CS 247

Software Engineering Principles

Chapter 2

Values vs. Entity Objects, Information Hiding

Victoria Sakhnini

Table of Contents

Introduction

The previous chapter showed how to make a user-defined type behave as comfortably as a built-in one. We chose how to construct it, compare it, and print it. Implicitly, we made a deeper choice as well: that two Rationals with the same numerator and denominator are the same thing. Two distinct variables holding 2/3 are interchangeable; copying one onto the other loses nothing.

That assumption is not always true. Two reservations for the same flight, on the same day, in the same seat, with the same passenger name are not the same reservation - they are a real-world problem. Two airplanes with identical specifications are not the same airplane. Two patients with identical medical histories are not the same patient. These objects have an identity that exists independently of their attributes, and treating them as values causes program bugs and real-world confusion.

This distinction - entity versus value - is the most important early design decision when modelling a domain. Every other choice (whether to allow copying, how to compare for equality, whether to expose mutators, whether to use pointers or values) follows from it. This chapter walks through both kinds of object, the design rules they impose, and how to combine them with a second important idea: information hiding, with a final look at how C++ lets us hide implementation behind a private pointer using the Pimpl idiom.

Entity Objects

An entity object is the computer embodiment of a real-world entity. Each instance corresponds to a specific real-world thing - a particular airplane, a particular customer, a particular reservation - and that correspondence is what makes the object meaningful. Two entity objects that happen to have identical attribute values are still distinct objects; they model two distinct real-world things.

Identity, Lifecycle, and Equality

Three properties characterize entity objects:

Identity. Entities have a unique identity that persists through time and through changes in attribute values. A Student entity might have a name, an email, and a student ID. Even if the name or email changes, the student ID - and the underlying identity - does not. Two students with the same name are not the same student.

Lifecycle. Entities are created, change state over time, and are eventually destroyed. A Reservation is booked, modified, paid for, fulfilled, and eventually archived. The object exists for the duration of that lifecycle, mirroring its real-world counterpart's lifecycle.

Equality by identity, not attributes. Two entity objects with identical attribute values are still two distinct entities. Two reservations booked separately, even for the same flight and seat, are two separate reservations. Asking "Are these two reservations equal?" without adding "equal in what respect?" does not yield a meaningful answer.

Examples of Entity ADTs

Several broad categories of real-world things naturally model as entities:

- Physical objects: airplane, runway, taxiway, terminal.
- People and accounts: passenger, booking agent, employee, student.
- Records: customer information, boarding pass, flight schedule, medical chart.
- Transactions: reservations, cancellations, receipts, payments.

In each case, an object in the program corresponds to a specific real-world counterpart - the airplane on gate 14, the passenger holding ticket 7B, the receipt with confirmation number XJ29F1. The job of the object is to track the state of that one specific thing, and operations on the object usually correspond to real-world events that affect it.

Value Objects

A value object simply represents a value. There is no real-world counterpart to which a specific instance corresponds - there is only the value itself, and any number of objects can carry that value at the same time. Two value objects with the same attribute values are considered identical: they represent the same value, and they are interchangeable.

Because they have no identity beyond their attributes, value objects are usually immutable: once constructed, their state never changes. If you need a different value, you do not modify the existing object; you create a new one. This mirrors how we treat values like integers and strings - you do not "change" the number 5 into 6, you simply use a different number.

Examples of Value ADTs

Several categories are naturally modelled as values:

- Mathematical types: rational numbers, polynomials, matrices, complex numbers, vectors.
- Measurements: size, distance, weight, mass, energy, duration.
- Other quantities: money, currency.
- Properties: colour, location, date, time.
- Restricted value sets: names, addresses, postal codes, phone numbers, number ranges.

An Address with street 123 Main, city Waterloo, postal code N2L 3G1 is the same address no matter how many times you construct it. If you need a different address, you build a new Address object - you do not "modify" the existing one to point to a different street.

Entity or Value? A Worked Example

The same domain often contains both kinds of object, and getting the distinction right is more important than getting any individual design detail right. Consider a video game of Euchre, a card game:

- Card - entity. The Five of Hearts in the deck is a specific card; if it is dealt to a hand, it has moved, not been duplicated. Two physical "Five of Hearts" objects in the same deck would be a bug.
- Deck - entity. There is one deck in the game, with a particular shuffled order and a position indicating the next card to deal. A second Deck object is a second deck, not "the same deck."
- Hand - entity. Each player has their own hand. Two players holding the same five cards would still have distinct hands; transferring a card from one hand to another is a real-world event.
- Player - entity. A specific person at the table.
- Score - value. The current score "12 to 7" is just a value. Two scores that happen to be equal are equal; there is no notion of "this 12-7 score" versus "that 12-7 score."

