C++面向對象編程 (C++Object-Oriented Programming)

課程簡介

這是我的所有 C++技術課程中最基礎最根本的一門課。

C++可說是第一個高度普及的 Object-Oriented (面向對象) 編程語言。"第一個"或 "最早的" 並非重點，重點是經過多年的淬煉和考驗 C++的影響深入各層面，擁有眾多使用者和極其豐富的資源 (書籍、論文、期刊、視頻、第三方程序庫…)。

作為 Object-Oriented (面向對象) 技術的主流語言，C++ 其實還擁有另一支編程主線：模板與泛型編程 (Templates and Generic Programming)。

本課程涵蓋上述兩條主線：Object-Oriented (面向對象) 和泛型編程 (Generic Programming)。

由於視頻錄製時程的因素，本課程分為 Part I 和 Part II.
Part I 主要討論 OO (面向對象) 的基礎概念和編程手法。基礎最是重要，勿在浮沙築高台，我著重的是大器與大氣。課程首先探討 Class without pointer members 和 Class with pointer members 兩大類型，而後晉昇至 OOP/OOD，包括 classes 之間最重要的三種關係：繼承(Inheritance)、複合(Composition)、委託(Delegation)。
Part II 繼續探討更多相關主題，並加上低階的對象模型 (Object Model)，以及高階的 Templates (模板) 編程。

本課程所談主題都隸屬 C++1.0 (C++98)；至於 C++ 2.0 (C++11/14) 帶來的嶄新內容則由我的另一門課程涵蓋。C++2.0 在語言和標準庫兩方面都帶來了很多新事物，份量足以形成另一門課程。

你將獲得整個 video 課程的完整講義 (也就是 video 呈現的每一張投影片畫面)，和完整的示例程序。你可以 (也必要) 在視聽過程中隨時停格思考和閱讀講義。

侯捷簡介：程序員，軟件工程師，臺灣工研院副研究員，教授，專欄主筆。曾任教於中壢元智大學、上海同濟大學、南京大學。著有《無責任書評》三卷、《深入淺出 MFC》、《STL 源碼剖析》…，譯有《Effective C++》《More Effective C++》《C++ Primer》《The C++ Standard Library》…。

以下這份不算太細緻的主題劃分，協助您認識整個課程內容。

C++面向對象編程 (C++Object-Oriented Programming)

Part I

- Introduction of C++98, TR1, C++11, C++14
  - Bibliography
  - Data and Functions, from C to C++
  - Basic forms of C++ programs
  - About output
  - Guard declarations of header files
  - Layout of headers
  - Object Based
  - Class without pointer member
    - Class declarations
    - Class template, introductions and overviews
    - What is 'this'
    - Inline functions
    - Access levels
    - Constructor (ctor)
    - Const member functions
    - Parameters : pass by value vs. pass by reference
    - Return values : return by value vs. return by reference
    - Friend
    - Definitions outside class body
    - Operator overloading, as member function
    - Return by reference, again
    - Operator overloading, as non-member function
    - Temporary objects
    - Expertise
    - Code demonstration
  - class with pointer members
    - The "Big Three"
      - Copy Constructor
      - Copy Assignment Operator
      - Destructor
    - Ctor and Dtor, in our String class

- "MUST HAVE" in our String class
  - Copy Constructor
  - Copy assignment operator
- Deal with "self assignment"
- Another way to deal with "self assignment" : Copy and Swap
- Overloading output operator (<<)
- Expertise
- Code demonstration

Objects from stack vs. objects from heap

- Objects lifetime
- new expression : allocate memory and then invoke ctor
- delete expression : invoke dtor and then free memory
- Anatomy of memory blocks from heap
- Allocate an array dynamically
- new[] and delete[]

MORE ISSUES :

- static
- private ctors
- cout
- Class template
- Function template
- namespace
- Standard Library : Introductions and Overviews

Object Oriented

- Composition means "has-a"
  - Construction : from inside to outside
  - Destruction : from outside to inside
- Delegation means "Composition by reference"
- Inheritance means "is-a"
  - Construction : from inside to outside
  - Destruction : from outside to inside
- Construction and Destruction, when Inheritance+Composition
- Inheritance with virtual functions
- Virtual functions typical usage 1 : Template Method
- Virtual functions typical usage 2 : Polymorphism
- Virtual functions inside out : vptr, vtbl, and dynamic binding
- Delegation + Inheritance : Observer
- Delegation + Inheritance : Composite

Delegation + Inheritance : Prototype

Part II

-- the end

C++面向對象程序設計

(Object Oriented Programming, OOP)

侯捷

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)

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

Object Oriented
(面向對象)

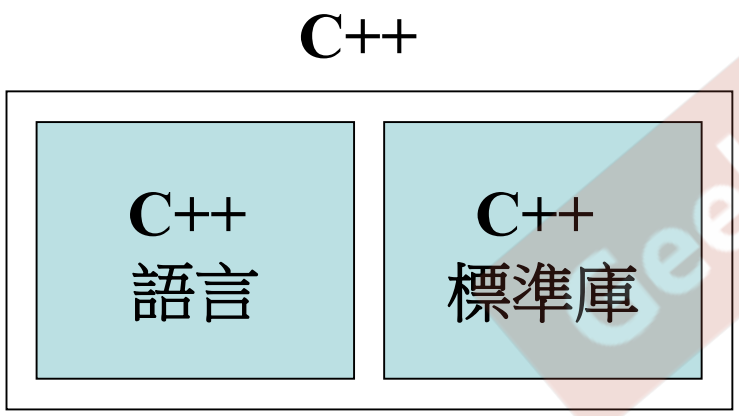
C++ 的歷史

- B 語言 (1969)
- C 語言 (1972)
- C++ 語言 (1983)
  (new C ➔ C with Class ➔ C++)

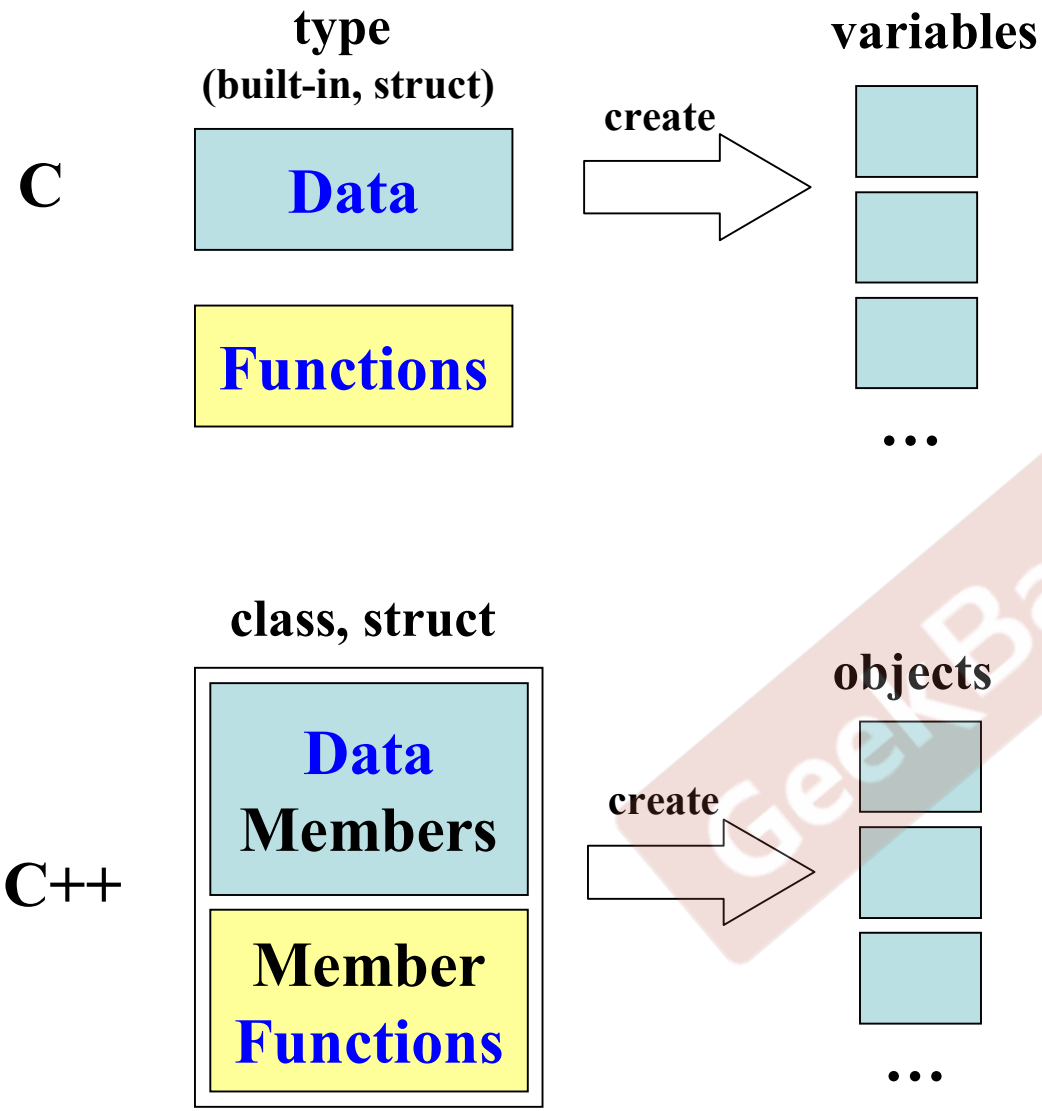
- Java 語言
- C# 語言

C++ 演化

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

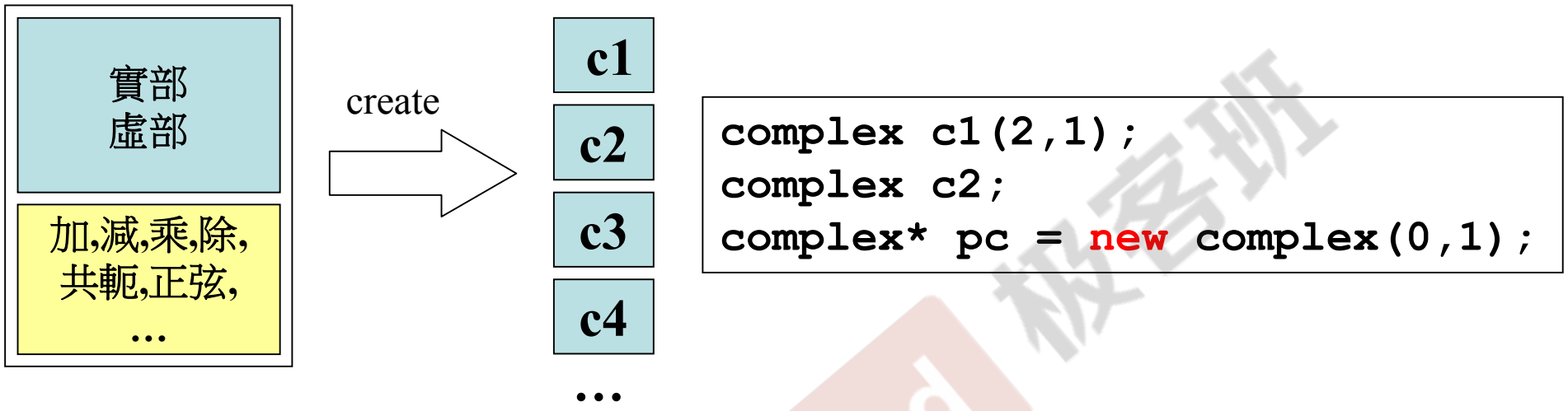


C vs. C++, 關於數據和函數



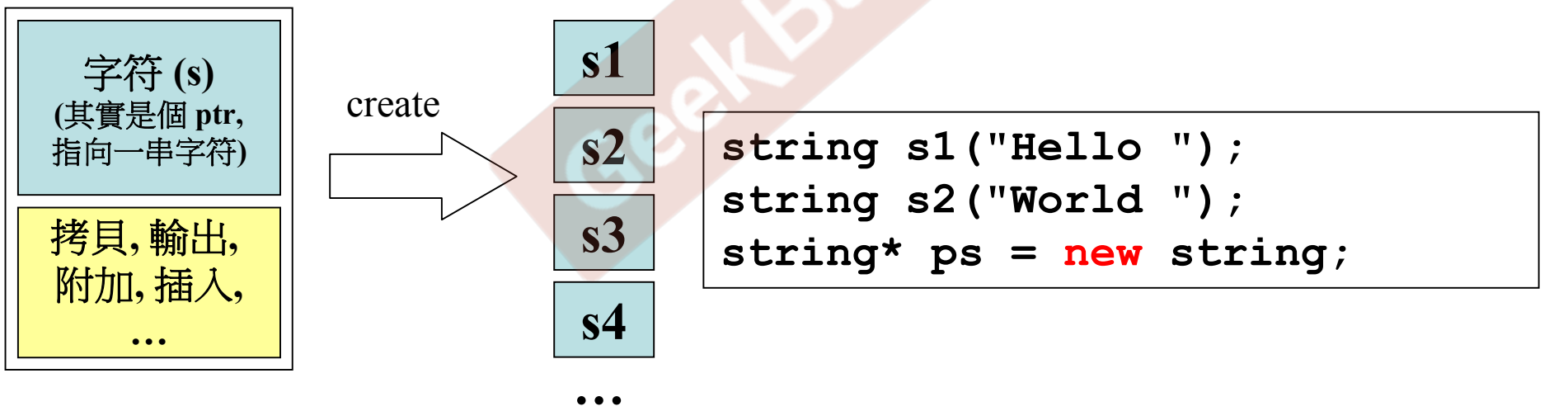
C++, 關於數據和函數

complex




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

string




```
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：

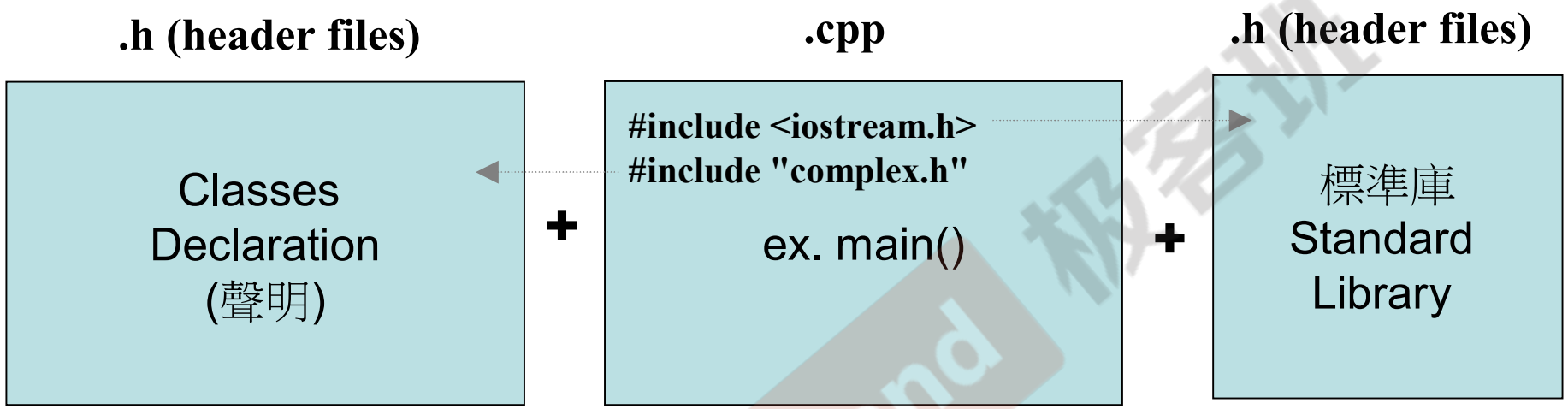
- **Class without pointer member(s)**

  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>
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)

```
#pragma once
```

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__

#include <cmath>

class ostream;
class complex;

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

class complex
{
...
};

complex::function ...

#endif
```

0 forward declarations (前置聲明)

1 class declarations (類 - 聲明)

2 class definition (類 - 定義)

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;
  ...
}
```

inline (內聯) 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&);
};
```

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 ();
}
```

编译器对 inline 函数的处理步骤:
* 将 inline 函数体复制到 inline 函数调用点处；
* 为所用 inline 函数中的局部变量分配内存空间；
* 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
* 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）

内联是以代码膨胀(复制)为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率

如果函数体内的代码比较长，或函数体内出现循环 则内联代价比较高。

内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联(不是指报错。
内联是在编译期建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码

access level (訪問級別)

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

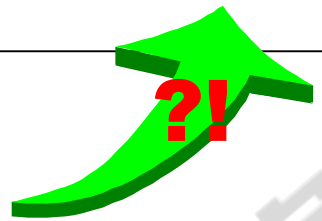
```
{
  complex c1(2,1);
  cout << c1.real();
  cout << c1.imag();
}
```

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);
  ...
}
```

ctor (構造函數) 可以有很多個 – overloading (重載)

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

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

上述给出了默认参数，无参ctor冲突，不允许 ?!

