPACK_PACKAGE_VERSION "1.0.0")
include(CPack)

# After build
cpack -G DEB -C Release
```

15.2. RPM basics

```
# build with rpmbuild
# ensure proper spec file and buildroot
rpmbuild -ba mypackage.spec
```

15.3. Distribution tips

- Provide clear dependency metadata.
- Use CI to produce packages for multiple distros.
- Sign packages and repositories for secure delivery.

16. Embedded Linux and Cross-Compilation

16.1. Toolchain and sysroot

```
export CC=arm-linux-gnueabihf-gcc
export CXX=arm-linux-gnueabihf-g++
export SYSROOT=/opt/arm/sysroot

cmake -S . -B build-arm \
  -DCMAKE_SYSTEM_NAME=Linux \
  -DCMAKE_SYSROOT=$SYSROOT \
  -DCMAKE_C_COMPILER=$CC \
  -DCMAKE_CXX_COMPILER=$CXX
```

16.2. Remote debugging

```
# On target:
gdbserver :2345 ./app

# On host:
arm-linux-gnueabihf-gdb ./app
(gdb) target remote 192.168.0.10:2345
```

🚀 **Project:** Set up a simple cross-compilation workflow and debug a "Hello world" remotely via gdbserver. Document latency and steps.

17. Real-time Linux and Kernel Hints

17.1. PREEMPT_RT basics

```
# check kernel info
uname -a
# check if PREEMPT_RT is enabled (if config exposed)
zcat /proc/config.gz | rg PREEMPT

# test latency with cyclictest
sudo apt-get install rt-tests
sudo cyclictest -p 80 -n -i 100 -l 1000
```

17.2. Kernel build quickstart

```
git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
cd linux-stable
make defconfig
# optionally run menuconfig to tweak options
make -j$(nproc)
# install modules (careful on production systems)
\end{minted}

\begin{WarningBox}
Compiling and installing kernels on production machines is risky. Use VM or test
    hardware. Keep recovery options (rescue kernel, bootable USB).
\end{WarningBox}

\clearpage

\section{Networking Deep Dive}

\subsection{Socket basics (C/C++)}
\begin{lstlisting}
// Basic TCP server skeleton (pseudocode)
int sock = socket(AF_INET, SOCK_STREAM, 0);
bind(sock, ...);
listen(sock, 128);
int client = accept(sock, ...);
read(client, buffer, len);
```

17.3. iptables / nftables

```
# List current rules
sudo iptables -L -n -v
# nftables example
sudo nft list ruleset
```

17.4. Secure access and SSH

```
ssh-keygen -t ed25519 -C "dev@company"
ssh-copy-id user@host
# Agent
```

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

🚀 **Exercise:** Create a small client/server pair, run server in tmux, use ss to verify listener, and simulate network failure with iptables to test reconnection logic.

18. Monitoring, Logging and Observability

18.1. journalctl and logs

```
# view unit logs
journalctl -u myservice -f

# logs since a time
journalctl --since "2025-01-01" --no-pager

# use rate-limit
journalctl --disk-usage
```

18.2. Metrics stack basics

- $node_exporter (Prometheus) for host metrics$.
- cadvisor for container metrics.
- Prometheus + Grafana for alerting and dashboards.

🚀 **Project:** Deploy a minimal Prometheus + Grafana stack using Docker Compose and create a dashboard for CPU, memory, and process counts.

19. Security Practices

19.1. Hardening tips

- Keep packages updated and apply security patches.
- Use minimal images for containers (distroless when possible).
- Run services with least privilege and dedicated users.
- Use SELinux/AppArmor and learn to debug denials.
- Use fail2ban to block brute-force attempts.

19.2. Secrets management

- Never store secrets in VCS.
- Use environment variables from secret stores or mounted volumes.
- For CI, use encrypted secrets supported by platform (GitHub Actions secrets).

20. Interview Prep and Common Questions

20.1. Key topics to master

- Process lifecycle, signals, wait/waitpid.
- Memory layout (stack, heap, BSS, data, text).
- ELF structure and dynamic linking.
- Debugging workflows and core dump analysis.
- Performance analysis: hotspots, cache behavior.
- Build systems and cross-compilation basics.

20.2. Sample questions

1. **What does `set -euo pipefail` do?**
   Answer: `-e` exits on non-zero status, `-u` treats unset vars as error, `-o pipefail` makes whole pipeline fail if any element fails.
2. **Explain `fork()` vs `exec()`.**
   Answer: `fork()` duplicates current process; `exec()` replaces the process image with a new program (often called in child after fork).
3. **How do you debug a segmentation fault in production?**
   Answer: Reproduce with debug build if possible, enable core dumps, use gdb on core, analyze backtrace, inspect memory, check for UB with sanitizers.

21. Mini Projects and Exercises

21.1. Project 1 — C++ Library with CI

- Build a small library (header + implementation).
- Add unit tests with GoogleTest.
- Add CMake, CPack packaging, and GitHub Actions CI for build + tests.
- Add clang-tidy checks and sanitizer runs on PRs.

21.2. Project 2 — Log Analyzer for Compiler Dumps

- Parse compiler/assembly dumps.
- Identify registers written but not read (data flow hint).
- Output CSV with file,line,register,status.
- Visualize using simple bar charts (e.g., Python matplotlib).

21.3. Project 3 — Telemetry Daemon

- Small daemon that collects CPU, memory, disk and exposes HTTP metrics for Prometheus.
- Run as a systemd service and add health checks.
- Containerize for deployment.

A. Appendix A: Quick Reference Cheat Sheet

Command	Purpose
`ls -la`	List files with details and hidden files
`find . -name "*.cpp"`	Find C++ sources
`rg "TODO"`	Fast code search (ripgrep)
`tmux new -s dev`	Create a tmux session
`gdb -q ./app`	Debugging session
`perf record -g ./app`	Profile program
`valgrind -leak-check=full ./app`	Memory debugging
`docker build -t myapp .`	Build docker image
`cmake -S . -B build`	Configure CMake build
`ninja -C build`	Build with ninja

B. Appendix B: 90-Day Mastery Plan (Detailed)

B.1. Phase 1 — Foundation (Days 1–30)

- Week 1: Basic commands, navigation, editors.
- Week 2: Shell scripting basics, text processing.
- Week 3: Build systems (CMake), simple C++ builds.
- Week 4: Git workflows, code search tools (rg, fzf).

B.2. Phase 2 — Development (Days 31–60)

- Week 5-6: Debugging (gdb), sanitizers, valgrind.
- Week 7-8: Containers, CI, packaging, performance profiling.

B.3. Phase 3 — Mastery (Days 61–90)

- Kernel internals, system internals, real-time concepts.
- Contribute to an open-source C++ project (LLVM recommended).
- Interview prep and mock interviews.

C. Appendix C: Resources and Further Reading

- **Books**: *The Linux Programming Interface* (Michael Kerrisk), *Linux Kernel Development* (Robert Love)
- **Web**: man7.org, llvm.org, kernel.org
- **Tools**: perf, valgrind, gdb, clang-tidy, ccache, rg, fzf, tmux
- **Courses**: Explore CompilerSutra tutorials at compilersutra.com

Keep Learning, Keep Building!

CompilerSutra