

# C++面向对象程序设计（上）--侯捷

笔记本： A01.Tem

创建时间： 2020/2/12 10:49

更新时间： 2020/2/27 17:00

作者： 1023442559@qq.com

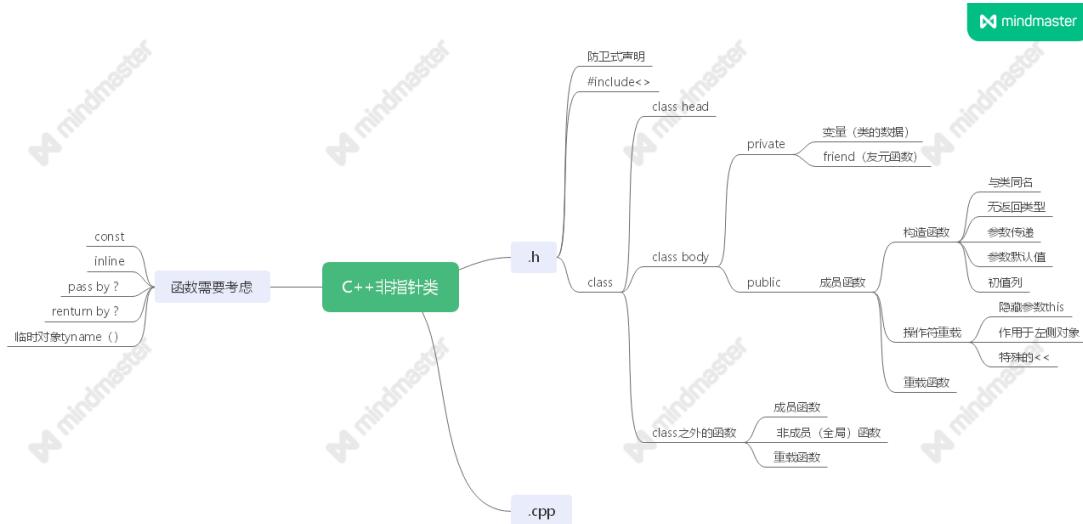
URL： <https://list.yinxiang.com/markdown/eef42447-db3f-48ee-827b-1bb34c03eb83.p...>

---

- [C++编程简介](#)
  - [目标](#)
  - [C++介绍](#)
  - [C vs C++ 关于数据和函数](#)
  - [Object Based \(基于对象\) vs Object Oriented \(面向对象\)](#)
  - [C++ programs 代码基本形式](#)
  - [Output C++ vs C](#)
- [头文件与类的声明](#)
  - [Header \(头文件\) 中的防卫式声明](#)
  - [Header \(头文件\) 的布局](#)
  - [class的声明 \(declaration\)](#)
  - [Class template \(模板\) 简介](#)
- [构造函数](#)
  - [inline \(内联\) 函数](#)
  - [access level \(访问级别\)](#)
  - [constructor \(ctor 构造函数\)](#)
  - [ctor \(构造函数\) 可以有多个overloading \(重载\)](#)
  - [constructor \(ctor 构造函数\) 被放在private区](#)
  - [const member function \(常成员函数\)](#)
- [参数传递与返回值](#)
  - [参数传递: pass by value vs pass by reference \(to const\)](#)
  - [返回值传递: return by value vs return by reference \(to const\)](#)
  - [friend \(友元\)](#)
  - [相同class的各个objects互为friends \(友元\)](#)
  - [class body 外的各种定义 \(definitions\) 什么情况下pass by reference 什么情况下return by reference](#)
- [操作符重载与临时对象](#)
  - [operator overloading \(操作符重载-1 成员函数\) this](#)
  - [return by reference语法分析: 传递者无需知道接收者是以reference形式接受](#)
  - [class body之外的各种定义 \(definitions\)](#)
  - [operator overloading \(操作符重载-2 非成员函数\) \(无this\)](#)
  - [temp object \(临时对象\) typename\(\);](#)
  - [class body 之外的各种定义 \(definitions\)](#)
  - [operator overloading \(操作符重载\), 非成员函数](#)

- 三大函数：拷贝函数、拷贝复制、析构
  - String class及三大函数
  - ctor和dtor (构造函数和析构函数)
  - class with pointer members必须有copy ctor和copy op=
  - copy ctor (拷贝构造函数)
  - copy assignment operator (拷贝赋值函数)
  - 一定要在operator中检测是否self assignment
  - output 函数
- 堆、栈与内存管理
  - 所谓stack (栈) , 所谓heap (堆)
  - stack objects的生命期
  - static local objects的生命期
  - global objects的生命期
  - heap objects的生命期
  - new: 先分配memory, 在调用ctor
  - delete: 先调用dtor, 再释放memory
  - 动态分配得到的内存块 (memory block) , in vc
  - 动态分配所得的array, in vc
  - array new一定要搭配array delete
- 扩展
  - 关键字 static
- 组合与继承
  - Composition (复合)
  - Delegation (委托) Composition by reference
  - Inheritance (继承) , 表示is-a
- 虚函数与多态
  - Inheritance (继承) with virtual functions (虚函数)

## 非指针类：



# C++ 编程简介

## 目标

- 培养正规的、大气的编程习惯
- 以良好的方式编写C++ class (**基于对象 (Object Based)**)
  - class without pointer members--Complex (复数)
  - class with pointer members--String (字符串)

这里的复数与字符串为举例

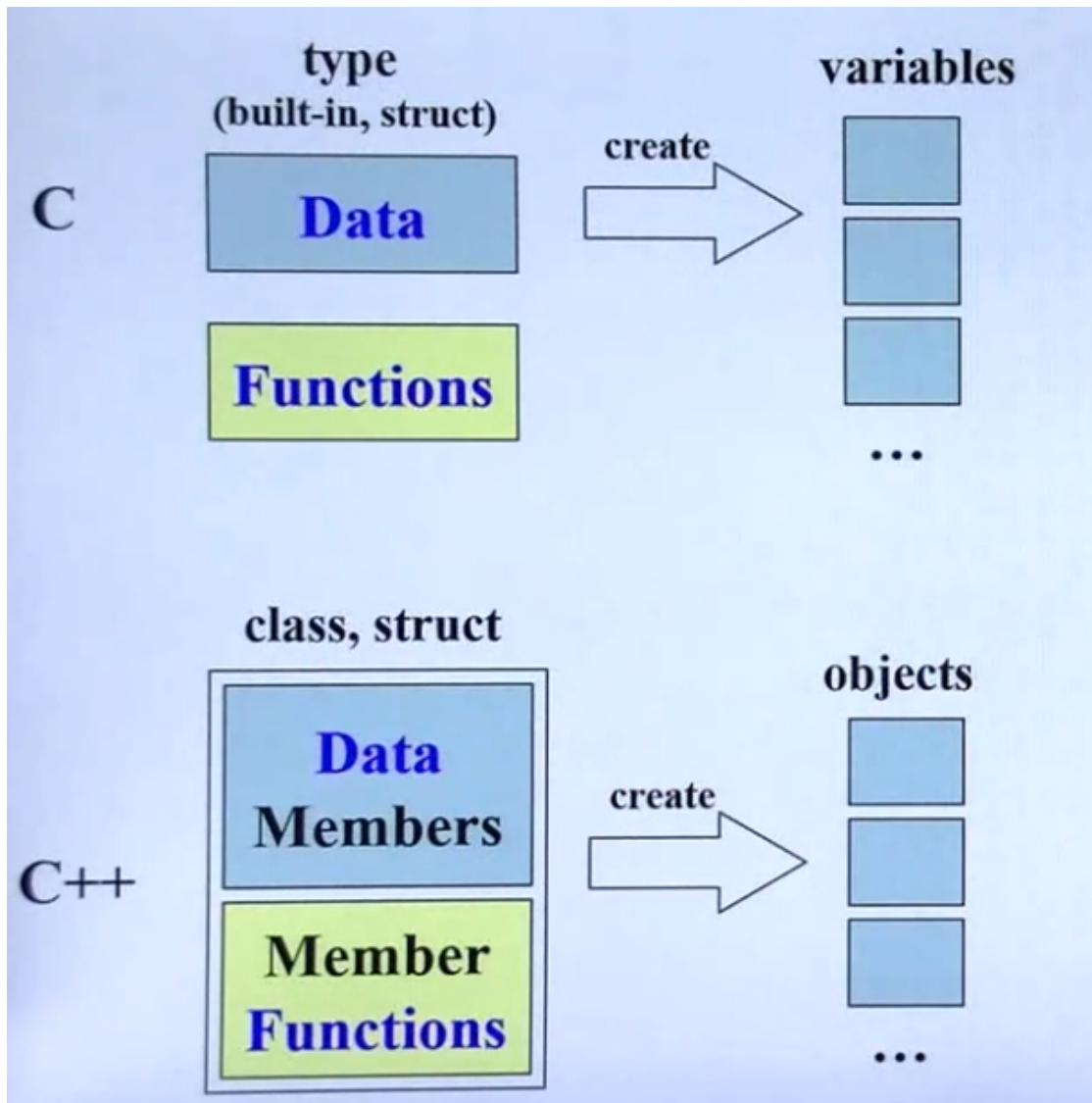
- 学习Classes之间的关系 (**面向对象 (Object Oriented)**)
  - 继承 (inheritance)
  - 复合 (composition)
  - 委托 (delegation)

