Module 10 of 20

# Inheritance and polymorphism

Deriving and the order of construction, virtual functions with override and final, abstract classes and interfaces held through unique_ptr, virtual destructors and slicing, the vtable, dynamic_cast and std::variant, and when composition beats inheritance.

---

Inheritance is how one C++ class is built on another, and polymorphism is what you get when a function called through a base reference runs the derived class's version. Java learners know the shape — `extends`, `@Override`, interfaces — but C++ makes every part of it explicit and puts the consequences in your hands: the base's access specifier, a construction order the compiler fixes, a `virtual` keyword that turns a compile-time call into a run-time one, a destructor that must be virtual or the delete is undefined behaviour, and a copy into a base value that silently drops the derived half. Get these right and a hierarchy is the cleanest way to say "many kinds of thing, one interface"; get them wrong and the program compiles and lies.

The six lessons move from mechanics to design. Inheritance basics fixes what is inherited, what `protected` means, the base-then-members-then-body order and its reverse, `using B::B` and name hiding. Virtual functions separates static from dynamic type, adds `override` and `final`, and shows the two places dispatch does not reach: default arguments and constructors. Abstract classes and interfaces introduces `= 0`, the interface class, a `Shape` hierarchy and the container that holds it, `std::vector<std::unique_ptr<Shape>>`. Virtual destructors and slicing covers the two ways a hierarchy loses data and the rules that prevent both. RTTI and the vtable opens the implementation — vptr, vtable, one indirect call — then `dynamic_cast`, `typeid`, the cast-chain smell and `std::variant` as the closed-set alternative. Composition versus inheritance closes with is-a against has-a, the Liskov rule and its square-and-rectangle failure, private and multiple inheritance, and the reasons a member usually beats a base.

The exercises are whole programs that print what the language does: a construction and destruction tracer whose order is fully defined, an account whose hidden `withdraw` calls the base version, a side-by-side of virtual and non-virtual dispatch naming which function ran, a pricing hierarchy with `override` and `final`, a `Shape` catalogue and an encoder pipeline through `unique_ptr`, a slicing demonstration and a resource stack released in reverse, a `dynamic_cast` census and the same shapes redone as a `std::variant`, a service that owns its logger and an

undo history composed of two vectors. The checkpoint's three programs — a bank of polymorphic accounts, an expression tree and a plugin registry with a load and unload trace — draw on all of it.

CONTENTS


| | | |
|---|---|---|
| 1 | Inheritance basics — deriving, protected and the order of construction | 14 min |
| 2 | Virtual functions — static type, dynamic type and override | 15 min |
| 3 | Abstract classes and interfaces — programming to a contract | 14 min |
| 4 | Virtual destructors and slicing — the two ways a hierarchy loses data | 14 min |
| 5 | RTTI and the vtable — how dispatch works and when to ask an object its type | 15 min |
| 6 | Composition versus inheritance — is-a, has-a and the shapes to avoid | 14 min |
| ✓ | Checkpoint — Inheritance and polymorphism | 25 min |


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

LESSON 1 · 14 MIN

# Inheritance basics — deriving, protected and the order of construction

A class can be built on another: `class Car : public Vehicle` makes every `Car` contain a `Vehicle` and gives it that class's members. Java has the same idea with `extends`; C++ differs in the details that matter — an access specifier on the base itself, a `protected` level between public and private, a construction sequence the compiler fixes and you cannot reorder, constructors that are not inherited unless you ask, and a name-lookup rule that hides base functions wholesale. This lesson settles those mechanics. The next lesson adds `virtual`, which is where inheritance stops being code reuse and becomes polymorphism.

---

## Deriving a class

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

class Vehicle {
public:
    explicit Vehicle(std::string name) : name_(std::move(name)) {}
    const std::string& name() const { return name_; }
    int wheels() const { return wheels_; }
protected:
    int wheels_ = 4;
private:
    std::string name_;
};

class Car : public Vehicle {
public:
    Car(std::string name, int doors) : Vehicle(std::move(name)), doors_(doors)
    int doors() const { return doors_; }
private:
    int doors_;
};

int main() {
    Car car("mini", 3);
    std::cout << car.name() << ' ' << car.wheels() << ' ' << car.doors() << '
    return 0;
}
```

`Car` is the *derived* class and `Vehicle` its *base* — C++ says base and derived where Java says super and sub. A `Car` object contains a complete `Vehicle` subobject (on this platform the base's members are laid out first, then the derived class's own), and every public member of `Vehicle` is callable on a `Car` as though `Car` had declared it. What is inherited: data members, member functions including `static` ones, and nested types. What is not: constructors, the destructor and the assignment operators — the derived class gets its own, which call the base's — and friendships. `private` members are present in the object but inaccessible to the derived class's code.

The word `public` before `Vehicle` is the *inheritance access specifier*, and in practice it is not optional. `class Car : Vehicle` means **private** inheritance — the default for `class`, just as private is the default for members — which makes every inherited member private to `Car` and stops a `Car` from being passed where a `Vehicle&` is expected. `struct` defaults to public. Write `public` every time you mean is-a; lesson 6 covers the rare uses of the private form.

A derived object converts implicitly to a reference or pointer to its base — `const Vehicle& v = car;` compiles because a `Car` *is a* `Vehicle`. That upcast is what makes the next lesson possible. A conversion in the other direction never happens by itself.

---

## Three levels of access

`protected` sits between `public` and `private`: a protected member is accessible inside the class, inside its friends, and inside member functions of derived classes — but only through an object of the derived type. `wheels_` above may be read by `Car`'s member functions; a free function or `main` cannot touch it, and `Car`'s functions cannot read `wheels_` through a plain `Vehicle&` either, only through a `Car`.

| Specifier | Inside the class | In a derived class | Everywhere else |
|---|---|---|---|
| `public` | yes | yes | yes |
| `protected` | yes | yes | no |
| `private` | yes | no | no |

Protected data is a weak form of public: every derived class becomes a client of the base's representation, and changing that representation breaks them all. Prefer private data with protected *functions* when derived classes need a controlled way in; keep protected data for small, stable hierarchies where the convenience is worth it.

---

## Construction order

A derived class's constructor runs in three fixed stages, whatever order the initialiser list happens to write them in:

1. The **base class** subobject is constructed — by the constructor named in the initialiser list, or by the base's default constructor when none is named.
2. The derived class's **members** are constructed in declaration order — the same rule as a class with no base (Module 8, lesson 2).
3. The **constructor body** runs.

```
struct Engine { Engine() { std::cout << "Engine "; } ~Engine() { std::cout <<
struct Base   { Base()   { std::cout << "Base "; }   ~Base()   { std::cout <<
struct Derived : Base {
    Engine engine;
    Derived()  { std::cout << "Derived "; }
    ~Derived() { std::cout << "~Derived "; }
};

int main() {
    Derived d;                  // Base Engine Derived
}                               // ~Derived ~Engine ~Base
```

Destruction is the exact reverse: the derived destructor's body, then the derived members in reverse declaration order, then the base. That symmetry is why a derived destructor may still use the base's members — the base stays alive until the derived body has finished — and why a base destructor must never rely on derived state, which is already gone. It also means the base is fully built before the derived body starts, so a derived constructor can call base member functions in its body safely.

---

## Calling the base constructor

The base constructor is called from the initialiser list, and only from there:

```
class Car : public Vehicle {
public:
    Car(std::string name, int doors) : Vehicle(std::move(name)), doors_(doors)
};
```

If the list does not name the base, the compiler inserts `Vehicle()`. When the base has no default constructor — `Vehicle` has only `Vehicle(std::string)` — that is a compile error: `no matching function for call to 'Vehicle::Vehicle()'`. Writing `Vehicle(name);` inside the body instead of the list is legal C++ that does something else entirely: it constructs a temporary `Vehicle` and throws it away, after the base subobject was already default-constructed (or failed to be). The base always goes in the list, and it goes first by rule even when you write it second.

---

## Inheriting constructors

Constructors are not inherited: a `Car` needs its own, even one that only forwards. When the derived class adds no state that needs arguments, C++11 lets you pull the base's constructors in wholesale:

```
class Taxi : public Vehicle {
public:
    using Vehicle::Vehicle;      // Taxi(std::string) now exists and forwards
    int fares = 0;               // derived members get their default member i
};

Taxi t("cab-7");
```

`using Base::Base;` declares a constructor in `Taxi` for every constructor of `Vehicle`, each forwarding its arguments to the base and leaving the derived members to their in-class initialisers. It is the right tool for thin wrappers and tag types. A derived class with members that must be set from arguments still writes its own constructor.

---

## Name hiding

```
struct Printer {
    void print(int v)    { std::cout << "int " << v << '\n'; }
    void print(double v) { std::cout << "double " << v << '\n'; }
};
struct Fancy : Printer {
    void print(const std::string& s) { std::cout << "string " << s << '\n'; }
};

Fancy f;
f.print("hi");      // string hi
f.print(3);         // error: no conversion from int to std::string
```

Declaring `print` in `Fancy` **hides every** `print` in `Printer`, not only the one with matching parameters. Name lookup stops at the first class, walking upward, that declares the name; overload resolution then runs on what it found there and nothing else. Java merges overloads across the hierarchy; C++ does not. Two fixes: `using Printer::print;` inside `Fancy` brings the base overloads into the derived scope so all three overload together, or call the hidden one explicitly as `f.Printer::print(3)`. The same qualified form, `Printer::print(v)`, is how a derived function calls the base version of the function it replaces — a habit that becomes essential the moment functions are `virtual`.

## Pitfalls

| Mistake | What happens |
| --- | --- |
| `class D : B` | Private inheritance; `D` is not usable as a `B` outside itself. |
| Base has no default constructor and the list omits it | Compile error at the derived constructor. |
| `Base(x);` in the constructor body | A temporary; the base subobject was default-constructed already. |
| Relying on initialiser-list order | Ignored: base first, then members in declaration order. |
| A derived `f(double)` next to a base `f(int)` | The base overload is hidden; add `using Base::f;`. |
| Reading a protected member through a `Base&` inside `Derived` | Not allowed; only through a `Derived` object. |
| Using derived state in the base destructor | Already destroyed — the reverse order guarantees it. |

## Key takeaways

`class D : public B` embeds a complete `B` in every `D` and exposes `B`'s public members; write `public` explicitly or you get private inheritance.

`protected` is visible to derived classes and friends; prefer protected functions over protected data.

Construction is base, then members in declaration order, then the body; destruction is the exact reverse.

The base constructor is named in the initialiser list; omit it and the base's default constructor is used or the build fails.

`using B::B;` inherits constructors for a derived class that adds nothing needing arguments.

A derived name hides every base overload of that name; `using B::f;` or `B::f()` gets them back.

---

EXERCISE 1

## Construction and destruction tracer

Complete a three-level hierarchy whose constructors and destructors announce themselves, then let the scaffold build the objects and tear them down.

`Vehicle` holds a protected `std::string name_` and prints `Vehicle(<name>)` when constructed and `~Vehicle(<name>)` when destroyed. `Car : public Vehicle` adds an `Engine` member (which prints `Engine(<name>)` / `~Engine(<name>)`) and prints `Car(<name>)` / `~Car(<name>)`. `SportsCar : public Car` adds a `Turbo` member (`Turbo(<name>)` / `~Turbo(<name>)`) and prints `SportsCar(<name>)` / `~SportsCar(<name>)`. Every constructor takes the name and passes it to its base in the **initialiser list**; the members are initialised from the same name.

The starter's `build` function constructs the objects one per recursive call, so they are all alive at once and die in reverse order when the calls return. It prints `-- all built --` between the two phases. Your job is the classes; the order is the language's.

**Input:** `n`, then `n` lines of `kind name` where kind is `vehicle`, `car` or `sportscar`. **Output:** the construction trace, the separator, the destruction trace.

```
2
car mini
sportscar zed
```

prints

```
Vehicle(mini)
Engine(mini)
Car(mini)
Vehicle(zed)
Engine(zed)
Car(zed)
Turbo(zed)
SportsCar(zed)
-- all built --
~SportsCar(zed)
~Turbo(zed)
~Car(zed)
~Engine(zed)
~Vehicle(zed)
~Car(mini)
~Engine(mini)
~Vehicle(mini)
```

INPUT 1

```
2
car mini
sportscar zed
```

OUTPUT 1

```
Vehicle(mini)
Engine(mini)
Car(mini)
Vehicle(zed)
Engine(zed)
Car(zed)
Turbo(zed)
SportsCar(zed)
-- all built --
~SportsCar(zed)
~Turbo(zed)
~Car(zed)
~Engine(zed)
~Vehicle(zed)
~Car(mini)
~Engine(mini)
~Vehicle(mini)
```

### INPUT 2

```
1
sportscar ace
```

### OUTPUT 2

```
Vehicle(ace)
Engine(ace)
Car(ace)
Turbo(ace)
SportsCar(ace)
-- all built --
~SportsCar(ace)
~Turbo(ace)
~Car(ace)
~Engine(ace)
~Vehicle(ace)
```

### REFERENCE SOLUTION

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

class Engine {
public:
    explicit Engine(const std::string& tag) : tag_(tag) { std::cout << "Engine(" <<
    ~Engine() { std::cout << "~Engine(" << tag_ << ")\n"; }
private:
    std::string tag_;
};

class Turbo {
public:
    explicit Turbo(const std::string& tag) : tag_(tag) { std::cout << "Turbo(" << ta
    ~Turbo() { std::cout << "~Turbo(" << tag_ << ")\n"; }
private:
    std::string tag_;
};

class Vehicle {
public:
    explicit Vehicle(const std::string& name) : name_(name) { std::cout << "Vehicle(
    ~Vehicle() { std::cout << "~Vehicle(" << name_ << ")\n"; }
protected:
    std::string name_;
};

class Car : public Vehicle {
public:
    explicit Car(const std::string& name) : Vehicle(name), engine_(name) {
        std::cout << "Car(" << name_ << ")\n";
    }
    ~Car() { std::cout << "~Car(" << name_ << ")\n"; }
private:
    Engine engine_;
};

class SportsCar : public Car {
public:
    explicit SportsCar(const std::string& name) : Car(name), turbo_(name) {
        std::cout << "SportsCar(" << name_ << ")\n";
    }
    ~SportsCar() { std::cout << "~SportsCar(" << name_ << ")\n"; }
private:
    Turbo turbo_;
};

struct Spec {
```

```
    std::string kind;
    std::string name;
};

// Builds specs[i] on this frame, then the rest on deeper frames; returning unwinds
void build(const std::vector<Spec>& specs, std::size_t i) {
    if (i == specs.size()) {
        std::cout << "-- all built --\n";
        return;
    }
    const Spec& s = specs[i];
    if (s.kind == "vehicle") {
        Vehicle v(s.name);
        build(specs, i + 1);
    } else if (s.kind == "car") {
        Car c(s.name);
        build(specs, i + 1);
    } else {
        SportsCar sc(s.name);
        build(specs, i + 1);
    }
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<Spec> specs(n);
    for (auto& s : specs) std::cin >> s.kind >> s.name;
    build(specs, 0);
    return 0;
}
```

EXERCISE 2

## Plain account, fee account

Write `Account` with a `protected long long balance_`, a constructor taking the opening balance, `void deposit(long long)`, `bool withdraw(long long)` (returns `false` and changes nothing when the amount exceeds the balance) and `long long balance() const`.

Write `FeeAccount : public Account` that **inherits the constructor** with `using Account::Account;`, keeps a class constant `kFee = 2`, and declares its own `bool withdraw(long long amount)` — which **hides** the base's — implemented as one call to the base version: `Account::withdraw(amount + kFee)`. A withdrawal that would overdraw after the fee is rejected and nothing changes.

The program opens one of each with the same opening balance and applies every command to both.

