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

"Is a" relationship

Alternative to composition
## Inheritance Examples 1

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

- **Base class** (或super class)
- **CommunityMember**
  - **Employee**
  - **Student**
  - **Alumnus**
  - **Faculty**
  - **Staff**
  - **Administrator**
  - **Teacher**
  - **AdministratorTeacher**

- **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 {
    ...
};
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;`
    - `derivedPtr = (derivedClass *) basePtr;`
# Casting Inherited Objects and Pointers

<table>
<thead>
<tr>
<th>Downcasting</th>
<th>Object</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed</td>
<td>bo = do;</td>
<td>Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bp = dp;</td>
</tr>
</tbody>
</table>

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

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

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

Use `operator=` to enable casting

or C++ `reinterpret_cast`

```c
X \ast px;
Y \ast py;
```
Casting Inherited Objects

**base class object ↔ derived class object**

```cpp
class B {
...
int b;
};
...
B x; D y;
```

```cpp
x = y;  // ok
```

**derived class object ↔ 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

- **X class object ↔ Y class object**

```cpp
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
```

- Use `operator=` to enable assignment
Downcasting Pointers

pointer to base class object \(\leftarrow\) pointer to derived class object

class B {
    ...
    void bf();
    int b;
    
    ...
    
    B *px; D y,*py=&y;
    px = py;    // ok: downcast
    px = &y;    // ok: downcast
    px->bf();   // ok
    px->df();   // error
}

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

pointer to derived class object ↔ pointer to base class object

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

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

```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()?

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

**Solution:** 設成 virtual function

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

Circle
  draw()
  ...()

Rectangle
  draw()
  ...()

Triangle
  draw()
  ...()
```

Does NOT work
## 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

设成virtual function

**Java**?

```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++ 允許使用 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**
  
  ```c++
  ShapePtr->Draw();
  ```

Static binding

- Function determined during **compile-time**
  
  ```c++
  ShapeObject.Draw();
  ```

**Pointer** 可能指向任何 **derivedObject**
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

```cpp
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 class**
- **Concrete class**
- **Not pure virtual**
Abstract and Concrete Classes

UML example

- **Shape**
  - `draw()`
  - `setColor()`
  - `...()`

- **Circle**
  - `draw()`
  - `...()`

- **Rectangle**
  - `draw()`
  - `...()`

- **Triangle**
  - `draw()`
  - `...()`
Virtual Destructors

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

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

```
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
Each class has a vtable.

Each object has a vtable pointer.

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

The flow of `virtual` function call:

`baseClassPtr->printShapeName();` is illustrated by the bold arrows above.

1. pass `&circle` to `baseClassPtr`
2. get to `Circle` object
3. get to `Circle` vtable
4. get to `printShapeName` pointer in vtable
5. execute `printShapeName` for `Circle`