



博览网，高端IT在线学习平台

www.boolan.com

更多精彩课程推荐



C++设计模式

课程从设计之道和设计之术两方面，通过大量的代码实践与演练，深入剖析经典GOF 23种设计模式。



C++ 面向对象开发

助你正确理解面向对象(OO)的精神和实现手法，课程涵盖对象模型、关键机制、编程风格、动态分配。



STL标准库与泛型编程

深入剖析标准库之六大部件：分配器、容器、算法、迭代器、仿函数、适配器之间的体系结构，并分析其源码，引导高阶泛



C++新标准C++11-14

使你在极短时间内深刻理解C++2.0的诸多新特性，涵盖语言和标准库两层面，只谈新东西。

內存管理

從平地到萬丈高樓

Memory Management 101

第一講 primitives

第二講 std::allocator

第三講 malloc/free

第四講 loki::allocator

第五講 other issues



侯捷

萬丈高樓平地起

源碼之前
了無秘密



你應具備的基礎

- 曾經動態分配並使用 **memory**
- 曾經使用過 C++ 標準庫的容器 (**containers**)



我們的目標

從平地到萬丈高樓，

從最基礎的 C++ 語言構件
到高知名度的內存管理器，

徹底了解高高低低的方方面面。



這些工具對你有幫助

(Dev-C++ 5.11 畫面; with GNU 4.9.2)

The screenshot shows the Dev-C++ 5.11 IDE interface. The main window displays a C++11 project with a single source file, Test-C++11.cpp. The code in the editor window is as follows:

```
2195 // 從語法上試用各式各樣的 allocators
2196 cout << sizeof(__gnu_cxx::malloc_allocator<int>) << endl; // 1. 大小 1 個其實為 0, fields 都是 static
2197 cout << sizeof(__gnu_cxx::__pool_alloc<int>) << endl; // 1
2198 cout << sizeof(__gnu_cxx::__mt_alloc<int>) << endl; // 1
2199 cout << sizeof(__gnu_cxx::bitmap_allocator<int>) << endl; // 1
2200 cout << sizeof(__gnu_cxx::array_allocator<int>) << endl; // 8 ==> 因為它有一個 ptr 指向 array 和一個
2201 cout << sizeof(__gnu_cxx::debug_allocator<std::allocator<double>>) << endl; // 8
2202 // cout << sizeof(__gnu_cxx::throw_allocator<int>) << endl; // 只有 throw_allocator_base, throw_allocator
2203
2204 // 搞配容器
2205 list<int, __gnu_cxx::malloc_allocator<int>> list_malloc;
2206 deque<int, __gnu_cxx::debug_allocator<std::allocator<int>>> deque_debug;
2207 vector<int, __gnu_cxx::__pool_alloc<int>> vector_pool;
2208
2209
2210
2211
2212 // 測試 cookie
2213 cout << "sizeof(int)=" << sizeof(int) << endl; // 4
2214 cout << "sizeof(double)=" << sizeof(double) << endl; // 8
2215
2216 cookie_test(std::allocator<int>(), 1); // 相距 10h (表示帶 cookie)
2217 cookie_test(__gnu_cxx::malloc_allocator<int>(), 1); // 相距 10h (表示帶 cookie)
```

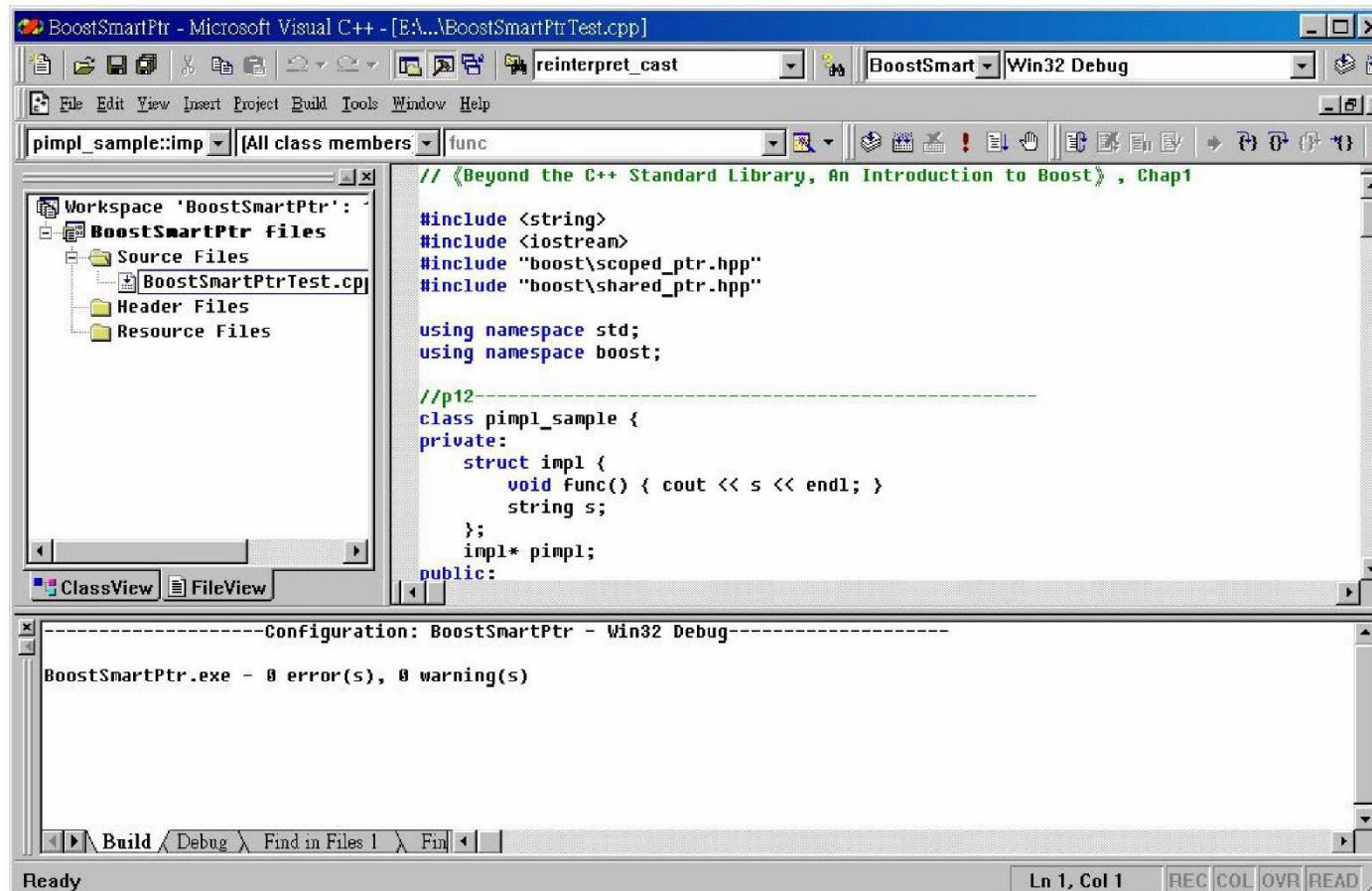
The compiler log at the bottom shows the command and results of the compilation:

```
g++.exe Test-C++11.o -o C++11.exe -L"C:/Program Files/Dev-Cpp/MinGW64/lib32" -LC:/Program Files/Dev-Cpp/MinGW64/lib
Compilation results...
-----
- Errors: 0
- Warnings: 0
- Output Filename: D:\handout\C++11-test-DevC++\Test-C++11\C++11.exe
- Output Size: 2.10485744476318 MiB
- Compilation Time: 7.05s
```

Bottom status bar: Line: 2161 Col: 1 Set: 0 Lines: 2989 Length: 105275 Insert Done parsing 511 files in 1.281 seconds (398.91 files per second)

■■■■ 這些工具對你有幫助

(Visual C++ 6.0)





網絡有許多資源 …

Doug Lea's Home Page

Surface mail: Doug Lea, Computer Science Department, State University of New York at Oswego, Oswego, NY 13126 | Voice: 315-312-2688 | Fax: 315-312-5424 | E-Mail: d1@cs.oswego.edu | [vita](#) | [shortbio](#)

Documents

Books

- Online supplement to the book [Concurrent Programming in Java: Design Principles and Patterns](#), (second edition) published November 1, 1999 by [Addison-Wesley](#)
- HTML edition of the book [Object Oriented System Development](#), by Dennis Lea, and Penelope Faure, originally published in 1993 by [Addison-Wesley](#).
- Companion site for [Java Concurrency in Practice](#) by Brian Goetz, with Tim Peierls, Bowbeer, David Holmes, Doug Lea. Addison-Wesley, 2006.

