C++ 面向對象程序設計

(Object Oriented Programming, OOP)

侯捷

勿在浮沙築高台

1

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 **Object Based**
  - class without pointer members **(基於對象)**
    - Complex
  - class with pointer members
    - String

- 學習 Classes 之間的關係 **Object Oriented**
  - 繼承 (inheritance) **(面向對象)**
  - 複合 (composition)
  - 委託 (delegation)

complex.h
complex-test.cpp
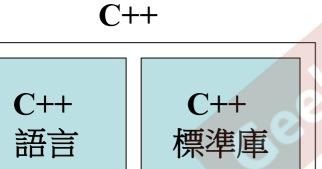
string.h
string-test.cpp
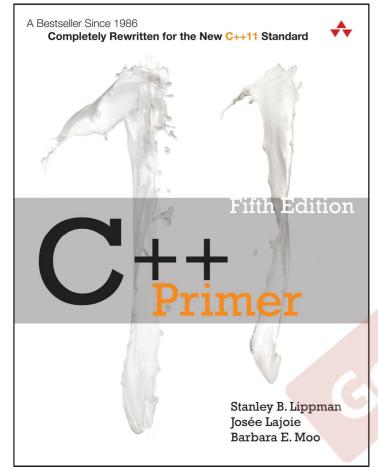
■■■ C++ 的歷史
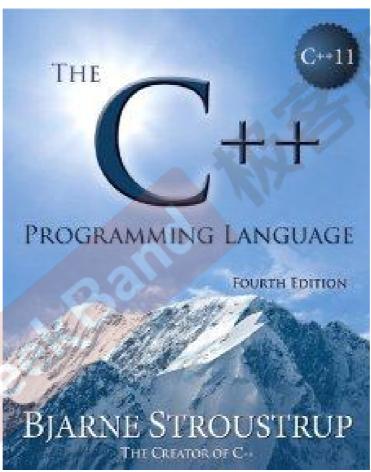
- **B 語言 (1969)**
- **C 語言 (1972)**
- **C++ 語言 (1983)**
  (new C ➔ C with Class ➔ C++)

- **Java 語言**
- **C# 語言**

C++ 演化

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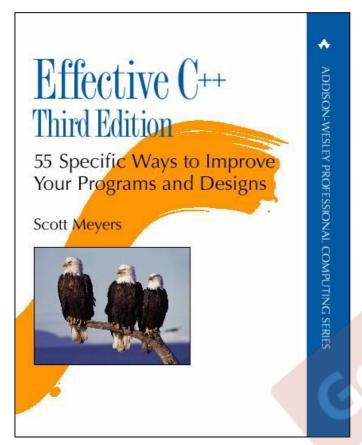- **C++ 03 (TR1, Technical Report 1)**
- **C++ 11 (2.0)**
- **C++ 14**

C++

C++ 語言	C++ 標準庫
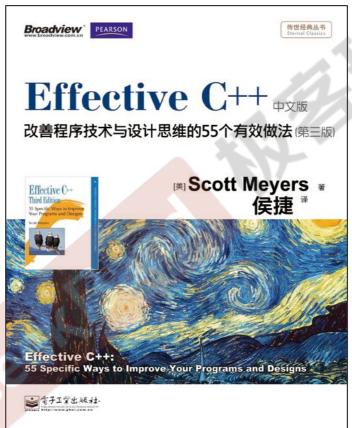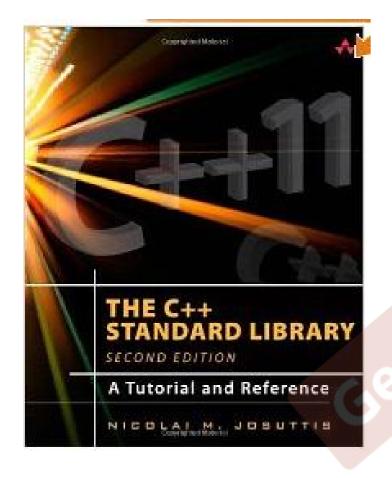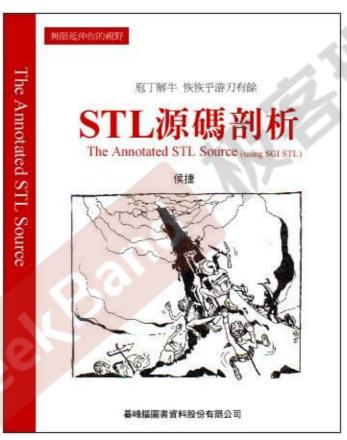
Bibliography (書目誌)

Bibliography (書目誌)

C

type
(built-in, struct)

Data

Functions

create ⟹

variables

...