## C++介绍

C++:

- C++语言
- C++标准库

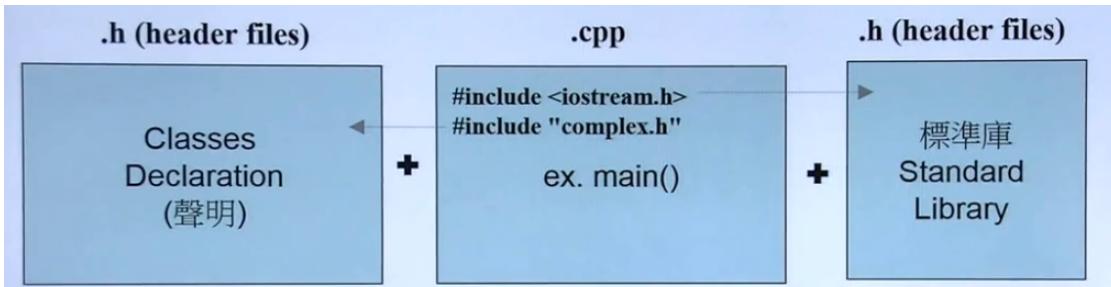
## C vs C++ 关于数据和函数



## Object Based (基于对象) vs Object Oriented (面向对象)

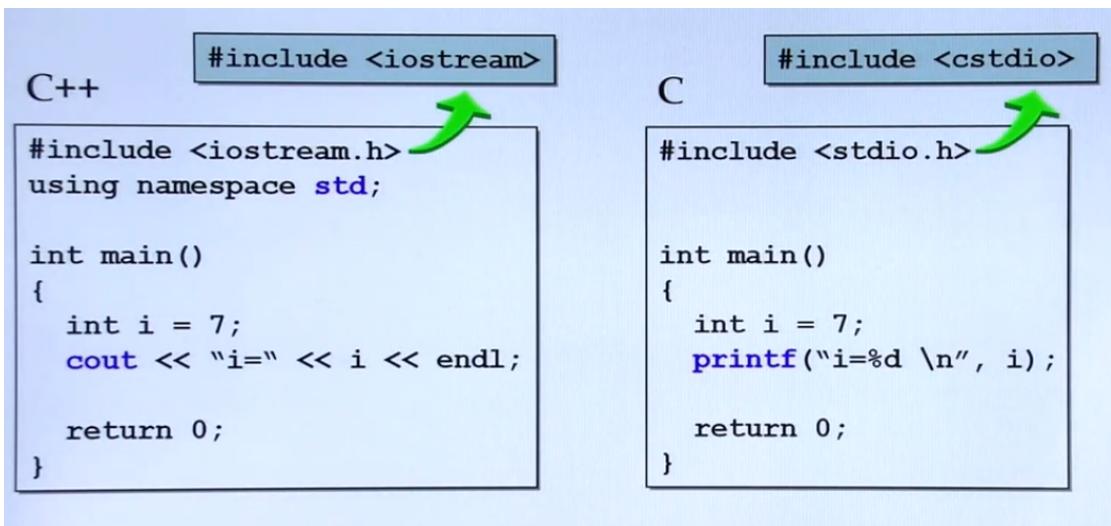
- **Object Based**: 面对的是单一class的设计
- **Object Oriented**: 面对的是多重classes的设计, class和class之间的关系

## C++ programs 代码基本形式



延伸文件名 (extension file name) 不一定是 .h 或 .cpp ,  
也可能是 .hpp 或其他或甚至無延伸名。

## Output C++ vs C



## 头文件与类的声明

### Header (头文件) 中的防卫式声明

### complex.h

```
#ifndef __COMPLEX__
#define __COMPLEX__      guard
                      (防衛式聲明)

...
#endif
```

```
#include <iostream>
#include "complex.h"
using namespace std;

int main()
{
    complex c1(2,1);
    complex c2;
    cout << c1 << endl;
    cout << c2 << endl;

    c2 = c1 + 5;
    c2 = 7 + c1;
    c2 = c1 + c2;
    c2 += c1;
    c2 += 3;
    c2 = -c1;

    cout << (c1 == c2) << endl;
    cout << (c1 != c2) << endl;
    cout << conj(c1) << endl;
    return 0;
}
```

防卫式声明：当头文件在第一次被included的时候进行定义，重复included时不会进入中间的代码区域，起到了防止重定义的作用

## Header (头文件) 的布局

```
#ifndef __COMPLEX__
#define __COMPLEX__

① #include <cmath>

class ostream;
class complex;           forward declarations
                        (前置聲明)

complex&
    __doapl (complex* ths, const complex& r);

② class complex
{
    ...
};

complex::function ...      class declarations
                           (類 - 聲明)

③ #endif                  class definition
                           (類 - 定義)
```

## class的声明 (declaration)

```
① 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&);
};
```

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

class head  
class body  
有些函數在此直接定義，  
另一些在 body 之外定義

## Class template (模板) 简介

```
① template<typename T>
class complex
{
public:
    complex (T r = 0, T i = 0)
        : re (r), im (i)
    { }
    complex& operator += (const complex&);
    T real () const { return re; }
    T imag () const { return im; }
private:
    T re, im;

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

{  
 complex<double> c1(2.5,1.5);  
 complex<int> c2(2,6);  
 ...  
}

## 构造函数

## inline (内联) 函数

```
1 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&);
};
```

2-2 `inline double  
imag(const complex& x)  
{  
 return x.imag ();  
}`

函數若在 class body  
內定義完成，便自動  
成為 inline 候選人

内联函数：函数在class body内完成定义。内联函数具备宏定义的特性（优点），但通常函数过于复杂便无法inline，inline只是相当于对编译器的一种建议，能否成为inline是编译器自行决定

## access level (访问级别)

```
1 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;

