CS101A(H) Homework 3

Brief Overview

This homework consists of four parts:

- **Multiple Choices (30 pts):** A set of multiple-choice questions covering is-a relationships; inheritance; abstract base classes; upcasting; downcasting; method overriding; dynamic binding; polymorphism.
- **Codes (19 pts):** Mandatory programming exercises. You should fix the code in 'User.cpp'.
- **Check (9 pts):** This part will be done after the discussion class in person. We will examine the reasons of 'why' questions in Codes and your understanding about Multiple Choices. This part will take place in the recitation class, and you will need to answer the TA's questions.

Your grade for this homework will be computed as

$$\min\Big(\text{Multiple Choices} + \text{Codes} + \text{Check},\ 50\text{ pts}\Big).$$

Usage of AI. The use of AI tools for searching information or obtaining assistance on this homework is strictly prohibited. All solutions must be derived from your own understanding and effort. Submissions will be reviewed carefully, and any indication of reliance on AI-generated content will result in severe penalties.

Academic Integrity. This course upholds the highest standards of academic integrity. Any form of academic dishonesty, including plagiarism, unauthorized collaboration, or the use of prohibited resources such as AI tools, will be treated as a serious violation. Such actions not only undermine your own learning but also violate university policies.

Due Date: Oct 21, 2025, **8:00 PM**

1 Multiple Choices

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`.

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.

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.

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.

5. Inheriting interface vs inheriting implementation. (3 pts) 5. Inheriting interface vs inheriting implementation. (3 pts)

```
class ILogger {
    virtual ~ILogger() = default;
    virtual void log(const char*) = 0;
};

class ConsoleLogger : public ILogger {
public:
    void log(const char* s) override {
        std::cout << s << std::endl;
    }
};

class PrefixLogger : public ConsoleLogger {
    std::string prefix = "[DBG] ";
public:
    void log(const char* s) override {
        std::string message = prefix + s;
        ConsoleLogger::log(message.c_str());
    }
};
```

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

A. `ILogger` models *interface inheritance* (pure virtuals).

B. `ConsoleLogger` inherits the interface *and* supplies a concrete implementation.

C. `PrefixLogger` inherits *implementation* and may extend/override behavior.

Answer: A, B, C

- A: Correct. `ILogger` exposes a contract with no data/behavior implemented.
- B: Correct. It realizes the interface by implementing `log`.
- C: Correct. It reuses implementation and refines behavior by overriding the same interface.

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.

7. Cross-casting siblings. (3 pts)

```
class A { virtual ~A() = default; };
class B : A {};
class C : A {};

A* q = new B;
C* pc1 = static_cast<C*>(q);    // X
```

```
C* pc2 = dynamic_cast<C*>(q);   // Y
```

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

A. (X) compiles but is unsafe; calling through `pc1` is undefined behavior.

B. (Y) returns `nullptr` because the dynamic type is `B`, not `C`.

C. Cross-casts between siblings require `dynamic_cast` and a polymorphic base.

D. If `A` lacked virtuals, (Y) would still succeed for sibling casts.

Answer: A, B, C

- A: Correct. `static_cast` does not check dynamic type; using the wrong result is UB.
- B: Correct. RTTI check fails; pointer version returns `nullptr`.
- C: Correct. Cross-cast needs RTTI on a polymorphic base.
- D: Incorrect. Without virtuals, `dynamic_cast` down/cross-casts are not available.

8. Name hiding and overload resolution. (3 pts)

```
class B {
public:
  virtual void f(int)   { /* ... */ }
  void f(double)        { /* ... */ } // non-virtual overload
};
class D : public B {
public:
  void f(int) override { /* D::f(int) */ }   // overrides B::f(int)
};

D d;
B* pb = &d;
```

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

A. `D::f(int)` hides `B::f(double)`; `d.f(3.14)` is ill-formed (no matching function).

B. `pb->f(3.14)` selects `B::f(double)` by overload resolution on static type `B*`; it is non-virtual.

C. `pb->f(3)` calls `D::f(int)` because `B::f(int)` is virtual and overridden in `D`.

Answer: A, B, C

- A: Correct. When a derived class declares a function with the same name (`f`), it hides all overloads of that name in the base class, even those with different parameter lists. Therefore, `B::f(double)` is hidden. Calling `d.f(3.14)` fails because only `D::f(int)` is visible; no matching overload for `double`.
- B: Correct. Overload resolution depends on the static type of the expression (`B*` here). The compiler sees both `B::f(int)` and `B::f(double)`, and selects the best match for a `double` argument—namely, `B::f(double)`. Since it is non-virtual, the call is statically bound.
- C: Correct. For `pb->f(3)`, the static type `B*` provides virtual `f(int)`, which is overridden in `D`. Hence, dynamic dispatch ensures `D::f(int)` is called at runtime.

9. Virtual calls in constructors/destructors. (3 pts)

```
class B {
  B()          { init();  }
  virtual void init()  { /* B init */ }
  virtual ~B() { clean(); }
  virtual void clean() { /* B clean */ }
};
class D : B {
  void init()  override { /* D init */ }
  void clean() override { /* D clean */ }
};

D obj;
```

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

A. During `B()`'s execution, the dynamic type is effectively `B`. `B::init()` runs, not `D::init()`.

B. During $\sim$ `B()`'s execution as part of destroying `D`, `B::clean()` runs, not `D::clean()`.

C. It is recommended to avoid depending on derived state within virtuals called from base ctors/dtors.

D. dynamic binding always uses the most-derived override, even inside base constructors.

Answer: A, B, C

- A: Correct. While building the base, the derived subobject does not yet exist; base virtuals call base implementations.
- B: Correct. During base destructor, the derived subobject is already destroyed; base virtuals call base implementations.
- C: Correct. Relying on derived resources in base ctor/dtor virtuals is a known pitfall.
- D: Incorrect. This is precisely what *does not* happen during construction/destruction.

10. Multiple inheritance: interface + implementation (3 pts).

```
class IPrintable {        // interface
  virtual ~IPrintable() = default;
  virtual void print() const = 0;
};
class Timestamped {       // implementation base
  std::string stamp() const { return "[t]"; }
};

class Note : IPrintable, Timestamped {
  void print() const override { /* uses stamp() + content */ }
};

IPrintable* ip = new Note;
auto* t1 = dynamic_cast<Timestamped*>(ip);   // A
auto* t2 = static_cast<Timestamped*>(ip);    // B
Note n; IPrintable& r = n; r.print();        // C
```

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

A. `IPrintable` is an *interface* (pure virtuals); `Timestamped` provides reusable *implementation*.

B. `Note` inherits interface *and* implementation; this is a common pattern for code reuse.

C. (A) is valid and yields non-null because `ip` actually points to `Note`, which also is-a `Timestamped`.

D. (B) is safe in this specific hierarchy: both bases are subobjects of `Note`, and `ip` points to a `Note`.

E. (C) demonstrates dynamic binding: `r.print()` calls `Note::print()`.

Answer: A, B, C, E

- A: Correct. Clean separation of interface vs implementation inheritance.
- B: Correct. This leverages MI for behavior reuse while exposing a uniform interface.
- C: Correct. `dynamic_cast` across bases of a polymorphic object succeeds and adjusts the pointer.
- D: Incorrect.
- E: Correct. `print` is virtual in the interface; calling via `IPrintable&` dispatches to `Note::print()`.

2 Code: polymorphism of Users in the library

The objective of this part is to complete the implementation of the library management system in 'User.cpp'. You need to finish the implementation of four classes:

- `class User`: abstract base class.
- `class Member:public User`: basic member who can borrow 2 books at the same time.
- `class VIPMember:public Member`: VIP member who can borrow 5 books at the same time.
- `class Admin:public User`: administrator of the library, who cannot borrow books, but has some advanced management privileges.

In the primary interface of the library system, you can:

1. Login with userID;
2. Register as a basic member with a new userID.

If you log in as a `Member`, in the secondary interface you can:

1. Display all books;
2. Display your borrowed books;
3. Borrow book;
4. Return book;
5. Check whether you are VIP member.

Otherwise, you log in as an `Admin`, in the secondary interface you can:

1. Display all books;
2. Add book;
3. Remove book;
4. Change VIP identity of a Member;

5. Add a new `Admin` account.

2.1 Tasks and Requirements

Complete the code in 'Users.cpp'. The frame structure has been provided, so you only need to be aware of the following things:

- Add access specifiers (`public`, `protected` or `private`) to satisfy the requirements;
- Add (`= 0`, `= default` or `= delete`) to satisfy the requirements;
- Add (`virtual`, `override`) where it's needed.
- Some methods like `Member(const std::string& id)` are declared but not defined. You are required to implement them.
- Some methods are left with blanks. You are required to fix them.

Besides that the code should compile and all functions should work properly, there are some additional requirements:

- Copy constructor and copy assignment should be banned in `class User` and its derived classes.
- Class member variables including `userId` and `maxBooksAllowed` should be inaccessible outside the class.
- `class Admin` is not allowed to be constructed outside the class.

2.2 Tests and Points

Submit 'User.cpp' to OJ.

2.2.1 **basic compilation and requirement test (5 pts).**
The code should compile, and meet the requirements.

2.2.2 **Member operations test (5 pts).**
Only members will log in and do what they can. There will also be new member registrations.

2.2.3 **Admin operations test (5 pts).**
Only administrators will log in and do what they can. There will also be new member registrations.

2.2.4 **all possible operations test (4 pts).**
Not only members but also administrators will log in and do what they can. There will also be new member registrations.

2.2.5 **check in recitation (9 pts).**
Prepare to answer following questions:

- Why are copy constructor and copy assignment not allowed in `class User` and its derived classes?
- Why is `class User` not allowed to be instantiated?
- Why should member variables including `userId` and `maxBooksAllowed` be inaccessible outside the class?
- Why is `class Admin` not allowed to be constructed outside the class?