虚函数与多态

1. 函数调用捆绑

把函数体与函数调用相联系称为捆绑(binding)。即，将函数体实现代码的**入口地址**，与调用的**函数名**绑定。执行到调用代码时进入函数体内部。

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

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

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

void tune(Instrument& i) {
  i.play();
}

int main() {
  Wind flute;
  tune(flute); /// 向上类型转换
}
```

问题在哪?

运行结果

Instrument::play

- 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)。即，在运行之前已经决定了函数调用代码到底进入哪个函数。
- 当捆绑根据对象的**实际类型**，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。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。

2.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**（如下所示），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。

问题解决

```cpp
#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
}
```

如何在运行时确定对象的实际类型？这与虚函数有何关联？

运行结果

```
Wind::play
```

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。并且，晚捆绑只对**指针和引用**有效(上图)，对对象无效（下图）：

【虚函数表】

那么，怎么能够在运行的时候确定对象的实际类型，从而调用正确的函数呢？实际上，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。

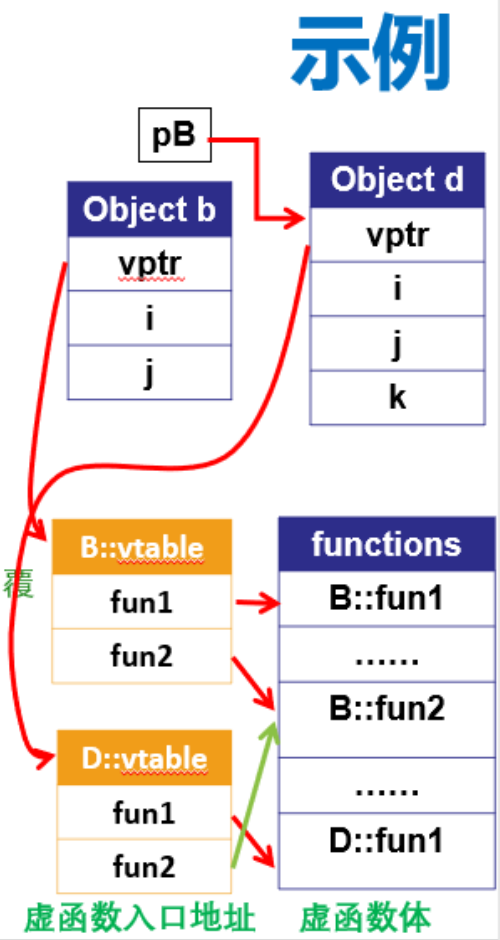
当通过基类指针做虚函数调用时，编译器静态地插入能取得这个VPTR并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数并引起晚捆绑的发生。

- **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。