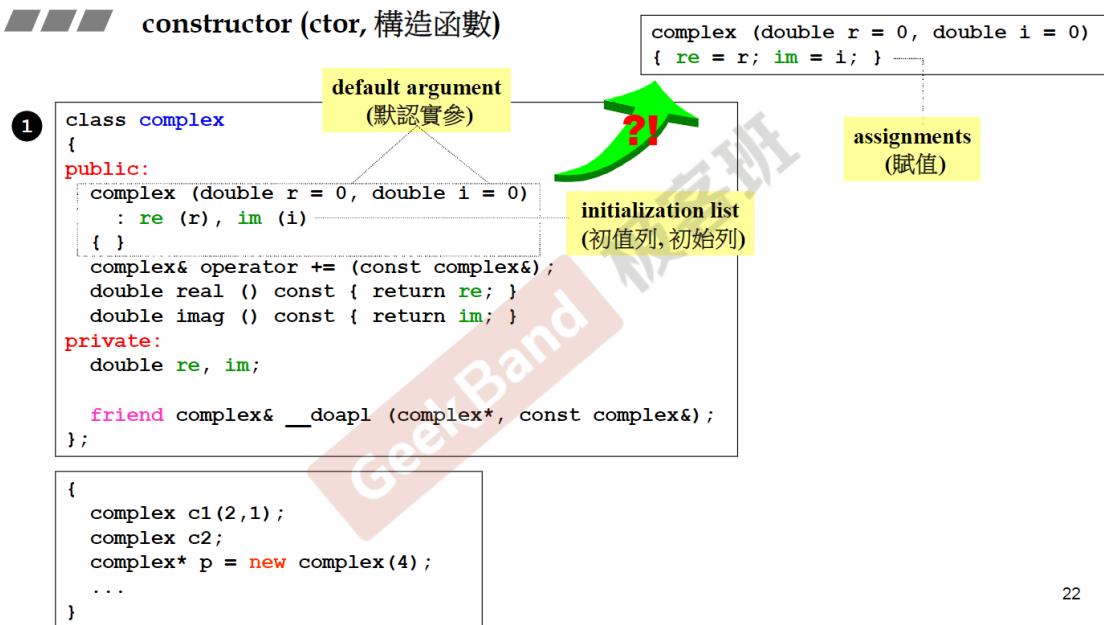
    friend complex& __doapl (complex*, const complex&);
};
```

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

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

访问级别还包含：protected，通常函数为public，变量为

## constructor (ctor 构造函数)

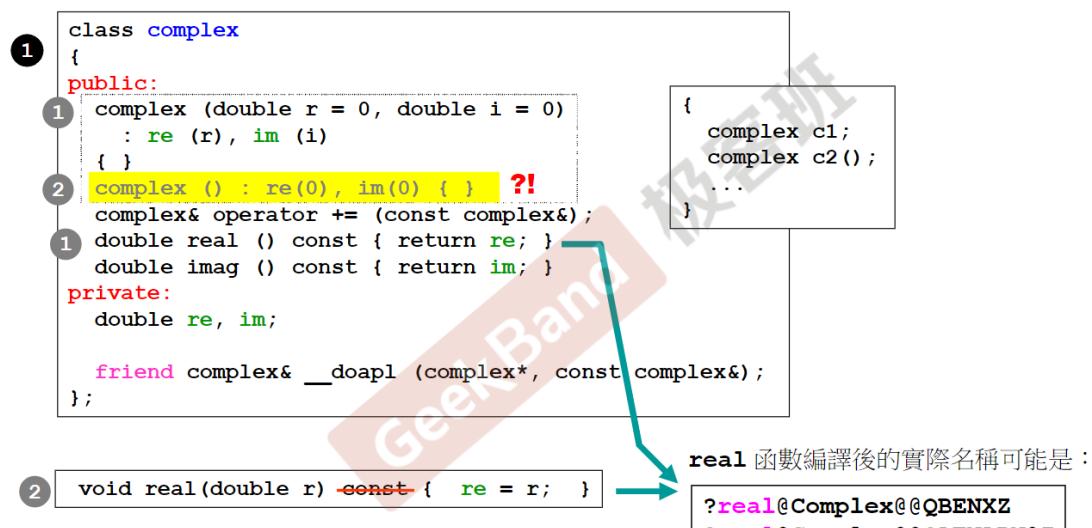


### 构造函数:

- 创建对象时自动被调用
- 构造函数名称==类的名称
- 没有返回值类型
- 可以拥有初始列 (初值列initialization list)

构造函数可以对变量进行初始化或对其进行赋值 (assignments) , 建议使用初始化, 且还须知道, 函数 (不只构造函数) 是可以有默认实参的 (default argument) , 此时介绍的 class中是不包含指针的, 通常不包含指针的类无需析构函数

## ctor (构造函数) 可以有多个overloading (重载)



取决于编译器

实际应用中，创建一个class往往需要多种方法，因此需要对其ctor进行多次overloading，重载的函数实际名称取决于编译器，当然不只构造函数可以被重载

图中的两个ctor会发生冲突，因为ctor1的参数有默认值，在创建对象时，编译器会不知道应调用ctor1还是ctor2

## constructor (ctor 构造函数) 被放在private区

```
1 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);
    complex c2;
    ...
}
```

当构造函数被放置private区，外界无法创建该class的对象，单例模式就是如此。

## Singleton

```
class A {  
public:  
    static A& getInstance();  
    setup() { ... }  
private:  
    A();  
    A(const A& rhs);  
    ...  
};  
  
A& A::getInstance()  
{  
    static A a;  
    return a;  
}
```

A::getInstance().setup();

## const member function (常成员函数)

1

```
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&);  
};
```

0

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

?!  
{

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

class中的function包含：

- 改变数据内容

- 不改变数据内容 (加上const)

在创建class的const实例，且class中的function没有加const会出现无法调用函数的error

## 参数传递与返回值

### 参数传递: pass by value vs pass by reference (to const)

```

1 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&);
};

2-7 ostream&
operator << (ostream& os, const complex& x)
{
    return os << '(' << real (x) << ','
                  << imag (x) << ')';
}
{
    complex c1(2,1);
    complex c2;

    c2 += c1;
    cout << c2;
}

```

- pass by value: value整包传过去, value压到栈 (**此处感觉是传递了复制的**)
- pass by reference: 加快传参的速度,
- pass by reference to const: 传递引用有被更改数据的风险, 若不想更改可以在参数前加上const
- C语言中, 当value过大, 可以传指针

### 返回值传递: return by value vs return by reference (to const)

1

```

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&);
};

```

2-7

```

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

```

```

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

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

```

## friend (友元)

1

```

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&);
};

```

2-1

```

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 成員

friend可以获取class的private成员  
friend不易太多，因为其打破了封装性

相同class的各个objects互为friends (友元)

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

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

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

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

    c2.func(c1);
}
```

相同class的各个objects互为friends (友元) , 因此图中的函数不需要friend关键词

**class body 外的各种定义 (definitions) 什么情况下pass by reference 什么情况下return by reference**

## do assignment plus

2-1

```
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);
}
```

第一參數將會被改動  
第二參數不會被改動

local object 不要被return by reference

## 操作符重载与临时对象

### operator overloading (操作符重载-1 成员函数) this

2-1

```
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);
}
```

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

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

所有的成员函数包含了一个隐藏参数this，调用该成员函数的object即是this，例中c2便是this，而this是一个指针，编译器会将c2的指针传进函数

## return by reference语法分析：传递者无需知道接收者是以reference形式接受

2-1

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

c3 += c2 += c1;

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

```
//接收是complex&, 按reference接收的  
inline complex&  
__doapl(complex* this, const complex& r)  
{  
    ...  
    //返回的是by value, 返回了一个object  
    return *this  
}
```

