香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Introduction to Linux

King Kai Chan

21-02-2026

Table of Contents

Agenda

Fundamentals

1. **Introduction** —What is Linux, distributions, why CP
2. **Getting Started** —WSL, setup, resources
3. **Terminal Basics** —Tips, wildcards, man, arguments
4. **Filesystem** —Directory structure, paths, nano

Practical Skills

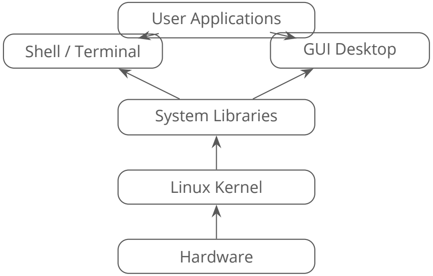
5. **Essential Commands** —ls, cd, cp, mv, rm, cat, ...
6. **Permissions & System** —chmod, sudo, apt
7. **C++ Compilation** —g++, flags, warnings, Makefile
8. **I/O Redirection** —Redirects, piping, chaining

Interactive Playground

Interactive playground available at `https://github.com/hkoi/linux-playground/`

What is Linux?

- A **free, open-source** operating system — like Windows or macOS
- Built on the **Linux kernel**, created by Linus Torvalds in 1991
- Powers most servers, supercomputers, and Android devices
- Highly customizable — you can inspect and modify anything



Linux Distributions

A **distribution** bundles the Linux kernel with software, package managers, and desktop environments.

Ubuntu
Most popular desktop distro. Great GUI & community.

Red Hat
Enterprise-focused. Corporate servers.

Fedora
Cutting-edge features. Community-driven.

Arch Linux
Minimalist. Highly customizable.

Linux Distributions

A **distribution** bundles the Linux kernel with software, package managers, and desktop environments.

Ubuntu
Most popular desktop distro. Great GUI & community.

Red Hat
Enterprise-focused. Corporate servers.

Fedora
Cutting-edge features. Community-driven.

Arch Linux
Minimalist. Highly customizable.

Note
All distros share the same kernel and shell commands —skills transfer across distributions.

Why Linux for Competitive Programming?

- **Competition environments use Linux** —NOI, TFT, and many ICPC regionals run on Linux

Why Linux for Competitive Programming?

- **Competition environments use Linux** —NOI, TFT, and many ICPC regionals run on Linux
- **Free and open source** —no licensing costs, install on any machine

Why Linux for Competitive Programming?

- **Competition environments use Linux** —NOI, TFT, and many ICPC regionals run on Linux
- **Free and open source** —no licensing costs, install on any machine
- **Superior shell** —bash/zsh syntax is far more powerful than Windows cmd

Why Linux for Competitive Programming?

- **Competition environments use Linux** —NOI, TFT, and many ICPC regionals run on Linux
- **Free and open source** —no licensing costs, install on any machine
- **Superior shell** —bash/zsh syntax is far more powerful than Windows cmd
- **Portable scripts** —shell commands work across all Linux distributions

Why Linux for Competitive Programming?

- **Competition environments use Linux** —NOI, TFT, and many ICPC regionals run on Linux
- **Free and open source** —no licensing costs, install on any machine
- **Superior shell** —bash/zsh syntax is far more powerful than Windows cmd
- **Portable scripts** —shell commands work across all Linux distributions
- **Developer ecosystem** —g++, gdb, valgrind, and other tools are first-class citizens

Which Distribution Should I Use?

Ubuntu

- Excellent GUI out of the box
- Largest community & docs
- Best for beginners

Get it: ubuntu.com

WSL (Windows)

- Linux shell inside Windows
- No VM needed, great perf.
- Access Windows files

Get it: Built into Win 10/11

Arch Linux

- Extremely lightweight
- Highly customizable
- Learn Linux internals

Get it: archlinux.org

Recommendation

Use WSL if you're on Windows. Use Ubuntu if you want a dedicated Linux machine.

Table of Contents

Windows Subsystem for Linux (WSL)

WSL lets you run a full Linux environment on Windows —no VM or dual boot needed.

WSL 1

- Translates Linux syscalls to Windows
- **Faster** cross-OS filesystem access
- No full Linux kernel
- Limited hardware support

WSL 2

- Runs a **real Linux kernel**
- Full system call compatibility
- **Docker** and **CUDA** support
- Slightly slower cross-OS file access

Windows Subsystem for Linux (WSL)

WSL lets you run a full Linux environment on Windows —no VM or dual boot needed.

WSL 1

- Translates Linux syscalls to Windows
- **Faster** cross-OS filesystem access
- No full Linux kernel
- Limited hardware support

WSL 2

- Runs a **real Linux kernel**
- Full system call compatibility
- **Docker** and **CUDA** support
- Slightly slower cross-OS file access

Recommendation

For competitive programming, WSL 2 is recommended —full kernel means everything just works.

