CS 247

Software Engineering Principles

Chapter 1

ADT Design

Victoria Sakhnini

Table of Contents

Introduction

A central goal of object-oriented programming is to let us define new types that feel as natural to use as the built-in ones. We want a Rational, a Date, or a Subscription to behave like an int - variables of these types should have well-defined values, support meaningful operations, and be checked by the compiler the same way the built-in types are. The mechanism C++ gives us for this is the abstract data type, or ADT, implemented as a class. This chapter walks through how to design ADTs that are safe, evolvable, and pleasant to use, using a Rational number class as a running example.

Abstract Data Types

An abstract data type (ADT) is a user-defined type that bundles together two things:

- the range of values that variables of that type can hold, and
- the operations that manipulate variables of that type.

A well-designed ADT is implemented in such a way that client code knows the type only through its interface - the public set of operations and the legal values they accept. The data representation and the implementation of those operations are hidden behind that interface. This separation is what makes the type abstract: the client programmer can use a Rational without knowing whether it is stored as a numerator/denominator pair, a string, or something more exotic, and the implementer can change that representation later without breaking any client code.

Why ADTs?

There are four interrelated reasons to invest in ADTs rather than working directly with built-in types or plain structs:

Safety. By controlling how values are constructed and modified, the ADT can guarantee that every variable of its type holds a sensible value. The constructor establishes a valid value, and every operation is responsible for leaving the object in a valid state. A Rational object never has a zero denominator. A Date object never has month 13. These constraints, called invariants, become impossible to violate from outside the class.

Tool-supported safety. The compiler enforces the restrictions you put on the type. If you mark a method `const` and then try to assign to a member inside it, you get a compile error - not a bug discovered at runtime. Mistakes that would otherwise become silent bugs are caught as type errors before the program ever runs. The compiler becomes a collaborator in maintaining your invariants.

Evolvability. Because the client code depends only on the interface, you can change the internal representation later without rewriting the clients. You could swap a numerator/denominator pair for a double, switch to arbitrary-precision integers, or add caching - as long as the public operations behave the same, client code keeps compiling and running.

Productivity. Once the ADT enforces its invariants, every other piece of code that uses it can stop checking them. Range checks live inside constructors and mutators, not scattered across the program. The client programmer needs to learn

only the interface, not the internal representation. Both effects scale: the larger the program, the more leverage you get from a well-defined ADT.

> 💡 **Tip:** Aggressive use of ADTs: anytime you have a variable with a limited range of legal values, define your own type for it. Anytime you want to restrict the set of operations that can be applied to a variable, define your own ADT. Investing a few minutes in design saves hours of debugging later.

ADT vs typedef

It is worth distinguishing an ADT from a simple typedef (or its modern equivalent, using). A typedef gives an existing type a new name, but adds nothing - no new operations, no restrictions on values, no compiler-enforced safety. Writing

```
using Age = int;
```

lets you write Age in your code, but does nothing to stop you assigning -5 to an Age variable or multiplying two Ages together. An ADT, by contrast, is a real type. The compiler treats it as distinct from int; only the operations you define are legal, and the constructor decides what counts as a legal value. Reach for an ADT whenever a typedef would feel like a lie about the actual constraints.

Outside-In Design

A natural temptation when designing a class is to start with the data - what fields it has - and work outward, exposing operations as the implementation suggests them. This is an inside-out design, and it tends to produce classes whose interfaces match their implementation rather than their users' needs.

Outside-in design inverts the process. You start from the environment of the application - the code that will use this class - and work inward:

1. Develop use scenarios first: write sample client code that exercises the class the way you want it to be used. This is "test code" that does not compile yet, because the class does not exist.
2. Design the interface to satisfy those scenarios: what public operations does the class need? What are their signatures? What invariants do they preserve?
3. Implement the class so that the interface works. Internal data structure choices fall out of the operations you committed to in step 2.
4. Refine the test code in the validation phase, once you can compile and run the class for real.

During the interface-design step, you look across the set of operations and ask whether they are consistent (similar things named similarly), complete (you can do everything a user would reasonably want), and encapsulated (each operation hides one decision). The result is a class designed for its users, not for its implementation.

> 💡 **Tip:** Outside-in design keeps you honest. If you cannot write a convincing sample client code before implementing the class, you do not yet understand the use cases well enough to design the class.