```
//该语句是先将c1加到c2, 结果加到c3上  
c3 += c2 += c1;  
//c2与c1结果需要作为右值 加到c3身上  
//所以本例中的重载函数不可以为void
```

## class body之外的各种定义 (definitions)

全局函数：

2-2

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

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

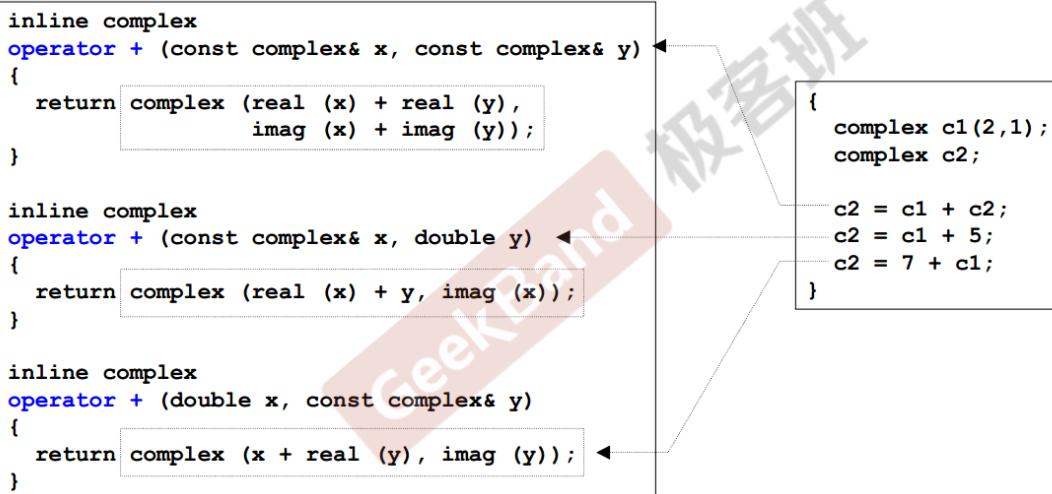
## operator overloading (操作符重载-2 非成员函数) (无 this)

多次重载+：

2-3

為了對付 client 的三種可能用法，這兒對應開發三個函數

```
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));  
}
```



temp object (临时对象) typename();

2-3

下面這些函數絕不可 return by reference，  
因為，它們返回的必定是個 local object.

```
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));
}
```

```
{
int(7);

complex c1(2,1);
complex c2;
complex();
complex(4,5);

cout << complex(2);
}
```

typename();该形式会创造临时对象，是一个local object不可以return by reference

```
//这两句话皆是临时对象，生命周期到这句话运行结束
complex();
complex(4, 5);
```

## class body 之外的各种定义 (definitions)

2-4

**negate**  
反相  
(取反)

```
inline complex
operator + (const complex& x)
{
    return x;
}

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

```
{
complex c1(2,1);
complex c2;
cout << -c1;
cout << +c1;
}
```

這個函數絕不可  
return by reference，  
因為其返回的  
必定是個 local object。

此处重载的不是加法而是正号和负号，同样重载负号时，因为对改变了传进来的 complex，因此需要临时存储也就是local object，返回时不能return by reference 而正号重载时没有更改变量，没有local object，理论上是可以改为return by reference

## operator overloading (操作符重载)，非成员函数

2-7

```

inline complex
conj (const complex& x)
{
    return complex (real (x), -imag (x));
}

#include <iostream.h>
ostream&
operator << (ostream& os, const complex& x)
{
    return os << '(' << real (x) << ','
                  << imag (x) << ')';
}

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

```

(2,-1)  
(2,1)(2,-1)

```

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

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

```

40

重载操作符可以有两种选择：

- 定义为全局函数
- 定义为成员函数

**重载<<(特殊):**

- <<是作用于左侧的cout，所以重载<<时应选择为全局函数，如果选择成员函数大概要写成complex << cout;
- 而cout的type那么为ostream，在其作为参数时不可以加const，因为每次在cout时相当于都改变了os的状态
- 考虑到连续cout的情况，返回值不应该是void，应该还返回cout即ostream

## 三大函数：拷贝函数、拷贝复制、析构

### String class及三大函数

```

#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
    ...
};

String::function(...)

Global-function(...)

#endif

```

string.h

```

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

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

    cout << s3 << endl;
}

```

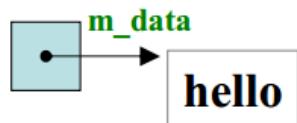
string-test.cpp

1

```

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;
};

```



## ctor和dtor (构造函数和析构函数)

2-1

```

inline
String::String(const char* cstr = 0)
{
    if (cstr) {
        m_data = new char[strlen(cstr)+1];
        strcpy(m_data, cstr);
    }
    else { // 未指定初值
        m_data = new char[1];
        *m_data = '\0';
    }
}

inline
String::~String()
{
    delete[] m_data;
}

```

```

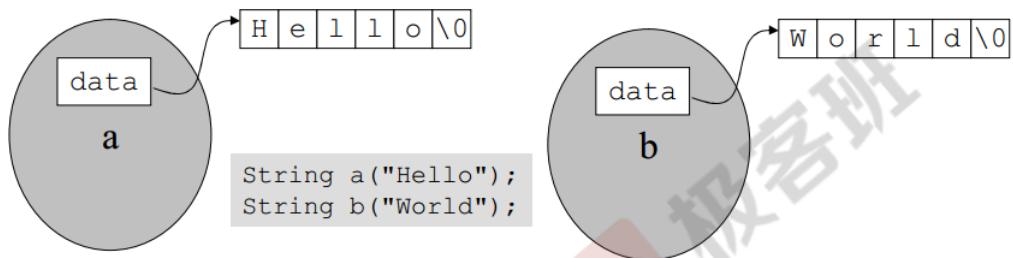
{
    String s1(),
    String s2("hello");

    String* p = new String("hello");
    delete p;
}

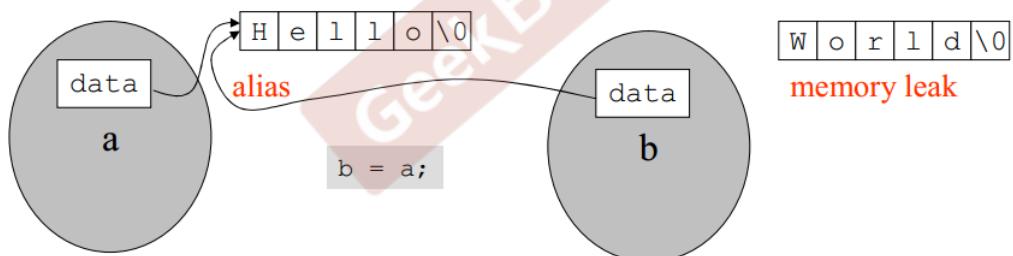
```

A diagram illustrating the creation and destruction of String objects. It shows a pointer member m\_data pointing to the string "hello". The pointer is created in the constructor's code (new char[1]), assigned to m\_data, and then copied to the heap (strcpy). In the destructor's code (delete[]), the pointer is deleted, which releases the heap-allocated memory.

class with pointer members必须有copy ctor和copy op=



使用 default copy ctor 或 default op= 就會形成以下局面



**浅拷贝：**因为a、b为指针并不是字符串本身，使用默认拷贝构造或=，使得b的指针指向了a指针指向的内容，b原先指向的那块内存发生了内存泄漏memory leak，而ab同时指向一块内存会造成改变a时b也跟着改变的问题。为了避免发生浅拷贝所造成的问题，需要采用**深拷贝**

## copy ctor (拷贝构造函数)

2-2

```
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;
}
```

直接取另一個 object 的 private data.  
(兄弟之間互為 friend)

深拷贝：

- 分配足够的空间
- 将内容复制到新的内存空间

## copy assignment operator (拷贝赋值函数)

2-3

```

inline
String& String::operator=(const String& str)
{
    if (this == &str) 檢測自我賦值
        return *this; (self assignment)

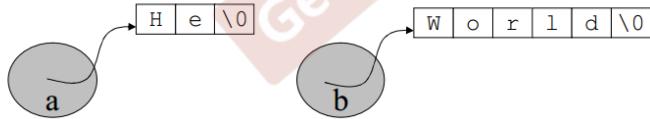
    ① delete[] m_data;
    ② m_data = new char[ strlen(str.m_data) + 1 ];
    ③ strcpy(m_data, str.m_data);
        return *this;
}

