📘 C++ Notes

1️⃣ Principles of Object-Oriented Programming (OOP)

🔹 Software Crisis

- 1960s: Programs became **large & complex** 🏗️
- Problems: ❌ Cost, ❌ Time, ❌ Maintenance

🔹 Software Evolution

- Programs need **upgrades** 🔄
- Easy maintenance = Better software lifecycle ⚡

🔹 Procedure-Oriented Programming (POP)

- Focus 👉 **Functions**
- Data = **Global** (less secure 🔓)
- Example: `C language`

🔹 Object-Oriented Programming (OOP)

- Focus 👉 **Objects** (data + functions together)
- Data hidden 🔒
- Reusable & modular ✅

🔹 Basic OOP Concepts

- Class 🏫
- Object 🎭

- Encapsulation 📦
- Abstraction 🎭
- Inheritance 👪
- Polymorphism 🎭➡️🎭
- Message Passing ✉️

🔹 Benefits of OOP

- ✅ Reusability
- ✅ Security
- ✅ Scalability
- ✅ Maintainability

🔹 Applications

- Games 🎮
- Banking 💳
- Real-time systems ⏱️
- Simulation 🛫

2️⃣ Beginning with C++

🔹 What is C++?

- Developed by **Bjarne Stroustrup (1979)** 👨‍💻
- Extension of **C** ➕ OOP features

🔹 Applications

- Operating systems ⚙️
- Compilers 🔧
- Databases 💾
- Games 🎮

🔹 Structure of a C++ Program

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

// 👇 main function
int main() {
    cout << "Hello, C++ 🚀" << endl;
    return 0;
}
```

3️⃣ Tokens, Expressions & Control Structures

🔹 Tokens in C++

- Keywords 🔑 (`int`, `class`, `return`)
- Identifiers 🏷️ (user-defined names)
- Constants 📌 (`5`, `3.14`, `'A'`)
- Operators ➕➖✖️➗
- Punctuation `;` `,` `{` `}`

🔹 Data Types

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

🔹 Storage Classes

- `auto`, `register`, `static`, `extern`, `mutable`

🔹 Operators

- Arithmetic ➕➖✖️➗%
- Relational `< > == !=`
- Logical `&& || !`
- Scope Resolution `::`
- Memory Mgmt: `new`, `delete`

🔹 Control Structures

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

int main() {
    int n = 5;

    // if-else
    if (n % 2 == 0)
        cout << "Even ✅";
    else
        cout << "Odd ❌";

    // Loop
    cout << "\nNumbers: ";
    for (int i = 1; i <= n; i++) {
        cout << i << " ";
    }
}
```

4️⃣ Functions in C++

🔹 Example

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

```cpp
// Function Prototype
int add(int a, int b);

// Inline Function
inline int square(int x) { return x * x; }

// Call by Reference
void swapNums(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

// Main
int main() {
    cout << "Sum = " << add(3, 4) << endl;
    cout << "Square = " << square(5) << endl;

    int a = 10, b = 20;
    swapNums(a, b);
    cout << "After Swap: a=" << a << " b=" << b << endl;

    return 0;
}

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

🔹 Key Concepts

- Call by Reference 🔄
- Return by Reference ↩️
- Inline Functions ⚡ (faster execution)
- Default Arguments 🛠️
- Friend Functions 👥 (access private data)
- Virtual Functions 🎭 (runtime polymorphism)

- Recursion 🔁

📘 Classes and Objects in C++

---

1. 🔹 Introduction

- **Class** = blueprint 🏗️ for objects.
- **Object** = real entity (created from class).
- **Class = Data + Functions (together).**

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

class Student {
    string name;  // data member
    int age;

public:
    void setData(string n, int a) { // member function
        name = n; age = a;
    }
    void display() {
        cout << "Name: " << name << " 🎓 Age: " << age << endl;
    }
};

int main() {
    Student s1;  // object
    s1.setData("Harshith", 20);
    s1.display();
}
```

---

2. 🔹 C structure revisited vs Class

- **C struct** → members are public by default.

- **C++ class** → members are private by default.