Setting Up WSL

❶ Open **PowerShell as Administrator** and run:

```
wsl --install
```

Setting Up WSL

1. Open **PowerShell as Administrator** and run:

```
wsl --install
```

2. **Restart** your computer when prompted

Setting Up WSL

❶ Open **PowerShell as Administrator** and run:

```
wsl --install
```

❷ **Restart** your computer when prompted

❸ Open **Ubuntu** from the Start menu —first launch completes setup

Setting Up WSL

❶ Open **PowerShell as Administrator** and run:

```
wsl --install
```

❷ **Restart** your computer when prompted

❸ Open **Ubuntu** from the Start menu —first launch completes setup

❹ Create a **username and password** when prompted

Setting Up WSL

1. Open **PowerShell as Administrator** and run:

```
wsl --install
```

2. **Restart** your computer when prompted
3. Open **Ubuntu** from the Start menu —first launch completes setup
4. Create a **username and password** when prompted
5. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

Setting Up WSL

1. Open **PowerShell as Administrator** and run:

```
wsl --install
```

2. **Restart** your computer when prompted
3. Open **Ubuntu** from the Start menu —first launch completes setup
4. Create a **username and password** when prompted
5. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

Setting Up WSL

1. Open **PowerShell as Administrator** and run:

```
wsl --install
```

2. **Restart** your computer when prompted
3. Open **Ubuntu** from the Start menu —first launch completes setup
4. Create a **username and password** when prompted
5. Update your system:

```
sudo apt update && sudo apt upgrade -y
```

Requirements

Windows 10 version 2004+ or any Windows 11. Run `winver` to check.

How to Learn Linux Efficiently

Resources

- **Google it** —most problems have been solved
- **Stack Overflow** —Q&A for specific issues
- **ChatGPT / AI** —great for explaining commands
- **`man` pages** —built-in manuals
- **This presentation** —bookmark for reference

Practice

- **Use Linux daily** —muscle memory matters
- **Try the Linux Playground** —interactive missions
- **Break things** —that's how you learn (use WSL)
- **Read error messages** —they tell you what's wrong

Linux Playground —Interactive Practice

Try commands in your browser with guided missions.

`https://github.com/hkoi/linux-playground/`

6 mission levels from basic navigation to contest simulation.

Table of Contents

Opening the Terminal

Ubuntu

Press `Ctrl+Alt+T` to open the default terminal.

Or search for “Terminal” in the application menu.

WSL

- Open **Windows Terminal** → select Ubuntu tab
- Or type `wsl` in PowerShell
- Or search “Ubuntu” in the Start menu

What You’ll See

```
user@hostname: $
```

This is your **shell prompt** —it shows the username, machine name, and current directory (~ = home).

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates
- **Command history** —↑ / ↓ to cycle through previous commands

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates
- **Command history** —↑ / ↓ to cycle through previous commands
- **Cancel input** —`Ctrl+C` to discard the current line

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates
- **Command history** —↑ / ↓ to cycle through previous commands
- **Cancel input** —`Ctrl+C` to discard the current line
- **Clear screen** —`clear` or `Ctrl+L`

Terminal Tips & Shortcuts

- **Tab completion** — type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates
- **Command history** — ↑ / ↓ to cycle through previous commands
- **Cancel input** — `Ctrl+C` to discard the current line
- **Clear screen** — `clear` or `Ctrl+L`
- **Move cursor** — `Ctrl+A` (start), `Ctrl+E` (end of line)

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
  - Press `Tab` twice to list all matching candidates
- **Command history** —↑ / ↓ to cycle through previous commands
- **Cancel input** —`Ctrl+C` to discard the current line
- **Clear screen** —`clear` or `Ctrl+L`
- **Move cursor** —`Ctrl+A` (start), `Ctrl+E` (end of line)

Terminal Tips & Shortcuts

- **Tab completion** —type a few characters, press `Tab` to auto-complete
    - Press `Tab` twice to list all matching candidates
- **Command history** —↑ / ↓ to cycle through previous commands
- **Cancel input** —`Ctrl+C` to discard the current line
- **Clear screen** —`clear` or `Ctrl+L`
- **Move cursor** —`Ctrl+A` (start), `Ctrl+E` (end of line)

Example

To reference `this_is_a_super_long_filename`, type `this_is` then press `Tab`.

Wildcard Character —*

The * matches **any sequence of characters** in file/directory names.

```
# Directory contains: 1.txt, 2.txt, 3.txt, notes.md
rm *.txt            # Removes 1.txt, 2.txt, 3.txt (not notes.md)
ls *.md             # Lists only notes.md
```

Wildcard Character — *

The * matches **any sequence of characters** in file/directory names.

```
# Directory contains: 1.txt, 2.txt, 3.txt, notes.md
rm *.txt          # Removes 1.txt, 2.txt, 3.txt (not notes.md)
ls *.md           # Lists only notes.md
```

More examples with files `file123.txt` and `file456.txt`:

```
ls file*.txt      # Matches both
ls file*          # Matches both
ls f*.txt         # Matches both
```

Wildcard Character —*

The * matches **any sequence of characters** in file/directory names.

```
# Directory contains: 1.txt, 2.txt, 3.txt, notes.md
rm *.txt          # Removes 1.txt, 2.txt, 3.txt (not notes.md)
ls *.md           # Lists only notes.md
```

More examples with files `file123.txt` and `file456.txt`:

```
ls file*.txt      # Matches both
ls file*          # Matches both
ls f*.txt         # Matches both
```

Warning

`rm *` deletes **everything** in the current directory. Always verify with `ls` first!

