1️⃣ C++ Fundamentals

🧮 Data Types, Sizes, and Ranges

✅ Detailed Theory

C++ provides a wide variety of data types, ranging from fixed-width integer types (`int32_t` , `uint64_t`), floating-point types (`float` , `double` , `long double`), extended types (`__int128` , `uint128_t`), and special-purpose types (`wchar_t` , `size_t` , `intptr_t`).

These types differ by:

- **Size (in bytes)**: platform and compiler-dependent
- **Signedness**: whether they can hold negative values
- **Storage class / type modifiers**: `signed` , `unsigned` , `short` , `long`

🔢 Type Size & Range Table

Data Type	Size (bytes)	Range
`short int`	2	-32,768 to 32,767
`unsigned short int`	2	0 to 65,535
`int`	4	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long int`	8	-2^63 to 2^63 - 1
`unsigned long int`	8	0 to 2^64 - 1
`long long int`	8	same as `long int`
`__int128` / `int128_t` *	16	-2^127 to 2^127-1
`uint128_t` *	16	0 to 2^128-1

Data Type	Size (bytes)	Range
`int256_t` *	32	(non-standard, used via libraries like Boost or custom SIMD)
`signed char`	1	-128 to 127
`unsigned char`	1	0 to 255
`float`	4	±3.4e±38 (~6-7 decimal digits)
`double`	8	±1.7e±308 (~15-16 decimal digits)
`long double`	12 / 16	Platform dependent
`wchar_t`	2 / 4	Platform dependent
`bool`	1	true / false
`size_t`	4/8	Platform dependent (unsigned type)
`ptrdiff_t`	4/8	Signed version of difference between pointers
`intmax_t` , `uintmax_t`	Platform max	
`uintptr_t`	Unsigned integer capable of holding pointer	

🔸 Note: Types like `__int128` , `uint128_t` , `int256_t` are not part of standard C++ but are supported by GCC/Clang or third-party libraries.

🧠 When to Use Which Data Type

✅ Guidelines

- Use `int` for general-purpose integer values.
- Use `size_t` for array indices and sizes.
- Use `int64_t` , `uint64_t` for platform-independent large integers.
- Use `float` , `double` for floating-point math. Prefer `double` for precision.
- Use `char` , `wchar_t` for character and Unicode handling.

💡 Interview Tip

Prefer `fixed-width types` (`int32_t` , `uint64_t`) in systems programming or cross-platform projects for predictability.

🧠 Advanced Deep Dives

🔍 Signed vs Unsigned

✅ **Detailed Theory**:

- **Signed integers** use **2's complement** representation.
- The MSB (Most Significant Bit) is the **sign bit**:
  - `0` means positive or zero
  - `1` means negative (in 2's complement)

💡 **Key Formula**:

- For `n` -bit signed integers:
  - Range: `-2^(n-1)` to `2^(n-1) - 1`
- For `n` -bit unsigned integers:
  - Range: `0` to `2^n - 1`

💻 **Example**:

```
 unsigned int u = UINT_MAX;
std::cout << u + 1 << "\n"; // Outputs 0 due to wrap-around
```

🧠 **Advanced**:

- Mixing signed and unsigned types in arithmetic may lead to **unexpected conversions** and bugs.
- Always enable warnings with `-Wsign-compare` in `g++` .

🔁 **Common Question**:
**Q:* _*Why is unsigned arithmetic defined to wrap around?*_
***A:** Because it allows predictable low-level behavior and matches how hardware works with bits.

🔍 Overflow Behavior

✅ **Detailed Theory**:

- **Signed integer overflow → undefined behavior**

- **Unsigned integer overflow → defined wrap-around**

💡 **Why This Matters**:

- Compilers like `g++` may optimize assuming overflow **does not happen**, leading to hard-to-find bugs.

💻 **Example**:

```
 int x = INT_MAX;
std::cout << x + 1 << "\n"; // UB: could crash or wrap
```

🧠 **Advanced**:

- Use compiler flags like `-fwrapv` or `-fsanitize=undefined` to detect overflows during testing.

🔁 **Common Question**:
Q:* ***_How do I detect signed overflow at runtime?_***
***A:** Use compiler sanitizers (`-fsanitize=signed-integer-overflow`).

🔍 **Alignment & Padding**

✅ **Detailed Theory**:

- **Alignment** means data must be stored at memory addresses divisible by a certain number (power of 2).
- **Padding** is inserted between members to satisfy alignment.

💡 **Why It Exists**:

- Misaligned accesses are **slower** or **illegal** on some architectures.

💻 **Example**:

```
struct A { char c; int x; };
```

This has **3 bytes padding** between `c` and `x` .

🧠 **Advanced**:

- Use `alignas(T)` to explicitly control alignment.
- Use `offsetof()` to inspect member layout.

📊 **Diagram**:

```
struct A {
  char c;   // 0
  padding   // 1-3
  int x;    // 4-7
}
sizeof(A) == 8
```

🔁 **Common Question**:
**Q:* _*How do I pack a structure without padding?*_
***A:** Use `#pragma pack(1)` or compiler-specific attributes like `__attribute__((packed))`.

🧠 More on Alignment, Padding, Offset

✅ Detailed Theory

- **Alignment**: Restriction that data be stored at memory addresses divisible by a power of two.
- **Padding**: Extra bytes inserted by the compiler to maintain alignment.
- `alignas`: Specifies desired alignment for a variable or type.
- `alignof`: Queries the required alignment.
- `offsetof`: Finds byte offset of struct member.

💻 Example

```
struct X {
    char a;   // 1 byte
    // 3 bytes padding
    int b;    // 4 bytes
};

std::cout << sizeof(X); // Outputs 8
```

📊 Diagram

```
Offset:
0   [char a]
1   [padding]
4   [int b]
8   [total size]
```

💡 Tips

- Use `#pragma pack(1)` to remove padding if needed (non-portable).
- Use `alignas(16)` when interfacing with SIMD instructions or hardware.

🔮 Extended Types (`__int128`)

✅ **Detailed Theory**:

- `__int128` is a **non-standard** 128-bit integer type.
- Supported in **GCC and Clang** on 64-bit platforms.

💡 **Limitations**:

- Not printable with `std::cout`
- No standard format specifier

💻 **Example**:

```
__int128 x = (__int128)1 << 100;
```

🧠 **Advanced**:

- Useful in:
  - **Cryptography**
  - **Big Integer Arithmetic**
  - **Simulating fixed-point math**

🔁 **Common Question**:
Q:* _***How to print*** `__int128` ***?***_
***A:** Convert to `std::string` manually or write digit-by-digit using a loop.

💻 Code Examples with Expected Output

```
 #include <iostream>
#include <climits>
#include <cstdint>

int main() {
    std::cout << "int: " << sizeof(int) << " bytes, Range: " << INT_MIN << " to " << INT_
    std::cout << "uint64_t: " << sizeof(uint64_t) << " bytes\n";

      int128 x = (  int128)1 << 100;
    std::cout << "128-bit integer (1 << 100) stored successfully (cannot print directly)\

    return 0;
}
```

Output:

```
 int: 4 bytes, Range: -2147483648 to 2147483647
uint64_t: 8 bytes
128-bit integer (1 << 100) stored successfully (cannot print directly)
```

🔍 Dry-run Walkthrough

- `sizeof(int)` returns 4 on most systems (32-bit).
- `INT_MIN` and `INT_MAX` are defined in `<climits>`.
- `__int128` can store extremely large integers but you can't `std::cout` them directly —requires manual conversion.
- Use loops or character buffers to print `__int128`.

📊 Diagram – Signed vs Unsigned Layout (8-bit)

```
[ signed char ]  (2's complement)
Range: -128 to 127

   10000000  => -128
   11111111  => -1
   00000000  => 0
   01111111  => +127

[ unsigned char ]
Range: 0 to 255

   00000000  => 0
   11111111  => 255
```

💡 Interview Tips and Gotchas

- Avoid mixing signed and unsigned types in comparisons (`int` vs `size_t`).
- `sizeof(char)` is always 1, but it doesn't mean 1 bit — it's 1 **byte** (8 bits).
- Don't assume `int` is always 4 bytes — use fixed-width types (`int32_t` , `uint64_t`) for portability.
- Using `__int128` in an interview? Clarify it's platform/compiler-specific.
- `size_t` is unsigned — subtracting it can result in large values if one operand is smaller.
- Always be explicit about overflow behavior in performance-critical code.

🔁 Common Interview Questions and Answers

Q1: _Why should we prefer fixed-width types like `int32_t` instead of `int`?_
A: Because `int` size may vary across platforms; fixed-width types ensure consistent memory layout, important for serialization, networking, and embedded.

Q2: _What happens when an unsigned int overflows?_
A: It wraps around (modulo arithmetic). For example, `UINT_MAX + 1 == 0`.

Q3: _What is `size_t` and why is it unsigned?_
A: It's the type returned by `sizeof()` and used for indexing/length. It's unsigned because size can't be negative.

Q4: _What is the difference between `intptr_t` and `uintptr_t`?_
A: `intptr_t` is a signed integer type that can store a pointer, `uintptr_t` is its unsigned counterpart.

Q5: _What does `alignof()` return?_
A: It returns the alignment requirement of a type (e.g., 4 for `int`, 8 for `double`).

🧠 References vs Pointers

✅ Detailed Theory

- **Pointer**: Stores the memory address of a variable.
- **Reference**: An alias for another variable; must be initialized during declaration.

Feature	Pointer	Reference
Can be null?	Yes	No
Reassignable?	Yes	No
Syntax	`int* p`	`int& r`
Dereferencing	`*p`	`r` (no special symbol)

💡 Interview Tips

- Use references for parameters when you don't want to copy large objects.
- Use `const &` when passing read-only arguments to functions.

💻 Code Example

```
 void increment(int& r) { r++; }
void incrementPtr(int* p) { (*p)++; }

int main() {
    int a = 5;
    increment(a);      // by reference
    incrementPtr(&a);  // by pointer
    std::cout << a;    // prints 7
}
```

🔁 Common Questions

- _Can a reference be reseated?_ → No.
- _Why use pointers instead of references?_ → When you need nullability or dynamic memory management.

🧠 Pass-by-Value vs Pass-by-Reference

✅ Detailed Theory

- **Pass-by-value**: A **copy** of the variable is passed. Modifications don't affect the original.
- **Pass-by-reference**: The original variable is passed; changes inside the function are reflected outside.

💻 Example

```
 void byValue(int x) { x = 100; }
void byRef(int& x) { x = 100; }

int a = 5;
byValue(a); // a is still 5
byRef(a);   // a becomes 100
```

🧠 Stack vs Heap Memory

✅ Detailed Theory

- **Stack**:
    - Fast, automatically managed
    - Limited size
    - Used for function calls and local variables

- **Heap**:
    - Manually managed (`new` / `delete`)
    - Larger in size
    - Used for dynamic allocations

💡 Notes

- Stack has size limits and is automatically freed.
- Heap is manually managed and may fragment.

💻 Example

```
 int* p = new int(42); // heap
int x = 10;           // stack
```

✅ Visual Diagram

```
 Memory Layout:

| High Address |
|--------------|
|   Heap       | ← grows upward
|              |
|--------------|
|   Stack      | ← grows downward
|--------------|
| Low Address  |
```

🔁 Question

Q: _What happens if you forget `delete` ?_
A: Memory leak.

🧠 Arrays vs Vectors

✅ Detailed Theory

- **Arrays**:
    - Fixed size, static
    - No built-in size tracking
- **Vectors**:

- Dynamic size
- Built-in methods like `push_back`, `size`, `clear`

```
 int arr[5];               // Array
std::vector<int> vec(5); // Vector
vec.push_back(6);
```

🧠 Function Basics, Overloading, Default Arguments

✅ Detailed Theory

- C++ supports function overloading.
- You can define default values for parameters.

💻 Example

```
 void greet(std::string name = "Guest") {
    std::cout << "Hello, " << name << "\n";
}

void greet(int id) {
    std::cout << "Hello, user #" << id << "\n";
}
```

🧠 Control Structures: if, switch, for, while

✅ Summary

- `if`, `else if`, `else`: Conditional branching
- `switch`: Multi-branch based on constant expressions
- `for`, `while`, `do-while`: Looping constructs

💻 Example

```
 for (int i = 0; i < 3; i++) {
    std::cout << i << " ";
}
```

🧠 Endl vs \n

✅ Theory

- `std::endl` inserts a newline **and flushes** the output buffer.
- `'\n'` just inserts a newline without flushing.

💡 Interview Tip

Use `'\n'` for performance unless flushing is necessary.

🧠 cerr vs cout vs clog

✅ Theory

- `std::cout` : Standard output stream.
- `std::cerr` : Standard **error** stream (unbuffered).
- `std::clog` : Standard error (buffered).

💻 Example

```
 std::cout << "Normal output\n";
std::cerr << "Immediate error\n";
std::clog << "Buffered error\n";
```

💡 Interview Tip

Use `std::cerr` for logging errors in competitive programming or debugging.

🧠 Buffering and Flushing

✅ Detailed Theory

- **Buffered output**: Data is stored in memory temporarily before being written to the output device.
- **Flushing**: Forces the buffer to be written immediately.
- `std::endl` flushes the buffer.
- `std::cerr` is unbuffered (written immediately).
- `std::clog` is buffered (writes like `cout`).

💡 When to Use

- Use `\n` instead of `std::endl` to avoid unnecessary flushing in performance-sensitive code.
- Use `std::cerr` for debugging or immediate error logs.

2️⃣ C++ OOP Concepts (Core & Advanced)

🧱 Encapsulation & Abstraction

✅ Detailed Theory

🔐 Encapsulation

Encapsulation is the concept of **binding data (variables)** and the **functions that operate on that data** into a single unit — the **class**.

Encapsulation helps in:

- Data hiding (private members)
- Controlled access via public interfaces (getters/setters)
- Better maintainability and modularity

🎭 Abstraction

Abstraction focuses on **hiding internal implementation details** and showing only the relevant interface to the outside world.

In C++, abstraction is achieved using:

- Abstract classes (classes with at least one pure virtual function)
- Interfaces (pure virtual methods only)
- Public methods that expose only what's necessary

🧠 Deep Dives

- Private members cannot be accessed outside the class. This protects internal state.
- Abstraction and encapsulation often work together — encapsulation **implements** abstraction.
- Pure virtual functions (`= 0`) force derived classes to implement them.
- Use abstraction to define **what** an object does, and encapsulation to define **how** it does it.

💻 Code Examples

```
 class Account {
private:
    double balance;  // Encapsulated data

public:
    Account(double initial) : balance(initial) {}

    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    double getBalance() const { return balance; }
};

int main() {
    Account acc(100);
    acc.deposit(50);
    std::cout << acc.getBalance(); // Output: 150
}
```

Output:

```
 150
```

🔍 Dry Run

- `balance` is private → cannot be directly modified
- `deposit()` performs internal validation
- `getBalance()` safely exposes state to the outside

📊 Diagram (Encapsulation)

```
 +---------------------+
|      Account        |
|---------------------|
| - balance: double   | ← private data (hidden)
|---------------------|
| + deposit()         | ← public interface
| + getBalance()      |
+---------------------+
```

💡 Interview Tips

- Always explain encapsulation as “data + functions bundled in a class with restricted access.”
- For abstraction, emphasize "interface-focused design."
- Distinguish clearly between **data hiding (encapsulation)** and **interface hiding (abstraction)**.

🔁 Common Interview Questions

Q1: _How does C++ support encapsulation?_
A: Through access specifiers: `private`, `protected`, and `public`.

Q2: _How is abstraction implemented in C++?_
A: Using abstract classes, pure virtual functions, and well-designed interfaces.

Q3: _Can encapsulation exist without abstraction?_
A: Yes. Encapsulation is about access control; abstraction is about hiding complexity.

2️⃣ C++ OOP Concepts (Core & Advanced)

🧱 Encapsulation & Abstraction

✅ Detailed Theory

Encapsulation is the bundling of data and functions that operate on that data into a single unit (i.e., class). It restricts direct access to some components, enhancing data integrity and security.

Abstraction is the concept of hiding complex implementation details and showing only the necessary features of an object.

🧠 Deep Dives

- Encapsulation enables modular programming and information hiding.
- Abstraction is implemented via abstract classes and interfaces.

💻 Code Examples

```
 class Account {
private:
    double balance;
public:
```

```
    Account(double initial) : balance(initial) {}
    void deposit(double amount) { if (amount > 0) balance += amount; }
    double getBalance() const { return balance; }
};
```

🔍 Dry Run

1. Create account with 100.
2. Deposit 50 → new balance = 150.

📊 Diagram

```
 +---------------------+
|      Account        |
|---------------------|
| - balance: double   |
|---------------------|
| + deposit()         |
| + getBalance()      |
+---------------------+
```

💡 Interview Tips

- Emphasize separation of concerns and access control.

🔁 Common Questions

- Difference between abstraction and encapsulation?
- How is abstraction implemented in C++?

🧬 Inheritance (Single, Multiple, Multilevel, Hybrid)

✅ Detailed Theory

Inheritance enables a class (derived) to acquire properties and behavior (methods) from another class (base).

Types of inheritance:

- **Single**: One base class
- **Multiple**: Multiple base classes
- **Multilevel**: Inheritance through multiple levels
- **Hybrid**: Combination (e.g., multilevel + multiple)

💡 Access Control in Inheritance

Inheritance Type	Public Base Members	Protected Base Members	Private Base Members
public	public	protected	not accessible
protected	protected	protected	not accessible
private	private	private	not accessible

💻 Code Example

```
 class A { public: void foo() { std::cout << "A"; } };
class B : public A { public: void bar() { std::cout << "B"; } };
class C : public B { public: void baz() { std::cout << "C"; } };
```

📊 Diagram

```
A → B → C
```

🔷 Diamond Problem & Virtual Inheritance

💎 Diamond Problem & Virtual Inheritance

✅ Theory

The **Diamond Problem** arises in **multiple inheritance** when a class inherits from two classes that both inherit from a common base class. This can lead to **ambiguity** and **duplication**.

