GeekBand 极客班

互联网人才 + 油站!

C++面向对象高级编程

www.geekband.com

GeekBand 极客班 互联网人才+油站！

极客班携手网易云课堂，针对热门IT互联网岗位，联合业内专家大牛，紧贴企业实际需求，量身打造精品实战课程。

专业课程 ＋ 项目碾压

- 顶尖专家技能私授
- 贴合企业实际需求
- 互动交流直播答疑

- 学员混搭线上组队
- 一线项目实战操练
- 业内大牛辅导点评

www.geekband.com

C++面向對象程序設計

(Object Oriented Programming, OOP)

侯捷

勿在浮沙築高台

你應具備的基礎

- 曾經學過某種 procedural language (C 語言最佳)
  - 變量 (variables)
  - 類型 (types) : int, float, char, struct …
  - 作用域 (scope)
  - 循環 (loops) : while, for,
  - 流程控制 : if-else, switch-case
- 知道一個程序需要編譯、連結才能被執行
- 知道如何編譯和連結
  (如何建立一個可運行程序）

我們的目標

- 培養正規的、大氣的編程習慣

- 以良好的方式編寫 C++ class
  - class without pointer members
    - Complex
  - class with pointer members
    - String

Object Based
(基於對象) → 单一

- 學習 Classes 之間的關係
  - 繼承 (inheritance)
  - 複合 (composition)
  - 委託 (delegation)

Object Oriented
(面向對象) → # of class 互动

你將獲得的代碼

complex.h
complex-test.cpp

string.h
string-test.cpp

C++ 的歷史

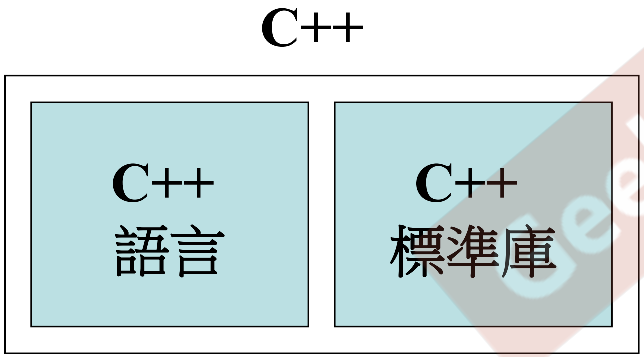
- **B 語言 (1969)**
- **C 語言 (1972)**
- **C++ 語言 (1983)**

  (new C ➔ C with Class ➔ C++)

- **Java 語言**
- **C# 語言**

C++ 演化

- C++ 98 (1.0)
- C++ 03 (TR1, Technical Report 1)
- C++ 11 (2.0)
- C++ 14

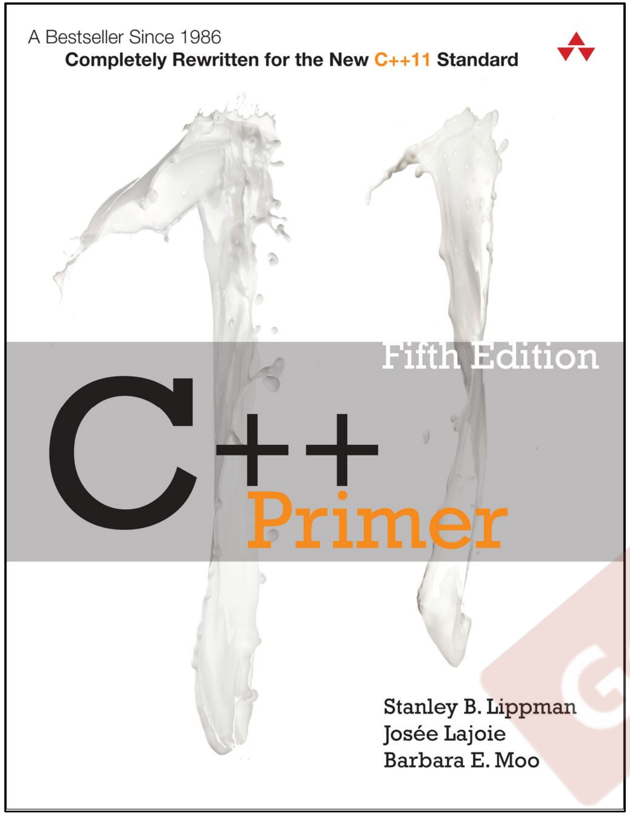


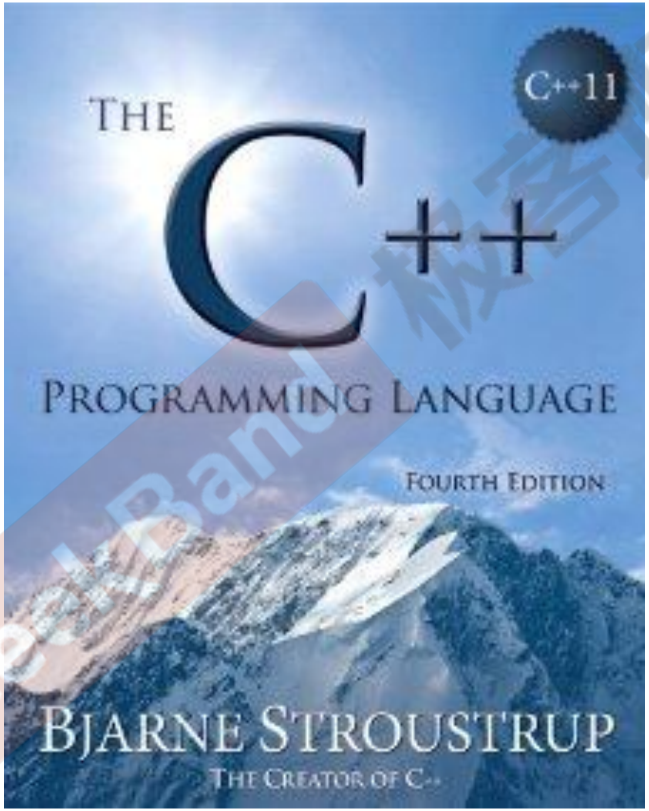
C++

C++
語言

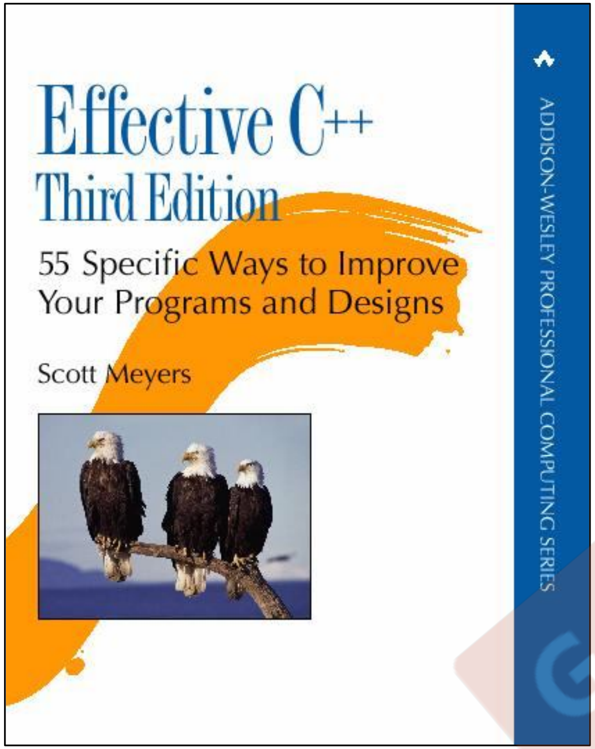
C++
標準庫

Bibliography (書目誌)





Bibliography (書目誌)

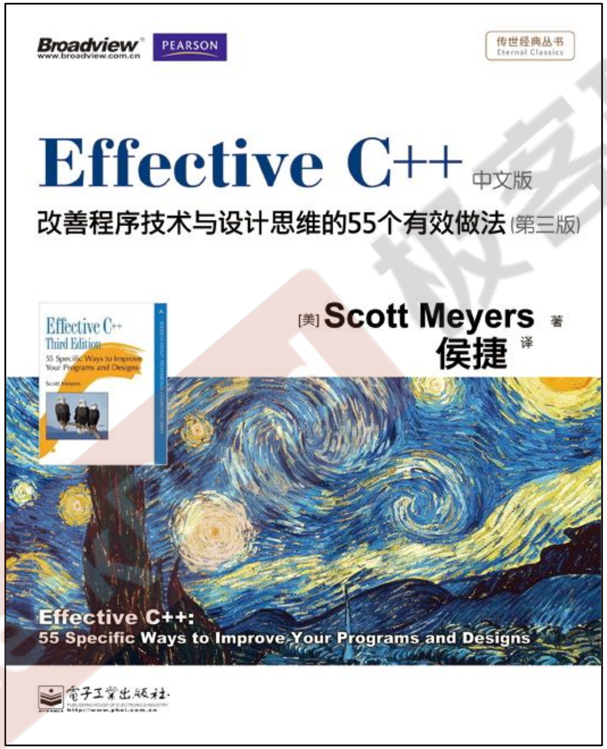


Effective C++
Third Edition

55 Specific Ways to Improve Your Programs and Designs

Scott Meyers

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Broadview
www.broadview.com.cn

PEARSON

传世经典丛书
Eternal Classics

Effective C++ 中文版

改善程序技术与设计思维的55个有效做法 (第三版)

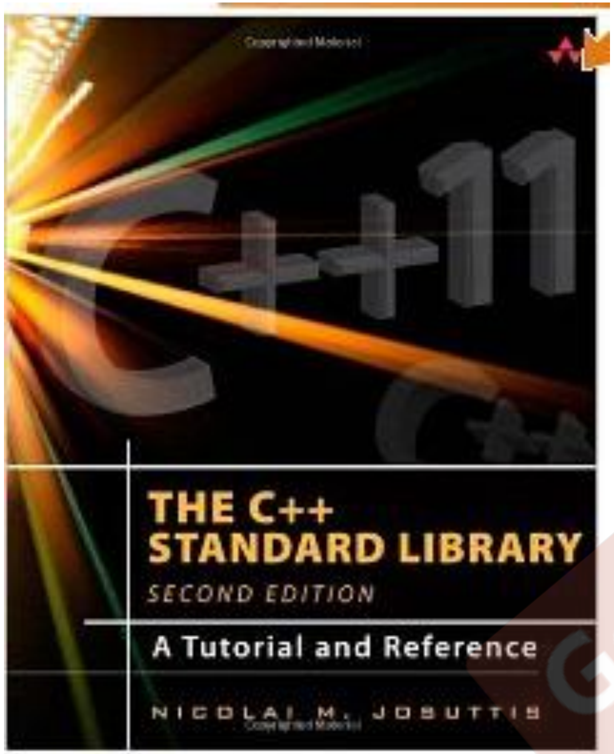
[美] Scott Meyers 著

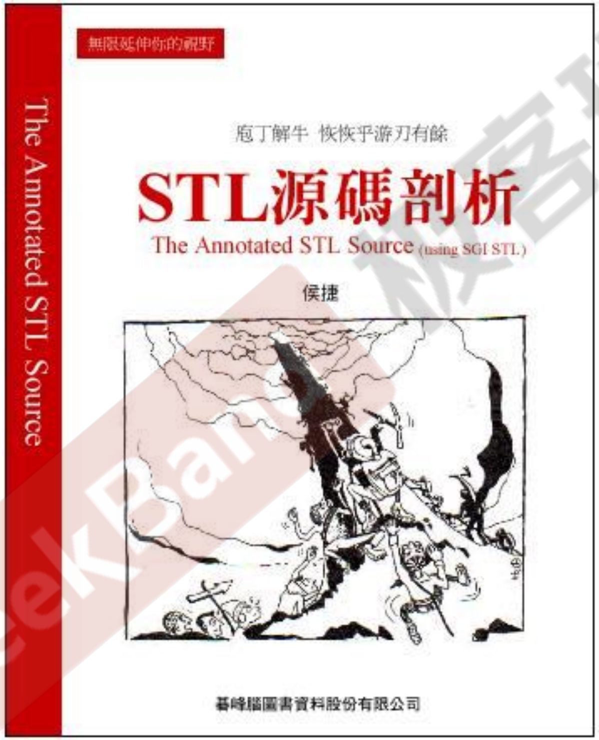
侯捷 译

Effective C++:
55 Specific Ways to Improve Your Programs and Designs

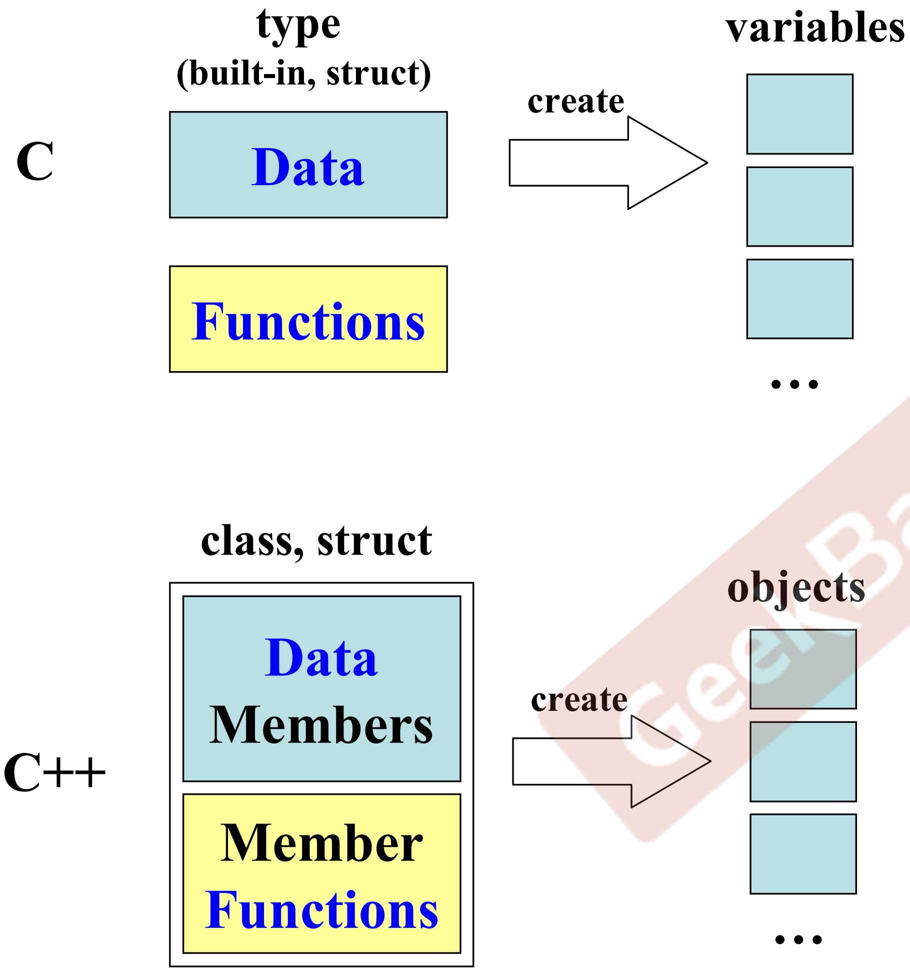
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Bibliography (書目誌)