Reading the Manual — `man`

Every command has a built-in manual page:

```
man cp        # Manual for the cp (copy) command
man ls        # Manual for the ls (list) command
man zip       # Manual for the zip command
```

Reading the Manual — `man`

Every command has a built-in manual page:

```
man cp        # Manual for the cp (copy) command
man ls        # Manual for the ls (list) command
man zip       # Manual for the zip command
```

Navigating man pages

Key	Action
↑ / ↓	Scroll line by line
`Space`	Scroll one page down
`/pattern`	Search for text
`q`	Quit

Reading the Manual — `man`

Every command has a built-in manual page:

```
man cp        # Manual for the cp (copy) command
man ls        # Manual for the ls (list) command
man zip       # Manual for the zip command
```

Navigating man pages

Key	Action
↑ / ↓	Scroll line by line
`Space`	Scroll one page down
`/pattern`	Search for text
`q`	Quit

Tip

Most commands support `-h` or `--help` for a shorter summary.

Command Arguments

Arguments modify how a command behaves —like function parameters in code.

Short form (single -)

```
ls -a           # all files
ls -l           # long format
ls -la          # combined
rm -rf folder/  # recursive + force
```

Long form (double --)

```
ls --all
rm --recursive --force folder/
g++ --version
```

Command Arguments

Arguments modify how a command behaves —like function parameters in code.

Short form (single -)

```
ls -a          # all files
ls -l          # long format
ls -la         # combined
rm -rf folder/ # recursive + force
```

Long form (double --)

```
ls --all
rm --recursive --force folder/
g++ --version
```

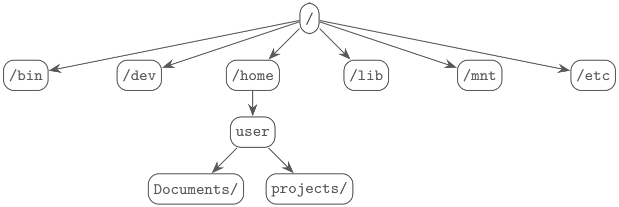
Common flag conventions

`-f` force	`-v` verbose	`-a` all	`-r` recursive	`-i` interactive	`-o` output	`-h` help

Table of Contents

Filesystem Overview

Linux organizes everything in a single tree rooted at /.



Unlike Windows (`C:\`, `D:\`), Linux has a **single root** / —everything is a subdirectory, including external drives.

Special Directories

Symbol	Name	Meaning
/	Root	Top of entire filesystem
~	Home	User's home
.	Current	Directory you're in now
..	Parent	One level up

```
# Current directory: /home/user/projects
pwd                # /home/user/projects
cd ..              # Now at /home/user
cd                # Now at /home/user
cd /               # Now at root
cd  /projects     # Back to projects
```

Special Directories

Symbol	Name	Meaning
/	Root	Top of entire filesystem
~	Home	User's home
.	Current	Directory you're in now
..	Parent	One level up

```
# Current directory: /home/user/projects
pwd               # /home/user/projects
cd ..             # Now at /home/user
cd               # Now at /home/user
cd /              # Now at root
cd  /projects    # Back to projects
```

Note

Your terminal always starts at ~ (home directory) unless you change it with `cd`.

Key System Directories

`/bin`
Essential command executables: `ls`, `cp`, `mv`, `cat`

`/lib`
Shared libraries needed by `/bin` and system binaries

`/dev`
Device files. `/dev/null` discards all data (a “black hole”)

`/mnt`
Where external drives and filesystems are temporarily mounted

`/home`
User directories: `/home/user1`, `/home/user2`, etc.

`/etc`
System-wide config files (network, users, services)

Relative vs Absolute Paths

Absolute path —starts from root /; always the same.

```
/home/user/projects/solution.cpp
```

Relative path —relative to your current directory.

```
./projects/solution.cpp
../user2/file.txt
```

Example from `/home/user`

Target	Absolute	Relative
File in projects	`/home/user/projects/a.cpp`	`./projects/a.cpp`
Parent	`/home`	`..`
Other user	`/home/user2/file.txt`	`../user2/file.txt`

Text Editor in Terminal — `nano`

`nano` is a simple, beginner-friendly terminal text editor.

```
nano myfile.cpp        # Open (or create) myfile.cpp for editing
```

Essential shortcuts

Shortcut	Action	Shortcut	Action
`Ctrl+X`	Exit nano	`Ctrl+O`	Save without exiting
`Y` → `Enter`	Save on exit	`Ctrl+K`	Cut current line
`N` → `Enter`	Discard on exit	`Ctrl+W`	Search text

For advanced editing, look into **vim** or **VS Code** with remote SSH.

