Contents

1 One Definition Rule (ODR)

- Function/variable defined fully in a header → still has **external linkage**.
- Each translation unit compiled independently → every object file gets its own copy of the symbol.
- Linker sees the symbol multiple times → **"multiple definition"** error.
- `inline` fixes it: entity may be defined in many translation units (promise: all definitions identical). Linker keeps one copy.
- ⚠️ Nothing checks the promise. Same signature + different bodies = **UB**.

2 Include Guards

- Purpose: header processed at most once per translation unit (preprocessing) → prevents redefinition from repeated inclusion.
- Common choice: `#pragma once`. Nonstandard but all major compilers support it.
- Advantages:
  - one line instead of three (less boilerplate)
  - keys on the file itself → no risk of two headers sharing a guard name

3 Brace Initialization `{}`

Prefer `{}` as the default initializer.

- **No narrowing**: `int x{2.5};` = compile error; `int x = 2.5;` silently truncates to `2`.
- **Defined state**: `int x{};` value initializes to `0` (avoids indeterminate value / UB of `int x;`).
- **No most vexing parse**: `Car c{};` = object; `Car c();` = function declaration.
- **Uniform**: same syntax for scalars, aggregates, members, containers.

Two cases where `{}` is wrong:

- `std::initializer_list` **wins ties**: `std::vector<int>{10, 20}` = two elements `{10,20}`, not ten of value `20`. Use `(10, 20)` for the constructor call.
- `auto` **+** `=` **+ braces**: `auto x = {0.0};` deduces `std::initializer_list<double>`. For `auto` write `auto x = 0.0;` or `auto x{0.0};`.

4 Most Vexing Parse

- Rule: if a line **can** be parsed as a function declaration, the compiler **must** treat it as one.
- Problem: `()` for default init looks like a function declaration.
- Fix: use braces `{}` for object instantiation.

```
1 Widget w1(); // declares a function
2 Widget w2{}; // creates a Widget
```

5 Pointer vs Pointee

- **Pointer**: variable holding a memory address (e.g. `0x7ffe...`).
- **Pointee**: the actual object at that address.
- **Dereference** `*ptr`: access/modify the pointee.

```
int age = 25;    // pointee
int* ptr = &age; // pointer
```

6 Classes

6.1 Special Member Functions

- Destructor, default constructor, copy/move constructor, copy/move assignment
- Destructor, move constructor and move assignment must **not** throw. Hence, Move Ctor/Assignment should be marked `noexcept`. Dtor is implicitly `noexcept`.
- Can be defaulted with `= default`

6.2 `explicit`

- In general, mark all Ctor (except copy/move ctor) as `explicit` → prevents **implicit conversions**.

```
class Wallet final {
  public:
    explicit Wallet(int amount) : cash{amount} {}
  private:
    int cash{};
};

void pay(Wallet w);
pay(50);         // error: no implicit int -> Wallet
pay(Wallet{50}); // ok
```

6.3 const Member Variables (avoid)

- A `const` member → compiler **deletes** the default copy assignment `operator=`.
- `const` member cannot change after construction → object not assignable later.

```
struct Player {
    const int id; // blocks copy assignment
    std::string name;
};

// p1 = p2; // error
```

6.4 Constructor Delegation

```cpp
class Employee {
  public:
    Employee(std::string name, int id) : name_(name), id_(id) {}
    Employee(std::string name) : Employee(name, -1) {} // delegates
  private:
    std::string name_;
    int id_;
};
```

6.5 Operator Overloading

Output operator as a **free function** (member version awkward to call):

```cpp
inline auto operator<<(std::ostream& os, const Date& date) -> std::ostream& {
    return date.print(os);
}
```

or mark it as `friend`:

```cpp
class DayOfWeek final {
  public:
    friend auto operator<<(std::ostream& os, const Date& date) -> std::ostream&;
  private:
    std::string name_{};
};
```

Increment operator:

- Prefix: `auto operator++(DayOfWeek&) -> DayOfWeek`
- Postfix: `auto operator++(DayOfWeek&, int) -> DayOfWeek`

6.6 `friend` Keyword

- Grants free functions / other classes access to `private` + `protected` members.
- **Friend function**: free function allowed inside the private barrier (e.g. binary `<<`).
- **Friend class**: another class gets full access.

Rules:

1. **Granted, not taken**: the owner declares the friendship.
2. **Not symmetric**: A friends B does not imply B friends A.
3. **Not transitive**: A friends B, B friends C does not imply A friends C.

Prefer a **free function** over `friend`: friendship is not inherited; a free function calling a `virtual` member supports polymorphism.

6.7 Abstract Classes

- Class is abstract if it has at least one **pure virtual function** (`= 0`).
- Instantiation blocked by the compiler.
- Do **not** delete the constructor: derived classes call the base constructor to init their base layers.

```
struct Shape {
    virtual void draw() = 0;
    virtual ~Shape() = default; // needed for safe polymorphic cleanup
};
```

6.8 Inheritance: Construction and Destruction Order

- Construction order: **Base** → **Sub** → **SubSub**.
- Destruction order: reverse (**SubSub** → **Sub** → **Base**); first created = last destroyed.
- Override is implicit once a base member is `virtual`.
- A **nonvirtual** call uses the **static** type.
- A reference binding (`Animal& a = hummingbird;`) has **no** destructor call (not an object).

```
struct Animal {
    void makeSound(); // nonvirtual: uses static type
    virtual void move(); // virtual: dynamic dispatch
    Animal()  { /* "animal born"  */ }
    ~Animal() { /* "animal died"  */ }
};

struct Bird : Animal { /* override move(), ctor/dtor */ };
struct Hummingbird : Bird { /* override move(), ctor/dtor */ };

Hummingbird h; // ctor: Animal -> Bird -> Hummingbird
Animal& a = h; // binds reference only
a.makeSound(); // Animal version (nonvirtual)
a.move();      // Hummingbird version (virtual)
```

6.9 Object Slicing

- Happens when a derived object is assigned to a base **by value**.
- Result: a new independent **base** object.
- Loss: derived data stripped, virtual overrides lost.
- Destructs strictly as the base type.

```
Derived d;
Base b = d; // slicing: b is a pure Base
```

6.10 Rule of 3 / Rule of 5 / Rule of Zero

- **Problem:** Default copies are **shallow** (copy raw pointers) → leaks, double-free crashes.
- **Rule of 3:** If you need a custom **destructor**, **copy constructor**, or **copy assignment** → you need all three (to implement a deep copy).
- **Rule of 5:** Add **move constructor** + **move assignment** → else the compiler falls back to expensive copies.
- **Rule of Zero:** Design your classes so they **do not** manage raw resources directly. Rely entirely on existing RAII types (`std::string`, `std::unique_ptr`, `std::vector`) to handle ownership. If you write no custom special member functions, the compiler generates all of them perfectly for you.
- ⚠️ Writing a custom destructor or copy operation **blocks** compiler-generated move operations.
- **The Polymorphic Exception:** When a base class requires a `virtual ~destructor()`, it automatically blocks implicit moves. You must explicitly restore them using `= default`.
- ⚠️ **Slicing Hazard:** To prevent dangerous object slicing in inheritance hierarchies, polymorphic base classes should usually `= delete` their copy/move operations instead of defaulting them.
- Move operations must **not** throw.
- Assignment operators must be **member functions**.