🔺 Example (Diamond Problem Without Virtual Inheritance)

```
 class A {
public:
    void show() { std::cout << "A::show\n"; }
};

class B : public A {};
```

```
class C : public A {};
class D : public B, public C {}; // Diamond problem

int main() {
    D obj;
    // obj.show(); // ❌ Ambiguous: which A::show?
}
```

💡 How `virtual` avoids Diamond Problem

Using **virtual inheritance**, the compiler ensures that only **one shared instance** of the base class is inherited.

✅ Corrected Example Using Virtual Inheritance

```
 class A {
public:
    void show() { std::cout << "A::show\n"; }
};

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

int main() {
    D obj;
    obj.show(); // ✅ Unambiguous
}
```

📊 Diagram:

```
 Without Virtual:
     A
    / \
   B   C
    \ /
     D  → Two copies of A

With Virtual:
     A
    / \
 virtual virtual
   B     C
    \   /
      D → One shared A
```

🧬 Virtual Inheritance (Detailed Explanation)

- **Virtual inheritance** ensures only one base class subobject when a class is indirectly inherited multiple times.
- It's used to **resolve ambiguity** and **eliminate redundancy** in the inheritance tree.
- When a class is **virtually inherited**, the compiler ensures it's shared across all inheritance paths.

🧠 Key Notes:

- The constructor of the **most derived class** initializes the virtual base.
- Common in situations with **diamond-shaped hierarchies**.

🔐 Access Specifiers

✅ Detailed Theory

Controls the accessibility of class members.

Specifier	Class Access	Derived Class	Outside Access
public	✅	✅	✅
protected	✅	✅	❌
private	✅	❌	❌

💻 Code Example

```
 class Base {
public: int pub;
protected: int prot;
private: int priv;
};

class Derived : public Base {
    void access() {
        pub = 1;    // ✅
        prot = 2;   // ✅
        // priv = 3; // ❌
    }
};
```

🔁 Polymorphism (Compile-time, Runtime, CRTP)

✅ Detailed Theory

- **Compile-time**: Function Overloading, Operator Overloading
- **Runtime**: Virtual functions
- **Static Polymorphism**: CRTP (Curiously Recurring Template Pattern)

💻 Example – Runtime

```
 class Base {
public: virtual void speak() { std::cout << "Base\n"; } };
class Derived : public Base {
public: void speak() override { std::cout << "Derived\n"; } };

Base* b = new Derived();
b->speak();  // Derived
```

🧠 CRTP (Curiously Recurring Template Pattern)

✅ Theory

CRTP is a C++ idiom where a class `Derived` inherits from a template base class which takes `Derived` as a template parameter.

🔁 Purpose

- Used for **static polymorphism**.
- Enables **compile-time binding** (avoiding virtual overhead).

💻 Example

```
 template<typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Derived implementation\n";
```

```
    }
};

int main() {
    Derived d;
    d.interface(); // Calls Derived::implementation()
}
```

📊 Diagram

```
 Base<Derived>
    ↑
 Derived
```

💻 Example – CRTP

```
 template <typename T>
class Base {
public:
    void speak() { static_cast<T*>(this)->impl(); }
};

class Derived : public Base<Derived> {
public:
    void impl() { std::cout << "CRTP Derived\n"; }
};

Derived d; d.speak();
```

🧩 Virtual Functions, vtable & vptr (In-depth)

✅ Virtual Function

- A function declared with `virtual` in the base class.
- Enables **runtime polymorphism**.

🔧 How It Works

1. Compiler generates a **vtable** (virtual table) for classes with virtual functions.
2. Each object has a **vptr** (virtual pointer) pointing to its class's vtable.
3. At runtime, the correct function is looked up via vtable.

💻 Example:

```
 class Base {
public:
    virtual void foo() { std::cout << "Base::foo\n"; }
};

class Derived : public Base {
public:
    void foo() override { std::cout << "Derived::foo\n"; }
};

int main() {
    Base* b = new Derived();
    b->foo(); // Calls Derived::foo via vtable
    delete b;
}
```

📊 Diagram:

```
 Base* ptr = new Derived();

Memory Layout:

[ Base Object (vptr) ] → [ vtable for Derived ]
                                ↓
                     +--------------------+
                     | &Derived::foo      |
                     +--------------------+
```

🧠 Notes:

- `vtable` is per class.
- `vptr` is per object.
- `vptr` is automatically set by constructor.

🧠 Interview Tips:

- Use `override` to catch mistakes.
- No `vtable` generated unless at least one virtual function.
- Virtual destructors are important in polymorphic base classes to avoid leaks.

🧠 Virtual Dispatch & Overhead

✅ What is Virtual Dispatch?

Virtual Dispatch is the process of selecting which function to invoke at **runtime**, based on the actual (dynamic) type of the object, not the type of the pointer/reference.

🔧 How Virtual Dispatch Works:

1. The class with virtual functions has a **vtable** (virtual table of function pointers).
2. Each object has a **vptr** pointing to its class's vtable.
3. When calling a virtual function via base class pointer/reference, the correct function is fetched from the vtable via the vptr.

💻 Example:

```
 class Animal {
public:
    virtual void speak() { std::cout << "Animal sound\n"; }
};

class Dog : public Animal {
public:
    void speak() override { std::cout << "Woof!\n"; }
};

int main() {
    Animal* a = new Dog();
    a->speak(); // Virtual dispatch: calls Dog::speak
    delete a;
}
```

📊 Virtual Dispatch Diagram:

```
 Animal* ptr = new Dog();

[ ptr (vptr) ] → [ vtable of Dog ]
                        ↓
            +------------------+
            | &Dog::speak()    |
            +------------------+
```

💸 What is Virtual Overhead?

1. **Runtime Indirection**
    - One extra memory fetch per virtual call.
    - Slightly slower than static dispatch.

2. **Memory Overhead**
    - Each object stores a `vptr` (usually 8 bytes).
    - Each class with virtual functions has a static `vtable`.
3. **Inlining Limitations**
    - Virtual functions generally **cannot be inlined**, limiting compiler optimizations.

💡 Interview Tips

- Always add `virtual` to base destructors when using polymorphism.
- CRTP enables polymorphism without vtables.

🔁 Common Questions

- What's the difference between static and dynamic polymorphism?
- What's the cost of virtual functions?
- When is CRTP better than runtime polymorphism?

#

🔁 Common Questions

Q1. What is the diamond problem? → Ambiguity from multiple inheritance of a common base. Resolved with virtual inheritance.

Q2. How does virtual inheritance solve the diamond problem? → Ensures only one shared base instance across inheritance paths.

Q3. What is CRTP and its advantage? → Static polymorphism pattern using templates to achieve compile-time binding.

Q4. Difference between static vs dynamic polymorphism? → Static via templates (compile-time), dynamic via virtual functions (runtime).

Q5. How does vtable and vptr work? → `vtable` holds function pointers, `vptr` points to vtable. Enables virtual dispatch.

Q6. What is virtual dispatch? → The runtime mechanism of choosing the correct function via vtable based on the dynamic type.

Q7. What is the overhead of virtual functions? → Extra memory (vptr), indirection cost, and less optimization (no inlining).

🧠 Virtual Functions, Cost of Dynamic Dispatch & Virtual Destructors

✅ Virtual Functions Recap

- Enable runtime polymorphism.
- Introduce indirection via vtable lookup.

💸 Cost of Dynamic Dispatch

1. **Indirect Call Overhead**:
    - Virtual calls require a lookup in the vtable.
    - One extra memory access compared to direct (static) calls.
2. **Inlining Prevention**:
    - Compilers cannot inline virtual calls easily.
    - Misses out on instruction-level optimization.
3. **Branch Prediction Penalties**:
    - Virtual calls can be less predictable, especially in tight loops.
4. **Memory Cost**:
    - Each polymorphic object stores a `vptr`.
    - Each class with virtual functions has a static `vtable`.

❓ Can you remove virtual functions using CRTP?

Yes ✅

- **CRTP (Curiously Recurring Template Pattern)** allows compile-time polymorphism.
- Replaces virtual dispatch with **static dispatch**, eliminating `vtable` and `vptr`.

```
 template<typename Derived>
class Shape {
public:
    void draw() { static_cast<Derived*>(this)->draw_impl(); }
};
```

```
class Circle : public Shape<Circle> {
public:
    void draw_impl() { std::cout << "Circle\n"; }
};
```

- ✅ Faster (no indirection)
- ❌ Less flexible (requires templates and known types at compile-time)

⚡ Improving Performance in Polymorphic Systems

💡 Techniques to Improve Virtual Call Performance:

1. **Avoid Virtual Calls in Hot Paths**
    - Use virtual functions in setup/configuration.
    - Use non-virtual functions inside performance-critical loops.
2. **Use `final` and `override`**
    - `final` allows compilers to devirtualize calls.
    - Prevents further subclassing → more optimizations.
3. **Use CRTP Where Possible**
    - CRTP avoids runtime dispatch entirely.
4. **Profile and Inline Manually**
    - Identify bottlenecks.
    - Refactor with static dispatch.
5. **Enable Link-Time Optimization (LTO)**
    - Modern compilers can devirtualize across TU boundaries during link time.
6. **Use Jump Tables (Advanced)**
    - In systems like game engines, use custom dispatch tables for performance-critical paths.

🔐 Friend Classes and Friend Functions

✅ What is a Friend?

- A `friend` class or function is granted access to the **private and protected** members of another class.
- Defined using the `friend` keyword.

💡 Friend Function

```
 class Box {
    int width;
    friend void printWidth(Box);
public:
    Box(int w) : width(w) {}
};

void printWidth(Box b) {
    std::cout << b.width << '\n'; // Allowed due to friendship
}
```

💡 Friend Class

```
 class Engine;

class Car {
    int speed;
    friend class Engine; // Engine can access Car’s private members
public:
    Car() : speed(100) {}
};

class Engine {
public:
    void boost(Car& c) {
        c.speed += 50; // Legal
    }
};
```

📌 Use Cases

- Operator overloading (e.g., `operator<<` for `std::ostream`)
- Tight coupling for performance or encapsulation
- Testing private members

⚠️ Caution

- Use sparingly: breaks encapsulation
- Prefer public APIs unless performance or design demands direct access

---

🔁 Common Questions

Q8. What is the cost of dynamic dispatch? → Extra indirection, vtable lookup, blocked inlining, branch mispredictions.

Q9. Can virtuals be removed with CRTP? → Yes. CRTP allows static dispatch, avoiding vtable/vptr.

Q10. How to speed up systems using many virtual calls? → Use `final`, CRTP, LTO, avoid hotpath virtuals.

Q11. What is a friend function/class? → Non-member function/class with access to private/protected members.

🔧 Functions in C++ (Member, Non-Member, Static, Lambda)

✅ Types of Functions in C++

1. **Member Functions**
    - Defined inside a class.
    - Can access all members (including private).

```
 class MyClass {
public:
    void greet() { std::cout << "Hello!\n"; }
};
```

2. **Static Member Functions**
    - Do not require an object.
    - Can only access static members.

```
 class MyClass {
    static int count;
public:
    static void increment() { count++; }
};
```

3. **Non-Member Functions**
    - Defined outside any class.

- Can be made `friend` to access private members.

```
void print(int x) {
    std::cout << x << '\n';
}
```

4. **Inline Functions**

    - Suggested to be inlined by the compiler.
    - Useful for short, frequently-used functions.

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

5. **Const Member Functions**

    - Do not modify the object.
    - Can be called on `const` instances.

```
class MyClass {
public:
    int getVal() const { return 42; }
};
```

6. **Lambda Functions**

    - Anonymous inline functions.
    - Capture local context.

```
auto add = [](int a, int b) { return a + b; };
std::cout << add(3, 4); // Outputs 7
```

🔁 Function Overloading

- Multiple functions with the same name but different signatures.
- Resolved at compile-time.

```
void print(int);
void print(double);
```

🧠 Function Pointers

- Store address of a function.
- Enable callbacks and dispatching.

```
void hello() { std::cout << "Hi"; }
void (*funcPtr)() = hello;
funcPtr();
```

📌 Use Cases

- Abstraction (interfaces)
- Event-driven programming
- Callbacks and hooks (e.g., in GUI or OS-level APIs)

🔁 Common Questions

Q12. What are the different types of functions in C++? → Member, static, non-member, inline, const, lambda.

Q13. What is the difference between static and non-static member functions? → Static functions don't access instance data; no `this` pointer.

Q14. How are lambdas different from function pointers? → Lambdas can capture context; function pointers can't.

📐 Rule of 0 / 3 / 5 and Constructor Delegation

✅ Rule of 0

- Rely on compiler-generated special member functions.
- Prefer when using only RAII-compliant types (e.g., STL containers).

```
class Safe {
   std::string name;
   // No manual destructor, copy/move
};
```

✅ Rule of 3

If you need to define one of these, you likely need all three:

1. Destructor
2. Copy constructor
3. Copy assignment operator

```
class MyResource {
    ~MyResource();
    MyResource(const MyResource&);
    MyResource& operator=(const MyResource&);
};
```

✅ Rule of 5 (C++11+)

Adds move semantics: 4. Move constructor 5. Move assignment operator

```
class MyResource {
    ~MyResource();
    MyResource(const MyResource&);
    MyResource(MyResource&&);
    MyResource& operator=(const MyResource&);
    MyResource& operator=(MyResource&&);
};
```

✅ Constructor Delegation (C++11)

- A constructor can call another constructor in the same class.

```
class MyClass {
    int a, b;
public:
    MyClass(int x) : MyClass(x, 0) {}
    MyClass(int x, int y) : a(x), b(y) {}
};
```

💣 Destructors, Inheritance, and RAII

✅ RAII (Resource Acquisition Is Initialization)

- Acquire resources in constructor.
- Release in destructor.
- Ensures exception-safe cleanup.

```
class File {
    std::fstream f;
public:
    File(const std::string& path) : f(path) {}
    ~File() { f.close(); }
};
```

⚠️ Why Destructor Might Not Be Called?

1. **Object not destroyed**:
    - Leaked due to `new` without `delete`.
2. **Sliced objects**:
    - Base class destructor not virtual → derived destructor not called.
3. **Exception thrown before object fully constructed**:
    - Destructor not invoked.

❌ What if base destructor is not virtual?

```
 class Base {
public:
    ~Base() { std::cout << "Base Dtor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Derived Dtor\n"; }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Only Base destructor is called ❌
}
```

✅ Corrected with virtual destructor:

```
 class Base {
public:
    virtual ~Base() { std::cout << "Base Dtor\n"; }
};
```

📌 Best Practice

- Always declare base class destructor `virtual` if it's intended for inheritance.
- Or make base class destructor `= default` and `virtual`.

🔁 Common Questions

Q15. What is the Rule of 0 / 3 / 5? → Guides writing/omitting special member functions based on resource ownership.

Q16. What is constructor delegation? → A constructor calling another constructor in the same class.

Q17. Why is my destructor not called? → Possibly due to slicing or missing virtual destructor in base class.

Q18. What happens if base destructor is not virtual? → Undefined behavior when deleting through base pointer; derived destructor won't run.

🧿 Simulating Polymorphism Without Virtual Functions

✅ Alternatives to Virtual Functions

1. **Function Pointers / Function Objects**

```
 struct Strategy {
    std::function<void()> execute;
};

Strategy s1 { [](){ std::cout << "A\n"; } };
Strategy s2 { [](){ std::cout << "B\n"; } };
s1.execute();
s2.execute();
```

2. **CRTP (Curiously Recurring Template Pattern)**

- Achieves compile-time polymorphism.

```
 template <typename Derived>
class Shape {
public:
    void draw() {
        static_cast<Derived*>(this)->draw_impl();
    }
};

class Circle : public Shape<Circle> {
public:
    void draw_impl() { std::cout << "Circle\n"; }
};
```

3. **Tagged Dispatch (Manual vtable)**

```
enum class Type { A, B };

void handle(Type t) {
    switch(t) {
        case Type::A: std::cout << "A\n"; break;
        case Type::B: std::cout << "B\n"; break;
    }
}
```

➕ Operator Overloading

✅ Why and When to Overload Operators

- To make user-defined types behave like built-in types.
- Improves readability and expressiveness.

💡 Common Operators to Overload

- `<<` for output
- `>>` for input
- `==`, `!=`, `<` for comparisons
- `[]` for container-style access
- `()` for functor-style behavior
- `+`, `-`, `*`, `/` for arithmetic

💻 Example: Overload `<<`

```
class Point {
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};
```

🧠 Example: Overload `[]`

```
class Array {
    int data[10];
public:
    int& operator[](int idx) { return data[idx]; }
    const int& operator[](int idx) const { return data[idx]; }
};
```

📌 Best Practices

- Use `friend` when operator needs access to private members.
- Maintain symmetry and semantics of built-in operators.
- Use `const` correctness.
- Keep overloaded operators intuitive.

🧱 SOLID Principles (OOP Design)

✅ S – Single Responsibility Principle

- A class should have only one reason to change.
- Each class should encapsulate a single part of the functionality.

✅ O – Open/Closed Principle

- Software entities should be **open for extension**, but **closed for modification**.
- Use interfaces and abstract base classes.

✅ L – Liskov Substitution Principle

- Derived classes must be substitutable for their base classes without altering program correctness.

✅ I – Interface Segregation Principle

- Clients should not be forced to depend on interfaces they do not use.
- Favor many small, focused interfaces over a large one.

✅ D – Dependency Inversion Principle

- Depend on abstractions, not on concrete implementations.
- High-level modules should not depend on low-level modules.

💡 Example of Good SOLID:

```
 class IPrinter {
public:
    virtual void print() = 0;
    virtual ~IPrinter() {}
};

class PDFPrinter : public IPrinter {
```

```
public:
    void print() override { std::cout << "PDF Print\n"; }
};
```

🔁 Common Questions

Q19. How can polymorphism be achieved without virtual functions? → Using CRTP, function pointers, lambdas, or switch-based dispatch.

