Module 8 of 20

# Classes and objects

Structs, aggregates and designated initialisers; constructors, the member-initialiser list and declaration-order initialisation; this, const member functions, mutable and chaining; static members and named constructors; invariants, friends and operator<<; the header/source split and a worked class design.

---

A `struct` or `class` is a type you define, and in C++ it is a *value*: held in the variable itself, copied on assignment, destroyed with its scope, laid out in memory exactly as its members are declared. Everything a Java programmer knows about objects is here, but with the handle removed — which is what makes a class able to own a heap block or a file, and what makes the questions of construction, constness and copying matter in a way they do not elsewhere.

The module builds a class from the outside in. Structs and aggregates first: default access, member functions inside and outside the class, brace and designated initialisation, and when a transparent bundle of data is the right design. Then constructors — the ones you write, the ones the compiler writes, the member-initialiser list, the rule that members are initialised in declaration order whatever the list says, delegation, `explicit`, and validation as the constructor's job. `this` and `const` member functions turn "does not modify" into a checked promise, with `mutable` for caches and `return *this` for chaining. Static members hold state that belongs to the type, from instance counters to named constructors. Encapsulation makes the case for private data precisely — an invariant the constructor establishes, mutators preserve and readers assume — and introduces `friend` for the few operations that need private access from outside, `operator<<` first among them. The last lesson splits a class across a header and a source file and walks one design from representation to interface.

Every exercise is a judged program on the Clang 18 / C++20 runtime: a bounding box built from aggregates, inventory records filled with designated initialisers, `Money` with three delegating constructors, a construction trace that proves the declaration-order rule, a statistics class whose cached mean counts its rebuilds, a query builder whose chain silently breaks when a mutator returns by value, counted sessions with copying deleted, temperatures built by named constructors, `Money` that prints itself through a friend, an `Account` audited by a friend class, a `Date` with `dayOfYear` and `nextDay`, and a `Fraction` written from a blank file through the eight-question checklist. The checkpoint adds a chaining gradebook, a grid with const and non-const `at()`, and an instrumented assembly line whose trace shows construction and destruction order.

CONTENTS


| | | |
|---|---|---|
| 1 | Structs and classes — a type of your own | 13 min |
| 2 | Constructors — building a valid object | 15 min |
| 3 | this, const member functions and chaining | 14 min |
| 4 | Static members — state that belongs to the class | 13 min |
| 5 | Encapsulation and friends — the interface a class publishes | 14 min |
| 6 | Designing a class — headers, sources and a worked design | 15 min |
| ✓ | Checkpoint — Classes and objects | 25 min |


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

LESSON 1 · 13 MIN

# Structs and classes — a type of your own

Everything so far has used types the language or the standard library defined. A `struct` or `class` is a type *you* define: a bundle of named members, the functions that operate on them, and — the part that separates C++ from Java — value semantics: a `Point` is a value like an `int`, held in the variable itself, copied on assignment, destroyed with its scope. This lesson settles the difference between `struct` and `class`, how member functions are declared and defined, what an aggregate is and how C++20's designated initialisers fill one in, and when a plain struct is the right design.

---

## Declaring a struct

```
#include <iostream>

struct Point {
    int x = 0;      // default member initialiser
    int y = 0;
    int manhattan() const { return (x < 0 ? -x : x) + (y < 0 ? -y : y); }
};   // the semicolon is part of the definition

int main() {
    Point p{3, -4};
    Point q;                        // {0, 0} from the initialisers
    std::cout << p.manhattan() << ' ' << q.x << '\n';   // 7 0
    return 0;
}
```

A struct definition names the type and lists its **members** between the braces: data members (`x`, `y`) and member functions (`manhattan`). After the definition `Point` is a type like any other — you can declare variables of it, put it in a `std::vector<Point>`, pass it by `const Point&`, return it by value. The trailing semicolon is required, and its absence produces a confusing error on the *next* declaration in the file.

Access members with `.` on an object and `→` through a pointer (Module 6, lesson 2). Inside a member function the members are simply in scope: `manhattan` reads the `x` and `y` of whichever `Point` it was called on.

## struct versus class

```
class Point {
    int x;          // private by default
public:
    int getX() const { return x; }
};

struct Point2 {
    int x;          // public by default
};
```

The two keywords define the same kind of thing. There are exactly two differences: members of a `struct` are `public` until an access specifier says otherwise, members of a `class` are `private`; and a base class (Module 10) is inherited publicly by default for a `struct` and privately for a `class`. That is all — a `class` can have public data and a `struct` can have private data and member functions, and the compiler treats them identically. The choice is a signal to readers. The convention across the community, and in this track: `struct` when every member is public and any combination of values is valid — a bundle of data; `class` when the type has an invariant to protect (lesson 5) and therefore private members behind a public interface. An access specifier (`public:`, `private:`, `protected:`) applies to everything after it until the next one, and a definition may switch sections as often as it likes; the usual order is the public interface first, because that is what readers want.

## Member functions inside and outside the class

A member function can be defined in the class body, or only *declared* there and defined later under the qualified name `Type::name`:

```
struct Rect {
    int w;
    int h;
    int area() const;                       // declaration
    bool isSquare() const { return w == h; }   // definition inside: implicit
};

int Rect::area() const {                    // definition outside, qualifie
    return w * h;
}
```

`Rect::area` says "the `area` that belongs to `Rect`", and inside its body the members are in scope exactly as they were inside the class. The outside form is what the header/source split needs (lesson 6): the class with its declarations lives in the `.h`, the bodies in the `.cpp`. A function defined inside the class body is implicitly `inline` (Module 4, lesson 6), so a header holding it can be included from many files without breaking the one-definition rule. Short one-liners belong inside; anything with a loop belongs outside, where it does not clutter the interface a reader is scanning. The `const` after the parameter list is part of the signature and must appear on both the declaration and the definition; lesson 3 explains what it promises.

---

## Aggregates and designated initialisers

A struct with no user-declared constructors, no private or protected non-static data members, no virtual functions and no base classes is an **aggregate**, and it can be initialised member by member with braces, in declaration order:

```
struct Config {
    std::string host = "localhost";
    int port = 8080;
    bool verbose = false;
};

Config a{"example.org", 443, true};      // positional
Config b{"example.org"};                 // port and verbose keep their defaul
Config c{.port = 9000, .verbose = true}; // C++20 designated initialisers
Config d{};                              // all defaults
```

Positional aggregate initialisation fills members in order and leaves the rest to their default member initialisers (or value-initialises them — zero — when there is none). C++20's **designated initialisers** name the member: `.port = 9000` is readable at the call site and immune to a reordering of the members before it. Two rules: designators must appear in declaration order (`{.verbose = true, .port = 9000}` is an error, not a reordering), and you cannot mix designated and positional values in one list. Brace initialisation also refuses narrowing (Module 2, lesson 4): `Config{"h", 3.5}` does not compile.

An aggregate can be returned with a bare brace list — `return {host, port, false};` — which is how a function hands back several named values without a `std::pair` whose `.first` says nothing (Module 4, lesson 2).

## Default member initialisers

```
struct Counter {
    int count = 0;          // every Counter starts at 0
    std::string label{};    // empty string — explicit
    double ratio;           // indeterminate unless initialised!
};
```

A default member initialiser gives the member a value whenever no constructor or brace list provides one. Without it, a member of fundamental type is left **indeterminate** by a plain `Counter c;`, and reading it is undefined behaviour — the same rule as for locals (Module 2, lesson 1). Give every fundamental member an initialiser; `= 0` or `{}` costs nothing and turns a silent garbage value into a predictable zero. Members of class type such as `std::string` construct themselves and need no help.

## Objects are values

```
Point a{1, 2};
Point b = a;      // b is a copy; two objects
b.x = 10;
std::cout << a.x << '\n';   // 1
```

A struct variable *is* the object, not a handle to one. Assignment copies every member (memberwise copy — Module 9), `==` is not provided until you ask for it (`bool operator== (const Point&) const = default;` in C++20 — Module 11), and a struct passed by value is copied, which is why anything bigger than a couple of words goes by `const&`. The layout in memory is the members in declaration order, possibly with padding between them (Module 19, lesson 3); `sizeof(Point)` is 8 on this platform for two `int`s.

## When a struct is the right choice

Use a plain struct when the type is *transparent*: any values the members can hold make sense together, there is nothing to enforce, and naming the fields is the whole point — a coordinate, a configuration, a record read from a file, a pair of results returned from a function. Adding member functions that compute from the fields (`manhattan`, `area`) keeps a struct a struct. The moment one member depends on another — a `size` that must match a

buffer, a `denominator` that must not be zero, a `balance` that must never go negative — the members can disagree, and the design needs private data and a constructor: the subject of the next four lessons.

## Pitfalls

**The vexing parse.** `Point p();` declares a *function* named `p` returning `Point`. Write `Point p;` or `Point p{};`.

**Missing semicolon** after the closing brace: the error appears on the following declaration.

**Uninitialised members**: `Point p;` with no default member initialisers holds garbage; `Point p{};` zeroes them.

**Designated initialisers out of order** or mixed with positional values: a compile error, not a reordering.

**Defining a member function outside the class without `Type::`** : you have written an unrelated free function, and the member is left undefined — a linker error when it is called.

**Forgetting `const` on the outside definition** when the declaration has it: "no declaration matches".

## Key takeaways

`struct` and `class` differ only in default access (and default inheritance); pick by intent — `struct` for transparent data, `class` for an invariant.

Member functions are declared in the class and defined inside (implicitly inline) or outside with `Type::name`.

An aggregate is initialised with braces in declaration order; C++20 designated initialisers name the members, in order, without mixing.

Give every fundamental member a default member initialiser; an unset one is indeterminate.

Objects are values: copying makes an independent object; pass by `const&`.

EXERCISE 1

## Bounding box

Complete two structs and a free function. `Point` has `int x = 0; int y = 0;` and a member function `int manhattan() const` returning `|x| + |y|`, **declared inside the struct and defined outside it** with `Point::manhattan`. `Rect` has `x`, `y`, `w`, `h` (all `int`, defaulting to 0), `long long area() const` defined outside the struct, and `bool isSquare() const` defined inside it. `Rect bounds(const std::vector<Point>&)` returns the smallest rectangle containing every point — `x` / `y` its lowest corner, `w` / `h` its extent — **returned as a bare brace list** (`return {minX, minY, ...};`), since `Rect` is an aggregate.

Read `n` points and build each with a designated initialiser (`Point{.x = x, .y = y}`). Print the box, its area, whether it is square (a zero-by-zero box is), and the largest Manhattan distance of any point from the origin.

**Input:** `n` (at least 1), then `n` lines `x y`. **Output:** four lines — `bbox <x> <y> <w> <h>`, `area <area>`, `square yes|no`, `farthest <distance>`.

```
4
1 2
5 -1
-3 4
2 2
```

→

```
bbox -3 -1 8 5
area 40
square no
farthest 7
```

INPUT 1

```
4
1 2
5 -1
-3 4
2 2
```

OUTPUT 1

```
bbox -3 -1 8 5
area 40
square no
farthest 7
```

INPUT 2

```
1
3 -3
```

OUTPUT 2

```
bbox 3 -3 0 0
area 0
square yes
farthest 6
```

REFERENCE SOLUTION

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

struct Point {
    int x = 0;
    int y = 0;
    int manhattan() const;
};

struct Rect {
    int x = 0;
    int y = 0;
    int w = 0;
    int h = 0;
    long long area() const;
    bool isSquare() const { return w == h; }
};

int Point::manhattan() const {
    return (x < 0 ? -x : x) + (y < 0 ? -y : y);
}

long long Rect::area() const {
    return static_cast<long long>(w) * h;
}

Rect bounds(const std::vector<Point>& pts) {
    int minX = pts[0].x, maxX = pts[0].x;
    int minY = pts[0].y, maxY = pts[0].y;
    for (const Point& p : pts) {
        if (p.x < minX) minX = p.x;
        if (p.x > maxX) maxX = p.x;
        if (p.y < minY) minY = p.y;
        if (p.y > maxY) maxY = p.y;
    }
    return {minX, minY, maxX - minX, maxY - minY};
}

int main() {
    int n;
    std::cin >> n;
    std::vector<Point> pts;
    pts.reserve(static_cast<std::size_t>(n));
    for (int i = 0; i < n; ++i) {
        int x, y;
        std::cin >> x >> y;
        pts.push_back(Point{.x = x, .y = y});
    }
    Rect r = bounds(pts);
```

```
    int farthest = 0;
    for (const Point& p : pts) {
        if (p.manhattan() > farthest) farthest = p.manhattan();
    }
    std::cout << "bbox " << r.x << ' ' << r.y << ' ' << r.w << ' ' << r.h << '\n';
    std::cout << "area " << r.area() << '\n';
    std::cout << "square " << (r.isSquare() ? "yes" : "no") << '\n';
    std::cout << "farthest " << farthest << '\n';
    return 0;
}
```

---

EXERCISE 2

## Inventory records

Define the aggregate `struct Item { std::string name; int quantity = 0; long long unitCents = 0; bool discontinued = false; }` with one member function, `long long value() const`, returning `quantity * unitCents`. Read `n` stock lines and build every `Item` with a **designated initialiser** naming the members you set — `Item{.name = name, .quantity = qty, .unitCents = cents, .discontinued = flag == "x"}` — so that a reader sees which value went where.

Print one line per item: `<name>: <quantity> at <unitCents> → <value>`, with `discontinued` appended when the flag is `x`. Then `stock value <sum of value() over items still stocked>` and `discontinued <count>`.

**Input:** `n`, then `n` lines `name quantity unitCents flag` where `flag` is `ok` or `x`. **Output:** `n + 2` lines.

```
3
bolt 10 25 ok
nut 40 5 ok
gear 2 1500 x
```

→

```
bolt: 10 at 25 → 250
nut: 40 at 5 → 200
gear: 2 at 1500 → 3000 discontinued
stock value 450
discontinued 1
```

### INPUT 1

```
3
bolt 10 25 ok
nut 40 5 ok
gear 2 1500 x
```

### OUTPUT 1

```
bolt: 10 at 25 → 250
nut: 40 at 5 → 200
gear: 2 at 1500 → 3000 discontinued
stock value 450
discontinued 1
```

### INPUT 2

```
1
widget 0 999 ok
```

### OUTPUT 2

```
widget: 0 at 999 → 0
stock value 0
discontinued 0
```

### REFERENCE SOLUTION

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

struct Item {
    std::string name;
    int quantity = 0;
    long long unitCents = 0;
    bool discontinued = false;
    long long value() const { return quantity * unitCents; }
};

int main() {
    int n;
    std::cin >> n;
    std::vector<Item> items;
    for (int i = 0; i < n; ++i) {
        std::string name, flag;
        int qty;
        long long cents;
        std::cin >> name >> qty >> cents >> flag;
        items.push_back(Item{.name = name, .quantity = qty, .unitCents = cents, .dis
    }
    long long stock = 0;
    int gone = 0;
    for (const Item& it : items) {
        std::cout << it.name << ": " << it.quantity << " at " << it.unitCents << " -
        if (it.discontinued) {
            std::cout << " discontinued";
            ++gone;
        } else {
            stock += it.value();
        }
        std::cout << '\n';
    }
    std::cout << "stock value " << stock << '\n';
    std::cout << "discontinued " << gone << '\n';
    return 0;
}
```

---

QUIZ

**1.** What are the differences between `struct` and `class` in C++?

A. A `struct` cannot have member functions

B. Only default member access (public vs private) and default inheritance access

C. A `class` is allocated on the heap, a `struct` on the stack

D. A `struct` cannot have a constructor

**2.**

```
Point p();
p.x = 3;
```

Why does the second line fail to compile?

