

数据结构与算法

DATA STRUCTURE

第五讲 类的继承和虚函数

信息管理与工程学院

2017 - 2018 第一学期

回顾

- 类封装
- 动态内存分配和管理
- 运算符重载
- 写一个自定义类**IntArray**

创建类IntArray

- 要求内部动态分配数组内存
- 注意释放资源 (rule of three)
- 支持常用操作 (Length(), [], <<, etc)
- 注意边界检查
- 注意常量const对象使用

步骤一：决定数据元素

1. 指向int数组的指针

- 因为我们要动态决定数组的大小，不希望固定大小
- `private int * _pData;`

2. 数组的长度

- 用来告诉用户数组大小
- 用来检查数组元素读取时候的越界问题
- `private int _size;`

步骤二：如何分配动态内存和释放

- 配对进行
 - `pData = new int[sz]`
 - `delete [] pData`
 - 分配应该在构造函数，释放应该在析构函数里
- 别忘了 **rule of three**
 - 写了析构函数释放资源 `~IntArray()`
 - 必须写复制构造函数 `IntArray(const IntArray & other)`
 - 和赋值重载函数 `IntArray & operator= (const IntArray & rhs)`

步骤三：数组下标操作符[]

- 对自定义类型IntArray重载[]
 - `int & operator[] (int index)`
 - 返回数组内部元素的引用
 - 用assert做越界检查
- Const对象也要能用下表操作符[]
 - `const int & operator[] (int index) const`
 - 加const重载[], 实现内容一样

步骤四：别的补充

- 返回长度
 - `int Length() const`
 - 这样`const`对象也能调用
- 重新申请数组
 - `void Resize(int size);`
 - 同样需要释放之前的内存，然后申请新的数组内存
- 添加功能过程中，有些重复用到的代码块，可以考虑变成私有函数调用，比如分配动态数组内存`getArray()`，释放内存`Reset()`

类的继承

对象之间关系

- 基本的数据结构
 - array
 - List
- 衍生的数据结构
 - sortedArray, stack, queue
 - 树, 图

对象之间关系

- 抽象出问题的本质
- 把复杂事物划分成相对独立的小对象
- 可以重用代码 (引用次数)
- 使得逻辑分明，易于扩展和维护

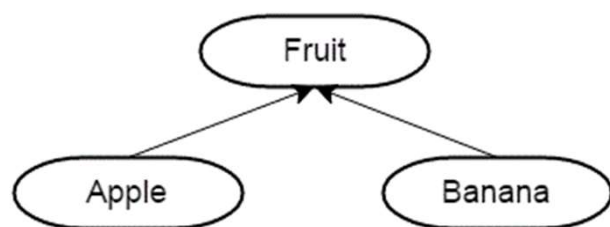
一种对象之间关系（组合）

对象具体实例	组合Composition	聚合Aggregation	联合Association
关系类型	部分--整体	部分--整体	没有从属关系
成员可以属于多个类	No（心脏不能共享）	Yes（老师可以共享）	Yes（病人可以看多个医生）
成员由类建立/销毁	Yes（心脏不能单独存在）	No（老师可以单独存在）	No（病人和医生不共生）
关系描述语	Part-of	Has-a	Uses-a
例子	人有一颗心脏	系里有几个老师	医生和病人
使用	一般使用正常成员定义(指针也可以) 负责建立和销毁	使用指针或者引用定义成员 不负责建立和销毁	使用指针或者引用定义成员 不负责建立和销毁

还有一种更弱的依赖关系（dependency），只是在类要执行某个任务时候才有的关系。
比如你只有腿扭了才需要拐杖，Point class输出到屏幕时候才需要ostream。