A Running Example: The Rational ADT

Throughout the rest of this chapter, we develop a single example: a Rational class representing a rational number as a numerator/denominator pair. Each design idea - constructors, member initialization lists, accessors and mutators, const correctness, operator overloading - is introduced in the context of this one class, so that by the end of the chapter, you have seen a complete, idiomatic C++ ADT built piece by piece.

Sample Client Code

Outside-in design tells us to start with sample client code. Here is what we would like to be able to write:

```
int main () {
    Rational r, s;
    cout << "Enter rational number (a/b): ";
    cin >> r;
    cout << "Enter rational number (a/b): ";
    cin >> s;
    Rational t(r + s);
    cout << r << endl;
    cout << s << endl;
    cout << t << endl;
    t.setnum(100);
    cout << t << endl;
    cout << r << " + " << s << " = " << r + s << endl;
    Rational m(2);
    cout << m << endl;
    m = m + 5;
    cout << m << endl;
    m++;
    cout << m << endl;
    m += 10;
    cout << m << endl;
    r = s = m;
    cout << s << endl;
    cout << m << endl;
    return 0;
}
```

Reading through this snippet, we can already enumerate everything the class will need to support:

- default construction (`Rational r, s;`) and construction from a single int (`Rational m(2);`),
- input and output streaming (`cin >> r` and `cout << r`),
- addition of two Rationals (`r + s`), and mixed addition with an int on either side (`m + 5`),
- a mutator `setnum` that updates the numerator,
- compound assignment (`m += 10`) and post-increment (`m++`),
- chained assignment (`r = s = m`).

That gives us our interface. The implementation comes next.

Restricting Legal Values: Constructors

A constructor has one job: initialize a new object to a legal value. The class invariant says what "legal" means for this type - for Rational, the constraint is that the denominator must not be zero, since division by zero is undefined.

A first cut at the class shows three overloaded constructors:

```
class Rational {
private:
    int num_;
    int denom_;
public:
    // Default constructor
    Rational() noexcept: num_{0}, denom_{1} {}

    // Single-parameter constructor (explicit prevents implicit conversion)
    explicit Rational(int n) noexcept : num_{n}, denom_{1} {}

    // Two-parameter constructor
    Rational(int num, int denom): num_{num}, denom_{denom} {
        if (denom == 0) {
            throw std::out_of_range("Denominator cannot be zero");
        }
    }
};
```

Each constructor handles the client's way of creating a Rational differently. The default constructor produces 0/1 = 0. The single-int constructor produces n/1 = n. The two-int constructor takes both pieces, but only after checking that the denominator is non-zero. If it is zero, the constructor throws an exception, refusing to construct an object that would violate the invariant.

> ? How should a constructor deal with an illegal initial value?

There are several options, and the right choice depends on the application:

- Throw an exception (the option chosen above). The client must handle it; the broken object does not exist.
- Set the variable to a different legal value (e.g., default the denominator to 1). Quietly correct the input.
- Call `exit(1)` or `abort()`. Fail fast and stop the program.
- Print an error message and continue with a sentinel value. Diagnostic-friendly but error-prone.
- Return an error code. Not directly available from a constructor - would require a static factory function instead.

The one thing a constructor cannot do is silently fail and leave an invalid object behind. That is the worst of all worlds: the rest of the program continues to use an object whose invariant is broken, and bugs surface far from their cause.

The Default Constructor

A default constructor - one with no arguments - should only be defined when there is a reasonable default value. For Rational, 0/1 is a sensible default. For a Money class, 0 dollars is reasonable. For a Date class, the default is less obvious: today's date? The Unix epoch? Often, the right choice for a type with no clear default is to omit the default constructor entirely and force the client to supply meaningful values at construction time.

Member Initialization Lists (MIL)

Notice the colon-separated list in the constructors above:

```
Rational(int num, int denom): num_{num}, denom_{denom} { /* ... */ }
```

That syntax - between the parameter list and the constructor body - is a member initialization list, or MIL. It sets the private members directly during the object's construction, rather than assigning to them inside the body. Compare the two styles:

```
// With MIL - fields are initialized in place
Rational(int num, int denom): num_{num}, denom_{denom} {}

// Without MIL - fields are default-initialized, then assigned
Rational(int num, int denom) {
    this->num_ = num;
    this->denom_ = denom;
}
```

The MIL version is preferred for two reasons. First, it is often more efficient - assignment inside the body is preceded by an implicit default initialization, so the field is set twice. Second, and more importantly, in certain fields, the MIL is not just preferred but mandatory: the assignment style will not compile.