```

```

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

```



拷贝赋值：

- $a = b;$
- $a = a;$  自我赋值

当a与b都有内容的时候，想完成如上的赋值操作，需要先将a的内存清空，重新分配足够容纳b内容的内存，再将b的内容拷贝过来

自我赋值的检测

```

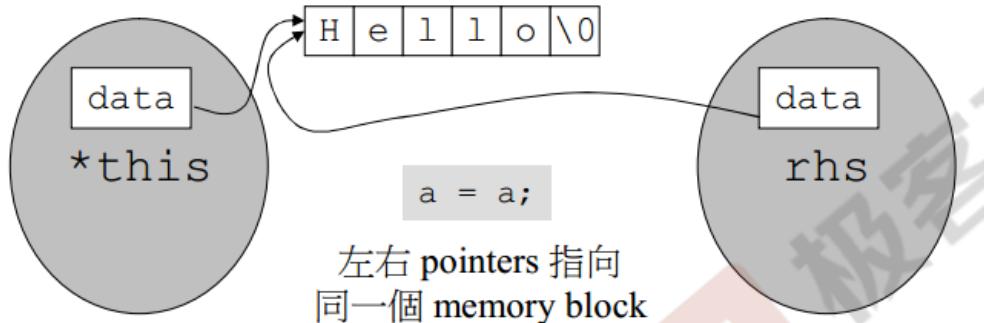
if(this == &str)
    return *this;

```

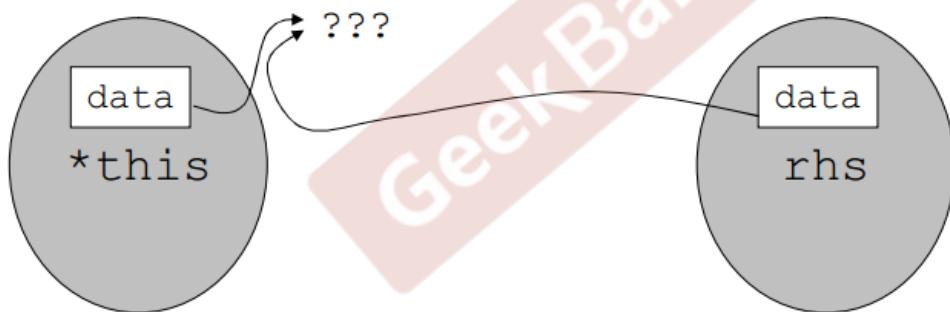
`this`是函数的默认参数

判断传进来的引用与`this`指向是否为同一块内存

## 一定要在operator中检测是否self assignment



前述 `operator=` 的第一件事情就是 `delete`，造成這般結果：



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

## output 函数

非成员函数 (全局函数) :

把指针丢给 <<

```
2-4 #include <iostream.h>
ostream& operator<<(ostream& os, const String& str)
{
    os << str.get_c_str();
    return os;
}

{
    String s1("hello ");
    cout << s1;
}
```

## 堆、栈与内存管理

所谓stack (栈)， 所谓heap (堆)

**Stack**，是存在於某作用域 (scope) 的一塊內存空間 (**memory space**)。例如當你調用函數，函數本身即會形成一個 **stack** 用來放置它所接收的參數，以及返回地址。

在函數本體 (**function body**) 內聲明的任何變量，其所使用的內存塊都取自上述 **stack**。

**Heap**，或謂 **system heap**，是指由操作系統提供的  
一塊 **global** 內存空間，程序可動態分配 (**dynamic allocated**) 從某中獲得若干區塊 (**blocks**)。

```
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。

## stack objects的生命期

```
class Complex { ... };
...
{
    Complex c1(1, 2);
}
```

c1 便是所謂 **stack object**，其生命在作用域 (scope) 結束之際結束。  
這種作用域內的 object，又稱為 **auto object**，因為它會被「自動」清理。

## static local objects的生命期

```
class Complex { ... };
...
{
    static Complex c2(1,2);
}
```

c2 便是所謂 **static object**，其生命在作用域 (scope) 結束之後仍然存在，直到整個程序結束。

### global objects的生命期

```
class Complex { ... };
...
Complex c3(1,2);

int main()
{
    ...
}
```

c3 便是所謂 **global object**，其生命在整個程序結束之後才結束。你也可以把它視為一種 **static object**，其作用域是「整個程序」。

### heap objects的生命期

```

class Complex { ... };

...
{
    Complex* p = new Complex;
    ...
    delete p;
}

```

```

class Complex { ... };