Table of Contents

Command Overview

Navigation

- `pwd` —print working dir
- `ls` —list directory
- `cd` —change directory

File Operations

- `mkdir` —create dirs
- `touch` —create files
- `cp` —copy
- `mv` —move / rename
- `rm` —remove

Content & Search

- `cat` —display contents
- `echo` —print / write
- `diff` —compare files
- `grep` —search contents
- `zip` / `unzip`

List Directory Contents — `ls`

```
ls                # List files in current directory
ls -a             # Include hidden files (names starting with .)
ls -l             # Long format -- permissions, size, date
ls -la            # Both: hidden files + long format
ls /some/path     # List a specific directory
```

List Directory Contents — `ls`

```
ls              # List files in current directory
ls -a           # Include hidden files (names starting with .)
ls -l           # Long format -- permissions, size, date
ls -la          # Both: hidden files + long format
ls /some/path   # List a specific directory
```

Example output of `ls -la`

```
drwxr-xr-x  4 user user 4096 Feb 15 10:30 .
drwxr-xr-x  3 user user 4096 Feb 14 09:00 ..
-rw-r--r--  1 user user  220 Feb 14 09:00 .bashrc
drwxr-xr-x  2 user user 4096 Feb 15 10:30 projects
-rw-r--r--  1 user user  128 Feb 15 10:25 solution.cpp
```

`d` = directory, `-` = file. Hidden files start with `.` (only shown with `-a`).

Change Directory — `cd`

```
cd /            # Go to root directory
cd             # Go to home directory (/home/user)
cd ..           # Go up one level (parent directory)
cd projects     # Enter the "projects" subdirectory
cd  /projects  # Go to projects under home (absolute via  )
```

Change Directory —cd

```
cd /            # Go to root directory
cd              # Go to home directory (/home/user)
cd ..           # Go up one level (parent directory)
cd projects     # Enter the "projects" subdirectory
cd  /projects   # Go to projects under home (absolute via  )
```

Practical navigation

```
user@linux: $ cd projects/contest
user@linux: /projects/contest$ ls
A.cpp  B.cpp  input.txt
user@linux: /projects/contest$ cd ../..
user@linux: $ pwd
/home/user
```

The prompt updates to show your current path —always know where you are.

Create Directories — `mkdir`

```
mkdir mydir                        # Create a single directory
mkdir dir1 dir2 dir3               # Create multiple directories
mkdir -p CP/Codeforces/Round900    # Create nested directories
```

Create Directories — `mkdir`

```
mkdir mydir                     # Create a single directory
mkdir dir1 dir2 dir3            # Create multiple directories
mkdir -p CP/Codeforces/Round900   # Create nested directories
```

Why `-p`?

Without it, creating `CP/Codeforces/Round900` fails if parent dirs don't exist.

```
mkdir CP/Codeforces/Round900      # Error: No such file or directory
mkdir -p CP/Codeforces/Round900   # Creates all missing parents
```

Create Files —`touch`

```
touch file.txt                  # Create an empty file
touch a.cpp b.cpp c.cpp         # Create multiple files at once
```

Create Files — `touch`

```
touch file.txt                  # Create an empty file
touch a.cpp b.cpp c.cpp         # Create multiple files at once
```

Alternative: create with >

```
> file.txt                      # Also creates an empty file
echo "content" > file.txt       # Create a file with content
```

Create Files —`touch`

```
touch file.txt                  # Create an empty file
touch a.cpp b.cpp c.cpp         # Create multiple files at once
```

Alternative: create with >

```
> file.txt                      # Also creates an empty file
echo "content" > file.txt       # Create a file with content
```

Note

`touch` is safe on existing files —it only updates the timestamp. `> file.txt` will **erase** existing content.

Copy —cp

```
cp source.txt dest.txt                  # Copy a file
cp file1.txt file2.txt dest_folder/     # Copy multiple files
cp -r source_dir/ dest_dir/             # Copy a directory recursively
```

Copy —cp

```
cp source.txt dest.txt                  # Copy a file
cp file1.txt file2.txt dest_folder/     # Copy multiple files
cp -r source_dir/ dest_dir/             # Copy a directory recursively
```

Key points

- **File** copy —no flags needed
- **Directory** copy —requires `-r` (recursive)
- Overwrites silently (`-i` to prompt, `-f` to force)

```
cp solution.cpp backup_solution.cpp     # Quick backup
cp -r contest/ contest_backup/          # Backup entire directory
```

Move / Rename — `mv`

```
mv source.txt dest_folder/          # Move file into a directory
mv old_name.cpp new_name.cpp        # Rename a file
mv dir1/ dir2/                      # Move (or rename) a directory
```

Move / Rename — mv

```
mv source.txt dest_folder/         # Move file into a directory
mv old_name.cpp new_name.cpp       # Rename a file
mv dir1/ dir2/                     # Move (or rename) a directory
```

mv vs cp

	cp	mv
Original file	**Kept**	**Removed**
Directory flag	Needs -r	No flag needed
Use case	Duplicate	Relocate or rename

`mv` is the standard way to **rename** files and directories in Linux.