A. `Point` has no default constructor

B. `p` is a function declaration — the most vexing parse — so it has no member `x`

C. `x` is private

D. A `Point` must be initialised with braces

**3.**

```
struct Config { std::string host = "localhost"; int port = 8080; bool verbose =
Config c{.verbose = true, .port = 9000};
```

What happens?

A. `c` is `{"localhost", 9000, true}`

B. Compile error: designators must appear in declaration order

C. `c` is `{"localhost", 8080, true}` — `port` is ignored

D. Undefined behaviour

**4.** With the same `Config`, what does `Config b{"example.org"};` produce?

A. Compile error: all three members must be given

B. `host` is set; `port` and `verbose` keep their default member initialisers (8080, false)

C. `host` is set; `port` and `verbose` are indeterminate

D. `host` is set; `port` and `verbose` are zero and false regardless of the defaults

**5.**

```
struct Rect { int w, h; int area() const; };
int area() { return 0; }   // intended as Rect::area
```

What is the result?

A. It works: the compiler matches the name to the declaration

B. Compile error on the definition

C. An unrelated free function; calling `r.area()` fails at link time with an undefined reference

D. `Rect::area` returns 0

**6.**

```
Point a{1, 2};
Point b = a;
b.x = 10;
std::cout << a.x;
```

What is printed?

A. `10`

B. `1`

C. It depends on whether `Point` is a `struct` or a `class`

D. Undefined behaviour

**7.** Which of these types is an aggregate?

A.

```
struct A { int x; explicit A(int v) : x(v) {} };
```

B.

```
class B { int x; public: int get() const { return x; } };
```

C.

```
struct C { std::string name; int count = 0; double ratio = 1.0; };
```

D.

```
struct D { int x; virtual int f() { return x; } };
```

ANSWER KEY · STRUCTS AND CLASSES — A TYPE OF YOUR OWN

**1. B** The two keywords define the same kind of type. Members of a `struct` are public and its bases inherited publicly by default; for a `class` both default to private. Where an object lives depends on how it is declared, not on the keyword.

**2. B** `Point p();` declares a function named `p` taking no arguments and returning `Point`. Write `Point p;` or `Point p{};` to declare an object.

**3. B** Designated initialisers must name members in the order they are declared; naming `verbose` before `port` is an error, not a reordering. `Config c{.port = 9000, .verbose = true};` is the valid spelling.

**4. B** Positional aggregate initialisation fills members in order and leaves the rest to their default member initialisers. Only a member with no initialiser would be value-initialised to zero.

**5. C** Without the `Rect::` qualifier the definition is a free function named `area`, and the member is declared but never defined. The compiler accepts the call; the linker reports `Rect::area() const` as undefined.

**6. B** `b` is a copy: a separate object with its own members. Objects are values, not handles, so changing `b` never affects `a`.

**7. C** An aggregate has no user-declared constructors, no private or protected non-static data members, no virtual functions and no base classes; default member initialisers are allowed. `A` declares a constructor, `B` has private data, `D` has a virtual function.

LESSON 2 · 15 MIN

# Constructors — building a valid object

A constructor turns raw storage into an object that keeps its promises. It runs exactly once per object, before any other member function can, and it is the only code that ever sees the members before they hold meaningful values. This lesson covers the constructors you write and the ones the compiler writes for you, the member-initialiser list and why it beats assignment in the body, the rule that members are initialised in declaration order whatever the list says, delegating constructors, `explicit`, and `= default` / `= delete`.

---

## Declaring constructors

```
#include <iostream>
#include <string>

class Account {
    std::string owner_;
    long long balance_ = 0;          // default member initialiser
public:
    Account() = default;                              // (1) default constructor
    Account(std::string owner, long long opening)  // (2) parameterised
        : owner_(std::move(owner)), balance_(opening) {}
    long long balance() const { return balance_; }
};

int main() {
    Account a;                        // (1)
    Account b{"ada", 500};            // (2)
    Account c("bob", 20);             // (2), parentheses also work
    std::cout << b.balance() << '\n';
    return 0;
}
```

A constructor has the class's name, no return type — not even `void` — and any number of parameters; constructors overload like functions (Module 4, lesson 3). A constructor with no parameters (or with defaults for all of them) is the **default constructor**, used by `Account a;`, by `std::vector<Account> v(10)`, and by every array of `Account`. Braces and parentheses both call a constructor; braces reject narrowing and cannot be mistaken for a function declaration, which is why this track prefers them for objects.

## What the compiler writes for you

If a class declares **no** constructor at all, the compiler generates a public default constructor that default-initialises each member: class-type members run their own default constructors, members with a default member initialiser take it, and fundamental members with neither are left indeterminate. Declare *any* constructor and the implicit default constructor is **not generated** — `Account a;` becomes a compile error the moment you add `Account(std::string, long long)`. That is exactly the situation the `= default` line above handles: it asks for the compiler-generated version explicitly. The mirror image is `= delete`, which declares a constructor in order to forbid it:

```
class Session {
public:
    Session(int id) : id_(id) {}
    Session() = delete;          // "no Session without an id" — the error name
private:
    int id_;
};
```

Copy and move constructors are generated by the same mechanism and are the subject of Module 9; for now, know that the rules for *those* are separate from the rule for the default constructor.

## The member-initialiser list

The part between the `:` and the body is the **member-initialiser list**. It is not decoration — it is where members are constructed, and it is the difference between initialising and assigning:

```
class Widget {
    std::string name_;
    const int id_;
    std::vector<int>& log_;
public:
    Widget(std::string name, int id, std::vector<int>& log)
        : name_(std::move(name)), id_(id), log_(log) {}   // initialised

    // Widget(std::string name, int id, std::vector<int>& log) {
    //     name_ = name;   // name_ was default-constructed first, then assigr
    //     id_ = id;       // error: assignment to a const member
    //     log_ = log;     // error: a reference must be bound at initialisat
    // }
};
```

By the time the body's opening brace runs, **every member already exists**. A member you did not list was default-initialised — an empty `std::string`, a vector with nothing in it, an indeterminate `int` — and assigning to it in the body is a second operation on top of the first. For `std::string` that is a wasted construction; for a `const` member, a reference member, or a member whose type has no default constructor, it is a compile error, because the body is too late. The rule: initialise every member in the list; the body is for work that needs all the members to exist, such as validation or a log line. A default member initialiser (`balance_ = 0` above) is the fallback the list does not override, which lets a class with several constructors state each default once.

---

## Initialisation order is declaration order

```
class Buffer {
    std::vector<int> data_;
    std::size_t size_;
public:
    Buffer(std::size_t n) : size_(n), data_(size_) {}   // BUG: data_ is built
};
```

Members are initialised in the order they are **declared in the class**, never in the order they appear in the list. Here `data_` is declared first, so `data_(size_)` runs before `size_(n)`, reading `size_` before it holds anything — undefined behaviour that happens to work in a debug build and produces a random-length vector under `-O2`. GCC and Clang warn with `-Wreorder` (part of `-Wall`) whenever the list disagrees with the declaration order; treat it as

an error. Write the list in declaration order, and never let one member's initialiser read a member declared after it — read the *parameter* (`data_(n)`) instead. Destruction runs in the reverse of construction: members are destroyed after the destructor body, last-declared first (Module 7, lesson 3).

---

## Delegating constructors

```
class Money {
    long long cents_;
public:
    explicit Money(long long cents) : cents_(cents) {}
    Money(long long units, int hundredths) : Money(units * 100 + hundredths) 
    Money() : Money(0) {}
};
```

A constructor whose initialiser list names *another constructor of the same class* delegates the whole job to it; the delegating constructor's own body, if any, runs after the target finishes. The target must be the **only** entry in the list — you cannot delegate and also initialise a member. Delegation keeps validation in one place: whatever `Money(long long)` checks, every other constructor gets for free. Before C++11 the same effect needed a private `init()` member called from each constructor, which could not initialise `const` members and left objects half-built if it failed.

---

## explicit

```
void charge(Money amount);

Money m = 5;      // without explicit: 5 converts to Money(5) — five cents, o
charge(250);      // without explicit: an int silently becomes a Money
```

A constructor callable with one argument is a **converting constructor**: the compiler may use it to convert a value of that argument type into your class, implicitly, anywhere a value of your class is expected. That is what makes `std::string s = "text";` work — and it is wrong for almost any class of your own, where a bare `250` has no obvious meaning. `explicit` switches the conversion off: `Money m{250};` and `Money(250)` still work, `Money m = 250;` and `charge(250)` do not compile. The rule the standard library follows and this track adopts:

**every constructor that can be called with one argument is `explicit` unless the conversion is the whole point**. `explicit` also applies to multi-argument constructors, where it blocks the brace form `charge({2, 50})`.

---

## Validation belongs in the constructor

```
#include <stdexcept>

class Percentage {
    int value_;
public:
    explicit Percentage(int v) : value_(v) {
        if (v < 0 || v > 100) throw std::invalid_argument("percentage out of r
    }
};
```

If an object can be built with inconsistent members, every member function has to defend against it forever. Checking in the constructor means a `Percentage` that exists is valid. When the arguments cannot make a valid object, the constructor's only escape is to throw — there is no return value to signal failure with — and a constructor that throws produces no object at all. Module 15 covers exceptions in full; lesson 6 of this module puts validation to work in a complete design.

---

## Pitfalls

| Mistake | What happens |
| --- | --- |
| `Account a();` | Declares a function; `a.balance()` fails with "request for member of non-class type". |
| Adding a constructor, keeping `Account a;` callers | The implicit default constructor vanished; add `Account() = default;` or initialise them. |
| Assigning members in the body | Double work at best; a compile error for `const`, references and non-default-constructible members. |
| Initialiser list out of declaration order | `-Wreorder`; a member initialised from a later one reads garbage. |
| Delegating and initialising in the same list | Compile error — the delegation must stand alone. |
| A single-argument constructor without `explicit` | Silent conversions from `int`, `double` or `const char*` at every call site. |

## Key takeaways

- Constructors have the class's name, no return type, and overload; declaring any constructor removes the implicit default one — `= default` restores it, `= delete` forbids it.
- Initialise members in the member-initialiser list; the body runs after every member exists.
- Members are initialised in **declaration order** regardless of the list; never read a later member.
- A constructor can delegate to another; the target is then the whole list.
- Mark single-argument constructors `explicit` unless an implicit conversion is intended.
- Validate in the constructor, so that an object that exists is valid.

EXERCISE 1

## Money, three ways

Write `class Money` holding `long long cents_` with three constructors. `explicit Money(long long cents)` is the target: it initialises the member in its **member-initialiser list** and throws `std::invalid_argument` when `cents` is negative. `Money(long long units, int hundredths)` **delegates** to it with `units * 100 + hundredths`, then checks in its own body that `hundredths` is 0–

99 and throws otherwise. `Money()` delegates with 0. Add `long long cents() const`, `Money add(const Money&) const` (a new object; `Money` is a value) and `std::string text() const` printing `<units>.<two digits>`.

Read `n` commands. `c <cents>` builds with the first constructor, `u <units> <hundredths>` with the second, `z` with the third. Print each amount's `text()`, or `invalid` when the constructor threw (catch `const std::invalid_argument&`), and keep a running total of the valid ones starting from `Money()`. Finish with `total <text>`.

**Input:** `n`, then `n` command lines. **Output:** `n + 1` lines.

```
4
c 1234
u 5 7
z
u 2 150
```

→

```
12.34
5.07
0.00
invalid
total 17.41
```

INPUT 1

```
4
c 1234
u 5 7
z
u 2 150
```

OUTPUT 1

```
12.34
5.07
0.00
invalid
total 17.41
```

INPUT 2

```
2
c -5
u 0 99
```

OUTPUT 2

```
invalid
0.99
total 0.99
```

REFERENCE SOLUTION

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

class Money {
    long long cents_;
public:
    explicit Money(long long cents) : cents_(cents) {
        if (cents < 0) throw std::invalid_argument("negative amount");
    }
    Money(long long units, int hundredths) : Money(units * 100 + hundredths) {
        if (hundredths < 0 || hundredths > 99) throw std::invalid_argument("bad hund
    }
    Money() : Money(0) {}
    long long cents() const { return cents_; }
    Money add(const Money& other) const;
    std::string text() const;
};

Money Money::add(const Money& other) const {
    return Money(cents_ + other.cents_);
}

std::string Money::text() const {
    std::string tail = std::to_string(cents_ % 100);
    if (tail.size() < 2) tail = "0" + tail;
    return std::to_string(cents_ / 100) + "." + tail;
}

Money readMoney(char kind) {
    if (kind == 'c') {
        long long c;
        std::cin >> c;
        return Money(c);
    }
    if (kind == 'u') {
        long long u;
        int h;
        std::cin >> u >> h;
        return Money(u, h);
    }
    return Money();
}

int main() {
    int n;
    std::cin >> n;
    Money total;
    for (int i = 0; i < n; ++i) {
```

```
        char kind;
        std::cin >> kind;
        try {
            Money m = readMoney(kind);
            std::cout << m.text() << '\n';
            total = total.add(m);
        } catch (const std::invalid_argument&) {
            std::cout << "invalid\n";
        }
    }
    std::cout << "total " << total.text() << '\n';
    return 0;
}
```

---

EXERCISE 2

## Robots, traced

`Part` is an instrumented member type: its constructor prints `build <name>` and its destructor `scrap <name>`. `Robot` has three `Part` members declared in the order `head_`, `arm_`, `leg_`, each with a **default member initialiser** naming it `head`, `arm` and `leg`, and a defaulted default constructor. Add two constructors:

- `explicit Robot(const std::string& prefix)` initialises all three members **in the member-initialiser list** as `<prefix>-head`, `<prefix>-arm`, `<prefix>-leg`, then its body prints `robot <prefix> ready`;
- `explicit Robot(int serial)` **delegates** to the first with `"r" + std::to_string(serial)`, then its body prints `serial <serial>`.

Read `n` commands, each run in its own block so the robot is destroyed at the closing brace: `d` builds `Robot r;`, `p <prefix>` builds `Robot r(prefix);`, `n <serial>` builds `Robot r(serial);`. Print `--` after each block. Predict the trace before you run it: members are built in declaration order and scrapped in reverse, whatever any list says.

**Input:** `n`, then `n` commands. **Output:** the trace.

