

directory

- review
- 组合
- 继承
- 成员访问权限
- 重写隐藏与重载
- 多重继承

review

- 拷贝构造函数：对象之间的拷贝
- 右值引用：延长临时对象的生命周期
- 移动构造函数：避免频繁的拷贝

组合

对象之间的关系

- 整体-部分 has-a
- 一般-特殊 is-a

组合

- 对象组合的两种实现方法：
 - 已有类的对象作为新类的公有数据成员，这样通过允许直接访问子对象而“提供”旧类接口
 - 已有类的对象作为新类的私有数据成员。新类可以调整旧类的对外接口，可以不使用旧类原有的接口（相当于对接口作了转换）

```
class Car{
private:
    Wheel w;
public:
    Engine e; /// 公有成员，直接访问其接口
    void setWheel(int n){w.set(n);} /// 提供私有成员的访问接口
};

int main()
{
    Car c;
    c.e.set(1);
    c.setWheel(4);
    return 0;
}
// 方法一：公有成员 直接访问对象e
// 方法二：私有成员 接口----对象w
```

- 对象构造与析构函数的次序

- 先完成子对象构造，再完成当前对象构造
- 子对象构造的次序仅由在类中声明的次序所决定
- 析构函数的次序与构造函数相反
- 回忆：隐式定义的拷贝构造与赋值运算
 - 如果调用拷贝构造函数且没有给类显式定义拷贝构造函数，编译器将提供“隐式定义的拷贝构造函数”。该函数的功能为：
 - 递归调用所有子对象的拷贝构造函数
 - 对于基础类型，采用位拷贝
- 赋值运算的默认操作类似

```
#include <iostream>
using namespace std;

class C1{
public:
    int i;
    C1(int n):i(n){}
    C1(const C1 &other) /// 显式定义拷贝构造函数
        {i=other.i; cout << "C1(const C1 &other)" << endl;}
};

class C2{
public:
    int j;
    C2(int n):j(n){}
    C2& operator= (const C2& right){/// 显式定义赋值运算符
        if(this != &right){
            j = right.j;
            cout << "operator=(const C2&)" << endl;
        }
        return *this;
    }
};

class C3{
public:
    C1 c1;
    C2 c2;
    C3():c1(0), c2(0){}
    C3(int i, int j):c1(i), c2(j){}
    void print(){cout << "c1.i = " << c1.i << " c2.j = " << c2.j << endl;}
};

int main(){
    C3 a(1, 2);
    C3 b(a); //C1执行显式定义的拷贝构造
            //C2执行隐式定义的拷贝构造

    cout << "b: ";
    b.print();
    cout << endl;
```

```

    C3 c;
    cout << "c: ";
    c.print();
    c = a; //C1执行隐式定义的拷贝赋值
           //C2执行显式定义的拷贝赋值
    cout << "c: ";
    c.print();
    return 0;
}
/*
C1(const C1 &other)
b: c1.i = 1 c2.j = 2

c: c1.i = 0 c2.j = 0
operator=(const C2&)
c: c1.i = 1 c2.j = 2

*/

```

继承

- 被继承的已有类，被称为基类(base class)，也称“父类”。
- 通过继承得到的新类，被为派生类(derived class)，也称“子类”、“扩展类”。
- 常见的继承方式：public, private
 - class Derived : [private] Base { .. }; 缺省继承方式为private继承。
 - class Derived : public Base { ... };
- protected 继承很少被使用
 - class Derived : protected Base { ... };
- 什么不能被继承？
 - 构造函数：创建派生类对象时，必须调用派生类的构造函数，派生类构造函数调用基类的构造函数，以创建派生对象的基类部分。C++11新增了继承构造函数的机制（使用using），但默认不继承
 - 析构函数：释放对象时，先调用派生类析构函数，再调用基类析构函数
 - 赋值运算符：
 - 编译器不会继承基类的赋值运算符（参数为基类）
 - 但会自动合成隐式定义的赋值运算符（参数为派生类），其功能为调用基类的赋值运算符。
 - 友元函数：本质不是类成员

```

#include <iostream>
using namespace std;

class Base{
public:
    int k = 0;
    void f(){cout << "Base::f()" << endl;}
    Base & operator= (const Base &right){
        if(this != &right){
            k = right.k;
            cout << "operator= (const Base &right)" << endl;

```

```

        }
        return *this;
    }
};
class Derive: public Base{};
int main(){
    Derive d, d2;
    cout << d.k << endl; //Base数据成员被继承
    d.f(); //Base::f()被继承

    Base e;
    //d = e; //编译错误, Base的赋值运算符不被继承
    d = d2; //调用隐式定义的赋值运算符
    return 0;
}