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

struct StudentStruct {
    int age;
};

class StudentClass {
    int age;  // private by default
public:
    void setAge(int a) { age = a; }
    void display() { cout << "Age: " << age << endl; }
};
```

---

3. 🔹 Outside function inline

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

class Test {
public:
    inline void show();  // declaration
};

inline void Test::show() {   // defined outside but inline
    cout << "Inline Function ⚡" << endl;
}

int main() {
    Test t;
    t.show();
}
```

---

4. 🔹 Nesting of member functions

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

class Number {
    int x;
```

```
public:
    void read() {
        cout << "Enter a number: ";
        cin >> x;
    }
    void display() {
        if(isPositive())  // function inside another
            cout << "Positive ✅" << endl;
        else
            cout << "Negative ❌" << endl;
    }
    bool isPositive() { return x >= 0; }
};

int main() {
    Number n;
    n.read();
    n.display();
}
```

---

5. 🔹 Private member function

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

class Demo {
    void secret() { cout << "Secret 🔒 function\n"; }
public:
    void accessSecret() { secret(); }
};

int main() {
    Demo d;
    d.accessSecret();  // allowed ✅
}
```

---

6. 🔹 Arrays within a class

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

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] << " ";
    }
};

int main() {
    Marks m;
    m.input();
    m.display();
}
```

---

7. 🔹 Static data & Static member function

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

class Counter {
    static int count;  // shared by all objects
public:
    Counter() { count++; }
    static void showCount() {
        cout << "Objects created: " << count << endl;
    }
};
int Counter::count = 0;

int main() {
    Counter c1, c2, c3;
    Counter::showCount();
}
```

---

8. 🔹 Friend function

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

class Box {
    int width;
```

```
public:
    Box(int w) { width = w; }
    friend void showWidth(Box b);  // friend function
};

void showWidth(Box b) {
    cout << "Width = " << b.width << " 📦" << endl;
}

int main() {
    Box b(10);
    showWidth(b);
}
```

📘 Constructors & Destructors

---

1. 🔹 Simple constructor

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

class Student {
    string name;
public:
    Student(string n) {  // constructor
        name = n;
        cout << "Constructor called 🎉 for " << name << endl;
    }
    ~Student() { // destructor
        cout << "Destructor called ❌ for " << name << endl;
    }
};

int main() {
    Student s1("Harshith"), s2("Rahul");
}
```

---

2. 🔹 Parameterized constructor + Default arguments

```
#include <iostream>
```

```
using namespace std;

class Point {
    int x, y;
public:
    Point(int a=0, int b=0) { x=a; y=b; }
    void show() { cout << "(" << x << "," << y << ")" << endl; }
};

int main() {
    Point p1(2,3), p2(10), p3;
    p1.show(); p2.show(); p3.show();
}
```

---

3. 🔹 Copy constructor

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

class Demo {
    int x;
public:
    Demo(int a) { x=a; }
    Demo(const Demo &d) {  // copy constructor
        x = d.x;
    }
    void show() { cout << "x=" << x << endl; }
};

int main() {
    Demo d1(5);
    Demo d2(d1);  // copy constructor
    d1.show();
    d2.show();
}
```

📘 Operator Overloading

---

1. 🔹 Unary operator overloading (++ operator)

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

class Number {
    int x;
public:
    Number(int a=0) { x=a; }
    void operator++() { x++; } // overload ++
    void show() { cout << "x=" << x << endl; }
};

int main() {
    Number n(5);
    ++n;
    n.show();
}
```

---

2. 🔹 Binary operator overloading (+ operator)

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

class Complex {
    int real, imag;
public:
    Complex(int r=0, int i=0) { real=r; imag=i; }
    Complex operator+(Complex c) {   // binary overloading
        return Complex(real+c.real, imag+c.imag);
    }
    void show() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    Complex c1(2,3), c2(4,5), c3;
    c3 = c1 + c2;
    c3.show();
}
```

📘 Inheritance in C++

---

1. 🔹 Introduction

- **Inheritance** = mechanism to reuse code (base → derived).
- **Base Class** = parent 👨‍👦
- **Derived Class** = child class that extends parent.

---

2. 🔹 Defining Derived Classes

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

class Person {   // base
public:
    string name;
    void show() { cout << "Name: " << name << endl; }
};

class Student : public Person {   // derived
public:
    int roll;
    void display() {
        cout << "Roll: " << roll << endl;
    }
};

int main() {
    Student s;
    s.name = "Harshith";
    s.roll = 101;
    s.show();
    s.display();
}
```

---

3. 🔹 Single Inheritance

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

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

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

int main() {
    B obj;
    obj.displayA();
    obj.displayB();
}
```

---

4. 🔹 Making Private Members Inheritable

👉 Use **protected** instead of private.

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

class Base {
protected:
    int x;   // protected = like private but inheritable
public:
    void setX(int a) { x=a; }
};

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

int main() {
    Derived d;
    d.setX(10);
    d.show();
}
```