C++

class, struct

Data
Members

Member
Functions

create ⟹

objects

...

complex

實部
虛部

加,減,乘,除,
共軛,正弦,
...

create →

c1
c2
c3
c4
...

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

string

字符 (s)
(其實是個 ptr,
指向一串字符)

拷貝,輸出,
附加,插入,
...

create →

s1
s2
s3
s4
...
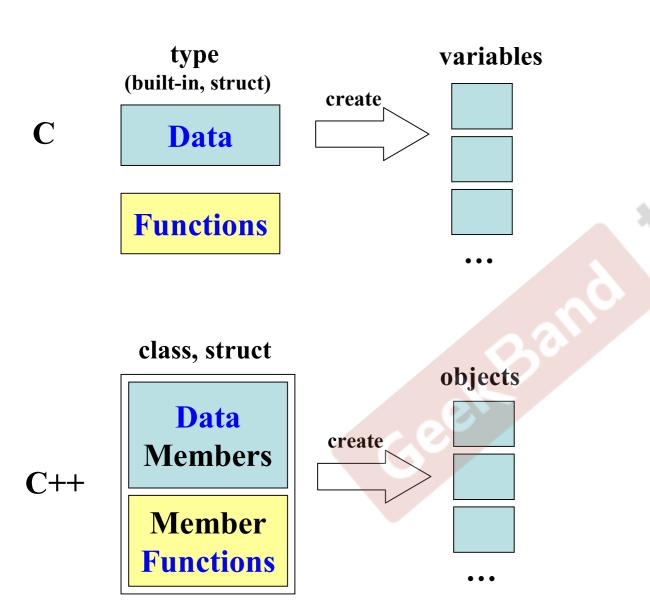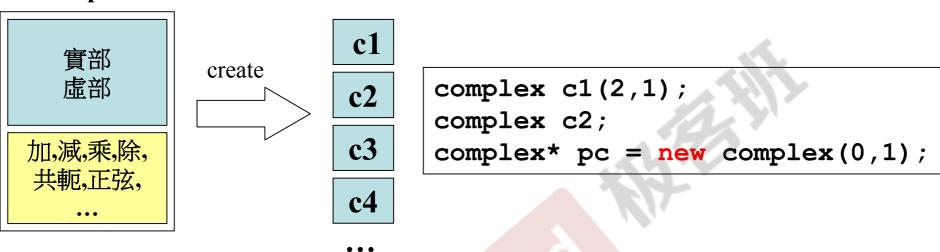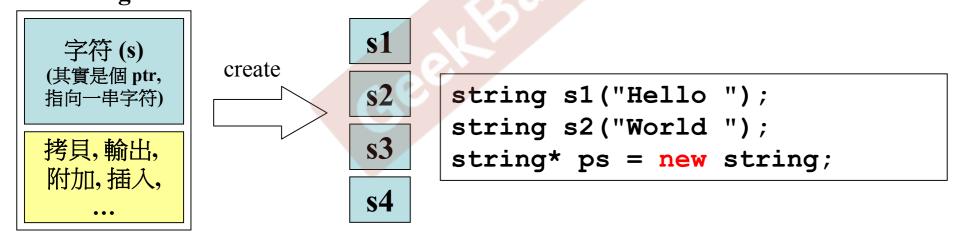
```
string s1("Hello ");
string s2("World ");
string* ps = new string;
```

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

Object Based : 面對的是單一 **class** 的設計

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

Classes 的兩個經典分類：

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

  complex
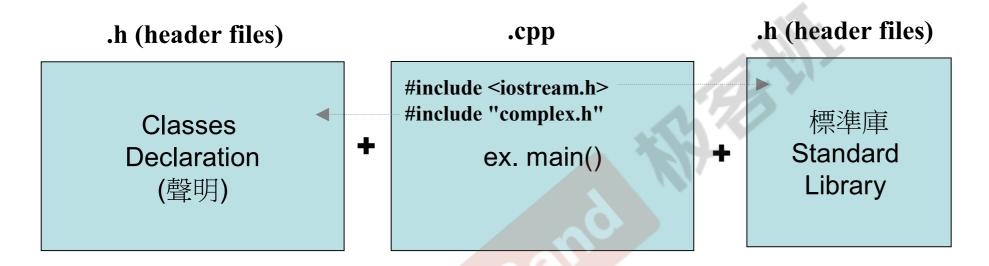
- **Class with pointer member(s)**

  string

C++ programs 代碼基本形式

.h (header files)   **.cpp**   **.h (header files)**

Classes Declaration (聲明)	**+**	#include <iostream.h> #include "complex.h" ex. main()	**+**	標準庫 Standard Library

