Binding

binding 顾名思义为绑定

涉及到的一些知识点如同virtual function等

首先我们有一个base类

在base类中我们定义了一个虚函数virtual function

那么这个虚函数就可以被下面的派生类重写

具体写法如下

```
class Base{
    public:
    virtual VirtualFunction(){
        //
    }

}

class Derived : public Base{
    public:
    VirtualFunction() override{
        //
    }
}
```

这个就是一个最简单的虚函数重写

那么这个override可以不用写，照样可以运行

这里有一个例子

```
class Scientist {
    char name[40];
public:
    virtual void show_name();
    virtual void show_all();
    // ...
};

class Physicist : public Scientist {
    char field[40];
public:
    void show_all(); // Redefined
    virtual void show_field(); // New virtual function
    // ...
};
```

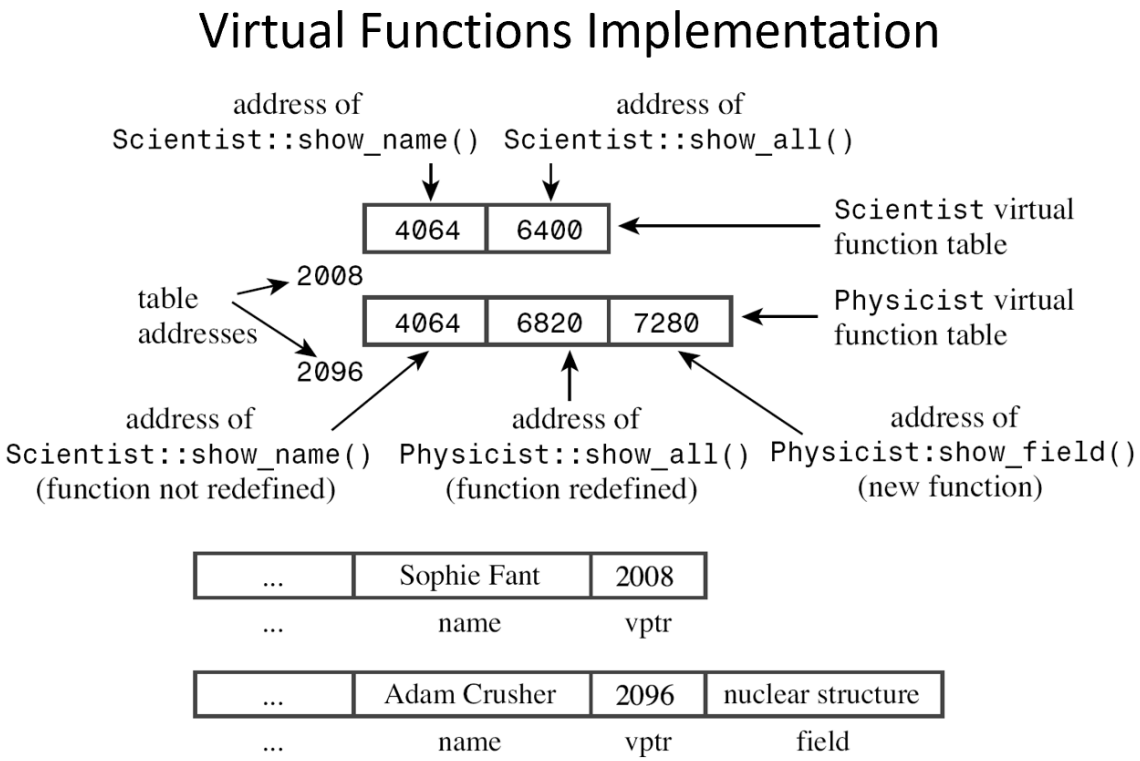
注意看这个基类Scientist里面有两个虚函数

virtual void show_name();
virtual void show_all();

派生类中有一个重写函数void show_all();

和一个新的虚构函数virtual void show_field(); // New virtual function

那现在我们来讨论一下这些函数的地址都是多少



首先两个类的地址都是不一样的

其次关于直接继承下来的函数，也就是没有被重写的函数

这两个类中的这种函数都是指向同一个地址

其次是重写过的函数，那么这两个函数肯定地址就不一样了

```
Physicist adam (“Adam Crusher”, “Nuclear structure”);
Scientist* psc = &adam;
psc->show_all();
```

1. Find value of psc-vptr, which is 2096.
2. Go to table at 2096.
3. Find address of second function in table, which is 6820.
4. Go to that address (6820) and execute the function found there.

Pure Virtual Function

关于这个纯虚函数

写法上和普通的虚函数有一定的区别

```
virtual ReturnType FunctionName(Parameters...) const = 0;
```

关于这个const写不写，都可以，但是如果你要在这里写的话，那么下面派生类重写的函数后面也要加上const在同样的位置

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

// 抽象基类
class Character {
public:
    // 纯虚函数，定义了一个接口
    virtual void attack() const = 0;

    virtual ~Character() {}
};

// 派生类：战士
class Warrior : public Character {
public:
    void attack() const override {
        cout << "Warrior attacks with a sword!" << endl;
    }
};

// 派生类：弓箭手
class Archer : public Character {
public:
    void attack() const override {
        cout << "Archer attacks with a bow!" << endl;
    }
};

// 派生类：法师
class Mage : public Character {
public:
    void attack() const override {
        cout << "Mage casts a spell!" << endl;
    }
};

void performAttack(const Character& character) {//一般来说传入引用都加一个const在前面好一些
    character.attack();  // 多态性
}

int main() {
    Warrior warrior;
    Archer archer;
    Mage mage;

    performAttack(warrior);
```

```
    performAttack(archer);
    performAttack(mage);

    return 0;
}
```

虚函数后面一定写的是等于0，后面{}都不可以加

这种写法也只有虚函数能写

后面重写的部分和上面差不太多

最后再提一嘴调用的问题

两种方法：

1. **点操作符（`.`）**：
    - 用于直接通过对象实例来访问成员函数或成员变量。
    - 示例：如果 `character` 是一个对象实例，你会使用 `character.attack()` 来调用成员函数。
2. **箭头操作符（`->`）**：
    - 用于通过指向对象的指针来访问成员函数或成员变量。
    - 示例：如果 `characterPtr` 是一个指向对象的指针，你会使用 `characterPtr->attack()` 来调用成员函数。

总而言之，如果你确实实例化了一个对象，那就用点

如果你用的是指向对象的一个指针，那就用->

```
class Base {
public:
    virtual void show() {
        cout << "Base class show function called" << endl;
    }
    virtual ~Base() {}  // 虚析构函数
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class show function called" << endl;
    }
};

// 在主函数中
Base* bptr;
Derived d;
bptr = &d;

// 动态绑定
bptr->show();  // 输出: Derived class show function called
```

这个就是指针用->访问的例子

```
//关于upcasting的两种写法
Student s;
Person p = s;
Person* ptr = &s;//方法1
Person &ptr = s;//方法2
//ptr则被定义成了一个指向Student类对象s的一个指针，其本质还是属于Person类的
//自己可以调用Person类的成员函数或者对象
//也可以通过虚重写的方式来访问派生类中的成员函数
//但是要注意方法1的ptr本质上为一个指针，访问派生类成员函数需要通过->操作符
//方法2的ptr相当于是一个实例化对象，访问派生类成员函数可以用.操作符
```