

## 基于对象(Object Based)的程序设计

不带有指针成员变量的类——以复数类 `complex` 为例

头文件的结构

访问级别

函数设计

内联函数

构造函数

常量成员函数

参数的值传递和引用传递

返回值的值传递和引用传递

友元

操作符重载

在类内声明 `public` 函数重载 `+=`

在类外声明或函数重载 `+`

在类外声明函数重载 `<<`

总结:在编写类的时候应该注意的5件事

带有指针成员变量的类——以字符串类 `string` 为例

3个特殊函数:拷贝构造函数、拷贝赋值函数和析构函数

构造函数和析构函数

拷贝构造函数和拷贝赋值函数

堆栈与内存管理

堆栈及对象的生命周期

`new` 和 `delete` 过程中的内存分配

static成员

## 面向对象(Object Oriented)的程序设计——类之间的关系

类之间的关系

复合(composition)

委托(aggregation;composition by reference)

继承(extension)

虚函数

面向对象设计范例

使用委托+继承实现Observer模式

使用委托+继承实现Composite模式

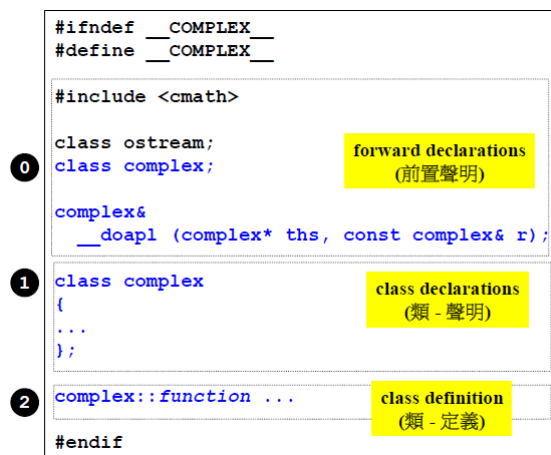
使用委托+继承实现Prototype模式

# 基于对象(Object Based)的程序设计

## 不带有指针成员变量的类——以复数类 `complex` 为例

### 头文件的结构

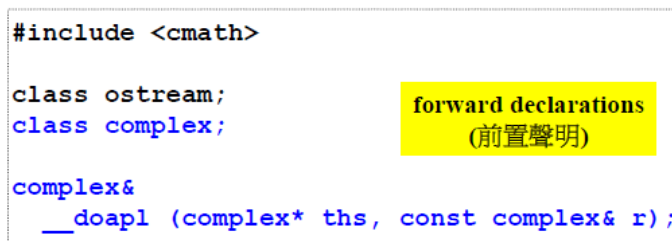
头文件 `complex.h` 的结构如下,主要分为4个部分



1. 防卫式声明,防止头文件被重复包含:



2. 前置声明: 声明头文件中用到的类和函数



3. 类声明: 声明类的函数和变量,部分简单的函数可以在这一部分加以实现



4. 类定义: 实现前面声明的函数



## 访问级别

C++的访问级别有3种,这个程序中展示了两种:

访问级别	意义
private	只能被本类的函数访问
protected	能被本类的函数和子类的函数访问
public	可以被所有函数访问

```
class complex
{
public:
    complex (double r = 0, double i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    double real () const { return re; }
    double imag () const { return im; }
private:
    double re, im;

    friend complex& __doapl (complex*, const complex&);
};
```

**X**

```
{
    complex c1(2,1);
    cout << c1.re;
    cout << c1.im;
}
```

**O**

```
{
    complex c1(2,1);
    cout << c1.real();
    cout << c1.imag();
}
```

类的声明内可以交叉定义多个不同级别的访问控制块:

```
1  class complex
2  {
3  public:
4      // public访问控制块1...
5
6  private:
7      // private访问控制块1...
8
9  public:
10     // public访问控制块2...
11
12 }
```

## 函数设计

### 内联函数

在类声明内定义的函数,自动成为 `inline` 函数;在类声明外定义的函数,需要加上 `inline` 关键字才能成为 `inline` 函数.

类声明内定义的函数,自动成为 `inline` 函数

```
class complex
{
public:
    complex (double r = 0, double i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    double real () const { return re; }
    double imag () const { return im; }
private:
    double re, im;

    friend complex& __doapl (complex*, const complex&);
};

inline double
imag(const complex& x)
{
    return x.imag ();
}
```

函数若在 class body  
内定义完成,便自动  
成为 inline 候选人

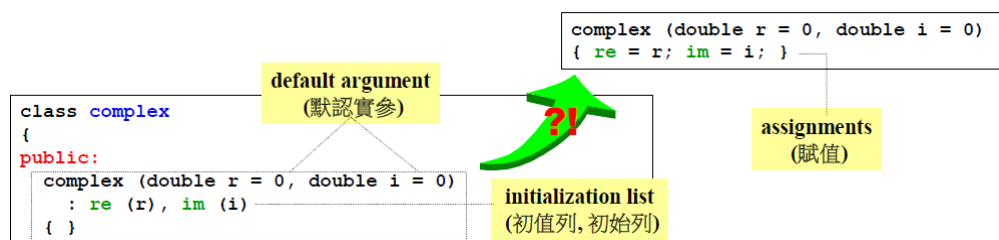
类声明外定义的函数,需要加上 `inline` 关键字才能成为 `inline` 函数

```
inline double
imag(const complex& x)
{
    return x.imag ();
}
```

`inline` 只是编程者给编译器的一个建议,在编译时未必会真正被编译为 `inline` 函数.因此如果函数足够简单,我们就把它声明为 `inline` 就好了.

## 构造函数

与其他语言类似,C++的构造函数也可以有默认实参.C++构造函数的特殊之处在于列表初始化 (initialization list).



上面两种构造函数的效果是一样的,但是使用列表初始化的效率更高,应尽量使用列表初始化.

默认实参使得类的使用者更灵活地创建对象.

```
1 complex c1(2,1);           // complex(2, 1)
2 complex c2;                // complex(0, 0)
3 complex* p = new complex(4); // complex(4, 0)
```

默认实参不应造成歧义,下图定义的两个构造函数会造成歧义,使得编译失败:

```
class complex
{
public:
    complex (double r = 0, double i = 0)
        : re (r), im (i)
    { }
    complex () : re(0), im(0) { }    ?!
```

```
1 complex c1(2, 1);    // 这一行编译通过
2 complex c2;          // 这一行编译失败: error: call of overloaded 'complex()' is
                        // ambiguous
```

## 常量成员函数

若成员函数中不改变成员变量,应加以 `const` 修饰

```
class complex
{
public:
    double real () const { return re; }
    double imag () const { return im; }
```

若这类函数不加以 `const` 修饰,则常量对象将不能调用这些函数:

```
1 const complex c(2, 1);    // 定义常量变量
2 c.real();                 // 若 real() 函数不加以const修饰,则编译时会报错:
                        // error: passing 'const complex' as 'this' argument
```

## 参数的值传递和引用传递

```

class complex
{
public:
    complex (double r = 0, double i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    double real () const { return re; }
    double imag () const { return im; }
private:
    double re, im;

    friend complex& __doapl (complex*, const complex&);
};

ostream&
operator << (ostream& os, const complex& x)
{
    return os << '(' << real (x) << ', '
               << imag (x) << ')';
}

{
    complex c1(2,1);
    complex c2;

    c2 += c1;
    cout << c2;
}

```

为了提高效率,使用引用传递参数,避免了参数的复制.若不希望函数体内对输入参数进行修改,应使用 `const` 修饰输入参数

函数的参数应尽量使用引用传递.

## 返回值的值传递和引用传递

```

class complex
{
public:
    complex (double r = 0, double i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    double real () const { return re; }
    double imag () const { return im; }
private:
    double re, im;

    friend complex& __doapl (complex*, const complex&);
};

ostream&
operator << (ostream& os, const complex& x)
{
    return os << '(' << real (x) << ', '
               << imag (x) << ')';
}

{
    complex c1(2,1);
    complex c2;

    cout << c1;
    cout << c2 << c1;
}

```

为提高效率,若函数的返回值是原本就存在的对象,则应以引用形式返回.

若函数的返回值是临时变量,则只能通过值传递返回.

```

inline complex& __doapl (complex *ths, const
complex & r)
{
    ths->re+=r.re;
    ths->im+=r.im;
    return *ths;
}

```

## 友元

友元函数不受访问级别的控制,可以自由访问对象的所有成员.

```

class complex
{
private:
    double re, im;

    friend complex& __doapl (complex*, const complex&);
};

inline complex&
__doapl (complex* ths, const complex& r)
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}

```

自由取得 friend 的 private 成员

同一类的各个对象互为友元,因此在类定义内可以访问其他对象的私有变量

```
class complex
{
public:
    int func(const complex& param)
    { return param.re + param.im; }

private:
    double re, im;
};
```

```
1 | complex c1, c2;
2 | c2.func(c1);           // 因为c1和c2互为友元,因此c2可以在func()函数内调用c1的私有变量
```

## 操作符重载

在C++中的操作符重载有两种形式,一种是在类内声明 `public` 函数实现操作符重载(这种情况下,操作符是作用在左操作数上的);另一种是在类外声明全局函数实现操作符重载.

例如对于如下语句,有两种方式都可以实现操作符 `+` 的重载.

```
1 | complex c1;
2 | c1 + 2;           // 需要重载操作符 +
```

1. 在类内声明 `public` 函数 `complex::operator += (int)`
2. 在类外声明全局函数 `complex operator + (const complex&, double)`

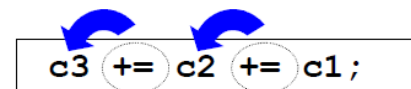
这两种方式均可以实现操作符重载,为便于调用该类的用户使用,不同的操作符使用不同的方式进行重载

### 在类内声明 `public` 函数重载 `+=`

重载 `+=` 函数的 `complex::operator += (const complex& r)` 的输入参数和输出参数均使用引用传值,输入参数在函数中不应被改动,因此使用 `const` 修饰.

输出参数类型为 `complex&`,这是为了支持将多个 `+=` 操作符串联起来.若返回参数类型设为 `void`,也支持形如 `c2+=c1` 的运算,但不支持形如 `c3+=c2+=c1` 的运算.

```
inline complex&
complex::operator += (const complex& r)
{
    return __doapl(this, r);
}
```



函数体内调用友元函数 `__doapl(complex *, const complex &)` 实现功能,其第一个参数接收成员函数内隐含的 `this` 指针,其内容在函数中会被改变;第二个参数接收重载函数的参数,该参数在函数中不会被改变,以 `const` 修饰.

```
inline complex&
__doapl(complex* ths, const complex& r)
{
    ths->re += r.re;
    ths->im += r.im;
    return *ths;
}

inline complex&
complex::operator += (const complex& r)
{
    return __doapl (this, r);
}
```

第一参数将会被改动  
第二参数不会被改动

返回的是一个val

```
{
    complex c1(2,1);
    complex c2(5);

    c2 += c1;
}
```

传递者无需知道接收者以什么方式接受

```
inline complex&
complex::operator += (this, const complex& r)
{
    return __doapl (this, r);
}
```

从这个例子中也可以看出使用引用传递参数和返回值的好处在于传送者无需知道接收者是否以引用形式接收,只需要和值传递一样写代码就行,不需要改动.

## 在类外声明或函数重载 +

考虑到 + 操作符有三种可能的用法如下:

```
1  complex c1(2,1);
2  complex c2;
3
4  c2 = c1 + c2;    // 用法1: complex + complex
5  c2 = c1 + 5;     // 用法2: complex + double
6  c2 = 7 + c1;     // 用法3: double + complex
```

因为重载操作符的成员函数是作用在左操作数上的,若使用类内声明 public 函数重载操作符的方法,就不能支持第3种用法了.因此使用在类外声明函数重载 + 运算符.

只要需要生成一个新的结果,就需要一个临时对象

```
inline complex
operator + (const complex& x, const complex& y)
{
    return complex (real (x) + real (y),
                    imag (x) + imag (y));
}

inline complex
operator + (const complex& x, double y)
{
    return complex (real (x) + y, imag (x));
}

inline complex
operator + (double x, const complex& y)
{
    return complex (x + real (y), imag (y));
}
```

临时对象

都是临时对象:  
complex()  
complex(3, 2)

这3个函数返回的是局部对象(local object),在退出函数时对象就会被销毁,因此不能使用引用传递返回值.

## 在类外声明函数重载 <<

与重载 + 的考虑方法类似, << 操作符通常的使用方式是 cout<<c1 而非 c1<<cout, 因此不能使用成员函数重载 << 运算符.

考虑到形如 cout<<c1<<c2<<c3 的级联用法,重载函数的返回值为 ostream& 而非 void.

```
ostream&
operator << (ostream& os, const complex& x)
{
    return os << '(' << real (x) << ', '
        << imag (x) << ')';
}
```



```
void
operator << (ostream& os,
            const complex& x)
{
    return os << '(' << real (x) << ', '
        << imag (x) << ')';
}
```

```
{
    complex c1(2,1);
    cout << conj(c1);
    cout << c1 << conj(c1);
}
```

```
{
    complex c1(2,1);
    cout << conj(c1);
    cout << c1 << conj(c1);
}
```

## 总结:在编写类的时候应该注意的5件事

在编写类的时候应该注意的5件事,通过这5件事可以看出你写的代码是否大气:

1. 构造函数中使用列表初始化(initialization list)为成员变量赋值.
2. 常量成员函数使用 `const` 修饰.
3. 参数的传递尽量考虑使用引用传递,若函数体内不改变传入的参数,应加以 `const` 修饰.
4. 返回值若非局部变量,其传递尽量考虑使用引用传递,
5. 数据放入 `private` 中,大部分函数放入 `public` 中.

## 带有指针成员变量的类——以字符串类 String 为例

`String` 类定义在头文件 `string.h` 中,其结构与 `complex.h` 类似.

只要类带有指针,必须有拷贝构造和赋值函数

```
#ifndef MYSTRING
#define MYSTRING

class String
{
    ...
};

String::function(...) ...
Global-function(...) ...

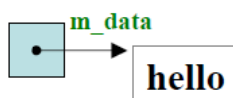
#endif
```

```
int main()
{
    String s1(),
    String s2("hello");

    String s3(s1);
    cout << s3 << endl;
    s3 = s2;
    cout << s3 << endl;
}
```

类声明如下,使用指针成员变量 `m_data` 管理 `String` 类中的字符串数据.

```
class String
{
public:
    String(const char* cstr = 0);
    String(const String& str);
    String& operator=(const String& str);
    ~String();
    char* get_c_str() const { return m_data; }
private:
    char* m_data;
};
```



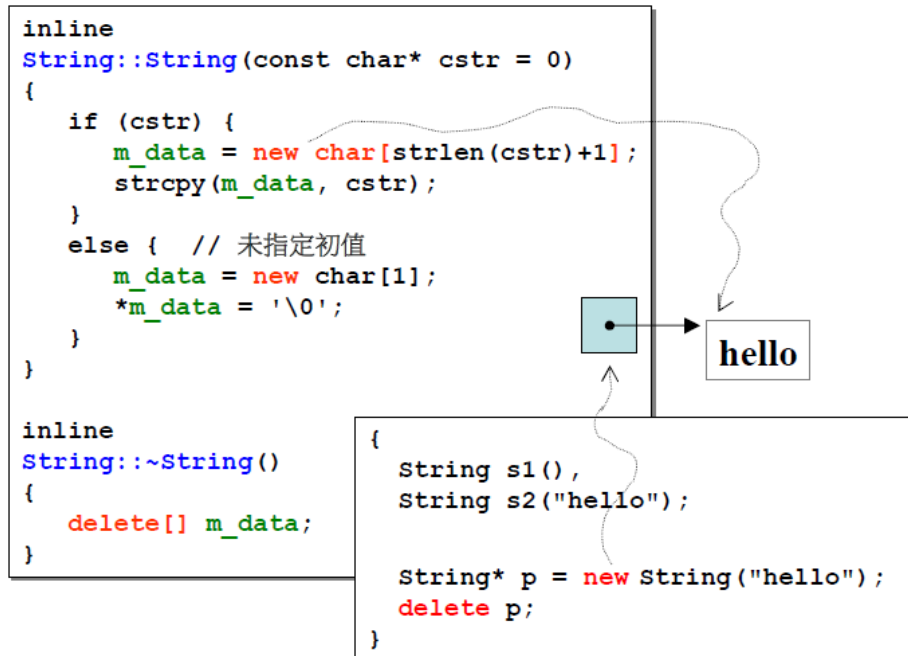


## 3个特殊函数:拷贝构造函数、拷贝赋值函数和析构函数

对于不带有指针的类,这3个函数可以使用编译器默认为我们生成的版本;但是编写带有指针的类时就有必要定义这3个特殊函数.

### 构造函数和析构函数

构造函数和析构函数中执行数据的深拷贝和释放.



值得注意的是使用 delete[] 操作符释放数组内存,若直接使用 delete 操作符释放数组内存虽然能够通过编译,但有可能产生内存泄漏.

### 拷贝构造函数和拷贝赋值函数

拷贝构造函数和拷贝赋值函数函数的使用场景不同,下面程序的拷贝3虽然使用了 = 赋值,但是因为是在初始化过程中使用的,所以调用的是拷贝构造函数.

```
1 String s1 = "hello";
2 String s2(s1);      // 拷贝1: 调用拷贝构造函数
3 String s3;
4 s3 = s1;            // 拷贝2: 调用拷贝赋值函数
5 String s4 = s1;     // 拷贝3: 调用拷贝构造函数
```

如果没有赋值函数,那么  
string a;  
string b;  
b=a;  
之后, b就会指向a的空间,并没有把a空间的内容复制到b中, b的空间就会泄露

拷贝构造函数的实现较为简单,直接调用友元对象的数据指针进行拷贝即可.

```
inline
String::String(const String& str)
{
    m_data = new char[ strlen(str.m_data) + 1 ];
    strcpy(m_data, str.m_data);
}

{
    String s1("hello ");
    String s2(s1);
    // String s2 = s1;
}
```

直接取另一个 object 的 private data.  
(兄弟之间互为 friend)

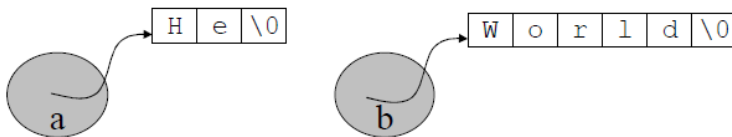
拷贝赋值函数中要检测自我赋值,这不仅是效率考虑,也是为了防止出现bug.

```
inline
String& String::operator=(const String& str)
{
    if (this == &str) 检测自我赋值 (self assignment)
        return *this;

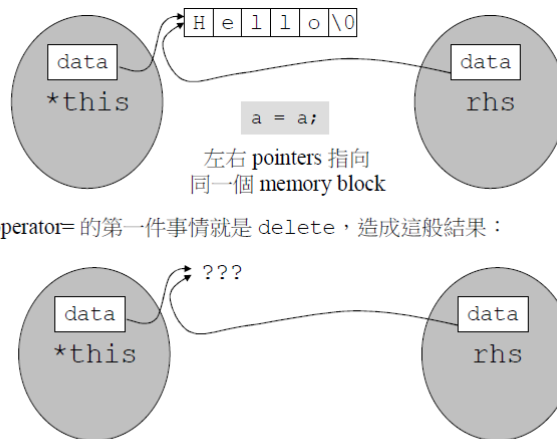
    1 delete[] m_data;
    2 m_data = new char[ strlen(str.m_data) + 1 ];
    3 strcpy(m_data, str.m_data);
    return *this;
}
```

```
{
    String s1("hello ");
    String s2(s1);
    s2 = s1;
}
```

三步走：  
先把自己的空间销毁掉  
重新分配和右边的空间一样大的内存  
进行拷贝



若在拷贝赋值函数中不检测自我赋值,在第2步中会出现bug.



前述 operator= 的第一件事情就是 delete, 造成这般结果：

然後，當企圖存取（訪問）rhs，產生不確定行為（undefined behavior）

## 堆栈与内存管理

### 堆栈及对象的生命周期

栈(stack),是存在于某作用域(scope)的一块内存空间.例如当你调用函数,函数本身就会形成一个stack用来防治它所接收的参数以及返回地址.在函数本体内声明的任何变量,其所使用的内存块都取自上述stack.

堆(heap),是指由操作系统提供的一块global内存空间,程序可动态分配从其中获得若干区块.

```
class Complex { ... };
...
{
    Complex c1(1,2);
    Complex* p = new Complex(3);
}
```

c1 所佔用的空間來自 stack

Complex(3) 是個臨時對象，其所佔用的空間乃是以 new 自 heap 動態分配而得，並由 p 指向。

1. stack object的生命周期:

```

1 | class Complex { ... };
2 | // ...
3 |
4 | {
5 |     Complex c1(1,2);
6 | }

```

程序中 `c1` 就是stack object,其生命周期在作用域(大括号)结束之际结束.这种作用域内的对象又称为 auto object,因为它会被自动清理.

## 2. static object的生命周期

```

1 | class Complex { ... };
2 | // ...
3 |
4 | {
5 |     static Complex c2(1,2);
6 | }

```

程序中 `c2` 就是static object,其生命周期在作用域(大括号)结束之后仍然存在,直到整个程序结束.

## 3. global object的生命周期

```

1 | class Complex { ... };
2 | // ...
3 |
4 | Complex c3(1,2);
5 |
6 | int main()
7 | {
8 |     ...
9 | }

```

程序中 `c3` 就是global object,其生命在在在整个程序结束之后才结束,也可以将其视为一种static object,其作用域是整个程序.

## 4. heap object的生命周期

```

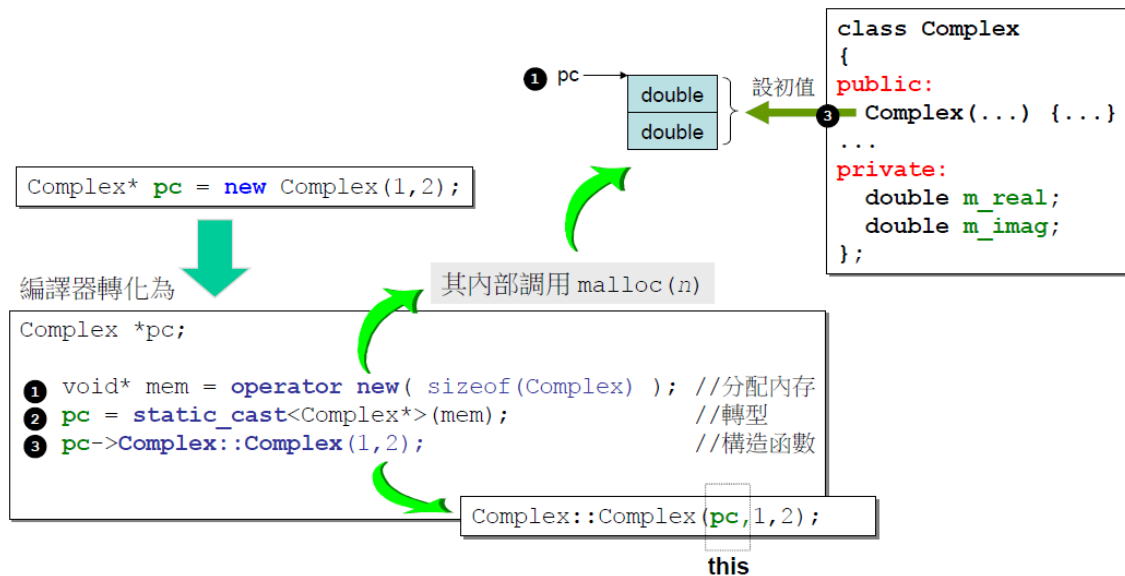
1 | class Complex { ... };
2 | // ...
3 |
4 | {
5 |     Complex* p = new Complex;
6 |     // ...
7 |     delete p;
8 | }

```

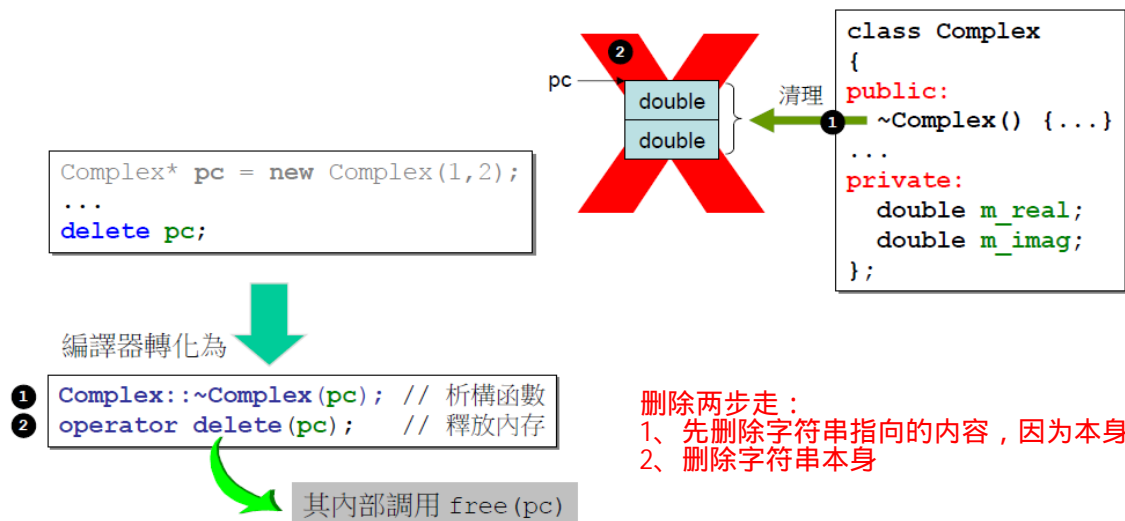
程序中 `p` 指向的对象就是heap object,其生命周期在它被deleted之际结束.若推出作用域时忘记 delete指针 `p` 则会发生内存泄漏,即 `p` 所指向的heap object 仍然存在,但指针 `p` 的生命周期却结束了,作用域之外再也无法操作 `p` 指向的heap object.

## new 和 delete 过程中的内存分配

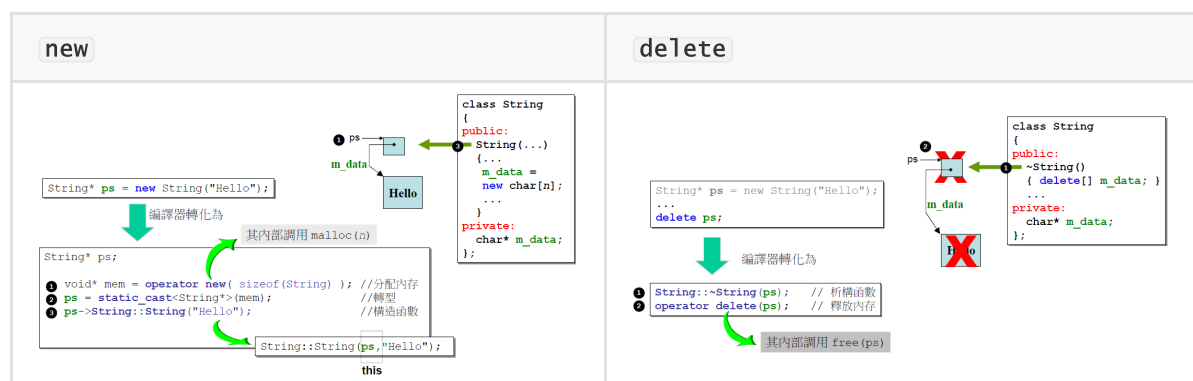
- new 操作先分配内存,再调用构造函数.



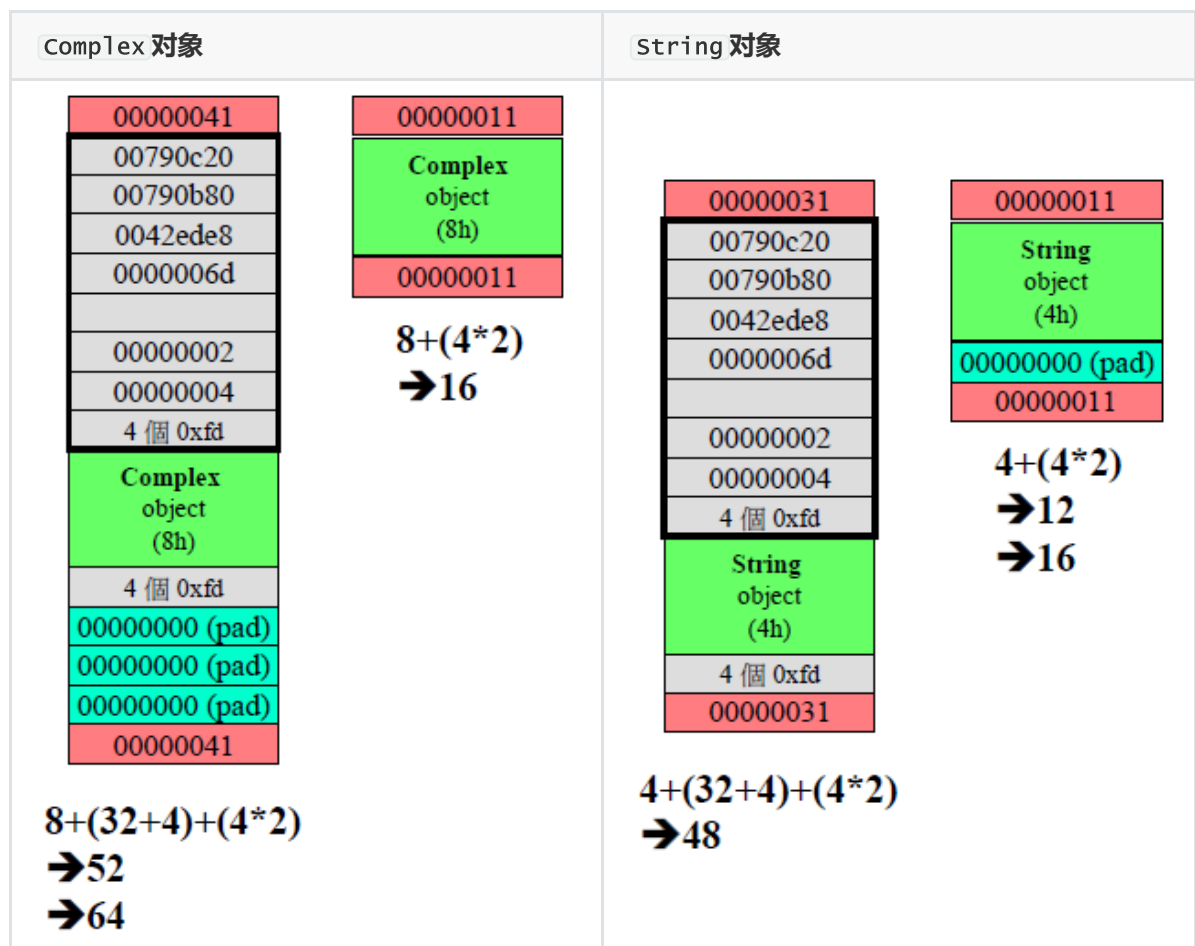
- delete 操作先调用析构函数,再释放内存.



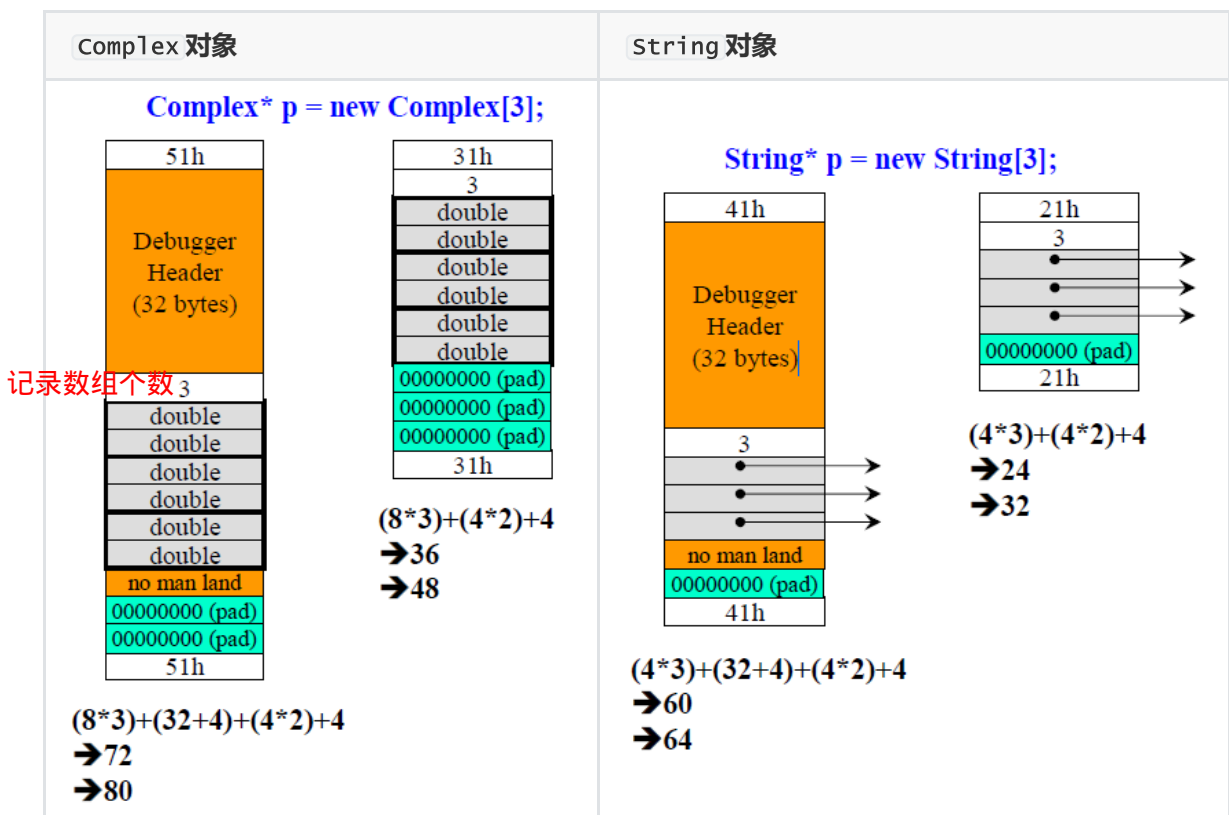
对于带有指针的 String 类, new 操作和 delete 操作的示意图如下:



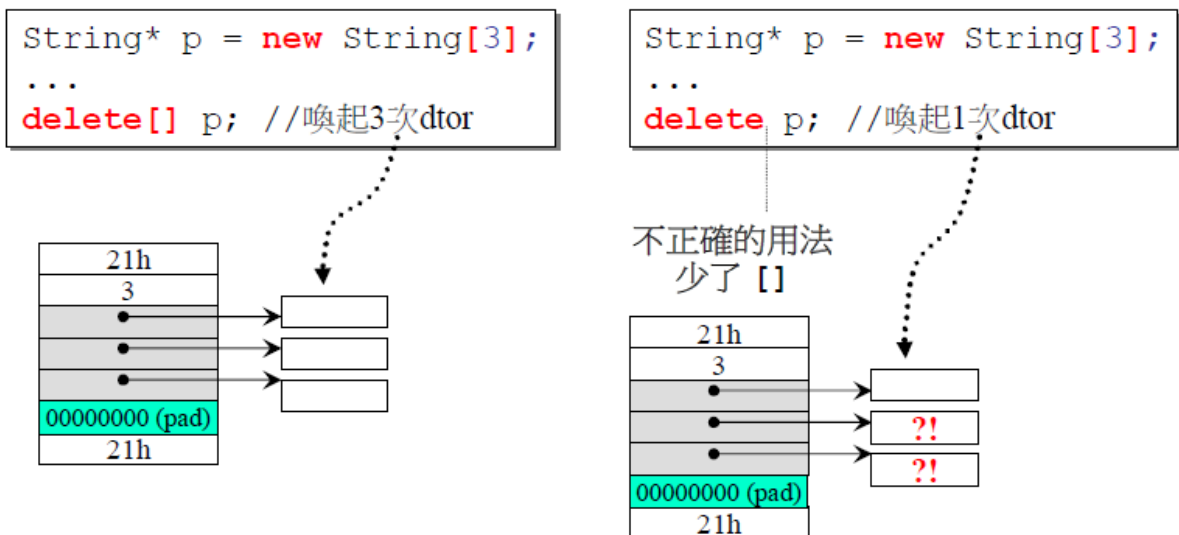
VC中对象在 debug 模式和 release 模式下的内存分布如下图所示,变量在内存中所占字节数必须被补齐为16的倍数,红色代表 cookie 保存内存块的大小,其最低位的 1 和 0 分别表示内存是否被回收。  
回收的边界



数组中的元素是连续的,数组头部4个字节记录了数组长度:



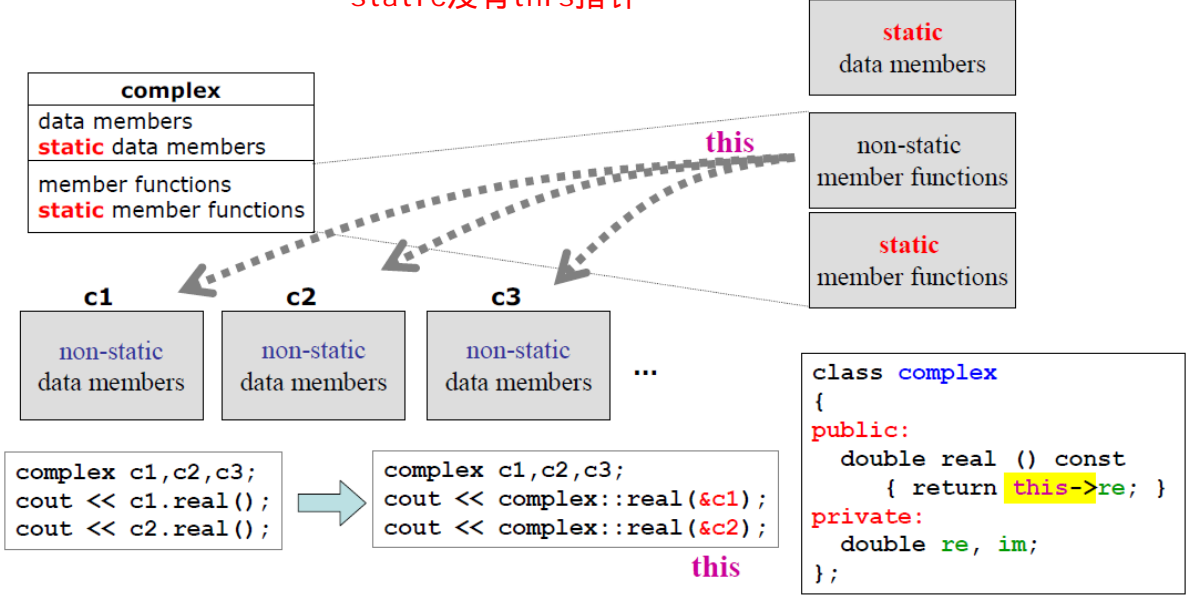
根据数组在内存中的状态,自然可以理解为什么 new[] 和 delete[] 应该配对使用了: delete 操作符仅会调用一次析构函数,而 delete[] 操作符依次对每个元素调用析构函数.对于 String 这样带有指针的类,若将 delete[] 误用为 delete 会引起内存泄漏.



## static成员

对于类来说, `non-static` 成员变量每个对象均存在一份, `static` 成员变量、 `non-static` 和 `static` 成员函数在内存中仅存在一份. 其中 `non-static` 成员函数通过指定 `this` 指针获得函数的调用权, 而 `non-static` 函数不需要 `this` 指针即可调用.

static没有this指针



`static` 成员函数可以通过对象调用,也可以通过类名调用.

```
1 class Account {
2 public:
3     static double m_rate;
4     static void set_rate(const double& x) { m_rate = x; }
5 };
6 double Account::m_rate = 8.0;
7
8 int main() {
9     Account::set_rate(5.0); 通过类调用
10    Account a;
11    a.set_rate(7.0);
12 }
```

`static` 成员变量需要在类声明体外进行初始化.

# 面向对象(Object Oriented)的程序设计——类之间的关系

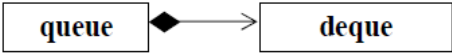
类之间的关系有**复合**(composition)、**委托**(aggregation)和**继承**(extension)3种.

## 类之间的关系

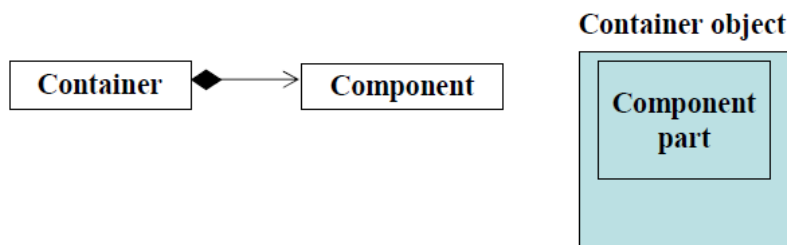
### 复合(composition) 有了外部的就有内部

复合表示一种 has-a 的关系,STL中 queue 的实现就使用了复合关系.这种结构也被称为**adapter模式**.

```
template <class T>
class queue {
    ...
protected:
    deque<T> c;      // 底層容器
public:
    // 以下完全利用 c 的操作函数完成
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    reference back() { return c.back(); }
    //
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```



复合关系下构造由内而外,析构由外而内:



构造由内而外

**Container** 的构造函数首先调用 **Component** 的 default 构造函数，  
然后才执行自己。  
编译器做

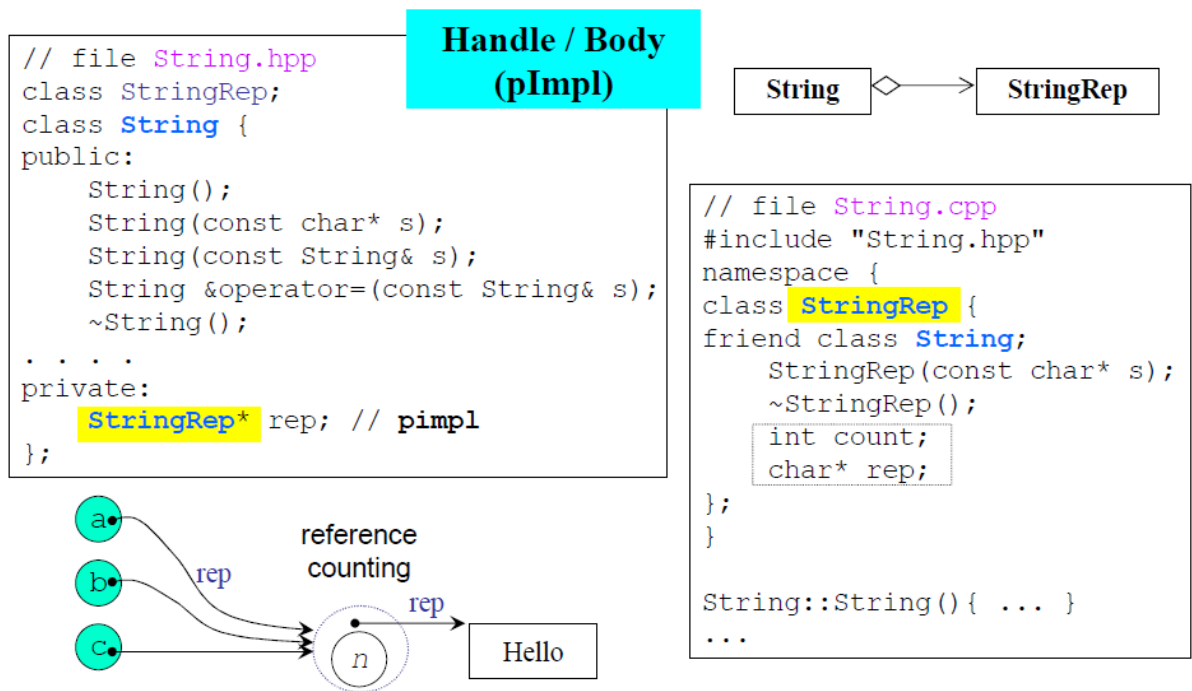
```
Container::Container(...) : Component() { ... };
```

析构由外而内

**Container** 的析构函数首先执行自己，然后才调用 **Component** 的析构函数。

```
Container::~Container(...) { ... ~Component() };
```

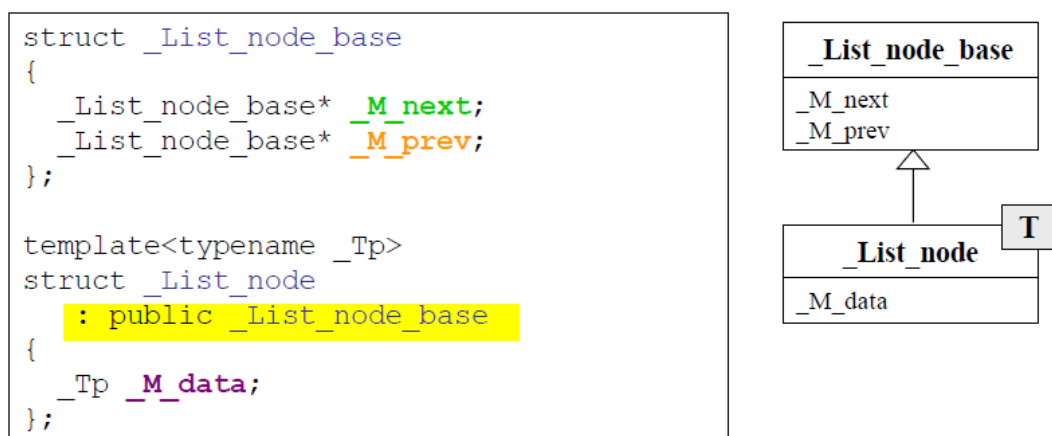
### 委托(aggregation;composition by reference) 有了外部不一定有内部



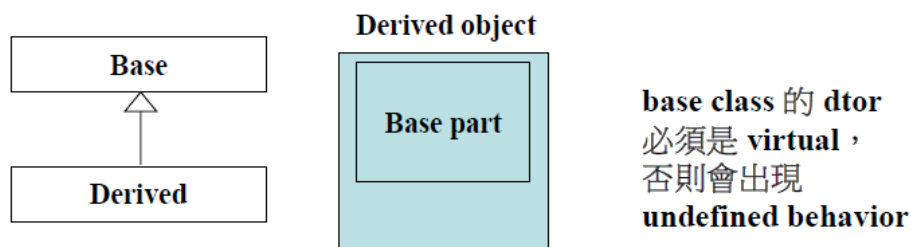
委托将类的定义与类的实现分隔开来,也被称为**编译防火墙**.

## 继承(extension)

继承表示一种 is-a 的关系, STL 中 `_List_node` 的实现就使用了继承关系.



继承关系下构造由内而外,析构由外而内:



### 构造由内而外

**Derived** 的构造函数首先调用 **Base** 的 **default** 构造函数, 然後才执行自己。

```
Derived::Derived(...) : Base() { ... };
```

### 析构由外而内

**Derived** 的析构函数首先执行自己, 然後才调用 **Base** 的析构函数。

```
Derived::~Derived(...) { ... ~Base() };
```



## 虚函数

成员函数有3种:非虚函数、虚函数和纯虚函数:

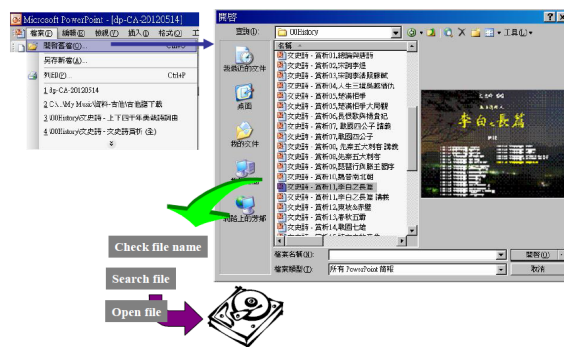
- 非虚(non-virtual)函数: 不希望子类重新定义(override)的函数.
- 虚(virtual)函数: 子类可以重新定义(override)的函数,且有默认定义.
- 纯虚(pure virtual)函数: 子类必须重新定义(override)的函数,没有默认定义.

```
class Shape {
public:
    virtual void draw( ) const = 0;
    virtual void error(const std::string& msg);
    int objectID( ) const;
    ...
};

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

pure virtual  
impure virtual  
non-virtual

使用虚函数实现框架: 框架的作者想要实现一般的文件处理类,由框架的使用者定义具体的文件处理过程,则可以用虚函数来实现.