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。

14

Output, C++ vs. C

C++

> #include <iostream>

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

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

  return 0;
}
```

C

> #include <cstdio>

```
#include <stdio.h>

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

  return 0;
}
```

Header (頭文件) 中的防衛式聲明

complex.h

```
#ifndef __COMPLEX__
#define __COMPLEX__


...


#endif
```

guard
(防衛式聲明)

```cpp
#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;
}
```

DEFINE

16

```
#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
                            (類 - 定義)

#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

```cpp
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&);
};
```

有些函數在此直接定義，
另一些在 body 之外定義

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

18

class template (模板) 簡介

①

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

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

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

19

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 ();
}
```

access level (訪問級別)

①

```cpp
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

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

O

```cpp
{
  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; }
```

default argument
(默認實參)

initialization list
(初值列, 初始列)

assignments
(賦值)

?!

①

```cpp
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&);
};
```

```cpp
{
  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)

①

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

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

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

②

```cpp
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

23

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

1

```cpp
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&);
};
```

✗

```cpp
{
  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;
}
```

private ,
  static

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

const member functions (常量成員函數)

1

```cpp
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&);
};
```

const.

this

const   this

const

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

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

real ()
const

c1   real
        const   this

26

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

by value   by reference

const

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

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

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

27

1

```cpp
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

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

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

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

friend (友元)

①

```cpp
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

```cpp
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

29

相同 class 的各個 objects 互為 friends (友元)

```cpp
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;
};
```

```cpp
{
  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

pass by ref:

return by ref

c1+c2

c1

return by ref

this

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

2-1

```cpp
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);
}
```

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

c2 += c1;
}
```

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


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

c3 += c2 += c1;

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

  c2 += c1;
}
```

33

class body 之外的各種定義 (definitions)

2-2

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


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

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

  cout << imag(c1);
  cout << real(c1);
}
```

34

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

```cpp
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));
}
```

```cpp
{
    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;
}
```

37

operator overloading (操作符重載), 非成員函數

2-5

```cpp
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;
}
```

```cpp
{
  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

```cpp
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;
}
```

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

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

39

operator overloading (操作符重載), 非成員函數

共軛複數

2-7

```cpp
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) << ')';
}
```

?!

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

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

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

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

編程示例

```cpp
#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
```

42

```cpp
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));
}
```

44

```
#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;
```

complex.h
complex-test.cpp

string.h
string-test.cpp

Classes 的兩個經典分類

• **Class without pointer member(s)**

   complex

• **Class with pointer member(s)**

   string

String class

string.h

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

①
```
class String
{
...
};
```

②
```
String::function(...) ...

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

```
#endif
```

string-test.cpp
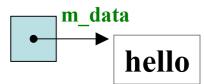
```
int main()
{
   String s1(),
   String s2("hello");

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

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

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;
};
```


m_data → hello
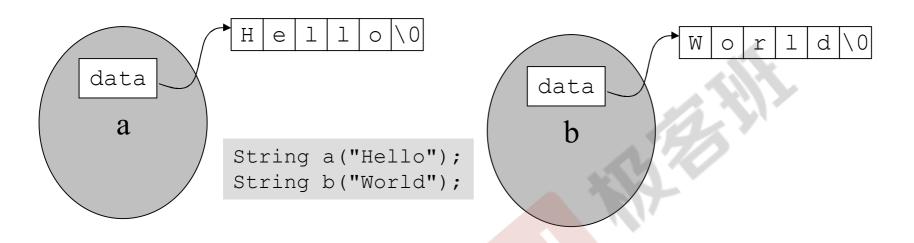
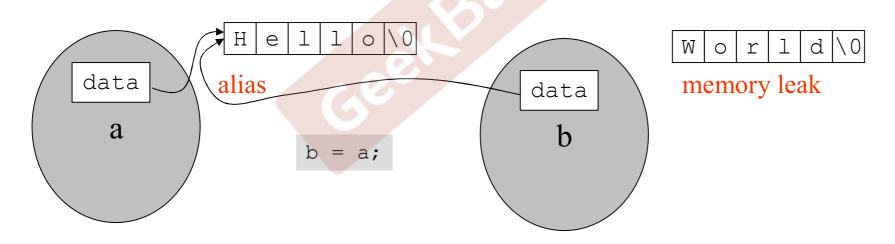# ctor 和 dtor (構造函數 和 析構函數)

2-1