Remove — `rm`

```
rm file.txt              # Remove a single file
rm file1 file2 file3     # Remove multiple files
rm -r directory/         # Remove a directory and all contents
rm -rf directory/        # Force remove -- no prompts
```

Remove —`rm`

```
rm file.txt              # Remove a single file
rm file1 file2 file3     # Remove multiple files
rm -r directory/         # Remove a directory and all contents
rm -rf directory/        # Force remove -- no prompts
```

Warning

`rm` is **permanent** —there is no recycle bin. Always double-check paths and verify with `ls` first!

Remove —`rm`

```
rm file.txt             # Remove a single file
rm file1 file2 file3    # Remove multiple files
rm -r directory/        # Remove a directory and all contents
rm -rf directory/       # Force remove -- no prompts
```

Warning

`rm` is **permanent** —there is no recycle bin. Always double-check paths and verify with `ls` first!

```
# Safe workflow: verify before deleting
ls *.o                # Check which .o files exist
rm *.o                # Now remove them
```

Display File Contents — `cat`

```
cat file.txt                 # Print file contents to terminal
cat file1.txt file2.txt      # Print multiple files (concatenate)
```

Display File Contents —cat

```
cat file.txt                # Print file contents to terminal
cat file1.txt file2.txt     # Print multiple files (concatenate)
```

Example

```
user@linux: $ cat hello.cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
}
```

Display File Contents —`cat`

```
cat file.txt                # Print file contents to terminal
cat file1.txt file2.txt     # Print multiple files (concatenate)
```

Example

```
user@linux: $ cat hello.cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
}
```

For long files, use `less file.txt` (scrollable) or `head -n 20 file.txt` (first 20 lines).

Print & Write —`echo`

```
echo "Hello, World!"                # Print text to terminal
echo "Hello" > file.txt             # Write to file (overwrites)
echo "More text" >> file.txt        # Append to file
```

Print & Write —echo

```
echo "Hello, World!"            # Print text to terminal
echo "Hello" > file.txt         # Write to file (overwrites)
echo "More text" >> file.txt    # Append to file
```

> vs >>

Operator	Behavior
>	**Overwrites** the file (creates if doesn’t exist)
>>	**Appends** to the end of the file

```
echo "line 1" > output.txt    # output.txt: "line 1"
echo "line 2" > output.txt    # output.txt: "line 2" (line 1 gone!)
echo "line 3" >> output.txt   # output.txt: "line 2\nline 3"
```

Compare Files — `diff`

```
diff file1.txt file2.txt        # Compare two files line by line
diff -Z a.out b.out             # Ignore trailing whitespace
```

Compare Files —`diff`

```
diff file1.txt file2.txt        # Compare two files line by line
diff -Z a.out b.out             # Ignore trailing whitespace
```

Reading diff output

```
user@linux: $ diff expected.out my.out
3c3
< 42
---
> 41
```

3c3 = line 3 **c**hanged · < = first file · > = second file · **No output** = identical

Compare Files —`diff`

```
diff file1.txt file2.txt        # Compare two files line by line
diff -Z a.out b.out             # Ignore trailing whitespace
```

Reading diff output

```
user@linux: $ diff expected.out my.out
3c3
< 42
---
> 41
```

`3c3` = line 3 **c**hanged · < = first file · > = second file · **No output** = identical

CP tip

Use `diff -Z` to ignore trailing spaces —matches how most judges compare output.

Search in Files — grep

```
grep "hello" file.txt           # Find lines containing "hello"
grep -i "hello" file.txt        # Case-insensitive search
grep -e "regex" file.txt        # Match a regular expression
grep -r "pattern" directory/    # Search recursively in all files
grep -n "pattern" file.txt      # Show line numbers
```

Search in Files — `grep`

```
grep "hello" file.txt          # Find lines containing "hello"
grep -i "hello" file.txt       # Case-insensitive search
grep -e "regex" file.txt       # Match a regular expression
grep -r "pattern" directory/   # Search recursively in all files
grep -n "pattern" file.txt     # Show line numbers
```

Example

```
user@linux: $ grep -n "int" solution.cpp
3:int main() {
4:    int n;
7:    int result = n * 2;
```

Archive — `zip` / `unzip`

```
zip archive.zip file1 file2 file3        # Zip multiple files
zip -r archive.zip directory/            # Zip a directory (requires -r)
unzip archive.zip                        # Extract all contents
```

Archive — `zip` / `unzip`

```
zip archive.zip file1 file2 file3       # Zip multiple files
zip -r archive.zip directory/           # Zip a directory (requires -r)
unzip archive.zip                       # Extract all contents
```

Example workflow

```
zip -r contest.zip CP/Codeforces/Round900/  # Package solutions
unzip starter_code.zip                      # Extract downloaded archive
```

`-r` is required when zipping directories (just like `cp` and `rm`).

