# Object Oriented Program Design

A comprehensive cheat sheet for C++ programming, Object-Oriented Programming (OOP), file I/O, classes, and its application in embedded systems, including communication protocols, OpenCV library, image and color processing, networking with sockets, 3D and PCB design, dynamic data management, version control, timing, threads, and Linux-Pi integration.

## C++ Fundamentals & OOP

### Basic Syntax & Data Types

**Variables:**

```
int age = 30;
float pi = 3.14;
char initial = 'J';
bool is_valid = true;
```

**Operators:**

`+` (Addition), `-` (Subtraction)
`*` (Multiplication), `/` (Division)
`%` (Modulo), `=` (Assignment)
`==` (Equal), `!=` (Not equal)

**Control Structures:**

```
if (condition) { ... } else { ... }
for (int i = 0; i < 10; ++i) { ... }
while (condition) { ... }
switch (variable) { case value: ... break; }
```

**Functions:**

```
int add(int a, int b) {
  return a + b;
}
```

**Pointers and References:**

```
int x = 10;
int *ptr = &x;
// Pointer to x
int &ref = x;
// Reference to x
```

**Memory Management:**

`new` (allocate memory), `delete` (free memory)

```
int *arr = new int[10];
delete[] arr;
```

### Object-Oriented Programming (OOP)

**Classes and Objects:**

```
class Dog {
public:
  std::string breed;
  void bark() { ... }
};

Dog myDog;
myDog.breed = "Labrador";
```

**Encapsulation:**
Bundling data and methods that operate on that data within a class.
Use access specifiers (`private`, `protected`, `public`).

**Inheritance:**
Creating new classes from existing ones.

```
class GermanShepherd : public Dog { ... };
```

**Polymorphism:**
The ability of an object to take on many forms. (e.g., Function Overloading, Virtual Functions)

**Abstraction:**
Showing only essential attributes and hiding unnecessary information.

**Constructors and Destructors:**

```
Dog() { ... }  // Constructor
~Dog() { ... } // Destructor
```

### File I/O

**Reading from a File:**

```
#include <fstream>

std::ifstream file("example.txt");
std::string line;
if (file.is_open()) {
  while (getline(file, line)) {
    std::cout << line << std::endl;
  }
  file.close();
}
```

**Writing to a File:**

```
#include <fstream>

std::ofstream file("example.txt");
if (file.is_open()) {
  file << "Hello, file!\n";
  file.close();
}
```

**File Modes:**
`std::ios::in` (input), `std::ios::out` (output)
`std::ios::app` (append), `std::ios::binary` (binary mode)

## Embedded Systems & Communication

### Embedded Systems Fundamentals

**Microcontrollers:**
Small, self-contained computers on a single chip.
(e.g., Arduino, STM32)

**Real-Time Operating Systems (RTOS):**
Operating systems designed for real-time applications.
(e.g., FreeRTOS)

**Memory Types:**
`RAM` (Random Access Memory), `ROM` (Read-Only Memory)
`Flash Memory`, `EEPROM`

**Peripherals:**
`GPIO` (General Purpose Input/Output)
`UART`, `SPI`, `I2C` (Communication Interfaces)
`ADC/DAC` (Analog-to-Digital/Digital-to-Analog Converters)

**Interrupts:**
Hardware or software signals that cause the processor to suspend its current execution and handle a specific event.

**Timers:**
Used for timing events, generating PWM signals, etc.

### Embedded Communication Protocols

**UART (Universal Asynchronous Receiver/Transmitter):**
Simple serial communication protocol.
Used for point-to-point communication.

**SPI (Serial Peripheral Interface):**
Synchronous serial communication protocol.
Used for short-distance, high-speed communication.

**I2C (Inter-Integrated Circuit):**
Two-wire serial communication protocol.
Used for connecting multiple devices to a single bus.

**CAN (Controller Area Network):**
Robust communication protocol for automotive and industrial applications.

**Bluetooth:**
Wireless communication protocol for short-range communication.

**WiFi:**
Wireless communication protocol for longer-range communication.

## Timing and Threads

**Timing:**

```
#include <chrono>
#include <thread>

auto start =
std::chrono::high_resolution_clock::now(
);
std::this_thread::sleep_for(std::chrono:
:milliseconds(100));
auto end =
std::chrono::high_resolution_clock::now(
);
auto duration =
std::chrono::duration_cast<std::chrono::
microseconds>(end - start);
```

**Threads:**

```
#include <thread>

void task() { ... }
std::thread t(task);
t.join(); // Wait for thread to finish
```

**Mutexes:**

Used to protect shared resources from concurrent access.

```
#include <mutex>

std::mutex mtx;
mtx.lock();
// Access shared resource
mtx.unlock();
```

