Object Oriented Programming & Object Oriented Design

> Inheritance 继承
>
> Composition 复合
>
> Delegation 委托

Composition & Adapter 模式 has-a

Definition: 某个类的里面有另外的类对象(class1 has class2,class3.....)

```
template<class T, class Sequence = deque<T>>
class queue
{
protected:
    Sequence c; // 底层容器
public:
    // 以下完全用 c 的操作函数完成
    size_type size() const {return c.size();}
    bool empty() const {return c.empty();}
    void push(const value_type& x) { c.push_back(x);}
    pass
}; // queue 中有 Sequence, 称为复合Composition
// queue *---> deque
// queue 称为 Adapter
```

> 内存角度
>
> 嵌套包含

Composition 下的构造和析构函数

联想盖房子:

> 构造由内而外调用： Container::Container(): /*默认构造*/Component() {}; // <---
>
> 析构由外而内调用： Container::~Container(){ ~Component() }; // --->

Delegation 委托. Composition by reference has-a-re

Delegation: class1 里有其他类的 指针 (class1 has a pointer of class2,class3...)

```
/*Handle and Body*/
class StringRep;
class String{ // Handle
public:
    String();
    String(const char* s);
    ~String();
private:
    StringRep* rep; // pimpl = point to implementation class
};
// String o---> StringRep
class StringRep { // body
    friend class String;
public:
    StringRep(const char* s);
    ~StringRep();
    int count; // 共享计数器
    char* rep;
};
// body 再怎么变动都无法影响 handle (编译防火墙)
```

Inheritance 继承 is-a

```
struct _List_node_base
{
    _List_node_base* _M_next;
    _List_node_base* _M_prev;
};

template<typename _Tp>
struct _List_node : public _List_node_base // public private protected
{
    _Tp _M_data;
};
```

> 父类数据被完整继承下来（子类的对象里有父类的成分 base part）
>
> 如果一个类有成为父类的可能就该把他的析构函数定义成 virtual 的
>
> 构造由内而外调用： Derived::Derived(): /*先父类默认构造*/Base() {then ...derived}; // <---
>
> 析构由外而内调用： Derived::~Derived(){ /*先子类*/ derived... then ~Base() }; // --->

Inheritance with virtual functions 虚函数

> 函数的继承继承的是 调用权 可以通过子类的对象调用父类的函数
>
> non-virtual 函数：你不希望 derived class 重新定义(override)
>
> virtual 函数：你希望 derived class 重新定义(override)
>
> pure virtual 纯虚函数：你希望 derived class 一定要 重新定义(override)，因为你对他没有默认定义

```
class Shape {
public:
    virtual void draw() const = 0; // pure virtual
    virtual void error(const std::string& msg); // impure virtual
    int objectID() const; // non-virtual
};

class Rectangle : public Shape {...};
class Ellipse : public Shape {...};
//
```

Template Method 模式(模板函数(直译))

> 将固定的函数方法写好，留下现在还无法决定的函数让子类定义，框架 如 MFC

```
// Application Framework
class CDocument {
public:
    void OnFileOpen();
    virtual Serialize();
}
void CDocument::OnFileOpen() // 2 // myDoc 的this被传进来
{
    ...
    Serialize(); // 3 // 通过this来调用
    ... // 5
}

class CMyDoc : public CDocument {
    virtual Serialize() {...} // 4 // this 就是 &myDoc
};
int main()
{
    CMyDoc myDoc;
    ...
    myDoc.OnFileOpen(); // 1     CDocument::OnFileOpen(&myDoc); // this = &myDoc
}



#include <iostream>
using namespace std;

class CDocument {
```

```
public:
    void OnFileOpen()
    {
        // 这是个算法，每个 cout 输出代表一个实际动作
        cout << "dialog ..." << endl;
        cout << "check file status..." << endl;
        cout << "open file..." << endl;
        Serialize();
        cout << "close file..." << endl;
        cout << "update all views..." << endl;
    }

    virtual void Serialize() { }; // impure virtual, derived class must override this method
};

class CMyDoc : public CDocument {
public:
    virtual void Serialize() {
        // 自有应用程序本身才知道如何读取自己的文件格式
        cout << "CMyDoc::Serialize called" << endl;
    }
};

int main()
{
    CMyDoc myDoc;
    myDoc.OnFileOpen();
}
```

Inheritance + Composition 关系下的构造和析构(自己观察)

```
class Base {
    public:
        Base()
        {
            cout << "Base's constructor called" << endl;
        }
        virtual ~Base(){
            cout << "Base's destructor called" << endl;
        }
};
class Component {
    public:
        Component()
        {
            cout << "Component's constructor called" << endl;
        }
        ~Component()
        {
            cout << "Component's destructor called" << endl;
        }
};
```

```
class Derived : public Base {
    public:
        Derived()
        {
            cout << "Derived's constructor called" << endl;
        }
        ~Derived()
        {
            cout << "Derived's destructor called" << endl;
        }
    private:
        Component c;
};

int main()
{
    Derived b;
    return 0;
}
Result:
    Base's constructor called
    Component's constructor called
    Derived's constructor called
    Derived's destructor called
    Component's destructor called
    Base's destructor called
```

构造： Base --> Component --> Derived

析构： Derived --> Component --> Base

Delegation(委托) + Inheritance(继承)

> 一个文件，多个窗口试图

```
class Subject { // subject O--> Observer
    int m_value;
    vector<Observer> m_views; // 容器
public:
    void attach(Observer* obs) // 注册
    {
        m_views.push_back(obs);
    }
    void set_val(int value)
    {
        m_value = value;
        notify();
    }
    void notify() // 通知所有观察者更新数据
    {
        for(int i = 0; i < m_views.size();++i)
        {
```

```
            m_views[i] -> update(this,m_value);
        }
    }
};

class Observer { // 窗口类
    public:
        virtual void update(Subject* sub,int value) = 0; // pure virtual
}

class OBSDocument : public Observer {
    public:
        virtual void update(Subject* sub,int value){
            // 具体实现
        }
};
```

几种设计模式

单例模式 Singleton

```
class A {
public:
    static A& getInstance(); // 唯一接口
    setup() { pass }
private:
    // 构造函数为private,外界不可实例化
    A();
    A(const A& rhs);
    //Static A a; // 类内唯一实例化一个对象
};
// 在静态函数中实例化唯一对象，防止浪费内存空间
A& A::getInstance()
{
    static A a;
    return a;
}
int main()
{
    // 外界只能通过A::getInstance()函数来得到唯一的一个A的实例
    A::getInstance().setup();
}
```

Composite 模式

> A，C 继承至 B, C 中有 B 的指针，则 C 中有 A

```
class Component
{
    int value;
public:
    Component(int val) : value(val) {}
    virtual void add(Component*) {}
}
class Primitive : Component
{
public:
    Primitive(int val) : Component(val) {}
}

class Composite : public Component {
    vector<Component*> c;
public:
    Composite(int val) : Component(val) {}
    void add(Component* elem)
    {
        c.push_back(elem);
    }
}
```

Prototype 模式(原型)