A second example from a university domain:

- Student - entity. A specific person enrolled in a program. Two students named "Alex Chen" are still two students.
- Student ID - value. The number 20945672 is a value. Two variables holding 20945672 have the same value; there is no distinction between "this 20945672" and "that 20945672."

Notice the pattern: the Student is the thing the university tracks; the Student ID is one piece of information attached to it. The same number can be used to look up the student, be transmitted in a message, or be compared to another ID, but it does not have a lifecycle or behaviour. It is just data.

> 💡 **Tip:** When designing a class, ask first: "Is each instance a specific real-world thing with its own lifecycle, or is it just a value that any number of variables could carry simultaneously?" The answer tells you which of the design rules in the next two sections to follow.

Designing Entity ADTs

Once you have decided that a class models an entity, several design choices follow almost automatically. The goal is to make sure objects of the class behave like the real-world entities they represent - and that means actively forbidding some of the operations that would be natural for a value type.

Construction reflects creation. Building a new entity object should correspond to a real-world creation event: a new reservation is booked, a new patient is admitted, or a new airplane enters service. There is no "default" entity, because the real world does not have a default airplane. Entity classes typically have no default constructor, and their constructors take whatever information is required to identify the new real-world thing.

Copying an entity is meaningless. What would it mean to copy a reservation? There are now two reservation objects in the program, but only one reservation in the real world. The program no longer reflects reality. Worse, operations on the two copies are uncoordinated: a cancellation through one copy leaves the other copy "still booked," and when one copy disappears, its history disappears with it. The standard discipline is to prohibit the copy constructor and the assignment operator outright.

```
class Reservation {
public:
    Reservation(/* identifying info */);

    // Forbid copy and assignment - copying entities corrupts the model.
    Reservation(const Reservation&)            = delete;
    Reservation& operator=(const Reservation&) = delete;

    // ... operations representing real-world events on this reservation ...
};
```

Prohibit type conversions. Implicit conversion between unrelated entity types is rarely meaningful - converting a Passenger to a Reservation, say, would tell the compiler to invent an entity that does not exist. The only conversions that make sense for entities are upcasts and downcasts within a class hierarchy, and even those are usually explicit via static_cast or dynamic_cast.

Avoid built-in equality. Asking "are these two entities the same entity?" by comparing attribute values is misleading. Two passengers with the same name and date of birth are still two passengers. If a comparison is meaningful, force the client to be explicit about which attribute it cares about:

```
// Don't do this - what does "equal" mean for two entities?
if (a1 == a2) { ... }

// Do this - the comparison is explicit about what is being compared.
if (a1->cost() == a2->cost()) { ... }
if (a1->id()   == a2->id())   { ... }
```

Computations on entities are not meaningful. What is "Reservation A plus Reservation B"? Or "Passenger X times 2"? Arithmetic on entities almost never has a sensible meaning, and overloading + or * on an entity type tends to produce code that compiles but does the wrong thing. Resist the urge.

⚠️ Overloading arithmetic operators on entity types invites confusion with pointer arithmetic on the corresponding pointers. If clients work with Reservation* (as they should, since entities are usually accessed

through pointers), then r1 + r2 might be either "the sum of two reservations" or "advance a pointer by an offset" depending on whether the operands are objects or pointers. Both look identical at the call site. The simplest defence is not to overload arithmetic on entity types at all.

operator< can be useful for identity-based ordering. If you need to put entities into an ordered container - a std::map keyed by passenger, say - define operator< to compare by some unique identifier (name, account number, ID). The ordering is for the container's sake, not for any meaningful arithmetic interpretation.

A clone operation may be useful. Sometimes the program legitimately needs a second entity that is "like" an existing one - a duplicate booking, a copy of a record for editing. Provide an explicit clone() method that constructs and returns a new entity. Making it explicit forces the caller to acknowledge that a new entity is coming into existence, with its own identity.

Entities are referred to by a pointer. Because copying is forbidden, you cannot pass entity objects around by value. The natural way to refer to an entity in code is through a pointer (or, when ownership matters, a smart pointer like std::unique_ptr or std::shared_ptr). Two pointers to the same entity are two references to the one real-world thing; two pointers to different entities clearly distinguish them.

💡 **Tip:** Inheritance and virtual functions are common in entity hierarchies. An Account base class with derived classes CheckingAccount and SavingsAccount, all manipulated through Account* pointers, is a natural design. The base class lets you write code that works on any account; the virtual functions let each subclass behave differently when asked to perform an operation.

Designing Value-based ADTs

Value ADTs play by almost the opposite rules. Where entity types forbid copying, value types embrace it. Where entity types reject equality on attributes, value types depend on it. The design choices, again, follow from the underlying notion that values have no identity beyond their attributes.

Equality and ordering are central. Two value objects with identical attributes are identical, full stop. The class needs a meaningful operator== and, where ordering makes sense, operator<, operator<=, and so on. For some value types, the equality check is straightforward (two integers are equal if they hold the same number); for others (matrices, large records), it can be nontrivial and expensive, but it is still well-defined.

