Lecture-9: C++ Inheritance

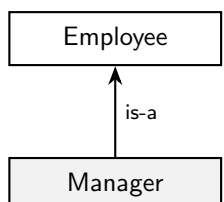
Bernard Nongpoh

Motivation: Why we need inheritance

```
struct Employee {
  string name;
};

struct Manager {
  Employee emp;     // "has an Employee"
  int level;
};

void f(Employee* e);

Manager m;
f(&m);          // ERROR: Manager* !=
Employee*
f(&m.emp);      // Works, but awkward

// Better: Manager : public Employee
```

A **Manager** is also an **Employee**; the **Employee** data is stored inside a **Manager** object.

Without inheritance, the compiler does *not* know that `Manager` can be used wherever `Employee` is expected (e.g., a `Manager*` is not an `Employee*`).



The Problem: Code Duplication

```
struct Employee {
  string first_name;
  string family_name;
  int department;
  Date hiring_date;
};
```

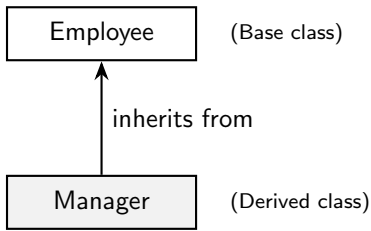
```
struct Manager {
  string first_name;    // duplicate!
  string family_name;   // duplicate!
  int department;       // duplicate!
  Date hiring_date;     // duplicate!
  int level;
  list<Employee*> group;
};
```

Problems:

- Redundancy: Same fields declared multiple times
- Maintenance: Changes must be made everywhere
- No type relationship: Can't use Manager where Employee expected

The "Is-A" Relationship

"A Manager **is an** *Employee with extra responsibilities."*



Real-world examples:

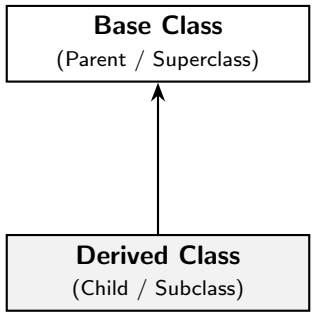
- A Circle **is a** Shape
- A Dog **is an** Animal

The Solution: Inheritance

```
class Employee {
public:
    string first_name, last_name;
    int department;

    void print() const;
    string full_name() const {
        return first_name + " " + last_name;
    }
};

class Manager : public Employee {   // Manager IS-A Employee
public:
    list<Employee*> group;          // people managed
    short level;

    void print() const;             // can provide own version
};
```

Result: Manager automatically has all Employee members!

Terminology



What derived classes can do:

1. **Inherit** all members from base class
2. **Add** new data members and functions
3. **Override** base class function behavior

Two Types of Inheritance

Implementation Inheritance

- Reuse code from base class
- Share common functionality
- Reduce redundancy

```
// Manager reuses Employee's
// data and functions
class Manager : public Employee {
  // Gets name, dept, print()
  // for free!
};
```

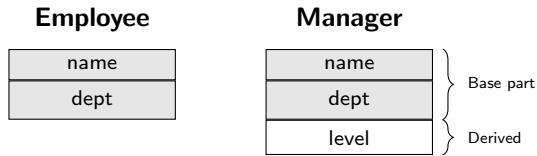
Interface Inheritance

- Define a common interface
- Enable polymorphism
- Allow interchangeable use

```
class Shape {
public:
  virtual void draw() = 0;
};

class Circle : public Shape {
  void draw() override;
};
```

Memory Layout of Derived Classes

```
class Employee {
    string name;
    int dept;
};

class Manager
  : public Employee {
    int level;
};
```



Key Points:

- Derived = Base members + Derived members (appended)
- No memory overhead from inheritance itself
- Base part comes first in memory

Using Inherited Members

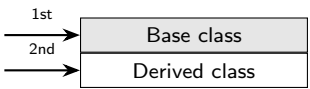
```cpp
class Employee {
public:
    string name;
    int department;
    void print() const {
        cout << name << " in dept " << department << endl;
    }
};

class Manager : public Employee {
public:
    int level;
    void printAll() const {
        print();  // Calls inherited Employee::print()
        cout << "Level: " << level << endl;
    }
};

int main() {
    Manager m;
    m.name = "Alice";       // Accessing inherited member
    m.department = 42;      // Accessing inherited member
    m.level = 3;            // Manager's own member
}
```

Construction Order: Base First

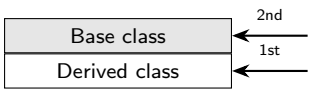
```
class Animal {
    string name;
public:
    Animal(const string& n) : name(n) {
        cout << "Animal constructed" << endl;
    }
};

class Dog : public Animal {
    int age;
public:
    Dog(const string& n, int a)
        : Animal(n),     // MUST call base constructor first!
          age(a)         // Then initialize own members
    {
        cout << "Dog constructed" << endl;
    }
};

Dog d("Rex", 3);
// Output: Animal constructed
//         Dog constructed
```

Construction and Destruction Order

Construction: Base first



Destruction: Derived first



Complete Order:

Construction: Base → Members → Derived body

Destruction: Derived body → Members → Base

Important Constructor Rules

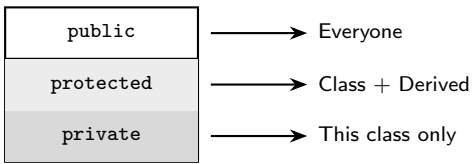
```
class Base {
public:
    Base(int x) { }  // No default constructor!
};

class Derived : public Base {
public:
    // ERROR: Base has no default constructor
    Derived() { }

    // CORRECT: Must explicitly call base constructor
    Derived() : Base(0) { }

    // CORRECT: Passing parameter to base
    Derived(int x, int y) : Base(x) { }
};
```

Rules:

- If base has no default constructor, you **must** call it explicitly
- Base constructor is called **before** member initializers

The Three Access Levels

Specifier	Own Class	Derived	Outside
`private`	✓	×	×
`protected`	✓	✓	×
`public`	✓	✓	✓



Access Control Example

```
class Employee {
private:
    double salary;        // Only Employee can access
protected:
    int employee_id;      // Employee + derived classes
public:
    string name;          // Everyone can access
};

class Manager : public Employee {
public:
    void test() {
        salary = 50000;       // ERROR: salary is private
        employee_id = 42;     // OK: employee_id is protected
        name = "Alice";       // OK: name is public
    }
};

void external() {
    Manager m;
    m.name = "Bob";           // OK: public
    m.employee_id = 1;        // ERROR: protected
}
```

Inheritance Access Specifiers

```
class D1 : public Base { };     // Most common - IS-A relationship
class D2 : protected Base { };  // Rare - implementation detail
class D3 : private Base { };    // HAS-A via inheritance
```

Base Member	: public	: protected	: private
public	public	protected	private
protected	protected	protected	private
private	—	—	—

Defaults:

- `class` defaults to **private** inheritance
- `struct` defaults to **public** inheritance

Protected Members: Use With Caution

```
class Buffer {
public:
    char& operator[](int i); // Checked

protected:
    char& access(int i);     // Unchecked

};

class FastBuffer : public Buffer {
public:
    void copy(const char* src, int n) {
        for (int i = 0; i < n; ++i)
            access(i) = src[i]; // Fast
    }
};
```

Protected functions: ✓
Useful for derived class implementers

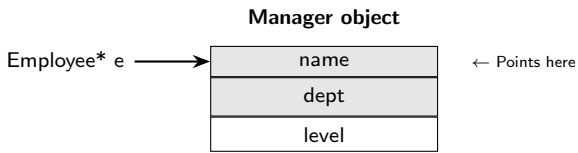
Protected data: ×
Usually a design error!

Problems with protected data:

- Open to corruption
- Hard to refactor
- Tight coupling

Upcasting: Always Safe

```
Manager m;
Employee* e = &m;      // OK: implicit conversion
Employee& r = m;       // OK: reference too

void process(Employee* emp);
process(&m);           // OK: Manager* -> Employee*
```



Upcasting = derived* to base*. Always safe and implicit.

Downcasting: Be Careful

```
Employee e;
Manager* m = &e;     // ERROR: Not every Employee is a Manager

// If we KNOW it's really a Manager:
Employee* eptr = &someManager;

// Option 1: static_cast - Trust programmer (fast, unsafe)
Manager* m1 = static_cast<Manager*>(eptr);

// Option 2: dynamic_cast - Runtime check (safe)
Manager* m2 = dynamic_cast<Manager*>(eptr);
if (m2 != nullptr) {
    cout << m2->level << endl;  // Safe to use
}
```

Cast	Speed	Safety
`static_cast`	Fast	Unsafe (undefined if wrong)
`dynamic_cast`	Slower	Safe (returns nullptr if wrong)

Why Pointers and References Matter

By Value (Copies):

```
void byValue(Employee e) {
    e.print();
}

Manager m;
byValue(m);
// Copies only Employee part!
// SLICING occurs!
```

By Reference/Pointer:

```
void byRef(Employee& e) {
    e.print();
}
void byPtr(Employee* e) {
    e->print();
}

Manager m;
byRef(m);  // No copy, full object
byPtr(&m); // No copy, full object
```

Rule: Polymorphism only works through pointers or references.

Virtual Functions & Polymorphism

Polymorphism (1/3)

Polymorphism (meaning *"having multiple forms"*) is the capability of an entity to mutate its behavior according to the specific usage context.

Polymorphism (2/3)

Polymorphism (meaning *"having multiple forms"*) is the capability of an entity to mutate its behavior according to the specific usage context.

Types of Polymorphic Dispatch

- **Compile-time (Static Polymorphism)**
  - The called instance is known **before program execution**.
  - Examples: function overloading, operator overloading, templates.

Polymorphism (3/3)

Polymorphism (meaning *"having multiple forms"*) is the capability of an entity to mutate its behavior according to the specific usage context.

Types of Polymorphic Dispatch

- **Compile-time (Static Polymorphism)**
  - The called instance is known **before program execution**.
  - Examples: function overloading, operator overloading, templates.
- **Run-time (Dynamic Polymorphism)**
  - The called instance is known **during execution**.
  - Depends on run-time values.
  - Achieved using `virtual` functions (overriding).

In C++, the term **polymorphic** is strongly associated with **dynamic polymorphism** (function overriding via virtual functions).

https://federico-busato.github.io/Modern-CPP-Programming/htmls/10.Object_Oriented_II.html

C++ Mechanisms for Polymorphism (1/2)

- *Preprocessing*

```
#define ADD(x, y) x + y  // ADD(3, 4) or ADD(3.0, 4.0)
```

- *Function/Operator overloading*

```
void f(int);
void f(double);
```

- *Templates*

```
template<typename T>
void f(T);               // f(3) or f(3.0)
```

- *Virtual functions*

https://federico-busato.github.io/Modern-CPP-Programming/htmls/10.Object_Oriented_II.html

C++ Mechanisms for Polymorphism (2/2)

Mechanism	**Implementation**	**Form**
Preprocessing	static	Parametric
Function / Operator Overloading	static	Ad-hoc
Template	static	Parametric
Virtual Function	dynamic	Subtyping

- **Static** → Resolved at compile time
- **Dynamic** → Resolved at runtime (via vtable)
- **Parametric** → Generic code (e.g., templates)
- **Ad-hoc** → Same name, different behavior
- **Subtyping** → IS-A relationship

https://federico-busato.github.io/Modern-CPP-Programming/htmls/10.Object_Oriented_II.html

Dynamic Polymorphism in C++ (1/2)

- At run-time, objects of a *base class* behave as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

Dynamic Polymorphism in C++ (2/2)

- At run-time, objects of a *base class* behave as objects of a *derived class*
- A **Base** class may define and implement polymorphic methods, and **derived** classes can **override** them, which means they provide their own implementations, invoked at run-time depending on the context

```
struct A {
    void f()  { cout << "A"; }
};
struct B : A {
    void f()  { cout << "B"; }
};
void g(A& a) { a.f(); }  // accepts A and B
                         // note: g(B&) would only accept B

A a; B b;
g(a);    // print "A"
g(b);    // print "A" not "B"!!!
```

Without `virtual`, the compiler uses **static binding** — resolved at compile time.