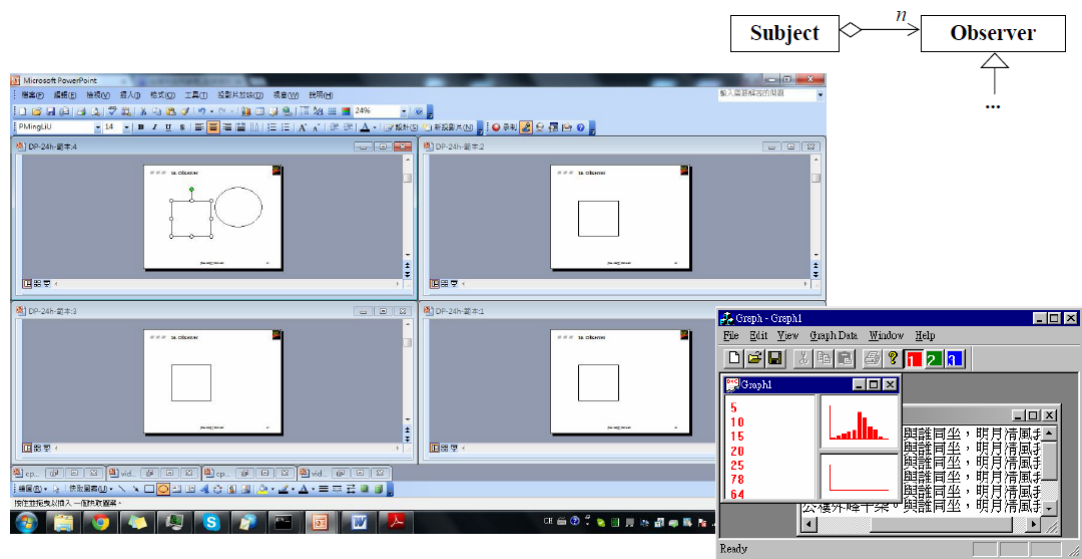
将框架中父类 CDocument 的 Serialize() 函数设为虚函数,由框架使用者编写的子类 CMyDoc 定义具体的文件处理过程,流程示意图和代码如下:



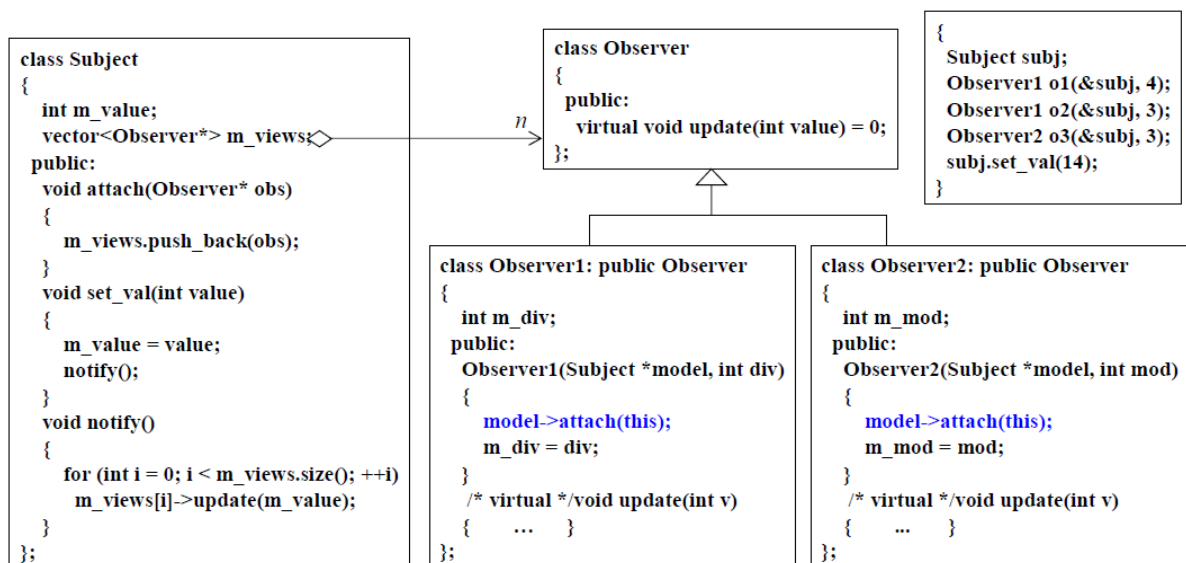
## 面向对象设计范例

### 使用委托+继承实现Observer模式

使用Observer模式实现多个窗口订阅同一份内容并保持实时更新

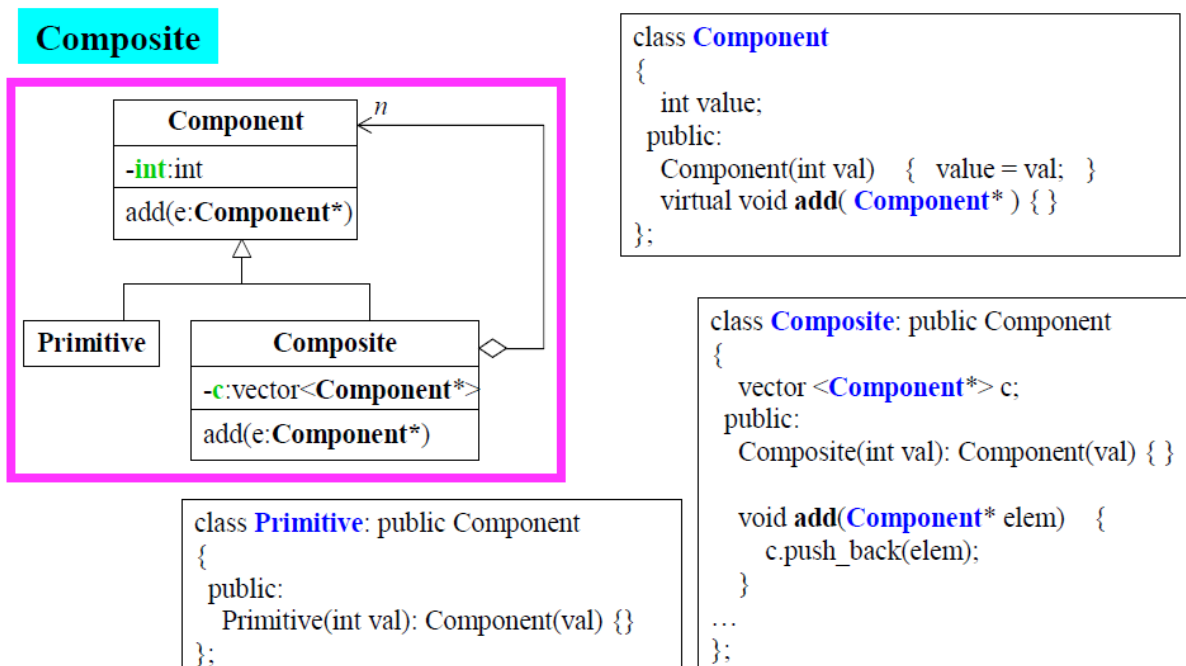


类结构图如下:



## 使用委托+继承实现Composite模式

使用Composite模式实现多态,类结构图如下



# 使用委托+继承实现Prototype模式

Prototype模式示意图如下:

