Object-Oriented Programming

C++

Inheritance and Polymorphism

CSIE Department, NTUT

Woei-Kae Chen
Introduction to inheritance

- **Inheritance**
  - **Derived class**
    - inherits data members and member functions from **base class**
  - **Single inheritance**
    - inherits from one base class
  - **Multiple inheritance**
    - inherits from multiple base classes
  - **Types of inheritance**
    - **public**
    - **private**
    - **protected**

Alternative to composition

"Is a" relationship
# Inheritance Examples 1

<table>
<thead>
<tr>
<th>Base class</th>
<th>Derived classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>GraduateStudent, UndergraduateStudent</td>
</tr>
<tr>
<td>Shape</td>
<td>Circle, Triangle, Rectangle</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan, HomeImprovementLoan, MortgageLoan</td>
</tr>
<tr>
<td>Employee</td>
<td>FacultyMember, StaffMember</td>
</tr>
<tr>
<td>Account</td>
<td>CheckingAccount, SavingsAccount</td>
</tr>
</tbody>
</table>
Inheritance Examples 2

Base class (或super class)

CommunityMember

Employee

Student

Alumnus

Faculty

Staff

Administrator

Teacher

AdministratorTeacher

Single inheritance

Single inheritance

Single inheritance

Derived class (或subclass)

Multiple inheritance
Inheritance Examples 3

- Shape
  - TwoDimensionalShape
    - Circle
    - Square
    - Triangle
  - ThreeDimensionalShape
    - Sphere
    - Cube
    - Tetrahedron
“is-a” vs. “has-a” relationship

◆ “is-a”
  - Inheritance
  - Derived class object treated as base class object
  - Example: Car is a vehicle

◆ “has-a”
  - Composition
  - Object contains one or more objects of other classes as members
  - Example: Car has a steering wheel
UML Example (C++): Inheritance

```
class X {
...
};
class Y : public X {
...
...}
```

UML: Inheritance

UML: Class

Shape

Circle

X

Y
UML Example (C++): Composition

class Employee {  
    ...  
    ...  
    Date d;  // Composition  
};
Casting Inherited Objects and Pointers

**Casting objects**
- **Downcasting:** allowed; safe
  
  ```
  baseObject = derivedObject;
  ```
- **Upcasting:** not allowed
  - Use operator overload to allow upcasting

**Casting pointers**
- **Downcasting:** allowed; safe
  
  ```
  basePtr = derivedPtr;
  ```
- **Upcasting:** allowed (**casting required**); unsafe
  
  ```
  derivedPtr = static_cast<derivedClass *> (basePtr);
  ```
Casting Inherited Objects and Pointers

<table>
<thead>
<tr>
<th>Downcasting</th>
<th>Upcasting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td><strong>Pointer</strong></td>
</tr>
<tr>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>bo = do;</td>
<td>bp = dp;</td>
</tr>
<tr>
<td>Not allowed</td>
<td>Casting allowed</td>
</tr>
<tr>
<td>dp = (D *) bp;</td>
<td></td>
</tr>
</tbody>
</table>

Use operator= to enable upcasting or C++ static_cast
Casting Unrelated Objects and Pointers

<table>
<thead>
<tr>
<th>Object</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X \leftarrow Y ) Not allowed</td>
<td>Casting allowed ( px = (X*) ) py;</td>
</tr>
</tbody>
</table>

- Use operator= to enable casting
- or C++ reinterpret_cast
### Casting Inherited Objects

#### base class object $\leftrightarrow$ derived class object

```cpp
class B {
    ...
    int b;
    
};

class D : public B {
    ...
    int d;

};

B x; D y;
x = y; // ok
```

#### derived class object $\leftrightarrow$ base class object

```cpp
y = x; // error
y = (D) x; // error
y = static_cast<D> (y); // error
```

Use operator= to enable upcasting
Casting Unrelated Objects

\[ \text{X class object} \leftarrow \text{Y class object} \]

class X {
    ...
    int x;
};

class Y {
    ...
    int y;
}

...

X x, x1; Y y;
x = x1;  // ok
x = y;   // error
x = (X) y; // error
x = static_cast<X>(y); // error
Downcasting Pointers

pointer to base class object 🃶 pointer to derived class object

class B {
    ...
    void bf();
    int b;
    
    ...
}

class D : public B {
    ...
    void df();
    int d;
    
    ...
}

B *px; D y,*py=&y;
px = py;   // ok: downcast
px = &y;   // ok: downcast
px->bf();  // ok
px->df();  // error
Upcasting Pointers

pointer to derived class object $\leftrightarrow$ pointer to base class object

```cpp
class B {
    ...
    void bf();
    int b;
};

class D : public B {
    ...
    void df();
    int d;
};

B x,*px; D y, *py;
px = &y;  // ok: downcast
py = px;  // error
py = (D *) px;  // ok: upcast
py->bf();  // ok
py = (D *) &x;  // ok: but, buggy
```
Casting Pointer of Unrelated Objects

pointer to X class object \(\longleftrightarrow\) pointer to Y class object

```cpp
class X {

    ...

    int x;

};

class Y {

    ...

    int y;

};

...

X px; Y py;

px = py;       // error
px = (X *) py; // ok
px = static_cast<X *> (py); // error
px = reinterpret_cast<X *> (py); // ok
```
Introduction to polymorphism

Problem:
A list of unknown shapes?

```
Shape
| draw()  |
| ...()   |

Circle
| draw()  |
| ...()   |

Rectangle
| draw()  |
| ...()   |

Triangle
| draw()  |
| ...()   |
```
Declaring a list of **unknown** shapes

Shape s[100];   // Bad
Circle s[100];    // Bad
Shape s[100];    // Bad
Shape *s[100];   // OK, 但怎麼呼叫 draw()?

