第五讲 继承

五、虚函数

六、多态的代价

五、 虚函数

5.1 虚函数使用

当父类指针或引用指向子类对象，而子类中又覆盖了父类的函数，希望用父类指针或父类引用，调用到正确版本的函数，需要把该成员函数声明为虚函数。

【例5-1】虚函数使用

```
/** Base.h  **/
#ifndef Base_hpp
#define Base_hpp

#include <iostream>
using namespace std;
class Base{

public:
    virtual void func(){
        cout << "Base class function.\n";
    }
};
#endif
```

```
/** Derived.h  **/
#ifndef Derived_hpp
#define Derived_hpp

#include <iostream>
using namespace std;
#include "Base.hpp"
class Derived : public Base{

public:
    void func(){
        cout << "Derived class function.\n";
    }
};
#endif
```

```
/** main.cpp **/
#include "Base.hpp"
#include "Derived.hpp"

void foo(Base & b){
    b.func( );
}

int main(int argc, const char * argv[]) {

    //之前我们写的子类对象当父类对象使用结果?
    Derived d;
    Base b;
    Base* p = &d;
    Base& br = d;

    b = d;
    b.func( );
    d.func( );
    p -> func( );
    foo(d);
    foo(b);
    br.func( );

    return 0;
}
```

程序运行结果如下:

```
Base class function.
Derived class function.
Derived class function.
Derived class function.
Base class function.
Derived class function.
```

【例5-2】虚函数使用

```
/** Thing.h  **/

#ifndef Thing_hpp
#define Thing_hpp

#include <iostream>
using namespace std;

class Thing{

public:
    virtual void what_Am_I( ){
        cout << "I am a Thing.\n";
    }
    ~Thing(){
        cout<<"Thing destructor"<<endl;
    }
};
#endif
```

```
/** Animal.h  **/

#ifndef Animal_hpp
#define Animal_hpp

#include <iostream>
using namespace std;
#include "Thing.hpp"

class Animal : public Thing{

public:

    //如果父类的成员函数是虚函数，则子类覆盖后，不写virtual，也是虚函数。
    void what_Am_I( ){
        cout << "I am an Animal.\n";
    }

    ~Animal(){
        cout<<"Animal destructor"<<endl;
    }
};
#endif
```

```cpp
/** main.cpp **/

#include "Thing.hpp"
#include "Animal.hpp"

int main(int argc, const char * argv[]) {

    Thing t ;
    Animal x ;
    Thing* array[2];

    array[0] = &t; //指向父类
    array[1] = &x; //指向子类

    for (int i=0; i<2; i++){
        array[i]->what_Am_I( );
    }

    return 0;
}
```

程序运行结果如下:

```
I am a Thing.
I am an Animal.
Animal destructor
Thing destructor
Thing destructor
```

5.2 虚析构函数

如果一个类有子类，则这个类(父类)的析构函数必须是虚函数，即虚析构！

如果父类的析构不是虚析构，则当（用delete）删除一个指向子类对象的父类指针时，将调用父类版本的析构函数，子类只释放了来自于父类的那部分成员变量，而子类自己扩展的成员没有被释放，造成内存泄露。

【例5-3】非虚析构函数使用

```
/** Thing.h  **/

#ifndef Thing_hpp
#define Thing_hpp

#include <iostream>
using namespace std;

class Thing{
public:
    ~Thing(){
        cout<<"Thing destructor"<<endl;
    }
};
#endif
```

```
/** Animal.h  **/

#ifndef Animal_hpp
#define Animal_hpp

#include <iostream>
using namespace std;
#include "Thing.hpp"

class Animal : public Thing{
public:
    ~Animal(){
        cout<<"Animal destructor"<<endl;
    }
};

#endif
```

```
/** main.cpp **/

#include "Thing.hpp"
#include "Animal.hpp"

int main(int argc, const char * argv[]) {
    Thing* t = new Thing();
    Animal*  x = new Animal();

    Thing* array[2]; //父类指针数组
    // array[0]指向父类对象；array[1]指向子类对象。
    array[0] = t;
    array[1] = x;

    delete array[0];
    //因为析构函数不是虚函数，所以调用非真实身份(调用基类析构函数)
    delete array[1];

    return 0;
}
```

程序运行结果如下：

```
Thing destructor
Thing destructor
```

示例分析:

在上面程序中我们看到最后的析构函数只是调用了两次基类的析构函数，而子类扩展出来的成员堆上空间没有进行释放，所以造成内存泄漏。

我们把上面程序基类中的析构函数改成:

```
virtual ~Thing(){
   cout<<"Thing destructor"<<endl;
}
```

输出结果:

```
I am a Thing.
I am an Animal.
Thing destructor
Animal destructor
Thing destructor
```

所以:如果一个类有子类，一般则把基类的析构写成virtual。

5.3 动态绑定

虚函数被调用的时候，到底调用那个版本，在编译的时候无法确定，只有在执行时才能确定，称为动态绑定。

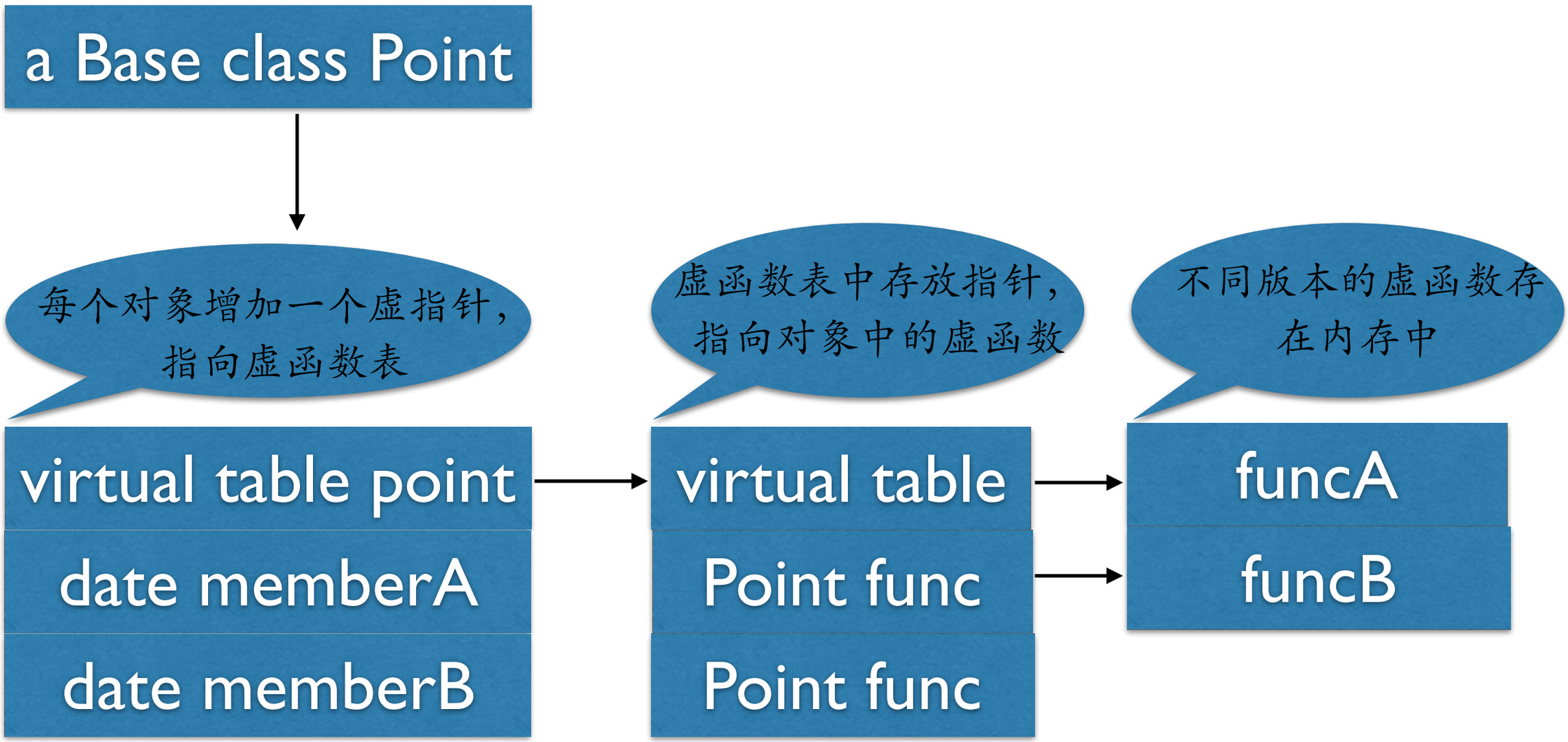
绑定使得程序可以照顾到未来增加的代码，比如创建一个新的子类，并在子类中覆盖了父类的虚函数。用之前的父类指针，依然可以正确的调用到新子类里的函数，而无需对旧有代码进行更改。

六、 多态的代价

多态：用父类指针or引用，统一操作各种子类对象。

为了实现动态绑定，编译器为每一个包含虚函数的类提供一个虚函数表，这个虚函数表被一个虚指针指向，这个类的虚函数表包含一个数组用于存放虚函数的地址，每一个指针指向了类中的虚函数地址。

当虚函数被调用时，编译器会使用该对象中的虚指针来查找虚函数表，然后遍历虚函数表，以查找到虚函数的指针（地址），最终找到正确版本的函数。


a Base class Point
每个对象增加一个虚指针，指向虚函数表
虚函数表中存放指针，指向对象中的虚函数
不同版本的虚函数存在内存中
virtual table point
date memberA
date memberB
virtual table
Point func
Point func
funcA
funcB

Q: 如果创建了一个新的子类C，会怎样?

A: 编译器为子类C创建一个虚函数表，表里存储的是C实现的虚函数的函数指针;

在每个C的对象里，添加一个虚指针成员变量，虚指针指向虚函数表。每个对象都多出4个字节的内存开销，来存储虚指针。

通过这个过程，我们可以看到，为了实现虚拟功能，必须构建一个虚拟表，用于每个类具有虚拟功能。

此外，在一类具有虚拟功能的类中的每一个对象都有一个隐藏的成员，即虚拟指针。所有这些都增加了程序使用的内存空间。

因此，当一个虚拟函数被调用时，程序必须在到达该函数的代码之前进行两次寻址，所以速度是比一般函数慢10% - 20% 。