> 💡 **Tip:** Fields are initialized in the order they were declared in the class, not the order they appear in the MIL. The compiler will warn if your MIL order disagrees with your declaration order - listen to it.

When the MIL is Mandatory

There are four cases where the constructor body cannot do the job, and the MIL is the only option.

1. **const fields.** A const member can only be initialized; it cannot be assigned to after construction. The body-assignment style fails because assignment is exactly what is forbidden.

```
class Student {
    const int id;
public:
    Student(int id) {
        this->id = id;   // ERROR: cannot assign to const member
    }
};

// Correct: use the MIL
class Student {
    const int id;
public:
    Student(int id): id{id} {}
};
```

2. **Reference fields.** References must be bound at the moment the object is created - they cannot be left unbound and assigned later. The MIL is the only place this binding can happen.

```
class Student {
    University& myUni;
public:
    Student(University& uni) {
        this->myUni = uni;   // ERROR: cannot rebind a reference
    }
};
```

```
// Correct: bind the reference in the MIL
class Student {
    University& myUni;
public:
    Student(University& uni): myUni{uni} {}
};
```

3. **Fields whose type has no default constructor.** When a field is itself a class object, the compiler must call some constructor on it before the constructor body runs. If the field's class has no default constructor, the compiler does not know which one to call - unless you tell it explicitly via the MIL.

```
class A {
public:
    A(int x) {}   // No default constructor
};

class B {
    A myA;        // What constructor does the compiler call for myA?
public:
    B() {
        myA = A{5};   // ERROR: A has no default ctor; can't construct myA first
    }
};

// Correct: tell the compiler which A constructor to use, via the MIL
class B {
    A myA;
public:
    B(): myA{5} {}
};
```

4. **Inheriting from a base class with no default constructor.** A derived class must construct its base sub-object before its own fields. If the base has only non-default constructors, the derived class must invoke one of them by name in the MIL - there is no other way to pass arguments up to it.

```
class A {
    int x;
public:
    A(int x): x{x} {}    // No default constructor
};
```

```
class B: public A {
    int y;
public:
    B(int x, int y) {
        this->x = x;      // ERROR: x is private in A, not accessible
        this->y = y;
    }
};

// Correct: forward x to A's constructor via the MIL
class B: public A {
    int y;
public:
    B(int x, int y): A{x}, y{y} {}   // Works even though A has no default ctor
};
```

Note also the protection point: in the broken version above, even if we did not have the no-default-ctor problem, we still could not write `this->x = x;` from `B`, because `x` is private to A. Inheritance does not grant access to private base-class members - only protected ones. The MIL sidesteps the issue by going through A's constructor, which can see its own private fields.

> 💡 **Tip:** Embrace the MIL. Use it for every field, every time, even when the body-assignment style would work. Consistency makes the four mandatory cases feel like a natural extension rather than special syntax.

Accessors, Mutators, and const

Public Accessors and Mutators

Data members should be private. Always. That is the rule that makes everything else in this chapter work: if clients can poke at the fields directly, none of the invariants are safe. Instead, the class exposes accessors (functions that read a member) and mutators (functions that update a member, after checking that the new value is legal).

```
class Rational {
private:
    int num_;
    int denom_;

public:
    // Accessors
    int numerator() const noexcept { return num_; }
    int denominator() const noexcept { return denom_; }
```

```
    // For backwards compatibility
    int getnum() const noexcept { return num_; }
    int getdenom() const noexcept { return denom_; }

    // Mutators
    void setnum(int n) noexcept {
        num_ = n;
    }
    void setdenom(int n) {
        if (n == 0) {
            throw std::out_of_range("Denominator cannot be zero");
        }
        denom_ = n;
    }
};
```

Notice how `setdenom` repeats the same legality check as the two-argument constructor. Every entry point that could put the object into an illegal state must enforce the invariant. Anywhere a client can change a field, the mutator gets the chance to refuse.

⚠️ Mutators that allow any value to be written are essentially the same as making the field public - the encapsulation buys you nothing. If a mutator does not need to check anything, ask whether the field should be writable at all. Sometimes the right answer is "no mutator": the field is set once at construction and never changed.