**Input:** the opening balance, then lines of `deposit x` or `withdraw x` until the input ends. **Output:** after each command, `plain=<balance> fee=<balance>`; at the end, `rejected plain=<count> fee=<count>`.

```
100
deposit 50
withdraw 20
withdraw 130
withdraw 128
```

prints

```
plain=150 fee=150
plain=130 fee=128
plain=0 fee=128
plain=0 fee=128
rejected plain=1 fee=2
```

INPUT 1

```
100
deposit 50
withdraw 20
withdraw 130
withdraw 128
```

OUTPUT 1

```
plain=150 fee=150
plain=130 fee=128
plain=0 fee=128
plain=0 fee=128
rejected plain=1 fee=2
```

INPUT 2

```
10
withdraw 10
withdraw 8
```

OUTPUT 2

```
plain=0 fee=10
plain=0 fee=0
rejected plain=1 fee=1
```

REFERENCE SOLUTION

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

class Account {
public:
    explicit Account(long long opening) : balance_(opening) {}
    void deposit(long long amount) { balance_ += amount; }
    bool withdraw(long long amount) {
        if (amount > balance_) return false;
        balance_ -= amount;
        return true;
    }
    long long balance() const { return balance_; }
protected:
    long long balance_;
};

class FeeAccount : public Account {
public:
    using Account::Account;                       // FeeAccount(long long) forwards
    static constexpr long long kFee = 2;
    bool withdraw(long long amount) {             // hides Account::withdraw
        return Account::withdraw(amount + kFee);  // the base version, by qualified
    }
};

int main() {
    long long opening = 0;
    std::cin >> opening;
    Account plain(opening);
    FeeAccount charged(opening);
    int rejectedPlain = 0;
    int rejectedFee = 0;
    std::string command;
    long long amount = 0;
    while (std::cin >> command >> amount) {
        if (command == "deposit") {
            plain.deposit(amount);
            charged.deposit(amount);
        } else if (command == "withdraw") {
            if (!plain.withdraw(amount)) ++rejectedPlain;
            if (!charged.withdraw(amount)) ++rejectedFee;
        }
        std::cout << "plain=" << plain.balance() << " fee=" << charged.balance() <<
    }
    std::cout << "rejected plain=" << rejectedPlain << " fee=" << rejectedFee << '\r
    return 0;
}
```

QUIZ

**1.** What does this print?

```
struct A { A() { std::cout << "A"; } };
struct M { M() { std::cout << "M"; } };
struct B : A {
    M m;
    B() { std::cout << "B"; }
};
int main() { B b; }
```

A. `AMB`

B. `ABM`

C. `MAB`

D. `BAM`

**2.** `class D : B { … };` — with no access specifier before `B`, what kind of inheritance is this?

A. Public, as in Java's `extends`

B. Private — the default for `class`, as for members

C. Protected

D. A compile error: the specifier is required

**3.** `class Base { public: Base(int); };` and `class Derived : public Base { public: Derived() {} };`. What happens?

A. It compiles; the base is left uninitialised

B. It compiles; `Base(0)` is called implicitly

C. Compile error: the base has no default constructor and the initialiser list does not call `Base(int)`

D. Compile error: a derived class cannot have a default constructor

**4.** What does `using Vehicle::Vehicle;` inside `class Taxi : public Vehicle` do?

A. Makes `Vehicle`'s private members accessible to `Taxi`

B. Declares constructors in `Taxi` matching every `Vehicle` constructor, each forwarding to the base

C. Calls the `Vehicle` constructor before every `Taxi` member function

D. Brings `Vehicle`'s virtual functions into `Taxi`'s scope

**5.** What does this print?

```
struct B { void f(int) { std::cout << "int"; } };
struct D : B { void f(double) { std::cout << "double"; } };
int main() { D d; d.f(3); }
```

A. `int` — an exact match beats a conversion

B. `double` — `D::f` hides every `B::f`, and `3` converts to `double`

C. Compile error: ambiguous call

D. `intdouble`

**6.** Which statement about `protected` members is correct?

A. They are accessible anywhere in the same translation unit

B. They are accessible in derived classes' member functions, but only through an object of the derived type

C. They are accessible in derived classes through any `Base&`

D. They are private to the base and inaccessible to derived classes

**7.** In what order are the parts of a `Derived` object destroyed?

A. Base subobject, then members, then the derived destructor body

B. Derived destructor body, then the derived members in reverse declaration order, then the base

C. Members first, then base, then the derived destructor body

D. The order is unspecified

ANSWER KEY · INHERITANCE BASICS — DERIVING, PROTECTED AND THE ORDER OF CONSTRUCTION

**1. A** The base subobject is constructed first, then the derived class's members in declaration order, then the derived constructor body: `A`, then `M`, then `B`. The body always runs last, whatever the initialiser list says.

**2. B** For a `class` the default is private, both for members and for bases, so every inherited member becomes private to `D` and a `D` cannot be passed as a `B&`. `struct` defaults to public. Write `public` explicitly when you mean is-a.

**3. C** When the initialiser list omits the base, the compiler inserts a call to the base's default constructor. `Base` has only `Base(int)`, so there is none to call and the derived constructor fails to compile. The fix is `Derived() : Base(0) {}`.

**4. B** Constructors are not inherited by default; the `using` declaration inherits them, so `Taxi t("cab")` works if `Vehicle(std::string)` exists. Derived members get their default member initialisers. It has nothing to do with private access or virtual functions.

**5. B** Name lookup stops at the first class walking upward that declares `f`, which is `D`; overload resolution then only sees `D::f(double)`, and `3` converts. `B::f(int)` is hidden, not overloaded. `using B::f;` in `D` would make both visible and select `int`.

**6. B** A derived member function may read a protected member of `*this` or of another object of its own type, but not through a plain `Base&` — the rule stops a derived class from poking at unrelated siblings' state. Free functions and `main` never see protected members.

**7. B** Destruction is the exact reverse of construction (base, members in declaration order, body). The base is still alive during the derived destructor body, which is why it may still use base members; by the time the base destructor runs, the derived part is gone.

LESSON 2 · 15 MIN

# Virtual functions — static type, dynamic type and override

A `Car` bound to a `Vehicle&` is still a `Car`, but which `describe()` runs when you call it through the reference? Without `virtual`, the one `Vehicle` declared — the compiler chose it from the reference's type before the program ran. With `virtual`, the one `Car` provides — chosen at run time from the object's actual type. That single keyword is the difference between inheritance as code reuse and inheritance as polymorphism. This lesson covers what it changes: the two types every expression has, `override` and `final`, calling through a base reference, reaching the base version, and the two places where dispatch does not behave the way intuition says — default arguments and constructors.

## Static type and dynamic type

```
Car car("mini", 3);
Vehicle& v = car;          // static type: Vehicle&   dynamic type: Car
Vehicle* p = &car;         // static type: Vehicle*   dynamic type: Car
Vehicle copy = car;        // static and dynamic type: Vehicle (slicing — lesso
```

The **static type** is the declared type of the expression, known at compile time. The **dynamic type** is the type of the object it actually refers to, known only at run time and only meaningful for references and pointers — an object itself is exactly its declared type. A non-virtual call is resolved from the static type; a virtual call is resolved from the dynamic type. This is why polymorphism in C++ happens through references and pointers and never through values.

## virtual

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

class Vehicle {
public:
    virtual ~Vehicle() = default;                                   // lesso
    virtual std::string describe() const { return "a vehicle"; }
    std::string plate() const { return "generic"; }                 // not v
};

class Car : public Vehicle {
public:
    std::string describe() const override { return "a car"; }
    std::string plate() const { return "car-plate"; }               // hides
};

void show(const Vehicle& v) {
    std::cout << v.describe() << " / " << v.plate() << '\n';
}

int main() {
    Car car;
    show(car);                          // a car / generic
    std::cout << car.plate() << '\n';   // car-plate — on the object itself th
}
```

`describe` is virtual, so `v.describe()` consults the dynamic type and finds `Car::describe`. `plate` is not, so `v.plate()` is bound at compile time to `Vehicle::plate`; `Car::plate` merely *hides* it (lesson 1) and is reached only through a `Car`. A function declared `virtual` in a base stays virtual in every derived class whether or not the keyword is repeated — `Car::describe` is virtual, and a `SportsCar::describe` would override it in turn.

Two costs come with the keyword. A virtual call is an indirect call through a table the compiler builds for the class (lesson 5 shows the mechanism), which the optimiser usually cannot inline; and every object of a class with at least one virtual function carries a hidden pointer to that table. Neither is large. Both are why the standard containers have no virtual functions and why `virtual` is a design decision rather than a default.

---

## override and final

`override` says "this function overrides a virtual function in a base", and the compiler checks the claim. Without it, a mistake in the signature silently declares a *new* function, and the base version keeps running:

```
class Vehicle {
public:
    virtual std::string describe() const;
};
class Car : public Vehicle {
public:
    std::string describe() { return "a car"; }            // no const: a new f
    std::string describe() const override;                 // the real overrid
    std::string descrbe() const override;                  // error: marked 'o
    std::string describe(int) const override;              // error: marked 'o
};
```

A missing `const`, a typo, an extra parameter, a different return type — each compiles without `override` and fails quietly at run time. With it, every one is a compile error. Mark every override; GCC and Clang offer `-Wsuggest-override` to enforce the habit.

`final` closes the door: on a function, no further derived class may override it; on a class, nothing may derive from it at all.

```
class Taxi final : public Car {
public:
    std::string describe() const final { return "a taxi"; }
};
class Cab : public Taxi {};                                // error: cannot de
```

`final` documents a design decision, and it lets the compiler replace some virtual calls with direct ones, since it knows no override can exist below.

---

## Calling the base version

An override often extends rather than replaces. The base version is reached with a qualified name, which is always a non-virtual call:

```
class Car : public Vehicle {
public:
    std::string describe() const override {
        return Vehicle::describe() + " with four wheels";      // "a vehicle wi
    }
};
```

Forget the qualifier — `return describe() + …` — and the function calls itself until the stack is exhausted.

---

## Default arguments are bound statically

```
struct Base {
    virtual void greet(const std::string& who = "base") const { std::cout << "
};
struct Derived : Base {
    void greet(const std::string& who = "derived") const override { std::cout
};

Derived d;
Base& b = d;
b.greet();          // Derived greets base
```

The function chosen is `Derived::greet` — dynamic dispatch — but the default argument comes from the static type, `Base`, because default arguments are filled in by the compiler at the call site, which only knows the static type. The result is a call nobody wrote. Rule: never redefine default arguments in an override; give them in the base only, or use an overload instead.

---

## Virtual calls in constructors and destructors

```
struct Base {
    Base() { std::cout << name() << '\n'; }
    virtual ~Base() = default;
    virtual std::string name() const { return "Base"; }
};
struct Derived : Base {
    std::string name() const override { return "Derived"; }
};

Derived d;          // prints Base
```

During `Base`'s constructor the object *is* a `Base`: the derived part has not been constructed yet, and C++ sets the dynamic type stage by stage as construction proceeds (lesson 5 explains how). So the virtual call resolves to `Base::name`, not to the override. Java does the opposite — it calls the override on a subclass whose fields are still zero — and C++'s choice is the safer one, but it surprises people arriving from Java. The same applies in destructors, in the other direction: by the time `~Base` runs the derived part is gone and the dynamic type is `Base` again. A call to a *pure* virtual function (lesson 3) from a constructor has no body to reach and terminates the program.

If a derived class needs a hook to run after construction, provide a separate `init()` the creator calls, or use a factory function (Module 8, lesson 2) that constructs the object and then calls it.

---

## Pitfalls

| Mistake | What happens |
| --- | --- |
| Calling a virtual function on a value rather than a reference or pointer | The static type is the dynamic type; no dispatch — and a copy may have sliced. |
| Override without `override` and a signature that differs | A new, hidden function; the base version keeps running. |
| Different default arguments in an override | The base's default with the derived body. |
| Virtual call in a constructor expecting the derived version | The base version runs. |
| `describe()` instead of `Vehicle::describe()` inside the override | Infinite recursion. |
| Deleting through a base pointer with a non-virtual destructor | Undefined behaviour — lesson 4. |

## Key takeaways

Static type: what the expression is declared as. Dynamic type: what the object really is. Non-virtual calls use the first, virtual calls the second.

Polymorphism happens only through references and pointers; a value has no separate dynamic type.

`override` turns a mismatched signature into a compile error; `final` forbids further overriding or deriving.

`Base::f()` calls the base version non-virtually — the way an override extends behaviour.

Default arguments are chosen from the static type; virtual calls inside constructors and destructors resolve to the class being constructed.

EXERCISE 1

## Which function ran

Write `Animal` with a **virtual** `std::string sound() const` returning `Animal::sound` and a **non-virtual** `std::string describe() const` returning `Animal::describe`. Write `Dog` and `Cat` deriving publicly from it; each **overrides** `sound` (marked `override`) and also declares its own `describe` —

which, being non-virtual, only hides the base's. Every function returns its own qualified name as a string, so the output tells you which one ran.

The starter's `viaBase` takes a `const Animal&` and calls both functions through it. For each kind, print one line through the base reference and one line calling directly on the object.

**Input:** `n`, then `n` kinds: `dog`, `cat` or `animal`. **Output:** two lines per kind.

```
2
dog
animal
```

prints

```
dog via Animal&: Dog::sound, Animal::describe
dog direct: Dog::sound, Dog::describe
animal via Animal&: Animal::sound, Animal::describe
animal direct: Animal::sound, Animal::describe
```

INPUT 1

```
2
dog
animal
```

OUTPUT 1

```
dog via Animal&: Dog::sound,
Animal::describe
dog direct: Dog::sound, Dog::describe
animal via Animal&: Animal::sound,
Animal::describe
animal direct: Animal::sound,
Animal::describe
```

INPUT 2

```
1
cat
```

OUTPUT 2

```
cat via Animal&: Cat::sound,
Animal::describe
cat direct: Cat::sound, Cat::describe
```

REFERENCE SOLUTION

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

class Animal {
public:
    virtual ~Animal() = default;
    virtual std::string sound() const { return "Animal::sound"; }
    std::string describe() const { return "Animal::describe"; }     // not virtual:
};

class Dog : public Animal {
public:
    std::string sound() const override { return "Dog::sound"; }
    std::string describe() const { return "Dog::describe"; }        // hides Animal:
};

class Cat : public Animal {
public:
    std::string sound() const override { return "Cat::sound"; }
    std::string describe() const { return "Cat::describe"; }
};

void viaBase(const std::string& kind, const Animal& a) {
    std::cout << kind << " via Animal&: " << a.sound() << ", " << a.describe() << '\
}

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string kind;
        std::cin >> kind;
        if (kind == "dog") {
            Dog d;
            viaBase(kind, d);
            std::cout << kind << " direct: " << d.sound() << ", " << d.describe() <<
        } else if (kind == "cat") {
            Cat c;
            viaBase(kind, c);
            std::cout << kind << " direct: " << c.sound() << ", " << c.describe() <<
        } else {
            Animal a;
            viaBase(kind, a);
            std::cout << kind << " direct: " << a.sound() << ", " << a.describe() <<
        }
    }
    return 0;
}
```

EXERCISE 2

## Pricing policies with override and final

Write a small hierarchy of pricing policies and apply each one to a list of prices.

- `Pricing` has a virtual destructor and `virtual long long price(long long base) const` returning `base` unchanged.
- `Percent(int p)` overrides it: `base - base * p / 100` (integer arithmetic).
- `Flat(long long off)` overrides it: `base - off`, but never below 0.
- `Clamped(int p, long long floor)` derives from `Percent`, is marked `final`, and overrides `price` as the larger of `Percent::price(base)` — call the base version by its qualified name — and `floor`.

Mark every override `override`. The starter's `makePolicy` returns a `std::unique_ptr<Pricing>` for a command line; complete it.

**Input:** `m`, then a line of `m` prices, then one policy per line until the input ends: `none`, `percent p`, `flat off` or `clamped p floor`. **Output:** one line per policy: the policy line as given, a colon, then each price after the policy.

