Module 11 of 20

Operator overloading

Which operators to overload and where they live, the compound-first idiom, == and the C++20 spaceship with its ordering categories, << and >> that respect the stream, operator[] and operator() with functors and std::hash, and explicit on constructors and conversions.

A value type earns its keep when it reads like a built-in: `total += price * qty`, `if (a < b)`, `std::cout << money`, `grid[r][c]`, `if (result)`. C++ lets a class define almost every operator, and that freedom is exactly what makes the rules matter — a member `operator*` that refuses `2 * v`, a `<` that is not strict and breaks `std::sort`, an `operator<<` that swallows the caller's `setw`, an implicit `operator bool` that lets `result + 1` compile. Every one of those is a bug that compiles, and this module is where you learn to write the operator set that does not contain them.

The five lessons go operator family by operator family. Overloading rules settles what can be overloaded, member versus free function (and the hidden-friend idiom), the compound-first idiom that derives `+` from `+=`, and prefix versus postfix increment. Comparison covers `==`, the strict weak ordering `<` must be, C++20's `operator<=>` with its rewritten expressions, `strong_ordering` / `weak_ordering` / `partial_ordering`, and what `std::sort`, `std::set` and `std::map` actually call. Stream operators covers `<<` and `>>` as free functions that return the stream, the failure contract of `>>`, respecting `setw` and sticky fill, and a `std::formatter` specialisation as reading. Subscript, call and function objects covers the const/non-const `operator[]` pair, `operator()` for two-dimensional access and for callable state, functors as comparators for `std::priority_queue` and `std::set`, and a `std::hash` specialisation for a custom key. Conversion operators covers converting constructors, `explicit`, `explicit operator bool`, the standard library's own surprising conversions and user-defined literals.

The exercises build the types the lessons describe: a `Fraction` normalised by `std::gcd` with `+ - * /` derived from the compound forms, a grid `Vec2` with unary minus and a scalar on either side, a `Version` sorted and deduplicated by a defaulted `<=>`, a case-insensitive `Tag` returning `std::weak_ordering`, a `Money` that round-trips `12.34` through `>>` and `<<` and honours `setw`, a `Point` reader that fails cleanly on `[1,1]`, a `Matrix` with `operator()(r, c)` and a `std::span` row view, a task queue driven by a functor comparator, a `Result` with `explicit operator bool`, and `Celsius` / `Fahrenheit` joined by explicit conversion operators. The checkpoint adds a `Polynomial` with arithmetic, `operator()` evaluation and a formatted `<<`; a grid walk keyed on a hashed `Point`

printed in sorted order; and a `Duration` ledger that reads `h:mm:ss`, sorts by a defaulted `<=>` and prints without leaking a fill character.

CONTENTS

1	Overloading rules — members, free functions and the compound-first idiom	14 min
2	Comparison and the spaceship operator	15 min
3	Stream operators — printing and reading your own types	14 min
4	Subscript, call and function objects	15 min
5	Conversion operators and explicit	14 min
✓	Checkpoint — Operator overloading	25 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Overloading rules — members, free functions and the compound-first idiom

`a + b` means addition for `int`, `double` and `std::string`, and C++ lets it mean addition for your `Fraction`, `Money` or `Vec2` too: `total += price * qty` instead of `total.add(price.times(qty))`. The mechanism is only a function with a strange name, but the rules about *which* operators, *where* the function lives and *what* it returns decide whether the result reads like arithmetic or like a trap. This lesson settles those rules and the one idiom — compound first, binary from compound — that produces a consistent set every time.

What an overloaded operator is

```
struct Vec2 {
    double x = 0;
    double y = 0;
};

Vec2 operator+(const Vec2& a, const Vec2& b) {
    return Vec2{a.x + b.x, a.y + b.y};
}

Vec2 p{1, 2}, q{3, 4};
Vec2 r = p + q;             // the compiler rewrites this as operator+(p, q)
Vec2 s = operator+(p, q);   // legal, and exactly the same call
```

An expression `p + q` where at least one operand has class or enumeration type is looked up as a call to a function named `operator+`. Nothing else changes: precedence, associativity and arity are fixed by the grammar, so `a + b * c` still multiplies first however you define `*`, and you cannot invent `**` or give `+` three operands. Nor can you redefine an operator on built-in types alone — `int operator+(int, int)` is an error.

Almost every operator can be overloaded: the arithmetic and bitwise set, the comparisons, `<<` and `>>`, `[]`, `()`, `->`, `=`, the compound assignments, `++` / `--`, `!`, `,` and the conversion operators (lesson 5). Five cannot: `.`, `.*`, `::`, `?:` and `sizeof`. Three that *can* be overloaded should not be — `&&`, `||` and `,` — because an overloaded version is an ordinary function call: both operands are always evaluated, and the short-circuit guarantee is gone.

Member or free function

An operator can be a member, in which case the left operand is `*this` and the parameter list holds only the right one, or a free function taking both operands:

```
class Money {
public:
    explicit Money(long long cents) : cents_(cents) {}
    long long cents() const { return cents_; }
    Money& operator+=(const Money& o) { cents_ += o.cents_; return *this; }
    friend Money operator+(Money a, const Money& b) { return a += b; }
private:
    long long cents_;
};

Money operator*(const Money& m, long long k) { return Money(m.cents() * k); }
Money operator*(long long k, const Money& m) { return m * k; }
```

The choice is not taste. Four operators **must** be members: `=`, `[]`, `()` and `→` (and the conversion operators). Everything that mutates its left operand — the compound assignments, `++`, `--` — is naturally a member too. Symmetric binary operators — `+`, `-`, `*`, `==`, `<`, `<<` — belong outside the class, for one concrete reason: a member operator's left operand is never subject to implicit conversion. With `Money::operator*(long long)` as a member, `price * 3` compiles and `3 * price` does not, because `3` is an `int` and there is no `int::operator*`. A free function treats both sides alike. In `std::cout << m` the left operand is a `std::ostream`, a class you cannot add members to, so `operator<<` is always free (lesson 3).

A free operator that needs private members is declared `friend` inside the class, and defining it there — as `operator+` above — gives the *hidden friend* idiom: a non-member (`*this` does not exist inside it) that is found only when an argument is a `Money`, which keeps it out of unrelated overload sets and error messages. It is the modern default for a class's binary operators.

Compound first, binary from compound

Write `+=` as the member that does the work; write `+` as a free function that copies its left operand, applies `+=` and returns the copy:

```
Vec2& Vec2::operator+=(const Vec2& o) { x += o.x; y += o.y; return *this; }
Vec2& Vec2::operator*=(double k)      { x *= k;   y *= k;   return *this; }

Vec2 operator+(Vec2 a, const Vec2& b) { return a += b; }   // a is the copy; -
Vec2 operator*(Vec2 v, double k)      { return v *= k; }
Vec2 operator*(double k, Vec2 v)      { return v *= k; }
```

Taking the left operand **by value** is deliberate: the operator needs a copy to modify anyway, and a by-value parameter lets the compiler build that copy straight from a temporary, so `(a + b) + c` moves the intermediate rather than copying it. The idiom guarantees `a + b` and `a += b` can never disagree — there is one implementation — and the same shape serves `-`, `*`, `/`, `%` and the bitwise pairs.

Unary operators and increment

Unary minus, `!` and `~` take no parameter as members (`Money operator-() const`) or one as free functions, return a new value and leave the operand alone. Increment and decrement come in two spellings that the language tells apart by a dummy parameter:

```
class Counter {
public:
    Counter& operator++()   { ++n_; return *this; }                      // p
    Counter operator++(int) { Counter old = *this; ++*this; return old; } // p
    int value() const { return n_; }
private:
    int n_ = 0;
};
```

The `int` is never used; it exists so the two overloads have different signatures. Prefix returns a reference to the object; postfix must return the previous value, which costs a copy — the reason `++it` is preferred over `it++` in loops. Implement postfix through prefix so the two cannot drift.

Return types

Operator	Return	Why
`+ - * / %`, unary `-`	by value	A new value; the operands are untouched
`==`, `<`	`bool`	A comparison, not a value of the type
`+= -= *= /=`, `=`, prefix `++`	`T&` (`*this`)	The same object, updated; chains like the built-ins
postfix `++` / `--`	`T` by value	The old value exists nowhere else
`<<` on a stream	`std::ostream&`	So `os << a << b` chains
`[]`	`T&` / `const T&`	The element itself, assignable (lesson 4)

Never return a reference to a local — it dangles the moment the function returns (Module 6, lesson 4). Mark every operator that does not modify `*this` as `const`, or it cannot be used on a `const Money` or through a `const&` parameter.

Do as the ints do

The reader of `a + b` brings expectations from the built-in types, and an overload that violates them is worse than a named function. `+` adds and does not mutate; `==` is symmetric and agrees with `!=`; `<` is a strict order consistent with `==` (lesson 2); `+=` does what `+` then `=` would do. A `Matrix` whose `*` is element-wise, a `String` whose `-` removes a substring — each compiles, and each costs every future reader a trip to the definition. The standard library breaks the rule once, deliberately: `std::filesystem::path` defines `/` as join because `dir / "file.txt"` reads like a path. If your overload needs a comment to say what it does, write a named function.

Pitfalls

Mistake	What happens
`operator*` as a member, then `2 * v`	No conversion on a member's left operand: no match
`+` written independently of `+=`	The two drift; one gets fixed, the other does not
`operator+=` returning `void` or by value	`a += b += c` fails, or copies
Postfix `++` implemented as prefix	`c++` yields the new value; loops off by one
No `const` on `operator==` or `operator-()`	Cannot be used on `const` objects or `const&` parameters
`friend` operator declared but never defined	Linker error: undefined reference

Key takeaways

An overloaded operator is a function named `operator@`; precedence, arity and the built-in meanings are untouchable.

`=`, `[]`, `()`, `→` and conversions must be members; symmetric binary operators should be free functions (hidden friends), so conversions apply to both operands.

Write `+=` as the member that works, and `+` as a one-line free function taking its left operand by value and returning `a += b`.

Prefix `++` returns `T&`; postfix takes a dummy `int` and returns the old value — implement it via prefix.

Return by value for results, `*this` by reference for assignments; mark non-mutating operators `const`.

Do as the ints do: an operator that needs a comment should be a named function.

EXERCISE 1

Fraction arithmetic

Complete the `Fraction` class so that it supports `+`, `-`, `*` and `/`. Every fraction is kept **normalised**: the sign lives in the numerator (the denominator is always positive) and numerator and denominator are divided by their `std::gcd`, so `2/4` is stored as `1/2`, `1/-3` as `-1/3` and `0/7` as `0/1`. Write `+=`, `-=`, `*=` and `/=` as members that normalise after the arithmetic, then the four

binary operators as free functions that take their left operand by value and return `a += b` (and so on) — the compound-first idiom.

Input: an integer `n`, then `n` lines of the form `a/b op c/d` where `op` is one of `+ - * /`, the four numbers are integers and no denominator is zero. The input never divides by a zero fraction. **Output:** `n` lines: the two operands **normalised**, the operator, and the result as `p/q`, e.g. `1/2 + 1/3 = 5/6`.

```
3
1/2 + 1/3
2/4 - 1/-3
3/4 * 2/3
```

prints

```
1/2 + 1/3 = 5/6
1/2 - -1/3 = 5/6
3/4 * 2/3 = 1/2
```

Use `long long` for the parts: the intermediate products can exceed `int`.

INPUT 1

```
3
1/2 + 1/3
2/4 - 1/-3
3/4 * 2/3
```

OUTPUT 1

```
1/2 + 1/3 = 5/6
1/2 - -1/3 = 5/6
3/4 * 2/3 = 1/2
```

INPUT 2

```
2
1/2 / 1/4
-6/8 / -3/2
```

OUTPUT 2

```
1/2 / 1/4 = 2/1
-3/4 / -3/2 = 1/2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <numeric>

class Fraction {
public:
    Fraction(long long num = 0, long long den = 1) : num_(num), den_(den) { normalis

    long long num() const { return num_; }
    long long den() const { return den_; }

    Fraction& operator+=(const Fraction& o) {
        num_ = num_ * o.den_ + o.num_ * den_;
        den_ *= o.den_;
        normalise();
        return *this;
    }
    Fraction& operator-=(const Fraction& o) { return *this += Fraction(-o.num_, o.de
    Fraction& operator*=(const Fraction& o) {
        num_ *= o.num_;
        den_ *= o.den_;
        normalise();
        return *this;
    }
    Fraction& operator/=(const Fraction& o) { return *this *= Fraction(o.den_, o.num

private:
    long long num_;
    long long den_;

    void normalise() {
        if (den_ < 0) {
            num_ = -num_;
            den_ = -den_;
        }
        const long long g = std::gcd(num_, den_);   // gcd(0, d) is d, so 0/7 become
        if (g > 1) {
            num_ /= g;
            den_ /= g;
        }
    }
};

Fraction operator+(Fraction a, const Fraction& b) { return a += b; }
Fraction operator-(Fraction a, const Fraction& b) { return a -= b; }
Fraction operator*(Fraction a, const Fraction& b) { return a *= b; }
Fraction operator/(Fraction a, const Fraction& b) { return a /= b; }

std::ostream& operator<<(std::ostream& os, const Fraction& f) {
    return os << f.num() << '/' << f.den();
```

```
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        long long a, b, c, d;
        char slash, op;
        std::cin >> a >> slash >> b >> op >> c >> slash >> d;
        const Fraction x(a, b), y(c, d);
        Fraction result;
        switch (op) {
            case '+': result = x + y; break;
            case '-': result = x - y; break;
            case '*': result = x * y; break;
            default:  result = x / y; break;
        }
        std::cout << x << ' ' << op << ' ' << y << " = " << result << '\n';
    }
    return 0;
}
```

EXERCISE 2

Displacement on a grid

Give the `Vec2` struct (integer `x` and `y`) the operators a vector needs: `+=`, `-=` and `*=` by an integer scalar as members; `+`, `-`, `v * k` **and** `k * v` as free functions derived from them; unary `-` as a `const` member; and a `length()` member returning `std::sqrt` of the squared length as a `double`. Then simulate a walk: a start position and a series of steps, each a velocity vector and a number of ticks, so that the position advances by `ticks * velocity` — use the scalar-on-the-left form to prove it exists.