Software

- [JSR166 \(JDK 1.5 java.util.concurrent and jsr166x\) specs, etc.](#)
- [Concurrency utilities for Java EE](#)
- Version 1.3.4 of [util.concurrent](#), a [Java](#) package containing locks, queues, threads, and other primitives, a precursor to `java.util.concurrent`.
- [collections](#), a [Java](#) package that was a precursor to some JDK collections.
- [Microscope](#): a cute Java applet.
- Version 2.8.6 of [malloc.c](#), and [malloc.h](#). Also a short article on [the design of this](#) some [plots](#). Also a [malloc tracer](#) written by [Wolfram Gloger](#) and a [sample trace](#) of [malloc](#).

A Memory Allocator

by [Doug Lea](#)

[A German adaptation and translation of this article appears in unix/mail December, 1996. This article is now out of date, and doesn't reflect details of current version of malloc.]

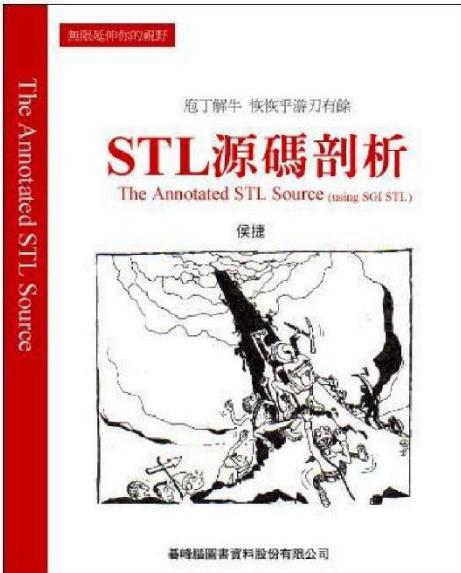
Introduction

Memory allocators form interesting case studies in the engineering of infrastructure software. I started writing one in 1987, and have maintained and evolved it (with the help of many volunteer contributors) ever since. This allocator provides implementations of the standard C routines `malloc()`, `free()`, and `realloc()`, as well as a few auxiliary utility routines. The allocator has never been given a specific name. Most people just call it *Doug Lea's Malloc*, or *d1malloc* for short.

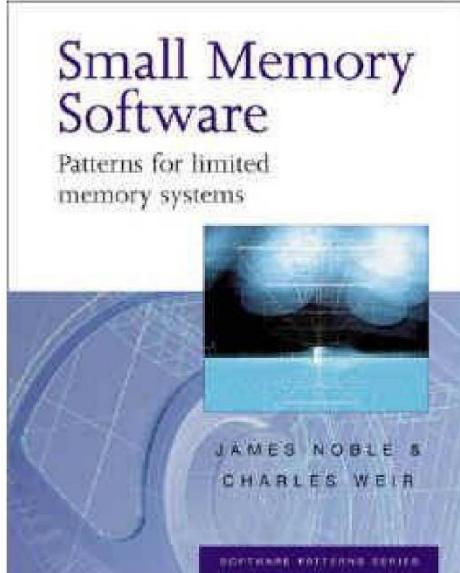
Doug Lea 自1986年起潛心研究 malloc 算法，其作品被稱為 DL Malloc。目前 Linux 的 glibc 中的 malloc 算法就是直接來自 Doug Lea，其他平台的 malloc 實現也或多或少受到 DL 的影響。DL Malloc 源碼可下載自 Doug Lea 個人主頁：
<http://gee.cs.oswego.edu/dl/>



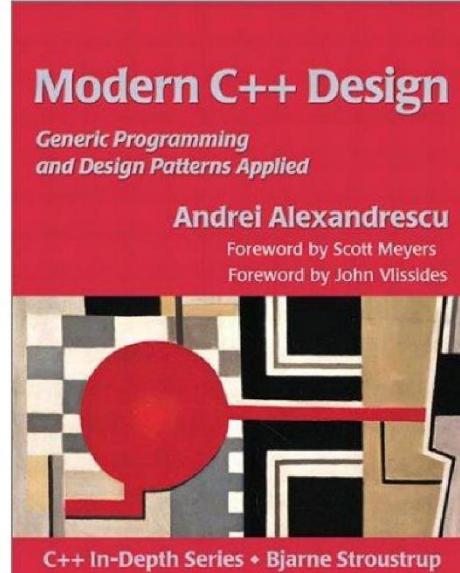
Bibliography



by 侯捷
Chap2 : allocator



By James Noble &
Charles Weir,
AW 2001



by Andrei Alexandrescu,
Chap4: Small-Object Allocation

- Libraries :
- STL Allocators
 - MFC CPlex+CFixedAlloc
 - Boost.Pool
 - Loki SmallObjAllocator
 - VC malloc/free
 - jemalloc
 - tcmalloc
 - ...



你將獲得的代碼

Test-Mem.cpp

內有出現於本講義中的所有代碼



天寶當年, DOS 640K, extented memory.

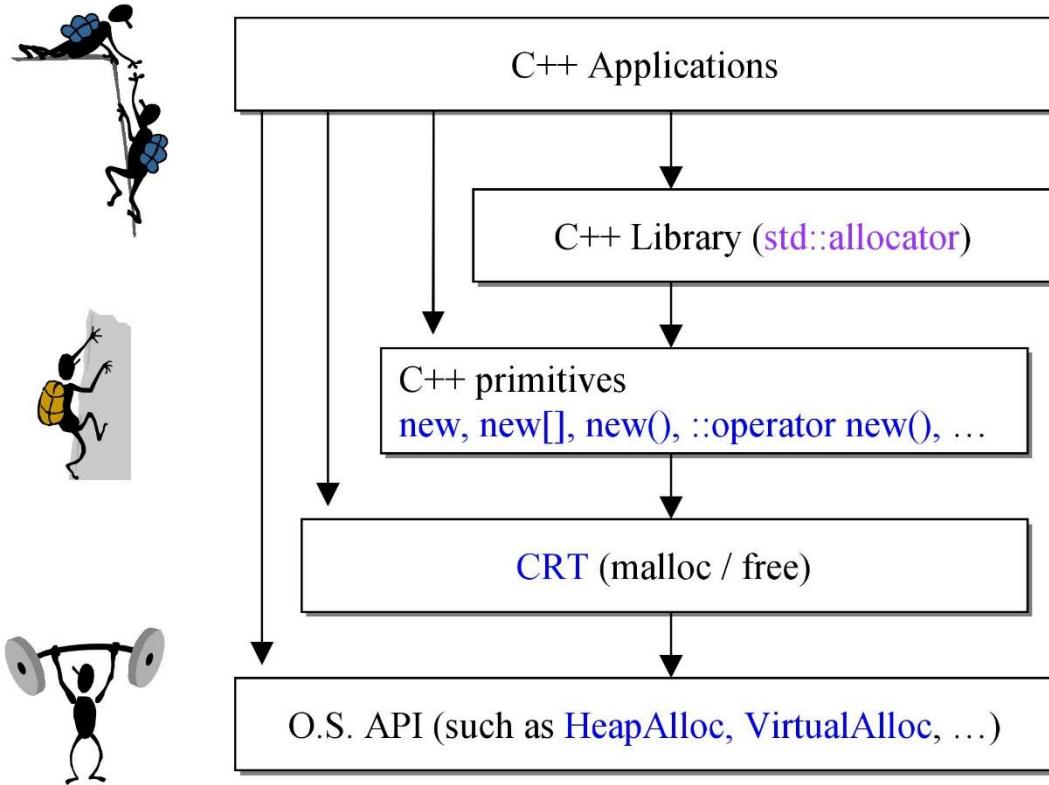


錙銖必較

俱往矣 且看今朝



C++應用程序, 使用memory的途徑





C++ memory primitives

分配	釋放	類屬	可否重載
<code>malloc()</code>	<code>free()</code>	C 函數	不可
<code>new</code>	<code>delete</code>	C++表達式 (expressions)	不可
<code>::operator new()</code>	<code>::operator delete()</code>	C++函數	可
<code>allocator<T>::allocate()</code>	<code>allocator<T>::deallocate()</code>	C++標準庫	可自由設計並以之搭配任何容器



C++ memory primitives

```
#01 void* p1 = malloc(512); //512 bytes
#02 free(p1);
#03
#04 complex<int>* p2 = new complex<int>; //one object
#05 delete p2;
#06
#07 void* p3 = ::operator new(512); //512 bytes
#08 ::operator delete(p3);
#09
#10 //以下使用 C++ 標準庫提供的 allocators。
#11 //其接口雖有標準規格，但實現廠商並未完全遵守；下面三者形式略異。
#12 #ifdef _MSC_VER
#13     //以下兩函數都是 non-static，定要通過 object 調用。以下分配 3 個 ints.
#14     int* p4 = allocator<int>().allocate(3, (int*)0);
#15     allocator<int>().deallocate(p4, 3);           ↗ 無用
#16 #endif
#17 #ifdef _BORLANDC_
#18     //以下兩函數都是 non-static，定要通過 object 調用。以下分配 5 個 ints.
#19     int* p4 = allocator<int>().allocate(5);
#20     allocator<int>().deallocate(p4, 5);
#21 #endif
#22 #ifdef _GNUC_
#23     //以下兩函數都是 static，可通過全名調用之。以下分配 512 bytes.
#24     void* p4 = alloc::allocate(512);
#25     alloc::deallocate(p4, 512);
#26 #endif
```



C++ memory primitives

```
#01 void* p1 = malloc(512); //512 bytes
#02 free(p1);
#03
#04 complex<int>* p2 = new complex<int>; //one object
#05 delete p2;
#06
#07 void* p3 = ::operator new(512); //512 bytes
#08 ::operator delete(p3);
#09
#10 #ifdef __GNUC__
#11     //以下兩函數都是 non-static，定要通過 object 調用。以下分配 7 個 ints
#12     void* p4 = allocator<int>().allocate(7);
#13     allocator<int>().deallocate((int*)p4, 7);
#14
#15     //以下兩函數都是 non-static，定要通過 object 調用。以下分配 9 個 ints.
#16     void* p5 = __gnu_cxx::__pool_alloc<int>().allocate(9);
#17     __gnu_cxx::__pool_alloc<int>().deallocate((int*)p5, 9);
#18 #endif
```



new expression

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

編譯器
轉為

```
Complex *pc;
try {
    1 void* mem = operator new( sizeof(Complex) ); //allocate
    2 pc = static_cast<Complex*>(mem);           //cast
    3 pc->Complex::Complex(1,2);                  //construct
    ... //注意：只有編譯器才可以像上面那樣直接呼叫 ctor
}
catch( std::bad_alloc ) {
    //若allocation失敗就不執行constructor
}
```

→ App. 欲直接調用 ctor, 可運用 placement new :
`new(p) Complex(1, 2);`

void *operator new(size_t size, const std::nothrow_t&)
 _THROW0()
{ // try to allocate size bytes
void *p;
while ((p = malloc(size)) == 0)
{ // buy more memory or return null pointer
 _TRY_BEGIN
 if (_callnewh(size) == 0) break;
 _CATCH(std::bad_alloc) return (0);
 _CATCH_END
}
return (p);
}

The struct is used as a function parameter to `operator new` to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

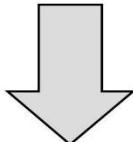
```
struct std::nothrow_t {};
```

—侯捷—



delete expression

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



編譯器轉為...

```
pc->~Complex();           //先析構  
operator delete(pc);      //然後釋放內存
```

```
void __cdecl operator delete(void *p) _THROW0()  
{ // free an allocated object  
    free(p);  
}
```

...\\vc98\\crt\\src\\delop.cpp



Ctor & Dtor 直接調用



```
96     string* pstr = new string;
97     cout << "str= " << *pstr << endl;
98
99 //! pstr->string::string("jjhou");
100 //! [Error] 'class std::basic_string<char>' has no member named 'string'
101 //! pstr->~string(); //crash
102     cout << "str= " << *pstr << endl;
```

```
84 class A
85 {
86 public:
87     int id;
88     A(int i) : id(i) { cout << "ctor. this=" << this << " id=" << id << endl; }
89     ~A()           { cout << "dtor. this=" << this << endl; }
90 };
```



```
107     A* pA = new A(1);          //ctor. this=000307A8 id=1
108     cout << pA->id << endl;   //!
109 //! pA->A::A(3);           //in VC6 : ctor. this=000307A8 id=3
110 //! [Error] cannot call constructor 'jj02::A::A' directly
111
112 //! A::A(5);                //in VC6 : ctor. this=0013FF60 id=5
113 //!                         //      dtor. this=0013FF60
114 //! [Error] cannot call constructor 'jj02::A::A' directly
115 //!                         //      [Note] for a function-style cast, remove the redundant '::A'
116
117     cout << pA->id << endl;   //in VC6 : 3
118 //!                         //in GCC : 1
119
120     delete pA;              //dtor. this=000307A8
```



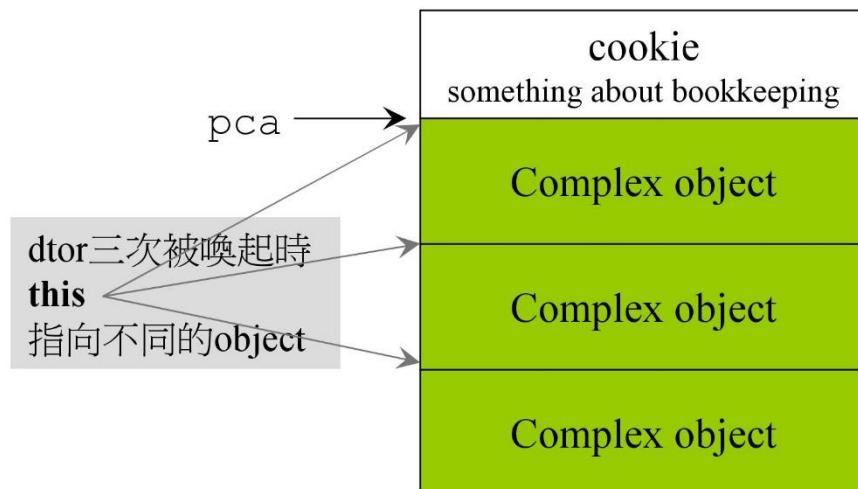
array new, array delete

```
Complex* pca = new Complex[3];  
//喚起三次ctor。  
//無法藉由參數給予初值  
...  
delete[] pca; //喚起3次dtor
```

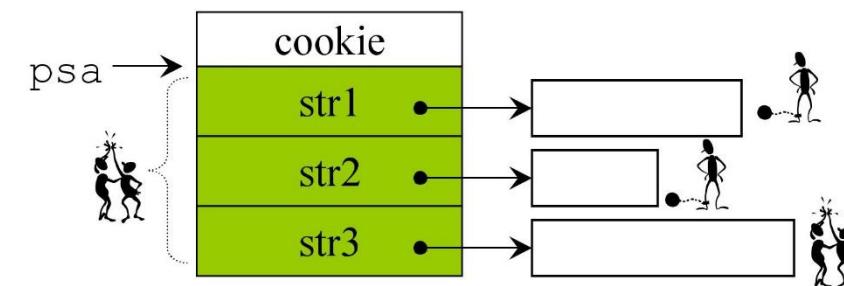


沒對每個 object 調用 dtor，有什麼影響？

- 對 class without ptr member 可能沒影響。
- 對 class with pointer member 通常有影響。



```
string* psa = new string[3];  
...  
delete psa; //喚起1次dtor
```





array new, array delete

```
class A
{
public:
    int id;

    A() : id(0) { cout << "default ctor. this=" << this << " id=" << id << endl; }
    A(int i) : id(i) { cout << "ctor. this=" << this << " id=" << id << endl; }
    ~A() { cout << "dtor. this=" << this << " id=" << id << endl; }
};
```

```
default ctor. this=0x3e398c id=0
default ctor. this=0x3e3990 id=0
default ctor. this=0x3e3994 id=0
buf=0x3e398c tmp=0x3e398c
ctor. this=0x3e398c id=0
ctor. this=0x3e3990 id=1
ctor. this=0x3e3994 id=2
buf=0x3e398c tmp=0x3e3998
dtor. this=0x3e3994 id=2
dtor. this=0x3e3990 id=1
dtor. this=0x3e398c id=0
```

```
A* buf = new A[size]; //default ctor 3 次. [0]先於[1]先於[2])
//A必須有 default ctor,
//否則 [Error] no matching function for call to 'A::A()'
A* tmp = buf;

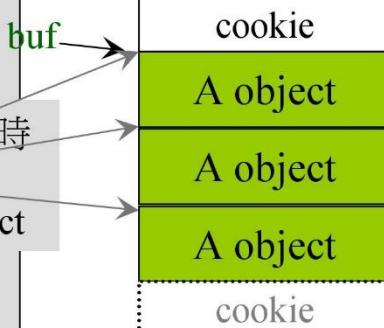
cout << "buf=" << buf << " tmp=" << tmp << endl;

for(int i = 0; i < size; ++i)
    new (tmp++) A(i); //ctor 3次

cout << "buf=" << buf << " tmp=" << tmp << endl;

delete [] buf; //dtor 3 次 (次序逆反, [2]先於[1]先於[0])
```

dtor 三次被喚起時
this 指向不同的object

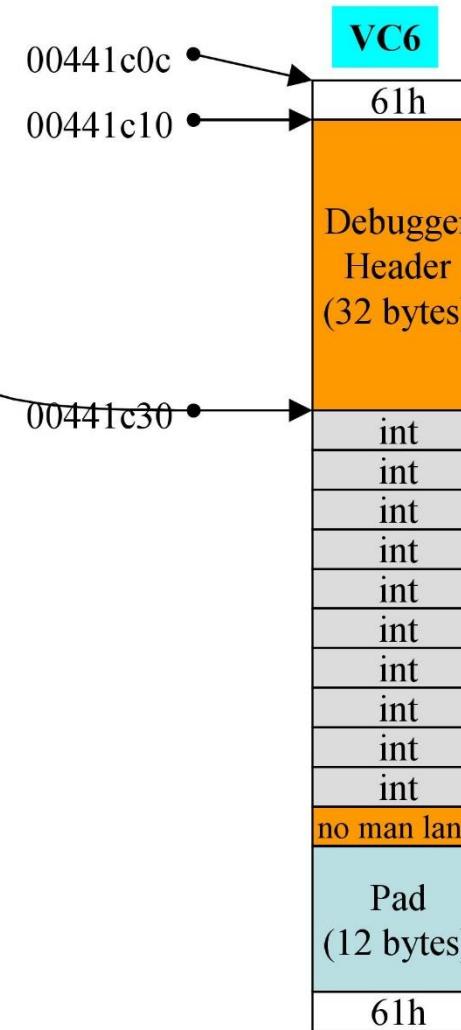




array size, in memory block

```
int* pi = new int[10]; //sizeof(pi) : 4  
delete pi;
```

```
int ia[10]; //from stack but not heap  
cout << sizeof(ia); //40
```

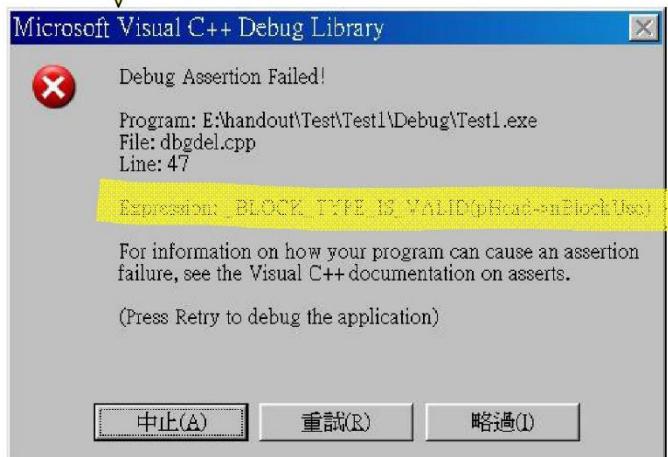




array size, in memory block

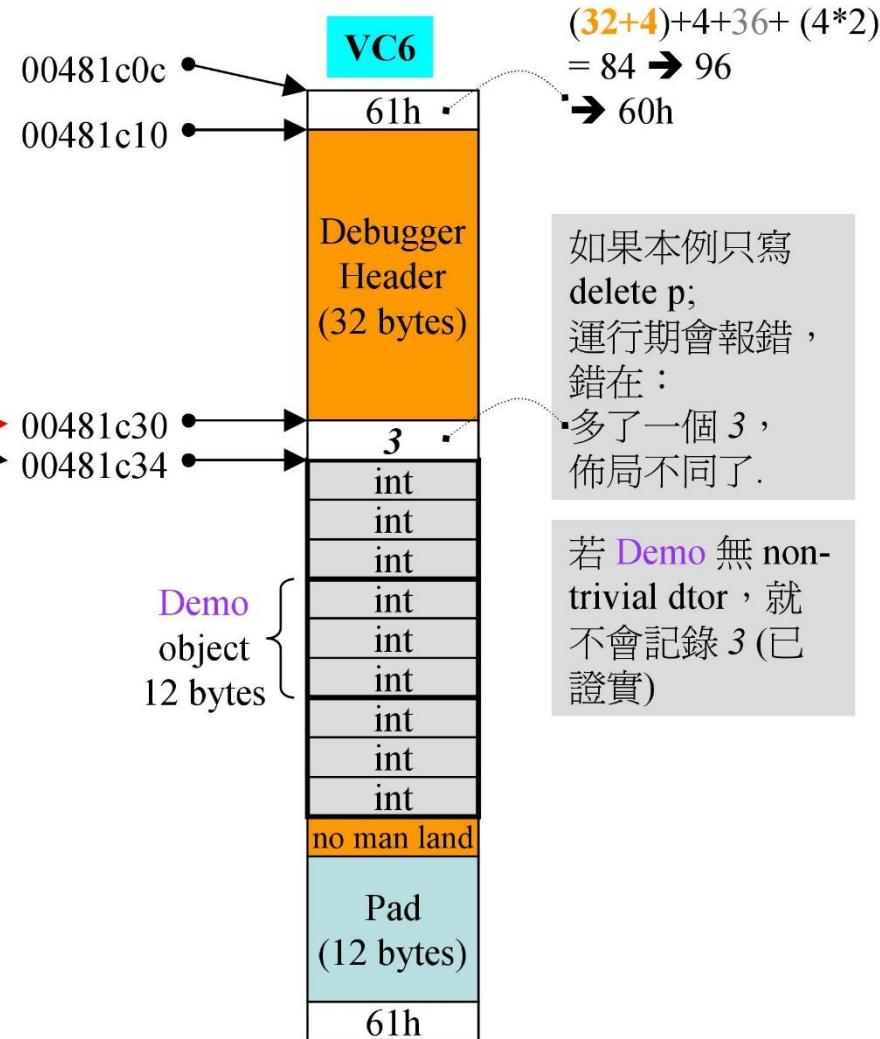
```
Demo* p = new Demo[3];
delete [] p;
```