```
3
100 250 40
none
percent 10
flat 30
clamped 50 30
```

prints

```
none: 100 250 40
percent 10: 90 225 36
flat 30: 70 220 10
clamped 50 30: 50 125 30
```

INPUT 1

```
3
100 250 40
none
percent 10
flat 30
clamped 50 30
```

OUTPUT 1

```
none: 100 250 40
percent 10: 90 225 36
flat 30: 70 220 10
clamped 50 30: 50 125 30
```

### INPUT 2

```
2
7 1000
percent 33
flat 8
```

### OUTPUT 2

```
percent 33: 5 670
flat 8: 0 992
```

### REFERENCE SOLUTION

```
#include <algorithm>
#include <iostream>
#include <memory>
#include <sstream>
#include <string>
#include <vector>

class Pricing {
public:
    virtual ~Pricing() = default;
    virtual long long price(long long base) const { return base; }
};

class Percent : public Pricing {
public:
    explicit Percent(int percent) : percent_(percent) {}
    long long price(long long base) const override { return base - base * percent_ /
private:
    int percent_;
};

class Flat : public Pricing {
public:
    explicit Flat(long long off) : off_(off) {}
    long long price(long long base) const override { return std::max(0LL, base - off
private:
    long long off_;
};

class Clamped final : public Percent {
public:
    Clamped(int percent, long long floor) : Percent(percent), floor_(floor) {}
    long long price(long long base) const override {
        return std::max(Percent::price(base), floor_);    // the base version, non-v
    }
private:
    long long floor_;
};

std::unique_ptr<Pricing> makePolicy(const std::string& line) {
    std::istringstream in(line);
    std::string kind;
    in >> kind;
    if (kind == "percent") {
        int p = 0;
        in >> p;
        return std::make_unique<Percent>(p);
    }
    if (kind == "flat") {
```

```
        long long off = 0;
        in >> off;
        return std::make_unique<Flat>(off);
    }
    if (kind == "clamped") {
        int p = 0;
        long long floor = 0;
        in >> p >> floor;
        return std::make_unique<Clamped>(p, floor);
    }
    return std::make_unique<Pricing>();
}

int main() {
    int m = 0;
    std::cin >> m;
    std::vector<long long> prices(m);
    for (auto& p : prices) std::cin >> p;
    std::string line;
    std::getline(std::cin, line);                  // the rest of the prices line
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        const auto policy = makePolicy(line);
        std::cout << line << ':';
        for (long long p : prices) std::cout << ' ' << policy->price(p);
        std::cout << '\n';
    }
    return 0;
}
```

---

QUIZ

**1.** What does this print?

```
struct B { void hello() const { std::cout << "B"; } };
struct D : B { void hello() const { std::cout << "D"; } };
int main() {
    D d;
    const B& r = d;
    r.hello();
}
```

A. `D` — the object is a `D`

B. `B` — `hello` is not virtual, so the static type `B` decides

C. Compile error: `D::hello` needs `override`

D. Undefined behaviour

**2.** `struct B { void f(); }; struct D : B { void f() override; };` — what happens?

A. It compiles and `D::f` overrides `B::f`

B. Compile error: `override` requires a virtual function in the base to override

C. It compiles with a warning

D. Undefined behaviour at run time

**3.** What does this print?

```
struct B { virtual void f(int x = 1) const { std::cout << "B" << x; } };
struct D : B { void f(int x = 2) const override { std::cout << "D" << x; } };
int main() {
    D d;
    const B& r = d;
    r.f();
}
```

A. `B1`

B. `D2`

C. `D1`

D. Compile error: an override cannot change a default argument

**4.** What does this print?

```
struct B {
    B() { init(); }
    virtual ~B() = default;
    virtual void init() { std::cout << "B::init"; }
};
struct D : B { void init() override { std::cout << "D::init"; } };
int main() { D d; }
```

A. `D::init`

B. `B::init`

C. Nothing — virtual calls are disabled in constructors

D. Undefined behaviour

**5.** Which statement about `final` is correct?

A. `final` on a class makes all its member functions `const`

B. `final` on a virtual function prevents further overriding; on a class it prevents deriving; both are enforced at compile time

C. `final` on a function prevents it from being called through a base pointer

D. `final` is only a hint and has no effect on compilation

**6.** Compared with a non-virtual call, what does a virtual call cost?

A. A hash-table lookup of the function name at run time

B. One indirect call through the class's table of function pointers, and the optimiser usually cannot inline it

C. A linear search up the class hierarchy for an override

D. Nothing — the compiler resolves it statically

ANSWER KEY · VIRTUAL FUNCTIONS — STATIC TYPE, DYNAMIC TYPE AND OVERRIDE

**1. B** Without `virtual`, the call is bound at compile time from the static type of `r`, which is `const B&`. `D::hello` merely hides `B::hello`. Add `virtual` in `B` (and `override` in `D`) to get `D`.

**2. B** `override` makes the compiler check that the function overrides a virtual function in some base. `B::f` is not virtual, so the claim is false and the build fails — exactly the mistake the keyword exists to catch.

**3. C** The function is chosen dynamically (`D::f`), but the default argument is filled in at the call site from the static type `B`, giving `x = 1`. Changing defaults in an override is legal and is the bug.

**4. B** During `B`'s constructor the dynamic type of the object is `B` — the `D` part does not exist yet — so the virtual call resolves to `B::init`. This is well-defined, unlike calling a *pure* virtual function from a constructor.

**5. B** `void f() final` refuses any override in a further-derived class, and `class X final` refuses any derived class — both as compile errors. The compiler may also devirtualise calls to a `final` function since no override can exist.

**6. B** The compiler emits a load of the object's hidden vptr, a load of the slot for that function, and an indirect call. There is no name lookup at run time. The cost is small but real, and it is why `virtual` is opt-in.

LESSON 3 · 14 MIN

# Abstract classes and interfaces — programming to a contract

Some base classes exist only to be derived from. A `Shape` has an area, but there is no formula for the area of "a shape" — only of a circle or a rectangle. C++ expresses that with a *pure virtual function*: a function the base declares and refuses to define, which makes the class *abstract* and forces every concrete derived class to supply the body. This lesson covers the syntax, what an abstract class can and cannot do, the interface class as C++'s version of Java's `interface`, a `Shape` hierarchy used the way library code uses one, and the container that holds a mixed collection of shapes without losing anything: `std::vector<std::unique_ptr<Shape>>`.

## Pure virtual functions

```
#include <algorithm>
#include <iostream>
#include <memory>
#include <numbers>
#include <string>
#include <vector>

class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual std::string name() const = 0;
};
```

`= 0` after the declaration marks the function *pure virtual*: the base has no implementation and a derived class must provide one. A class with at least one pure virtual function is **abstract**, and the compiler refuses to create an object of it:

```
Shape s;                              // error: cannot declare variable 's' to
Shape* p = new Shape;                 // same error
```

References and pointers to `Shape` are fine — that is the whole point — so `const Shape&` parameters and `std::unique_ptr<Shape>` members work as usual. A derived class that overrides *every* pure virtual function is concrete and can be instantiated; one that leaves any of them out is abstract too, and the error message when you try to create it names the missing function, which is the fastest way to find out what an interface requires.

A pure virtual function may still have a body, defined outside the class — `double Shape::area() const { return 0; }` — which derived overrides can call as `Shape::area()`. The class stays abstract. The common use is a pure virtual destructor, a way to make a class abstract when no other function is naturally pure.

---

## Concrete shapes

```
class Circle : public Shape {
public:
    explicit Circle(double r) : r_(r) {}
    double area() const override { return std::numbers::pi * r_ * r_; }
    double perimeter() const override { return 2 * std::numbers::pi * r_; }
    std::string name() const override { return "circle"; }
private:
    double r_;
};

class Rect : public Shape {
public:
    Rect(double w, double h) : w_(w), h_(h) {}
    double area() const override { return w_ * h_; }
    double perimeter() const override { return 2 * (w_ + h_); }
    std::string name() const override { return "rect"; }
private:
    double w_, h_;
};
```

Each concrete class carries its own representation — a radius, a width and a height — and answers the three questions in its own way. Nothing in `Shape` knows either representation exists. `override` on each function makes the compiler confirm that the concrete class really satisfies the contract, rather than silently adding a differently spelled function and remaining abstract.

## Interface classes

An abstract class with no data members and nothing but pure virtual functions (plus a virtual destructor) is an **interface class** — what Java calls an `interface`. `Shape` above is one. The convention matters because such a class promises nothing about representation: any type that can answer `area`, `perimeter` and `name` can be a `Shape`, whether it stores a radius, a polygon's vertices, or a handle into a graphics library.

```
class Drawable {
public:
    virtual ~Drawable() = default;
    virtual void draw(std::ostream& out) const = 0;
};
class Serializable {
public:
    virtual ~Serializable() = default;
    virtual std::string serialize() const = 0;
};

class Sprite : public Drawable, public Serializable { /* implements both */ };
```

A class may implement several interfaces by deriving from all of them — the one form of multiple inheritance (lesson 6) that is routine in C++, because interfaces carry no state and so cannot collide.

## Programming to the interface

```
void report(const Shape& s) {
    std::cout << s.name() << ": area " << s.area() << ", perimeter " << s.peri
}

double largest(const std::vector<std::unique_ptr<Shape>>& shapes) {
    double best = 0;
    for (const auto& s : shapes) best = std::max(best, s->area());
    return best;
}
```

`report` and `largest` compile once and work for every shape that will ever be written — a `Triangle` added next year needs no change here. That is the *open–closed* property

polymorphism exists for: code is open for extension by new derived classes and closed for modification. Functions should take the most abstract type that gives them what they need, which is usually `const Shape&`; taking `const Circle&` where only `area()` is used couples the function to one representation for no gain.

---

## A container of shapes

`std::vector<Shape>` is impossible — `Shape` is abstract — and would be wrong even for a concrete base, because copying a `Circle` into a `Shape` slot keeps only the `Shape` part (lesson 4). A mixed collection needs indirection, and the owning indirection is `std::unique_ptr` (Module 7, lesson 4):

```
std::unique_ptr<Shape> makeShape(const std::string& kind, std::istream& in) {
    if (kind == "circle") {
        double r;
        in >> r;
        return std::make_unique<Circle>(r);
    }
    double w, h;
    in >> w >> h;
    return std::make_unique<Rect>(w, h);
}

int main() {
    std::vector<std::unique_ptr<Shape>> shapes;
    std::string kind;
    while (std::cin >> kind) shapes.push_back(makeShape(kind, std::cin));
    for (const auto& s : shapes) report(*s);
    std::cout << "largest: " << largest(shapes) << '\n';
    return 0;
}
```

`std::make_unique<Circle>(r)` yields a `std::unique_ptr<Circle>`, which converts implicitly to `std::unique_ptr<Shape>` — the same upcast as `Circle*` to `Shape*`, carried through the smart pointer. The vector owns every shape; when it is destroyed, each `unique_ptr` deletes its object through a `Shape*`, which is exactly why `Shape` needs its virtual destructor (lesson 4). The factory `makeShape` is the one place that knows the concrete types; everything after it sees `Shape`. Iterate with `const auto&` — copying a `unique_ptr` is a compile error, and the reference is what you wanted anyway.

When shared ownership is genuinely required, `std::shared_ptr<Shape>` works the same way; when the container does not own the shapes at all, a `std::vector<Shape*>` observes objects that live elsewhere. Owning is the common case, and `unique_ptr` is the default.

---

## Non-virtual interface

A refinement worth knowing: keep the public function non-virtual and let it call a private virtual one.

```
class Shape {
public:
    double area() const { return computeArea(); }       // public, non-virtual
private:
    virtual double computeArea() const = 0;              // derived classes ov
};
```

Derived classes override `computeArea` — a private virtual can be overridden, because access and virtuality are separate rules — while the public `area` stays under the base's control, free to add checks around the call. The idiom is called NVI. A hierarchy with pre- and post-conditions benefits from it; a plain `Shape` does not need it.

---

## Pitfalls

| Mistake | What happens |
| --- | --- |
| Creating an abstract class by value (`Shape s;`, `std::vector<Shape>`) | Compile error naming the pure function. |
| Overriding with a different signature and no `override` | The class stays abstract; the error appears where it is instantiated. |
| Calling a pure virtual function from the base constructor | No body to reach; the program terminates. |
| Returning `Shape` by value from a factory | Impossible for an abstract base; return `std::unique_ptr<Shape>`. |
| `for (auto s : shapes)` over a vector of `unique_ptr` | Tries to copy a `unique_ptr`; use `const auto&`. |
| Omitting the virtual destructor in the interface | Deleting through `unique_ptr<Shape>` is undefined behaviour. |

## Key takeaways

- `= 0` makes a function pure virtual and its class abstract: no objects, only references and pointers.
- A derived class is concrete once it overrides every pure virtual function; miss one and it is abstract too.
- An interface class has only pure virtual functions and a virtual destructor; a class may implement several.
- Take `const Base&` in functions and let the dynamic type decide — new derived classes need no changes.
- Hold a mixed collection as `std::vector<std::unique_ptr<Base>>`, built by a factory that returns `std::unique_ptr<Base>`.

EXERCISE 1

## A catalogue of shapes

Write an abstract `Shape` with a virtual destructor and three pure virtual functions: `std::string name() const`, `double area() const` and `double perimeter() const`. Derive `Circle(r)`,

`Rect(w, h)` and `Triangle(a, b, c)` (Heron's formula: with `s = (a + b + c) / 2`, the area is `sqrt(s (s - a) (s - b) (s - c))`). Use `std::numbers::pi`.

Read shapes until the input ends into a `std::vector<std::unique_ptr<Shape>>`, then print each shape through the base interface and a summary. Print every number with `std::fixed` and two decimals.

**Input:** lines of `circle r`, `rect w h` or `triangle a b c` (whitespace-separated; the input may be empty). **Output:** one line per shape, `<name>: area=<a> perimeter=<p>`, then `count=<n> total area=<sum>`.

```
circle 1
rect 2 3
triangle 3 4 5
```

prints

```
circle: area=3.14 perimeter=6.28
rect: area=6.00 perimeter=10.00
triangle: area=6.00 perimeter=12.00
count=3 total area=15.14
```

INPUT 1

```
circle 1
rect 2 3
triangle 3 4 5
```

OUTPUT 1

```
circle: area=3.14 perimeter=6.28
rect: area=6.00 perimeter=10.00
triangle: area=6.00 perimeter=12.00
count=3 total area=15.14
```

INPUT 2

```
rect 1.5 2
circle 2.5
```

OUTPUT 2

```
rect: area=3.00 perimeter=7.00
circle: area=19.63 perimeter=15.71
count=2 total area=22.63
```

REFERENCE SOLUTION

```cpp
#include <cmath>
#include <iomanip>
#include <iostream>
#include <memory>
#include <numbers>
#include <string>
#include <vector>

class Shape {
public:
    virtual ~Shape() = default;
    virtual std::string name() const = 0;
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
};

class Circle : public Shape {
public:
    explicit Circle(double r) : r_(r) {}
    std::string name() const override { return "circle"; }
    double area() const override { return std::numbers::pi * r_ * r_; }
    double perimeter() const override { return 2 * std::numbers::pi * r_; }
private:
    double r_;
};

class Rect : public Shape {
public:
    Rect(double w, double h) : w_(w), h_(h) {}
    std::string name() const override { return "rect"; }
    double area() const override { return w_ * h_; }
    double perimeter() const override { return 2 * (w_ + h_); }
private:
    double w_;
    double h_;
};

class Triangle : public Shape {
public:
    Triangle(double a, double b, double c) : a_(a), b_(b), c_(c) {}
    std::string name() const override { return "triangle"; }
    double area() const override {
        const double s = (a_ + b_ + c_) / 2;
        return std::sqrt(s * (s - a_) * (s - b_) * (s - c_));
    }
    double perimeter() const override { return a_ + b_ + c_; }
private:
    double a_;
    double b_;
```

```
    double c_;
};

int main() {
    std::vector<std::unique_ptr<Shape>> shapes;
    std::string kind;
    while (std::cin >> kind) {
        if (kind == "circle") {
            double r = 0;
            std::cin >> r;
            shapes.push_back(std::make_unique<Circle>(r));
        } else if (kind == "rect") {
            double w = 0, h = 0;
            std::cin >> w >> h;
            shapes.push_back(std::make_unique<Rect>(w, h));
        } else if (kind == "triangle") {
            double a = 0, b = 0, c = 0;
            std::cin >> a >> b >> c;
            shapes.push_back(std::make_unique<Triangle>(a, b, c));
        }
    }
    std::cout << std::fixed << std::setprecision(2);
    double total = 0;
    for (const auto& s : shapes) {
        std::cout << s->name() << ": area=" << s->area() << " perimeter=" << s->peri
        total += s->area();
    }
    std::cout << "count=" << shapes.size() << " total area=" << total << '\n';
    return 0;
}
```

EXERCISE 2

## An encoder pipeline

Define an interface class `Encoder` — a virtual destructor and one pure virtual `std::string encode(const std::string& text) const` — and four implementations: `upper` (every letter upper-cased), `lower`, `reverse` (the characters reversed) and `rot13` (each letter shifted 13 places within its case; other characters unchanged). Use `<cctype>` with the `unsigned char` cast.

Build the pipeline named on the first line as a `std::vector<std::unique_ptr<Encoder>>` through a factory `std::unique_ptr<Encoder> makeEncoder(const std::string& name)` that returns `nullptr` for a name it does not know. Then push every remaining line through the pipeline in order.

**Input:** a line of encoder names, then text lines until the input ends. **Output:** for each unknown name, `unknown encoder: <name>` (while building the pipeline); then for each text line, `<text> →`

`<encoded>`.

