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



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

• 學習 Classes 之間的關係

- 繼承 (inheritance)
- 複合 (composition)
- 委託 (delegation)

Object Based (基於對象)

Object Oriented (面向對象)

你將獲得的代碼

complex.h complex-test.cpp

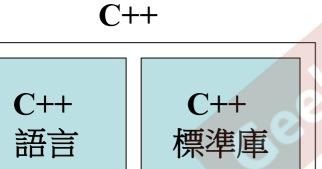
string.h string-test.cpp



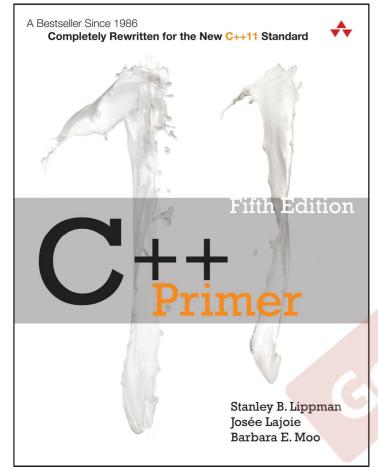
C++ 的歷史

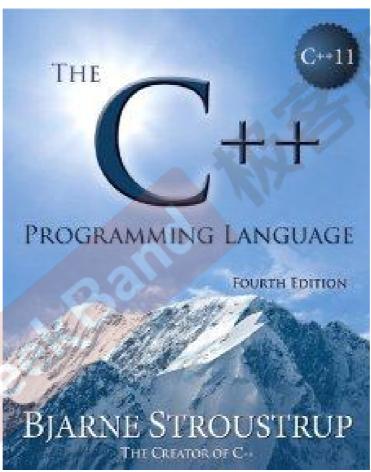
- B 語言 (1969)
- C 語言 (1972)
- C++ 語言 (1983) (new C → C with Class → C++)
- Java 語言
- C# 語言

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

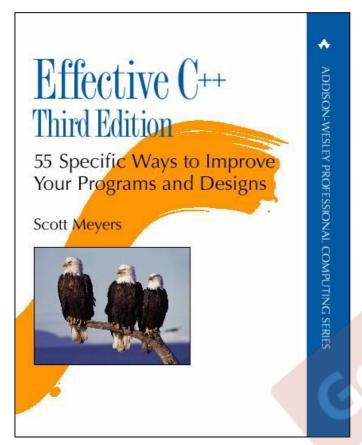


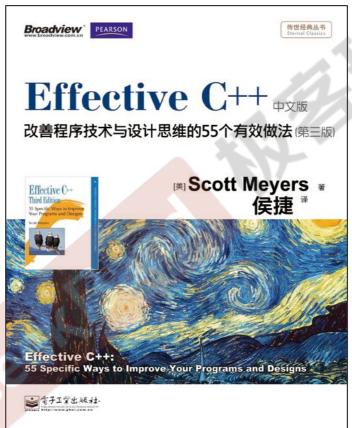
Bibliography (書目誌)



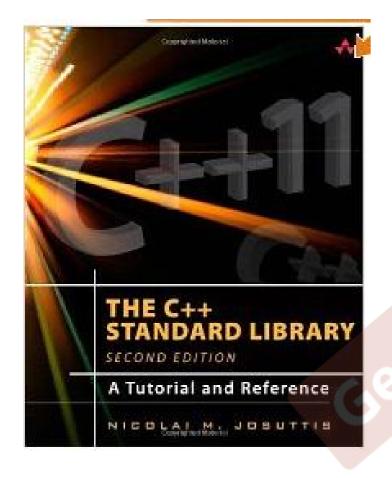


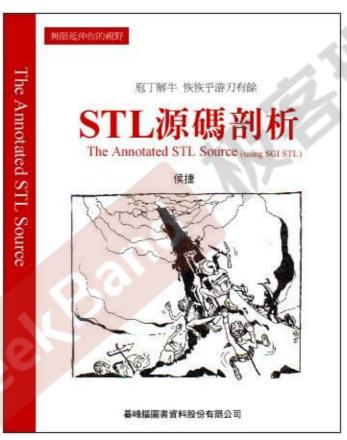
Bibliography (書目誌)



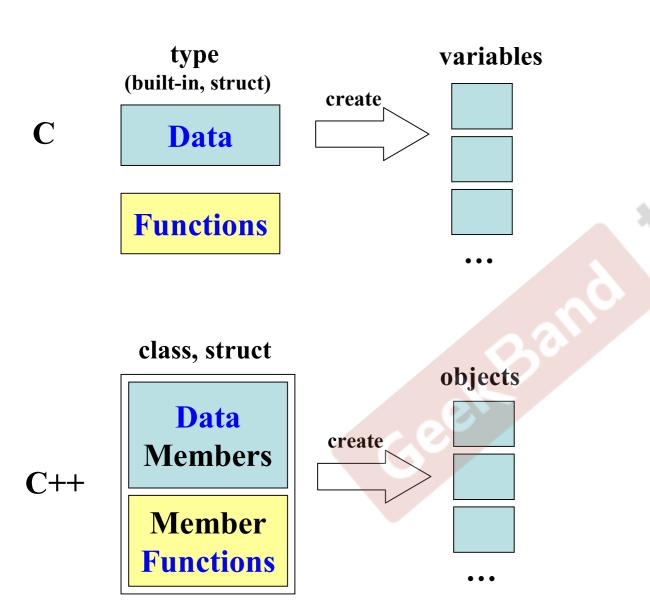


Bibliography (書目誌)



