Recitation Class

Week 5

Inheritance and Polymorphism

Arrangements for Future Quizzes and Homeworks

- Homework
  - Hw4 has been released, the deadline is Oct 28, 20:00 P.M. 2025
  - Hw4 covers the contents from slides DS-1 to DS-5 (Array, Linked List, Stack, Queue, Hash Table, Asymptotic Analysis)
  - Please Submit Homeworks via Gradescope and remember to match each question correctly.
  - Answers should be written in English.
  - Starting from Homework4, homework will be released at the middle of each chapter but will cover all contents of this chapter.
- Quiz
  - Quiz 2 will be held during the recitation class next week. Remember to bring a pencil or pen!
  - It covers the contents from slides DS-1 to DS-5(Array, Linked List, Stack, Queue, Hash Table, Asymptotic Analysis)
  - In the future, quiz will be held after the content of each chapter is finished.
- Check:
  - In the future, checks will only be held for projects, to ensure your code is done by yourself.
  - There is no checks for homework after C++ Basics.
- Projects
  - Small projects will be released after the content of each chapter is finished. (Project 1 will be released next week)
  - Final project will be released several weeks before the course ends.
  - You should submit small projects via OJ.
  - The submission for final project will be announced in the future.

Syllabus

Please note that these course schedules are subject to change based on actual circumstances and are for reference only.

Week1-Week3	C/C++ Programmiong，OOP	**Week10-Week11**	Graph Traversal, Topological Sorting, Minimum Spanning Tree
Week4-Week5	Arrays, Linked Lists, Stacks, Queues, Hash Tables, Asymptotic Notation	**Week12-Week 13**	Dijkstra, Bellman-Ford, A*, Floyd-Warshall
Week6-Week7	Bubble Sort, Insertion Sort, Merge Sort, Quick Sort	**Wekk14-Week16**	AI Tools
Week8 - Week9	BFS, DFS, Binary Trees, Heaps, Balanced Binary Trees		

Today

- Homework 3
- Quiz 1
- Plugin Installation
- Check

Is-a Relationship

Public inheritance: The "is-a" relationship

By writing that class `D` publicly inherits from class `B`, you are telling the compiler (as well as human readers of your code) that

- Every object of type `D` ***is*** also ***an*** object of type `B`, but not vice versa.
- `B` represents a **more general concept** than `D`, and that `D` represents a **more specialized concept** than `B`.

More specifically, you are asserting that **anywhere an object of type `B` can be used, an object of type `D` can be used just as well**.

- On the other hand, if you need an object of type `D`, an object of type `B` won't do.

Conclusion: Public inheritance means "is-a". Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object.

Upcasting

If `D` is a subclass of `B`:

- A `B*` can point to a `D`, and
- A `B&` can be bound to a `D`.

```
DiscountedItem di = someValue();
Item &ir = di; // correct
Item *ip = &di; // correct
```

Reason: The **is-a** relationship! A `D` **is a** `B`.

But on such references or pointers, only the members of `B` can be accessed.

Inheritance

```
class Item {
public:
  virtual double netPrice(int cnt) const {
    return m_price * cnt;
  }
  // other members
};
class DiscountedItem : public Item {
public:
  double netPrice(int cnt) const override {
    return cnt < m_minQuantity ? cnt * m_price : cnt * m_price * m_discount;
  }
  // other members
};
```

Homework

1. "is-a" relationship and substitutability. (3 pts)

Which statements about "is-a" and substitutability are correct? (Select all that apply.)

A. If `Derived` publicly inherits `Base`, then a `Derived` object "is-a" `Base`, i.e., it can be bound to a `Base&` or `Base*`.

B. If `Derived` privately inherits `Base`, code outside `Derived` can still upcast a `Derived*` to `Base*`.

C. "is-a" implies substitutability: any function that accepts a `Base&` should work correctly when passed a `Derived&`, respecting the base's contract.

D. "is-a" also means the reverse is true: a `Base` object may be used anywhere a `Derived` is required.

Answer: A, C

- A: Correct. Public inheritance models "is-a"; upcasting to `Base&`/`Base*` is allowed by the language.
- B: Incorrect. Private inheritance *hides* the base interface from clients; outside code cannot upcast `Derived*` to `Base*`.
- C: Correct. A derived must honor the base's behavioral contract to be substitutable.
- D: Incorrect. The reverse is false; a `Base` is *not* necessarily a `Derived`.

Upcasting & Downcasting

Upcasting

If `D` is a subclass of `B`:

- A `B*` can point to a `D`, and
- A `B&` can be bound to a `D`.

