NED University of Engineering and Technology
IC Design Special Course
Batch: 01

Hands on Training: C++, GCC, Git

Course Instructor: Engr. Firdous Riaz

Week 1,2,3,4

Contact Details of Course Instructor

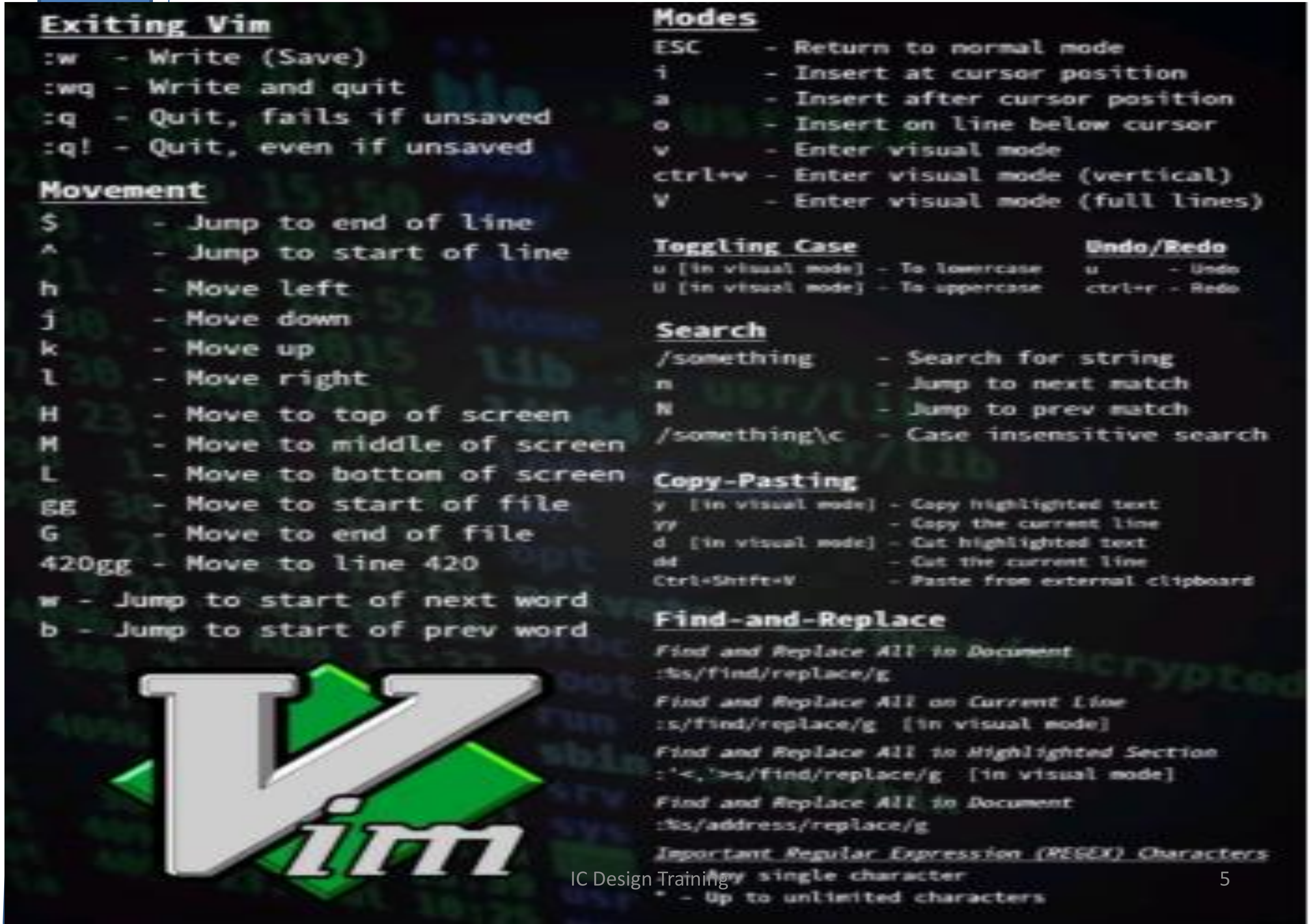
- Name: Engr. Firdous Riaz (TSCS Class Advisor & Lecturer - CS&IT Department)
- Email: firdousriaz@cloud.neduet.edu.pk
- Office: Ext: 2498
- Consultation hours: Tuesday (2:30 pm to 3:30 pm)
- Office: Room 5, First Floor, CSIT department (Office Area)