```
upper rot13
hello world
Abc xyz
```

prints

```
hello world → URYYB JBEYQ
Abc xyz → NOP KLM
```

INPUT 1

```
upper rot13
hello world
Abc xyz
```

OUTPUT 1

```
hello world → URYYB JBEYQ
Abc xyz → NOP KLM
```

INPUT 2

```
reverse lower
Stressed
```

OUTPUT 2

```
Stressed → desserts
```

REFERENCE SOLUTION

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

class Encoder {
public:
    virtual ~Encoder() = default;
    virtual std::string encode(const std::string& text) const = 0;
};

class Upper : public Encoder {
public:
    std::string encode(const std::string& text) const override {
        std::string out = text;
        for (char& c : out) c = static_cast<char>(std::toupper(static_cast<unsigned
        return out;
    }
};

class Lower : public Encoder {
public:
    std::string encode(const std::string& text) const override {
        std::string out = text;
        for (char& c : out) c = static_cast<char>(std::tolower(static_cast<unsigned
        return out;
    }
};

class Reverse : public Encoder {
public:
    std::string encode(const std::string& text) const override {
        return std::string(text.rbegin(), text.rend());
    }
};

class Rot13 : public Encoder {
public:
    std::string encode(const std::string& text) const override {
        std::string out = text;
        for (char& c : out) {
            if (c >= 'a' && c <= 'z') c = static_cast<char>('a' + (c - 'a' + 13) % 2
            else if (c >= 'A' && c <= 'Z') c = static_cast<char>('A' + (c - 'A' + 13
        }
        return out;
    }
};
```

```cpp
std::unique_ptr<Encoder> makeEncoder(const std::string& name) {
    if (name == "upper") return std::make_unique<Upper>();
    if (name == "lower") return std::make_unique<Lower>();
    if (name == "reverse") return std::make_unique<Reverse>();
    if (name == "rot13") return std::make_unique<Rot13>();
    return nullptr;
}

int main() {
    std::string line;
    std::getline(std::cin, line);
    std::istringstream names(line);
    std::vector<std::unique_ptr<Encoder>> pipeline;
    std::string name;
    while (names >> name) {
        auto encoder = makeEncoder(name);
        if (encoder) pipeline.push_back(std::move(encoder));
        else std::cout << "unknown encoder: " << name << '\n';
    }
    while (std::getline(std::cin, line)) {
        std::string out = line;
        for (const auto& e : pipeline) out = e->encode(out);
        std::cout << line << " -> " << out << '\n';
    }
    return 0;
}
```

---

QUIZ

1. `class Shape { public: virtual double area() const = 0; };` and then `Shape s;`. What happens?

   A. It compiles; `s.area()` returns 0

   B. It compiles; `s.area()` throws at run time

   C. Compile error: cannot declare a variable of abstract type `Shape`

   D. Undefined behaviour

**2.** `Shape` has two pure virtual functions. `class Circle : public Shape` overrides only one of them. Which is true?

A. `Circle` is concrete; the missing function defaults to the base's

B. `Circle` is still abstract; creating a `Circle` is a compile error naming the missing function

C. `Circle` compiles and the missing function returns 0

D. It is a compile error at the class definition

**3.** Can a pure virtual function have a body?

A. No — `= 0` means there is no implementation anywhere

B. Yes, defined outside the class; derived classes can call it as `Base::f()`, and the class remains abstract

C. Yes, and then the class is no longer abstract

D. Only for destructors

**4.** Which describes an *interface class* in C++?

A. A class with only `static` functions

B. An abstract class with pure virtual functions, a virtual destructor and no data members

C. Any class declared with the `struct` keyword

D. A class template with no member functions

**5.** `std::vector<std::unique_ptr<Shape>> v; v.push_back(std::make_unique<Circle>(1.0));` — does this compile?

A. No: `unique_ptr<Circle>` and `unique_ptr<Shape>` are unrelated types

B. Yes: `unique_ptr<Circle>` converts implicitly to `unique_ptr<Shape>` because `Circle*` converts to `Shape*`

C. Only with an explicit `static_cast`

D. Only if `Shape` is not abstract

**6.** Why is `std::vector<Shape>` the wrong container for a mixed collection of shapes?

A. Vectors cannot hold class types with virtual functions

B. `Shape` is abstract so no element can exist — and even for a concrete base, each element would be a copy holding only the base part

C. It would work but be slower than `std::vector<std::unique_ptr<Shape>>`

D. It compiles only if `Shape` has a default constructor

ANSWER KEY · ABSTRACT CLASSES AND INTERFACES — PROGRAMMING TO A CONTRACT

**1. C** A class with a pure virtual function is abstract and no object of it can be created, by value or with `new`. References and pointers to `Shape` are fine — that is how the type is used.

**2. B** Abstractness is inherited until every pure virtual function has been overridden. The class definition itself is fine — abstract intermediate classes are common — but instantiating `Circle` fails, and the message lists the unoverridden function.

**3. B** `= 0` says derived classes must override; it does not forbid a definition. `double Shape::area() const { … }` can be supplied and called by name. The common case is a pure virtual destructor, which must have a body because it is always called.

**4. B** C++ has no `interface` keyword; the convention is an abstract class that promises behaviour and nothing about representation. A class may implement several by multiple inheritance, safely, because interfaces carry no state.

**5. B** `std::unique_ptr` has a converting move constructor for any pointer type that converts implicitly, so the upcast carries through the smart pointer. The temporary is moved into the vector; ownership transfers.

**6. B** The element type must be complete and constructible, which an abstract class is not. For a concrete base the vector compiles but slices every derived object pushed into it (lesson 4). Polymorphism needs indirection: `unique_ptr<Shape>` elements.

LESSON 4 · 14 MIN

# Virtual destructors and slicing — the two ways a hierarchy loses data

Two bugs follow polymorphic classes everywhere. The first: an object created as a `Derived` and deleted through a `Base*` runs only the base destructor — unless that destructor is virtual — so the derived part is never cleaned up, and the standard calls the whole operation undefined behaviour. The second: an object copied *into* a `Base` — a by-value parameter, an element of a `std::vector<Base>`, a plain `Base b = derived;` — keeps only the base part, silently. Neither is a compile error. This lesson explains both, shows what the compiler and the sanitizers say, and gives the rules that make them impossible: a virtual destructor on every polymorphic base, and references, pointers or `unique_ptr`s wherever a derived object travels.

---

## Deleting through a base pointer

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

class Resource {
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource released\n"; }          // not virtu
};

class Logged : public Resource {
public:
    Logged() : log_(1000, 'x') { std::cout << "Logged opened\n"; }
    ~Logged() { std::cout << "Logged closed\n"; }
private:
    std::string log_;
};

int main() {
    Resource* r = new Logged;
    delete r;                    // undefined behaviour: static type Resource,
}
```

In practice, on this platform, this prints `Resource acquired`, `Logged opened`, `Resource released` — and nothing else. `delete r` uses the static type of `r`: it calls `~Resource`, then frees the block as though it held a `Resource`. `~Logged` never runs, and `log_`'s thousand bytes leak. The standard's wording is that deleting an object through a pointer to a base whose destructor is not virtual is **undefined behaviour**; the leak is only the visible part, and AddressSanitizer reports the mismatch as `new-delete-type-mismatch`.

`std::unique_ptr<Resource> p = std::make_unique<Logged>();` has the same problem: `unique_ptr<Resource>` deletes through a `Resource*`. `std::shared_ptr` happens to escape it — `std::make_shared<Logged>()` records the deleter for `Logged` in the control block at creation — but a design that relies on that detail breaks the moment someone changes the pointer type.

---

## The fix: a virtual destructor

```
class Resource {
public:
    virtual ~Resource() { std::cout << "Resource released\n"; }
};
```

A virtual destructor is dispatched like any virtual function: `delete r` now finds `~Logged` through the dynamic type, which runs its body, destroys `log_`, then calls `~Resource` — the reverse of construction (lesson 1), exactly as if the object had been a local `Logged`. Output: `Logged closed`, `Resource released`. When the base has nothing to do in its destructor, write `virtual ~Resource() = default;`. On the derived side, `~Logged() override` is legal and useful: it fails to compile if the base destructor is not virtual.

The rule, stated by the C++ Core Guidelines as C.35: *a base class destructor should be either public and virtual, or protected and non-virtual.* Public and virtual is the normal case — anyone may delete through the base. Protected and non-virtual is for a base that is never deleted polymorphically: nobody outside can `delete` a `Base*`, so the question does not arise and the class avoids a vptr. Practical shortcut: if a class has any virtual function, give it a virtual destructor.

One cost to remember from Module 9: declaring a destructor — even `= default` — suppresses the implicit move constructor and move assignment. A polymorphic base that should be movable declares all five special members, usually all `= default`.

## Slicing

```
class Employee {
public:
    explicit Employee(std::string name) : name_(std::move(name)) {}
    virtual ~Employee() = default;
    virtual std::string role() const { return "employee"; }
protected:
    std::string name_;
};

class Manager : public Employee {
public:
    Manager(std::string name, int reports) : Employee(std::move(name)), report
    std::string role() const override { return "manager of " + std::to_string(
private:
    int reports_;
};

void byValue(Employee e)      { std::cout << e.role() << '\n'; }
void byRef(const Employee& e) { std::cout << e.role() << '\n'; }

int main() {
    Manager m("ada", 5);
    byValue(m);          // employee          — sliced
    byRef(m);            // manager of 5
    Employee copy = m;   // copy is an Employee; reports_ is gone
    std::cout << copy.role() << '\n';   // employee
}
```

`byValue(Employee e)` copy-constructs an `Employee` from `m`. The copy constructor of `Employee` takes a `const Employee&`, to which `m` binds by the usual upcast, and it copies exactly the `Employee` members — `name_` — because that is all an `Employee` has. `reports_` is not copied; there is nowhere to put it. The result is a complete, valid `Employee` whose dynamic type is `Employee`, so `e.role()` calls the base version. This is **object slicing**: the derived part is cut off by a copy into a base-typed value. Nothing is undefined; the program simply and quietly does the wrong thing.

Slicing happens wherever a derived object is copied into a base object:

| Site | What slices |
| --- | --- |
| `void f(Base b)` called with a `Derived` | The parameter. |
| `Base b = derived;` | The initialisation. |
| `std::vector<Base> v;` `v.push_back(derived);` | Each element. |
| `Base& r = base; r = derived;` | The assignment — only the base members are assigned, and `r` keeps its dynamic type. |
| `catch (std::exception e)` | The caught exception (Module 15). |
| `return derived;` from a function returning `Base` | The return value. |

The assignment row is the subtle one: a reference does not slice, but assigning *through* it does, because `Base::operator=` is what runs.

---

## Preventing slicing

The fix is the same in every row: do not copy polymorphic objects into base-typed values. Take parameters as `const Base&` or `Base*`; hold collections as `std::vector<std::unique_ptr<Base>>` (lesson 3); return `std::unique_ptr<Base>` from factories. When a hierarchy must be copied polymorphically, give it a virtual `clone()`:

```
class Employee {
public:
    virtual std::unique_ptr<Employee> clone() const { return std::make_unique<
};
class Manager : public Employee {
public:
    std::unique_ptr<Employee> clone() const override { return std::make_unique
};
```

`clone()` copies the whole dynamic object because each override copies its own type. The compiler can also be asked to forbid the accidental copy outright. An abstract base cannot be copied into by value at all — it cannot exist as a value, so `void f(Shape s)` is itself an error — and for a concrete base the guideline C.67 suggests deleting or protecting the copy

operations: `Employee(const Employee&) = delete;` turns `byValue(m)` into a compile error instead of a wrong answer. Derived classes that need copying for `clone()` declare their own copy constructor and call the base's protected one.

## Pitfalls

| Mistake | What happens |
| --- | --- |
| A virtual function without a virtual destructor | `delete` through the base is undefined behaviour; the derived destructor is skipped. |
| Trusting `shared_ptr` to compensate | It does, until the pointer becomes a `unique_ptr` or a raw `delete`. |
| `void f(Base b)` in a polymorphic hierarchy | Every call slices; virtual calls inside `f` go to the base. |
| `std::vector<Base>` for a mixed collection | Every element is a base; no polymorphism survives. |
| `*base = *derived` or `base = derived` through references | The base part is assigned, the rest ignored. |
| Declaring `~Base()` and forgetting the moves | Implicit move is suppressed; copies happen instead (Module 9). |

## Key takeaways

- Deleting a derived object through a base pointer needs a virtual base destructor; without it the behaviour is undefined and the derived destructor does not run.
- Any class with a virtual function gets `virtual ~Base() = default;` — or a protected non-virtual destructor when it is never deleted through the base.
- Copying a derived object into a base value keeps only the base part: by-value parameters, `std::vector<Base>`, `Base b = d;`, assignment through a reference.
- Pass and hold polymorphic objects by reference, pointer or `std::unique_ptr<Base>`; copy them through a virtual `clone()`.
- Make the base abstract or delete its public copy operations to turn slicing from a silent bug into a compile error.

EXERCISE 1

## Sliced

Write `Employee(name, salary)` with a virtual destructor, `virtual std::string role() const` returning `employee` and `virtual long long bonus() const` returning `salary / 10`, plus a non-virtual `name()` accessor. Write `Manager(name, salary, reports)` overriding `role` (`manager`) and `bonus` (`salary / 5 + 100 * reports`).

Then write two functions that print `<role>/<bonus>` for what they receive: `std::string byValue(Employee e)` and `std::string byRef(const Employee& e)`. Call both with each object. Finally push every object into a `std::vector<Employee>` and print the roles it holds. The program demonstrates slicing on purpose; the output shows where the derived part survives and where it is lost.

**Input:** `n`, then `n` lines of `employee name salary` or `manager name salary reports`. **Output:** `n` lines of `<name>: by-value=<role>/<bonus> by-ref=<role>/<bonus>`, then `vector<Employee>: <name>=<role> ...`.