```
2
d
n 7
```

→

```
build head
build arm
build leg
scrap leg
scrap arm
scrap head
--
build r7-head
build r7-arm
build r7-leg
robot r7 ready
serial 7
scrap r7-leg
scrap r7-arm
scrap r7-head
--
```

INPUT 1

```
2
d
n 7
```

OUTPUT 1

```
build head
build arm
build leg
scrap leg
scrap arm
scrap head
--
build r7-head
build r7-arm
build r7-leg
robot r7 ready
serial 7
scrap r7-leg
scrap r7-arm
scrap r7-head
--
```

INPUT 2

```
1
p x
```

OUTPUT 2

```
build x-head
build x-arm
build x-leg
robot x ready
scrap x-leg
scrap x-arm
scrap x-head
--
```

REFERENCE SOLUTION

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

struct Part {
    std::string name;
    explicit Part(std::string n) : name(std::move(n)) { std::cout << "build " << nam
    ~Part() { std::cout << "scrap " << name << '\n'; }
};

class Robot {
    Part head_{"head"};
    Part arm_{"arm"};
    Part leg_{"leg"};
public:
    Robot() = default;
    explicit Robot(const std::string& prefix)
        : head_(prefix + "-head"), arm_(prefix + "-arm"), leg_(prefix + "-leg") {
        std::cout << "robot " << prefix << " ready\n";
    }
    explicit Robot(int serial) : Robot("r" + std::to_string(serial)) {
        std::cout << "serial " << serial << '\n';
    }
};

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        char cmd;
        std::cin >> cmd;
        if (cmd == 'd') {
            Robot r;
        } else if (cmd == 'p') {
            std::string prefix;
            std::cin >> prefix;
            Robot r(prefix);
        } else {
            int serial;
            std::cin >> serial;
            Robot r(serial);
        }
        std::cout << "--\n";
    }
    return 0;
}
```

QUIZ

**1.** A class had only compiler-generated constructors and `Account a;` compiled. After `Account(std::string owner, long long opening)` is added, `Account a;` fails. Why?

A. Every constructor must now take arguments

B. Declaring any constructor stops the compiler generating the default one; add `Account() = default;`

C. `std::string` members cannot be default-constructed

D. The class became an aggregate

**2.**

```
class Buffer {
    std::vector<int> data_;
    std::size_t size_;
public:
    Buffer(std::size_t n) : size_(n), data_(size_) {}
};
```

What does `Buffer b(10);` do?

A. Creates a vector of 10 elements; the list order is what counts

B. Undefined behaviour: `data_` is declared first, so `data_(size_)` runs before `size_` is initialised

C. Compile error: the initialiser list is out of order

D. Creates an empty vector, then resizes it to 10

**3.** Why must a `const` member or a reference member be initialised in the member-initialiser list rather than assigned in the constructor body?

A. Because the body cannot see private members

B. Because by the time the body runs every member already exists, and neither a `const` nor a reference can be assigned afterwards

C. Because the initialiser list is faster and the compiler enforces the faster form

D. It is a style rule only; both compile

**4.**

```
Money(long long units, int hundredths) : Money(units * 100 + hundredths), cents
```

What happens?

A. Delegates, then sets `cents_` to 0

B. Compile error: a delegating constructor's initialiser list may contain nothing but the delegation

C. `cents_` is initialised twice; the second wins

D. Undefined behaviour

**5.** With `explicit Money(long long cents);` and `void charge(Money m);`, which line does **not** compile?

A. `Money m{250};`

B. `Money m(250);`

C. `Money m = 250;`

D. `charge(Money{250});`

**6.** A constructor validates its arguments and throws `std::invalid_argument`. What exists after the throw?

A. A half-built object whose destructor runs at scope exit

B. No object at all; members already constructed are destroyed during unwinding

C. An object in a default state

D. A null object that must be checked before use

**7.** What does `Session() = delete;` achieve?

A. Removes the destructor

B. Declares the default constructor so that using it is a compile error naming the deleted function

C. Makes the class abstract

D. Prevents the class from being copied

ANSWER KEY · CONSTRUCTORS — BUILDING A VALID OBJECT

**1. B** The implicit default constructor exists only while the class declares no constructor at all. `= default` asks for it explicitly; the members' types are irrelevant.

**2. B** Members are initialised in declaration order, so `data_` is built from an indeterminate `size_`. `-Wreorder` warns; the fix is `data_(n)` — read the parameter, never a later member.

**3. B** Initialisation happens before the body; a member not in the list is default-initialised, which is a compile error for a reference and leaves a `const` unassignable. For a `std::string` the body form merely does two operations instead of one.

**4. B** When a constructor delegates, the target constructor is the entire initialiser list. Anything else there is an error; extra work belongs in the delegating constructor's body.

**5. C** `explicit` forbids implicit conversion from the argument type, and copy-initialisation with `=` is an implicit conversion. Direct initialisation with braces or parentheses and an explicit `Money{250}` argument all still work.

**6. B** A constructor that exits by exception produces no object, so the destructor never runs; the fully constructed members are destroyed automatically. That is what makes "an object that exists is valid" a guarantee.

**7. B** `= delete` declares a function and forbids its use; `Session s;` then fails with "use of deleted function". Copying is governed by the copy constructor and assignment (Module 9), not by the default constructor.

LESSON 3 · 14 MIN

# this, const member functions and chaining

Every member function runs on behalf of one object, and two questions follow from that: how does the function name that object, and what may it do to it? The first answer is `this`. The second is `const`, which turns a member function's promise not to modify its object into something the compiler checks — and, through const objects, `const T&` parameters and const/non-const overloads, into a rule that shapes every interface you will write. This lesson settles both, adds `mutable` for the members the promise should not cover, shows why mutators that return `*this` can be chained, and ends with the question of when an operation should not be a member at all.

---

## What `this` is

Inside a non-static member function, `this` is a pointer to the object the function was called on. It is a keyword, not a variable you declare; its type is `Account*` in an ordinary member function of `Account` and `const Account*` in a const one. Every unqualified member name in the body is short for `this->member`, so you rarely write `this` at all:

```
class Account {
    std::string owner_;
    long long balance_ = 0;
public:
    void rename(const std::string& owner) { owner_ = owner; }          // this
    void rename2(const std::string& owner_) { this->owner_ = owner_; }  // a p
    const Account& self() const { return *this; }                      // the
};
```

There are two occasions for writing `this` explicitly. When a parameter or local shadows a member, `this->` is the only way to name the member — the trailing-underscore convention exists so that this never happens. And when the function needs the whole object: to return it (`*this`), to pass it to another function, or to compare identity (`this == &other` in an assignment operator, Module 9). `this` is never null in a well-formed program; calling a member function through a null pointer is undefined behaviour, whatever it happens to print. Static member functions (lesson 4) have no `this`, because they are not called on an object.

## const member functions

```
class Account {
    long long balance_ = 0;
public:
    long long balance() const { return balance_; }     // promises not to char
    void deposit(long long amount) { balance_ += amount; }
};
```

The `const` after the parameter list qualifies `this`: inside `balance()` the object is a `const Account`, every data member is read-only, and the body may call only other const member functions. Assign to `balance_` there and the message names the mechanism: "assignment of member 'balance_' in read-only object". The qualifier is part of the function's type, so a declaration and its out-of-class definition must both carry it, and `Type::name() const` is the form of the definition.

What the promise buys is on the caller's side. A `const Account`, and — far more often — a `const Account&` parameter, can call *only* const member functions:

```
void print(const Account& a) {
    std::cout << a.balance() << '\n';   // fine: balance() is const
    a.deposit(5);                       // error: passing 'const Account' as
}
```

Since `const T&` is the default way to pass anything larger than a register (Module 4, lesson 2), a member function that is not marked `const` when it could be is a member function that most of the program cannot call. Const-correctness is contagious in the useful direction (Module 6, lesson 5): mark every member function that does not modify the object `const` on the day you write it, and the compiler will tell you when a later edit breaks the promise.

## const and non-const overloads

A member function may be overloaded on the constness of the object. The standard containers do it everywhere:

```
class Readings {
    std::vector<double> values_;
public:
    double& at(std::size_t i) { return values_.at(i); }               // (1) j
    const double& at(std::size_t i) const { return values_.at(i); }   // (2) j
};

Readings r;                // ... fill ...
r.at(0) = 3.5;             // (1): a reference you can assign through
const Readings& view = r;
double x = view.at(0);     // (2): read-only access
view.at(0) = 1.0;          // error: assignment of read-only location
```

Overload resolution picks (2) when the object is const and (1) otherwise — exactly the choice `std::vector::operator[]`, `front()` and `at()` make. Without (2), a `const Readings&` could not be indexed at all; without (1), nobody could write through the result. When the two bodies are identical apart from constness, the non-const one may call the const one and cast the constness off the *result* — `return const_cast<double&>(std::as_const(*this).at(i));` — which is the one respectable use of `const_cast`, because the object was never really const. C++23's "deducing this" writes the pair once; on this track's runtime it is reading only.

---

## mutable

Sometimes a const member function must change a member that is not part of the object's *logical* value: a cache of an expensive result, a count of how often the object was read, a mutex that must be locked even to read (Module 17). `mutable` exempts a member from the const promise:

```
class Stats {
    std::vector<double> values_;
    mutable double cachedMean_ = 0;
    mutable bool cacheValid_ = false;
public:
    void add(double v) { values_.push_back(v); cacheValid_ = false; }
    double mean() const {
        if (!cacheValid_) {
            double sum = 0;
            for (double v : values_) sum += v;
            cachedMean_ = values_.empty() ? 0 : sum / static_cast<double>(valu
            cacheValid_ = true;     // allowed: the member is mutable
        }
        return cachedMean_;
    }
};
```

`mean()` is const from the caller's point of view — calling it twice gives the same answer and changes nothing observable — even though the second call skips the loop. That is the test: a `mutable` member is one whose changes are invisible through the interface. A `mutable` member that *is* part of the value is a lie about constness, and the compiler cannot catch it. The first exercise builds exactly this cache and counts how often it is rebuilt.

---

## Returning `*this` for chaining

A mutator that returns a reference to its own object lets calls be strung together:

```
class Query {
    std::string table_, where_;
    int limit_ = 0;
public:
    Query& from(std::string t) { table_ = std::move(t); return *this; }
    Query& where(std::string w) { where_ = std::move(w); return *this; }
    Query& limit(int n) { limit_ = n; return *this; }
    std::string text() const;
};

std::string sql = Query{}.from("users").where("age > 18").limit(10).text();
```

`from` returns `*this` — the object itself, by reference — so `.where(...)` is called on the same `Query`, and so on down the line. This is how `std::string::append`,

`std::ostream::operator<<` and every builder-style interface work. The return type must be `Query&`, not `Query`: returning by value would hand the next call a *copy*, the chain would modify the copy, and the original would be untouched — a bug that compiles cleanly and is found only by printing. A const member that returns the object returns `const Query&`.

---

## Member function or free function?

Not every operation on a class should be a member. A member can see the private state and is the natural home for anything that modifies the object or reads a representation detail. But an operation that can be written entirely in terms of the public interface — `isEmpty()` when there is a `size()`, a `print()` that only calls accessors — gains nothing from being a member and costs something: one more function that could break the invariant (lesson 5), one more thing to rewrite if the representation changes. Scott Meyers's rule: **prefer non-member non-friend functions** whenever the public interface suffices, because they *increase* encapsulation — fewer functions touch the private data. Symmetric binary operations (`==`, `+`, `<`) are free functions for a second reason: a member `a.equals(b)` converts only `b` (Module 11, lesson 1). An operation that needs private access to two objects at once (`transfer`) is a free function declared `friend` (lesson 5). Keep as members what needs private access to one object; put everything else beside the class, in the same header and namespace.

---

## Pitfalls

| Mistake | What happens |
| --- | --- |
| Forgetting `const` on a getter | Every `const T&` parameter and const object in the program cannot call it. |
| `const` on the declaration but not the definition (or vice versa) | "No declaration matches" — the qualifier is part of the signature. |
| A chaining mutator returning `T` instead of `T&` | The chain modifies temporaries; the original is unchanged. |
| A `mutable` member that is part of the logical value | The const promise is a lie; callers see a "const" object change. |
| `const_cast` to modify a genuinely const object | Undefined behaviour; the cast is only for a non-const object seen through a const reference. |
| Calling a member function through a null pointer | Undefined behaviour, even if the function never touches a member. |

## Key takeaways

- `this` is a pointer to the object a member function was called on; `*this` is the object, and `this→m` is the explicit form of a member name.
- A `const` member function promises not to modify the object and is the only kind a const object or a `const T&` can call — mark every non-mutating member `const`.
- Overload on constness (`T& at()` / `const T& at() const`) so const and non-const objects each get the right access.
- `mutable` exempts a member from the promise; use it for caches and counters that are not part of the logical value.
- Mutators that return `T&` and `return *this;` chain; returning by value silently breaks the chain.
- Prefer a non-member non-friend function whenever the public interface suffices.

EXERCISE 1

## A cached mean, counted

Complete `class Stats`: `void add(double)` appends a value and invalidates the cache; `double mean() const` returns the mean (0 for no values), recomputing it only when the cache is invalid and counting every recomputation; `int rebuilds() const` reports that count; `std::size_t count() const` the number of values. The cache — `cachedMean_`, `cacheValid_` and `rebuilds_` — must be `mutable`, because `mean()` is `const` and is called through the free function `report(const Stats&)`, which can reach only const members.

Read `n` commands: `add <x>`, `mean` (call `report`, which prints `mean <m> of <count>` with two decimals), `rebuilds` (prints `rebuilds <k>`).

**Input:** `n`, then `n` commands. **Output:** one line per `mean` or `rebuilds` command.

```
7
add 4
add 6
mean
mean
add 11
mean
rebuilds
```

→

```
mean 5.00 of 2
mean 5.00 of 2
mean 7.00 of 3
rebuilds 2
```

INPUT 1

```
7
add 4
add 6
mean
mean
add 11
mean
rebuilds
```

OUTPUT 1

```
mean 5.00 of 2
mean 5.00 of 2
mean 7.00 of 3
rebuilds 2
```

INPUT 2

```
3
mean
mean
rebuilds
```

OUTPUT 2

```
mean 0.00 of 0
mean 0.00 of 0
rebuilds 1
```

REFERENCE SOLUTION

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

class Stats {
    std::vector<double> values_;
    mutable double cachedMean_ = 0;
    mutable bool cacheValid_ = false;
    mutable int rebuilds_ = 0;
public:
    void add(double v) {
        values_.push_back(v);
        cacheValid_ = false;
    }
    std::size_t count() const { return values_.size(); }
    double mean() const {
        if (!cacheValid_) {
            double sum = 0;
            for (double v : values_) sum += v;
            cachedMean_ = values_.empty() ? 0 : sum / static_cast<double>(values_.si
            cacheValid_ = true;
            ++rebuilds_;
        }
        return cachedMean_;
    }
    int rebuilds() const { return rebuilds_; }
};

void report(const Stats& s) {
    std::cout << "mean " << s.mean() << " of " << s.count() << '\n';
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int n;
    std::cin >> n;
    Stats s;
    for (int i = 0; i < n; ++i) {
        std::string cmd;
        std::cin >> cmd;
        if (cmd == "add") {
            double v;
            std::cin >> v;
            s.add(v);
        } else if (cmd == "mean") {
            report(s);
        } else {
            std::cout << "rebuilds " << s.rebuilds() << '\n';
```

```
        }
    }
    return 0;
}
```

---

EXERCISE 2

## A chaining query builder

Complete `class Query`. The three mutators — `from(std::string table)`, `where(std::string condition)` (appends; any number may be given) and `limit(int n)` — must each **return** `Query&` with `return *this;` so that calls chain on the same object. `std::string text() const` renders `SELECT * FROM <table>`, then `WHERE <c1> AND <c2> ...` when there are conditions, then `LIMIT <n>` when the limit is positive.

For each of `n` queries, read `table limit k` and then `k` condition lines (they may contain spaces). Build it as `q.from(table).limit(limit);` followed by one `q.where(c)` per condition, and print `q.text()` through a `const Query&`. If your mutators return by value, the chain modifies a temporary and the limit is silently lost — the visible case catches it.

**Input:** `n`, then for each query a line `table limit k` and `k` lines. **Output:** `n` lines of SQL.