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

```
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;
  ...
}
```

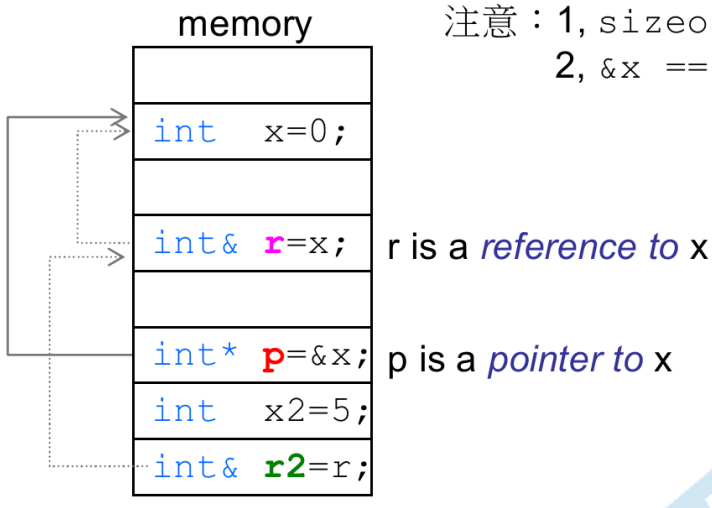
ctors 放在 private 區

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();
```

reference



注意：1, `sizeof(r) == sizeof(x)`
2, `&x == &r;`

object 和其 reference 的大小相同，地址也相同（全都是假象）

Java 裡頭所有變量都是 reference

```
int x=0;
int* p = &x;
int& r = x;   //r 代表 x。現在 r,x 都是0
int x2=5;

r = x2;         //r 不能重新代表其他物體。現在 r,x 都是 5
int& r2 = r; //現在 r2 是 5（r2 代表 r；亦相當於代表 x）
```

引用必须初始化，而指针可以不初始化。
引用不能为空，而指针可以为空。
引用不能更换目标

左值引用：常规引用，一般表示对象的身份。

右值引用：就是必须绑定到右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。
    可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding）
    消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
    能够更简洁明确地定义泛型函数。

引用折叠
X& &、X& &&、X&& & 可折叠成 X&
X&& && 可折叠成 X&&

C++中的引用只是C++对指针操作的一个“ 语法糖 ”，在底层实现时C++编译器实现这两种操作的方法完全相同

reference

```
typedef struct Stag {  int a,b,c,d; } S;
int main() {
  double  x=0;
  double* p = &x; //p指向x, p的值是x的地址
  double& r = x;  //r代表x, 現在r,x都是0

  cout << sizeof(x) << endl;   //8
  cout << sizeof(p) << endl;   //4
  cout << sizeof(r) << endl;   //8
  cout <<  p << endl;          //0065FDFC
  cout << *p << endl;          //0
  cout <<  x << endl;          //0
  cout <<  r << endl;          //0
  cout << &x << endl;          //0065FDFC
  cout << &r << endl;          //0065FDFC

  S s;
  S& rs = s;
  cout << sizeof(s) << endl;    //16
  cout << sizeof(rs) << endl;   //16
  cout << &s << endl;           //0065FDE8
  cout << &rs << endl;          //0065FDE8
}
```

object 和其 reference 的大小相同，地址也相同（全都是假象）

reference 的常見用途

```
void func1(Cls* pobj) { pobj->xxx(); }
void func2(Cls  obj)  { obj.xxx();   }
void func3(Cls& obj)  { obj.xxx();   }
.....
Cls obj;
func1(&obj);
func2(obj);
func3(obj);
```

被調用端 寫法相同, 很好

func1(&obj); —— 接口不同, 困擾

func2(obj); / func3(obj); > 調用端接口相同, 很好

reference 通常不用於聲明變量，而用於參數類型 (parameters type) 和返回類型 (return type) 的描述。

以下被視為 “same signature”（所以二者不能同時存在）：

```
double imag(const double& im) { ... }
double imag(const double  im) { ... }   //Ambiguity
```

signature

Q: const 是不是函數簽名的一部分？
A: Yes

參數傳遞：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;
}
```

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&);
};
```

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

提供了一种 普通函数或者类成员函数 访问另一个类中的私有或保护成员 的机制

友元函数在类声明的任何区域中声明，而定义则在类的外部
友元函数只是一个普通函数，并不是该类的类成员函数，它可以在任何地方调用，友元函数中通过对象名来访问该类的私有或保护成员

友元类的声明在该类的声明中，而实现在该类外。
类B是类A的友元，那么类B可以直接访问A的私有成员。

友元关系没有继承性
友元关系没有传递性(即不存在“友元的友元”

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);
}
```

this作用域是在类内部，当在类的非静态成员函数中访问类的非静态成员的时候，编译器会自动将对象本身的地址作为一个隐含参数传递给函数。

this是const指针
类的成员默认是private，而结构是public。this是类的指针，如果换成结构，那this就是结构的指针了。

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

  c2 += c1;
}
```

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

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);
}
```

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

  c2 += c1;
}
```

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

连串使用-考虑返回值

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* ();

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));
}
```

negate 反相 (取反)

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

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

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) << ')';
}
```

?!

```
void
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)

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

Operator Overloading, 操作符重載

http://en.cppreference.com/w/cpp/language/operators

Overloaded operators

When an operator appears in an expression, and at least one of its operands has a class type or an enumeration type, then overload resolution is used to determine the user-defined function to be called among all the functions whose signatures match the following:

Expression	As member function	As non-member function	Example
@a	(a).operator@ ( )	operator@ (a)	`!std::cin` calls `std::cin.operator!()`
a@b	(a).operator@ (b)	operator@ (a, b)	`std::cout << 42` calls `std::cout.operator<<(42)`
a=b	(a).operator= (b)	cannot be non-member	`std::string s; s = "abc";` calls `s.operator=("abc")`
a[b]	(a).operator[](b)	cannot be non-member	`std::map<int, int> m; m[1] = 2;` calls `m.operator[](1)`
a->	(a).operator-> ( )	cannot be non-member	`std::unique_ptr<S> ptr(new S); ptr->bar()` calls `ptr.operator->()`
a@	(a).operator@ (0)	operator@ (a, 0)	`std::vector<int>::iterator i = v.begin(); i++` calls `i.operator++(0)`
in this table, **@** is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b			

Operator Overloading, 操作符重載

http://en.cppreference.com/w/cpp/language/operators

Overloaded operators (but not the built-in operators) can be called using function notation:

```
std::string str = "Hello, ";
str.operator+=("world");                          // same as str += "world";
operator<<(operator<<(std::cout, str) , '\n'); // same as std::cout << str << '\n';
```

Restrictions

- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), and `?:` (ternary conditional) cannot be overloaded
- New operators such as `**`, `<>`, or `&|` cannot be created
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing.
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded.
- It is not possible to change the precedence, grouping, or number of operands of operators.

不能重载

Operator Overloading, 操作符重載

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
  typedef __list_iterator<T, Ref, Ptr> self;
  typedef bidirectional_iterator_tag iterator_category; //(1)
  typedef T    value_type;                    //(2)
  typedef Ptr pointer;                        //(3)
  typedef Ref reference;                      //(4)
  typedef __list_node<T>* link_type;
  typedef ptrdiff_t difference_type;          //(5)

  link_type node;

  reference operator*() const { return (*node).data; }
  pointer operator->()  const { return &(operator*()); }
  self& operator++()          { node = (link_type)((*node).next); return *this; }
  self operator++(int)        { self tmp = *this;  ++*this; return tmp; }
  ...
};
```

```
#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
```

```
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);
}
```

```
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));
}
```

```
#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;
```

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
```

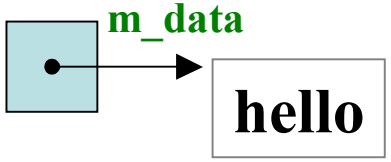
string-test.cpp

```
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

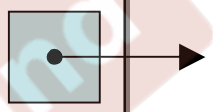
```
class String
{
public:
   String(const char* cstr = 0);
   String(const String& str);
   String& operator=(const String& str);
   ~String();
   char* get_c_str() const { return m_data; }
private:
   char* m_data;
};
```



ctor 和 dtor (構造函數 和 析構函數)

2-1

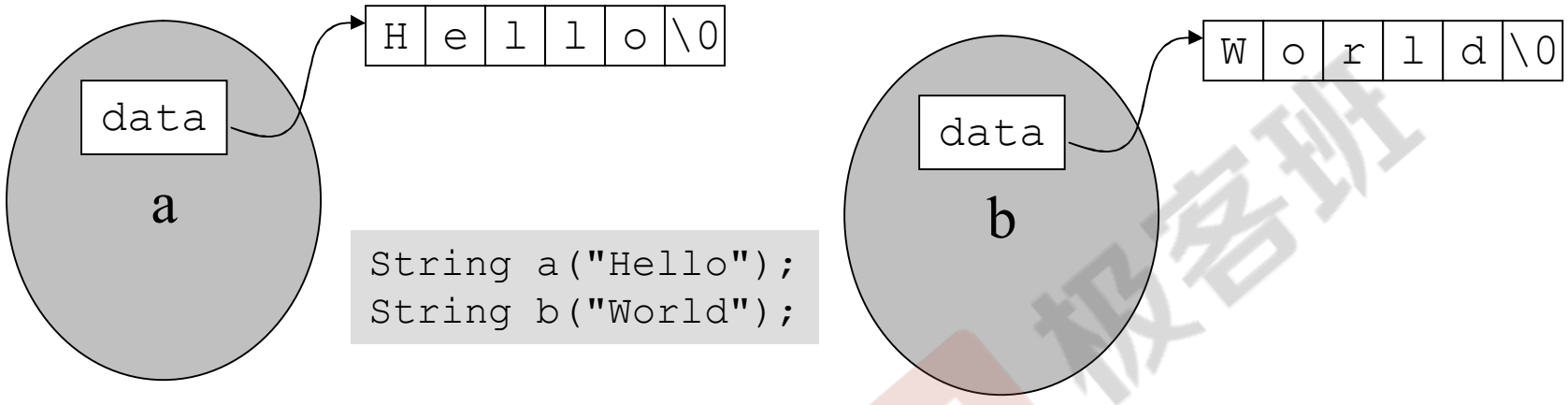
```
inline
String::String(const char* cstr = 0)
{
   if (cstr) {
      m_data = new char[strlen(cstr)+1];
      strcpy(m_data, cstr);
   }
   else {  // 未指定初值
      m_data = new char[1];
      *m_data = '\0';
   }
}

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



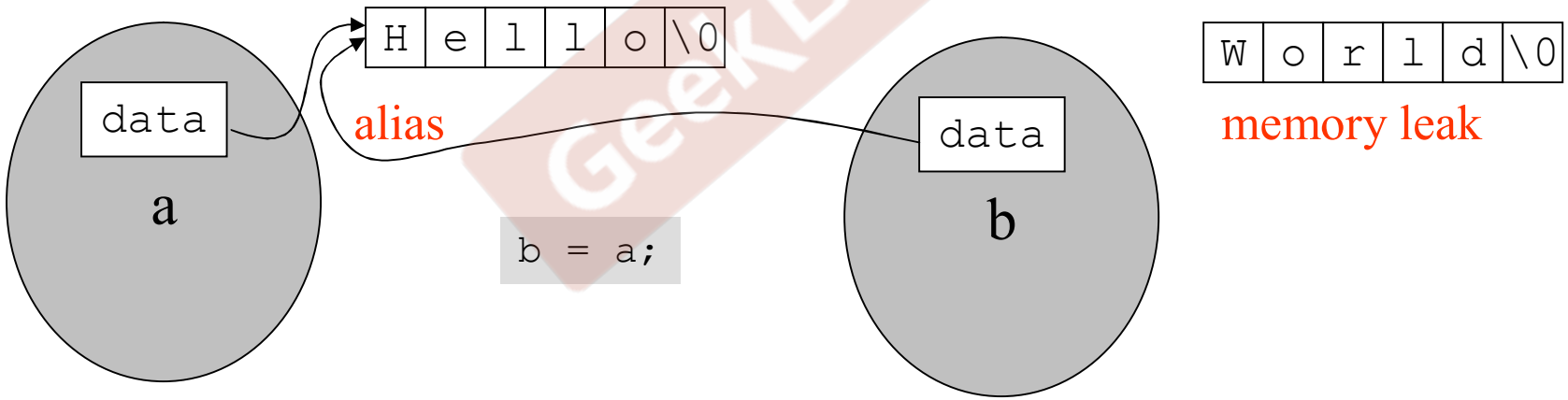
hello

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

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

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



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



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;
}
```

等同

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

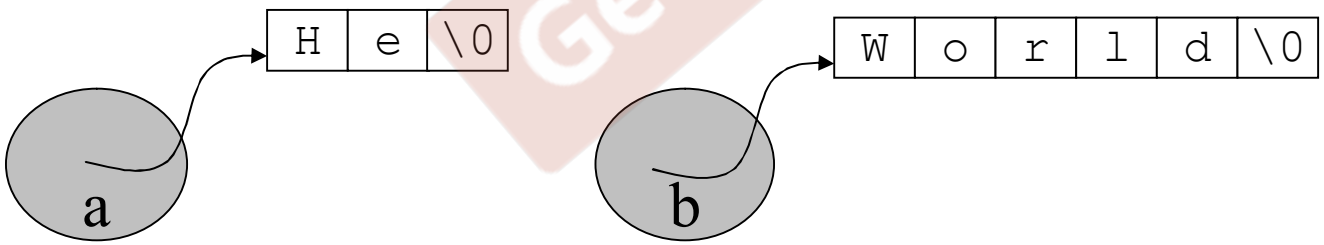
copy assignment operator (拷貝賦值函數)

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;
}
```



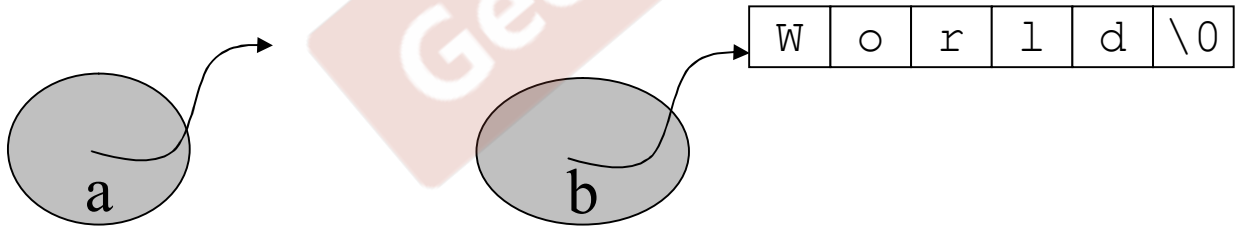
copy assignment operator (拷貝賦值函數)

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



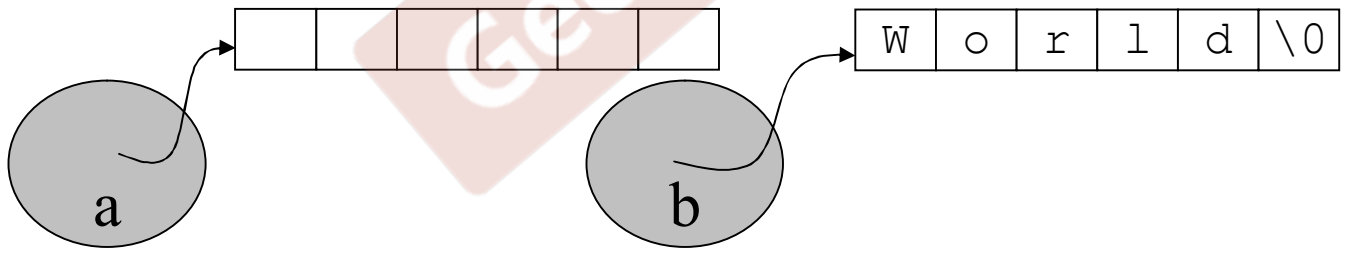
copy assignment operator (拷貝賦值函數)

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)

2



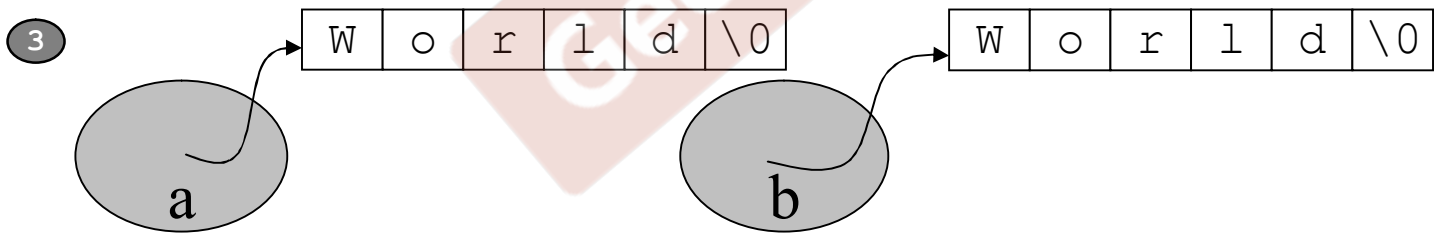
copy assignment operator (拷貝賦值函數)

2-3

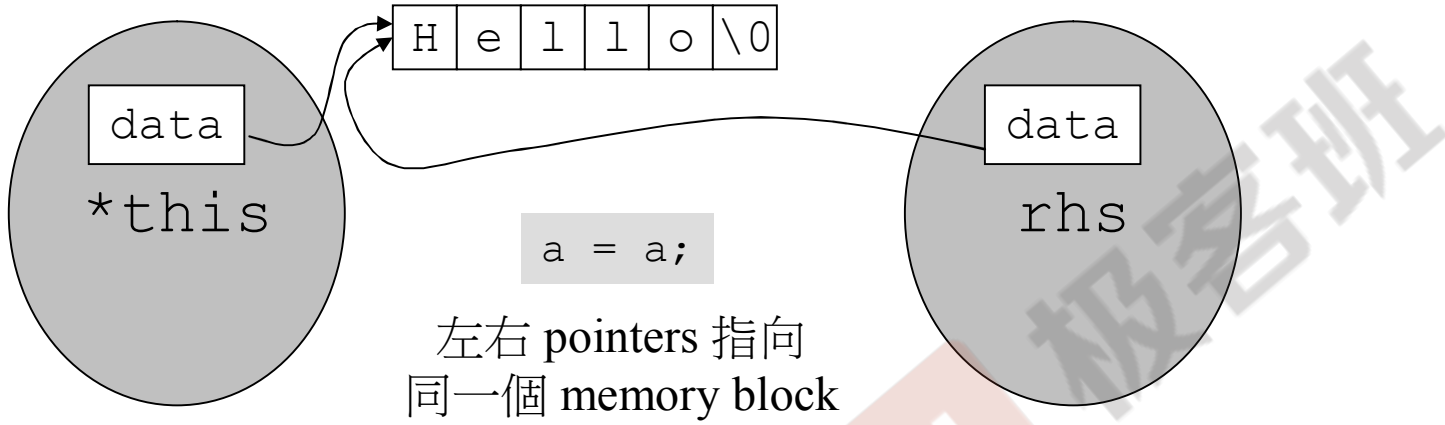
```
inline
String& String::operator=(const String& str)
{
    if (this == &str)
        return *this;

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)

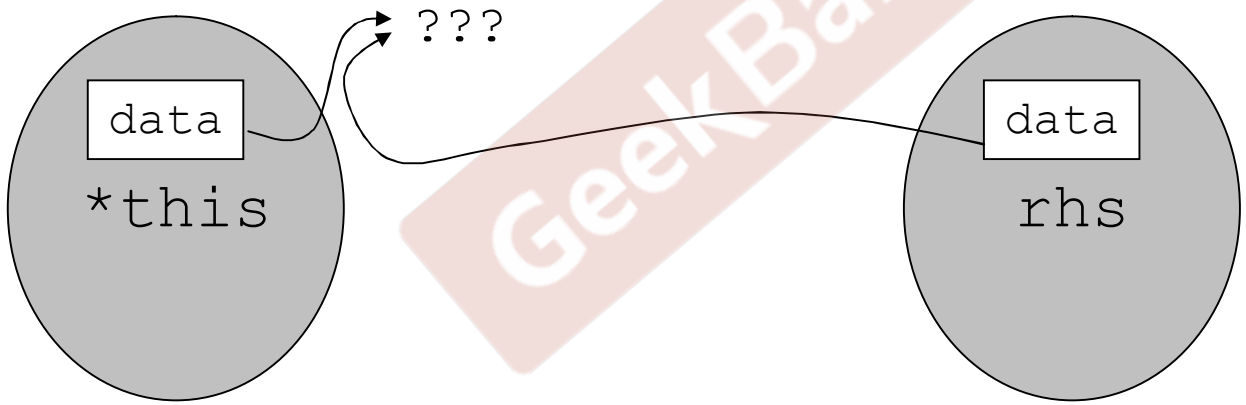
3



一定要在 operator= 中檢查是否 self assignment



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);
}
```