# Image Processing & OpenCV

## OpenCV Basics

**Loading an Image:**

```
#include <opencv2/opencv.hpp>

cv::Mat image = cv::imread("image.jpg");
if (image.empty()) { ... }
```

**Displaying an Image:**

```
cv::imshow("Image", image);
cv::waitKey(0); // Wait for a key press
```

**Image Data Structure:**
`cv::Mat` (Matrix) - Stores image data. Access pixel values using `image.at<cv::Vec3b>(row, col)[channel]`

**Basic Image Operations:**
Resizing, Cropping, Color Conversion (e.g., BGR to Grayscale)

**Saving an Image:**

```
cv::imwrite("output.jpg", image);
```

**Video Capture**

```
cv::VideoCapture cap(0); // open default camera
if(!cap.isOpened()){return -1;}

cv::Mat frame;
cap >> frame; // get a new frame from camera
```

## Image and Color Processing

**Image Filtering:**
Blurring, Sharpening, Edge Detection (e.g., `cv::GaussianBlur`, `cv::Sobel`)

**Color Spaces:**
`BGR`, `Grayscale`, `HSV` (Hue, Saturation, Value)
Converting between color spaces using `cv::cvtColor`

**Thresholding:**
Converting an image to binary using a threshold value. (e.g., `cv::threshold`)

**Color Detection:**
Isolating specific colors in an image by thresholding in HSV color space.

**Morphological Operations:**
Erosion, Dilation, Opening, Closing (e.g., `cv::erode`, `cv::dilate`)

**Histogram Equalization**

```
cv::equalizeHist( image, hist_equalized_image );
```

## Advanced Image Processing

**Feature Detection:**
Detecting key points and features in an image. (e.g., `cv::SIFT`, `cv::ORB`, `cv::HoughLines`)

**Object Detection:**
Identifying objects in an image using pre-trained models.
(e.g., `cv::CascadeClassifier` for face detection, `YOLO`, `SSD` using DNN module)

**Image Segmentation:**
Dividing an image into multiple segments or regions.
(e.g., Watershed Algorithm, K-Means Clustering)

# Networking, Data, and Version Control

## Network Sockets

**Creating a Socket:**

```
#include
<sys/socket.h>
#include
<netinet/in.h>

int sockfd =
socket(AF_INET,
SOCK_STREAM, 0);
```

**Binding a Socket:**

```
sockaddr_in addr;
addr.sin_family =
AF_INET;
addr.sin_port =
htons(8080);
addr.sin_addr.s_addr
= INADDR_ANY;
bind(sockfd,
(sockaddr*)&addr,
sizeof(addr));
```

**Listening for Connections:**

```
listen(sockfd,
5); // Max 5
pending
connections
```

**Accepting a Connection:**

```
sockaddr_in
client_addr;
socklen_t client_len
=
sizeof(client_addr);
int client_sockfd =
accept(sockfd,
(sockaddr*)&client_a
ddr, &client_len);
```

**Sending and Receiving Data:**

```
send(client_sock
fd, buffer,
length, 0);
recv(client_sock
fd, buffer,
length, 0);
```

**Closing a Socket:**

```
close(sockfd);
```

## Dynamic Data and Memory Management

**Dynamic Arrays:**

```
int *arr = new
int[size];
// ...
delete[] arr;
```

**Linked Lists:**
Dynamic data structure for storing elements in a sequence. Requires manual memory management.

**Smart Pointers:**
`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`
Automatically manage memory and prevent memory leaks.

**Memory Leaks:**
Occur when dynamically allocated memory is not properly deallocated. Use smart pointers or manual memory management carefully.

**RAII (Resource Acquisition Is Initialization):**
A programming idiom where resources are acquired during object construction and released during object destruction.

**Placement new**

```
#include <new>

void* buffer =
malloc(sizeof(M
yObject));
MyObject* obj =
new (buffer)
MyObject();
```

## Version Control with Git

**Basic Git Commands:**
`git init` (initialize a new repository)
`git clone <url>` (clone an existing repository)
`git add <file>` (stage changes)
`git commit -m "message"` (commit changes)

**Branching and Merging:**
`git branch <branch_name>` (create a new branch)
`git checkout <branch_name>` (switch to a branch)
`git merge <branch_name>` (merge a branch into the current branch)

**Remote Repositories:**
`git remote add origin <url>` (add a remote repository)
`git push origin <branch_name>` (push changes to remote repository)
`git pull origin <branch_name>` (pull changes from remote repository)