Downcasting: For polymorphic class only

`dynamic_cast<Target>(expr)`.

```
Base *bp = new Derived{};
Derived *dp = dynamic_cast<Derived *>(bp);
Derived &dr = dynamic_cast<Derived &>(*bp);
```

- `Target` must be a **reference** or a **pointer** type.
- `dynamic_cast` will perform **runtime type identification (RTTI)** to check the dynamic type of the expression.
  - If the dynamic type is `Derived`, or a derived class (direct or indirect) of `Derived`, the downcasting succeeds.
  - Otherwise, the downcasting fails. If `Target` is a pointer, returns a null pointer. If `Target` is a reference, throws an exception `std::bad_cast`.

dynamic_cast & static_cast

`dynamic_cast` can be very slow

`dynamic_cast` performs a runtime **check** to see whether the downcasting should succeed, which uses runtime type information.

This is **much slower** than other types of casting, e.g. `const_cast`, or arithmetic conversions.

[Best practice] Avoid `dynamic_cast` whenever possible.

Guaranteed successful downcasting: Use `static_cast`.

If the downcasting is guaranteed to be successful, you may use `static_cast`

```
auto dp = static_cast<Derived *>(bp); // quicker than dynamic_cast,
// but performs no checks. If the dynamic type is not Derived, UB.
```

Homework

2. Upcasting and downcasting . (3 pts)

```
class Base { virtual ~Base() = default; };
class Derived : Base { void ext() {} };

Base  b;
Derived d;

Base* pb1 = &d;                 // (I) upcast pointer
Base& rb1 = d;                  // (II) upcast reference

Derived* pd1 = static_cast<Derived*>(&b);    // (III)
Derived* pd2 = dynamic_cast<Derived*>(pb1);  // (IV)
Derived& rd1 = dynamic_cast<Derived&>(b);    // (V)
```

Which statements are correct? (Select all that apply.)

A. (I) and (II) are always safe: upcasting preserves dynamic identity and only narrows the visible interface.

B. (III) compiles but is unsafe at runtime; using `pd1->ext()` is undefined behavior.

C. (IV) is safe; `pd2` becomes non-null because `pb1` actually points to a `Derived`.

D. (V) throws `std::bad_cast`, because `b` is a *base* object, not a `Derived`.

Answer: A, B, C, D

- A: Correct. Upcasting is implicit and safe; the dynamic type remains `Derived`.
- B: Correct. `b` is a *standalone* `Base`; `static_cast` to `Derived*` compiles but does not change reality; dereferencing as `Derived` is UB.
- C: Correct. `dynamic_cast` checks RTTI; since `pb1` points at `d`, it succeeds.
- D: Correct. `b`'s dynamic type is `Base`, so a reference downcast fails and throws.

Override

To **override** (覆盖/覆写) a `virtual` function,

- The function parameter list must be the same as that of the base class's version.
- The return type should be **identical to** (or ***covariant with***) that of the corresponding function in the base class.
- **The `const` ness should be the same!**

To make sure you are truly overriding the `virtual` function (instead of making a overloaded version), use the `override` keyword.

*** Not to be confused with "overloading"（重载）.**

```
class DiscountedItem : public Item {
  virtual double netPrice(int cnt) const override; // correct, explicitly virtual
};
class DiscountedItem : public Item {
  double netPrice(int cnt) const; // also correct, but not recommended
};
```

Overload

Such overloading is allowed:

```
void fun(const std::string &);
void fun(std::string &&);
```

- `fun(s1 + s2)` matches `fun(std::string &&)`, because `s1 + s2` is an rvalue.
- `fun(s)` matches `fun(const std::string &)`, because `s` is an lvalue.
- Note that if `fun(std::string &&)` does not exist, `fun(s1 + s2)` also matches `fun(const std::string &)`.

We will see how this kind of overloading benefit us soon.

Dynamic Binding

```
void printItemInfo(const Item &item) {
  std::cout << "Name: " << item.getName()
            << ", price: " << item.netPrice(1) << std::endl;
}
```

The dynamic type of `item` is determined at run-time.

Since `netPrice` is a `virtual` function, which version is called is also determined at run-time:

- If the dynamic type of `item` is `Item`, it calls `Item::netPrice`.
- If the dynamic type of `item` is `DiscountedItem`, it calls `DiscountedItem::netPrice`.

late binding, or **dynamic binding**

Homework

3. Dynamic binding & overriding subtleties. (3 pts)

```
class B {
  virtual B* clone() const { return new B(*this); }
  virtual int f(int) const { return 1; }
};
class D : B {
  D* clone() const override { return new D(*this); }
  int f(int) override { return 2; }
  int f(double) const { return 3; }
};
```

Which statements are correct? (Select all that apply.)