Provide a copy constructor and assignment operator. Copies of a value are not a problem - they are the point. A second variable holding the same Rational value is exactly the same value used elsewhere. The copy and assignment operations should perform a deep copy: after the operation, the two objects represent the same value but share no underlying state, so modifying one (if mutation is allowed at all) does not affect the other.

Computations are meaningful, so overload arithmetic. You can sensibly add two rationals, scale a matrix, and concatenate two strings. The conventional arithmetic and comparison operators are usually the right way to expose these

computations to the client; r + s reads naturally and matches the mathematical operation. The chapter on operator overloading covers the mechanics.

Virtual functions and inheritance are uncommon. Value types describe values; they are rarely the kind of thing one extends with new subtypes. A Rational does not have specialized subclasses; a Date does not have a PolymorphicDate variant. Most value hierarchies are flat - one class, one set of operations, and any "variants" handled by storing different values rather than by deriving new types.

> 💡 **Tip:** A useful sanity check: if you can write down two attribute-identical instances and they should be treated as different, it is an entity. If two attribute-identical instances should be interchangeable, it is a value. The Rational 2/3 is interchangeable with another Rational 2/3 - value. A Reservation booked on Tuesday is not interchangeable with another Reservation booked on Wednesday, even if they share the same flight and seat entity.

Mutable vs. Immutable Objects

Mutability is closely tied to the entity/value distinction. Entity objects are mutable: a reservation changes state when payment is recorded, a patient's chart accumulates appointments, and an account's balance updates. Mutability is part of the lifecycle that makes *entities* entities.

Value objects, by contrast, are usually immutable: once constructed, their state cannot change. To "modify" a value, you construct a new value and assign it to the variable. This sounds restrictive, but it pays off in two ways. First, immutable objects are safe to share: there is no way for one part of the program to surprise another by mutating an object they both reference. Second, invariants only need to be checked once, at construction. Once the object exists, it cannot escape into an invalid state.

> 💡 **Tip:** Values (i.e., objects) of an immutable type do not change. But variables of an immutable type can still be assigned different values. Assignment changes which value the variable holds; the values themselves are inert. Integers work this way: you cannot "change 5 into 6," but you can certainly assign 6 to a variable that previously held 5.

Achieving Immutability in C++

C++ does not enforce immutability automatically - you have to design the class so that no operation can change an existing object's state. A class is immutable when all of the following hold:

1. No mutators. The class exposes only read or compute operations, never write operations.
2. All data members are private. With public fields, clients can write to them directly; with mutators or no mutators, the encapsulation is lost.

3. Member functions cannot be overridden. If a derived class can override a method, it can change what the method does or store extra mutable state - silently breaking the immutability promise of the base class. There are two ways to defend against this: make the class itself final (the preferred approach, sometimes called strong immutability), or make every method final.
4. Copy and assignment operations perform deep copies. Two variables holding the same value should not share underlying storage, since any "modification" to that storage would affect both.
5. All data members are themselves immutable, either primitive types or other immutable classes - otherwise, mutable fields leak the mutability they were supposed to prevent.

That last requirement is subtle and often missed: if your "immutable" class stores a std::vector<int>, clients can ask for the vector and modify it. Even if you marked your accessor const, returning a reference exposes the underlying mutable state. The next section explains how to handle this.

Protecting Mutable Fields

When a field is itself a mutable type, immutability requires two extra defences:

Copy on the way in. When the constructor receives a mutable parameter, take a copy of the parameter rather than storing a reference to it. Otherwise, the caller still holds a handle to the same underlying state, and any modifications they make leak into your supposedly-immutable object.

Copy on the way out. When an accessor returns a mutable field, return a copy rather than a reference. Otherwise, the caller can modify the returned object, and through it, the field. Returning a const reference helps somewhat - the caller cannot directly call non-const methods - but they can still bind a non-const reference elsewhere and circumvent the restriction in some patterns. A by-value return is the safer default.

Two patterns of bugs to watch for, both of which are easy to write by accident:

```
class AlmostImmutable final {
    std::vector<int> data_;
public:
    // BUG 1: storing a reference rather than copying.
    AlmostImmutable(const std::vector<int>& d): data_{d} {}
    //   This particular line is fine (the field is initialized by copy),
    //   but contrast with:

    // BUG 2: returning a reference.
    std::vector<int>& getData() { return data_; }     // wrong
    const std::vector<int>& getData() const { return data_; }  // better

    // The safe pattern: return a copy.
    std::vector<int> getDataCopy() const { return data_; }  // best
};
```

The first variant of `getData` lets a client write `obj.getData().push_back(99)`, mutating the field directly. The second variant blocks the obvious mutation but still hands out a reference into your private storage; the third variant gives the client a copy that they can do anything they like with, while your data remains untouched.

A Worked Immutable Class

Putting all the rules together, here is a small but genuinely immutable class:

```
class ImmutableObject final {
private:
    const std::string      name_;
    const int              age_;
    const std::vector<int> data_;
public:
    // Constructor takes const references and copies into const members.
    ImmutableObject(const std::string& name, int age,
                    const std::vector<int>& data)
        : name_{name}, age_{age}, data_{data} {}

    // Delete copy and move operations to ensure true immutability.
    ImmutableObject(const ImmutableObject&)            = delete;
    ImmutableObject& operator=(const ImmutableObject&) = delete;
    ImmutableObject(ImmutableObject&&)                 = delete;
    ImmutableObject& operator=(ImmutableObject&&)      = delete;

    ~ImmutableObject() = default;

    const std::string& getName() const noexcept { return name_; }
    int getAge() const noexcept { return age_; }

    // Return a copy to protect internal state from mutation.
    std::vector<int> getData() const { return data_; }

    // Alternative: return a const reference (cannot directly mutate
    // through the reference, but a copy is still safer).
    const std::vector<int>& getDataRef() const noexcept { return data_; }
};
```

Several details are worth noting. The class is declared final, so no derived class can sneak in mutable state or override the methods. Every data member is const and private, so neither the methods nor the outside world can write to them after construction. The copy and move operations are deleted, locking down the value of any specific instance. And the accessor that returns the vector returns it by value - clients get a copy they can do whatever they like with, while the private data remains untouched.

⚠️ Note 1: Deleting the copy and move operations makes the class strongly immutable but also harder to use - you cannot return an ImmutableObject from a function by value, store it in a std::vector, or pass it around easily. Most production immutable classes allow copying (which produces a new immutable value with the same attributes) but never expose mutation. The deleted-copy version above is closer to a singleton-like single-instance object; choose your immutability level based on how the class will be used.

Note 2: const members plus no mutators already prevent an existing object's state from changing. The deletion of copy/move adds two further guarantees. Move semantics is fundamentally incompatible with immutability, because a move leaves the source in a "valid but unspecified" state, it mutates the supposedly immutable source. Deleting move makes this explicit instead of letting it silently degrade to a copy. Note also that const members already cause the compiler to implicitly delete the copy and move assignment operators, so writing = delete on those mostly just documents what the language is already doing.

The copy constructor is the interesting case, since copying doesn't mutate the source and so doesn't strictly violate immutability. Deleting it is a design choice that pushes the type toward reference/entity semantics. Immutable objects are uniquely safe to share, no mutation means no aliasing hazards, so the natural way to use them is via shared pointers or references, not duplication. Allowing free copying encourages wasteful duplication and obscures whether identity matters.

Client code that tries to mutate one of these objects fails at compile time:

```
int main() {
    const std::vector<int> initialData{1, 2, 3};
    const ImmutableObject obj{"Alice", 30, initialData};

    std::cout << "Name: " << obj.getName() << '\n';
    std::cout << "Age: "  << obj.getAge()  << '\n';

    const auto dataCopy = obj.getData();
    for (const auto& v : dataCopy) std::cout << v << ' ';
    // All of the following fail to compile:
    //   obj.setName("Bob");            // no such method exists
    //   ImmutableObject obj2 = obj;    // copy constructor deleted
    //   auto obj3 = std::move(obj);    // move constructor deleted
}
```

The Singleton Design Pattern

Some types are meant to have exactly one instance in the entire program. A configuration manager that holds the application's settings; a hardware controller that talks to a piece of physical equipment; a database connection that is expensive to set up and pointless to duplicate. For these, allowing the client to construct a second instance would be a mistake - the second instance would either fight the first for the shared resource, or worse, silently allow the program to behave as if the resource were duplicated when it is not.

The Singleton design pattern enforces "exactly one instance" structurally, at the language level, so that the constraint cannot be violated by accident.

How the Pattern Works

The pattern has four ingredients:

1. A private constructor, so external code cannot create new instances.
2. A static member variable of the class type, holding the single instance, initialized once when the program starts.
3. A public static accessor that hands out a pointer (or reference) to the singleton.
4. Deleted copy and move operations, so the singleton cannot be duplicated through copying.

Here is a minimal singleton, modelled on a class Egg that allows exactly one Egg in the universe:

```
class Egg final {
private:
    static Egg instance_;   // the one and only Egg
    int value_;
    // Private constructor: only the class itself can build an Egg.
    explicit Egg(int val) noexcept : value_{val} {}
public:
    // Delete copy and move to prevent duplication.
    Egg(const Egg&)            = delete;
    Egg& operator=(const Egg&) = delete;
    Egg(Egg&&)                 = delete;
    Egg& operator=(Egg&&)      = delete;

    ~Egg() = default;
    // Static accessor for the singleton instance.
    static Egg* instance() noexcept { return &instance_; }

    int  val() const noexcept { return value_; }
    void setVal(int v) noexcept  { value_ = v; }
};

// Define and initialize the static instance with the value 42.
Egg Egg::instance_{42};
```

Several mechanisms cooperate here. The constructor is private, so a line like Egg e(7); fails - external code cannot invoke the constructor. The static instance_ is initialized exactly once, by the line outside the class definition, which the compiler arranges to run before `main()`. The accessor `instance()` returns a pointer to that single object. The deleted copy and move operations close the remaining loopholes - without them, clients could write `Egg e = *Egg::instance();` and get a second Egg via the implicitly-generated copy constructor.

> 💡 **Tip:** Why delete copy and move rather than make them private? Both have the same effect: the operations cannot be called from outside. But = delete is documentation: it tells anyone reading the code that the operation is intentionally forbidden, not an oversight. Making them private without a definition also works, but produces a less clear error message at the call site.

Using the Singleton

Client code can access the single instance through the static accessor:

```
int main() {
    // Both pointers refer to the SAME Egg.
    Egg* egg1 = Egg::instance();
    Egg* egg2 = Egg::instance();

    std::cout << "egg1 value: "   << egg1->val() << '\n';
    std::cout << "egg2 value: "   << egg2->val() << '\n';
    std::cout << "egg1 address: " << egg1 << '\n';
    std::cout << "egg2 address: " << egg2 << '\n';
    std::cout << "Same object? "  << (egg1 == egg2 ? "YES" : "NO") << '\n';

    // Modify through one pointer; the change is visible through the other.
    egg1->setVal(100);
    std::cout << "After setVal(100): egg1=" << egg1->val()
              << ", egg2=" << egg2->val() << '\n';

    // All of the following are prevented at compile time:
    //   Egg e = *Egg::instance();  // copy constructor deleted
    //   Egg e2(42);                // constructor is private
    //   *egg1 = *egg2;             // assignment operator deleted
}
```

Notice what the output of this program will show: `egg1` and `egg2` have identical addresses, because they point to the same underlying object; a modification through one is immediately visible through the other. That is exactly the property the pattern is designed to guarantee.

⚠️ The singleton pattern is easy to overuse. It is appropriate when "exactly one" is genuinely a property of the domain (one hardware device, one shared configuration), but it introduces global state, which makes code harder to test and reason about. Before reaching for a singleton, ask whether the same goal could be achieved by passing the object explicitly to the code that needs it.

💡 **Tip:** The version shown above uses a static member object initialized at program start (eager initialization). Another common variant returns &instance_ via a static local variable inside the accessor, which delays construction until the first call (lazy initialization, sometimes called the Meyers Singleton). For now, the eager version is enough; we will see the lazy form later when we cover static initialization order.

Hiding the Implementation

We have spent the chapter making classes well-behaved on the outside - entities behave like entities, values like values, and immutables like immutables. But there is one part of every class so far that does not hide anything: the class definition itself, written in the header file, lays bare the private data members for every client to see.

Why the Implementation is Exposed

Recall a typical class definition from the previous chapter:

```
class Rational {
private:
    int num_;
    int denom_;
public:
    Rational() noexcept : num_{0}, denom_{1} {}
    explicit Rational(int n) noexcept : num_{n}, denom_{1} {}
    Rational(int num, int denom) : num_{num}, denom_{denom} {
        if (denom == 0) {
            throw std::out_of_range("Denominator cannot be zero");
        }
    }
    // ...
};
```

The private fields `num_` and `denom_` appear in the header file. Client code that includes rational.h sees them - not because the client can use them (they are private, after all), but because the compiler needs them. When the client writes Rational r, the compiler needs to know how big a Rational is so it can allocate the right amount of memory. The only way to compute that size is to look at the class definition, including the private parts.

Private, in other words, means "clients cannot name these members," not "clients cannot see them." The information is in the header, visible to anyone who opens the file.

The Cost of Exposure

Three distinct costs follow from this exposure:

A larger slice of code for clients to wade through. A client trying to understand how to use Rational reads the header and is greeted with implementation details - private fields, helper methods marked private, perhaps friend declarations. None of this is relevant to using the class, but it is all there, inviting confusion about what is important.

Accidental dependence on the representation. Once the layout is visible, clever clients sometimes start depending on it - accessing fields directly through pointer arithmetic, casting between layouts, or simply assuming a particular memory size in serialization code. Even where no one writes such code intentionally, debuggers and tools may surface the field names and encourage clients to read them. The class is no longer free to evolve its representation.

Mandatory recompilation. The most concrete and frustrating cost. When you change a class's implementation - adding a private field, renaming an internal helper, switching from int to long long for an internal counter - the header file changes. Every translation unit that includes the header must be recompiled. For a small project, this is barely noticeable; for a large project, where the header is included by hundreds of files, a small change to a private detail can trigger a multi-minute rebuild. This is sometimes called the fragile base-class problem: a base class is "fragile" in the sense that changes to it cascade through every dependent.

> 💡 **Tip:** Some projects also have strategic reasons to hide implementation. A library may not want to disclose its internal algorithms because they represent proprietary intellectual property. Cryptographic code may not want to expose hints about key layouts or buffer sizes that could help attackers. A library being shipped into a "hostile" environment, where some clients are willing to bypass access modifiers with casts, benefits from making its implementation literally absent from the header. The PImpl idiom, covered next, addresses all of these concerns.

The PImpl Idiom

PImpl - short for "Pointer to Implementation," sometimes called the handle/body or Cheshire-cat pattern - is the standard C++ technique for actually hiding implementation. The idea is simple: the class's public type, the one clients see, contains nothing but a single pointer to an implementation struct. The implementation struct itself is defined only in the .cpp file, where the compiler can see it. Everything that used to live in the private section of the class - the data members, the helpers, the invariants - moves into that struct.

From the client's perspective, every Rational object is just a pointer. The compiler can compute the size of a Rational (the size of one pointer) without ever seeing what the pointer points to. When the implementation changes, only the .cpp file is recompiled; the header is unchanged, so every client that uses Rational remains untouched.

The metaphor of the Cheshire Cat captures it: everything about the implementation disappears, except for the smile (the pointer) that lets you find it again.

The Header File

The header declares the class, declares (but does not define) the nested implementation struct, and holds a pointer to it:

```
// rational.h

#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream>
#include <memory>


class Rational {
private:
    // Forward declaration only - Impl is incomplete here.
    struct Impl;

    // std::unique_ptr<Impl> automatically manages the Impl object's
    // lifetime; modern C++ best practice over a raw Impl*.
    std::unique_ptr<Impl> pImpl_;

public:
    explicit Rational(int num = 0, int denom = 1);

    // Destructor MUST be declared here but defined in the .cpp file,
    // where Impl is a complete type. (More on this below.)
    ~Rational();

    Rational(const Rational& other);
    Rational& operator=(const Rational& rhs);
    Rational(Rational&& other) noexcept;
    Rational& operator=(Rational&& rhs) noexcept;

    int  getnum()   const;
    int  getdenom() const;
    void setnum(int n);
    void setdenom(int n);
};

// Non-member operators, as before.
std::ostream& operator<<(std::ostream& out, const Rational& r);

#endif
```

Look at what is missing: there are no `int num_;` or `int denom_;` fields. The class definition contains only a forward declaration of a nested type `Impl` and a `unique_ptr` to it. A client reading this header sees the interface and

nothing else - neither the choice of two ints versus one double, nor the names of the helper methods, nor any other private detail.

⚠ When a class manages a `std::unique_ptr<Impl>` with `Impl` being an incomplete type, the destructor cannot be defaulted in the header. The compiler-generated default destructor would try to delete the `unique_ptr` inline at every place the class is destroyed, which requires the full Impl type to be visible. The standard fix is to declare the destructor in the header (`~Rational();`) and define it in the .cpp file (`Rational::~Rational() = default;`) - the .cpp is where Impl is fully defined, so the default destructor works there. The same applies to the move operations.

The Implementation File

The .cpp file defines the `Impl struct`, the constructor, the special member functions, and all the operations. Because Impl is a nested type, every reference to it from outside the class definition must use scope resolution: `Rational::Impl`.

```
// rational.cpp
#include "rational.h"
#include <stdexcept>
#include <numeric>

// Full definition of the nested implementation struct. This is invisible
// to clients - they only ever see the forward declaration in rational.h.
struct Rational::Impl {
    int num_;
    int denom_;
    Impl(int num, int denom) : num_{num}, denom_{denom} { reduce(); }

    void reduce() {
        const int g = std::gcd(num_, denom_);
        if (g != 0) { num_ /= g; denom_ /= g; }
        if (denom_ < 0) { num_ = -num_; denom_ = -denom_; }
    }
};

Rational::Rational(int num, int denom)
    : pImpl_{std::make_unique<Impl>(num, denom)} {
    if (denom == 0) {
        throw std::invalid_argument("Denominator cannot be zero");
    }
}

// Destructor defined here, where Impl is a complete type.
Rational::~Rational() = default;
```

```
// Copy constructor: allocate a new Impl that is a copy of the other one.
// This is a DEEP copy - the two Rationals do not share state.
Rational::Rational(const Rational& other)
    : pImpl_{std::make_unique<Impl>(other.pImpl_->num_,
                                    other.pImpl_->denom_)} {}

Rational& Rational::operator=(const Rational& rhs) {
    if (this != &rhs) {
        pImpl_->num_   = rhs.pImpl_->num_;
        pImpl_->denom_ = rhs.pImpl_->denom_;
    }
    return *this;
}


// Move operations can be defaulted, but only where Impl is complete.
Rational::Rational(Rational&& other) noexcept = default;
Rational& Rational::operator=(Rational&& rhs) noexcept = default;

int  Rational::getnum()   const { return pImpl_->getnum_(); }
int  Rational::getdenom() const { return pImpl_->denom_; }

void Rational::setnum(int n) {
    pImpl_->num_ = n;
    pImpl_->reduce();
}

void Rational::setdenom(int n) {
    if (n == 0) {
        throw std::invalid_argument("Denominator cannot be zero");
    }
    pImpl_->denom_ = n;
    pImpl_->reduce();
}
```