Table of Contents

User Permissions

Linux controls file access with three classes and three operations:

Permission classes

- **Owner** —user who created the file
- **Group** —users in the file's group
- **Others** —everyone else

Operations: **r** (read) · **w** (write) · **x** (execute)

Reading `ls -l` output

```
-rwxr-xr-- 1 user group  script.sh
 ^^^   owner permissions   (rwx)
    ^^^ group permissions (r-x)
       ^^^ other perms     (r--)
```

User Permissions

Linux controls file access with three classes and three operations:

Permission classes

- **Owner** —user who created the file
- **Group** —users in the file's group
- **Others** —everyone else

Operations: **r** (read) · **w** (write) · **x** (execute)

Reading `ls -l` output

```
-rwxr-xr-- 1 user group  script.sh
 ^^^   owner permissions   (rwx)
    ^^^ group permissions (r-x)
       ^^^ other perms     (r--)
```

Note

If you see **"Permission denied"**, you lack the required permission for that operation.

Change Permissions —`chmod`

```
chmod +x script.sh         # Add execute permission
chmod -x script.sh         # Remove execute permission
chmod +r file.txt          # Add read permission
chmod +w file.txt          # Add write permission
chmod +rwx file.txt        # Add all permissions
```

Change Permissions —`chmod`

```
chmod +x script.sh          # Add execute permission
chmod -x script.sh          # Remove execute permission
chmod +r file.txt           # Add read permission
chmod +w file.txt           # Add write permission
chmod +rwx file.txt         # Add all permissions
```

When you'll need this —making a compiled binary or script executable:

```
g++ solution.cpp -o solution
chmod +x solution         # Usually not needed -- g++ sets this
./solution

chmod +x compile.sh       # Make a provided shell script runnable
./compile.sh
```

Superuser — `sudo`

`sudo` runs a command with **root (administrator) privileges**.

```
sudo apt update                # Update package catalogue as root
sudo apt install g++            # Install a package as root
```

Superuser —`sudo`

`sudo` runs a command with **root (administrator) privileges**.

```
sudo apt update                # Update package catalogue as root
sudo apt install g++            # Install a package as root
```

Why is `sudo` needed?

- System-wide changes require root permissions
- Your normal account is intentionally restricted for safety
- `sudo` temporarily elevates privileges —prompts for your password

Warning

The **root user** has unrestricted access. Never log in as root daily —use `sudo` when needed.

Package Manager —`apt`

`apt` is Ubuntu's package manager —an **app store for the terminal**.

```
# Always update the catalogue first
sudo apt update

# Install packages
sudo apt install g++
sudo apt install python3
```

Package Manager —`apt`

`apt` is Ubuntu's package manager —an **app store for the terminal**.

```
# Always update the catalogue first
sudo apt update

# Install packages
sudo apt install g++
sudo apt install python3
```

```
# Upgrade all installed packages
sudo apt upgrade
```

Package Manager —`apt`

`apt` is Ubuntu’s package manager —an **app store for the terminal**.

```
# Always update the catalogue first
sudo apt update

# Install packages
sudo apt install g++
sudo apt install python3
```

```
# Upgrade all installed packages
sudo apt upgrade
```

```
# Remove a package
sudo apt remove g++           # Remove the package
sudo apt purge g++            # Remove package AND config files
```

Table of Contents

Installing and Using g++

g++ is the GNU C++ compiler —the standard tool for CP.

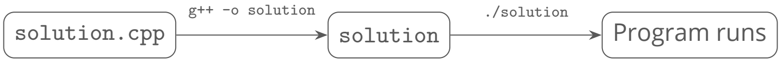
```
sudo apt update && sudo apt install g++     # Install
g++ solution.cpp -o solution                # Compile
```

Installing and Using g++

g++ is the GNU C++ compiler —the standard tool for CP.

```
sudo apt update && sudo apt install g++     # Install
g++ solution.cpp -o solution                # Compile
```

What this does



`-o solution` sets the output name. Without `-o`, the default is `a.out`.

Compilation Flags

Add flags for C++ standard, optimization, and warnings:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow
```

Compilation Flags

Add flags for C++ standard, optimization, and warnings:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow
```

Flag	Purpose
`-std=c++17`	Use C++17 standard
`-O2`	Optimize for execution speed (most judges use this)
`-Wall`	Enable most compiler warnings
`-Wshadow`	Warn when a variable shadows another
`-Wextra`	Even more warnings beyond `-Wall`

Compilation Flags

Add flags for C++ standard, optimization, and warnings:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow
```

Flag	Purpose
`-std=c++17`	Use C++17 standard
`-O2`	Optimize for execution speed (most judges use this)
`-Wall`	Enable most compiler warnings
`-Wshadow`	Warn when a variable shadows another
`-Wextra`	Even more warnings beyond `-Wall`

Tip

Always compile with `-Wall` during practice —it catches bugs that cost hours in contests.

`-Wall` —Warning Examples

Format string mismatch

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    printf("%lld\n", x);
    // Warning: '%lld' expects
    // 'long long int', not 'int'
}
```

Unused variable

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    int y = 9;  // Warning: unused 'y'
    printf("%d\n", x);
}
```

-Wall —Warning Examples

Format string mismatch

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    printf("%lld\n", x);
    // Warning: '%lld' expects
    // 'long long int', not 'int'
}
```