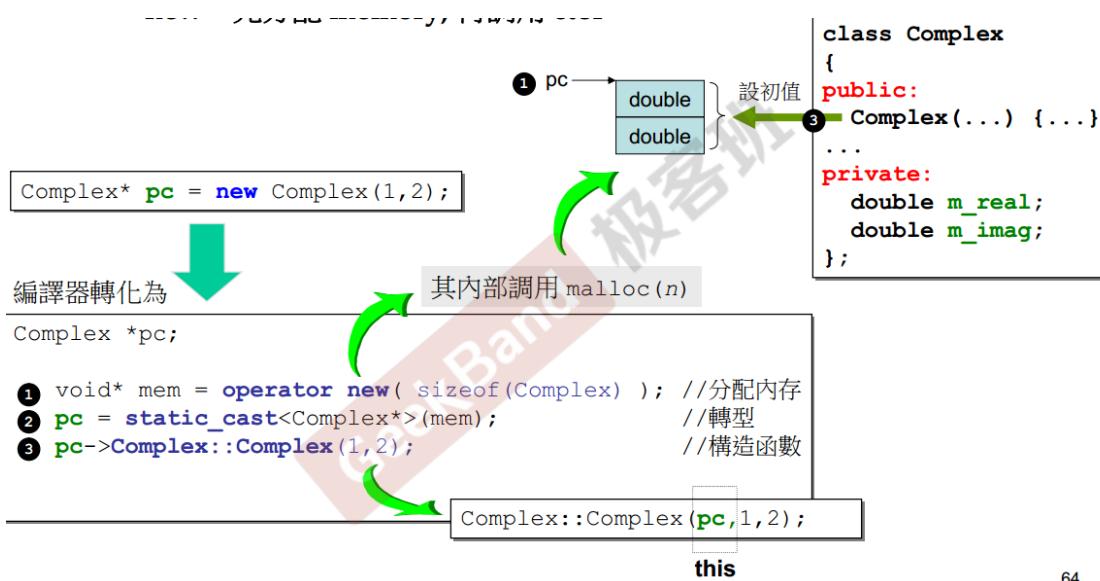
...
{
    Complex* p = new Complex;
}

```

`p` 所指的便是 heap object，其生命在它被 `deleted` 之際結束。

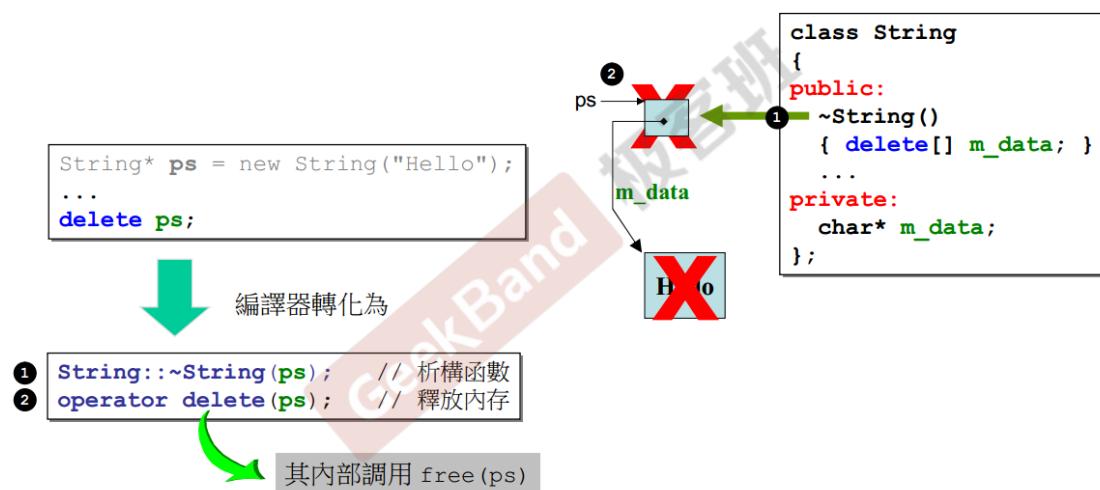
`new` 完要记得 `delete`，否则会造成内存泄漏

## new: 先分配memory，在调用ctor



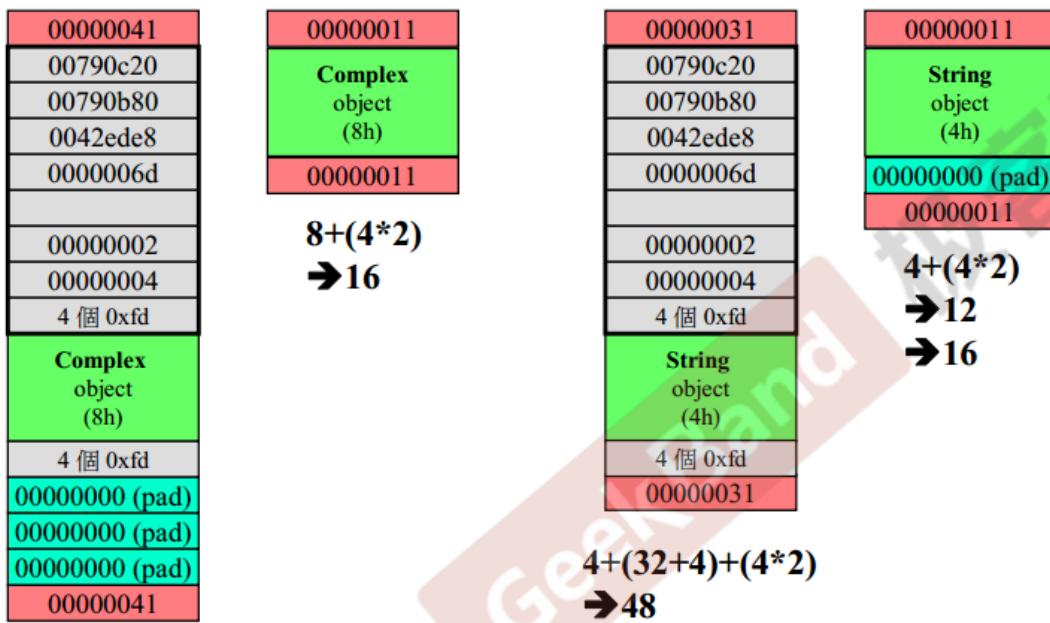
64

## delete: 先调用dtor, 再释放memory



operator new与operator delete 是C++的特殊函数

## 动态分配得到的内存块 (memory block) , in vc



$$8 + (32 + 4) + (4 * 2)$$

$$\rightarrow 52$$

$$\rightarrow 64$$

**class complex:**

- class本身8个字节
- **debug mode:** 前面 $8 * 4$ 个字节, 后面4个字节
- cookie: 前后各4字节
- 占用内存: 每一块分配内存必须是16的倍数, 所以得到的内存块是64字节

**release mode:**

- $\text{complex} + \text{cookie} = 16$

**cookie的作用:** 记录分配内存的大小。

cookie的值:

- debug mode: 64的16进制数是40, cookie的数字是41, 最后一个数字1代表着获得内存
- release mode: 16的16进制为10, 最后一位1, 所以为11

这也说明了为什么使用16的倍数, 最后四个位都是零

**class string:**

debug:

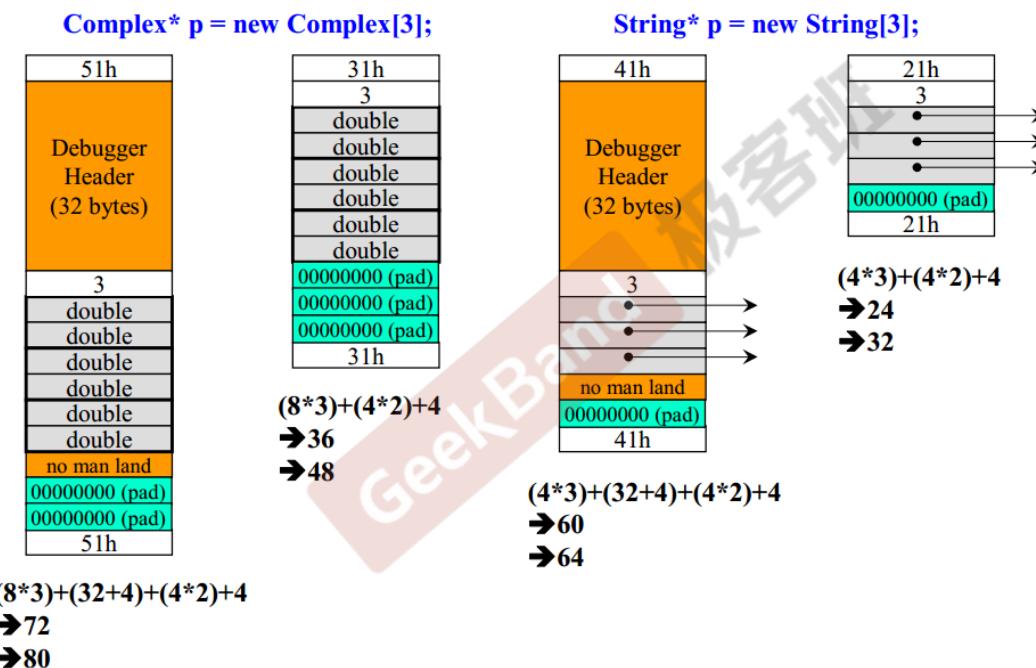
- class: 4字节
- debug mode:  $32 + 4$

- cookie:  $4 * 2$
- 占用内存: 48
- cookie值: 31

release:

- class: 4字节
- cookie:  $4 * 2$
- 补充: 4
- 占用内存: 16
- cookie值: 11

## 动态分配所得的array, in vc

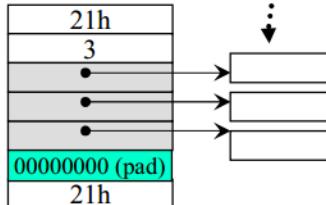


- new typename[];: 术名array new
- delete[] name;: array delete

-array情况下会再多4字节：用一个整数记录储存数组的个数

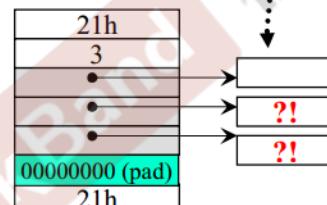
## array new一定要搭配array delete