Constant Member Functions

Look again at the accessors above. Each one ends with the keyword const just before the opening brace:

```
int numerator() const noexcept { return num_; }
```

This is a constant member function - a method that promises not to modify the object on which it is called. The compiler enforces the promise: if you accidentally write to a member inside a const method, the code does not compile.

There are two benefits. First, the promise is documented in the function signature and backed by the compiler, so clients reading the header know which operations are read-only. Second, const methods are the only methods that can be called on a const object. If a function receives a Rational by const reference, only the accessors are callable on it - the mutators are not.

> 💡 **Tip:** Use const member functions liberally - every accessor, every operation that does not modify the object. The discipline pays off when you start passing objects by const reference, because anything that is not const-correct becomes painful very quickly.

Pass-by-Const-Reference

When a function takes an object as a parameter, there are three plausible ways to pass it: by value, by reference, or by const reference. For ADT objects, the right default is almost always pass-by-const-reference.

- Pass-by-value copies the object - fine for an int, expensive for anything larger.
- Pass-by-reference avoids the copy, but lets the function silently modify the object - usually not what the caller expects.
- Pass-by-const-reference avoids the copy AND prevents modification - the best of both.

A function that takes its parameters by const reference is also more general. It can accept variables, named constants, and temporary objects (including the result of an expression like `r + s`). A pass-by-non-const-reference parameter can only bind to a named variable - temporaries are rejected, because there is nothing for the function to "modify back to."

> 💡 **Tip:** For class types, pass-by-const-reference is the standard. It is efficient like pass-by-reference, safe like pass-by-value, and works with temporaries and literals as arguments. Reach for it by default and only diverge when you have a specific reason.

Function Overloading

Look once more at the constructors of Rational:

```
Rational() noexcept: num_{0}, denom_{1} {}
explicit Rational(int n) noexcept: num_{n}, denom_{1} {}
Rational(int num, int denom): num_{num}, denom_{denom} { /* ... */ }
```

Three functions, same name, three different parameter lists. This is function overloading: defining several functions with the same name and letting the compiler pick the right one based on the arguments at the call site.

Function overloading is allowed wherever the parameter lists are sufficiently different. Two rules govern when an overload set is well-formed:

- the functions must have different argument signatures (different numbers or types of parameters), and
- The return type alone is not enough - two functions that differ only by return type cannot coexist.

When you call one of an overloaded set, the compiler performs overload resolution. It looks at the arguments you provided and ranks the candidates based on how well each matches them. The ranking proceeds in tiers:

1. Exact match - argument types are exactly the parameter types.
2. Match via promotions - lossless conversions like bool→int, char→int, float→double.
3. Match via standard conversions - int→double, double→int, derived*→base*, etc.
4. Match via user-defined conversions - constructors and conversion operators you wrote.

The compiler picks the best match. If two candidates tie at the same tier, the call is ambiguous, and the compiler reports an error rather than making a guess. With multiple arguments, the rule is that one candidate must be at least as good as every other candidate on every argument, and strictly better on at least one.

> 💡 **Tip:** Overload-resolution rules are detailed, but in practice, the right intuition is simple: write the overloads you actually need, give them clearly different parameter lists, and let the compiler tell you when something is ambiguous. Trying to anticipate every conversion path is more confusing than just letting the error message tell you what went wrong.

Operator Overloading

Function overloading extends naturally to operators: the `+`, `==`, `<<`, and other built-in operators can be overloaded so that they work on your ADT objects, just as they work on int or double. This is what lets us write `r + s` instead of `r.plus(s)`, and `cout << r` instead of `r.print(cout)`.

Designing an operator overload is mostly the same exercise as designing any other function: pick the parameter types, the return type, whether the parameters are const, and whether the function is a member or non-member. There are also a few constraints:

- You cannot invent new operators. Trying to define operator`**` for exponentiation will not compile - only existing operators can be overloaded.
- You cannot change the number of operands. Binary `+` must take two arguments; you cannot make it ternary.
- You cannot change operator precedence. `+` still binds less tightly than `*`, no matter how you overload them.

Within those constraints, you have a lot of freedom - you choose the argument types, the return type, whether to pass by value or reference, whether parameters are const, and whether the operator is implemented as a member function or a non-member.

💡 **Tip:** Use operator signatures that the client programmer expects. `operator==` should return a bool. `operator+` should return a new object, not modify one of the operands. `operator<<` should return a reference to the stream. Deviating from convention surprises your users and breaks code that should "just work."