Polymorphism — `virtual` Method

```
struct A {
    virtual void f() { cout << "A"; }
};  // now "f()" is virtual, evaluated at run-time

struct B : A {
    void f() override { cout << "B"; }
    // now B::f() overrides A::f(), run-time dispatch
    // 'virtual void f()' is also valid
};  // 'override' is a C++11 feature, more details in the next slides

void g(A& a) { a.f(); }  // accepts A and B

A a;
B b;
g(a);    // print "A"
g(b);    // NOW, print "B"!!!
```

The `virtual` keyword enables **dynamic binding** — the actual type's function is called at runtime.

When `virtual` Works

```
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() override { cout << "B"; }
};

void f(A& a) { a.f(); }   // ok, print "B"
void g(A* a) { a->f(); }  // ok, print "B"
void h(A  a) { a.f(); }   // does not work with pass-by value!!
                          // print "A"
B b;
f(b);    // print "B"
g(&b);   // print "B"
h(b);    // print "A" (cast to A)
```

Rule: Polymorphism only works through pointers or references, NOT by value.

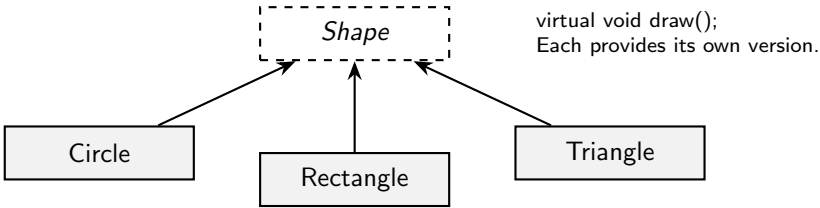
Polymorphism Dynamic Behavior

```
struct A {
    virtual void f() { cout << "A"; }
};

struct B : A {
    void f() override { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

get_object(true)->f();   // print "A"
get_object(false)->f();  // print "B"
```

The actual function called depends on the **runtime type** of the object, not the static type of the pointer. This is the essence of **dynamic polymorphism**.

What is Polymorphism?

"One interface, multiple implementations."




```
void render(Shape* s) { s->draw(); }  // Works for ANY shape!

render(new Circle());     // Draws a circle
render(new Rectangle());  // Draws a rectangle
```

Virtual Table

The **virtual table** (`vtable`) is a lookup table of functions used to resolve function calls and support *dynamic dispatch* (late binding).

A *virtual table* contains one entry for each `virtual` function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the *most-derived* function accessible by that class.

The compiler adds a *hidden* pointer to the base class which points to the virtual table for that class (`sizeof` considers the vtable pointer).

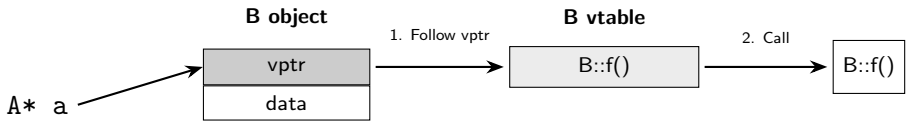
Does the `vtable` Really Exist? (Answer: YES)

```
struct A {
    int x = 3;
    virtual void f() { cout << "abc"; }
};

A* a1 = new A;
A* a2 = (A*) malloc(sizeof(A));

cout << a1->x;    // print "3"
cout << a2->x;    // undefined value!!
a1->f();           // print "abc"
a2->f();           // segmentation fault
```

Lesson learned: Never use `malloc` in C++.

`new` calls the constructor (which initializes the `vptr`), while `malloc` only allocates raw memory without setting up the vtable pointer.

Virtual Function Call Mechanism



`a->f()` becomes:

1. Read the vptr from the object
2. Look up `f()` slot in the vtable
3. Call the function pointer found there

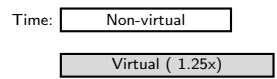
Virtual Function Overhead

Space Overhead:

- One vptr per object
- One vtable per class

Time Overhead:

- Extra pointer indirection
- About 25% slower
- Harder to inline



Guideline:
Use `virtual` when needed.
Don't use "just in case."