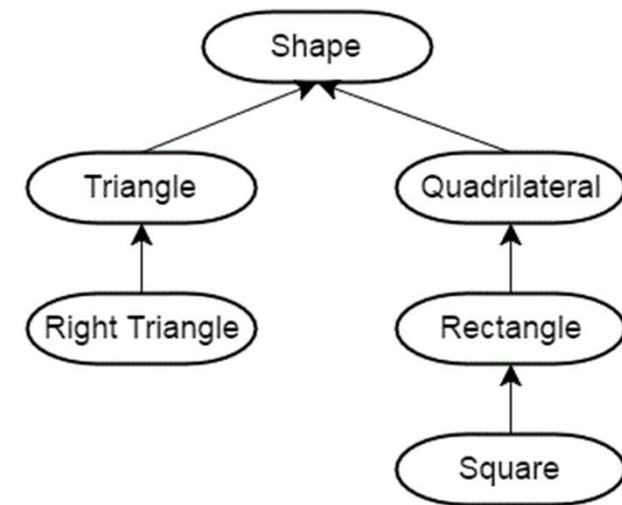
另一种对象之间关系（继承）

- 组合和聚合适合于has-a关系
- 另外一种主要内因关系：is-a关系，用继承来描述
- 比如苹果和香蕉都属于（is-a）水果。



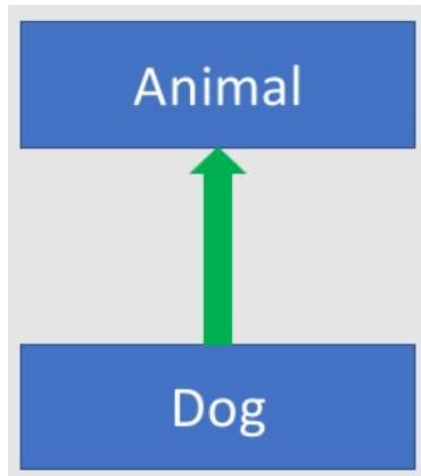
继承Inheritance

- 基类/衍生类，父类/子类
- 子类包含父类的数据和函数
- 子类不能使用父类private数据和函数



```
Class Shape(...); //Base class Shape
Class Triangle : public Shape(...); // Triangle derived from Shape
Class RightTriangle : public Triangle(...); // RightTriangle derived from Triangle
```

继承例子



```
class Animal {
    public:
        int legs = 4;
};
class Dog : public Animal {
    public:
        int tail = 1;
};
int main() {
    Dog d;
    cout << "Legs are: " << d.legs << endl;
    cout << "Tail is: " << d.tail;
}
```

子类和父类的构造函数和析构函数

- 构造函数和析构函数不能被继承
- 子类的构造函数执行时，父类的构造函数先被执行
- 子类的析构函数先被执行，然后是父类的析构函数

构造函数和析构函数顺序

```
class A
{
public:
    A() { cout << " A" << endl; }
    ~A() { cout << "~A" << endl; }
};

class B : public A
{
public:
    B() { cout << " B" << endl; }
    ~B() { cout << "~B" << endl; }
};

class C : public B
{
public:
    C() { cout << " C" << endl; }
    ~C() { cout << "~C" << endl; }
};

int main()
{
    cout << "Construct C object" << endl;
    C object;
    cout << "Destroy C object" << endl;
    return 0;
}
```



```
Construct C object
A
B
C
Destroy C object
~C
~B
~A
```


基类构建过程

- 分配内存
- 调用基类的构造函数
- 初始化列表
- 构造函数体
- 返回caller

```
class Base
{
public:
    int _id;
    Base(int id) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0) : _cost(cost) {}
    double getCost() const { return _cost; }
};

int main()
{
    Base base(5);
    return 0;
}
```

子类构建过程

- 分配内存（子类和父类）
- 调用子类的构造函数
- 首先调用基类构造函数，如果不指定就调用基类缺省构造函数
- 初始化子类列表
- 构造函数体
- 返回caller

```
class Base
{
public:
    int _id;
    Base(int id = 0) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0) : _cost(cost) {}
    double getCost() const { return _cost; }
};

int main()
{
    Derived obj(1.3);
    return 0;
}
```