⚠️ You can technically change the meaning of an operator - make `operator+` subtract, for example - but doing so is one of the few uniformly bad ideas in C++. Operator overloads should preserve the operator's intuition, just extended to a new type.

operator+

Adding two rationales `a/b + c/d` gives `(ad + bc) / (bd)`. A straightforward non-member implementation:

```
Rational operator+(const Rational& lhs, const Rational& rhs) {
    const auto a = lhs.getnum();
    const auto b = lhs.getdenom();
    const auto c = rhs.getnum();
    const auto d = rhs.getdenom();
    return Rational{a * d + b * c, b * d};
}
```

A few things worth noticing. Both parameters are const references - adding does not modify either operand, and a const reference is the right default for ADT inputs. The return type is Rational by value, because the result is a fresh object, not a reference to anything that already exists. And the entire function lives outside the class - it is a non-member, defined later in the .cpp file.

💡 **Tip:** const auto is a modern C++ feature that combines two ideas: auto lets the compiler deduce the variable's type from its initializer (so you do not have to write int explicitly), and const makes it immutable after initialization. The combination reduces typing, prevents accidental modifications, and means the code keeps working even if getnum()'s return type changes someday.

We also want to support mixed addition - an int plus a Rational, or a Rational plus an int. We can add two more overloads:

```
Rational operator+(int lhs, const Rational& rhs) {
    return Rational(lhs) + rhs;        // promote int to Rational, recurse
}
```

```
Rational operator+(const Rational& lhs, int rhs) {
    return lhs + Rational(rhs);        // same idea on the other side
}
```

Each mixed overload promotes the int to a Rational using the single-argument constructor and then defers to the Rational + Rational overload we already wrote. This is a common pattern: implement the operation once for the canonical types and write thin adapters for the mixed cases. The first function does the real work; everything else routes through it, so a bug fix in one place fixes the whole family.

Streaming Operators (>>, <<)

Input and output for ADT objects use the same << and >> operators as built-in types, overloading them with the appropriate signatures.

> 💡 **Tip:** Streaming operators should be non-member functions. The first operand is the stream (a std::istream& or std::ostream&), not a Rational - so a member operator<< would put the stream on the right of the dot, which is impossible. A non-member function has no such constraint.

Both should return a reference to the modified stream, so that operations can be chained: `cout << r << " + " << s` reads as `(((cout << r) << " + ") << s)`, with each call returning cout so the next one can be applied to it. A typical pair:

```
std::istream& operator>>(std::istream& in, Rational& r) {
    int num, denom;
    char slash;
    in >> num >> slash >> denom;
    r = Rational(num, denom);
    return in;
}

std::ostream& operator<<(std::ostream& out, const Rational& r) {
    out << r.getnum() << '/' << r.getdenom();
    return out;
}
```

operator>> takes the Rational by non-const reference, because its job is exactly to write into it. operator<< takes it by const reference, because output only reads. Both take and return the stream by non-const reference, since the stream has internal state (position, error flags) that the operation does change.

Type Conversion of ADT Objects

Constructors do more than initialize objects. Whenever a constructor takes a single argument (or has multiple arguments but defaults for all but the first), the compiler treats it as a conversion from the argument type to the class type. The compiler will silently apply that conversion to make a function call work - sometimes helpfully, sometimes not.

Consider the single-int constructor for Rational. Without further annotation, this code compiles:

```
void printRational(const Rational& r);

printRational(7);    // 7 is silently converted to Rational(7) = 7/1
```

The compiler sees that `printRational` expects a Rational, but you passed an int. It searches for a way to construct a Rational from an int, finds the single-int constructor, and automatically inserts the conversion.

Sometimes you want this - implicit conversion makes mixed-type expressions like `r + 5` work without writing explicit casts everywhere. Sometimes you do not - silent conversions can hide bugs, especially when the conversion is expensive or semantically dubious. The keyword explicit on a constructor tells the compiler, "do not use me for implicit conversions; require the client to write the conversion out by hand.":

```
explicit Rational(int n) noexcept : num_{n}, denom_{1} {}

// With 'explicit':
//   Rational r = 5;            // ERROR: no implicit conversion
//   Rational r(5);             // OK: direct construction
//   Rational r = Rational(5);  // OK: explicit conversion
```

> 💡 **Tip:** A good default is to mark single-argument constructors explicit unless you have a specific reason to allow implicit conversion. The cases where implicit conversion genuinely helps are rare and easy to recognize; the cases where it hides bugs are common and easy to miss.