```
2
manager bob 5000 3
employee ann 3000
```

prints

```
bob: by-value=employee/500 by-ref=manager/1300
ann: by-value=employee/300 by-ref=employee/300
vector<Employee>: bob=employee ann=employee
```

INPUT 1

```
2
manager bob 5000 3
employee ann 3000
```

OUTPUT 1

```
bob: by-value=employee/500 by-
ref=manager/1300
ann: by-value=employee/300 by-
ref=employee/300
vector<Employee>: bob=employee
ann=employee
```

INPUT 2

```
1
manager cy 999 0
```

OUTPUT 2

```
cy: by-value=employee/99 by-
ref=manager/199
vector<Employee>: cy=employee
```

REFERENCE SOLUTION

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

class Employee {
public:
    Employee(std::string name, long long salary) : name_(std::move(name)), salary_(s
    virtual ~Employee() = default;
    virtual std::string role() const { return "employee"; }
    virtual long long bonus() const { return salary_ / 10; }
    const std::string& name() const { return name_; }
protected:
    std::string name_;
    long long salary_;
};

class Manager : public Employee {
public:
    Manager(std::string name, long long salary, int reports)
        : Employee(std::move(name), salary), reports_(reports) {}
    std::string role() const override { return "manager"; }
    long long bonus() const override { return salary_ / 5 + 100LL * reports_; }
private:
    int reports_;
};

std::string byValue(Employee e) {                // copies the Employee part only:
    return e.role() + "/" + std::to_string(e.bonus());
}

std::string byRef(const Employee& e) {           // refers to the whole object: dis
    return e.role() + "/" + std::to_string(e.bonus());
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<Employee> sliced;
    for (int i = 0; i < n; ++i) {
        std::string kind, name;
        long long salary = 0;
        std::cin >> kind >> name >> salary;
        if (kind == "manager") {
            int reports = 0;
            std::cin >> reports;
            Manager m(name, salary, reports);
            std::cout << name << ": by-value=" << byValue(m) << " by-ref=" << byRef(
            sliced.push_back(m);                 // copies the Employee subobject
        } else {
```

```
            Employee e(name, salary);
            std::cout << name << ": by-value=" << byValue(e) << " by-ref=" << byRef(
            sliced.push_back(e);
        }
    }
    std::cout << "vector<Employee>:";
    for (const auto& e : sliced) std::cout << ' ' << e.name() << '=' << e.role();
    std::cout << '\n';
    return 0;
}
```

---

EXERCISE 2

## Release in reverse

Write a `Resource` base with a name, a **virtual** destructor, and a static count of live resources (incremented in the constructor, decremented in the destructor). Derive `File`, `Socket` and `Lock`; each constructor prints an opening line and each destructor — marked `override`, so it will not compile unless the base destructor is virtual — prints a closing line:

| kind | on construction | on destruction |
|---|---|---|
| `file` | `open file <name>` | `close file <name>` |
| `socket` | `connect socket <name>` | `disconnect socket <name>` |
| `lock` | `acquire lock <name>` | `release lock <name>` |

Hold them in a `std::vector<std::unique_ptr<Resource>>`. `release` destroys the most recently acquired one (`pop_back`) or prints `nothing to release`. When the input ends, print `still held: <count>`, release everything from the most recent back to the first, then print `alive: <Resource::alive()>`.

**Input:** lines of `file <name>`, `socket <name>`, `lock <name>` or `release`. **Output:** the trace as described.

```
file a.txt
socket 8080
lock db
release
socket 9090
```

prints

```
open file a.txt
connect socket 8080
acquire lock db
release lock db
connect socket 9090
still held: 3
disconnect socket 9090
disconnect socket 8080
close file a.txt
alive: 0
```

### INPUT 1

```
file a.txt
socket 8080
lock db
release
socket 9090
```

### OUTPUT 1

```
open file a.txt
connect socket 8080
acquire lock db
release lock db
connect socket 9090
still held: 3
disconnect socket 9090
disconnect socket 8080
close file a.txt
alive: 0
```

### INPUT 2

```
release
lock m
release
release
```

### OUTPUT 2

```
nothing to release
acquire lock m
release lock m
nothing to release
still held: 0
alive: 0
```

### REFERENCE SOLUTION

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

class Resource {
public:
    explicit Resource(std::string name) : name_(std::move(name)) { ++alive_; }
    virtual ~Resource() { --alive_; }             // virtual: deleted through unique
    const std::string& name() const { return name_; }
    static int alive() { return alive_; }
private:
    std::string name_;
    inline static int alive_ = 0;
};

class File : public Resource {
public:
    explicit File(const std::string& name) : Resource(name) { std::cout << "open fil
    ~File() override { std::cout << "close file " << name() << '\n'; }
};

class Socket : public Resource {
public:
    explicit Socket(const std::string& name) : Resource(name) { std::cout << "connec
    ~Socket() override { std::cout << "disconnect socket " << name() << '\n'; }
};

class Lock : public Resource {
public:
    explicit Lock(const std::string& name) : Resource(name) { std::cout << "acquire
    ~Lock() override { std::cout << "release lock " << name() << '\n'; }
};

int main() {
    std::vector<std::unique_ptr<Resource>> held;
    std::string command;
    while (std::cin >> command) {
        if (command == "release") {
            if (held.empty()) std::cout << "nothing to release\n";
            else held.pop_back();                 // destroys the derived object thr
            continue;
        }
        std::string name;
        std::cin >> name;
        if (command == "file") held.push_back(std::make_unique<File>(name));
        else if (command == "socket") held.push_back(std::make_unique<Socket>(name))
        else if (command == "lock") held.push_back(std::make_unique<Lock>(name));
    }
```

```
    std::cout << "still held: " << held.size() << '\n';
    while (!held.empty()) held.pop_back();        // reverse order, one at a time
    std::cout << "alive: " << Resource::alive() << '\n';
    return 0;
}
```

---

QUIZ

**1.** `Base* p = new Derived; delete p;` where `Base` has a non-virtual destructor. What does the standard say?

A. Both destructors run; nothing is wrong

B. Only `~Base` runs, which is fine as long as `Derived` has no members

C. The behaviour is undefined — in practice `~Derived` never runs

D. Compile error: cannot delete through a base pointer

**2.** `std::unique_ptr<Base> p = std::make_unique<Derived>();` — does `Base` need a virtual destructor here?

A. No — smart pointers always know the real type

B. Yes — `unique_ptr<Base>` deletes through a `Base*`, exactly like a raw `delete`

C. No — `make_unique` records the deleter

D. Only if `Derived` has a destructor of its own

**3.** What does this print?

```
struct B { virtual std::string name() const { return "B"; } virtual ~B() = defa
struct D : B { std::string name() const override { return "D"; } };
void show(B b) { std::cout << b.name(); }
int main() { D d; show(d); }
```

A. `D`

B. `B`

C. Compile error: cannot pass a `D` as a `B`

D. Undefined behaviour

**4.** `std::vector<B> v; D d; v.push_back(d);` — what happens?

A. Compile error: element type mismatch

B. It compiles and stores a sliced copy holding only the `B` part

C. It stores a `D` because the vector remembers the dynamic type

D. Undefined behaviour

**5.** Core Guideline C.35 says a base class destructor should be…

A. Always public and virtual

B. Either public and virtual, or protected and non-virtual

C. Always `= default`

D. Private, so nobody can delete the object

**6.** What does this print?

```
struct B { int x = 1; virtual int get() const { return x; } virtual ~B() = defa
struct D : B { int y = 2; int get() const override { return x + y; } };
int main() {
    D d1, d2;
    d2.x = 10;
    B& r = d1;
    r = d2;
    std::cout << r.get();
}
```

A. `3`

B. `12`

C. `10`

D. `1`

ANSWER KEY · VIRTUAL DESTRUCTORS AND SLICING — THE TWO WAYS A HIERARCHY LOSES DATA

**1. C** Deleting through a pointer to base with a non-virtual destructor is undefined behaviour by the standard. The typical outcome is that only `~Base` runs, leaking whatever `Derived` owned, and sanitizers report a size mismatch. Make `~Base` virtual.

**2. B** The default deleter of `unique_ptr<Base>` calls `delete` on a `Base*`, so the same rule applies. `std::shared_ptr` does record the deleter at creation and would survive this, but relying on that is fragile.

**3. B** `show` takes `B` by value, so a new `B` is copy-constructed from `d`'s base part — the `D` part is sliced away. The dynamic type of `b` is `B`, and the virtual call finds `B::name`. Take `const B&` to dispatch.

**4. B** `push_back(const B&)` accepts `d` by the implicit upcast and copies exactly a `B`. No error, no warning, no polymorphism: every element is a `B`. Use `std::vector<std::unique_ptr<B>>`.

**5. B** Public and virtual when objects may be deleted through the base; protected and non-virtual when they never are, which prevents the dangerous `delete` at compile time and avoids the vptr. The practical rule: any class with a virtual function gets a virtual destructor.

**6. B** `r = d2` runs `B::operator=`, which assigns only the `B` members: `d1.x` becomes 10, `d1.y` stays 2. `r` still refers to `d1`, whose dynamic type is `D`, so `get()` returns 10 + 2. The reference did not slice; the assignment through it did.

LESSON 5 · 15 MIN

# RTTI and the vtable — how dispatch works and when to ask an object its type

`virtual` is a promise the compiler keeps with a data structure. Knowing what it is — a table of function pointers per class and one hidden pointer per object — explains the cost of a virtual call, why a constructor's virtual calls stay in the base, and what `dynamic_cast` and `typeid` have to work with. This lesson opens that box, then covers run-time type information: casting a base pointer down safely, why the reference form throws, what `typeid` reports and must never be used to print, and the design signal a pile of `dynamic_cast`s sends. It closes with `std::variant` and `std::visit`, the alternative for a fixed set of types.

---

## The vtable and the vptr

The standard says only that a virtual call resolves to the dynamic type's override. Every mainstream compiler implements it the same way, and on this platform (the Itanium C++ ABI that GCC and Clang share on Linux) the layout is fixed:

> Each class with at least one virtual function gets one **vtable** — a static array of function pointers, one slot per virtual function in declaration order, filled with the addresses of *that class's* final overriders. A derived class's vtable begins with the base's slots (overridden ones replaced) and appends any new virtual functions.
>
> Each object of such a class carries a hidden **vptr** as its first member, pointing at its class's vtable. On this platform a vptr adds 8 bytes to every object — an `int` wrapper with one virtual function is 16 bytes, not 4.
>
> A virtual call `v.describe()` compiles to: load the vptr from the object, load the `describe` slot from the vtable, call through it. One indirect call.

```
struct Vehicle { virtual std::string describe() const; virtual ~Vehicle(); };
struct Car : Vehicle { std::string describe() const override; };

// vtable for Vehicle: [ &Vehicle::describe, &Vehicle::~Vehicle ]
// vtable for Car:     [ &Car::describe,     &Car::~Car         ]
// v.describe()  becomes  (*(v.vptr[0]))(&v)
```

Two consequences follow. The cost of a virtual call is the indirection plus the fact that the compiler, not knowing the target, cannot inline it — a few nanoseconds, noticeable only in the

innermost loop of something hot. And the constructor rule from lesson 2 has a mechanism: each constructor sets the vptr to *its own* class's vtable before running its body, so during `Vehicle`'s constructor a virtual call finds `Vehicle::describe`; `Car`'s constructor then repoints it. Destructors do the same in reverse. The vtable also holds a pointer to the class's type information, which is what the rest of this lesson uses.

---

## dynamic_cast

An upcast — `Derived*` to `Base*` — is implicit and always valid. A downcast goes the other way and may be wrong, because a `Base*` may point at any derived type or at a plain `Base`. `dynamic_cast` checks at run time:

```
#include <iostream>
#include <memory>
#include <string>
#include <typeinfo>

struct Node { virtual ~Node() = default; };
struct Text : Node { std::string words; };
struct Image : Node { int w = 0, h = 0; };

void inspect(const Node& n) {
    if (const auto* t = dynamic_cast<const Text*>(&n)) {
        std::cout << "text with " << t->words.size() << " chars\n";
    } else if (const auto* i = dynamic_cast<const Image*>(&n)) {
        std::cout << "image " << i->w << 'x' << i->h << '\n';
    } else {
        std::cout << "some other node\n";
    }
}
```

On a pointer, `dynamic_cast<T*>(p)` yields `p` converted to `T*` when the object really is a `T` (or something derived from `T`), and `nullptr` otherwise; the `if (auto* t = …)` shape tests and binds in one line. On a reference there is no null reference to return, so `dynamic_cast<T&>(r)` **throws** `std::bad_cast` on failure — use it when a failure is a bug the caller cannot handle, and the pointer form when either outcome is expected. Both require the source type to be *polymorphic* — at least one virtual function — because the check reads the vtable; on a class without one the cast is a compile error (`source type is not polymorphic`). The check walks the type-information graph and costs more than a virtual call, and it is the one cast

that can cross a hierarchy sideways, from one interface to another the same object implements.

`static_cast<Derived*>(basePtr)` performs the same conversion with no check: fast, and undefined behaviour when the object is not a `Derived`. Use it only where the type is already known — right after a `dynamic_cast` that succeeded, or in a closed piece of code that created the object a line earlier.

---

## typeid

`typeid(expr)` yields a `const std::type_info&` (header `<typeinfo>`) describing the *dynamic* type when the expression is a polymorphic class type, and the static type otherwise:

```
std::unique_ptr<Node> n = std::make_unique<Image>();
std::cout << (typeid(*n) == typeid(Image)) << '\n';     // 1
std::cout << (typeid(n) == typeid(Image)) << '\n';      // 0: the static type
```

`type_info` objects compare with `==` and can key a map through `std::type_index`. They also carry a `name()`, and that string is the one thing not to print: it is implementation-specific and on this platform mangled — `5Image`, `St6vectorIiSaIiEE` — so any output that includes it is unportable and, for a judge, wrong. Print your own tag from a virtual `name()` function instead. `typeid` is rarer than `dynamic_cast` in application code; its natural home is an exact-type check such as "these two objects are the same concrete type" inside an `operator==`.

The two facilities together are *run-time type information* (RTTI). Some code bases compile with `-fno-rtti` to save the type-info tables — game engines, embedded targets — and there `dynamic_cast` and `typeid` are unavailable. The study judge has RTTI on.

---

## The smell

```
double area(const Shape& s) {
    if (auto* c = dynamic_cast<const Circle*>(&s)) return 3.14159 * c->r * c-
    if (auto* r = dynamic_cast<const Rect*>(&s)) return r->w * r->h;
    return 0;
}
```

This works and is wrong. Every new shape requires finding and editing this function — and every other function shaped like it — and forgetting one produces a silent `0`. The chain is a virtual function that has not been written: `virtual double area() const = 0;` puts each formula next to its representation and makes the compiler enforce that new shapes supply one. Tell the object what to do; do not ask it what it is. The legitimate uses of `dynamic_cast` are narrow: a genuinely optional capability (`if (auto* r = dynamic_cast<Resizable*>(w)) r->resize();`), a single special case at a system boundary, or a census over a closed set where adding an `isImage()` to the interface would be worse than the cast.

---

## The closed-set alternative: std::variant

When the set of types is fixed and known up front, C++17 offers a different tool:

```
#include <variant>

struct Circle { double r; double area() const { return 3.14159 * r * r; } };
struct Rect   { double w, h; double area() const { return w * h; } };
using Shape = std::variant<Circle, Rect>;

double area(const Shape& s) {
    return std::visit([](const auto& shape) { return shape.area(); }, s);
}
```

A `std::variant<Circle, Rect>` holds exactly one of its alternatives, by value — no base class, no vptr, no heap, and `std::vector<Shape>` is a plain contiguous vector. `std::visit` calls the function with whichever alternative is present; a generic lambda covers every alternative that shares a member name, and a visitor with one overload per type is checked at compile time for completeness — leaving a type out is an error, not a silent `0`. `std::holds_alternative<Circle>(s)` and `s.index()` ask the type question directly, which is fine here because the set is closed by construction.

| | Virtual functions | `std::variant` + `std::visit` |
|---|---|---|
| Set of types | Open: anyone can derive | Closed: listed in the type |
| Set of operations | Fixed by the base interface | Open: any visitor |
| Storage | Heap objects behind pointers | Values, inline |
| Adding a type | Write a class; nothing else changes | Edit the variant and every visitor |
| Adding an operation | Edit every class | Write one visitor |

Choose by which axis is open. A plugin system needs virtual functions; an expression tree with five node kinds and thirty passes over them is a variant. Module 15, lesson 4 covers `variant` in full.

## Pitfalls

| Mistake | What happens |
|---|---|
| `dynamic_cast` on a class with no virtual functions | Compile error; give the base a virtual destructor. |
| Ignoring the null from a pointer `dynamic_cast` | Null dereference — undefined behaviour. |
| `dynamic_cast<T&>` where failure is possible and uncaught | `std::bad_cast` propagates and terminates the program. |
| Printing `typeid(x).name()` | Mangled, compiler-specific text. |
| A chain of casts where a virtual function belongs | Every new type breaks it silently. |
| `static_cast` down to the wrong type | No check; undefined behaviour. |

## Key takeaways

A virtual call is one indirect call through the class's vtable; each polymorphic object carries a vptr (8 bytes on this platform) that constructors set stage by stage.

`dynamic_cast<T*>` returns `nullptr` on failure; `dynamic_cast<T&>` throws `std::bad_cast`; both need a polymorphic source type.

`typeid(*p)` names the dynamic type; compare `type_info` objects, never print `name()`.

A chain of `dynamic_cast`s is a virtual function waiting to be written; keep casts for optional capabilities and closed-set censuses.

`std::variant` + `std::visit` is the value-semantics alternative for a closed set of types with open operations.

---

EXERCISE 1

## A census by dynamic_cast

Write a polymorphic `Node` base (a virtual destructor is enough to make it polymorphic) and three concrete nodes: `Text` holding a line of words with `int wordCount() const`, `Image` holding a width and height with `long long pixels() const`, and `Link` holding a URL with `bool secure() const` (true when the URL starts with `https://`; `std::string::starts_with` is C++20).

Read the document into a `std::vector<std::unique_ptr<Node>>`. Then take a census with `dynamic_cast` on each `node.get()`: count each concrete type and total the words, pixels and secure links. This is one of the legitimate uses of the cast — a closed-set count over an interface that has no reason to know about counting.

**Input:** lines of `text <words...>`, `image <w> <h>` or `link <url>` until the input ends. **Output:** three lines: `text=<n> words=<total>`, `image=<n> pixels=<total>`, `link=<n> secure=<n>`.