Two things to notice. First, the actual data layout (two ints plus a helper method) lives entirely in this file. Recompiling rational.cpp does not require any client to be recompiled, because the public header is unchanged. Switching to one double, or to arbitrary-precision integers, or to caching the reduced form alongside the unreduced form - any of these changes is confined to this file.

Second, the scope of every Rational member function is the same as before (`Rational::getnum`, `Rational::setnum`, etc.); only the path to the data has an extra level of indirection. Where the previous chapter's implementations wrote `num_` directly, they now write `pImpl_->num_`. The interface to clients is unchanged.

Drawbacks of PImpl

PImpl is not free. It buys you implementation hiding, but it costs you something in return:

More work for the implementer. Every member function has one extra layer of indirection (pImpl_->...). Copy and move semantics have to be written explicitly rather than defaulted in the header. The constructor and destructor have to allocate and free the Impl. None of this is hard, but it is work.

No protected members. PImpl hides the implementation from subclasses as well, since subclasses cannot see the Impl struct. If you have a class hierarchy where derived classes legitimately need access to base-class data, PImpl is awkward at best.

Slightly harder to read. Some readers expect to see the data layout in the header. With PImpl, they have to open the .cpp file to learn what the class actually stores, which is exactly the point, but it is also a small extra cost for anyone debugging or extending the class.

Small runtime cost. Every access to a member is an indirection through pImpl_, which costs a pointer dereference. For most code, this is negligible, but in performance-critical inner loops (or for classes that are accessed millions of times per second), it is measurable. Virtual function dispatch through PImpl is doubly indirect, which compounds the cost.

In practice, the right rule of thumb is: use PImpl for classes whose implementation you actually want to change without breaking client code, or whose implementation contains genuinely sensitive details. Do not reach for it as a default - for many classes, the small win in compile times does not justify the extra implementation complexity.

Summary

Two cross-cutting design distinctions emerged in this chapter, and they shape almost every design decision a C++ class faces:

- Entity vs. value. Entity objects model specific real-world things with identity, lifecycle, and behaviour; they are referred to by pointer, do not support copying, and are usually mutable. Value objects represent values; they support copying and equality on attributes, and are usually immutable.
- Mutable vs. immutable. Mutable objects can change state via mutators; immutable objects cannot. Immutability requires more than removing mutators - private fields, no overrides (preferably final), deep copies, and either primitive members or copies-on-input-and-output for mutable members.
- The Singleton pattern enforces exactly one instance via a private constructor, a static instance, a public accessor, and deleted copy and move operations.
- Information hiding via PImpl moves all private members into a struct defined only in the .cpp file, leaving the header with a single pointer. This decouples client compilation from implementation changes, but costs a layer of indirection and some implementation complexity.

None of these patterns is universal: each is the right answer to a specific question. The skill is recognizing which question your class is answering, and then applying the matching pattern systematically.

Practice Problems

1.

Immutable Range with PImpl.

Implement an immutable Range abstract data type that represents a closed integer interval [start, end]. The internal representation of the range must be hidden using the PImpl idiom. The class definition may only contain a pointer to an incomplete implementation type; all data members must reside in the implementation file.

A Range object is immutable. Once constructed, the start and end values cannot be modified. Any operation that conceptually changes a range must return a new Range object instead of modifying the existing one. A range is valid only if start <= end; if an invalid range is created, the constructor must throw an exception with the message "Invalid range".

The class must support copy construction, copy assignment, move construction, and move assignment. Copying a range must create an independent copy of its implementation. Moving a range must transfer ownership of the implementation and leave the source object in a safe, destructible state. Proper dynamic memory management is required.

The following operations must be supported:

• Output of a range using operator<<, displaying the range in the form [start, end].

• Equality comparison using operator==, which returns true if two ranges have the same start and end values.

• An intersect operation that returns a new Range representing the intersection of two ranges. If the two ranges do not overlap, the operation must throw an exception with the message "No intersection".

2.

Money: an immutable value type.

Design an immutable Money class that holds an amount and a currency code (e.g., "CAD", "USD", "EUR"). The amount is stored in the smallest unit of the currency - cents for CAD/USD, etc. - to avoid floating-point rounding issues.

Requirements:

• Constructor takes the integer amount in the smallest units and the three-letter currency code. Reject anything that is not exactly three uppercase letters.