---

5. 🔹 Multilevel Inheritance

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

```
class A { public: void showA(){ cout<<"A"<<endl; } };
class B : public A { public: void showB(){ cout<<"B"<<endl; } };
class C : public B { public: void showC(){ cout<<"C"<<endl; } };

int main() {
    C obj;
    obj.showA();
    obj.showB();
    obj.showC();
}
```

---

6. 🔹 Multiple Inheritance

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

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

class C : public A, public B { };

int main() {
    C obj;
    obj.showA();
    obj.showB();
}
```

---

7. 🔹 Hierarchical Inheritance

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

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

int main() {
    B obj1;
    C obj2;
    obj1.showA();
    obj2.showA();
}
```

---

8. 🔹 Virtual Base Class (Diamond Problem 💎)

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

class A { public: int x; };
class B : virtual public A { };
class C : virtual public A { };
class D : public B, public C { };

int main() {
    D obj;
    obj.x = 10;   // only one copy ✅
    cout << "x=" << obj.x << endl;
}
```

---

9. 🔹 Abstract Class & Pure Virtual Function

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

class Shape {
public:
    virtual void area() = 0;  // pure virtual function
};

class Circle : public Shape {
public:
    void area() { cout << "Area of Circle 🔵" << endl; }
};

int main() {
    Shape* s;   // pointer to abstract class
    Circle c;
    s = &c;
    s->area();
}
```

---

10. 🔹 Constructor in Derived Class

```
#include <iostream>
```

```
using namespace std;

class Base {
public:
    Base() { cout << "Base constructor 🚀" << endl; }
};

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

int main() {
    Derived d;
}
```

---

11. 🔹 Nesting of Classes

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

class Outer {
public:
    class Inner {   // nested class
    public:
        void display() { cout << "Inner class 🎯" << endl; }
    };
};

int main() {
    Outer::Inner obj;
    obj.display();
}
```

📘 Pointers, Virtual Functions & Polymorphism

---

1. 🔹 Pointer to Object

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

```
class Demo {
    int x;
public:
    void set(int a){ x=a; }
    void show(){ cout << "x=" << x << endl; }
};

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

---

2. 🔹 this Pointer

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

class Demo {
    int x;
public:
    Demo(int x){ this->x = x; }  // this pointer
    void show(){ cout << "x=" << this->x << endl; }
};

int main() {
    Demo d(50);
    d.show();
}
```

---

3. 🔹 Polymorphism (Compile-time vs Run-time)

👉 **Run-time polymorphism** uses **virtual functions**.

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

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

```
};

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

int main() {
    Base* b;
    Derived d;
    b = &d;
    b->show();  // calls Derived (runtime binding)
}
```

---

4. 🔹 Pure Virtual Function (Abstract Class)

👉 Already shown above ✅

---

5. 🔹 Virtual Destructor

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

class Base {
public:
    virtual ~Base(){ cout<<"Base Destructor ❌"<<endl; }
};

class Derived : public Base {
public:
    ~Derived(){ cout<<"Derived Destructor ❌"<<endl; }
};

int main() {
    Base* b = new Derived();
    delete b;  // calls both destructors correctly ✅
}
```

📘 Managing Console I/O Operations

---

1. 🔹 Introduction

- **C++ I/O system** uses **streams** (flow of data).
- **Stream** = sequence of bytes (input/output).
- Types:
  - `cin` → standard input (keyboard ⌨️)
  - `cout` → standard output (screen 🖥️)
  - `cerr` → error output
  - `clog` → logging

---

2. 🔹 C++ Streams & Stream Classes

- All streams are implemented using classes in `<iostream>`.
- Hierarchy:
  - `ios` → base class
  - `istream`, `ostream` → input/output
  - `iostream` → both

---

3. 🔹 Unformatted I/O

👉 Functions like `get()`, `put()`, `getline()`, `ignore()`

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

int main() {
    char ch;
    cout << "Enter a character: ";
    cin.get(ch);       // unformatted input
```

```
    cout.put(ch);      // unformatted output
}
```

---

4. 🔹 Formatted I/O

👉 Use **insertion (<<)** and **extraction (>>)** operators

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

int main() {
    int age;
    string name;
    cout << "Enter name & age: ";
    cin >> name >> age;   // formatted input
    cout << "Name: " << name << " 🎓 Age: " << age << endl;
}
```

---

5. 🔹 Manipulators (in <iomanip>)

👉 Control formatting with `setw()`, `setprecision()`, `fixed`, etc.