```
2
users 10 2
age > 18
city is pune
orders 0 0
```

→

```
SELECT * FROM users WHERE age > 18 AND city is pune LIMIT 10
SELECT * FROM orders
```

INPUT 1

```
2
users 10 2
age > 18
city is pune
orders 0 0
```

OUTPUT 1

```
SELECT * FROM users WHERE age > 18 AND
city is pune LIMIT 10
SELECT * FROM orders
```

### INPUT 2

```
1
logs 5 1
level above warn
```

### OUTPUT 2

```
SELECT * FROM logs WHERE level above
warn LIMIT 5
```

### REFERENCE SOLUTION

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

class Query {
    std::string table_;
    std::vector<std::string> where_;
    int limit_ = 0;
public:
    Query& from(std::string t) { table_ = std::move(t); return *this; }
    Query& where(std::string w) { where_.push_back(std::move(w)); return *this; }
    Query& limit(int n) { limit_ = n; return *this; }
    std::string text() const {
        std::string out = "SELECT * FROM " + table_;
        for (std::size_t i = 0; i < where_.size(); ++i) {
            out += (i == 0 ? " WHERE " : " AND ") + where_[i];
        }
        if (limit_ > 0) out += " LIMIT " + std::to_string(limit_);
        return out;
    }
};

void print(const Query& q) {
    std::cout << q.text() << '\n';
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string table;
        int lim, k;
        std::cin >> table >> lim >> k;
        std::cin.ignore();
        Query q;
        q.from(table).limit(lim);
        for (int j = 0; j < k; ++j) {
            std::string cond;
            std::getline(std::cin, cond);
            q.where(cond);
        }
        print(q);
    }
    return 0;
}
```

QUIZ

**1.** Inside `long long Account::balance() const`, what is the type of `this`?

A. `Account*`

B. `const Account*`

C. `Account&`

D. `const Account`

**2.**

```
class Query {
public:
    Query from(std::string t) { table_ = std::move(t); return *this; }
    Query limit(int n) { limit_ = n; return *this; }
    // ...
};
Query q;
q.from("users").limit(5);
```

After these lines, what does `q` hold?

A. Table `users`, limit 5

B. Table `users`, limit 0 — `limit` ran on a copy returned by value

C. Nothing: both calls ran on temporaries

D. Compile error: `*this` cannot be returned by value

**3.** A function takes `const Account& a`. Which of `a`'s member functions can it call?

A. All of them; `const` only prevents reassigning `a`

B. Only those declared `const` after the parameter list

C. Only static member functions

D. Only those that return `void`

**4.** Which member is a correct use of `mutable`?

A. The `balance_` of an `Account`, so `deposit` can be `const`

B. A `cachedMean_` that a const `mean()` recomputes lazily and callers cannot observe changing

C. Every member, so that const objects can be modified freely

D. A member that must be assigned in the constructor body

**5.**

```
struct Rect {
    int area() const;
};
int Rect::area() { return w * h; }
```

What happens?

A. Compiles; `const` on the declaration is enough

B. Compile error: no declaration matches `int Rect::area()` — the `const` is part of the signature

C. Compiles, but `area` is no longer callable on const objects

D. Linker error

**6.**

```
class Readings {
public:
    double& at(std::size_t i);
    const double& at(std::size_t i) const;
};
const Readings& view = r;
auto x = view.at(0);
```

Which overload is called, and what is `x`?

A. The non-const one; `x` is `double&`

B. The const one; `x` is a `double` copy of the element

C. The const one; `x` is `const double&`

D. Ambiguous: compile error

7. `bool isEmpty(const Stack& s)` can be written using only `s.size()`. Where should it live, and why?

   A. As a member, because members are faster

   B. As a non-member non-friend function, because that increases encapsulation — one fewer function can touch the private data

   C. As a friend, so it can read the private size directly

   D. As a static member function

ANSWER KEY · THIS, CONST MEMBER FUNCTIONS AND CHAINING

**1. B** The trailing `const` qualifies `this`: it becomes a pointer to a `const Account`, which is why the body cannot assign to members or call non-const member functions.

**2. B** `q.from(...)` modifies `q` and returns a copy; `.limit(5)` modifies that temporary, which dies at the semicolon. Returning `Query&` makes the whole chain act on `q`.

**3. B** Through a reference to `const`, the object is const, and only const member functions are callable; calling `a.deposit(5)` is rejected with "discards qualifiers". This is why every non-mutating member should be `const`.

**4. B** `mutable` exempts a member from the const promise and is right only when its changes are invisible through the interface — caches, counters, a mutex. Marking the logical value `mutable` makes the promise a lie.

**5. B** The const qualifier belongs to the function's type, so the definition must repeat it. The compiler sees a definition for an `area()` that was never declared.

**6. B** `view` is const, so only the const overload is viable. Plain `auto` drops the reference and the const, so `x` is an independent `double`.

**7. B** Anything expressible through the public interface gains nothing from private access and adds one more function that could break the invariant. Prefer non-member non-friend functions when the public interface suffices.

LESSON 4 · 13 MIN

# Static members — state that belongs to the class

Every data member so far has lived inside each object: two `Account`s, two balances. Some state belongs to the *type* instead — how many objects exist, the next id to hand out, a constant every object shares, a lookup table built once. A `static` member has exactly one instance per class, held in static storage (Module 7, lesson 1) and alive for the whole program whether or not any object exists. This lesson covers static data members and the C++17 `inline` that made them painless, the copy trap that makes instance counters drift, static member functions and the named-constructor pattern they enable, constants inside classes, and the singleton — the static-member design that interviews ask about and codebases regret.

---

## Static data members

```
class Session {
    std::string user_;
    int id_;
    inline static int alive_ = 0;      // one counter for the whole class
    inline static int nextId_ = 1;
public:
    explicit Session(std::string user) : user_(std::move(user)), id_(nextId_++
    ~Session() { --alive_; }
    static int alive() { return alive_; }
};
```

`alive_` is not part of any `Session`; `sizeof(Session)` counts `user_` and `id_` and nothing else. There is one `alive_`, initialised before `main` starts, and every constructor and destructor touches the same one. Inside member functions it is in scope like any other member; from outside it is `Session::alive_` (or `s.alive_` through an object, which is legal and misleading — write the class name).

Before C++17 a static data member was only *declared* inside the class and had to be *defined* exactly once, in one `.cpp` file: `int Session::alive_ = 0;`. A definition in a header that two files included was a linker error (multiple definition); no definition at all was a linker error the other way (undefined reference to `Session::alive_`). `inline static` (C++17) lets the in-

class declaration be the definition, the same way an inline function can be defined in a header (Module 4, lesson 6): every translation unit sees a definition and the linker keeps one. Use it for every static data member you write; the two-part form survives in code that predates 2017 and in interview questions about it.

---

## Instance counters, and the copy trap

The counter above has a hole. The compiler generates a copy constructor for `Session` (Module 9), and that generated constructor copies `user_` and `id_` — it does not run the body that increments `alive_`. Copy a `Session` and the count is one short; destroy both and it goes negative:

```
Session a("ada");        // alive 1
Session b = a;           // generated copy: alive still 1, but two objects ex
// ... both destroyed: alive is -1
```

Two correct fixes. Either write the copy constructor and count in it too, or — for a class where copying makes no sense, which is most classes with an id — delete it: `Session(const Session&) = delete; Session& operator=(const Session&) = delete;`. The first exercise takes the second route. The trap is worth remembering beyond counters: any class whose constructor has a side effect must ask what the generated copy does.

---

## Static member functions

A static member function belongs to the class rather than to an object: it has no `this`, cannot touch non-static members, and is called as `Session::alive()`. It can read private static members, which is why the counter is private and published through a public static function. Three uses recur.

**Access to class-level state**, as above. **Utilities tied to the type** — `Date::isLeapYear(int)` and `Date::daysInMonth(int, int)` need no `Date` to exist and belong nowhere else. And **named constructors**: a constructor can only be called by the class's name, so two ways of building an `Angle` from a `double` cannot both be constructors. Static factory functions give each a name, and a private constructor makes them the only way in:

```cpp
#include <numbers>

class Angle {
    double degrees_;
    explicit Angle(double d) : degrees_(d) {}          // private: only the fa
public:
    static Angle fromDegrees(double d) { return Angle(d); }
    static Angle fromRadians(double r) { return Angle(r * 180.0 / std::numbers
    double degrees() const { return degrees_; }
};

Angle a = Angle::fromRadians(1.0);
```

The standard library does the same: `std::chrono::system_clock::now()` is a static member function that builds a time point, and `std::thread::hardware_concurrency()` is a static utility that needs no thread. A factory can also validate and return `std::optional<T>` (Module 15, lesson 3) instead of throwing, return a cached instance, or return a derived type (Module 10) — none of which a constructor can do.

---

## Constants inside classes

```cpp
class Grid {
public:
    static constexpr int kMaxSide = 1024;
    static constexpr double kEpsilon = 1e-9;
    static inline const std::string kName = "grid";   // non-literal type: ini
};

int side = std::min(n, Grid::kMaxSide);
```

A constant that belongs to a type is a `static constexpr` data member: usable in constant expressions (array sizes, `static_assert`, template arguments), implicitly `inline` since C++17, and named through the class. `std::string::npos` is such a constant; `std::numeric_limits<int>::max()` is the function form. Prefer `static constexpr` to a `#define` (no scope, no type) and to the pre-C++11 trick of an unscoped `enum { kMaxSide = 1024 }`. A constant of a non-literal type such as `std::string` cannot be `constexpr`; `static inline const` gives it its one definition.

## Function-local statics in members

A `static` local inside a member function (Module 4, lesson 6) is initialised once, on first use, and shared by every object — a lazily built lookup table, for instance. It behaves like a static data member with tighter scope; use it when only one function needs the value.

## Why singletons are a trap

A **singleton** is a class arranged so that exactly one instance can exist, reached through a static function:

```
class Logger {
    Logger() = default;
public:
    static Logger& instance() {
        static Logger only;      // constructed on first call, destroyed at ex
        return only;
    }
    Logger(const Logger&) = delete;
    Logger& operator=(const Logger&) = delete;
    void log(const std::string& line);
};

Logger::instance().log("started");
```

The function-local static — the "Meyers singleton" — is the correct way to write one: construction is lazy and thread-safe since C++11, and it sidesteps the **static initialisation order fiasco**: globals in different translation units are constructed in an unspecified order, so a global that reads another global from a different file may read it before it exists. What the pattern cannot fix is the design. A singleton is a global variable with a class around it: every function that calls `Logger::instance()` has a hidden dependency its signature does not show; tests cannot substitute a fake; two of them cannot exist even when a test or a second tenant needs two; and destruction order at exit is again unspecified, so a singleton used from another object's destructor may already be gone. The alternative is almost always to construct the object once in `main` and pass a reference to whatever needs it. Interviews ask for the pattern; the answer that scores is the Meyers form *plus* why you would rather not.

## Pitfalls

| Mistake | What happens |
| --- | --- |
| `static int n;` in the class with no definition (pre-C++17 style) | Linker: undefined reference to `T::n`. Use `inline static`. |
| Defining `int T::n = 0;` in a header | Linker: multiple definition. Use `inline static`. |
| Counting in the constructor but not the copy constructor | The count drifts; delete copying or write the copy constructor. |
| Touching a non-static member from a static function | "Invalid use of member in static member function" — there is no `this`. |
| A global object that depends on a global in another file | Unspecified construction order; use a function-local static. |
| Reaching for a singleton because two functions need the same object | Pass it; the singleton hides the dependency. |

## Key takeaways

- A static data member has one instance per class, in static storage, initialised before `main`; write it `inline static` so the class body is its definition.
- A constructor's side effects are not repeated by the generated copy constructor: count in it too, or delete copying.
- Static member functions have no `this`; they hold class-level state, type utilities and named constructors (private constructor plus static factories).
- Constants belong in the class as `static constexpr` (or `static inline const` for non-literal types), named `Type::kName`.
- The Meyers function-local static is the correct singleton; the design is still a hidden global — pass the object instead.

EXERCISE 1

## Sessions, counted

Complete `class Session`. Two `inline static` data members hold class-level state: `alive_` (how many sessions exist now) and `nextId_` (the next id to hand out, starting at 1). The constructor takes a user name, assigns `id_ = nextId_++` and increments `alive_`; the destructor decrements `alive_`. **Delete the copy constructor and copy assignment** — a copied session would carry a duplicate id and skip the count. Publish the counters through static member functions `alive()` and `created()` (ids handed out so far).

Sessions are kept in a `std::vector<std::unique_ptr<Session>>` (a non-copyable type cannot live in a vector by value). Commands: `open <user>` creates one and prints `#<id> <user> opened (alive <n>)`; `close <id>` destroys it and prints `#<id> closed (alive <n>)`, or `no session #<id>` when no open session has that id; `report` prints `alive <a> created <c>`. Every count is read *after* the change.