```
text the quick brown fox
image 640 480
link https://example.org
text jumps
link http://old.example.org
```

prints

```
text=2 words=5
image=1 pixels=307200
link=2 secure=1
```

### INPUT 1

```
text the quick brown fox
image 640 480
link https://example.org
text jumps
link http://old.example.org
```

### OUTPUT 1

```
text=2 words=5
image=1 pixels=307200
link=2 secure=1
```

### INPUT 2

```
image 1920 1080
image 1 1
```

### OUTPUT 2

```
text=0 words=0
image=2 pixels=2073601
link=0 secure=0
```

### REFERENCE SOLUTION

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

class Node {
public:
    virtual ~Node() = default;                    // one virtual function makes the
};

class Text : public Node {
public:
    explicit Text(std::string words) : words_(std::move(words)) {}
    int wordCount() const {
        std::istringstream in(words_);
        std::string word;
        int n = 0;
        while (in >> word) ++n;
        return n;
    }
private:
    std::string words_;
};

class Image : public Node {
public:
    Image(int w, int h) : w_(w), h_(h) {}
    long long pixels() const { return 1LL * w_ * h_; }
private:
    int w_;
    int h_;
};

class Link : public Node {
public:
    explicit Link(std::string url) : url_(std::move(url)) {}
    bool secure() const { return url_.starts_with("https://"); }
private:
    std::string url_;
};

int main() {
    std::vector<std::unique_ptr<Node>> nodes;
    std::string line;
    while (std::getline(std::cin, line)) {
        std::istringstream in(line);
        std::string kind;
        in >> kind;
```

```
        if (kind == "text") {
            std::string rest;
            std::getline(in >> std::ws, rest);
            nodes.push_back(std::make_unique<Text>(rest));
        } else if (kind == "image") {
            int w = 0, h = 0;
            in >> w >> h;
            nodes.push_back(std::make_unique<Image>(w, h));
        } else if (kind == "link") {
            std::string url;
            in >> url;
            nodes.push_back(std::make_unique<Link>(url));
        }
    }
    int texts = 0, images = 0, links = 0, secure = 0;
    long long words = 0, pixels = 0;
    for (const auto& node : nodes) {
        if (const auto* t = dynamic_cast<const Text*>(node.get())) {
            ++texts;
            words += t->wordCount();
        } else if (const auto* img = dynamic_cast<const Image*>(node.get())) {
            ++images;
            pixels += img->pixels();
        } else if (const auto* l = dynamic_cast<const Link*>(node.get())) {
            ++links;
            if (l->secure()) ++secure;
        }
    }
    std::cout << "text=" << texts << " words=" << words << '\n';
    std::cout << "image=" << images << " pixels=" << pixels << '\n';
    std::cout << "link=" << links << " secure=" << secure << '\n';
    return 0;
}
```

EXERCISE 2

## The same shapes as a variant

Redo the shape catalogue with **no base class**: three plain structs `Circle { double r; }`, `Rect { double w, h; }` and `Square { double side; }`, each with a `double area() const` and a `const char* name() const` member. Define `using Shape = std::variant<Circle, Rect, Square>;` and hold the input in a `std::vector<Shape>`.

Print each shape's name and area using `std::visit` with a generic lambda (`[](const auto& s) { … }`), count each alternative with `std::holds_alternative`, and print the total. Two decimals, `std::fixed`; use `std::numbers::pi`.

**Input:** `circle r`, `rect w h` or `square s`, whitespace-separated, until the input ends. **Output:** one line per shape, `<name>: area=<a>`; then `circle=<n> rect=<n> square=<n>`; then `total=<sum>`.

```
circle 1
rect 2 3
square 4
```

prints

```
circle: area=3.14
rect: area=6.00
square: area=16.00
circle=1 rect=1 square=1
total=25.14
```

INPUT 1

```
circle 1
rect 2 3
square 4
```

OUTPUT 1

```
circle: area=3.14
rect: area=6.00
square: area=16.00
circle=1 rect=1 square=1
total=25.14
```

INPUT 2

```
square 1.5
square 2
```

OUTPUT 2

```
square: area=2.25
square: area=4.00
circle=0 rect=0 square=2
total=6.25
```

REFERENCE SOLUTION

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

struct Circle {
    double r;
    double area() const { return std::numbers::pi * r * r; }
    const char* name() const { return "circle"; }
};

struct Rect {
    double w, h;
    double area() const { return w * h; }
    const char* name() const { return "rect"; }
};

struct Square {
    double side;
    double area() const { return side * side; }
    const char* name() const { return "square"; }
};

using Shape = std::variant<Circle, Rect, Square>;   // a closed set, held by value

int main() {
    std::vector<Shape> shapes;
    std::string kind;
    while (std::cin >> kind) {
        if (kind == "circle") {
            double r = 0;
            std::cin >> r;
            shapes.push_back(Circle{r});
        } else if (kind == "rect") {
            double w = 0, h = 0;
            std::cin >> w >> h;
            shapes.push_back(Rect{w, h});
        } else if (kind == "square") {
            double s = 0;
            std::cin >> s;
            shapes.push_back(Square{s});
        }
    }
    std::cout << std::fixed << std::setprecision(2);
    double total = 0;
    int circles = 0, rects = 0, squares = 0;
    for (const Shape& s : shapes) {
```

```
        const double a = std::visit([](const auto& shape) { return shape.area(); },
        const char* name = std::visit([](const auto& shape) { return shape.name(); }
        std::cout << name << ": area=" << a << '\n';
        total += a;
        if (std::holds_alternative<Circle>(s)) ++circles;
        else if (std::holds_alternative<Rect>(s)) ++rects;
        else ++squares;
    }
    std::cout << "circle=" << circles << " rect=" << rects << " square=" << squares
    std::cout << "total=" << total << '\n';
    return 0;
}
```

---

QUIZ

**1.** `Base* p` points at a `Base` that is not a `Derived`. What does `dynamic_cast<Derived*>(p)` return?

A. `p` unchanged

B. `nullptr`

C. It throws `std::bad_cast`

D. Undefined behaviour

**2.** `Base& r` refers to a plain `Base`. What does `dynamic_cast<Derived&>(r)` do?

A. Returns a reference to a default-constructed `Derived`

B. Throws `std::bad_cast`

C. Returns a null reference

D. Fails to compile: references cannot be cast

**3.** `struct P { int v; }; struct Q : P {};` and `P* p = new Q; auto* q = dynamic_cast<Q*>(p);`. What happens?

A. `q` points at the `Q`

B. `q` is `nullptr`

C. Compile error: `P` is not polymorphic — it has no virtual functions

D. Undefined behaviour

**4.** What does a class pay for having virtual functions, on this platform?

A. One extra pointer per virtual function in every object

B. A hidden 8-byte vptr in every object, plus one vtable per class

C. Nothing until `dynamic_cast` is used

D. A copy of every virtual function in every object

**5.** `std::unique_ptr<Node> n = std::make_unique<Image>();` — what is `typeid(*n)`?

A. `typeid(Node)` — the static type of `*n`

B. `typeid(Image)` — the dynamic type, because `Node` is polymorphic

C. `typeid(std::unique_ptr<Node>)`

D. It does not compile without `-frtti`

**6.** A function is a chain of `if (dynamic_cast<A*>(p)) … else if (dynamic_cast<B*>(p)) … else if (dynamic_cast<C*>(p)) …`. The best diagnosis is…

A. Correct use of RTTI; nothing to change

B. A virtual function is missing from the base interface — each branch's body belongs in the class it tests for

C. It should use `static_cast` for speed

D. It should use `typeid` instead

**7.** When is `std::variant` + `std::visit` the better choice over a virtual-function hierarchy?

A. When new types are added frequently by code you do not control

B. When the set of types is closed and known up front, and you want value semantics with no heap or vptr

C. Whenever performance matters

D. Never — `variant` is only for error handling

ANSWER KEY · RTTI AND THE VTABLE — HOW DISPATCH WORKS AND WHEN TO ASK AN OBJECT ITS TYPE

**1. B** The pointer form reports failure with a null pointer, which is why the idiom is `if (auto* d = dynamic_cast<Derived*>(p))`. The reference form has no null to return and throws instead.

**2. B** A reference cannot be null, so failure is an exception. Use the reference form when a failed cast is a programming error; use the pointer form and test for null when either outcome is expected.

**3. C** `dynamic_cast` reads the object's vtable to check the type, so the source type must have at least one virtual function. `P` has none; the compiler rejects the cast. Adding `virtual ~P() = default;` fixes it.

**4. B** Each object of a polymorphic class carries one vptr (8 bytes on x86-64) pointing at the class's single shared vtable. The number of virtual functions changes the vtable's size, not the object's.

**5. B** For an expression of polymorphic class type, `typeid` reports the dynamic type. `typeid(n)` without the dereference would be the static type `unique_ptr<Node>`. Compare `type_info` objects with `==`; never print `name()`, which is mangled.

**6. B** Asking an object its type and then doing something per type is what dynamic dispatch does automatically and extensibly. A new derived class silently falls off the end of the chain; a pure virtual function would force it to be handled.

**7. B** A variant lists its alternatives in the type, so adding a type means editing it and every visitor — fine for a fixed set, wrong for a plugin system. In exchange the objects are values, visitors are checked for completeness, and operations are open.

LESSON 6 · 14 MIN

# Composition versus inheritance — is-a, has-a and the shapes to avoid

Inheritance is the most expensive relationship in C++: it couples the derived class to the base's representation, promises that a derived object can stand in for a base one everywhere, and cannot be undone without touching every user. Composition — a member object — makes none of those promises. This lesson gives the test for choosing between them (is-a versus has-a, and what "is-a" actually obliges you to), shows the classic square-and-rectangle failure, covers the two forms of inheritance that are not is-a — private and multiple — and states the rule the Core Guidelines and thirty years of practice agree on: prefer composition, and reach for public inheritance only to get run-time polymorphism through a base interface.

---

## is-a versus has-a

```
class Engine {
public:
    void start() { running_ = true; }
    bool running() const { return running_; }
private:
    bool running_ = false;
};

class Car {                          // a Car HAS an Engine
public:
    void start() { engine_.start(); }
    bool running() const { return engine_.running(); }
private:
    Engine engine_;
};

class ElectricCar : public Car {};   // an ElectricCar IS a Car
```

A `Car` is not an `Engine`; handing a `Car` to a function that expects an `Engine&` would be nonsense. So the engine is a **member**, and `Car` *delegates* — `start()` forwards to `engine_.start()`. Composition lets `Car` decide what to expose: nothing of `Engine`'s interface leaks unless `Car` chooses to forward it. Public inheritance exposes everything public

in the base, and it makes a promise composition does not: an `ElectricCar` can be used wherever a `Car&` is accepted, and every function that works on a `Car` must keep working.

The quick test is to say the sentence aloud. "A car is an engine" fails; "a car has an engine" passes. When the sentence is ambiguous — "a stack is a vector"? — the harder test decides.

---

## The substitution rule

The **Liskov substitution principle** is the harder test: a derived class must honour every promise the base makes, so that code written against the base keeps working when handed a derived object. Not "has the same functions" — *behaves compatibly*. The classic failure:

```
class Rectangle {
public:
    Rectangle(int w, int h) : w_(w), h_(h) {}
    virtual ~Rectangle() = default;
    virtual void setWidth(int w)  { w_ = w; }
    virtual void setHeight(int h) { h_ = h; }
    int area() const { return w_ * h_; }
protected:
    int w_, h_;
};

class Square : public Rectangle {                // a square IS a rectangle...
public:
    explicit Square(int side) : Rectangle(side, side) {}
    void setWidth(int w) override  { w_ = h_ = w; }     // keep it square
    void setHeight(int h) override { w_ = h_ = h; }
};

void stretch(Rectangle& r) {
    r.setWidth(5);
    r.setHeight(2);
    std::cout << r.area() << '\n';        // a Rectangle prints 10
}

Square sq(3);
stretch(sq);                              // prints 4
```

`stretch` was written against `Rectangle`'s promise: after `setWidth(5)` and `setHeight(2)` the area is 10. A `Square` cannot keep that promise — keeping its own invariant (sides equal) breaks the base's (width and height independent). The mathematics says a square is a

rectangle; the *mutable* interface says it is not. Either drop the setters — an immutable `Rectangle` and an immutable `Square` substitute fine, because nothing can be broken — or do not derive. The general lesson: is-a is decided by the base's interface and invariants, not by the nouns.

---

## Why composition wins by default

**Coupling.** A derived class depends on the base's protected members and on the behaviour of every virtual function it does not override. Changing the base can break the derived class without touching its source — the *fragile base class* problem. A member's interface is a smaller, published surface.

**Choice of interface.** Composition exposes what you forward; inheritance exposes the base's whole public interface, including functions that make no sense for the derived type.

**Flexibility.** A member can be swapped at run time — a `std::unique_ptr<Logger>` that points at a file logger or a console logger — while a base class is fixed at compile time.

**Testing.** A class whose collaborators are members can be given test doubles; one whose collaborator is its base cannot.

The Java standard library's `Stack extends Vector` is the cautionary tale: every `Stack` exposes `insertElementAt`, so nothing guarantees it is a stack. C++ chose composition for `std::stack`: it *contains* a `std::deque` and forwards `push`, `pop` and `top`. Do the same with `std::vector` — a class that needs a vector holds one; it does not derive from one, which would also inherit a non-virtual destructor (lesson 4).

---

## Private inheritance

`class Stack : private std::vector<int>` is legal and means "implemented in terms of": the base's public members become private in `Stack`, no one outside can treat a `Stack` as a `std::vector<int>`, and the relationship is invisible to users. It is composition wearing inheritance's syntax.