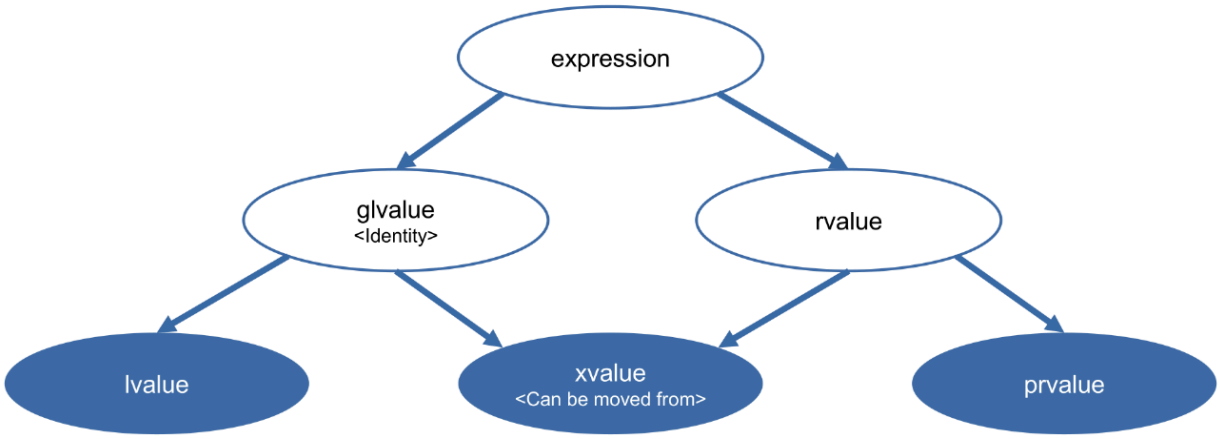
What you get

What you write ↓ / Where you want to be →	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted (!)	defaulted (!)	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted (!)	not declared	not declared
copy assignment	defaulted	defaulted	defaulted (!)	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Avoid if possible

7 Move Semantics and Value Categories

- **lvalue reference** `T&`: binds to an lvalue; original must outlive the reference. String literals are always lvalues.
- **rvalue reference** `T&&`: binds to an rvalue; can extend the lifetime of a temporary.



has identity?	can be moved from?	Value Category
Yes	No	lvalue
Yes	Yes	xvalue (expiring value)
No	No (Since C++17)	prvalue (pure rvalue)
No	Yes (Since C++17)	- (doesn't exist anymore)

- **lvalue**: has identity; cannot be moved from; address can be taken; can init an lvalue reference.
- **prvalue**: no identity; cannot be moved from; no address; not the left side of builtin assignment.
  - **Temporary materialization**: when a glvalue is required, prvalue → xvalue (compiler allocates temp memory). Type must be complete with a destructor.
  - Triggers: binding a reference to a prvalue; accessing a member of a prvalue; accessing an element of a prvalue array; prvalue array → pointer; init `std::initializer_list<T>` from a brace list.
- **xvalue**: has identity; can be moved from; not the left side of builtin assignment; result of prvalue temporary materialization; `std::move(lvalue)` produces an xvalue.

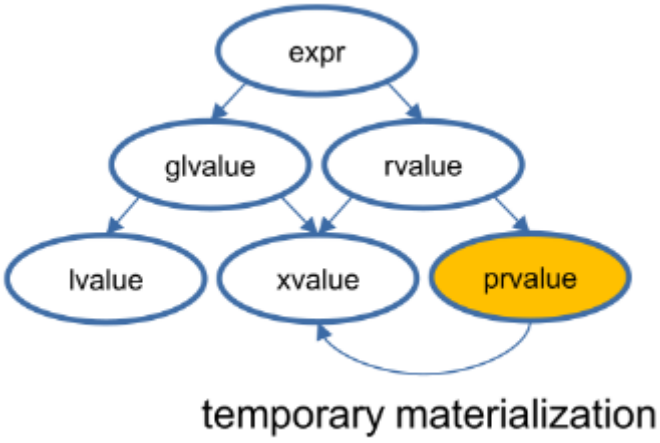


Figure 4: Temporary materialization

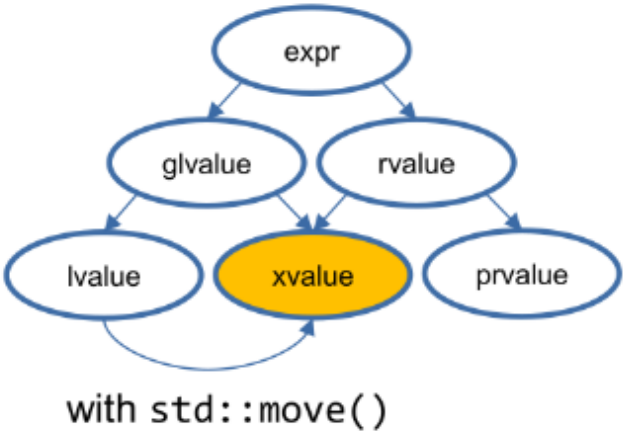


Figure 5: `std::move()` is a static-cast mechanism that converts an lvalue into an xvalue, without creating a prvalue or any new object.

8 Copy Elision

Compiler optimization: omits copy/move by constructing the object directly in its final memory location. Three contexts:

1. **NRVO** (Named Return Value Optimization): returning a local by value → constructed directly in the caller's slot (ignores `const` for type matching).
2. **Throw expressions**: throwing a local from the innermost `try` → exception object built directly in the exception storage.
3. **Catch clauses**: caught type matches the thrown type exactly → accessed in place (same profile as catch by reference).

9 Return Value: Elision, Move, Copy

Priority for `return expr;` by value: **elide** → **move** → **copy**.

- **Temporary (prvalue)** (`return T();`): **guaranteed elision** (C++17). No move, no copy.
- **Named local / by value parameter** (`return t;`): try **NRVO**. Else treat as **rvalue** → **move**. Else **copy**.
- **Other lvalue** (global, member, `*ptr`, reference param): **copy**. No implicit move. `return std::move(x);` needed (rarely wanted).
- **Scalars** (`int`, `double`, pointers): trivial copy.

10 Overload Resolution

Two key sort (templates and nontemplates compete together):

1. **Primary**: best implicit conversion sequence (exact > promotion > conversion). Template vs nontemplate irrelevant here.
2. **Tiebreak** (only when conversions are equal): nontemplate beats template; among templates, more specialized beats more generic.

Consequence: an **exact matching template** beats a **nontemplate needing a conversion** (primary key dominates). Template is never its own tier; it is the runner up **within** its conversion tier.

	`f(S)`	`f(S &)`	`f(S const &)`	`f(S &&)`
`S s{};` `f(s);`	✓	✓ (preferred over const &)	✓	✗
`S const s{};` `f(s);`	✓	✗	✓	✗
`f(S{});`	✓	✗	✓	✓ (preferred over const &)
`S s{};` `f(std::move(s));`	✓	✗	✓	✓ (preferred over const &)

Figure 6: Overload resolution rules and binding preferences for different value categories

11 Template Type Deduction

11.1 Value `T param`

- Strips **references** (`&`, `&&`) and **top level** (right-most) `const`.
- Reason: function gets an independent copy.
- `int x` ⟶ `T = int`; `int const cx` ⟶ `T = int`; `int const& crx` ⟶ `T = int`; `const int* const cpc` ⟶ `T = const int*`

11.2 Reference `T&` or `const T&`

- Strips the reference, **keeps** **`const`**.
- `int x` ⟶ `T = int`; `int const cx` ⟶ `T = int const`; `int const& crx` ⟶ `T = int const`.

11.3 Forwarding Reference `T&&`

- Uses **reference collapsing**.
- lvalue arg ⟶ `T = Arg&` ⟶ param is an lvalue reference.
- rvalue arg ⟶ `T = Arg` ⟶ param is an rvalue reference.
- `int x` ⟶ `T = int&`; `int const cx` ⟶ `T = int const&`; `int const& crx` ⟶ `T = int const&`; `foo(27)` ⟶ `T = int`.