```
String* p = new String[3];
...
delete[] p; // 喚起3次dtor
```



```
String* p = new String[3];
...
delete p; // 喚起1次dtor
```

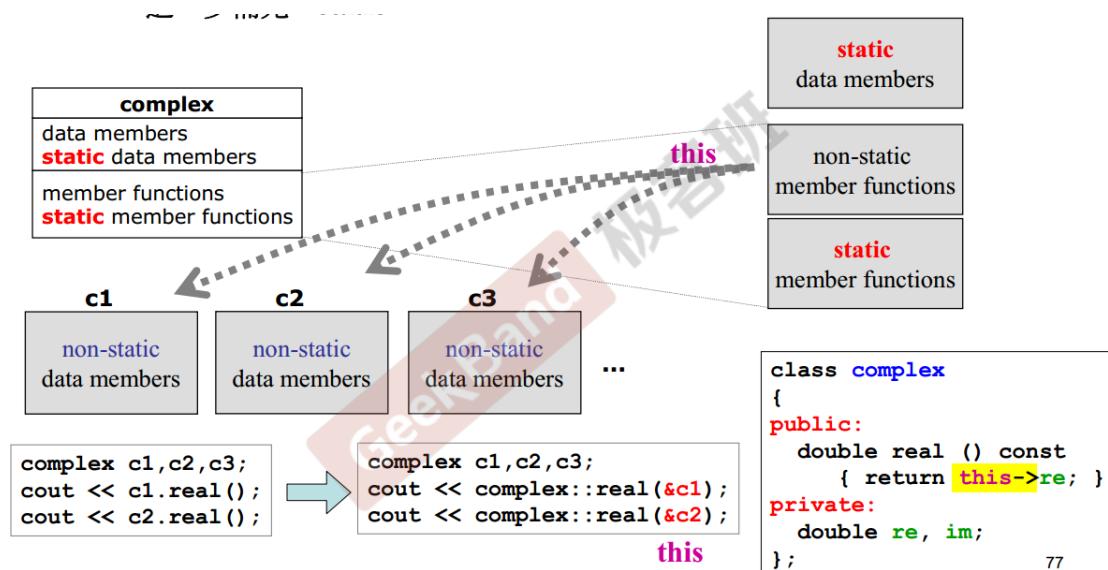
不正確的用法  
少了 []



没有搭配使用array delete：会发生内存泄漏，但并不是指针数组占用的内存泄漏，而是  
dtor只被调用了一次，指针数组的未被析构部分**指向的内存**会发生泄漏  
本质上：没有指针的类是不会造成这样的内存泄漏

## 扩展

### 关键字 static



static 的对象没有 this pointer

```

class Account {
public:
    static double m_rate;
    static void set_rate(const double& x) { m_rate = x; }
};

double Account::m_rate = 8.0;

int main() {
    Account::set_rate(5.0);

    Account a;
    a.set_rate(7.0);
}

```

調用 static 函數的方式有二：  
 (1) 通過 object 調用  
 (2) 通過 class name 調用

把ctors放在private区

## Singleton

```

class A {
public:
    static A& getInstance( return a; );
    setup() { ... }

private:
    A();
    A(const A& rhs);
    static A a;
    ...
};

```

**A::getInstance().setup();**

## Meyers Singleton

```
class A {  
public:  
    static A& getInstance();  
    setup() { ... }  
private:  
    A();  
    A(const A& rhs);  
    ...  
};  
  
A& A::getInstance()  
{  
    static A a;  
    return a;  
}
```

```
A::getInstance().setup();
```

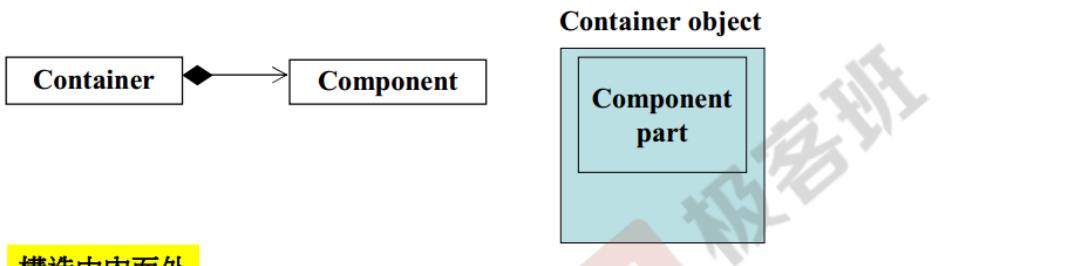
## 组合与继承

- Inheritance (继承)
- Composition (复合)
- Delegation (委托)

### Composition (复合)

Composition (复合) , 表示has-a  
一个复合类型中包含了某一种类型的变量

Composition (复合) 关系下的构造和析构



構造由內而外

**Container** 的構造函數首先調用 **Component** 的 **default** 構造函數，然後才執行自己。

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

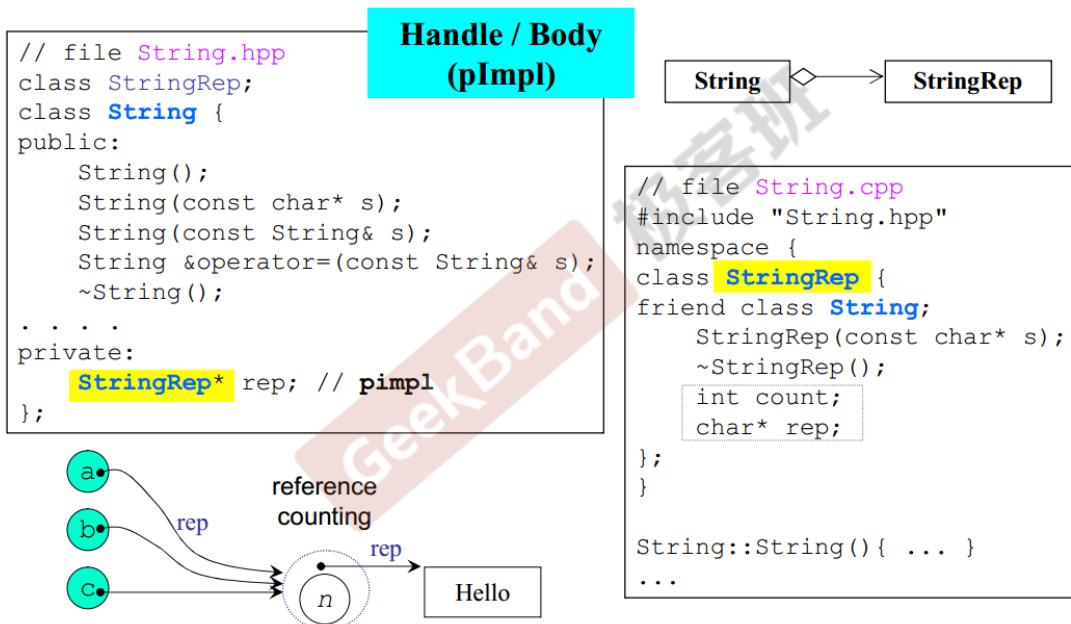
析構由外而內

**Container** 的析構函數首先執行自己，然後才調用 **Component** 的析構函數。

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

关于复合：若你有一个外部的对象，你便会有内部的对象，因为你的外部对象的构造函数会在初值列对内部对象进行初始化，二者的生命期机会一致。

## Delegation (委托) Composition by reference



注意：没有Compostion by pointer 的说法

### 委托的特点

- 委托不同于复合其内外部对象的生命不保持同步
- 起到编译防火墙的作用，外部接口不需要编译，只编译内部指向的对象

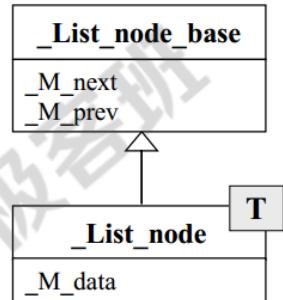
引用计数、写时复制

## Inheritance (继承)，表示is-a

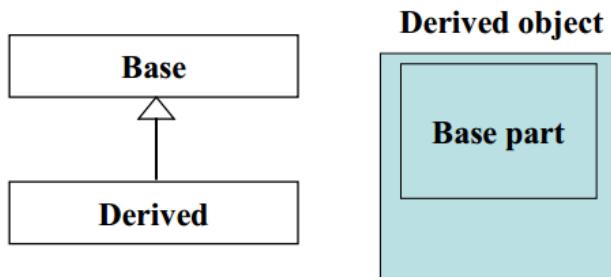
语法：(黄色)

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

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



Inheritance (继承) 关系下的构造和析构



base class 的 dtor  
必須是 virtual，  
否則會出現  
undefined behavior

構造由內而外

Derived 的構造函數首先調用 Base 的 default 構造函數，  
然後才執行自己。

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

析構由外而內

Derived 的析構函數首先執行自己，然後才調用 Base 的  
析構函數。

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

父类的析构函数必须是virtual

## 虚函数与多态

### Inheritance (继承) with virtual functions (虚函数)

**non-virtual** 函數：你不希望 derived class 重新定義 (override, 覆寫) 它。

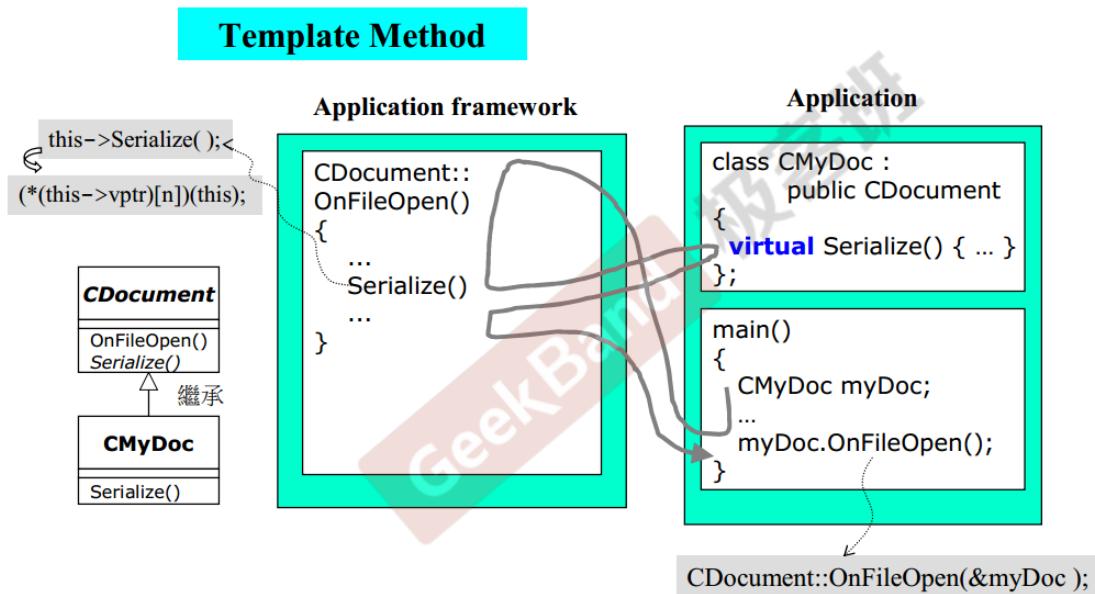
**virtual** 函數：你希望 derived class 重新定義 (override, 覆寫) 它，且你對它已有默認定義。

**pure virtual** 函數：你希望 derived class 一定要重新定義 (override 覆寫) 它，你對它沒有默認定義。