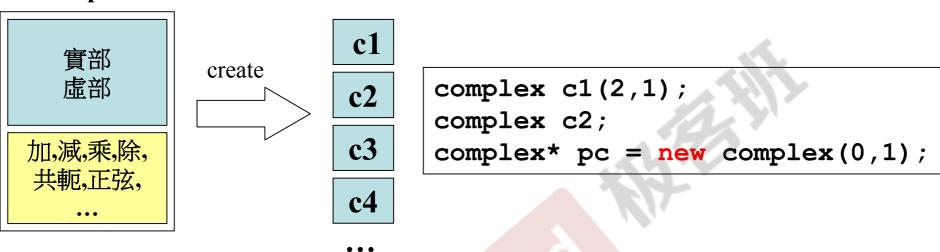


C vs. C++, 關於數據和函數

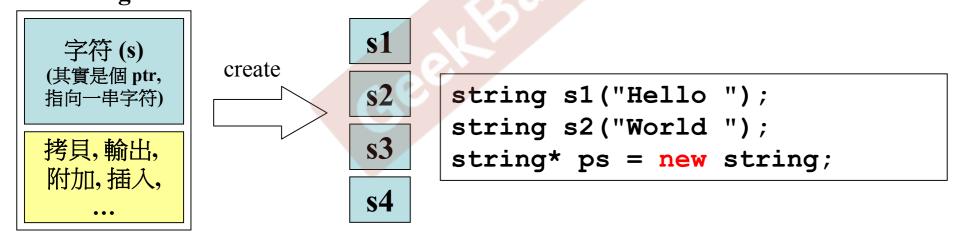


■ C++, 關於數據和函數

complex



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:

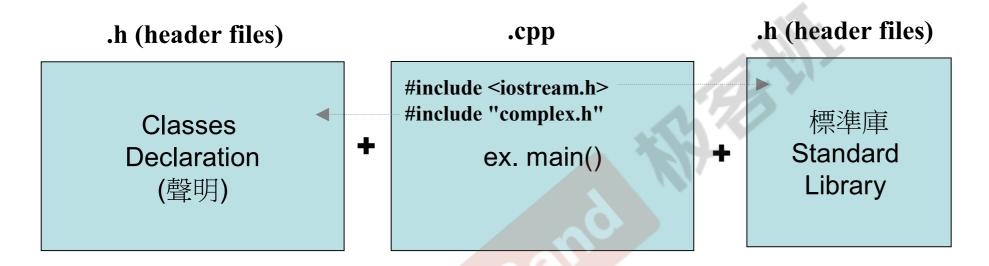
•Class without pointer member(s)

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

```
#include <iostream>
C++
#include <iostream.h>
using namespace std;
int main()
  int i = 7;
  cout << "i=" << i << endl;</pre>
  return 0;
```

```
#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-test.h

complex.h

```
#ifndef __COMPLEX___ guard (防衛式聲明)

....
#endif
```

```
#include <iostream>
#include "complex.h"
using namespace std;
int main()
  complex c1(2,1);
  complex c2;
  cout << c1 << endl;</pre>
  cout << c2 << endl;</pre>
  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;</pre>
  return 0;
```

DEFINE条件编译 避免头文件被多次 包含

Header (頭文件) 的佈局

```
#ifndef
              COMPLEX
   #define COMPLEX
   #include <cmath>
   class ostream;
                               forward declarations
   class complex;
                                   (前置聲明)
   complex&
       doapl (complex* ths, const complex& r);
   class complex
                                  class declarations
                                    (類 - 聲明)
   complex::function ...
                                  class definition
2
                                    (類 - 定義)
   #endif
```

class 的聲明 (declaration)

class head class complex class body public: complex (double r = 0, double i = 0) : re (r), im (i) **{** } complex& operator += (const complex&); double real () const { return re; } 有些函數在此直接定義, double imag () const { return im; } 另一些在 body 之外定義 private: double re, im; friend complex& doapl (complex*, const complex&); **}**;

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

class template (模板) 簡介

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

内联函数要求短小

```
class complex
public:
  complex (double r = 0, double i = 0)
    : re (r), im (i)
                                         函數若在 class body
                                         內定義完成,便自動
  complex& operator += (const complex&);
 double real () const { return re; }
                                         成為 inline 候選人
 double imag () const { return im; }
private:
 double re, im;
 friend complex& doapl (complex*, const complex&);
};
```

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

access level (訪問級別)

};

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);
  cout << c1.re;
  cout << c1.im;
}</pre>
cout << c1.im;
}</pre>
cout << c1.imag();
}</pre>
```

constructor (ctor, 構造函數)

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

default argument (默認實參)

```
(默認實參)
class complex
public:
  complex (double r = 0, double i = 0)
                                          initialization list
    : re (r), im (i)
                                          (初值列,初始列)
  complex& operator += (const complex&);
 double real () const { return re; }
 double imag () const { return im; }
private:
 double re, im;
  friend complex& doapl (complex*, const complex&);
};
```

这种写法不好,建议使 用列表初始化

assignments

(賦值)

```
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)
                                             complex c1;
   : re (r), im (i)
                                             complex c2();
 complex () : re(0), im(0) { }
                                                     构造函数经常重载
 complex& operator += (const complex&);
 double real () const { return re; }
                                                    都使用的默认值,有歧
 double imag () const { return im; }
                                                     义,因此不对
private:
 double re, im;
 friend complex& doapl (complex*, const complex&);
};
                                            real 函數編譯後的實際名稱可能是:
void real(double r) const {
                                             ?real@Complex@@QBENXZ
                                             ?real@Complex@@QAENABN@Z
```