A. `int f(int) override` in `D` (missing `const`) does *not* override `B::f(int) const`.

B. `int f(double) const` is an overload (new signature), not an override.

C. If `B::f(int) const` is called via `B&` bound to `D`, it can select `D`'s proper override when one exists.

D. When called through a `B&` reference bound to a `D` object, `D::clone` returns `D*` only when the base declares the *same* function as virtual.

Answer: A, B, C, D

- A: Correct. Signature must match (including `const`) to override; otherwise it is a different function.
- B: Correct. Changing parameter type to `double` yields an overload.
- C: Correct. Virtual call resolution uses dynamic type; if the exact override exists, it's dispatched.
- D: Correct. Covariance applies only to *overrides* of virtuals; the base must declare the function virtual.

Covariant Return Type

Pure Virtual Functions And Abstract Class

If a `virtual` function does not have a reasonable definition in the base class, it should be declared as **pure** `virtual` by writing `=0`.

```
class Shape {
public:
  virtual void draw(ScreenHandle &) const = 0;
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual ~Shape() = default;
};
```

Any class that has a **pure** `virtual` **function** is an **abstract class**. Pure `virtual` functions (usually) cannot be called [1], and abstract classes cannot be instantiated.

Slicing

Dynamic binding only happens on references or pointers to base class.

```
DiscountedItem di("A", 10, 2, 0.8);
Item i = di; // What happens?
auto x = i.netPrice(3); // Which netPrice?
```

`Item i = di;` calls the **copy constructor of** `Item`

- but `Item` 's copy constructor handles only the base part.
- So `DiscountedItem` 's own members are **ignored**, or **"sliced down"**.
- `i.netPrice(3)` calls `Item::netPrice` .

Homework

4. Abstract Base Class & interface-only design. (3 pts)

```
class Shape {
  virtual ~Shape() = default;
  virtual void draw() const = 0;
  virtual double area() const = 0;
};

class Circle : Shape {
  double r{};
  void draw() const override { /* ... */ }
  double area() const override { return 3.14159 * r * r; }
};

class Bad : Shape {
  // forgot to implement draw()
  double area() const override { return 0; }
};
```

Which statements are correct? (Select all that apply.)

A. `Shape` is abstract because it has at least one pure virtual function.

B. `Circle` is concrete since it provides definitions for all pure virtuals.

C. `Bad` remains abstract because it fails to implement `draw()`.

D. You can declare `std::vector<Shape>` and push `Circle{}` into it to achieve polymorphism.

E. Using `std::vector<std::unique_ptr<Shape»` allows storing heterogeneous shapes without slicing.

Answer: A, B, C, E

- A: Correct. Any class declaring (or inheriting) a pure virtual is abstract.
- B: Correct. `Circle` implements both pure virtuals and can be instantiated.
- C: Correct. Missing an override for a pure virtual keeps `Bad` abstract.
- D: Incorrect. `std::vector<Shape>` stores objects by value; `Shape` is abstract and cannot be instantiated; even if it weren't, this would cause slicing.
- E: Correct. Smart pointers to base enable polymorphic storage without slicing and with automatic lifetime.

Smart Pointer

`<memory>` provides two types of smart pointers:

- `std::unique_ptr<T>`, which uniquely **owns** an object of type `T`.
  - No other smart pointer pointing to the same object is allowed.
  - Disposes of the object (calls its destructor) once this `unique_ptr` gets destroyed or assigned a new value.
- `std::shared_ptr<T>`, which **shares** ownership of an object of type `T`.
  - Multiple `shared_ptr`s pointing to a same object is allowed.
  - Disposes of the object (calls its destructor) when the last `shared_ptr` pointing to that object gets destroyed or assigned a new value.

Note the `<T>` in smart pointers: they are templates, like `std::vector`. `T` indicates the type of their managed objects:

- `std::unique_ptr<int> pi;` points to an `int`, like a raw pointer `int *`.
- `std::shared_ptr<std::vector<double>> pv;`, like an `std::vector<double> *`.

Dereferencing operators `*` and `->` can be used the same way as for raw pointers:

- `*pi = 3;`
- `pv->push_back(2.0);`

unique_ptr

Use `std::unique_ptr` to create an object in dynamic memory,

- if no other pointer to this object is needed.