怎么设定基类的成员数据

- 尝试在初始化列表里进行
- 错误，因为已经在基类构造函数里初始化了，不能初始化两次
- 可以改成在构造函数体里赋值。
- 但如果_id是const，这样也不行。

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
    :
        _cost(cost),
        _id(id) // ERROR: cannot initialize here
    {}
    double getCost() const { return _cost; }
};
```

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
    :
        _cost(cost)
    {
        _id = id; // OK, 赋值可以
    }
    double getCost() const { return _cost; }
};
```

调用基类构造函数

衍生类构造函数 基类构造函数

Derived:: Derived(cost, id) : **Base**(id)

 衍生类构造函数参数 基类构造函数参数

- 如果基类没有缺省构造函数的话，必须显式调用别的构造函数

设定基类的成员数据

- 分配内存（子类和父类）
- 调用子类的构造函数
- 首先调用基类构造函数，就是指定的Base(id)
- 初始化基类数据
- 返回到子类构造函数
- 初始化子类列表
- 构造函数体
- 返回caller

```
class Derived : public Base
{
public:
    double _cost;
    Derived(double cost = 0.0, int id = 0)
    :
        Base(id),
        _cost(cost)
    {
    }
    double getCost() const { return _cost; }
};
```

把数据变成private

```
class Base
{
private:
    int _id;
public:
    Base(int id = 0) : _id(id) {}
    int getId() const { return _id; }
};

class Derived : public Base
{
private:
    double _cost;
public:
    Derived(double cost = 0.0, int id = 0)
    :
        Base(id),
        _cost(cost)
    {
    }
    double getCost() const { return _cost; }
};
```

类继承方式

- 公有继承
 - `Class Triangle : public Shape(...);`
- 保护继承
 - `Class Triangle : protected Shape(...);`
- 私有继承
 - `Class Triangle : private Shape(...);`
 - 如果子类和父类关系不大，不想子类公开父类方法

继承方式和权限

Base class members

子类对父类数据的权限

```
private: x  
protected: y  
public: z
```

private
base class

```
x is inaccessible  
private: y  
private: z
```

```
private: x  
protected: y  
public: z
```

protected
base class

```
x is inaccessible  
protected: y  
protected: z
```

```
private: x  
protected: y  
public: z
```

public
base class

```
x is inaccessible  
protected: y  
public: z
```


继承方式和权限

class Grade
<pre>private members: char letter; float score; void calcGrade(); public members: void setScore(float); float getScore(); char getLetter();</pre>

class Test : public Grade
<pre>private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int);</pre>


Test **class** 共有继承 Grade
class 后: —————→

<pre>private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int); void setScore(float); float getScore(); char getLetter();</pre>

继承方式和权限

class Grade
private members: char letter; float score; void calcGrade(); public members: void setScore(float); float getScore(); char getLetter();

class Test : protected Grade
private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int);

Test **class** 保护继承 Grade
class 后: 

private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int); protected members: void setScore (float); float getScore (); float getLetter ();

继承方式和权限

class Grade
private members: char letter; float score; void calcGrade(); public members: void setScore(float); float getScore(); char getLetter();

class Test : private Grade
private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int);

Test class 私有继承 Grade
class 后: →

private members: int numQuestions; float pointsEach; int numMissed; void setScore(float) ; float getScore() ; float getLetter() ; public members: Test(int, int);
--

问题

- 子类继承了基类public函数
- 子类对于父类的有些行为表现不一样
 - 比如动物都会叫，但是叫法不一样
 - 车辆都能跑，速度和方式不一样

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    int identity() const { cout << "Base " << _id << endl; }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
    : Base(id)
    {
    }
};

int main()
{
    Base base(5);
    base.identity();

    Derived derived(7);
    derived.identity();
    return 0;
}
```

重定义基类函数

- 子类的函数可以用一样的函数名和参数列表（数据类型，数目，顺序）来重定义
- 一般是用来替换基类的函数
- 不是重载，参数列表不一样
- 子类对象用子类函数；父类对象用父类函数
- 子类对象调用父类重定义函数，必须加scope， `Base::Identity()`

重定义基类函数

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    int identity() const { cout << "Base " << _id << endl; }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
    : Base(id)
    {
    }

    int identity() const { cout << "Derived " << endl; }
};

int main()
{
    Base base(5);
    base.identity();

    Derived derived(7);
    derived.identity();
    return 0;
}
```



```
int identity() const
{
    cout << "Derived " << endl;
    Base::identity();
}
```

重定义问题

- 考虑:
 - **Class** Base 定义函数 `identity()` 和 `ShowMe()`.
 - `ShowMe()` 调用 `identity()`.
 - **Class** Derived 继承 Base 并重定义函数 `identity()`.
 - 建立一个 Derived 对象, 并调用函数 `ShowMe()`.
 - 调用 `ShowMe()` 时, `identity()` 也被调用了

问题是基类还是子类里的`identity()`被调用了?

重定义问题

- 基类的identity()被调用了。
- 因为函数调用是在编译阶段决定的，叫静态绑定

```
class Base
{
private:
    int _id;
public:
    Base(int id) : _id(id) {}
    void Identity() const { cout << "Base " << _id << endl; }
    void ShowMe() const { Identity(); }
};

class Derived : public Base
{
public:
    Derived(int id = 0)
    :
        Base(id)
    {
    }

    void Identity() const
    {
        cout << "Derived " << endl;
        Base::Identity();
    }
};

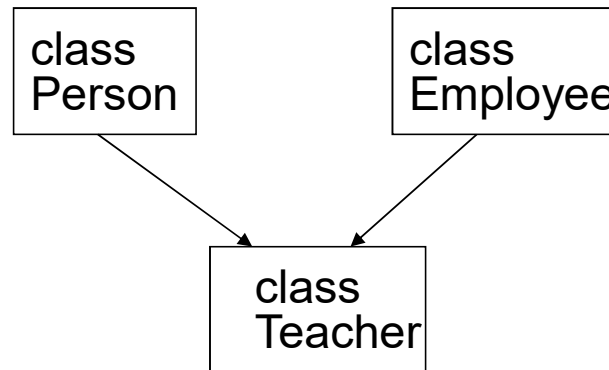
int main()
{
    Derived derived(7);
    derived.ShowMe();

    Base * pDerived = new Derived(2);
    pDerived->ShowMe();
    return 0;
}
```


多继承

- 子类可以有不止一个父类

```
class Teacher : public Person,  
                public Employee;
```



多继承问题

- 多个基类包含相同的函数，名字冲突
- 如果是菱形继承的话，问题尤其严重
- 解决方法：
 - 子类重定义相同的函数（重定义问题）
 - 子类在调用时加上scope 运算符 ::
- 例子：iostream里的cin, cout
- 一般来说避免多继承

动态绑定

基类指针

- 可以把子类的指针**cast**到基类
- 基类指针只知道基类的数据和函数
- 子类的重定义函数（静态绑定）会被忽略

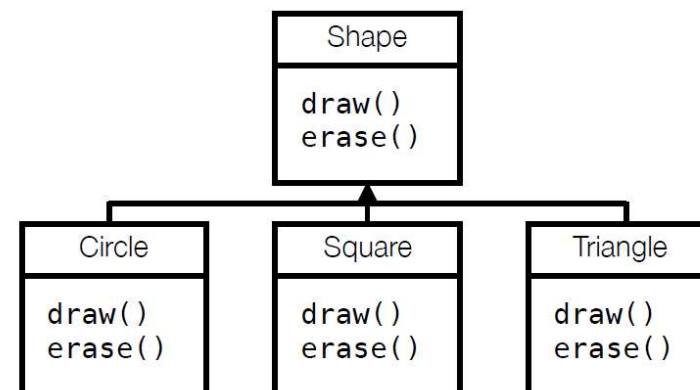
```
Base * pObject = new Derived(2);
```

虚函数和多态

- 虚函数:是指通过基类访问派生类定义的函数
- 需要在基类里加上关键词virtual:
`virtual void func() {...}`
- 虚函数支持动态绑定, 就是同一段代码在执行中根据输入决定用哪个版本, 这种行为就是多态
- 不用virtual重定义就是在编译阶段静态绑定

多态

- 多态就是不同类型的对象对于同一个事情行为不一样
- 模板和重载都是编译阶段（静态）多态
 - `Void DoSomething(const Shape& obj)`
 - `Void DoSomething(const Circle& obj)`
 - `Void DoSomething(const Square& obj)`
- 继承提供了动态绑定，多态
 - `Void DoSomething(Base * pObj)`
 - 关键需要通过基类访问派生类定义的函数



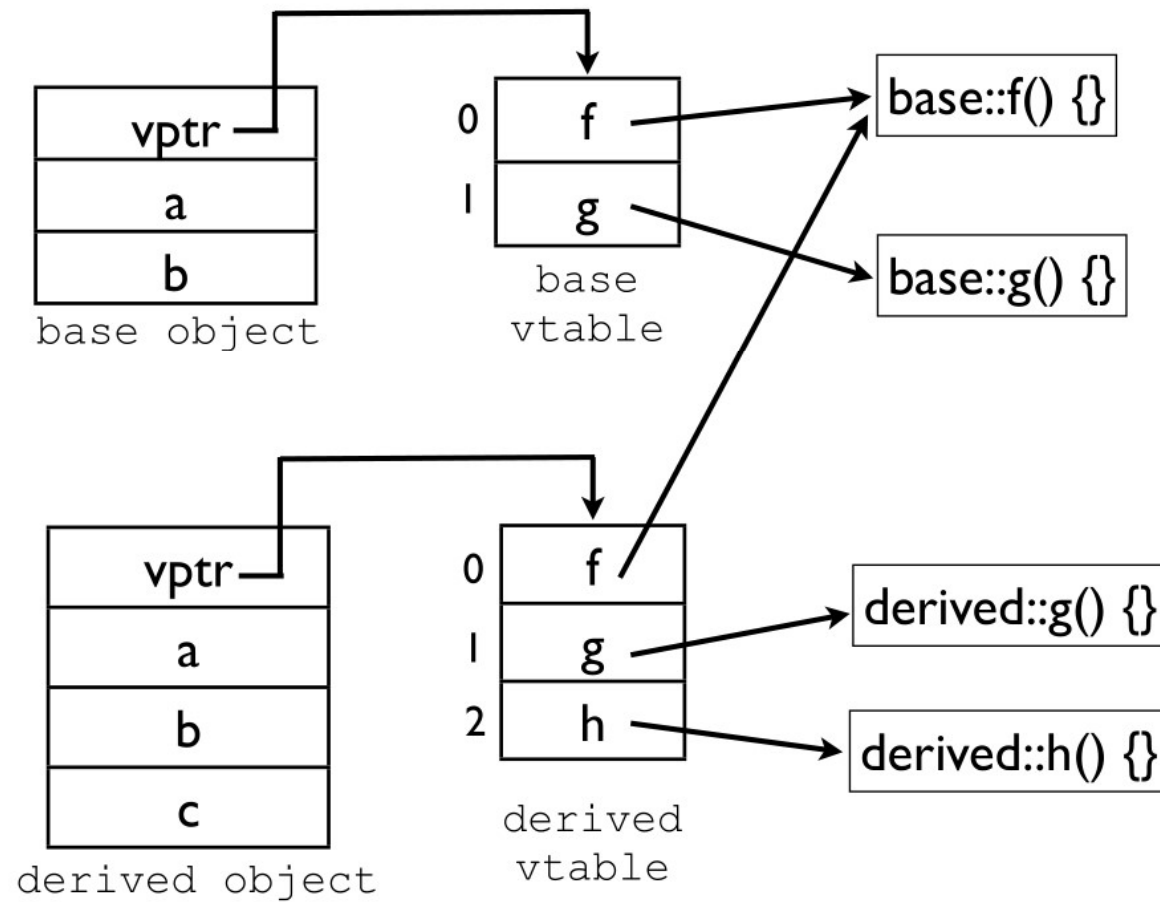
The vtable

```
struct base
{
    virtual void f();
    virtual void g();
    int a,b;
};

struct derived : base
{
    virtual void g();
    virtual void h();
    int c;
};

void poly(base * ptr)
{
    ptr->f();
    ptr->g();
}

int main()
{
    poly(&base());
    poly(&derived());
}
```



