第7课 类和对象(多态深入)

内容概述

1. “is-a”关系
2. “has-a”关系
3. 依赖倒置原则
4. 虚函数表
5. 运行时类型识别
6. 例子:链表
7. 例子:shared_ptr与容器
8. 构造函数中调用虚函数
9. 析构函数中调用虚函数
10. 成员函数中调用虚函数
11. 虚函数中的默认参数
12. 类设计的一些建议

public继承：“is-a”关系

public继承：意味着“is-a”关系，“是一种，是一个”关系。（设计的时候要牢记这一点）
class A : public B，这样的写法表明：每一个类型为A对象，同时也是一个类型为B的对象，反之不成立（每一个类型为B的对象，并不是一个类型为A的对象!）
意思是：B对象可以派上用处的地方，A对象一样可以派上用处。反之不行！
比如：class Superman : public Person 表明：每一个超人都是一个人，但：不是每个人都是超人！人能做的事情，叫超人去做肯定没问题。反之不行。

“is-a”：适用于基类身上的每一件事情一定适用于派生类身上，因为每一个派生类对象也都是基类对象。

```
#include <iostream>
#include <string>
using namespace std;
class Person {
public:
    void eat() { cout << "吃饭" << endl; }
    void sleep() { cout << "睡觉" << endl; }
};
class Teacher :public Person { //is-a
public:
    void teach() { cout << "教书" << endl; }
};
class Student :private Person { //not is-a
public:
    void study() { cout << "学习" << endl; }
};
```

```
int main() {
    //Teacher public继承自Person, Teacher is a Person
    Teacher t; //Teacher对象可以做Person的任何事情
    t.eat();
    t.sleep();
    Person *pt = &t; //is-a 关系 才能这样赋值
    Person p;  //Person对象不能做Teacher的事情
    //p.teach();//错,无法调用

    //Student private继承自Person,不存在is-a的关系
    Student s; //Student对象不能做Person的事情
    //s.eat();  //错,无法调用
    //s.sleep();//错,无法调用
    //Person *ps = &s; //错，不是is-a关系不能这样赋值
    return 0;
}
```

public继承：“is-a”关系

例：鸟都可以飞，企鹅是鸟，但是企鹅不会飞，这看似违反了is-a关系，其实，这个和语言表达方式有关，当我们说鸟会飞的时候，意思是表达一部分鸟会飞，而不是说所有的鸟都会飞，因此，在设计继承关系是就要注意这点。

```
#include <iostream>
using namespace std;
class Bird {
public:
    void eat() { cout << "鸟吃东西" << endl; }
    void fly() { cout << "鸟飞行" << endl; }
};
class Qie :public Bird { //is-a
public:
    //somecode...
};
int main() {
    Qie q;
    //企鹅不会飞，这里却可以调用，设计不合理
    q.fly();
    return 0;
}
```

```
class Bird {
public:
    void eat() { cout << "鸟吃东西" << endl; }
};
class FlyingBird :public Bird {
public:
    void fly() { cout << "鸟飞行" << endl; }
};
class Qie :public Bird { //is-a
public:
    //somecode...
};
int main() {
    Qie q;
    //企鹅不会飞，这里也不能调用，ok
    //q.fly(); //没有fly函数
    //其他会飞的鸟 可以 public继承 FlyingBird
    return 0;
}
```

复合：“has-a”和“根据某物实现出”

复合：意味着has-a（有一个）或者is-implemented-in-terms-of（根据某物实现出）。

人：有一个名字，有一个地址，有一个电话号码… 是 has-a 关系
不能说：人是一个名字，人是一个地址… 不是 is-a 关系

计算机：有一个cpu，有一个内存，有一个硬盘… 是 has-a关系

```
class Address{ /*...somecode...*/ };
class PhoneNo{ /*...somecode...*/ };
class Person {
public:
    //...somecode...
private:
    std::string name;
    Address address;
    PhoneNo phoneno;
};
```

```
class Cpu { /*...somecode...*/ };
class Mem { /*...somecode...*/ };
class HDisk { /*...somecode...*/ };
class Computer {
public:
    //...somecode...
private:
    Cpu cpu;
    Mem mem;
    HDisk hdisk;
};
```

复合：“has-a”和“根据某物实现出”

前面的企鹅继承Bird类的例子，假如一定要用带fly()接口的Bird类，那么可以用“根据某物实现出”这样的关系来做：

```
#include <iostream>
using namespace std;
class Bird {
public:
    void eat() { cout << "鸟吃东西" << endl; }
    void fly() { cout << "鸟飞行" << endl; }
};
class Qie { //根据Bird实现出
public:
    void eat() { bird.eat(); }
private:
    Bird bird;
};
int main() {
    Qie q;
    q.eat();
    //q.fly(); //不行
    return 0;
}
```

复合：“has-a”和“根据某物实现出”

做一个简单的set(集合,无重复元素,元素是string)类，想通过list链表类来实现，用“根据某物实现出”关系来做：

```
class List {        //List代码只是模拟
public:
    void push_back(const string& item){}     //插入链表
    bool find_item(const string& item)const{//查找链表
        return true; }
    void remove(const string& item){}     //删除链表元素
    int size()const {return 1;}           //求链表元素个数
    //其他接口函数
private:
    //...
};
class Set {  //根据List实现出Set
public:
    bool is_member(const string& item)const;//item是否在集合中
    void insert(const string& item);    //将item插入集合
    void remove(const string& item);    //将item从集合中删除
    int size()const;                    //求集合中元素个数
private:
    List LS;
};
```

```
bool Set::is_member(const string& item)const{
    return LS.find_item(item);
}
void Set::insert(const string& item) {
    if(is_member(item))
        LS.push_back(item);
}
void Set::remove(const string& item) {
    LS.remove(item);
}
int Set::size()const {
    return LS.size();
}
```

面向对象设计原则：依赖倒置原则

例子：
妈妈讲故事：

当需求更改：需要讲的内容从Book扩充到Newspaper 或者其他时，Mother类必须要跟着修改。

“耦合度”很高。

思考：如何降低“耦合度”

```
#include <iostream>
#include <string>
using namespace std;
class Book {
public:
    string getContent()const {
        return "很久很久以前，有一个...";
    }
};
class Mother {
public:
    void tellStory(const Book& b) {
        cout << "妈妈开始讲故事: "
            << b.getContent() << endl;
    }
};
int main() {
    Book book;
    Mother m;
    m.tellStory(book);
    return 0;
}
```

```
妈妈开始讲故事: 很久很久以前，有一个...
请按任意键继续. . .
```

```
class Newspaper {
public:
    string getContent()const {
        return "昨日凌晨，台风...";
    }
};
class Mother {
public:
    void tellStory(const Book& b) {
        cout << "妈妈开始讲故事: "
            << b.getContent() << endl;
    }
    void tellStory(const Newspaper& b) {
        cout << "妈妈开始讲故事: "
            << b.getContent() << endl;
    }
};
int main() {
    Book book;
    Mother m;
    m.tellStory(book);
    Newspaper paper;
    m.tellStory(paper);
    return 0;
}
```

```
妈妈开始讲故事: 很久很久以前，有一个...
妈妈开始讲故事: 昨日凌晨，台风...
请按任意键继续. . .
```

面向对象设计原则：依赖倒置原则

Mother类不再依赖于Book、Newspaper等底层模块，而是依赖于抽象接口IReader。
Book、Newspaper也依赖于抽象接口Ireader。

依赖倒置原则的核心：面向接口编程

```
class IReader {
public:
    //纯虚函数，接口
    virtual string getContent()const = 0;
};
class Book:public IReader {
public:
    virtual string getContent()const override{
        return "很久很久以前，有一个...";
    }
};
class Newspaper:public IReader {
public:
    virtual string getContent()const override{
        return "昨日凌晨，台风...";
    }
};
```

```
class Mother {
public:
    void tellStory(const IReader& b) {
        cout << "妈妈开始讲故事: "
            << b.getContent() << endl;
    }
};
int main() {
    Book book;
    Mother m;
    m.tellStory(book);
    Newspaper paper;
    m.tellStory(paper);
    return 0;
}
```

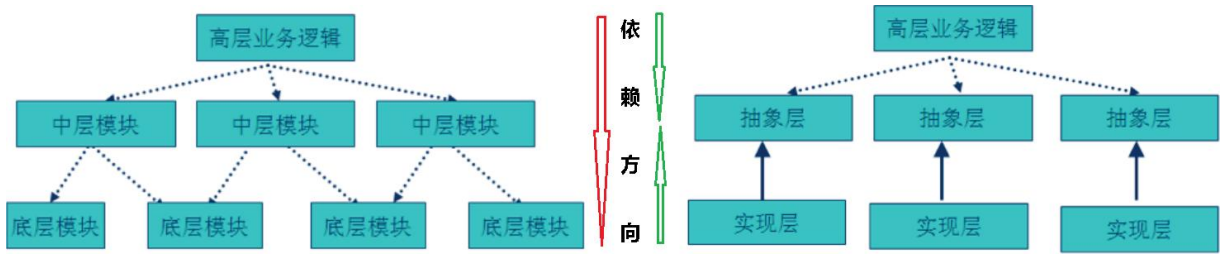
面向对象设计原则：依赖倒置原则

依赖倒置原则（基于多态）：

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。抽象指的是抽象基类（通过纯虚函数），细节就是具体的实现类，使用抽象基类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的核心：面向接口编程



前面的例中，采用依赖倒置原则后无论以后怎样扩展底层类(Book、Newspaper)，都不要再修改Mother类了。
代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。
所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如前例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。
修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。
参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。

面向对象设计原则：依赖倒置原则

```
class Cpu { //Cpu接口(抽象基类)
public:
    virtual void work() = 0;
    virtual ~Cpu(){}
};
class Mem { //Mem接口
public:
    virtual void work() = 0;
    virtual ~Mem() {}
};
class HDisk { //HDisk接口
public:
    virtual void work() = 0;
    virtual ~HDisk() {}
};
class IntelCpu :public Cpu {
public:
    virtual void work() override { cout << "Intel Cpu working..." << endl;}
};
class AmdCpu :public Cpu {
public:
    virtual void work() override { cout << "AMD Cpu working..." << endl;}
};
class KSTMem :public Mem {
public:
    virtual void work() override { cout << "KST Mem working..." << endl;}
};
class SamsungMem :public Mem {
public:
    virtual void work() override { cout << "SamSung Mem working..." << endl;}
};
class WDHDisk :public HDisk {
public:
    virtual void work() override { cout << "WD HDisk working..." << endl; }
};
```

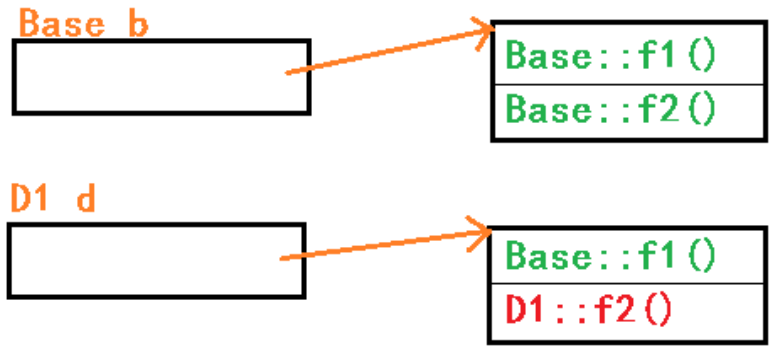
```
class Computer {
public:
    Computer(Cpu *pcpu,Mem *pmem,HDisk *phdisk)
    :cpu(pcpu),mem(pmem),hdisk(phdisk){}
    void work() {
        cpu->work();
        mem->work();
        hdisk->work();
    }
private:
    Cpu *cpu;
    Mem *mem;
    HDisk *hdisk;
};
int main() {
    IntelCpu intel;
    KSTMem kst;
    WDHDisk wd;
    Computer pc(&intel,&kst,&wd);
    pc.work();
    return 0;
}
```

```
Intel Cpu working...
KST Mem working...
WD HDisk working...
请按任意键继续. . .
```

Computer类依赖于Cpu,Mem,HDisk接口以后，新加各种品牌的Cpu,Mem,HDisk都不会影响Computer类，不需要修改。

虚函数表

```cpp
#include <iostream>
using namespace std;
class Base {
public:
    void func() { cout << "Base::func()\n"; }
    virtual void f1() { cout << "Base::f1()\n"; }
    virtual void f2() { cout << "Base::f2()\n"; }
};
class D1 :public Base {
public:
    void func1() { cout << "D1::func1()\n"; }
    virtual void f2()override { cout << "D1::f2()\n"; }
};
```



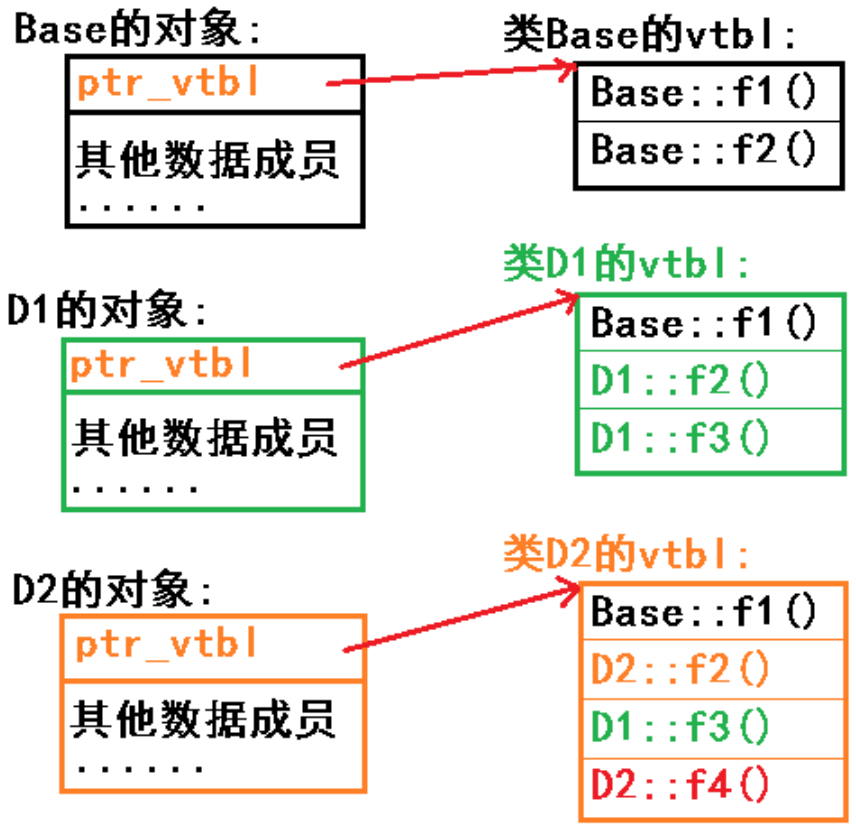

```
4--4
Base::f1()
Base::f2()
----------
Base::f1()
D1::f2()
请按任意键继
```

```cpp
int main() {
    //class生成对象的size发生变化,32bit下是4
    //(没有数据成员时：以前是1,说明有虚函数时多了一些东西)
    cout << sizeof(Base) << "--" << sizeof(D1) << endl;
    Base b;
    int *p_vtbl = reinterpret_cast<int*>(&b); //重解释
    int *vtbl = reinterpret_cast<int*>(*p_vtbl);
    typedef void(*PFUN)();
    PFUN pf1 = reinterpret_cast<PFUN>(vtbl[0]);
    pf1();
    PFUN pf2 = reinterpret_cast<PFUN>(vtbl[1]);
    pf2();
    cout << "----------" << endl;
    D1 d;
    p_vtbl = reinterpret_cast<int*>(&d); //重解释
    vtbl = reinterpret_cast<int*>(*p_vtbl);
    pf1 = reinterpret_cast<PFUN>(vtbl[0]);
    pf1();
    pf2 = reinterpret_cast<PFUN>(vtbl[1]);
    pf2();
    return 0;
}
```

虚函数表

通过基类的指针或引用操作派生类对象，在调用虚函数时，执行的代码必须是其真实类型中的函数。
C++通过虚函数表(Virtual Table)来实现。表中存放的是指向虚函数地址的函数指针。

```
class Base {
public:
    void func();
    virtual void f1();
    virtual void f2();
};

class D1 :public Base {
public:
    void func2();
    virtual void f2()override;
    virtual void f3();
};

class D2 :public D1 {
public:
    void func3();
    virtual void f2()override;
    virtual void f4();
};
```



1. 有虚函数的类，都有一个虚函数表vtbl，该类所有的实例共享该vtbl。

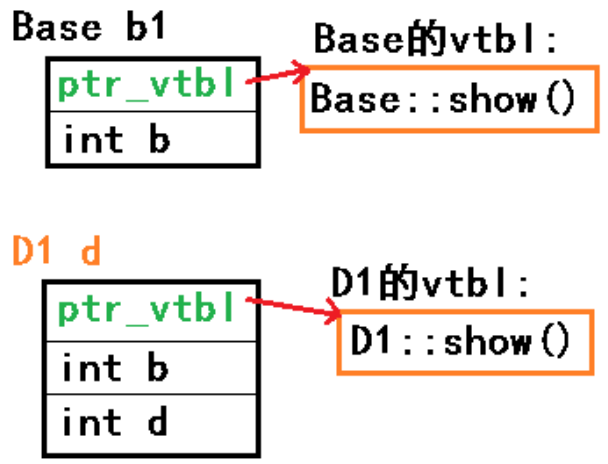
2. 有虚函数的类的实例对象中：
有一个隐藏的数据成员(ptr_vtbl)，用来指向该类的虚函数表(vtbl)

3. 虚函数不能inline

虚函数表

```
class Base {
public:
    Base() {
        cout << "Base构造\n";
        memset(this, 0, sizeof(*this));
    }   //构造时,强制将对象的所有数据置为0
    void func() {cout<<"Base::func()\n";}
    virtual void show()const {
        cout<<"Base::show()\n";
    }

private:
    int b;
};
class D1 :public Base {
public:
    D1() { cout << "D1构造\n";   }
    virtual void show()const override {
        cout << "D1::show()\n";
    }
private:
    int d;
};
```

```
int main() {
    Base b1;
    b1.func(); //非虚函数调用,ok
    b1.show(); //虚函数调用,ok
    //上面虚函数调用ok,是因为通过对象调用虚函数,
    //是在编译时确定的,没有通过虚函数表
    Base *pb1 = &b1;
    pb1->func(); //非虚函数调用,ok
    //pb1->show(); //崩溃,指向虚函数表的指针被破坏了.
    //通过指针或引用调用虚函数,会使用虚函数表,运行时确定。
    cout << "------------\n";
    D1 d;
    Base &rd = d;
    rd.func(); //ok,非虚函数调用
    rd.show();
    //ok,D1构造时,重建了指向虚函数表的指针
    //(因为Base和D1的虚函数表不同)
    return 0;
}
```



虚函数表

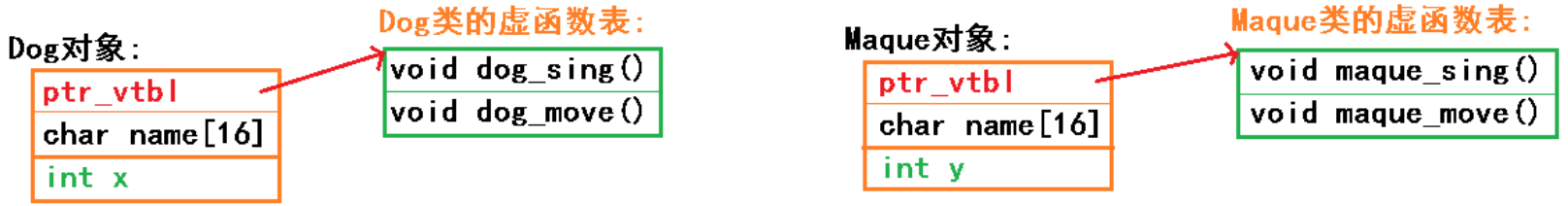
```
//不同的动物：不同的叫声,不同的行动方式
typedef void(*SING)();
typedef void(*MOVE)();
struct vtbl { //模拟虚函数表
    SING sing; //函数指针
    MOVE move;
};

//基类：Animal
struct Animal {
    //指向虚表的指针
    const struct vtbl *ptr_vtbl;
    //动物的名称：(公有属性)
    char name[16];
};

int main() {
    struct Animal *p = NULL;
    //基类指针 指向派生类对象
    p = create_dog("小狗");
    p->ptr_vtbl->sing();
    p->ptr_vtbl->move();
    free(p); printf("-------\n");
    p = create_maque("麻雀");
    p->ptr_vtbl->sing();
    p->ptr_vtbl->move();
    free(p);
    return 0;
}
```

```
Dog:汪汪汪
Dog:地面狂奔
-------
Maque:唧唧唧
Maque:低空任我飞
请按任意键继续. .
```

C语言模拟多态

```
//派生类：Dog
struct Dog {
    struct Animal base;
    int x; //Dog私有的属性
};
//Dog的sing和move函数
void dog_sing() { printf("Dog:汪汪汪\n"); }
void dog_move() { printf("Dog:地面狂奔\n"); }
//Dog类的虚函数表vtbl
const struct vtbl dog_vtbl={dog_sing,dog_move};
//创建Dog对象
struct Animal *create_dog(const char* name) {
    assert(name);
    struct Dog *p_dog =
        (struct Dog*)malloc(sizeof(struct Dog));
    assert(p_dog);
    p_dog->base.ptr_vtbl = &dog_vtbl;
    strcpy(p_dog->base.name, name);
    p_dog->x = 10;
    //上面：将Dog对象的虚函数表绑定到 虚表指针
    return (struct Animal*)p_dog;
}
```

```
//派生类：Maque (麻雀)
struct Maque {
    struct Animal base;
    int y; //Maque的私有属性
};
//Maque的sing和move函数
void maque_sing() { printf("Maque:唧唧唧\n"); }
void maque_move() { printf("Maque:低空任我飞\n"); }
//Maque的虚函数表vtbl
const struct vtbl maque_vtbl={maque_sing,maque_move};
//创建Maque对象
struct Animal *create_maque(const char* name) {
    assert(name);
    struct Maque *p_maque =
        (struct Maque*)malloc(sizeof(struct Maque));
    assert(p_maque);
    p_maque->base.ptr_vtbl = &maque_vtbl;
    strcpy(p_maque->base.name, name);
    p_maque->y = 20;
    return (struct Animal*)p_maque;
}
```

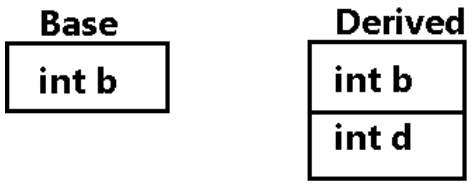


运行时类型识别：RTTI

通过RTTI，能够通过基类的指针或引用来检索其所指对象的实际类型。c++通过下面两个操作符提供RTTI。
（1）typeid：返回指针或引用所指对象的实际类型。
（2）dynamic_cast：将基类类型的指针或引用安全的转换为派生类型的指针或引用。
注意：基类至少有一个虚函数(虚析构也算)

如果dynamic_cast转换指针类型失败，则返回0；如果转换引用类型失败，则抛出一个bad_cast类型的异常。

```
class Base {
public:    //基类至少有一个虚函数
    virtual ~Base() = default;
private:
    int b;
};
class Derived :public Base {
private:
    int d;
};
```




```
int main() {
    Derived d1;
    Base *pd1 = &d1;
    //将基类指针指向的对象，转为派生类指针指向的对象
    Derived *p1 = dynamic_cast<Derived*>(pd1); //成功
    if (p1 != nullptr)
        cout << "转换成功" << endl;
    else
        cout << "转换失败" << endl;
    Base b1;
    Base *pb1 = &b1;
    Derived *p2 = dynamic_cast<Derived*>(pb1); //失败
    if (p2 != nullptr)
        cout << "转换成功" << endl;
    else
        cout << "转换失败" << endl;
    return 0;
}
```

```
转换成功
转换失败
请按任意键继续
```

运行时类型识别：RTTI

typeid能够获取一个表达式的类型：typeid(e)。
如果操作数不是类类型或者是没有虚函数的类，则获取其静态类型；
如果操作数是定义了虚函数的类类型，则计算运行时类型。

type_info类:

typeid操作符的返回类型就是type_info；type_info的默认构造函数、拷贝构造函数、赋值操作符都定义为private，创建type_info对象的唯一方法就是使用typeid操作符。
name()函数返回类型名字的c-style字符串，但字符串的格式可能不同的编译器略有不同。

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base {
public:
    virtual ~Base() = default;
private:
    int b = 1;
};
class Derived :public Base {
private:
    int d1 = 2;
};
```

```
int main() {
    cout << typeid(int).name() << endl;          //int
    cout << typeid(double).name() << endl;       //double
    cout << typeid(char *).name() << endl;       //char *
    //顶层const不识别
    cout << typeid(char * const).name() << endl;//char *
    //底层const识别
    cout << typeid(const char *).name() << endl;//char const *
    cout << typeid(Base).name() << endl;     //class Base
    cout << typeid(Derived).name() << endl; //class Derived

    Derived d;
    Base *pb = &d;
    cout << typeid(pb).name() << endl; //class Base * 指针看不出真实类型
    cout << typeid(*pb).name() << endl; //class Derived
    cout << typeid(d).name() << endl; //class Derived
    Base &rb = d;
    cout << typeid(rb).name() << endl; //class Derived
```

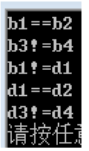
```
int
double
char *
char *
char const *
class Base
class Derived
class Base *
class Derived
class Derived
class Derived
类型一致
是Derived类对象
请按任意键继续.
```

```
    Base *pb1 = &d;
    Derived *pd1 = &d;
    //1.判断两个对象的真实类型是否一致
    if (typeid(*pb1) == typeid(*pd1))
        cout << "类型一致" << endl;//注意 加*
    else
        cout << "类型不一致" << endl;
    //2.判断某对象的真实类型是否某种类类型
    if(typeid(*pb1)==typeid(Derived))
        cout << "是Derived类对象" << endl;
    else
        cout << "不是Derived类对象" << endl;
    return 0;
}
```

运行时类型识别：RTTI

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base {
public:
    Base(int b1 = 0) :b(b1) {}
    virtual ~Base() = default;
    virtual bool equal(const Base& rhs)const {
        return b == rhs.b;
    }
private:
    int b;
};
class Derived :public Base {
public:
    Derived(int b1=0, int d1=0) :Base(b1), d(d1) {}
    //注意：这里的参数只能写 Base，否则就不是 override
    virtual bool equal(const Base& rhs)const override {
        //注意下面 const的用法
        const Derived *pd = dynamic_cast<const Derived*>(&rhs);
        if (pd)
            return Base::equal(rhs) && d == pd->d;
        return false;
    }
private:
    int d;
};
```

```
bool operator==(const Base& lhs, const Base& rhs) {
    return typeid(lhs)==typeid(rhs) && lhs.equal(rhs);
}
bool operator!=(const Base& lhs, const Base& rhs) {
    return !(lhs == rhs);
}
int main() {
    Base b1(1), b2(1);
    if (b1 == b2) cout << "b1==b2" << endl; //b1==b2
    Base b3(1), b4(2);
    if (b3 != b4) cout << "b3!=b4" << endl; //b3!=b4
    Derived d1(1, 2), d2(1, 2);
    if (b1 != d1) cout << "b1!=d1" << endl; //b1!=d1
    if (d1 == d2) cout << "d1==d2" << endl; //d1==d2
    Derived d3(1, 2), d4(1, 4);
    if (d3 != d4) cout << "d3!=d4" << endl; //d3!=d4
    return 0;
}
```



例子：链表1

实现一个链表，可存放多种类型的数据。
实现接口：尾插，遍历打印，拷贝，排序 。

1. List中的结点Node，以前存放的是具体的类型数据，现在改为存放指针；(什么类型的指针呢？)
2. 定义一个基类Obj，让需要存放的数据都从Obj派生，这样结点Node中就是存放 Obj *类型的指针;

链表结点：

```
class Node {
      ......
private:
    Obj *p_data;
    Node *next;
};
```

链表：

```
class List {
      ......
private:
    Node *head;
    Node *tail;
    int size;
};
```

Obj基类：

```
class Obj {
      ......
public:
 virtual ~Obj();
},
```

int类型的数据：

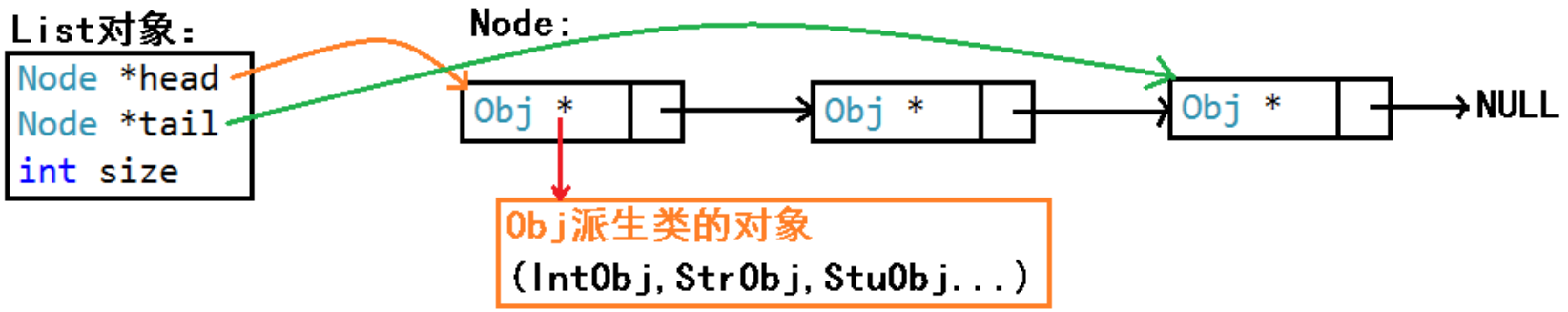
```
class IntObj :public Obj{
      ......
private:
    int data;
};
```

字符串类型的数据：

```
class StrObj :public Obj {
      ......
private:
    char *data;
};
```

学生类数据：

```
class StuObj :public Obj {
      ......
private:
    int id;
    string name;
};
```



例子：链表2

```
class Obj { //基类
public:
    virtual ~Obj() = default;
    virtual void print()const = 0;
};
class IntObj :public Obj{ //派生类:int数据
public:
    IntObj(int i=0):data(i){}
    ~IntObj(){}
    virtual void print()const override{
        cout << data << " -> ";
    }
private:
    int data;
};
class StrObj :public Obj { //派生类:str数据
public:
    StrObj(const char *str=nullptr){
        if (!str)
            data = new char[1]{ '\0' };
        else {
            data = new char[strlen(str) + 1];
            strcpy(data, str);
        }
    }
    ~StrObj() { delete data; }
    virtual void print()const override {
        cout << "\""<<data << "\" -> ";
    }
private:
    char *data;
};
```

```
class StuObj :public Obj { //派生类:学生数据
public:
    StuObj(int _id,const string &_name)
        :id(_id),name(_name){}
    ~StuObj(){}
    virtual void print()const override {
        cout << id << ":" << name << " -> ";
    }
private:
    int id;
    string name;
};
```

```
class List;
class Node {
    friend List;
public:
    Node(Obj *p_obj=nullptr)
        :p_data(p_obj),next(nullptr){}
    ~Node() { delete p_data; }
private:
    Obj *p_data;
    Node *next;
};
```

结点Node

```
int main() {
    List L;
    L.push_back(new IntObj(10));
    L.push_back(new StuObj(1001, "张三"));
    L.push_back(new StrObj("abc"));
    L.print_list();
    return 0;
}
```

测试：

```
10 -> 1001:张三 -> "abc" -> NULL
请按任意键继续. . .
```

```
class List {
public:
    List():head(nullptr),tail(nullptr),size(0){}
    ~List(){
        Node *p = head;
        while (p) {
            Node *tmp = p->next;
            delete p;
            p = tmp;
        }
        head = tail = nullptr;
    }
    void push_back(Obj *p_obj) {
        assert(p_obj);
        Node *p_new_node = new Node(p_obj);
        if (!head)  head = p_new_node;
        else  tail->next = p_new_node;
        tail = p_new_node;
        size++;
    }
    void print_list()const {
        Node *p = head;
        while (p) {
            p->p_data->print(); //多态
            p = p->next;
        }
        cout << "NULL\n";
    }
private:
    Node *head;
    Node *tail;
    int size;
};
```

链表List

注意List资源的释放

虚析构的重要性

例子：链表3

List链表的拷贝，利用多态，虚clone。

```
class List {
public:
List(const List &other):List() {
    //思路：遍历other链表,将每个元素复制一份push_back
    //但是元素是以Obj*存放的,并不知道其真实类型是啥
    //尝试这样写：
    Node * p = other.head;
    Obj * p_new_obj = nullptr;
    while (p) { //遍历other链表
        if (typeid(*(p->p_data)) == typeid(IntObj)) {
            //先将Node结点中的Obj*转为真实类型的IntObj*
            IntObj * p_int = dynamic_cast<IntObj*>(p->p_data);
            p_new_obj = new IntObj(*p_int);
        }
        else if(typeid(*(p->p_data)) == typeid(StrObj)){
            //先将Node结点中的Obj*转为真实类型的StrObj*
            StrObj * p_int = dynamic_cast<StrObj*>(p->p_data);
            p_new_obj = new StrObj(*p_int);
        }
        else if (typeid(*(p->p_data)) == typeid(StuObj)) {
            //先将Node结点中的Obj*转为真实类型的StuObj*
            StuObj * p_int = dynamic_cast<StuObj*>(p->p_data);
            p_new_obj = new StuObj(*p_int);
        }
        else {
            cout << "没有找到匹配的类型\n";
            throw other;
        }
        push_back(p_new_obj); //push_back
        p = p->next;
    }
    size = other.size;
}
```

```
class Obj { //基类
public:
    virtual Obj *clone()const = 0; //虚clone
};
class IntObj :public Obj{ //派生类:int数据
public:
    virtual IntObj *clone()const override{
        return new IntObj(*this);
    }
};
class StrObj :public Obj { //派生类:str数据
public:
    StrObj(const StrObj& other) :Obj(other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data); //深拷贝
    }
    virtual StrObj *clone()const override {
        return new StrObj(*this);
    }
};
class StuObj :public Obj { //派生类:学生数据
public:
    virtual StuObj *clone()const override {
        return new StuObj(*this);
    }
};
```

```
class List {
public:
List(const List &other) :List() {
    //遍历other链表,将每个元素复制一份push_back
    //新思路：虚clone（利用多态）
    Node * p = other.head;
    while (p) { //遍历other链表
        push_back(p->p_data->clone());
        p = p->next;
    }
    size = other.size;
}
int main() {
    List L;
    L.push_back(new IntObj(10));
    L.push_back(new StuObj(1001, "张三"));
    L.push_back(new StrObj("abc"));
    L.print_list();
    List L1(L);
    L1.print_list();
    return 0;
}
```

```
10 -> 1001:张三 -> "abc" -> NULL
10 -> 1001:张三 -> "abc" -> NULL
请按任意键继续. . .
```

左边的方法用了运行时类型识别，但是如有新类型加入，则必须改代码！（不好！）

右边的方法用了多态来实现拷贝，虚clone。

例子：链表4

链表排序：要排序就要比大小。
Obj派生出的IntObj、StrObj、StuObj如何比大小，当然比较法则可以由我们自己定。(全部转字符串比较)

尝试重载 <运算符：由于链表中存放的是 Obj* 类型的数据，所以参数写法是 Obj &类型。

```
bool operator<(const Obj &lhs, const Obj &rhs); //普通函数
bool XxxObj::operator<(const Obj &rhs);          //成员函数
```

但是，< 运算符比较，必须要进行真实类型的比较才行。**思考：如何实现？**

1. 虚函数 + 运行时类型识别
2. 虚函数
3. 模拟虚表

lhs	rhs
IntObj	IntObj
IntObj	StrObj
IntObj	StuObj
StrObj	IntObj
StrObj	StrObj
StrObj	StuObj
StuObj	IntObj
StuObj	StrObj
StuObj	StuObj

Obj基类：

```
class Obj {
      ......
public:
  virtual ~Obj();
},
```

int类型的数据：

```
class IntObj :public Obj{
      ......
private:
     int data;
};
```

字符串类型的数据：

```
class StrObj :public Obj {
      ......
private:
     char *data;
};
```

学生类数据：

```
class StuObj :public Obj {
      ......
private:
     int id;
     string name;
};
```

例子：链表5

```
class Obj { //基类  加一个 operator< 纯虚函数
};   virtual bool operator<(const Obj&)const = 0;
class IntObj :public Obj { //派生类:int数据
    int get_data()const { return data; }
    virtual bool operator<(const Obj& rhs)const override;
private:
};   int data;
```

class StrObj 和 StuObj同样实现operator<纯虚函数

```
bool IntObj::operator<(const Obj& rhs)const {
    const type_info& obj_type = typeid(rhs);//获取rhs的type_info
    if (obj_type == typeid(IntObj)) { //IntObj < IntObj
        const IntObj& rrhs = dynamic_cast<const IntObj&>(rhs);
        if (data < rrhs.data) return true;
    }   return false;
    else if (obj_type == typeid(StrObj)) { //IntObj < StrObj
        const StrObj& rrhs = dynamic_cast<const StrObj&>(rhs);
        string tmp = std::to_string(data);
        if (strcmp(tmp.c_str(), rrhs.c_str()) < 0) return true;
    }   return false;
    else if (obj_type == typeid(StuObj)) { //IntObj < StuObj
        const StuObj& rrhs = dynamic_cast<const StuObj&>(rhs);
        string tmp1 = std::to_string(data);
        string tmp2 = std::to_string(rrhs.get_id());
        tmp2 += rrhs.get_name();
        if (tmp1 < tmp2) return true;
    }   return false;
    else {
        cout << "Type error\n";  throw "TypeError";
    }
}
```

```
bool StrObj::operator<(const Obj& rhs)const;
bool StuObj::operator<(const Obj& rhs)const;
```

方法1. 虚函数 + 运行时类型识别

增加新类型的时候，需要修改老类型的代码！（不好）

```
class List {   加一个sort函数
    void sort() { //从大到小排序
        if (size <= 1) return;
        int i = 0, j = 0;
        Node *p = head, *q = nullptr;
        for (; i < size - 1; ++i, p = p->next) {
            for (j = i + 1, q = p->next; j<size; ++j, q = q->next){
                if ( *p->p_data < *q->p_data )
                    std::swap(p->p_data, q->p_data);
            }
        }
    }

int main() {
    List L;
    L.push_back(new IntObj(1));
    L.push_back(new IntObj(2));
    L.push_back(new StrObj("12a"));
    L.push_back(new IntObj(3));
    L.push_back(new StuObj(2001, "张三"));
    L.push_back(new IntObj(4));
    L.push_back(new StrObj("1001abc"));
    L.push_back(new StuObj(3002, "李四"));
    L.push_back(new IntObj(5));
    L.print_list();
    L.sort();
    L.print_list();
}   return 0;
```

这里operator< 都是Obj类型

```
1 -> 2 -> "12a" -> 3 -> 2001:张三 -> 4 -> "1001abc" -> 3002:李四 -> 5 -> NULL
5 -> 4 -> 3002:李四 -> 3 -> 2001:张三 -> 2 -> "12a" -> "1001abc" -> 1 -> NULL
请按任意键继续. . .
```

例子：链表6

```
class Obj { //基类  加4个 operator< 纯虚函数
      virtual bool operator<(const Obj&)const = 0;
      virtual bool operator<(const IntObj&)const = 0;
      virtual bool operator<(const StrObj&)const = 0;
};    virtual bool operator<(const StuObj&)const = 0;
```

IntObj, StrObj, StuObj 都必须实现这4个纯虚函数

IntObj的4个operator<函数实现：

```
bool IntObj::operator<(const Obj& rhs)const {
    return !(rhs < *this);
}
```

这里会调用谁？

```
bool IntObj::operator<(const IntObj& rhs)const {
    if (data < rhs.data) return true;
    return false;
}

bool IntObj::operator<(const StrObj& rhs)const {
    string tmp = std::to_string(data);
    if (strcmp(tmp.c_str(), rhs.c_str())<0)return true;
    return false;
}

bool IntObj::operator<(const StuObj& rhs)const {
    string tmp1 = std::to_string(data);
    string tmp2 = std::to_string(rhs.get_id());
    tmp2 += rhs.get_name();
    if (tmp1 < tmp2) return true;
    return false;
}
```

方法2：虚函数

在增加新类型的时候，Obj基类需要增加纯虚函数

其他所有类型的类都要增加这个纯虚函数的实现。

所以：也不好。

StrObj:

```
virtual bool operator<(const Obj& rhs)const override;
virtual bool operator<(const IntObj& rhs)const override;
virtual bool operator<(const StrObj& rhs)const override;
virtual bool operator<(const StuObj& rhs)const override;
```

StuObj:

```
virtual bool operator<(const Obj& rhs)const override;
virtual bool operator<(const IntObj& rhs)const override;
virtual bool operator<(const StrObj& rhs)const override;
virtual bool operator<(const StuObj& rhs)const override;
```

List的sort函数：

```
if ( *p->p_data < *q->p_data )
    std::swap(p->p_data, q->p_data);
```

```
1 -> 2 -> "12a" -> 3 -> 2001:张三 -> 4 -> "1001abc" -> 3002:李四 -> 5 -> NULL
5 -> 4 -> 3002:李四 -> 3 -> 2001:张三 -> 2 -> "12a" -> "1001abc" -> 1 -> NULL
请按任意键继续. . .
```

例子：链表7

```
bool Int_Int(const Obj &lhs, const Obj &rhs) {
    const IntObj &llhs = dynamic_cast<const IntObj&>(lhs);
    const IntObj &rrhs = dynamic_cast<const IntObj&>(rhs);
    if (llhs.get_data() < rrhs.get_data()) return true;
}   return false;
bool Int_Str(const Obj &lhs, const Obj &rhs) {
    const IntObj &llhs = dynamic_cast<const IntObj&>(lhs);
    const StrObj &rrhs = dynamic_cast<const StrObj&>(rhs);
    string tmp = std::to_string(llhs.get_data());
    if (strcmp(tmp.c_str(), rrhs.c_str()) < 0) return true;
}   return false;
bool Int_Stu(const Obj &lhs, const Obj &rhs) {
    const IntObj &llhs = dynamic_cast<const IntObj&>(lhs);
    const StuObj &rrhs = dynamic_cast<const StuObj&>(rhs);
    string tmp1 = std::to_string(llhs.get_data());
    string tmp2 = std::to_string(rrhs.get_id());
    tmp2 += rrhs.get_name();
    if (tmp1 < tmp2) return true;
}   return false;
```

9个比较函数，参数一样

```
bool Str_Int(const Obj &lhs, const Obj &rhs) {
}   return !(Int_Str(rhs, lhs)); //直接调用 Int_Str
bool Str_Str(const Obj &lhs, const Obj &rhs) {···}
bool Str_Stu(const Obj &lhs, const Obj &rhs) {···}

bool Stu_Int(const Obj &lhs, const Obj &rhs) {
}   return !(Int_Stu(rhs, lhs));//直接调用 Int_Stu
bool Stu_Str(const Obj &lhs, const Obj &rhs) {
}   return !(Str_Stu(rhs, lhs));//直接调用 Str_Stu
bool Stu_Stu(const Obj &lhs, const Obj &rhs) {···}
```

```
typedef bool(*LESS_FUNC_PTR)(const Obj &, const Obj &);
struct LessVecNode {
    LessVecNode() {}
    LessVecNode(const char* s1, const char* s2, LESS_FUNC_PTR p) :
        obj1(s1), obj2(s2), ptr(p) {}
    string obj1 = "";
    string obj2 = "";
    LESS_FUNC_PTR ptr = nullptr;
};
vector<LessVecNode> init_vec() { //初始化 函数表
    vector<LessVecNode> vec;
    vec.push_back(LessVecNode("class IntObj", "class IntObj", Int_Int));
    vec.push_back(LessVecNode("class IntObj", "class StrObj", Int_Str));
    ·····
    vec.push_back(LessVecNode("class StuObj", "class StuObj", Stu_Stu));
}   return vec;
LESS_FUNC_PTR lookup(const Obj &lhs, const Obj &rhs) {//查找 函数表
    static vector<LessVecNode> func_vec = init_vec();
    //可以用 map 来替代vector,效率高
    vector<LessVecNode>::iterator end = func_vec.end();
    vector<LessVecNode>::iterator it = func_vec.begin();
    for (; it != end; ++it) {
        if ((*it).obj1 == typeid(lhs).name() && (*it).obj2 == typeid(rhs).name())
    }       break;
    if (it != func_vec.end())
        return (*it).ptr;
}   return nullptr;
bool operator<(const Obj &lhs, const Obj &rhs) {
    LESS_FUNC_PTR f = lookup(lhs, rhs);
    if (f) return f(lhs, rhs);
}   throw "No Match Type!";
```

方法3：模拟虚函数表
采用普通函数(Obj类系中没有<)
先建立一张函数表，运用运行时类型识别来查表决定调用哪个函数。
新类型添加时，不需要修改类定义和比较函数。

例子：shared_ptr存入vector

shared_ptr与容器实现多态且保证安全性的一种方式：**基类析构函数设置为protected**。
多态：vector中存入的是类Base的指针，通过指针实现了多态行为；
安全性：类Base的析构函数设置为protected，这样就不能手动delete智能指针get()到的裸指针；

```
class Base {
public:
    virtual void show()const = 0;
protected:
    virtual ~Base() =default;
    //不能是private的析构,否则D1无法析构
};

class D1 : public Base {
public:
    D1(int i) :data(i) { }
    virtual void show()const override{
        cout << data << " ";
    }
private:
    int data;
};
```

```
shared_ptr<Base> createBase(int data) {
    shared_ptr<Base> p = make_shared<D1>(data);
    return p;
}
int main() {
    vector<shared_ptr<Base>> vec;
    for (int i = 0; i<10; ++i)
        vec.push_back(createBase(i));
    vector<shared_ptr<Base>>::iterator end = vec.end();
    for (auto it = vec.begin(); it != end; ++it)
        (*it)->show();
    cout << endl;
    Base* p = vec[0].get();
    //delete p; //错误,防止delete 裸指针
    //通过get获得的裸指针不能delete,因为Base的析构是protected
    //只有shared_ptr可以，unique_ptr不行。
}
```

构造函数中调用虚函数

基类构造函数执行时，派生类构造函数还没有执行；
意味着基类构造函数执行时，派生类部分数据成员还没有完成初始化，假如基类构造函数调用的虚函数是派生类中的，那么就可能会用到派生类中没有完成初始化的数据成员，这是危险的行为，所以C++禁止了这样的做法。

基类构造函数中调用虚函数只调用基类的虚函数实现
构造函数中调用虚函数会退化到调用本类的虚函数实现。

```
class Base {
public:
    Base() {
        this->f2();   //就是 f2();
        this->func(); //就是 func();
        //这里调用虚函数，只会调用本类的，而不会是派生类的
        cout<<"typeid: "<<typeid(*this).name()<<endl;//Base
    }
    virtual void func() { cout << "Base func()\n"; }
    void f2() { cout << "Base f2()\n"; }
    virtual ~Base() {}
private:
    int i;
};
```

```
class Derived :public Base {
public:
    Derived() :Base() {
        this->f2();
        this->func();
    }   cout<<"typeid: "<<typeid(*this).name()<<endl;//Derived
    virtual void func() override{ cout << "Derived func()\n";}
    void f2() { cout << "Derived f2()\n"; }
private:
    int j;
};
int main() {
    Derived d1;
    return 0;
}
```

```
Base f2()
Base func()
typeid: class Base
Derived f2()
Derived func()
typeid: class Derived
请按任意键继续. . .
```

析构函数中调用虚函数

基类析构函数执行时，派生类部分的数据成员已经销毁，假如调用派生类的虚函数，则可能会用到已销毁的数据成员，危险行为，所以C++禁止了这样的操作。

析构函数中调用虚函数会退化到调用本类的虚函数实现。　　(小问题：析构函数纯虚函数?)

建议：在构造函数和析构函数中不要调用虚函数，因为这类调用从不下降至派生类

```
class Base {
public:
    virtual ~Base() {
        f2();   //就是 this->f2();
        func(); //就是 this->func();
        //这里调用虚函数，只会调用本类的，而不会是派生类的
        cout << "typeid: "<<typeid(*this).name()<<endl;//Base
    }
    virtual void func() { cout << "Base func()" << endl; }
    void f2() { cout << "Base f2()" << endl; }
private:
    int i;
};
```

```
class Derived :public Base {
public:
    virtual ~Derived() {
        f2();
        func();
        cout<<"typeid: "<<typeid(*this).name()<<endl;//Derived
    }
    virtual void func()override {cout<<"Derived func()"<<endl;}
    void f2() { cout << "Derived f2()" << endl; }
private:
    int j;
};
int main() {
    Derived d1;
    return 0;
}
```

```
Derived f2()
Derived func()
typeid: class Derived
Base f2()
Base func()
typeid: class Base
请按任意键继续. . .
```

成员函数中调用虚函数

成员函数中调用虚函数/纯虚函数：调用真实类型的虚函数实现。

```
public:
    void f() {
        cout << "Base f()" << endl;
        cout <<"typeid: "<<typeid(*this).name()<<endl;  //Derived
        //可以做一些虚函数调用前的准备工作
        vf();
        //可以做一些虚函数调用后的善后工作
    }
    virtual ~Base(){}
private:
    virtual void vf() { cout<<"Base vf(): "<<i<<endl; }
    //假如派生类中需要调用父类的虚函数,可以将权限改为protected/public
    int i = 100;
};
```

```
class Derived :public Base {
private:
    virtual void vf()override {
        cout<<"Derived vf(): "<<j<< endl;
    }
    int j = 200;
};
int main() {
    Derived d;
    d.f();
    cout << "---------\n";
    Base *p = &d;
    p->f();
    return 0;
}
```

```
Base f()
typeid: class Derived
Derived vf(): 200
---------
Base f()
typeid: class Derived
Derived vf(): 200
请按任意键继续. . .
```

虚函数的默认参数

通过基类指针或引用调用虚函数时，属于动态绑定（运行时才决定运行哪个函数）
但是：默认参数是静态绑定（编译时决定）

所以：不要重新定义一个继承而来的默认参数值。
派生类中override的虚函数不应该再写默认值。

```
class Base {
public:
    virtual ~Base() = default;
    void f1(int i=0) {
        cout << "Base f1():" << i << endl;
    }
    virtual void vf2(int i = 0) {
        cout << "Base vf2():" << i << endl;
    }
};
class D1 :public Base {
public:
    void f1(int i=1) {
        cout << "D1 f1():"<<i << endl;
    }
    virtual void vf2(int i = 1)override{
        cout << "D1 vf2():" << i << endl;
    }
};
```

```
int main() {
    Base b;
    b.f1(); //0
    b.vf2();//0
    D1 d;
    d.f1(); //1
    d.vf2();//1
    Base *pb = &d;
    pb->f1(); //0
    pb->vf2();//0 不符合预期：默认参数是 Base::vf2中的
    D1 *pd = &d;
    pd->f1(); //1
    pd->vf2();//1 编译时确定哪个函数,哪个默认参数
    return 0;
}
```

```
Base f1():0
Base vf2():0
D1 f1():1
D1 vf2():1
Base f1():0
D1 vf2():0
D1 f1():1
D1 vf2():1
请按任意键继续.
```

不重新定义继承来的非虚函数

在派生类中重新定义继承而来的非虚函数，通过不同的指针或引用类型调用时，会出现不一致的情况。

假如想表达特异性（多态），那么就使用虚函数。

public继承，表达的意思是每个D1都是一个Base，不使用虚函数而重新定义，违背“is-a”的关系。

所以：不要重新定义继承而来的非虚函数。

```
class Base {
public:
    virtual ~Base() = default;
    void f1() {
        cout << "Base f1()\n";
    }
};

class D1 :public Base {
public:
    void f1() {
        cout << "D1 f1():\n";
    }
};
```

```
int main() {
    D1 d;
    Base *pb = &d;
    pb->f1(); //Base f1()
    D1 *pd = &d;
    pd->f1(); //D1 f1()
    //非虚函数调用:
    //通过不同的指针类型调用结果不同！
    //(语法没有错误,但是对用户造成伤害！)
    //虚函数不会如此！
    return 0;
}
```

数据成员尽量private

数据成员（成员变量）尽量使用private，而不是public和protected。

1. 用户在使用类的接口时，不再要确认是否要加( )。
2. 精确的权限控制，暴露的数据成员都是读写全开放的，而通过成员函数提供操作数据成员的接口可以轻松实现只读、只写或者读写的权限控制。
3. 封装：
   将成员变量隐藏在函数接口的后面，为“所有可能的实现”提供弹性。例如这可使得成员变量被读或被写时轻松通知其他对象，可以验证约束条件以及函数的前提和事后状态，可以在多线程环境中执行同步控制等等。Public意味着不封装，不封装意味着不可改变。使用函数接口，修改实现方式不会让用户改代码。假设使用了public数据成员，一旦改变，则所有用户代码中使用了该数据成员的代码都要修改，或者基类中使用了protected数据成员，它的改变同样会影响所有的派生类代码。所以：public数据成员意味着不可更改。