```
class Stack : private std::vector<int> {
public:
    using std::vector<int>::empty;              // re-expose selected members
    using std::vector<int>::size;
    void push(int v) { push_back(v); }
    int pop() { int v = back(); pop_back(); return v; }
};
```

Prefer a member. Private inheritance earns its place in three narrow cases: the derived class must override a virtual function of the implementation class; it needs access to that class's protected members; or the base is empty and the *empty base optimisation* saves the byte a member would cost — a concern for library authors, not application code.

---

## Multiple inheritance and the diamond

A class may have several bases. With interface classes (lesson 3) it is routine and safe: interfaces carry no state, so nothing can collide. With bases that carry data, a shape called the **diamond** appears:

```
struct Device  { std::string id; };
struct Printer : Device { void print() {} };
struct Scanner : Device { void scan() {} };
struct Copier  : Printer, Scanner {};

Copier c;
c.id = "x";           // error: request for member 'id' is ambiguous — two Dev
c.Printer::id = "x";  // one of them
```

A `Copier` contains *two* `Device`s, one through each path, and `c.id` cannot choose. **Virtual inheritance** makes the paths share one:

```
struct Printer : virtual Device { void print() {} };
struct Scanner : virtual Device { void scan() {} };
struct Copier  : Printer, Scanner {};       // one Device; c.id is unambiguous
```

With virtual bases the *most derived* class constructs the shared base — `Copier`'s initialiser list is where `Device(…)` is called, and the `Device(…)` in `Printer`'s and `Scanner`'s lists is ignored when they are parts of a `Copier` — and each object gains a pointer to locate the shared subobject. The standard library's own diamond, `std::iostream` deriving from `std::istream`

and `std::ostream`, which both derive virtually from `std::ios`, is why the mechanism exists. It is rare in application code; when a diamond appears in a design, the usual answer is that one of the edges should have been composition.

---

## Choosing

| Use public inheritance when | Use composition when |
| --- | --- |
| Callers must handle objects through a base interface without knowing the concrete type | You want to reuse an implementation |
| The derived type honours every promise of the base, invariants included | The derived type would have to hide or weaken base functions |
| The set of types is open and the operations are fixed (lesson 5) | The collaborator may change at run time or be a test double |
| The base is an interface: pure virtual functions and no data | The relationship is "uses" or "has", not "is" |

The Core Guidelines summarise it as C.129: *when designing a class hierarchy, distinguish between implementation inheritance and interface inheritance* — and use public inheritance for the interface kind only.

---

## Pitfalls

| Mistake | What happens |
| --- | --- |
| Deriving to reuse code (`class Stack : public std::vector<int>`) | The whole base interface leaks; no virtual destructor. |
| `Square : Rectangle` with setters | Substitution fails; functions written for the base compute wrong answers. |
| Two data-carrying bases sharing a grandparent | Two copies of the grandparent; ambiguous member access. |
| Protected data reached from many derived classes | Every change to the base ripples through all of them. |
| Multiple non-interface bases | Diamonds, ambiguity and initialisation puzzles; keep to one base plus interfaces. |

## Key takeaways

Public inheritance says *is-a and substitutable*: the derived class must keep every promise of the base's interface, invariants included.

Composition (a member) says *has-a*: you choose what to expose, can swap the part, and stay decoupled from its representation.

Prefer composition; use public inheritance to get dynamic dispatch through an interface class.

Private inheritance is composition in disguise — "implemented in terms of" — and rarely better than a member.

Multiple inheritance is safe for interfaces; data-carrying diamonds need `virtual` bases and are usually a design mistake.

EXERCISE 1

## A service that owns its logger

Write `Logger` with `void info(const std::string&)`, `void error(const std::string&)`, a `const std::vector<std::string>& lines() const` accessor and `int infoCount() const` / `int errorCount() const`. Each call appends `[<n>] INFO <message>` or `[<n>] ERROR <message>`, where `n` is the line's 1-based number.

Write `OrderService` that **has a** `Logger` member (not a base class) and a `std::map<std::string, int>` of open orders, exposing the logger through `const Logger& logger() const`:

- `order <item> <qty>` adds `qty` to the item and logs `order <item> x<qty>` at INFO.
- `cancel <item>` removes the item and logs `cancel <item>` at INFO, or logs `cancel <item>: not found` at ERROR.
- `ship` logs `ship <orders> orders, <units> units` at INFO (the number of open items and the sum of their quantities) and clears them.

**Input:** commands until the input ends. **Output:** every log line, then `info=<n> error=<n>`.

```
order pen 3
order ink 1
cancel cup
ship
cancel pen
```

prints

```
[1] INFO order pen x3
[2] INFO order ink x1
[3] ERROR cancel cup: not found
[4] INFO ship 2 orders, 4 units
[5] ERROR cancel pen: not found
info=3 error=2
```

INPUT 1

```
order pen 3
order ink 1
cancel cup
ship
cancel pen
```

OUTPUT 1

```
[1] INFO order pen x3
[2] INFO order ink x1
[3] ERROR cancel cup: not found
[4] INFO ship 2 orders, 4 units
[5] ERROR cancel pen: not found
info=3 error=2
```

INPUT 2

```
ship
order a 2
order a 5
ship
```

OUTPUT 2

```
[1] INFO ship 0 orders, 0 units
[2] INFO order a x2
[3] INFO order a x5
[4] INFO ship 1 orders, 7 units
info=4 error=0
```

REFERENCE SOLUTION

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

class Logger {
public:
    void info(const std::string& message) {
        add("INFO", message);
        ++info_;
    }
    void error(const std::string& message) {
        add("ERROR", message);
        ++errors_;
    }
    const std::vector<std::string>& lines() const { return lines_; }
    int infoCount() const { return info_; }
    int errorCount() const { return errors_; }
private:
    void add(const std::string& level, const std::string& message) {
        lines_.push_back("[" + std::to_string(lines_.size() + 1) + "] " + level + "
    }
    std::vector<std::string> lines_;
    int info_ = 0;
    int errors_ = 0;
};

class OrderService {
public:
    void order(const std::string& item, int qty) {
        open_[item] += qty;
        logger_.info("order " + item + " x" + std::to_string(qty));
    }
    void cancel(const std::string& item) {
        const auto it = open_.find(item);
        if (it == open_.end()) {
            logger_.error("cancel " + item + ": not found");
            return;
        }
        open_.erase(it);
        logger_.info("cancel " + item);
    }
    void ship() {
        long long units = 0;
        for (const auto& [item, qty] : open_) units += qty;
        logger_.info("ship " + std::to_string(open_.size()) + " orders, " + std::to_
        open_.clear();
    }
    const Logger& logger() const { return logger_; }   // delegation: expose the par
```

```
private:
    Logger logger_;                                  // has-a: the service owns its log
    std::map<std::string, int> open_;
};

int main() {
    OrderService service;
    std::string command;
    while (std::cin >> command) {
        if (command == "order") {
            std::string item;
            int qty = 0;
            std::cin >> item >> qty;
            service.order(item, qty);
        } else if (command == "cancel") {
            std::string item;
            std::cin >> item;
            service.cancel(item);
        } else if (command == "ship") {
            service.ship();
        }
    }
    for (const auto& line : service.logger().lines()) std::cout << line << '\n';
    std::cout << "info=" << service.logger().infoCount() << " error=" << service.log
    return 0;
}
```

EXERCISE 2

## Undo history from two vectors

Write `History`, a class composed of two `std::vector<std::string>` members — `done_` and `undone_` — and nothing else. It is not a vector and must not derive from one; it exposes exactly:

- `void apply(const std::string& action)` — pushes onto `done_` and **clears** `undone_` (a new action discards the redo trail).
- `bool canUndo() const` / `std::string undo()` — moves the most recent action from `done_` to `undone_` and returns it.
- `bool canRedo() const` / `std::string redo()` — the reverse.
- `std::string show() const` — the done actions oldest first, separated by single spaces, or `(empty)`.

**Input:** commands until the input ends: `do <word>`, `undo`, `redo`, `show`. **Output:** `do` prints nothing; `undo` prints `undo <word>` or `nothing to undo`; `redo` prints `redo <word>` or `nothing to redo`; `show` prints the list. At the end print `undoable=<n> redoable=<n>`.

```
do type
do bold
undo
show
redo
do italic
redo
show
```

prints

```
undo bold
type
redo bold
nothing to redo
type bold italic
undoable=3 redoable=0
```

INPUT 1

```
do type
do bold
undo
show
redo
do italic
redo
show
```

OUTPUT 1

```
undo bold
type
redo bold
nothing to redo
type bold italic
undoable=3 redoable=0
```

INPUT 2

```
undo
redo
show
```

OUTPUT 2

```
nothing to undo
nothing to redo
(empty)
undoable=0 redoable=0
```

REFERENCE SOLUTION

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

class History {
public:
    void apply(const std::string& action) {
        done_.push_back(action);
        undone_.clear();                          // a new action invalidates the re
    }
    bool canUndo() const { return !done_.empty(); }
    std::string undo() {
        std::string action = std::move(done_.back());
        done_.pop_back();
        undone_.push_back(action);
        return action;
    }
    bool canRedo() const { return !undone_.empty(); }
    std::string redo() {
        std::string action = std::move(undone_.back());
        undone_.pop_back();
        done_.push_back(action);
        return action;
    }
    std::string show() const {
        if (done_.empty()) return "(empty)";
        std::string out;
        for (std::size_t i = 0; i < done_.size(); ++i) {
            if (i > 0) out += ' ';
            out += done_[i];
        }
        return out;
    }
    std::size_t undoable() const { return done_.size(); }
    std::size_t redoable() const { return undone_.size(); }
private:
    std::vector<std::string> done_;               // composition: two parts, one inv
    std::vector<std::string> undone_;
};

int main() {
    History history;
    std::string command;
    while (std::cin >> command) {
        if (command == "do") {
            std::string word;
            std::cin >> word;
            history.apply(word);
        } else if (command == "undo") {
```

```
            if (history.canUndo()) std::cout << "undo " << history.undo() << '\n';
            else std::cout << "nothing to undo\n";
        } else if (command == "redo") {
            if (history.canRedo()) std::cout << "redo " << history.redo() << '\n';
            else std::cout << "nothing to redo\n";
        } else if (command == "show") {
            std::cout << history.show() << '\n';
        }
    }
    std::cout << "undoable=" << history.undoable() << " redoable=" << history.redoab
    return 0;
}
```

QUIZ

**1.** `Square : public Rectangle` with `setWidth` and `setHeight` overridden to keep the sides equal breaks the Liskov substitution principle because…

A. A square has fewer members than a rectangle

B. Code written against `Rectangle` — set width 5, set height 2, expect area 10 — gets a different answer from a `Square`, so a `Square` cannot substitute for a `Rectangle`

C. `Square` should have used private inheritance

D. Virtual setters are always a design error

**2.** A `Car` needs an `Engine`. Which relationship is right?

A. `class Car : public Engine` — a car is an engine with extras

B. `class Car { Engine engine_; }` — a car has an engine and forwards what it chooses to expose

C. `class Car : private Engine` — a car is implemented in terms of an engine

D. Either public inheritance or composition; they are equivalent here

**3.** What does `class Stack : private std::vector<int> { … };` mean?

A. A `Stack` can be passed wherever a `std::vector<int>&` is expected

B. A `Stack` is implemented in terms of a vector; the vector's public interface is private inside `Stack` and invisible to users

C. The vector's members are deleted

D. It is a compile error: standard containers cannot be base classes

**4.** What happens here?

```
struct A { int x = 0; };
struct B : A {};
struct C : A {};
struct D : B, C {};
int main() { D d; d.x = 1; }
```

A. `d.x` is 1; there is one `A`

B. Compile error: `d.x` is ambiguous because `D` contains two `A` subobjects

C. Undefined behaviour

D. `d.x` sets both copies

**5.** With `struct B : virtual A` and `struct C : virtual A` and `struct D : B, C`, which class's constructor initialises the shared `A`?

A. `B`'s, because it is listed first

B. `C`'s, because it is constructed last

C. `D`'s — the most derived class constructs virtual bases; the `A(…)` in `B`'s and `C`'s initialiser lists is ignored

D. `A` must be default-constructible; nobody may pass arguments

**6.** When is public inheritance the right tool?

A. Whenever two classes share some code

B. When callers must handle objects through a base interface without knowing the concrete type, and the derived type honours every promise of the base

C. When you want to hide some of the base's functions

D. When the base class has no virtual functions

ANSWER KEY · COMPOSITION VERSUS INHERITANCE — IS-A, HAS-A AND THE SHAPES TO AVOID

**1. B** Substitutability is about behaviour, not vocabulary. The base promises that width and height are independent; the derived class cannot keep that promise while keeping its own invariant. Immutable shapes, or no inheritance, resolve it.

**2. B** A car cannot stand in for an engine, so it is has-a: a member. Composition also keeps `Engine`'s interface from leaking into `Car` and lets the engine be swapped. Private inheritance is legal but says nothing a member would not say better.

**3. B** Private inheritance is composition in inheritance's syntax: no is-a relationship is visible outside the class, and only members `Stack` re-exposes (with `using`) or wraps are callable. A member vector would express the same thing more plainly.

**4. B** Without virtual inheritance each path to `A` brings its own subobject, so `D` has `B::x` and `C::x` and the unqualified name cannot choose. `d.B::x` names one; `struct B : virtual A` and `struct C : virtual A` would share a single `A`.

**5. C** A virtual base is constructed once, before any non-virtual base, by the most derived class's constructor. This is why virtual inheritance is rare and awkward: every class that might be most-derived has to know how to build `A`.

**6. B** Public inheritance buys dynamic dispatch through an interface and commits you to substitutability. Sharing implementation, hiding members or swapping parts at run time are jobs for composition.

CHECKPOINT · 25 MIN

# Checkpoint — Inheritance and polymorphism

This checkpoint covers the whole module: deriving with `public` and `protected` and the fixed order of construction and destruction; `virtual`, `override` and `final` and the two places dispatch does not reach; pure virtual functions, interface classes and a container of `std::unique_ptr<Base>`; virtual destructors and slicing; the vtable, `dynamic_cast`, `typeid` and `std::variant`; and is-a against has-a.

**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:

- In what order do the base subobject, the derived class's members and the derived constructor body run — and in what order are they destroyed?
- What does `override` catch that the compiler would otherwise accept silently?
- Why does a virtual call inside a base constructor run the base version?
- What happens when `delete` runs through a `Base*` whose destructor is not virtual?
- What does `void f(Base b)` do to a `Derived` argument?
- What does `dynamic_cast<T*>` return on failure, and what does `dynamic_cast<T&>` do instead?
- Why is `Square : Rectangle` with setters a broken design?

The three programs are a bank of polymorphic accounts held through `std::unique_ptr`, an expression tree built on an abstract `Expr`, and a plugin registry whose load and unload trace pins the construction and destruction order. Read each input format before writing a class.

---

EXERCISE 1

## Accounts at month end

Write an abstract `Account(id, opening)` with a virtual destructor, a protected `balance_`, non-virtual `deposit`, `withdraw` (returns `false` and changes nothing when the amount exceeds the balance), `id()` and `balance()`, and two pure virtual functions: `std::string kind() const` and `void monthEnd()`.

`Savings(id, opening, ratePercent)`: `kind()` is `savings`; `monthEnd()` adds `balance_ * rate / 100` (integer arithmetic).

`Checking(id, opening, minimum, fee)`: `kind()` is `checking`; `monthEnd()` subtracts `fee` when the balance is below `minimum`.

Hold the accounts in a `std::vector<std::unique_ptr<Account>>` in input order and look them up by id with a linear search.

**Input:** `n`, then `n` account lines — `savings <id> <opening> <rate>` or `checking <id> <opening> <minimum> <fee>` — then commands until the input ends: `deposit <id> <x>`, `withdraw <id> <x>`, `monthend`. Ids always exist. **Output:** `rejected <id>` for each refused withdrawal, in order; at the end one line per account in input order: `<id> <kind> <balance>`.

```
2
savings s1 1000 5
checking c1 200 500 25
deposit s1 200
withdraw c1 300
monthend
withdraw s1 5000
monthend
```

prints

```
rejected c1
rejected s1
s1 savings 1323
c1 checking 150
```

INPUT 1

```
2
savings s1 1000 5
checking c1 200 500 25
deposit s1 200
withdraw c1 300
monthend
withdraw s1 5000
monthend
```

OUTPUT 1

```
rejected c1
rejected s1
s1 savings 1323
c1 checking 150
```

### INPUT 2

```
1
checking main 600 500 25
withdraw main 101
monthend
monthend
```

### OUTPUT 2

```
main checking 449
```

### REFERENCE SOLUTION

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

class Account {
public:
    Account(std::string id, long long opening) : id_(std::move(id)), balance_(openir
    virtual ~Account() = default;
    virtual std::string kind() const = 0;
    virtual void monthEnd() = 0;
    void deposit(long long amount) { balance_ += amount; }
    bool withdraw(long long amount) {
        if (amount > balance_) return false;
        balance_ -= amount;
        return true;
    }
    const std::string& id() const { return id_; }
    long long balance() const { return balance_; }
protected:
    std::string id_;
    long long balance_;
};

class Savings : public Account {
public:
    Savings(std::string id, long long opening, int ratePercent)
        : Account(std::move(id), opening), rate_(ratePercent) {}
    std::string kind() const override { return "savings"; }
    void monthEnd() override { balance_ += balance_ * rate_ / 100; }
private:
    int rate_;
};

class Checking : public Account {
public:
    Checking(std::string id, long long opening, long long minimum, long long fee)
        : Account(std::move(id), opening), minimum_(minimum), fee_(fee) {}
    std::string kind() const override { return "checking"; }
    void monthEnd() override {
        if (balance_ < minimum_) balance_ -= fee_;
    }
private:
    long long minimum_;
    long long fee_;
};

Account* find(std::vector<std::unique_ptr<Account>>& accounts, const std::string& id
    for (auto& a : accounts) {
```

```
        if (a->id() == id) return a.get();
    }
    return nullptr;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<std::unique_ptr<Account>> accounts;
    for (int i = 0; i < n; ++i) {
        std::string kind, id;
        long long opening = 0;
        std::cin >> kind >> id >> opening;
        if (kind == "savings") {
            int rate = 0;
            std::cin >> rate;
            accounts.push_back(std::make_unique<Savings>(id, opening, rate));
        } else {
            long long minimum = 0, fee = 0;
            std::cin >> minimum >> fee;
            accounts.push_back(std::make_unique<Checking>(id, opening, minimum, fee)
        }
    }
    std::string command;
    while (std::cin >> command) {
        if (command == "monthend") {
            for (auto& a : accounts) a->monthEnd();
            continue;
        }
        std::string id;
        long long amount = 0;
        std::cin >> id >> amount;
        Account* account = find(accounts, id);
        if (!account) continue;
        if (command == "deposit") {
            account->deposit(amount);
        } else if (command == "withdraw") {
            if (!account->withdraw(amount)) std::cout << "rejected " << id << '\n';
        }
    }
    for (const auto& a : accounts) std::cout << a->id() << ' ' << a->kind() << ' ' <
    return 0;
}
```

EXERCISE 2

## An expression tree

Build a tree of polymorphic nodes from a prefix expression and evaluate it.

`Expr` is abstract: a virtual destructor, `virtual long long eval() const = 0` and `virtual std::string show() const = 0`. `Num` holds a value. `Binary` is an abstract intermediate class holding two `std::unique_ptr<Expr>` children; it implements `eval` and `show` once in terms of two new pure virtual functions, `char symbol() const` and `long long combine(long long a, long long b) const`, which `Add`, `Sub` and `Mul` supply. `show` renders `(<left> <symbol> <right>)`.

Write a recursive `std::unique_ptr<Expr> parse(std::istream& in)` that reads one token: an operator (`+`, `-`, `*`) parses two sub-expressions; anything else is a number. Parse the left child, **then** the right — in two statements, because C++ does not fix the order in which two arguments of one call are evaluated.

**Input:** one prefix expression per line until the input ends. Numbers fit in `long long`; results do too.
**Output:** `<show()> = <eval()>` per line.

```
+ 3 * 4 5
- 10 4
7
```

prints

```
(3 + (4 * 5)) = 23
(10 - 4) = 6
7 = 7
```

INPUT 1

```
+ 3 * 4 5
- 10 4
7
```

OUTPUT 1

```
(3 + (4 * 5)) = 23
(10 - 4) = 6
7 = 7
```

INPUT 2

```
* - 8 2 + 1 1
```

OUTPUT 2

```
((8 - 2) * (1 + 1)) = 12
```

REFERENCE SOLUTION

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

class Expr {
public:
    virtual ~Expr() = default;
    virtual long long eval() const = 0;
    virtual std::string show() const = 0;
};

class Num : public Expr {
public:
    explicit Num(long long value) : value_(value) {}
    long long eval() const override { return value_; }
    std::string show() const override { return std::to_string(value_); }
private:
    long long value_;
};

class Binary : public Expr {                        // abstract: symbol() and combine
public:
    Binary(std::unique_ptr<Expr> left, std::unique_ptr<Expr> right)
        : left_(std::move(left)), right_(std::move(right)) {}
    long long eval() const override { return combine(left_->eval(), right_->eval()); }
    std::string show() const override {
        return "(" + left_->show() + " " + symbol() + " " + right_->show() + ")";
    }
protected:
    virtual char symbol() const = 0;
    virtual long long combine(long long a, long long b) const = 0;
private:
    std::unique_ptr<Expr> left_;
    std::unique_ptr<Expr> right_;
};

class Add : public Binary {
public:
    using Binary::Binary;
protected:
    char symbol() const override { return '+'; }
    long long combine(long long a, long long b) const override { return a + b; }
};

class Sub : public Binary {
public:
    using Binary::Binary;
protected:
```

```
    char symbol() const override { return '-'; }
    long long combine(long long a, long long b) const override { return a - b; }
};

class Mul : public Binary {
public:
    using Binary::Binary;
protected:
    char symbol() const override { return '*'; }
    long long combine(long long a, long long b) const override { return a * b; }
};

std::unique_ptr<Expr> parse(std::istream& in) {
    std::string token;
    in >> token;
    if (token == "+" || token == "-" || token == "*") {
        auto left = parse(in);                    // left first, then right: two sta
        auto right = parse(in);
        if (token == "+") return std::make_unique<Add>(std::move(left), std::move(ri
        if (token == "-") return std::make_unique<Sub>(std::move(left), std::move(ri
        return std::make_unique<Mul>(std::move(left), std::move(right));
    }
    return std::make_unique<Num>(std::stoll(token));
}

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        std::istringstream in(line);
        auto tree = parse(in);
        if (tree) std::cout << tree->show() << " = " << tree->eval() << '\n';
    }
    return 0;
}
```

EXERCISE 3

## Plugins loaded and unloaded

Write `Plugin(name)`: its constructor prints `register <name>`, its **virtual** destructor prints `unregister <name>`, and `virtual std::string status() const = 0` makes it abstract. Derive `Renderer` and `Audio`; each constructor prints `<kind> <name> ready` after the base has printed, each destructor (marked `override`) prints `<kind> <name> stopped` before the base does, and `status()` returns `renderer <name> drawing` or `audio <name> playing`.

Hold the plugins in a `std::vector<std::unique_ptr<Plugin>>`. When the input ends, unload every remaining plugin from the most recent back to the first, then print `done`.

**Input:** commands until the input ends: `load renderer <name>`, `load audio <name>`, `status` (one line per loaded plugin, oldest first), `unload` (the most recent; prints `nothing loaded` when none). **Output:** the trace.

```
load renderer gl
load audio alsa
status
unload
status
```

prints

```
register gl
renderer gl ready
register alsa
audio alsa ready
renderer gl drawing
audio alsa playing
audio alsa stopped
unregister alsa
renderer gl drawing
renderer gl stopped
unregister gl
done
```

INPUT 1

```
load renderer gl
load audio alsa
status
unload
status
```

OUTPUT 1

```
register gl
renderer gl ready
register alsa
audio alsa ready
renderer gl drawing
audio alsa playing
audio alsa stopped
unregister alsa
renderer gl drawing
renderer gl stopped
unregister gl
done
```

### INPUT 2

```
unload
load audio pulse
unload
unload
```

### OUTPUT 2

```
nothing loaded
register pulse
audio pulse ready
audio pulse stopped
unregister pulse
nothing loaded
done
```

### REFERENCE SOLUTION

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

class Plugin {
public:
    explicit Plugin(std::string name) : name_(std::move(name)) {
        std::cout << "register " << name_ << '\n';
    }
    virtual ~Plugin() { std::cout << "unregister " << name_ << '\n'; }
    virtual std::string status() const = 0;
protected:
    std::string name_;
};

class Renderer : public Plugin {
public:
    explicit Renderer(const std::string& name) : Plugin(name) {
        std::cout << "renderer " << name_ << " ready\n";
    }
    ~Renderer() override { std::cout << "renderer " << name_ << " stopped\n"; }
    std::string status() const override { return "renderer " + name_ + " drawing"; }
};

class Audio : public Plugin {
public:
    explicit Audio(const std::string& name) : Plugin(name) {
        std::cout << "audio " << name_ << " ready\n";
    }
    ~Audio() override { std::cout << "audio " << name_ << " stopped\n"; }
    std::string status() const override { return "audio " + name_ + " playing"; }
};

int main() {
    std::vector<std::unique_ptr<Plugin>> loaded;
    std::string command;
    while (std::cin >> command) {
        if (command == "load") {
            std::string kind, name;
            std::cin >> kind >> name;
            if (kind == "renderer") loaded.push_back(std::make_unique<Renderer>(name
            else loaded.push_back(std::make_unique<Audio>(name));
        } else if (command == "status") {
            for (const auto& p : loaded) std::cout << p->status() << '\n';
        } else if (command == "unload") {
            if (loaded.empty()) std::cout << "nothing loaded\n";
            else loaded.pop_back();
        }
```

```
    }
    while (!loaded.empty()) loaded.pop_back();    // reverse order, through Plugin*
    std::cout << "done\n";
    return 0;
}
```

---

### CHECKPOINT QUESTIONS

**1.** What does this print?

```
struct Base { Base() { std::cout << "B"; } ~Base() { std::cout << "b"; } };
struct Part { Part() { std::cout << "P"; } ~Part() { std::cout << "p"; } };
struct Derived : Base {
    Part part;
    Derived() { std::cout << "D"; }
    ~Derived() { std::cout << "d"; }
};
int main() { Derived x; }
```

A. `BPDdpb`

B. `PBDbpd`

C. `BPDbpd`

D. `DPBdpb`

**2.** `struct B { void f(int); void f(double); }; struct D : B { void f(const std::string&); };` — how do you make all three `f` overloads callable on a `D`?

A. Nothing is needed; overloads merge across the hierarchy

B. Add `using B::f;` inside `D`

C. Mark `D::f` as `override`

D. Make `B::f` virtual

**3.** `struct B { virtual void f() const; }; struct D : B { void f() override; };` — what happens?

A. `D::f` overrides `B::f`

B. Compile error: `D::f` is not `const`, so it does not override anything, and `override` reports it

C. It compiles; `D::f` hides `B::f`

D. It compiles with a warning

**4.** `Derived d; Base& r = d;` — what are the static and dynamic types of `r`?

A. Both `Derived`

B. Both `Base`

C. Static `Base`, dynamic `Derived`

D. Static `Derived`, dynamic `Base`

**5.** A class declares `virtual double area() const = 0;`. Which is true?

A. Objects of the class can be created, but `area()` returns 0

B. The class is abstract; only references and pointers to it can exist, and derived classes must override `area` to be instantiable

C. `= 0` sets the default return value

D. The function cannot have a body anywhere

**6.** What does this print?

```
struct B {
    B() { std::cout << tag(); }
    virtual ~B() = default;
    virtual char tag() const { return 'B'; }
};
struct D : B { char tag() const override { return 'D'; } };
int main() { D d; }
```

A. `D`

B. `B`

C. Nothing

D. Undefined behaviour

**7.** `std::unique_ptr<Base> p = std::make_unique<Derived>();` goes out of scope. `Base` has a non-virtual destructor. Which is true?

A. Both destructors run

B. Undefined behaviour; in practice only `~Base` runs and `Derived`'s members leak

C. A compile error

D. `unique_ptr` refuses to delete and leaks deliberately

**8.** What does this print?

```
struct B { virtual int v() const { return 1; } virtual ~B() = default; };
struct D : B { int v() const override { return 2; } };
int byValue(B b) { return b.v(); }
int byRef(const B& b) { return b.v(); }
int main() { D d; std::cout << byValue(d) << byRef(d); }
```

A. `22`

B. `12`

C. `11`

D. `21`

**9.** `dynamic_cast<Derived&>(baseRef)` when `baseRef` does not refer to a `Derived` …

A. Returns a null reference

B. Throws `std::bad_cast`

C. Returns a reference to the base part

D. Is undefined behaviour

**10.** How does a virtual call reach the right function at run time?

A. The runtime searches the hierarchy by function name

B. Through the object's hidden vptr to its class's vtable, where the function's slot holds the address to call — one indirect call

C. The compiler generates a `switch` over all derived classes

D. By comparing `typeid` values until one matches

**11.** Why is `Square : public Rectangle` (with `setWidth` / `setHeight`) considered a broken design?

A. A square needs only one dimension, so it wastes memory

B. A `Square` cannot honour `Rectangle`'s promise that width and height change independently, so code written for `Rectangle` computes wrong results with a `Square`

C. C++ forbids overriding setters

D. Squares should always be implemented as a `std::variant`

**12.** `struct A { int x; }; struct B : A {}; struct C : A {}; struct D : B, C {};` — what fixes the ambiguity of `d.x`?

A. Making `x` `protected`

B. Declaring `B` and `C` as `: virtual A` so `D` contains one shared `A`

C. Marking `D` as `final`

D. Nothing; multiple inheritance always duplicates bases

**13.** Which situation calls for `std::variant` + `std::visit` rather than a class hierarchy?

A. A plugin API where third parties add new types

B. A fixed set of five node kinds with many operations over them, held by value in a vector

C. Any code that must avoid exceptions

D. A hierarchy that needs a virtual destructor

**14.** `std::vector<std::unique_ptr<Shape>> v; v.push_back(std::make_unique<Circle>(2.0));` — why does this compile?

A. Because `std::vector` accepts any pointer type

B. Because `unique_ptr<Circle>` converts implicitly to `unique_ptr<Shape>`, mirroring the `Circle*` to `Shape*` upcast, and the temporary is moved in

C. Because `make_unique` returns `unique_ptr<Shape>` when the argument is a shape

D. It does not compile without a cast

ANSWER KEY · CHECKPOINT — INHERITANCE AND POLYMORPHISM

**1. A** Construction: base, then members, then body — `B P D`. Destruction is the exact reverse: derived body, members, base — `d p b`.

**2. B** A name declared in the derived class hides every base overload of that name. `using B::f;` brings the base overloads into `D`'s scope so all three take part in overload resolution. `override` and `virtual` are about dispatch, not lookup.

**3. B** The signatures differ by `const`, so `D::f` would be a new function that hides the base's. Without `override` that compiles silently and the base version keeps running through a `B&`; with `override`, the mismatch is an error.

**4. C** The static type is what the reference is declared as; the dynamic type is what it refers to. Non-virtual calls through `r` use `Base`'s functions; virtual calls use `Derived`'s overrides.

**5. B** A pure virtual function makes the class abstract. It may still be given a body outside the class, callable as `Base::area()`, but the class stays abstract and a derived class must override the function before it can be instantiated.

**6. B** Inside `B`'s constructor the object's dynamic type is `B` — the vptr points at `B`'s vtable until `D`'s constructor repoints it — so `tag()` resolves to `B::tag`.

**7. B** `unique_ptr<Base>` deletes through a `Base*`, the same operation as a raw `delete`, and the standard makes that undefined without a virtual destructor. Give any class with virtual functions a virtual destructor.

**8. B** `byValue` copies only the `B` part — slicing — so its `b` has dynamic type `B` and `v()` returns 1. `byRef` refers to the `D` itself and dispatches to `D::v`, returning 2.

**9. B** References cannot be null, so the reference form signals failure with an exception. The pointer form, `dynamic_cast<Derived*>(basePtr)`, returns `nullptr` instead.

**10. B** Each polymorphic class has one vtable; each object carries a vptr to it. The call loads the vptr, loads the slot, and jumps. No names, no searching — and the constructor sets the vptr stage by stage, which is why constructors do not dispatch to derived overrides.

**11. B** Substitutability is about behaviour. `stretch(Rectangle&)` sets 5 by 2 and expects 10; a `Square` yields 4. Either remove the mutating interface or drop the inheritance.

**12. B** Virtual inheritance makes every path to `A` share a single subobject, constructed by the most derived class. Without it `D` holds two `A`s and the unqualified `x` cannot choose between them.

**13. B** A closed set with open operations is the variant's shape: alternatives are listed in the type, visitors are checked for completeness, and the values live inline. An open set of types is the hierarchy's shape.

**14. B** `unique_ptr` has a converting move constructor for pointer types that convert implicitly. Ownership of the new `Circle` moves into the vector, which will later delete it through `Shape*` — hence `Shape`'s virtual destructor.

---