多态例子

```
class Snow {
public:
    virtual void method2() {
        cout << "Snow 2" << endl;
    }
    virtual void method3() {
        cout << "Snow 3" << endl;
    }
};

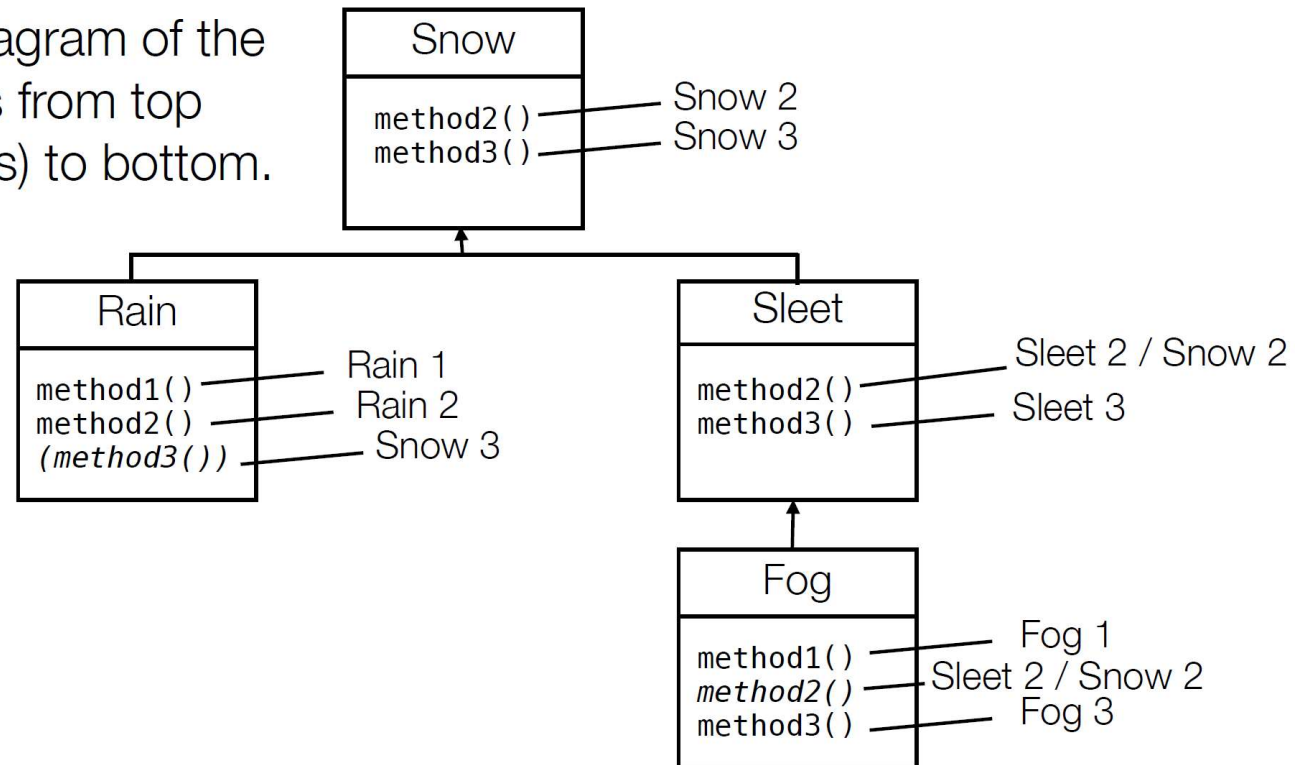
class Rain : public Snow {
public:
    virtual void method1() {
        cout << "Rain 1" << endl;
    }
    virtual void method2() {
        cout << "Rain 2" << endl;
    }
};
```

```
class Sleet : public Snow {
public:
    virtual void method2() {
        cout << "Sleet 2" << endl;
        Snow::method2();
    }
    virtual void method3() {
        cout << "Sleet 3" << endl;
    }
};

class Fog : public Sleet {
public:
    virtual void method1() {
        cout << "Fog 1" << endl;
    }
    virtual void method3() {
        cout << "Fog 3" << endl;
    }
};
```