Input: the start `x y`, then `n`, then `n` lines of `vx vy ticks` (all integers; `n` may be 0). **Output:** the position after each step as `(x, y)`; then `displacement (dx, dy)` (final minus start), `return (-dx, -dy)` (the unary minus of the displacement), and `distance <length of the displacement, one decimal>`.

```
0 0
2
3 4 1
-1 2 2
```

prints

```
(3, 4)
(1, 8)
displacement (1, 8)
return (-1, -8)
distance 8.1
```

INPUT 1

```
0 0
2
3 4 1
-1 2 2
```

OUTPUT 1

```
(3, 4)
(1, 8)
displacement (1, 8)
return (-1, -8)
distance 8.1
```

INPUT 2

```
10 -3
1
-2 1 3
```

OUTPUT 2

```
(4, 0)
displacement (-6, 3)
return (6, -3)
distance 6.7
```

REFERENCE SOLUTION

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>

struct Vec2 {
    long long x = 0;
    long long y = 0;

    Vec2& operator+=(const Vec2& o) { x += o.x; y += o.y; return *this; }
    Vec2& operator-=(const Vec2& o) { x -= o.x; y -= o.y; return *this; }
    Vec2& operator*=(long long k) { x *= k; y *= k; return *this; }
    Vec2 operator-() const { return Vec2{-x, -y}; }
    double length() const { return std::sqrt(static_cast<double>(x * x + y * y)); }
};

Vec2 operator+(Vec2 a, const Vec2& b) { return a += b; }
Vec2 operator-(Vec2 a, const Vec2& b) { return a -= b; }
Vec2 operator*(Vec2 v, long long k) { return v *= k; }
Vec2 operator*(long long k, Vec2 v) { return v *= k; }   // the scalar on the left r

std::ostream& operator<<(std::ostream& os, const Vec2& v) {
    return os << '(' << v.x << ", " << v.y << ')';
}

int main() {
    Vec2 start;
    std::cin >> start.x >> start.y;
    int n;
    std::cin >> n;
    Vec2 pos = start;
    for (int i = 0; i < n; ++i) {
        Vec2 v;
        long long ticks;
        std::cin >> v.x >> v.y >> ticks;
        pos += ticks * v;
        std::cout << pos << '\n';
    }
    const Vec2 moved = pos - start;
    std::cout << "displacement " << moved << '\n';
    std::cout << "return " << -moved << '\n';
    std::cout << "distance " << std::fixed << std::setprecision(1) << moved.length()
    return 0;
}
```

QUIZ

1. What happens here?

```
struct V {
    int x;
    V operator*(int k) const { return V{x * k}; }
};
int main() {
    V v{2};
    V w = 3 * v;
}
```

A. `w.x` is 6

B. Compile error: no match for `operator*` with `int` and `V`

C. `w.x` is 3

D. Undefined behaviour

2. Which group of operators **must** be implemented as member functions?

A. `+`, `-`, `*`, `/`

B. `=`, `[]`, `()`, `→`

C. `<<` and `>>`

D. `==` and `⟺`

3. What does this print?

```
struct C {
    int n = 0;
    C& operator++() { ++n; return *this; }
    C operator++(int) { C old = *this; ++*this; return old; }
};
int main() {
    C c;
    C a = c++;
    C b = ++c;
    std::cout << a.n << b.n << c.n;
}
```

A. `022`

B. `122`

C. `012`

D. `011`

4. Why is `Vec2 operator+(Vec2 a, const Vec2& b) { return a += b; }` written with its first parameter by value?

A. So the function can modify the caller's object

B. The operator needs a copy to work on anyway, and a by-value parameter lets a temporary argument be moved straight into it

C. Operators cannot take reference parameters

D. To make the operator a member function

5. Which of these overloads compiles but breaks a guarantee readers rely on?

A. `operator&&` on a `Condition` type

B. `operator+=` returning `*this` by reference

C. `operator==` as a free function

D. Unary `operator-` as a `const` member

6. What is the outcome?

```
class M {
    long long c;
public:
    explicit M(long long v) : c(v) {}
    friend M operator+(M, const M&);
};
int main() {
    M a(1), b(2);
    M s = a + b;
}
```

A. Runs and exits 0

B. Compile error: `c` is private inside `operator+`

C. Linker error: `operator+` is declared but never defined

D. Compile error: a friend cannot be an operator

ANSWER KEY · OVERLOADING RULES — MEMBERS, FREE FUNCTIONS AND THE COMPOUND-FIRST IDIOM

1. B A member operator's left operand must be the class itself — `3` is an `int`, and there is no `int::operator*(V)`. Only a free `V operator*(int, V)` makes the scalar-on-the-left form work; `v * 3` would have compiled.

2. B Assignment, subscript, call and arrow are required to be members; `<<` on a stream is required to be a free function, and the arithmetic and comparison operators may be either but are usually free so conversions apply to both operands.

3. A Postfix returns the *old* value: `a` is a copy taken while `n` was 0. Prefix returns the updated object, so `b.n` is 2, and `c` itself has been incremented twice.

4. B `+` must not change its operands, so it works on a copy; making the parameter the copy avoids a second one and lets `(a + b) + c` move the intermediate. The caller's object is never touched.

5. A An overloaded `&&` is an ordinary function call: both operands are evaluated before it runs, so the short-circuit behaviour of the built-in `&&` is lost. The other three are the recommended shapes.

6. C The `friend` line only *declares* a free function; nothing defines it, so the call compiles and the linker reports an undefined reference. Defining it inside the class body (a hidden friend) is the usual fix.

LESSON 2 · 15 MIN

Comparison and the spaceship operator

A type that can be compared can be sorted, searched, deduplicated and used as a `std::map` key — most of the standard library opens up the moment `<` and `==` exist. Before C++20 that meant writing six operators by hand and keeping them consistent; since C++20 one line, `auto operator<=>(const T&) const = default;`, produces the whole set from the members in declaration order. This lesson settles what `==` and `<` must promise, how the *three-way comparison* `<=>` and the rewritten expressions work, which of `strong_ordering`, `weak_ordering` and `partial_ordering` a type should return, and what `std::sort` and `std::set` actually call.

Equality

```
struct Version {
    int major, minor, patch;
    bool operator==(const Version&) const = default;   // memberwise, in decl
};

Version a{1, 2, 3}, b{1, 2, 4};
a == b;   // false
a != b;   // true — rewritten by the compiler as !(a == b)
```

`operator==` returns `bool` and is a `const` member (or a free function with two `const&` parameters). Defaulting it compares each member with its own `==`, in the order they are declared, and stops at the first difference. C++20 also *rewrites*: `a != b` becomes `!(a == b)`, and when the operand types differ, `b == a` is tried as `a == b` reversed — so one `==` serves both directions and `!=` no longer needs writing. Equality must be an equivalence relation: reflexive, symmetric, transitive. A hand-written `==` that ignores a member (a cached hash, a timestamp) is legitimate as long as `<` ignores it too.

Ordering before C++20

Ordered containers and `std::sort` need `<` and nothing else, and `<` must be a **strict weak ordering**: `a < a` is false, `a < b` implies `!(b < a)`, and "neither is less" is transitive. The pre-C++20 way to write it for several members is `std::tie`, which compares lexicographically:

```cpp
bool operator<(const Version& a, const Version& b) {
    return std::tie(a.major, a.minor, a.patch) < std::tie(b.major, b.minor, b.
}
bool operator>(const Version& a, const Version& b)  { return b < a; }
bool operator⩽(const Version& a, const Version& b) { return !(b < a); }
bool operator⩾(const Version& a, const Version& b) { return !(a < b); }
```

Four functions derived from one, and `==` / `≠` separately: six operators for one idea. The classic bug is `⩽` written as the comparison itself (`return a.x ⩽ b.x;`) — not strict, so `std::sort` may read out of bounds and `std::set` cannot find what it just inserted.

The spaceship operator

`a ⟺ b` asks "less, equal or greater?" once and answers with a *comparison category* value that itself compares to `0`:

```cpp
#include <compare>

struct Version {
    int major, minor, patch;
    auto operator⟺(const Version&) const = default;   // also declares opera
};

Version a{1, 2, 10}, b{1, 2, 9};
a < b;          // rewritten as (a ⟺ b) < 0    → false
a ⩾ b;          // rewritten as (a ⟺ b) ⩾ 0    → true
a == b;         // the defaulted ==, NOT ⟺     → false
std::strong_ordering r = a ⟺ b;   // std::strong_ordering::greater
```

The compiler rewrites each of `<`, `>`, `⩽` and `⩾` as a call to `⟺` compared with the literal `0` (only `0` — `(a ⟺ b) < 1` is an error by design). A **defaulted** `⟺` compares members in declaration order and, as a courtesy, also declares a defaulted `operator==`, so one line yields all six. The order of the members is therefore the order of comparison: `major, minor, patch` sorts `1.2.9` before `1.2.10`, which a string comparison would not.

`==` is deliberately not derived from `⟺`. For a `std::string` the three-way comparison must scan the characters, while equality can reject two strings of different length instantly, so the language keeps them separate — and that has a consequence: if you write `⟺` **yourself**, you get the four relational operators and no `==`. Add `bool operator==(const T&) const = default;` (or your own) alongside, or `a == b` does not compile.

Three orderings

Category	Meaning	Examples	Converts to
`std::strong_ordering`	Equal values are substitutable	`int`, `std::string`, a defaulted `Version`	weak, partial
`std::weak_ordering`	Equivalent but distinguishable	Case-insensitive names, a struct with a member `⟺` ignores	partial
`std::partial_ordering`	Some pairs are `unordered`	`double` (NaN), a struct with a `double` member	—

`auto` as the return type of a defaulted `⟺` deduces the *weakest* category among the members, so adding one `double` field turns a `strong_ordering` type into a `partial_ordering` one — and `a < b` becomes `false` in both directions for a NaN. Declaring the return type as `std::strong_ordering` instead makes the defaulted operator *deleted* when a member cannot supply that category, so the first comparison fails to compile — which is the check you want. Stronger categories convert implicitly to weaker ones: a function returning `std::weak_ordering` may `return x.size() ⟺ y.size();` even though that expression is strong.

Writing `<=>` by hand

Default when the members in declaration order *are* the ordering. Otherwise compose it: compare the most significant key, return if decided, fall through to the next:

```
struct Date {
    int y, m, d;
    std::strong_ordering operator⟺(const Date& o) const {
        if (auto c = y ⟺ o.y; c ≠ 0) return c;
        if (auto c = m ⟺ o.m; c ≠ 0) return c;
        return d ⟺ o.d;
    }
    bool operator==(const Date&) const = default;   // must be declared: a use
};
```

A case-insensitive `Tag` compares lower-cased characters and returns `std::weak_ordering`, because `"Apple"` and `"apple"` are *equivalent* — neither is less — yet remain distinguishable objects; its `==` must agree, and `(*this <=> o) == 0` is the simplest way to guarantee that. Keep the two consistent: a `std::set` that finds an element by `<` and a `std::find` that finds it by `==` must agree about which elements are the same.

Sort, set and map