`override` Keyword (C++11) (1/2)

The `override` keyword ensures that the function is `virtual` and is overriding a `virtual` function from a base class.

- It forces the compiler to check the base class to see if there is a `virtual` function with this exact signature
- `override` clearly expresses the intent of the function, making the code easier to understand

`override` implies `virtual` (`virtual` should be omitted).

Always use `override`!

- Catches typos and signature mismatches at compile time
- Documents intent clearly
- Zero runtime cost

override Keyword (C++11) (2/2)

```
struct A {
    virtual void f(int a);      // a "float" value is casted to "int"
};                              // ***

struct B : A {
    void f(int   a) override;   // ok
    void f(float a);            // (still) very dangerous!!
                                // ***
//  void f(float a) override;       // compile error not safe
//  void f(int   a) const override; // compile error not safe
};

// *** f(3.3f) has a different behavior between A and B
```

Without override, a function with the wrong signature silently creates a *new* function instead of overriding — a common and dangerous bug.

`final` Keyword (C++11)

The `final` keyword prevents inheriting from classes or overriding methods in derived classes.

```
struct A {
    virtual void f(int a) final;  // "final" method
};

struct B : A {
//  void f(int a);      // compile error f(int) is "final"
    void f(float a);    // dangerous (still possible)
};                      // "override" prevents these errors

struct C final {        // cannot be extended
};
// struct D : C {       // compile error C is "final"
// };
```

Note: Use `final` sparingly. Prefer `override` for safety.

Calling Base Class Version

```
class Employee {
public:
    virtual void print() const {
        cout << name << " in dept " << department << endl;
    }
};

class Manager : public Employee {
public:
    void print() const override {
        Employee::print();   // Call base version explicitly
        cout << "Level: " << level << endl;
    }
};
```

Important: Using `::` bypasses the virtual mechanism.

```
void Manager::print() const {
    print();            // INFINITE RECURSION!
    Employee::print();  // Correct: calls base version
}
```

Virtual Methods (Common Error 1): Virtual Destructors

All classes with at least one `virtual` method should declare a `virtual` destructor.

```
struct A {
    ~A() { cout << "A"; }   // <-- here the problem (not virtual)
    virtual void f(int a) {}
};
struct B : A {
    int* array;
    B()  { array = new int[1000000]; }
    ~B() { delete[] array;           }
};
//-----------------------------------------------------
void destroy(A* a) {
    delete a;     // call ~A()
}
B* b = new B;
destroy(b); // without virtual, ~B() is not called
            // destroy() prints only "A" -> huge memory leak!!
```

Fix: `virtual ~A() = default;`

Virtual Methods (Common Error 2): Virtual in Constructor/Destructor

Do not call virtual methods in constructor and destructor.

- *Constructor*: The derived class is not ready until constructor is completed
- *Destructor*: The derived class is already destroyed

```
struct A {
    A() { f(); }  // what instance is called? "B" is not ready
                  // it calls A::f(), even though A::f() is virtual
    virtual void f() { cout << "Explosion"; }
};
struct B : A {
    B() = default;  // call A(). Note: A() may be also implicit

    void f() override { cout << "Safe"; }
};

B b;  // call B(), print "Explosion", not "Safe"!!
```

Virtual Methods (Common Error 3): Default Parameters

Do not use default parameters in virtual methods.

Default parameters are not inherited.

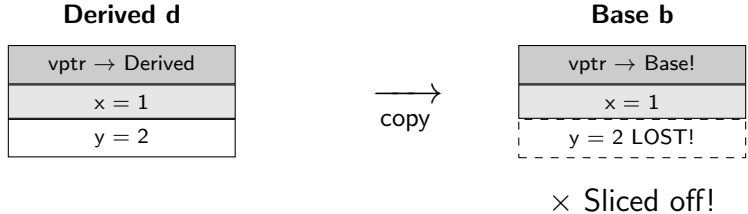
```
struct A {
    virtual void f(int i = 5) { cout << "A::" << i << "\n"; }
    virtual void g(int i = 5) { cout << "A::" << i << "\n"; }
};
struct B : A {
    void f(int i = 3) override { cout << "B::" << i << "\n"; }
    void g(int i)     override { cout << "B::" << i << "\n"; }
};
A a; B b;
a.f();      // ok, print "A::5"
b.f();      // ok, print "B::3"

A& ab = b;
ab.f();     // !!! print "B::5"  // the virtual table of A
            // contains f(int i = 5) and g(int i = 5) but it points
ab.g();     // !!! print "B::5"  // to B implementations
```