Friends

Non-member operators, such as operator+, have a problem: they cannot access the class's private fields. Our `operator+` above worked around this by going through the public accessors (`getnum`, `getdenom`). Sometimes that is fine; sometimes the indirection is annoying, or the accessors do not exist yet.

The friend declaration lets a class grant a specific non-member function (or another class) access to its private members. A friend function is not a member - it does not have a `this` pointer, is not called with the dot operator, and is not part of the class's interface - but it can see inside the class as if it were.

```
class Rational {
private:
    int num, denom;
public:
    // ...
    friend Rational operator+(const Rational& lhs, const Rational& rhs);
};

Rational operator+(const Rational& lhs, const Rational& rhs) {
    return Rational{ lhs.num * rhs.denom + lhs.denom * rhs.num,
                     lhs.denom * rhs.denom };
}
```

The friend version is shorter and a touch more efficient (no function-call overhead through accessors), but it broadens the codebase by allowing direct access to private data, which is exactly what we worked to restrict in the first place. The accessor-based version, by contrast, looks like:

```
Rational operator+(const Rational& lhs, const Rational& rhs) {
    return Rational{ lhs.getnum() * rhs.getdenom() + lhs.getdenom() * rhs.getnum(),
                     lhs.getdenom() * rhs.getdenom() };
}
```

> 💡 **Tip:** Prefer the accessor-based style when accessors exist, and the overhead does not matter. Reach for friend when you genuinely need direct access - when the accessors do not exist, when the function is performance-critical, or when the operator must inspect multiple private fields in a way that public accessors would clutter.

Other Tools for ADT Design

Helper Functions

Most non-trivial classes have small internal routines used by several members but not meaningful to clients - a private `reduce()` that divides a Rational by its GCD, for example, or a `normalize()` that puts a date into canonical form. These helper functions modularize code that would otherwise repeat across member functions, but they should not appear in the public interface.

> 💡 **Tip:** Hide helper functions as private methods of the class, or, if they are genuinely free functions, place them inside a namespace dedicated to the class. Helpers should not pollute the global namespace, and they should not be callable by clients who do not need them.

override and final

Two keywords introduced in C++11 - override and final - help the compiler catch mistakes in inheritance hierarchies, and let you forbid further inheritance when the design calls for it. They appear after the parameter list of a virtual function (or, for `final` on a class, after the class name).

override on a method signals "this method is intentionally overriding a virtual function from the base class." The compiler verifies that the claim is true. If you mistype the function name, or get the signature subtly wrong, the compiler tells you immediately - rather than silently treating your method as an unrelated new function that never gets called.

```
class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Animal makes a generic sound" << std::endl;
    }
    virtual void eat() const {
        std::cout << "Animal is eating" << std::endl;
    }
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    // 'override' confirms we're overriding a base virtual function
    void makeSound() const override {
        std::cout << "Dog barks: Woof! Woof!" << std::endl;
    }

    void eat() const override {
        std::cout << "Dog is eating dog food" << std::endl;
    }

    // The following would NOT compile (typo in function name):
    // void makeSond() const override { ... }
    //   error: 'makeSond' marked 'override' but does not override
};
```

Without override, the typo would silently define a brand-new method `makeSond` on `Dog`. Polymorphic calls through an `Animal*` would call `Animal::makeSound` instead of the dog's version, the program would compile cleanly, and the bug would only show up at runtime when "Animal makes a generic sound" appeared where you expected the dog. override turns that runtime mystery into a compile error.

final on a virtual method does the opposite: it forbids further overriding. Once a function is declared final, derived classes are not allowed to override it. This is useful when the implementation must not change - perhaps for invariant reasons, or because you want the compiler to perform optimizations that depend on the call being non-virtual in practice.

```
class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a generic shape" << std::endl;
    }

    // 'final' prevents any derived class from overriding this method
    virtual void calculateArea() const final {
        std::cout << "Area calculation is finalized" << std::endl;
    }

    virtual ~Shape() = default;
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a circle" << std::endl;
    }

    // The following would NOT compile (cannot override a final method):
    // void calculateArea() const override { ... }
    //   error: 'calculateArea' marked 'final' cannot be overridden
};
```