若未寫 [],
debug mode 下出現
Assertion Failed.



```
Demo d[3]; //from stack but not heap
cout << sizeof(d); //36
```

注意，重載 operator delete[] 後
觀察到傳入的 p 竟指向

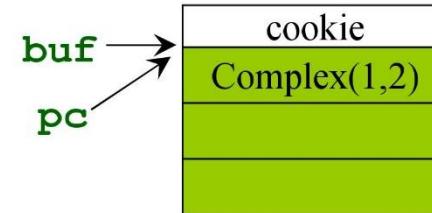


placement new

- placement new 允許我們將 object 建構於 allocated memory 中。
- 沒有所謂 placement delete，因為 placement new 根本沒分配 memory.
亦或稱呼與 placement new 對應的 operator delete 為 placement delete.

注意，關於“placement new”，
或指 `new(p)`，
或指 `::operator new(size_t, void*)`

```
#include <new>
char* buf = new char[sizeof(Complex)*3];
Complex* pc = new(buf) Complex(1,2);
...
delete [] buf; //
```



編譯器轉為

```
Complex *pc;
try {
    1 void* mem = operator new(sizeof(Complex), buf); //allocate
    2 pc = static_cast<Complex*>(mem); //cast
    3 pc->Complex::Complex(1,2); //construct
}
catch( std::bad_alloc ) {
    //若allocation失敗就不執行constructor
}
```

```
void* operator new(size_t, void* loc)
{ return loc; }
```



C++應用程序, 分配內存的途徑

應用程序

```
Foo* p = new Foo(x);  
...  
delete p;
```

expression
(不可改變)
(不可重載)

```
Foo* p = (Foo*)operator new(sizeof(Foo));  
new (p) Foo(x);  
...  
p->~Foo();  
operator delete (p);
```

so, 我也可以模仿 new expression :

```
Foo* p = (Foo*)malloc(sizeof(Foo));  
new (p) Foo(x); //invoke ctor  
...  
p->~Foo(); //invoke dtor  
free (p);
```

member functions

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

1

可重載



我們有上↑下↓
兩個機會(兩個
地點)可以改變
內存分配機制

global functions

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

2

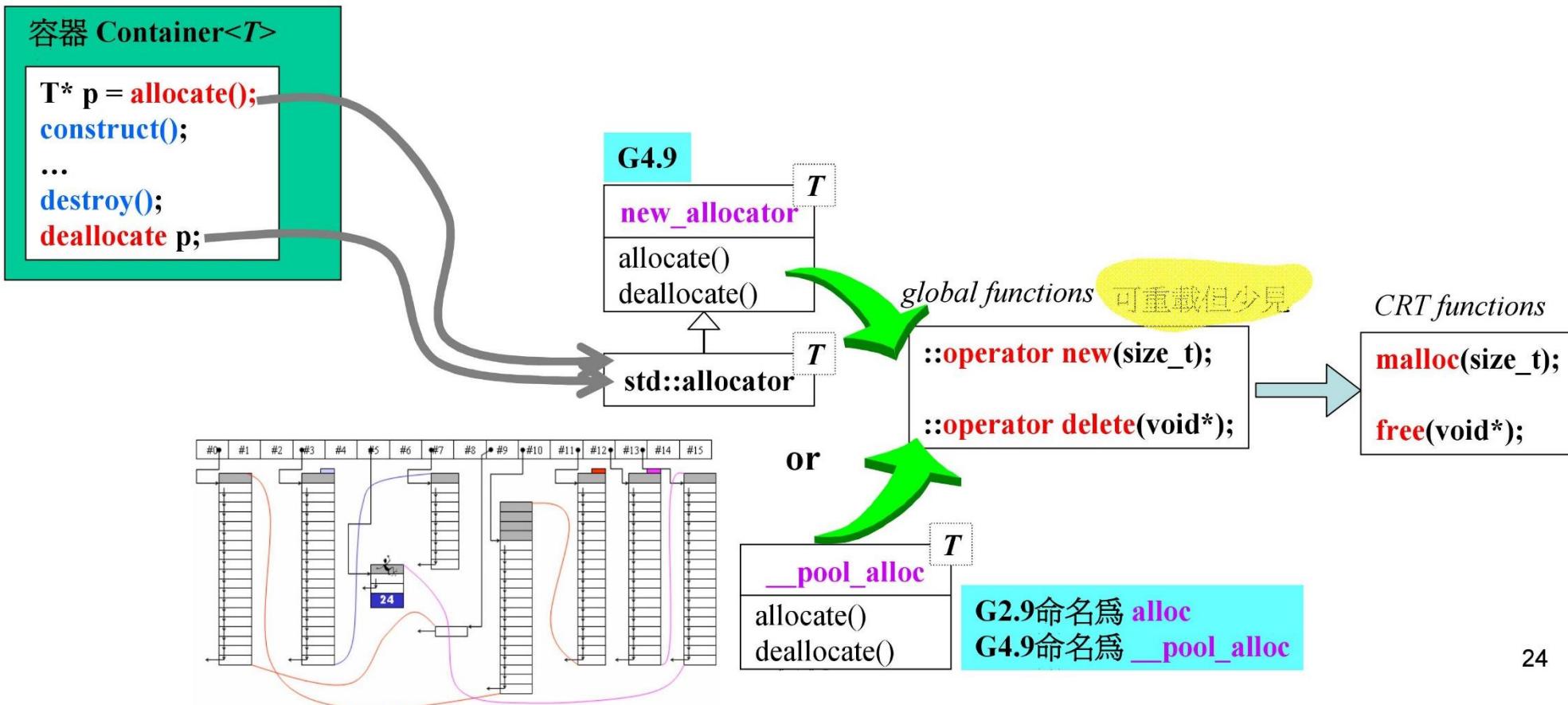
CRT functions

malloc(size_t);
free(void*);

可重載但少見



C++容器, 分配內存的途徑





重載 ::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 ); }
```



—侯捷—

...\\vc98\\crt\\src\\newop2.cpp