Linux Distribution

- For training purpose, **Ubuntu 22.04 LTS** or **Linux Mint** is easiest.
- Download ISO from https://ubuntu.com/download/desktop
- Install it directly on hardware or use a **virtual machine** (VirtualBox / VMware Workstation).
- Minimum requirements:
  - 2 CPU cores
  - 8 GB RAM
  - 80 GB HardDisk

- Download virtual machine box,
- Download Ubuntu version
- Setup virtual environment
- Run the linux based system
- Download and test Vim Editor

VIM Commands



Exiting Vim

```
:w   - Write (Save)
:wq  - Write and quit
:q   - Quit, fails if unsaved
:q!  - Quit, even if unsaved
```

Movement

```
$       - Jump to end of line
^       - Jump to start of line
h       - Move left
j       - Move down
k       - Move up
l       - Move right
H       - Move to top of screen
M       - Move to middle of screen
L       - Move to bottom of screen
gg      - Move to start of file
G       - Move to end of file
420gg - Move to line 420
w - Jump to start of next word
b - Jump to start of prev word
```

Modes

```
ESC      - Return to normal mode
i        - Insert at cursor position
a        - Insert after cursor position
o        - Insert on line below cursor
v        - Enter visual mode
ctrl+v - Enter visual mode (vertical)
V        - Enter visual mode (full lines)
```

Toggling Case

```
u [in visual mode] - To lowercase
U [in visual mode] - To uppercase
```

Undo/Redo

```
u      - Undo
ctrl+r - Redo
```

Search

```
/something      - Search for string
n               - Jump to next match
N               - Jump to prev match
/something\c    - Case insensitive search
```

Copy-Pasting

```
y  [in visual mode] - Copy highlighted text
yy                  - Copy the current line
d  [in visual mode] - Cut highlighted text
dd                  - Cut the current line
Ctrl+Shift+V        - Paste from external clipboard
```

Find-and-Replace

Find and Replace All in Document

```
:%s/find/replace/g
```

Find and Replace All on Current Line

```
:s/find/replace/g  [in visual mode]
```

Find and Replace All in Highlighted Section

```
:'<,'>s/find/replace/g  [in visual mode]
```

Find and Replace All in Document

```
:%s/address/replace/g
```

Important Regular Expression (REGEX) Characters

```
Any single character
* - Up to unlimited characters
```

Install the C++ Build Toolchain

Once the OS is installed,
open Terminal (Ctrl + Alt + T).

Run these commands:

```
sudo apt update
sudo apt install build-essential –y
```

This installs:

gcc → GNU C compiler
g++ → GNU C++ compiler
make → Build automation tool
gdb → Debugger

Verify installation:

```
gcc –version
g++ --version
make --version
```

Compiling and Building C++ Programs on Linux Using GCC and Make

Learning Outcomes:

By the end of the class, students will be able to:

- Use the Linux terminal to create, compile, and run C++ programs.
- Use g++ manually to compile single and multiple files.
- Automate builds using a Makefile.
- Understand basic Makefile rules and dependency structure.

Introduction & Environment Setup

- Explain compiler, linker, build process. Verify tools installed. kernels

Manual Compilation with g++

- Practice compiling single and multiple source files.

Using Make and Makefile

- Write and run a Makefile. Modify and observe behavior.

Debugging Tips

- Discuss errors, common mistakes, cleaning builds.

Compiling and Running C++ Programs on Linux using GCC and Make

❑Objective:

- To learn how to compile, link, and automate C++ builds using g++ and make.

❑Prerequisites:

- Ubuntu or any Linux distro with build-essential installed.
- Basic knowledge of C++ programming.

