fit@hcmus

Object-Oriented Programming

Polymorphism

Tran Duy Hoang

Object Slicing

- When a **derived class object** is assigned to a **base class variable**
- The extra **attributes and methods** of the derived class are **sliced off**

```
class Animal {};
class Dog : public Animal {};

int main()
{
    Dog d;
    Animal a = d; // object slicing
}
```

Base Class Pointer

- A **base class pointer** can point to a **derived class object**

```
class Animal {};
class Dog : public Animal {};

int main()
{
    Animal* pA; // base class pointer
    Dog d;

    pA = &d; // point to derived object
    pA = new Dog(); // point to allocated derived object
}
```

Base Class Reference

- A **base class reference** can alias a **derived class object**

```
class Animal {};
class Dog : public Animal {};

void makeTalk(Animal& a) {}

int main()
{
    Dog d;
    makeTalk(d); // pass a derived object
}
```

Static Binding

- **Static binding** (or **early binding**) means that the function to be called is **determined at compile time**

```
class Animal {
public:
    void talk() {
        cout << "Animal talks\n";
    }
};

class Dog : public Animal {
public:
    void talk() {
        cout << "Dog barks\n";
    }
};
```

```
int main() {
    Dog d;
    d.talk(); // Dog barks

    Animal* pA = &d;
    pA->talk(); // Animal talks
}
```

Virtual Function

- A **virtual function** is a member function in a **base class** that you can **override** in a **derived class**, and which is **resolved at runtime**
  - Declared using the “virtual” keyword
  - Enables dynamic binding (late binding)

```
class Animal {
public:
    virtual void talk() {
        cout << "Animal talks\n";
    }
};
```

Dynamic Binding

- **Dynamic binding** (or **late binding**) means the decision of **which function to invoke** is made at **runtime**, **based on the actual object type**
- Requires
  - A virtual function in the base class
  - Use of a pointer or reference to base class

Dynamic Binding

```
class Animal {
public:
    virtual void talk() {
        cout << "Animal talks\n";
    }
};

class Dog : public Animal {
public:
    void talk() {
        cout << "Dog barks\n";
    }
};

void makeTalk(Animal& a) {
    a.talk();
}
```

```
int main() {
    Dog d;
    d.talk(); // Dog barks

    Animal* pA = &d;
    pA->talk(); // Dog barks

    makeTalk(d); // Dog barks
}
```

Exercise

```
class Animal {
public:
    virtual void talk() {
        cout << "Animal talks\n";
    }
};

class Dog : public Animal {
public:
    void talk() {
        cout << "Dog barks\n";
    }
};

void makeTalk(Animal& a) {
    a.talk();
}
```

```
int main() {
    Animal a;
    a.talk(); // A?

    Animal* pA = new Animal();
    a->talk(); // B?

    Dog d;
    d.talk(); // C?

    Dog* pD = new Dog();
    pD->talk(); // D?

    a = d;
    a.talk(); // E?

    pA = &d;
    pA->talk(); // F?

    pA = pD;
    pA->talk(); // G?
}
```

Exercise

```
class A {
public:
[yyy] void f1() { cout << “Good morning.\n”; f2(); }
[zzz] void f2() { cout << “Good afternoon.\n”; }
};
class B: public A {
public:
        void f1() { cout << “Good evening.\n”; f2(); }
        void f2() { cout << “Good night.\n”; }
};
void main()
{
        A   *p = new B;
        p->f1();
}
```

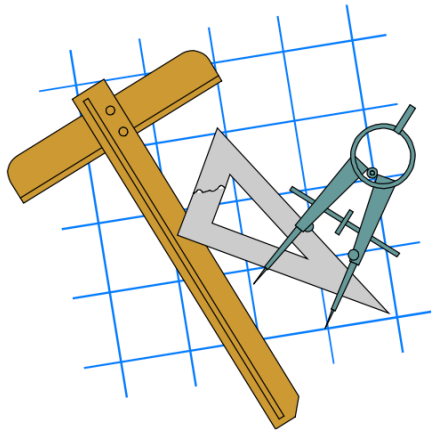
What are displayed on screen of each of the following cases:

a) [yyy] empty, [zzz] empty.

b) [yyy] empty, [zzz] virtual.

c) [yyy] virtual, [zzz] empty.

d) [yyy] virtual, [zzz] virtual.



What is Polymorphism?

- The ability of a function or an object to **behave differently** based on the context

- Types of Polymorphism in C++
  - Compile-time (Function Overloading, Operator Overloading)
  - Runtime (Virtual Functions + Dynamic Binding)

Keywords “override” & “final”

- “override” ensures that a function is overriding a virtual function
  - the use of “override” is optional
- “final” prevents further overriding

```
class Dog : public Animal {
public:
    void talk override final {
        cout << "Dog barks\n";
    }
};
```

Pure Virtual Functions

- Has declaration only, no implementation
- Syntax: virtual <function signature> **= 0**;
- Derived class provides implementation

```
class Animal {
public:
    virtual void talk() = 0;
};
```

```
class Dog : public Animal {
public:
    void talk() {
        cout << "Dog barks\n";
    }
};
```

Abstract Class

- A class which has one or more pure virtual functions becomes abstract class
- No objects of the abstract class can be created

```
int main() {
    Animal a; // error

    Dog d; // OK
    d.talk();
}
```

Abstract Class

- A pure virtual function that is not overridden in a derived class remains a pure virtual function, so the derived class is also an abstract class.

- An important use of abstract classes is to provide an **interface** without exposing any implementation details.

Quiz - Abstract Class

- Contains pure virtual functions?
- Can have regular member functions?
- Can have data members?
- Can be inherited?
- Can be instantiated?

Virtual Destructor

- NO virtual constructor!
  - Constructor is used to initialize the class itself
- Virtual destructor: YES!
  - The destructor should be virtual in order to free the memory/resource of the correct object

```
class Base {
public:
    virtual ~Base();
};
```

Common Mistakes

- Forgetting to use "**virtual**" → no polymorphism
- Slicing: assigning derived object to base object directly (not pointer/reference)
- Not declaring destructor **virtual** (may lead to memory leaks)

Summary

- Virtual functions allow dynamic dispatch in inheritance hierarchies
- Key to runtime polymorphism in C++
- Proper usage ensures flexible and maintainable OOP design
- Use “override”, “final”, and virtual destructors for safety

Quick Quiz

- What happens if a base pointer calls a non-virtual function of a derived object?
- How do virtual functions enable runtime polymorphism?
- What is the effect of not having a virtual destructor?

Exercise 7.1

- You are going to design a small system for a company that have different types of employees, such as engineers and managers. The company wants to be able to:
  - Store a list of employees (*hint. vector<Employee*> empList*)
  - Ask each employee to describe their work (*hint. empList[i]->describeWork()*)
  - Identify the role of each employee (*hint. empList[i]->getRole()*)
  - Calculate the total employee's salary (*hint. totalSalary =+ empList[i]->getSalary()*)
    - Engineer: a base salary plus an overtime bonus (*base + time*bonus_rate*)
    - Manager: a base salary plus bonus based on team size (*base + size*bonus_rate*)