C vs. C++, 關於數據和函數



type
(built-in, struct)

C

Data

Functions

create

variables

…

class, struct

C++

Data
Members

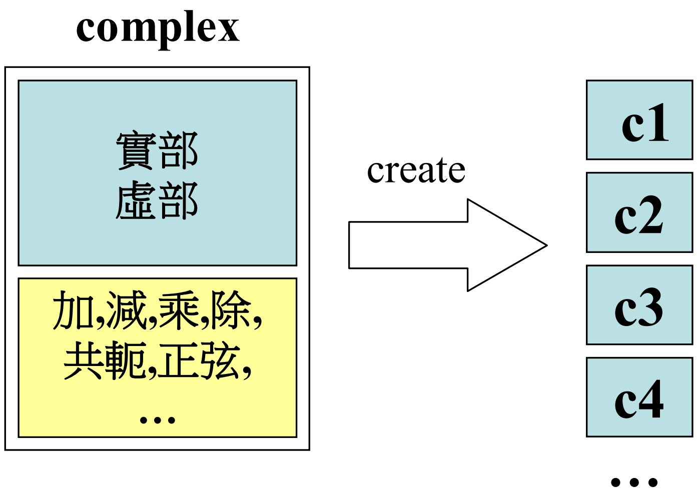
Member
Functions

create

objects

…

C++, 關於數據和函數



complex

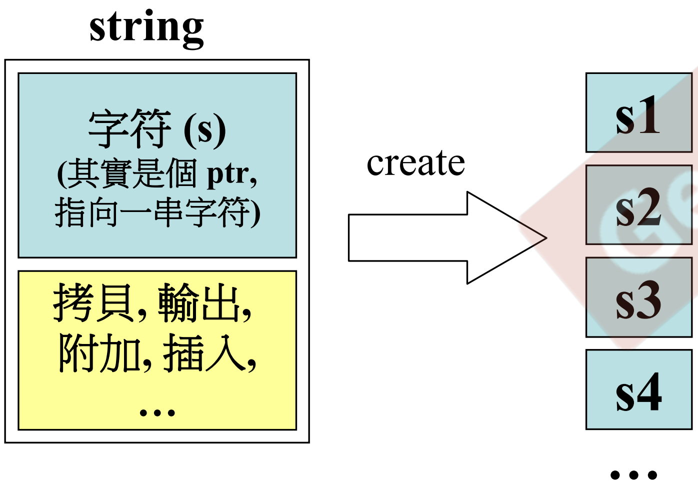
實部
虛部

加,減,乘,除,
共軛,正弦,
…

create

c1
c2
c3
c4
…

```
complex c1(2,1);
complex c2;
complex* pc = new complex(0,1);
```



string

字符 (s)
(其實是個 ptr,
指向一串字符)

拷貝, 輸出,
附加, 插入,
…

create

s1
s2
s3
s4
…

```
string s1("Hello ");
string s2("World ");
string* ps = new string;
```

Object Based (基於對象) vs. Object Oriented (面向對象)

Object Based : 面對的是單一 class 的設計

**Object Oriented : 面對的是多重 classes 的設計，
classes 和 classes 之間的關係。**

我們的第一個 C++ 程序

Classes 的兩個經典分類：

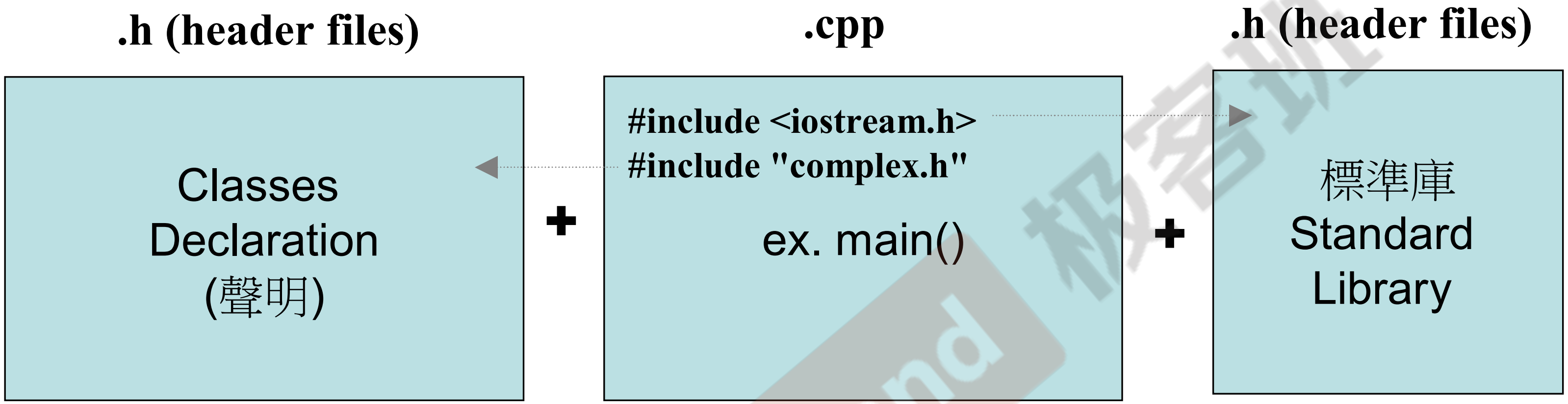
- Class without pointer member(s)

  complex

- Class with pointer member(s)

  string

C++ programs 代碼基本形式



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

C++

```
#include <iostream>
```

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

int main()
{
  int i = 7;
  cout << “i=“ << i << endl;

  return 0;
}
```

C

```
#include <cstdio>
```

```
#include <stdio.h>


int main()
{
  int i = 7;
  printf(“i=%d \n”, i);

  return 0;
}
```

Header (頭文件) 中的防衛式聲明

complex.h

```
#ifndef __COMPLEX__
#define __COMPLEX__



...



#endif
```

guard
(防衛式聲明)

complex-test.h

```
#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;
}
```

Header (頭文件) 的佈局

```
#ifndef __COMPLEX__
#define __COMPLEX__
```

0

```
#include <cmath>

class ostream;
class complex;

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

forward declarations
(前置聲明)

1

```
class complex
{
...
};
```

class declarations
(類 - 聲明)

2

```
complex::function ...
```

class definition
(類 - 定義)

```
#endif
```

class 的聲明 (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&);
};
```

class head

class body

有些函數在此直接定義，
另一些在 body 之外定義

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

class template (模板) 簡介

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);
  ...
}
```

inline (內聯) 函數

in line

→ faster.

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&);
};
```

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

2-2

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

access level (訪問級別)

❶

```
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();
}
```

constructor (ctor, 構造函數)

```
complex (double r = 0, double i = 0)
{ re = r; im = i; }
```

assignments
(賦值)

?!

default argument
(默認實參)

initialization list
(初值列, 初始列)

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;
  complex* p = new complex(4);
  ...
}
```

→ static : on the stack

→ dynamic : on the heap

ctor (構造函數) 可以有很多個 – overloading (重載)

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

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

冲突

```
{
  complex c1;
  complex c2();
  ...
}
```

```
2   void real(double r) ~~const~~ {   re = r;   }
```

real 函數編譯後的實際名稱可能是：

```
?real@Complex@@QBENXZ
?real@Complex@@QAENABN@Z
```

取決於編譯器

constructor (ctor, 構造函數) 被放在 private 區

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;
  ...
}
```

ctors 放在 private 區

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 functions (常量成員函數)

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&);
};
```

→如果没写 const 这里，
那下边，就不能调用了.

O

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

?!

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

→内容不能动了.

參數傳遞：pass by value vs. pass by reference (to const)

C++ 才有的 vs. C

传的其实就是 point (4bytes)

1

-> by value -> stack -> 传进去

```
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;

  c2 += c1;
  cout << c2;
}
```

2-7

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

返回值傳遞：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&);
};
```

→比直接用更慢一点

Friend.

在这里，其它的 Complex 可以用这里边的 private member.

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

相同 class 的各個 objects 互為 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);
}
```

class body 外的各種定義 (definitions)

什麼情況下可以 pass by reference

什麼情況下可以 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

operator overloading (操作符重載-1, 成員函數) 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;
}
```

作用在左边身上

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

Pointer

感觉和python 一样.

自己写时 不用这个this

return by reference 語法分析

傳遞者無需知道接收者是以 reference 形式接收

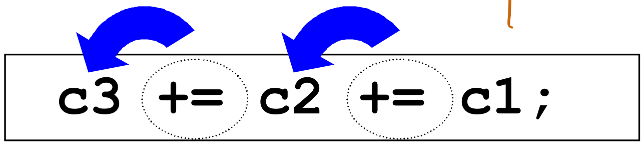
2-1

```
inline complex&
__doapl(complex* ths, const complex& r)
{
   ...
   return *ths;
}

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

pointer

object

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

   c2 += c1;
}
```



```
c3 += c2 += c1;
```

順序: c2 += c1
c3 += c2

class body 之外的各種定義 (definitions)

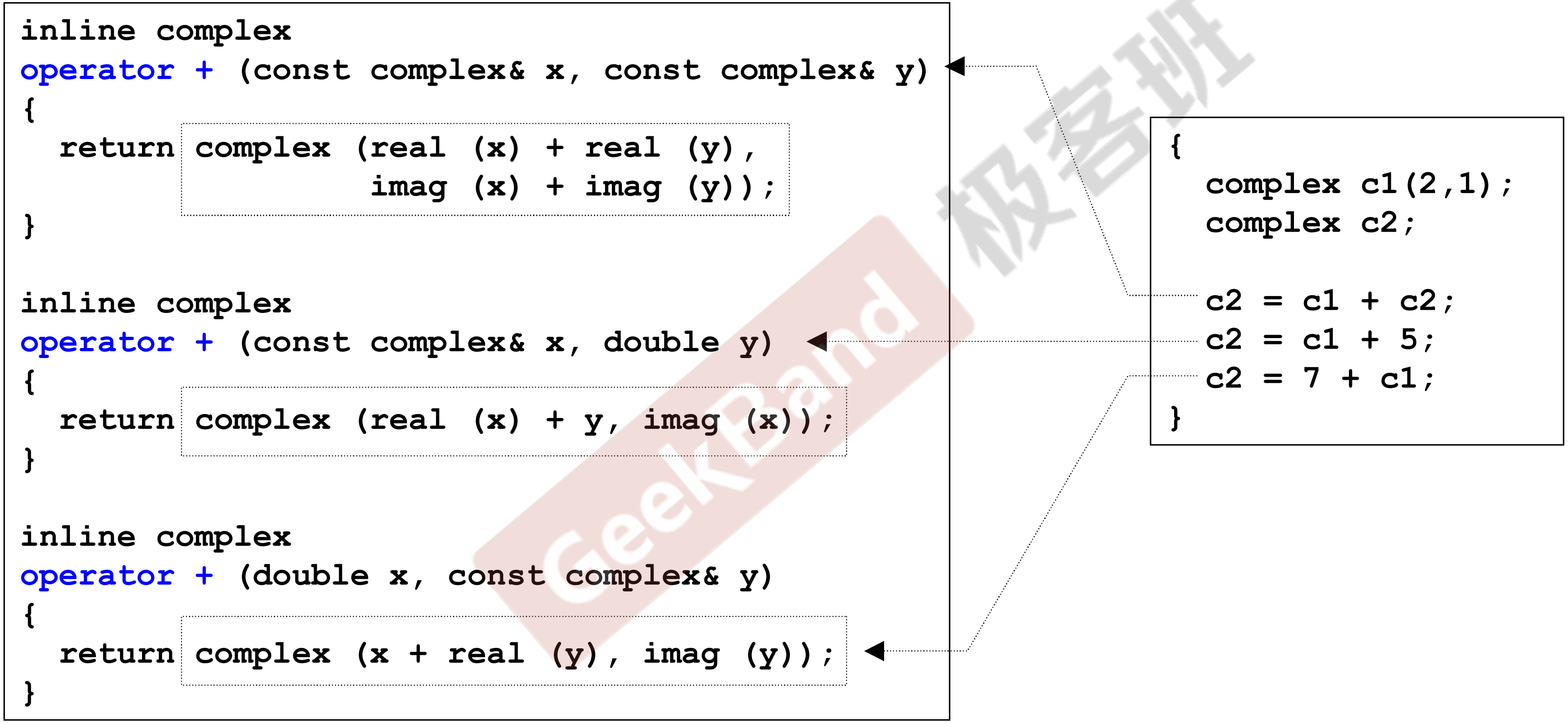
2-2

```
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 (操作符重載-2, 非成員函數) (無 this)

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));
}
```

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

  c2 = c1 + c2;
  c2 = c1 + 5;
  c2 = 7 + c1;
}
```

temp object (臨時對象) *typename ();*

创建

temp object ?

R-value ?

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);
}
```

class body 之外的各種定義 (definitions)

2-4

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

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

只有个 input variable.

negate
反相
(取反)

```
{
  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**。

operator overloading (操作符重載), 非成員函數

2-5

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

inline bool
operator == (const complex& x, double y)
{
  return real (x) == y && imag (x) == 0;
}

inline bool
operator == (double x, const complex& y)
{
  return x == real (y) && imag (y) == 0;
}
```

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

  cout << (c1 == c2);
  cout << (c1 == 2);
  cout << (0 == c2);
}
```

operator overloading (操作符重載), 非成員函數

2-6

```
inline bool
operator != (const complex& x,
             const complex& y)
{
  return real (x) != real (y)
      || imag (x) != imag (y);
}

inline bool
operator != (const complex& x, double y)
{
  return real (x) != y || imag (x) != 0;
}

inline bool
operator != (double x, const complex& y)
{
  return x != real (y) || imag (y) != 0;
}
```

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

  cout << (c1 != c2);
  cout << (c1 != 2);
  cout << (0 != c2);
}
```

operator overloading (操作符重載), 非成員函數

2-7

共軛複數

```
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);
}
```

作用到左边的身上、

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

?!

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

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

編程示例

```
#ifndef __COMPLEX__
#define __COMPLEX__

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&);
};

#endif
```

編程-動畫

編程-動畫

```
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);
}
```

編程-動畫

```
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));
}
```

編程-動畫

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

?!

```
complex c1(9,8);
cout << c1;
~~c1 << cout;~~

cout << c1 << endl;
```

你將獲得的代碼

complex.h
complex-test.cpp

string.h
string-test.cpp

Classes 的兩個經典分類

- Class without pointer member(s)

  complex

- Class with pointer member(s)

  string

String class

string.h

```
#ifndef __MYSTRING__
#define __MYSTRING__
```

1

```
class String
{
...
};
```

2

```
String::function(...) ...

Global-function(...) ...
```

```
#endif
```

string-test.cpp

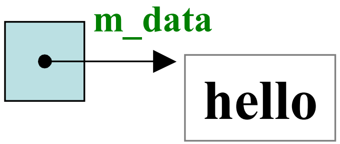
```
int main()
{
  String s1(),
  String s2("hello");

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

  cout << s3 << endl;
}
```

Big Three, 三個特殊函數

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;
};
```

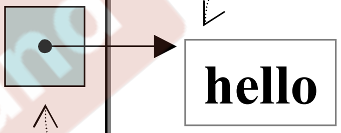
big three.



ctor 和 dtor (構造函數 和 析構函數)

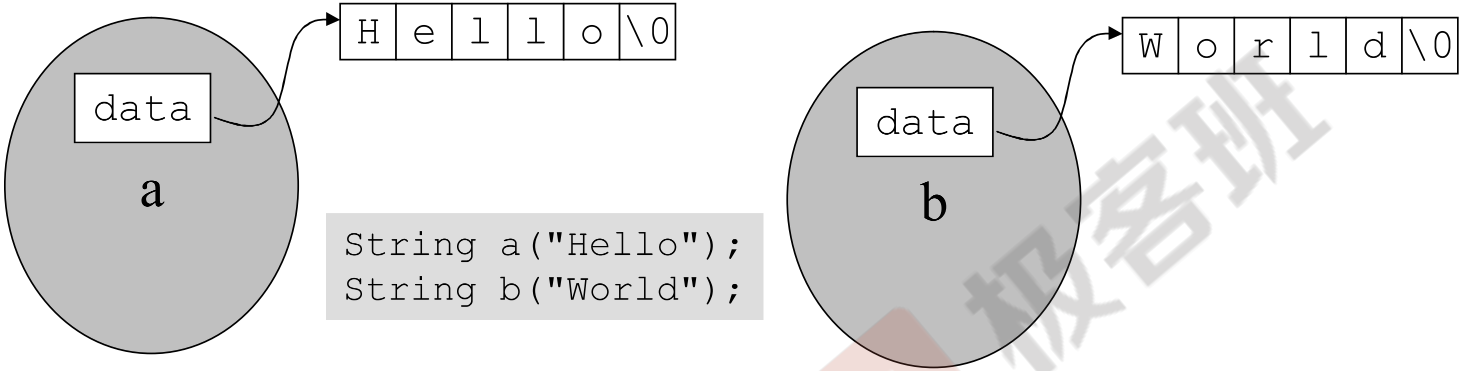
2-1

```
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;
}
```



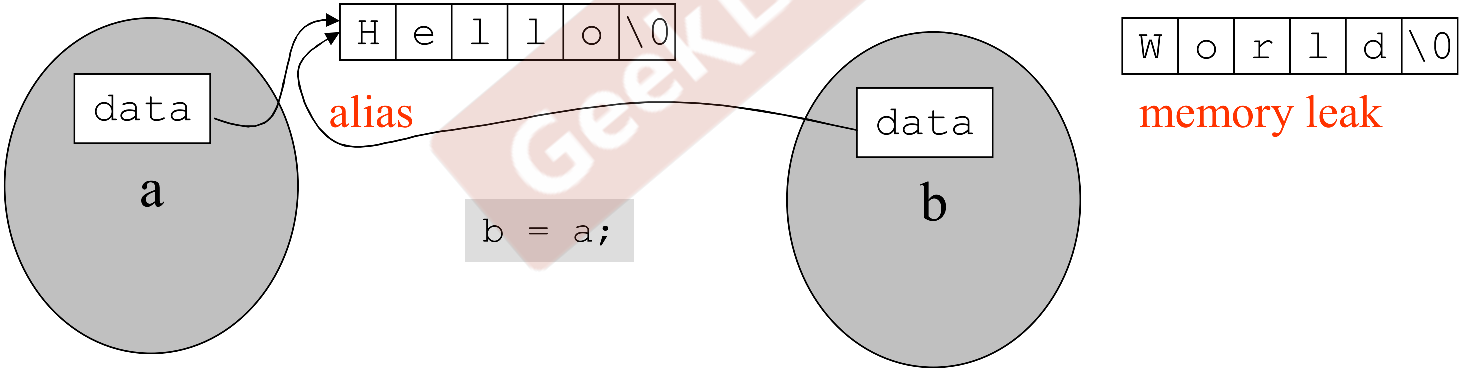
```
{
  String s1(),
  String s2("hello");

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

class with pointer members 必須有 copy ctor 和 copy op=



```
String a("Hello");
String b("World");
```

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



alias

memory leak

```
b = a;
```

copy ctor (拷貝構造函數)

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)

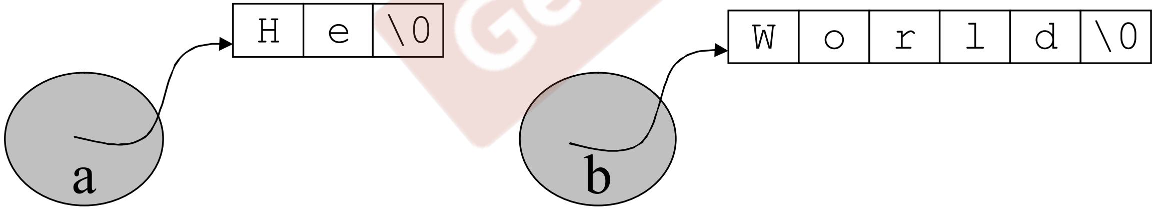
copy assignment operator (拷貝賦值函數)

2-3

```
inline
String& String::operator=(const String& str)
{
    if (this == &str)
        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;
}
```

檢測自我賦值
(self assignment)

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



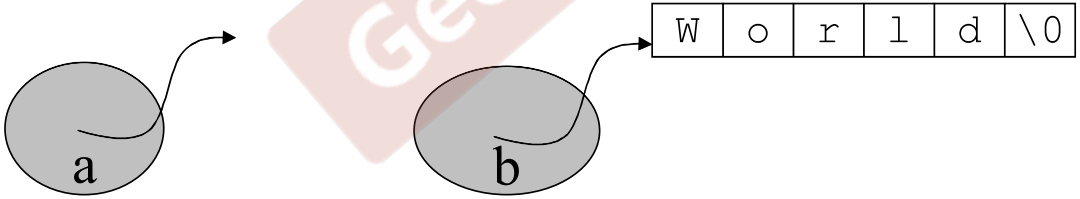
copy assignment operator (拷貝賦值函數)

2-3

```
inline
String& String::operator=(const String& str)
{
    if (this == &str)
        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;
}
```

檢測自我賦值
(self assignment)

1

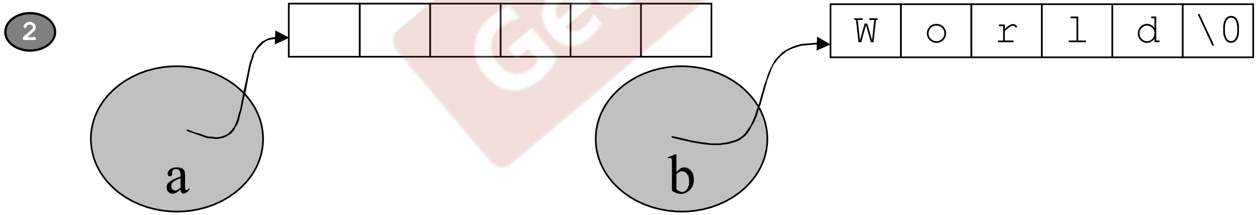


copy assignment operator (拷貝賦值函數)

2-3

```
inline
String& String::operator=(const String& str)
{
    if (this == &str)
        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;
}
```

檢測自我賦值
(self assignment)



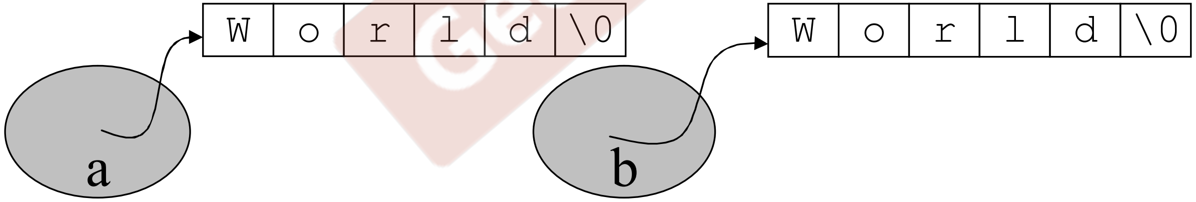
copy assignment operator (拷貝賦值函數)

2-3

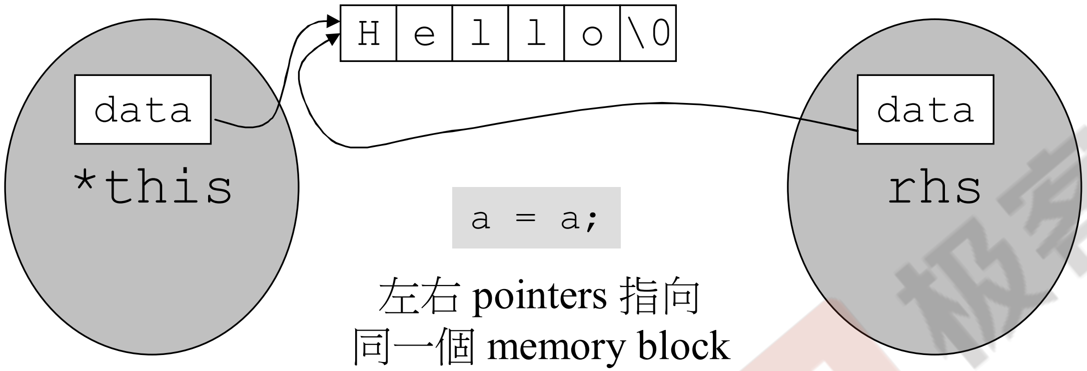
```
inline
String& String::operator=(const String& str)
{
   if (this == &str)
      return *this;

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

檢測自我賦值
(self assignment)

3



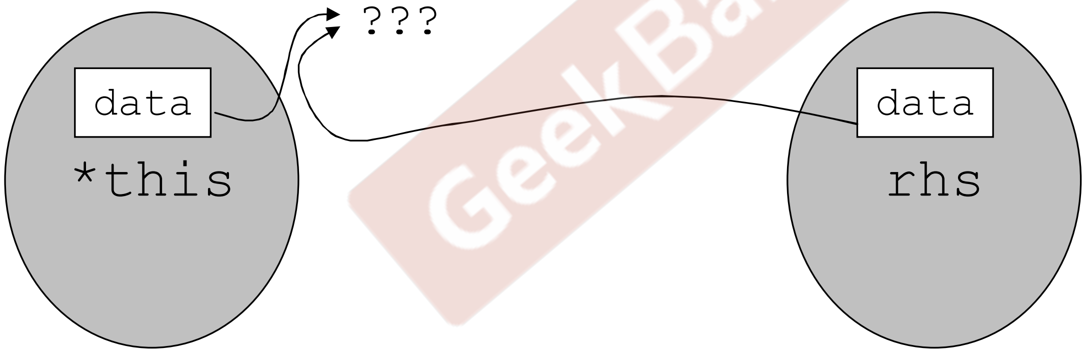
一定要在 operator= 中檢查是否 self assignment



```
a = a;
```

左右 pointers 指向
同一個 memory block

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 函數

2-4

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

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

所謂 stack (棧), 所謂 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);
}
```

Scope 之后 p就没用了

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

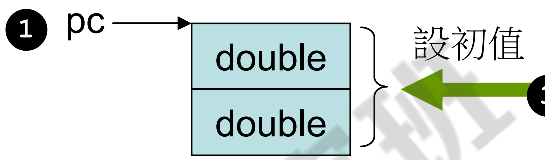
```
class Complex { … };
...

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

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

```
class Complex { … };
...

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

以上出現內存洩漏 (memory leak)，因為當作用域結束，**p** 所指的 heap object 仍然存在，但指針 **p** 的生命卻結束了，作用域之外再也看不到 **p**（也就沒機會 **delete p**）

new：先分配 memory, 再調用 ctor



```
class Complex
{
public:
  Complex(...) {...}
...
private:
  double m_real;
  double m_imag;
};
```

```
Complex* pc = new Complex(1,2);
```

編譯器轉化為

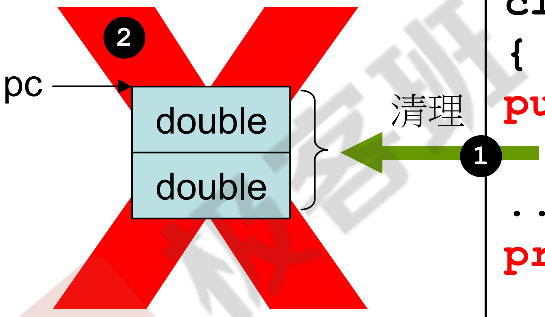
其內部調用 malloc(*n*)

```
Complex *pc;

1 void* mem = operator new( sizeof(Complex) ); //分配內存
2 pc = static_cast<Complex*>(mem);             //轉型
3 pc->Complex::Complex(1,2);                   //構造函數
```

```
Complex::Complex(pc,1,2);
```

this

delete：先調用 dtor, 再釋放 memory



```
class Complex
{
public:
   ~Complex() {...}
...
private:
   double m_real;
   double m_imag;
};
```

```
Complex* pc = new Complex(1,2);
...
delete pc;
```

編譯器轉化為

```
❶ Complex::~Complex(pc);  // 析構函數
❷ operator delete(pc);    // 釋放內存
```

其內部調用 free(pc)

new：先分配 memory, 再調用 ctor

```
String* ps = new String("Hello");
```

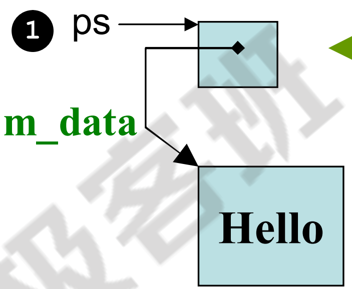
編譯器轉化為

其內部調用 malloc(*n*)

```
String* ps;

❶ void* mem = operator new( sizeof(String) );  //分配內存
❷ ps = static_cast<String*>(mem);              //轉型
❸ ps->String::String("Hello");                 //構造函數
```

```
String::String(ps,"Hello");
```

this



```
class String
{
public:
 ❸ String(...)
    {...
     m_data =
     new char[n];
     ...
    }
private:
    char* m_data;
};
```

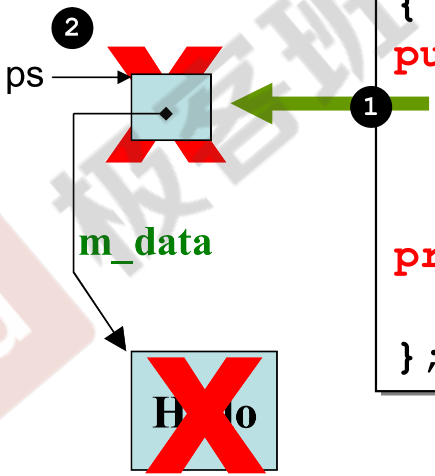
delete：先調用 dtor, 再釋放 memory

```
String* ps = new String("Hello");
...
delete ps;
```

編譯器轉化為

```
❶ String::~String(ps);      // 析構函數
❷ operator delete(ps);      // 釋放內存
```

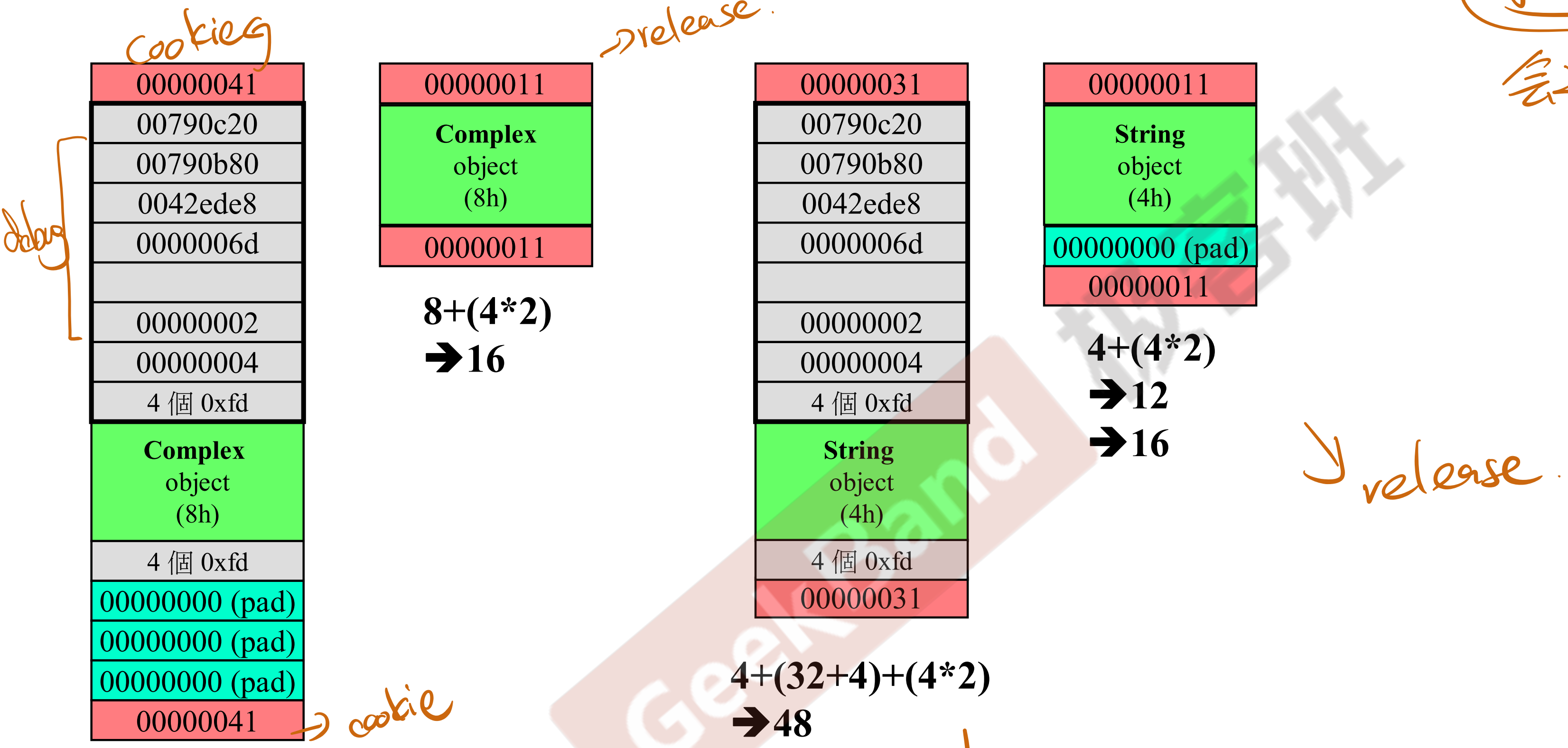
其內部調用 free(ps)



```
class String
{
public:
   ~String()
   { delete[] m_data; }
   ...
private:
   char* m_data;
};
```

動態分配所得的內存塊 (memory block), in VC

堆 heap 会这样.



cookie

debug

00000041
00790c20
00790b80
0042ede8
0000006d

00000002
00000004
4 個 0xfd
Complex object (8h)
4 個 0xfd
00000000 (pad)
00000000 (pad)
00000000 (pad)
00000041

cookie

8+(32+4)+(4*2)
→52
→64

→release.

00000011
Complex object (8h)
00000011

8+(4*2)
→16

00000031
00790c20
00790b80
0042ede8
0000006d

00000002
00000004
4 個 0xfd
String object (4h)
4 個 0xfd
00000031

4+(32+4)+(4*2)
→48

debug

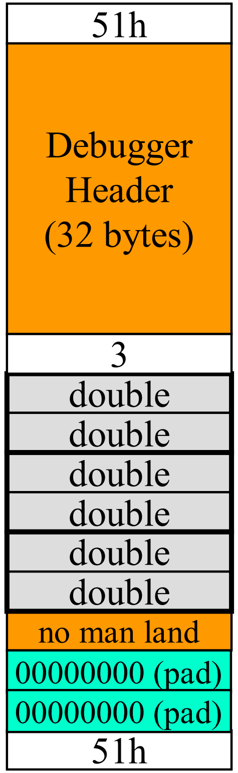
00000011
String object (4h)
00000000 (pad)
00000011

4+(4*2)
→12
→16

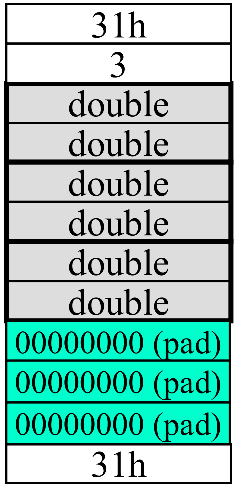
release.

動態分配所得的 array

Complex* p = new Complex[3];

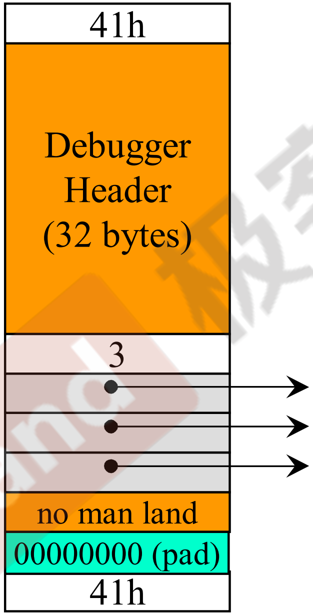


(8*3)+(32+4)+(4*2)+4
→72
→80

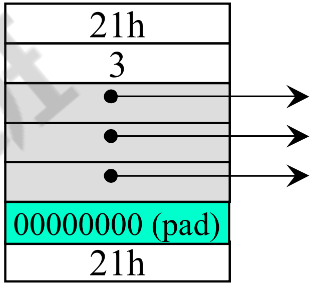


(8*3)+(4*2)+4
→36
→48

String* p = new String[3];



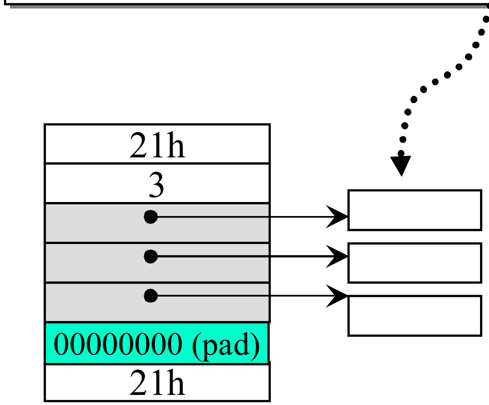
(4*3)+(32+4)+(4*2)+4
→60
→64



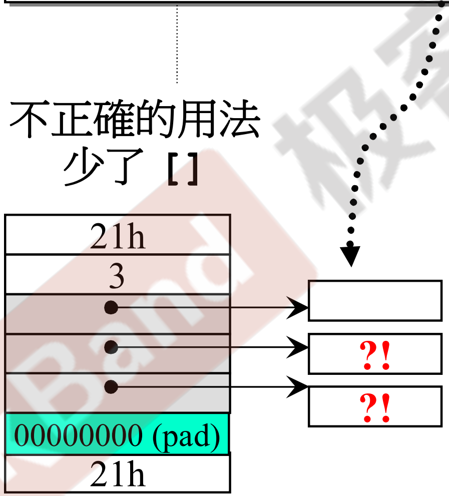
(4*3)+(4*2)+4
→24
→32

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 [ ]

編程示例

編程-動畫

```
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 (構造函數 和 析構函數)

```
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;
}
```

copy ctor (拷貝構造函數)

編程-動畫

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

copy assignment operator (拷貝賦值函數)

編程-動畫

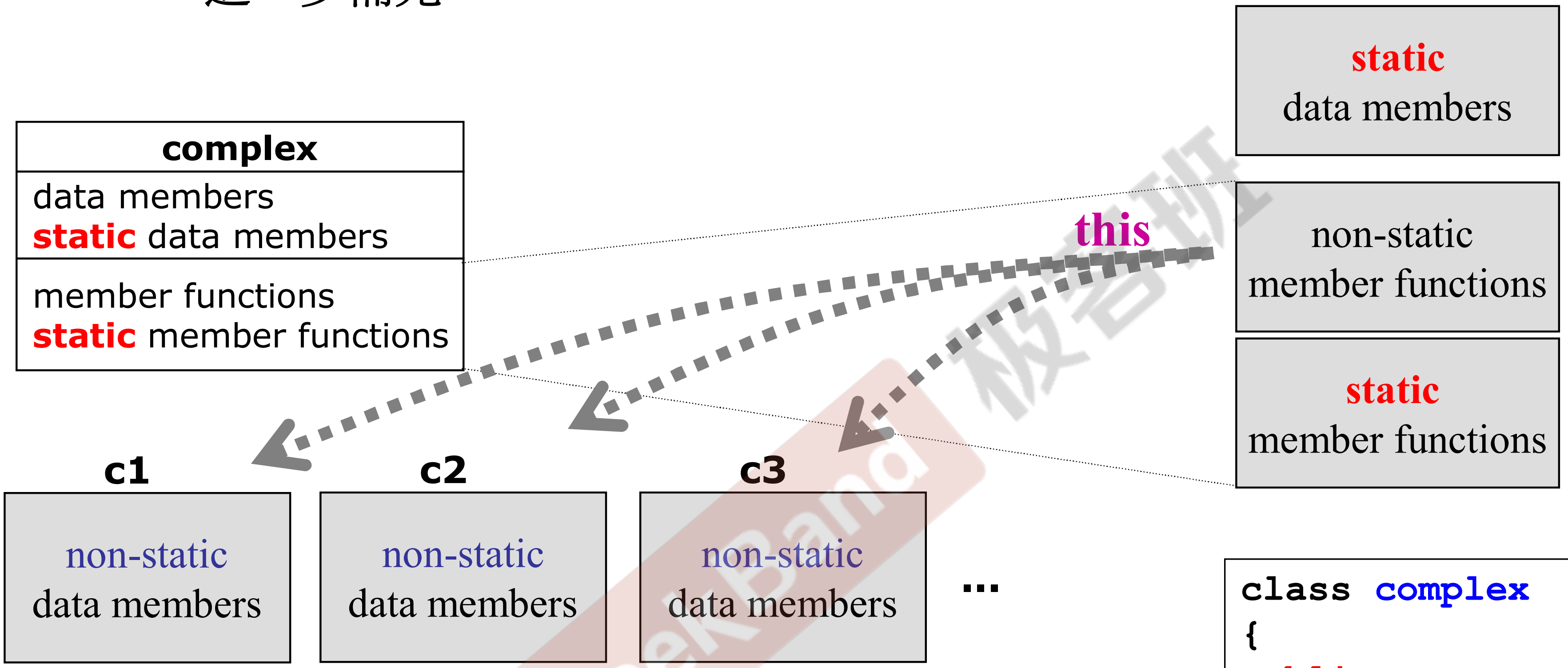
```
inline
String& String::operator=(const String& str)
{
   if (this == &str)
      return *this;

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

你將獲得的代碼

complex.h
complex-test.cpp

string.h
string-test.cpp

進一步補充：static



complex

data members
static data members

member functions
static member functions

static
data members

non-static
member functions

static
member functions

this

c1

non-static
data members

c2

non-static
data members

c3

non-static
data members

...

```
complex c1,c2,c3;
cout << c1.real();
cout << c2.real();
```

```
complex c1,c2,c3;
cout << complex::real(&c1);
cout << complex::real(&c2);
```

this

```
class complex
{
public:
  double real () const
     { return this->re; }
private:
  double re, im;
};
```

進一步補充：static

```
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);
}
```

→ static 在 class 外边

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

進一步補充：把 ctors 放在 private 區

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();
```

進一步補充：把 ctors 放在 private 區

Singleton

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

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

進一步補充：cout

```
class _IO_ostream_withassign
   : public ostream {
...
};

extern _IO_ostream_withassign cout;
```

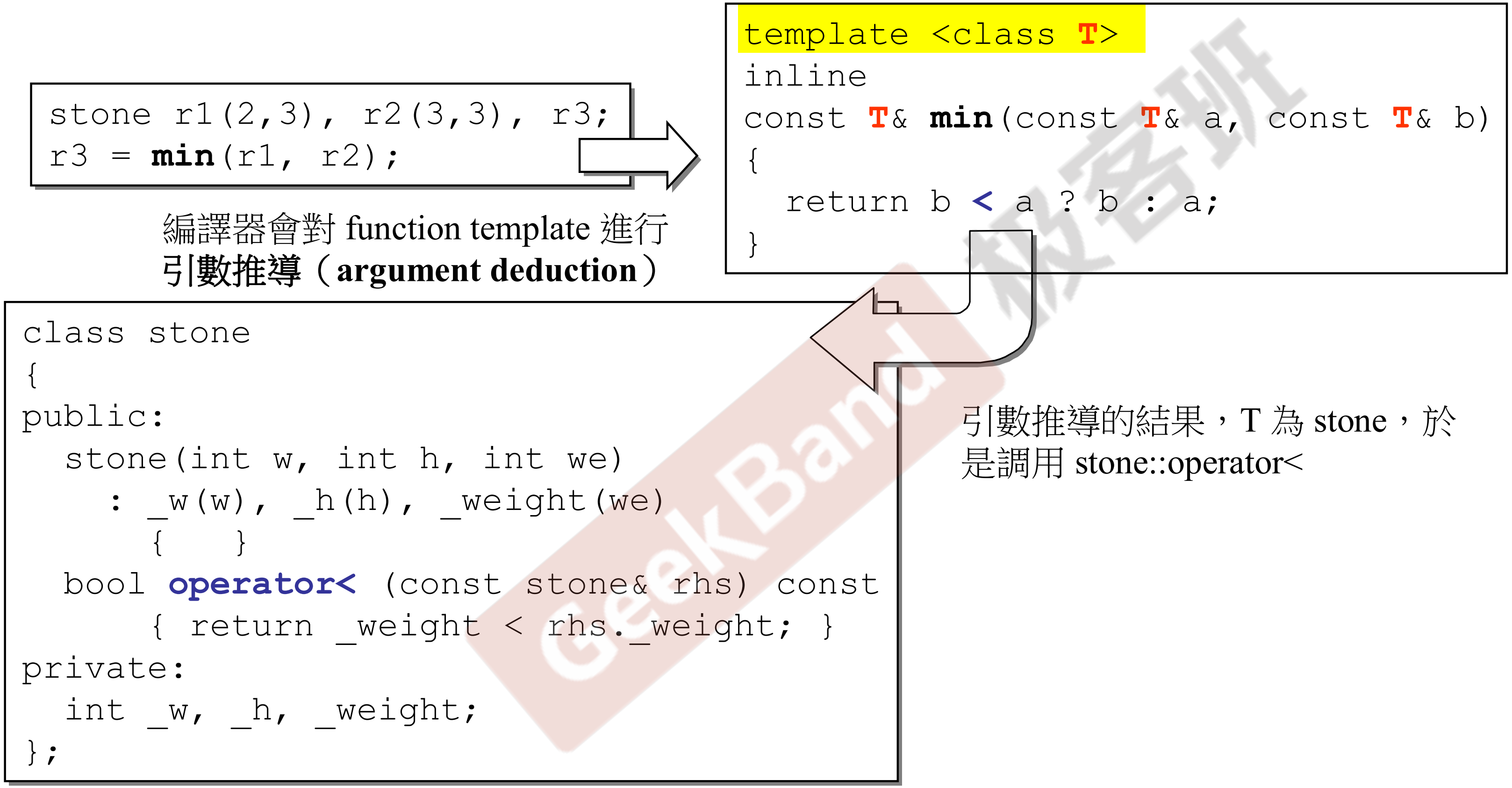
```
class ostream : virtual public ios
{
  public:
    ostream& operator<<(char c);
    ostream& operator<<(unsigned char c) { return (*this) << (char)c; }
    ostream& operator<<(signed char c) { return (*this) << (char)c; }
    ostream& operator<<(const char *s);
    ostream& operator<<(const unsigned char *s)
       { return (*this) << (const char*)s; }
    ostream& operator<<(const signed char *s)
       { return (*this) << (const char*)s; }
    ostream& operator<<(const void *p);
    ostream& operator<<(int n);
    ostream& operator<<(unsigned int n);
    ostream& operator<<(long n);
    ostream& operator<<(unsigned long n);
    ...
}
```

進一步補充：class template, 類模板

```
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);
  ...
}
```

進一步補充：function template, 函數模板



```
stone r1(2,3), r2(3,3), r3;
r3 = min(r1, r2);
```

編譯器會對 function template 進行
引數推導（argument deduction）

```
template <class T>
inline
const T& min(const T& a, const T& b)
{
  return b < a ? b : a;
}
```

```
class stone
{
public:
  stone(int w, int h, int we)
    : _w(w), _h(h), _weight(we)
      {  }
  bool operator< (const stone& rhs) const
      { return _weight < rhs._weight; }
private:
  int _w, _h, _weight;
};
```

引數推導的結果，T 為 stone，於是調用 stone::operator<

進一步補充：namespace

```
namespace std
{
   ...
}
```

using directive

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

int main()
{
  cin << ...;
  cout << ...;

  return 0;
}
```

using declaration

```
#include <iostream.h>
using std::cout;

int main()
{
  std::cin << ...;
  cout << ...;

  return 0;
}
```

```
#include <iostream.h>


int main()
{
  std::cin << ;
  std::cout << ...;

  return 0;
}
```

更多細節與深入

- •operator *type*() const;
- •explicit complex(…) : *initialization list* { }
- •pointer-like object ?
- •function-like object ?
- •Namespace
- •template specialization ?
- •Standard Library
- •variadic template (since C++11)
- •move ctor (since C++11)
- •Rvalue reference (since C++11)
- •auto (since C++11)
- •lambda (since C++11)
- •range-base for loop (since C++11)
- •unordered containers (Since C++)
- •…

革命尚未成功

同志仍需努力

Object Oriented Programming, Object Oriented Design

OOP, OOD

- **Inheritance (繼承)**
- **Composition (複合)**
- **Delegation (委託)**

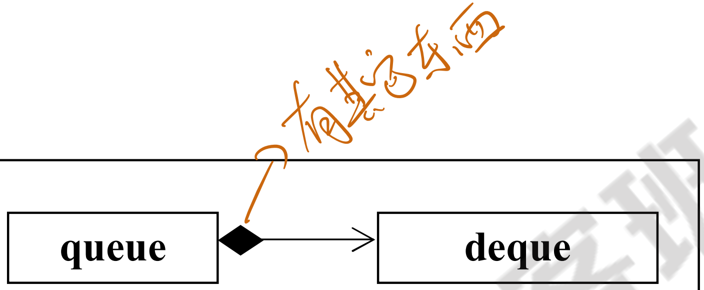
Composition (複合), 表示 has-a

Composition = has a

```
template <class T, class Sequence = deque<T> >
class queue {
   ...
protected:
  Sequence c;     // 底層容器
public:
  // 以下完全利用 c 的操作函數完成
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  reference back() { return c.back(); }
  // deque 是兩端可進出，queue 是末端進前端出（先進先出）
  void push(const value_type& x) { c.push_back(x); }
  void pop() { c.pop_front(); }
};
```

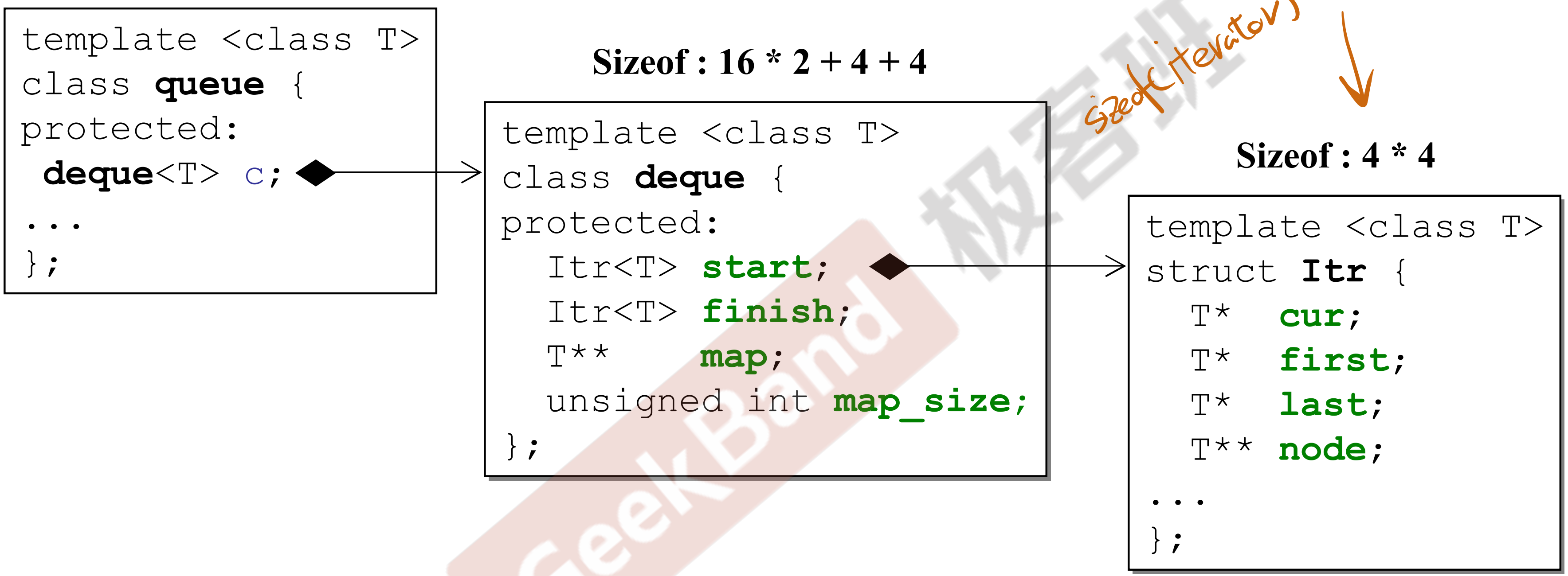
Composition (複合), 表示 has-a

Adapter



```
template <class T>
class queue {
  ...
protected:
  deque<T> c;       // 底層容器
public:
  // 以下完全利用 c 的操作函數完成
  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(); }
};
```

Composition (複合), 表示 has-a



Sizeof : 40

```
template <class T>
class queue {
protected:
 deque<T> c;
...
};
```

Sizeof : 16 * 2 + 4 + 4

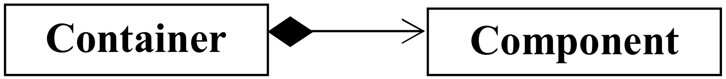
```
template <class T>
class deque {
protected:
   Itr<T> start;
   Itr<T> finish;
   T**    map;
   unsigned int map_size;
};
```

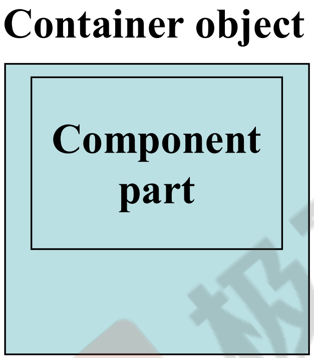
sizeof(iterator)=16

Sizeof : 4 * 4

```
template <class T>
struct Itr {
   T*   cur;
   T*   first;
   T*   last;
   T**  node;
...
};
```

Composition (複合) 關係下的構造和析構





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() };
```

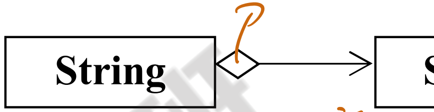
Delegation (委託). Composition by reference.

Handle / Body
(pImpl)

```
// file String.hpp
class StringRep;
class String {
public:
     String();
     String(const char* s);
     String(const String& s);
     String &operator=(const String& s);
     ~String();
. . . .
private:
     StringRep* rep; // pimpl
};
```

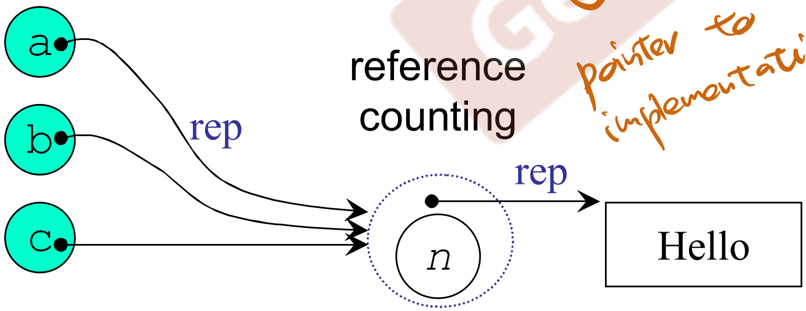
Big THREE

pointer to implementation



这样我们可以改变右边而不改变左边.

```
// file String.cpp
#include "String.hpp"
namespace {
class StringRep {
friend class String;
     StringRep(const char* s);
     ~StringRep();
     int count;
     char* rep;
};
}

String::String(){ ... }
...
```



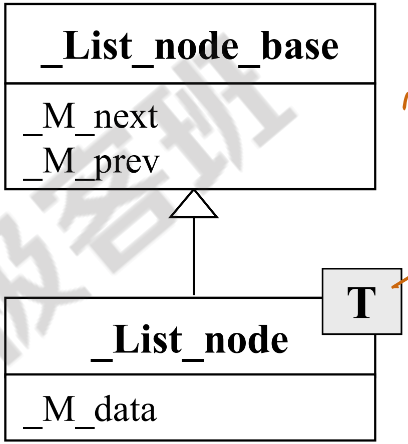
改变的时候 copy on write

Inheritance (繼承), 表示 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
{
  _Tp _M_data;
};
```

三种 Inheritance

-> public
-> private
-> protected

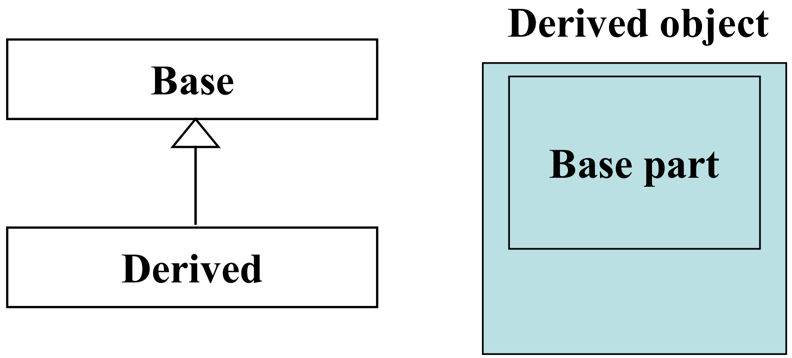


父类

-> Template<Template T>

子类

Inheritance (繼承) 關係下的構造和析構



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

!!

dstructor →外→内

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() { … };
```

Constructor

内→外

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() };
```

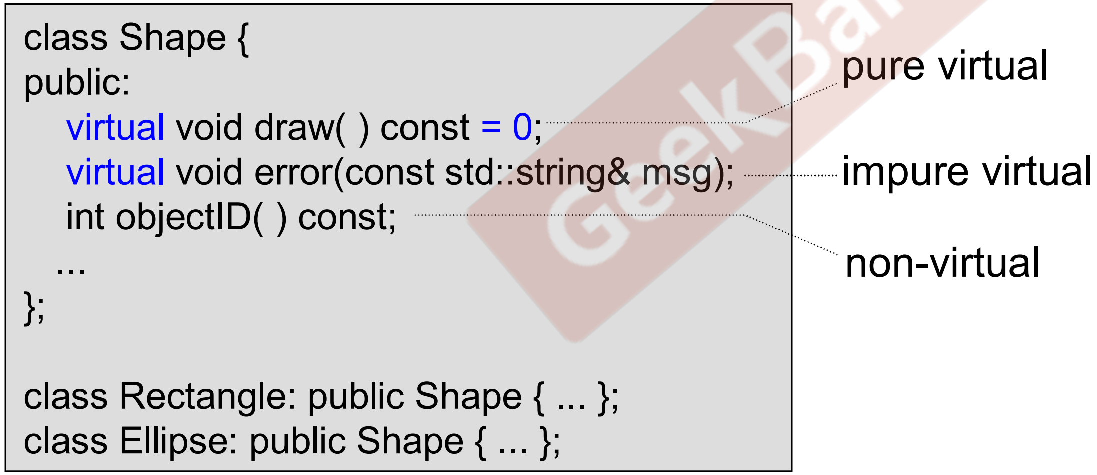
Inheritance (繼承) with virtual functions (虛函數)

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。

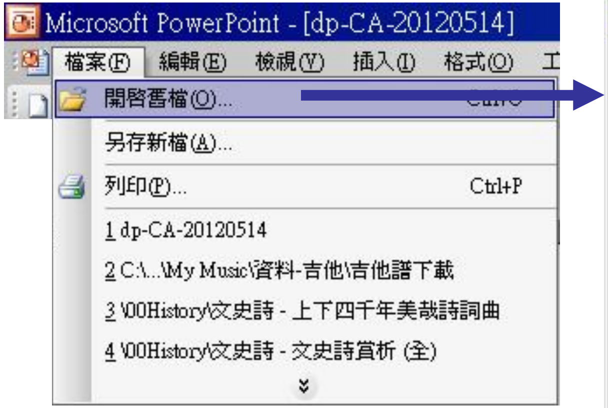
→ You have a defaut defination for it

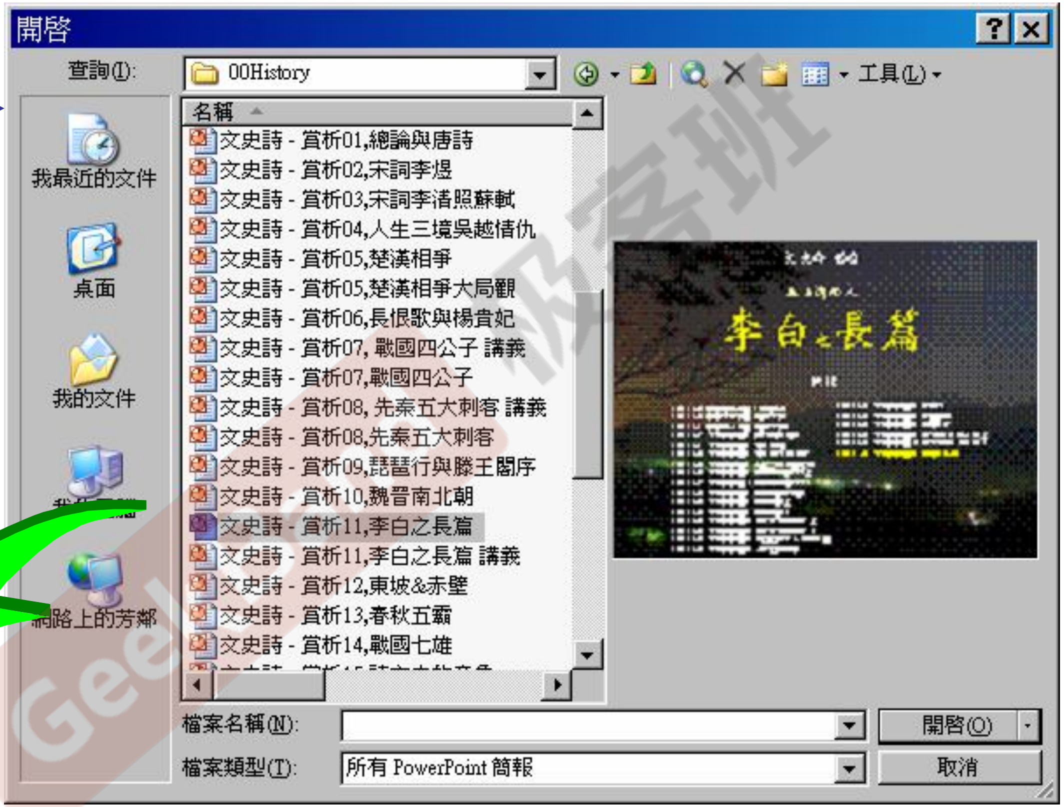
pure virtual 函數：你希望 derived class 一定要重新定義 (override 覆寫) 它，你對它沒有默認定義。



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

Inheritance (繼承) with virtual


Microsoft PowerPoint - [dp-CA-20120514]
檔案(F)
編輯(E)
檢視(V)
插入(I)
格式(O)
開啟舊檔(O)...
另存新檔(A)...
列印(P)... Ctrl+P
1 dp-CA-20120514
2 C:\...\My Music\資料-吉他\吉他譜下載
3 \00History\文史詩 - 上下四千年美哉詩詞曲
4 \00History\文史詩 - 文史詩賞析 (全)



文史詩 - 賞析01,總論與唐詩
文史詩 - 賞析02,宋詞李煜
文史詩 - 賞析03,宋詞李清照蘇軾
文史詩 - 賞析04,人生三境吳越情仇
文史詩 - 賞析05,楚漢相爭
文史詩 - 賞析05,楚漢相爭大局觀
文史詩 - 賞析06,長恨歌與楊貴妃
文史詩 - 賞析07, 戰國四公子 講義
文史詩 - 賞析07,戰國四公子
文史詩 - 賞析08, 先秦五大刺客 講義
文史詩 - 賞析08,先秦五大刺客
文史詩 - 賞析09,琵琶行與滕王閣序
文史詩 - 賞析10,魏晉南北朝
文史詩 - 賞析11,李白之長篇
文史詩 - 賞析11,李白之長篇 講義
文史詩 - 賞析12,東坡&赤壁
文史詩 - 賞析13,春秋五霸
文史詩 - 賞析14,戰國七雄

檔案名稱(N):

檔案類型(T): 所有 PowerPoint 簡報

Check file name

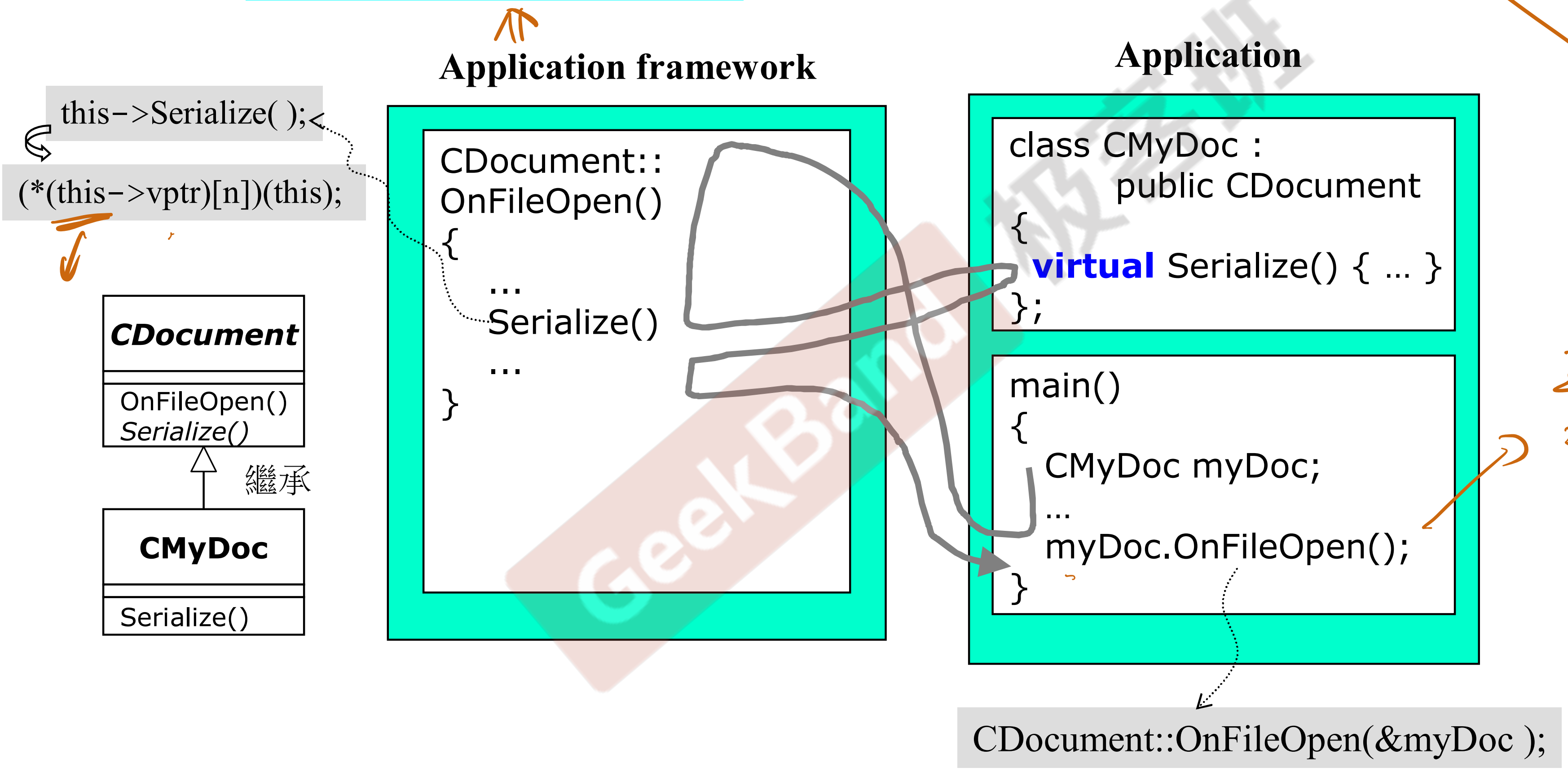
Search file

Open file

Inheritance (繼承) with virtual

Template Method

Template Method



```
this->Serialize( );
```

```
(*(this->vptr)[n])(this);
```

Application framework

```
CDocument::
OnFileOpen()
{
    ...
    Serialize()
    ...
}
```

Application

```
class CMyDoc :
        public CDocument
{
  virtual Serialize() { ... }
};
```

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

```
CDocument::OnFileOpen(&myDoc );
```

通过子类object
调用父类function

Inheritance (繼承), 表示 is-a

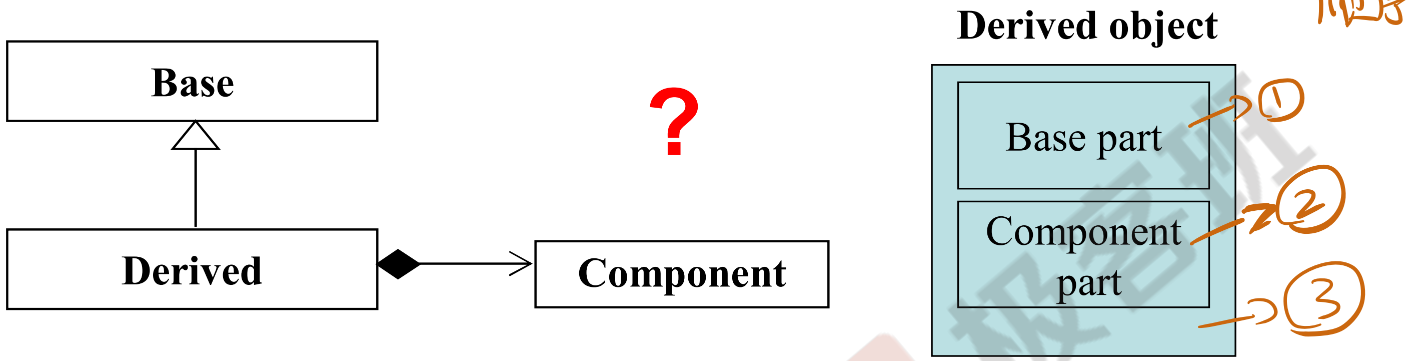
```
#include <iostream>
using namespace std;


class CDocument
{
public:
    void OnFileOpen()
    {
     // 這是個算法，每個 cout 輸出代表一個實際動作
     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() { };
};
```

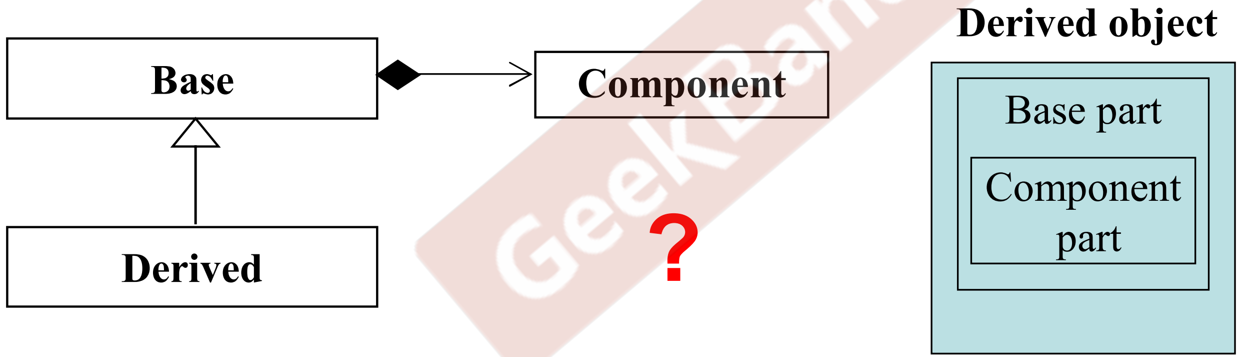
```
class CMyDoc : public CDocument
{
public:
    virtual void Serialize()
    {
      // 只有應用程序本身才知道如何讀取自己的文件(格式)
      cout << "CMyDoc::Serialize()" << endl;
    }
};
```

```
int main()
{
  CMyDoc myDoc;  // 假設對應[File/Open]
  myDoc.OnFileOpen();
}
```

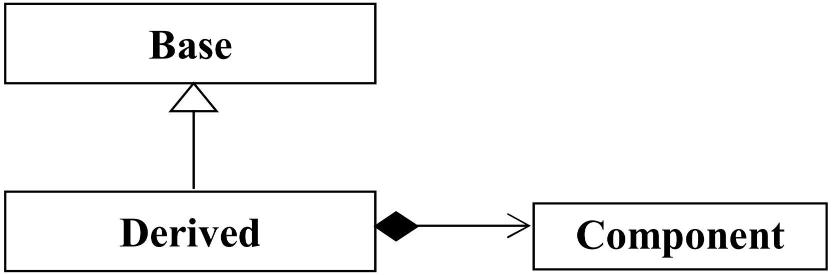
Inheritance+Composition 關係下的構造和析構

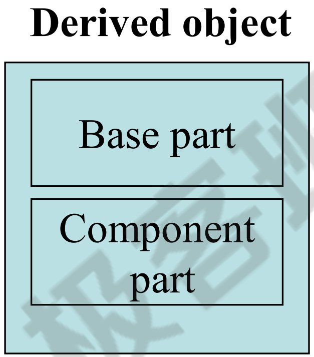
顺序 ctor ①②③
dtor ③②①





Inheritance+Composition 關係下的構造和析構





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。

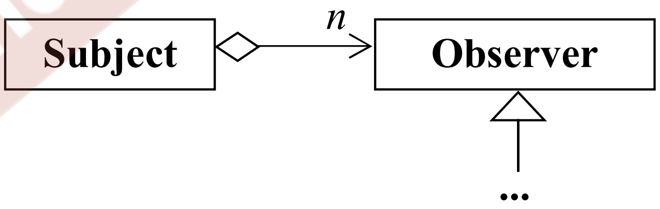
```
Derived::~Derived(…){ … ~Component(), ~Base() };
```

Delegation (委託) + Inheritance (繼承)

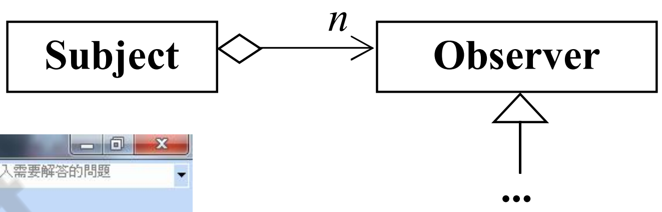
```
class Subject
{
   int m_value;
   vector<Observer*> m_views;
 public:
   void attach(Observer* obs)
   {
      m_views.push_back(obs);
   }
   void set_val(int value)
   {
      m_value = value;
      notify();
   }
   void notify()
   {
      for (int i = 0; i < m_views.size(); ++i)
       m_views[i]->update(this,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

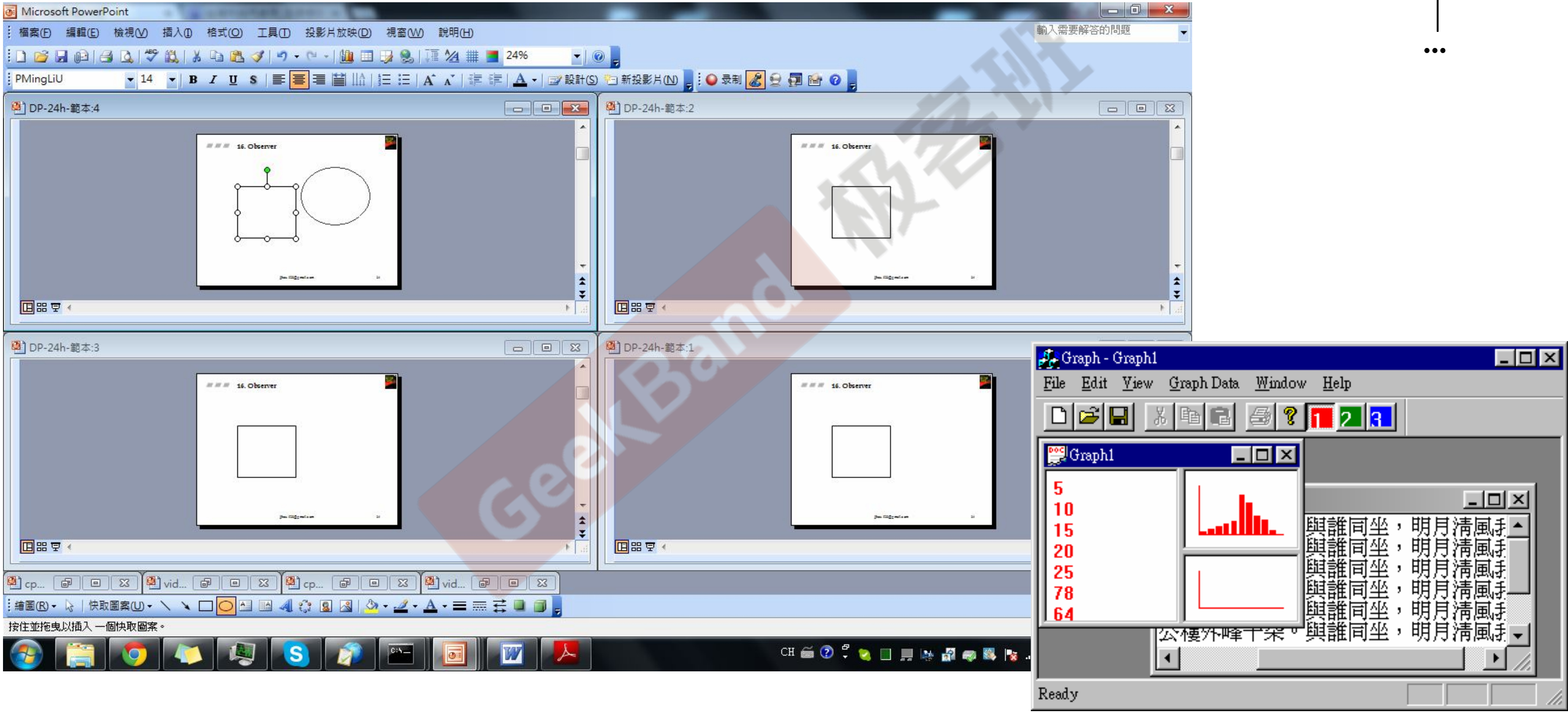
Observer

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



Delegation (委託) + Inheritance (繼承)