• Provide accessors getAmount() and getCurrency().

• Overload + and - to combine two Money values. Throw if the currencies do not match - adding USD to EUR has no meaning without a conversion rate. Operations on Money return new Money objects; the operands are unchanged.

• Overload * to scale a Money by an integer factor (e.g., 5 * pricePerItem).

• Overload == and !=. Two Money values are equal if and only if their amount AND currency match.

• Overload <<: print as "12.34 CAD" (amount divided by 100, two decimal places, then the code).

Verify your design satisfies all five rules for immutability from the chapter. Identify which rules each piece of your design satisfies.

3.

Patient: an entity ADT.

Design an entity ADT Patient representing a hospital patient. Each patient has a unique numeric patient ID (assigned at construction by an internal counter), a name, and a list of appointments. An Appointment is itself an entity, with a date, a doctor's name, and an outcome string ("scheduled", "completed", "cancelled").

Requirements:

• A Patient cannot be copied or assigned. Document why in a comment.

• Patient provides scheduleAppointment(Appointment*), cancelAppointment(int appointmentId), and listAppointments() - the last printing a summary of every appointment for the patient.

• Two patients with the same name are NOT equal; provide a sameId(const Patient& other) method that compares patient IDs explicitly.

• Provide a clone() method that creates a new Patient (with a new ID) carrying the same name and the same appointments. Decide whether the cloned patient should share appointments with the original or have independent copies, and justify your decision.

• Implement an inheritance hierarchy where InpatientPatient and OutpatientPatient extend Patient. Each subclass adds one extra field (admission date for inpatient, follow-up clinic for outpatient) and one method specific to its kind. Client code should manipulate all patients through Patient* base pointers.

4.

Singleton ConfigManager with PImpl.

Implement a singleton ConfigManager class that stores key-value configuration settings (keys are strings, values are strings). Combine two of the patterns from this chapter:

• Singleton: ConfigManager has a private constructor, deleted copy/move, and a static accessor ConfigManager::instance() that returns the single instance.

• PImpl: the actual storage (a std::unordered_map<std::string, std::string>) lives in a private Impl struct defined only in the .cpp file. Clients of config_manager.h see no map.

The interface should support:

• get(const std::string& key) const, returning the value or throwing std::out_of_range if the key is absent.

• set(const std::string& key, const std::string& value), which inserts or updates.

• has(const std::string& key) const, returning bool.

• remove(const std::string& key), which erases the key if present.

• Output of the entire configuration using operator<<.

Test your implementation in a small main() that loads a few key-value pairs, queries them, and prints the whole configuration. Verify that constructing a second ConfigManager fails to compile.

Reflection question to answer in a comment at the end of your file: this class is mutable (set and remove change its state), but it is also a singleton. Does that combination make sense, or does it create the kind of "global mutable state" the chapter warned about? Under what conditions is it acceptable?

5.

Refactoring exercise - converting an existing class to PImpl.

You are given the following class, written without PImpl:

```
class Image {
public:
   Image(int width, int height);
   Image(const Image& other);
   Image& operator=(const Image& other);
   ~Image();
   int  width()  const;
   int  height() const;
   void setPixel(int x, int y, uint32_t colour);
   uint32_t getPixel(int x, int y) const;
   void fill(uint32_t colour);
private:
   int      width_;
   int      height_;
   uint32_t* pixels_;   // heap-allocated array of width*height pixels
   void bounds_check(int x, int y) const;
};
```

(a) Identify everything in this class that a client sees by including the header, but that is none of the client's business.

(b) Rewrite the header file using the PImpl idiom. Be precise about which functions need to be declared in the header but defined in the .cpp file, and explain why.

(c) Rewrite the .cpp file accordingly. Take care of: the Impl struct definition; the constructor allocating an Impl with a heap-allocated pixel array; the destructor freeing both; deep-copy semantics in copy construction and copy assignment; move semantics that transfer the Impl pointer; and forwarding every public method through pImpl_.

(d) Suppose six months from now we want to change the pixel storage from uint32_t* to std::vector<uint32_t>, and to keep an extra cached field for "average colour" computed on demand. With the original code, how many files have to recompile? With the PImpl version, how many? Why does the difference matter?

6.

Stretch: invariant-preservation puzzle.

Below is the skeleton of an "immutable" PostalCode class. Find at least three distinct ways a client could violate its supposed immutability, and propose a fix for each.

```
class PostalCode {
public:
   PostalCode(std::string code) : code_{code} {}
   std::string& getCode() { return code_; }
   void setCodeIfEmpty(const std::string& c) {
      if (code_.empty()) code_ = c;
   }
protected:
   std::string code_;
};
```

For each violation:

- Describe the client code that exploits it.
- Explain which of the chapter's five rules for immutability is being broken.
- Show the corrected version of the offending line.

Then write the fully correct, strongly immutable PostalCode class.