Q20. When should you overload operators? → When it improves usability and semantics match built-in behavior.

Q21. What are the SOLID principles? → A set of 5 OOP design principles: SRP, OCP, LSP, ISP, DIP.

🏛️ Abstract Classes vs Interfaces

✅ Abstract Class

- A class with at least one **pure virtual function** (`= 0`).
- Cannot be instantiated.
- Used to define **interfaces** or **contracts**.

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

✅ Interface in C++

- C++ does not have a built-in `interface` keyword.
- An interface is modeled using a class with **only pure virtual functions** and a virtual destructor.

```
 class IPrintable {
public:
    virtual void print() = 0;
    virtual ~IPrintable() {}
};
```

🚫 Instantiation

```
Shape s;        // ❌ Error: Cannot instantiate abstract class
Shape* s = new Circle(); // ✅
```

🔁 Interface Implementation

- Any class implementing all pure virtual methods becomes concrete and can be instantiated.

🔢 Object Counting Pattern

✅ Pattern: Count Number of Instances

```
class Tracker {
    static int count;
public:
    Tracker() { ++count; }
    ~Tracker() { --count; }
    static int getCount() { return count; }
};

int Tracker::count = 0;
```

🧠 Use Cases:

- Debugging resource leaks
- Tracking live objects (e.g., singleton lifecycle)
- Limiting instances (e.g., max connection limit)

🧱 Constructor & Destructor Order in Inheritance

🧩 Constructor Call Order

1. Base class constructor
2. Member object constructors (in order of declaration)
3. Derived class constructor

```
class A {
public:
    A() { std::cout << "A ctor\n"; }
};

class B : public A {
```

```
public:
    B() { std::cout << "B ctor\n"; }
};

// Output: A ctor → B ctor
```

🧨 Destructor Call Order

1. Derived class destructor
2. Member object destructors (reverse of declaration)
3. Base class destructor

```
 class A {
public:
    ~A() { std::cout << "A dtor\n"; }
};

class B : public A {
public:
    ~B() { std::cout << "B dtor\n"; }
};

int main() {
    B obj;
} // Output: B dtor → A dtor
```

🧠 Best Practice:

- Always declare base class destructors as `virtual` for polymorphic base classes.

🔁 Common Questions

Q19. How can polymorphism be achieved without virtual functions? → Using CRTP, function pointers, lambdas, or switch-based dispatch.

Q20. When should you overload operators? → When it improves usability and semantics match built-in behavior.

Q21. What are the SOLID principles? → A set of 5 OOP design principles: SRP, OCP, LSP, ISP, DIP.

Q22. What is an abstract class in C++? → A class with at least one pure virtual function. Cannot be instantiated.

Q23. How do interfaces work in C++? → Modeled via abstract base classes with only pure virtual functions.

Q24. How can I count live object instances? → Use static counter variable incremented in constructor, decremented in destructor.

Q25. What is the order of constructor and destructor calls in inheritance? → Constructor: base → members → derived. Destructor: derived → members → base.

🧨 Virtual Destructors vs Virtual Constructors

✅ Virtual Destructors

- Allow proper cleanup of derived class objects via base class pointers.
- Must be used if you intend to delete derived objects polymorphically.

```
 class Base {
public:
    virtual ~Base() { std::cout << "Base Dtor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Derived Dtor\n"; }
};

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

⚠️ If Destructor is Not Virtual

- Only base class destructor is called → potential **resource leak**.

🧠 Best Practice:

- Always make base destructors `virtual` in polymorphic base classes.

🚫 Virtual Constructors?

- C++ **does not support virtual constructors**.
- Constructor call must be bound **at compile-time**, not runtime.

✅ Alternatives:

1. **Factory Pattern**

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

    static Shape* createCircle();
};

class Circle : public Shape {
public:
    void draw() override { std::cout << "Circle\n"; }
};

Shape* Shape::createCircle() {
    return new Circle();
}
```

2. **Virtual Clone (Prototype Pattern)**

```
 class Shape {
public:
    virtual Shape* clone() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    Circle* clone() const override { return new Circle(*this); }
};
```

🔁 Common Questions

Q26. What is a virtual destructor? → A destructor declared `virtual` in base class to ensure derived destructors are called via base pointer.

Q27. Why doesn't C++ support virtual constructors? → Constructors need compile-time binding; use factory or clone pattern instead.

🔄 Types of Inheritance & Interface Segregation

✅ Public Inheritance ("is-a" relationship)

- The derived class is a subtype of the base.
- Public members of base remain public in derived.

```
 class Animal {
public:
    void eat() {}
};

class Dog : public Animal {
    void bark() {}
};
```

⚠️ Private Inheritance ("implemented-in-terms-of")

- Public and protected members become private in derived.
- No subtype relationship.

```
 class Timer {
public:
    void start();
};

class Engine : private Timer {
public:
    void useTimer() { start(); } // Access allowed inside
};
```

🧱 Protected Inheritance

- Public and protected members of base become protected in derived.
- Rarely used in practice.

🧩 Interface Segregation (SOLID - I)

- Clients should not be forced to implement interfaces they do not use.
- Split large interfaces into smaller, more specific ones.

```
 class IReadable {
public:
    virtual void read() = 0;
};

class IWritable {
```

```
public:
    virtual void write() = 0;
};

class File : public IReadable, public IWritable {
    void read() override {}
    void write() override {}
};
```

🧬 Class vs Struct vs Union vs Enum vs Typedef

✅ Class vs Struct

- Both define types with data + functions.
- **Default access:**
  - `class` → private
  - `struct` → public

```
 class A {
    int x; // private by default
};

struct B {
    int y; // public by default
};
```

📌 struct Details

- `struct` is often used for **POD** (Plain Old Data) types.
- Allows public data grouping with minimal encapsulation.
- Supports all OOP features (inheritance, methods, etc.)

```
 struct Point {
    int x, y;
    void print() const { std::cout << x << ", " << y << "\n"; }
};
```

✅ Union

- All members share the **same memory** location.
- Only **one member should be accessed** at a time.
- Useful for memory optimization in low-level programming.

```
union Data {
    int i;
    float f;
    char c;
};
```

⚠️ Important Notes:

- Changing one member affects others.
- Cannot have non-trivial constructors/destructors unless explicitly defined (C++11+).

📊 Visualization:

```
union Data {
    int i;      // 4 bytes
    float f;    // 4 bytes
    char c;     // 1 byte (but shares full 4-byte space)
};

Memory:
+------------+
|  i / f / c | ← all occupy the same space (largest member defines size)
+------------+
```

✅ Enum

- Define **named constants**.
- **Default underlying type** is `int`.

```
enum Color { RED, GREEN, BLUE }; // RED = 0, GREEN = 1, etc.
```

🎯 Changing Enum Underlying Type (C++11+)

```
enum class Status : unsigned char {
    OK = 1,
    FAIL = 2
};
```

📌 Benefits of `enum class`

- Strongly typed: avoids accidental conversions.
- Scoped: `Status::OK`, not just `OK`.
- Can define smaller storage types (like `char`, `short`) to optimize memory.

```
void handle(Status s) {
    if (s == Status::OK) std::cout << "Success";
}
```

✅ Typedef vs Using vs #define

📌 typedef

- Legacy C-style syntax for creating type aliases.
- Cannot be used with templates easily.

```
typedef unsigned int uint;
typedef int (*funcPtr)(double);
```

✅ using (C++11+)

- Modern, clearer alternative to `typedef`.
- Works seamlessly with templates.

```
using uint = unsigned int;
using funcPtr = int(*)(double);
```

⚠️ #define

- Preprocessor macro.
- No type checking.
- Replaced by the preprocessor before compilation.

```
#define PI 3.1415
#define SQUARE(x) ((x)*(x))
```

🧠 Comparison Table:

Feature	`typedef`	`using`	`#define`
Type checking	✅ Yes	✅ Yes	❌ No
Works with templates	❌ No	✅ Yes	❌ No
Preprocessor	❌ No	❌ No	✅ Yes
Readability	Moderate	✅ High	❌ Low

🔁 Common Questions

Q. What is a union and how does it differ from struct? → A union shares memory for all members, struct allocates separately.

Q. How can you change the default type of an enum? → Use `enum class Name : type {}` syntax (C++11+).

Q. Difference between typedef, using, and #define? → `typedef` is legacy, `using` is modern and supports templates, `#define` is unsafe macro substitution.

🧱 Constructor Call: Base → Derived

- **Constructor call order:**
    1. Base class constructor
    2. Member initializers (in order of declaration)
    3. Derived class constructor

```
 class A {
public:
    A() { std::cout << "A ctor\n"; }
};

class B : public A {
public:
    B() { std::cout << "B ctor\n"; }
};

// Output: A ctor → B ctor
```

📞 Calling Base Constructor from Derived

- Use **constructor initializer list**.

```
 class A {
public:
    A(int x) { std::cout << "A(" << x << ")\n"; }
};

class B : public A {
public:
```

```
    B() : A(5) { std::cout << "B ctor\n"; }
};
```

- If no explicit call made, default constructor is called automatically.

🔁 Common Questions

Q26. What is a virtual destructor? → A destructor declared `virtual` in base class to ensure derived destructors are called via base pointer.

Q27. Why doesn't C++ support virtual constructors? → Constructors need compile-time binding; use factory or clone pattern instead.

Q28. What is private inheritance and when to use it? → Implementation reuse without establishing an is-a relationship.

Q29. What's the difference between class, struct, union, enum, typedef? → Access control, memory layout, type aliasing.

Q30. In what order are constructors called? → Base → members → derived.

Q31. How do you call a base class constructor from a derived class? → Use initializer list: `Derived() : Base(args) {}`

🧬 Implicit & Explicit Constructor Inheritance with `using`

✅ Implicit Constructor Inheritance (C++11+)

- Use `using Base::Base;` to inherit all accessible constructors from base class.
- Implicit: no need to manually redefine constructors.

```
 class Base {
public:
    Base(int x) { std::cout << "Base(" << x << ")\n"; }
};

class Derived : public Base {
    using Base::Base; // Inherits Base(int)
};

Derived d(5); // OK: calls Base(5)
```

✅ Explicit Constructor Inheritance

- You can explicitly define constructors and manually call base constructors.

```
 class Derived : public Base {
public:
    Derived(int x) : Base(x) {}
};
```

🧠 Notes:

- Implicit is concise and automatic.
- Explicit gives full control (e.g., logging, additional init).

✂️ Object Slicing

❌ What is Object Slicing?

- When a derived object is assigned to a base class object **by value**, only the base part is copied.
- Derived-specific members and behavior are lost.

```
 class Base { public: int x = 1; };
class Derived : public Base { public: int y = 2; };

Base b = Derived(); // Object slicing: only Base part remains
```

⚠️ Avoid By:

- Using **base class pointers or references**:

```
 Derived d;
Base& ref = d;   // no slicing
Base* ptr = &d;  // no slicing
```

- These maintain full polymorphic identity of the derived object.

🔁 Assigning Between Base and Derived

✅ Base ← Derived (Upcasting)

- Safe and implicit.

- Common when using polymorphism (base pointer/reference to derived object).

```
Derived d;
Base* b = &d; // OK
```

⚠️ Derived ← Base (Downcasting)

- Needs `static_cast` or `dynamic_cast`.
- Unsafe unless the base actually points to a derived instance.

```
Base* b = new Derived();
Derived* d = static_cast<Derived*>(b); // OK if b really points to Derived
```

✅ Why Use This?

- To allow generalized interfaces while enabling specialized behavior.
- Essential in polymorphic systems:

```
std::vector<Base*> objects;
objects.push_back(new Derived());
```

🧠 Best Practice:

- Use `dynamic_cast` when RTTI is available and correctness matters:

```
if (Derived* d = dynamic_cast<Derived*>(b)) { /* safe */ }
```

🔄 Copy/Move Constructors Across Base and Derived

✅ Base Class Copy/Move

- Derived constructors **must call base constructors** explicitly.

```
class Base {
public:
    Base(const Base&) { std::cout << "Base copy\n"; }
};

class Derived : public Base {
public:
    Derived(const Derived& d) : Base(d) {
        std::cout << "Derived copy\n";
    }
};
```

⚙️ Move Semantics

- Same rules apply for move constructors and move assignment operators.
- Must explicitly call base class move constructor in derived.

```
Derived(Derived&& d) : Base(std::move(d)) {}
```

🧠 Note:

- If base class copy/move constructors are deleted or private, derived constructors must handle it accordingly.

🧨 Virtual Destructors & Polymorphic Cleanup

✅ Why Virtual Destructors Matter

- When deleting a derived object through a base pointer, **only base destructor is called** unless it's virtual.
- Virtual destructors ensure proper **polymorphic cleanup**.

```
 class Base {
public:
    virtual ~Base() { std::cout << "Base dtor\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Derived dtor\n"; }
};

Base* b = new Derived();
delete b; // Calls Derived dtor → Base dtor (if Base has virtual dtor)
```

❌ If Not Virtual:

- Only Base's destructor is called → **memory/resource leak** if Derived allocates.

📌 Polymorphic Cleanup

- Cleanup performed via a **base pointer** to an object of **derived type**.
- Without virtual destructor, base can’t correctly clean up derived parts.

🧠 Best Practice:

- Always declare destructors as `virtual` in base classes with virtual functions or intended for polymorphism.

🔁 Common Questions

Q35. How do you inherit constructors from a base class? → Use `using Base::Base;` in the derived class.

Q36. What is object slicing? → Losing derived members when assigning derived object to base object by value.

Q37. Is assigning base to derived safe? → No. Use `dynamic_cast` or `static_cast` with care.

Q38. How do copy/move constructors work with inheritance? → Derived must explicitly invoke base copy/move constructors.

Q39. Why are virtual destructors needed in polymorphic hierarchies? → To ensure derived destructors are called correctly when deleting via base pointer.

Q40. What is polymorphic cleanup? → Deleting a derived object via a base pointer with a virtual destructor to ensure full cleanup.

3️⃣ C++ Language Features

🔁 `this` Pointer

✅ What is `this`?

- `this` is an implicit pointer available inside **non-static** member functions.
- It points to the **calling object**.

```
 class A {
    int x;
public:
    A(int x) { this->x = x; }
    A& set(int x) { this->x = x; return *this; }
};
```

📌 Use Cases:

- Return *this for chaining.
- Disambiguate shadowed variables.
- Pass current object to another method.

🚀 `constexpr`, `consteval`, `constinit`

✅ `constexpr` (C++11+)

- Tells the compiler to **evaluate at compile-time** if possible.
- Applies to variables, functions, and constructors.

```
constexpr int square(int x) { return x * x; }
constexpr int val = square(5); // evaluated at compile-time
```

✅ `consteval` (C++20)

- Stronger version of `constexpr`.
- Must be evaluated at **compile-time**, not optionally.

```
consteval int alwaysCompileTime(int x) { return x * 2; }
constexpr int result = alwaysCompileTime(10); // OK
```

✅ `constinit` (C++20)

- Ensures that a **static/global variable** is initialized **at compile-time**.
- Prevents runtime initialization surprises.

```
constinit int staticVal = 42; // must be initialized at compile-time
```

📌 Summary Table

Keyword	When Evaluated	Applies To	Notes
`constexpr`	Compile-time (if possible)	vars, functions, ctors	Can fall back to runtime if needed
`consteval`	Must be compile-time	functions only	Always evaluated during compilation
`constinit`	Compile-time init only	static/global vars	Prevents static runtime init surprise

---

🔐 Const Correctness

✅ `const` Member Functions

- Cannot modify any member variables.
- Can be called on `const` objects.

```
class A {
    int x;
public:
    int get() const { return x; } // const method
};
```

✅ `const` Arguments & Return Types

```
void print(const std::string& s); // avoids copy, prevents modification
const int getValue(); // return is const (not very useful in most cases)
```

✅ `const` Pointers

- `const int* ptr` → pointer to const int (can't change value)
- `int* const ptr` → const pointer (can't change address)
- `const int* const ptr` → both are const

```
const int* a = &val;     // value protected
int* const b = &val;     // pointer protected
const int* const c = &val; // both protected
```

🧠 Why Const Correctness?

- Helps compiler catch unintended modifications.
- Enables function overloading by const qualifier.
- Improves API clarity and safety.

🧪 `decltype` , `auto` , `decltype(auto)`

✅ `auto`

- Deduces the type of the initializer (value type).

```
auto x = 1;         // int
const auto y = 2;   // const int
```

✅ `decltype`

- Yields the type of an expression **without evaluating it**.

```
 int a = 5;
decltype(a) b = 10; // int
```

✅ `decltype(auto)` (C++14)

- Combines `decltype` with `auto` to **preserve reference and const qualifiers**.

```
 int x = 5;
int& getRef() { return x; }
decltype(auto) r = getRef(); // r is int&
```

🔁 Comparison Table

Feature	Captures	Preserves CV/Ref?	Example Result
`auto`	Value	❌ No	`int`
`decltype(expr)`	Exact type	✅ Yes	`int&`
`decltype(auto)`	Exact init	✅ Yes	`const int&`

📦 `std::optional`, `std::variant`, `std::any`

✅ `std::optional<T>`

- May or may not hold a value.
- Used as a **safe alternative to nullable pointers** or sentinel values.

```
 std::optional<int> getValue(bool ok) {
    if (ok) return 42;
    return std::nullopt;
}
```

✅ `std::variant<Ts...>`

- Type-safe union: holds one of several specified types.
- Replaces `union` or tagged structs.

```
 std::variant<int, float, std::string> val = "hello";
if (std::holds_alternative<std::string>(val)) {
    std::cout << std::get<std::string>(val);
}
```

✅ `std::any`

- Holds a value of **any type**, type-erased.
- Less type-safe but more flexible.

```
 std::any a = 5;
a = std::string("text");

if (a.type() == typeid(std::string)) {
    std::cout << std::any_cast<std::string>(a);
}
```

📊 Summary Table

Feature	Holds	Type-safe	Use Case
`optional<T>`	0 or 1 value	✅ Yes	Optional returns, nullable alt
`variant<Ts>`	1 of N types	✅ Yes	Tagged unions, type-dispatch
`any`	Any type	❌ No	Flexible but less safe container

🚫 Namespaces & Anonymous Namespaces

✅ Namespaces

- Prevent name conflicts in large projects.

```
 namespace math {
    int square(int x) { return x * x; }
}

int val = math::square(4);
```

✅ Anonymous Namespaces

- Provide **internal linkage** within the current translation unit.
- Replaces `static` globals.

```
namespace {
    int internalOnly = 42;
}
```

⚠️ Namespace Pollution

- Occurs when symbols are dumped into the global namespace.
- Can cause ambiguity and symbol collisions.

```
using namespace std; // BAD in headers or global scope
```

✅ Best Practice:

- Use scoped access: `std::vector`
- Prefer `using` inside functions or limited scope.

⚙️ Macros vs `using` vs `typedef` vs Templates

✅ Macros (`#define`)

- Preprocessor directives.
- No type safety, no scope.

```
#define PI 3.14159
#define SQUARE(x) ((x)*(x))
```

✅ typedef

- Creates type aliases.
- Limited support for templates.

```
typedef unsigned int uint;
```

✅ using (C++11+)

- Modern replacement for `typedef`.
- Works with templates.

```
using uint = unsigned int;
```

✅ Templates

- Compile-time type-safe code generation.
- Replaces macro-based generics.

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

📊 Comparison Table

Feature	Type-Safe	Template-Safe	Scope Aware	Recommended
`#define`	❌ No	❌ No	❌ No	❌ Avoid
`typedef`	✅ Yes	❌ No	✅ Yes	⚠️ Legacy
`using`	✅ Yes	✅ Yes	✅ Yes	✅ Yes
Template	✅ Yes	✅ Yes	✅ Yes	✅ Yes

🧊 `static` Keyword

✅ Static Global Variable

- Visible only within the file.

```
static int counter = 0; // internal linkage
```

✅ Static Member Variable

- Shared across all objects of the class.

```
class A {
   static int count;
};
int A::count = 0;
```

✅ Static Member Function

- Can only access static members.
- No `this` pointer.

```
class Logger {
public:
```

```
    static void log(const std::string& msg); // no access to non-static members
};
```

✅ Static Local Variable (Inside Function)

- Retains value across function calls.

```
void countCalls() {
    static int calls = 0;
    ++calls;
    std::cout << calls << "\n";
}
```

📌 Class-Level vs Object-Level Members

Scope	Belongs To	Keyword	Access Type
Static	Class	`static`	No `this`
Non-Static	Each object	—	Uses `this`

🔗 Preprocessor Directives: `#pragma`, `#define`, `#include`

3️⃣ C++ Language Features

...(existing sections remain unchanged)

🔗 Preprocessor Directives: `#pragma`, `#define`, `#include`

✅ `#pragma`

- Compiler-specific directive.
- Used for optimization hints, diagnostic control, header guards, etc.

```
#pragma once  // modern alternative to include guards
```

🚀 Common `#pragma` Optimizations

Pragma	Purpose
`#pragma once`	Ensures file is only included once per compilation unit
`#pragma GCC optimize`	Tells GCC to use specific optimization flags (e.g., O3, Ofast, inline)
`#pragma GCC target`	Specify CPU-specific instruction sets (e.g., "sse4", "avx2")
`#pragma inline`	Suggest inlining specific functions (non-standard, rarely used)
`#pragma pack(n)`	Change structure alignment to `n` bytes (affects padding)
`#pragma warning(disable:xyz)`	Disable specific compiler warning (e.g., MSVC)
`#pragma region/endregion`	Used in Visual Studio to create collapsible code regions
`#pragma message("text")`	Display a custom message during compilation

🔧 GCC-Specific Example

```
 #pragma GCC optimize("O3")
#pragma GCC target("avx2")
```

- Applies high optimization and AVX2 instruction set (for numeric loops, SIMD, etc.)

🔧 MSVC-Specific Example

```
#pragma warning(disable : 4996) // Disable deprecation warning
```

⚠️ Note:

- Always verify that your compiler supports the pragma.
- Misuse may reduce portability or silently degrade performance.

✅ `#define`

- Defines constants or macros.
- Preprocessor replaces before compilation.

```
#define PI 3.1415
#define SQUARE(x) ((x)*(x))
```

✅ `#include`

- Inserts content of a file into current file.

```
#include <iostream>   // standard library
#include "myheader.h" // user-defined
```

🔢 `enum` vs `enum class`

✅ Traditional `enum`

- Plain constants, unscoped.
- Values leak into enclosing scope.

```
enum Color { RED, GREEN, BLUE };
int c = RED;
```

✅ `enum class` (C++11+)

- Scoped, strongly typed.
- Safer: avoids collisions and implicit conversions.

```
enum class Direction { LEFT, RIGHT };
Direction d = Direction::LEFT;
```

📊 Comparison Table

Feature	`enum`	`enum class`
Scope	Global	Scoped
Implicit Convert	✅ Yes	❌ No
Type Safety	❌ Weak	✅ Strong
Preferred	❌ Legacy	✅ Modern

📐 `size_t` and Fixed-Width Types

✅ `size_t`

- Unsigned type returned by `sizeof`.
- Platform-dependent (typically 32/64-bit).
- Used for indexing, memory sizes.

```
size_t len = sizeof(arr);
```

✅ Fixed-Width Types (`<cstdint>`)

- Precise control over integer size and sign.
- Introduced in C++11 for portability.

```
#include <cstdint>

int8_t a;     // 8-bit signed
uint32_t b;   // 32-bit unsigned
int64_t c;    // 64-bit signed
```

📌 Best Practice

- Use `size_t` for sizes and indexing.
- Use fixed-width types when size consistency matters (e.g., networking, serialization).

📥 `istream` and `ostream`

✅ `std::istream`

- Represents input streams (e.g., `cin`, file input).
- Provides extraction (`>>`) operators and functions like `.get()`, `.getline()`.

```
std::string name;
std::cin >> name;
```

✅ `std::ostream`

- Represents output streams (e.g., `cout`, file output).
- Provides insertion (`<<`) operators and functions like `.put()`.

```
std::cout << "Hello, " << name << std::endl;
```

🔄 Difference: `istream` vs `ostream`

Feature	`istream`	`ostream`
Direction	Input (read from source)	Output (write to sink)
Common Classes	`cin`, `ifstream`, `istringstream`	`cout`, `ofstream`, `ostringstream`
Operator	`>>`	`<<`

✅ Standard Stream Types in Detail

1. `std::cin` (console input)

- Reads input from standard input (keyboard).
- Type-safe and formatted.

2. `std::cout` (console output)

- Writes output to standard output (console).
- Can be chained using `<<`.

3. `std::cerr` (standard error)

- Unbuffered output stream for error messages.
- Doesn't wait for buffer flush like `std::cout`.

4. `std::clog` (standard log)

- Buffered stream used for logging or diagnostics.

5. `std::ifstream` (file input)

- Reads from a file.

```
 std::ifstream fin("file.txt");
int x; fin >> x;
```

6. `std::ofstream` (file output)

- Writes to a file.

```
 std::ofstream fout("out.txt");
fout << "Hello";
```

7. `std::fstream` (file input + output)

- Reads and writes from the same file stream.

```
 std::fstream io("data.txt", std::ios::in | std::ios::out);
io << "data"; io >> val;
```

8. `std::stringstream` (string-based input/output)

- Allows treating strings as input/output buffers.

```
 std::stringstream ss;
ss << 123;
int x;
ss >> x;
```

9. `std::istringstream`

- Reads data from a string as an input stream.

```
 std::istringstream iss("42 3.14");
int a; float b;
iss >> a >> b;
```

10. `std::ostringstream`

- Writes data to a string.

```
 std::ostringstream oss;
oss << "result: " << 10;
std::string s = oss.str();
```

✅ Custom Overload Example

```
 class Point {
    int x, y;
public:
    Point(int x, int y) : x(x), y(y) {}
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        return os << "(" << p.x << ", " << p.y << ")";
    }
};
```

🧪 STL Debug Mode: `_GLIBCXX_DEBUG`

✅ What it does:

- Enables runtime checks in STL containers for **iterator validity**, **bounds**, and **memory safety**.
- Defined by compiling with:

```
g++ -D_GLIBCXX_DEBUG yourfile.cpp
```

🧠 Benefits:

- Detects use-after-free, invalid iterator dereference, concurrent container modifications.

⚠️ Notes:

- Adds performance overhead.
- Binary incompatible with non-debug STL code.
- Should be used in development, not production.

⚠️ Exception Handling in C++

✅ Syntax: `try` , `catch` , `throw`

```
try {
    if (some error)
        throw std::runtime_error("Error occurred");
} catch (const std::exception& e) {
    std::cerr << "Caught: " << e.what();
}
```

🔁 Stack Unwinding

- When an exception is thrown:
    - Control jumps to the nearest `catch` block.
    - All destructors for objects in scope are called in reverse order (stack unwinding).

✅ RAII-Based Exception Safety

- RAII ensures resources are released when objects go out of scope.
- No need for manual cleanup.

```
class File {
    std::ifstream f;
public:
    File(const std::string& name) { f.open(name); }
    ~File() { f.close(); } // guaranteed even during exception
};
```

🧠 Types of Standard Exceptions

Exception Type	Description
`std::exception`	Base class for all standard exceptions
`std::runtime_error`	Errors detectable only at runtime
`std::logic_error`	Violation of logical precondition
`std::invalid_argument`	Invalid arguments passed to functions
`std::out_of_range`	Access beyond bounds
`std::length_error`	Too large container
`std::bad_alloc`	Memory allocation failure
`std::bad_cast`	Invalid `dynamic_cast`
`std::bad_typeid`	`typeid` on null polymorphic ptr
`std::ios_base::failure`	I/O stream error

🔺 Best Practices

- Use standard exceptions where appropriate.
- Catch exceptions by reference (`catch (const std::exception& e)`).
- Prefer RAII for managing memory/files/locks.
- Avoid catching all exceptions via `catch(...)` unless in final fallback.

🚀 Low-Level Performance Tuning

✅ Cache-Friendly Data Layout

- Arrange data in memory to favor **spatial locality**.
- Avoid pointer chasing (e.g., AoS → SoA transformation).

```
// Bad: Array of structures (AoS)
struct Point { float x, y; } points[1000];

// Good: Structure of arrays (SoA)
float x[1000], y[1000];
```

✅ Inlining

- Reduces function call overhead.
- Use `inline` keyword or rely on compiler optimizations.
- Helps only for small, frequently called functions.

✅ Loop Unrolling

- Compiler expands loop body to reduce branching.
- Done manually or via flags like `-funroll-loops` (GCC).

```
// Original
for (int i = 0; i < 4; ++i) a[i] = b[i];

// Unrolled
a[0] = b[0]; a[1] = b[1]; a[2] = b[2]; a[3] = b[3];
```

✅ Branch Prediction

- Branch = conditional jump instruction (like `if`, `switch`, loop exits).
- Modern CPUs predict branch outcome to reduce pipeline stalls.
- Mis-predicted branches incur performance penalties.

```
if (likely(x > 0)) { ... } // GCC: __builtin_expect
```

🧠 Tips

- Minimize unpredictable branches in tight loops.
- Keep hot paths branch-free or predictable when possible.

🔁 Ranges Library (C++20)

✅ What is Ranges?

- Modern alternative to raw iterators/algorithms.
- Lazy, composable views using pipes (`|`).

```
 #include <ranges>
#include <vector>

std::vector<int> v = {1,2,3,4,5};
for (int x : v | std::views::filter([](int i){ return i % 2 == 0; })) {
    std::cout << x << " ";
}
```

✅ Common Views in `<ranges>`

View	Description
`filter`	Keep only elements satisfying predicate
`transform`	Apply a function to each element
`take(n)`	Take first `n` elements
`drop(n)`	Skip first `n` elements
`reverse`	Reversed iteration order
`join`	Flattens nested ranges
`split(delim)`	Splits a range by delimiter
`iota(start, end)`	Generate range from `start` to `end-1`
`enumerate`	Not standard, use `ranges::views::zip` + counter

✅ Benefits

- Lazy evaluation, efficient pipelines
- Works on STL containers and custom ranges
- Composable: can chain multiple views

```
auto result = vec | std::views::filter(...) | std::views::transform(...);
```

🧪 Casting in C++ (Detailed)

✅ `static_cast`

- Compile-time checked.

- Used for:
    - Numeric conversions (e.g., `double` → `int`)
    - Up/down casts (non-polymorphic)
    - Calling explicit constructors

```
 int i = static cast<int>(3.14);
Base* b = static_cast<Base*>(new Derived());
```

✅ `dynamic_cast`

- Runtime-checked for polymorphic base classes (with virtual functions).
- Returns `nullptr` if cast fails.

```
 Base* b = new Derived();
if (Derived* d = dynamic_cast<Derived*>(b)) { d->func(); }
```

✅ `const_cast`

- Used to **remove or add constness**.
- Cannot cast away const on const objects safely.

```
 void print(int* p);
const int* cp = ...;
print(const_cast<int*>(cp));
```

✅ `reinterpret_cast`

- Reinterpret the bit pattern of the object.
- Used for low-level conversions (e.g., between function/data pointers).
- **Highly unsafe unless you know exactly what you're doing.**

```
 int i = 65;
char* p = reinterpret_cast<char*>(&i); // now treat integer as char buffer
```

✅ C-style Cast

- Combines static, const, reinterpret into a single ambiguous operation.
- Dangerous; **use only in legacy code.**

```
int x = (int)3.14;
```

📊 Summary Table

Cast Type	Safety	Use Case
`static_cast`	✅ Safe	Type conversion, base-derived (non-RTTI)
`dynamic_cast`	✅ Runtime	RTTI polymorphic downcast
`const_cast`	⚠️ Careful	Add/remove const
`reinterpret_cast`	❌ Risky	Low-level bit manipulation
C-style cast	❌❌ Avoid	Legacy code; bypasses all checks

4️⃣ STL, Containers & Algorithms

📦 STL Containers: Time & Space Complexity

✅ `std::vector`

- **Dynamic array**
- Fast random access, amortized push_back.

Operation	Time Complexity
`push_back`	O(1) amortized
`insert/erase`	O(n)
`[]`, `at()`	O(1)
Memory Overhead	Contiguous memory

🧠 Memory Layout

```
| val0 | val1 | val2 | val3 | ... |
^ contiguous block in heap
```

✅ `std::deque`

- Double-ended queue
- Fast insertion/removal from both ends.

Operation	Time Complexity
`push_front/back`	O(1)
`[]`, `at()`	O(1)
insert/erase (mid)	O(n)

🧠 Memory Layout

```
 | block0 | block1 | block2 |
Each block is a contiguous array, managed as a circular buffer.
```

✅ `std::list`

- Doubly linked list
- Slow random access, efficient insertion/deletion.

Operation	Time Complexity
`push_front/back`	O(1)
`insert/erase`	O(1) (with iterator)
Access	O(n)

🧠 Memory Layout

```
 [Node] -> [Node] -> [Node]
Each node contains value + prev/next pointer
```

✅ `std::map`

- Ordered key-value (Red-Black Tree)
- Always sorted.

Operation	Time Complexity
Insert	O(log n)
Find/Erase	O(log n)

Operation	Time Complexity
Iteration	O(n) in order

🔧 Custom Comparator Example

```
std::map<int, std::string, std::greater<>> descending_map;
```

✅ `std::unordered_map`

- Hash table
- Fast average-time access.

Operation	Avg Time	Worst Time
Insert	O(1)	O(n)
Find/Erase	O(1)	O(n)

🔧 Custom Hash Example

```
 struct CustomHash {
    size_t operator()(const pair<int,int>& p) const {
        return hash<int>()(p.first) ^ hash<int>()(p.second);
    }
};

unordered_map<pair<int,int>, int, CustomHash> mp;
```

✅ `std::set`

- Ordered unique elements
- Red-Black Tree-based

Operation	Time Complexity
Insert	O(log n)
Find/Erase	O(log n)
Sorted	Yes

🔧 Set of Structs with Comparator

```
struct Data {
    int id;
    bool operator<(const Data& other) const {
        return id < other.id;
    }
};

std::set<Data> dataSet;
```

✅ `std::priority_queue`

- Max-heap by default
- Backed by a vector + make_heap

Operation	Time Complexity
Insert	O(log n)
Extract Max	O(log n)
Access Max/top	O(1)

🔧 Min-Heap Example

```
priority_queue<int, vector<int>, greater<>> minHeap;
```

🆚 `unordered_map` vs `map` : Comparison

Feature	`unordered_map`	`map`
Internal Structure	Hash Table	Red-Black Tree (Balanced BST)
Ordering	No	Sorted by key
Insertion Time	Avg O(1), Worst O(n)	O(log n)
Lookup Time	Avg O(1), Worst O(n)	O(log n)
Memory Usage	More (due to hash buckets)	Less (but with pointer overhead)
Use Case	Fast access with no order needed	Ordered traversal, range queries

🔧 Implementation Notes

- `unordered_map` uses **buckets + chaining** internally.
- `map` maintains **tree balance** during insertion/deletion.
- `unordered_map` may degrade to O(n) if hash collisions are severe.

🔁 Iterators, Ranges & Invalidation

✅ What is an Iterator?

- A **generalized pointer-like** object used to traverse containers.
- Types: `begin()`, `end()`, `rbegin()`, `const_iterator`, `reverse_iterator`

```
 std::vector<int> v = {1, 2, 3};
for (auto it = v.begin(); it != v.end(); ++it) std::cout << *it;
```

✅ Iterator Invalidation

- Happens when structure of container changes.

