📘 Principles of Object-Oriented Programming (OOP)

1. Software Crisis ⚠️

- 1960s–80s: Software projects → too big, too costly, late, buggy.
- Problems: Poor modularity, low code reuse, hard maintenance.

2. Software Evolution 🔁

- Software must **change over time**: bug fixes, upgrades, user needs.
- OOP helps make evolution easier (due to modular classes & reuse).

3. Procedure-Oriented Programming (POP) 🔧

- Focus on **functions**.
- Data is **global** and shared.
- Difficult for complex systems.

4. Object-Oriented Programming Paradigm 🧠

- Combines **data + functions** into **objects**.
- Models real-world entities.

5. Basic OOP Concepts 🏗️

- **Class**: Blueprint (defines data + functions).
- **Object**: Instance of class.
- **Encapsulation** 🔒: Hiding details, exposing only what's needed.

- **Abstraction** 🎭: Showing only *essential features*.
- **Inheritance** 📥: Reuse code from another class.
- **Polymorphism** 🔁: Same interface, many forms.
- **Message Passing** ✉️: Objects talk to each other via methods.

6. Benefits of OOP 🌟

- Code reuse, modular, secure, easy to maintain, real-world mapping.

7. Object-Oriented Languages 🧾

- **C++**, Java, Python, C#, Ruby, Smalltalk.

8. Applications 💼

- Games, GUIs, simulations, enterprise software, embedded systems.

🟦 Beginning with C++

1. What is C++? 🤔

- Developed by **Bjarne Stroustrup**.
- Extension of C (C with classes).
- Multi-paradigm (procedural + OOP + generic).

2. Applications 🚀

- Operating systems, game engines, database systems, compilers, real-time apps.

3. Simple C++ Program 👋

```
#include <iostream>
using namespace std;
```

```
int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

4. More C++ Statements 📝

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;         // declaration + initialization
    double pi = 3.14;   // floating point
    char grade = 'A';   // character

    cout << "Value of a: " << a << endl;
    cout << "Value of pi: " << pi << endl;
    cout << "Grade: " << grade << endl;

    return 0;
}
```

5. Example with Class 🏷️

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
    }

    // Parameterized constructor
    Person(string n, int a) {
        name = n;
        age = a;
    }
```

```
    void greet() {
        cout << "Hi, I'm " << name
             << " and I'm " << age
             << " years old." << endl;
    }
};

int main() {
    Person p1;              // calls default constructor
    Person p2("Alice", 25); // calls parameterized constructor

    p1.greet();
    p2.greet();

    return 0;
}
```

6. Structure of C++ Program 🧱

- Header files → `#include`
- Namespace → `using namespace std;`
- Class/Functions → user-defined code
- `main()` function → entry point

7. Creating Source File 💾

- Save as `program.cpp`.

8. Compiling and Linking ⚙️

- Compile: `g++ program.cpp -o program`
- Run: `./program` (Linux/macOS) or `program.exe` (Windows).

🔤 Tokens, Expressions and Control Structures

1. Tokens

- Smallest elements: **keywords**, **identifiers**, **constants**, **operators**, **punctuators**.

2. Keywords 🔑

- Reserved words like `int`, `for`, `class`, `virtual`.

3. Identifiers 🏷️

- Names for variables, functions, classes. Must start with letter/underscore.

4. Constants 📌

```
const double PI = 3.14159;
```

5. Data Types 📦

- Basic: `int`, `char`, `float`, `double`, `bool`.
- User-defined: `class`, `struct`, `enum`.
- Derived: arrays, pointers, references.

6. Storage Classes 🗂️

- `auto`, `static`, `extern`, `mutable`.

```
static int count = 0;  // retains value
```

7. Reference Variables 🔗

```
int x = 10;
int &ref = x;
ref = 20;   // changes x
```

8. Operators ➕➖

- Arithmetic, relational, logical, bitwise, assignment, increment/decrement.

9. Scope Resolution :: 🔎

```
int a = 10;
int main() {
    int a = 20;
    cout << ::a;  // prints global a = 10
}
```

10. Memory Management 🧹

```
int* p = new int(5);
delete p;
```

11. Manipulators 🎛️

```
#include <iomanip>
cout << fixed << setprecision(2) << 3.14159;
```

12. Operator Overloading Example (Expanded) ⚡

```
#include <iostream>
using namespace std;

class Complex {
private:
    double r; // real
    double i; // imaginary

public:
    Complex(double rr = 0.0, double ii = 0.0) {
        r = rr;
        i = ii;
    }

