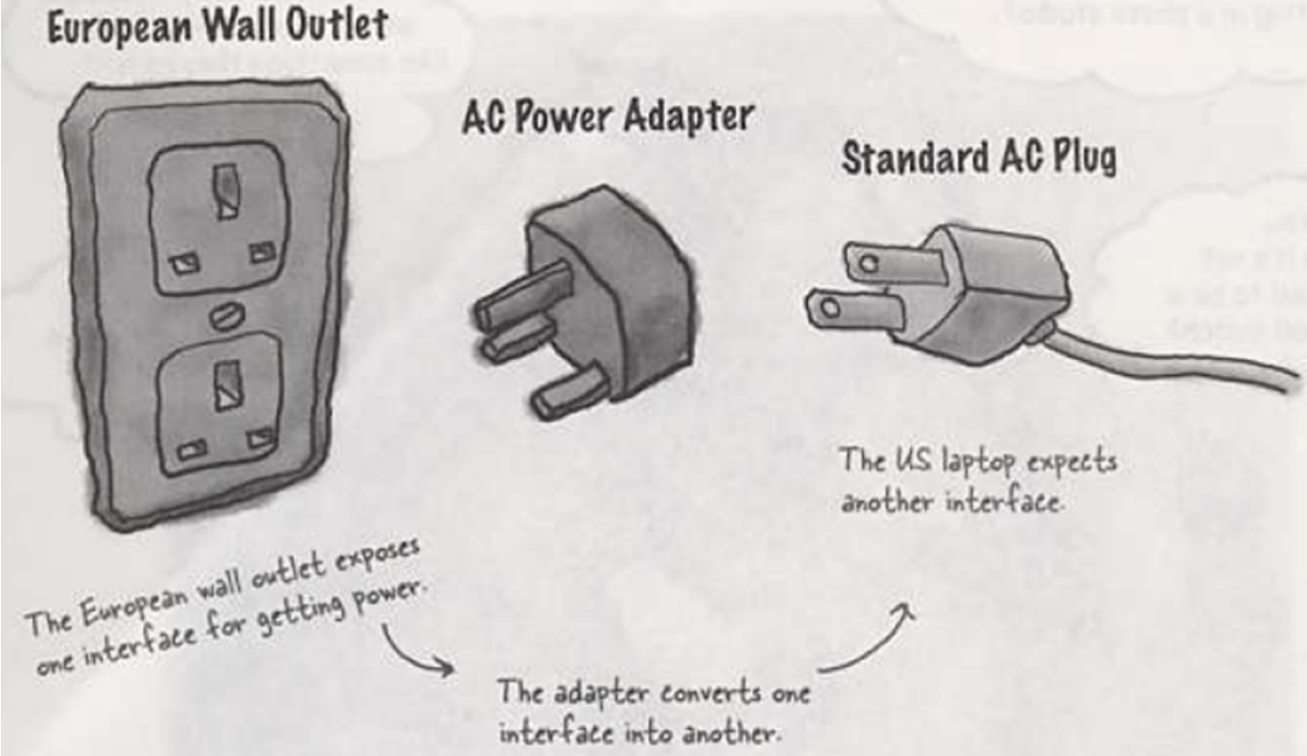
设计模式：结构型模式

1. 适配器模式(Adapter)

就像咱们生活中使用电源的转接口来给不同型号的电子设备来充电一样，适配器模式将一个类的接口转换成**客户希望的另一个接口**，从而使得原本由于接口不兼容而不能一起工作的类可以在统一的接口环境下工作。这个"转接口"就是我们要实现的适配器。



下面用一个例子进行说明。我们想要自己实现一个栈的数据结构，当然了，我们可以在类的内部使用自己开辟的数组int* 来模拟一个栈，但是我们知道stl中已经实现了vector数据结构，那么能不能利用vector来实现栈呢?

在这个例子里，我们的**目标**(Target, 用户所期待的接口)就是栈；**需要适配的类**（Adaptee）就是vector; 适配器（Adapter）就是我们要实现的类Vector2Stack.