Container	When Invalidated
`vector`	On resize, insert, erase
`deque`	On insert/erase at ends
`list`	Safe: insert/erase does not invalidate others
`map/set`	Safe for all except erased iterator
`unordered_map`	On rehash (insertion past load factor)

⚠️ Example

```
 std::vector<int> v = {1, 2, 3};
auto it = v.begin();
v.push_back(4);   // may invalidate `it`
```

🔁 `push_back` vs `emplace_back`

✅ `push_back`

- Constructs a copy of the object and adds it to the container.
- Requires the object to be created first.

```
std::vector<std::string> v;
std::string s = "hello";
v.push_back(s); // copy s into vector
```

✅ `emplace_back`

- Constructs the object in-place inside the container.
- More efficient for complex types.

```
v.emplace_back("world"); // constructs string directly in vector
```

🧠 When to use

- Use `emplace_back` for performance-sensitive code where direct construction avoids copy/move.
- Prefer `push_back` if object is already constructed.

🧳 Passing STL Containers to Functions

✅ Default is pass-by-value

```
void print(std::vector<int> v); // copies the entire vector
```

✅ Use const reference to avoid copy

```
void print(const std::vector<int>& v); // read-only, no copy
```

✅ Use non-const reference to modify

```
void modify(std::vector<int>& v) {
    v.push_back(10);
}
```

✅ Applies to all containers: `map`, `queue`, `stack`, `set`, etc.

- STL containers are **not** passed by reference by default.
- Use `&` to pass by reference when needed.

🔧 STL Utility Operations

✅ `assign`

- Replace contents of container with specified value or range.

```
v.assign(5, 10); // five elements of value 10
```

✅ `fill`

- Fill a range with value.

```
std::fill(v.begin(), v.end(), 0);
```

✅ `iota`

- Fill range with sequentially increasing values.

```
std::iota(v.begin(), v.end(), 1); // 1, 2, 3...
```

✅ `clear`

- Removes all elements.

```
v.clear();
```

✅ `erase`

- Removes specific element or range.

```
v.erase(v.begin() + 2); // remove 3rd element
```

✅ `replace`

- Replace all matching elements.

```
std::replace(v.begin(), v.end(), 2, 5); // replace 2 with 5
```

✅ `substr`

- Substring from a string.

```
std::string s = "hello";
auto sub = s.substr(1, 3); // "ell"
```

🧹 `erase-remove` Idiom

✅ Problem:

- `std::remove` only moves values to the end. It doesn't erase them.

```
std::vector<int> v = {1, 2, 3, 2, 4};
// Incorrect:
v.erase(std::remove(v.begin(), v.end(), 2)); // ❌ compile error
```

✅ Correct idiom:

```
v.erase(std::remove(v.begin(), v.end(), 2), v.end());
```

- `std::remove` returns iterator to new logical end.
- `erase` trims off the tail.

📐 Common STL Algorithms

✅ `std::find_if`

Find first element matching a condition.

```
auto it = std::find_if(v.begin(), v.end(), [](int x){ return x % 2 == 0; });
```

✅ `std::accumulate`

Compute sum/product/etc.

```
int sum = std::accumulate(v.begin(), v.end(), 0);
```

✅ `std::all_of`, `std::any_of`, `std::none_of`

Check for all/any/none satisfying condition.

```
bool allEven = std::all_of(v.begin(), v.end(), [](int x){ return x % 2 == 0; });
```

📍 String Algorithms: `find_first_not_of`, `find_last_not_of`

✅ `find_first_not_of`

- Finds the **first character** in the string **not present** in the given set.

```
 std::string s = "aabc";
size_t pos = s.find_first_not_of("a"); // pos = 2
```

✅ `find_last_not_of`

- Finds the **last character** not in the set.

```
 std::string s = "aabcc";
size_t pos = s.find_last_not_of("c"); // pos = 3
```

📩 Deep vs Shallow Copy for Containers

✅ By Value (Deep Copy)

- Creates a full copy of the container and its elements.

```
void f(std::vector<int> v) { v[0] = 100; } // original unchanged
```

✅ By Reference (Shallow Copy / No Copy)

- Allows modification of original container.

```
void f(std::vector<int>& v) { v[0] = 100; } // modifies original
```

✅ By `const` Reference (Read-Only, No Copy)

- Prevents modification but avoids copy cost.

```
void f(const std::vector<int>& v) { std::cout << v[0]; }
```

📌 Notes:

- STL containers are always passed **by value unless explicitly marked**.
- Copying large containers is expensive.

- Prefer `const &` for read-only and `&` for mutability.

🔁 `next_permutation` and `prev_permutation`

✅ Header

```
#include <algorithm>
```

✅ `next_permutation`

- Rearranges the elements into the **next lexicographically greater** permutation.
- Returns `false` if no next permutation exists.

```
 std::vector<int> v = {1, 2, 3};
do {
    for (int i : v) std::cout << i;
    std::cout << "\n";
} while (std::next_permutation(v.begin(), v.end()));
```

✅ `prev_permutation`

- Rearranges to **previous lexicographical order**.

```
 std::vector<int> v = {3, 2, 1};
while (std::prev_permutation(v.begin(), v.end())) { ... }
```

🧠 Best Practice

- Useful for brute-force/backtracking problems.
- Ensure input is sorted (for `next_permutation`) or reverse-sorted (for `prev_permutation`).

⚙️ Bit Manipulation

✅ Common Builtins

```
 __builtin_popcount(x);      // count set bits in int
__builtin_clz(x);           // count leading zeros
__builtin_ctz(x);           // count trailing zeros
__builtin_parity(x);        // returns 1 if set bits are odd
```

- Available in GCC/Clang. Use `<bit>` header in C++20+ for portable functions.

✅ Example: Check if power of 2

```
bool isPowerOfTwo(int x) {
    return x && !(x & (x - 1));
}
```

📥 `stringstream`, `cin.clear()`, Buffer Flushing, and Input Utilities

✅ `stringstream`

- A class from `<sstream>` used for **in-memory text parsing**, similar to `cin` / `cout`, but with strings.
- Can be used to tokenize strings, convert between strings and other types.

```
std::stringstream ss("42 3.14 hello");
int x; float y; std::string word;
ss >> x >> y >> word; // parses just like cin
```

- You can insert and extract like `cin` / `cout`:

```
std::stringstream s;
s << 123 << " " << 3.14;
std::string result = s.str(); // "123 3.14"
```

- Resetting a stringstream:

```
ss.str("");     // clear content
ss.clear();     // reset error flags
```

✅ `cin.*` Methods

- `cin.clear()` – clears error state bits like `failbit`, `eofbit`.
- `cin.ignore(n, delim)` – skips `n` characters or until `delim`.
- `cin.get()` – gets a single char (including whitespace).
- `cin.getline()` – reads an entire line including spaces.
- `cin.peek()` – checks next character without removing it from buffer.
- `cin.fail()` – returns true if last input operation failed.

```
int x;
std::cin >> x;
if (std::cin.fail()) {
    std::cin.clear();             // reset error flags
```

```
    std::cin.ignore(INT MAX, '
'); // discard bad input
}
```

✅ Buffer Flushing

```
std::cout << std::flush;  // flushes buffer
std::cout << std::endl;   // flushes + newline
```

- Flush ensures output appears immediately.

🧰 `pair`, `tuple`, and Structured Bindings

✅ `std::pair`

```
std::pair<int, std::string> p = {1, "hello"};
auto [id, name] = p; // structured binding
```

✅ `std::make_pair`

- Type deduction helper.

```
auto p = std::make_pair(1, "hi");
```

✅ `std::tuple`

```
std::tuple<int, float, std::string> t = {1, 3.14, "pi"};
auto [a, b, c] = t; // C++17 structured binding
```

✅ Accessing elements

```
std::get<0>(t); // get first element
```

🧠 Use Case

- Use `pair` for 2 elements, `tuple` for more.
- Prefer structured bindings over `get<>()` where possible.

✂️ String Manipulation Functions

✅ Accessors / Queries

- `size()`, `length()` – get number of characters
- `empty()` – check if string is empty
- `front()`, `back()` – get first/last character
- `at(i)` – bounds-checked access

✅ Modifiers

- `push_back(char)` – add character at end
- `pop_back()` – remove last character
- `insert(pos, str)` – insert string at position
- `erase(pos, len)` – remove substring
- `replace(pos, len, str)` – replace portion with another
- `clear()` – remove all contents
- `resize(n)` – change size, pads if needed
- `swap(str)` – swap with another string

✅ Find / Search

- `find(str)` – returns first occurrence index or `npos`
- `rfind(str)` – last occurrence
- `find_first_of(chars)` – index of any of `chars`
- `find_first_not_of(chars)` – index of first char not in `chars`
- `substr(pos, len)` – returns a substring

🔗 `std::span` (C++20) – In-Depth

✅ What is `std::span`?

- A **non-owning view** over a contiguous block of memory.
- Safer alternative to raw pointers.
- Requires `<span>` header.

```
 #include <span>
void print(std::span<int> data) {
    for (int x : data) std::cout << x << " ";
}

std::vector<int> v = {1, 2, 3};
print(v); // automatic conversion
```

🧠 Benefits

- Prevents buffer overflows by tracking size
- Eliminates the need for separate pointer and size parameters
- Safer than raw pointers but just as efficient
- Interoperable with C-style arrays, `std::array`, `std::vector`
- Enables slicing using `first`, `last`, or `subspan()`

```
std::array<int, 5> arr = {10, 20, 30, 40, 50};
std::span<int> s1(arr);           // entire array
std::span<int> s2 = s1.subspan(1); // skip first
std::span<int> s3 = s1.first(3);  // first 3 elements
std::span<int> s4 = s1.last(2);   // last 2 elements
```

📌 Notes

- `span` is non-owning: original container must outlive the span.
- Avoid dangling span references to temporaries.
- `span<T>` vs `span<const T>` controls read/write access.

```
std::span<int> sub = s.subspan(1, 2); // elements 1 and 2
```

⚡ `std::execution` (C++17/20) – Parallel Algorithms

✅ What is it?

- Header `<execution>` provides **execution policies** for STL algorithms:
  - `seq` – sequential (default)
  - `par` – parallel
  - `par_unseq` – parallel + vectorized

```
#include <execution>
std::sort(std::execution::par, v.begin(), v.end());
```

🧠 Use Cases

- Improve performance of sorting, transformation, counting, etc. over large data
- Combine with STL algorithms like `sort`, `transform`, `for_each`, `reduce`

```
std::vector<int> nums(1000000);
std::iota(nums.begin(), nums.end(), 0);

// Parallel transform
```

```
std::transform(std::execution::par, nums.begin(), nums.end(), nums.begin(),
               [](int x) { return x * x; });
```

📌 Policies Summary

Policy	Description
`seq`	Sequential (default STL behavior)
`par`	Parallel using threads
`par_unseq`	Parallel + SIMD vectorization

⚠️ Notes

- Not all algorithms support parallelism
- Requires support in compiler + standard library
- May cause race conditions if writing to shared memory

5️⃣ Templates & Meta Programming

🧩 Function & Class Templates

✅ Function Templates

Templates allow generic functions that work with any type.

```
 template <typename T>
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}

int x = maxVal(3, 7);      // works for int
double y = maxVal(2.5, 9); // works for double
```

- `typename` and `class` are interchangeable in this context.
- Type is inferred unless explicitly specified.

✅ Class Templates

```
 template <typename T>
class Stack {
    std::vector<T> v;
  public:
    void push(T x) { v.push_back(x); }
    void pop()     { v.pop_back(); }
    T top() const  { return v.back(); }
};

Stack<int> s1;
s1.push(42);
```

🧠 Template Specialization & Partial Specialization – In-Depth

✅ Full Specialization

Used when a specific implementation is needed for a specific type.

```
 template <typename T>
class Printer {
  public:
    void print(T val) { std::cout << val << "\n"; }
};

template <>
class Printer<std::string> {
  public:
    void print(std::string val) {
        std::cout << "[str] " << val << "\n";
    }
};
```

✅ Partial Specialization

Specializes only some parameters.

```
 template <typename T, typename U>
class Pair {};

template <typename T>
class Pair<T, int> {}; // specialize when second type is int
```

🧠 Detailed Explanation

- **Full Specialization:**

  - Overrides the template entirely for a specific type.
  - Used when a type needs completely custom logic.
  - Must match the template parameter list exactly.

```
 template <typename T>
class Box {
  public:
    void print() { std::cout << "Generic Box
"; }
};

template <>
class Box<int> {
  public:
    void print() { std::cout << "Box for int
"; }
};
```

- **Partial Specialization:**

  - Only customizes part of a generic template.
  - Allows maintaining general behavior while handling edge cases.

```
 template <typename T, typename U>
class Box {
  public:
    void show() { std::cout << "Generic
"; }
};

template <typename T>
class Box<T, int> {
  public:
    void show() { std::cout << "Int specialization
"; }
};
```

- **Function Template Specialization (not overload)**

  - You can't partially specialize a function template, only fully specialize or overload.

```
 template <typename T>
void print(T val) {
    std::cout << val << "
";
```

```
}

template <>
void print<std::string>(std::string val) {
    std::cout << "String: " << val << "
";
}
```

🧠 Tips

- Prefer full specialization for performance-optimized or hard-typed logic.
- Partial specialization is useful for SFINAE/meta-programming.
- Avoid ambiguous partial matches.
- You can't partially specialize a function template—use overload or `if constexpr` instead.

🔁 Variadic Templates

✅ What are Variadic Templates?

Templates that accept **a variable number of arguments**.

```
 template<typename T>
void print(T t) {
    std::cout << t << " ";
}

template<typename T, typename... Args>
void print(T t, Args... args) {
    std::cout << t << " ";
    print(args...); // recursion
}
```

🧠 Uses

- Logging frameworks
- Tuple/parameter pack expansions
- Generic forwarding

✅ Fold Expression (C++17)

```
 template<typename... Args>
void printAll(Args... args) {
```

```
    (std::cout << ... << args) << '\n';
}
```

🧠 SFINAE & `std::enable_if`

✅ What is SFINAE?

Substitution Failure Is Not An Error: if template instantiation fails, compiler tries other overloads.

✅ `std::enable_if` (C++11)

Enable/disable template based on a condition.

```
 template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
process(T val) {
    std::cout << "Integral: " << val << '\n';
}

template<typename T>
typename std::enable_if<!std::is_integral<T>::value>::type
process(T val) {
    std::cout << "Non-integral: " << val << '\n';
}
```

🧠 Tips

- Useful to enforce compile-time constraints
- Often used in generic libraries for overload control
- Alternative in C++17: `if constexpr`

🧠 Concepts (C++20)

✅ What are Concepts?

Type constraints for templates — make errors clearer and templates safer.

✅ Example:

```
 #include <concepts>

template<typename T>
```

```
concept Integral = std::is_integral_v<T>;

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

✅ Built-in Concepts:

- `std::integral`, `std::floating_point`, `std::same_as<T>`, `std::derived_from<Base>`

🧠 Benefits

- Better compile-time error messages
- Replaces complex SFINAE chains
- Enhances template readability

🧠 Type Traits & Detection Idioms

✅ Type Traits (from `<type_traits>`)

- Compile-time type inspection utilities

Trait	Description
`std::is_integral<T>`	Checks if T is integral type
`std::is_same<T, U>`	Checks if types are exactly same
`std::is_base_of<B, D>`	Checks if B is base of D
`std::remove_reference<T>`	Strips reference
`std::decay<T>`	Strips cv-ref and arrays

🧠 Detection Idiom (C++17+)

Used to conditionally detect whether a class/member/function exists.

```
 template<typename T>
using has_push_back_t = decltype(std::declval<T>().push_back(std::declval<int>()));

template<typename, typename = void>
struct has_push_back : std::false_type {};
```

```
template<typename T>
struct has_push_back<T, std::void_t<has_push_back_t<T>>> : std::true_type {};
```

🧠 CRTP – Curiously Recurring Template Pattern

✅ What is CRTP?

A class derives from a template instantiation of itself.

```
 template<typename Derived>
class Base {
  public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class Derived : public Base<Derived> {
  public:
    void implementation() { std::cout << "Derived impl\n"; }
};
```

🧠 Use Cases

- Static (compile-time) polymorphism
- CRTP avoids virtual dispatch cost
- Used in libraries like **Eigen**, **Boost**, **Curiously Recurring Visitor**, etc.

🧬 CRTP in Eigen (Library Example)

🔍 About Eigen

- Eigen is a C++ template library for **linear algebra**.
- It heavily uses **CRTP** to implement expression templates.

✅ How Eigen Uses CRTP

Eigen matrices inherit from `Eigen::MatrixBase<Derived>` where `Derived` is the specific matrix type.