    Complex operator+(const Complex &other) const {
        Complex temp;
        temp.r = r + other.r;
        temp.i = i + other.i;
        return temp;
    }

    void display() {
        cout << r << " + " << i << "i" << endl;
    }
};
```

```cpp
int main() {
    Complex c1(3, 4);
    Complex c2(1, 2);
    Complex sum = c1 + c2;

    sum.display();

    return 0;
}
```

🔁 Functions in C++

1. Prototype 📜

```cpp
int add(int a, int b);
```

2. Call by Reference 🔗

```cpp
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

3. Inline ⚡

```cpp
inline int square(int x) {
    return x * x;
}
```

4. Default Arguments 🎯

```cpp
void greet(string name = "Guest") {
    cout << "Hello, " << name << endl;
}
```

5. Const Arguments 🛡️

```cpp
void print(const string &s) {
    cout << s << endl;
}
```

6. Recursion 🔁

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

7. Function Overloading 🧩

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

8. Friend Function 🤝

```
class Box {
private:
    int value;
public:
    Box(int v) { value = v; }
    friend void show(Box b);
};

void show(Box b) {
    cout << "Value: " << b.value << endl;
}
```

9. Virtual Functions 🪄

```
class Base {
public:
    virtual void display() {
        cout << "Base class" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived class" << endl;
    }
};
```

10. Math Functions ➗

```
#include <cmath>
cout << sqrt(16);   // 4
```

```
cout << pow(2, 5);
```

1️⃣ **Classes and Objects**
2️⃣ **Constructors and Destructors**
3️⃣ **Operator Overloading and Type Conversions**

I'll prepare them **step by step** so nothing is missed. Here's the plan:

🔹 1. Classes and Objects

📌 Introduction

- A **class** is a blueprint (template) for creating objects.
- An **object** is an instance of a class.
- Classes combine **data (attributes)** + **functions (methods)** = **Encapsulation** 🎁

📌 C Structure Revisited

- In **C language**, `struct` groups data, but no functions inside.
- In **C++**, `class` allows **both data + functions**.

```
#include <iostream>
using namespace std;

struct Student {
    int id;
    char name[20];
};
```

📌 Specifying a Class

```
class Student {
    int id;        // data member
    string name;   // data member

public:
```

```
    void setData(int i, string n) {
        id = i;
        name = n;
    }

    void display() {
        cout << "ID: " << id << ", Name: " << name << endl;
    }
};
```

📌 Defining Member Functions

- **Inside class** (inline automatically).
- **Outside class** using `::` (scope resolution).

```
class Student {
    int id;
    string name;

public:
    void setData(int i, string n);
    void display();
};

void Student::setData(int i, string n) {
    id = i;
    name = n;
}

void Student::display() {
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

📌 Making an Outside Function Inline

```
inline void Student::display() {
    cout << "ID: " << id << ", Name: " << name << endl;
}
```

📌 Nesting of Member Functions

```
class Number {
    int x;

public:
    void read() {
        cout << "Enter number: ";
        cin >> x;
    }

    void displaySquare() {
        cout << "Square = " << square() << endl;
    }

private:
    int square() {
        return x * x;
    }
};
```

📌 Private Member Functions

- Functions declared `private` can be used **only inside the class**.
- Example above: `square()` is private.

📌 Arrays Within a Class

```
class Marks {
    int arr[5];

public:
    void input() {
        cout << "Enter 5 marks: ";
        for (int i = 0; i < 5; i++) {
            cin >> arr[i];
        }
    }

    void display() {
        for (int i = 0; i < 5; i++) {
            cout << arr[i] << " ";
```

```
        }
        cout << endl;
    }
};
```

📌 Memory Allocation for Objects

- Each object gets **separate memory for data members**.
- Member functions are **shared** by all objects (not duplicated).

📌 Static Data Members

```
class Counter {
    static int count;

public:
    Counter() {
        count++;
    }

    void showCount() {
        cout << "Count = " << count << endl;
    }
};

int Counter::count = 0;
```

📌 Static Member Functions

```
class Demo {
    static int val;

public:
    static void setVal(int v) {
        val = v;
    }

    static void showVal() {
        cout << "Value = " << val << endl;
    }
```

```
};

int Demo::val = 0;
```

📌 Arrays of Objects

```
class Student {
    int id;
    string name;

public:
    void setData(int i, string n) {
        id = i;
        name = n;
    }

    void display() {
        cout << id << " " << name << endl;
    }
};

int main() {
    Student s[3];
    s[0].setData(1, "A");
    s[1].setData(2, "B");
    s[2].setData(3, "C");

    for (int i = 0; i < 3; i++) {
        s[i].display();
    }
}
```

📌 Object as Function Arguments