```cpp
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

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



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

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

```
H e l l o \0
```

```
data
```
a

```
W o r l d \0
```

```
data
```
b

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

使用 default copy ctor 或 default op= 就會形成以下局面
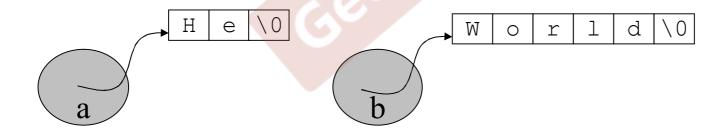
```
H e l l o \0
```

```
data
```
a

alias

```
data
```
b

```
W o r l d \0
```

memory leak

```
b = a;
```

copy ctor (拷貝構造函數)

2-2

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

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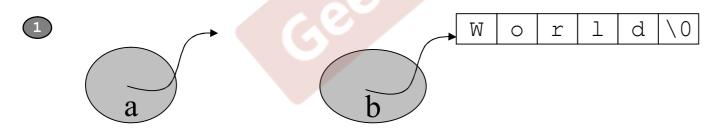```

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

# copy assignment operator (拷貝賦值函數)

**2-3**

```cpp
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;

}
```

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

}
```

H	e	\0

W	o	r	l	d	\0

a

b

copy assignment operator (拷貝賦值函數)

2-3

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


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

}
```
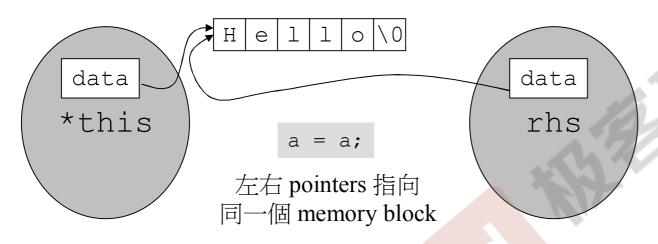
1

| W | o | r | l | d | \0 |

a   b

copy assignment operator (拷貝賦值函數)

2-3

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

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

②

| | | | | | → | W | o | r | l | d | \0 |

a   b

copy assignment operator (拷貝賦值函數)

2-3

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

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

③

| W | o | r | l | d | \0 |

| W | o | r | l | d | \0 |

a

b
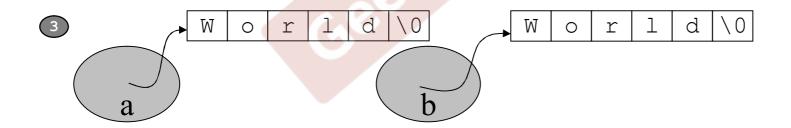
■■■■ 一定要在 **operator=** 中檢查是否 **self assignment**


```
H e l l o \0
```

`a = a;`

左右 pointers 指向
同一個 memory block

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


```
???
```

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

output 函數

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

```cpp
{
    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。

59

■■■■ 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;
}
```

```
class Complex { … };
...

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

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

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

new：先分配 memory, 再調用 ctor

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

❶ pc → double / double 設初值 ❸

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

編譯器轉化為

其內部調用 malloc(*n*)

```
Complex *pc;

❶ void* mem = operator new( sizeof(Complex) ); //分配內存
❷ pc = static_cast<Complex*>(mem);             //轉型
❸ 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;
```

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

double
double

pc

清理

編譯器轉化為

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

其內部調用 free(pc)

new：先分配 memory，再調用 ctor

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

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

編譯器轉化為

其內部調用 malloc(n)

```
String* ps;

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

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

this

66

delete：先調用 dtor, 再釋放 memory

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

編譯器轉化為

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

其內部調用 free(ps)

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

ps

m_data

Hello

00000041
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0xfd
Complex object (8h)
4 個 0xfd
00000000 (pad)
00000000 (pad)
00000000 (pad)
00000041

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

➔52

➔64

00000011
Complex object (8h)
00000011

8+(4*2)

➔16

00000031
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0xfd
String object (4h)
4 個 0xfd
00000031

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

➔48

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

4+(4*2)

➔12

➔16

68
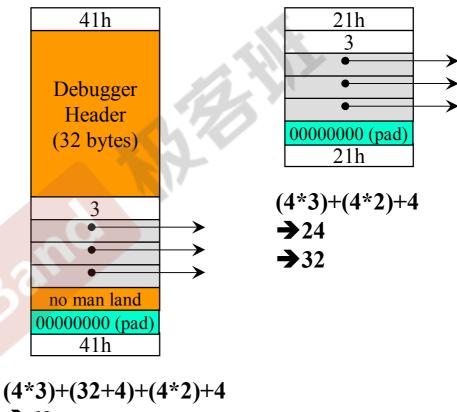
動態分配所得的 array

Complex* p = new Complex[3];

51h
Debugger Header (32 bytes)
3
double
double
double
double
double
double
no man land
00000000 (pad)
00000000 (pad)
51h

(8*3)+(32+4)+(4*2)+4
➜**72**
➜**80**

31h
3
double
double
double
double
double
double
00000000 (pad)
00000000 (pad)
00000000 (pad)
31h

(8*3)+(4*2)+4
➜**36**
➜**48**

String* p = new String[3];

41h
Debugger Header (32 bytes)
3
no man land
00000000 (pad)
41h

(4*3)+(32+4)+(4*2)+4
➜**60**
➜**64**

21h
3
00000000 (pad)
21h

(4*3)+(4*2)+4
➜**24**
➜**32**
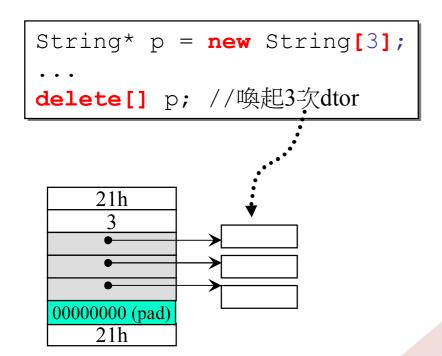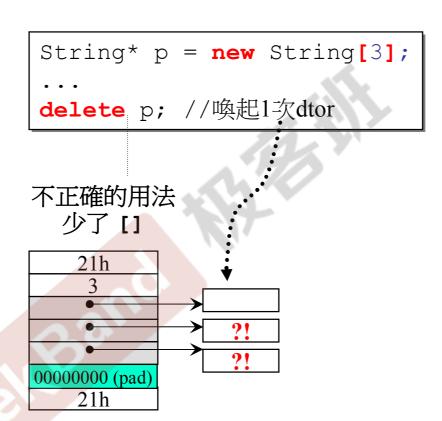
array new 一定要搭配 array delete

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

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

不正確的用法
少了 **[ ]**

編程示例

```cpp
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;
};
```

```cpp
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;
}
```

73

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

```cpp
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;
}
```

complex.h
complex-test.cpp

string.h
string-test.cpp

進一步補充：**static**

complex
data members **static** data members
member functions **static** member functions

static
data members

this

non-static
member functions

static
member functions

c1

non-static
data members

c2

non-static
data members

c3

non-static
data members

...

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

→

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

this

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

77

進一步補充：**static**

```cpp
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

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


A& A::getInstance()
{
  static A a;
  return a;
}
```

```cpp
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();`

80

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

extern _IO_ostream_withassign cout;
```

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

進一步補充：class template, 類模板

```cpp
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&);
};
```

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

82

進一步補充：**function template,** 函數模板

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

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

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

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

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

進一步補充：**namespace**

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

using directive

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

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

  return 0;
}
```

using declaration

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

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

  return 0;
}
```

```
#include <iostream.h>


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

  return 0;
}
```

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

革命尚未成功

同志仍需努力

Object Oriented Programming, Object Oriented Design
OOP, OOD

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

Composition (複合), 表示 has-a
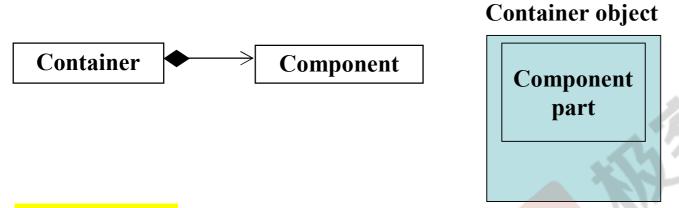
```
template <class T, class Sequence = deque<T> >
class queue {
  ...
protected:
  Sequence c;      // 底層容器
public:
  // 以下完全利用 c 的操作函數完成
  bool empty() const { return c.empty(); }
  size_type size() const { return c.size(); }
  reference front() { return c.front(); }
  reference back() { return c.back(); }
  // 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(); }
};
```

queue ◆———————▷ deque

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

Container ▸────▸ **Component**

Container object

Component part

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。
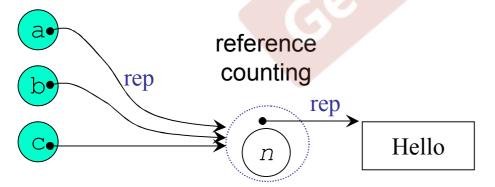
```
Container::Container(…): Component() { … };
```
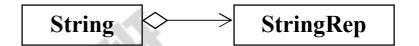
析構由外而內

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.

```cpp
// 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
};
```
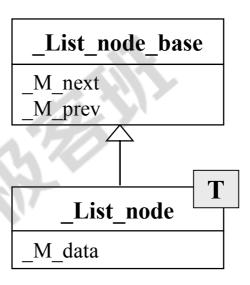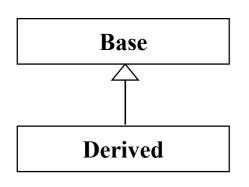
```
String  ◇──────▷  StringRep
```

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

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

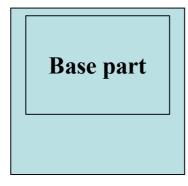reference counting

a
b
rep
c
rep
n
Hello

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;
};
```

```
┌─────────────────────┐
│  _List_node_base    │
├─────────────────────┤
│ _M_next             │
│ _M_prev             │
└─────────────────────┘
          △
          │
     ┌────┴────┐  ┌───┐
     │         │  │ T │
┌────┴─────────────┐
│   _List_node     │
├──────────────────┤
│ _M_data          │
└──────────────────┘
```

Derived object

```
┌─────────┐        ┌───────────────────┐
│  Base   │        │  ┌─────────────┐  │
└─────────┘        │  │             │  │
     △             │  │  Base part  │  │
     │             │  │             │  │
     │             │  └─────────────┘  │
┌─────────┐        └───────────────────┘
│ Derived │
└─────────┘
```

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

构造由內而外

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)

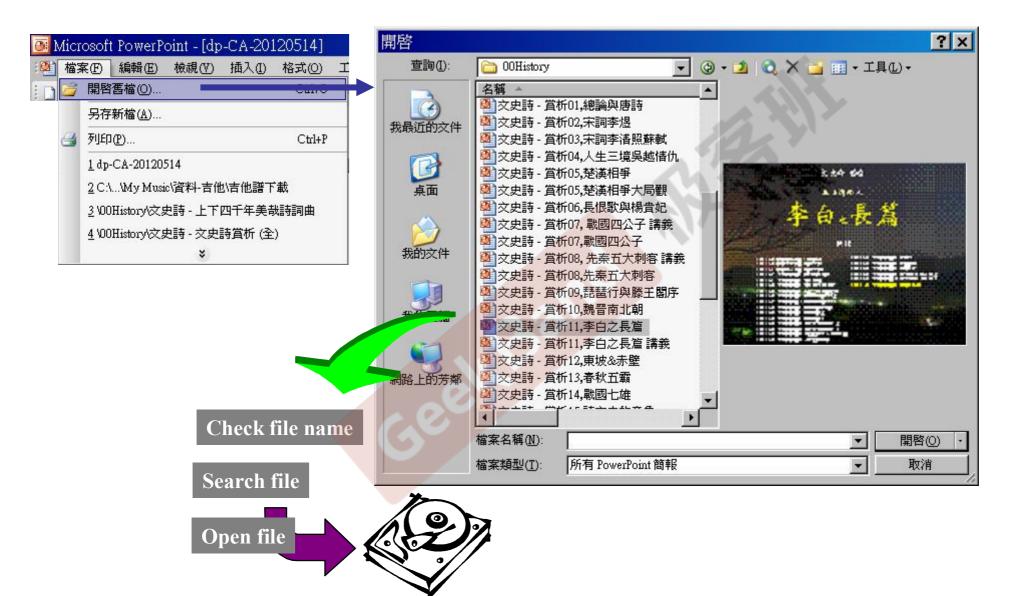**non-virtual** 函數：你不希望 **derived class** 重新定義 (override, 覆寫) 它.

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 { ... };
```
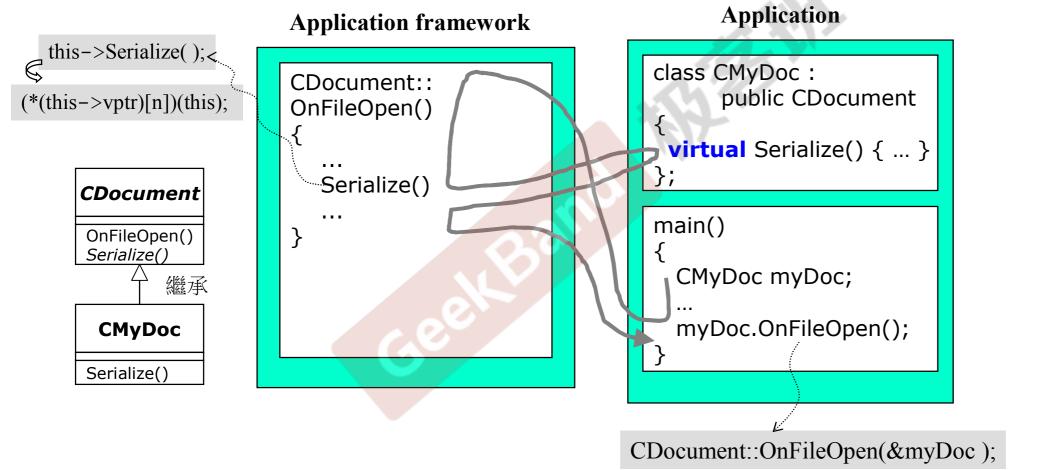
pure virtual

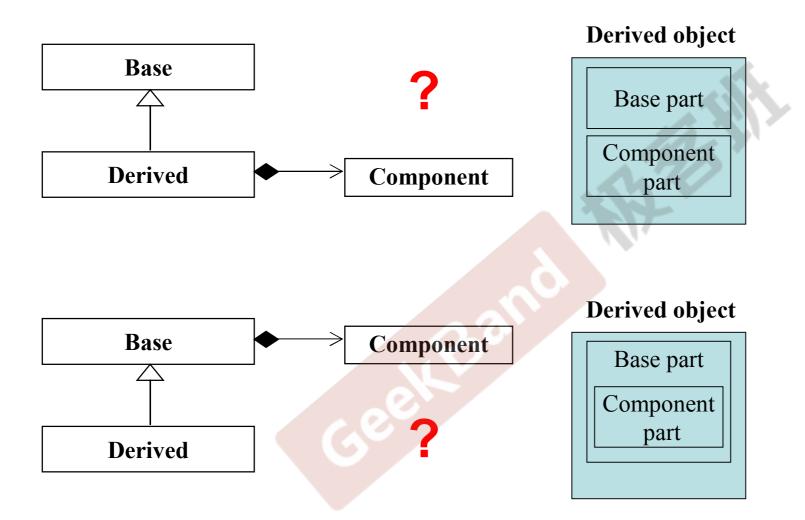impure virtual

non-virtual

Inheritance (繼承) with virtual

Check file name

Search file

Open file

Inheritance (繼承) with virtual

Template Method

Application framework

Application

this->Serialize( );

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

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

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

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

CDocument

OnFileOpen()
Serialize()

繼承

CMyDoc

Serialize()

CDocument::OnFileOpen(&myDoc);

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

Base ← **Derived** ◆→ **Component**

?

Derived object
- Base part
- Component part

Base ◆→ **Component** ← **Derived**

?