```
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 { ... };
```

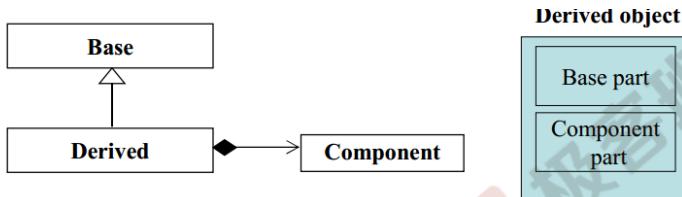
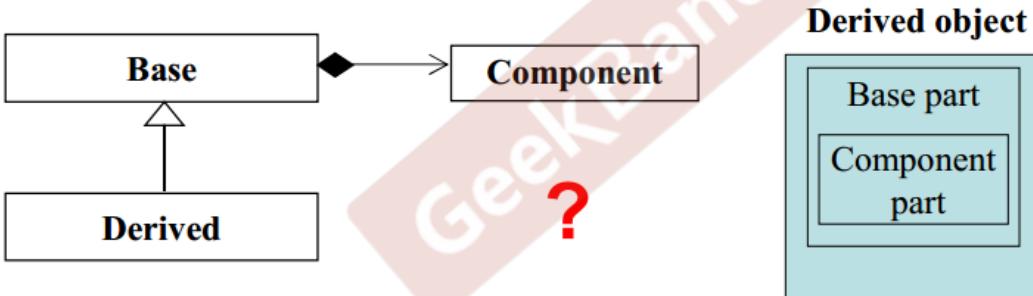
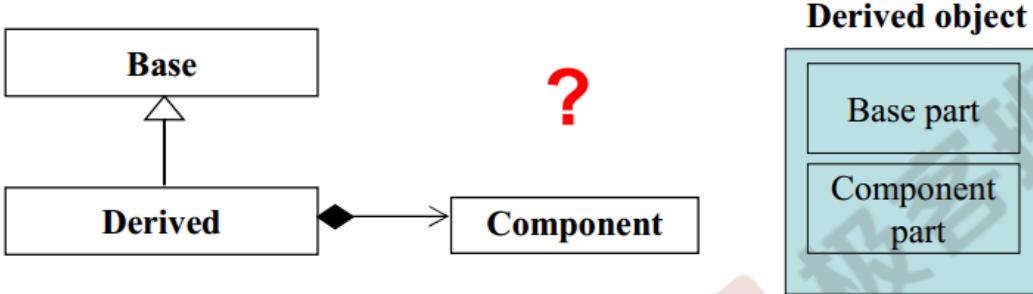
pure virtual  
impure virtual  
non-virtual

模板方法（设计模式）：



子类调用父类的函数等同于，子类作为this pointer，被传进了父类调用的该函数中，  
CDocument中的serialize ()之所以能找到子类重写的虚函数，就是靠这个this pointer

Inheritance+composition关系下的构造与析构



構造由內而外

Derived 的構造函數首先調用 Base 的 default 構造函數，  
 然後調用 Component 的 default 構造函數，  
 然後才執行自己。

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

析構由外而內

Derived 的析構函數首先執行自己，  
 然後調用 Component 的析構函數，  
 然後調用 Base 的析構函數。

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

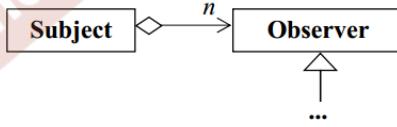
Delegation+Inheritance

## 观察者模式Observer:

```
class Subject
{
    int m_value;
    vector<Observer*> m_views;
public:
    void attach(Observer* obs)
    {
        m_views.push_back(obs);
    }
    void set_val(int value)
    {
        m_value = value;
        notify();
    }
    void notify()
    {
        for (int i = 0; i < m_views.size(); ++i)
            m_views[i]->update(this, m_value);
    }
};
```

## Observer

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



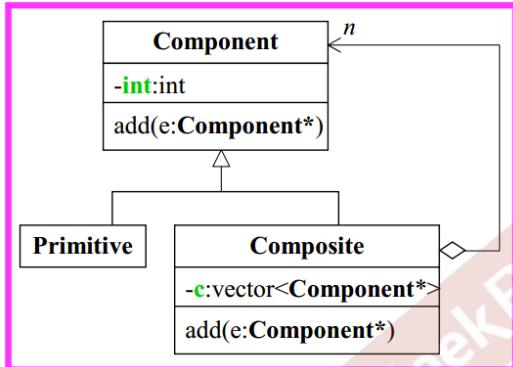
vector指针 (delegation) 指向Observer,  
而Observer是可以被继承的，这些子类都可以放到vector这个容器中

## 组合模式Composite:

Primitive: 个体

Composite: 组合物

### Composite



### Component

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

### Composite: public Component

```
{ 
    vector <Component*> c;
public:
    Composite(int val): Component(val) { }

    void add(Component* elem) { 
        c.push_back(elem);
    }
    ...
};
```

Prototype: 没懂