```
void* operator new(size_t size, const std::nothrow_t&)
    _THROW0()
```

```
{ // try to allocate size bytes
```

```
void *p;
```

```
while ((p = malloc(size)) == 0)
```

```
{ // buy more memory or return null pointer
```

```
_TRY_BEGIN
```

```
if (_callnewh(size) == 0) break;
```

```
_CATCH(std::bad_alloc) return (0);
```

```
_CATCH_END
```

```
}
```

```
return (p);
```

```
}
```

...\\vc98\\crt\\src\\delop.cpp

```
void __cdecl operator delete(void *p) _THROW0()
```

```
{ // free an allocated object
```

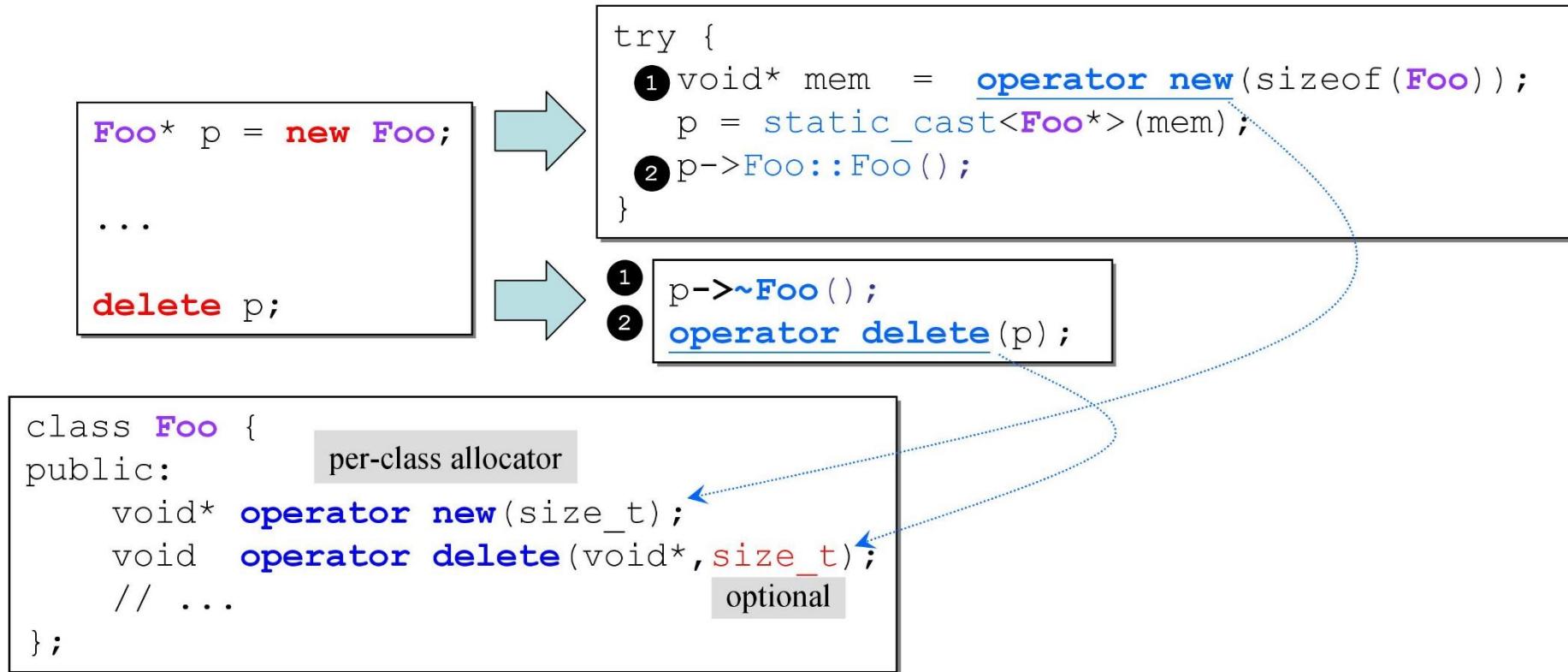
```
free(p);
```

```
}
```



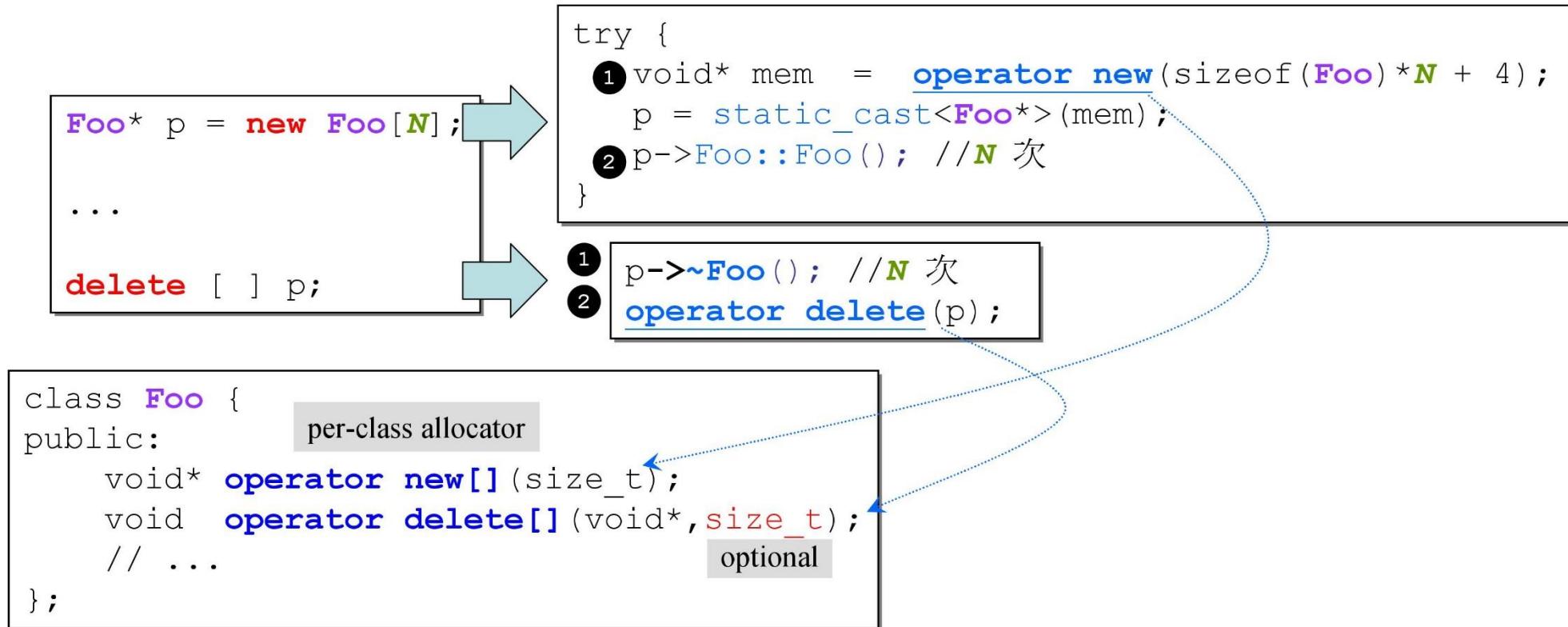


重载 operator new/operator delete



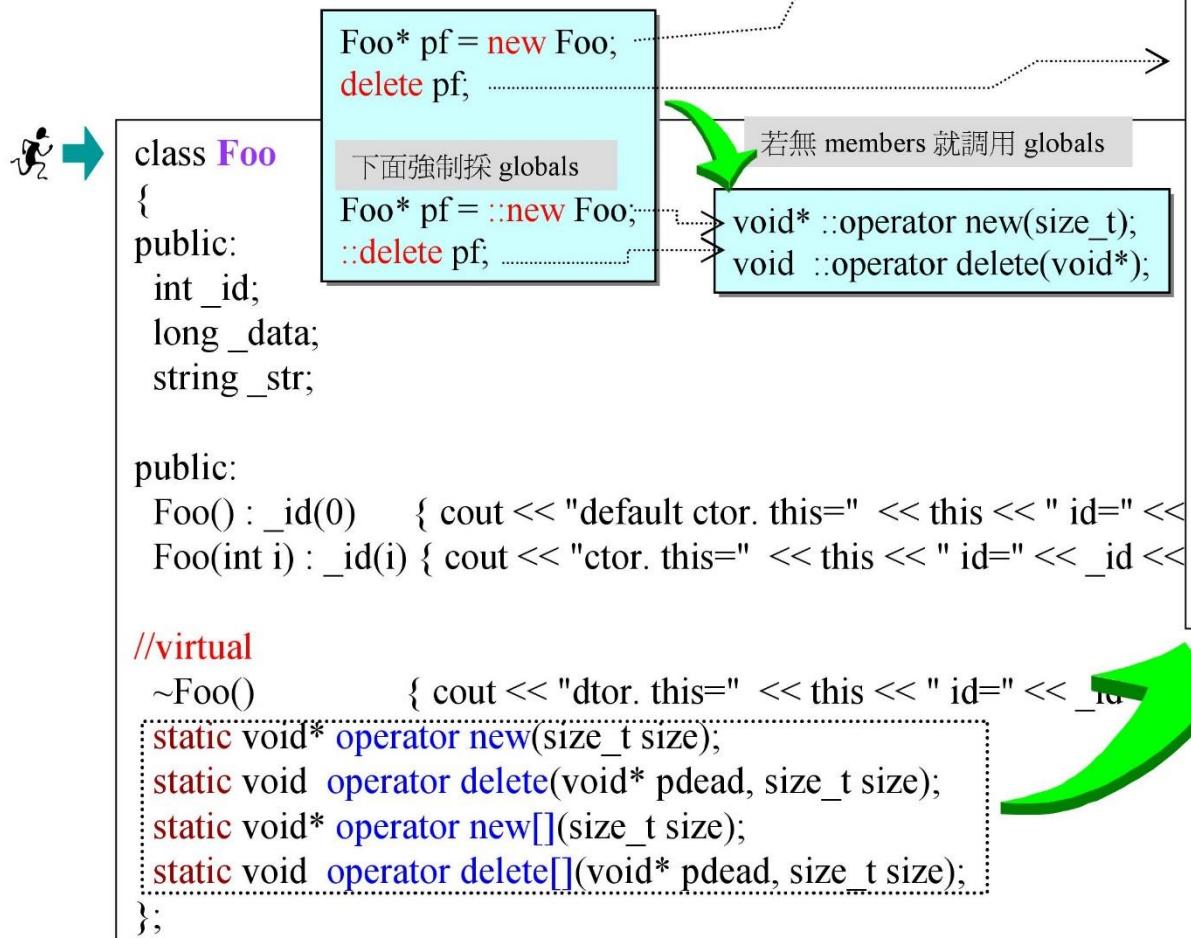


重載 operator new[]/ operator delete[]





示例, 接口