Unused variable

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    int y = 9;  // Warning: unused 'y'
    printf("%d\n", x);
}
```

Note

Unused variables often indicate a typo —warnings catch these before they become bugs.

`-Wshadow` —Variable Shadowing

When an inner scope re-declares a variable name, the outer one is **shadowed**:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    for (int i = 0; i < 3; i++) {
        int x = 9;              // Warning: 'x' shadows a previous local
        printf("%d\n", x);      // Prints 9, not 5!
    }
}
```

`-Wshadow` —Variable Shadowing

When an inner scope re-declares a variable name, the outer one is **shadowed**:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int x = 5;
    for (int i = 0; i < 3; i++) {
        int x = 9;              // Warning: 'x' shadows a previous local
        printf("%d\n", x);      // Prints 9, not 5!
    }
}
```

- Common source of bugs in nested loops
- Inner `x` hides outer `x` —compiles silently without this flag
- **Always compile with** `-Wshadow` to catch these early

Sanitizers —Catch Runtime Bugs

Add sanitizer flags to detect bugs that compile cleanly but crash:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow \
    -fsanitize=address,undefined
```

Sanitizers —Catch Runtime Bugs

Add sanitizer flags to detect bugs that compile cleanly but crash:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow \
    -fsanitize=address,undefined
```

What they catch

Sanitizer	Catches
`address`	Out-of-bounds array access, use-after-free, memory leaks
`undefined`	Integer overflow, null pointer dereference, shift errors

Sanitizers —Catch Runtime Bugs

Add sanitizer flags to detect bugs that compile cleanly but crash:

```
g++ solution.cpp -o solution -std=c++17 -O2 -Wall -Wshadow \
    -fsanitize=address,undefined
```

What they catch

Sanitizer	Catches
`address`	Out-of-bounds array access, use-after-free, memory leaks
`undefined`	Integer overflow, null pointer dereference, shift errors

Warning

Sanitizers slow your program ~2–3x —use for **debugging only**, not timing.

Running Binaries

After compilation, run with `./`:

```
g++ solution.cpp -o solution
./solution
```

Running Binaries

After compilation, run with `./`:

```
g++ solution.cpp -o solution
./solution
```

Why `./` is needed —Linux looks for executables in `PATH` dirs (`/bin`, `/usr/bin`). Your current directory isn't in `PATH` —`./` tells the shell to look here.

Measuring execution time

```
time ./solution < input.txt
# real    0m0.032s  <- wall clock time (the one you care about)
# user    0m0.028s  <- CPU time in user mode
# sys     0m0.004s  <- CPU time in kernel mode
```

Running Binaries

After compilation, run with `./`:

```
g++ solution.cpp -o solution
./solution
```

Why `./` is needed —Linux looks for executables in `PATH` dirs (`/bin`, `/usr/bin`). Your current directory isn't in `PATH` —`./` tells the shell to look here.

Measuring execution time

```
time ./solution < input.txt
# real    0m0.032s  <- wall clock time (the one you care about)
# user    0m0.028s  <- CPU time in user mode
# sys     0m0.004s  <- CPU time in kernel mode
```

CP tip

If `real` time exceeds ~1–2s for a typical test case, your solution is likely too slow.

Grader Tasks —Multiple Source Files

Many TFT/IOI problems provide a **grader** —you implement functions, the grader handles I/O.

```
# solution.cpp    -- your code (implements functions)
# sample_grader.cpp -- provided (contains main)

g++ -o solution solution.cpp sample_grader.cpp -std=c++17 -O2
```

Grader Tasks —Multiple Source Files

Many TFT/IOI problems provide a **grader** —you implement functions, the grader handles I/O.

```
# solution.cpp     -- your code (implements functions)
# sample_grader.cpp -- provided (contains main)

g++ -o solution solution.cpp sample_grader.cpp -std=c++17 -O2
```

Using a provided compile script

```
chmod +x compile.sh && ./compile.sh     # Make executable and run
cat compile.sh                          # Peek at the compilation command
```

You can also compile manually and add any extra flags you need.

Makefile —Automate Compilation

A **Makefile** saves you from retyping the full g++ command:

```
% : %.cpp
^^Ig++ $< -o $@ -std=c++17 -Wall -Wshadow -O2
```

Makefile —Automate Compilation

A **Makefile** saves you from retyping the full `g++` command:

```
% : %.cpp
^^Ig++ $< -o $@ -std=c++17 -Wall -Wshadow -O2
```

How it works

- `%` matches any name → `make solution` looks for `solution.cpp`
- `$<` = source file, `$@` = target name
- **Must use tab** for indentation (not spaces)

```
sudo apt install make      # Install make (if needed)
make solution              # Compiles solution.cpp -> solution
make A                     # Compiles A.cpp -> A
```

Table of Contents

I/O Redirection

Redirect input/output from/to files —essential for testing contest problems.

```
./solution < input.txt              # Read stdin from file
./solution > output.txt             # Write stdout to file
./solution < input.txt > output.txt     # Both at once
```

I/O Redirection

Redirect input/output from/to files —essential for testing contest problems.