```
std::vector<Version> v = {{1, 2, 10}, {1, 2, 9}, {2, 0, 0}};
std::sort(v.begin(), v.end());        // uses operator<  → (v[i] <=> v[j]) < 0
std::set<Version> distinct(v.begin(), v.end());
std::map<Version, std::string> notes; // keys ordered by the same <
distinct.contains(Version{1, 2, 9});  // found by "neither is less", not by ==
```

Every ordered facility in the library — `std::sort`, `std::set`, `std::map`, `std::lower_bound`, `std::min` — is written in terms of `<` alone, through the default comparator `std::less<T>`. A defaulted `<=>` supplies that `<` and nothing more is needed. Two keys are the *same* key for a `std::set` when `!(a < b) && !(b < a)`, so a `Tag` with a case-insensitive ordering deduplicates spellings that differ only in case, keeping the one inserted first. When the natural order is not the one a particular container needs, pass a comparator instead of changing `<` (lesson 4).

Pitfalls

Mistake	What happens
`⩽` or `⩾` used as the sort order	Not strict: `std::sort` is undefined behaviour, `std::set` loses elements
Hand-written `⟺`, no `==`	`a == b` and `a != b` do not compile
Members declared in the wrong order, `⟺` defaulted	Sorts by the wrong key first
`double` member, `auto` return	`partial_ordering`; NaN compares neither less nor greater
`(a ⟺ b) < 1`	Compile error: an ordering compares only with the literal `0`
Forgetting `#include <compare>`	The category types are undeclared
`==` and `<` that disagree	`std::set` and `std::find` see different duplicates

Key takeaways

`operator==` is enough for `!=` (rewritten) and, when defaulted, compares members in declaration order.

`auto operator⟺(const T&) const = default;` gives `<`, `>`, `⩽`, `⩾` **and** `==`; a hand-written `⟺` gives the four relations only — add `==` yourself.

The result compares with `0`: `a < b` is `(a ⟺ b) < 0`.

`strong_ordering` for substitutable values, `weak_ordering` for equivalent-but-distinguishable, `partial_ordering` when some pairs are unordered (`double`).

`std::sort`, `std::set` and `std::map` use only `<`; a strict weak ordering is the contract, and "same key" means neither is less.

Compose a custom `⟺` key by key with `if (auto c = x ⟺ o.x; c != 0) return c;`.

EXERCISE 1

Sort the versions

Write a `Version` struct with `major`, `minor` and `patch` members **in that order** and a defaulted three-way comparison: `auto operator⟺(const Version&) const = default;` — nothing else.

Read `n` versions, sort them ascending with `std::sort`, print them, then put them in a `std::set<Version>` and report how many distinct versions there are and the oldest and newest.

Numeric order, not text order: `1.2.9` comes before `1.2.10`, and `1.10.0` before `2.0.0`.

Input: `n` (at least 1), then `n` lines each `major.minor.patch`. **Output:** the `n` versions sorted, one per line; then `distinct=<k>`, `oldest=<version>`, `newest=<version>`.

```
5
1.2.10
1.2.9
2.0.0
1.10.0
1.2.9
```

prints

```
1.2.9
1.2.9
1.2.10
1.10.0
2.0.0
distinct=4
oldest=1.2.9
newest=2.0.0
```

INPUT 1

```
5
1.2.10
1.2.9
2.0.0
1.10.0
1.2.9
```

OUTPUT 1

```
1.2.9
1.2.9
1.2.10
1.10.0
2.0.0
distinct=4
oldest=1.2.9
newest=2.0.0
```

INPUT 2

```
3
0.0.1
0.0.0
10.0.0
```

OUTPUT 2

```
0.0.0
0.0.1
10.0.0
distinct=3
oldest=0.0.0
newest=10.0.0
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <compare>
#include <iostream>
#include <set>
#include <vector>

struct Version {
    int major = 0;
    int minor = 0;
    int patch = 0;
    auto operator<=>(const Version&) const = default;   // <, >, <=, >= and == from
};

std::ostream& operator<<(std::ostream& os, const Version& v) {
    return os << v.major << '.' << v.minor << '.' << v.patch;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<Version> versions(n);
    for (Version& v : versions) {
        char dot;
        std::cin >> v.major >> dot >> v.minor >> dot >> v.patch;
    }
    std::sort(versions.begin(), versions.end());        // uses the rewritten operat
    for (const Version& v : versions) std::cout << v << '\n';
    const std::set<Version> distinct(versions.begin(), versions.end());
    std::cout << "distinct=" << distinct.size() << '\n';
    std::cout << "oldest=" << *distinct.begin() << '\n';
    std::cout << "newest=" << *distinct.rbegin() << '\n';
    return 0;
}
```

EXERCISE 2

Case-insensitive tags

Write a `Tag` class wrapping a `std::string` whose ordering ignores case: `"Apple"`, `"apple"` and `"APPLE"` are *equivalent* — neither is less than the other — but they remain different strings, so the three-way comparison must return `std::weak_ordering`, not `std::strong_ordering`. Write `operator<=>` by hand (compare the lower-cased characters one by one, then the lengths) and an `operator==` that agrees with it.

Read `n` tags into a `std::set<Tag>`: a spelling equivalent to one already present is not inserted, so the first spelling seen is the one kept. Print how many distinct tags there are and the tags in set

order. Then read two more words and print how the first compares to the second.

Input: `n`, then `n` words (one per line), then a line with two words. **Output:** `distinct=<k>`, the `k` tags in order, then `less`, `greater` or `equivalent`.

```
5
Apple
banana
APPLE
Cherry
Banana
apple Apple
```

prints

```
distinct=3
Apple
banana
Cherry
equivalent
```

INPUT 1

```
5
Apple
banana
APPLE
Cherry
Banana
apple Apple
```

OUTPUT 1

```
distinct=3
Apple
banana
Cherry
equivalent
```

INPUT 2

```
3
zeta
Alpha
alpha
Zeta beta
```

OUTPUT 2

```
distinct=2
Alpha
zeta
greater
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cctype>
#include <compare>
#include <iostream>
#include <set>
#include <string>
#include <utility>

class Tag {
public:
    explicit Tag(std::string text) : text_(std::move(text)) {}
    const std::string& text() const { return text_; }

    std::weak_ordering operator<=>(const Tag& o) const {
        const std::size_t n = std::min(text_.size(), o.text_.size());
        for (std::size_t i = 0; i < n; ++i) {
            const int a = std::tolower(static_cast<unsigned char>(text_[i]));
            const int b = std::tolower(static_cast<unsigned char>(o.text_[i]));
            if (a != b) return a < b ? std::weak_ordering::less : std::weak_ordering
        }
        return text_.size() <=> o.text_.size();   // a strong_ordering converts to w
    }
    bool operator==(const Tag& o) const { return (*this <=> o) == 0; }   // a hand-w

private:
    std::string text_;
};

int main() {
    int n;
    std::cin >> n;
    std::set<Tag> tags;
    for (int i = 0; i < n; ++i) {
        std::string word;
        std::cin >> word;
        tags.insert(Tag(word));   // an equivalent spelling is already there: insert
    }
    std::cout << "distinct=" << tags.size() << '\n';
    for (const Tag& t : tags) std::cout << t.text() << '\n';
    std::string left, right;
    std::cin >> left >> right;
    const auto order = Tag(left) <=> Tag(right);
    if (order < 0) std::cout << "less\n";
    else if (order > 0) std::cout << "greater\n";
    else std::cout << "equivalent\n";
    return 0;
}
```

QUIZ

1. Given `struct V { int a, b; auto operator⟺(const V&) const = default; };`, which comparisons compile for two `V` objects?

A. `<`, `>`, `⩽`, `⩾` only

B. All six: the four relational operators, `==` and `≠`

C. `⟺` only

D. `==` and `≠` only

2. A class has a hand-written `std::strong_ordering operator⟺(const D&) const` and no `operator==`. What does `a == b` do?

A. Returns `true` when `(a ⟺ b) == 0`

B. Compile error: `==` is not rewritten in terms of `⟺`

C. Compares the objects byte by byte

D. Calls an implicitly defaulted `==`

3. What is the type of `R{} ⟺ R{}` for `struct R { int id; double value; auto operator⟺(const R&) const = default; }`?

A. `std::strong_ordering`

B. `std::weak_ordering`

C. `std::partial_ordering`

D. It does not compile: `double` has no `⟺`

4. `std::sort` is run on a vector of `T` whose `operator<` is written as `return a.x <= b.x;`. What happens?

A. The vector is sorted and equal elements keep their order

B. Undefined behaviour: `<=` is not a strict weak ordering

C. Compile error

D. The vector is sorted in descending order

5. How does `std::set<T>` decide that a key being inserted is already present?

A. It calls `operator==`

B. It compares `std::hash` values

C. It checks that neither `a < b` nor `b < a` holds

D. It requires `a <=> b` to be `std::strong_ordering::equal`

6. What does this print?

```
struct V {
    int major, minor;
    auto operator<=>(const V&) const = default;
};
int main() {
    V a{1, 10}, b{1, 9};
    std::cout << (a < b) << (a > b) << (a == b);
}
```

A. `010`

B. `100`

C. `001`

D. `110`

7. Which comparison of a three-way result compiles?

A. `(a <=> b) < 0`

B. `(a <=> b) < 1`

C. `(a <=> b) == 0.0`

D. `(a <=> b) < -1`

ANSWER KEY · COMPARISON AND THE SPACESHIP OPERATOR

1. B A *defaulted* $\Longleftrightarrow$ also declares a defaulted `operator==`, and $\neq$ is rewritten from $=$. Only a *hand-written* $\Longleftrightarrow$ leaves $=$ undeclared.

2. B Equality is deliberately kept separate from ordering (for a string, $=$ can reject on length without scanning). A user-provided $\Longleftrightarrow$ yields the four relational operators only; declare `bool operator==(const D&) const = default;` alongside it.

3. C `auto` deduces the weakest category among the members. `int` gives `strong_ordering` but `double` gives `partial_ordering` because NaN is unordered, and the common category is the weaker one.

4. B `std::sort` requires irreflexivity (`a < a` must be false). With $\leqslant$, an element is 'less than' itself, and the implementation may loop past the end of the range or leave it unsorted; `std::set` with such a comparator cannot find elements it holds.

5. C Ordered containers are written in terms of `<` alone: two keys are equivalent when neither is less than the other. That is why a case-insensitive `<` deduplicates different spellings, and why $=$ and `<` must agree.

6. A The majors tie, so the minors decide: 10 is greater than 9. `a < b` is false, `a > b` is true, and the defaulted $=$ is false because a member differs.

7. A An ordering value may only be compared with the literal `0`; the library implements this with a type that only a literal zero converts to, so `1`, `-1` and `0.0` are compile errors by design.

LESSON 3 · 14 MIN

Stream operators — printing and reading your own types

`std::cout << total` works for `int` and `std::string` because the library wrote an `operator<<` for each; it works for your `Money` the moment you write one too, and then every stream — `std::cout`, a file, an `std::ostringstream` — can print it. The reverse, `std::cin >> m`, is the same idea with a harder contract, because input can be malformed and the stream has a state the reader is expected to check. This lesson settles the two signatures, why they are free functions, what "return the stream" and "respect the stream's state" mean in practice, how to read a structured value back safely, and where `std::format` fits.

---

operator<<

```
#include <iostream>

class Money {
public:
    explicit Money(long long cents) : cents_(cents) {}
    friend std::ostream& operator<<(std::ostream& os, const Money& m);
private:
    long long cents_;
};

std::ostream& operator<<(std::ostream& os, const Money& m) {
    long long c = m.cents_;
    if (c < 0) { os << '-'; c = -c; }
    os << c / 100 << '.';
    if (c % 100 < 10) os << '0';
    return os << c % 100;
}

Money price(1999);
std::cout << "price: " << price << '\n';   // price: 19.99
```

The left operand is the stream, a class you cannot add members to, so `operator<<` is necessarily a free function: `std::ostream&` in, `const T&` in, `std::ostream&` out. Returning the stream is what makes chaining work — `std::cout << a << b` is

`operator<<(operator<<(std::cout, a), b)`, and the inner call must yield something the outer one can insert into. It is a `friend` only because it reads `cents_`; with a public accessor it needs no friendship at all. Two habits matter. Insert into the `os` you were given, never into `std::cout`, or the operator silently misbehaves for files and string streams. And do not print a trailing newline: the caller decides how the value sits in its line.

---

operator>>

```
std::istream& operator>>(std::istream& is, Point& p) {
    char open, comma, close;
    int x, y;
    if (is >> open && open == '(' && is >> x >> comma && comma == ',' && is >>
        p = Point{x, y};                    // assign only once the whole valu
    } else {
        is.setstate(std::ios::failbit);     // leave the stream failed and the
    }
    return is;
}

Point p;
if (std::cin >> p) { /* p is valid */ } else { /* bad or exhausted input */ }
```

Input has a contract that output does not. The reader must read into *locals*, validate, and assign to the target only when the whole value is good, so a half-read `(3,` never leaves `p` with a new `x` and an old `y`. On a malformed token it sets `failbit` — the same signal the built-in `>>` gives when `int` meets `"abc"` — and returns the stream, so `if (std::cin >> p)` and `while (std::cin >> p)` work exactly as they do for `int`. Whitespace is skipped by each `>>` into a `char` or a number, which is why `(3, 4)` and `(3,4)` both parse; if a format must reject spaces, read with `is.get()` instead. For a token-shaped value, the simplest robust reader takes a `std::string` with `is >> tok` and parses it by hand — then the stream is already positioned after the token and the only decision left is whether to set `failbit`.

Respecting the stream's state

A stream carries formatting state — `width`, `fill`, `precision`, the `fixed` / `hex` / `left` flags — that the *caller* set and expects to keep. Two rules follow.

First, `std::setw` is one-shot: it applies to the next single insertion and then resets to 0. So `std::cout << std::setw(10) << price` widens only whatever the operator inserts first — the `'-'` or the units — and the rest of the value follows unpadded. If a type should honour `setw`, the operator must build the whole text and insert it in **one** operation:

```
#include <sstream>
#include <iomanip>

std::ostream& operator<<(std::ostream& os, const Money& m) {
    std::ostringstream out;                                  // a private stre
    long long c = m.cents();
    if (c < 0) { out << '-'; c = -c; }
    out << c / 100 << '.' << std::setw(2) << std::setfill('0') << c % 100;
    return os << out.str();                                  // one insertion.
}

std::cout << '[' << std::setw(10) << Money(-1999) << "]\n";   // [    -19.99]
```

Second, sticky state must be put back. `std::setfill('0')` and `std::setprecision(2)` stay in force after the operator returns, and a caller who then prints an `int` gets it zero-padded. Either apply such manipulators to a temporary `std::ostringstream`, as above, or save and restore: `const auto fill = os.fill('0'); … os.fill(fill);`, and `os.flags(saved)` for the flag set. An operator that changes the caller's stream is a bug that surfaces three print statements later.

Reading a value back

Print and read should be inverses: whatever `<<` writes, `>>` accepts. For `Money` that means `>>` parses `19.99`, `-0.05` and `7` (no fraction), and pads a single decimal — `7.5` is 750 cents — while rejecting `1.2.3` and `12.345` with `failbit`. Writing the two together, and testing them by printing a value, reading it into a second object and comparing, is the discipline that keeps a serialised format honest.