`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）

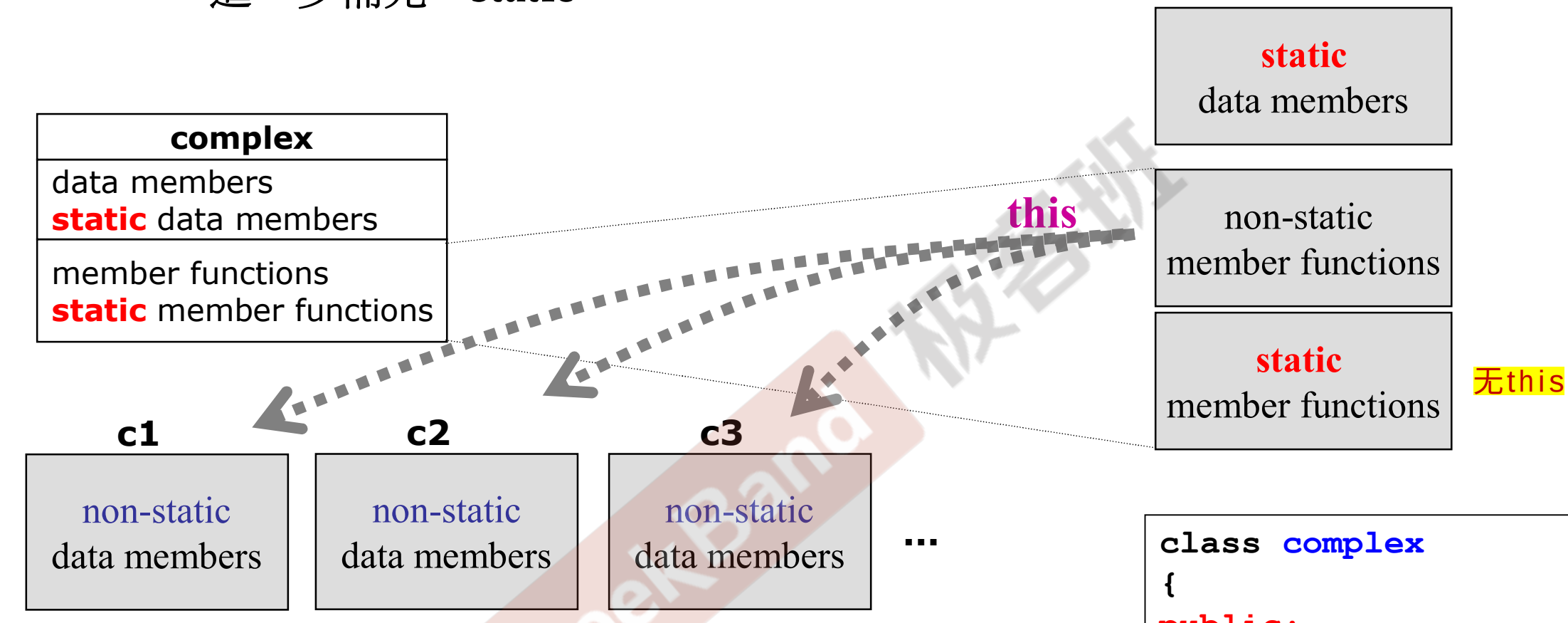
```
class String
{
public:
    String(const char* cstr = 0);
    String(const String& str);
    String& operator=(const String& str);
    ~String();
    char* get_c_str() const { return m_data; }
private:
    char* m_data;
};
```

```
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;
}
```

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

```
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;
}
```

進一步補充：static




```
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;
};
```

函数中的静态变量:
空间只分配一次 将在程序的生命周期内分配

类中的静态变量:
由对象共享 不能使用构造函数初始化
应由用户使用类外的类名和范围解析运算符显式初始化

类对象为静态:
像变量一样 范围直到程序的生命周期

类中的静态函数:
不依赖于类的对象
允许静态成员函数仅访问静态数据成员或其他静态成员函数，它们无法访问类的非静态数据成员或成员函数

限定访问范围:
使extern失效

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);
}
```

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;
}
```

只有调用它单例才会出现

```
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();
```

OOP, OOD

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

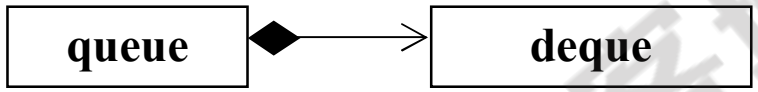
Composition (複合), 表示 has-a

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

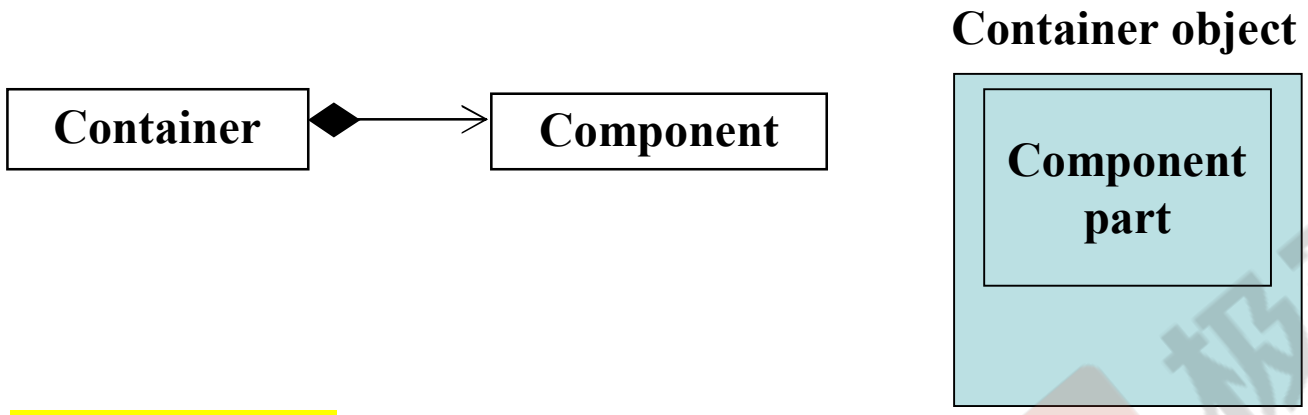
Sizeof : 16 * 2 + 4 + 4

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

Sizeof : 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()  { … };
```

编译器所加，非default则自己加

析構由外而內

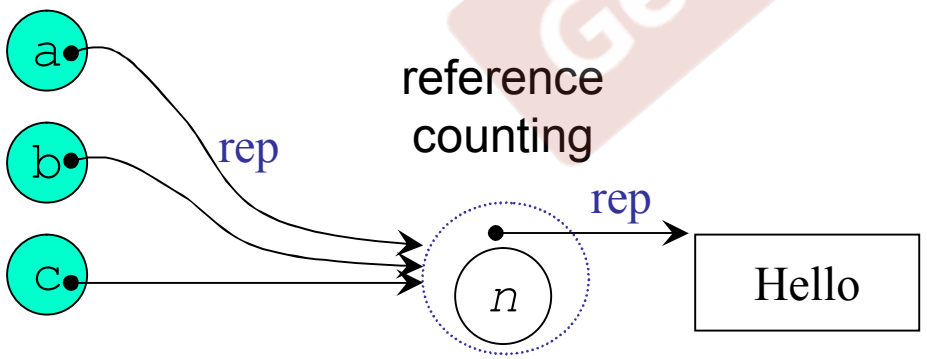
Container 的析構函數首先執行自己，然後才調用 Component 的析構函數。

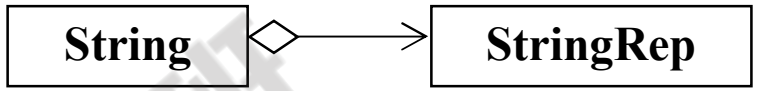
```
Container::~Container(…){ … ~Component()  };
```

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
};
```

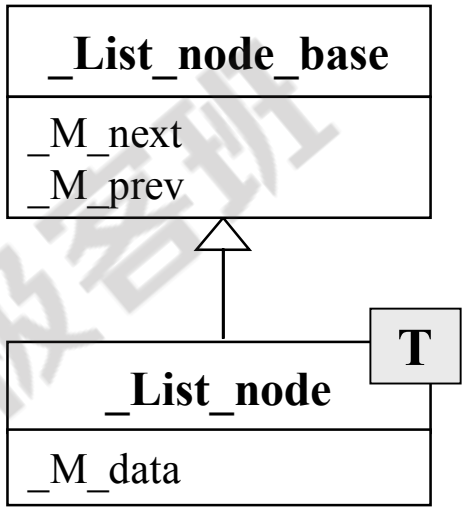




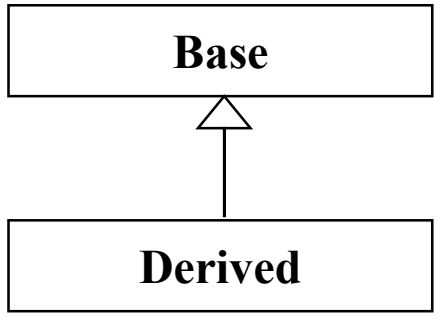

```
// file String.cpp
#include "String.hpp"
namespace {
class StringRep {
friend class String;
    StringRep(const char* s);
    ~StringRep();
    int count;
    char* rep;
};
}

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

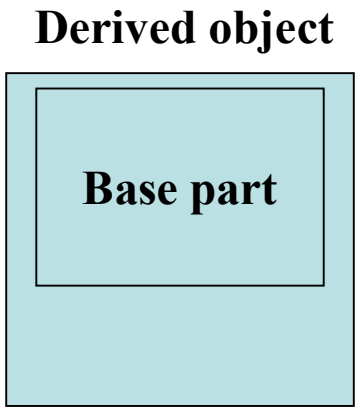
Inheritance (繼承), 表示 is-a

```
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;
};
```



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

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() };
```

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。

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 { ... };
```

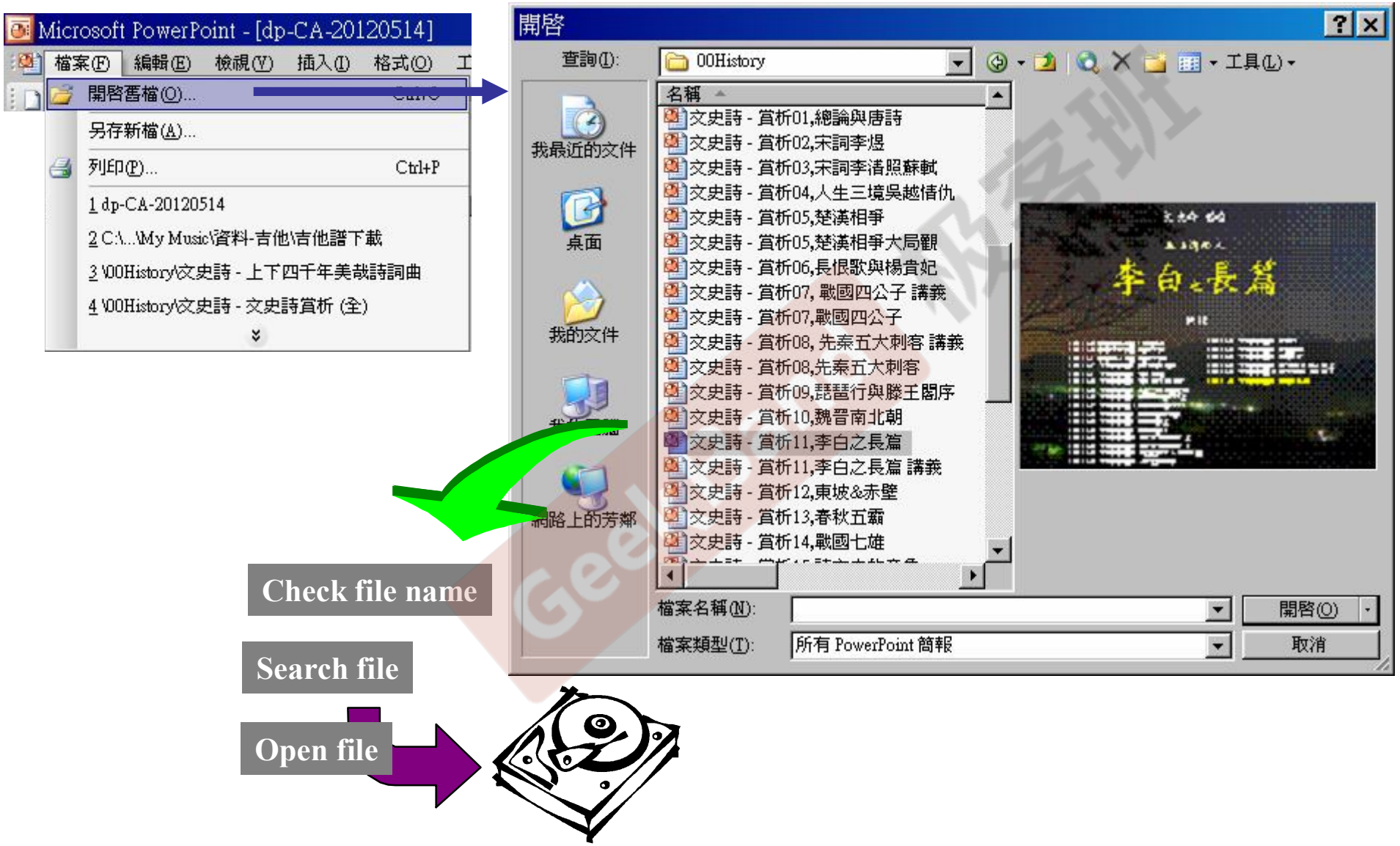
pure virtual

impure virtual

non-virtual

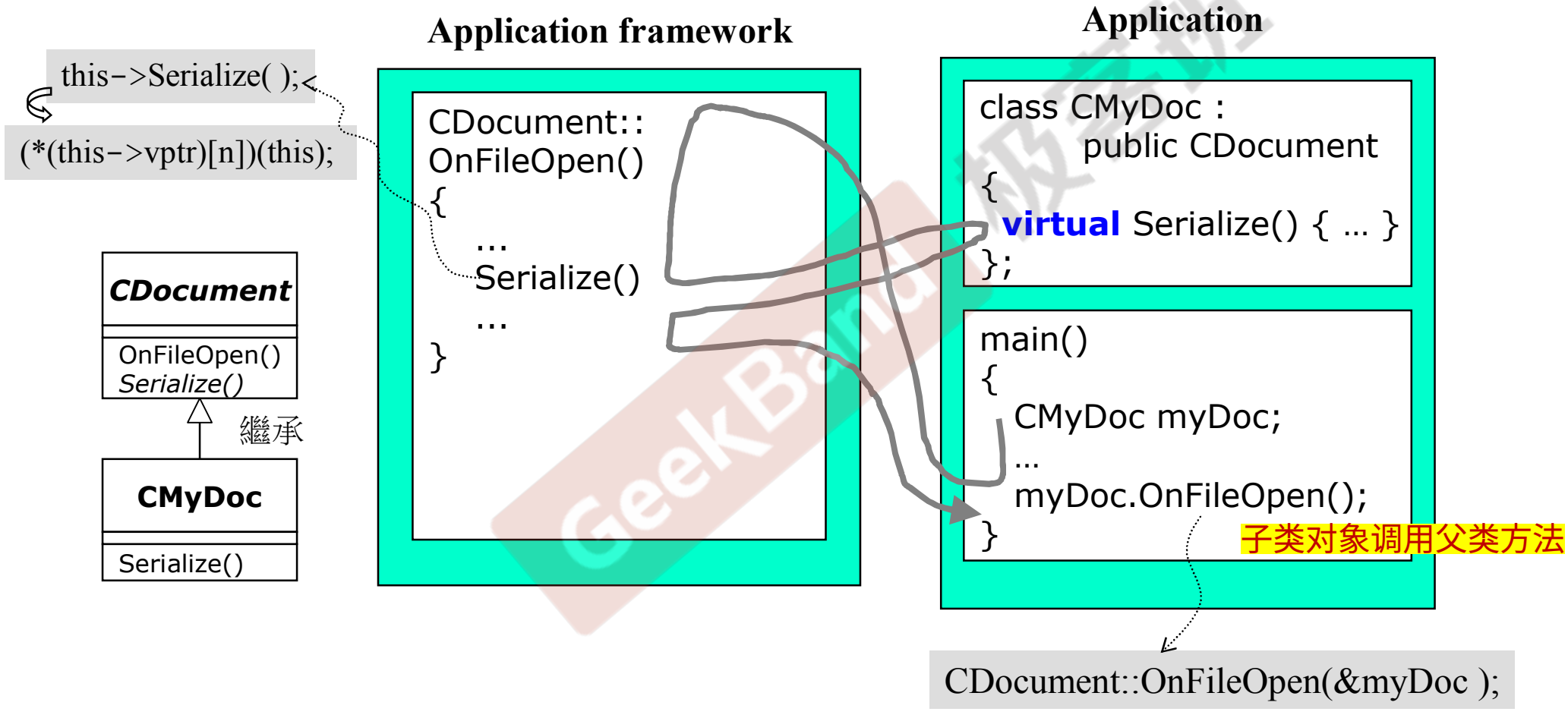
对函数 子类继承的是调用权

Inheritance (繼承) with virtual



Inheritance (繼承) with virtual

Template Method



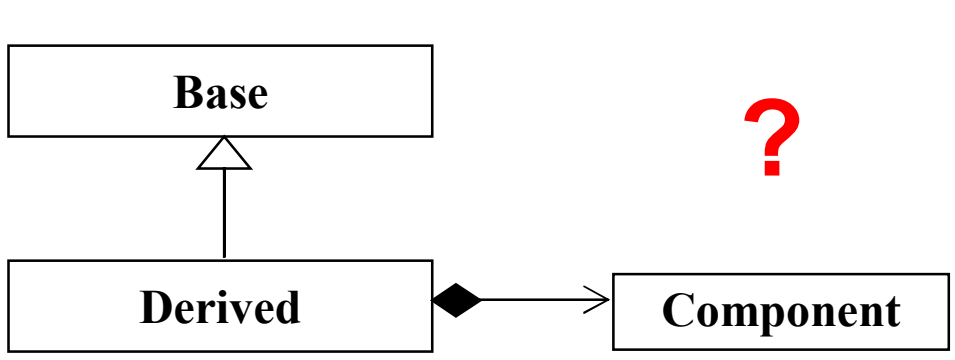
Inheritance (繼承), 表示 is-a

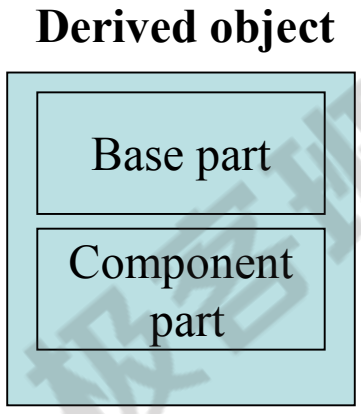
```
01 #include <iostream>
02 using namespace std;
03
04
05 class CDocument
06 {
07 public:
08     void OnFileOpen()
09     {
10       // 這是個算法，每個 cout 輸出代表一個實際動作
11       cout << "dialog..." << endl;
12       cout << "check file status..." << endl;
13       cout << "open file..." << endl;
14       Serialize();
15       cout << "close file..." << endl;
16       cout << "update all views..." << endl;
17     }
18
19     virtual void Serialize() { };
20 };
```

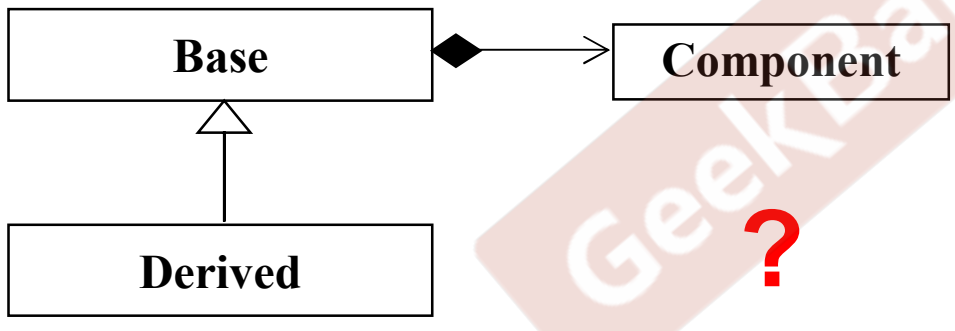
```
22 class CMyDoc : public CDocument
23 {
24 public:
25     virtual void Serialize()
26     {
27       // 只有應用程序本身才知道如何讀取自己的文件(格式)
28       cout << "CMyDoc::Serialize()" << endl;
29     }
30 };
```

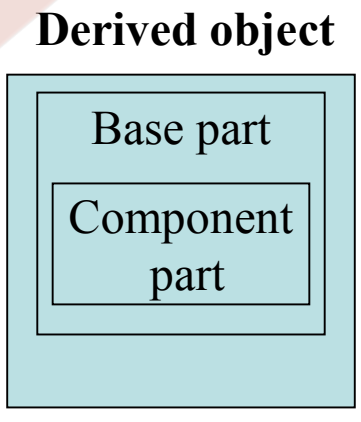
```
31 int main()
32 {
33   CMyDoc myDoc;  // 假設對應[File/Open]
34   myDoc.OnFileOpen();
35 }
```

Inheritance+Composition 關係下的構造和析構

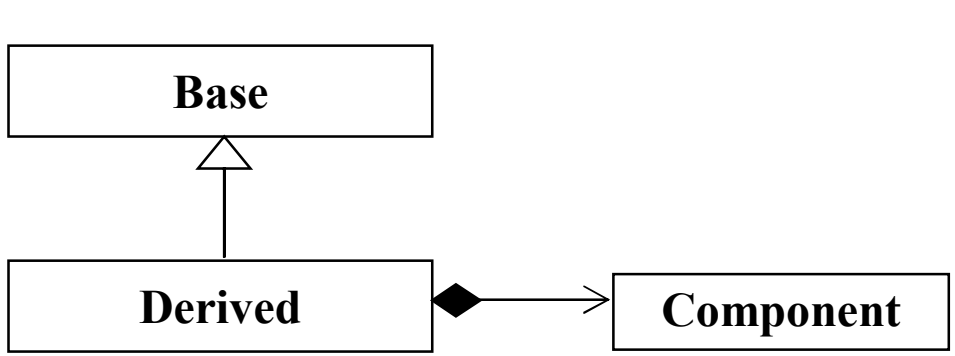


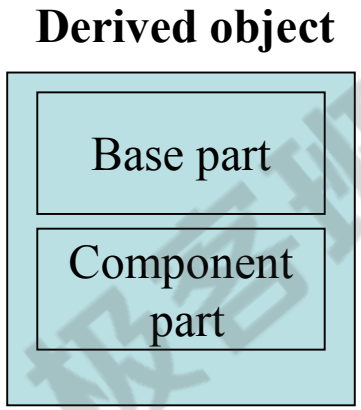






Inheritance+Composition 關係下的構造和析構





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。

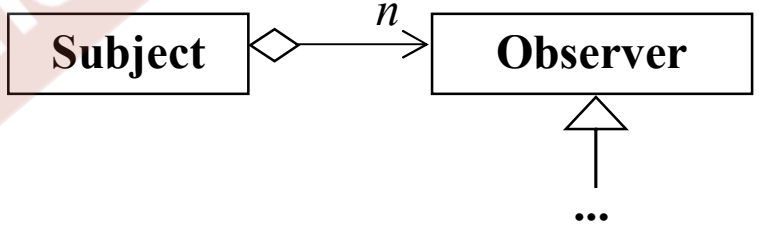
```
Derived::~Derived(…){ … ~Component(), ~Base() };
```

Delegation (委託) + Inheritance (繼承)

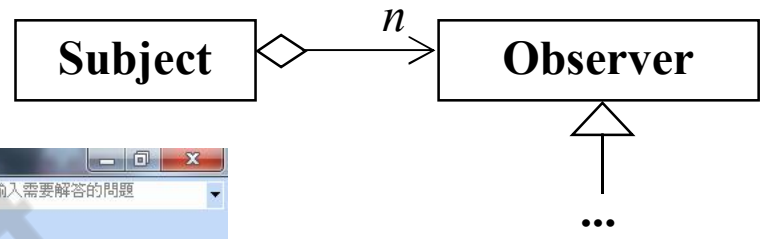
```
class Subject
{
    int m_value;
    vector<Observer*> m_views;
  public:
    void attach(Observer* obs)    注册
    {
        m_views.push_back(obs);
    }
    void set_val(int value)
    {
        m_value = value;
        notify();
    }
    void notify()    通知
    {
        for (int i = 0; i < m_views.size();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