```cpp
#include <iostream>
using namespace std;
class B{
public:
  virtual void fun1() {
cout << "B::fun1()" << endl; }
  virtual void fun2() {
cout << "B::fun2()" << endl; }
private:
  int i;
  float j;
};
class D: public B{
public:
  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
  double k;
};
int main() {
  B b; D d;
  B *pB = &d;
  pB->fun1();
}
```

运行结果

D::fun1()



上图中，object b和object d都有指针vptr, 指向B的虚函数表和D的虚函数表。这个vptr就是在构造函数构造的。

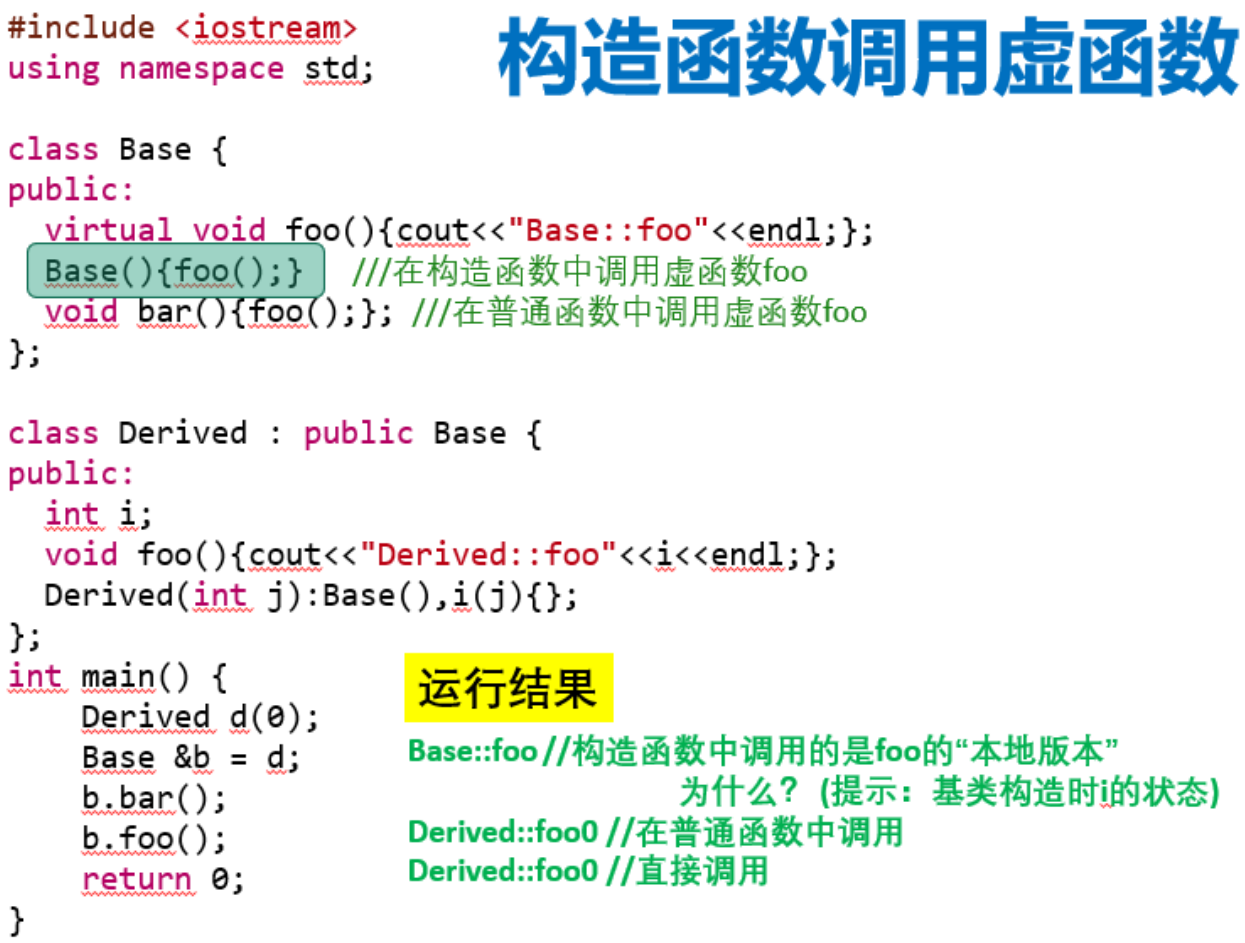
3. 虚函数和构造函数、析构函数

3.1 虚函数与构造函数

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VPTR本身就是在构造函数中构造的，在构造函数调用的时候，VPTR根本就没有初始化！
- 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。

既然构造函数本身不是虚函数，那么假如在构造函数中调用虚函数会怎么样呢？其实，**虚机制在构造函数中不工作**。看下面这个例子：



值得注意的是，main()函数的第一行派生类构造函数 `Derived d(0)` 调用基类构造函数 `Base(){foo();}`，这里的 `foo()` 调用的是基类的版本而不是派生类中重写的版本。

这是因为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(如上例中Derive中的数据成员 `i`)。如果允许调用实际对象的虚函数 `foo()`，则可能会用到未初始化的派生类成员。所以说，虚机制在构造函数中不工作。

3.2 虚函数与析构函数

析构函数能是虚的，且常常是虚的。为什么要这样做呢？这是因为，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。

4. 重写覆盖，override和final

4.1 重写覆盖

- 重载(overload)：函数名必须相同，**函数参数必须不同**，作用域相同(同一个类)，返回值可以相同或不同。
- 重写覆盖(**override**)：派生类重新定义基类中的**虚函数**，函数名必须相同，**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重新定义的虚函数指针覆盖掉。
- 重写隐藏(redefining)：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+虚函数->不是重写隐藏，而是重写覆盖)

```
#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;}; /// 误把参数写错了，不是重写覆盖，是重写隐藏
};
```

4.2 override关键字

重写覆盖要满足的条件很多，很容易写错，出现本来想实现的是重写覆盖、实际却是重写隐藏的情况。因此可以使用**override**关键字辅助检查，正确的重写覆盖才能通过编译。当然了，如果没有override关键字，但是满足了重写覆盖的各项条件，也能实现重写覆盖。

4.3 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。**
- 不想让使用者继承？-> final关键字! 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{ ///不可被继承的类
  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
};
```