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

```
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;
```

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;
```

单例模式 构造函数在pri vate中,因此不允许外部声明类对象。 只能通过一个stati c成员函数来声明一个单例对象

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

const member functions (常量成員函數)

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&); **}**;

养成好习惯,不改变类成员变量 的函数后面加const.

成员函数的参数默认含thi s指针, 函数后面const是对thi s指针const。

```
complex c1(2,1);
cout << c1.real();
cout << c1.imag();
}</pre>
```

?!

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

如果real ()定义时没加const,那么这个就是错误的。 因为c1是常量,而real 的参数是非const的this指针,传参时导致权限扩大,发送错误。

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

class complex 参数传递时 要考虑by value还是by reference. public: complex (double r = 0, double i = 0) 按引用传递相当于按指针传递,不需要 复制一份变量(当变量占空间大时,可 : re (r), im (i) 以节约复制变量的时间),只在栈上存一 complex& operator += (const complex&); 个指针。 double real () const { return re; } double imag () const { return im; } 加const保证参数不被改变。 private: double re, im; friend complex& doapl (complex*, const complex&); };

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

c2 += c1;
cout << c2;
}</pre>
```

返回值傳遞:return by value vs. return by reference (to const)

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;

  cout << c1;
  cout << c2 << c1;
}</pre>
```

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
```

友元函数可以访问类的私有 成员。 它的声明可以放在共有、私 有、保护三者任意部分

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;
};
```

相同类的不同实例之间,可以利用成员函数巧妙的 访问其他实例的私有成员

```
{
   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

```
inline complex&
 doapl(complex* ths, const complex& r)
                    第一參數將會被改動
 ths->re += r.re;
 ths->im += r.im;
                    第二參數不會被改動
 return *ths;
inline complex&
complex::operator += (const complex& r)
 return doapl (this, r);
```

pass by ref:参数占用空间大,要靠考虑是否按引用传参 return by ref:是否需要额外空间保存结果

c1+c2时,不需要额外空间存储结 果,结果保存在了c1中,因此可以 return by ref

thi s是默认存在的参数

operator overloading (操作符重載-1, 成員函數) this

```
inline complex&
 doapl(complex* ths, const complex& r)
 ths->re += r.re;
 ths->im += r.im;
  return *ths;
inline complex&
complex::operator += (const complex& r)
 return doapl (this, r);
```

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

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

return by reference 語法分析

傳遞者無需知道接收者是以 reference 形式接收

```
inline complex&
  doap1(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);
                                       c1;
```

class body 之外的各種定義 (definitions)



```
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);
}</pre>
```

operator overloading (操作符重載-2, 非成員函數) (無 this)



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),
                                                             complex c1(2,1);
                  imag(x) + imag(y));
                                                             complex c2;
inline complex
                                                             c2 = c1 + c2;
                                                             c2 = c1 + 5;
operator + (const complex& x, double y) ◀
                                                             c2 = 7 + c1;
 return complex (real (x) + y, imag (x));
inline complex
operator + (double x, const complex& y)
 return complex (x + real (y), imag (y));
```

temp object (臨時對象) typename ();



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);
}</pre>
```

class body 之外的各種定義 (definitions)



```
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)

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

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



```
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);
}</pre>
```

operator overloading (操作符重載), 非成員函數



```
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);
}</pre>
```

operator overloading (操作符重載), 非成員函數

```
inline complex
conj (const complex& x)
 return complex (real (x), -imag (x));
#include <iostream.h>
ostream&
operator << (ostream& os, const complex& x)</pre>
 return os << '(' << real (x) << ','
            << imag (x) << ')';
```

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

```
2-7
```

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

<< imag (x) << ')';

編程示例



```
編程-動畫
```

```
#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

```
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-動書

```
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

```
complex c1(9,8);
cout << c1;
cl << cout;
cout << c1 << endl;</pre>
```

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

```
#ifndef __MYSTRING__ string.h
#define __MYSTRING__
```

class String
{
...
};

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

Global-function(...) ...

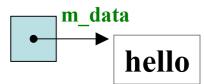
#endif
```

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

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

Big Three, 三個特殊函數

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; **}**;

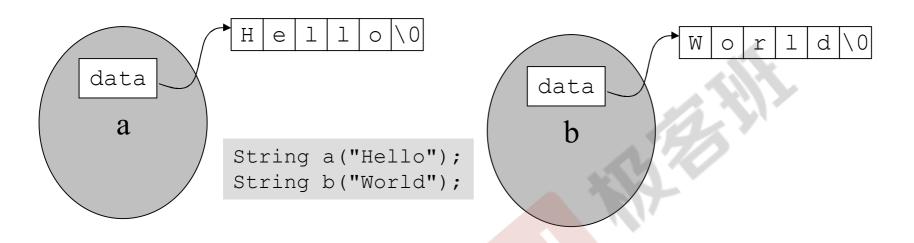


ctor和dtor(構造函數和析構函數)

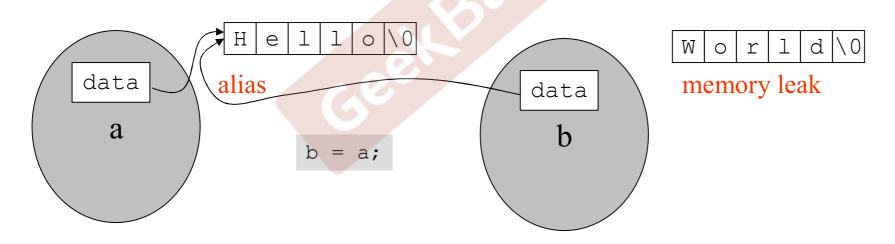
2-1

```
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 = ' \setminus 0';
                                               hello
inline
String::~String()
                         String s1(),
                         String s2("hello");
   delete[] m data;
                         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

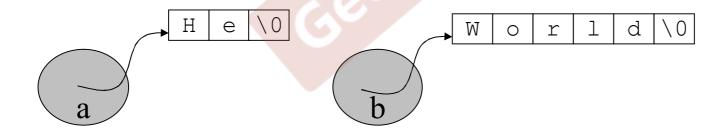
```
inline
String::String(const String& str)
{
    m_data = new char[ strlen(str.m_data) + 1 ];
    strcpy(m_data, str.m_data);
}
```

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

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

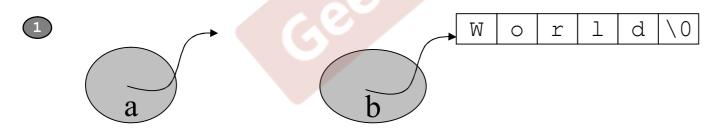
2-3

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



2-3

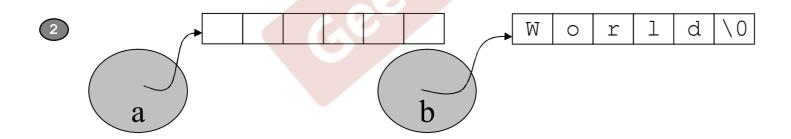
```
inline
String& String::operator=(const String& str)
  if (this == &str) 檢測自我賦值
     return *this;
                    (self assignment)
  delete[] m data;
  m data = new char[ strlen(str.m data) + 1 ];
  strcpy(m_data, str.m data);
  return *this;
```



inline
String& String::operator=(const String& str)
{

if (this == &str) 檢測自我賦值
 return *this; (self assignment)

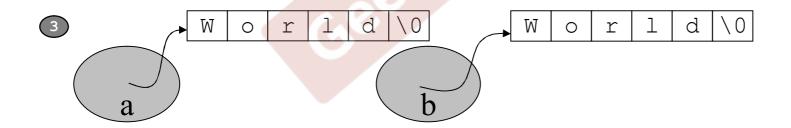
① delete[] m_data;
② m_data = new char[strlen(str.m_data) + 1];
③ strcpy(m_data, str.m_data);
 return *this;



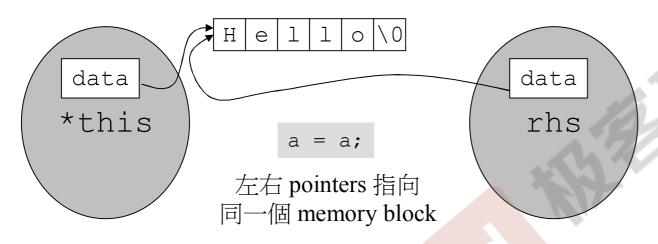
inline
String& String::operator=(const String& str)
{

if (this == &str) 檢測自我賦值
 return *this; (self assignment)

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



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



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

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

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。



```
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
                                   1 pc
                                                  設初值
                                                        public:
                                           double
                                                       Complex(...) {...}
                                           double
                                                        private:
Complex* pc = new Complex(1,2);
                                                          double m real;
                                                          double m imag;
                                                        };
                             其內部調用 malloc (n)
編譯器轉化為
Complex *pc;
  void* mem = operator new( sizeof(Complex) ); //分配內存
  pc = static cast<Complex*>(mem);
                                               //轉型
                                               //構造函數
  pc->Complex::Complex(1,2);
                               Complex::Complex(pc,1,2);
                                               this
```

delete: 先調用 dtor, 再釋放 memory

```
class Complex
                                    2
                                рс
                                                    public:
                                      double
                                                   1 ~Complex() {...}
                                      double
Complex* pc = new Complex(1,2);
                                                    private:
                                                      double m real;
delete pc;
                                                      double m imag;
                                                    };
編譯器轉化為
Complex::~Complex(pc); // 析構函數
operator delete(pc);
                         釋放內存
                其內部調用 free (pc)
```

new: 先分配 memory, 再調用 ctor 1 ps m_data| String* ps = new String("Hello"); Hello 編譯器轉化為 其內部調用 malloc (n) String* ps; void* mem = operator new(sizeof(String)); //分配內存 ps = static_cast<String*>(mem); //轉型 //構造函數 ps->String::String("Hello"); String::String(ps, "Hello");

this

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

delete: 先調用 dtor, 再釋放 memory

```
public:
                                    ps
                                                    - ~String()
                                                     { delete[] m data; }
String* ps = new String("Hello");
                                       m_data
                                                   private:
delete ps;
                                                     char* m data;
                                                   };
            編譯器轉化為
                     // 析構函數
String::~String(ps);
                         釋放內存
operator delete(ps);
               其內部調用 free (ps)
```

class String

動態分配所得的內存塊 (memory block), in VC

| 00000041 |
|----------|
| 00790c20 |
| 00790b80 |
| 0042ede8 |
| 0000006d |
| |