Two ways of creating an `std::unique_ptr`:

- passing a pointer created by `new` in the constructor:

```
std::unique_ptr<Student> p(new Student("Bob", 2020123123));
```

- use `std::make_unique<T>`, pass initializers to it:

```
std::unique_ptr<Student> p1 = std::make_unique<Student>("Bob", 2020123123);
auto p2 = std::make_unique<Student>("Alice", 2020321321);
```

An `std::unique_ptr` automatically calls the destructor once it gets destroyed or assigned a new value.

- No manual `delete` needed!

shared_ptr

A `unique_ptr` uniquely owns an object, but sometimes this is not convenient:

```
std::vector<Object *> objects;
Object *get_object(int i) {
  return objects[i];
}
```

```
std::vector<unique_ptr<Object>> objects;
unique_ptr<Object> get_object(int i) {
  return objects[i]; // Error
}
```

A smart pointer that uses **reference counting** to manage shared objects.

Create a `shared_ptr`:

```
std::shared_ptr<Type> sp2(new Type(args));
auto sp = std::make_shared<Type>(args); // equivalent, but better
```

For example:

```
auto sp = std::make_shared<std::string>(10, 'c');
// sp points to a string "cccccccccc".
```

Homework

6. Destruction semantics & ownership pitfalls. (3 pts)

```
class B { /* no virtual dtor */ ~B(){} };
class D : B { ~D(){ /* free big buffer */ } };

std::unique_ptr<B> p1(new D);                    // (I)
std::shared_ptr<B> p3 = std::make_shared<D>();   // (II)
B* raw = new D; delete raw;                      // (III)
```

Which statements are correct? (Select all that apply.)

A. (I) is dangerous: `unique_ptr<B>` deletes via `B`'s destructor, which is non-virtual.

B. (II) leaks resources because `shared_ptr` never uses dynamic binding.

C. (III) is undefined behavior for the same reason as (I).

D. If `B` had a virtual destructor, (I), (II), (III) would destroy `D` correctly.

E. Arrays must use array-delete with the *exact* dynamic type; mixing base arrays and derived arrays is a design smell.

Answer: A, C, D, E

- A: Correct. Non-virtual base destructor means deleting via base only runs `B`'s dtor; the derived cleanup is skipped (UB).
- B: Incorrect. `shared_ptr` *will* call the correct deleter (`delete`) which, if `B` has a virtual dtor, will dispatch; the leak claim is false.
- C: Correct. `delete raw;` via non-virtual base is UB (derived destructor not run).
- D: Correct. With a virtual base destructor, polymorphic deletion works for (I), (III), (IV).
- E: Correct. Arrays require array-delete of the precise dynamic type; mixing base arrays/derived arrays is fragile and typically wrong.

Homework

E. Arrays must use array-delete with the *exact* dynamic type; mixing base arrays and derived arrays is a design smell.

```
struct Base {
    virtual ~Base() { cout << "Base dtor\n"; }
};
struct Derived : Base {
    ~Derived() { cout << "Derived dtor\n"; }
};

int main() {
    Base* arr = new Derived[3];  // ❌ allocate Derived array but store as Base*
    delete[] arr;                // ❌ UB: deletes as Base[] not Derived[]
}
```

Quiz - From Homework One

9. Pointer difference. (2 pts)

Given

```
int z[8]{0,1,2,3,4,5,6,7};
int* p = &z[6];
int* q = &z[1];
```

Which of the following statements are *correct*?

A. The expression `p - q` is well-defined and its value is `5`.

B. The type of `(p - q)` is `std::ptrdiff_t` (a signed integer type).

C. The expression `q - p` is well-defined and its value is `-5`.

D. The expression `q - (z + 9)` is well-defined and equals `-8`.

E. The expression `(z + 8) - z` is well-defined and equals `8`.

Answer: A, B, C, E

- A: Correct. `p` points to `z[6]`, `q` points to `z[1]`. The difference `p - q` is `6 - 1 = 5`.
- B: Correct. The result of subtracting two pointers is of type `std::ptrdiff_t`, which is a signed integer type.
- C: Correct. `q - p` is `1 - 6 = -5`.
- D: Incorrect. `z + 9` points to one element past the end of the array of 8 elements (`z[8]`), which is a valid "one-past-the-end" pointer.
- E: Correct. `z + 8` is the "one-past-the-end" pointer of the array `z`. Subtracting the pointer to the first element (`z`) from it yields the size of the array, which is `8`. This is well-defined.