```
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;
```

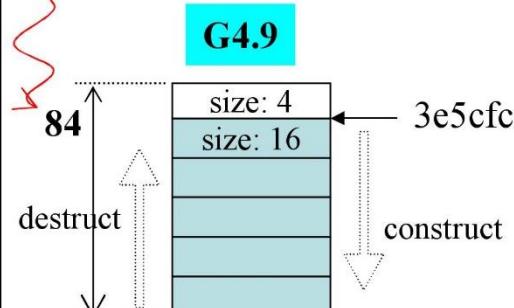
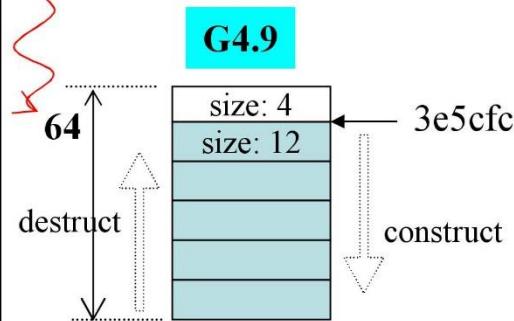
Foo with
virtual dtor



```
...._operator_new_array_no...  
sizeof(Foo)= 12  
Foo::operator new(), size=12      return: 0x3e3988  
ctor. this=0x3e3988 id=7  
dtor. this=0x3e3988 id=7  
Foo::operator delete(), pdead= 0x3e3988 size= 12  
Foo::operator new[], size=64     return: 0x3e5cf8  
default ctor. this=0x3e5cf8 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=0x3e5cf8 id=0  
Foo::operator delete[], pdead= 0x3e5cf8 size= 64
```



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





示例



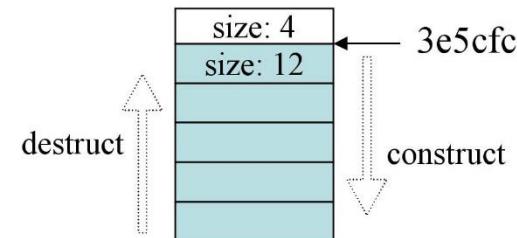
- ① Foo* p = ::new Foo(7);
- ② ::delete p;
- ③ Foo* pArray = ::new Foo[5];
- ④ ::delete [] pArray;

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

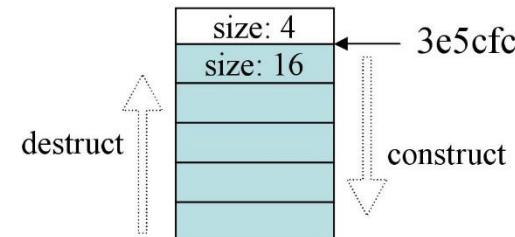
```
ctor. this=0x3e5ce0 id=7
dtor. this=0x3e5ce0 id=7
default ctor. this=0x3e5cf0 id=0
default ctor. this=0x3e5d08 id=0
default ctor. this=0x3e5d14 id=0
default ctor. this=0x3e5d20 id=0
default ctor. this=0x3e5d2c id=0
dtor. this=0x3e5d2c id=0
dtor. this=0x3e5d20 id=0
dtor. this=0x3e5d14 id=0
dtor. this=0x3e5d08 id=0
dtor. this=0x3e5cf0 id=0
```

↑ 都沒有進入我重載的
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.

```
void operator delete(void*,size_t)
{ cout << "operator delete(void*,size_t) " << er
```

//(2) 這是對應上頁的 (2)

```
void operator delete(void*,void*)  
{ cout << "operator delete(void*,vo
```

//(3) 這是對應上面的 (3)

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

//(4) 這是對應上面的 (4)

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