Creating and Compiling a C++ Program

Open the terminal and create a folder:

- mkdir cpp_lab
- cd cpp_lab

Create a simple program:

- vim hello.cpp
- #include <iostream>
- using namespace std;
- int main() {
- cout << "Hello, Linux World!" << endl;
- return 0;}

Compile and run manually:

- g++ hello.cpp -o hello
- ./hello

Compiling Multiple Source Files

Create files:

- vim main.cpp
- vim mathlib.cpp
- vim mathlib.h

Code:

- mathlib.h
- int add(int, int);
- mathlib.cpp
- #include "mathlib.h“
- int add(int a, int b) { return a + b; }
- main.cpp
- #include <iostream>
- #include "mathlib.h“
- using namespace std;
- int main() {
- cout << "Sum = " << add(3, 4) << endl;
- return 0;}

Compile manually:

- g++ -c mathlib.cpp
- g++ -c main.cpp
- g++ main.o mathlib.o -o program
- ./program

Automating Builds with Makefile

- Create Makefile:
- vim Makefile
- Content:
- # Makefile Example
- all: program
- program: main.o mathlib.o
- 	g++ main.o mathlib.o -o program
- main.o: main.cpp mathlib.h
- 	g++ -c main.cpp
- mathlib.o: mathlib.cpp mathlib.h
- 	g++ -c mathlib.cpp
- clean:
- 	rm -f *.o program
- Important: use Tab, not spaces, before each command.
- Run:
- Make
- ./program
- Clean the project:
- make clean

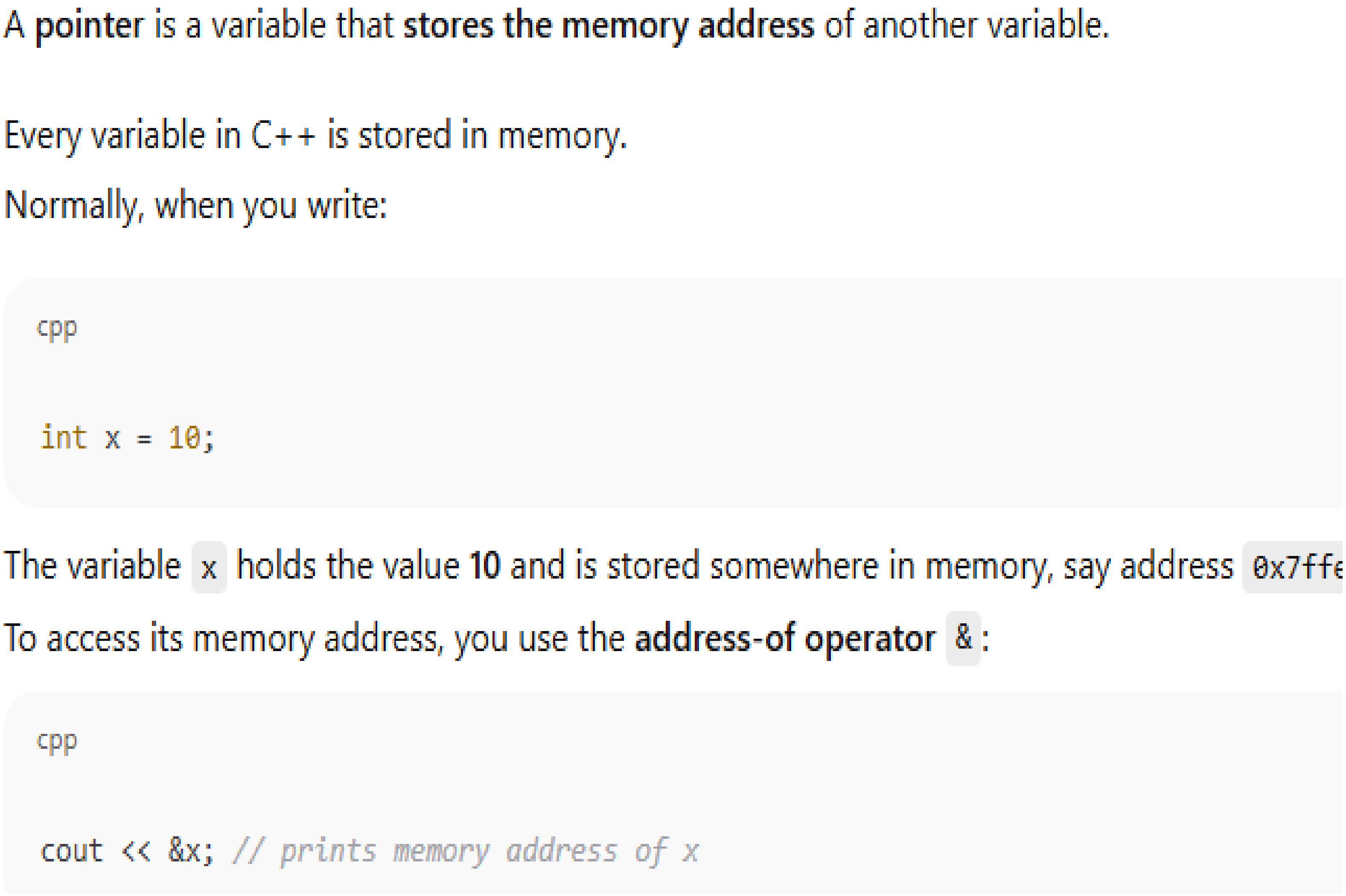
Additional Points

- Common errors: missing headers, forgetting tabs in Makefile.
- Check dependency updates by modifying one .cpp file and rerunning make.
- .PHONY and variables
- debug symbols: g++ -g
- Use make run target to automate running after build.

TASK1

- Build a Calculator in C++
- Compile/run using g++

Pointers in C++



A **pointer** is a variable that **stores the memory address** of another variable.

Every variable in C++ is stored in memory.

Normally, when you write:

```cpp
int x = 10;
```

The variable `x` holds the value **10** and is stored somewhere in memory, say address `0x7ffe`

To access its memory address, you use the **address-of operator** `&`:

```cpp
cout << &x; // prints memory address of x
```

Pointers in C++

Declaring a Pointer

A pointer variable is declared using the * symbol:

int* ptr;

Here, ptr is a pointer that can store the **address of an integer** variable.

To assign the address of a variable to it:

int x = 10; int* ptr = &x;

Now:

•ptr → stores the address of x

•*ptr → accesses the **value** stored at that address

So:

cout << ptr; // prints address of x cout << *ptr; // prints value of x (10)

Pointers in C++

Changing Value Using a Pointer

Because *ptr refers to the same memory location as x, you can change x via the pointer:

*ptr = 20; // modifies x directly cout << x; // prints 20

Pointer to Pointer

A pointer can also store the address of another pointer:

int x = 5; int* p1 = &x; int** p2 = &p1; cout << **p2; // prints 5

Pointers and Arrays

Arrays and pointers are closely related:

int arr[3] = {10, 20, 30}; int* p = arr; // arr gives address of first element cout << *p; // prints 10 cout << *(p+1); // prints 20

Pointers in C++

Pointer and Functions

Pointers are used to **pass data by reference** to functions:

```
void increment(int* num) { (*num)++; } int main() { int x = 10; increment(&x); cout << x; // prints 11 }
```

Pointers in C++

Dynamic Memory Allocation (new / delete)

C++ allows allocating memory at runtime using new:

```
int* p = new int; // allocate memory for one int *p = 50;
cout << *p; // prints 50 delete p; // free memory
```

For arrays:

```
int* arr = new int[5]; for(int i=0; i<5; i++) arr[i] = i*10;
delete[] arr;
```

TASK 2

Write a C++ program that uses pointers to:

1.Store and display the value and address of an integer variable.
2.Create a pointer to pointer and display all levels of indirection.
3.Dynamically allocate an array of integers (size entered by user).
4.Fill the array with user input using pointer arithmetic.
5.Display the array elements and their memory addresses.
6.Free the allocated memory using delete[].

Bonus Questions to explore

- What are Structs?
- What are def () ,def end declarations ?
- What are if , endif ?

OOP in C++

Summary of Core OOP Concepts

Concept	Meaning	Example
Class/Object	Blueprint / Instance	class Car {}
Encapsulation	Hide data, expose through methods	private balance
Inheritance	Reuse and extend behavior	class Car : public Vehicle
Polymorphism	Same interface, different forms	virtual void speak()
Abstraction	Hide implementation details	Pure virtual function

TASK 3

Implementation Task — “Bank Management System”
Goal: Implement a small program that demonstrates **all major OOP concepts**.

□ **Requirements**

1. **Create a base class** Account
   - Data members: accountNumber, balance
   - Functions: deposit(), withdraw(), displayBalance()
2. **Encapsulation**
   - Make data members private.
   - Access them via public functions.
3. **Inheritance**
   - Create derived classes:
     - SavingsAccount
     - CurrentAccount
   - Each has its own version of withdraw() (for example, apply limits or charges).
4. **Polymorphism**
   - Use a virtual void withdraw() in the base class.
   - Override it in derived classes.
5. **Abstraction**
   - Use a pointer of type Account* to refer to derived objects.
6. **Object Creation**
   - Create multiple account objects and use them to deposit, withdraw, and display balances.