00000002

4個 0xfd

Complex

object (8h)

4個 0xfd

00000000 (pad)

00000000 (pad)

00000000 (pad)

00000041

00000011

Complex object

(8h)

00000011

→16

00000031

00790c20 00790b80

0042ede8

0000006d

00000002

00000004

4個 0xfd

String

object (4h)

4個 0xfd

00000031

4+(32+4)+(4*2)

→48

00000011

String object

(4h)

00000000 (pad) 00000011

4+(4*2)

→12

→16

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

→52

→64

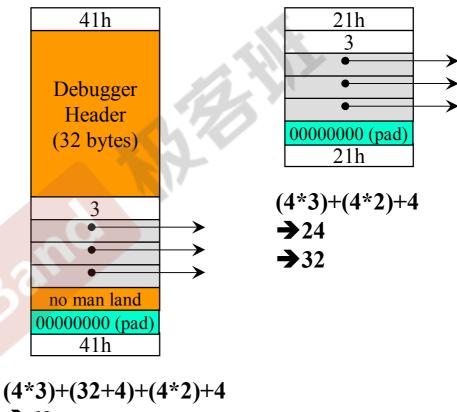
動態分配所得的 array

Complex* p = new Complex[3];

| - |
|----------------------------------|
| 51h |
| Debugger Header (32 bytes) |
| 3 |
| double |
| no man land |

| _ |
|----------------|
| 31h |
| 3 |
| double |
| 00000000 (pad) |
| 00000000 (pad) |
| 00000000 (pad) |
| 31h |

String* p = new String[3];



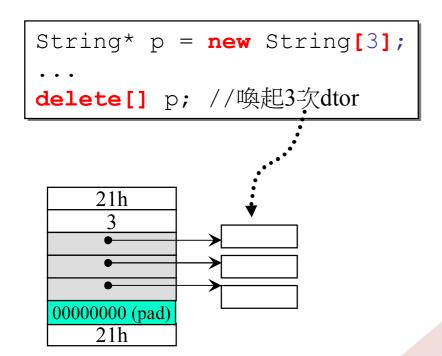
00000000 (pad) 00000000 (pad)

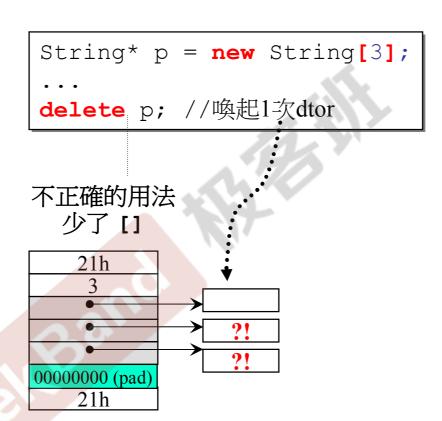
51h

→72

→80

array new 一定要搭配 array delete







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;
```

```
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;
```

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

c1

non-static data members

complex c1,c2,c3;

cout << c1.real();</pre>

cout << c2.real();</pre>

non-static data members

c2

non-static data members

c3

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

this</pre>

static

data members

non-static member functions

this

static

member functions

```
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);
                                調用 static 函數的方式有二:
  Account a;
                                (1) 通過 object 調用
  a.set_rate(7.0);
                                (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 ostream : virtual public ios
                                              extern IO ostream withassign cout;
 public:
    ostream& operator<<(char c);
    ostream& operator << (unsigned char c) { return (*this) << (char)c; }
    ostream& operator<<(signed char c) { return (*this) << (char)c; }</pre>
    ostream& operator<<(const char *s);</pre>
    ostream& operator<<(const unsigned char *s)</pre>
       { return (*this) << (const char*)s; }
    ostream& operator<<(const signed char *s)</pre>
       { return (*this) << (const char*)s; }
    ostream& operator<<(const void *p);</pre>
    ostream& operator<<(int n);</pre>
    ostream& operator<<(unsigned int n);</pre>
    ostream& operator << (long n);
    ostream& operator<<(unsigned long n);</pre>
    . . .
```

class IO ostream withassign

: public ostream {

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)**

```
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;
};</pre>
```

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

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;
}</pre>
```

using declaration

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

int main()
{
   std::cin << ...;
   cout << ...;
   return 0;
}</pre>
```

```
#include <iostream.h>

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

    return 0;
}</pre>
```

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

•...



- •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

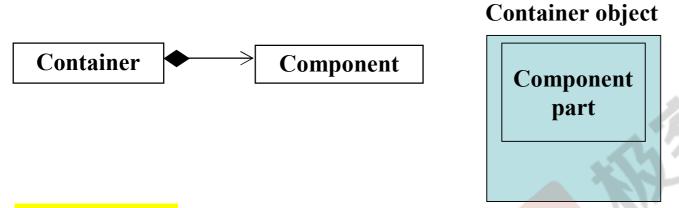
```
template <class T>
class queue {
                                          deque
                          queue
protected:
deque<T> c; // 底層容器
public:
  // 以下完全利用 c 的操作函數完成
 bool empty() const { return c.empty(); }
  size type size() const { return c.size(); }
 reference front() { return c.front(); }
  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>
                            Sizeof : 16 * 2 + 4 + 4
class queue {
protected:
                       template <class T>
                                                            Sizeof: 4 * 4
deque<T> c; ◀
                       class deque {
                       protected:
                                                       template <class T>
};
                          Itr<T> start;
                                                       struct Itr {
                          Itr<T> finish;
                                                              cur;
                          丁**
                                 map;
                                                              first;
                          unsigned int map_size;
                                                              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() { ... };
```

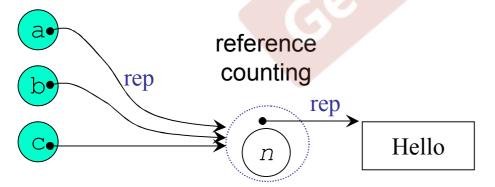
析構由外而內

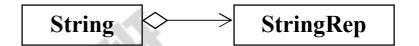
Container 的析構函數首先執行自己,然後才調用 Component 的析構函數。

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

Delegation (委託). Composition by reference.

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

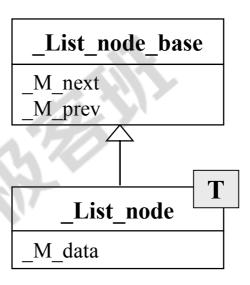




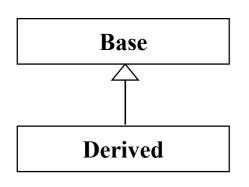
```
// file String.cpp
#include "String.hpp"
namespace ·
class StringRep {
friend class String;
    StringRep(const char* s);
    ~StringRep();
    int count;
    char* rep;
};
String::String() { ... }
```

Inheritance (繼承), 表示 is-a

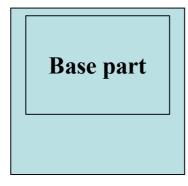
```
struct _List_node_base
  List node base* M next;
 List node base* M prev;
};
template<typename Tp>
struct List node
  : public List node base
 _Tp _M_data;
```



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 (虚函數)

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

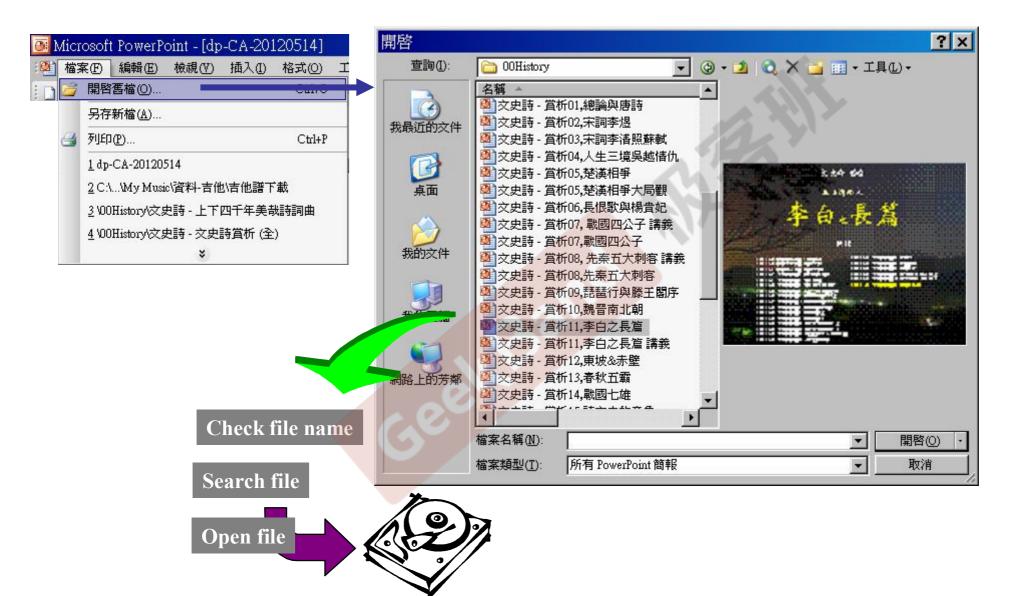
virtual 函數:你希望 derived class 重新定義 (override, 覆寫) 它,且你對 它已有默認定義。

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

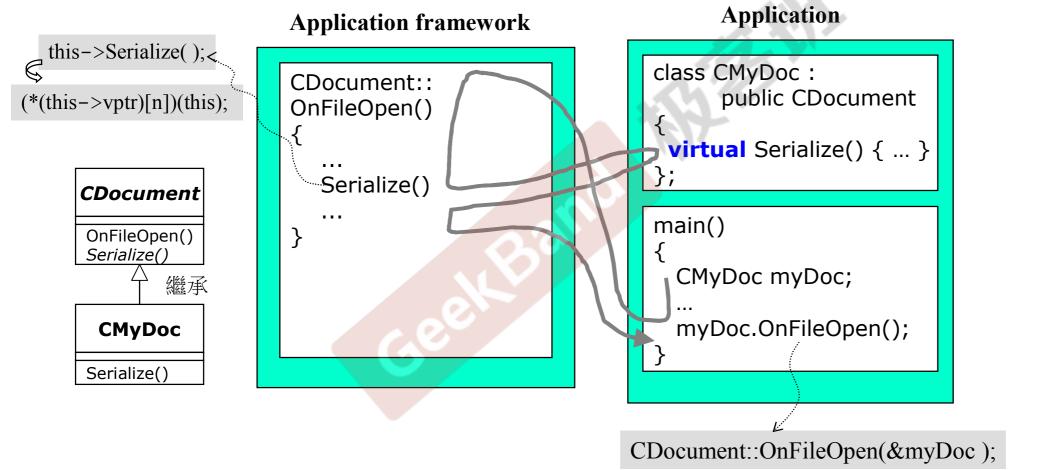
impure virtual

Inheritance (繼承) with virtual



Inheritance (繼承) with virtual

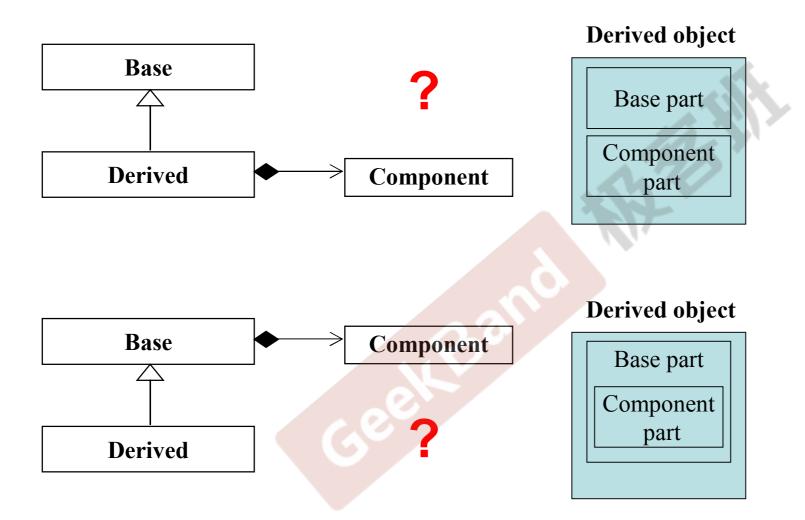
Template Method



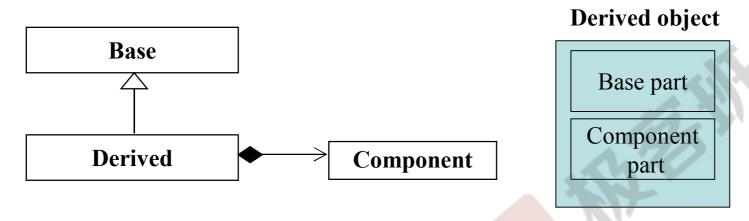
Inheritance (繼承), 表示 is-a

```
22 class CMyDoc : public CDocument
01 #include <iostream>
                                          23 {
02 using namespace std;
                                          24 public:
0.3
                                          25
                                                 virtual void Serialize()
04
                                          2.6
05 class CDocument
                                          27
                                                   // 只有應用程序本身才知道如何讀取自己的文件(格式)
06 {
                                          28
                                                   cout << "CMyDoc::Serialize()" << endl;</pre>
07 public:
                                          29
       void OnFileOpen()
0.8
                                          30 };
09
10
        // 這是個算法,每個 cout 輸出代表一個實際動作
11
        cout << "dialog..." << endl;</pre>
12
        cout << "check file status..." << endl;</pre>
13
        cout << "open file..." << endl;</pre>
14
        Serialize();
15
        cout << "close file..." << endl;</pre>
                                                  31 int main()
16
        cout << "update all views..." << endl;</pre>
                                                  32 {
17
                                                  33
                                                        CMyDoc myDoc; // 假設對應[File/Open]
18
                                                  34
                                                       myDoc.OnFileOpen();
19
      virtual void Serialize() { };
                                                  35 }
20 };
```

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

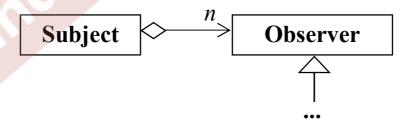
Derived 的析構函數首先執行自己, 然後調用 Component 的 析構函數,

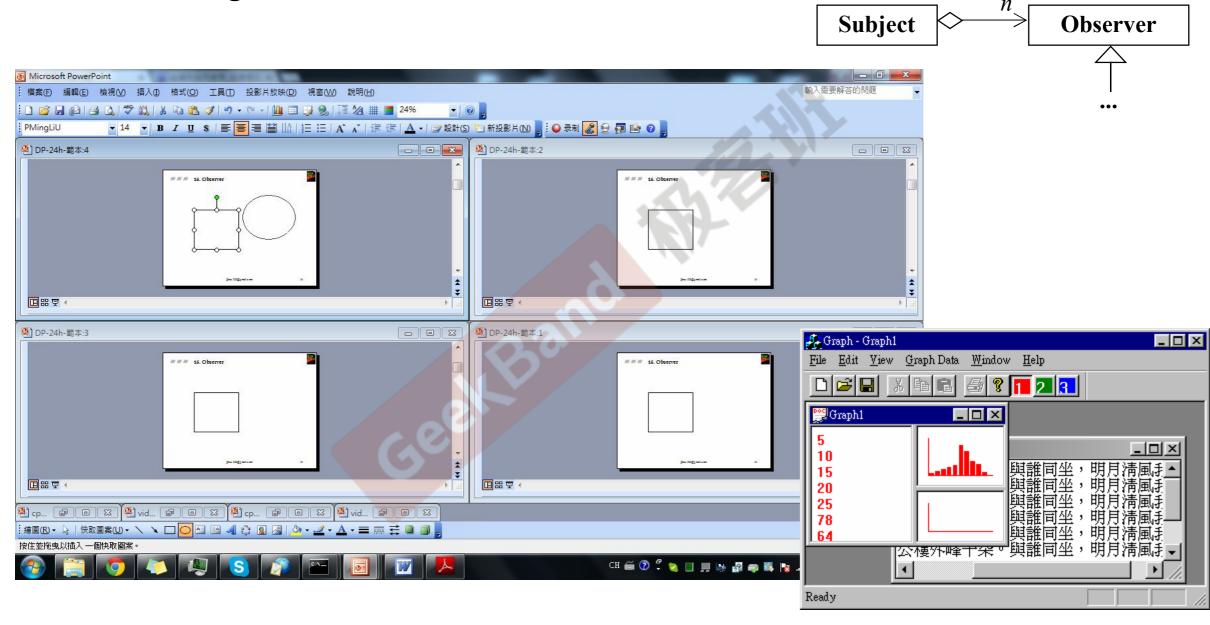
然後調用 Base 的析構函數。 Derived::~Derived(...) { ... ~Component(), ~Base() }

```
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

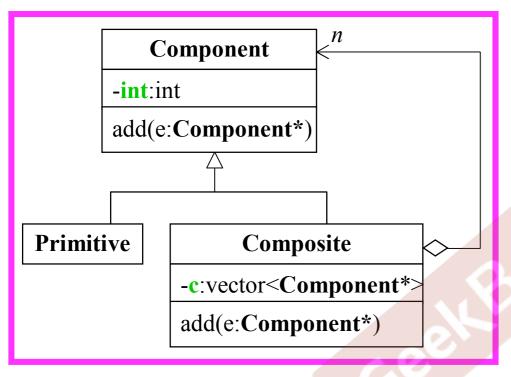
```
class Observer
{
  public:
    virtual void update(Subject* sub, int value) = 0;
};
```





```
class Observer
                                                                                                   Subject subj;
class Subject
                                                                                                   Observer1 o1(&subj, 4);
                                                          public:
                                                                                                   Observer1 o2(&subj, 3);
  int m value;
                                                           virtual void update(int value) = 0;
                                                                                                   Observer2 o3(&subj, 3);
  vector<Observer*> m_views
                                                                                                   subj.set val(14);
 public:
  void attach(Observer* obs)
    m views.push back(obs);
                                             class Observer1: public Observer
                                                                                      class Observer2: public Observer
  void set val(int value)
                                               int m div;
                                                                                        int m mod;
                                              public:
    m value = value;
                                                                                       public:
                                               Observer1(Subject *model, int div)
                                                                                        Observer2(Subject *model, int mod)
    notify();
                                                 model->attach(this);
                                                                                          model->attach(this);
  void notify()
                                                 m div = div;
                                                                                          m \mod = \mod;
    for (int i = 0; i < m views.size(); ++i)
     m_views[i]->update(m_value);
                                                                                         /* virtual */void update(int v)
                                                /* virtual */void update(int v)
                                                     • • •
                                             };
                                                                                      };
```

Composite

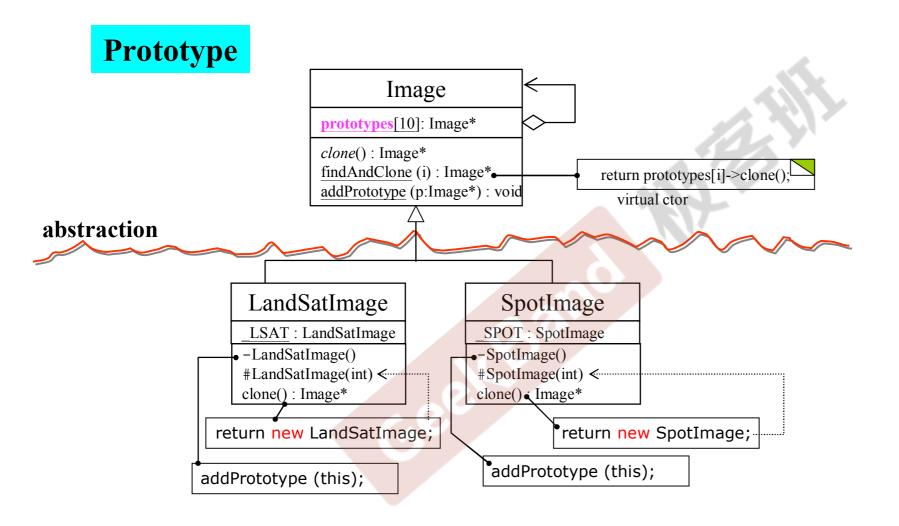


```
class Component
{
  int value;
  public:
    Component(int val) { value = val; }
    virtual void add( Component* ) { }
};
```

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







```
Design Patterns
Explained Simply
```

```
#include <iostream.h>
   enum imageType
    LSAT, SPOT
05
   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
                                                      enum imageType
     public:
                                                      { LSAT, SPOT };
      imageType returnType() {
         return LSAT;
06
      void draw()
07
         cout << "LandSatImage::draw " << iid << endl;</pre>
09
      // 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)
16
17
         id = count++;
18
     private:
19
      // Mechanism for initializing an Image subclass - this causes the
20
      // 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
23
       LandSatImage()
24
         addPrototype(this);
26
       // Nominal "state" per instance mechanism
      int id;
28
29
      static int count;
30
    // Register the subclass's prototype
    LandSatImage LandSatImage: landSatImage;
    // Initialize the "state" per instance mechanism
    int LandSatImage:: count = 1;
```

```
class Spotlmage: public Image
      public:
       imageType returnType()
         return SPOT;
06
       void draw()
         cout << "SpotImage::draw " << iid << endl;</pre>
09
10
       Image *clone()
11
          return new SpotImage(1);
12
13
      protected:
       SpotImage(int dummy) {
          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;
     int SpotImage:: count = 1;
```





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

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