1. 混合type的shape 怎麼儲存在array?
2. 怎麼呼叫draw()? 

**Solution:** 設成virtual function

```
Shape draw()
...()  // does not work
```

```
s[0] = new Circle;
s[1] = new Triangle;
...
for (i=0; i<100; i++)
  s[i]->draw(); // ???
```
Effect of Virtual Functions

Example (non-virtual)
```cpp
class X {
public:
  void f() {
    cout << 'x';
  }
};
class Y : public X {
public:
  void f() {cout<<'y';}
};
...
X a; a.f();  // x
Y b; b.f();  // y
X *p=&a;
p->f();      // x
X *p=&b;
p->f();      // x
```

Example (virtual)
```cpp
class X {
public:
  virtual void f() {
    cout << 'x';
  }
};
class Y : public X {
public:
  void f() {cout<<'y';}
};
...
X a; a.f();  // x
Y b; b.f();  // y
X *p=&a;
p->f();      // x
X *p=&b;
p->f();      // y
```

virtual只對pointer或reference有效(dynamic binding)
Virtual Functions for the shape problem

```java
Shape *s;
s = new Circle;    s->draw(); // 呼叫circle的draw()
s = new Triangle;  s->draw(); // 呼叫Triangle的draw()
...
Shape *s[100];
... // 設定s[0..99]分別指向哪一種物件
for (int i=0;i<100;i++)
    s[i]->draw(); // 依據實際的type呼叫draw()
```
Virtual Functions & Polymorphism

Polymorphism
- Ability for objects of different classes to respond differently to the same function call
- Implemented through virtual functions
  - C++ 允许使用 base class pointer (或 reference) 呼叫 derived class 的 member function
  - C++ chooses the correct overridden function in object

Dynamic binding
- Function determined during execution time
  - ShapePtr->Draw();

Static binding
- Function determined during compile-time
  - Pointer 可能指向任何 derivedObject
  - ShapeObject.Draw();
Dynamic Binding ⇔ Static Binding

- **Dynamic binding (run-time)**
  - 用base class的pointer或reference呼叫virtual function

- **Static binding (compile time)**
  - 其他(用object呼叫virtual/non-virtual function，用pointer或reference呼叫non-virtual function)

```cpp
class X {
public:
    void f1() {
        cout << "xf1";
    }
    virtual void f2() {
        cout << "xf2";
    }
};
class Y : public X {
public:
    void f1(){cout<<"yf1";}
    void f2(){cout<<"yf2";}
};

X a;
a.f1();    // xf1 static
a.f2();    // xf2 static
Y b;
b.f1();    // yf1 static
b.f2();    // yf2 static
X *p=&a;
p->f1();    // xf1 static
p->f2();    // xf2 dynamic
X *pb=&b;
pb->f1();    // xf1 static
pb->f2();    // yf2 dynamic
```
Abstract and Concrete Classes

- **Abstract classes**
  - To provide a base class for other classes
  - *No* objects can be instantiated
  - Declare one or more **pure virtual** functions
  - Example
    ```
    virtual void draw() = 0;
    ```

- **Concrete classes**
  - Objects can be instantiated
  - *No* unimplemented **virtual** functions
    - Implement all unimplemented **virtual** functions
Abstract and Concrete Classes

C++ example

class Shape {
public:
    virtual void draw() = 0;
    void setColor(int c) {
        color = c;
    }
private:
    int color;
};

class Rectangle : public Shape {
public:
    void draw() {
        ...
    }
    void getArea() {
        ...
    }
    virtual void f() {
        ...
    }
};
Abstract and Concrete Classes
UML example

Abstract class

Concrete class

pure virtual function

implementation
Virtual Destructors

Problem:
- How to delete an object with a base class pointer?

Solution:
- declare a **virtual** (base-class) destructor

```cpp
class X {
public:
    virtual ~X(); // virtual destructor
    ...
private:
    ...
};
class Y : public X {
    ...
};
X *pa = new Y;
... // use *pa
... delete pa;
```
How does compiler handle virtual functions?

**virtual** functions

- overheads?
  - memory
  - speed
- How it works?
  - dynamic binding
  - vtable

Diagram:

```
  Shape
     ↑
  Point
     ↑
  Circle
     ↑
Cylinder
```

Point

Circle

Cylinder

Abstract class
Each class has a vtable

Each object has a vtable's pointer

**virtual functions**

- Shape
  - area()
  - volume()
  - PSN()
  - print()
- Point
  - x
  - y
  - PSN()
  - print()
- Circle
  - radius
  - area()
  - PSN()
  - print()
- Cylinder
  - height
  - area()
  - volume()
  - PSN()
  - print()

Key:
- `a` = area function
- `v` = volume function
- `psn` = printShapeName function
- `pr` = print function
- `0` entry means pure virtual function
- `r` = radius; `h` = height

Example:
- `bp->PSN();`