```
./solution < input.txt                  # Read stdin from file
./solution > output.txt                 # Write stdout to file
./solution < input.txt > output.txt     # Both at once
```

Example workflow

```
echo "23382338" > input.txt             # Create sample input
g++ solution.cpp -o solution            # Compile
./solution < input.txt > output.txt     # Run with redirected I/O
cat output.txt                          # Check the result
```

I/O Redirection

Redirect input/output from/to files —essential for testing contest problems.

```
./solution < input.txt                # Read stdin from file
./solution > output.txt               # Write stdout to file
./solution < input.txt > output.txt   # Both at once
```

Example workflow

```
echo "23382338" > input.txt           # Create sample input
g++ solution.cpp -o solution          # Compile
./solution < input.txt > output.txt   # Run with redirected I/O
cat output.txt                        # Check the result
```

Tip

Large inputs cause terminal lag. File redirection is instant and reproducible.

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow
- **Speed** —terminal output is much slower than writing to a file

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow
- **Speed** —terminal output is much slower than writing to a file
- **Reproducibility** —keep input files for repeated testing

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow
- **Speed** —terminal output is much slower than writing to a file
- **Reproducibility** —keep input files for repeated testing
- **Automated testing** —combine with `diff` to verify output:

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow
- **Speed** —terminal output is much slower than writing to a file
- **Reproducibility** —keep input files for repeated testing
- **Automated testing** —combine with `diff` to verify output:

Why Use I/O Redirection?

- **Large inputs** —pasting thousands of lines into the terminal is slow
- **Speed** —terminal output is much slower than writing to a file
- **Reproducibility** —keep input files for repeated testing
- **Automated testing** —combine with `diff` to verify output:

```
./solution < input.txt > my_output.txt
diff -Z my_output.txt expected_output.txt
```

No output from `diff` = your answer matches exactly.

Command Chaining — &&

Run commands in sequence —next runs only if previous **succeeds**:

```
make solution && ./solution < input.txt
sudo apt update && sudo apt install g++
g++ sol.cpp -o sol && ./sol < in.txt > out.txt && diff -Z out.txt ans.txt
```

Command Chaining —&&

Run commands in sequence —next runs only if previous **succeeds**:

```
make solution && ./solution < input.txt
sudo apt update && sudo apt install g++
g++ sol.cpp -o sol && ./sol < in.txt > out.txt && diff -Z out.txt ans.txt
```

&& vs ;

Operator	Behavior
&&	Next runs **only if** previous succeeded
;	Next runs **regardless** of previous result

Using && is safer —if compilation fails, you won’t run the old binary.

Piping — |

The pipe sends **output of one command** as **input to another**:

```
./solution < input.txt | sort         # Sort program output
./solution < input.txt | wc -l       # Count output lines
./solution < input.txt | grep "Error" # Search output
```

Piping — |

The pipe sends **output of one command** as **input to another**:

```
./solution < input.txt | sort          # Sort program output
./solution < input.txt | wc -l        # Count output lines
./solution < input.txt | grep "Error" # Search output
```

Copying output to clipboard

```
cat solution.cpp | clip.exe                    # WSL (Windows)
cat solution.cpp | xclip -selection clipboard  # Linux with xclip
```

Useful when output is too long to select with the mouse.

Terminal Control

Stopping a command

Press `Ctrl+C` to **terminate** the current command.
Use when:

- Program enters an infinite loop
- Output is flooding the terminal
- You want to cancel an operation

Useful shortcuts

Shortcut	Action
`Ctrl+C`	Kill current process
`Ctrl+Z`	Suspend process
`Ctrl+D`	Send EOF (end input)
`Ctrl+L`	Clear screen

Terminal Control

Stopping a command

Press `Ctrl+C` to **terminate** the current command.
Use when:

- Program enters an infinite loop
- Output is flooding the terminal
- You want to cancel an operation

Useful shortcuts

Shortcut	Action
`Ctrl+C`	Kill current process
`Ctrl+Z`	Suspend process
`Ctrl+D`	Send EOF (end input)
`Ctrl+L`	Clear screen

CP tip

If your solution runs for more than a few seconds, it’s likely TLE —`Ctrl+C` and optimize.

Table of Contents

Summary

What we covered

- What Linux is and why it matters for CP
- Navigating the filesystem and terminal
- Essential commands: `ls`, `cd`, `cp`, `mv`, `rm`, `cat`, `mkdir`, `echo`, `diff`, `grep`, `zip`
- Permissions with `chmod` and `sudo`
- Package management with `apt`
- C++ compilation with `g++` and flags
- I/O redirection, piping, and chaining

Compilation cheat sheet

```
# Compile
g++ sol.cpp -o sol -std=c++17 \
    -O2 -Wall -Wshadow

# Run with I/O redirection
./sol < input.txt > output.txt

# Check output
diff -Z output.txt expected.txt

# All in one
make sol && ./sol < in.txt \
    > out.txt && diff -Z out.txt ans.txt
```

Try It Yourself

Practice everything you've learned in the interactive Linux Playground:

`https://github.com/hkoi/linux-playground/`

Complete all 6 mission levels —from basic navigation to a full contest simulation workflow.

Questions?

`https://github.com/hkoi/linux-playground/`