The Slicing Problem

```
class Base {
public:
    int x = 1;
    virtual void print() { cout << "Base: " << x << endl; }
};

class Derived : public Base {
public:
    int y = 2;
    void print() override { cout << "Derived: " << x << ", " << y << endl; }
};

Derived d;
Base b = d;      // SLICING! y is lost, vptr points to Base!
b.print();       // Prints "Base: 1" -- NOT "Derived: 1, 2"!
```

Object Slicing: When derived is copied to base, the derived part is "sliced off."

Visualizing Object Slicing



What happens:

- Base b only has space for Base members
- Copy constructor copies only the Base part
- vptr set to **Base's vtable**, not Derived's

Where Slicing Occurs

```cpp
// 1. Direct assignment
Derived d;
Base b = d;                     // SLICED

// 2. Function parameters (pass by value)
void process(Base b);
process(d);                     // SLICED

// 3. Containers of values
vector<Base> v;
v.push_back(Derived());         // SLICED

// 4. Return by value
Base createObject() {
    return Derived();
}                               // SLICED
```

Pattern: Any operation that **copies** derived to base causes slicing.

Preventing Object Slicing

```cpp
// Solution 1: Use references
void process(Base& b);          // No copy
process(d);                     // OK!

// Solution 2: Use pointers
void process(Base* b);          // No copy
process(&d);                    // OK!

// Solution 3: Use smart pointers for containers
vector<unique_ptr<Base>> v;
v.push_back(make_unique<Derived>());   // OK!

vector<shared_ptr<Base>> v2;
v2.push_back(make_shared<Derived>());  // OK!
```

Rule: For polymorphism, always use pointers or references.

Pure Virtual Functions (1/2)

A **pure virtual Functions** is a function that must be implemented in derived classes (concrete implementation).

Pure virtual functions can have or not have a body:

```
struct A {
    virtual void f() = 0;  // pure virtual without body
    virtual void g() = 0;  // pure virtual with body
};
void A::g() {}  // pure virtual implementation (body) for g()

struct B : A {
    void f() override {}  // must be implemented
    void g() override {}  // must be implemented
};
```

Note: Even with a body, a pure virtual function **must** still be overridden in derived classes.

Pure Virtual Functions (2/2)

```
class Shape {
public:
    virtual void draw() const = 0;      // Pure virtual
    virtual double area() const = 0;    // Pure virtual
    virtual ~Shape() = default;         // Virtual destructor!
};

Shape s;  // ERROR: Cannot instantiate abstract class!
```

The `= 0` makes a function **pure virtual**:

- No implementation required in base class
- Derived classes **must** provide an implementation
- Makes the class **abstract** — cannot be instantiated
- Defines an **interface** that derived classes must implement

Abstract Class (C++)

An **abstract class** is a class that contains **at least one pure virtual function** and therefore **cannot be instantiated**. It is used to define a common **interface and behavior** for derived classes.

Key Properties

- Cannot create objects directly
- Used only as a base class
- Enables runtime polymorphism
- Can contain data members and implemented functions

Example

```
class Shape {
public:
    virtual void draw() = 0;  // pure virtual
};
```

Note: Any class with at least one pure virtual function is abstract.

Implementing Abstract Classes

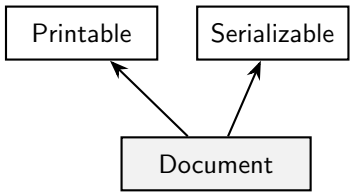
```
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}

    void draw() const override {
        cout << "Drawing circle" << endl;
    }
    double area() const override {
        return 3.14159 * radius * radius;
    }
};

Circle c(5.0);  // OK: All pure virtuals implemented!

// Partial implementation - still abstract!
class Polygon : public Shape {
public:
    double area() const override { /* ... */ }
    // draw() not implemented
};

Polygon p;  // ERROR: draw() not implemented!
```

Virtual Destructor Rules

Rule: If a class has any virtual function, make destructor virtual.