Derived object
- Base part
  - Component part

Inheritance+Composition 關係下的構造和析構

Base

Derived ◆───▶ **Component**

Derived object

Base part

Component part

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
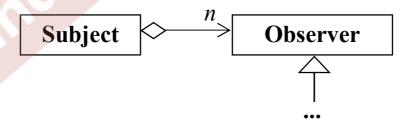
Derived 的析構函數首先執行自己，
然後調用 **Component** 的 析構函數，
然後調用 **Base** 的析構函數。
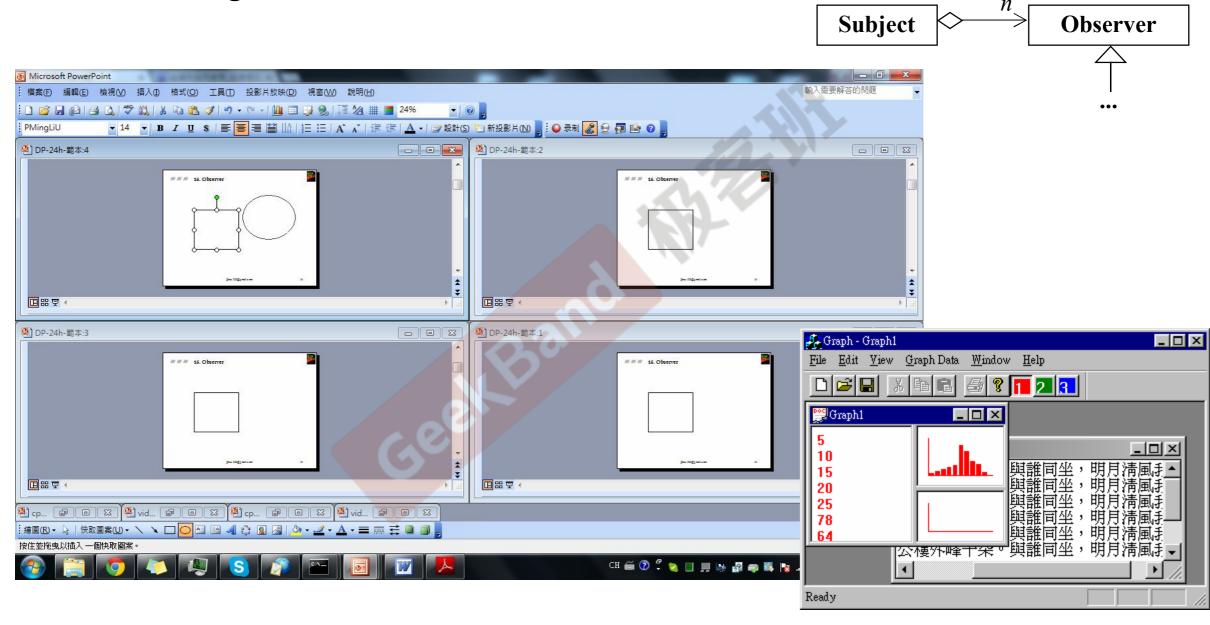
```
Derived::~Derived(…){ … ~Component(), ~Base() };
```

Delegation (委託) + Inheritance (繼承)

```cpp
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);
  }
};
```
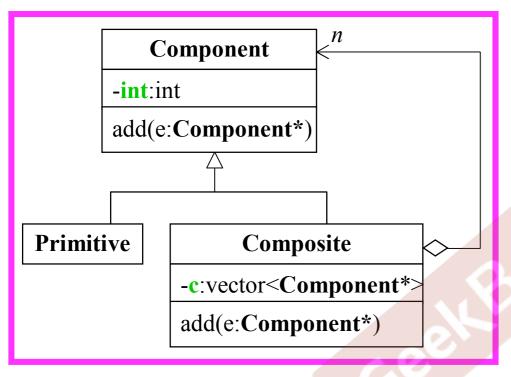
Observer

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


Subject ◇—————n————▷ Observer
                                        △
                                        ┆
                                       ...

Delegation (委託) + Inheritance (繼承)

Delegation (委託) + Inheritance (繼承)

```cpp
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);
  }
};
```

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

n

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

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

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

102

Composite

```
          n
┌──────────────────┐◄─────────┐
│    Component     │          │
├──────────────────┤          │
│ -int:int         │          │
├──────────────────┤          │
│ add(e:Component*)│          │
└──────────────────┘          │
          △                   │
     ┌────┴──────┐            │
┌─────────┐ ┌──────────────────┐
│Primitive│ │    Composite     │◇─┘
└─────────┘ ├──────────────────┤
            │ -c:vector<Component*>│
            ├──────────────────┤
            │ add(e:Component*)│
            └──────────────────┘
```

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

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

```cpp
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

abstraction

Image
prototypes[10]: Image*
clone() : Image* findAndClone (i) : Image* addPrototype (p:Image*) : void

return prototypes[i]->clone();
virtual ctor

LandSatImage
LSAT : LandSatImage
−LandSatImage() #LandSatImage(int) clone() : Image*

SpotImage
SPOT : SpotImage
−SpotImage() #SpotImage(int) clone() : Image*

return new LandSatImage;

return new SpotImage;

addPrototype (this);

addPrototype (this);

```
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();
}
```

105

Prototype

```cpp
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;
```

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

```cpp
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;
```

106

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  }
```

The End