final on a class prevents the class from being used as a base at all. No derivation, no override, no surprises. This is appropriate when the class is meant to be used as-is, and any inheritance hierarchy would be a mistake - utility classes, immutable types, and tightly-locked-down implementations.

```
class Vehicle {
public:
    virtual void start() const {
        std::cout << "Vehicle starting..." << std::endl;
    }
    virtual ~Vehicle() = default;
};

// 'final' class - cannot be inherited from
class ElectricCar final: public Vehicle {
public:
    void start() const override {
        std::cout << "Electric car starting silently..." << std::endl;
    }
    void charge() const {
        std::cout << "Electric car is charging" << std::endl;
    }
};
// The following would NOT compile (cannot inherit from final class):
// class TeslaModel3 : public ElectricCar { };
//   error: cannot derive from 'final' base 'ElectricCar'
```

> 💡 **Tip:** Key takeaways. (1) override is essentially free safety - apply it to every method that is supposed to override a base virtual function, every time. (2) final on methods says "this is sealed." (3) final on classes says "this hierarchy ends here." All three help the compiler catch mistakes that would otherwise become runtime bugs or design erosion.

Putting It All Together

We can now assemble the full Rational class. The header file declares the class and the non-member operators; the .cpp file defines the operator implementations.

rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream>
#include <stdexcept>
#include <numeric>

class Rational {
private:
    int num_;
    int denom_;
    void reduce() noexcept {}      // helper: num/denom by their GCD

public:
    // Default constructor
    Rational() noexcept: num_{0}, denom_{1} {}

    // Single-parameter constructor (explicit)
    explicit Rational(int n) noexcept: num_{n}, denom_{1} {}

    // Two-parameter constructor
    Rational(int num, int denom): num_{num}, denom_{denom} {
        if (denom == 0) {
            throw std::out_of_range("Denominator cannot be zero");
        }
        reduce();
    }

    // Copy operations (defaults are fine, shown for clarity)
    Rational(const Rational& other) noexcept = default;
    Rational& operator=(const Rational& rhs) noexcept = default;

    // Move operations
    Rational(Rational&& other) noexcept = default;
```

```
    Rational& operator=(Rational&& rhs) noexcept = default;

    // Destructor
    ~Rational() = default;

    // Accessors
    int numerator()   const noexcept { return num_; }
    int denominator() const noexcept { return denom_; }
    int getnum()      const noexcept { return num_; }
    int getdenom()    const noexcept { return denom_; }

    // Mutators
    void setnum(int n) noexcept {
        num_ = n;
        reduce();
    }
    void setdenom(int n) {
        if (n == 0) {
            throw std::out_of_range("Denominator cannot be zero");
        }
        denom_ = n;
        reduce();
    }
};

// Streaming operators
std::istream& operator>>(std::istream& in, Rational& r);
std::ostream& operator<<(std::ostream& out, const Rational& r);

// Arithmetic operators
Rational  operator+(const Rational& lhs, const Rational& rhs);
Rational  operator+(int             lhs, const Rational& rhs);
Rational  operator+(const Rational& lhs, int             rhs);

// Compound assignment, increment
Rational& operator+=(Rational& r, int n);
//(++r)
Rational& operator++(Rational& r);
// (r++)
Rational operator++(Rational& r, int);

// Comparison operators
bool operator==(const Rational& lhs, const Rational& rhs) noexcept;
bool operator!=(const Rational& lhs, const Rational& rhs) noexcept;
bool operator< (const Rational& lhs, const Rational& rhs) noexcept;
bool operator<=(const Rational& lhs, const Rational& rhs) noexcept;
bool operator> (const Rational& lhs, const Rational& rhs) noexcept;
bool operator>=(const Rational& lhs, const Rational& rhs) noexcept;

// Multiplication
Rational operator*(const Rational& lhs, const Rational& rhs);

#endif
```

rational.cpp

```
#include "rational.h"

std::istream& operator>>(std::istream& in, Rational& r) {
    int num, denom;
    char slash;
    in >> num >> slash >> denom;
    r = Rational(num, denom);
    return in;
}

std::ostream& operator<<(std::ostream& out, const Rational& r) {
    out << r.getnum() << '/' << r.getdenom();
    return out;
}