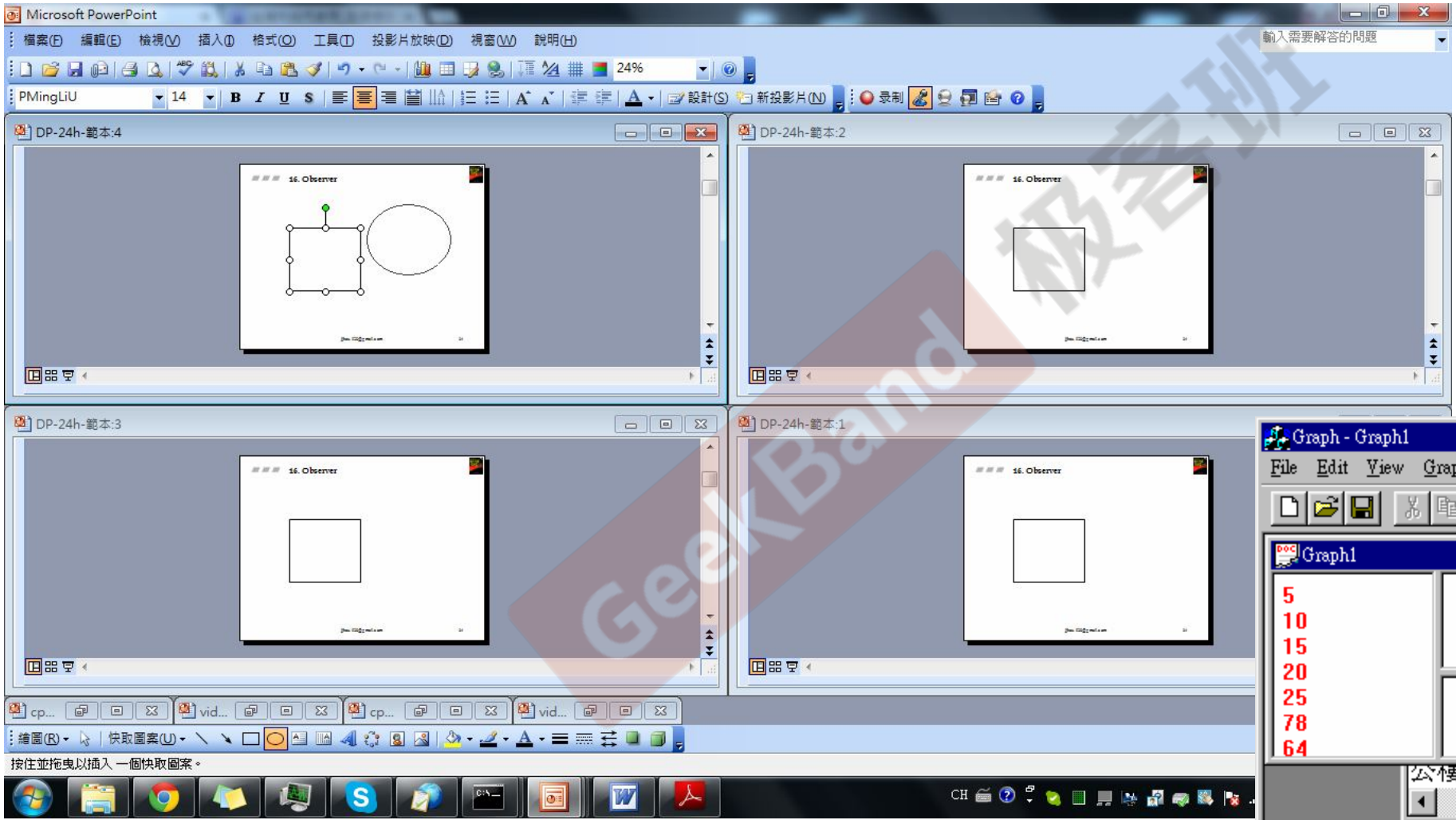
Observer

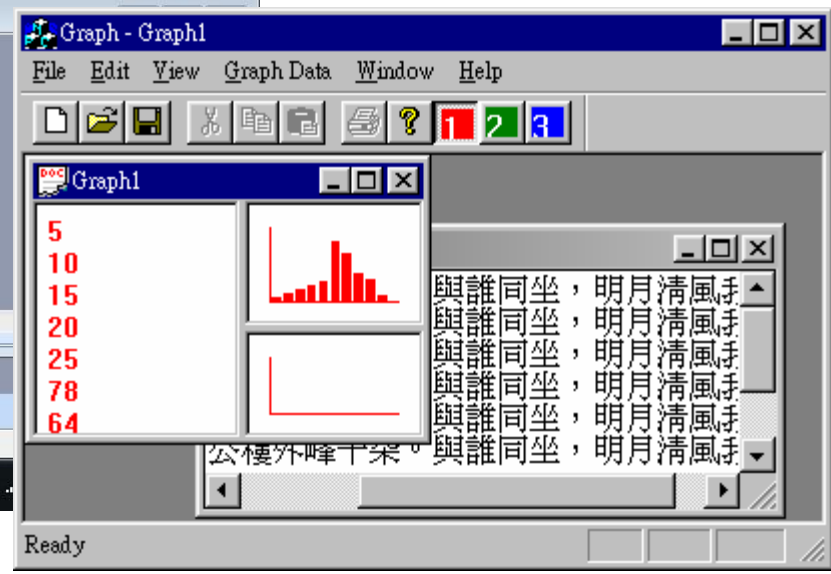
```
class Observer          观察数据
{
  public:
    virtual void update(Subject* sub, int value) = 0;
};
```



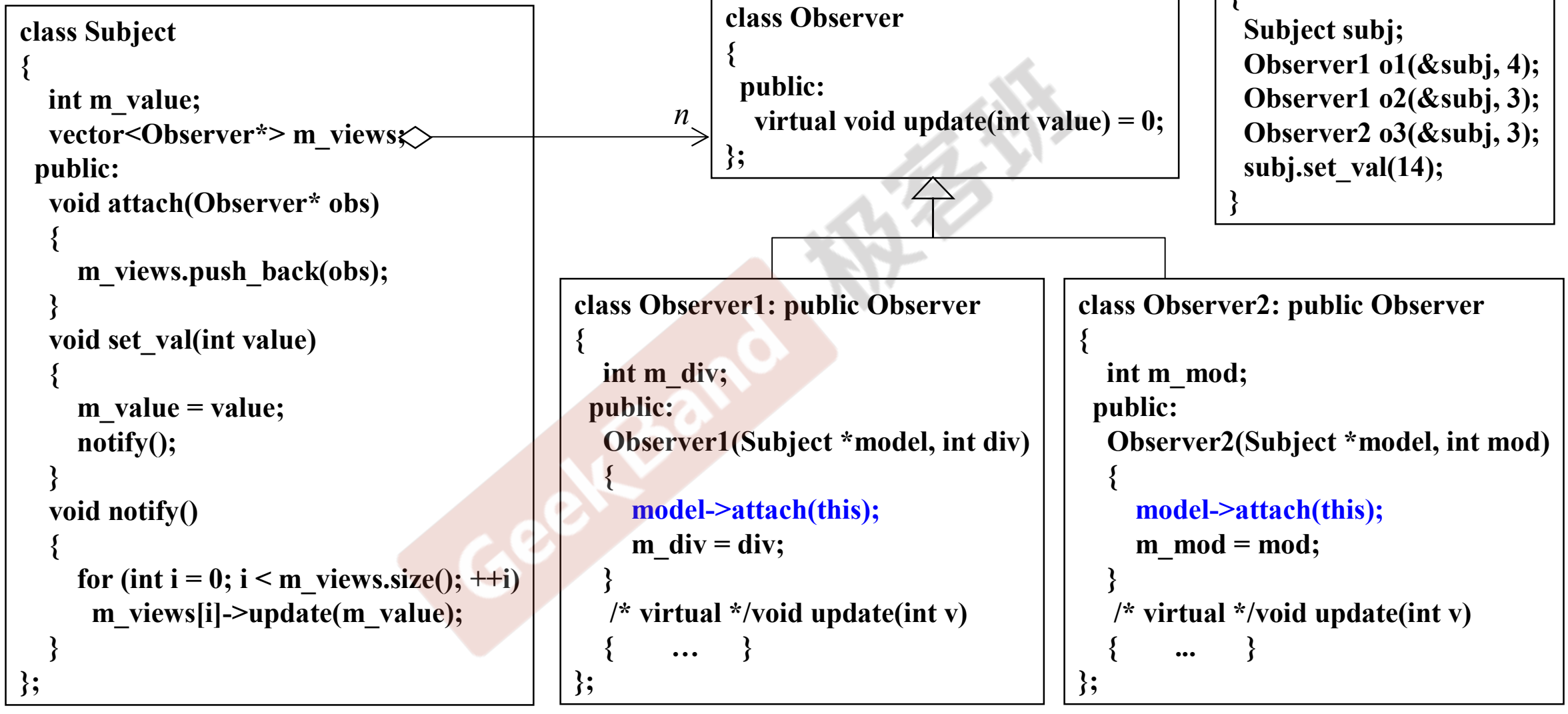
Delegation (委託) + Inheritance (繼承)







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);
   }
};
```

n

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

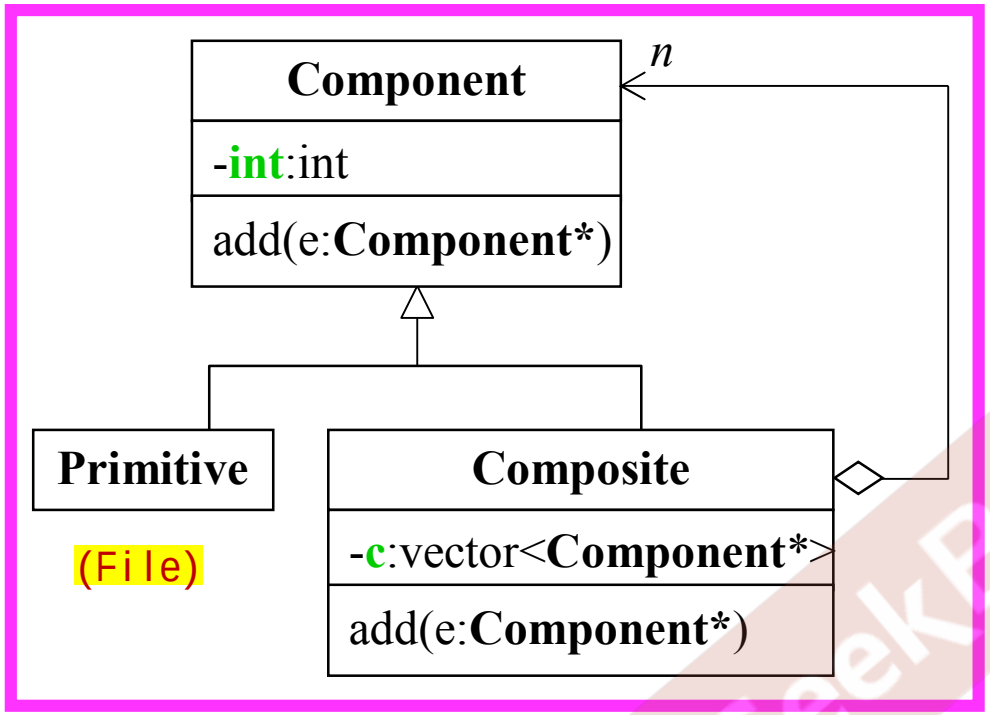
```
{
  Subject subj;
  Observer1 o1(&subj, 4);
  Observer1 o2(&subj, 3);
  Observer2 o3(&subj, 3);
  subj.set_val(14);
}
```

```
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)