```

- 若没有显式调用，则编译器会自动生成一个对基类的默认构造函数的调用。

```

class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }/// 默认构造函数
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() { cout << "Derive::Derive()" << endl; }
    /// 无显式调用基类构造函数，则调用基类默认构造函数
};
int main() {
    Derive obj;
    return 0;
}
// g++ 1.cpp -o 1.out -std=c++11

```

- 若想要显式调用，则只能在派生类构造函数的初始化成员列表中进行。

```

class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }
    /// 默认构造函数
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive(int i) : Base(i) { cout << "Derive::Derive()" << endl; }
}

```

```

    /// 显式调用基类构造函数
};
int main() {
    Derive obj(356);
    return 0;
} // g++ 1.cpp -o 1.out -std=c++11

```

- 在派生类中使用 `using Base::Base;` 来继承基类构造函数，相当于给派生类“定义”了相应参数的构造函数，如

```

class Base
{
    int data;
public:
    Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
};
class Derive : public Base {
public:
    using Base::Base;          ///相当于 Derive(int i):Base(i){};
};
int main() {
    Derive obj(356);

    return 0;
} // g++ 1.cpp -o 1.out -std=c++11

```

- 当基类存在多个构造函数时，使用 `using` 会给派生类自动构造多个相应的构造函数。

```

class Base
{
    int data;
public:
    Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
    Base(int i, int j)
    { cout << "Base::Base(" << i << ", " << j << ")\n"; }
};
class Derive : public Base {
public:
    using Base::Base;          ///相当于 Derive(int i):Base(i){};
                                ///加上 Derive(int i, int j):Base(i, j){};
};
int main() {
    Derive obj1(356);
    Derive obj2(356, 789);
    return 0;
} // g++ 1.cpp -o 1.out -std=c++11

```

- 如果基类的某个构造函数被声明为私有成员函数，则不能在派生类中声明继承该构造函数。

- 如果派生类使用了继承构造函数，编译器就不会再为派生类生成隐式定义的默认构造函数。

成员访问权限

- 基类中的私有成员，**不允许在派生类成员函数中访问**，也不允许派生类的对象访问它们。
 - 真正体现“基类私有”，对派生类也不开放其权限！
- 基类中的公有成员：
 - 允许在派生类成员函数中被访问
 - 若是使用**public**继承方式，则成为派生类公有成员，可以被派生类的对象访问；
 - 若是使用**private/protected**继承方式，则成为派生类私有/保护成员，不能被派生类的对象访问。若想某成员能被派生类的对象访问，可在派生类public部分用关键字**using**声明它的名字。
- 基类中的保护成员
 - 保护成员**允许在派生类成员函数**中被访问，但不能被外部函数访问。

```
#include <iostream>
using namespace std;

class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive1: public Base {}; // D1类的继承方式是public继承

int main() {
    Derive1 obj1;
    cout << "calling obj1.baseFunc()..." << endl;
    obj1.baseFunc(); // 基类接口成为派生类接口的一部分，派生类对象可调用

    return 0;
}
```

```
#include <iostream>
using namespace std;
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive2: private Base
{/// 私有继承， is-implementing-in-terms-of: 用基类接口实现派生类功能
public:
    void deriveFunc() {
        cout << "in Derive2::deriveFunc(), calling Base::baseFunc()..." << endl;
        baseFunc(); /// 私有继承时，基类接口在派生类成员函数中可以使用
    }
};

int main() {
```

```

Derive2 obj2;
cout << "calling obj2.deriveFunc()..." << endl;
obj2.deriveFunc();
//obj2.baseFunc(); ERROR: 基类接口不允许从派生类对象调用

return 0;
}

```

```

#include <iostream>
using namespace std;
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive3: private Base { // B的私有继承
public:
    /// 私有继承时, 在派生类public部分声明基类成员名字
    using Base::baseFunc;
};

int main() {
    Derive3 obj3;
    cout << "calling obj3.baseFunc()..." << endl;
    obj3.baseFunc(); //基类接口在派生类public部分声明, 则派生类对象可调用

    return 0;
}

```

- 基类中的私有, 保护成员访问

```

#include <iostream>
using namespace std;

class Base{
private:
    int a{0};
protected:
    int b{0};
};

class Derive : private Base{
public:
    void getA(){cout<<a<<endl;} ///编译错误, 不可访问基类中私有成员
    void getB(){cout<<b<<endl;} ///可以访问基类中保护成员
};

int main()
{

```

```

    Derive d;
    d.getB();
    //cout<<d.b; ///编译错误，派生类对象不可访问基类中保护成员
    return 0;
}

```

```

#include <iostream>
using namespace std;
class Base {
    private:
        int data{0};
    public:
        int getData(){ return data;}
        void setData(int i){ data=i;} };
class Derive1 : private Base {
    public:
        using Base::getData; };
int main() {
    Derive1 d1;
    cout<<d1.getData();
    //d1.setData(10);    ///隐藏了基类的setData函数，不可访问
    //Base& b = d1;      ///不允许私有继承的向上转换
    //b.setData(10);     ///否则可以绕过D1，调用基类的setData函数
    return 0; }