实现方法1: 组合方式实现适配器Vector2Stack.

```
//堆栈基类
class Stack {
public:
virtual ~Stack() { }
    virtual     bool full() = 0;
    virtual     bool empty() = 0;
    virtual     void push(int i) = 0;
    virtual     void pop() = 0;
    virtual     int size() = 0;
    virtual     int top() = 0;
}
```

```
class Vector2Stack : public Stack{
private:
    std::vector<int> m_data; //将vector的接口组合进来实现具体功能
    const int m_size;
public:
    Vector2Stack(int size) : m_size(size) { }
    bool full() { return (int)m_data.size() >= m_size; }    //满栈检测
    bool empty() { return (int)m_data.size() == 0; }    //空栈检测
    void push(int i) { m_data.push_back(i); }       //入栈
    void pop() { if (!empty()) m_data.pop_back(); } //出栈
    int size() { return m_data.size(); }                //获取堆栈已用空间
    int top() {                                         //获取栈头内容
        if (!empty())
            return m_data[m_data.size()-1];
        else
            return INT_MIN;
    }
};

//在main函数中调用：Vector2Stack stack(10);
```

实现方法二：继承方式实现适配器Vector2Stack.

直接继承并使用std::vector的接口。

```
//直接继承vector并改造接口，采用私有继承可以使得外界只能接触到Vector2Stack中的接口

class Vector2Stack : private std::vector<int>, public Stack {
public:
    Vector2Stack(int size) : vector<int>(size) { }
    bool full() { return false; }
    bool empty() { return vector<int>::empty(); }
    void push(int i) { push_back(i); }
    void pop() { pop_back(); }
    int size() { return vector<int>::size(); }
    int top() { return back(); }
};
```

2. 代理/委托模式(Proxy)

C++中，指针是一个使用起来需要格外注意的东西，尤其是class中有指针，析构与释放是一个及其棘手的事情，稍有不慎就会程序错误或者内存泄漏。因此，我们要实现智能指针类，**能够包裹指针**，具有指针的各项功能，并能够**进行引用计数**，在计数为0时**自动释放指针空间**。使用适配器模式(Adapter)可以进行指针的封装，并对外提供指针各项功能的接口。但是，适配器模式仅仅只是**接口的转换**，其本身无法在提供接口的同时进行计数这样的**功能控制**。

这就需要用到代理/委托模式了。代理就相当于咱们生活中的中介。

在一些应用中，**直接访问对象**往往会带来诸多问题。例如，要访问的对象在远程的机器上，或者被访问对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问。直接访问会给使用者或者系统结构带来不必要的麻烦。因此，我们可以在被访问对象上加上一个**访问层**，将**复杂操作包裹在内部**不对外部类开放，仅**对外开放功能接口**，即可完成上述要求，这就是代理/委托模式。

对于上述智能指针的例子，我们可以实现一个智能指针类smart_ptr.

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

template <typename T> class SmartPtr; //提前声明智能指针模板类

//辅助指针，用于存储指针计数以及封装实际指针地址
template <typename T>
class U_Ptr {
private:
    friend class SmartPtr<T>;

    U_Ptr(T *ptr) :p(ptr), count(1) { }
    ~U_Ptr() { delete p; }

    int count;
    T *p; //数据存放地址
};
```

(U_ptr在实际调用的时候，就指向被代理指针的地址，还维护了该指针的引用计数)

```
template <typename T>
//智能指针
class SmartPtr {
private:
    U_Ptr<T> *rp;    //进行实际指针操作的辅助指针
public:
    SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
    //调动拷贝构造即增加引用计数
    SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) { ++rp->count; }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count; //赋值号后的指针引用加1
        if (--rp->count == 0) delete rp; //原内部指针引用减1
        rp = rhs.rp;     //代理新的指针
        return *this;
    }
    ~SmartPtr() { //只有引用次数为0才会释放
        if (--rp->count == 0) delete rp;
    }
    //对智能指针操作等同于对内部辅助指针操作
    T & operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};
```

smartptr实现了指针的引用计数统计、在引用计数为0的时候还完成了释放。

```
int main(int argc, char *argv[]) {
    //声明指针
    int *i = new int(2);
    //使用代理来包裹指针
    SmartPtr<int> ptr1(i);
    SmartPtr<int> ptr2(ptr1);
    SmartPtr<int> ptr3 = ptr2;
    //之后的操作均通过代理进行
    cout << *ptr1 << endl;
    *ptr1 = 20;
    cout << *ptr2 << endl;
    return 0;
}
```

```
2
20
```

3. 装饰器模式

我们想对基类增加一些功能，但是又不想直接修改基类代码（如果直接修改基类，其子类可能也都需要修改）。于是，我们可以创建一个装饰类，用来包装原有的基类，就可以在保持基类类方法完整性的前提下，提供额外的功能。