Composite 文件系统




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

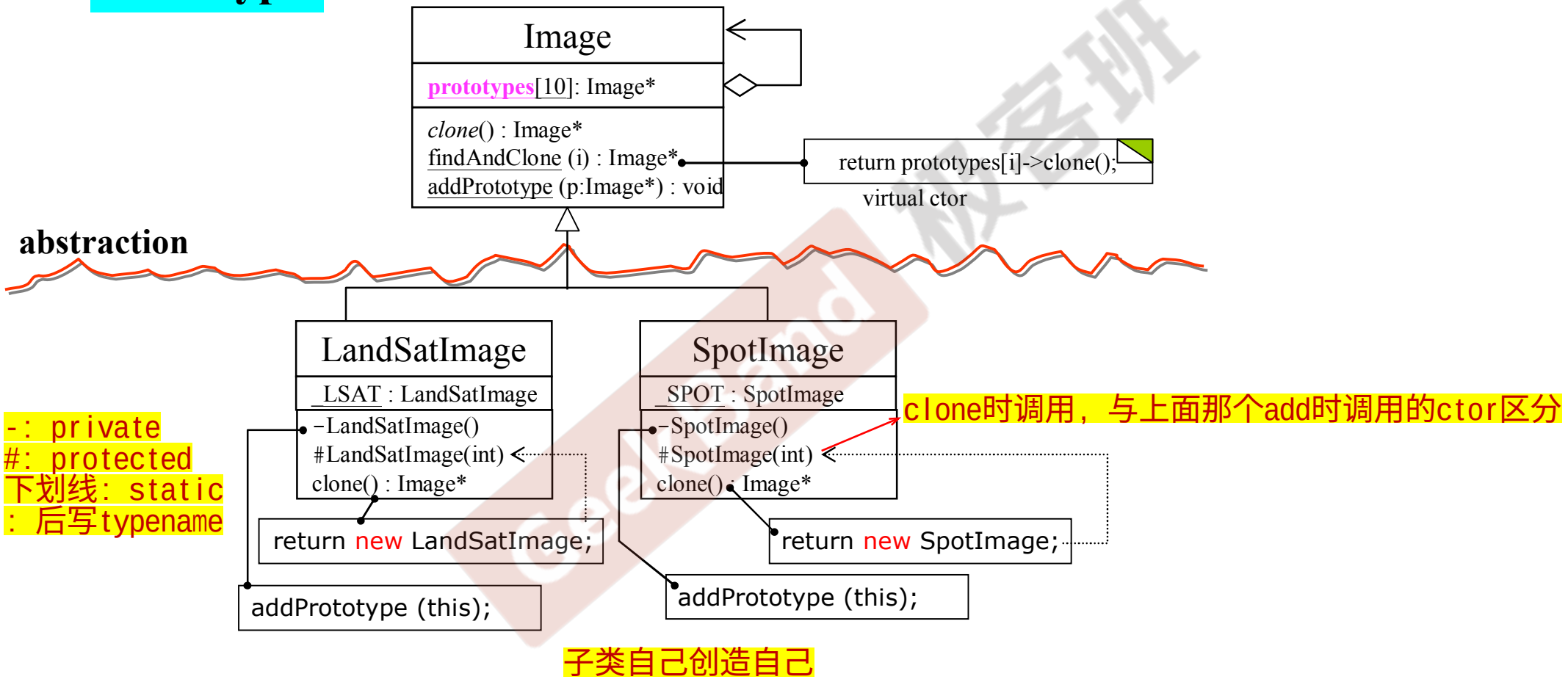
```
class Primitive: public Component
{
 public:
   Primitive(int val): Component(val) {}
};
```

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

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

Delegation (委託) + Inheritance (繼承)

Prototype

创建未来的class



-: private
#: protected
下划线: static
: 后写typename

clone时调用，与上面那个add时调用的ctor区分

子类自己创造自己

Prototype

```
01 #include <iostream.h>
02 enum imageType
03 {
04   LSAT, SPOT
05 };
06 class Image
07 {
08   public:
09     virtual void draw() = 0;
10     static Image *findAndClone(imageType);
11   protected:
12     virtual imageType returnType() = 0;
13     virtual Image *clone() = 0;
14     // As each subclass of Image is declared, it registers its prototype
15     static void addPrototype(Image *image)
16     {
17         _prototypes[_nextSlot++] = image;
18     }
19   private:
20     // addPrototype() saves each registered prototype here
21     static Image *_prototypes[10];
22     static int _nextSlot;
23 };
24 Image *Image::_prototypes[];
25 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();
}
```

add后制造副本

Prototype

```
01  class LandSatImage: public Image
02  {
03   public:
04    imageType returnType()    {
05       return LSAT;
06    }
07    void draw()    {
08       cout << "LandSatImage::draw " << _id << endl;
09    }
10    // When clone() is called, call the one-argument ctor with a dummy arg
11    Image *clone()    {
12       return new LandSatImage(1);
13    }
14   protected:
15    // This is only called from clone()
16    LandSatImage(int dummy)    {
17       _id = _count++;
18    }
19   private:
20    // Mechanism for initializing an Image subclass - this causes the
21    // default ctor to be called, which registers the subclass's prototype
22    static LandSatImage _landSatImage;
23    // This is only called when the private static data member is inited
24    LandSatImage()    {
25       addPrototype(this);
26    }
27    // Nominal "state" per instance mechanism
28    int _id;
29    static int _count;
30  };
31  // Register the subclass's prototype
32  LandSatImage LandSatImage::_landSatImage;
33  // Initialize the "state" per instance mechanism
34  int LandSatImage::_count = 1;
```

enum imageType { LSAT, SPOT };

```
01  class SpotImage: public Image
02  {
03   public:
04    imageType returnType()    {
05       return SPOT;
06    }
07    void draw()    {
08       cout << "SpotImage::draw " << _id << endl;
09    }
10    Image *clone()    {
11       return new SpotImage(1);
12    }
13   protected:
14    SpotImage(int dummy)  {
15       _id = _count++;
16    }
17   private:
18    SpotImage()    {
19       addPrototype(this);
20    }
21    static SpotImage _spotImage;
22    int _id;
23    static int _count;
24  };
25  SpotImage SpotImage::_spotImage;
26  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
};
```

```
01  int main()
02  {
03    Image *images[NUM_IMAGES];
04    // Given an image type, find the right prototype, and return a clone
05    for (int i = 0; i < NUM_IMAGES; i++)
06      images[i] = Image::findAndClone(input[i]);
07    // Demonstrate that correct image objects have been cloned
08    for (i = 0; i < NUM_IMAGES; i++)
09      images[i]->draw();
10    // Free the dynamic memory
11    for (i = 0; i < NUM_IMAGES; i++)
12      delete images[i];
13  }
```

C++程序設計 (II)
兼談對象模型

C++ Programming (part II), and Object Model

侯捷

勿在浮沙築高台

我們的目標

- 在先前基礎課程所培養的正規、大器的編程素養上，繼續探討更多技術。

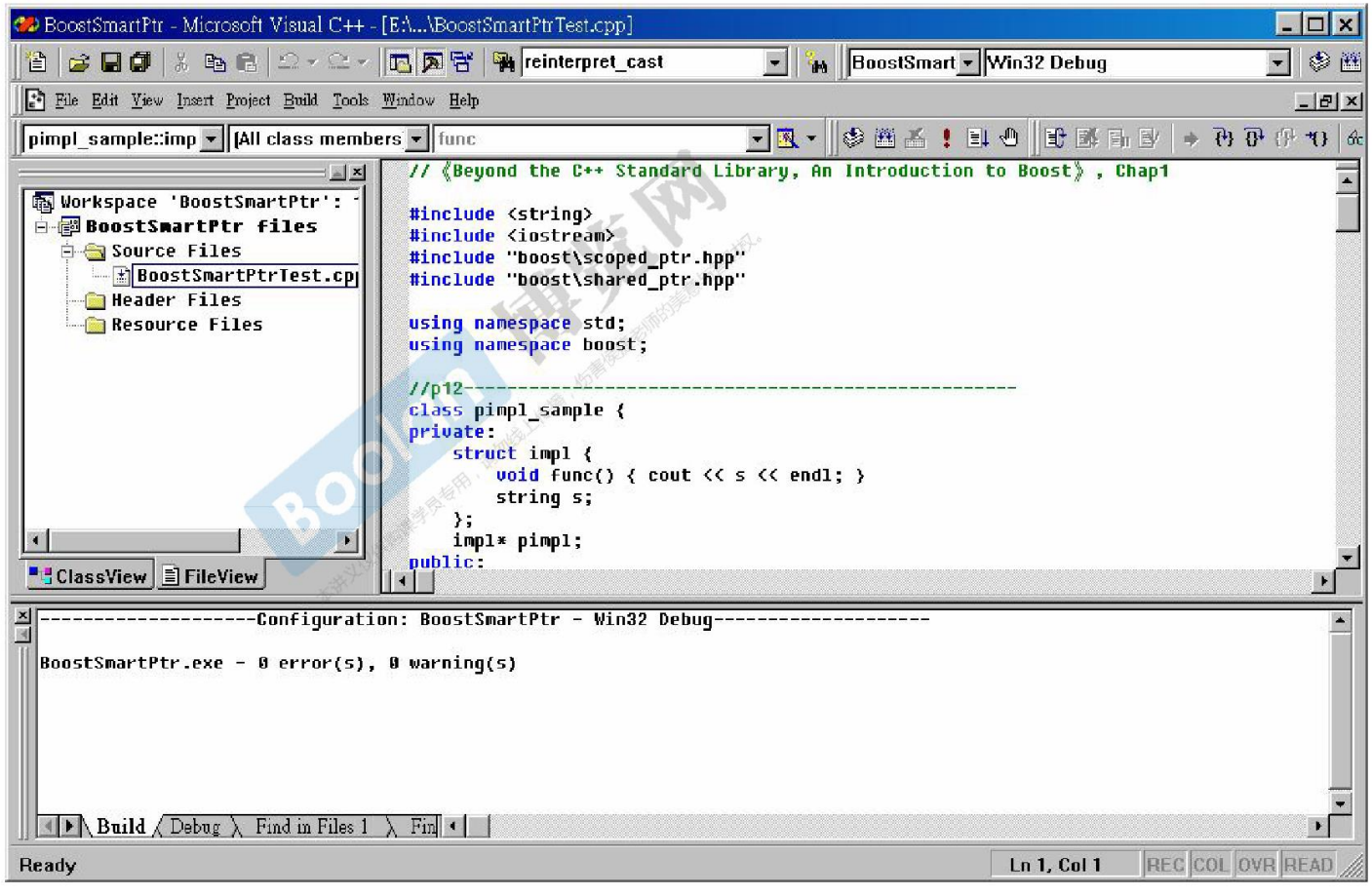
- 泛型編程 (Generic Programming) 和面向對象編程 (Object-Oriented Programming) 雖然分屬不同思維，但它們正是 C++ 的技術主線，所以本課程也討論 template (模板)。

- 深入探索面向對象之繼承關係 (inheritance) 所形成的對象模型 (Object Model)，包括隱藏於底層的 this 指針, vptr (虛指針), vtbl (虛表), virtual mechanism (虛機制), 以及虛函數 (virtual functions) 造成的 polymorphism (多態) 效果。

C++ 編譯器

- 編譯 (compile)
- 連結 (link)

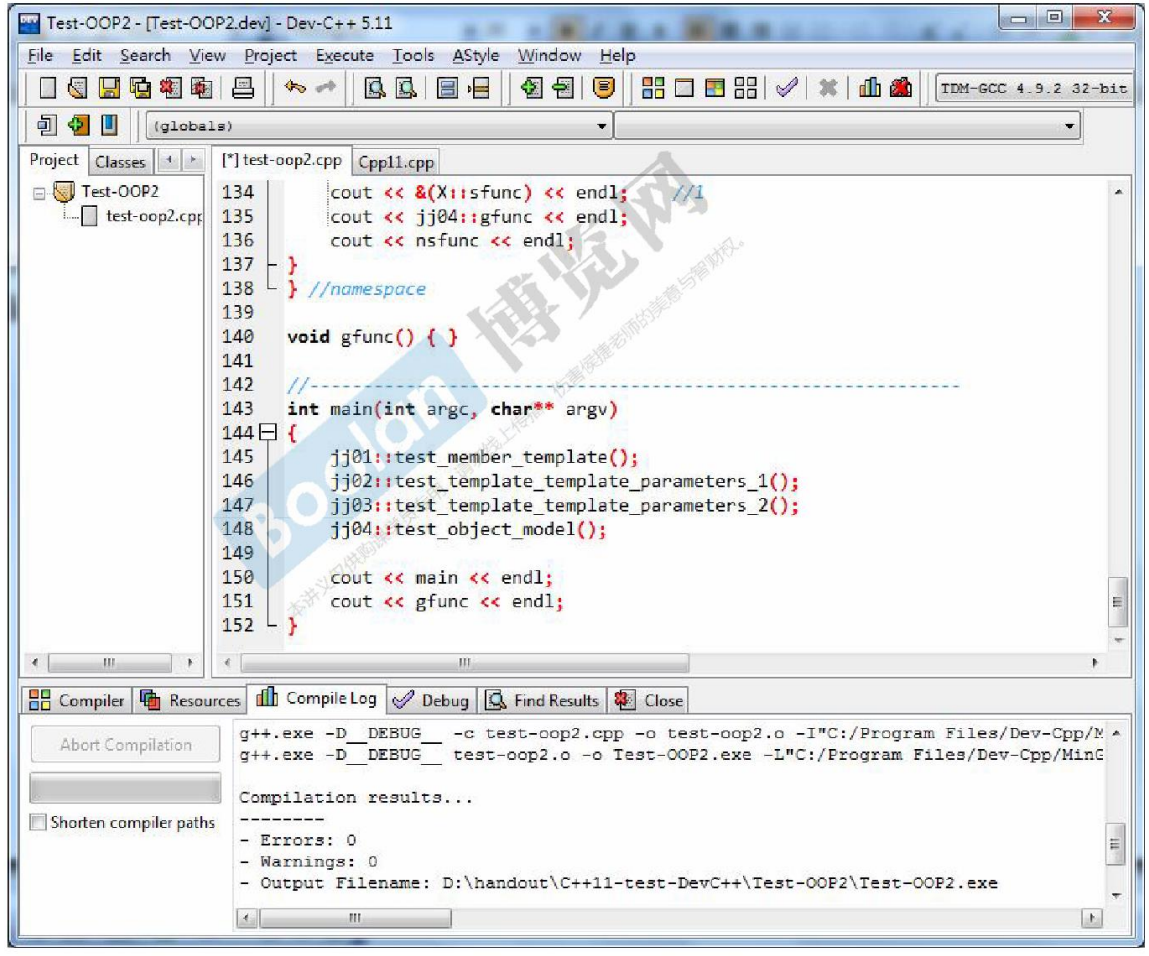
(本圖為 Visual C++ 6.0 畫面)



C++ 編譯器

- 編譯 (compile)
- 連結 (link)

(本圖為 Dev-C++ 5.6 畫面)



conversion function, 轉換函數

```
class Fraction
{
public:
   Fraction(int num, int den=1)
    : m_numerator(num), m_denominator(den) { }
   operator double() const {
     return (double)(m_numerator / m_denominator);
   }
private:
   int m_numerator;     //分子
   int m_denominator;   //分母
};
```

```
Fraction f(3,5);
double d=4+f;  //調用operator double()將 f 轉為 0.6
```

non-explicit-one-argument ctor

```
class Fraction
{
public:
   Fraction(int num, int den=1)
    : m_numerator(num), m_denominator(den) { }

   Fraction operator+(const Fraction& f) {
     return Fraction(……);
   }
private:
   int m_numerator;
   int m_denominator;
};
```

```
Fraction f(3,5);
Fraction d2=f+4; //調用 non-explicit ctor 將 4 轉為 Fraction(4,1)
                 //然後調用operator+
```

```
explicit 修饰构造函数时，可以防止隐式转换和复制初始化
explicit 修饰转换函数时，可以防止隐式转换，但按语境转换除外

struct B
{
    explicit B(int) {}
    explicit operator bool() const { return true; }
};
 B b1(1);        // OK：直接初始化
//    B b2 = 1;         // 错误：被 explicit 修饰构造函数的对象不可以复制初始化
    B b3{ 1 };        // OK：直接列表初始化
//    B b4 = { 1 };         // 错误：被 explicit 修饰构造函数的对象不可以复制列表初始化
    B b5 = (B)1;        // OK：允许 static_cast 的显式转换
//    doB(1);            // 错误：被 explicit 修饰构造函数的对象不可以从 int 到 B 的隐式转换
    if (b1);        // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换
    bool b6(b1);        // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境
转换
//    bool b7 = b1;         // 错误：被 explicit 修饰转换函数 B::operator bool() 的对象不可以隐式转换
    bool b8 = static_cast<bool>(b1);  // OK：static_cast 进行直接初始化
```

conversion function vs. non-explicit-one-argument ctor

```
class Fraction
{
public:
   Fraction(int num, int den=1)
    : m_numerator(num), m_denominator(den) { }
   operator double() const {
     return (double)(m_numerator / m_denominator);
   Fraction operator+(const Fraction& f) {
     return Fraction(……);
   }
private:
   int m_numerator;
   int m_denominator;
};
```

```
Fraction f(3,5);
Fraction d2=f+4; //[Error] ambiguous
```

explicit-one-argument ctor

```
class Fraction
{
public:
   explict Fraction(int num, int den=1)
    : m_numerator(num), m_denominator(den) { }
   operator double() const {
     return (double)(m_numerator / m_denominator);
   Fraction operator+(const Fraction& f) {
     return Fraction(……);
   }
private:
   int m_numerator;
   int m_denominator;
};
```

```
Fraction f(3,5);
Fraction d2=f+4; //[Error] conversion from 'double' to 'Fraction' requested
```

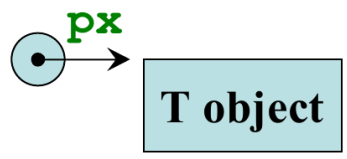
conversion function, 轉換函數

```
template<class Alloc>
class vector<bool, Alloc>
{
public:
  typedef __bit_reference reference;
protected:
  reference operator[](size_type n) {
     return *(begin() + difference_type(n));
  }
...
```

```
struct __bit_reference {
  unsigned int* p;
  unsigned int mask;
  ...
public:
  operator bool() const { return !(!(*p & mask)); }
...
```

pointer-like classes, 關於智能指針

```
template<class T>
class shared_ptr
{
public:
    T& operator*() const
    {  return *px;  }

    T* operator->() const
    {  return px;  }

    shared_ptr(T* p) : px(p) { }

private:
    T*      px;
    long*   pn;
……
};
```




```
struct Foo
{
  ……
  void method(void) { …… }
};
```

```
shared_ptr<Foo> sp(new Foo);

Foo f(*sp);

sp->method();
```

```
px->method();
```

作用后还会继续作用下去

pointer-like classes, 關於迭代器

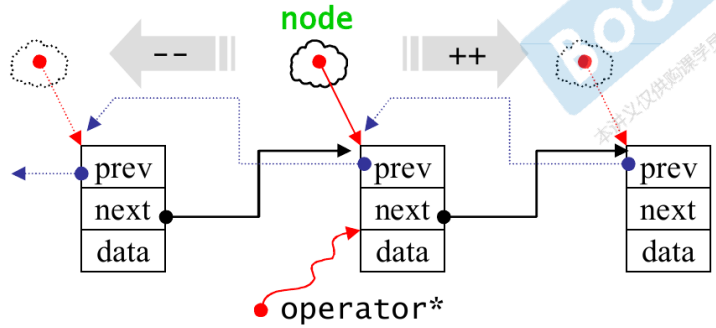
```
template <class T>
struct __list_node {
  void* prev;
  void* next;
  T data;
};
```