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

int main() {
    double pi = 3.14159265;
    cout << "Default: " << pi << endl;
    cout << "Fixed: " << fixed << setprecision(2) << pi << endl;
    cout << "Width: " << setw(10) << pi << endl;
}
```

📘 Working with Files

---

1. 🔹 Introduction

- Files allow **permanent storage** 📂
- Streams:
    - `ifstream` → read
    - `ofstream` → write
    - `fstream` → both

---

2. 🔹 Opening & Closing Files

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

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

    ifstream fin("data.txt");   // open file for reading
    string line;
    getline(fin, line);
    cout << "Read: " << line << endl;
    fin.close();
}
```

---

3. 🔹 Detecting End-of-File (EOF)

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

int main() {
    ifstream fin("data.txt");
    char ch;
    while(fin.get(ch)) {   // loop until EOF
        cout << ch;
    }
    fin.close();
```

```
}
```

📘 Exception Handling

---

1. 🔹 Introduction

- Errors at **run-time** (divide by 0, file not found, etc.).
- Exception handling = detect + handle gracefully.

---

2. 🔹 Basics (try-catch)

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

int main() {
    int a, b;
    cout << "Enter 2 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;
    }
}
```

---

3. 🔹 Throwing & Catching Multiple Exceptions

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

int main() {
    try {
        int x;
```

```
        cout << "Enter value: ";
        cin >> x;
        if(x < 0) throw x;
        else if(x == 0) throw "Zero entered ❌";
        else cout << "Valid: " << x << endl;
    }
    catch(int n) { cout << "Negative number: " << n << endl; }
    catch(const char* msg) { cout << msg << endl; }
}
```

✅ That covers:

1. **Console I/O** (Streams, unformatted, formatted, manipulators)
2. **Files** (open, close, EOF)
3. **Exceptions** (try-throw-catch)

.

📘 Templates in C++

---

1. 🔹 Introduction

- Templates = **write once, use for any data type** 🪄
- Used for **generic programming**.
- Types:
    1. **Function Templates**
    2. **Class Templates**

---

2. 🔹 Function Template (Single Parameter)

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

template <typename T>   // template keyword
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add<int>(3, 4) << endl;     // int
    cout << add<double>(2.5, 3.1) << endl;  // double
}
```

👉 Compiler creates separate versions at compile-time.

---

3. 🔹 Function Template with Multiple Parameters

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

template <typename T1, typename T2>
void display(T1 a, T2 b) {
    cout << "a = " << a << " b = " << b << endl;
}

int main() {
    display<int, double>(5, 3.14);
    display<char, string>('A', "Hello");
}
```

---

4. 🔹 Overloading Template Functions

👉 A **normal function** + **template function** can coexist.

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

void show(int a) { cout << "Normal function: " << a << endl; }

template <typename T>
void show(T x) { cout << "Template function: " << x << endl; }
```

```
int main() {
    show(10);        // calls normal function
    show(3.14);      // calls template
    show('A');       // calls template
}
```

---

5. 🔹 Class Template (Single Parameter)

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

template <class T>
class Box {
    T value;
public:
    Box(T v) { value = v; }
    void show() { cout << "Value: " << value << endl; }
};

int main() {
    Box<int> b1(100);
    Box<string> b2("Hello Templates");
    b1.show();
    b2.show();
}
```

---

6. 🔹 Class Template with Multiple Parameters

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

template <class T1, class T2>
class Pair {
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) { first=a; second=b; }
    void show() { cout << "First: " << first << " Second: " << second << endl; }
};

int main() {
    Pair<int, double> p1(10, 3.14);
    Pair<string, char> p2("Hello", 'X');
```

```
    p1.show();
    p2.show();
}
```

---

7. 🔹 Member Function Template inside Class

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

class Printer {
public:
    template <typename T>
    void print(T data) {
        cout << "Printing: " << data << endl;
    }
};

int main() {
    Printer p;
    p.print<int>(100);
    p.print<string>("Generic Function inside Class");
}
```

✅ Covered:

1. Function templates
2. Multiple parameters
3. Overloading template functions
4. Class templates (single + multiple parameters)
5. Member function templates