类关系图

Draw a diagram of the classes from top (superclass) to bottom.

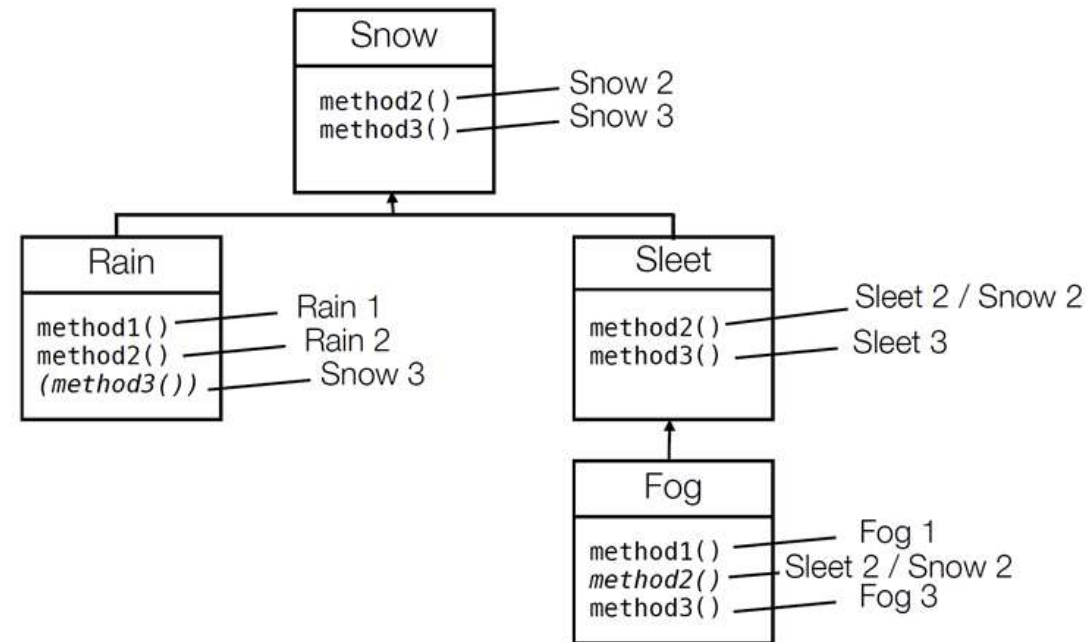


例子

```
Snow* var1 = new Sleet();  
var1->method2();
```

```
Snow* var2 = new Rain();  
var2->method1();
```

```
Snow* var3 = new Rain();  
var3->method2();
```