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
  typedef __list_iterator<T, Ref, Ptr> self;
  typedef Ptr pointer;
  typedef Ref reference;
  typedef __list_node<T>* link_type;
  link_type node;
  bool operator==(const self& x) const { return node == x.node; }
  bool operator!=(const self& x) const { return node != x.node; }
  reference operator*() const { return (*node).data; }
  pointer operator->() const { return &(operator*()); }
  self& operator++() { node = (link_type)((*node).next); return *this; }
  self operator++(int) { self tmp = *this;  ++*this; return tmp; }
  self& operator--() { node = (link_type)((*node).prev); return *this; }
  self operator--(int) { self tmp = *this; --*this;  return tmp; }
};
```

pointer-like classes, 關於迭代器

```
T&
reference operator*() const
{ return (*node).data; }
T*
pointer operator->() const
{ return &(operator*()); }
```

```
list<Foo>::iterator ite;
...
*ite; //獲得一個 Foo object
ite->method();
 //意思是調用 Foo::method()
 //相當於 (*ite).method();
 //相當於(&(*ite))->method();
```



function-like classes, 所謂仿函數

```
template <class T>
struct identity {
  const T&
  operator()(const T& x) const { return x; }
};

template <class Pair>
struct select1st {
  const typename Pair::first_type&
  operator()(const Pair& x) const
  { return x.first; }
};

template <class Pair>
struct select2nd {
  const typename Pair::second_type&
  operator()(const Pair& x) const
  { return x.second; }
};
```

```
template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b)
    : first(a), second(b) {}
……
};
```

標準庫中的仿函數的奇特模樣

```
template <class T>
struct identity {
  const T&
  operator()(const T& x) const { return x; }
};

template <class Pair>
struct select1st {
  const typename Pair::first_type&
  operator()(const Pair& x) const
  { return x.first; }
};

template <class Pair>
struct select2nd {
  const typename Pair::second_type&
  operator()(const Pair& x) const
  { return x.second; }
};
```

標準庫中的仿函數的奇特模樣

```
template <class T>
struct plus {
    T operator()(const T& x, const T& y) const { return x + y; }
};
template <class T>
struct minus {
    T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct equal_to {
    bool operator()(const T& x, const T& y) const { return x == y; }
};
template <class T>
struct less {
    bool operator()(const T& x, const T& y) const { return x < y; }
};
```

標準庫中,仿函數所使用的奇特的 base classes

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

less<int>::result_type ➔ bool

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;
}
```

namespace 經驗談

全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间

类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的

命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的

```
using namespace std;
//-----------------------------------------------------
#include <iostream>
#include <memory> //shared_ptr
namespace jj01
{
void test_member_template()
{   ……  }
} //namespace
//-----------------------------------------------------
#include <iostream>
#include <list>
namespace jj02
{
template<typename T>
using Lst = list<T, allocator<T>>;
void test_template_template_param()
{   ……  }
} //namespace
//-----------------------------------------------------
```

```
int main(int argc, char** argv)
{
   jj01::test_member_template();
   jj02::test_template_template_param();
}
```

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);
  ...
}
```

function template, 函數模板

不必指明type

編譯器會對 function template 進行
實參推導（argument deduction）

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

```
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;
};
```

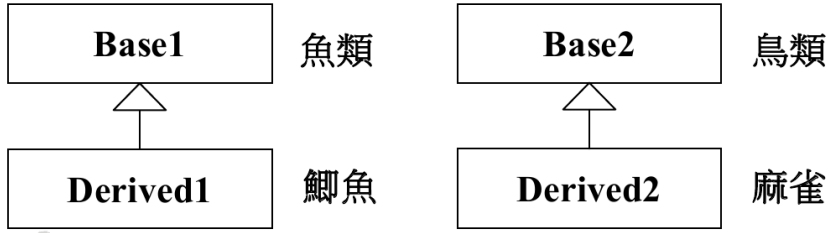
實參推導的結果，T 為 stone，於是調用 stone::operator<

member template, 成員模板

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;

  pair()
    : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b)
    : first(a), second(b) {}

  template <class U1, class U2>
  pair(const pair<U1, U2>& p)
    : first(p.first), second(p.second) {}
};
```

member template, 成員模板

```
class Base1 { };
class Derived1: public Base1 { };

class Base2 { };
class Derived2: public Base2 { };
```




```
pair<Derived1, Derived2> p;

pair<Base1, Base2> p2(p);
```

T1 T2

```
pair<Base1, Base2> p2(pair<Derived1, Derived2>());
```

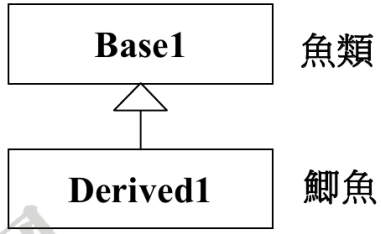
T1 T2 U1 U2

把一個由鯽魚和麻雀構成的 pair，放進 (拷貝到) 一個由魚類和鳥類構成的 pair 中，可以嗎？
反之，可以嗎？

```
template <class T1, class T2>
struct pair {
…
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) :
       first(a), second(b) {}

  template <class U1, class U2>
  pair(const pair<U1, U2>& p) :
      first(p.first), second(p.second) {}
};
```

member template, 成員模板

```
template<typename _Tp>
class shared_ptr : public __shared_ptr<_Tp>
{
…
      template<typename _Tp1>
      explicit shared_ptr(_Tp1* __p)
        : __shared_ptr<_Tp>(__p) { }
…
};
```




```
Base1* ptr = new Derived1;       //up-cast

shared_ptr<Base1> sptr(new Derived1);  //模擬 up-cast
```

specialization, 模板特化

```
template <class Key>
struct hash { };
```

```
template<>
struct hash<char> {
  size_t operator()(char x) const { return x; }
};

template<>
struct hash<int> {
  size_t operator()(int x) const { return x; }
};

template<>
struct hash<long> {
  size_t operator()(long x) const { return x; }
};
```

```
cout << hash<long>()(1000);
```

partial specialization, 模板偏特化 —— 個數的偏

```
template<typename T, typename Alloc=……>
class vector
{
...
};
```

綁定

```
template<typename Alloc=……>
class vector<bool, Alloc>
{
...
```

partial specialization, 模板偏特化 -- 範圍的偏

```
template <typename T>
class C
{
  ...
};
```

```
template <typename T>
class C<T*>
{
  ...
};
```

這樣寫
也可以

```
template <typename U>
class C<U*>
{
  ...
};
```



```
C<string> obj1;

C<string*> obj2;
```

template template parameter, 模板模板參數

```
template<typename T,
         template <typename T>
             class Container
        >
class XCls
{
private:
      Container<T> c;
public:
     ……
};
```

只有在<>中typename和class共通

```
template<typename T>
using Lst = list<T, allocator<T>>;
```

```
  x   XCls<string, list> mylst1;
      XCls<string, Lst>  mylst2;
```

template template parameter, 模板模板參數

```
template<typename T,
         template <typename T>
             class SmartPtr
          >
class XCls
{
private:
      SmartPtr<T> sp;
public:
      XCls() : sp(new T) { }
};
```

```
      XCls<string, shared_ptr> p1;
  x   XCls<double, unique_ptr> p2;
  x   XCls<int, weak_ptr> p3;
      XCls<long, auto_ptr> p4;
```

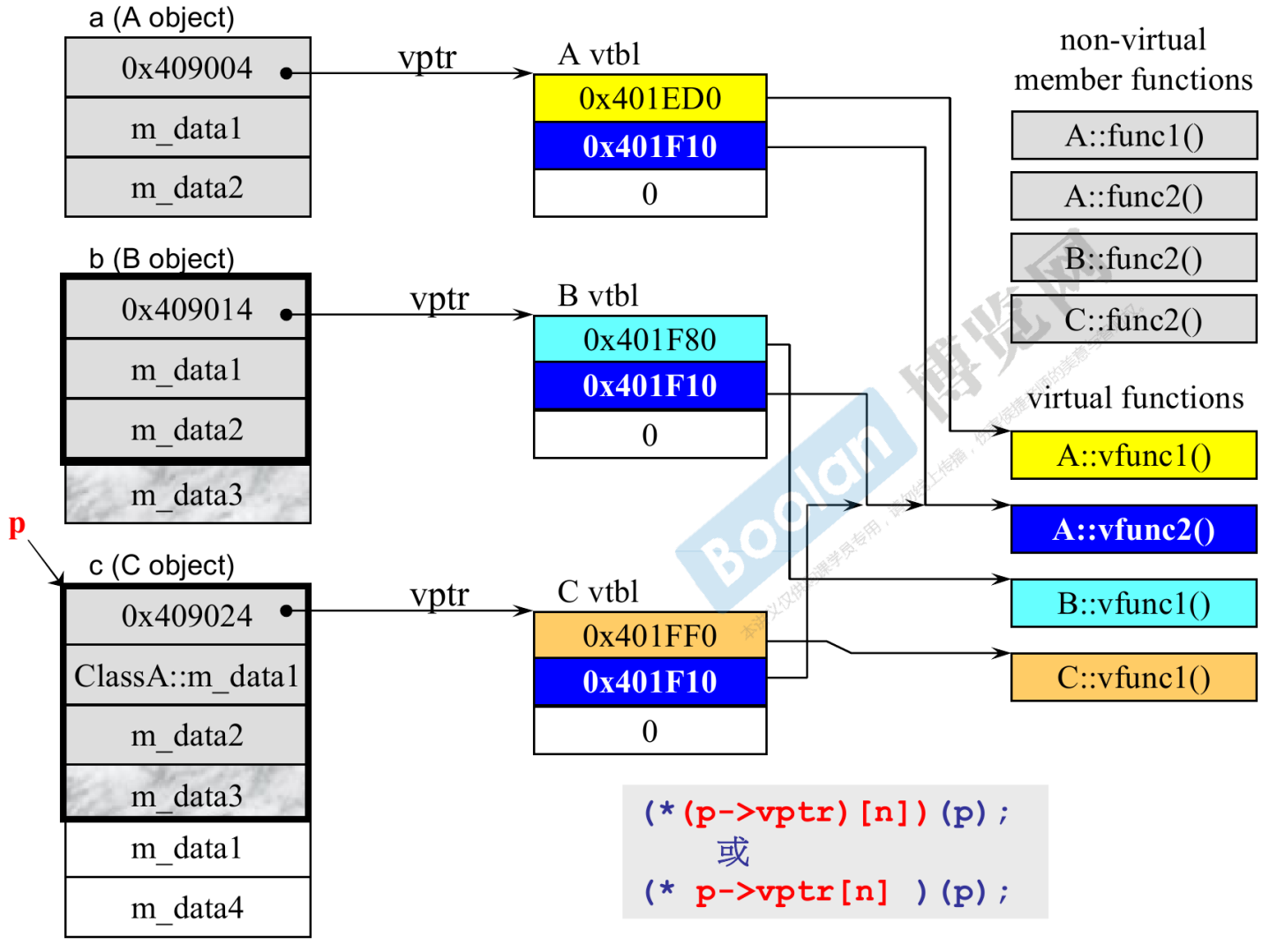
這不是 template template parameter

```
template <class T, class Sequence = deque<T>>
class stack {
  friend bool operator== <> (const stack&, const stack&);
  friend bool operator< <> (const stack&, const stack&);

protected:
  Sequence c;   //底層容器
……
};
```

- `stack<int> s1;`
- `stack<int,list<int>> s2;`

已经明确 不是模板

對象模型 (Object Model) ：關於 vptr 和 vtbl




```
(*(p->vptr)[n])(p);
    或
(* p->vptr[n] )(p);
```

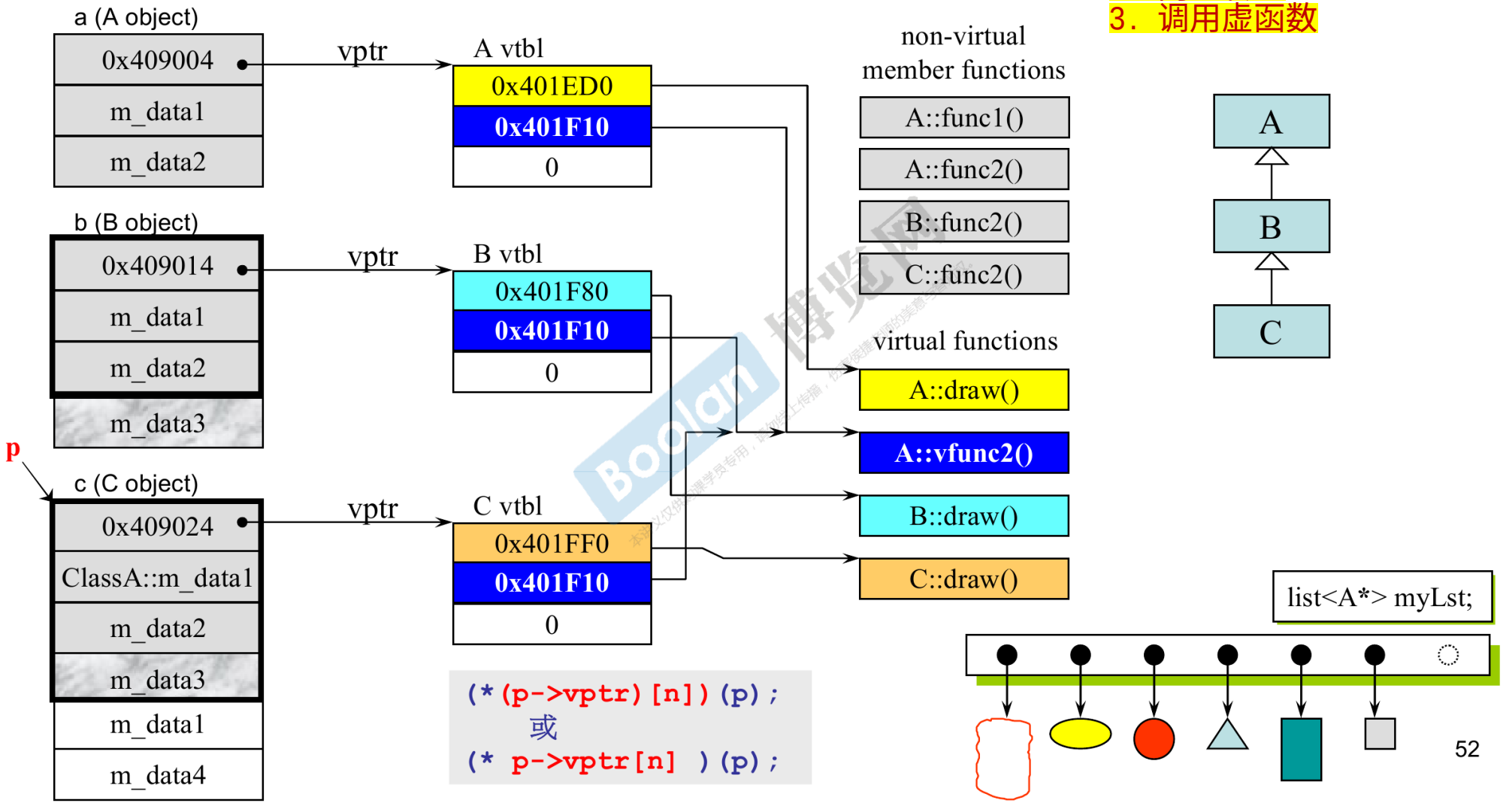
```
class A {
public:
  virtual void vfunc1();
  virtual void vfunc2();
           void func1();
           void func2();
private:
  int m_data1, m_data2;
};
```

```
class B : public A {
public:
  virtual void vfunc1();
           void func2();
private:
  int m_data3;
};
```

```
class C : public B {
public:
  virtual void vfunc1();
           void func2();
private:
  int m_data1, m_data4;
};
```

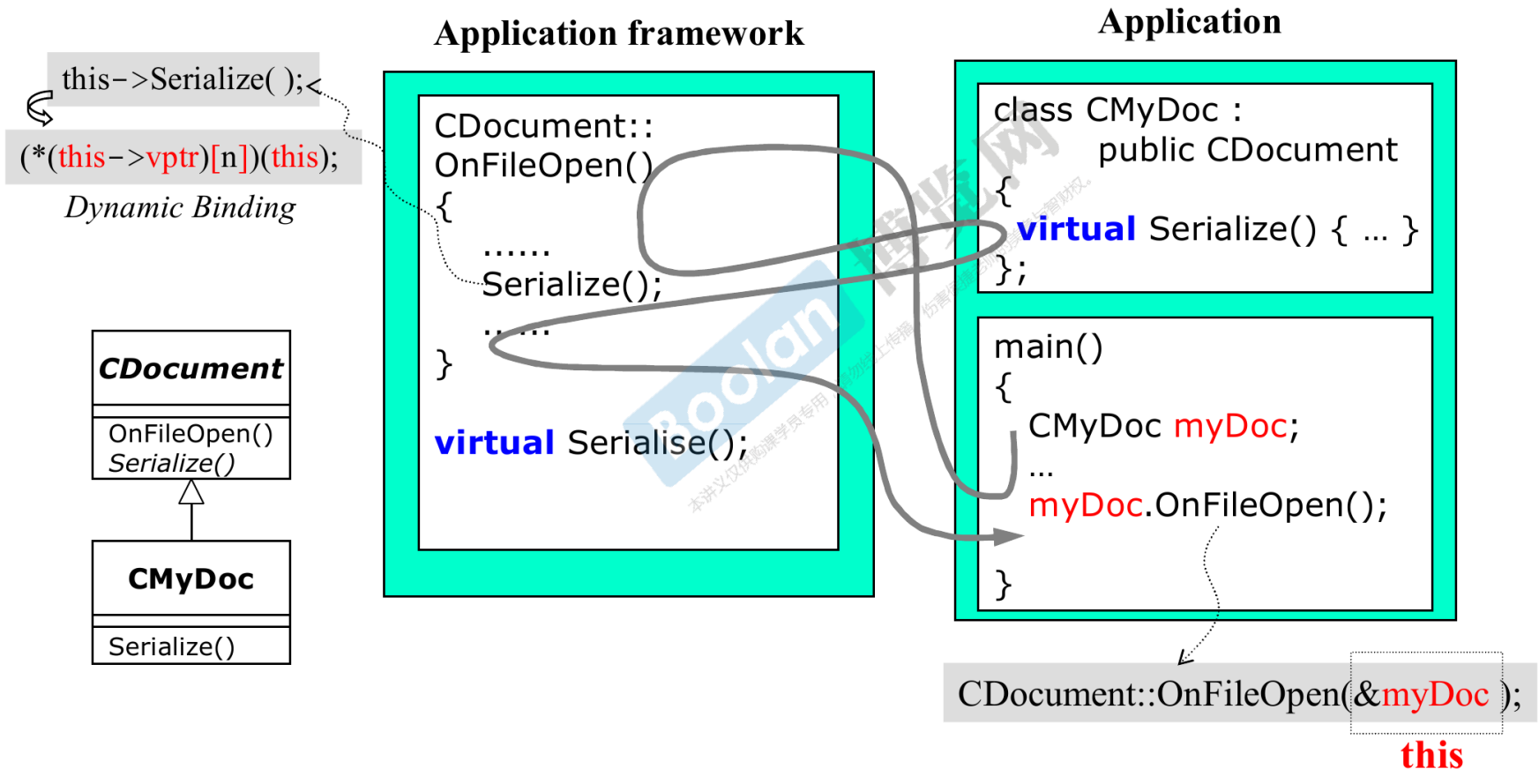
對象模型 (Object Model) ：關於 vptr 和 vtbl

动态绑定(虚机制)：
1．指针调用
2．向上转型
3．调用虚函数

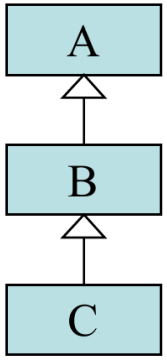


對象模型 (Object Model) ：關於 this

Template Method



對象模型 (Object Model) ：關於 Dynamic Binding




```
B b;
A a = (A)b;
a.vfunc1();

A* pa = new B;
pa->vfunc1();

pa = &b;          //up cast too!
pa->vfunc1();     //dynamic binding. B::vfunc1()
```