Rational operator+(const Rational& lhs, const Rational& rhs) {
    const auto a = lhs.getnum();
    const auto b = lhs.getdenom();
    const auto c = rhs.getnum();
    const auto d = rhs.getdenom();
    return Rational{a * d + b * c, b * d};
}

Rational operator+(int lhs, const Rational& rhs) {
    return Rational(lhs) + rhs;
}

Rational operator+(const Rational& lhs, int rhs) {
    return lhs + Rational(rhs);
}
// Remaining operators left as practice (see end of chapter).
```

Sample Output

Running the client code from the start of the chapter with inputs 2/7 and 16/3 produces:

```
Enter rational number (a/b): 2/7
Enter rational number (a/b): 16/3
2/7
16/3
118/21
100/21
2/7 + 16/3 = 118/21
2/1
7/1
8/1
18/1
18/1
18/1
```

Summary

A class designed as an abstract data type combines a private representation with a public interface in a way that makes the type easy to use, hard to misuse, and evolvable. The recurring techniques in this chapter all serve those goals:

- Constructors establish the invariant. Make data members private, validate inputs in the constructor, and throw on invalid initial values rather than letting broken objects exist.
- The member initialization list initializes fields in place. Prefer it everywhere, and recognize the four cases where it is mandatory: const fields, reference fields, fields without default constructors, and base classes without default constructors.
- Mutators preserve the invariant. Every public path that can change a field must first check the new value.
- const correctness is enforced by the compiler. Mark accessors const, pass parameters by const reference, and your invariants become unbreakable from outside the class.
- Overload functions and operators where it makes the interface natural. Use the conventional signatures clients expect, and use explicit to forbid implicit conversions you do not want.
- override and final document inheritance intent and turn would-be runtime bugs into compile errors.

Each individual technique is small. Together, they let you define types that the rest of the program can use as comfortably as the built-in ones, which is the whole point.

Practice Problems

1.

Complete the Rational class.

(a) Implement `reduce()` so that the numerator and denominator are always divided by their GCD after every constructor and mutator call. Use `std::gcd` from `<numeric>`.

(b) Implement `operator-`, `operator*`, and `operator/` for two Rationals, plus the two mixed-type overloads for each (int with Rational and Rational with int).

(c) Implement the comparison operators `!=`, `==`, `<`, `<=`, `>`, `>=`.

(d) Implement compound assignment (`+=`, `-=`, `*=`, `/=`) and pre- and post-increment/decrement (`++`, `--`).

2.

Design a `Date` ADT.

The `Date` class should store a day, a month, and a year. Provide:

• A constructor that validates its input (a Date object never holds an invalid date - no February 30, no month 13, no day 0).

• Three accessors (g`etDay`, `getMonth`, `getYear`) and three mutators (`setDay`, `setMonth`, `setYear`). Mutators must preserve the invariant.

• Overloaded ==, !=, <, <=, >, >= (one `Date` is "less than" another if it comes earlier in time).

• Overloaded ++, −−, +=, −=. The increment operators advance the date by one day; += and −= take an integer number of days.

• Overloaded << and >> for streaming.

3.

Streaming-subscription system (polymorphism warm-up).

Implement a system that manages streaming subscriptions for customers. Every subscription has an automatically generated account number and a balance. A positive balance is a credit; a negative balance is a debit. All subscriptions support making payments, querying the balance, printing account information, recording streaming usage, and applying monthly billing.

Two subscription types inherit from a common abstract base class Subscription. The base class stores the account number and balance, and provides a constructor, a payment function, a balance accessor, and a print function. It also declares two pure virtual functions: one for recording streaming usage, one for applying monthly billing.

Basic Subscription costs $10 per month and includes up to 5 hours of streaming at no additional cost. Any streaming beyond five hours costs $2 per extra hour. A basic subscription tracks how many hours have been streamed in the current billing period; when streaming occurs, the count increases; when monthly billing is applied, the balance is reduced by $10 plus any per-hour overage, and the hour count is reset to zero.

Premium Subscription costs $25 per month and includes unlimited streaming, so it does not track hours. Recording streaming is a no-op; monthly billing just reduces the balance by $25.

The print function should identify the subscription type and display the account number and balance. For basic subscriptions, it should also display the hours used this month. A test harness should interact with subscriptions through base-class pointers to demonstrate polymorphic behaviour: create both types, record streaming activity, apply payments, bill at the end of the month, and print each subscription's state. Assume all input to the harness is valid.