```
std::istream& operator>>(std::istream& is, Money& m) {
    std::string tok;
    if (!(is >> tok)) return is;                    // EOF or an earlier failu
    long long cents = 0;
    if (!parseCents(tok, cents)) { is.setstate(std::ios::failbit); return is;
    m = Money(cents);
    return is;
}
```

std::format and a formatter specialisation

`std::format("{:>10}", price)` does not compile until `std::formatter<Money>` exists. The shortest correct specialisation inherits the string formatter, which already understands width, alignment and fill, and hands it the text:

```
#include <format>

template <>
struct std::formatter<Money> : std::formatter<std::string> {
    auto format(const Money& m, std::format_context& ctx) const {
        return std::formatter<std::string>::format(toString(m), ctx);    // to
    }
};

std::cout << std::format("[{:>10}]", Money(1999)) << '\n';   // [     19.99]
```

This compiles on the C++20 judge, but treat it as reading: `std::format` has no input counterpart, so a type that must round-trip still needs `>>`, and `<<` remains what `std::ostringstream`, files and every logging library call.

Pitfalls

Mistake	What happens
`operator<<` as a member	The left operand would have to be your type: `m << std::cout`
Returning `void` from `<<`	`std::cout << m << '\n'` does not compile
Writing to `std::cout` inside the operator	Wrong output for files and string streams
Assigning to the target before validation in `>>`	A failed read leaves a half-updated object
Not setting `failbit` on bad input	The caller's `if (is >> x)` says success for garbage
`std::setfill('0')` applied to the caller's stream	Every later number is zero-padded
Printing `'\n'` inside `<<`	The caller cannot put the value mid-line
`-` from `c / 100` when `-5 < c < 0`	`-5 / 100` is `0`: the sign is lost unless printed explicitly

Key takeaways

`std::ostream& operator<<(std::ostream&, const T&)` and `std::istream& operator>>(std::istream&, T&)` are free functions (friends when they need private members) that return the stream for chaining.

Insert into the stream you were given, never `std::cout`, and never a trailing newline.

In `>>`, read into locals, validate, assign on success only, and `setstate(failbit)` otherwise so `if (is >> x)` keeps its meaning.

`setw` is one-shot: build the text in an `std::ostringstream` and insert it once so callers can pad your type; restore any sticky fill or precision you change.

Make `<<` and `>>` inverses and test the round trip.

`std::formatter<T>` inheriting `std::formatter<std::string>` enables `std::format` with width and alignment — useful, but not a replacement for `>>`.

EXERCISE 1

Money in and out

Give the `Money` class (an integer number of cents) an `operator>>` and an `operator<<` that are exact inverses.

`>>` reads one whitespace-delimited token and accepts an optional `-`, one or more digits, and optionally a `.` followed by **one or two** digits: `12.34`, `7`, `0.5` (fifty cents) and `-0.05` are valid; `7.`, `1.2.3`, `12.345` and `abc` are not. On a bad token set `failbit` and leave the target unchanged. `<<` prints the sign, the units, a point and exactly two digits — and it must build the text first and insert it into the stream **once**, so that a caller's `std::setw` pads the whole amount.

Input: `n`, then `n` amounts (one per line). **Output:** each amount echoed through `<<` on its own line; then `total` followed by the sum printed with `std::setw(10)`. If an amount is bad, print `bad amount` instead and stop.

```
4
12.34
7
0.5
-0.05
```

prints

```
12.34
7.00
0.50
-0.05
total      19.79
```

INPUT 1

```
4
12.34
7
0.5
-0.05
```

OUTPUT 1

```
12.34
7.00
0.50
-0.05
total      19.79
```

INPUT 2

```
3
-3.5
1.25
2.25
```

OUTPUT 2

```
-3.50
1.25
2.25
total       0.00
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>

class Money {
public:
    Money() = default;
    explicit Money(long long cents) : cents_(cents) {}
    long long cents() const { return cents_; }
    Money& operator+=(const Money& o) { cents_ += o.cents_; return *this; }
private:
    long long cents_ = 0;
};

std::ostream& operator<<(std::ostream& os, const Money& m) {
    std::ostringstream out;                 // a private stream: setfill here never
    long long c = m.cents();
    if (c < 0) {
        out << '-';                         // printed explicitly: -5 / 100 is 0 and
        c = -c;
    }
    out << c / 100 << '.' << std::setw(2) << std::setfill('0') << c % 100;
    return os << out.str();                 // one insertion, so the caller's setw p
}

std::istream& operator>>(std::istream& is, Money& m) {
    std::string tok;
    if (!(is >> tok)) return is;            // nothing to parse: the stream already
    std::size_t i = 0;
    const bool negative = tok[0] == '-';
    if (negative) i = 1;
    long long units = 0;
    int unitDigits = 0;
    while (i < tok.size() && std::isdigit(static_cast<unsigned char>(tok[i]))) {
        units = units * 10 + (tok[i] - '0');
        ++i;
        ++unitDigits;
    }
    long long cents = 0;
    int fractionDigits = 0;
    if (i < tok.size() && tok[i] == '.') {
        ++i;
        while (i < tok.size() && std::isdigit(static_cast<unsigned char>(tok[i])) &&
            cents = cents * 10 + (tok[i] - '0');
            ++i;
            ++fractionDigits;
        }
```

```
        if (fractionDigits == 0) {
            is.setstate(std::ios::failbit);
            return is;
        }
    }
    if (unitDigits == 0 || i != tok.size()) {   // no digits at all, or something le
        is.setstate(std::ios::failbit);
        return is;
    }
    if (fractionDigits == 1) cents *= 10;       // "0.5" is fifty cents
    const long long total = units * 100 + cents;
    m = Money(negative ? -total : total);       // assigned only now, when the whole
    return is;
}

int main() {
    int n;
    std::cin >> n;
    Money total;
    for (int i = 0; i < n; ++i) {
        Money m;
        if (!(std::cin >> m)) {
            std::cout << "bad amount\n";
            return 0;
        }
        std::cout << m << '\n';
        total += m;
    }
    std::cout << "total " << std::setw(10) << total << '\n';
    return 0;
}
```

EXERCISE 2

Points in parentheses

Write `operator<<` and `operator>>` for a `Point` of two integers. The printed form is `(x, y)`; the reader accepts `(x,y)` with any amount of whitespace around the numbers and punctuation — `(1,2)`, `(3, -4)` and `( -5 , 6 )` all parse. Read the pieces into **local** variables with `>>` into `char` and `int`, check each punctuation character, and assign to the point only when the whole value was read; otherwise set `failbit` and leave the point untouched.

Input: `n` (at least 1), then `n` points separated by whitespace in any layout. **Output:** each point printed as `(x, y)` on its own line as it is read; then `bbox (minx, miny) (maxx, maxy)` — the bounding box of all the points. If a point fails to parse, print `bad point` and stop.

```
3
(1,2) (3, -4)
( -5 , 6 )
```

prints

```
(1, 2)
(3, -4)
(-5, 6)
bbox (-5, -4) (3, 6)
```

INPUT 1

```
3
(1,2) (3, -4)
( -5 , 6 )
```

OUTPUT 1

```
(1, 2)
(3, -4)
(-5, 6)
bbox (-5, -4) (3, 6)
```

INPUT 2

```
2
(0,0)
[1,1]
```

OUTPUT 2

```
(0, 0)
bad point
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>

struct Point {
    int x = 0;
    int y = 0;
};

std::ostream& operator<<(std::ostream& os, const Point& p) {
    return os << '(' << p.x << ", " << p.y << ')';
}

std::istream& operator>>(std::istream& is, Point& p) {
    char open = 0, comma = 0, close = 0;
    int x = 0, y = 0;
    if (is >> open && open == '(' && is >> x >> comma && comma == ',' && is >> y >>
        p = Point{x, y};                    // only once the whole value has been re
    } else {
        is.setstate(std::ios::failbit);     // the stream reports the failure; p is
    }
    return is;
}

int main() {
    int n;
    std::cin >> n;
    Point lo, hi;
    for (int i = 0; i < n; ++i) {
        Point p;
        if (!(std::cin >> p)) {
            std::cout << "bad point\n";
            return 0;
        }
        std::cout << p << '\n';
        if (i == 0) {
            lo = p;
            hi = p;
        }
        lo.x = std::min(lo.x, p.x);
        lo.y = std::min(lo.y, p.y);
        hi.x = std::max(hi.x, p.x);
        hi.y = std::max(hi.y, p.y);
    }
    std::cout << "bbox " << lo << ' ' << hi << '\n';
    return 0;
}
```

QUIZ

1. Why is `operator<<` for a user-defined type written as a free function rather than a member?

A. Members cannot return references

B. The left operand is a `std::ostream`, a library class you cannot add members to

C. Free functions are faster

D. A member `<<` would be ambiguous with the bit-shift operator

2. What does this print?

```
struct P { int x, y; };
std::ostream& operator<<(std::ostream& os, const P& p) {
    os << p.x << ',' << p.y;
    return os;
}
int main() {
    std::cout << '[' << std::setw(6) << P{1, 2} << "]\n";
}
```

A. `[ 1,2]`

B. `[ 1,2]`

C. `[1,2 ]`

D. `[1,2]`

3. Inside `operator>>`, the token read does not match the expected format. What should the operator do?

A. Throw `std::invalid_argument`

B. Assign a default value to the target and return the stream

C. Set `failbit` on the stream, leave the target unchanged and return the stream

D. Print an error and call `std::exit`

4. Which `operator>>` mistake can leave the target half-updated?

A. Reading the pieces into locals and assigning at the end

B. Reading straight into `p.x` and `p.y` before checking the punctuation

C. Calling `is.setstate(std::ios::failbit)` on failure

D. Returning `is` at the end

5. An `operator<<` applies `std::setfill('0')` to the caller's stream and does not restore it. Afterwards the caller runs `std::cout << std::setw(5) << 42;`. What appears?

A. `42`

B. `00042`

C. `42`

D. A compile error

6. What does specialising `std::formatter<Money>` (inheriting `std::formatter<std::string>`) give you?

A. An `operator<<` for `Money`

B. An `operator>>` for `Money`

C. Support for `std::format("{:>10}", money)` with the string formatter's width and alignment handling

D. Nothing — `std::formatter` cannot be specialised for user types

ANSWER KEY · STREAM OPERATORS — PRINTING AND READING YOUR OWN TYPES

1. B In `std::cout << m` the left operand is the stream; a member operator would have to be `std::ostream::operator<<(const Money&)`, which you cannot add. As a member of `Money` it would read `m << std::cout`.

2. B `setw` is one-shot and applies to the *next single insertion*, which inside the operator is `p.x` : `1` is padded to width 6, then `,2` follows unpadded. Building the text in an `ostringstream` and inserting it once would pad the whole `1,2`.

3. C That is the contract the built-in `>>` follows, and it is what keeps `if (is >> x)` meaningful. Assigning a default would hide the failure; throwing would break every loop that expects a failed stream.

4. B If `p.x` is written before the comma check fails, the caller's object now has a new `x` and an old `y`. Locals first, one assignment at the end.

5. B Fill is sticky: it stays set until changed. Width applies once, so `42` is padded to 5 with the leaked `'0'`. Format into a private `std::ostringstream`, or save and restore with `os.fill()`.

6. C The specialisation is what `std::format` looks up for a `Money` argument; delegating to the string formatter makes the fill/align/width specs work on the text you produce. It has nothing to do with streams or input.

LESSON 4 · 15 MIN

Subscript, call and function objects

Two operators turn a class into something that *behaves like* a container or a function. `operator[]` makes `grid[r]` and `table["key"]` possible and is why `std::vector` and `std::map` read like arrays; `operator()` makes an object callable, and an object that can be called — a *function object*, or functor — is what `std::sort`, `std::priority_queue`, `std::map` and `std::unordered_map` accept wherever they need a comparison or a hash. This lesson settles the const/non-const pair every `operator[]` needs, bounds checking, `operator()` for both multi-index access and callable state, comparators as types, and the `std::hash` specialisation that lets your own struct be a hashed key.

---

operator[] — always two of them

