虚函数

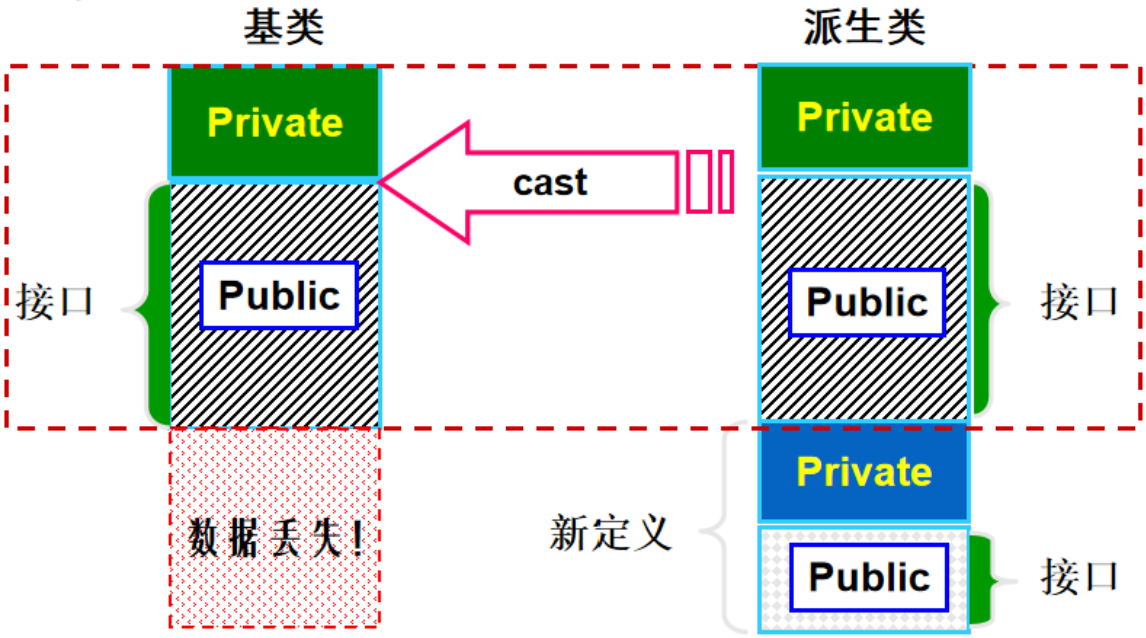
directory

向上类型转化

- **派生类**对象/引用/指针转换成**基类**对象/引用/指针。
- **只对public继承有效**，在继承图上是上升的；对private、protected继承无效。
- 向上类型转换（**派生类到基类**）可以由编译器**自动完成**，是一种隐式类型转换。
- 凡是接受基类对象/引用/指针的地方（如函数参数），**都可以使用**派生类对象/引用/指针，编译器会自动将派生类对象转换为基类对象以便使用。

对象切片

- 派生类对象转换位基类对象时，派生类对象被切片成基类的子对象。



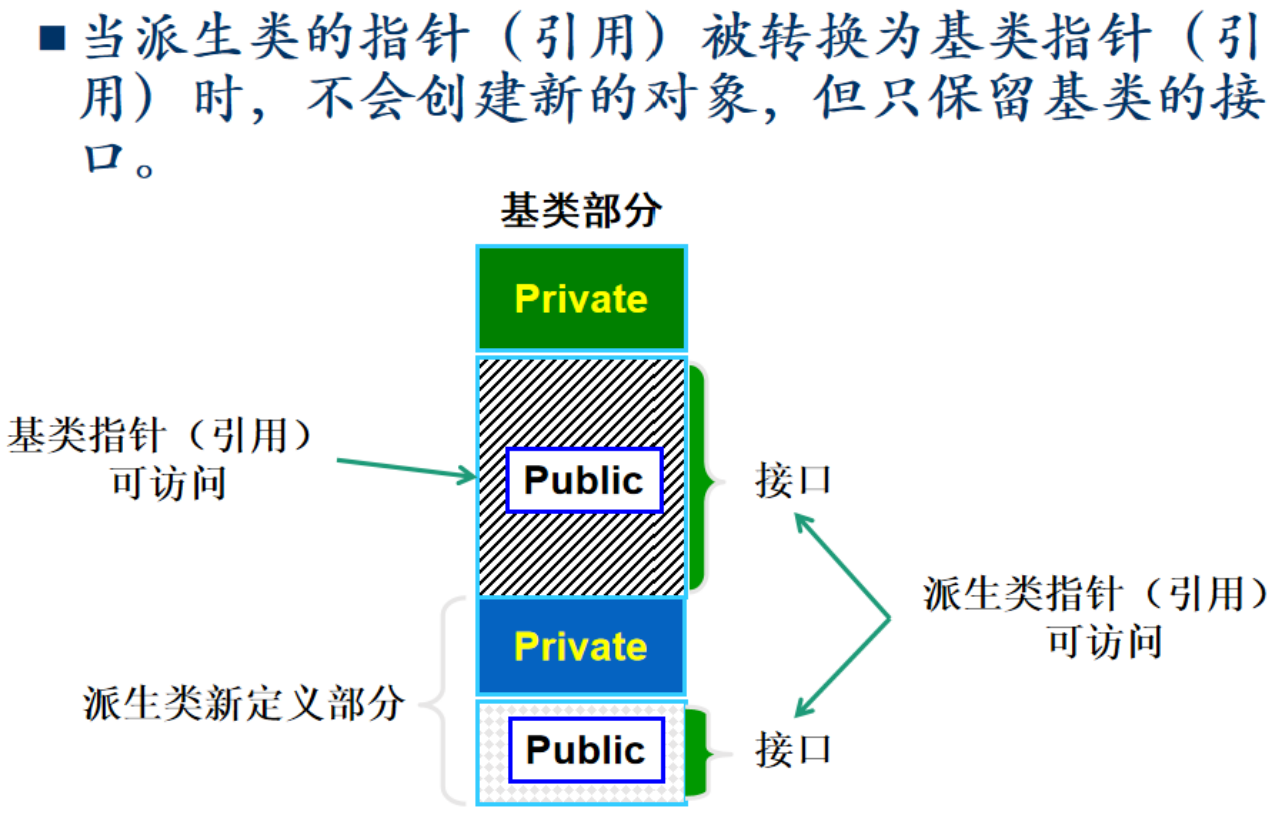
派生类新数据丢失

派生类新方法丢失

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

class Pet {
```

```
public:
  void name(){ cout << "Pet::name()" << endl; }
};
class Dog: public Pet {
public:
  void name(){ cout << "Dog::name()" << endl; }
};
void getName(Pet p){
  p.name();
}
int main() {
  Dog g;
  g.name();
  getName(g);   /// 对象切片（传参），调用基类的 name 函数
  Pet p = g;
  p.name();   /// 对象切片（赋值），调用基类的 name 函数
  return 0;
}

/*
输出:
Dog::name()
Pet::name()
Pet::name()
*/
```

引用的向上类型转换



- 派生类的新方法和基类的方法都可以使用

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

class Instrument {
    public:
        void play() { cout << "Instrument::play" << endl; }
};
class Wind : public Instrument {
    public:
        // Redefine interface function:
        void play() { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) {
    i.play();
}
int main() {
    Wind flute;
    tune(flute); /// 引用的向上类型转换(传参)，编译器早绑定，无对象切片产生
    Instrument &inst = flute;  /// 引用的向上类型转换(赋值)
    inst.play();
    return 0;
}

/*
输出:
Instrument::play
Instrument::play
*/
```

私有继承不能实现

```
#include <iostream>
using namespace std;
class B {
    private:
        int data{0};
    public:
        int getData(){ return data;}
        void setData(int i){ data=i;}
};
class D1 : private B {
    public:
        using B::getData;
};
int main() {
    D1 d1;
    cout<<d1.getData();
    //d1.setData(10);   ///隐藏了基类的setData函数，不可访问
    //B& 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;
}
```

函数调用捆绑

- 把函数体与函数调用相联系称为**捆绑**(binding)
    - 即将函数体的具体实现代码，与调用的函数名绑定。执行到调用代码时进入直接进入捆绑好的函数体内部。
- 当捆绑在程序运行之前（由编译器和连接器）完成时，称为**早捆绑**(early binding)
    - 运行之前已经决定了函数调用代码到底进入哪个函数。
    - 上面程序中的问题是早捆绑引起的，编译器将tune中的函数调用i.play()与Instrument::play()绑定。
- 当捆绑根据对象的实际类型(上例中即子类Wind而非Instrument)，发生在程序运行时，称为**晚捆绑**(late binding)，又称动态捆绑或运行时捆绑。
    - 要求在运行时能确定对象的实际类型(思考：如何确定？)，并绑定正确的函数。
    - 晚捆绑**只对类中的虚函数起作用**，使用 **virtual 关键字** 声明虚函数。

虚函数

- 对于被派生类重新定义的成员函数，若它**在基类中被声明为虚函数**，则通过**基类指针**或**引用**调用该成员函数时，编译器将**根据所指（或引用）对象的实际类型决定**是调用基类中的函数，还是调用派生类重写的函数。

```
  class Base {
   public:
    virtual ReturnType FuncName(argument); //虚函数
    ...
};
```

- 若某成员函数**在基类中**声明为虚函数，**当派生类重写覆盖它时(同名，同参数函数)**，无论是否声明为虚函数，该成员函数都仍然是虚函数。

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

class Instrument {
    public:
        virtual void play() { cout << "Instrument::play" << endl; }
};
class Wind : public Instrument {
    public:
        void play() { cout << "Wind::play" << endl; }
     /// 重写覆盖(稍后：重写隐藏和重写覆盖的区别)
```

```
};

void tune(Instrument& ins) {
    ins.play(); /// 由于 Instrument::play 是虚函数，编译时不再直接绑定，运行时根据 ins 的实际类型调用。
}
int main() {
    Wind flute;
    tune(flute); /// 向上类型转换
    return 0;
}

/*
输出:
Wind::play
*/
```

晚绑定只对指针和引用有效

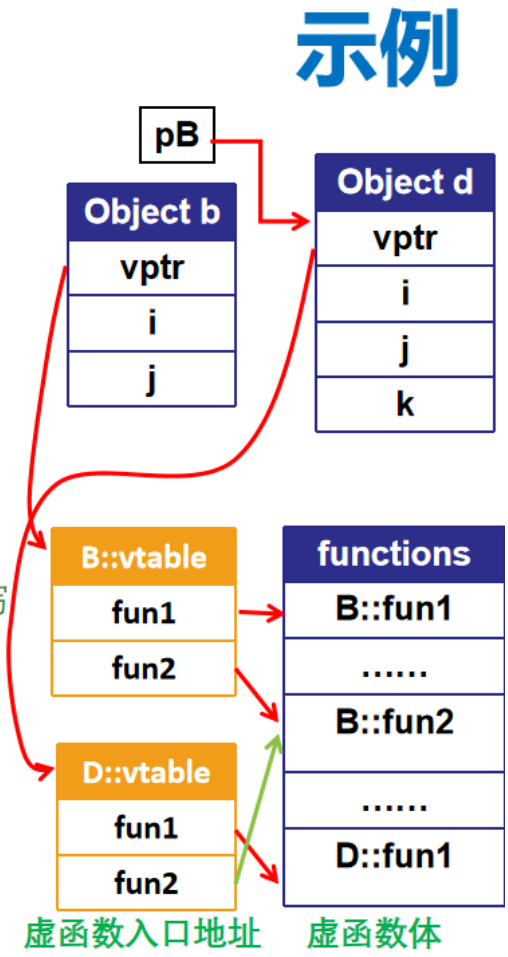
```
#include <iostream>
using namespace std;

class Instrument {
public:
  virtual void play() { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  void play() { cout << "Wind::play" << endl; }
};

void tune(Instrument ins) { ///
  ins.play(); /// 晚绑定只对指针和引用有效，这里早绑定 Instrument::play
}

int main() {
  Wind flute;
  tune(flute); /// 向上类型转换，对象切片
  return 0;
}

/*
输出:
Instrument::play
*/
```

虚函数表

- 对象自身要包含自己实际类型的信息：用虚函数表表示。运行时通过虚函数表确定对象的实际类型

- 虚函数表(VTABLE)：每个**包含虚函数的类用于存储虚函数地址的表**
  - 虚函数表有唯一性，即使没有重写虚函数
- 每个包含虚函数的**类对象**中，编译器秘密地放一个指针，称为**虚函数指针(vpointer/VPTR)，指向这个类的VTABLE。**
- 当通过基类指针做虚函数调用时，编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起**晚捆绑**的发生。
  - **编译期间**：建立虚函数表VTABLE，记录每个类或该类的基类中所有已声明的虚函数入口地址。
  - **运行期间**：建立虚函数指针VPTR，在构造函数中发生，指向相应的VTABLE。

```
#include <iostream>
using namespace std;
class B{
  int i;
  float j;
public:
  virtual void fun1() {
cout << "B::fun1()" << endl; }
  virtual void fun2() {
cout << "B::fun2()" << endl; }
};
class D: public B{
public:
  double k;
  virtual void fun1() {
cout << "D::fun1()" << endl; } ///对fun1重写
覆盖，对fun2没有，则fun2使用基类的虚函数地址
};
int main() {
  B b; D d;
  B *pB = &d;
  pB->fun1();
  return 0;
}
```



运行结果

D::fun1()

虚函数表的存放类型信息

```
#include <iostream>
using namespace std;
#pragma pack(4) //按照4字节进行内存对齐

class NoVirtual{ //没有虚函数
  int a;
public:
  void f1() const {}
  int f2() const {return 1;}
};

class OneVirtual{ //一个虚函数
```

```
  int a;
public:
  virtual void f1() const {}
  int f2() const {return 1;}
};

class TwoVirtual{//两个虚函数
  int a;
public:
  virtual void f1() const {}
  virtual int f2() const {return 1;}
};


int main(){
    cout<<"int: "<<sizeof(int)<<endl; // int: 4
    cout<<"NoVirtual: "<<sizeof(NoVirtual)<<endl; // NoVirtual: 4
    cout<<"void* : "<<sizeof(void*)<<endl; // void*: 8
    cout<<"OneVirtual: "<<sizeof(OneVirtual)<<endl; // OneVirtual: 12
    cout<<"TwoVirtual: "<<sizeof(TwoVirtual)<<endl; // TwoVirtual: 12
    return 0;
}
```

- 对不带虚函数的类NoVirtual,对象的大小就是单个int的大小。
- 对带有单个虚函数的类OneVirtual，对象的大小是单个int的大小加上一个void指针(**实际上是VPTR**)的大小。
- 带有多个虚函数的类TwoVirtual与OneVirtual大小相同，因为VPTR**指向一个存放所有虚函数地址的表。**

虚函数和构造函数、析构函数

虚函数与构造函数

- 当创建一个包含有虚函数的对象时，必须初始化它的VPTR以指向相应的VTABLE。设置VPTR的工作由构造函数完成。编译器在构造函数的开头秘密的插入能初始化VPTR的代码。
- **构造函数不能也不必是虚函数。**
  - **不能**：如果构造函数是虚函数，则创建对象时**需要先知道VPTR**，而在构造函数调用前，VPTR未初始化。
  - **不必**：构造函数的作用是提供类中成员初始化，调用时明确指定要创建对象的类型，没有必要是虚函数。
- 虚函数动态绑定：如果一个函数被声明为虚函数，那么当它被调用时，会动态地调用最具体的那个实现，也就是说，**如果在派生类中重写了虚函数，那么在调用这个虚函数时，会动态地调用派生类中的实现**
  - 只有通过指针或引用调用虚函数时才会发生动态绑定
  - 如果直接通过对象名调用虚函数，那么会发生静态绑定，即调用基类中的实现

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

class Base {
```

```
public:
  virtual void foo(){cout<<"Base::foo"<<endl;}
  Base(){foo();}      ///在构造函数中调用虚函数foo
  void bar(){foo();};    ///在普通函数中调用虚函数foo
};

class Derived : public Base {
public:
  int _num;
  void foo(){cout<<"Derived::foo"<<_num<<endl;}
  Derived(int j):Base(),_num(j){}
};
int main() {
  Derived d(0);
  Base &b = d;
  b.bar();
  b.foo();
  return 0;
}
/*
输出:
Base::foo      //构造函数中调用的是foo的“本地版本”
                 为什么? (提示:基类构造时_num的状态)
Derived::foo0 //在普通函数中调用
Derived::foo0 //直接调用

过程:
因为派生类的数据成员还没有被初始化。因此,虚函数的调用会被解析为基类的foo()函数,输出
“Base::foo”。

虚函数动态绑定:在调用b.bar()时,调用基类base中的bar()函数,在bar()中调用了虚函数
foo(),所以会动态地调用最具体的那个实现,即Derived类中的foo()函数,输出“Derived::foo0”

在调用b.foo()时,因为b是基类的引用,因此会静态地调用Base类中的foo()函数,输出
“Derived::foo0”
*/
```

- 在构造函数中调用一个虚函数，被调用的只是这个函数的本地版本(即当前类的版本)，即虚机制在构造函数中不工作。
- 初始化顺序：(与构造函数初始化列表顺序无关)
  - **基类**初始化
  - **对象成员**初始化
  - **构造函数体**
- 原因：**基类的构造函数比派生类先执行**，调用基类构造函数时派生类中的数据成员还没有初始化(上例中Derive中的数据成员i)。如果允许调用实际对象的虚函数(如b.foo())，则可能会用到未初始化的派生类成员。

虚析构函数

- 析构函数能是虚的，且常常是虚的。虚析构函数仍需定义函数体。
- 虚析构函数的用途：

    - 当删除基类对象指针时，编译器将根据指针所指对象的实际类型，调用相应的析构函数。
- 若基类析构不是虚函数，则删除基类指针所指派生类对象时，编译器仅自动调用基类的析构函数，而不会考虑实际对象是不是基类的对象。这可能会导致内存泄漏。
- 在析构函数中调用一个虚函数，被调用的只是这个函数的本地版本，即虚机制在析构函数中不工作

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

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; } // 将基类的析构函数设置为虚析构函数
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1;
  delete bp; /// 只调用了基类的析构函数 只析构基类，不析构派生类

  Base2* b2p = new Derived2;
  delete b2p; /// 派生类虚析构函数调用完后调用基类的虚析构函数 析构派生类再析构基类
  return 0;
}

/*
输出:
~Base1()
~Derived2()
~Base2()
*/
```

- **重要原则：总是将基类的析构函数设置为虚析构函数**

重写覆盖，override和final

- 重载(overload)
    - 函数名、作用域相同(同一个类，或同为全局函数)，返回值可以相同/不同，**参数**列表不同

- 重写覆盖(override)
  - 派生类重新定义基类中的**虚函数**，**函数名**、函数**参数**必须相同，**返回值**一般情况应相同。
  - **派生类的虚函数表中原基类的虚函数指针会被派生类中重新定义的虚函数指针覆盖。**
- *即：如果基类成员函数被声明为虚函数，则在派生类中重新定义该函数时，如果函数名称、参数列表和返回类型都与基类中的虚函数相同，那么派生类中的函数将自动成为虚函数，覆盖基类中的虚函数*
- 重写隐藏(overdefining)
  - 派生类重新定义基类中的函数，函数名相同，但是**参数不同**或者**基类的函数不是虚函数。**(参数相同+基类虚函数->不是重写隐藏)
  - **重写隐藏中虚函数表不会发生覆盖**
- 相同点：
  - 都要求派生类定义的函数与基类**同名**
  - 都会屏蔽基类中的同名函数，即**派生类的实例无法调用基类的同名函数**
- 不同点：
  - 重写覆盖：**基类的函数是虚函数 + 函数参数相同，返回值一般情况应相同，**
  - 重写隐藏：基类的函数不是虚函数或者函数参数不同。
  - 重写覆盖**会使派生类虚函数表中 基类的虚函数的指针 被派生类的虚函数指针覆盖。**重写隐藏不会。

	重载(overload)	重写隐藏(redefining)	重写覆盖(override)
作用域	相同(同一个类中，或者均为全局函数)	不同(派生类和基类)	不同(派生类和基类)
函数名	相同	相同	相同
函数参数	不同	相同/不同	相同
其他要求	—	如果函数参数相同，则基类函数不能为虚函数	基类函数为虚函数

```
#include <iostream>
using namespace std;
class Base{
    public:
        void foo(float){}         //(1) 是(2)和(3)的重载
        virtual void foo(){}      //(2)
        virtual void foo(int){}   //(3)
};
class Derived : public Base {
    public:
        void foo() {}             //(4) 是重写覆盖
        virtual void foo(float){} //(5) 不是重写覆盖，因为基类中的(1)不是虚函数
};
```

```
int main(){
    Derived d;
    d.foo(1);                   //(6) 无编译错误，调用(5)
    return 0;
}
/*
在执行虚函数调用时，编译器会将参数1隐式地转换为浮点型参数，然后调用Derived类中的
foo(float)函数，而不是调用Base类中的foo(int)函数
*/
```

override关键字

- override关键字明确地告诉编译器一个函数是对基类中一个虚函数的重写覆盖，编译器将对重写覆盖要满足的条件进行检查，正确的重写覆盖才能通过编译。
- override关键字只是编译器的一个检查，正确实现重写覆盖一样可以通过。

```
#include <iostream>
using namespace std;
class Base{
    public:
        virtual void foo(){cout<<"Base::foo()"<<endl;}
        virtual void foo(int ){cout<<"Base::foo(int )"<<endl;} ///重载
        void bar(){};
};
class Derived1 : public Base {
    public:
        void foo(int ) {cout<<"Derived1::foo(int )"<<endl;} /// 是重写覆盖
};
class Derived2 : public Base {
    public:
        void foo(float ) {cout<<"Derived2::foo(float )"<<endl;} /// 参数不同，不是
重写覆盖，是重写隐藏
};
class Derived3 : public Base {
    public:
        void foo(int ) override {cout<<"Derived3::foo(int )"<<endl;}; /// 重写覆盖
正确，与Derived1等价
        //void foo(float ) override {}; /// 参数不同，不是重写覆盖，编译错误
        //void bar() override {}; /// bar 非虚函数，编译错误
};
int main() {
    Derived1 d1;
    Derived2 d2;
    Derived3 d3;
    Base* p1 = &d1;
    Base* p2 = &d2;
    Base* p3 = &d3; // 基类指针，指向派生类对象
    //d1.foo(); ///由于派生类都定义了带参数的foo，基类foo()对实例不可见
    //d2.foo();
    //d3.foo();
    ///但是虚函数表中有继承自基类的foo()虚函数
```

```
    // 当基类指针或引用调用虚函数时
    // 会根据实际指向的派生类对象的类型在虚函数表中查找相应的虚函数地址并调用
    p1->foo(); // Base::foo()
    p2->foo(); // Base::foo()
    p3->foo(); // Base::foo()
    d1.foo(3); // Derived1::foo(int ) ///重写覆盖
    d2.foo(3.0); // Derived2::foo(float ) ///调用的是派生类foo(float )
    d3.foo(3); // Derived3::foo(int ) ///重写覆盖
    p1->foo(3); // Derived1::foo(int ) ///重写覆盖
    p2->foo(3.0); // Base::foo(int ) ///重写隐藏，调用的是基类foo(int)
    p3->foo(3); // Derived3::foo(int )///重写覆盖
    return 0;
}
```

final关键字

- 在**虚函数声明或定义**中使用时，final**确保函数为虚且不可被派生类重写。**可在继承关系链的"中途"进行设定，禁止后续派生类对指定虚函数重写。
- 在**类定义**中使用时，final*指定此类不可被继承*。

```
class Base{
  virtual void foo(){};
};
class A: public Base {
  void foo() final {}; /// 重写覆盖，且是最终覆盖
  void bar() final {}; /// bar 非虚函数，编译错误
};
class B final : public A{ /// final 指定B类不能被继承
  void foo() override {}; /// A::foo 已是最终覆盖，编译错误
};
class C : public B{ /// B 不能被继承，编译错误
};
```

总结：OOP核心思想：数据抽象、继承与动态绑定

- 数据抽象：类的接口与实现分离
    - 回顾Animal\模板设计的例子
- 继承：建立相关类型的层次关系（基类与派生类）
    - Is-a、is-implementing-in-terms-of: 客观世界的认知关系
- 动态绑定：统一使用基类指针，实现多态行为
    - 虚函数
    - 类型转换，模板