**Input:** `n`, then `n` commands. **Output:** one line per command.

```
6
open ada
open bob
report
close 1
close 1
report
```

→

```
#1 ada opened (alive 1)
#2 bob opened (alive 2)
alive 2 created 2
#1 closed (alive 1)
no session #1
alive 1 created 2
```

INPUT 1

```
6
open ada
open bob
report
close 1
close 1
report
```

OUTPUT 1

```
#1 ada opened (alive 1)
#2 bob opened (alive 2)
alive 2 created 2
#1 closed (alive 1)
no session #1
alive 1 created 2
```

INPUT 2

```
2
report
close 5
```

OUTPUT 2

```
alive 0 created 0
no session #5
```

REFERENCE SOLUTION

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

class Session {
    std::string user_;
    int id_;
    inline static int alive_ = 0;
    inline static int nextId_ = 1;
public:
    explicit Session(std::string user) : user_(std::move(user)), id_(nextId_++) {
        ++alive_;
    }
    ~Session() {
        --alive_;
    }
    Session(const Session&) = delete;
    Session& operator=(const Session&) = delete;
    int id() const { return id_; }
    const std::string& user() const { return user_; }
    static int alive() { return alive_; }
    static int created() { return nextId_ - 1; }
};

int main() {
    int n;
    std::cin >> n;
    std::vector<std::unique_ptr<Session>> open;
    for (int i = 0; i < n; ++i) {
        std::string cmd;
        std::cin >> cmd;
        if (cmd == "open") {
            std::string user;
            std::cin >> user;
            open.push_back(std::make_unique<Session>(user));
            const Session& s = *open.back();
            std::cout << '#' << s.id() << ' ' << s.user() << " opened (alive " << Se
        } else if (cmd == "close") {
            int id;
            std::cin >> id;
            bool found = false;
            for (auto it = open.begin(); it != open.end(); ++it) {
                if ((*it)->id() == id) {
                    open.erase(it);
                    found = true;
                    break;
                }
            }
```

```
            if (found) {
                std::cout << '#' << id << " closed (alive " << Session::alive() << '
            } else {
                std::cout << "no session #" << id << '\n';
            }
        } else {
            std::cout << "alive " << Session::alive() << " created " << Session::cre
        }
    }
    return 0;
}
```

---

EXERCISE 2

## Temperatures by named constructor

`class Temperature` stores kelvin in a `double` behind a **private** constructor, so the only ways in are three static factories: `fromKelvin(double)` (throws `std::invalid_argument` below 0 K), `fromCelsius(double)` and `fromFahrenheit(double)`, each converting and delegating to `fromKelvin` so the check lives once. Keep the constant `static constexpr double kAbsoluteZeroC = -273.15;` in the class and use it for both conversions. Readers `kelvin()`, `celsius()` and `fahrenheit()` are `const`. A private `inline static int created_` counts how many temperatures the constructor built, published by `static int created()`.

Read `n` lines `<unit> <value>` with unit `C`, `F` or `K`. Print `<c> C | <f> F | <k> K` with two decimals, or `invalid` when the factory threw. Then `lowest <c> C` for the lowest valid temperature (`lowest none` if there was none) and `created <count>`.

**Input:** `n`, then `n` lines. **Output:** `n + 2` lines.

```
3
C 25
F 98.6
K 0
```

→

```
25.00 C | 77.00 F | 298.15 K
37.00 C | 98.60 F | 310.15 K
-273.15 C | -459.67 F | 0.00 K
lowest -273.15 C
created 3
```

### INPUT 1

```
3
C 25
F 98.6
K 0
```

### OUTPUT 1

```
25.00 C | 77.00 F | 298.15 K
37.00 C | 98.60 F | 310.15 K
-273.15 C | -459.67 F | 0.00 K
lowest -273.15 C
created 3
```

### INPUT 2

```
2
C -300
C 100
```

### OUTPUT 2

```
invalid
100.00 C | 212.00 F | 373.15 K
lowest 100.00 C
created 1
```

### REFERENCE SOLUTION

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

class Temperature {
    double kelvin_;
    inline static int created_ = 0;
    explicit Temperature(double k) : kelvin_(k) { ++created_; }
public:
    static constexpr double kAbsoluteZeroC = -273.15;
    static Temperature fromKelvin(double k) {
        if (k < 0) throw std::invalid_argument("below absolute zero");
        return Temperature(k);
    }
    static Temperature fromCelsius(double c) { return fromKelvin(c - kAbsoluteZeroC)
    static Temperature fromFahrenheit(double f) { return fromCelsius((f - 32.0) * 5.
    double kelvin() const { return kelvin_; }
    double celsius() const { return kelvin_ + kAbsoluteZeroC; }
    double fahrenheit() const { return celsius() * 9.0 / 5.0 + 32.0; }
    static int created() { return created_; }
};

Temperature build(char unit, double value) {
    if (unit == 'C') return Temperature::fromCelsius(value);
    if (unit == 'F') return Temperature::fromFahrenheit(value);
    return Temperature::fromKelvin(value);
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int n;
    std::cin >> n;
    bool any = false;
    double lowest = 0;
    for (int i = 0; i < n; ++i) {
        char unit;
        double value;
        std::cin >> unit >> value;
        try {
            Temperature t = build(unit, value);
            std::cout << t.celsius() << " C | " << t.fahrenheit() << " F | " << t.ke
            if (!any || t.celsius() < lowest) lowest = t.celsius();
            any = true;
        } catch (const std::invalid_argument&) {
            std::cout << "invalid\n";
        }
    }
    if (any) std::cout << "lowest " << lowest << " C\n"; else std::cout << "lowest r
```

```
    std::cout << "created " << Temperature::created() << '\n';
    return 0;
}
```

---

QUIZ

**1.** What does `inline static int alive_ = 0;` inside a class body mean?

A. Each object gets its own `alive_`, initialised to 0

B. One `alive_` for the whole class, and this line is its definition — no out-of-class `int T::alive_ = 0;` needed

C. A constant that cannot change

D. A local variable of every member function

**2.** A class increments a static `count_` in its constructor and decrements it in the destructor, but declares no copy operations. After `Session b = a;` and the destruction of both, what is `count_`?

A. 0

B. -1 — the generated copy constructor did not increment, but both destructors decremented

C. 1

D. 2

**3.**

```
class Session {
    int id_;
public:
    static int id() { return id_; }
};
```

Why does this fail to compile?

A. `id_` is private

B. A static member function has no `this`, so it cannot read a non-static member

C. Static functions cannot return `int`

D. `id_` is uninitialised

**4.** Why do `Angle::fromDegrees(double)` and `Angle::fromRadians(double)` exist as static functions with the constructor private?

A. Constructors cannot take `double`

B. Two constructors taking one `double` cannot be told apart, and a static factory gives each way of building an `Angle` a name

C. Static functions are faster than constructors

D. To make the class a singleton

**5.** Which is the preferred way to give a class a compile-time constant `kMaxSide` of 1024?

A. `#define kMaxSide 1024` above the class

B. `static constexpr int kMaxSide = 1024;` inside the class

C. `const int kMaxSide = 1024;` as a non-static member

D. `enum { kMaxSide = 1024 };` inside the class

**6.**

```
static Logger& instance() {
    static Logger only;
    return only;
}
```

What does the function-local static buy over a global `Logger g_logger;`?

A. Nothing; both are constructed before `main`

B. Lazy, thread-safe construction on first use, and immunity to the static initialisation order fiasco across translation units

C. A new `Logger` on every call

D. Automatic destruction when the last caller returns

**7.**

```
Session a("x");
{
    Session b("y");
    std::cout << Session::alive();
}
std::cout << Session::alive();
```

With a correct counter, what is printed?

A. `22`

B. `21`

C. `11`

D. `12`

ANSWER KEY · STATIC MEMBERS — STATE THAT BELONGS TO THE CLASS

**1. B** A static data member has one instance per class. Since C++17 `inline` lets the in-class declaration serve as the single definition, which is what makes it header-safe; before that the definition had to appear in exactly one source file.

**2. B** The compiler-generated copy constructor copies members and runs none of your constructor's body. Either count in a hand-written copy constructor or delete copying for a class whose identity should not be duplicated.

**3. B** A static member function is called on the class, not on an object; there is no object whose `id_` it could mean. It may read static members only.

**4. B** A constructor is only ever called by the class's name, so two meanings for one `double` need named entry points. The private constructor makes the factories the only way in, so every `Angle` went through one of them.

**5. B** A `static constexpr` member is typed, scoped to the class (`Grid::kMaxSide`), usable in constant expressions and implicitly `inline`. The macro has no scope or type, a non-static `const` member costs space in every object, and the enum trick predates `constexpr`.

**6. B** A function-local static is initialised on first call, once, with the initialisation guaranteed thread-safe since C++11, so no other translation unit's global can observe it before it exists. It remains a hidden global dependency — the design problem is not solved by the mechanism.

**7. B** Inside the block two sessions exist, so `2`; `b` is destroyed at the closing brace and its destructor decrements the counter, so the second print shows `1`.

LESSON 5 · 14 MIN

# Encapsulation and friends — the interface a class publishes

A class earns its `private` section by having something to protect. That something is an **invariant**: a statement about the members that is true of every object between operations — the denominator is never zero, the size never exceeds the capacity, the vector is sorted. Public data cannot keep such a promise, because any line of the program can break it. This lesson makes the case for private data precisely, sorts the accessors that earn their place from the ones that merely re-publish the members, introduces `friend` for the few operations that genuinely need private access from outside, previews `operator<<` as the commonest friend, and ends with the question every class should answer: what interface does it publish?

---

## Invariants

```
class Fraction {
    long long num_;
    long long den_;   // invariant: den_ > 0 and gcd(|num_|, den_) == 1
public:
    Fraction(long long n, long long d);              // establishes it (throws
    Fraction add(const Fraction& o) const;           // preserves it: returns
    long long denominator() const { return den_; }   // may assume it
};
```

The invariant is a contract between the constructor, the mutators and the readers. The **constructor establishes** it — an object that exists is valid (lesson 2). Every **public mutator preserves** it: it may leave the object in any state that satisfies the invariant, and in no other. Every **reader may assume** it: `add` divides by `den_` without checking for zero, because zero is impossible. That last part is the payoff. Without the invariant, every function that touches a `Fraction` defends against a zero denominator forever; with it, one check in the constructor covers the program.

Write the invariant down, as a comment beside the private members. If you cannot state one, the type has no reason to hide its data — it is a struct (lesson 1), and pretending otherwise adds ceremony without protection.

## Why data is private

Privacy is not secrecy: the private section sits in the header for anyone to read. It is *control*. Every write to the members goes through code the class author wrote, so the invariant can be checked at each write, and the representation can change without touching any caller. A `Fraction` stored as two `long long`s today could become a fixed-point number tomorrow, and `add`'s callers would never know.

```
struct Percentage { int value; };          // anyone can write p.value = 250

class Percentage {
    int value_;                            // invariant: 0 ⩽ value_ ⩽ 100
public:
    explicit Percentage(int v);            // the only way in, and it checks
    int value() const { return value_; }
};
```

The class version is four lines longer and infinitely more trustworthy: a `Percentage` that exists is between 0 and 100, wherever it came from and however many functions have handled it since.

## Accessors that earn their place — and the ones that do not

The reflex "make the data private and add `getX()` and `setX()` for every member" produces a struct with extra steps: every member can still be set to anything, and now it takes a function call. An accessor earns its place when it does something the bare member cannot:

| Accessor | Verdict |
| --- | --- |
| `void setBalance(long long b)` on an `Account` | Does not earn it: bypasses every rule about deposits and withdrawals. |
| `bool withdraw(long long amount)` | Earns it: checks funds, preserves `balance_ ⩾ 0`, reports refusal. |
| `int getYear() const` on a `Date` | Fine as a reader — but name it `year()`; the `get` prefix says nothing. |
| `int dayOfYear() const` | Earns it: a derived value computed from the representation. |
| `std::vector<int>& items()` (non-const reference) | Breaks encapsulation: the caller can now edit the vector behind the invariant's back. |
| `const std::vector<int>& items() const` | Acceptable for reading a large member without copying it. |

Two rules fall out. Readers return by value for small types and by `const T&` for large ones — never a non-const reference or pointer to private state, which is a public member with more syntax. And mutators are *operations*, named for what they do to the object (`deposit`, `withdraw`, `normalise`), not for the member they happen to change. A class whose public functions are all `getX` / `setX` pairs has not found its operations yet.

---

## friend functions

Occasionally the right function cannot be a member and still needs private access. `friend` grants it, one function at a time:

```
class Account {
    long long balance_;
public:
    explicit Account(long long opening) : balance_(opening) {}
    friend bool transfer(Account& from, Account& to, long long amount);   // d
};

bool transfer(Account& from, Account& to, long long amount) {             // d
    if (amount ⩽ 0 || from.balance_ < amount) return false;
    from.balance_ -= amount;      // allowed: transfer is a friend
    to.balance_ += amount;
    return true;
}
```

`transfer` acts on two accounts symmetrically and must change both or neither. As a member it would be lopsided — `from.transferTo(to, n)` — and the free function says what it is. The `friend` line inside the class does not make the function a member: it has no `this`, it is called as `transfer(a, b, 5)`, and its definition carries no `Account::`. Friendship has three properties worth memorising. It is **granted, not taken**: the class names its friends; a function cannot declare itself one. It is **not transitive**: a friend of `Account` gets no access to the private parts of `Account`'s members. And it is **not inherited** (Module 10). Use it sparingly — every friend is one more function that can break the invariant, and the count of such functions is the measure of how encapsulated a class really is.

---

## friend classes

`friend class Auditor;` inside `Account` lets every member function of `Auditor` read and write `Account`'s private members. This is for pairs designed together — a container and its iterator, a builder and the object it assembles, a test fixture and the class under test. A friend class is a statement that the two are one design in two types; if that is not true, the friendship is a shortcut around an interface that should exist. The second exercise pairs an `Account` with an `Auditor` that reads a counter no public function exposes.

---

## `operator<<` as a friend

The commonest friend in real code prints an object to a stream:

```
class Money {
    long long cents_;
public:
    explicit Money(long long cents) : cents_(cents) {}
    friend std::ostream& operator<<(std::ostream& os, const Money& m) {
        return os << m.cents_ / 100 << '.' << (m.cents_ % 100 < 10 ? "0" : "")
    }
};

std::cout << Money{1234} << '\n';    // 12.34
```

It cannot be a member of `Money`, because the left operand of `<<` is the stream, and a member `Money::operator<<` would put the `Money` on the left. Defined inside the class body with `friend`, it is a free function with access to `cents_` that the compiler finds through the `Money` argument. Module 11, lesson 3 covers stream operators properly — returning the stream, respecting its formatting state, reading a value back with `>>`; this is the shape to recognise, and the first exercise uses it.

---

## The interface a class publishes

The public section is the class's contract with the rest of the program. It should be **complete** — everything a caller legitimately needs is possible through it — and **minimal** — nothing that could be written from the other public functions (lesson 3's rule), no accessor added "in case", no mutator that merely sets a member. Put it first in the class body, because it is what readers come for; put the private data last, with its invariant in a comment. Then count: how many functions can modify the private members? Constructors, mutators, friends. That number is the surface across which the invariant can be broken, and a good class keeps it small.

---

## Pitfalls

**Public data with an invariant** — a `size_` anyone can set past the buffer's end.

**A setter for every member** — the invariant is unenforceable and the class is a verbose struct.

**Returning a non-const reference or pointer to a private member** — encapsulation in name only.

**Making a function a friend because it was convenient** — check whether the public interface already suffices.

**Assuming friendship is mutual or transitive** — `Auditor` being a friend of `Account` gives `Account` nothing, and `Auditor`'s own friends nothing.

**`operator<<` as a member** — the operands are the wrong way round; it must be a free function.

---

## Key takeaways

An invariant is what makes private data worth it: the constructor establishes it, mutators preserve it, readers assume it. Write it down.

Accessors earn their place by validating, deriving or protecting the representation; `getX` / `setX` for every member is a struct in disguise.

Never return a non-const reference or pointer to private state.

`friend` grants one function or class private access; it is granted by the class, not transitive, not inherited — and every friend widens the surface that can break the invariant.

`operator<<` is the archetypal friend: it must be a free function because the stream is the left operand.

Publish an interface that is complete and minimal; the private section, with its invariant, comes last.

---

EXERCISE 1

## Money that prints itself

`class Money` keeps `long long cents_` private with the invariant `cents_ ⩾ 0`, established by `explicit Money(long long cents)` (throws `std::invalid_argument` on a negative). Add `static Money parse(const std::string&)`, which accepts exactly `<digits>.<two digits>` (`0.25`, `15.00`) and throws `std::invalid_argument` for anything else — no sign, no missing dot, not one or three decimals. Add the const operations `add(const Money&)` and `times(int)`, each returning a new `Money`. Then write `operator<<` as a **friend free function** defined inside the class body — it must be a free function because the stream is the left operand — printing `<units>.<two digits>`.

Read `n` receipt lines `name quantity price`. For each, print `<name>: <quantity> x <price> → <line total>` using `operator<<` for both amounts, or `<name>: invalid price` when `parse` threw (the line is skipped). Finish with `total <sum of line totals>`.

**Input:** `n`, then `n` lines. **Output:** `n + 1` lines.

```
3
bolt 10 0.25
gear 2 15.00
widget 3 abc
```

→

```
bolt: 10 x 0.25 → 2.50
gear: 2 x 15.00 → 30.00
widget: invalid price
total 32.50
```

INPUT 1

```
3
bolt 10 0.25
gear 2 15.00
widget 3 abc
```

OUTPUT 1

```
bolt: 10 x 0.25 → 2.50
gear: 2 x 15.00 → 30.00
widget: invalid price
total 32.50
```

INPUT 2

```
2
pen 1 1.5
cap 4 0.05
```

OUTPUT 2

```
pen: invalid price
cap: 4 x 0.05 → 0.20
total 0.20
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <iostream>
#include <stdexcept>
#include <string>

class Money {
    long long cents_;   // invariant: cents_ ⩾ 0
public:
    explicit Money(long long cents) : cents_(cents) {
        if (cents < 0) throw std::invalid_argument("negative amount");
    }
    static Money parse(const std::string& text);
    Money add(const Money& other) const { return Money(cents_ + other.cents_); }
    Money times(int n) const { return Money(cents_ * n); }
    friend std::ostream& operator<<(std::ostream& os, const Money& m) {
        long long tail = m.cents_ % 100;
        return os << m.cents_ / 100 << '.' << (tail < 10 ? "0" : "") << tail;
    }
};

Money Money::parse(const std::string& text) {
    std::size_t dot = text.find('.');
    if (dot == std::string::npos || dot == 0 || text.size() - dot - 1 != 2) {
        throw std::invalid_argument("bad price");
    }
    for (std::size_t i = 0; i < text.size(); ++i) {
        if (i == dot) continue;
        if (!std::isdigit(static_cast<unsigned char>(text[i]))) throw std::invalid_a
    }
    long long units = std::stoll(text.substr(0, dot));
    int hundredths = std::stoi(text.substr(dot + 1));
    return Money(units * 100 + hundredths);
}

int main() {
    int n;
    std::cin >> n;
    Money total(0);
    for (int i = 0; i < n; ++i) {
        std::string name, price;
        int qty;
        std::cin >> name >> qty >> price;
        try {
            Money unit = Money::parse(price);
            Money line = unit.times(qty);
            std::cout << name << ": " << qty << " x " << unit << " -> " << line << '
            total = total.add(line);
        } catch (const std::invalid_argument&) {
            std::cout << name << ": invalid price\n";
```

```
        }
    }
    std::cout << "total " << total << '\n';
    return 0;
}
```

EXERCISE 2

## Account and Auditor

`class Account` holds a name, a balance with the invariant `balance_ ⩾ 0`, and a private `int rejected_` counting refused withdrawals — **no public function exposes it**. The public interface is `deposit(amount)` (refused, and nothing changes, when `amount ⩽ 0`), `bool withdraw(amount)` (refused and counted when `amount ⩽ 0` or above the balance), `balance()` and `name()`. Two friends complete it: the free function `bool transfer(Account& from, Account& to, long long amount)`, declared `friend` inside the class, which moves the amount or refuses (counted against `from`) when `amount ⩽ 0`, `from` cannot cover it, or the two are the same object; and `friend class Auditor`, whose `static void report(const Account&)` prints `<name>: balance <b> rejected <r>` — the only code outside the class that can see `rejected_`.

The first line names two accounts with opening balances. Then `n` commands: `deposit <name> <amount>`, `withdraw <name> <amount>`, `transfer <from> <to> <amount>`; print `ok` or `refused` for each. Finally audit both accounts in the order they were given.

**Input:** `nameA openingA nameB openingB`, then `n`, then `n` commands. **Output:** `n` result lines, then two audit lines.

```
ada 100 bob 20
5
withdraw ada 30
withdraw bob 50
transfer ada bob 70
transfer bob ada 500
deposit bob 5
```

→

```
ok
refused
ok
refused
ok
ada: balance 0 rejected 0
bob: balance 95 rejected 2
```

### INPUT 1

```
ada 100 bob 20
5
withdraw ada 30
withdraw bob 50
transfer ada bob 70
transfer bob ada 500
deposit bob 5
```

### OUTPUT 1

```
ok
refused
ok
refused
ok
ada: balance 0 rejected 0
bob: balance 95 rejected 2
```

### INPUT 2

```
x 0 y 0
2
withdraw x 1
transfer y y 0
```

### OUTPUT 2

```
refused
refused
x: balance 0 rejected 1
y: balance 0 rejected 1
```

### REFERENCE SOLUTION

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

class Account {
    std::string name_;
    long long balance_;   // invariant: balance_ ⩾ 0
    int rejected_ = 0;    // refused withdrawals and transfers out; not exposed
public:
    Account(std::string name, long long opening) : name_(std::move(name)), balance_(
    bool deposit(long long amount) {
        if (amount ⩽ 0) return false;
        balance_ += amount;
        return true;
    }
    bool withdraw(long long amount) {
        if (amount ⩽ 0 || amount > balance_) {
            ++rejected_;
            return false;
        }
        balance_ -= amount;
        return true;
    }
    long long balance() const { return balance_; }
    const std::string& name() const { return name_; }
    friend bool transfer(Account& from, Account& to, long long amount);
    friend class Auditor;
};

bool transfer(Account& from, Account& to, long long amount) {
    if (amount ⩽ 0 || from.balance_ < amount || &from == &to) {
        ++from.rejected_;
        return false;
    }
    from.balance_ -= amount;
    to.balance_ += amount;
    return true;
}

class Auditor {
public:
    static void report(const Account& a) {
        std::cout << a.name_ << ": balance " << a.balance_ << " rejected " << a.reje
    }
};

Account& pick(const std::string& name, Account& a, Account& b) {
    return name == a.name() ? a : b;
}
```

```
int main() {
    std::string nameA, nameB;
    long long openA, openB;
    std::cin >> nameA >> openA >> nameB >> openB;
    Account a(nameA, openA), b(nameB, openB);
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string cmd;
        std::cin >> cmd;
        bool ok = false;
        if (cmd == "transfer") {
            std::string from, to;
            long long amount;
            std::cin >> from >> to >> amount;
            ok = transfer(pick(from, a, b), pick(to, a, b), amount);
        } else {
            std::string who;
            long long amount;
            std::cin >> who >> amount;
            Account& acc = pick(who, a, b);
            ok = cmd == "deposit" ? acc.deposit(amount) : acc.withdraw(amount);
        }
        std::cout << (ok ? "ok" : "refused") << '\n';
    }
    Auditor::report(a);
    Auditor::report(b);
    return 0;
}
```

---

QUIZ

**1.** What is a class invariant, and who is responsible for it?

A. A `const` member; the compiler enforces it

B. A statement about the members that is true between operations; the constructor establishes it, mutators preserve it, readers may assume it

C. A rule that members must be initialised in declaration order

D. A comment the compiler checks with `static_assert`

**2.** Which accessor breaks encapsulation?

A. `long long balance() const`

B. `const std::vector<int>& items() const`

C. `std::vector<int>& items()` returning a non-const reference to a private member

D. `bool withdraw(long long amount)`

**3.** `Auditor` is a friend of `Account`, and `Inspector` is a friend of `Auditor`. What can `Inspector` access in `Account`?

A. Everything, because friendship is transitive

B. Only the public members of `Account` — friendship is neither transitive nor inherited

C. Only the private members that `Auditor` uses

D. Everything, because friendship is mutual

**4.** Why can `std::ostream& operator<<(std::ostream&, const Money&)` not be a member function of `Money`?

A. Members cannot return references

B. A member operator's left operand is the object it is called on, and the left operand of `<<` here is the stream, not the `Money`

C. Streams cannot be passed to members

D. It can, but only if declared `static`

**5.**

```
class Account {
    long long balance_;
public:
    friend bool transfer(Account& from, Account& to, long long amount);
};
```

What is `transfer`?

A. A member function called as `from.transfer(to, amount)`

B. A static member function

C. A free function with access to `balance_`, called as `transfer(a, b, amount)` and defined without `Account::`

D. A pure virtual function

**6.** A class makes every member private and adds a `getX()` / `setX()` pair for each one. What has been gained?

A. Full encapsulation

B. Nothing: every member can still be set to anything, so no invariant is enforced — it is a struct with extra steps

C. Thread safety

D. The compiler-generated copy is now safe

**7.** Which operation belongs as a `friend` rather than a member or a plain free function?

A. `bool isEmpty() const` when `size()` is public

B. `bool transfer(Account&, Account&, long long)` — it needs private access to two objects symmetrically

C. `void deposit(long long)`

D. `std::string name() const`

ANSWER KEY · ENCAPSULATION AND FRIENDS — THE INTERFACE A CLASS PUBLISHES

**1. B** The invariant is the contract that makes private data worth it: one check in the constructor covers the program, because no public operation may leave the object in a state that violates it.

**2. C** A non-const reference to private state lets any caller edit it without going through the class, so the invariant can be broken from outside. The const reference only allows reading; the others are operations.

**3. B** Friendship is granted by a class to specific functions or classes and stops there: a friend of a friend gets nothing, a derived class of a friend gets nothing, and `Account` gains no access to `Auditor`.

**4. B** A member `Money::operator<<` would be invoked as `money << stream`. To write `std::cout << m` the function must be free, and `friend` gives it the private access it usually needs.

**5. C** The `friend` line is a declaration of an ordinary non-member function plus a grant of access. It has no `this`, and its definition carries no class qualifier.

**6. B** Accessors earn their place by validating, deriving or protecting the representation. A setter for every member gives the invariant a second entrance and the readers no guarantee.

**7. B** A member would be lopsided (`from.transferTo(to, n)`) and a non-friend cannot reach `balance_`; a friend free function is symmetric and has the access. `isEmpty` needs no private access; the others modify or read one object.

LESSON 6 · 15 MIN

# Designing a class — headers, sources and a worked design

A class used from more than one file has to be *declared* wherever it is used and *defined* exactly once, and C++ makes that split visible: the class goes in a header, its member functions go in a source file, and the one-definition rule polices the boundary. This lesson covers that mechanism — what belongs in `date.h` versus `date.cpp`, include guards, why `inline` matters — and then walks one design from the first question to the last: choose the representation, state the invariant, write the constructor that enforces it, keep the interface small.

---

## The header/source split

A header holds what every user of the class needs in order to *compile* a call: the class definition — data members included, because the compiler must know the object's size and layout to create one — plus declarations of the member functions and of any free functions that go with the type. The source file holds the bodies. Both files, and every file that uses the class, `#include` the header:

```cpp
// date.h
#pragma once
#include <string>

class Date {
public:
    Date(int year, int month, int day);            // throws std::invalid_argu
    int year() const { return y_; }                // one-liners inline in the
    int month() const { return m_; }
    int day() const { return d_; }
    bool isLeapYear() const;
    std::string iso() const;
    static int daysInMonth(int year, int month);
private:
    int y_;
    int m_;
    int d_;     // invariant: 1 ≤ m_ ≤ 12 and 1 ≤ d_ ≤ daysInMonth(y_, m_)
};

bool operator==(const Date& a, const Date& b);
```

```cpp
// date.cpp
#include "date.h"
#include <format>
#include <stdexcept>

Date::Date(int year, int month, int day) : y_(year), m_(month), d_(day) {
    if (month < 1 || month > 12 || day < 1 || day > daysInMonth(year, month))
        throw std::invalid_argument("not a calendar date");
}

bool Date::isLeapYear() const { return (y_ % 4 == 0 && y_ % 100 != 0) || y_ %

int Date::daysInMonth(int year, int month) {
    static const int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    bool leap = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
    return month == 2 && leap ? 29 : days[month - 1];
}

std::string Date::iso() const { return std::format("{:04}-{:02}-{:02}", y_, m_

bool operator==(const Date& a, const Date& b) {
    return a.year() == b.year() && a.month() == b.month() && a.day() == b.day(
}
```

`g++ -std=c++20 -c date.cpp` and `g++ -std=c++20 -c main.cpp` produce two object files the linker joins (Module 1, lesson 2). Change a body in `date.cpp` and only that file recompiles; change the header and every file that includes it does — the argument for keeping headers small and stable. The study judge compiles a single file, so the exercises keep both halves together with a comment marking where the split would fall; the habit is the same.

---

## Include guards and `#pragma once`

`#include` pastes the header's text into the including file. If `main.cpp` includes `date.h` and also `calendar.h`, which itself includes `date.h`, the class definition appears twice in one translation unit — "redefinition of class Date". A guard makes the second inclusion empty:

```cpp
#ifndef DATE_H
#define DATE_H
// ... the header ...
#endif
```

`#pragma once` at the top does the same in one line. It is not in the standard, but every compiler in use honours it and it cannot be broken by a guard name copied from another header; this track uses it. One form or the other is mandatory on every header you write.

---

## The one-definition rule and `inline`

A *declaration* may be repeated; a *definition* of a function or variable must appear exactly once in the whole program, or the linker reports "multiple definition". Class definitions are the exception: the same class may be defined in every translation unit that includes its header, provided the definitions are identical — which they are, because they came from the same file. Member functions defined *inside* the class body are implicitly `inline`, and `inline` means "one definition per translation unit is allowed; the linker keeps one". So the header may carry the one-line readers above, but `Date::iso` with its body in the header would need the `inline` keyword — or, as here, the bodies belong in the `.cpp`. `inline static` data members (lesson 4) and templates (Module 12), which must be visible wherever they are used, live in headers by the same rule.

| Goes in the header | Goes in the source file |
|---|---|
| The class definition, private members included | Member function bodies longer than a line |
| Trivial members defined in the body (implicitly inline) | Free functions declared in the header |
| `inline` functions and `inline static` members | Helpers nobody outside needs (in an anonymous namespace) |
| Declarations of free functions | Static data member definitions in pre-C++17 style |
| Templates and `constexpr` functions | `#include`s that only the bodies need |

Two more header rules. Include only what the declarations need, so that including your header costs the user little. And never write `using namespace std;` in a header: it silently applies to every file that includes it (Module 1, lesson 3).

---

## A worked design

The `Date` above was not written top-down from its member list. It came from four decisions, taken in order.

**Choose the representation.** A date can be three integers, or one — days since some epoch. Three integers make `year()`, `month()` and `iso()` trivial and make "add 90 days" a loop over month lengths; a day count makes arithmetic and ordering trivial and makes printing a conversion. Which operations dominate decides it: a calendar type that mostly prints and compares takes the three integers; a scheduling type that mostly adds intervals takes the count. Either is right, and because the members are private, the choice can change later without touching a caller.

**State the invariant.** Three integers can hold `2024-02-30`; the invariant says they never will: the month is 1–12 and the day is 1 to that month's length in that year. It is written beside the members, and it decides what the constructor must check.

**Write the constructor to enforce it.** The constructor is the only entry; it validates and throws, so an existing `Date` is a real date and `iso()` never has to ask. The check calls `daysInMonth`, a static function (lesson 4) because it needs no `Date`. Copying a `Date` cannot break the invariant, so the compiler-generated copy is right and the class follows the rule of zero (Module 9, lesson 4).

**Keep the interface small.** Three readers, `isLeapYear`, `iso`, one static helper and `==`. No `setDay` — a date is a value, and a different date is a new object. `dayOfYear` and `nextDay` (the first exercise) earn their place as derived values (lesson 5); a setter for the month would not. Everything is `const` except the constructor; `==` is a free function because it needs only the readers (lesson 3), and C++20 could default it as `bool operator==(const Date&) const = default;` — memberwise comparison, which is correct here because the members are declared in significance order (Module 11, lesson 2).

---

## The checklist

Every class you write from here on answers the same questions, in the same order: what is the representation; what invariant do the members obey; does the constructor establish it, and is it `explicit` where one argument would convert; is every non-mutating member `const`; what does copying mean, and is the generated copy right (Module 9); is the public interface complete and minimal; are there friends, and does each one need to be; and which of it is in the header and which in the source. A class that answers all eight cleanly is one you can hand to another programmer with the header alone. The second exercise runs the list on a `Fraction` from a blank file.

## Pitfalls

| Mistake | What happens |
| --- | --- |
| A function body in a header without `inline` | "Multiple definition" as soon as two files include it. |
| A header without a guard | "Redefinition" when it is included twice through another header. |
| `using namespace std;` in a header | Every including file gets the pollution, and name clashes appear far from the cause. |
| Private members left out of the header "to hide them" | Cannot compile: the class definition must be complete to create an object. |
| Choosing the representation last | Every member function is written twice. |
| A setter on a value type | The invariant has a second entrance to defend. |

## Key takeaways

- The header carries the class definition (private members included), inline one-liners and declarations; the source file carries the bodies; every user includes the header.
- Guard every header (`#pragma once` or `#ifndef`); the ODR allows one definition of a function per program unless it is `inline`, and members defined in the class body are.
- Design in order: representation, invariant, a constructor that enforces it, a small `const`-correct interface.
- A value type has readers and derived operations, not setters; `==` and printing are free functions built on the readers.
- Run the eight-question checklist before declaring a class finished.

EXERCISE 1

## Date, dayOfYear and nextDay

The lesson's `Date` is given with its header half (the class definition) and source half (the bodies) marked by comments, as the judge compiles one file. Complete it with two derived operations that earn their place on a value type: `int dayOfYear() const` (1 for 1 January, 366 for 31 December of a

leap year) and `Date nextDay() const`, which returns a **new** `Date` — there is no setter — rolling over month and year ends, built through the constructor so the invariant is re-established rather than assumed. `iso()` formats with `std::format("{:04}-{:02}-{:02}", ...)`.

Read `n` lines `year month day`. For each, print `<iso> day <dayOfYear> next <iso of nextDay>`, or `<year>-<month>-<day> invalid` (the raw numbers, unpadded) when the constructor threw `std::invalid_argument`.

**Input:** `n`, then `n` lines. **Output:** `n` lines.

```
3
2024 2 28
2024 12 31
2023 2 29
```

→

```
2024-02-28 day 59 next 2024-02-29
2024-12-31 day 366 next 2025-01-01
2023-2-29 invalid
```

INPUT 1

```
3
2024 2 28
2024 12 31
2023 2 29
```

OUTPUT 1

```
2024-02-28 day 59 next 2024-02-29
2024-12-31 day 366 next 2025-01-01
2023-2-29 invalid
```

INPUT 2

```
2
2000 2 29
1900 2 29
```

OUTPUT 2

```
2000-02-29 day 60 next 2000-03-01
1900-2-29 invalid
```

REFERENCE SOLUTION

```cpp
#include <format>
#include <iostream>
#include <stdexcept>
#include <string>

// ---- date.h: what every user needs in order to compile a call ----
class Date {
public:
    Date(int year, int month, int day);            // throws std::invalid_argument
    int year() const { return y_; }
    int month() const { return m_; }
    int day() const { return d_; }
    bool isLeapYear() const;
    int dayOfYear() const;
    Date nextDay() const;
    std::string iso() const;
    static int daysInMonth(int year, int month);
private:
    int y_;
    int m_;
    int d_;     // invariant: 1 ⩽ m_ ⩽ 12 and 1 ⩽ d_ ⩽ daysInMonth(y_, m_)
};

// ---- date.cpp: the bodies ----
Date::Date(int year, int month, int day) : y_(year), m_(month), d_(day) {
    if (month < 1 || month > 12 || day < 1 || day > daysInMonth(year, month))
        throw std::invalid_argument("not a calendar date");
}

bool Date::isLeapYear() const { return (y_ % 4 == 0 && y_ % 100 != 0) || y_ % 400 ==

int Date::daysInMonth(int year, int month) {
    static const int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    bool leap = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
    return month == 2 && leap ? 29 : days[month - 1];
}

int Date::dayOfYear() const {
    int n = d_;
    for (int m = 1; m < m_; ++m) n += daysInMonth(y_, m);
    return n;
}

Date Date::nextDay() const {
    if (d_ < daysInMonth(y_, m_)) return Date(y_, m_, d_ + 1);
    if (m_ < 12) return Date(y_, m_ + 1, 1);
    return Date(y_ + 1, 1, 1);
}
```

```cpp
std::string Date::iso() const { return std::format("{:04}-{:02}-{:02}", y_, m_, d_);

// ---- main.cpp ----
int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int y, m, d;
        std::cin >> y >> m >> d;
        try {
            Date date(y, m, d);
            std::cout << date.iso() << " day " << date.dayOfYear() << " next " << da
        } catch (const std::invalid_argument&) {
            std::cout << y << '-' << m << '-' << d << " invalid\n";
        }
    }
    return 0;
}
```

EXERCISE 2

## Fraction from a blank file

Run the eight-question checklist on a `Fraction` and write it from the (nearly) blank starter. **Representation:** `long long num_` and `long long den_`. **Invariant:** `den_ > 0` and `gcd(|num_|, den_) == 1` (zero is stored as `0/1`), written as a comment beside the members. **Constructor:** `explicit Fraction(long long num, long long den)` throws `std::invalid_argument` when `den == 0`, moves the sign to the numerator and divides both by `std::gcd`; `explicit Fraction(long long whole)` delegates with denominator 1. **Interface**, all `const` and all returning a new `Fraction`: `add`, `subtract`, `times`, `dividedBy` (throws `std::domain_error` when the divisor is zero); readers `numerator()` and `denominator()`; a **friend** `operator<<` printing `<num>/<den>`, or just `<num>` when the denominator is 1. No setters: the compiler-generated copy is correct (rule of zero).

Read `n` lines `<lhs> <op> <rhs>`, where an operand is `a/b` or a whole number and `op` is `+`, `-`, `*` or `/`. Print the normalised result, `invalid` when an operand had a zero denominator, or `undefined` for division by a zero fraction.

**Input:** `n`, then `n` lines. **Output:** `n` lines.

```
4
1/2 + 1/3
2/4 * 3/5
5 - 7/2
1/2 / 0/1
```

→

```
5/6
3/10
3/2
undefined
```

INPUT 1

```
4
1/2 + 1/3
2/4 * 3/5
5 - 7/2
1/2 / 0/1
```

OUTPUT 1

```
5/6
3/10
3/2
undefined
```

INPUT 2

```
3
-1/2 + 1/2
3/-6 - 1/2
1/0 + 1/2
```

OUTPUT 2

```
0
-1
invalid
```

REFERENCE SOLUTION

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

class Fraction {
public:
    explicit Fraction(long long num, long long den) : num_(num), den_(den) {
        if (den == 0) throw std::invalid_argument("zero denominator");
        if (den_ < 0) {
            num_ = -num_;
            den_ = -den_;
        }
        long long g = std::gcd(num_ < 0 ? -num_ : num_, den_);
        num_ /= g;
        den_ /= g;
    }
    explicit Fraction(long long whole) : Fraction(whole, 1) {}

    long long numerator() const { return num_; }
    long long denominator() const { return den_; }

    Fraction add(const Fraction& o) const { return Fraction(num_ * o.den_ + o.num_ *
    Fraction subtract(const Fraction& o) const { return Fraction(num_ * o.den_ - o.r
    Fraction times(const Fraction& o) const { return Fraction(num_ * o.num_, den_ *
    Fraction dividedBy(const Fraction& o) const {
        if (o.num_ == 0) throw std::domain_error("division by zero");
        return Fraction(num_ * o.den_, den_ * o.num_);
    }

    friend std::ostream& operator<<(std::ostream& os, const Fraction& f) {
        os << f.num_;
        if (f.den_ != 1) os << '/' << f.den_;
        return os;
    }

private:
    long long num_;
    long long den_;   // invariant: den_ > 0 and gcd(|num_|, den_) == 1
};

Fraction parse(const std::string& token) {
    std::size_t slash = token.find('/');
    if (slash == std::string::npos) return Fraction(std::stoll(token));
    return Fraction(std::stoll(token.substr(0, slash)), std::stoll(token.substr(slas
}

int main() {
    int n;
```

```
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string lhs, op, rhs;
        std::cin >> lhs >> op >> rhs;
        try {
            Fraction a = parse(lhs);
            Fraction b = parse(rhs);
            Fraction r = op == "+" ? a.add(b) : op == "-" ? a.subtract(b) : op == "*
            std::cout << r << '\n';
        } catch (const std::invalid_argument&) {
            std::cout << "invalid\n";
        } catch (const std::domain_error&) {
            std::cout << "undefined\n";
        }
    }
    return 0;
}
```

---

QUIZ

**1.** Which of these belongs in `date.h` rather than `date.cpp`?

A. The body of `Date::iso()`

B. The class definition, private members included

C. Helper functions only the bodies use

D. `#include <format>`, which only `iso()` needs

**2.** A header defines `std::string Date::iso() const { ... }` outside the class body, without `inline`. Two source files include the header. What happens?

A. Nothing; headers may define functions

B. Linker error: multiple definition of `Date::iso() const`

C. Compile error in the second file

D. Only the first definition is used

**3.** `main.cpp` includes `date.h` and `calendar.h`, and `calendar.h` also includes `date.h`. Without a guard, what does the compiler report?

A. Nothing; identical definitions are merged

B. Redefinition of class `Date`

C. Undefined reference to `Date`

D. A warning about an unused include

**4.** Why is `using namespace std;` wrong in a header even if it is acceptable in a `.cpp`?

A. It does not compile in headers

B. It silently applies to every file that includes the header, causing name clashes far from the cause

C. It makes the header slower to compile

D. It is fine in headers; the rule is the other way round

**5.** The class body defines `int year() const { return y_; }` and the header is included from ten files. Why is there no multiple-definition error?

A. The linker discards duplicates for members

B. A member function defined inside the class body is implicitly `inline`

C. Getters are exempt from the one-definition rule

D. The compiler only compiles the first include

**6.** In what order does the lesson recommend taking the design decisions for a class?

A. Interface, then representation, then constructor, then invariant

B. Representation, then the invariant, then a constructor that enforces it, then a small const-correct interface

C. Constructor first, then members as needed

D. Write all getters and setters, then remove the unused ones

**7.** Why does the worked `Date` have `nextDay()` but no `setDay(int)`?

A. Setters are not allowed in C++20

B. A date is a value: a different date is a new object built through the constructor, so the invariant keeps one entrance

C. `setDay` would need to be a friend

D. Because `d_` is `const`

ANSWER KEY · DESIGNING A CLASS — HEADERS, SOURCES AND A WORKED DESIGN

**1. B** Users must see the whole class definition to create a `Date` — the compiler needs its size and layout — so it goes in the header. Bodies, private helpers and the includes only they need stay in the source file.

**2. B** Each translation unit gets its own definition, and the one-definition rule allows only one per program unless the function is `inline`. Members defined inside the class body are implicitly inline; an out-of-class definition in a header must say `inline` or move to the `.cpp`.

**3. B** `#include` pastes text, so the class definition appears twice in one translation unit. `#pragma once` or an `#ifndef` guard makes the second inclusion empty.

**4. B** A directive in a header is pasted into every includer, which cannot opt out. The pollution surfaces as ambiguities in files whose authors never wrote the directive.

**5. B** In-class definitions are inline by rule, and `inline` allows one definition per translation unit, of which the linker keeps one. That is what lets trivial members live in the header.

**6. B** The representation decides what is cheap; the invariant says what the members must obey; the constructor is the one entry that enforces it; only then is the interface chosen, and kept complete and minimal.

**7. B** A setter would be a second way to change the members, and it would have to repeat the validation. A derived operation that returns a new `Date` through the constructor gets the check for free.

CHECKPOINT · 25 MIN

# Checkpoint — Classes and objects

This checkpoint covers the whole module: structs, aggregates and designated initialisers, constructors and the member-initialiser list, `this` and const-correctness, static members, encapsulation with friends, and the header/source split behind a worked design.

**How it works.** Fourteen 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:

- What are the only two differences between `struct` and `class`?
- In what order are members initialised, and what does the order of the member-initialiser list change?
- Why does `Money m = 5;` fail when the constructor is `explicit`, while `Money m{5};` works?
- What type does `this` have inside a `const` member function?
- Why does a chaining mutator return `T&` rather than `T`?
- What goes wrong when an instance counter is incremented in the constructor but copying is left to the compiler?
- Which operand of `operator<<` makes it impossible to write as a member?

The three programs are a gradebook whose `Student` chains `add()` and validates every mark, a `Grid` with const and non-const `at()` and a half-turn written as one chain, and an assembly line whose instrumented parts show the construction and destruction order of a class with three members and a static count. Trace the instrumented one on paper before you run it.

---

EXERCISE 1

## A chaining gradebook

Write `class Student`: a name, a private `std::vector<int>` of marks with the invariant that every stored mark is 0–100, and a private count of rejected marks. `Student& add(int mark)` stores a valid mark or counts an invalid one, and **returns** `*this` so calls chain. Const readers: `name()`,

`average()` (0 when there are no marks), `best()` (the highest mark) and `rejected()`. Use `explicit` on the constructor.

Read `n` students, each as `name k m1 ... mk`. Print `<name> avg <average to two decimals> best <best or none> rejected <count>` per student, then `top <name>` for the highest average (the first one on a tie).

**Input:** `n` (at least 1), then `n` lines. **Output:** `n + 1` lines.

```
3
ada 3 90 80 101
bob 2 70 -5
cy 0
```

→

```
ada avg 85.00 best 90 rejected 1
bob avg 70.00 best 70 rejected 1
cy avg 0.00 best none rejected 0
top ada
```

INPUT 1

```
3
ada 3 90 80 101
bob 2 70 -5
cy 0
```

OUTPUT 1

```
ada avg 85.00 best 90 rejected 1
bob avg 70.00 best 70 rejected 1
cy avg 0.00 best none rejected 0
top ada
```

INPUT 2

```
2
x 1 100
y 1 100
```

OUTPUT 2

```
x avg 100.00 best 100 rejected 0
y avg 100.00 best 100 rejected 0
top x
```

REFERENCE SOLUTION

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

class Student {
    std::string name_;
    std::vector<int> marks_;   // invariant: every mark is 0..100
    int rejected_ = 0;
public:
    explicit Student(std::string name) : name_(std::move(name)) {}
    Student& add(int mark) {
        if (mark < 0 || mark > 100) {
            ++rejected_;
            return *this;
        }
        marks_.push_back(mark);
        return *this;
    }
    const std::string& name() const { return name_; }
    bool hasMarks() const { return !marks_.empty(); }
    double average() const {
        if (marks_.empty()) return 0;
        long long sum = 0;
        for (int m : marks_) sum += m;
        return static_cast<double>(sum) / static_cast<double>(marks_.size());
    }
    int best() const {
        int b = 0;
        for (int m : marks_) if (m > b) b = m;
        return b;
    }
    int rejected() const { return rejected_; }
};

int main() {
    std::cout << std::fixed << std::setprecision(2);
    int n;
    std::cin >> n;
    std::vector<Student> all;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int k;
        std::cin >> name >> k;
        Student s(name);
        for (int j = 0; j < k; ++j) {
            int mark;
            std::cin >> mark;
            s.add(mark);
```

```
        }
        all.push_back(s);
    }
    std::size_t top = 0;
    for (std::size_t i = 0; i < all.size(); ++i) {
        const Student& s = all[i];
        std::cout << s.name() << " avg " << s.average() << " best ";
        if (s.hasMarks()) std::cout << s.best(); else std::cout << "none";
        std::cout << " rejected " << s.rejected() << '\n';
        if (s.average() > all[top].average()) top = i;
    }
    std::cout << "top " << all[top].name() << '\n';
    return 0;
}
```

---

EXERCISE 2

## A grid with const and non-const at()

Write `class Grid` over a row-major `std::vector<int>` with `rows()`, `cols()`, and two overloads of `at(r, c)`: `int& at(int, int)` for writing and `const int& at(int, int) const` for reading, both throwing `std::out_of_range` for a bad cell. Two mutators return `Grid&` so they chain: `flipRows()` reverses the order of the rows and `flipCols()` reverses every row; a half-turn of the grid is the one chain `g.flipRows().flipCols()`. `long long sum() const` totals the cells. The free function `show(const Grid&)` prints the rows, space-separated, through the const `at`.

Read `R C`, the grid, then `q` commands: `set r c v` (writes through the non-const `at`; prints nothing), `get r c` (reads through a `const Grid&`; prints the value), `turn` (the chain; prints nothing), `show`, `sum`. A `set` or `get` on a bad cell prints `out of range`.

**Input:** `R C`, `R` rows of `C` integers, `q`, then `q` commands. **Output:** the lines the commands produce.

```
2 3
1 2 3
4 5 6
5
get 1 2
turn
show
set 0 0 9
get 0 0
```

→

```
6
6 5 4
3 2 1
9
```

### INPUT 1

```
2 3
1 2 3
4 5 6
5
get 1 2
turn
show
set 0 0 9
get 0 0
```

### OUTPUT 1

```
6
6 5 4
3 2 1
9
```

### INPUT 2

```
1 1
7
3
get 0 1
set 1 0 3
sum
```

### OUTPUT 2

```
out of range
out of range
7
```

### REFERENCE SOLUTION

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

class Grid {
    int rows_;
    int cols_;
    std::vector<int> cells_;   // row-major: cell (r, c) is cells_[r * cols_ + c]
    void check(int r, int c) const {
        if (r < 0 || r >= rows_ || c < 0 || c >= cols_) throw std::out_of_range("cel
    }
public:
    Grid(int rows, int cols) : rows_(rows), cols_(cols), cells_(static_cast<std::siz
    int rows() const { return rows_; }
    int cols() const { return cols_; }
    int& at(int r, int c) {
        check(r, c);
        return cells_[static_cast<std::size_t>(r) * cols_ + c];
    }
    const int& at(int r, int c) const {
        check(r, c);
        return cells_[static_cast<std::size_t>(r) * cols_ + c];
    }
    Grid& flipRows() {
        for (int r = 0; r < rows_ / 2; ++r) {
            for (int c = 0; c < cols_; ++c) std::swap(at(r, c), at(rows_ - 1 - r, c)
        }
        return *this;
    }
    Grid& flipCols() {
        for (int r = 0; r < rows_; ++r) {
            for (int c = 0; c < cols_ / 2; ++c) std::swap(at(r, c), at(r, cols_ - 1
        }
        return *this;
    }
    long long sum() const {
        long long s = 0;
        for (int v : cells_) s += v;
        return s;
    }
};

void show(const Grid& g) {
    for (int r = 0; r < g.rows(); ++r) {
        for (int c = 0; c < g.cols(); ++c) {
            if (c > 0) std::cout << ' ';
            std::cout << g.at(r, c);
        }
```

```
        std::cout << '\n';
    }
}

int main() {
    int R, C;
    std::cin >> R >> C;
    Grid g(R, C);
    for (int r = 0; r < R; ++r) {
        for (int c = 0; c < C; ++c) {
            int v;
            std::cin >> v;
            g.at(r, c) = v;
        }
    }
    const Grid& view = g;
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        std::string cmd;
        std::cin >> cmd;
        if (cmd == "set") {
            int r, c, v;
            std::cin >> r >> c >> v;
            try {
                g.at(r, c) = v;
            } catch (const std::out_of_range&) {
                std::cout << "out of range\n";
            }
        } else if (cmd == "get") {
            int r, c;
            std::cin >> r >> c;
            try {
                std::cout << view.at(r, c) << '\n';
            } catch (const std::out_of_range&) {
                std::cout << "out of range\n";
            }
        } else if (cmd == "turn") {
            g.flipRows().flipCols();
        } else if (cmd == "show") {
            show(view);
        } else {
            std::cout << view.sum() << '\n';
        }
    }
    return 0;
}
```

EXERCISE 3

## Assembly line, traced

`Part` prints `make <name>` when constructed and `scrap <name>` when destroyed. Write `class Widget` with three `Part` members declared in the order `frame_`, `motor_`, `shell_`, an `int serial_`, and an `inline static int built_` counting every widget assembled. The constructor takes a model name, initialises the parts as `<model>-frame`, `<model>-motor`, `<model>-shell` and the serial as `++built_` — all in the member-initialiser list — then prints `assembled #<serial> <model>`. The destructor prints `disassemble #<serial>`; the parts are then scrapped in reverse order by the language. Delete copying, so the count cannot drift, and publish the count through `static int built()`.

Commands, each in its own block followed by `--`: `one <model>` builds one widget; `two <a> <b>` builds `a` then `b` in the same block. Finish with `built <count>`. Trace it on paper first: the checkpoint says so.

**Input:** `n`, then `n` commands. **Output:** the trace, then the count.

```
1
two b c
```

→

```
make b-frame
make b-motor
make b-shell
assembled #1 b
make c-frame
make c-motor
make c-shell
assembled #2 c
disassemble #2
scrap c-shell
scrap c-motor
scrap c-frame
disassemble #1
scrap b-shell
scrap b-motor
scrap b-frame
--
built 2
```

### INPUT 1

```
1
two b c
```

### OUTPUT 1

```
make b-frame
make b-motor
make b-shell
assembled #1 b
make c-frame
make c-motor
make c-shell
assembled #2 c
disassemble #2
scrap c-shell
scrap c-motor
scrap c-frame
disassemble #1
scrap b-shell
scrap b-motor
scrap b-frame
--
built 2
```

### INPUT 2

```
1
one alpha
```

### OUTPUT 2

```
make alpha-frame
make alpha-motor
make alpha-shell
assembled #1 alpha
disassemble #1
scrap alpha-shell
scrap alpha-motor
scrap alpha-frame
--
built 1
```

### REFERENCE SOLUTION

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

struct Part {
    std::string name;
    explicit Part(std::string n) : name(std::move(n)) { std::cout << "make " << name
    ~Part() { std::cout << "scrap " << name << '\n'; }
};

class Widget {
    Part frame_;
    Part motor_;
    Part shell_;
    int serial_;
    inline static int built_ = 0;
public:
    explicit Widget(const std::string& model)
        : frame_(model + "-frame"), motor_(model + "-motor"), shell_(model + "-shell
        std::cout << "assembled #" << serial_ << ' ' << model << '\n';
    }
    ~Widget() { std::cout << "disassemble #" << serial_ << '\n'; }
    Widget(const Widget&) = delete;
    Widget& operator=(const Widget&) = delete;
    static int built() { return built_; }
};

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string cmd, a, b;
        std::cin >> cmd >> a;
        if (cmd == "one") {
            Widget w(a);
        } else {
            std::cin >> b;
            Widget x(a);
            Widget y(b);
        }
        std::cout << "--\n";
    }
    std::cout << "built " << Widget::built() << '\n';
    return 0;
}
```

---

## CHECKPOINT QUESTIONS

**1.** What are the only two differences between `struct` and `class`?

A. Structs cannot have constructors or destructors

B. Default member access and default inheritance access

C. Structs are values and classes are references

D. Classes can be templates; structs cannot

**2.**

```
class P {
    int a_;
    int b_;
public:
    P(int x) : b_(x), a_(b_ * 2) {}
};
```

What does `P p(5);` do?

A. Sets `b_` to 5 and `a_` to 10

B. Initialises `a_` first (declaration order) from an indeterminate `b_` — undefined behaviour, and `-Wreorder` warns

C. Compile error: the list is out of order

D. Sets both to 5

**3.** With `explicit Money(long long cents);`, why does `Money m = 5;` fail while `Money m{5};` works?

A. Braces are required for all class types

B. `= 5` is copy-initialisation, an implicit conversion that `explicit` forbids; `{5}` is direct initialisation

C. `5` is an `int`, not a `long long`

D. `explicit` constructors can only be called with parentheses

**4.** Inside a `const` member function of `Account`, what is `this`?

A. `Account*`

B. `const Account*`

C. `const Account&`

D. `Account* const`

**5.** Why must a chaining mutator return `Student&` rather than `Student`?

A. Returning by value is not allowed for members

B. Returning by value hands the next call a copy; the chain modifies temporaries and the original is unchanged

C. References are faster and the compiler requires them for chaining

D. `*this` cannot be copied

**6.** An instance counter is incremented in the constructor and decremented in the destructor; copying is left to the compiler. What goes wrong?

A. Nothing; the generated copy constructor also increments

B. The generated copy constructor copies members without running the body, so a copied object is not counted and the count goes negative on destruction

C. Compile error: a class with a destructor cannot be copied

D. The counter is reset on every copy

**7.** Which operand of `operator<<` makes it impossible to write as a member of `Money`?

A. The right operand, because it is `const`

B. The left operand: it is the stream, and a member's left operand is always the object of its class

C. Neither; it can be a member if declared `static`

D. The return value

**8.**

```
struct Config { std::string host; int port = 80; bool tls = false; };
Config c{.port = 443, .host = "x"};
```

What happens?

A. `c` is `{"x", 443, false}`

B. Compile error: designators must be in declaration order

C. `host` is ignored

D. `c` is `{"", 443, false}`

**9.** Which delegating constructor is valid?

A. `Money() : Money(0), cents_(0) {}`

B. `Money() : cents_(0), Money(0) {}`

C. `Money() : Money(0) {}`

D. `Money() { Money(0); }`

**10.** A `const` member function `mean()` recomputes a cached value when a flag says it is stale. What must the cache members be?

A. `static`

B. `mutable`

C. `constexpr`

D. `volatile`

**11.** Why is `inline static int count_ = 0;` preferred over `static int count_;` plus `int T::count_ = 0;` in a source file?

A. It is faster at run time

B. The in-class line is then the one definition, so the header is self-contained and no source file needs the definition

C. It makes the member `constexpr`

D. The two-part form does not compile in C++20

**12.** `friend class Auditor;` appears in `Account`. Which statement is true?

A. `Account` can now access `Auditor`'s private members too

B. Classes derived from `Auditor` also get access

C. Every member function of `Auditor` can access `Account`'s private members; nothing else follows

D. Friends of `Auditor` can access `Account`

**13.** A header has no `#pragma once` or include guard. When does the problem appear?

A. At link time, as a multiple definition

B. At compile time, as a redefinition, when one translation unit includes it twice through another header

C. Only at run time

D. Never; compilers deduplicate includes

**14.**

```
struct P {
    std::string n;
    explicit P(const char* s) : n(s) { std::cout << '+' << n; }
    ~P() { std::cout << '-' << n; }
};
struct W { P a{"a"}; P b{"b"}; };
int main() { W w; }
```

What is printed?

A. `+a+b-a-b`

B. `+a+b-b-a`

C. `+b+a-a-b`

D. `+a-a+b-b`

ANSWER KEY · CHECKPOINT — CLASSES AND OBJECTS

**1. B** Members and bases of a `struct` default to public, of a `class` to private. Everything else — constructors, value semantics, templates — is identical.

**2. B** The initialiser list's order is ignored; members are initialised as declared, so `a_(b_ * 2)` reads `b_` before `b_(x)` runs. Read the parameter instead: `a_(x * 2)`.

**3. B** `explicit` switches off implicit conversions from the argument type, and copy-initialisation is one. Direct initialisation with braces or parentheses names the constructor and is allowed.

**4. B** The trailing `const` makes the object const for the duration of the call, so `this` points to a `const Account` and members cannot be modified — except `mutable` ones.

**5. B** `s.add(1).add(2)` must call the second `add` on `s` itself. A by-value return creates a copy, so the second call modifies an object that dies at the semicolon.

**6. B** Generated special members do memberwise work only. Either count in a hand-written copy constructor or delete copying; the exercises take the second route.

**7. B** `std::cout << m` calls an operator whose left operand is `std::ostream`, which is not your class. So the operator is a free function, typically a `friend` for access to the private representation.

**8. B** C++20 designated initialisers must appear in the order the members are declared; `.host` after `.port` is an error rather than a reordering.

**9. C** The delegation must be the entire initialiser list. The last option compiles but creates and discards a temporary, leaving `cents_` uninitialised.

**10. B** `mutable` exempts a member from the const promise so a const function may write it. It is right here because the cache is not part of the object's logical value.

**11. B** Since C++17 an `inline` static data member is defined where it is declared; the linker keeps one copy across translation units. The two-part form still works but needs exactly one definition somewhere.

**12. C** Friendship is granted by the class, one way, and is neither transitive nor inherited: only `Auditor`'s own member functions gain access.

**13. B** `#include` is textual, so the class definition is pasted twice into one file. A guard makes the second inclusion empty; the linker is not involved.

**14. B** Members are constructed in declaration order ( `a` then `b` ) and destroyed in reverse ( `b` then `a` ) when `w` goes out of scope at the end of `main` .

---

CodeKairo · C++ study plan · Module 8: Classes and objects. Solve the exercises online to track your progress and earn XP.