```
#include <vector>
#include <stdexcept>

class Row {
public:
    explicit Row(std::size_t n) : data_(n) {}
    long long& operator[](std::size_t i) { return data_[i]; }               //
    const long long& operator[](std::size_t i) const { return data_[i]; }   //
    long long& at(std::size_t i) {
        if (i ⩾ data_.size()) throw std::out_of_range("Row::at");
        return data_[i];
    }
private:
    std::vector<long long> data_;
};

Row r(3);
r[0] = 7;                       // non-const overload: returns a reference, so
const Row& view = r;
long long v = view[0];          // const overload
```

`operator[]` must be a member and takes exactly one parameter in C++20. It returns a **reference** to the element — by value, `r[0] = 7` would assign to a temporary and change nothing — and it comes in two overloads, because a `const Row&` parameter (the way every function should receive a container it only reads) can call only `const` member functions.

Forget the const overload and `sum(const Row&)` fails to compile at the first `r[i]`. The non-const one returns `T&`, the const one `const T&`; the compiler picks by the constness of the object. Follow `std::vector`: `[]` does not check the index (undefined behaviour out of range, Module 6), and a checked `at()` throws `std::out_of_range`. A negative literal index converts to a huge `std::size_t` and hits the check, which is one reason to prefer `at()` while developing.

Two dimensions: operator() and a row view

Until C++23, `operator[]` takes one argument, so `m[r, c]` is not available on this track's C++20 runtime — the comma there is the comma operator, evaluating to `c`. The idiom for a matrix is `operator()(r, c)`, which takes any number of parameters:

```
#include <span>

class Matrix {
public:
    Matrix(std::size_t rows, std::size_t cols) : rows_(rows), cols_(cols), dat
    long long& operator()(std::size_t r, std::size_t c) { return data_[r * col
    const long long& operator()(std::size_t r, std::size_t c) const { return d
    std::span<long long> operator[](std::size_t r) { return {data_.data() + r
    std::span<const long long> operator[](std::size_t r) const { return {data_
private:
    std::size_t rows_, cols_;
    std::vector<long long> data_;   // one contiguous block, row-major
};

Matrix m(2, 3);
m(1, 2) = 5;                        // element access
for (long long x : m[1]) { /* the second row, as a view */ }
```

The elements live in one `std::vector` (row-major, `r * cols + c`), which is faster and simpler than a vector of vectors. `operator[](r)` returns a `std::span` over that row — a non-owning view (Module 16) — so `m[1][2]` works through the span's own `[]`, and range-for over a row needs no copy. Both operators again come in const and non-const pairs.

---

operator() and function objects

```
class RunningMean {
public:
    double operator()(double x) {           // callable: mean(x)
        sum_ += x;
        return sum_ / ++count_;
    }
private:
    double sum_ = 0;
    int count_ = 0;
};

RunningMean mean;
mean(10.0);          // 10
mean(20.0);          // 15
```

Any class with an `operator()` is a *function object*: it is called with the usual syntax and it carries state between calls, which a plain function cannot. A lambda (Module 4, lesson 5) is exactly this — the compiler generates a class whose captures are members and whose body is `operator()` — which is why a lambda and a hand-written functor are interchangeable in every algorithm. Write the class when the callable needs a name, several members, or a type you can spell in a template argument; write the lambda otherwise.

Functors as comparators

```
#include <algorithm>
#include <queue>
#include <set>

struct ByPriority {
    bool operator()(const Task& a, const Task& b) const {          // "a is wo
        if (a.priority != b.priority) return a.priority < b.priority;
        return a.seq > b.seq;                                       // earlier
    }
};

std::priority_queue<Task, std::vector<Task>, ByPriority> queue;   // top() is
std::sort(tasks.begin(), tasks.end(), ByPriority{});               // ascendi
std::set<Task, ByPriority> ordered;                                // ordered
std::map<std::string, int, std::greater<>> byNameDesc;             // a libra
```

A comparator is a callable returning `true` when its first argument sorts *before* its second — the strict weak ordering from lesson 2. `std::sort` takes it as a value; the containers take it as a **type** parameter, which is why a comparator that lives in a template argument must be a class (a lambda's type has no name you can write, short of `decltype`). Two rules: the ordering must be strict (`<`, never `<=`), and `operator()` must be `const` — `std::set` calls it on a const comparator and libstdc++ stops compilation with "comparison object must be invocable as const" otherwise. `std::priority_queue` inverts the intuition: with `std::less` the *largest* element is on top, so `std::greater<T>` gives a min-heap, and a custom comparator should say "a is lower priority than b". Ties need a deterministic tie-breaker (a sequence number) whenever output order matters.

---

std::hash for your own key

`std::unordered_map<Point, int>` needs two things a plain struct lacks: an `operator==` and a hash. The hash is a specialisation of `std::hash` — a functor type with a `const noexcept` call operator taking the key and returning `std::size_t`:

```
#include <functional>
#include <unordered_map>

struct Point {
    int x, y;
    bool operator==(const Point&) const = default;
};

template <>
struct std::hash<Point> {
    std::size_t operator()(const Point& p) const noexcept {
        const std::size_t h1 = std::hash<int>{}(p.x);
        const std::size_t h2 = std::hash<int>{}(p.y);
        return h1 ^ (h2 + 0x9e3779b9 + (h1 << 6) + (h1 >> 2));   // combine, r
    }
};

std::unordered_map<Point, int> visits;
++visits[Point{1, 2}];
```

Combine member hashes rather than XOR-ing them: `h1 ^ h2` makes `(1, 2)` and `(2, 1)` collide, and any point with `x == y` hashes to zero. The two members must agree — equal keys hash equal — and both must ignore the same fields. An alternative to specialising is

passing a hasher type as the third template argument, `std::unordered_map<Point, int, PointHash>`, which is the choice when the key type is not yours to specialise for. Whatever the hash, iteration order of an unordered container is unspecified and differs between libraries: copy the entries into a `std::vector` and `std::sort` them before printing anything a test compares. `std::string`, `std::string_view`, the integers, pointers, `std::optional` and `std::variant` already have a `std::hash`; `std::pair` and `std::tuple` do not, which is the usual reason a `PointHash` exists.

Pitfalls

Mistake	What happens
Only a non-const `operator[]`	A `const T&` parameter cannot be indexed
`operator[]` returning by value	`r[i] = x` assigns to a temporary
`m[r, c]` in C++20	The comma operator: `m[c]`
Comparator `operator()` not `const`	Static assertion in `std::set` / `std::map`
Comparator using `⩽`	Not a strict weak ordering: UB in `std::sort`
Printing an `unordered_map` in iteration order	Unspecified order; sort first
`std::hash` that ignores a member `==` uses	Equal keys in different buckets: lookups miss
`h1 ^ h2` as the combined hash	Symmetric collisions; `(a, a)` always hashes to 0

Key takeaways

`operator[]` is a member, returns a reference and needs both a const and a non-const overload; `[]` is unchecked, `at()` throws.

Use `operator()(r, c)` for two-dimensional access in C++20 and a `std::span` row view for `m[r][c]`.

A class with `operator()` is a function object; a lambda is one the compiler wrote.

Comparators are strict-weak-ordering functors with a `const` call operator; containers take them as a type, `std::priority_queue` puts the "largest" on top.

`template <> struct std::hash<Key>` plus `operator==` makes a struct an `unordered_map` key; combine member hashes, keep `==` and hash consistent, and sort before printing.

EXERCISE 1

A matrix with (r, c) and [r]

Complete the `Matrix` class, which stores its elements in **one** `std::vector<long long>` in row-major order. Provide `operator()(r, c)` in a const and a non-const overload, each checking the indices and throwing `std::out_of_range` when either is out of bounds; and `operator[](r)` in a const and a non-const overload returning a `std::span` over row `r` (`std::span<const long long>` from the const one). The free function `transposed(const Matrix&)` must build the transpose through the const `operator()`, and `rowSum` must take a `std::span<const long long>`.

Input: `R C`, then `R` rows of `C` integers, then `q`, then `q` queries `r c`. **Output:** the line `transpose`, the `C` rows of the transpose (space-separated), the line `rowsums` followed by the sum of each original row, then for each query the element `m(r, c)` or `out of range`.

```
2 3
1 2 3
4 5 6
3
0 2
1 0
2 0
```

prints

```
transpose
1 4
2 5
3 6
rowsums 6 15
3
4
out of range
```

INPUT 1

```
2 3
1 2 3
4 5 6
3
0 2
1 0
2 0
```

OUTPUT 1

```
transpose
1 4
2 5
3 6
rowsums 6 15
3
4
out of range
```

INPUT 2

```
1 1
7
1
0 0
```

OUTPUT 2

```
transpose
7
rowsums 7
7
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <span>
#include <stdexcept>
#include <vector>

class Matrix {
public:
    Matrix(std::size_t rows, std::size_t cols) : rows_(rows), cols_(cols), data_(row

    std::size_t rows() const { return rows_; }
    std::size_t cols() const { return cols_; }

    long long& operator()(std::size_t r, std::size_t c) { return data_[index(r, c)];
    const long long& operator()(std::size_t r, std::size_t c) const { return data_[i

    std::span<long long> operator[](std::size_t r) { return {data_.data() + r * cols
    std::span<const long long> operator[](std::size_t r) const { return {data_.data(

private:
    std::size_t rows_, cols_;
    std::vector<long long> data_;   // row-major: element (r, c) is at r * cols_ + c

    std::size_t index(std::size_t r, std::size_t c) const {
        if (r >= rows_ || c >= cols_) throw std::out_of_range("Matrix index");
        return r * cols_ + c;
    }
};

Matrix transposed(const Matrix& m) {
    Matrix t(m.cols(), m.rows());
    for (std::size_t r = 0; r < m.rows(); ++r)
        for (std::size_t c = 0; c < m.cols(); ++c)
            t(c, r) = m(r, c);          // const operator() on m, non-const on t
    return t;
}

long long rowSum(std::span<const long long> row) {
    long long s = 0;
    for (long long x : row) s += x;
    return s;
}

int main() {
    std::size_t rows, cols;
    std::cin >> rows >> cols;
    Matrix m(rows, cols);
    for (std::size_t r = 0; r < rows; ++r)
        for (std::size_t c = 0; c < cols; ++c)
            std::cin >> m(r, c);
```

```
    const Matrix t = transposed(m);
    std::cout << "transpose\n";
    for (std::size_t r = 0; r < t.rows(); ++r) {
        for (std::size_t c = 0; c < t.cols(); ++c) std::cout << (c == 0 ? "" : " ")
        std::cout << '\n';
    }
    std::cout << "rowsums";
    for (std::size_t r = 0; r < rows; ++r) std::cout << ' ' << rowSum(m[r]);   // sp
    std::cout << '\n';
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        long long r, c;
        std::cin >> r >> c;
        try {
            std::cout << m(static_cast<std::size_t>(r), static_cast<std::size_t>(c))
        } catch (const std::out_of_range&) {
            std::cout << "out of range\n";
        }
    }
    return 0;
}
```

EXERCISE 2

A task queue driven by a functor

Write a comparator **class** `LowerPriorityFirst` with a `const` `operator()(const Task&, const Task&)` and use it as the third template argument of a `std::priority_queue<Task, std::vector<Task>, LowerPriorityFirst>`. The task with the **highest** priority number must come out first; among equal priorities the one added **earliest** comes out first — stamp each task with a sequence number when it is added and use it as the tie-breaker. Remember that a priority queue's comparator says which element is *worse*: it returns `true` when its first argument should come out after its second.

Input: `n`, then `n` commands: `add <priority> <name>` or `run`. **Output:** for each `run`, `run <name>` for the task taken from the top, or `idle` when the queue is empty; finally `pending <count>`.

```
6
add 2 backup
add 5 deploy
add 5 hotfix
run
run
run
```

prints

```
run deploy
run hotfix
run backup
pending 0
```

INPUT 1

```
6
add 2 backup
add 5 deploy
add 5 hotfix
run
run
run
```

OUTPUT 1

```
run deploy
run hotfix
run backup
pending 0
```

INPUT 2

```
4
run
add 1 a
add 3 b
run
```

OUTPUT 2

```
idle
run b
pending 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <vector>

struct Task {
    int priority = 0;
    std::string name;
    int seq = 0;   // arrival order: the tie-breaker
};

struct LowerPriorityFirst {
    // true when a should come out AFTER b: a lower priority, or a later arrival at
    bool operator()(const Task& a, const Task& b) const {
        if (a.priority != b.priority) return a.priority < b.priority;
        return a.seq > b.seq;
    }
};

int main() {
    int n;
    std::cin >> n;
    std::priority_queue<Task, std::vector<Task>, LowerPriorityFirst> queue;
    int seq = 0;
    for (int i = 0; i < n; ++i) {
        std::string cmd;
        std::cin >> cmd;
        if (cmd == "add") {
            Task t;
            std::cin >> t.priority >> t.name;
            t.seq = seq++;
            queue.push(t);
        } else if (cmd == "run") {
            if (queue.empty()) {
                std::cout << "idle\n";
            } else {
                std::cout << "run " << queue.top().name << '\n';
                queue.pop();
            }
        }
    }
    std::cout << "pending " << queue.size() << '\n';
    return 0;
}
```

QUIZ

1. What happens?

```
class Row {
    std::vector<int> d{1, 2, 3};
public:
    int& operator[](std::size_t i) { return d[i]; }
};
int sum(const Row& r) { return r[0] + r[1] + r[2]; }
```

A. Returns 6

B. Compile error: `r` is const and `operator[]` is not a const member function

C. Undefined behaviour

D. Returns 0

2. In C++20, what does `m[1, 2]` mean for a `Matrix m` whose `operator[]` takes one `std::size_t`?

A. Element at row 1, column 2

B. The comma operator: `1` is evaluated and discarded, so it is `m[2]`

C. Compile error: too many arguments

D. A call to `operator()(1, 2)`

3. A comparator struct's `operator()` is not marked `const`. It is used as `std::set<Task, ByPriority>`. What does libstdc++ do?

A. Compiles and works

B. Compiles but the set is unordered

C. Fails to compile: a static assertion says the comparison object must be invocable as const

D. Compiles with a warning

4. What does `top()` return for `std::priority_queue<int, std::vector<int>, std::greater<int>>`?

A. The largest element

B. The smallest element

C. The first element pushed

D. The most recently pushed element

5. Why is `return std::hash<int>{}(p.x) ^ std::hash<int>{}(p.y);` a poor hash for a `Point`?

A. It is not `noexcept`

B. XOR is symmetric, so `(1, 2)` and `(2, 1)` collide and every point with `x == y` hashes to 0

C. `std::hash<int>` cannot be called on a temporary

D. It returns an `int` rather than a `std::size_t`

6. A program prints the contents of a `std::unordered_map` by iterating over it. What must a judged solution do instead?

A. Nothing; the order is insertion order

B. Use `std::map` in the loop only

C. Copy the entries into a `std::vector` and sort them before printing

D. Call `rehash(0)` first

7. What is a lambda expression, in terms of the mechanisms in this lesson?

A. A function pointer

B. A compiler-generated class whose captures are members and whose body is `operator()`

C. A `std::function`

D. A template function

ANSWER KEY · SUBSCRIPT, CALL AND FUNCTION OBJECTS

1. B A `const Row&` can only call `const` member functions. Every `operator[]` needs a second overload, `const int& operator[](std::size_t) const`, or read-only callers cannot index.

2. B A multi-argument `operator[]` arrives in C++23. In C++20 the expression inside the brackets is the comma operator (compilers warn about it), which yields its right operand. Use `operator()(r, c)` for two-dimensional access.

3. C The container stores the comparator and calls it through a const object. `std::sort` would still accept the functor (it takes it by value), which is why the bug shows up only when the same comparator moves into a container.

4. B A priority queue puts on top the element that the comparator says nothing is 'less' than — the maximum by its comparator. With `std::greater` the ordering is reversed, so the smallest element is on top: a min-heap.

5. B A plain XOR loses the order of the members and cancels equal ones. Mixing with a shift and a constant (`h1 ^ (h2 + 0x9e3779b9 + (h1 << 6) + (h1 >> 2))`) spreads the combinations out.

6. C Iteration order of an unordered container is unspecified and differs between standard libraries and bucket counts. Sorting the entries is what makes the output deterministic.

7. B The closure type is an unnamed class with a call operator; that is why lambdas and hand-written functors are interchangeable as comparators and why a lambda with captures can carry state.

LESSON 5 · 14 MIN

Conversion operators and explicit

C++ converts between types on its own more often than most languages: `int` to `double`, `const char*` to `std::string`, `3` to `Fraction` if `Fraction` has a constructor that takes one integer. Every one of those is a *user-defined* or built-in implicit conversion, and each is a decision someone made — sometimes without noticing. This lesson settles the two mechanisms a class controls, the converting constructor and the conversion operator, what `explicit` switches off, why `explicit operator bool` is the form every modern type uses, which surprising conversions the standard library itself contains, and how user-defined literals let a value be spelt as `2.5_km`.

Converting constructors

```
class Fraction {
public:
    Fraction(long long n, long long d = 1);    // callable with ONE argument: c
};

Fraction half(1, 2);
Fraction three = 3;          // copy-initialisation: Fraction(3)
Fraction sum = half + 3;     // if operator+ is a free function, the 3 convert
```

Any constructor that can be called with a single argument — one parameter, or several with defaults — is a **converting constructor**: the compiler may call it silently wherever a `Fraction` is expected and a `long long` was supplied. That is exactly what makes `half + 3` work with one free `operator+(const Fraction&, const Fraction&)` (lesson 1), and it is exactly what allows `void pay(Fraction); pay(3);` to compile when the author of `pay` never meant an integer to be accepted.

`explicit` keeps the constructor and removes the silence:

```
class Metres {
public:
    explicit Metres(double v) : v_(v) {}
    double value() const { return v_; }
private:
    double v_;
};

Metres a(5.0);                       // direct-initialisation: fine
Metres b{5.0};                       // fine
Metres c = static_cast<Metres>(5.0); // fine, and visibly a conversion
Metres d = 5.0;                      // error: copy-initialisation needs an i
void run(Metres); run(5.0);          // error: no implicit conversion from dou
```

The rule is the one the standard library follows: a single-argument constructor is `explicit` unless the conversion is lossless, obvious and intended. `std::string` converts from `const char*` implicitly because a string literal *is* a string; `std::vector<int>(5)` is explicit because `std::vector<int> v = 5;` would read as "a vector containing 5", which it is not. `Fraction(long long)` can reasonably stay implicit — an integer is a fraction. `Metres(double)` cannot: a bare `5.0` has no unit.

Conversion operators

The other direction — from your type *to* another — is a member function named after the target type, with no parameters and no return type:

```
class Fraction {
public:
    operator double() const { return static_cast<double>(num_) / den_; }   //
};

Fraction f(1, 3);
double d = f;                // 0.333…
std::cout << f << '\n';      // no operator<< for Fraction? prints 0.333333 thr
Fraction g(1, 6);
auto s = f + g;              // no operator+ for Fraction? both convert: s is d
```

Each of the last three lines is a silent decision. `std::cout << f` printing a decimal when you forgot `operator<<`, and `f + g` becoming floating-point arithmetic when you forgot `operator+`, are the reasons an implicit conversion operator is almost always wrong: it turns

missing overloads into wrong answers instead of compile errors. `explicit operator double() const` keeps the conversion available on request — `static_cast<double>(f)`, `double d(f)` — and refuses everything else.

---

explicit operator bool

```
class Result {
public:
    explicit operator bool() const { return ok_; }
    long long value() const { return value_; }
private:
    bool ok_ = false;
    long long value_ = 0;
};

Result r = parse("42");
if (r) { use(r.value()); }          // contextual conversion: allowed
if (!r) { report(); }               // allowed
bool b = r;                         // error
int n = r + 1;                      // error — the whole point
```

A plain `operator bool()` is the worst implicit conversion of all, because `bool` promotes to `int`: with it, `r + 1`, `r == other`, `std::cout << r` and `int n = r;` all compile and all mean nothing. `explicit operator bool` is honoured in exactly the places where "is it true?" is the question — the condition of `if`, `while`, `for` and `?:`, the operands of `!`, `&&` and `||`, and `static_cast<bool>` — and nowhere else. That is why `if (std::cin >> x)`, `if (ptr)` for a `std::unique_ptr`, and `if (opt)` for a `std::optional` all work while `int n = std::cin;` does not: every one of them is an `explicit operator bool`. Use the same form for anything that answers "did it succeed?"; readers already know the idiom.

Surprises in the standard library

Code	Result
`std::string s = "ab";`	Fine — `const char*` converts implicitly by design
`std::string s = 'a';`	Compile error: no constructor takes one `char`
`std::string s = 65;`	Compile error: no constructor takes one `int`
`std::string s; s = 65;`	Compiles: `operator=(char)` accepts the `int` — `s` is `"A"`
`std::string s{65};`	Compiles: the `initializer_list<char>` constructor — `"A"`
`std::string s('a', 3);`	Compiles: `string(size_type, char)` — 97 copies of the character `'\3'`
`std::string s(3, 'a');`	`"aaa"` — the argument order that was meant
`std::vector<int> v = 5;`	Compile error: `vector(size_type)` is `explicit`

The lesson of the table is not the individual rows but their shape: the conversions that compile are the ones with a plausible single-argument path (`int` to `char`), and the compiler never asks whether that was the intent. Brace-initialisation refuses *narrowing* conversions — `std::string s{300}` fails because 300 does not fit a `char` — which is one of the reasons Module 2 recommended braces. Two limits contain the damage. Only **one** user-defined conversion may appear in an implicit chain: with `Fraction(long long)` and `Money(Fraction)` , `Money m = 3;` is an error, since it would need two. And when two conversions match equally well — constructors from `int` and from `double` , called with a `long` — the call is ambiguous and refuses to compile: a diagnostic, not a guess.

User-defined literals

```
constexpr long double operator""_km(long double v) { return v * 1000.0L; }
constexpr unsigned long long operator""_kb(unsigned long long v) { return v *

using namespace std::string_literals;          // "text"s  → std::string
using namespace std::chrono_literals;          // 250ms, 2s → durations (Module

auto d = 2.5_km;                               // 2500.0 (long double)
auto size = 4_kb;                              // 4096
auto name = "ada"s;                            // std::string, not const char*
```

A literal operator is a function named `operator""` followed by a suffix, which for user code must begin with an underscore (suffixes without one are reserved for the standard library: `s`, `sv`, `ms`, `i`, `if`). The parameter is `unsigned long long` for an integer literal, `long double` for a floating one, or `const char*, std::size_t` for a string. They are ordinary functions, usually `constexpr`, and they earn their place when a unit is worth spelling at the point of use — `timeout(250ms)` reads better than `timeout(250)` and cannot be mistaken for seconds.

Pitfalls

Mistake	What happens
Single-argument constructor without `explicit`	Every function taking your type silently accepts the argument type
Implicit `operator double()`	Missing `operator+` / `operator<<` compile anyway and compute the wrong thing
`operator bool()` without `explicit`	`r + 1`, `r == s`, `int n = r;` all compile
`explicit` on a copy or move constructor	`T a = b;` and returning by value stop compiling
Expecting two user-defined conversions to chain	Only one is allowed; the call fails
Constructors from `int` and `double`, called with a `long`	Ambiguous call: neither conversion is better
A literal suffix without an underscore	Reserved: compile error or a conflict with the library

Key takeaways

A constructor callable with one argument is an implicit conversion from that argument's type; mark it `explicit` unless the conversion is obvious and lossless.

`operator T()` converts your type *to* `T`; make it `explicit` so that missing overloads are compile errors, not silent floating-point.

`explicit operator bool` is the standard idiom for "did it succeed?": it works in conditions and with `!`, `&&`, `||`, and nowhere else.

`s = 65` and `std::string('a', 3)` compile because an `int` converts to `char`; braces reject narrowing, `explicit` rejects the rest.

At most one user-defined conversion per implicit chain; two equally good candidates are an ambiguity error.

User-defined literals (`2.5_km`, `250ms`, `"text"s`) are `constexpr` functions named `operator""_suffix`.

EXERCISE 1

A result that tests like a bool

Write a `Result` class holding either a `long long` value or an error message, built through the static factories `Result::success(value)` and `Result::failure(message)`, with an `explicit operator bool` that is true for a success, plus `value()` and `error()` accessors. Then write `Result parseInt(std::string_view)` that accepts an optional `+` or `-` followed by one to eighteen digits and fails with `not a number` (any other character, or a sign with no digits) or `too long` (more than eighteen digits). Use the result the way the operator is meant to be used: `if (const Result r = parseInt(tok))`.

Input: whitespace-separated tokens until end of input. **Output:** one line per token — `<token> → <value>` or `<token> → error: <message>` — then `sum=<sum of the parsed values>` and `failed=<number of failures>`.

```
12 -7 abc 3x 0
```

prints

```
12 → 12
-7 → -7
abc → error: not a number
3x → error: not a number
0 → 0
sum=5
failed=2
```

INPUT 1

```
12 -7 abc 3x 0
```

OUTPUT 1

```
12 → 12
-7 → -7
abc → error: not a number
3x → error: not a number
0 → 0
sum=5
failed=2
```

INPUT 2

```
+5 -
```

OUTPUT 2

```
+5 → 5
- → error: not a number
sum=5
failed=1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <string_view>
#include <utility>

class Result {
public:
    static Result success(long long value) { Result r; r.ok_ = true; r.value_ = valu
    static Result failure(std::string message) { Result r; r.error_ = std::move(mess

    explicit operator bool() const { return ok_; }   // usable in if/while/!/&&, ref
    long long value() const { return value_; }
    const std::string& error() const { return error_; }

private:
    Result() = default;
    bool ok_ = false;
    long long value_ = 0;
    std::string error_;
};

Result parseInt(std::string_view text) {
    std::size_t i = 0;
    bool negative = false;
    if (!text.empty() && (text[0] == '-' || text[0] == '+')) {
        negative = text[0] == '-';
        i = 1;
    }
    if (i == text.size()) return Result::failure("not a number");
    if (text.size() - i > 18) return Result::failure("too long");
    long long value = 0;
    for (; i < text.size(); ++i) {
        if (text[i] < '0' || text[i] > '9') return Result::failure("not a number");
        value = value * 10 + (text[i] - '0');
    }
    return Result::success(negative ? -value : value);
}

int main() {
    std::string token;
    long long sum = 0;
    int failed = 0;
    while (std::cin >> token) {
        if (const Result r = parseInt(token)) {      // contextual conversion: the e
            std::cout << token << " -> " << r.value() << '\n';
            sum += r.value();
        } else {
            std::cout << token << " -> error: " << r.error() << '\n';
            ++failed;
```

```
        }
    }
    std::cout << "sum=" << sum << '\n';
    std::cout << "failed=" << failed << '\n';
    return 0;
}
```

EXERCISE 2

Celsius and Fahrenheit, explicitly

Write two classes, `Celsius` and `Fahrenheit`, each wrapping a `double` behind an `explicit` constructor and a `degrees()` accessor, and each with an **explicit conversion operator** to the other: `explicit operator Fahrenheit() const` on `Celsius` (F = C × 9/5 + 32) and `explicit operator Celsius() const` on `Fahrenheit` (C = (F − 32) × 5/9). Because both are explicit, `Celsius c = 36.6;` and `Fahrenheit f = c;` must not compile; conversions are requested with `static_cast`. Declare `Fahrenheit` before `Celsius` defines its operator body, since each class names the other.

Input: `n`, then `n` lines of `<value> <unit>` where the unit is `C` or `F`. **Output:** one line each, with one decimal place: `<v>C = <f>F` for a Celsius reading, `<v>F = <c>C` for a Fahrenheit one, or `bad unit` for anything else.

```
3
100 C
98.6 F
-40 C
```

prints

```
100.0C = 212.0F
98.6F = 37.0C
-40.0C = -40.0F
```

INPUT 1

```
3
100 C
98.6 F
-40 C
```

OUTPUT 1

```
100.0C = 212.0F
98.6F = 37.0C
-40.0C = -40.0F
```

INPUT 2

```
2
0 C
32 F
```

OUTPUT 2

```
0.0C = 32.0F
32.0F = 0.0C
```

REFERENCE SOLUTION

```cpp
#include <iomanip>
#include <iostream>
#include <string>

class Fahrenheit;

class Celsius {
public:
    explicit Celsius(double degrees) : degrees_(degrees) {}
    double degrees() const { return degrees_; }
    explicit operator Fahrenheit() const;   // Fahrenheit is incomplete here: declar
private:
    double degrees_;
};

class Fahrenheit {
public:
    explicit Fahrenheit(double degrees) : degrees_(degrees) {}
    double degrees() const { return degrees_; }
    explicit operator Celsius() const { return Celsius((degrees_ - 32.0) * 5.0 / 9.0
private:
    double degrees_;
};

Celsius::operator Fahrenheit() const { return Fahrenheit(degrees_ * 9.0 / 5.0 + 32.0

int main() {
    std::cout << std::fixed << std::setprecision(1);
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        double value;
        std::string unit;
        std::cin >> value >> unit;
        if (unit == "C") {
            const Celsius c(value);
            const auto f = static_cast<Fahrenheit>(c);   // the explicit operator, r
            std::cout << c.degrees() << "C = " << f.degrees() << "F\n";
        } else if (unit == "F") {
            const Fahrenheit f(value);
            const auto c = static_cast<Celsius>(f);
            std::cout << f.degrees() << "F = " << c.degrees() << "C\n";
        } else {
            std::cout << "bad unit\n";
        }
    }
    return 0;
}
```

QUIZ

1. `struct M { explicit M(double v); };` Which line fails to compile?

A. `M a(2.5);`

B. `M b{2.5};`

C. `M c = 2.5;`

D. `M d = static_cast<M>(2.5);`

2. `class F { public: operator double() const; };` with no `operator+` defined. What does `F a, b; auto s = a + b;` do?

A. Compile error: no `operator+` for `F`

B. Compiles: both operands convert to `double` and `s` is a `double`

C. Compiles and `s` is an `F`

D. Undefined behaviour

3. Why should `operator bool` almost always be `explicit`?

A. Non-explicit conversion operators are deprecated

B. `bool` promotes to `int`, so a non-explicit one lets `r + 1`, `r == s` and `int n = r;` compile and mean nothing

C. `explicit` makes the conversion faster

D. Without `explicit`, `if (r)` does not compile

4. Which of these compiles?

A. `std::string s = 65;`

B. `std::string s = 'a';`

C. `std::string s; s = 65;`

D. `std::vector<int> v = 5;`

5. What does `std::string s('a', 3);` produce?

A. `"aaa"`

B. A string of 97 characters, each with character code 3

C. A compile error

D. `"a3"`

6. `struct R { explicit operator bool() const; }; R r;` — which use compiles?

A. `if (r) { }`

B. `bool b = r;`

C. `int n = r;`

D. `bool ok() { return r; }`

7. A user-defined literal suffix in your own code must…

A. Be a single letter

B. Begin with an underscore

C. Be declared inside a class

D. Return `void`

ANSWER KEY · CONVERSION OPERATORS AND EXPLICIT

1. C Copy-initialisation with `=` needs an *implicit* conversion from `double` to `M`, which `explicit` forbids. Direct-initialisation, brace-initialisation and an explicit cast all name the constructor and are allowed.

2. B An implicit conversion operator lets the built-in `double + double` be chosen, so the missing overload is silently replaced by floating-point arithmetic. `explicit operator double()` turns this back into the compile error you want.

3. B The explicit form still works everywhere a condition is expected (`if`, `while`, `!`, `&&`, `||`, `?:`) — that is the contextual conversion — and refuses the arithmetic and assignment contexts where a bool was never meant.

4. C `std::string::operator=(char)` exists, and `65` converts to `char`, so `s` becomes `"A"`. No constructor takes a single `char` or `int` for copy-initialisation, and `vector(size_type)` is `explicit`.

5. B The matching constructor is `string(size_type count, char ch)`: `'a'` converts to the count 97 and `3` to the character `'\3'`. `std::string s(3, 'a')` is the intended call.

6. A Only contextual conversions honour an explicit `operator bool`: conditions and the operands of `!`, `&&`, `||`. Copy-initialising a `bool` — including returning `r` from a function that returns `bool` — needs `static_cast<bool>(r)`.

7. B Suffixes without a leading underscore are reserved for the standard library (`s`, `sv`, `ms`, `i`). `operator""_km(long double)` is the shape; the parameter is `long double` for floating literals and `unsigned long long` for integer ones.

CHECKPOINT · 25 MIN

Checkpoint — Operator overloading

This checkpoint covers the whole module: which operators to overload and where, the compound-first idiom, `==` and the defaulted `<=>` with its ordering categories, `<<` and `>>` as free functions that respect the stream's state, `operator[]` pairs, function objects as comparators and hashes, and `explicit` on constructors and conversions.

How it works. Thirteen questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why must `2 * v` be a free function rather than a member of `Vec2`?
- What does a defaulted `operator<=>` give you that a hand-written one does not?
- Which ordering category does a struct with a `double` member deduce, and why?
- Why does `std::setw(10) << money` need `operator<<` to insert one string?
- What goes wrong when a comparator's `operator()` is not `const`?
- Which contexts accept an `explicit operator bool`, and which refuse it?

The three programs are a `Polynomial` with arithmetic, evaluation through `operator()` and a formatted `<<`; a grid walk that counts visits through a `std::hash<Point>` specialisation and prints the cells sorted; and a `Duration` ledger that reads `h:mm:ss` through `>>`, orders with a defaulted `<=>` and prints through `<<` without leaking a fill character.

EXERCISE 1

Polynomial arithmetic

Write a `Polynomial` class over `long long` coefficients (stored lowest degree first, trailing zeros trimmed so the zero polynomial is empty) with `+=` and `*=` as members, `+` and `*` derived from them, `operator()(x)` that evaluates the polynomial at an integer `x`, and an `operator<<` that prints it in the usual form: highest degree first, zero terms skipped, a coefficient of 1 omitted except in the constant term, `x` for degree 1 and `x^k` above, terms joined by `+` or `-`, and a lone `0` for the zero polynomial. Examples: `3x^2 + 2x + 1`, `x - 1`, `-x^3 - 2x^2 - x`, `0`.

Input: two polynomials, each on a line as `d c0 c1 … cd` (degree, then the coefficients of `x^0` to `x^d`), then a line with an integer `x`. **Output:** seven lines: `p = …`, `q = …`, `p + q = …`, `p * q = …`,

`p(x) = …`, `q(x) = …`, `(p * q)(x) = …`, with the actual value of `x` in the parentheses.

```
2 1 2 3
1 -1 1
5
```

prints

```
p = 3x^2 + 2x + 1
q = x - 1
p + q = 3x^2 + 3x
p * q = 3x^3 - x^2 - x - 1
p(5) = 86
q(5) = 4
(p * q)(5) = 344
```

INPUT 1

```
2 1 2 3
1 -1 1
5
```

OUTPUT 1

```
p = 3x^2 + 2x + 1
q = x - 1
p + q = 3x^2 + 3x
p * q = 3x^3 - x^2 - x - 1
p(5) = 86
q(5) = 4
(p * q)(5) = 344
```

INPUT 2

```
0 5
1 0 2
3
```

OUTPUT 2

```
p = 5
q = 2x
p + q = 2x + 5
p * q = 10x
p(3) = 5
q(3) = 6
(p * q)(3) = 30
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <utility>
#include <vector>

class Polynomial {
public:
    Polynomial() = default;
    explicit Polynomial(std::vector<long long> coefficients) : c_(std::move(coeffici

    int degree() const { return static_cast<int>(c_.size()) - 1; }   // -1 for the z
    long long coefficient(int k) const {
        return k >= 0 && k <= degree() ? c_[static_cast<std::size_t>(k)] : 0;
    }

    long long operator()(long long x) const {          // Horner's rule
        long long result = 0;
        for (std::size_t i = c_.size(); i-- > 0;) result = result * x + c_[i];
        return result;
    }

    Polynomial& operator+=(const Polynomial& o) {
        if (o.c_.size() > c_.size()) c_.resize(o.c_.size(), 0);
        for (std::size_t i = 0; i < o.c_.size(); ++i) c_[i] += o.c_[i];
        trim();
        return *this;
    }
    Polynomial& operator*=(const Polynomial& o) {
        if (c_.empty() || o.c_.empty()) {
            c_.clear();
            return *this;
        }
        std::vector<long long> product(c_.size() + o.c_.size() - 1, 0);
        for (std::size_t i = 0; i < c_.size(); ++i)
            for (std::size_t j = 0; j < o.c_.size(); ++j)
                product[i + j] += c_[i] * o.c_[j];
        c_ = std::move(product);
        trim();
        return *this;
    }

private:
    std::vector<long long> c_;   // c_[k] is the coefficient of x^k
    void trim() { while (!c_.empty() && c_.back() == 0) c_.pop_back(); }
};

Polynomial operator+(Polynomial a, const Polynomial& b) { return a += b; }
Polynomial operator*(Polynomial a, const Polynomial& b) { return a *= b; }

std::ostream& operator<<(std::ostream& os, const Polynomial& p) {
```

```
    if (p.degree() < 0) return os << '0';
    bool first = true;
    for (int k = p.degree(); k >= 0; --k) {
        const long long c = p.coefficient(k);
        if (c == 0) continue;
        if (first) {
            if (c < 0) os << '-';
        } else {
            os << (c < 0 ? " - " : " + ");
        }
        const long long magnitude = c < 0 ? -c : c;
        if (magnitude != 1 || k == 0) os << magnitude;
        if (k >= 1) os << 'x';
        if (k >= 2) os << '^' << k;
        first = false;
    }
    return os;
}

Polynomial readPolynomial() {
    int degree;
    std::cin >> degree;
    std::vector<long long> c(static_cast<std::size_t>(degree) + 1);
    for (long long& v : c) std::cin >> v;
    return Polynomial(c);
}

int main() {
    const Polynomial p = readPolynomial();
    const Polynomial q = readPolynomial();
    long long x;
    std::cin >> x;
    std::cout << "p = " << p << '\n';
    std::cout << "q = " << q << '\n';
    std::cout << "p + q = " << p + q << '\n';
    std::cout << "p * q = " << p * q << '\n';
    std::cout << "p(" << x << ") = " << p(x) << '\n';
    std::cout << "q(" << x << ") = " << q(x) << '\n';
    std::cout << "(p * q)(" << x << ") = " << (p * q)(x) << '\n';
    return 0;
}
```

EXERCISE 2

Grid visits with a hashed key

A walker starts at `(0, 0)` on an integer grid and follows a string of moves: `U` (y + 1), `D` (y − 1), `L` (x − 1), `R` (x + 1); any other character is ignored. Count how many times each cell is visited — the start counts as one visit — in a `std::unordered_map<Point, int>`. That needs a `Point` struct with a defaulted `operator<=>` (which also provides the `==` the map requires and the `<` you will sort by) and a specialisation `template <> struct std::hash<Point>` that combines the two member hashes. Never print in the map's own order: copy the cells that were visited more than once into a `std::vector` and sort it.

Input: one line, the move string (possibly empty). **Output:** `distinct=<number of different cells visited>`, `revisited=<number of cells visited more than once>`, then those cells in ascending `(x, y)` order as `(x, y): <visits>`.