```
class MyMatrix : public Eigen::MatrixBase<MyMatrix> {
 // implements required methods
```

```
};
```

🧠 Why CRTP in Eigen?

- Avoids runtime overhead of virtual dispatch
- Enables chaining expressions like:

```
C = A + B + D;
```

- Instead of evaluating intermediate results, Eigen builds an **expression tree** at compile time and fuses operations.

✅ Benefits

- Extremely fast matrix computations
- Compile-time optimizations
- Cleaner syntax using operator overloading
- No dynamic memory allocations unless explicitly required

✅ Benefits

- Zero-cost abstraction
- Enables method overriding without virtual
- Type-safe interface enforcement

6️⃣ Memory Management

🧱 `new` / `delete` vs `malloc` / `free`

✅ `new` and `delete` (C++-style)

- `new` allocates memory **and calls constructor**
- `delete` deallocates memory **and calls destructor**

```
int* p = new int(5); // alloc + construct
...
delete p;           // destruct + dealloc
```

- Type-safe: No need to cast
- Works with classes/objects

✅ `malloc` and `free` (C-style)

- `malloc` only allocates raw memory
- `free` only deallocates — does **not** call destructor

```
int* p = (int*)malloc(sizeof(int));
...
free(p);
```

- Not type-safe: returns `void*`, must be cast
- Doesn't support constructors/destructors

⚠️ Never mix `new` with `free`, or `malloc` with `delete`

- Leads to undefined behavior!

🧠 Summary Table

Feature	`new/delete`	`malloc/free`
Type-safe	✅	❌
Constructor/Destructor	✅	❌
Returns proper type	✅	❌
C++ only	✅	❌ (C compatible)

🧠 Stack vs Heap Memory

✅ Stack

- Memory allocated in function scope
- Automatically managed (freed on return)
- Fast to allocate/deallocate
- Limited size (\~1MB to few MBs)

```
void func() {
   int a = 10; // stored on stack
} // a is destroyed automatically
```

✅ Heap

- Dynamically allocated using `new` / `malloc`
- Manual memory management required
- Larger pool of memory (\~GBs)

```
int* a = new int(42); // heap allocation
```

📌 Differences

Feature	Stack	Heap
Lifetime	Until function ends	Until explicitly freed
Allocation	Automatic	Manual
Speed	Fast	Slower
Size	Limited	Large

♻️ RAII Idiom (Resource Acquisition Is Initialization)

✅ What is RAII?

RAII binds resource management to object lifetime. When an object goes out of scope, its destructor automatically releases resources.

```
 class File {
    FILE* handle;
  public:
    File(const char* filename) { handle = fopen(filename, "r"); }
    ~File() { if (handle) fclose(handle); }
};

void readFile() {
    File f("data.txt"); // opens on construction, closes on destruction
}
```

🧠 Benefits

- No leaks
- Exception safe
- Deterministic cleanup

🔒 Smart Pointers in C++11+

✅ `std::unique_ptr`

- Represents **exclusive ownership** of a dynamically allocated object.
- When `unique_ptr` goes out of scope, it automatically deletes the object it owns.
- Cannot be copied, only moved.

```
std::unique_ptr<int> p = std::make_unique<int>(10);
```

🧠 Notes

- Use `std::move` to transfer ownership:

```
std::unique_ptr<int> q = std::move(p);
```

- Custom deleters are supported via constructor arguments.
- Automatically deletes object when out of scope

```
std::unique_ptr<int> p = std::make_unique<int>(10);
```

✅ `std::shared_ptr`

- Represents **shared ownership** over a dynamically allocated object.
- Maintains an internal reference count (thread-safe).
- Object is destroyed only when last `shared_ptr` to it is destroyed.

```
 std::shared_ptr<int> a = std::make_shared<int>(10);
std::shared_ptr<int> b = a; // count = 2
```

🧠 Notes

- Use `.use_count()` to check reference count
- Avoid circular references (use `weak_ptr`)
- Can be created using `std::make_shared<T>()` which is more efficient
- Reference count is maintained

```
 std::shared_ptr<int> a = std::make_shared<int>(10);
std::shared_ptr<int> b = a; // count = 2
```

✅ `std::weak_ptr`

- A non-owning smart pointer that **observes** a `shared_ptr`

- Does **not** affect reference count
- Used to break **cyclic references**

```
 std::shared_ptr<int> sp = std::make_shared<int>(100);
std::weak_ptr<int> wp = sp; // does not increase use_count

if (auto locked = wp.lock()) {
    std::cout << *locked;
}
```

🧠 Notes

- Must call `.lock()` to safely access the object
- Returns `nullptr` if the observed object has expired of `shared_ptr`
- Doesn't affect ref count; used to break cycles

```
std::weak_ptr<int> wp = a;
```

📌 Comparison Table

Pointer Type	Ownership	Ref Count	Thread-safe	Use Case
`unique_ptr`	Single	No	Yes	Exclusive resource
`shared_ptr`	Shared	Yes	Yes	Shared ownership
`weak_ptr`	None	No	Yes	Cache, avoid cycles

🔍 Accessing `shared_ptr` Reference Count

You can use `.use_count()` to check how many `shared_ptr` instances point to the same object:

```
 std::shared_ptr<int> sp1 = std::make_shared<int>(42);
std::shared_ptr<int> sp2 = sp1;
std::cout << sp1.use_count(); // Outputs 2
```

🛠️ Manual Implementation of `shared_ptr`

```
 template <typename T>
class MySharedPtr {
    T* ptr;
    int* ref_count;
```

```
  public:
    MySharedPtr(T* p) : ptr(p), ref_count(new int(1)) {}
    MySharedPtr(const MySharedPtr& other) {
        ptr = other.ptr;
        ref count = other.ref_count;
        (*ref_count)++;
    }
    ~MySharedPtr() {
        if (--(*ref_count) == 0) {
            delete ptr;
            delete ref_count;
        }
    }
};
```

⚠️ Pitfalls

- Thread safety: `ref_count` updates must be atomic
- Cyclic references leak memory (use `weak_ptr`)
- Manual management is error-prone compared to STL versions

🧪 `memset`

✅ What it Does

- Fills a block of memory with a specific value (usually 0 or -1)
- Defined in `<cstring>`

```
 int arr[5];
std::memset(arr, 0, sizeof(arr));
```

⚠️ Notes

- Works **byte-wise** — careful when using with non-`char` types
- May cause issues with objects with virtual tables or constructors

---

🧬 Deep Copy vs Shallow Copy

✅ Shallow Copy

- Copies pointers as-is

- Multiple objects point to same memory → shared ownership (may cause double free)

```
class A {
   int* data;
   A(const A& other) { data = other.data; } // shallow
};
```

✅ Deep Copy

- Allocates new memory and copies contents
- Each object manages its own copy

```
class A {
   int* data;
   A(const A& other) {
       data = new int(*other.data); // deep
   }
};
```

🧠 Tips

- Implement deep copy via copy constructor and assignment operator
- Always consider ownership semantics

🚚 Copy and Move Semantics

✅ Copy Constructor & Assignment

- Creates a new object with the **same content**
- Used when you pass/return by value

✅ Move Constructor & Assignment (C++11)

- Transfers ownership instead of copying
- Source object is left in a valid but unspecified state

```
class A {
   std::string s;
 public:
   A(std::string str) : s(std::move(str)) {}
};
```

🚀 `std::move` and `std::forward`

✅ `std::move`

- Casts an lvalue to an rvalue to enable move constructor
- Does **not** move by itself — just allows moving

```
 std::string s = "hello";
std::string t = std::move(s); // invokes move ctor
```

✅ `std::forward`

- Perfect forwarding: preserves lvalue/rvalue nature
- Used in template functions with universal references

```
 template <typename T>
void wrapper(T&& arg) {
    process(std::forward<T>(arg));
}
```

🔄 Move Semantics, Rvalue References, and Copy Elision

✅ Rvalue References (`&&`)

- Allows distinguishing between temporary (rvalue) and persistent (lvalue) objects
- Enables efficient move operations without copying

```
 void setName(std::string&& name) {
    this->name = std::move(name); // moves resource
}
```

✅ Move Constructors & Move Assignment

- Move Constructor: `ClassName(ClassName&&)`
- Move Assignment: `operator=(ClassName&&)`

```
class Buffer {
    char* data;
 public:
    Buffer(Buffer&& other) noexcept : data(other.data) {
        other.data = nullptr;
```

```
    }
    Buffer& operator=(Buffer&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }
};
```

🧠 Copy Elision & NRVO (Named Return Value Optimization)

NRVO (Named Return Value Optimization) is a compiler optimization that avoids creating unnecessary temporary objects during return-by-value. Instead of creating a temporary object and then copying/moving it, the object is directly constructed in the memory location of the caller's receiving variable.

✅ Example:

```
Buffer makeBuffer() {
    Buffer buf;
    return buf; // Compiler uses NRVO to construct 'buf' directly in the caller's space
}
```

- Even if copy/move constructor exists, the compiler **elides** it when safe.

🧠 C++17: Mandatory Copy Elision

Since C++17, copy elision is mandatory in certain cases such as:

- Returning a local object
- Returning a temporary object directly

🚀 Benefits of NRVO

- Zero-cost return optimization
- Eliminates unnecessary move/copy calls
- Essential for performance-critical APIs

🔐 Ownership Semantics

✅ What is Ownership?

- Defines which object is responsible for managing a resource
- C++ enforces this via smart pointers, move semantics, and RAII

🧠 Rules

- Only one `unique_ptr` should own a resource
- `shared_ptr` uses reference counting
- Copying should duplicate or deep copy
- Moving should transfer ownership and null the source

🧠 Good Practice

- Prefer moving resources rather than copying large data
- Clearly express ownership via smart pointers
- Avoid raw `new` / `delete` — prefer RAII and containers

🧨 Dangling Pointers

✅ What is a Dangling Pointer?

A pointer that **points to a memory location which has been freed or is invalid**.

🔥 Example:

```
 int* p = new int(42);
delete p;
*p = 10; // Undefined behavior: dangling pointer access
```

⚠️ Causes:

- Deleting memory and using pointer afterward
- Returning address of a local variable
- Misuse of raw pointers

🛡️ Prevention:

- Use smart pointers (`unique_ptr` , `shared_ptr`)
- Set pointer to `nullptr` after `delete`
- Avoid raw pointers unless necessary

🧱 Placement New, Alignment, and Padding

✅ Placement New

Constructs an object at a **pre-allocated memory location**.

```
 char buffer[sizeof(int)];
int* p = new (buffer) int(5); // placement new
```

- Requires manual destructor call: `p->~int();`

🧠 Use Cases:

- Embedded systems
- Custom allocators
- Performance-critical code avoiding dynamic memory

✅ Alignment

Ensures that objects are placed at memory addresses suitable for the CPU.

```
 struct alignas(16) MyAligned {
    int x;
};
```

✅ Padding

Compiler adds invisible bytes to meet alignment requirements.

```
struct A { char a; int b; }; // likely 8 bytes (3 padding)
```

🐛 Memory Leaks, Detection, and Circular References

✅ Memory Leak

Allocated memory that is never freed, leading to gradual memory exhaustion.

```
 void leak() {
    int* a = new int[100];
    // no delete → memory leak
}
```

🛠️ Detection Tools:

- `valgrind` (Linux)

- `Visual Leak Detector`, `CRT debug heap` (Windows)
- AddressSanitizer (GCC/Clang)

✅ Fixes:

- Use smart pointers to automate cleanup
- Ensure every `new` has a matching `delete`
- Avoid ownership ambiguity

🔁 Circular Reference Problem:

Occurs when two `shared_ptr`s refer to each other, preventing ref count from dropping to 0.

```
 struct A;
struct B {
    std::shared_ptr<A> a;
};
struct A {
    std::shared_ptr<B> b;
};
```

- Memory is never released
- Fix: make one of them a `weak_ptr`

7️⃣ Concurrency & Multithreading

🧵 std::thread

✅ What it Does

- Launches a new thread of execution.
- Constructor accepts a function/lambda and arguments.

```
 #include <thread>

void work(int id) {
    std::cout << "Working on thread " << id << '\n';
}

std::thread t(work, 1);
t.join();
```

- `join()` blocks until thread finishes
- `detach()` lets it run independently (risky if not managed)

🔐 std::mutex, std::lock_guard, std::scoped_lock

✅ std::mutex

Ensures only one thread accesses critical section.

```
 std::mutex mtx;
mtx.lock();
// critical section
mtx.unlock();
```

✅ std::lock_guard

RAII-style lock, auto unlocks on scope exit.

```
std::lock_guard<std::mutex> lock(mtx);
```

✅ std::scoped_lock (C++17)

Supports multiple mutexes and deadlock avoidance.

```
std::scoped_lock lock(mtx1, mtx2);
```

🔄 std::condition_variable

✅ Purpose

Used to block threads until a condition is met.

```
 std::mutex mtx;
std::condition_variable cv;
bool ready = false;

// Thread 1
std::unique_lock<std::mutex> lock(mtx);
cv.wait(lock, []{ return ready; });

// Thread 2
{
    std::lock_guard<std::mutex> lock(mtx);
```

```
    ready = true;
}
cv.notify_one();
```

⚛️ std::atomic

✅ Atomic Types

Ensure lock-free, thread-safe variable access.

```
 std::atomic<int> counter = 0;

void increment() {
    counter++;
}
```

🧠 Benefits

- No explicit locking
- Prevents data races
- Essential for performance in multi-threaded code

🧊 Thread-Safe Singleton

🧠 What is Singleton?

A **creational design pattern** that ensures a class has **only one instance** and provides a **global point of access** to that instance.

📦 Key Characteristics

- **Lazy Initialization**: Instance is created when needed
- **Global Access**: Access via static method `getInstance()`
- **Single Instance**: Maintains internal static pointer or reference

🔍 Typical Use Cases

- **Logger**: Shared logging instance across threads or modules
- **Configuration Manager**: Central access to app settings
- **Database Connection Pool**: Manage limited DB connections
- **Cache Manager**: Maintain shared memory cache

✅ Classic Singleton Issue

- Not thread-safe without protection.
- Risk of creating multiple instances under race.

✅ Thread-Safe Singleton (C++11+)

```
class Singleton {
 public:
   static Singleton& getInstance() {
       static Singleton instance; // thread-safe in C++11+
       return instance;
   }
 private:
   Singleton() {}
   Singleton(const Singleton&) = delete;
   Singleton& operator=(const Singleton&) = delete;
};
```

- Leverages function-local static initialization (guaranteed thread-safe)

🧩 Thread-Safe Design & OOP Integration

🧠 Key Concepts

- Encapsulate locking logic inside classes
- Use RAII wrappers for synchronization
- Avoid shared mutable state

✅ Example: Thread-Safe Counter Class

```
class ThreadSafeCounter {
   std::mutex m;
   int value = 0;
 public:
   void increment() {
       std::lock_guard<std::mutex> lock(m);
       value++;
   }
   int get() {
       std::lock_guard<std::mutex> lock(m);
       return value;
   }
};
```

📦 Concurrent Queue (Thread-Safe Queue)

✅ Motivation

Used for communication between threads (e.g. producer/consumer)

✅ Basic Implementation

```
 #include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class ConcurrentQueue {
    std::queue<T> q;
    std::mutex m;
    std::condition_variable cv;
  public:
    void push(const T& val) {
        {
            std::lock_guard<std::mutex> lock(m);
            q.push(val);
        }
        cv.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(m);
        cv.wait(lock, [&]{ return !q.empty(); });
        T val = q.front(); q.pop();
        return val;
    }
};
```

🔁 Object Pool Pattern

✅ Goal

Reuse a fixed number of preallocated objects to reduce allocation overhead.

✅ Example Pattern

```
 template<typename T, size_t Size>
class ObjectPool {
    std::vector<T*> pool;
```

```
    std::stack<T*> free_list;

  public:
    ObjectPool() {
        for (size t i = 0; i < Size; ++i) {
            T* obj = new T();
            pool.push_back(obj);
            free_list.push(obj);
        }
    }

    T* acquire() {
        if (free_list.empty()) return nullptr;
        T* obj = free_list.top(); free_list.pop();
        return obj;
    }

    void release(T* obj) {
        free_list.push(obj);
    }

    ~ObjectPool() {
        for (T* obj : pool) delete obj;
    }
};
```

🔁 Read/Write Locks

✅ What Are They?

Allow **multiple readers** or **one writer**, but not both at the same time. Used to improve concurrency when reads are frequent and writes are rare.

🔧 In C++

Use `std::shared_mutex` from `<shared_mutex>`

```
 #include <shared mutex>
std::shared_mutex rw_mutex;

void reader() {
    std::shared_lock lock(rw_mutex); // multiple readers allowed
}

void writer() {
    std::unique_lock lock(rw_mutex); // exclusive writer lock
}
```

🚀 Lock-Free Queues

✅ Concept

Data structures that allow concurrent access **without locks**, using atomic operations. Used where high throughput is required and locks are a bottleneck.

🧠 Example Use Cases

- Task schedulers
- Real-time systems
- Multi-producer/multi-consumer queues

🛠️ Notes

- Implemented using `std::atomic`, CAS (Compare-And-Swap)
- Hard to implement correctly — usually use libraries like:
  - Intel TBB
  - Boost Lockfree
  - moodycamel's `ConcurrentQueue`

⛓️ Semaphores and Barriers

✅ Semaphores

Counted signaling mechanism to **control access** to resources. C++20 adds `std::counting_semaphore`.

```
 #include <semaphore>
std::counting_semaphore<1> sem(1);

void worker() {
    sem.acquire(); // P()
    // critical section
    sem.release(); // V()
}
```

✅ Barriers

Used to **synchronize multiple threads** at a certain point. C++20 provides `std::barrier`.

```
#include <barrier>
std::barrier sync_point(3); // wait for 3 threads

void thread_func() {
    // do work
    sync_point.arrive_and_wait();
    // resume after all threads reach
}
```

🧨 Data Races

✅ What is a Data Race?

Occurs when **two or more threads access the same memory location concurrently**, and at least one of them writes without proper synchronization.

🔥 Example:

```
int counter = 0;
void thread func() {
    counter++; // potential data race
}
```

🛠️ How to Resolve:

- Use `std::mutex`, `std::atomic`, or `std::lock_guard`
- Avoid shared mutable state

🧠 Types:

- **Read/Write Race**: One thread reads from a memory location while another writes to it without synchronization.
- **Write/Write Race**: Two or more threads write to the same memory location simultaneously.
- **Read/Modify/Write Race**: A thread reads a value, modifies it, and writes it back without guarding it from other modifications in between.
- **Check-Then-Act Race**: A thread checks a condition and then acts on it, but the condition may change between check and act due to concurrent modifications.

🔒 Deadlocks

✅ What is a Deadlock?

Two or more threads wait on each other indefinitely due to **circular lock dependencies**.

🔥 Example:

```
 std::mutex a, b;

void thread1() {
    std::lock_guard<std::mutex> lock1(a);
    std::this_thread::sleep_for(10ms);
    std::lock_guard<std::mutex> lock2(b); // potential deadlock
}

void thread2() {
    std::lock_guard<std::mutex> lock2(b);
    std::lock_guard<std::mutex> lock1(a);
}
```

🛠️ How to Resolve:

- Always acquire locks in a consistent order
- Use `std::scoped_lock` to avoid deadlock
- Use `std::lock()` with `std::adopt_lock`

🧠 Types:

- **Mutual Deadlock**: Two or more threads wait on each other in a circular chain (A waits on B, B waits on A).
- **Hold and Wait Deadlock**: A thread holds one resource and waits for another, which is held by a different thread.
- **No Preemption Deadlock**: Resources cannot be forcibly taken from threads holding them.
- **Circular Wait Deadlock**: A closed chain of threads exists, where each thread holds a resource needed by the next.
- **Resource Starvation**: A thread is perpetually denied access to a needed resource due to biased scheduling or priority inversion.

🧬 False Sharing & Cache Line Effects

✅ What is False Sharing?

Occurs when multiple threads modify **different variables** that reside on the **same cache line**, causing unnecessary invalidation.

🔥 Example:

```
struct alignas(64) Padded {
    int a;
    char padding[60];
    int b;
};
```

- Both `a` and `b` are padded to avoid false sharing

🧠 Why It Happens:

- CPUs cache memory in blocks (typically 64 bytes)
- Even unrelated data in the same cache line causes performance hits if accessed by multiple threads

🛠️ Fixes:

- Use padding (`alignas` , char buffers)
- Use tools like `perf` , `VTune` to profile
- Prefer `std::hardware_destructive_interference_size` (C++17+)

📝 RAII + Thread Management

✅ Concept

RAII (Resource Acquisition Is Initialization) helps manage thread lifetimes safely using objects. Threads are joined or detached in the destructor, preventing resource leaks.

🔧 Example

```
class ThreadGuard {
    std::thread& t;
  public:
    explicit ThreadGuard(std::thread& t_) : t(t_) {}
    ~ThreadGuard() {
        if (t.joinable()) t.join();
    }
    ThreadGuard(const ThreadGuard&) = delete;
    ThreadGuard& operator=(const ThreadGuard&) = delete;
};
```

```
void task() { std::cout << "Running...\n"; }

int main() {
    std::thread t(task);
    ThreadGuard g(t);
    // no need to manually join
}
```

🧠 Benefits

- Prevents forgetting to `join()`
- Avoids leaks and premature terminations
- Composes well with exceptions and scope exits

🚀 Parallel Algorithms (C++17+)

✅ What Are They?

C++17 introduced parallel STL algorithms with execution policies to enable multicore speed-up.

```
 #include <execution>
#include <algorithm>

std::vector<int> v = {1, 2, 3, 4};
std::sort(std::execution::par, v.begin(), v.end());
```

✅ Execution Policies

- `std::execution::seq` : Sequential (default)
- `std::execution::par` : Parallel
- `std::execution::par_unseq` : Parallel + Vectorized

🧠 Notes

- Works on many STL algorithms like `sort` , `transform` , `for_each`
- Requires safe, side-effect-free operations
- May improve performance on multicore CPUs

🧰 Type Erasure (e.g., `std::function`, `std::any`, `std::variant`)

✅ What is Type Erasure?

A technique that allows you to **store and manipulate different types** through a **common interface** without knowing the exact type at compile time.

🔍 Real-world Example: `std::function`

`std::function` is a **type-erased, polymorphic wrapper** for any callable object — functions, lambdas, function pointers, or bind expressions. It can store, copy, and invoke them uniformly.

🔬 Difference from Normal Functions / Lambdas

Feature	`std::function`	Regular Function Pointer / Lambda
Type Erased	✅ Yes	❌ No — concrete type required
Runtime Polymorphism	✅ Can wrap any callable signature	❌ Only matches exact signature
Capturing Lambdas	✅ Can wrap capturing lambdas	❌ Only non-capturing (for function ptrs)
Copyable	✅ Copyable and assignable	❌ Capturing lambdas usually not copyable
Overhead	❗ Small (heap allocation + vtable lookup)	✅ Zero overhead (inlined, fast)

📌 When to Use

- When storing heterogeneous callables in containers
- When deferring execution (callbacks, event handlers)
- When writing generic APIs that accept arbitrary callables

```
 std::function<void()> f;
f = []() { std::cout << "Hello!\n"; };
```

```
f();
```

- Internally stores any callable object using a type-erased wrapper.
- Uses virtual dispatch behind the scenes.

🔧 Custom Type-Erased Wrapper Example

```
 class CallableBase {
  public:
    virtual void call() = 0;
    virtual ~CallableBase() {}
};

template <typename F>
class CallableImpl : public CallableBase {
    F func;
  public:
    CallableImpl(F&& f) : func(std::forward<F>(f)) {}
    void call() override { func(); }
};

class MyFunction {
    std::unique_ptr<CallableBase> cb;
  public:
    template<typename F>
    MyFunction(F&& f) : cb(std::make_unique<CallableImpl<F>>(std::forward<F>(f))) {}
    void operator()() { cb->call(); }
};
```

🧠 Benefits

- Flexibility in APIs
- Enables runtime polymorphism without inheritance
- Foundation of `std::function`, `std::any`, `std::variant`

📦 Custom Allocators & Memory Pooling

✅ What is a Custom Allocator?

User-defined memory management strategy for containers or objects.

🔧 Simple Example:

```
 template<typename T>
struct SimpleAllocator {
```

```
    using value_type = T;
    T* allocate(std::size_t n) { return static_cast<T*>(::operator new(n * sizeof(T))); }
    void deallocate(T* p, std::size_t) { ::operator delete(p); }
};
```

- Can be plugged into STL containers: `std::vector<int, SimpleAllocator<int>>`

✅ Memory Pooling

Pre-allocates a pool of memory to avoid repeated `malloc` / `free`.

🔧 Example Use Case:

- Game engines
- High-performance simulations
- Real-time systems

🧠 Benefits

- Predictable latency
- Reduced fragmentation
- High cache locality

🔁 ABI Stability & PImpl Idiom

✅ ABI (Application Binary Interface) Stability

Changing the internal layout of a class (e.g., adding a virtual function, changing data member order) **can break binary compatibility** with precompiled code.

❌ Why Changing Virtual Layout Breaks ABI

- Virtual function tables (`vtables`) have a fixed order in compiled binaries
- Adding/reordering virtual functions changes `vtable` layout, causing dynamic dispatch failures
- Old binaries expect the old `vtable` layout

✅ Solution: PImpl Idiom (Pointer to Implementation)

Separates interface from implementation to preserve binary compatibility.

```
 // header
class Widget {
```

```
  public:
    Widget();
    ~Widget();
    void draw();
  private:
    class Impl;
    Impl* pImpl;
};

// cpp file
class Widget::Impl {
  public:
    void drawImpl();
};
```

- Only the pointer size is exposed in the header
- Internals can change without recompiling dependents

🧠 Benefits

- ABI stability
- Encapsulation
- Faster compile times

🧬 Design Patterns in C++

🧊 Singleton (Thread-Safe)

```
class Singleton {
  public:
    static Singleton& getInstance() {
        static Singleton instance;
        return instance;
    }
  private:
    Singleton() {}
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
};
```

🏭 Factory Pattern

Creates objects without specifying exact class type.

```
class Shape { public: virtual void draw() = 0; };
class Circle : public Shape { void draw() override { ... } };

std::unique_ptr<Shape> createShape(std::string type) {
    if (type == "circle") return std::make_unique<Circle>();
    return nullptr;
}
```

🎯 Strategy Pattern

Encapsulate interchangeable algorithms.

```
class SortStrategy { public: virtual void sort(...) = 0; };
class QuickSort : public SortStrategy { ... };
class Context {
    SortStrategy* strategy;
    void setStrategy(SortStrategy* s) { strategy = s; }
};
```

🔔 Observer Pattern

Allows one-to-many event subscription.

```
class Observer { public: virtual void update() = 0; };
class Subject {
    std::vector<Observer*> observers;
    void notifyAll() { for (auto* o : observers) o->update(); }
};
```

📦 PImpl Idiom

Explained above — separates interface and implementation.

🧭 Visitor Pattern

Add new operations without modifying class structure.

```
class Element;
class Visitor { public: virtual void visit(Element*) = 0; };
class Element { public: virtual void accept(Visitor*) = 0; };
```

🧬 Prototype Pattern

Clone objects without knowing their concrete types.

```
 class Prototype {
  public:
    virtual Prototype* clone() const = 0;
};
```

🚫 Polymorphism Without Virtual (via CRTP)

✅ What is CRTP?

CRTP (Curiously Recurring Template Pattern) is a technique to achieve **static polymorphism** without virtual functions.

🔧 Example:

```
 template <typename Derived>
class Base {
  public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class Derived : public Base<Derived> {
  public:
    void implementation() {
        std::cout << "Derived implementation
";
    }
};
```

- No virtual calls, resolved at compile time
- Avoids vtable overhead

🧠 Pros:

- Zero runtime cost
- Works well in performance-critical systems

⚠️ Cons:

- Inflexible at runtime
- Requires compile-time knowledge of derived type

🚀 Improving Virtual Dispatch Performance

✅ Techniques

1. **Minimize Virtual Calls in Hot Paths**
    - Move virtual calls outside loops or performance-critical sections
2. **Use Final Classes/Functions**
    - Mark classes/methods `final` to allow devirtualization
3. **Devirtualization (Compiler Optimization)**
    - Compilers like GCC/Clang can inline virtual calls if type is known
4. **Switch to CRTP or Function Objects**
    - Prefer static dispatch where possible
5. **VTable Layout Optimization**
    - Keep virtual functions small and near the start of class layout
6. **Memory Layout**
    - Improve cache locality by placing frequently accessed data together

🧠 Tools

- `perf` , `VTune` , `objdump` to inspect call overhead

⚖️ Compile-Time vs Runtime Polymorphism

✅ Compile-Time Polymorphism

- Achieved via **function overloading**, **operator overloading**, **templates**, or **CRTP**.
- Resolved by the compiler at compile time.
- Fast: no runtime cost.

✅ Runtime Polymorphism

- Achieved using **inheritance** and **virtual functions**.
- Uses a vtable to determine the correct function to call at runtime.

- Slight runtime overhead due to indirection.

Feature	Compile-Time	Runtime
Examples	Templates, CRTP, Overloads	Virtual Functions, RTTI
Cost	Zero	Indirection, vtable
Flexibility	Limited to known types	Highly dynamic

⚙️ std::function vs Lambda vs Function Pointer

✅ Comparison Table

Feature	std::function	Lambda	Function Pointer
Type-erased	✅ Yes	❌ No (template-based)	❌ No
Capturing Support	✅ Yes	✅ Yes (if capturing)	❌ No
Can store any callable	✅ Yes	❌ Limited	❌ Limited
Performance	❗ Slower (heap, vtable)	✅ Fast (inline)	✅ Fast (inline)

🧠 Captures and Closures

- Lambdas can capture local variables by value/reference: `[x]() { return x; }`
- Lambdas with captures are **not convertible** to function pointers.
- `std::function` can wrap both capturing and non-capturing lambdas.

🧠 Implementing LRU Cache using OOP + STL

✅ Problem

Design a class with O(1) get and put for most recently used cache of fixed size.

🔧 Components Used

- `std::list` for maintaining order (front = most recent)
- `std::unordered_map` for fast key-node lookup

🛠️ Example:

```
class LRUCache {
   int capacity;
   std::list<std::pair<int, int>> lru; // {key, value}
   std::unordered_map<int, std::list<std::pair<int, int>>::iterator> map;

 public:
   LRUCache(int cap) : capacity(cap) {}

   int get(int key) {
       if (map.find(key) == map.end()) return -1;
       auto it = map[key];
       lru.splice(lru.begin(), lru, it); // move to front
       return it->second;
   }

   void put(int key, int value) {
       if (map.find(key) != map.end()) {
           auto it = map[key];
           it->second = value;
           lru.splice(lru.begin(), lru, it);
       } else {
           if (lru.size() == capacity) {
               auto last = lru.back();
               map.erase(last.first);
               lru.pop_back();
           }
           lru.emplace_front(key, value);
           map[key] = lru.begin();
       }
   }
};
```

🔎 RTTI (Run-Time Type Information)

✅ What is RTTI?

RTTI enables the identification of the **actual derived type** of an object at **runtime**, typically when accessed through a base class pointer or reference.

🔧 Enabled via:

- `typeid(obj)` → returns a `type_info` object describing the actual type.

- `dynamic_cast<T>(ptr)` → safely downcasts and checks if cast is valid at runtime.

🔍 Example:

```
 class Base { virtual void f() {} }; // Needs at least one virtual function
class Derived : public Base { };

Base* b = new Derived();
if (Derived* d = dynamic_cast<Derived*>(b)) {
    std::cout << "Downcast successful";
}
std::cout << typeid(*b).name(); // returns "Derived"
```

🧠 When to Use

- In polymorphic hierarchies where actual type needs to be known
- For dynamic plugin loading, serialization frameworks, and debug introspection

⚠️ Notes

- RTTI only works if the base class is **polymorphic** (i.e., has at least one virtual function)
- May incur **runtime overhead**, and not always available (can be disabled with `-fno-rtti`)

💾 File Handling & Streams

📂 ifstream, ofstream, fstream

✅ File Stream Types

- `std::ifstream`: Input file stream (read-only)
- `std::ofstream`: Output file stream (write-only)
- `std::fstream`: File stream for both input and output

🔧 Example:

```
 #include <fstream>
#include <string>

void readFile(const std::string& filename) {
```

```
    std::ifstream infile(filename);
    std::string line;
    while (std::getline(infile, line)) {
        std::cout << line << '\n';
    }
}

void writeFile(const std::string& filename) {
    std::ofstream outfile(filename);
    outfile << "Hello, File!\n";
}
```

🧠 File Modes

File modes determine how a file is opened or accessed:

- `ios::in` , `ios::out` , `ios::app` , `ios::binary` , `ios::trunc`
- `std::ios::in` — Open for reading (default for `ifstream`)
- `std::ios::out` — Open for writing (default for `ofstream`)
- `std::ios::app` — Append to the end of file
- `std::ios::ate` — Open and move to the end (can write anywhere)
- `std::ios::trunc` — Truncate file if it exists (default for `ofstream`)
- `std::ios::binary` — Open file in binary mode (no line ending translation)

You can combine modes using bitwise OR `|` :

```
std::ofstream fout("out.txt", std::ios::out | std::ios::app); // write + append
```

🔄 Input/Output Redirection

✅ What is Redirection?

Redirecting standard input/output from/to files instead of the console.

🔧 Terminal Usage:

```
$ ./program < input.txt > output.txt
```

- `< input.txt` reads stdin from file

- `> output.txt` writes stdout to file

🧠 Notes:

- Often used in competitive programming and batch processing
- Works with C++ code using `std::cin`, `std::cout`

🔧 C++ Program Example:

```
int main() {
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);

    int a, b;
    std::cin >> a >> b;
    std::cout << a + b << "\n";
}
```

📤 Flushing, Syncing, and Clearing Input Buffer

✅ Flushing

Forces buffered output to be written immediately.

```
std::cout << "Processing..." << std::flush;
```

- `std::endl` also flushes after inserting a newline.
- Useful for real-time output (e.g., progress bars).

✅ Syncing with `sync_with_stdio`

Disables synchronization between C and C++ standard streams to improve performance.

```
std::ios::sync_with_stdio(false);
std::cin.tie(nullptr);
```

- Speeds up `cin/cout`, especially in competitive programming.

✅ Clearing Input Buffer

To clear leftover input or errors from `cin`:

```
std::cin.clear();                // clear error flags
std::cin.ignore(INT_MAX, '\n');  // discard invalid input or leftovers
```

- Use when switching between numeric input and `getline`.

🆚 cerr vs cout

✅ Difference

Feature	`std::cout`	`std::cerr`
Purpose	Standard output (regular data)	Standard error (debug/errors)
Buffered	Yes (may delay output)	No (unbuffered — prints instantly)
Usage	Program output	Error messages, debug logging

🔧 Example:

```
 std::cout << "Result: " << value << std::endl;
std::cerr << "Invalid input!" << std::endl;
```

- `cerr` is often used for debugging without affecting redirected stdout

---

🔟 Build Systems & Toolchain

🧱 Compilers: `g++`, `gcc`, `clang`

✅ gcc / g++ (GNU Compiler Collection)

- `gcc` is the C compiler
- `g++` is the C++ compiler
- Popular, mature, open-source

```
 g++ main.cpp -o app               # basic compilation
g++ -Wall -Wextra -O2 -std=c++20 main.cpp -o app
```

🔧 Useful Flags

- `-Wall -Wextra` → enable warnings
- `-O0/O1/O2/O3/Ofast` → optimization levels
- `-g` → include debug symbols for gdb

- `-std=c++17` → specify C++ standard version
- `-I`, `-L`, `-l` → include/include path/lib

✅ clang

- Lightweight and modular compiler from LLVM
- Provides better error messages than g++
- Fast compilation, widely used for tooling

```
clang++ -std=c++20 -Wall file.cpp -o output
```

🧠 Comparison

Feature	g++	clang++
Speed	✅ Good	✅ Excellent
Error Output	Moderate	✅ Very Clear
Tool Support	✅ Widespread	✅ Strong (LLVM)

⚙️ Compiler Flags

🔧 Common Flags

- `-Wall` – Enable basic warnings
- `-Wextra` – Enable additional warnings
- `-O0/O1/O2/O3/Ofast` – Set optimization levels (O0 = none, O3 = aggressive)
- `-g` – Include debug symbols for use with `gdb`
- `-std=c++XX` – Specify C++ standard (e.g., `-std=c++20`)
- `-fsanitize=address` – Enable AddressSanitizer for detecting memory errors

🧠 Usage Example:

```
g++ -std=c++20 -Wall -Wextra -O2 -g -fsanitize=address main.cpp -o app
```

🧩 Debug Symbols

✅ What Are Debug Symbols?

Debug symbols provide metadata that maps compiled machine code back to the original source code, allowing a debugger like `gdb` to:

- Show variable names
- Display line numbers
- Trace function calls

🔧 How to Enable

Use the `-g` flag during compilation:

```
g++ -g main.cpp -o app
```

🧠 Notes:

- Debug symbols increase binary size but do not affect runtime performance
- Required for meaningful debugging, stack traces, and introspection

---

🐞 gdb Basics (GNU Debugger)

✅ Launch & Basic Commands

```
gdb ./app
```

- `break main` – Set breakpoint at `main()`
- `run` – Run the program
- `next` – Step over a line
- `step` – Step into a function
- `print x` – Inspect variable `x`
- `bt` – Show backtrace
- `quit` – Exit debugger

🧠 Notes:

- Requires compiling with `-g`
- Use `layout src` inside `gdb` for split-screen source view

🛠️ Build Automation Tools

✅ Makefile

A **Makefile** is a script used by the `make` utility to automate building programs by defining rules, targets, and dependencies.

📦 Basic Structure

```
target: dependencies
    command-to-build
```

- Each **target** is a file to be built
- **Dependencies** are files that the target depends on
- Commands must be indented using a **tab**, not spaces

🔧 Example

🧰 Sample Makefile Template (Modular C++ Project)

```
 # Compiler and flags
CXX = g++
CXXFLAGS = -std=c++20 -Wall -Wextra -O2 -g

# Target executable
TARGET = app

# Source and object files
SRCS = main.cpp utils.cpp module.cpp
OBJS = $(SRCS:.cpp=.o)

# Default target
all: $(TARGET)

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

- `all` is the default target
- Pattern rule `%.o: %.cpp` compiles each `.cpp` into `.o`
- `$@`, `$<`, and `$^` are automatic variables (target, first dependency, all dependencies)

```
app: main.cpp utils.cpp
    g++ -Wall -g main.cpp utils.cpp -o app
```

```
clean:
    rm -f app
```

▶️ Running Makefile

```
 make          # builds default (first) target: app
make clean    # runs the clean rule
```

🧠 Notes

- Rebuilds only what changed based on file timestamps
- Ideal for large projects with multiple source files Defines build rules in a declarative format.

```
app: main.cpp utils.cpp
   g++ main.cpp utils.cpp -o app
```

Run with:

```
make
```

✅ CMake

Cross-platform, modern build system.

```
 cmake_minimum_required(VERSION 3.10)
project(MyApp)
add_executable(MyApp main.cpp)
```

Build with:

```
 cmake .
make
```

🧪 Debug vs 🚀 Release Builds

Build Type	Purpose	Key Flags	Notes
Debug	Development & debugging	`-g -O0`	No optimization, full debug info

Build Type	Purpose	Key Flags	Notes
Release	Performance & deployment	`-O2` or `-O3`, `-DNDEBUG`	No debug symbols, aggressive opt

---

🗓️ C++ Versions and Feature Additions (C++98 → C++23)

🕰️ C++98 / C++03

- First standard versions
- Features: `STL`, templates, exceptions, namespaces, RAII
- Lacked modern features like `auto`, lambdas, smart pointers

🚀 C++11 (Biggest leap)

- `auto`, `nullptr`, `enum class`, `static_assert`
- **Move semantics**: `rvalue references`, `std::move`, `std::forward`
- **Lambda expressions**
- `range-based for` loops
- `std::shared_ptr`, `std::unique_ptr`, `std::weak_ptr`
- `thread`, `mutex`, `future`, `async`
- Variadic templates, `constexpr`, `decltype`
- `override`, `final` keywords
- `unordered_map`, `unordered_set`

🖌️ C++14 (Refinement)

- Generic lambdas: `auto` in lambda args
- `decltype(auto)`
- `std::make_unique`
- Return type deduction
- Binary literals (`0b1010`)
- `std::integer_sequence`, `std::exchange`

💡 C++17 (Polishing & Power Tools)

- `if constexpr`, `inline variables`
- Structured bindings: `auto [a, b] = pair;`
- `std::variant`, `std::optional`, `std::any`
- `std::filesystem`
- `std::string_view`, `std::byte`
- `std::apply`, `std::invoke`, fold expressions
- Parallel algorithms: `std::execution`
- `[[nodiscard]]`, `[[maybe_unused]]`

🧪 C++20 (Modern & Expressive)

- **Concepts**: type constraints in templates
- Ranges library: `views::filter`, `views::transform`
- `consteval`, `constinit`
- `std::span`, `std::bit_cast`, `std::is_constant_evaluated`
- `coroutines`: `co_await`, `co_yield`
- Modules (experimental)
- `three-way comparison (<=>)` "spaceship operator"
- Improved lambdas: lambdas in unevaluated contexts, template params
- Calendar/date utilities: `std::chrono`

🔥 C++23 (Refinements + New Libraries)

- `explicit` object parameters: `void func(this MyType self)`
- `std::expected`: error handling alternative to exceptions
- `std::print`, `std::println` (modern formatted I/O)
- More constexpr features
- `std::flat_map`, `flat_set`
- `std::generator` (coroutine-based)
- Multidimensional subscript operator (`operator[](...)`)
- Monadic operations on `optional` / `expected`

❌ Removed / Deprecated Across Versions

- `auto_ptr` (deprecated in C++11, removed in C++17)
- `register`, `gets()`
- `throw()` specifications (replaced with `noexcept`)
- `std::bind1st`, `std::bind2nd` (removed in C++17)

✳️ Modern C++ Features

✅ Assertions in C++

🔎 What Are Assertions?

Assertions are used to check conditions during runtime. If the condition fails, the program terminates with an error message.

🔧 Syntax

```
 #include <cassert>

int main() {
    int x = 5;
    assert(x > 0); // Passes
    assert(x < 0); // Fails at runtime
}
```

- Only active in debug builds (when `NDEBUG` is **not** defined)
- In release builds, assertions are disabled

🧠 When to Use

- Check invariants
- Validate assumptions in algorithms
- Detect logical errors early in development

⚠️ Note

- Do not use assertions for user input validation — they're intended for internal sanity checks only

---

🔁 Range-Based For Loops

✅ Syntax

```
 std::vector<int> nums = {1, 2, 3, 4};
for (int x : nums) {
```

```
    std::cout << x << " ";
}
```

🔍 With References and `auto`

```
for (auto& x : nums) {
    x *= 2; // Modifies the original
}
```

🧠 Advantages

- Simplifies iteration over containers
- Safer and less error-prone than index-based loops

🔧 Works With

- All containers with `begin()` and `end()`
- Arrays, initializer lists, custom iterable types

🔄 Async in Threads

✅ What is `std::async`?

`std::async` launches a task asynchronously (in a separate thread or lazily).

```
#include <future>

int compute() {
    return 42;
}

int main() {
    std::future<int> result = std::async(std::launch::async, compute);
    std::cout << result.get(); // Waits and retrieves the result
}
```

🚦 Launch Policies

- `std::launch::async` – forces a new thread
- `std::launch::deferred` – runs only when `.get()` or `.wait()` is called
- Default behavior is implementation-defined

🧠 Notes

- Returns a `std::future<T>` to retrieve the result
- Automatically manages thread lifecycle

🔒 `final` Keyword

✅ Purpose

Prevents a class or method from being overridden.

🔧 Usage

```
 class Base {
  virtual void show() final; // Cannot override
};

class Derived final : public Base { // Cannot be inherited
};
```

🧠 When to Use

- Enforce interface stability
- Optimize for performance (some compilers devirtualize final methods)
- Prevent unsafe inheritance or overrides

---

❺ Binary Literals

✅ What Are They?

C++14 introduced binary literals using the `0b` or `0B` prefix.

```
int x = 0b1010;  // Decimal 10
```

- Makes bit-level operations more readable
- Useful in hardware and embedded programming

---

❻ `std::integer_sequence` & `std::exchange`

✅ `std::integer_sequence`

Represents a compile-time sequence of integers, useful in template metaprogramming.

```
template<int... Ints>
void print(std::integer_sequence<int, Ints...>) {
    ((std::cout << Ints << " "), ...);
}

print(std::integer_sequence<int, 1, 2, 3>{});
```

- Commonly used with `std::index_sequence`

✅ **`std::exchange`**

Replaces the value of an object and returns the old value.

```
int a = 10;
int old = std::exchange(a, 20); // a = 20, old = 10
```

- Often used in move operations and reset functions

---

❼ C++17 Key Features

✅ **`if constexpr`**

Compile-time conditional branching inside templates.

```
template<typename T>
void print(T val) {
    if constexpr (std::is_integral<T>::value)
        std::cout << "int: " << val;
    else
        std::cout << "other: " << val;
}
```

✅ **`std::filesystem`**

C++17 standard library for file and path manipulation.

```
#include <filesystem>
namespace fs = std::filesystem;

fs::path p = "example.txt";
std::cout << fs::absolute(p);
```

✅ **`std::string_view`**

A lightweight, non-owning view over a string.

```
std::string s = "hello";
std::string_view sv = s;
```

- Avoids unnecessary copies
- Safer than raw C-strings

✅ `std::byte`

A type-safe alternative to `char` for raw byte manipulation.

```
std::byte b = std::byte{0x1F};
```

- Encourages explicit intent in low-level code

8️⃣ `std::apply`, `std::invoke`, Fold Expressions

✅ `std::apply`

Unpacks a tuple and calls a callable with its elements.

```
#include <tuple>
#include <functional>

auto func = [](int a, int b) { return a + b; };
auto tup = std::make_tuple(2, 3);
std::cout << std::apply(func, tup); // Outputs 5
```

✅ `std::invoke`

Universal function caller that works with member pointers, functions, lambdas, etc.

```
std::invoke([](int a){ return a * 2; }, 10); // returns 20
```

✅ Fold Expressions

C++17 feature to reduce parameter packs elegantly.

```
template<typename... Args>
auto sum(Args... args) { return (args + ...); } // left fold
```

9️⃣ `[[nodiscard]]`, `[[maybe_unused]]`

✅ `[[nodiscard]]`

Warns if a function's return value is ignored.

```
[[nodiscard]] int compute() { return 42; }
compute(); // warning if result ignored
```

✅ `[[maybe_unused]]`

Suppresses unused variable warnings.

```
void foo() {
    [[maybe_unused]] int x = 5;
}
```

9️⃣ Coroutines (`co_await`, `co_yield`, `co_return`)

✅ What Are Coroutines?

Functions that can be **paused and resumed**, ideal for async operations, generators.

🔧 Key Keywords:

- `co_await`: waits on a result
- `co_yield`: yields a value like a generator
- `co_return`: returns a value from coroutine

🔧 Example:

```
#include <coroutine>
#include <iostream>

struct Task {
  struct promise_type {
    Task get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend never final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {}
  };
};
```

```
Task example() {
    std::cout << "Start\n";
    co_await std::suspend_always{};
    std::cout << "Resumed\n";
}
```

- Requires C++20
- Powers async I/O, cooperative multitasking

⓫ Modules (Experimental)

✅ What Are Modules?

Modules are a modern alternative to the traditional header/include system.

- Reduce compile times
- Avoid macro collisions and multiple inclusions
- Improve encapsulation and build hygiene

🔧 Syntax (C++20)

```
 // math.ixx (interface)
export module math;
export int add(int a, int b) { return a + b; }

// main.cpp
import math;
int main() { return add(2, 3); }
```

- Still compiler-specific and tooling-dependent

⓬ `std::bit_cast`, `std::is_constant_evaluated`

✅ `std::bit_cast<T>(from)` (C++20)

Reinterprets bit representation safely without UB.

```
 float f = 3.14;
int i = std::bit_cast<int>(f); // safely view float bits as int
```

- Replacement for `reinterpret_cast`
- Requires both types to be trivially copyable and same size

✅ `std::is_constant_evaluated()`

Checks if code is running in a `constexpr` context.

```
 constexpr int square(int x) {
    if (std::is_constant_evaluated()) {
        return x * x;
    } else {
        std::cout << "runtime\n";
        return x * x;
    }
}
```

- Helps write hybrid compile-time/runtime logic

⓭ Three-Way Comparison (`<=>`) – "Spaceship Operator"

✅ Overview

The `<=>` operator simplifies and unifies comparisons.

- Introduced in C++20
- Enables default generation of all comparison operators

🔧 Example:

```
 #include <compare>
#include <iostream>

struct Point {
    int x, y;
    auto operator<=>(const Point&) const = default;
};

int main() {
    Point p1{1, 2};
    Point p2{1, 3};

    if (p1 < p2) std::cout << "p1 is less than p2
";
    if (p1 != p2) std::cout << "p1 and p2 are not equal
";

    auto cmp = (p1 <=> p2);
```

```
    if (cmp < 0) std::cout << "Result: p1 < p2
";
    else if (cmp == 0) std::cout << "Result: p1 == p2
";
    else std::cout << "Result: p1 > p2
";

    return 0;
}
```

```
 - Automatically provides `==`, `<`, `>`, `<=`, `>=`, `!=`

### ✅ Categories:
- `std::strong_ordering` (like int)
- `std::weak_ordering`
- `std::partial_ordering` (for NaN floats etc.)

---
```

⓮ `std::chrono`

✅ Purpose

A modern, type-safe library for handling time durations, timestamps, and clocks.

🔧 Usage Examples

```
 #include <chrono>
#include <iostream>

int main() {
    auto start = std::chrono::high_resolution_clock::now();

    // ... some work ...

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Elapsed: " << duration.count() << "ms\n";
}
```

🧠 Key Concepts

- `std::chrono::duration` – Time intervals
- `std::chrono::time_point` – Specific point in time
- `std::chrono::steady_clock` , `system_clock` , `high_resolution_clock`

⓯ Explicit Object Parameters (`this` in function args)

✅ Syntax

C++23 allows explicitly naming the `this` parameter:

```
struct MyType {
    void func(this MyType self) {
        std::cout << self.value;
    }
    int value;
};
```

- Allows **value**, **reference**, or **const-reference** qualification

🔧 Benefits

- Enables overloading based on object qualifier (like Rust-style method receivers)
- Improves clarity in non-member algorithms or extensions

⓰ `std::expected` – Error Handling Alternative

✅ Overview

A new type introduced in C++23 to replace exceptions in some cases with explicit, lightweight error handling.

🔧 Usage Example

```
#include <expected>

std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) return std::unexpected("Divide by zero");
    return a / b;
}

int main() {
    auto result = divide(10, 0);
    if (result) std::cout << "Result: " << *result;
    else std::cout << "Error: " << result.error();
}
```

🧠 Benefits

- Safer alternative to exceptions
- Forces caller to check for success/error explicitly
- Inspired by functional programming (like `Result` in Rust)

⓱ `std::print` , `std::println` (Modern Formatted I/O)

✅ Overview

Introduced in **C++23**, these are safer, faster, and simpler I/O functions.

🔧 Example

```
 #include <print>
int x = 42;
std::print("Value: {}\n", x);        // prints without flushing
std::println("Total: {}", x);        // prints with newline and flush
```

🧠 Benefits

- Replaces `std::cout` + manual formatting
- Inspired by Python's `print()` and Rust's `println!`
- Type-safe formatting using `{}` placeholders

⓲ `std::flat_map` , `std::flat_set`

✅ Overview

Sorted containers backed by **contiguous storage** (like `vector`) for faster iteration & lower memory use.

- Introduced in **C++23**
- Maintains sorted order but is more cache-friendly than `std::map` / `std::set`

🔧 Example

```
 #include <flat_map>
std::flat map<int, std::string> fmap = {{1, "one"}, {2, "two"}};
std::cout << fmap[2];
```

🧠 Benefits

- Better performance for read-heavy workloads
- Less overhead than tree-based maps

---

⑲ `std::generator` (Coroutine-Based)

✅ Overview

`std::generator` provides **lazy sequences** using `co_yield`, part of **C++23**.

🔧 Example

```
 #include <generator>

std::generator<int> counter(int start, int end) {
    for (int i = start; i <= end; ++i)
        co_yield i;
}

int main() {
    for (int i : counter(1, 5))
        std::cout << i << " ";
    // Output: 1 2 3 4 5
}
```

🧠 Benefits

- Memory-efficient iteration
- Replaces complex iterator logic
- Ideal for pipelined & async-like processing

---

⑳ Multidimensional Subscript Operator (`operator[](...)`)

✅ Overview

C++23 allows custom types to define **multidimensional subscript operators**:

```
struct Matrix {
   int data[10][10];
   int& operator[](std::size_t row, std::size_t col) {
       return data[row][col];
```

```
    }
};
```

🔧 Benefits

- Cleaner syntax for matrix/vector access
- No need for nested proxy classes or chaining
- Supports arbitrary number of indices

⓴❶ Monadic Operations on `std::optional` / `std::expected`

✅ Overview

Monadic operations simplify conditional logic chaining for optional/expected values.

🔧 Example: `and_then`, `transform`, `or_else`

```
 #include <optional>
std::optional<int> maybe_square(int x) {
    if (x < 0) return std::nullopt;
    return x * x;
}

std::optional<int> result = std::optional{5}
    .and_then(maybe_square)         // returns optional<int>
    .transform([](int val) { return val + 1; });
```

🧠 Benefits

- Avoids deeply nested `if` statements
- Functional programming style
- Safer and easier to compose conditional operations

⓴❷ `throw()` Specifications (Replaced by `noexcept`)

❌ Old Style

```
 void func() throw();       // C++98-style (deprecated)
void risky() throw(int);   // throws only int (not enforced)
```

✅ Modern Way (C++11+)

```
void func() noexcept;      // means function won't throw
```

🧠 Why `noexcept`?

- Enforced by compiler → can optimize more aggressively
- Affects exception safety, move operations, and termination handling
- `noexcept(true)` vs `noexcept(false)` for conditional expressions