Quiz

```
class Buf {
  std::size_t n_; int* d_;
  static inline std::size_t live_ = 0;
public:
  explicit Buf(std::size_t m): n_(m), d_(new int[m]{}) { ++live_; }
  Buf(const Buf&) = delete;
  Buf& operator=(const Buf&) = delete;
  Buf(Buf&& o) noexcept : n_(o.n_), d_(o.d_) { o.n_=0; o.d_=nullptr; ++live_; }
  Buf& operator=(Buf&& o) noexcept {
    if (this != &o) { delete[] d_; n_ = o.n_; d_ = o.d_; o.n_=0; o.d_=nullptr; }
    return *this;
  }
  ~Buf(){ delete[] d_; --live_; }
  int& at(std::size_t i){ if(i>=n_) throw; return d_[i]; }
  const int& at(std::size_t i) const { if(i>=n_) throw; return d_[i]; }
  int* begin() noexcept { return d_; }
  int* end()   noexcept { return d_ + n_; }
  const int* begin() const noexcept { return d_; }
  const int* end()   const noexcept { return d_ + n_; }
  static std::size_t live() noexcept { return live_; }
};
```

Q. 1 You expose `begin()/end()` so clients can iterate `Buf`. Judge the statements below. **(10 pts)** **ABC**

- ○ Dereferencing `end()` is undefined, even though it is a valid one-past-the-end pointer.
- ○ Pointer arithmetic like `begin()+k` is defined only if the result remains in `[begin(), end()]`.
- ○ Using `at(i)` provides checked access, while `begin()[i]` requires the caller to ensure `i<n`.
- ○ For `const` objects, dedicated `const` overloads of `begin()/end()` aren't needed; the non-`const` versions can still be called and return writable iterators.

Const-Correctness

https://isocpp.org/wiki/faq/const-correctness

Quiz

```
class Buf {
  std::size_t n_; int* d_;
  static inline std::size_t live_ = 0;
public:
  explicit Buf(std::size_t m): n_(m), d_(new int[m]{}) { ++live_; }
  Buf(const Buf&) = delete;
  Buf& operator=(const Buf&) = delete;
  Buf(Buf&& o) noexcept : n_(o.n_), d_(o.d_) { o.n_=0; o.d_=nullptr; ++live_; }
  Buf& operator=(Buf&& o) noexcept {
    if (this != &o) { delete[] d_; n_ = o.n_; d_ = o.d_; o.n_=0; o.d_=nullptr; }
    return *this;
  }
  ~Buf(){ delete[] d_; --live_; }
  int& at(std::size_t i){ if(i>=n_) throw; return d_[i]; }
  const int& at(std::size_t i) const { if(i>=n_) throw; return d_[i]; }
  int* begin() noexcept { return d_; }
  int* end()   noexcept { return d_ + n_; }
  const int* begin() const noexcept { return d_; }
  const int* end()   const noexcept { return d_ + n_; }
  static std::size_t live() noexcept { return live_; }
};
```

Q. 2 You want a robust interface for `Buf`. Judge the statements about the object model.**(7.5 pts)** **BC**

- ○ The `const` overload of `at` allows modifying elements through a `const Buf&`.
- ○ Since `live_` appears before `public:` label, it is `private`.
- ○ Exposing `n_` and `d_` as public would risk representation corruption.

Quiz

```
class Buf {
  std::size_t n_; int* d_;
  static inline std::size_t live_ = 0;
public:
  explicit Buf(std::size_t m): n_(m), d_(new int[m]{}) { ++live_; }
  Buf(const Buf&) = delete;
  Buf& operator=(const Buf&) = delete;
  Buf(Buf&& o) noexcept : n_(o.n_), d_(o.d_) { o.n_=0; o.d_=nullptr; ++live_; }
  Buf& operator=(Buf&& o) noexcept {
    if (this != &o) { delete[] d_; n_ = o.n_; d_ = o.d_; o.n_=0; o.d_=nullptr; }
    return *this;
  }
  ~Buf(){ delete[] d_; --live_; }
  int& at(std::size_t i){ if(i>=n_) throw; return d_[i]; }
  const int& at(std::size_t i) const { if(i>=n_) throw; return d_[i]; }
  int* begin() noexcept { return d_; }
  int* end()   noexcept { return d_ + n_; }
  const int* begin() const noexcept { return d_; }
  const int* end()   const noexcept { return d_ + n_; }
  static std::size_t live() noexcept { return live_; }
};
```

Q. 3 `Buf` is intended for `std::vector<Buf>`. Judge the statements below. **(7.5 pts)**

- ○ Marking move operations `noexcept` guarantees containers always move.
- ○ A moved-from `Buf` being `n_=0, d_=nullptr` makes destruction safe.
- ○ Because `Buf` manages a raw resource and defines a destructor, it should explicitly declare all five special members (destructor, copy constructor/assignment, move constructor/assignment).This implementation does so by deleting copy and providing move.