```
UDUD
```

prints

```
distinct=2
revisited=2
(0, 0): 3
(0, 1): 2
```

INPUT 1

```
RRUULLDD
```

OUTPUT 1

```
distinct=8
revisited=1
(0, 0): 2
```

INPUT 2

```
UDUD
```

OUTPUT 2

```
distinct=2
revisited=2
(0, 0): 3
(0, 1): 2
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <compare>
#include <functional>
#include <iostream>
#include <string>
#include <unordered_map>
#include <utility>
#include <vector>

struct Point {
    int x = 0;
    int y = 0;
    auto operator<=>(const Point&) const = default;   // == for the hash map, < for
};

template <>
struct std::hash<Point> {
    std::size_t operator()(const Point& p) const noexcept {
        const std::size_t h1 = std::hash<int>{}(p.x);
        const std::size_t h2 = std::hash<int>{}(p.y);
        return h1 ^ (h2 + 0x9e3779b9 + (h1 << 6) + (h1 >> 2));   // combine, so (1,
    }
};

std::ostream& operator<<(std::ostream& os, const Point& p) {
    return os << '(' << p.x << ", " << p.y << ')';
}

int main() {
    std::string moves;
    std::getline(std::cin, moves);
    std::unordered_map<Point, int> visits;
    Point pos;
    ++visits[pos];                          // the start is a visit
    for (const char m : moves) {
        switch (m) {
            case 'U': ++pos.y; break;
            case 'D': --pos.y; break;
            case 'L': --pos.x; break;
            case 'R': ++pos.x; break;
            default: continue;              // not a move: no visit
        }
        ++visits[pos];
    }
    std::vector<std::pair<Point, int>> revisited;
    for (const auto& [cell, count] : visits)
        if (count > 1) revisited.emplace_back(cell, count);
    std::sort(revisited.begin(), revisited.end());   // pair compares the Point firs
    std::cout << "distinct=" << visits.size() << '\n';
```

```
    std::cout << "revisited=" << revisited.size() << '\n';
    for (const auto& [cell, count] : revisited) std::cout << cell << ": " << count <
    return 0;
}
```

EXERCISE 3

A duration ledger

Write a `Duration` class holding a number of seconds with: an `operator>>` that reads the form `h:mm:ss` (hours with one to nine digits, then exactly two digits each for minutes and seconds, both 00–59) and sets `failbit` on anything else; an `operator<<` that prints the same form with minutes and seconds zero-padded — and **restores the stream's fill character** afterwards; `+=` and `+`; and a defaulted `operator⟺` so the durations can be sorted with `std::sort`.

Input: `n` (at least 1), then `n` durations, one per line. **Output:** `total <sum>`, `longest <max>`, `shortest <min>`, the `n` durations in ascending order one per line, then `count` followed by `n` printed with `std::setw(4)` — which comes out right only if your `<<` put the fill character back. If a duration is malformed, print `bad duration` and nothing else.

```
3
1:30:00
0:45:30
2:00:15
```

prints

```
total 4:15:45
longest 2:00:15
shortest 0:45:30
0:45:30
1:30:00
2:00:15
count   3
```

INPUT 1

```
3
1:30:00
0:45:30
2:00:15
```

OUTPUT 1

```
total 4:15:45
longest 2:00:15
shortest 0:45:30
0:45:30
1:30:00
2:00:15
count   3
```

INPUT 2

```
2
0:00:59
0:00:01
```

OUTPUT 2

```
total 0:01:00
longest 0:00:59
shortest 0:00:01
0:00:01
0:00:59
count   2
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cctype>
#include <compare>
#include <iomanip>
#include <iostream>
#include <string>
#include <vector>

class Duration {
public:
    Duration() = default;
    explicit Duration(long long seconds) : seconds_(seconds) {}
    long long seconds() const { return seconds_; }
    Duration& operator+=(const Duration& o) { seconds_ += o.seconds_; return *this;
    auto operator<=>(const Duration&) const = default;   // one member: ordering by
private:
    long long seconds_ = 0;
};

Duration operator+(Duration a, const Duration& b) { return a += b; }

std::ostream& operator<<(std::ostream& os, const Duration& d) {
    const long long s = d.seconds();
    const char previousFill = os.fill('0');              // fill is sticky: remember
    os << s / 3600 << ':' << std::setw(2) << (s / 60) % 60 << ':' << std::setw(2) <<
    os.fill(previousFill);                               // and put it back
    return os;
}

std::istream& operator>>(std::istream& is, Duration& d) {
    std::string tok;
    if (!(is >> tok)) return is;
    const std::size_t colon = tok.find(':');
    bool ok = colon != std::string::npos && colon >= 1 && colon <= 9
              && tok.size() == colon + 6 && tok[colon + 3] == ':';
    for (std::size_t i = 0; ok && i < tok.size(); ++i)
        if (i != colon && i != colon + 3 && !std::isdigit(static_cast<unsigned char>
    if (ok) {
        const long long h = std::stoll(tok.substr(0, colon));
        const int m = std::stoi(tok.substr(colon + 1, 2));
        const int sec = std::stoi(tok.substr(colon + 4, 2));
        if (m > 59 || sec > 59) ok = false;
        else d = Duration(h * 3600 + m * 60 + sec);      // assigned only for a valid
    }
    if (!ok) is.setstate(std::ios::failbit);
    return is;
}

int main() {
```

```
    int n;
    std::cin >> n;
    std::vector<Duration> entries;
    for (int i = 0; i < n; ++i) {
        Duration d;
        if (!(std::cin >> d)) {
            std::cout << "bad duration\n";
            return 0;
        }
        entries.push_back(d);
    }
    Duration total;
    for (const Duration& d : entries) total += d;
    std::sort(entries.begin(), entries.end());           // the rewritten < from the
    std::cout << "total " << total << '\n';
    std::cout << "longest " << entries.back() << '\n';
    std::cout << "shortest " << entries.front() << '\n';
    for (const Duration& d : entries) std::cout << d << '\n';
    std::cout << "count" << std::setw(4) << n << '\n';   // padded with spaces only
    return 0;
}
```

CHECKPOINT QUESTIONS

1. Which operators cannot be overloaded at all?

A. `,` , `&&` , `||`

B. `.` , `::` , `?:` , `sizeof`

C. `new` , `delete`

D. `<<` , `>>`

2. What does this print?

```
struct C {
    int n = 5;
    C& operator--() { --n; return *this; }
    C operator--(int) { C old = *this; --*this; return old; }
};
int main() {
    C c;
    std::cout << (c--).n << (--c).n << c.n;
}
```

A. `533`

B. `543`

C. `433`

D. `544`

3. `Vec2` has a member `Vec2 operator*(double k) const`. Which expression fails to compile?

A. `v * 2.0`

B. `v * 2`

C. `2.0 * v`

D. `(v * 2.0) * 3.0`

4. In `Vec2 operator+(Vec2 a, const Vec2& b) { return a += b; }`, what is the reason the first parameter is a value?

A. Operators cannot take two references

B. The copy is the working space: `+=` modifies it and the result is returned without touching the caller's objects

C. It makes the operator a member

D. By-value parameters are faster than references for every type

5. `struct V { int a, b; auto operator⟺(const V&) const = default; };` Which is true?

A. Only `⟺` exists; `<` must be written by hand

B. `<`, `>`, `⩽`, `⩾`, `=` and `≠` all work, comparing `a` then `b`

C. `=` compares `a` only

D. The struct cannot be used as a `std::map` key

6. Adding a `double` member to a struct with a defaulted `auto operator⟺` changes the deduced comparison category to…

A. `std::strong_ordering`

B. `std::weak_ordering`

C. `std::partial_ordering`

D. It no longer compiles

7. A `std::set<Tag>` whose `<` ignores case is given `Apple`, `apple` and `APPLE`. What does it hold?

A. Three elements

B. One element: the first inserted, `Apple`

C. One element: the last inserted, `APPLE`

D. A compile error: `std::set` needs `=`

8. `std::cout << std::setw(8) << money;` pads only the first digit of the amount. What is wrong?

A. `setw` does not work with user types

B. `operator<<` for `Money` inserts several pieces; `setw` applies to the first insertion only

C. `Money` is missing `operator>>`

D. The stream is in `std::fixed` mode

9. An `operator>>` reads `(3,` and then finds `x` instead of a number. The correct behaviour is to…

A. Assign `x = 3` to the target and return

B. Set `failbit`, leave the target unchanged and return the stream

C. Throw an exception

D. Print `bad input` and return

10. A class provides only `int& operator[](std::size_t)`. What fails?

A. `r[0] = 1;` on a non-const object

B. Indexing through a `const Row&` parameter

C. Passing `r` to a function by value

D. `r[0]++`

11. `std::priority_queue<int, std::vector<int>, std::greater<int>> q;` after pushing 5, 1, 3, what is `q.top()`?

A. 5

B. 1

C. 3

D. Unspecified

12. Which pair must agree for a `Point` used as an `unordered_map` key?

A. `operator<` and `operator<=>`

B. `operator==` and `std::hash<Point>`: equal points must hash equal

C. `operator<<` and `operator>>`

D. The copy constructor and the destructor

13. `struct M { explicit M(int); }; void f(M);` Which call compiles?

A. `f(3);`

B. `f(M(3));`

C. `M m = 3; f(m);`

D. `f({3});`

14. `struct R { explicit operator bool() const; }; R r;` Which line fails to compile?

A. `if (r) { }`

B. `while (!r) { }`

C. `bool ok = r && true;`

D. `bool ok = r;`

ANSWER KEY · CHECKPOINT — OPERATOR OVERLOADING

1. B Member access, scope resolution, the conditional operator and `sizeof` are fixed by the language. The comma and logical operators *can* be overloaded (and should not be); `new` / `delete` and the stream operators are overloaded routinely.

2. A Postfix returns the old value (5) and decrements to 4; prefix decrements to 3 and returns the object; `c.n` is then 3.

3. C A member operator's left operand must already be a `Vec2`; `2.0` is a `double` and no conversion is tried on it. A free `Vec2 operator*(double, Vec2)` is needed for the scalar-first form.

4. B `+` must leave both operands unchanged, so the work happens on a copy — and making the parameter that copy lets a temporary argument be moved in rather than copied again.

5. B A defaulted `<=>` supplies the four relational operators through rewriting and also declares a defaulted `==`; both compare members in declaration order, which is exactly what `std::map` and `std::sort` need.

6. C `double` supports only a partial ordering because NaN is unordered, and `auto` deduces the weakest category among the members.

7. B Ordered containers use only `<`; two keys are the same when neither is less than the other. `insert` does nothing for an equivalent key, so the first spelling stays.

8. B Width is one-shot. Build the whole text in an `std::ostringstream` and insert it once, and the caller's width covers the entire amount.

9. B Read into locals, assign only on complete success, and signal failure through the stream's state so `if (is >> p)` is false — the contract the built-in operators follow.

10. B A `const` object can call only `const` member functions, so the read-only caller cannot index at all. The second overload, `const int& operator[](std::size_t) const`, fixes it.

11. B With `std::greater` the queue is a min-heap: the smallest element is on top. With the default `std::less` it would be 5.

12. B The map finds a key by hash and then confirms it with `==`; if two equal points can land in different buckets, lookups miss. Ordering is irrelevant to an unordered container.

13. B `explicit` forbids the implicit conversion from `int` in a function argument, in copy-initialisation and in copy-list-initialisation (`{3}`); naming the constructor with `M(3)` is direct-initialisation and is allowed.

14. D Copy-initialising a `bool` is not a contextual conversion, so the explicit operator is not applied. Conditions and the operands of `!`, `&&` and `||` are; `static_cast<bool>(r)` is the explicit form.