12 decltype and decltype(auto)

- `decltype(expr)`: type at compile time, no evaluation.
- **Name rule** `decltype(variable)`: exact declared type, keeps `const` + references. `const int& cx` ⟶ `decltype(cx)` = `const int&`.
- **Expression rule** `decltype(expression)`: adds references by value category. lvalue (e.g. `(x)`, `c[i]`) ⟶ `T&`; rvalue (e.g. `x + 5`) ⟶ `T`.

Return type:

- **`auto`**: template deduction ⟶ strips references ⟶ value copy.
- **`decltype(auto)`**: applies decltype rules to the return expression ⟶ preserves references.

```
1 auto access(Container& c, Index i) { ... }           // returns int (copy)
2 decltype(auto) access(Container& c, Index i) { ... } // returns int& (reference)
```

⚠️ Parentheses trap with `decltype(auto)`: `return x;` ⟶ declared type (value). `return (x);` ⟶ lvalue expression ⟶ reference to a local (dangling).

13 Variadic Templates `...`

- `...` = parameter pack: variable number of args, arbitrary types. Two phases: pack, then expand.
- `sizeof...(args)` = element count.

```
template <typename... Args>
void log(Args... args) { std::size_t n = sizeof...(args); }
```

13.1 Pack Expansion (recursive, old style)

Needs a base case to terminate.

```
void print_all() { // base case
  std::cout << '\n';
}

template <typename T, typename... Args>
void print_all(const T& first, const Args&... args) {
    std::cout << first << " ";
    print_all(args...); // expand pack
}
```

13.2 Fold Expressions (C++17)

Unpack a pack with a binary operator in one expression. No recursion, no base case.

```
template <typename... Args>
void print_all(const Args&... args) {
    (std::cout << ... << args) << '\n'; // Unpacks to: (std::cout << a0) << a1) << a2...
}
```

- A function parameter pack may be **empty** (matches zero or more args); no default needed.
- A leading nonpack parameter (`First first`) is still **mandatory** → needs at least one arg.

13.3 Perfect Forwarding `std::forward`

Forward args preserving lvalue/rvalue. Both the type pack and the value pack use `...`

```
template <typename... Args>
void factory(Args&&... args) {
    auto p = std::make_unique<Widget>(std::forward<Args>(args)...);
}
```

14 Template Specialization

Override a generic template for specific types or families.

14.1 Full Specialization

- Dedicated impl for one exact type. Empty parameter list `template<>`, type after the name.
- Rule: a full specialization inherits **nothing** from the primary; redefine all needed members.

```
template <typename T>
struct Formatter { void print(T v); };

template <>
struct Formatter<bool> { void print(bool v); };
```

14.2 Partial Specialization

- For a **family** of types, some parameters left open.
- Only **class/struct** templates (functions cannot).

```
template <typename T>
struct Formatter<T*> { void print(T* p); }; // any pointer
```

14.3 Constraint Based Specialization

- Partially specialize using **concepts**; compiler picks the **most constrained** match.

```
template <std::integral T>
struct Formatter<T> { void print(T v); };
```

14.4 Selection Mechanism

- Class template specialization = **type/value selection** by matching the template argument: exact full specialization wins; else most specialized partial; else primary.
- It selects a **type**, never a function. Function choice is a separate **overload resolution** step.

15 Nontype Template Parameters (NTTP)

- Template parameter representing a **compile time value**, not a type.
- Valid: **Structural Types** e.g., integers (`int`, `size_t`), enums, pointers/references, floats, literal class types.
- `template <typename T, std::size_t N> struct FixedBuffer;`
- `auto` deduces the value type: `template <auto V> void print();` → `print<42>()` (int), `print<'A'>()` (char).

16 Variable Templates

- Family of variables/constants parameterized by type.
- Use: type traits, config flags (`std::is_integral_v<T>`).
- Fallback pattern: primary sets a default, specializations override.

```cpp
template <typename T>
constexpr bool is_integer = false; // primary (default)

template <>
constexpr bool is_integer<int> = true; // override
```

17 SFINAE

Substitution **F**ailure **I**s **N**ot **A**n **E**rror: if substituting deduced template arguments produces an invalid type in a candidate's signature, that candidate is silently **dropped from the overload set** instead of erroring (error only if **none** remain). Lets us constrain a template to certain types.

A bare trait is not enough: `std::is_class_v<T>` is just `true`/`false`. Both valid, so the overload is always picked. `std::enable_if_t<bool, T = void>` fixes this:

- condition `true` ⟶ it **is** the type `T`.
- condition `false` ⟶ it names **no type**, signature ill formed ⟶ candidate dropped.

The 3 places to apply `enable_if` (function template `increment`):

```cpp
// 1. Return type
template <typename T>
auto increment(T value) -> std::enable_if_t<std::is_class_v<T>, T> { ... }

// 2. Function parameter (impairs deduction: T cannot be deduced from the arg)
template <typename T>
auto increment(std::enable_if_t<std::is_class_v<T>, T> value) -> T { ... }

// 3. Defaulted unnamed template parameter
template <typename T, typename = std::enable_if_t<std::is_class_v<T>, void>>
auto increment(T value) -> T { ... }
```

NOTE: must be a real **substitution** failure. `enable_if_t<...>` inside the function **body** is not SFINAE; the overload is already chosen, so a failed condition there is a hard error.

18 Concepts

Modern replacement for `enable_if`: encode type requirements directly, more readable, **much better errors** (names the violated constraint). Unsatisfied constraints exclude the candidate from overload resolution, like SFINAE but explicit.

A **`requires` clause** constrains a template with a compile time bool (after the parameter list or the declarator):

```
template <typename T>
requires std::is_class_v<T>
auto increment(T value) -> T { return value.increment(); }
```

A **`requires` expression** is a bool expression checking that operations are valid (hence the double `requires`).

```
requires requires (T const v) {
    v.increment();                        // Simple:   statement compiles
    typename T::value_type;               // Type:     nested type exists
    { v.increment() } -> std::same_as<T>; // Compound: expr valid + optional return type
    requires std::is_class_v<T>;          // Nested:   another requires
}
```

A **`concept`** names a reusable constraint (combinable with `&&` / `||`):

```
template <typename T>
concept Incrementable = requires (T const v) { { v.increment() } -> std::same_as<T>; };

template <Incrementable T> // constrained template parameter
auto increment(T value) -> T { ... }

auto increment(Incrementable auto value) { ... } // terse / abbreviated syntax
```

Standard library (`#include <concepts>`): `std::equality_comparable`, `std::default_initializable`, `std::floating_point`, `std::same_as`, …, see: https://en.cppreference.com/w/cpp/concepts

19 Manual Storage for Nondefault Constructible Types

- Raw byte buffer instead of `T[]` → avoids default constructing elements.
- `std::construct_at(ptr, args...)` builds one element; `std::destroy_at(ptr)` destroys it.
- `reinterpret_cast<T*>` views the bytes as `T`.
- `emplace_back` uses perfect forwarding.

```cpp
template <typename T, std::size_t Capacity>
class FixedVector final {
  public:
    template <typename... Args>
    auto emplace_back(Args&&... args) -> T& {
        T* pos = ptr_at(size_++);
        return *std::construct_at(pos, std::forward<Args>(args)...);
    }
    auto operator[](std::size_t i) -> T& { return *ptr_at(i); }
    ~FixedVector() { while (size_ > 0) std::destroy_at(ptr_at(--size_)); }
  private:
    std::byte storage_[sizeof(T) * Capacity];
    std::size_t size_{};
    auto ptr_at(std::size_t i) -> T* { return reinterpret_cast<T*>(storage_) + i; }
};
```

20 Lambda Expressions

```cpp
[capture] (parameters) -> return_type { body }
```

- **Capture** `[ ... ]`: which surrounding variables are available inside. Can be empty.
- **Parameters**: like normal params; `auto` params (C++14+) make a generic lambda.
- **Return type** `-> T`: optional.