5. 纯虚函数与抽象类

包含纯虚函数的类，通常被称为“抽象类”。抽象类**不允许定义对象**，定义基类为抽象类的主要用途是为派生类规定共性“接口”。因此当继承一个抽象类时，必须实现所有纯虚函数，否则继承出的类也是抽象类。

```
class A {
public:
    virtual void f() = 0; /// 可在类外定义函数体提供默认实现。派生类通过 A::f() 调用
};
A obj; /// 不准抽象类定义对象！编译不通过！
```

纯虚函数的设计能避免对象切片：保证只有指针和引用能被向上类型转换。

6. 向下类型转换

基类指针/引用转换成**派生类**指针/引用，则称为向下类型转换。（类层次中向下移动）为什么要向下类型转换?

- 当我们用基类指针表示各种派生类时(向上类型转换)，保留了他们的共性，但是丢失了他们的特性。如果此时要表现**特性**，则可以使用向下类型转换。
- 比如我们可以使用基类指针数组对各种派生类对象进行管理，当具体处理时我们可以将基类指针转换为实际的派生类指针，进而调用派生类专有的接口。

如何确保转换的正确性?

示例

```
class Pet { public:  virtual ~Pet() {} };
class Dog : public Pet {
public: void run() { cout << "dog run" << endl; }
};
class Bird : public Pet {
public: void fly() { cout << "bird fly" << endl; }
};

void action(Pet* p)
{
    auto d = dynamic_cast<Dog*>(p);  /// 向下类型转换
    auto b = dynamic_cast<Bird*>(p); /// 向下类型转换
    if (d) /// 运行时根据实际类型表现特性
        d->run();
    else if(b)
        b->fly();
}

int main() {
    Pet* p[2];
    p[0] = new Dog; /// 向上类型转换
    p[1] = new Bird; /// 向上类型转换
    for (int i = 0; i < 2; ++i) {
        action(p[i]);
    }
}
```

运行结果

```
dog run
bird fly
```

【dynamic_cast】

c++提供了一个特殊的显式类型转换，称为**dynamic_cast**，是一种**安全**类型向下类型转换。

使用dynamic_cast必须有虚函数，因为它使用了存储在虚函数表中的信息判断实际的类型。

使用方法：

obj_p，obj_r分别是T1类型的指针和引用

- T2* pObj = dynamic_cast<T2*>(obj_p); //转换为T2指针，运行时失败返回nullptr
- T2& refObj = dynamic_cast<T2&>(obj_r);//转换为T2引用，运行时失败抛出bad_cast异常

T1必须是多态类型（声明或继承了至少一个虚函数的类），否则不过编译；T2不必。T1,T2没有继承关系也能通过编译，只不过运行时会转换失败。

【static_cast】

如果我们知道正在处理的是哪些类型，可以使用**static_cast**来避免这种开销。

- static_cast在编译时静态浏览类层次，只检查继承关系。没有继承关系的类之间，必须具有转换途径才能进行转换（要么自定义，要么是语言语法支持），否则不过**编译**。运行时无法确认是否正确转换。

static_cast使用方法：

obj_p，obj_r分别是T1类型的指针和引用

- T2* pObj = static_cast<T2*>(obj_p); //转换为T2指针
- T2& refObj = static_cast<T2&>(obj_r); //转换为T2引用

不安全：不保证转换后的目标是T2类型的。

7. 多态

按照基类的接口定义，调用**指针或引用**所指对象的接口函数，函数执行过程因对象实际所属派生类的不同而呈现不同的效果（表现），这个现象被称为“多态”。当利用基类**指针/引用**调用函数时，**虚函数**在**运行时**确定执行哪个版本，取决于引用或指针对象的真实类型；非虚函数则在**编译时**绑定。当利用类的对象直接调用函数时，无论什么函数，均在**编译时**绑定

产生多态效果的条件：**继承** && **虚函数** && (**引用 或 指针**)

【OOP的核心思想】：数据抽象、继承与多态

- **数据抽象**：类的接口与实现分离【怕【h'p
- **继承**：建立相关类型的层次关系（基类与派生类）
- **多态**