Git – Conceptual

- **What is Git?**
- Version control system to track code changes
- Helps collaborate with others
- Maintains project history & branches
- **How to Use Git (Basic Flow)**
- Install Git & create GitHub/GitLab account
- Create or clone repo
- Add files → Commit → Push to remote
- Pull updates to stay synced

Git Commands

Action	Command
Set username/email	git config --global user.name "Name"
Create repo	git init
Clone repo	git clone <url>
Check status	git status
Add files	git add .
Commit changes	git commit -m "message"
Push code	git push origin main
Pull latest	git pull
Create branch	git branch new-branch
Switch branch	git checkout new-branch

Explore - Git Commands

- Git Basics (Config, Clone, Status, Log, Add, Commit, Push, Pull)● https://docs.github.com/en/get-started/quickstart/create-a-repo● https://git-scm.com/docs/gittutorial● https://product.hubspot.com/blog/git-and-github-tutorial-for-beginnersGit Basics - Visual Tutorial● https://marklodato.github.io/visual-git-guide/index-en.html● https://agripongit.vincenttunru.com/Branches● https://www.atlassian.com/git/tutorials/using-branches● https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-MergingMerge Requests (also called Pull Requests)● https://docs.gitlab.com/ee/user/project/merge_requests/getting_started.htmlRebasing● https://www.atlassian.com/git/tutorials/rewriting-history/git-rebaseGitlab Issues● https://docs.gitlab.com/ee/user/project/issues/

- **Deploy/Upload Project to GitHub**
- Create new repo on GitHub
- In project folder run:
- git init git remote add origin <repo_url> git add . git commit -m "First commit" git branch -M main git push -u origin main
- **Working on Git**
- Write code → git add → git commit → git push
- Collaborate using branches & pull requests
- Regularly git pull to avoid conflicts

- https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners

Task 4

- Deploy the completed tasks on git (version control) and as well as create a project repository on git to do your final project in C++ using Vim.

Mega Project using all concepts explored until now

- **Project Title: FPGA-Based Image Processing Pipeline Simulator (C++ Hardware Accelerator Model)**
- **Project Goal**
- Simulate a simplified **hardware image-processing pipeline** (like in FPGA/SoC systems) using C++.
  The design mimics what hardware design companies do (Xilinx, Intel, NVIDIA embedded vision teams).

Features of the Project

Feature	Description
Image frame buffer	Use **pointers** + dynamic memory
Modules as structs	struct Filter, struct Buffer, struct ConvKernel, etc.
Pipeline stages	Load → Convert → Filter → Edge detect → Save
Custom data types	typedef, struct pixel { uint8_t r,g,b; };
Memory management	malloc/free or new/delete
Preprocessor	#ifdef USE_FIXED_POINT, #endif
Makefile	Build pipeline modules + main executable
Multiple files	main.cpp, pipeline.cpp, filters.cpp, etc.
OOP concepts	Polymorphism for filters (Base Filter class)
Hardware modeling mindset	Simulate registers, processing cores

Modules to add (hint: header files)

Module	Function
Frame Reader	Reads input image into pixel buffer
Color Converter	RGB → Grayscale
Smoothing Filter	(3x3 average filter)
Convolution Engine	Sharpening / Gaussian / Sobel edge detect
Memory Buffer / FIFO	Use pointers + dynamic arrays
Output Writer	Save final image
Debug Logger	#ifdef DEBUG show internal registers

Concepts to use

Concept	How it is used
struct	Pixel, buffer, module definitions
Pointers	Frame buffers, memory blocks
#ifdef / #endif	Enable FPGA-mode / Debug-mode / Fixed-point mode
Function pointers	Plug filters dynamically
Dynamic memory	Input/output frame buffers
Classes & Inheritance	Filter base class → SobelFilter, BlurFilter
Namespaces	hardware::pipeline
Makefile	Build and link multiple .cpp files
Templates	Template kernel size, pixel type

Possible extensions

Extension	Meaning
Add DMA simulation	Hardware style data streaming
Add threads	Multi-core processing
Add fixed-point simulation	Hardware datatype modeling
Integrate with OpenCV	Verify output
Add command shell	Control pipeline via CLI

Bonus Learning - Imp

- What are kernels? How can data and important things be defined there ? How to link kernels with system C / C++ project ?
- How to update something in kernels and then build and compile system c/C++ project.

THANK YOU