Capture	Meaning
`[]`	nothing (noncapturing → convertible to function pointer)
`[=]`	all used variables by copy (at lambda creation)
`[&]`	all used variables by reference
`[var]`	`var` by copy
`[&var]`	`var` by reference
`[=, &var]`	default copy, `var` by reference
`[&, var]`	default reference, `var` by copy
`[this]`	`this` pointer by value
`[*this]`	copy by value (C++17)
`[var = expr]`	init capture; the only place `=` precedes a name

20.1 `mutable` Lambdas

- Call operator `operator()` is `const` by default → cannot modify **by value** captures.
- `mutable` (after the parameter list) removes that constness.
- Affects **by value** captures only. By reference captures are modifiable regardless (if original is nonconst).

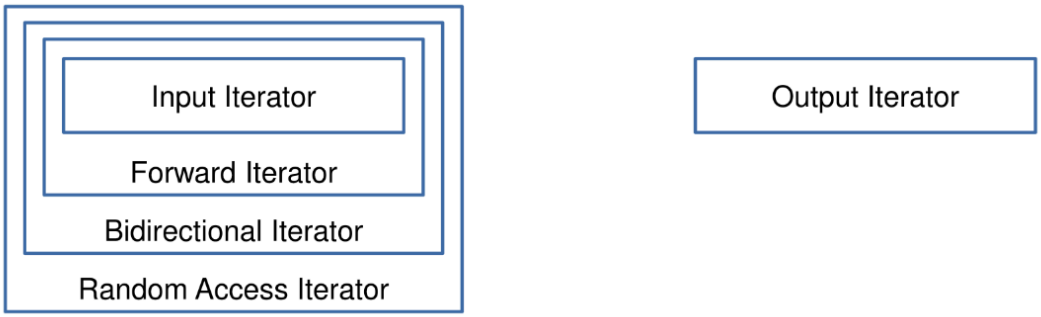
```cpp
auto counter = [count = 0]() mutable { return ++count; };
```

21 Containers

21.1 `std::vector`

- `std::vector v(10, 2);` → ten copies of `2` → `{2,2,2,...}`.
- `std::vector v{10, 2};` → `{10, 2}` (initializer_list wins).

22 Iterators



Each outer category includes all traits of the inner one (input → forward → bidirectional → random access). Output iterators are separate.

- **Input iterator**: read only, forward only, one element at a time; copyable; comparable with `==` / `!=`. E.g. `std::istream_iterator`.
- **Forward iterator**: also can change the current element. E.g. `std::forward_list`.
- **Bidirectional iterator**: also can go backwards. E.g. `std::set`.
- **Random access iterator**: also direct index access. E.g. `std::vector`.
- **Output iterator**: write to current element once, then must increment.

23 User Defined Literals (UDL)

Attach a custom **suffix** to a literal → builds an object via an `operator""` function.

```cpp
struct Distance { double meters; };
auto operator""_km(long double v) -> Distance {
    return Distance{static_cast<double>(v) * 1000};
}
auto d = 5.0_km; // calls operator""_km(5.0)
```

- **Suffix must start with `_`**. Suffixes without a leading underscore (`s`, `min`, `i`) are **reserved for the standard library** (`operator""s`, `std::chrono`). Using one is UB / warning.
- **Parameter type is restricted**: you **cannot** use arbitrary types like `std::string`. Allowed signatures only:

Literal kind	Allowed parameter
Integer	`unsigned long long` (or `const char*` raw)
Float	`long double` (or `const char*` raw)
Character	`char`, `wchar_t`, `char8_t`, `char16_t`, `char32_t`
String	`(const char* str, std::size_t len)` (+ char variants)

- `operator""_x(std::string)` is **ill formed**: input always arrives as a primitive. Convert **inside** the operator to get a class result.
- Widest types are used so any value fits (integers as `unsigned long long`, floats as `long double`).
- Make it `constexpr` where possible → evaluated at compile time.

24 Threads

- Every C++ program has at least one thread (runs synchronously).
- C++11 standard async support: `std::thread` (new thread), `std::async` (wrap a computation + result), `std::mutex` & co. (synchronization).
- Standard threading is **portable across platforms/compilers** because C++11 defines a memory model for concurrent execution, unlike raw `pthreads`.
- Terminology: `std::thread` = the C++ object; thread = the OS execution agent it manages.

24.1 `std::thread`

- Construction **creates and starts** a new OS thread immediately.
- Runs a lambda, free function, or functor; the callable's **return value is ignored**.
- Default constructible and **movable** (not copyable).

```
auto worker = std::thread{[] { std::cout << "Hello from thread\n"; }};
worker.join();
```

24.2 Passing Arguments

The constructor forwards extra arguments to the callable: `std::thread t{printFib, 46};`

- ⚠️ **Pass arguments by value** where possible. Passing by reference (or `[&]` capture) creates **shared data** and risks **dangling references** if the referent dies before the thread finishes.

24.3 join() vs detach()

Before a `std::thread` is **destroyed** you must `join()` or `detach()` it, else the destructor calls `std::terminate()`.

	`join()`	`detach()`
Effect	**blocks** until the thread finishes	thread runs **independently** in background
After	resources reclaimed	`std::thread` no longer owns it

- Destructor does **neither** by design: auto join could hang (e.g. on exception); auto detach could run on **deallocated** resources.
- With `detach()` beware lifetimes: when `main()` ends, globals like `std::cout` are destroyed while the detached thread may still use them ⟶ dangling references.

24.4 `std::jthread` (C++20)

RAII wrapper that **automatically joins** on destruction and supports **cooperative cancellation** via `std::stop_token`.

```
std::jthread t{[](std::stop_token token) {
    while (!token.stop_requested()) { /* work */ }
}};
t.request_stop(); // any thread may request stop; auto joins at scope end
```

24.5 `std::this_thread` Helpers

- `get_id()` ⟶ unique id of the OS thread (usable as a map key). Also a `std::thread` member.
- `sleep_for(duration)` / `sleep_until(time_point)` ⟶ suspend the thread.
- `yield()` ⟶ hint the OS to schedule another thread.
- NB: timing/sleeping does **not** guarantee execution order.

24.6 Standard Streams

- Writing to `std::cout` from multiple threads is **not a data race** (synchronized), but **character order can interleave** (`HHeellllloo MTaihrnead`) since the output order is not given.

25 Mutexes

A **mutex** (mutual exclusion) serializes access to **mutable shared state** to prevent a **data race**.

A **data race** occurs when:

1. two operations access the **same** memory location,
2. **at least one is not atomic**, and
3. **at least one modifies** it.

Fix: **lock** the access, or make it **atomic**.

25.1 Mutex Operations

Operation		
Acquire	`lock()`	blocking
	`try_lock()`	nonblocking (returns `bool`)
Release	`unlock()`	nonblocking
Timed	`try_lock_for(dur)` / `try_lock_until(tp)`	timed mutexes only

25.2 Mutex Types

Two properties: **recursive** (same thread may re acquire → prevents self deadlock) and **timed**.

	not recursive	**recursive**
not timed	`std::mutex`	`std::recursive_mutex`
timed	`std::timed_mutex`	`std::recursive_timed_mutex`

- **Shared mutex** (`#include <shared_mutex>`): reads need no exclusive access, only concurrent writes do. `std::shared_mutex` (C++17) / `std::shared_timed_mutex` (C++14) add `lock_shared()`, `try_lock_shared()`, `unlock_shared()` for read locking alongside exclusive `lock()`.