```
class Number {
    int x;

public:
    void setX(int a) {
        x = a;
    }

    void add(Number n1, Number n2) {
        x = n1.x + n2.x;
```

```cpp
    }

    void show() {
        cout << "Sum = " << x << endl;
    }
};
```

📌 Friendly Functions

```cpp
class Box {
    int length;

public:
    Box(int l) {
        length = l;
    }

    friend void showLength(Box b);
};

void showLength(Box b) {
    cout << "Length = " << b.length << endl;
}
```

📌 Returning Objects

```cpp
class Complex {
    int r, i;

public:
    void set(int x, int y) {
        r = x;
        i = y;
    }

    Complex add(Complex c) {
        Complex temp;
        temp.r = r + c.r;
        temp.i = i + c.i;
        return temp;
    }

    void display() {
        cout << r << " + " << i << "i" << endl;
```

```
    }
};
```

📌 Const Member Functions

```
class Demo {
    int x;

public:
    Demo(int a) {
        x = a;
    }

    void show() const {
        cout << "Value = " << x << endl;
    }
};
```

📌 Pointers to Members

```
class Demo {
public:
    int x;

    void display() {
        cout << "x = " << x << endl;
    }
};

int main() {
    Demo d;
    d.x = 10;

    int Demo::*ptr = &Demo::x;
    void (Demo::*fptr)() = &Demo::display;

    cout << "Value using pointer = " << d.*ptr << endl;
    (d.*fptr)();
}
```

🔹 1. Inheritance: Extending Classes

📌 Introduction

- **Inheritance** = process of creating a new class (derived class) from an existing class (base class).
- Promotes **code reusability** and **extensibility**.
- Syntax:

```
class Derived : access Base {
    // new members
};
```

📌 Defining Derived Classes

```
class Base {
public:
    void displayBase() {
        cout << "Base class function" << endl;
    }
};

class Derived : public Base {
public:
    void displayDerived() {
        cout << "Derived class function" << endl;
    }
};
```

📌 Single Inheritance

```
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
```

📌 Making Private Members Inheritable

- Private members are **not directly accessible** in derived class.
- Use `protected` in base → accessible in derived.

```
class Base {
protected:
    int x;
};

class Derived : public Base {
public:
    void setX(int a) {
        x = a;    // accessible because protected
    }

    void showX() {
        cout << "x = " << x << endl;
    }
};
```

📌 Multilevel Inheritance

```
class A {
public:
    void displayA() {
        cout << "Class A" << endl;
    }
};

class B : public A {
public:
    void displayB() {
        cout << "Class B" << endl;
    }
};

class C : public B {
public:
    void displayC() {
```

```
        cout << "Class C" << endl;
    }
};
```

📌 Multiple Inheritance

```
class A {
public:
    void displayA() {
        cout << "Class A" << endl;
    }
};

class B {
public:
    void displayB() {
        cout << "Class B" << endl;
    }
};

class C : public A, public B {
public:
    void displayC() {
        cout << "Class C" << endl;
    }
};
```

📌 Hierarchical Inheritance

```
class A {
public:
    void displayA() {
        cout << "Class A" << endl;
    }
};

class B : public A {
public:
    void displayB() {
        cout << "Class B" << endl;
    }
};

class C : public A {
```

```
public:
    void displayC() {
        cout << "Class C" << endl;
    }
};
```

📌 Virtual Base Classes (Diamond Problem)

```
class A {
public:
    void show() {
        cout << "Class A" << endl;
    }
};

class B : virtual public A { };
class C : virtual public A { };

class D : public B, public C { };
```

👉 Without `virtual`, `D` would inherit two copies of `A`.

📌 Abstract Classes

- A class with at least one **pure virtual function**.

```
class Shape {
public:
    virtual void draw() = 0;   // pure virtual
};

class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing Circle" << endl;
    }
};
```

📌 Constructors in Derived Classes

```
class Base {
```

```
public:
    Base() {
        cout << "Base Constructor" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor" << endl;
    }
};
```

📌 Member Classes (Nesting of Classes)

```
class Outer {
public:
    class Inner {
    public:
        void show() {
            cout << "Inner class function" << endl;
        }
    };
};
```

🔹 2. Pointers, Virtual Functions and Polymorphism

📌 Introduction

- **Pointer** → stores address of variable/object.
- **Polymorphism** → "one name, many forms" (function overriding, virtual functions).

📌 Pointers

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

```
cout << "Value = " << *ptr << endl;
```

📌 Pointers to Objects

```
class Demo {
    int x;

public:
    void setX(int a) {
        x = a;
    }

    void show() {
        cout << "x = " << x << endl;
    }
};

int main() {
    Demo d;
    Demo *ptr = &d;
    ptr->setX(20);
    ptr->show();
}
```

📌 This Pointer

```
class Demo {
    int x;

public:
    void setX(int x) {
        this->x = x;  // differentiates between local and member
    }

    void show() {
        cout << "x = " << x << endl;
    }
};
```

📌 Polymorphism

- **Compile-time**: Function overloading, Operator overloading.
- **Run-time**: Virtual functions.

📌 Pointers to Derived Classes

```
class Base {
public:
    void show() {
        cout << "Base class" << endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class" << endl;
    }
};

int main() {
    Base *ptr;
    Derived d;
    ptr = &d;
    ptr->show();   // Base version called (without virtual)
}
```

📌 Virtual Functions

```
class Base {
public:
    virtual void show() {
        cout << "Base class" << endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class" << endl;
    }
};
```

```cpp
int main() {
    Base *ptr;
    Derived d;
    ptr = &d;
    ptr->show();   // Derived version called
}
```

📌 Pure Virtual Functions

```cpp
class Shape {
public:
    virtual void draw() = 0;
};

class Square : public Shape {
public:
    void draw() {
        cout << "Drawing Square" << endl;
    }
};
```

📌 Virtual Constructors & Destructors

```cpp
class Base {
public:
    Base() {
        cout << "Base Constructor" << endl;
    }

    virtual ~Base() {
        cout << "Base Destructor" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor" << endl;
    }

    ~Derived() {
        cout << "Derived Destructor" << endl;
    }
};
```

🔹 1. Managing Console I/O Operations

📌 Introduction

- In C++, all input/output is done using **streams** (flow of data).
- **cin** → input stream
- **cout** → output stream
- **cerr** → standard error (unbuffered)
- **clog** → standard error (buffered)

📌 C++ Streams

- A **stream** = sequence of bytes flowing between program and device (keyboard, screen, file).
- Example:

```
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter a number: ";
    cin >> x;
    cout << "You entered: " << x << endl;
    return 0;
}
```

📌 C++ Stream Classes

- Important classes are in `<iostream>` header:
  - `istream` → input stream (for cin)
  - `ostream` → output stream (for cout, cerr, clog)

- `iostream` → for both input & output

📌 Unformatted I/O Operations

- Character-by-character I/O using `get()`, `put()`, `getline()`.

```
#include <iostream>
using namespace std;

int main() {
    char ch;
    cout << "Enter a character: ";
    ch = cin.get();    // unformatted input
    cout.put(ch);      // unformatted output
    return 0;
}
```

📌 Formatted Console I/O Operations

- Use `cin` and `cout` with **formatting**.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    int num = 255;

    cout << "Decimal: " << dec << num << endl;
    cout << "Hexadecimal: " << hex << num << endl;
    cout << "Octal: " << oct << num << endl;

    return 0;
}
```

📌 Managing Output with Manipulators

- Manipulators are in `<iomanip>`.
- Examples: `setw()`, `setprecision()`, `setfill()`, `left`, `right`.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    double pi = 3.14159265;

    cout << setw(10) << setfill('*') << 123 << endl;
    cout << fixed << setprecision(3) << pi << endl;

    return 0;
}
```

🔹 2. Working with Files

📌 Introduction

- File handling in C++ uses `<fstream>`.
- Streams:
  - `ifstream` → input (read file)
  - `ofstream` → output (write file)
  - `fstream` → both input/output

📌 Classes for File Stream Operations

- `ifstream` → derived from `istream`.
- `ofstream` → derived from `ostream`.
- `fstream` → derived from `iostream`.

📌 Opening and Closing Files

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream fout;
    fout.open("test.txt");   // open file for writing
    fout << "Hello File!" << endl;
    fout.close();

    ifstream fin;
    fin.open("test.txt");    // open file for reading
    string line;
    getline(fin, line);
    cout << "File contains: " << line << endl;
    fin.close();

    return 0;
}
```

📌 Detecting End-of-File

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ifstream fin("test.txt");
    string line;

    while (!fin.eof()) {
        getline(fin, line);
        cout << line << endl;
    }

    fin.close();
    return 0;
}
```

🔹 3. Exception Handling

📌 Introduction

- **Exception** = runtime error (like divide by zero, file not found).
- Exception handling prevents program crash.

📌 Basics of Exception Handling

- Keywords:
  - `try` → block of risky code
  - `throw` → raises an exception
  - `catch` → handles exception

📌 Exception Handling Mechanism

```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;

    try {
        if (b == 0) {
            throw "Division by zero!";
        }
        cout << "Result = " << a / b << endl;
    }
    catch (const char* msg) {
        cout << "Error: " << msg << endl;
    }

    return 0;
}
```

📌 Throwing Mechanism

- `throw` is used inside `try` when error occurs.
- Example: `throw 1;`, `throw "error";`, `throw exceptionObject;`

📌 Catching Mechanism

- Different catch blocks can handle different types.

```
try {
    throw 10;
}
catch (int x) {
    cout << "Caught integer: " << x << endl;
}
catch (...) {
    cout << "Caught unknown exception" << endl;
}
```

✅ Covered:

1. **Managing Console I/O Operations** (streams, stream classes, unformatted, formatted, manipulators)
2. **Working with Files** (intro, classes, open/close, EOF)
3. **Exception Handling** (intro, basics, try-throw-catch, mechanism)

🔹 Templates in C++

📌 Introduction

- **Template** = way to write **generic code** (works for any datatype).
- Helps in **code reusability** and **type-safety**.

- Two main types:
    1. **Function Templates**
    2. **Class Templates**

🟢 1. Class Templates

📌 Syntax

```
template <class T>
class MyClass {
    T data;

public:
    MyClass(T d) {
        data = d;
    }

    void show() {
        cout << "Data = " << data << endl;
    }
};
```

📌 Example

```
#include <iostream>
using namespace std;

template <class T>
class Box {
    T value;

public:
    Box(T v) {
        value = v;
    }

    void display() {
        cout << "Value = " << value << endl;
    }
};
```

```
int main() {
    Box<int> b1(10);
    Box<double> b2(3.14);
    Box<string> b3("Hello");

    b1.display();
    b2.display();
    b3.display();

    return 0;
}
```

🟢 2. Class Templates with Multiple Parameters

```
#include <iostream>
using namespace std;

template <class T1, class T2>
class Pair {
    T1 first;
    T2 second;

public:
    Pair(T1 f, T2 s) {
        first = f;
        second = s;
    }

    void display() {
        cout << "First = " << first << ", Second = " << second << endl;
    }
};

int main() {
    Pair<int, double> p1(10, 20.5);
    Pair<string, int> p2("Age", 25);

    p1.display();
    p2.display();

    return 0;
}
```

🟢 3. Function Templates

📌 Syntax

```
template <class T>
T add(T a, T b) {
    return a + b;
}
```

📌 Example

```
#include <iostream>
using namespace std;

template <class T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maximum(10, 20) << endl;
    cout << maximum(3.5, 2.8) << endl;
    cout << maximum('a', 'z') << endl;

    return 0;
}
```

🟢 4. Function Templates with Multiple Parameters

```
#include <iostream>
using namespace std;

template <class T1, class T2>
void show(T1 a, T2 b) {
    cout << "A = " << a << ", B = " << b << endl;
}

int main() {
    show(10, 20.5);
    show("Age", 25);

    return 0;
}
```

🟢 5. Overloading of Template Functions

- A template function can coexist with a normal function or another template.

```
#include <iostream>
using namespace std;

template <class T>
void display(T x) {
    cout << "Template version: " << x << endl;
}

// Normal function (overloaded)
void display(int x) {
    cout << "Normal int version: " << x << endl;
}

int main() {
    display(10);        // calls normal function
    display(3.14);      // calls template version
    display("Hello");   // calls template version

    return 0;
}
```

🟢 6. Member Function Templates

- A class can have **only one template type**, but some member functions can themselves be templates.

```
#include <iostream>
using namespace std;

class Demo {
public:
    template <class T>
    void show(T x) {
        cout << "Value = " << x << endl;
    }
};
```

```
int main() {
    Demo d;
    d.show(100);
    d.show(12.34);
    d.show("Hello");

    return 0;
}
```

✅ Covered Templates in detail:

1. Introduction
2. Class Templates
3. Class Templates with Multiple Parameters
4. Function Templates
5. Function Templates with Multiple Parameters
6. Overloading of Template Functions
7. Member Function Templates