```

基类成员访问权限与三种继承方式

- public继承
 - 基类的公有成员，保护成员，私有成员作为派生类的成员时，都保持原有的状态。
- private继承
 - 基类的公有成员，保护成员，私有成员作为派生类的成员时，都作为私有成员。
- protected继承
 - 基类的公有成员，保护成员作为派生类的成员时，都成为保护成员，基类的私有成员仍然是私有的。

成员访问权限

继承表		继承方法					
		public		private		protected	
基类中 成员类型	public	OK	pub/yes	OK	prv/no	OK	pro/no
	private	NO	prv/no	NO	prv/no	NO	prv/no
	protected	OK	pro/no	OK	prv/no	OK	pro/no

派生类成员函数
能否访问基类成员

基类成员在派生类中的成员类型，
派生类对象能否访问基类成员

prv: private
pro: protected
pub: public

类似集合交运算(成员类型与继承类型之间取交)
Order: public \supset protected \supset private

组合与继承

- 组合与继承的优点：支持增量开发。
 - 允许引入新代码而不影响已有代码正确性。
- 相似：
 - 实现代码重用。
 - 将子对象引入新类。
 - 使用构造函数的初始化成员列表初始化。
- 不同：
 - 组合：
 - 嵌入一个对象以实现新类的功能。
 - has-a 关系。
 - 继承：
 - 沿用已存在的类提供的接口。
 - public 继承：is-a。
 - private 继承：is-implementing-in-terms-of。

重写隐藏与重载

- 重载(overload)：目的：提供同名函数的不同实现，属于静态多态。函数名必须相同，函数参数必须不同，作用域相同（如位于同一个类中；或同名全局函数）。
- 重写隐藏(redefining)：

- 目的：在派生类中重新定义基类函数，实现派生类的特殊功能。
- 屏蔽了基类的所有其它同名函数。
- 函数名必须相同，函数参数可以不同

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
    void f() { cout << "Base::f()\n"; }
    void f(int i) { cout << "Base::f(" << i << ")\n"; }
    void f(double d) { cout << "Base::f(" << d << ")\n"; }
    void f(T) { cout << "Base::f(T)\n"; }
};
class Derive : public Base {
public:
    using Base::f;
    void f(int i) { cout << "Derive::f(" << i << ")\n"; }
};
int main() {
    Derive d;
    d.f(10);
    d.f(4.9);
    d.f();
    d.f(T());
    return 0;
}
/*
Derive::f(10)
B::f(4.9)
B::f()
B::f(T)
*/
```

using关键字

多重继承

- 数据存储
 - 如果派生类D继承的两个基类A,B，是同一基类Base的不同继承，则A,B中继承自Base的数据成员会在D有两份独立的副本，可能带来数据冗余。
- 二义性
 - 如果派生类D继承的两个基类A,B，有同名成员a，则访问D中a时，编译器无法判断要访问的哪一个基类成员。

```
#include <iostream>
using namespace std;

class Base {
```

```

public:
    int a{0};
};

class MiddleA : public Base {
public:
    void addA() { cout << "a=" << ++a << endl; };
    void bar() { cout << "A::bar" << endl; };
};

class MiddleB : public Base {
public:
    void addB() { cout << "a=" << ++a << endl; };
    void bar() { cout << "B::bar" << endl; };
};

class Derive : public MiddleA, public MiddleB{
};

int main() {
    Derive d;
    d.addA();          /// 输出 a=1。
    d.addB();          /// 仍然输出 a=1。
    d.addB();          /// 输出 a=2。
    //cout << d.a;      /// 编译错误, MiddleA和MiddleB都有成员a
    cout << d.MiddleA::a << endl;    /// 输出A中的成员a的值 1
    //d.bar();          /// 编译错误, MiddleA和MiddleB都有成员函数bar
    cout << d.MiddleB::a << endl;
    /// 输出B中的成员a的值 2
    return 0;
}

```

课后练习:

一家工厂生产飞机、汽车和摩托车。

一架飞机需要三个轮子, 和两个机翼; 一辆汽车需要四个轮子; 一辆摩托车需要两个轮子。

这些交通工具都具有一个run 函数, 其中汽车和摩托车调用时输出 “I am running”, 但是飞机调用时输出 “I am running and flying”。

编写以下几个类: Plane, Motor, Car, Wing, Wheel, Vehicle(交通工具), 设计合理的继承、组合关系以及使用合理使用函数的继承与重写实现add_wing, add_wheel, finished 以及 run 函数。

测试代码见下页: