

# 第十五章 多态性

## 1. 虚函数

情景：同一基类派生出多个子类

外部函数参数为基类对象但希望实现  
派生类内函数操作

**CASE 1: Why use virtual function?**

必须传入用基类对象

```
#include <iostream>
using namespace std;
class Instrument {
public:
    void play() const
    { cout << "Instrument::play()" << endl; }
};
class Wind : public Instrument {
public:
    void play() const
    { cout << "Wind::play()" << endl; }
};
class Stringed : public Instrument {
public:
    void play() const
    { cout << "Stringed::play()" << endl; }
};

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

int main()
{
    Wind flute;
    tune(flute);
    Stringed guitar;
    tune(guitar);
    return 0;
}
```

基类的

解决：使用虚函数

```
#include <iostream>
using namespace std;
class Instrument {
public:
    virtual void play() const
    { cout << "Instrument::play()" << endl; }
};
class Wind : public Instrument {
public:
    virtual void play() const
    { cout << "Wind::play()" << endl; }
};
class Stringed : public Instrument {
public:
    void play() const // virtual can be omitted
    { cout << "Stringed::play()" << endl; }
};

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

int main()
{
    Wind flute;
    tune(flute);
    Stringed guitar;
    tune(guitar);
    return 0;
}
```

原

因

由于参数类型为  
基类对象，因而  
编译器自动调用  
基类函数

虚函数帮助实现

由父类接口访问子类函数

↓  
虚函数使用前缀关键字 `virtual`  
使得该函数变成了一个接口，编译器将  
根据传入的对象实现 不同派生类的函数操作  
对象的动态绑定

※ 当基类中的函数被定义为虚函数时，  
派生类中的同型函数也会自动被指定为  
虚函数，`virtual`关键字可缺省。

※ 基类中定义的虚函数和子类中的虚函数格式务必  
一模一样 → ① 返回类型  
② 参数表  
③ 函数名

※ 虚函数一定为类的成员函数  
因而无法将友元函数定义为虚函数

※ 析构函数可以定义为虚函数  
但构造函数不行

※ 传入虚函数/调用虚函数的务必为对象的  
引用或指针 → 才能动态绑定

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

```
class Instrument {
public:
```

```
void tune() { play(); }
void play() const
{ cout << "Instrument::play()" << endl; }
```

```
};
```

```
class Wind : public Instrument
{
public:
```

```
void play() const
{ cout << "Wind::play()" << endl; }
```

```
};
```

↑ 此处  
↓ this->play()  
即 this 指针  
进行动态绑定

virtual void play() const;

```
class Stringed : public Instrument
```

```
{
public:
```

```
void play() const
```

```
{
```

```
cout << "Stringed ::play()";
}
```

```
};
```

```
int main() {
```

```
Wind flute;
```

```
flute.tune();
```

```
Stringed guitar;
```

```
guitar.tune();
```

```
return 0;
```

```
}
```

## 2. 析构函数虚函数化

从而所有子类的析构函数也为虚函数

一般将基类的析构函数定义为虚函数  
便于实现对基类和类的不同析构操作

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

```
class Item {
public:
```

```
Item() { id = 0; }
virtual ~Item() { cout << "Item deleted" << endl; }
```

```
private: int id;
};
```

```
class BookItem : public Item {
public:
```

```
BookItem() { title = new char [50]; }
~BookItem()
```

```
{
cout << "BookItem deleted" << endl;
if (title != nullptr) delete[] title;
}
```

```
private: char * title;
};
```

```
int main()
```

```
{
```

```
Item * p;
```

```
p = new Item();
```

```
delete p;
```

```
p = new BookItem();
```

```
delete p;
```

```
return 0;
```

```
}
```

Microsoft Visual Studio 调试控制台

```
Item deleted
BookItem deleted
Item deleted
```

此时不要求函数名一样

### 3. 虚函数的实现原理 → 动态绑定

普通函数的参数绑定类型均为静态绑定，  
即函数的调用与参数的数据类型相关  
不具有多态性

与之相对，虚函数的实现利用了动态绑定，  
即函数的调用与程序当前传入的引用或指针  
所动态指定的对象有关

具有多态性



含有虚函数的类将被编译器自行隐含地增加  
一个指针，指向包含该类所有虚函数的函数  
表 [作用域为该类的]，供动态绑定时使用

※ 子类中虚函数的返回类型可以与基类中  
的虚函数不同，但必须为父类虚函数  
返回类型的子类型

## 4. 抽象类和纯虚函数

一个纯虚函数是定义为不被实现的虚函数，该虚函数以 '=0' 为标明标记，是该类连接其子类虚函数的接口。


无函数体定义

`virtual void Function ( ) =0 ;`

当一个类中定义了纯虚函数，则在其子类中应当对该接口功能进行实现，否则其类也继承该纯虚函数化为抽象类

抽象类：即成员函数中包含有纯虚函数的类。这种类将作为父类存在，表示一种抽象概念，因而无法定义实例对象。

※ 特别的，对于纯虚析构函数，允许其在类外给予定义以满足对象销毁时调用析构函数的形式要求



### 15.6 Pure virtual destructor

In common sense, we don't give the source code for pure virtual function. But in the special, it's possible to provide a definition for a pure virtual function in the base class. *There may be a common piece of code that we want some or all of the derived class definitions to call rather than duplicating that code in every function.*

```
#include <iostream>
using namespace std;
class Pet {
public: virtual ~Pet() = 0;
};
// Don't implement in the class
Pet::~Pet() { cout << "~Pet()" << endl; }

class Dog: public Pet {
public: ~Dog() { cout << "~Dog()" << endl; }
};

int main()
{
    // Upcase
    Pet *p = new Dog();
    // Virtual destructor call
    delete p;
    return 0;
}
```

实现

## 5. 父类向子类的访问(自上而下) *downcasting*

① 可以用父类的引用或指针 指向其子类的对象,  
由该父类引用(指针)来访问 子类的部分成员  
↓  
应在基类中包含

② 不同的子类之间是平级且无联系的,  
不能互相转化.