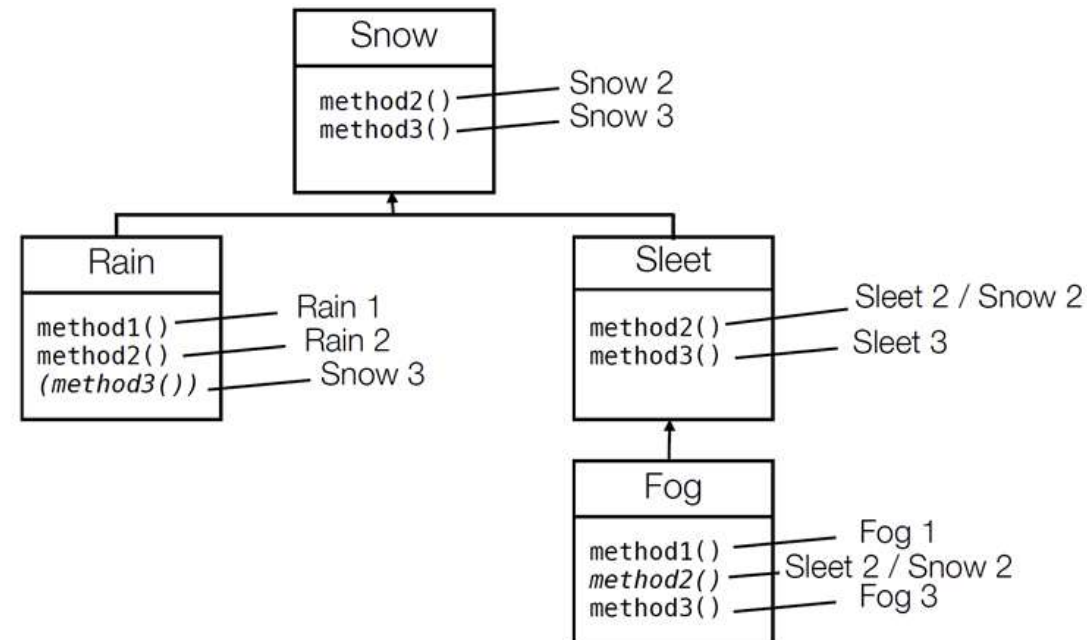


例子

```
Snow* var4 = new Rain();  
((Rain *) var4)->method1();
```

```
Snow* var5 = new Fog();  
((Sleet *) var5)->method1();
```

```
Snow* var7 = new Sleet();  
((Rain*) var7)->method1();
```



注意点

- 不要再构造函数和析构函数里调用虚函数
 - 在基类的构造函数里调用虚函数时，子类还没建立
 - 在基类的析构函数里调用虚函数时，子类已经销毁
- 虚函数有额外开销，不必要所有的函数都成为虚函数

虚析构函数

- 对于基类的析构函数，最好用virtual
- 要不然编译器会用静态绑定，容易造成内存泄漏

```
Snow* var4 = new Rain();
```

```
Delete var4;
```

- 赋值运算符重载，可以变成虚函数，不过最好不要。

抽象基类和纯虚函数

- 纯虚函数是指子类必须实现基类的虚函数
- 抽象基类包括至少一个纯虚函数:
`virtual void Y() = 0;`
- 这里 = 0 表示了纯虚函数
- 基类不能对纯虚函数定义
- 基类可以没有对象
- 抽象基类一般用来作为接口interface

抽象基类和纯虚函数

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

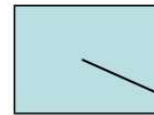
Circle's
draw()

Circle's
center()

vtbl:

draw
center

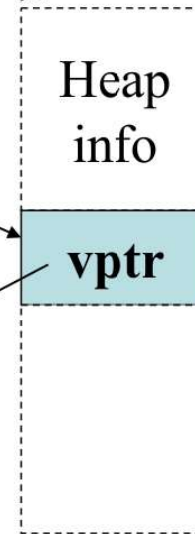
```
Shape* p = new Circle(x,20);
```



p->draw();

Heap
info

vptr



小结（重复利用代码的手段）

- 类class本身就是为了重利用代码
 - 对象有公共点（关系），也有自己的特色
 - 可以组合，也可以继承
- 函数和算术符重载`overload`
 - 重载是函数名一样，但是参数列表不一样，静态绑定
- 基类函数重定义`redefinition`
 - 是子类定义了和父类一样的函数(包括函数名和参数列表)，静态绑定
- 基类函数覆盖`override`
 - 是指基类带有`virtual`的函数被重定义，动态绑定，实现多态

Q&A

Thanks!