

GeekBand 极客班

互联网人才 + 油站!

C++ 面向对象高级编程

www.geekband.com

GeekBand 极客班 互联网人才+油站：

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

专业课程

+ 项目碾压

+ 习题&辅导

- 顶尖大牛亲授
- 紧贴课程内容
- 贴合企业实际需求
- 全程实战操练
- 找对重点深挖学习
- 作品就是最好的PASS卡

- 学前导读
- 定期作业点评
- 多项专题辅导



C++ 程序設計 (II)

兼談對象模型

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

侯捷



勿在浮沙築高台

GeekBand

你應具備的基礎

- 本課程是先前我的另一門視頻課程“面向對象程序設計”的續集
- 本課程將探討上述課程未及討論的主題

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



我們的目標

- 在先前基礎課程所培養的正規、大器的編程素養上，繼續探討更多技術。
- 泛型編程 (**Generic Programming**) 和面向對象編程 (**Object-Oriented Programming**) 雖然分屬不同思維，但它們正是 C++ 的技術主線，所以本課程也討論 **template** (模板)。
- 深入探索面向對象之繼承關係 (**inheritance**) 所形成的對象模型 (**Object Model**)，包括隱藏於底層的 **this** 指針, **vptr** (虛指針), **vtbl** (虛表), **virtual mechanism** (虛機制)，以及虛函數 (**virtual functions**) 造成的 **polymorphism** (多態) 效果。



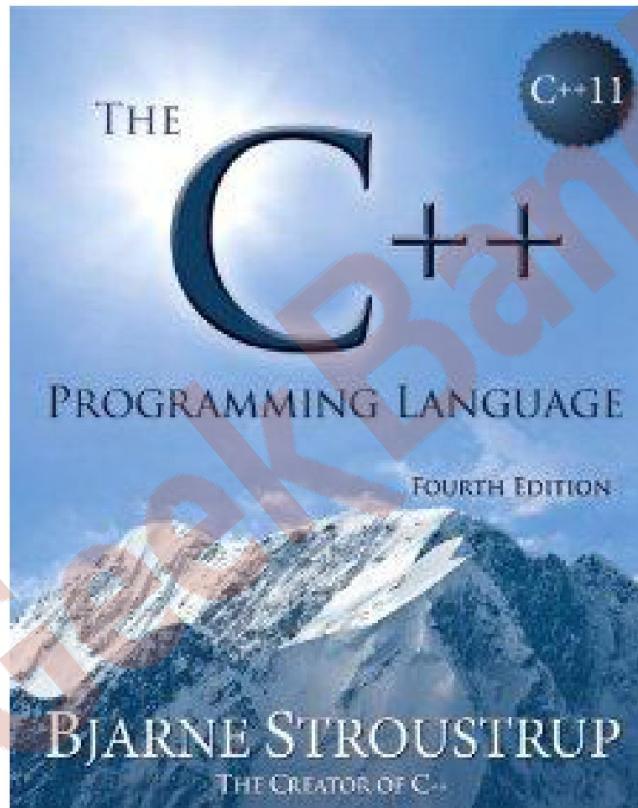
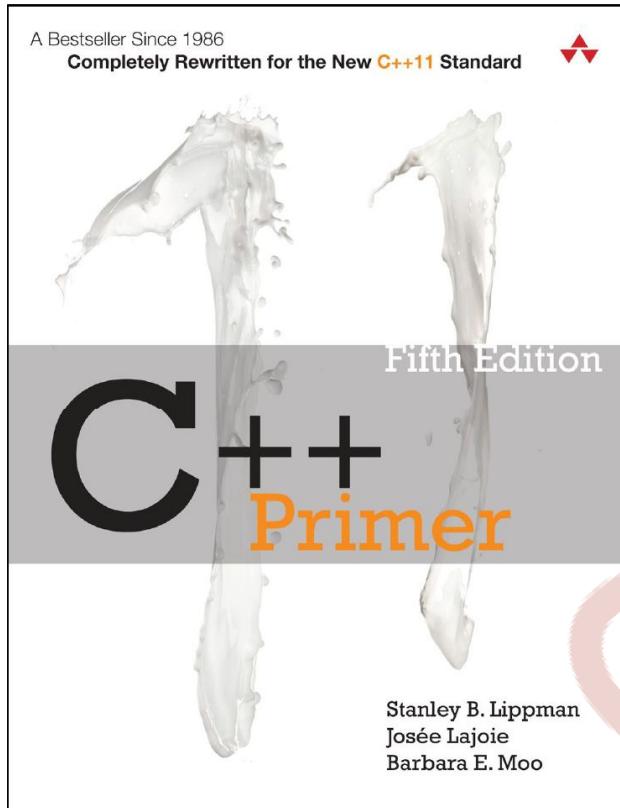
你將獲得的代碼

Test-Cpp.cpp

GeekBand

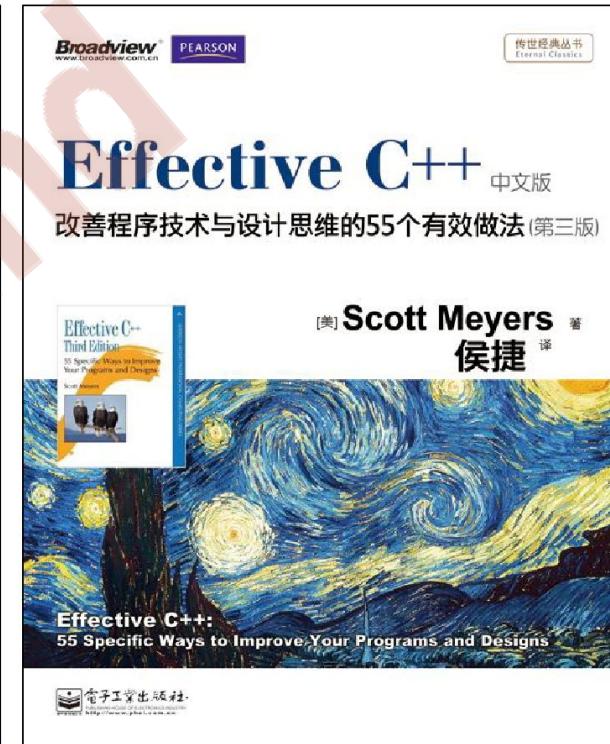
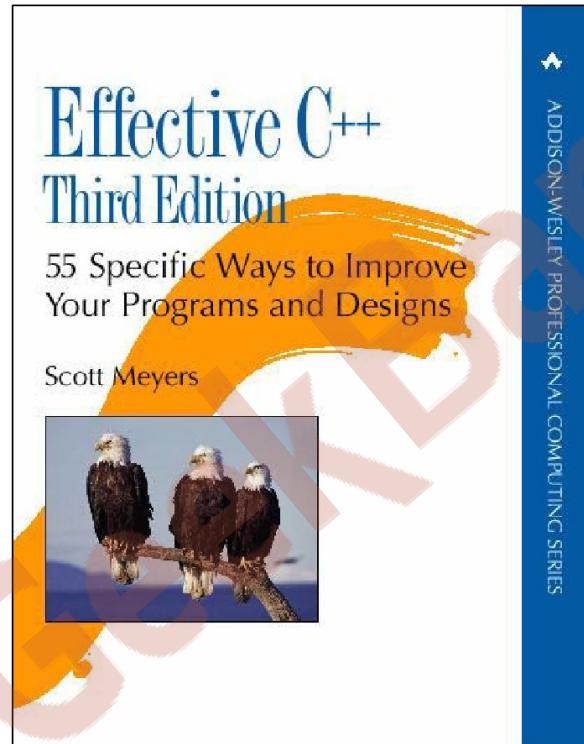
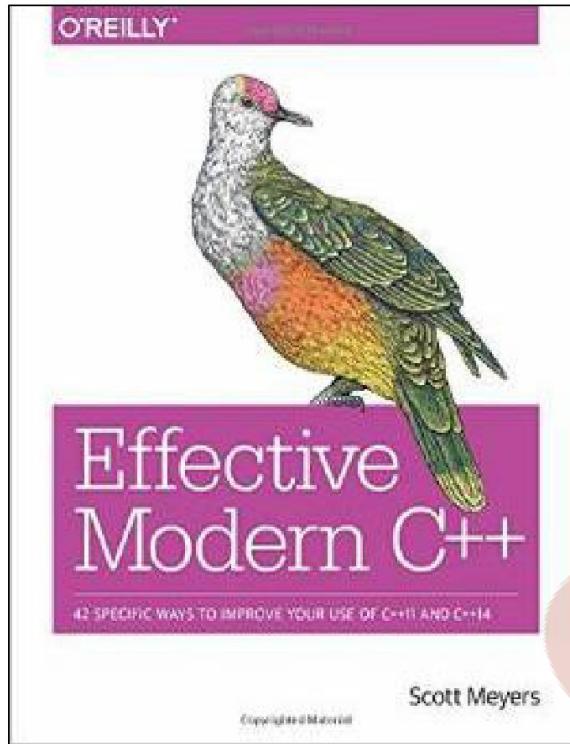


Bibliography (書目誌)



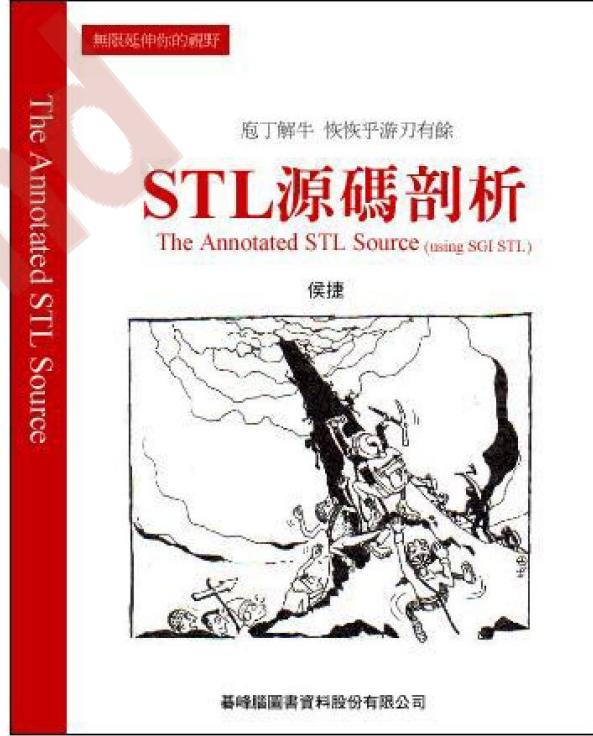
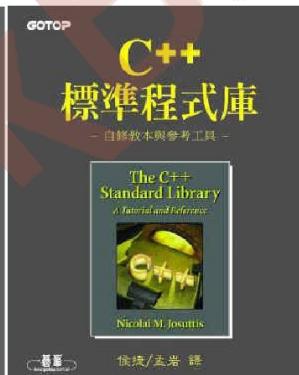
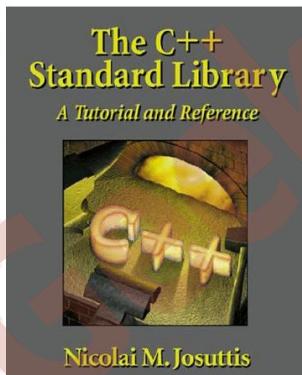
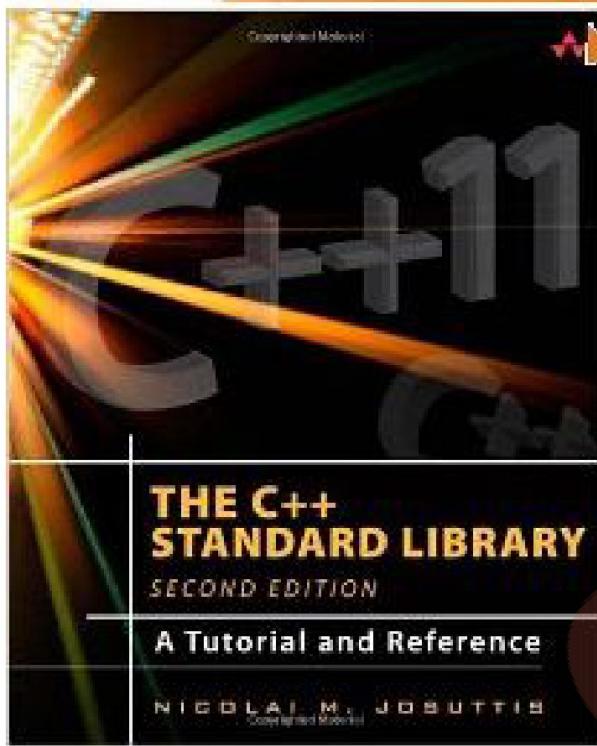


Bibliography (書目誌)





Bibliography (書目誌)





C++ 編譯器

(本圖為 Visual C++ 6.0 畫面)

- 編譯 (compile)
- 連結 (link)

```
// <Beyond the C++ Standard Library, An Introduction to Boost>, Chap1

#include <string>
#include <iostream>
#include "boost\scoped_ptr.hpp"
#include "boost\shared_ptr.hpp"

using namespace std;
using namespace boost;

//p12
class pimpl_sample {
private:
    struct impl {
        void func() { cout << s << endl; }
        string s;
    };
    impl* pimpl;
public:
```

--Configuration: BoostSmartPtr - Win32 Debug--

BoostSmartPtr.exe - 0 error(s), 0 warning(s)

Ready



C++ 編譯器

(本圖為 Dev-C++ 5.6 畫面)

- 編譯 (compile)
- 連結 (link)

The screenshot shows the Dev-C++ 5.6 IDE interface. The title bar reads "Test-OOP2 - [Test-OOP2.dev] - Dev-C++ 5.11". The menu bar includes File, Edit, Search, View, Project, Execute, Tools, AStyle, Window, Help. The toolbar has various icons for file operations. The left sidebar shows the project structure under "Test-OOP2": "test-oop2.cpp". The main code editor window shows the following C++11 code:

```
134     cout << &(X::func) << endl;    //1
135     cout << jj04::gfunc << endl;
136     cout << nsfunc << endl;
137 }
138 } //namespace
139
140 void gfunc() { }
141
142 /**
143 int main(int argc, char** argv)
144 {
145     jj01::test_member_template();
146     jj02::test_template_template_parameters_1();
147     jj03::test_template_template_parameters_2();
148     jj04::test_object_model();
149
150     cout << main << endl;
151     cout << gfunc << endl;
152 }
```

The bottom panel contains tabs for Compiler, Resources, Compile Log, Debug, Find Results, and Close. The Compiler tab is active, showing the command line:

```
g++.exe -D__DEBUG__ -c test-oop2.cpp -o test-oop2.o -I"C:/Program Files/Dev-Cpp/M
g++.exe -D__DEBUG__ test-oop2.o -o Test-OOP2.exe -L"C:/Program Files/Dev-Cpp/MinG
```

The Compile Log tab shows the compilation results:

```
Compilation results...
-----
- Errors: 0
- Warnings: 0
- Output Filename: D:\handout\C++11-test-DevC++\Test-OOP2\Test-OOP2.exe
```

- **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++)
- Object Model

革命尚未成功
同志仍需努力

conversion function, 轉換函數

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

Fraction f(3,5);

double d=4+f; //調用operator double()將 f 轉為 0.6

non-explicit-one-argument ctor

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

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

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

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

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

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

explicit-one-argument ctor

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

```
Fraction f(3,5);
```

```
Fraction d2=f+4; // [Error] conversion from 'double' to 'Fraction' requested
```

conversion function, 轉換函數

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

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

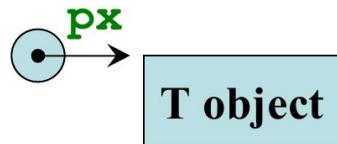
pointer-like classes, 關於智能指針

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

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

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

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



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

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

Foo f (*sp);

sp->method();
```

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

pointer-like classes, 關於迭代器

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

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

pointer-like classes, 關於迭代器

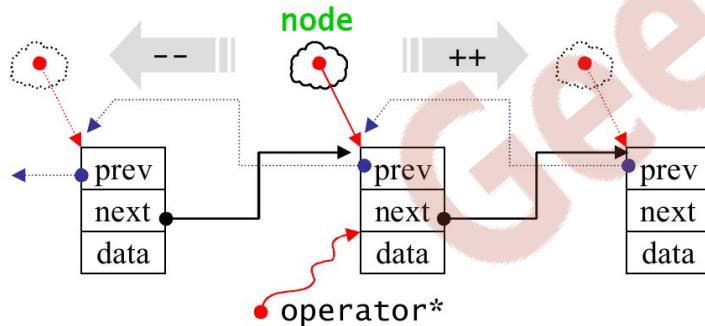
T&

```
reference operator*() const  
{ return (*node).data; }
```

T*

```
pointer operator->() const  
{ return &(operator*()); }
```

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



function-like classes, 所謂仿函數

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

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

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

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

標準庫中的仿函數的奇特模樣

```
template <class T>
struct identity {  
    const T&  
    operator()(const T& x) const { return x; }  
};  
  
template <class Pair>
struct select1st {  
    const typename Pair::first_type&  
    operator()(const Pair& x) const  
    { return x.first; }  
};  
  
template <class Pair>
struct select2nd {  
    const typename Pair::second_type&  
    operator()(const Pair& x) const  
    { return x.second; }  
};
```

標準庫中的仿函數的奇特模樣

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

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

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

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

標準庫中，仿函數所使用的奇特的 **base classes**

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

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

less<int>::result_type → bool

namespace 經驗談

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

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

class template, 類模板

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

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

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

function template, 函數模板

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

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

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

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

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

member template, 成員模板

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

    T1 first;
    T2 second;

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

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

};
```

member template, 成員模板

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

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



T1 T2

```
pair<Derived1, Derived2> p;
```

pair<Base1, Base2> p2(p);

T1 T2 U1 U2

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

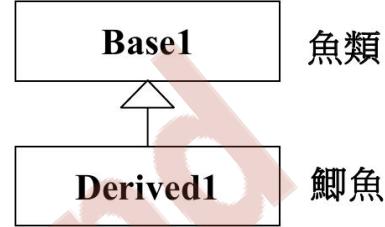
把一個由鯽魚和麻雀構成的 pair，放進 (拷貝到) 一個由
魚類和鳥類構成的 pair 中，可以嗎？
反之，可以嗎？

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

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

member template, 成員模板

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



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

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

specialization, 模板特化

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

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

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

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

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

partial specialization, 模板偏特化 -- 個數的偏

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

綁定

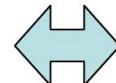
```
template<typename Alloc=.....>
class vector<bool, Alloc>
{
    ...
};
```

partial specialization, 模板偏特化 -- 範圍的偏

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

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

這樣寫
也可以



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

```
C<string> obj1;
```

```
C<string*> obj2;
```

template template parameter, 模板模板參數

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

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

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

template template parameter, 模板模板參數

```
template<typename T,
         template <typename T>
         class SmartPtr
     >

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

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

/// 這不是 template template parameter

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

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

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

關於 C++ 標準庫

Sequence containers:

- array C++
- vector
- deque
- forward_list C++
- list

Container adaptors:

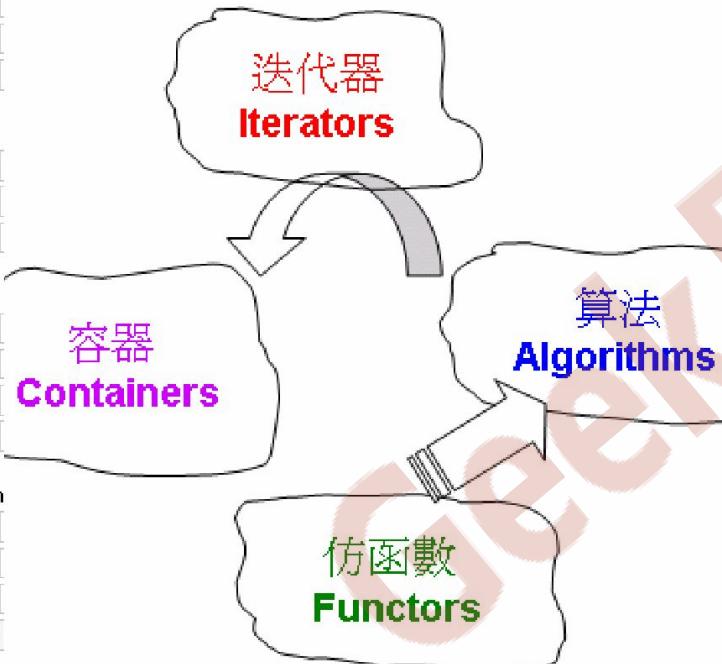
- stack
- queue
- priority_queue

Associative containers:

- set
- multiset
- map
- multimap

Unordered associative con

- unordered_set C++
- unordered_multiset C++
- unordered_map C++
- unordered_multimap C++



...

Sorting:

- sort
- stable_sort
- partial_sort
- partial_sort_copy
- is_sorted C++
- is_sorted_until C++
- nth_element

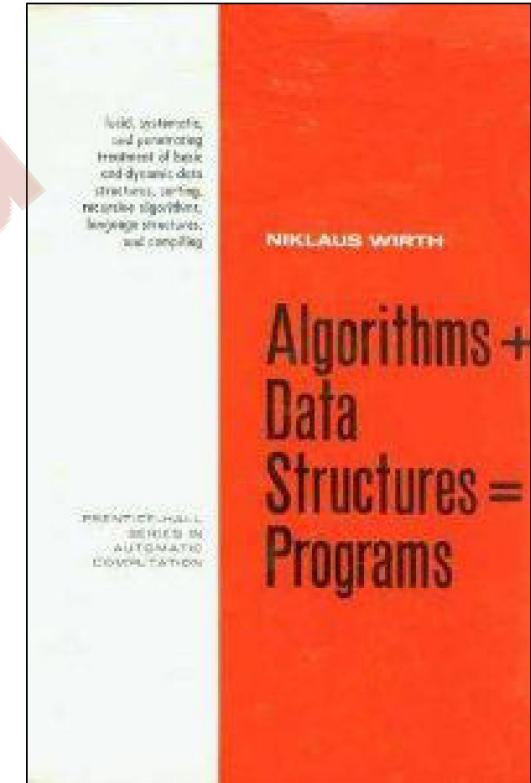
Binary search (operating on)

- lower_bound
- upper_bound
- equal_range
- binary_search

Merge (operating on sorted)

- merge
- inplace_merge
- includes
- set_union
- set_intersection
- set_difference
- set_symmetric_difference

...



1976 Niklaus Wirth



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++)
- Object Model

革命尚未成功
同志仍需努力

11



了解你的編譯器對 C++2.0 的支持度

→ C cpprocks.com/c11-compiler-support-shootout-visual-studio-gcc-clang-intel/

C++11 compiler support shootout: Visual Studio, GCC, Clang, Intel

March 14, 2013 | Tags: [C++11](#), [clang](#), [g++](#), [Intel C++](#), [vs2012](#)

Category: [Uncategorized](#) | 9 Comments

It's been more than half a year since my [last comparison](#) of the C++11 support across different compilers. This time I'd like to see how different compilers stack up based on the documentation for the pre-release versions of these compilers.

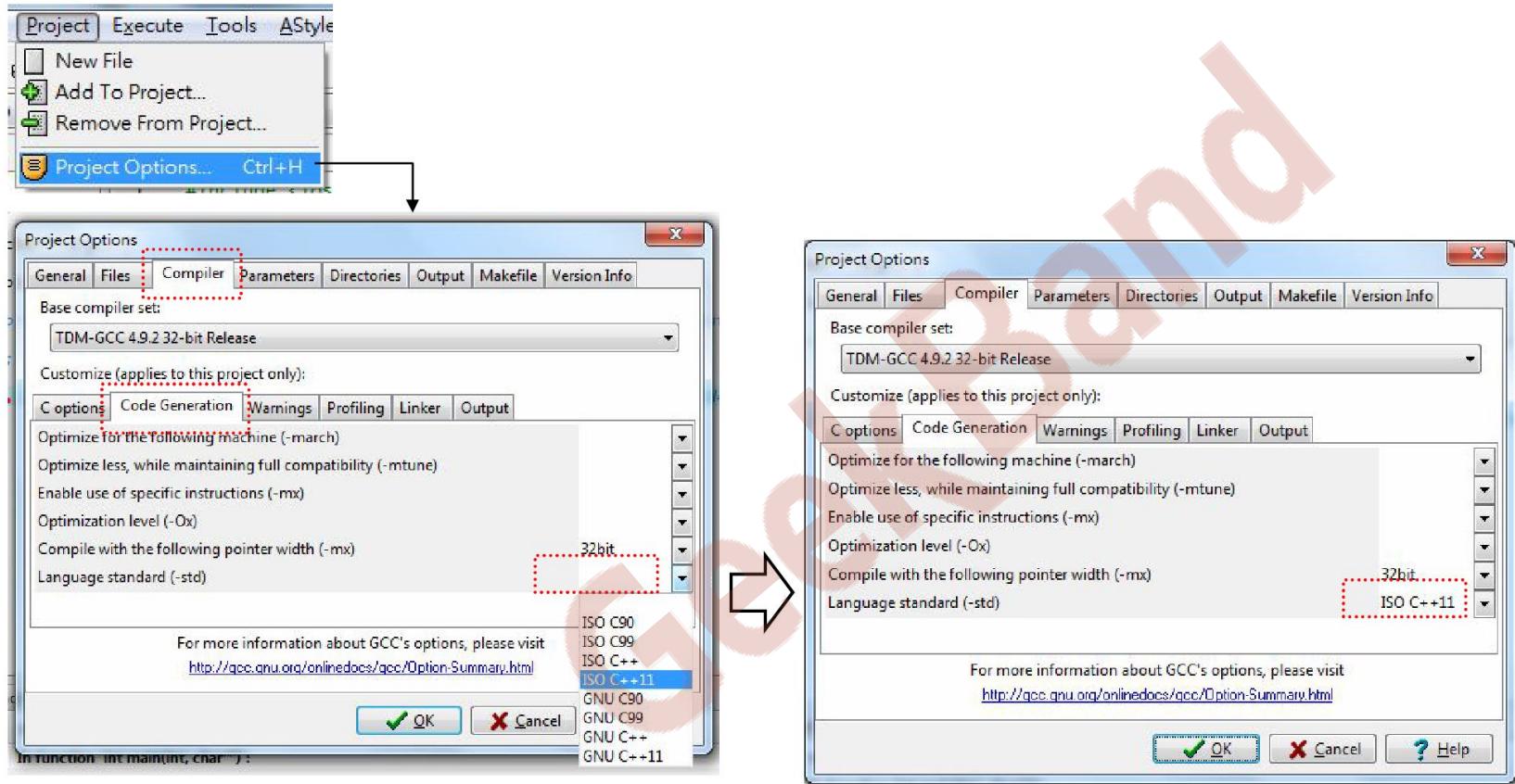
The next release of GCC is 4.8 and the upcoming version of Clang is 3.3. If you use Visual Studio 2012, you can install an experimental CTP update released in November 2012 to get additional C++11 features.

I've also thrown in v.13.0 of Intel's C++ compiler out of curiosity, although it isn't pre-release and there were a few features I couldn't find information about. I didn't find any information about the upcoming version of this compiler.

Feature	VS2012	g++ 4.8	Clang 3.3	Intel 13.0
	Nov CTP			
auto	Yes	Yes	Yes	Yes
decltype	Yes	Yes	Yes	Yes
Rvalue references and move semantics	Yes	Yes	Yes	Yes
Lambda expressions	Yes	Yes	Yes	Yes
nullptr	Yes	Yes	Yes	Yes
static_assert	Yes	Yes	Yes	Yes
Range based for loop	Yes	Yes	Yes	Yes
Trailing return type in functions	Yes	Yes	Yes	Yes
extern templates	Yes	Yes	Yes	Yes
>> for nested templates	Yes	Yes	Yes	Yes
Local and unnamed types as template arguments	Yes	Yes	Yes	Yes



DevC++ 上的 ISO C++11 開關



第一步, 確認支持 C++11 : macro _cplusplus

If available, you can evaluate the predefined macro _cplusplus. For C++11, the following definition holds when compiling a C++ translation unit:

```
#define _cplusplus 201103L
```

By contrast, with both C++98 and C++03, it was:

```
#define _cplusplus 199711L
```

Note, however, that compiler vendors sometimes provide different values here.

VS 2012

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

int main()
{
    cout << _cplusplus << endl;
    return 0;
}
```

C:\Windows\system32\cmd...
199711
Press any key to continue . . .

Dev-C++ 5

```
#include <iostream>
int main()
{
    std::cout << _cplusplus;
}
```

32bit.....
ISO C++11

D:\temp\Untitled3.exe
199711
Process exited after 0.06942 second
Press any key to continue . . .

D:\handout\C++11-test-DevC++\C++1
201103
Process exited after 0.01544 se
Press any key to continue . . .

variadic templates (since C++11)

數量不定的模板參數

1 → void **print** ()
{
}

print(7.5, "hello", bitset<16>(377), 42);

7.5
hello
0000000101111001
42

2 → template <typename T, typename... Types>
void **print** (const T& **firstArg**, const Types&... **args**)
{
 cout << **firstArg** << endl;
 print(**args...**);
}

Inside variadic templates,
sizeof...(args) yields the
number of arguments

... 就是一個所謂的 **pack** (包)

用於 template parameters, 就是 template parameters **pack** (模板參數包)

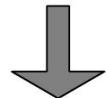
用於 function parameter types, 就是 function parameter types **pack** (函數參數類型包)

用於 function parameters, 就是 function parameters **pack** (函數參數包)



auto (since C++11)

```
list<string> c;  
...  
list<string>::iterator ite;  
ite = find(c.begin(),c.end(),target);
```



```
list<string> c;  
...  
auto ite = find(c.begin(),c.end(),target);
```

```
list<string> c;  
...  
x auto ite;  
ite = find(c.begin(),c.end(),target);
```

//// ranged-base **for** (since C++11)

```
for ( decl : coll ) {  
    statement  
}
```

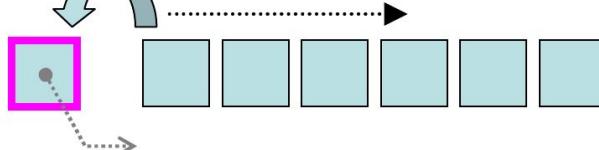
```
for (int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {  
    cout << i << endl;  
}
```

```
vector<double> vec;  
...  
for ( auto elem : vec ) {  
    cout << elem << endl;  
}  
  
for ( auto& elem : vec ) {  
    elem *= 3;  
}
```

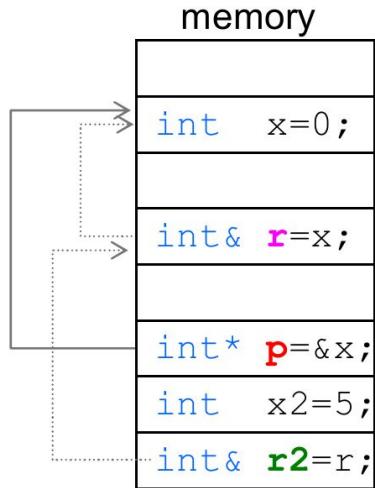
pass by value



pass by reference



reference



注意：1, sizeof(r) == sizeof(x)

2, &x == &r;

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

r is a *reference to x*

p is a *pointer to x*

Java 裡頭所有變量都是 reference

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

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

reference

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

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

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

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

reference 的常見用途

```
void func1(Cls* pobj) { pobj->xxx(); }
void func2(Cls obj) { obj.xxx(); }
void func3(Cls& obj) { obj.xxx(); }
.....
被調用端寫法相同, 很好
Cls obj;
func1(&obj); —— 接口不同, 困擾
func2(obj); > 調用端接口相同, 很好
func3(obj);
```

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

以下被視為“same signature”（所以二者不能同時存在）：

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

signature

Q: **const** 是不是函數簽名的一部分？
A: Yes

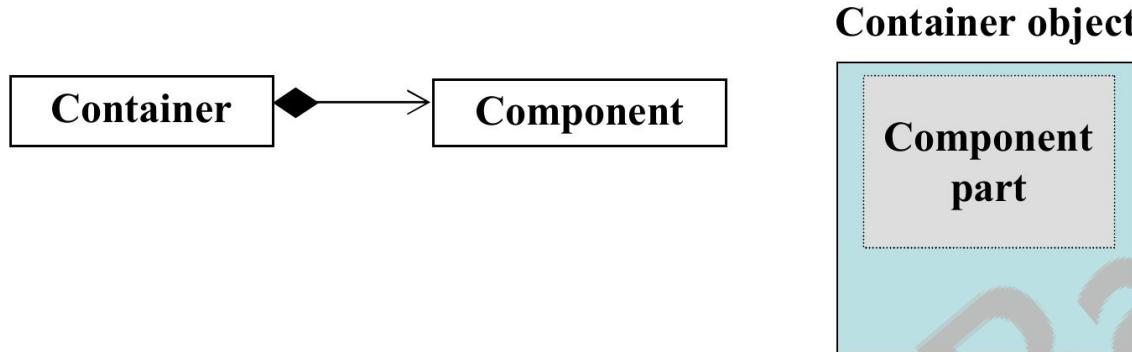


更多細節與深入

- 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++)
 - Object Model

革命尚未成功
同志仍需努力

///// 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。

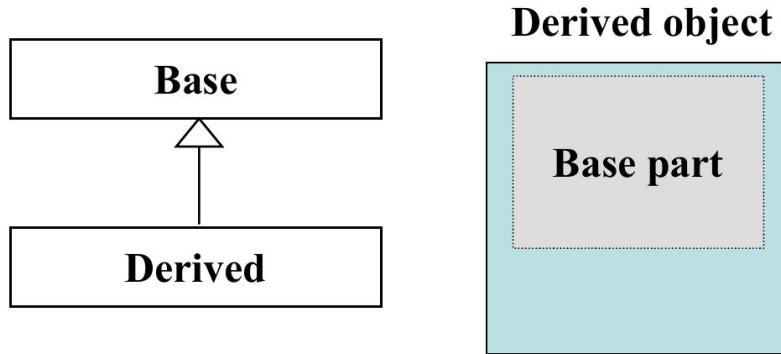
```
Container::Container(...) : Component() { ... };
```

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() };
```

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。

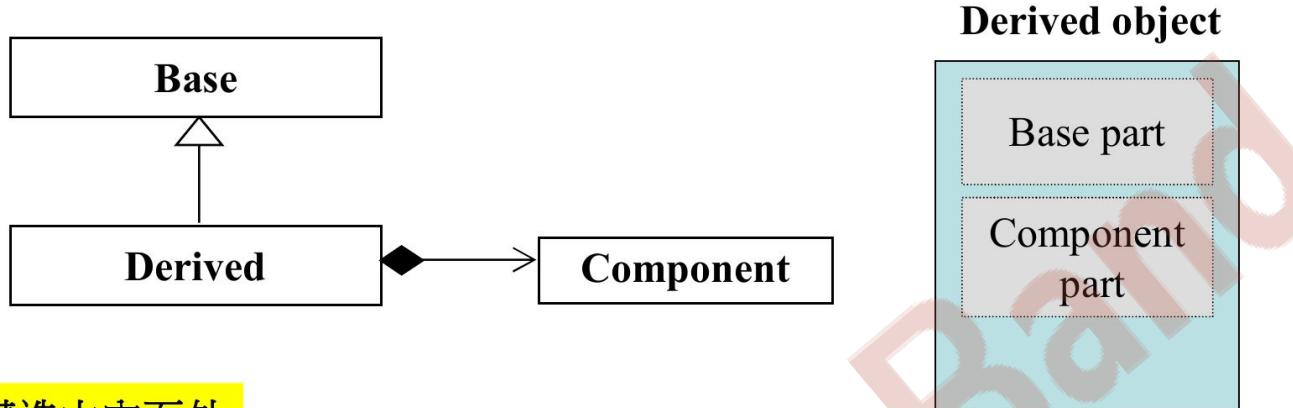
```
Derived::Derived(...) : Base () { ... };
```

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+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。

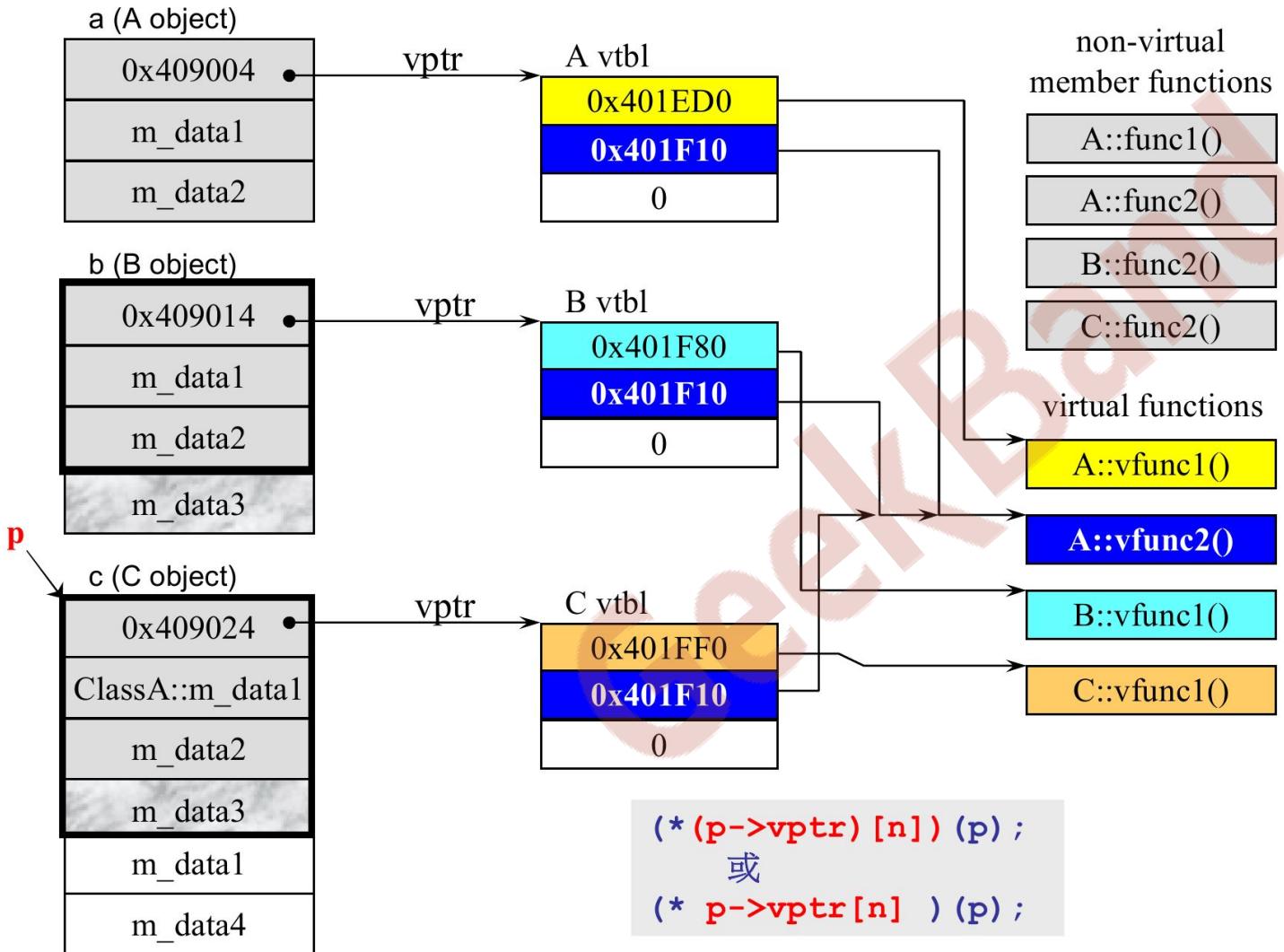
```
Derived::Derived(...): Base(), Component() { ... };
```

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() };
```

對象模型 (Object Model) : 關於 vptr 和 vtbl

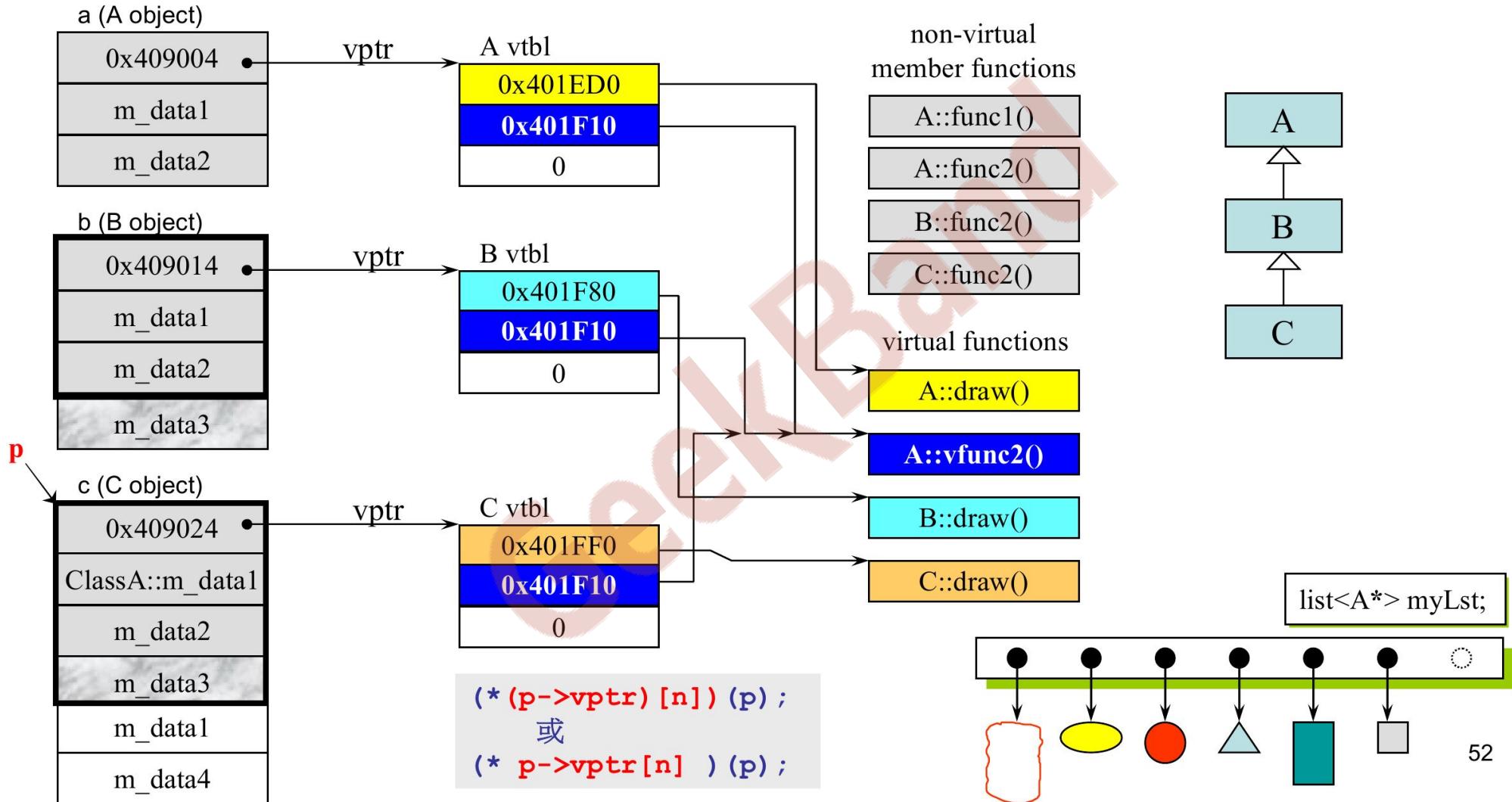


```
class A {
public:
    virtual void vfunc10;
    virtual void vfunc20;
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
```

```
class B : public A {
public:
    virtual void vfunc10;
    void func2();
private:
    int m_data3;
};
```

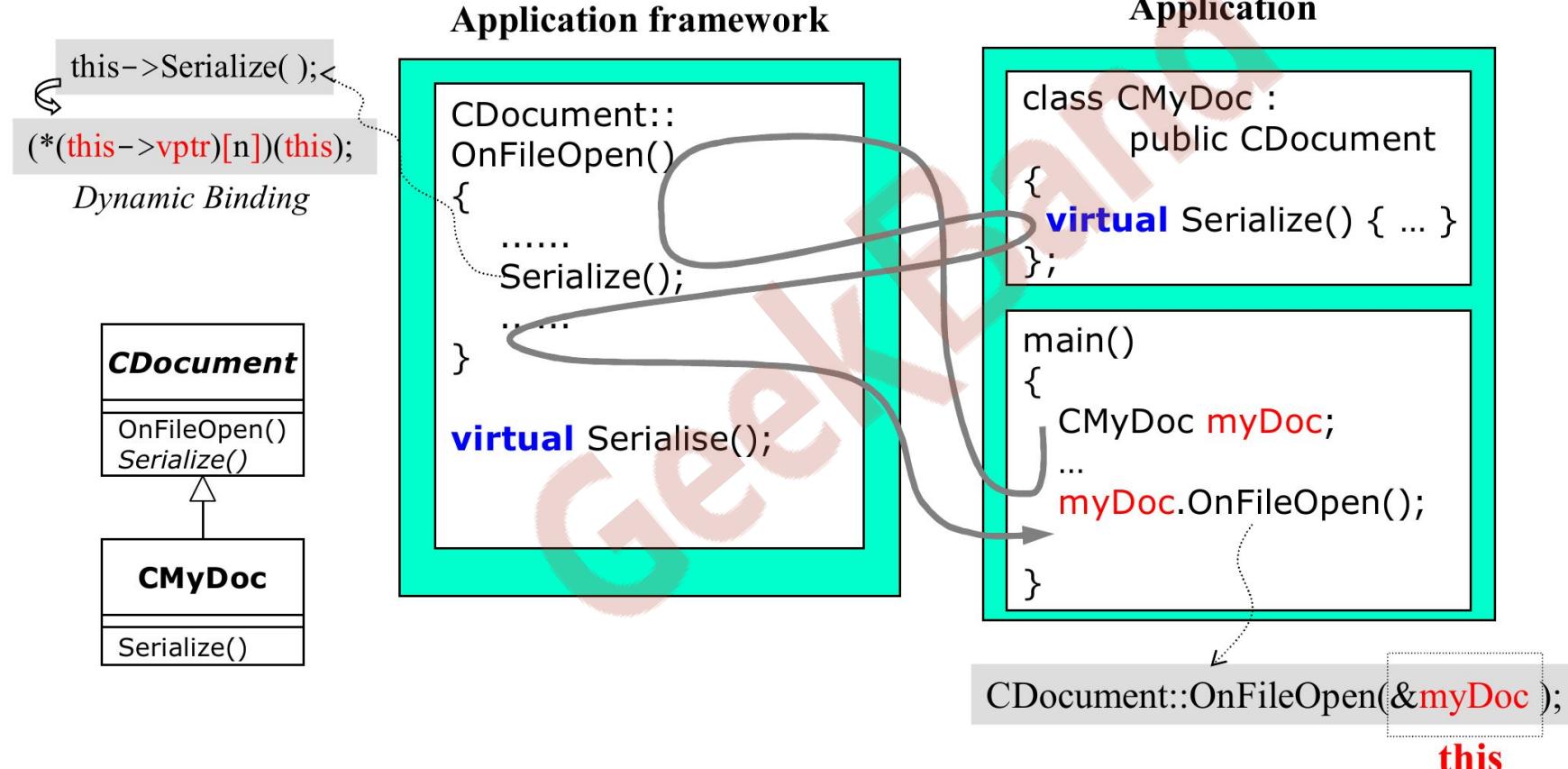
```
class C : public B {
public:
    virtual void vfunc10;
    void func2();
private:
    int m_data1, m_data4;
};
```

對象模型 (Object Model) : 關於 vptr 和 vtbl

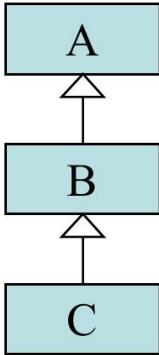


對象模型 (Object Model) : 關於 this

Template Method



對象模型 (Object Model) : 關於 Dynamic Binding



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

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

對象模型 (Object Model) : 關於 Dynamic Binding

The diagram illustrates the dynamic binding mechanism in C++ using inheritance. It shows a class hierarchy with classes C, B, and A. Class A is the base class, B is the derived class, and C is another derived class. Arrows indicate the inheritance relationship: C inherits from B, and B inherits from A.

On the left, the C++ code demonstrates dynamic binding:

```
B b;  
A a = (A)b;  
a.vfunc1();  
  
A* pa = new B;  
pa->vfunc1();  
  
pa = &b;  
pa->vfunc1();
```

On the right, the assembly output shows the generated machine code for these operations. The assembly code includes comments explaining the dynamic binding process:

123: pa->vfunc1(); //jj:dynamic binding. B::vfunc1()
00401D68 mov eax,dword ptr [pa]
00401D6E mov edx,dword ptr [eax]
00401D70 mov esi,esp
00401D72 mov ecx,dword ptr [pa]
00401D78 call **dword ptr [edx]**
00401D7A cmp esi,esp
00401D7C call __chkesp (00423590)

124:
125: pa = &b; //jj:up cast too!
00401D81 lea eax,[b]
00401D84 mov dword ptr [pa],eax

126: pa->vfunc1(); //jj:dynamic binding. B::vfunc1()
00401D8A mov ecx,dword ptr [pa]
00401D90 mov edx,dword ptr [ecx]
00401D92 mov esi,esp
00401D94 mov ecx,dword ptr [pa]
00401D9A call **dword ptr [edx]**
00401D9C cmp esi,esp
00401D9E call __chkesp (00423590)

127:

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



The End

GeekLand

談談 const

const member functions (常量成員函數)

1

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

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

0

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

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

談談 const

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

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

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

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

non-const member functions 可調用 **const** member functions，反之則不行，會引發：

(VC) error C2662: cannot convert 'this' pointer from 'const class X' to 'class X &'. Conversion loses qualifiers

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

charT

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

reference

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

COW : Copy On Write

關於 new, delete

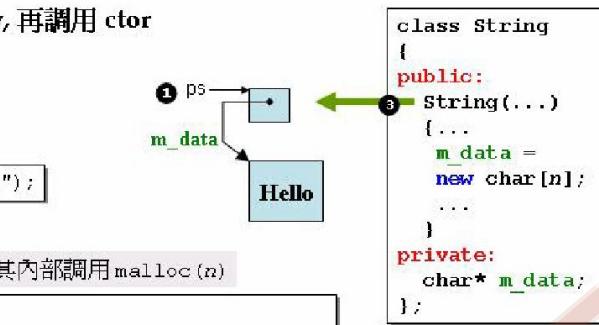
new : 先分配 memory, 再調用 ctor

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

編譯器轉化為

```
string* ps;
```

```
1 void* mem = operator new( 
2 ps = static_cast<String*>( 
3 ps->String::String("Hello"))
```



delete : 先調用 dtor, 再釋放 memory

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

編譯器轉化為

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

其內部調用 free(ps)

動態分配所得的內存塊 (memory block), in VC

00000041
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0x8d
Complex object (8a)
4 個 0x8d
00000000 (pad)
00000000 (pad)
00000000 (pad)
00000041

00000011
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0x8a
String object (4a)
4 個 0x8a
00000031

00000031
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0x8a
String object (4a)
4 個 0x8a
00000031

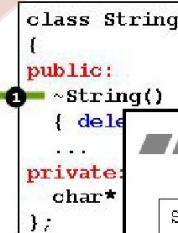
00000011
00790c20
00790b80
0042ede8
0000006d
00000002
00000004
4 個 0x8a
String object (4a)
4 個 0x8a
00000011

$$4 + (32+4) + (4 \times 2) \\ \rightarrow 48$$

$$8 + (32+4) + (4 \times 2) \\ \rightarrow 52 \\ \rightarrow 64$$

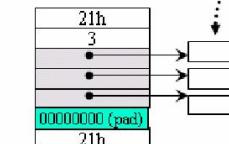
$$8 + (4 \times 2) \\ \rightarrow 16$$

$$4 + (4 \times 2) \\ \rightarrow 12 \\ \rightarrow 16$$



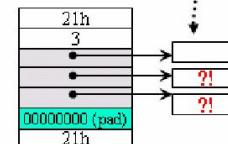
array new 一定要搭配 array delete

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



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

不正確的用法
少了 []



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

小心，這影響無遠弗屆

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

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

///它們不可以被聲明於一個 namespace 內

```
inline void* operator new(size_t size)
```

```
{ cout << "jjhou global new() \n"; return myAlloc( size ); }
```

```
inline void* operator new[](size_t size)
```

```
{ cout << "jjhou global new[]() \n"; return myAlloc( size ); }
```

```
inline void operator delete(void* ptr)
```

```
{ cout << "jjhou global delete() \n"; myFree( ptr ); }
```

```
inline void operator delete[](void* ptr)
```

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

重載 member operator new/delete

```
Foo* p = new Foo;  
...  
  
delete p;
```

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

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

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

重載 member operator new[]/ delete[]

```
Foo* p = new Foo[N];  
...  
delete [] p;
```

```
try {  
    1 void* mem = operator new(sizeof(Foo)*N + 4);  
    p = static_cast<Foo*>(mem);  
    2 p->Foo::Foo(); //N 次  
}  
  
1 p->~Foo(); //N 次  
2 operator delete(p);
```

```
class Foo {  
public:  
    per-class allocator  
    void* operator new[](size_t);  
    void operator delete[](void*, size_t);  
    // ...  
};
```

示例, 接口

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

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

//virtual
~Foo() { cout << "dtor. this=" << this << " id=" << _id << endl; }

static void* operator new(size_t size);
static void operator delete(void* pdead, size_t size);
static void* operator new[](size_t size);
static void operator delete[](void* pdead, size_t size);
};
```

若無 members 就調用 globals

下面強制採 globals

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

```
void* Foo::operator new(size_t size) {
    Foo* p = (Foo*)malloc(size);
    cout << .....
    return p;
}

void Foo::operator delete(void* pdead, size_t size) {
    cout << .....
    free(pdead);
}

void* Foo::operator new[](size_t size) {
    Foo* p = (Foo*)malloc(size);
    cout << .....
    return p;
}

void Foo::operator delete[](void* pdead, size_t size) {
    cout << .....
    free(pdead);
}
```

示例

Foo without
virtual dtor

```

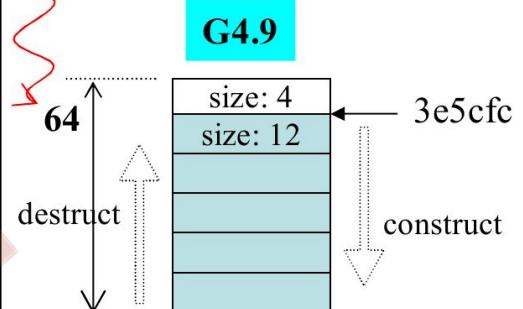
cout << "sizeof(Foo)= " << sizeof(Foo);

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

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

```

1 sizeof(Foo)= 12
2 Foo::operator new(), size=12 return: 0x3e3988
3 ctor. this=0x3e3988 id=7
4 dtor. this=0x3e3988 id=7
5 Foo::operator delete(), pdead= 0x3e3988 size= 12
6 Foo::operator new[], size=64 return: 0x3e5cf8
7 default ctor. this=0x3e5cf8 id=0
8 default ctor. this=0x3e5d08 id=0
9 default ctor. this=0x3e5d14 id=0
10 default ctor. this=0x3e5d20 id=0
11 default ctor. this=0x3e5d2c id=0
12 dtor. this=0x3e5d2c id=0
13 dtor. this=0x3e5d20 id=0
14 dtor. this=0x3e5d14 id=0
15 dtor. this=0x3e5d08 id=0
16 dtor. this=0x3e5cf8 id=0
17 Foo::operator delete[], pdead= 0x3e5cf8 size= 64



Foo with
virtual dtor

```

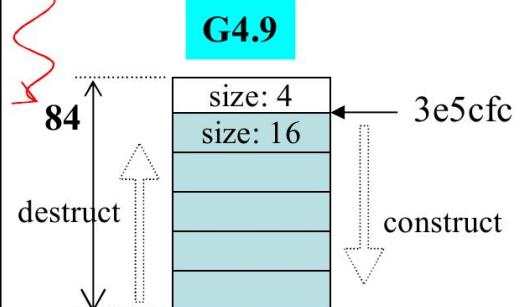
cout << "sizeof(Foo)= " << sizeof(Foo);

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

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

```

1 sizeof(Foo)= 16
2 Foo::operator new(), size=16 return: 0x3e3988
3 ctor. this=0x3e3988 id=7
4 dtor. this=0x3e3988 id=7
5 Foo::operator delete(), pdead= 0x3e3988 size= 16
6 Foo::operator new[], size=84 return: 0x3e5cf8
7 default ctor. this=0x3e5cf8 id=0
8 default ctor. this=0x3e5d0c id=0
9 default ctor. this=0x3e5d1c id=0
10 default ctor. this=0x3e5d2c id=0
11 default ctor. this=0x3e5d3c id=0
12 dtor. this=0x3e5d3c id=0
13 dtor. this=0x3e5d2c id=0
14 dtor. this=0x3e5d1c id=0
15 dtor. this=0x3e5d0c id=0
16 dtor. this=0x3e5cf8 id=0
17 Foo::operator delete[], pdead= 0x3e5cf8 size= 84



示例



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

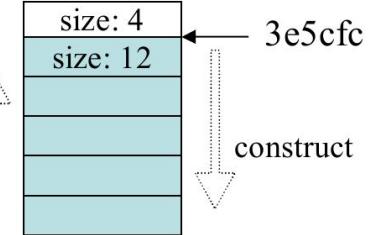
- ③ Foo* pArray = ::new Foo[5];
- ④ ::delete [] pArray;

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

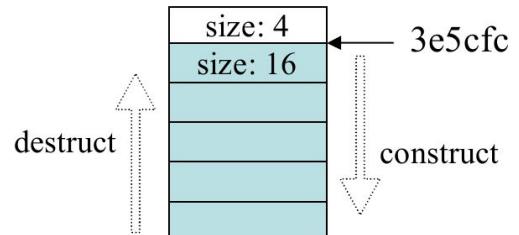
```
ctor. this=0x3e5cce0 id=7
dtor. this=0x3e5cce0 id=7
default ctor. this=0x3e5cfcc id=0
default ctor. this=0x3e5d08 id=0
default ctor. this=0x3e5d14 id=0
default ctor. this=0x3e5d20 id=0
default ctor. this=0x3e5d2c id=0
dtor. this=0x3e5d2c id=0
dtor. this=0x3e5d20 id=0
dtor. this=0x3e5d14 id=0
dtor. this=0x3e5d08 id=0
dtor. this=0x3e5cfcc id=0
```

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

G4.9



G4.9



■ ■ ■ 重載 new(), delete()

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

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

或稱此為
placement operator delete.

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

示例

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

//(1) 這個就是一般的 operator new() 的重載

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

//(2) 這個就是標準庫已提供的 placement new() 的重載 (的形式)

```
// (所以我也模擬 standard placement new, 就只是傳回 pointer)  
void* operator new(size_t size, void* start) {  
    return start;
```

//(3) 這個才是嶄新的 placement new

```
void* operator new(size_t size, long extra) {  
    return malloc(size+extra);  
}  
//(4) 這又是一個 placement new
```

```
void* operator new(size_t size, long extra, char init) {  
    return malloc(size+extra);  
}
```

.....(接下頁)

```
class Bad { };
```

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

//(5) 這又是一個 placement new, 但故意寫錯第一參數的 type
// (那必須是 size_t 以符合正常的 operator new)

```
!! void* operator new(long extra, char init) {  
// [Error] 'operator new' takes type 'size_t' ('unsigned int')  
// as first parameter [-fpermissive]  
//     return malloc(extra);  
// }  
...}
```

示例 (續)

..... (續上頁)

//以下是搭配上述 placement new 的各個所謂 placement delete.
//當 ctor 發出異常，這兒對應的 operator (placement) delete 就會被調用.

//其用途是釋放對應之 placement new 分配所得的 memory.

//(1) 這個就是一般的 operator delete() 的重載

```
void operator delete(void*,size_t)
```

```
{ cout << "operator delete(void*,size_t) " << endl; }
```

//(2) 這是對應上頁的 (2)

```
void operator delete(void*,void*)
```

```
{ cout << "operator delete(void*,void*) " << endl; }
```

//(3) 這是對應上頁的 (3)

```
void operator delete(void*,long)
```

```
{ cout << "operator delete(void*,long) " << endl; }
```

//(4) 這是對應上頁的 (4)

```
void operator delete(void*,long,char)
```

```
{ cout << "operator delete(void*,long,char) " << endl; }
```

private:

```
int m_i;
```

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



Foo start;

① Foo* p1 = new Foo;

② Foo* p2 = new (&start) Foo;

③ Foo* p3 = new (100) Foo;

④ Foo* p4 = new (100,'a') Foo;

⑤ Foo* p5 = new (100) Foo(1);

Foo* p6 = new (100,'a') Foo(1);

Foo* p7 = new (&start) Foo(1);

Foo* p8 = new Foo(1);

ctor拋出異常

Foo::Foo()

① operator new(size_t size), size= 4

Foo::Foo()

② operator new(size_t size, void* start), size= 4 start= 0x22fe8c

Foo::Foo()

③ operator new(size_t size, long extra) 4 100

Foo::Foo()

④ operator new(size_t size, long extra, char init) 4 100 a

Foo::Foo()

⑤ operator new(size_t size, long extra) 4 100

Foo::Foo<int>

terminate called after throwing an instance of 'jj07::Bad'

ctor 拋出異常

奇怪, G4.9 沒調用 operator delete (void*,long),
但 G2.9 確實調用了.

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

basic_string 使用 new(extra) 擴充申請量



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

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

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



The End

GeekLand