Situation	**Virtual Destructor?**
Class has virtual functions	✓ Yes, always
Class is designed for inheritance	✓ Yes
Will be deleted via base pointer	✓ Yes
Abstract base class (interface)	✓ Yes
Class is marked `final`	Not required
Simple value type, no inheritance	Not required

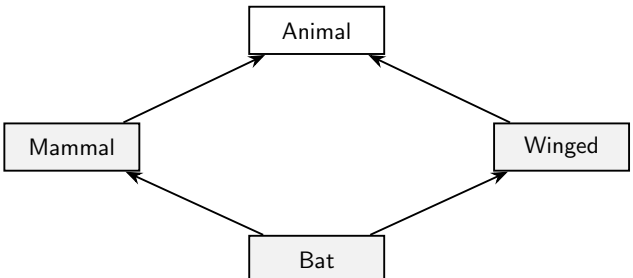
Multiple Inheritance




```
class Printable {
public:
    virtual void print() = 0;
};

class Serializable {
public:
    virtual void save() = 0;
};

class Document
  : public Printable,
    public Serializable {
public:
    void print() override { }
    void save() override { }
};
```

Caution: Multiple inheritance adds complexity. Use sparingly!

The Diamond Problem

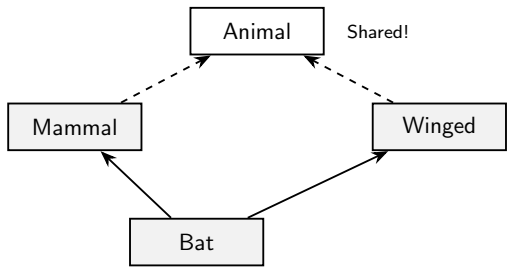


Two copies of `Animal`!

```
class Animal {
public:
    string name;
    void eat();
};

class Mammal : public Animal { };
class Winged : public Animal { };

class Bat : public Mammal,
            public Winged { };

Bat b;
b.eat();   // ERROR: Ambiguous!
b.name;    // ERROR: Which name?
```

Virtual Inheritance: The Solution

```
class Animal {
public:
    string name;
    void eat();
};

class Mammal
  : virtual public Animal { };

class Winged
  : virtual public Animal { };

class Bat
  : public Mammal,
    public Winged { };

Bat b;
b.eat();  // OK: One Animal!
b.name;   // OK: One name!
```



`virtual` inheritance ensures only **one copy** of the base.

Note: Adds overhead and complexity.

Inheritance vs Composition

Inheritance (is-a):

```
class Car : public Engine {
    // Car IS-A Engine?
    // WRONG!
};
```

When to use:

- True "is-a" relationship
- Need polymorphism
- Extending an interface

Composition (has-a):

```
class Car {
    Engine engine;
    // Car HAS-A Engine
    // CORRECT!
};
```

When to use:

- "Has-a" relationship
- Need flexibility
- Reusing implementation

Guideline: Prefer composition over inheritance.

Best Practices

1. Access polymorphic objects through pointers and references
2. Use abstract classes to focus design on clean interfaces
3. Use `override` to make overriding explicit
4. Only `use final` sparingly
5. A class with a virtual function should have a virtual destructor
6. Do not call virtual methods in constructors or destructors
7. Do not use default parameters in virtual methods
8. An abstract class typically does not need a constructor
9. Prefer private members for implementation details
10. Prefer public members for interfaces
11. Use protected members carefully; do not declare data members protected

References

- Bjarne Stroustrup, *The C++ Programming Language*, 4th Edition, Addison–Wesley, 2013.
- Chapter 20: **Derived Classes**
- Federico Busato, *Modern C++ Programming*, `https://federico-busato.github.io/Modern-CPP-Programming/`

Additional explanations and examples adapted from the official C++ language specification and cppreference.com.