静态绑定

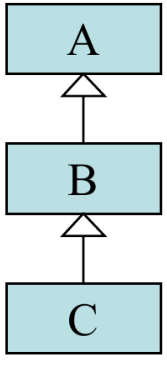
```
118:    B b;
00401CDE   lea          ecx,[b]
00401CE1   call         @ILT+535(B::B) (0040121c)
119:    A a = (A)b;
00401CE6   lea          eax,[b]
00401CE9   push         eax
00401CEA   lea          ecx,[ebp-114h]
00401CF0   call         @ILT+830(A::A) (00401343)
00401CF5   push         eax
00401CF6   lea          ecx,[a]
00401CFC   call         @ILT+830(A::A) (00401343)
120:    a.vfunc1();      //jj: static binding.  A::vfunc1()
00401D01   lea          ecx,[a]
00401D07   call         @ILT+420(A::vfunc1) (004011a9)
121:
```

call xxx

默认参数是静态绑定的，虚函数是动态绑定的。 默认参数的使用需要看指针或者引用本身的类型，而不是对象的类型。

對象模型 (Object Model) ：關於 Dynamic Binding

```
B b;
A a = (A)b;          //
a.vfunc1();          //

A* pa = new B;       //
pa->vfunc1();        //

pa = &b;             //
pa->vfunc1();        //
```




```
123:    pa->vfunc1();   //jj:dynamic binding.  B::vfunc1()
00401D68   mov          eax,dword ptr [pa]
00401D6E   mov          edx,dword ptr [eax]
00401D70   mov          esi,esp
00401D72   mov          ecx,dword ptr [pa]
00401D78   call         dword ptr [edx]
00401D7A   cmp          esi,esp
00401D7C   call         __chkesp (00423590)
124:
125:    pa = &b;         //jj:up cast too!
00401D81   lea          eax,[b]
00401D84   mov          dword ptr [pa],eax
126:    pa->vfunc1();    //jj:dynamic binding.  B::vfunc1()
00401D8A   mov          ecx,dword ptr [pa]
00401D90   mov          edx,dword ptr [ecx]
00401D92   mov          esi,esp
00401D94   mov          ecx,dword ptr [pa]
00401D9A   call         dword ptr [edx]
00401D9C   cmp          esi,esp
00401D9E   call         __chkesp (00423590)
127:
```

```
(*(p->vptr)[n])(p);
      或
(* p->vptr[n] )(p);
```

每个使用虚函数的类（或者从使用虚函数的类派生）都有自己的虚拟表。该表只是编译器在编译时设置的静态数组。虚拟表包含可由类的对象调用的每个虚函数的一个条目。此表中的每个条目只是一个函数指针，指向该类可访问的派生函数。

编译器还会添加一个隐藏指向基类的指针，我们称之为vptr。vptr在创建类实例时自动设置，以便指向该类的虚拟表。与this指针不同，this指针实际上是编译器用来解析自引用的函数参数，vptr是一个真正的指针。

通过virtual函数，指向子类的基类指针可以调用子类的函数。首先程序识别出fun1()是个虚函数，其次程序使用pt->vptr来获取Derived的虚拟表。第三，它查找Derived虚拟表中调用哪个版本的fun1()。这里就可以发现调用的是Derived::fun1()。因此pt->fun1()被解析为Derived::fun1()

静态函数不可以声明为虚函数，同时也不能被const 和 volatile关键字修饰
虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，静态成员函数没有this指针，无法访问vptr

构造函数不可以声明为虚函数。同时除了inline|explicit之外，构造函数不许使用其它关键字。
尽管虚函数表vtable是在编译阶段就已经建立的，但指向虚函数表的指针vptr是在运行阶段实例化对象时才产生的。 如果类含有虚函数，编译器会在构造函数中添加代码来创建vptr。 问题来了，如果构造函数是虚的，那么它需要vptr来访问vtable，可这个时候vptr还没产生。
构造函数是用来初始化实例的，实例的类型必须是明确的。使用虚函数，是因为需要在信息不全的情况下进行多态运行

析构函数可以声明为虚函数。如果我们需要删除一个指向派生类的基类指针时，应该把析构函数声明为虚函数。 事实上，只要一个类有可能会被其它类所继承， 就应该声明虚析构函数

通过基类指针或者引用调用的虚函数必定不能被内联。 实体对象调用虚函数或者静态调用时可以被内联，虚析构函数的静态调用也一定会被内联展开。当虚函数表现多态性的时候不能内联。

抽象类：包含纯虚函数的类.抽象类只能作为基类来派生新类使用，不能创建抽象类的对象,只可以抽象类的指针和引用->由抽象类派生出来的类的对象
抽象类中：在成员函数内可以调用纯虚函数，在构造函数/析构函数内部不能使用纯虚函数。
如果一个类从抽象类派生而来，它必须实现了基类中的所有纯虚函数，才能成为非抽象类。
当基类指针指向派生类对象并删除对象时，我们可能希望调用适当的析构函数。 如果析构函数不是虚拟的，则只能调用基类析构函数。

C实现C++的面向对象特性:
封装
C语言中是没有class类这个概念的，但是有struct结构体，我们可以考虑使用struct来模拟；
使用函数指针把属性与方法封装到结构体中。
继承
结构体嵌套
多态
类与子类方法的函数指针不同
在C语言的结构体内部是没有成员函数的，如果实现这个父结构体和子结构体共有的函数呢？我们可以考虑使用函数指针来模拟。但是这样处理存在一个缺陷就是：父子各自的函数指针之间指向的不是类似C++中维护的虚函数表而是一块物理内存，如果模拟的函数过多的话就会不容易维护了。
模拟多态，必须保持函数指针变量对齐

談談 const

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&);
};
```

O

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

?!

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


26

談談 const

當成員函數的 const 和 non-const 版本同時存在，
const object 只會 (只能) 調用 const 版本，
non-const object 只會 (只能) 調用 non-const 版本。

	const object (data members 不得變動)	non-const object (data members 可變動)
const member functions (保證不更改 data members)		
non-const member functions (不保證 data members 不變)		

```
const String str("hello world");
str.print();
```

如果當初設計 string::print() 時未指明 const，那麼上行便是
經由const object 調用non-const member function，會出錯。此非吾人所願

class template std::basic_string<…>
有如下兩個 member functions：

```
charT
operator[] (size_type pos) const
{ …… /* 不必考慮 COW */ }

reference
operator[] (size_type pos)
{ …… /* 必須考慮 COW */ }
```

COW : **C**opy **O**n **W**rite

non-const member functions 可調用 const member functions，反之則不行，會引發：
(VC) error C2662: cannot convert 'this' pointer from 'const class X' to 'class X &'. Conversion loses qualifiers

const常量与#define宏定义常量:
const常量具有类型，编译器可以进行安全检查；#define宏定义没有数据类型，只是简单的字符串替换，不能进行安全检查

const对象默认为文件局部变量
非const变量默认为extern。即要在其它文件中访问仅需要在其它文件中extern声明即可
要使const变量能够在其他文件中访问，必须在文件中显式地指定它为extern。即本文件中也要extern显示声明 并且需要做初始化

如果const位于*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；
允许把非const对象的地址赋给指向const对象的指针。必须使用const void*类型的指针保存const对象的地址。

如果const位于*的右侧，const就是修饰指针本身，即指针本身是常量。指针必须进行初始化

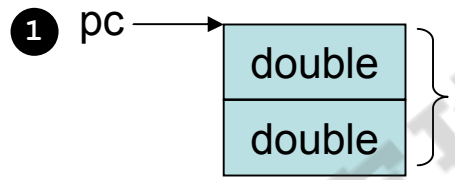
函数中使用const:
修饰函数返回值
const int 本身无意义，因为参数返回本身就是赋值给其他的变量

修饰函数参数
```
void func(const int var); // 传递过来的参数不可变
void func(int *const var); // 指针本身不可变
```
var本身就是形参，在函数内不会改变
对于非内部数据类型的输入参数，用引用传递提高效率,但引用传递可能改变参数，加const修饰即可，函数最终成为
```
void func(const A &a)
```
对于内部数据类型的输入参数，不要将“ 值传递 ”的方式改为“ const 引用传递 ”。否则既达不到提高效率的目的，又降低了函数的可理解性。

类中使用const:
任何不会修改数据成员的函数都应该声明为const类型.类中的const成员变量必须通过初始化列表进行初始化 或与static结合 或在外面初始化 C++11 直接可以在定义出初始化

只有常成员函数才有资格操作常量或常对象

new：先分配 memory, 再調用 ctor



設初值

```
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

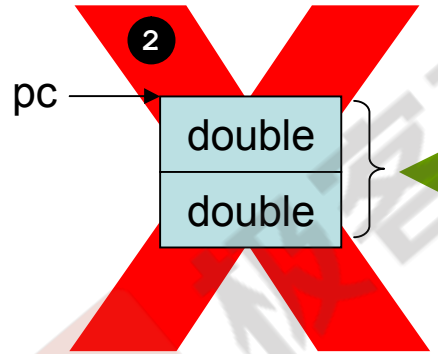
delete：先調用 dtor, 再釋放 memory

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

編譯器轉化為

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

其內部調用 free(pc)



清理

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

new：先分配 memory, 再調用 ctor

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

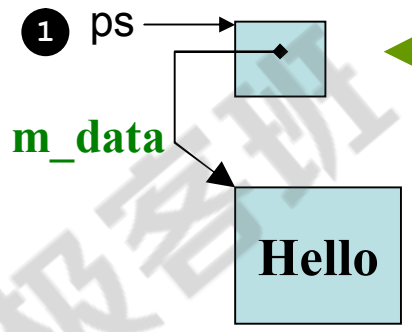
編譯器轉化為

```
String* ps;

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

其內部調用 malloc(*n*)

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

this




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

3

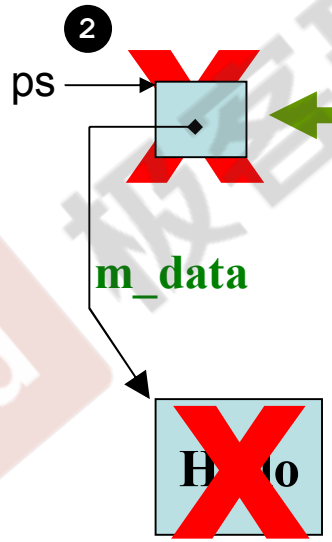
delete：先調用 dtor, 再釋放 memory

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

編譯器轉化為

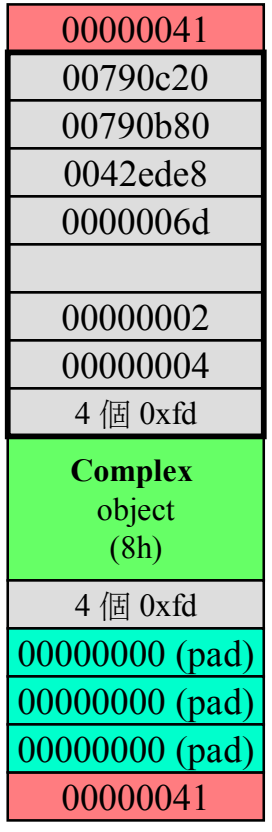
```
1 String::~String(ps);     // 析構函數
2 operator delete(ps);     // 釋放內存
```

其內部調用 free(ps)

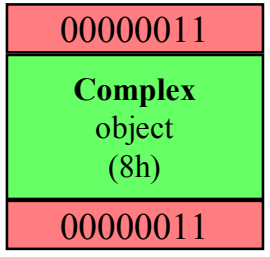



```
class String
{
public:
  ~String()
  { delete[] m_data; }
  ...
private:
  char* m_data;
};
```

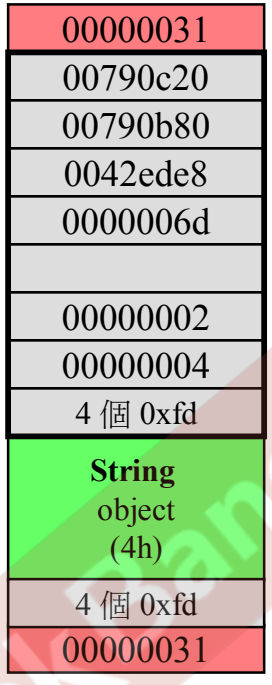
動態分配所得的內存塊 (memory block), in VC



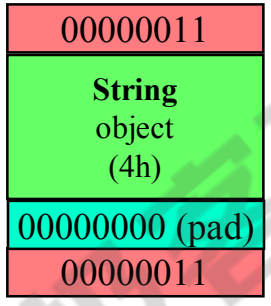
8+(32+4)+(4*2)
➔52
➔64



8+(4*2)
➔16



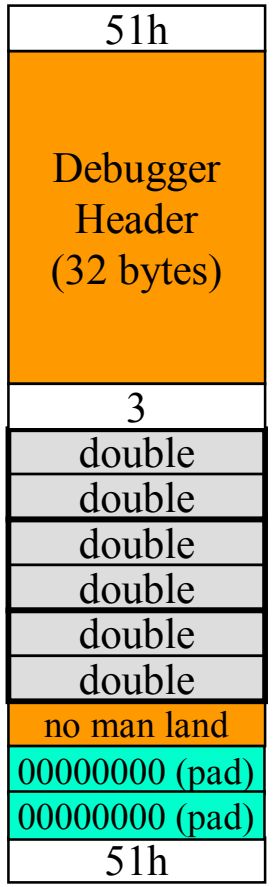
4+(32+4)+(4*2)
➔48



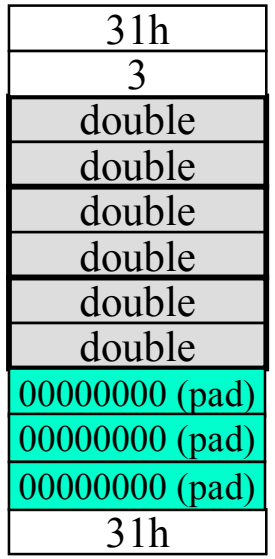
4+(4*2)
➔12
➔16

動態分配所得的 array

Complex* p = new Complex[3];

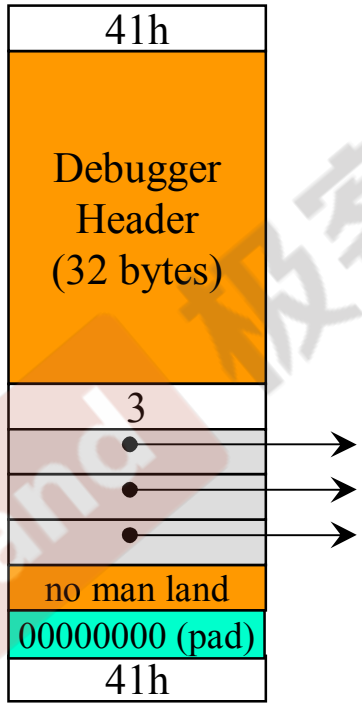


(8*3)+(32+4)+(4*2)+4
➔72
➔80

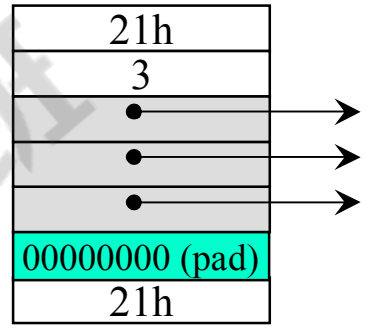


(8*3)+(4*2)+4
➔36
➔48

String* p = new String[3];



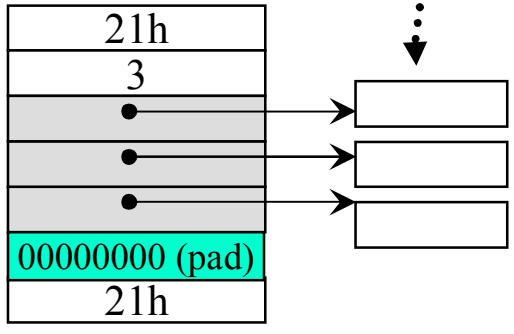
(4*3)+(32+4)+(4*2)+4
➔60
➔64



(4*3)+(4*2)+4
➔24
➔32

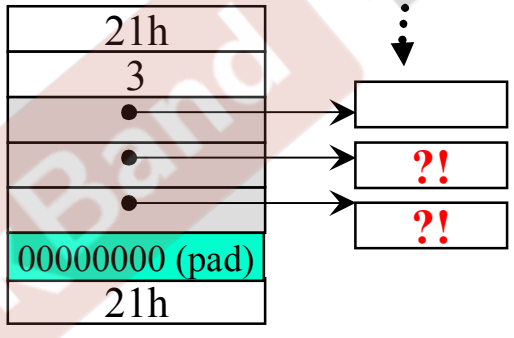
array new 一定要搭配 array delete

```
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 []



global

重載 ::operator new, ::operator delete
::operator new[], ::operator delete[]

小心，這影響無遠弗屆

```
void* myAlloc(size_t size)
{  return malloc(size); }

void myFree(void* ptr)
{  return free(ptr); }

////它們不可以被聲明於一個 namespace 內
inline void* operator new(size_t size)
  { cout << "jjhou global new()     \n";       return myAlloc( size ); }

inline void* operator new[ ](size_t size)
  { cout << "jjhou global new[]()   \n";       return myAlloc( size ); }

inline void  operator delete(void* ptr)
  { cout << "jjhou global delete()  \n";       myFree( ptr );   }

inline void  operator delete[ ](void* ptr)
  { cout << "jjhou global delete[]()\n";       myFree( ptr );   }
```

重載 member operator new/delete

```
Foo* p = new Foo;

...

delete p;
```

```
try {
  1 void* mem  =  operator new(sizeof(Foo));
    p = static_cast<Foo*>(mem);
  2 p->Foo::Foo();
}
```

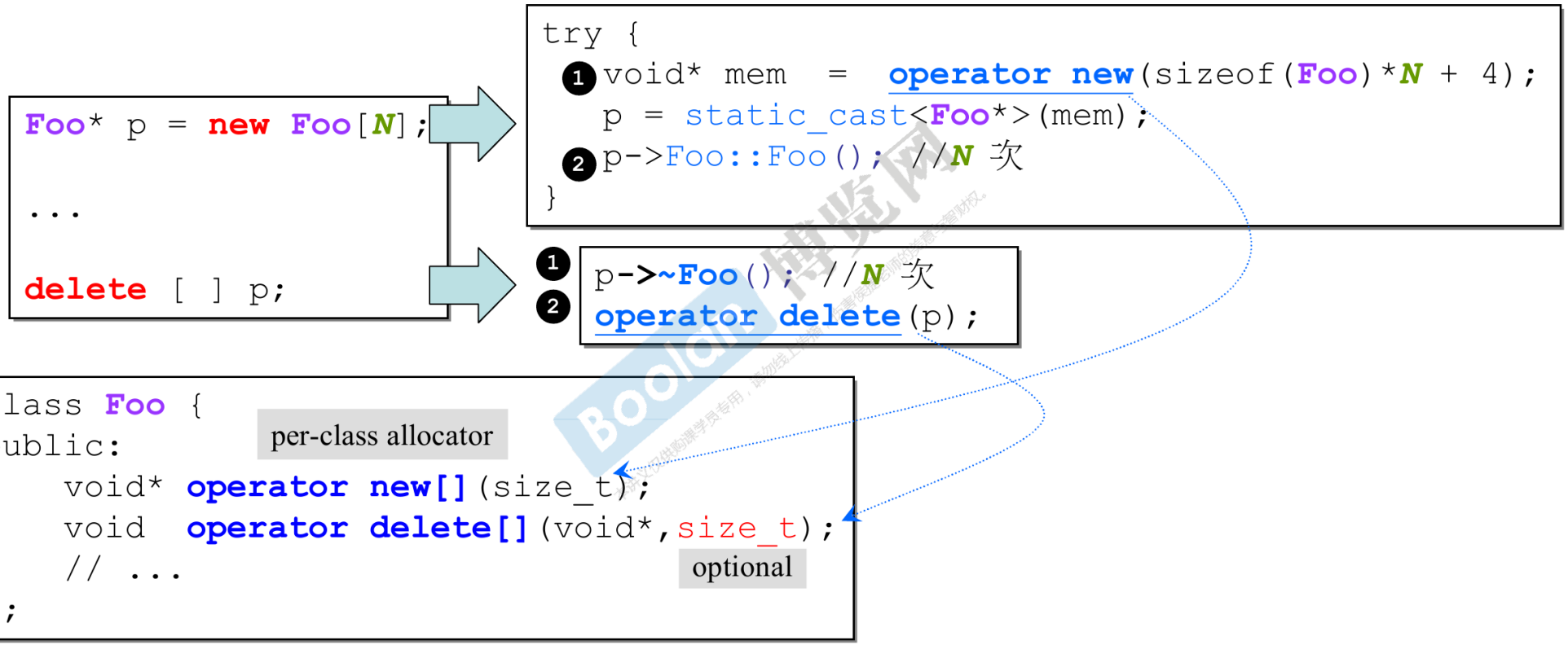
```
1 p->~Foo();
2 operator delete(p);
```

```
class Foo {
public:
    void* operator new(size_t);
    void  operator delete(void*,size_t);
    // ...
};
```

per-class allocator

optional

重載 member operator new[ ] / delete[ ]



示例, 接口

```
class Foo
{
public:
  int _id;
  long _data;
  string _str;

public:
  Foo() : _id(0)      { cout << "default ctor. this=" << this << " id=" <<
  Foo(int i) : _id(i) { cout << "ctor. this=" << this << " id=" << _id <<

//virtual
  ~Foo()              { cout << "dtor. this=" << this << " id=" << _id
  static void* operator new(size_t size);
  static void  operator delete(void* pdead, size_t size);
  static void* operator new[](size_t size);
  static void  operator delete[](void* pdead, size_t size);
};
```

```
Foo* pf = new Foo;
delete pf;

下面強制採 globals
Foo* pf = ::new Foo;
::delete pf;
```

若無 members 就調用 globals

```
void* ::operator new(size_t);
void  ::operator delete(void*);
```

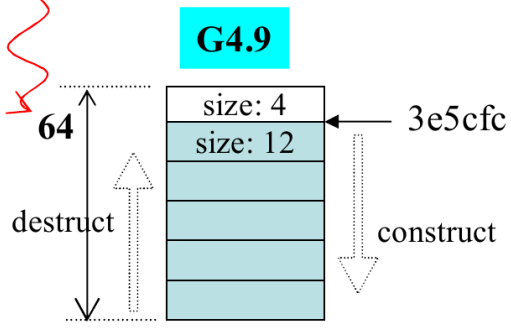
```
void* Foo::operator new(size_t size)  {
    Foo* p = (Foo*)malloc(size);
    cout << ……
    return p;
}
void Foo::operator delete(void* pdead, size_t size)  {
    cout << ……
    free(pdead);
}
void* Foo::operator new[](size_t size)  {
    Foo* p = (Foo*)malloc(size);
    cout << ……
    return p;
}
void Foo::operator delete[](void* pdead, size_t size)  {
    cout << ……
    free(pdead);
}
```

示例

```
cout << "sizeof(Foo)= " << sizeo

1 Foo* p = new Foo(7);
2 delete p;

3 Foo* pArray = new Foo[5];
4 delete [ ] pArray;
```

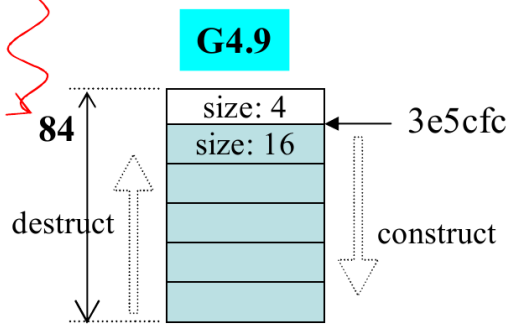
Foo without virtual dtor

```
sizeof(Foo)= 12
1 Foo::operator new(), size=12      return: 0x3e3988
  ctor. this=0x3e3988 id=7
2 dtor. this=0x3e3988 id=7
  Foo::operator delete(), pdead= 0x3e3988  size= 12
3 Foo::operator new[](), size=64    return: 0x3e5cf8
  default ctor. this=0x3e5cfc id=0
  default ctor. this=0x3e5d08 id=0
  default ctor. this=0x3e5d14 id=0
  default ctor. this=0x3e5d20 id=0
  default ctor. this=0x3e5d2c id=0
4 dtor. this=0x3e5d2c id=0
  dtor. this=0x3e5d20 id=0
  dtor. this=0x3e5d14 id=0
  dtor. this=0x3e5d08 id=0
  dtor. this=0x3e5cfc id=0
  Foo::operator delete[](), pdead= 0x3e5cf8  size= 64
```



Foo with virtual dtor

```
sizeof(Foo)= 16
1 Foo::operator new(), size=16      return: 0x3e3988
  ctor. this=0x3e3988 id=7
2 dtor. this=0x3e3988 id=7
  Foo::operator delete(), pdead= 0x3e3988  size= 16
3 Foo::operator new[](), size=84    return: 0x3e5cf8
  default ctor. this=0x3e5cfc id=0
  default ctor. this=0x3e5d0c id=0
  default ctor. this=0x3e5d1c id=0
  default ctor. this=0x3e5d2c id=0
  default ctor. this=0x3e5d3c id=0
4 dtor. this=0x3e5d3c id=0
  dtor. this=0x3e5d2c id=0
  dtor. this=0x3e5d1c id=0
  dtor. this=0x3e5d0c id=0
  dtor. this=0x3e5cfc id=0
  Foo::operator delete[](), pdead= 0x3e5cf8  size= 84
```



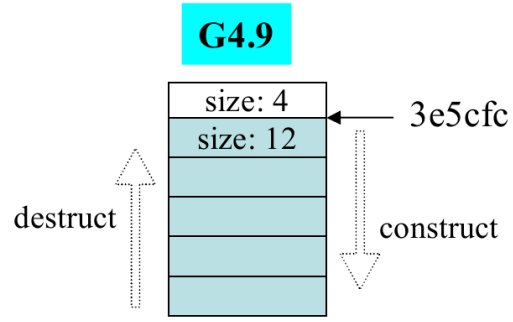
示例

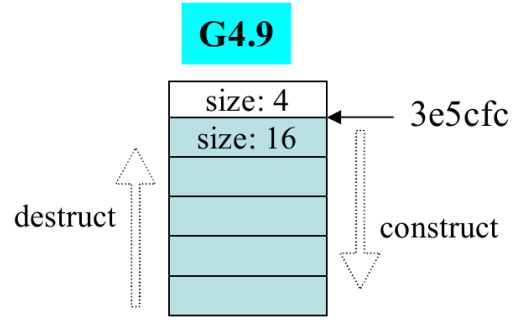
```
1 Foo* p = ::new Foo(7);
2 ::delete p;

3 Foo* pArray = ::new Foo[5];
4 ::delete [ ] pArray;
```

這樣調用 (也就是寫上 global scope operator ::)，會繞過前述所有 overloaded functions, 強迫使用 global version.

```
ctor. this=0x3e5ce0 id=7
dtor. this=0x3e5ce0 id=7
default ctor. this=0x3e5cfc id=0
default ctor. this=0x3e5d08 id=0
default ctor. this=0x3e5d14 id=0
default ctor. this=0x3e5d20 id=0
default ctor. this=0x3e5d2c id=0
dtor. this=0x3e5d2c id=0
dtor. this=0x3e5d20 id=0
dtor. this=0x3e5d14 id=0
dtor. this=0x3e5d08 id=0
dtor. this=0x3e5cfc id=0
```

↑ 都沒有進入我所重載的 operator new(), operator delete(), operator new[](), operator delete[]().





重載 new(), delete()

我們可以重載 class member operator new()，寫出多個版本，前提是每一版本的聲明都必須有獨特的參數列，其中第一參數必須是 **size_t**，其餘參數以 **new** 所指定的 placement arguments 為初值。出現於 **new (.....)** 小括號內的便是所謂 placement arguments。

```
Foo* pf = new (300, 'c') Foo;
```

或稱此為
placement operator delete.

我們也可以重載 class member operator delete()，寫出多個版本。但它們絕不會被 **delete** 調用。只有當 **new** 所調用的 ctor 拋出 exception，才會調用這些重載版的 operator delete()。它只可能這樣被調用，主要用來歸還未能**完全創建成功**的 object 所佔用的 memory。

```
new
operator new
array new
placement new
```

示例

```
class Foo {
public:
  Foo() { cout << "Foo::Foo()" << endl;  }
  Foo(int) { cout << "Foo::Foo(int)" << endl;  throw Bad();  }

  //(1) 這個就是一般的 operator new() 的重載
  void* operator new(size_t size) {
       return malloc(size);
  }
  //(2) 這個就是標準庫已提供的 placement new() 的重載 (的形式)
  //     (所以我也模擬 standard placement new, 就只是傳回 pointer)
  void* operator new(size_t size, void* start) {
       return start;
  }
  //(3) 這個才是嶄新的 placement new
  void* operator new(size_t size, long extra) {
       return malloc(size+extra);
  }
  //(4) 這又是一個 placement new
  void* operator new(size_t size, long extra, char init) {
        return malloc(size+extra);
  }
……(接下頁)
```

```
class Bad { };
```

故意在這兒拋出 expcetion，
測試 placement operator delete.

```
//(5) 這又是一個 placement new, 但故意寫錯第一參數的 type
//       (那必須是 size_t 以符合正常的 operator new)
//!  void* operator new(long extra, char init) {
//            [Error] 'operator new' takes type 'size_t' ('unsigned int')
//                       as first parameter [-fpermissive]
//!     return malloc(extra);
//!  }
…
```

示例 (續)

```
…… (續上頁)
  //以下是搭配上述 placement new 的各個所謂 placement delete.
  //當 ctor 發出異常，這兒對應的 operator (placement) delete 就會被調用.
  //其用途是釋放對應之 placement new 分配所得的 memory.
  //(1) 這個就是一般的 operator delete() 的重載
  void operator delete(void*,size_t)
  { cout << "operator delete(void*,size_t)  " << endl;  }

  //(2) 這是對應上頁的 (2)
  void operator delete(void*,void*)
  { cout << "operator delete(void*,void*)  " << endl;  }

  //(3) 這是對應上頁的 (3)
  void operator delete(void*,long)
  { cout << "operator delete(void*,long)  " << endl;  }

  //(4) 這是對應上頁的 (4)
  void operator delete(void*,long,char)
  { cout << "operator delete(void*,long,char)  " << endl; }

private:
      int m_i;
};
```

即使 operator delete(…) 未能一一對應於 operator new(…), 也不會出現任何報錯.
你的意思是 : 放棄處理 ctor 發出的異常.

ctor拋出異常

```
Foo start;
① Foo* p1 = new Foo;
② Foo* p2 = new (&start) Foo;
③ Foo* p3 = new (100) Foo;
④ Foo* p4 = new (100,'a') Foo;
⑤ Foo* p5 = new (100) Foo(1);
  Foo* p6 = new (100,'a') Foo(1);
  Foo* p7 = new (&start) Foo(1);
  Foo* p8 = new Foo(1);
```

```
Foo::Foo()
① operator new(size_t size), size= 4
Foo::Foo()
② operator new(size_t size, void* start), size= 4  start= 0x22fe8c
Foo::Foo()
③ operator new(size_t size, long extra)  4 100
Foo::Foo()
④ operator new(size_t size, long extra, char init)  4 100 a
Foo::Foo()
⑤ operator new(size_t size, long extra)  4 100
Foo::Foo(int)
terminate called after throwing an instance of 'jj07::Bad'
```

ctor 拋出異常 ⇨ 奇怪, G4.9 沒調用 operator delete (void*,long), 但 G2.9 確實調用了.

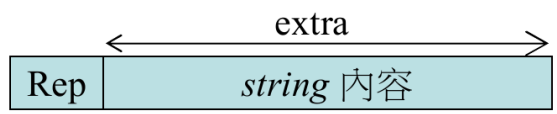
VC6 warning C4291: 'void *__cdecl Foo::operator new(~~~)' no matching operator delete found; memory will not be freed if initialization throws an exception

basic_string 使用 new(*extra*) 擴充申請量

```
template <…>
class basic_string
{
private:
  struct Rep {
    …
    void release () { if (--ref == 0) delete this; }
    inline static void * operator new (size_t, size_t);
    inline static void operator delete (void *);
    inline static Rep* create (size_t);
    …
  };
  …
};
```

```
template <class charT, class traits, class Allocator>
inline void * basic_string <charT, traits, Allocator>::Rep::
operator new (size_t s, size_t extra)
{
  return Allocator::allocate(s + extra * sizeof (charT));
}
template <class charT, class traits, class Allocator>
inline void basic_string <charT, traits, Allocator>::Rep::
operator delete (void * ptr)
{
  Allocator::deallocate(ptr, sizeof(Rep) +
                        reinterpret_cast<Rep *>(ptr)->res *
                        sizeof (charT));
}
```

```
template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>::Rep*
basic_string <charT, traits, Allocator>::Rep::
create (size_t extra)
{
  extra = frob_size (extra + 1);
  Rep *p = new (extra) Rep;
  …
  return p;
}
```



assert

断言，是宏，而非函数 如果它的条件返回错误，则终止程序执行。
可以通过#define NDEBUG来关闭 assert，但是需要在源代码的开头，include <assert.h> 前
assert(x==7);
主要用于检查逻辑上不可能的情况

extern "C"

C++调用C函数
C++文件中函数编译后生成的符号与C语言生成的不同。因为C++支持函数重载，C++函数编译后生成的符号带有函数参数类型的信息，而C则没有。C++中使用C语言实现的函数编译链接的时候，会出错
在头文件中

```
#ifdef __cplusplus
extern "C"{
#endif
int add(int x,int y);
#ifdef __cplusplus
}
#endif
```

既能被C调用，又能被C++调用
编译的时候一定要注意，先通过gcc生成c文件的中间文件.o

struct

C和C++中的Struct区别

C	C++
不能将函数放在结构体声明	能将函数放在结构体声明
在C结构体声明中不能使用C++访问修饰符。	public、protected、private 在C++中可以使用。
在C中定义结构体变量，如果使用了下面定义必须加struct。	可以不加struct
结构体不能继承（没有这一概念）。	可以继承
若结构体的名字与函数名相同，可以正常运行且正常的调用！	若结构体的名字与函数名相同，使用结构体，只能使用带struct定义！

struct/class
struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

enum

问题:作用域不受限,会容易引起命名冲突。(struct可解决) 会隐式转换为int。用来表征枚举变量的实际类型不能明确指定，从而无法支持枚举类型的前向声明。

C++11 的枚举类
新的enum的作用域不在是全局的 不能隐式转换成其他类型 可以指定用特定的类型来存储enum

```
enum class Color2
{
    RED=2,
    YELLOW,
    BLUE
};
Color2 c2 = Color2::RED;
cout << static_cast<int>(c2) << endl; //等价于enum class Color2:int
---
enum class Color3:char;  // 前向声明
enum class Color3:char   //定义
{
    RED='r',
    BLUE
};
char c3 = static_cast<char>(Color3::RED);
```

普通类中的枚举常量不会占用对象的存储空间，它们在编译时被全部求值。缺点是：它的隐含数据类型是整数，其最大值有限，且不能表示浮点。

union:
联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。

默认访问控制符为 public
可以含有构造函数、析构函数
不能含有引用类型的成员
不能继承自其他类，不能作为基类
不能含有虚函数
匿名 union 在定义所在作用域可直接访问 union 成员
匿名 union 不能包含 protected 成员或 private 成员
全局匿名联合必须是静态（static）的

The End