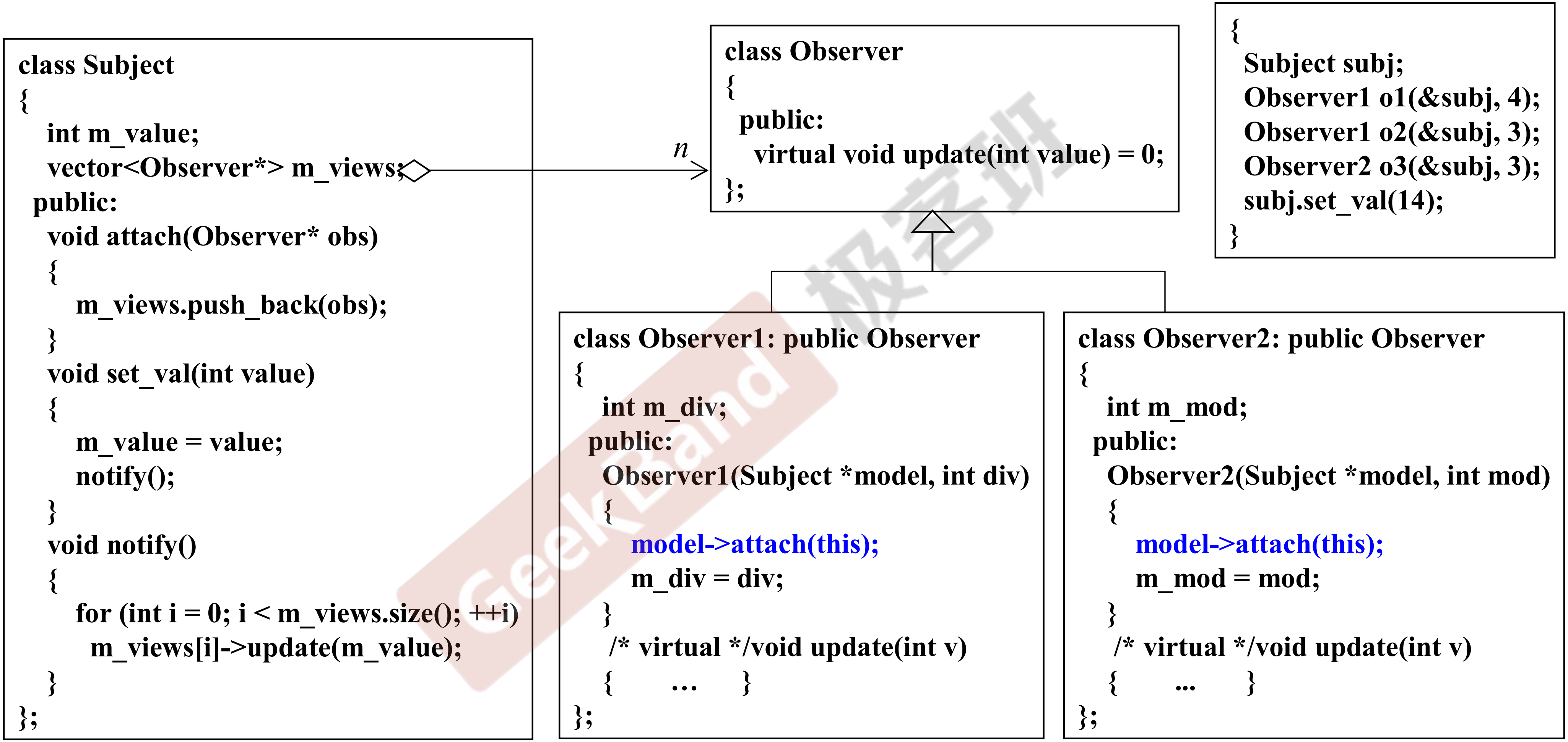




```
5
10
15
20
25
78
64
```

```
與誰同坐，明月清風
與誰同坐，明月清風
與誰同坐，明月清風
與誰同坐，明月清風
與誰同坐，明月清風
公樓外峰千朵，與誰同坐，明月清風
```

Delegation (委託) + Inheritance (繼承)



```
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(m_value);
   }
};
```

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

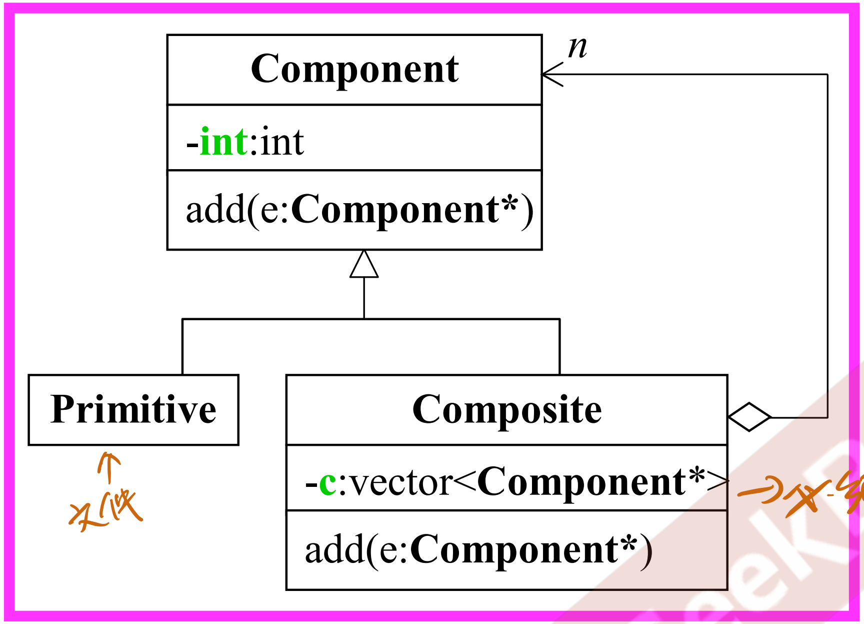
```
{
  Subject subj;
  Observer1 o1(&subj, 4);
  Observer1 o2(&subj, 3);
  Observer2 o3(&subj, 3);
  subj.set_val(14);
}
```

```
class Observer1: public Observer
{
   int m_div;
 public:
   Observer1(Subject *model, int div)
   {
      model->attach(this);
      m_div = div;
   }
   /* virtual */void update(int v)
   {      …      }
};
```

```
class Observer2: public Observer
{
   int m_mod;
 public:
   Observer2(Subject *model, int mod)
   {
      model->attach(this);
      m_mod = mod;
   }
   /* virtual */void update(int v)
   {      ...      }
};
```

Delegation (委託) + Inheritance (繼承) 写个 Window System

Composite



→ 个-须用pointer
↓
一样大小

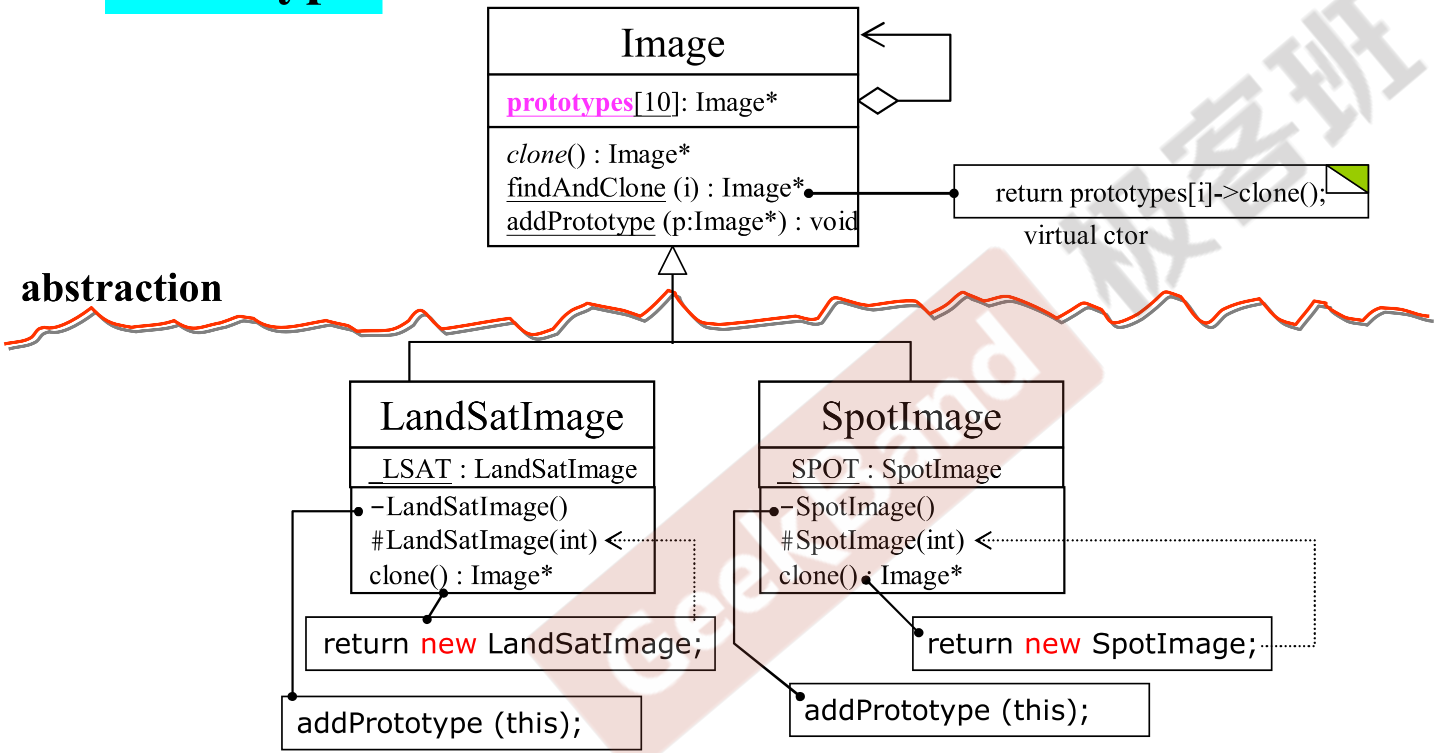
```
class Component
{
   int value;
 public:
   Component(int val)    {   value = val;   }
   virtual void add( Component* ) { }
};
```

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

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

```
class Primitive: public Component
{
 public:
   Primitive(int val): Component(val) {}
};
```

Delegation (委託) + Inheritance (繼承)

Prototype



Prototype

```
#include <iostream.h>
enum imageType
{
  LSAT, SPOT
};
class Image
{
  public:
    virtual void draw() = 0;
    static Image *findAndClone(imageType);
  protected:
    virtual imageType returnType() = 0;
    virtual Image *clone() = 0;
    // As each subclass of Image is declared, it registers its prototype
    static void addPrototype(Image *image)
    {
        _prototypes[_nextSlot++] = image;
    }
  private:
    // addPrototype() saves each registered prototype here
    static Image *_prototypes[10];
    static int _nextSlot;
};
Image *Image::_prototypes[];
int Image::_nextSlot;
```

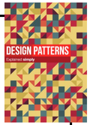
```
// Client calls this public static member function when it needs an instance
// of an Image subclass
Image *Image::findAndClone(imageType type)
{
  for (int i = 0; i < _nextSlot; i++)
    if (_prototypes[i]->returnType() == type)
      return _prototypes[i]->clone();
}
```

Prototype

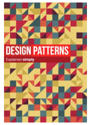
```
class LandSatImage: public Image
{
 public:
  imageType returnType()   {
     return LSAT;
  }
  void draw()   {
     cout << "LandSatImage::draw " << _id << endl;
  }
  // When clone() is called, call the one-argument ctor with a dummy arg
  Image *clone()   {
     return new LandSatImage(1);
  }
 protected:
  // This is only called from clone()
  LandSatImage(int dummy)   {
     _id = _count++;
  }
 private:
  // Mechanism for initializing an Image subclass - this causes the
  // default ctor to be called, which registers the subclass's prototype
  static LandSatImage _landSatImage;
  // This is only called when the private static data member is inited
  LandSatImage()   {
     addPrototype(this);
  }
  // Nominal "state" per instance mechanism
  int _id;
  static int _count;
};
// Register the subclass's prototype
LandSatImage LandSatImage::_landSatImage;
// Initialize the "state" per instance mechanism
int LandSatImage::_count = 1;
```

```
enum imageType
{ LSAT, SPOT };
```

```
class SpotImage: public Image
{
 public:
  imageType returnType()   {
     return SPOT;
  }
  void draw()   {
     cout << "SpotImage::draw " << _id << endl;
  }
  Image *clone()   {
     return new SpotImage(1);
  }
 protected:
  SpotImage(int dummy)  {
     _id = _count++;
  }
 private:
  SpotImage()   {
     addPrototype(this);
  }
  static SpotImage _spotImage;
  int _id;
  static int _count;
};
SpotImage SpotImage::_spotImage;
int SpotImage::_count = 1;
```



Prototype



```
// Simulated stream of creation requests
const int NUM_IMAGES = 8;
imageType input[NUM_IMAGES] =
{
  LSAT, LSAT, LSAT, SPOT, LSAT, SPOT, SPOT, LSAT
};
```

```
int main()
{
  Image *images[NUM_IMAGES];
  // Given an image type, find the right prototype, and return a clone
  for (int i = 0; i < NUM_IMAGES; i++)
    images[i] = Image::findAndClone(input[i]);
  // Demonstrate that correct image objects have been cloned
  for (i = 0; i < NUM_IMAGES; i++)
    images[i]->draw();
  // Free the dynamic memory
  for (i = 0; i < NUM_IMAGES; i++)
    delete images[i];
}
```

The End