25.3 Locks (RAII wrappers, prefer over manual `lock()`/`unlock()`)

Lock	**Purpose**
`std::lock_guard`	RAII for **one** mutex: locks on construction, unlocks on destruction
`std::scoped_lock`	RAII for **multiple** mutexes at once, **deadlock free** (all or none). C++17
`std::unique_lock`	movable; allows **deferred** (`std::defer_lock`), **timed**, explicit lock/unlock. Needed by condition variables
`std::shared_lock`	wrapper for **shared** (read) locking of a `shared_mutex`

- RAII matters: a `lock_guard` unlocks even if the protected code **throws** (manual `unlock()` would be skipped).

- The mutex member is usually **`mutable`** so it can be locked inside `const` member functions.

```
auto push(T const& t) -> void {
    std::lock_guard lk{mx}; // locks here, unlocks at scope end (even on throw)
    q.push(t);
} // CTAD deduces lock_guard<std::mutex>
```

25.4 Locking Multiple Mutexes (deadlock avoidance)

Locking two mutexes in different orders in different threads can cause a **deadlock**. Acquire all or nothing:

```
auto swap(threadsafe_queue<T>& other) -> void {
    if (this == &other) return;
    std::scoped_lock both{mx, other.mx}; // C++17: locks both, no deadlock
    std::swap(q, other.q);
}
```

25.5 Condition Variables (`std::condition_variable`)

Lets a thread **wait** until another signals a state change → avoids busy waiting.

- **Wait**: `wait(lock)` (needs a surrounding loop) or **`wait(lock, predicate)`** (loops internally, preferred, **guards spurious wakeups**). Also `wait_for` / `wait_until`.
- **Notify**: `notify_one()` / `notify_all()`.
- Requires a **`std::unique_lock`** (not `lock_guard`): `wait()` must **release** the lock while blocked and **reacquire** it on wakeup.

```
auto pop() -> T {
    std::unique_lock lk{mx};
    notEmpty.wait(lk, [this] { return !q.empty(); }); // releases lk while waiting
    T t = q.front(); q.pop();
    return t;
}
```

25.6 Standard Containers and Concurrency

- **No** thread safety wrapper for standard containers (yet).
- Accessing **different elements** of e.g. a `std::vector` from different threads is **OK**, provided the container itself does not change (no resize/insert/erase) during access.
- Almost every other concurrent use is dangerous.
- `std::shared_ptr`: copying the same pointer across threads is fine (the **refcount is atomic**), but accessing the **pointee** can race if it is nonconst.

26 Futures and Async

- `std::future<T>`: represents a result computed asynchronously.
  - `get()` → blocks until ready, returns the value or **rethrows** the stored exception.
  - `wait()` / `wait_for(dur)` / `wait_until(tp)` → block without consuming.
- `std::async(f, args...)` → runs `f` (maybe on a new thread), returns a `std::future` for its result. Cleanest way to get a value back from a task.
- `std::promise<T>`: the producer side, one origin of a future. `get_future()` to hand out the future; `set_value(v)` or `set_exception(eptr)` to publish.

```
auto fut = std::async([] { return 42; });
std::cout << fut.get(); // 42 (blocks until ready)
```

Promise + future across a thread:

```
std::promise<int> promise{};
std::future<int> result = promise.get_future();

std::thread worker{[&] {
    promise.set_value(42); // publish (or set_exception)
}};

std::cout << result.get() << '\n'; // blocks until the value is set -> 42
worker.join();
```

Advantage over shared state + mutex + condition variable: **exceptions propagate** through `get()`.

26.1 Launch Policy

`std::async(policy, f, args...)` takes an optional **launch policy** (enum `std::launch`) that controls **where/when** `f` runs. ⚠️ Without a policy, `std::async` does **not** guarantee a separate thread.

Policy	Behavior
`std::launch::async`	runs `f` on a **new thread** immediately
`std::launch::deferred`	**lazy evaluation**: `f` runs only when the future's `get()` is called, **on the calling thread** (no new thread)
default	implementation chooses either

```
auto a = std::async(std::launch::async,    [] { return 42; }); // own thread
auto d = std::async(std::launch::deferred, [] { return 42; }); // runs at d.get()
```

- `std::launch::async` tasks run **regardless** of whether the result is ever needed.
- `std::launch::deferred` runs **only if** `get()` is called, and then on the `get()` caller's thread.

27 Atomics

- `std::atomic<T>`: makes operations on `T` **data race free**. Lets threads share state without a mutex.
- `std::atomic_flag`: most basic atomic, the **only one guaranteed lock free**. `clear()` (set false), `test_and_set()` (set true, return old). Others might lock internally → check `is_lock_free()`.
- Core operations (all atomic):

Operation	Effect
`store(v)`	set value
`load()`	read value
`exchange(v)`	set new value, return **old**
`compare_exchange_weak(expected, desired)`	Compare-And-Swap: if current == `expected` write `desired` (true); else write current into `expected` (false). May **fail spuriously** → use in a loop
`compare_exchange_strong(...)`	never fails spuriously, may be slower

- `std::atomic<int>` etc. also provide atomic `++`, `--`, `+=`.
- Custom type in `std::atomic<T>` must be **trivially copyable** (no custom copy/move ctor, no custom copy/move assignment). Accessed **as a whole** only (no member access operator).

27.1 What atomics guarantee (and not)

- **Guarantee**: each single operation is **indivisible** (no torn read/write) + a chosen **memory ordering**.
- **Not guaranteed**:
  - compound logic is **not** atomic (e.g. `load` then `store` → use a single Compare-And-Swap).
  - multiple atomics are **not** kept mutually consistent → use a **mutex** for invariants across several variables.
  - does **not** remove logical race conditions / deadlocks.
- Rule of thumb: atomic = one variable / one operation; mutex = critical section / invariant across state.

28 Memory Model

- `volatile`: prevents the compiler from reordering within the same thread, but the hardware might still reorder. **Not** a threading tool.
- Every atomic operation takes an optional `std::memory_order` controlling how surrounding memory ops may be reordered. Default = `seq_cst` (strongest, safest).

Running example (each function on its own thread): `z` counts how many readers see **both** flags true:

```
std::atomic<bool> x, y;  std::atomic<int> z;
write_x():       x.store(true);
write_y():       y.store(true);
read_x_then_y(): while(!x.load());  if (y.load()) ++z;
read_y_then_x(): while(!y.load());  if (x.load()) ++z;
```

Memory order	possible `z`	meaning
`seq_cst` (default)	1 or 2	single global order all threads agree on. `z == 0` **impossible**
`acquire` / `release`	0, 1, 2	sync only **through the same atomic**, no global order
`relaxed`	0, 1, 2	atomicity only, no cross variable ordering

28.1 Sequential Consistency (`seq_cst`)

- One global total order of all atomic ops; every thread observes the same order.
- Whichever interleaving the scheduler picks (so `z` is 1 or 2 across runs), it is always explainable by one consistent order ⟶ `z == 0` cannot happen.
- Dashed arrow = **implied ordering**: seeing one flag false forces the other reader to see its flag true.

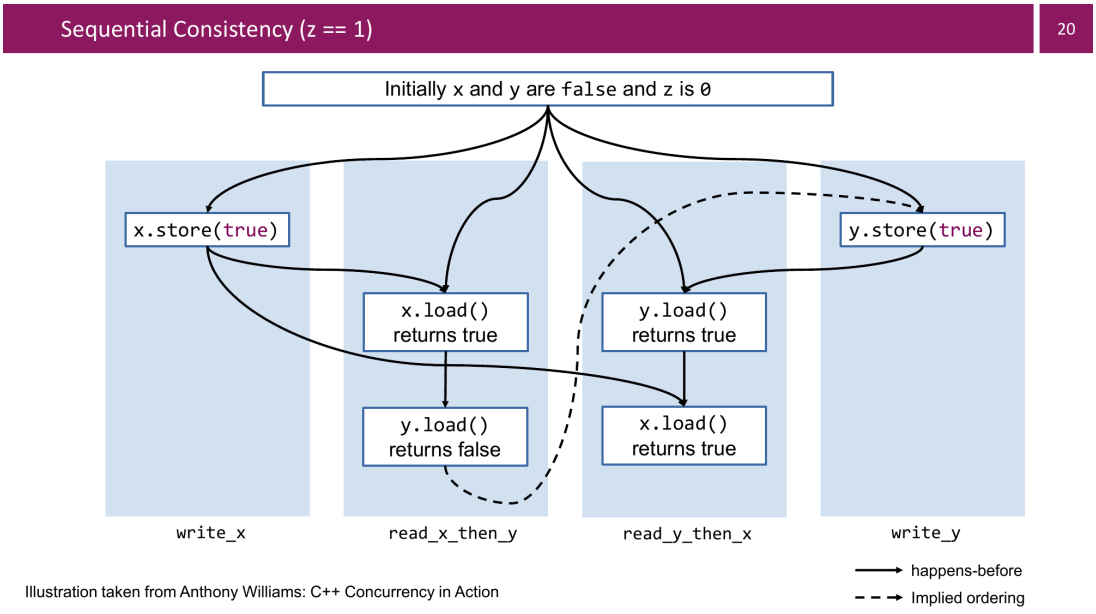


Illustration taken from Anthony Williams: C++ Concurrency in Action

28.2 Acquire / Release

- `acquire` (load): no later op in this thread moves **before** it; sees all writes a thread did before its matching `release` on the **same atomic**.
- `release` (store): no earlier op in this thread moves **after** it; publishes prior writes to a thread that **acquires the same atomic**.
- `acq_rel`: both, for read modify write ops.

- ⚠️ Not guaranteed to see the **latest** write, only a value consistent with the ordering of that **same atomic**.
- Because the `x` chain and `y` chain are independent (no global order), both readers can see the other flag false ⟶ `z == 0` **possible**.
- Good for the **publish / handshake** pattern (one flag): producer `store(release)`, consumer `load(acquire)`.

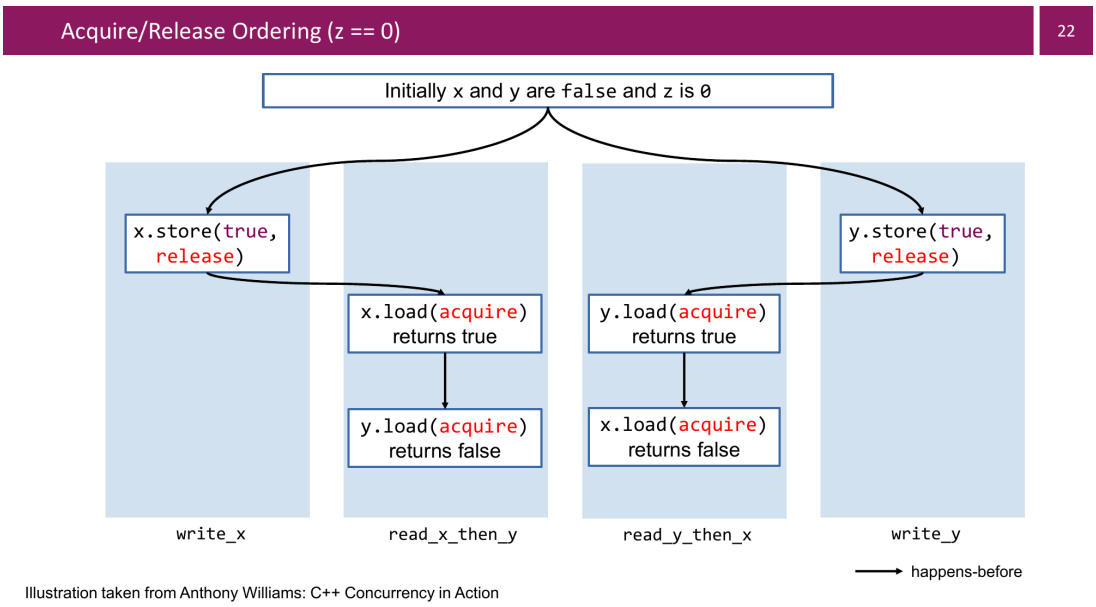


Illustration taken from Anthony Williams: C++ Concurrency in Action

28.3 Relaxed

- Atomicity only, **no** sequencing promises across variables.
- `load` / `store` may appear reordered / in parallel; threads can disagree on order of effects.
- Faster (less synchronization), but **very hard to get right** ⟶ must prove correctness.

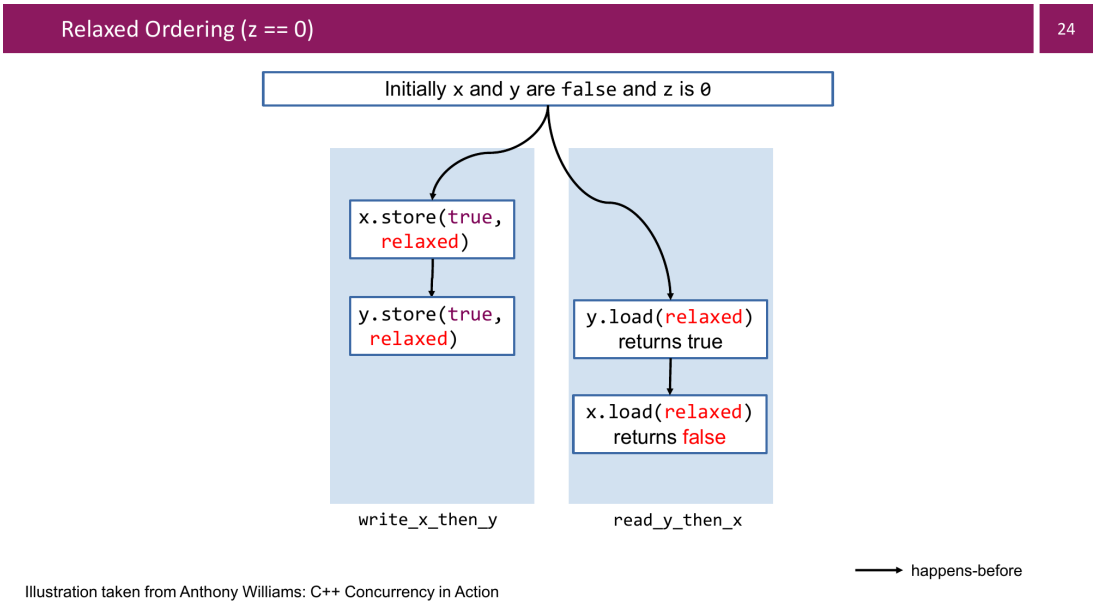


Illustration taken from Anthony Williams: C++ Concurrency in Action

28.4 Release / Consume

- Like acquire/release but synchronizes only **dependency ordered** data. Subtle and hard to use.
- **DO NOT USE** `consume`: the standard itself recommends `acquire` instead.

29 Networking (ASIO)

- Networking is **not** in the standard yet (networking TS, >C++23). Use **ASIO** (boost or standalone).
- `asio::io_context`: required by all operations; `io_context.run()` drives execution and dispatches **completion handlers**.

29.1 TCP vs UDP

	TCP	**UDP**
Reliability	reliable	packets may be lost / reordered
Style	stream oriented	datagram oriented (max 65k)
Connection	requires setup	connectionless (send/receive without connection)

29.2 Socket Connection Pattern (Berkeley primitives)

- **Server**: `socket → bind → listen → accept → read/write → close`
- **Client**: `socket → connect → write/read → close`
- `accept()` blocks until a client connects and returns a **new** socket for that client (the **acceptor** is a special socket bound to a local endpoint).

29.3 Synchronous vs Asynchronous

- **Synchronous**: blocks the calling thread; nothing else runs while blocked.
- **Asynchronous**: returns immediately, lets other work proceed while the OS performs the IO.

29.4 ASIO Async Model

- `async_*` operations (`async_read`, `async_write`, `async_accept`, …) **return immediately**.
- The OS performs the operation; when done, the **completion handler** is called from inside `io_context.run()`.
- Handler signature: `(asio::error_code ec, std::size_t length)`.

```
asio::async_read_until(socket, buffer, '\n',
    [](asio::error_code ec, std::size_t length) { /* handle result */ });
```

- Without callbacks: `asio::use_future` (returns `std::future`), `asio::detached` (ignore result), `asio::use_awaitable` (coroutine, C++20).

29.5 Session Lifetime (`enable_shared_from_this`)

- An async handler may run **after** the function that started it returned → the object must **outlive** the pending operation.
- Pattern: allocate the session on the heap as a `shared_ptr`, and each handler captures **`shared_from_this()`** so the `shared_ptr` keeps the object alive until the handler runs.

```
struct Session : std::enable_shared_from_this<Session> { ... };

// in accept handler:
auto session = std::make_shared<Session>(std::move(peer));
session->start();

// in a member starting an async op:
auto handler = [self = shared_from_this()](asio::error_code ec, std::size_t n) { ... };
```

29.6 Strands (avoiding data races between handlers)

- Multiple threads calling `run()` on the **same** `io_context` → handlers run **concurrently** → **data race** on shared state (e.g. `results.push_back(...)` from two read handlers).
- A **strand** guarantees **sequential** execution of the handlers posted through it → no concurrent access, no mutex needed.
- **Implicit strand**: only one thread calls `run()`, or program logic ensures one operation at a time.
- **Explicit strand**: `asio::make_strand(context)`, then wrap the handler with `asio::bind_executor(strand, handler)`.

```
auto strand = asio::make_strand(context);
asio::async_read(socket, asio::buffer(buffer),
    asio::bind_executor(strand, [&](auto ec, auto n) {
        results.push_back(parse(buffer)); // no data race: serialized by strand
    })
);
```

29.7 Signal Handling

- Portable signal subset in `<csignal>` (`SIGINT`, `SIGTERM`, `SIGSEGV`, `SIGABRT`, …); OS specific signals are not portable.
- ASIO: `asio::signal_set{context, SIGINT, SIGTERM}` + `async_wait(handler)`; handler gets the signal number (or an error if aborted). Used for **clean server shutdown**.

30 noexcept and Exception Safety

- `noexcept` is a **compile time** check: the condition must be a **constant expression** (`sizeof`, template parameter, type trait). It can **not** depend on a runtime value (`vec.size() > 9` is invalid). The throwing status is part of the function type.

30.1 Syntax

- `noexcept` = specifier `noexcept(true)`: function never throws.
- `noexcept(cond)` = specifier with a compile time bool: noexcept iff `cond` is `true` (e.g. `noexcept(sizeof(T) > 4)`).
- `noexcept(expr)` = the **operator**: compile time bool, `true` if evaluating `expr` can not throw (`expr` is unevaluated; its value is irrelevant).
- `noexcept(noexcept(expr))` = specifier fed by the operator: noexcept iff `expr` can not throw → **propagates** the noexcept status of a called function.

```
1 void f() noexcept;                      // never throws
2 void g() noexcept(sizeof(T) > 4);       // condition (compile time value)
3 void h() noexcept(noexcept(a.swap(b))); // noexcept if a.swap(b) is noexcept
```

30.2 Where no-throw matters

- **Destructors** must not throw, especially during **stack unwinding** (an exception already in flight + a throwing destructor → `std::terminate`).
- **Move construction / move assignment** better not throw → else containers fall back to copying (`move_if_noexcept`).
- `swap` should not throw; `std::swap` requires **non throwing move operations**.
- **Copying** might throw (it may need to **allocate** memory) → copies are generally not noexcept.

30.3 Exception Safety Guarantees

Each property holds **if an exception is thrown** during the operation. The levels are cumulative (each adds one column).

Guarantee	Invariant OK	All or Nothing	Will Not Throw
No Guarantee	✗	✗	✗
Basic Guarantee	✓	✗	✗
Strong Guarantee	✓	✓	✗
No-Throw Guarantee	✓	✓	✓

- **Invariant OK**: object stays **valid** (safe to use, assign, destroy); value may have changed.
- **All or Nothing**: rollback; if it throws, observable state is **unchanged**.
- **Will Not Throw**: operation can not fail.
- **No-throw (`noexcept`)**: guaranteed to succeed, never throws. Required for destructors, `swap`, and move operations used in strong guarantee commits.

What the guarantee level means:

- **Strong**: all or nothing. If it throws, observable state is unchanged, as if the call never happened. Typically via **copy and swap** (do risky work on a side copy, commit with a noexcept operation).
- **Basic**: if it throws, no resources leak and invariants still hold, but the object may be in a **valid but unspecified** state. Safe to destroy, assign to, or reset; just do not rely on its contents. **Minimum acceptable level** for most code.
- **None**: no promises. Invariants may be broken, resources may leak. The exception itself is well defined, but **using** the object afterwards (including its destructor) is likely UB.

30.4 Wide vs Narrow Contracts

- **Wide contract**: handles **all** argument values of the given parameter types successfully → it **cannot fail** → should be `noexcept`.
  - "Argument values" also covers **globals**, **external resources**, and the **heap**, not just parameters.
- **Narrow contract**: has **preconditions** on its arguments.
  - E.g. an `int` parameter must not be negative; a pointer parameter must not be `nullptr`.
  - Even if the precondition is **not checked** and **no exception** is thrown, such functions should **not** be `noexcept`.
  - Reason: leaving it non noexcept allows a **later** version to add a check and **throw** on a precondition violation (UB) instead of being locked into `std::terminate`.

31 Opaque Types (Incomplete Types)

- **Name declared, content/structure unknown** → introduced by a **forward declaration** (`struct S;`).
- Usable as **pointer / reference**; **not** dereferenceable, **no** member access, **no** object creation, **no** `sizeof` until the full definition is in scope.
- **C** uses pointers only → **`void*`** = universal opaque pointer; castable to any pointer type. Validity / UB avoidance is on the **programmer**.
- `std::byte*` = raw memory of a given size (see `BoundedBuffer`).

```cpp
struct S;               // forward declaration -> incomplete
auto foo(S& s) -> void; // OK: reference
// S s{};               // INVALID: definition not visible
struct S {};            // now complete
S s{};                  // now valid
```

Used by the **PIMPL idiom** and to **cut compile time dependencies** (forward declare instead of `#include`).

32 PIMPL (Pointer to IMPLementation)

- **Problem**: change a private member ⟶ **all clients recompile**.
- **Idiom**: header = pointer to forward declared `Impl` + public members only; real data + private members ⟶ `.cpp`.
- **Effect**: **compilation firewall** (impl changes do not touch header ⟶ no client recompile) + hides details.
- **Needs**: `Impl` is an **incomplete/opaque type** in the header ⟶ usable via pointer/reference, **not** dereferenceable.

Header (`Wizard.hpp`): minimal, no includes for implementation details.

```cpp
class Wizard {
    std::unique_ptr<class WizardImpl> pImpl; // forward declared, incomplete here
public:
    Wizard(std::string name);
    ~Wizard(); // see trap below (32.1 "The UNIQUE_PTR Destructor Trap")
    auto doMagic(std::string wish) -> std::string;
};
```

Source (`WizardImpl.cpp`): full `WizardImpl` definition + delegation.

```cpp
class WizardImpl { /* all real members + private methods */ };

Wizard::Wizard(std::string name)
    : pImpl{std::make_unique<WizardImpl>(name)} {}

auto Wizard::doMagic(std::string wish) -> std::string {
    return pImpl->doMagic(wish); // delegate to the impl
}

Wizard::~Wizard() = default; // defined HERE, where WizardImpl is complete
```

32.1 The unique_ptr Destructor Trap

- `unique_ptr<Impl>` deletes via **default deleter** ⟶ needs **complete** `Impl`.
- Implicit destructor sits at **end of class** in header ⟶ `Impl` still **incomplete** ⟶ compile error (`sizeof` of incomplete type).

- **Fix**: declare `~Wizard();` in header, **define** `= default` **in** `.cpp` (where `Impl` is complete).
- `shared_ptr<Impl>`: no trap (deleter type erased at construction). `unique_ptr` = lighter default.

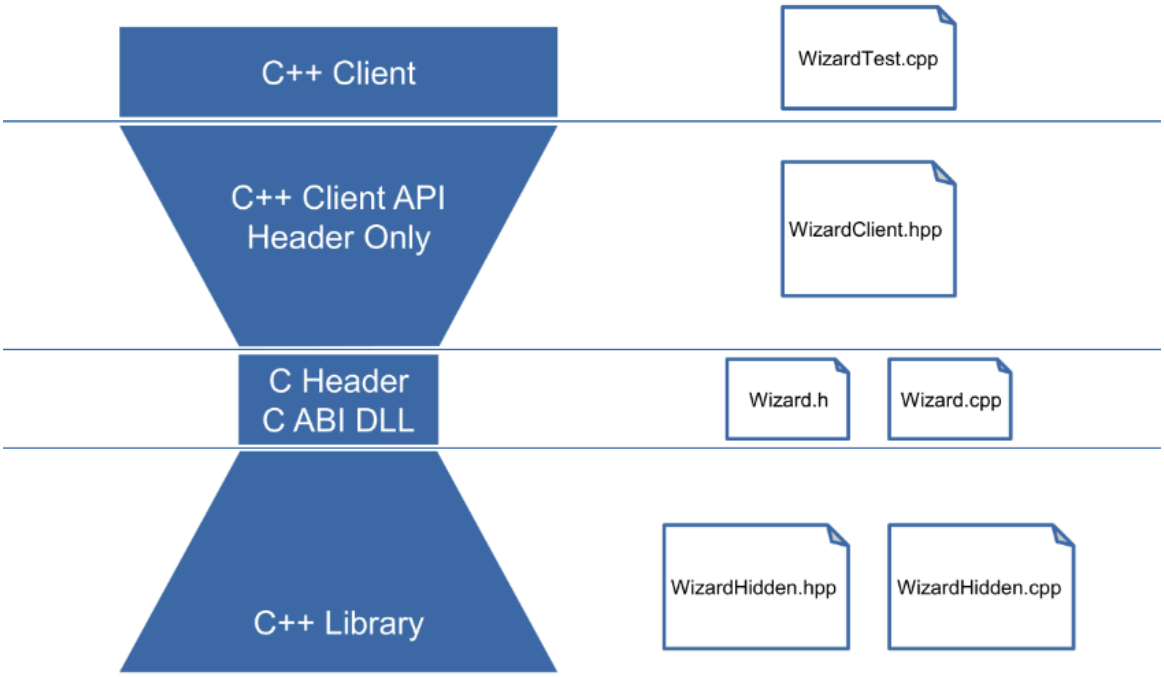
32.2 Copy Behavior (design decision)

Goal	Use
No copy, only move	`unique_ptr<Impl>` + declared destructor and move ops `= default`
Shallow copy (share the impl)	`shared_ptr<Impl>`
Deep copy (C++ default semantics)	`unique_ptr<Impl>` + hand written copy constructor (copies `*Impl`)

- `pImpl` should **never** be `nullptr`.
- Avoid inheriting from a PIMPL class.

33 Hourglass interface

- **Goal**: expose a C++ library to **any language**, despite C++ having **no stable ABI**.
- **Why**: C++ ABI is unstable (name mangling, differs per compiler/version/stdlib) → C++ does **not** define an ABI. **C has a stable ABI** (no namespaces, no name mangling).
- **Shape** (the "hourglass"): wide **C++ library** (bottom) → thin **C ABI** waist (C header + DLL) → wide **client API** in any language (top: C++, Java, …).
- C cannot do member functions / exceptions / templates → the waist works around all three.



33.1 The C ABI waist

- **`extern "C"` blocks**: disable name mangling so C can link. Guard with `#ifdef __cplusplus` so the same header compiles as C and C++.

```
#ifdef __cplusplus
extern "C" {
#endif
    typedef struct Wizard* wizard; // opaque handle
    wizard      createWizard(const char* name, error_t* out_error);
    void        disposeWizard(wizard w);
    const char* doMagic(wizard w, const char* wish, error_t* out_error);
#ifdef __cplusplus
}
#endif
```

- **Abstract data types → opaque pointers**: forward declared `struct` → `void*` / `typedef struct Wizard*`. (See Opaque Types.)
- **Member functions → free functions** taking the object pointer as **first argument**.
- **Lifetime → factory + disposal functions** (`createWizard` / `disposeWizard`); no constructors/destructors across C.
- **Strings →** `char const*` only; ownership must be clear; never return a pointer to a temporary.
- Usable parts in an `extern "C"` interface: functions (no overload/template/variadic), C primitives, pointers (incl. function pointers), forward declared structs, **unscoped** enums.

33.2 Trampoline class

- A **C linkage struct** (defined inside `extern "C"`) that **wraps the real C++ implementation** → lets the library use **full C++ (templates, etc.)** internally while the handle stays C compatible.

```
extern "C" {
struct Wizard { // C linkage trampoline
    Wizard(char const* name) : wiz{name} {}
    unseen::Wizard wiz; // real C++ implementation (hidden namespace)
};
}
```

33.3 Error handling (exceptions do not cross a C ABI)

- No references in C → use **pointer to pointer** (`error_t* out_error`).
- On error: **allocate the error value on the heap** → you **must provide a disposal function** (`error_dispose`).
- `translateExceptions`: run the body, **catch all**, map to an `Error` object, set `*out_error`.

```cpp
template<typename Fn>
bool translateExceptions(error_t* out_error, Fn&& fn) try {
    fn(); return true;
} catch (std::exception const& e) { *out_error = new Error{e.what()}; return false; }
  catch (...) { *out_error = new Error{"Unknown error"}; return false; }
```

- You **can use C++ types internally** (e.g. `std::string` member in `Error`).
- It is **safe to return** `char const*` (`error->message.c_str()`) → the **caller owns** the `Error` object providing the memory (it disposes it later).
- **Client side**: map error codes back to exceptions:
  - `ErrorRAII` → owns `error_t`, calls `error_dispose` in destructor.
  - `ThrowOnError` → converts to `error_t*` (via `operator error_t*`), **throws** `std::runtime_error{error_message(...)}` in its destructor if an error was set.

33.4 Client side + hiding symbols

- **Header only** inline wrappers delegate to the C functions; copy ops `= delete`; take care with passed/returned `char*` (no dangling).
- Really hide DLL symbols (compiler dependent): GCC/Clang `-fvisibility=hidden` + mark exported ABI functions with **default visibility**; Windows `__declspec(dllexport)`.

33.5 JNA (Java Native Access)

- Simple interface to **C libraries** from Java. Community library (not JDK), based on **JNI**, **generates interfaces at runtime**, single JAR, cross platform.
- Declare a Java `interface extends Library`; `Native.load("cpla", CplaLib.class)` loads `libcpla.so` / `.dylib` / `.dll`.
- **Function names must match**; ⚠️ parameter **types are not validated** (even at runtime, since C signatures carry no types). Parameter names do not matter.
- Type mappings: `char→byte`, `wchar_t→char`, `long→NativeLong`, `char*→String`, `T*→Pointer`, `struct→Structure`.