BC

The **move constructor** and the **move assignment operator**.

```
struct Widget {
  Widget(Widget &&) noexcept;
  Widget &operator=(Widget &&) noexcept;
  // Compared to the copy constructor and the copy assignment operator:
  Widget(const Widget &);
  Widget &operator=(const Widget &);
};
```

- Parameter type is **rvalue reference**, instead of lvalue reference-to-`const`.
- `noexcept` **is (almost always) necessary!** ⇒ Think about why.

Quiz

```
class IUser {
public:
  virtual ~IUser() = default;
  virtual int maxBooks() const = 0;
  virtual bool isAdmin() const = 0;
};

class Member : public IUser {
  int cap_ = 2;
public:
  int  maxBooks() const override { return cap_; }
  bool isAdmin() const override { return false; }
};
class VIPMember : public Member {
public:
  int maxBooks() const override { return 5; }
};
class Admin : public IUser {
  Admin() = default;
public:
  static Admin create() { return Admin(); }
  int  maxBooks() const override { return 0; }
  bool isAdmin() const override { return true; }
};
```

Q. 1 You want a uniform handle to different user roles. Judge the statements about `IUser` and its implementations.**(7.5 pts)**

- ○ `IUser` is abstract because it declares at least one pure virtual function.
- ○ `Member`, `VIPMember`, and `Admin` can be stored as `std::unique_ptr<IUser>` to avoid slicing.
- ○ Calls through `IUser&` bind to the most specific override of `maxBooks()` in the dynamic type.

ABC

Quiz

```
class IUser {
public:
  virtual ~IUser() = default;
  virtual int maxBooks() const = 0;
  virtual bool isAdmin() const = 0;
};

class Member : public IUser {
  int cap_ = 2;
public:
  int  maxBooks() const override { return cap_; }
  bool isAdmin() const override { return false; }
};
class VIPMember : public Member {
public:
  int maxBooks() const override { return 5; }
};
class Admin : public IUser {
  Admin() = default;
public:
  static Admin create() { return Admin(); }
  int  maxBooks() const override { return 0; }
  bool isAdmin() const override { return true; }
};
```

Q. 2 Consider the roles' behaviors and construction rules. Judge the statements about overriding and construction.**(10 pts)**

- ○ `VIPMember::maxBooks()` overrides the inherited interface and specializes the capacity.
- ○ Making `Admin`'s constructor private and exposing `Admin::create()` prevents uncontrolled construction.
- ○ You can pass a `VIPMember` wherever an `IUser&` is expected; calls through the `IUser` interface still invoke the `VIPMember` overrides.
- ○ Declaring `isAdmin()` non-virtual in `IUser` would still allow overrides in derived classes.

ABC

Quiz

```
class IUser {
public:
  virtual ~IUser() = default;
  virtual int maxBooks() const = 0;
  virtual bool isAdmin() const = 0;
};

class Member : public IUser {
  int cap_ = 2;
public:
  int  maxBooks() const override { return cap_;
  bool isAdmin() const override { return false; }
};
class VIPMember : public Member {
public:
  int maxBooks() const override { return 5; }
};
class Admin : public IUser {
  Admin() = default;
public:
  static Admin create() { return Admin(); }
  int  maxBooks() const override { return 0; }
  bool isAdmin() const override { return true; }
};
```

Q. 3 You need `auditBorrowing` to be easy to change and test, without depending on concrete roles or storage details. Choose the statements that best reflect simple, practical design.**(7.5 pts)**

- ○ Define a small `IAuditPolicy` (e.g., `bool canBorrow(const IUser&, int currentlyBorrowed)`), and let the auditor depend on this interface so different policies can be swapped in.
- ○ Hardcode the numeric limits (e.g., 2 or 5) inside for all roles to keep the code shorter.
- ○ Read and write global variables for audit thresholds so configuration is shared across the app without passing parameters.

A

Install plugin

本周五的课程我们会用通义灵码来做演示（由于copilot可能需要梯子）

请同学们提前安装一下这个插件

Check