```
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。

1

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

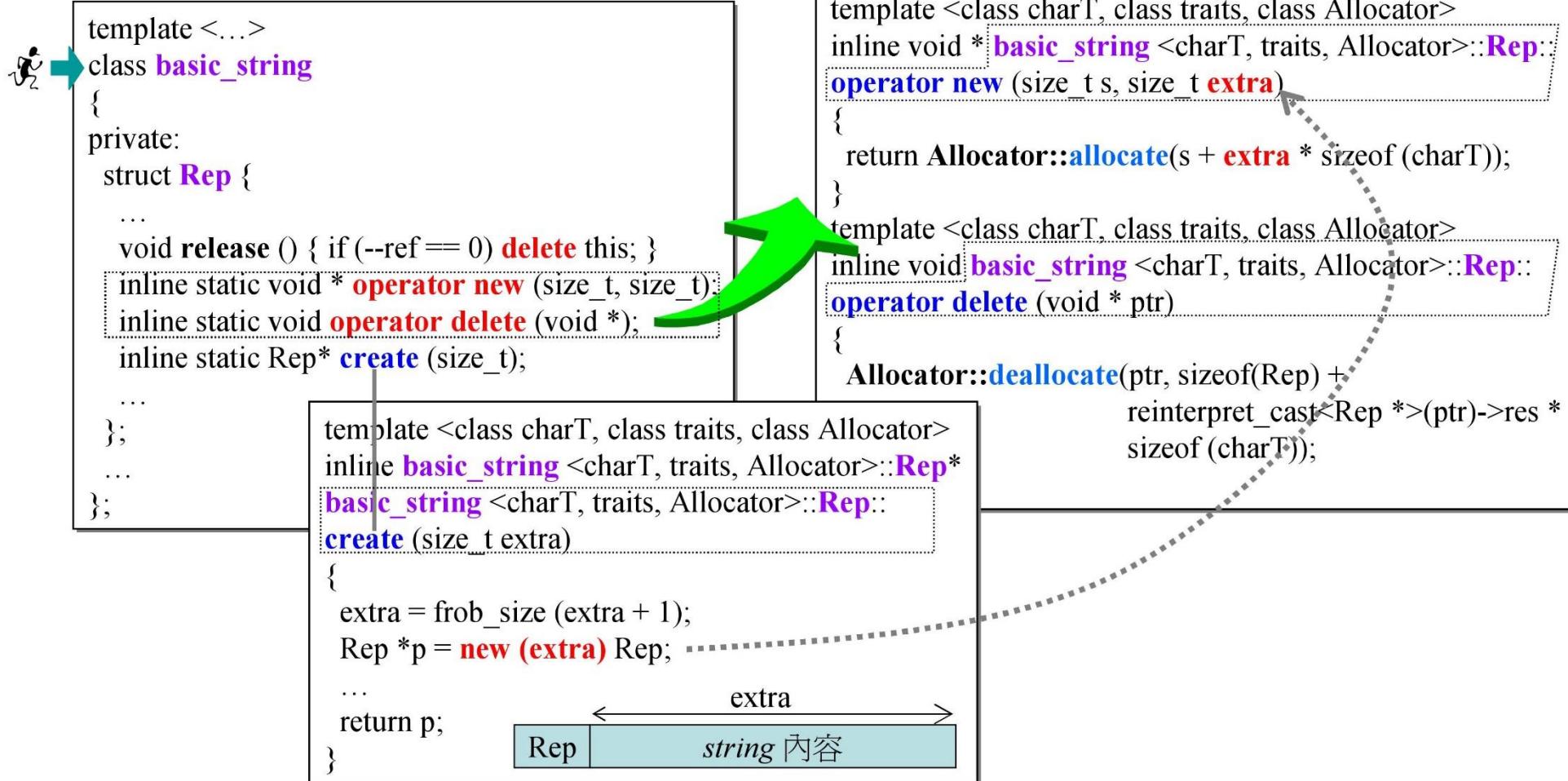
```
Foo::Foo()
① operator new(size_t size), size= 4
Foo::Foo()
② operator new(size_t size, void* start), size= 4  start= 0x22fe8c
Foo::Foo()
③ operator new(size_t size, long extra)  4 100
Foo::Foo()
④ operator new(size_t size, long extra, char init)  4 100 a
A Foo::Foo()
⑤ operator new(size_t size, long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

ctor {

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

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





per-class allocator, 1

ref. C++Primer 3/e, p.765

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

class Screen {
public:
    Screen(int x) : i(x) { }
    int get() { return i; }

    void* operator new(size_t);
    void operator delete(void*, size_t);
    // ...
    // 這種設計會引發多耗用一個
    // next 的疑慮。下頁設計更好
private:
    Screen* next;
    static Screen* freeStore;
    static const int screenChunk;
private:
    int i;
};

Screen* Screen::freeStore = 0;
const int Screen::screenChunk = 24;
```

```
void* Screen::operator new(size_t size)
{
    Screen *p;
    if (!freeStore) {
        //linked list 是空的，所以申請一大塊
        size_t chunk = screenChunk * size;
        freeStore = p =
            reinterpret_cast<Screen*>(new char[chunk]);
        //將一大塊分割片片，當做 linked list 串接起來
        for (; p != &freeStore[screenChunk-1]; ++p)
            p->next = p+1;
        p->next = 0;
    }
    p = freeStore;
    freeStore = freeStore->next;
    return p;
}
```

沒有對應的 **delete**，
但這不算 memory leak.

```
void Screen::operator delete(void *p, size_t)
{
    //將 deleted object 插回 free list 前端
    static_cast<Screen*>(p)->next = freeStore;
    freeStore = static_cast<Screen*>(p);
}
```



per-class allocator, 1

ref. C++Primer 3/e, p.765



```
cout << sizeof(Screen) << endl; //8  
  
size_t const N = 100;  
Screen* p[N];  
  
for (int i=0; i< N; ++i)  
    p[i] = new Screen(i);  
  
//輸出前 10 個 pointers, 比較其間隔  
for (int i=0; i< 10; ++i)  
    cout << p[i] << endl;  
  
for (int i=0; i< N; ++i)  
    delete p[i];
```

左邊是寫了 member operator new/delete 的結果，
右邊是沒寫因而使用 global operator new/delete 的結果

8
0x3e48f0
0x3e48f8
0x3e4900
0x3e4908
0x3e4910
0x3e4918
0x3e4920
0x3e4928
0x3e4930
0x3e4938

間隔8

8
0x3e48e0
0x3e48f0
0x3e4900
0x3e4910
0x3e4920
0x3e4930
0x3e4940
0x3e4950
0x3e4960
0x3e4970

間隔16

per-class allocator, 2

ref. Effective C++ 2e, item10

```
class Airplane {  
private:  
    struct AirplaneRep {  
        unsigned long miles;  
        char type;  
    };  
private:  
    union { AirplaneRep rep; //此欄針對使用中的objects  
            Airplane* next; //此欄針對 free list 上的object  
    };  
public:  
    unsigned long getMiles() { return rep.miles; }  
    char getType() { return rep.type; }  
    void set(unsigned long m, char t) {  
        rep.miles = m; rep.type = t;  
    }  
public:  
    static void* operator new(size_t size);  
    static void operator delete(void* deadObject, size_t size);  
private:  
    static const int BLOCK_SIZE;  
    static Airplane* headOfFreeList;  
};  
  
Airplane* Airplane::headOfFreeList;  
const int Airplane::BLOCK_SIZE = 512;
```

```
void* Airplane::operator new(size_t size)  
{  
    怎會有誤？當繼承發生時  
    //如果大小有誤，轉交給 ::operator new()  
    if (size != sizeof(Airplane))  
        return ::operator new(size);  
  
    Airplane* p = headOfFreeList;  
    if (p) //如果 p 有效，就把 list 頭部下移一個元素  
        headOfFreeList = p->next;  
    else {  
        //free list 已空，申請(分配)一大塊  
        Airplane* newBlock = static_cast<Airplane*>  
            (::operator new(BLOCK_SIZE * sizeof(Airplane)));  
  
        //將小塊串成一個 free list，  
        //但跳過 #0，因它將被傳回做為本次成果  
        for (int i = 1; i < BLOCK_SIZE-1; ++i)  
            newBlock[i].next = &newBlock[i+1];  
        newBlock[BLOCK_SIZE-1].next = 0; //結束 list  
        p = newBlock;  
        headOfFreeList = &newBlock[1];  
    }  
    return p;  
}
```



7



per-class allocator, 2

ref. Effective C++ 2e, item10

```
// operator delete 接獲一個內存塊  
// 如果大小正確，就把它加到 free list 前端  
void Airplane::operator delete(void* deadObject,  
                                size_t size)  
{  
    if (deadObject == 0) return;  
    if (size != sizeof(Airplane)) {  
        ::operator delete(deadObject);  
        return;  
    }  
  
    Airplane* carcass =  
        static_cast<Airplane*>(deadObject);  
  
    carcass->next = headOfFreeList;  
    headOfFreeList = carcass;  
}
```

```
8  
0x3e4ce0 A 1000  
0x3e4d00 B 2000  
0x3e4d20 C 5000000  
0x3e4cd8  
0x3e4ce0  
0x3e4ce8  
0x3e4cf0  
0x3e4cf8 間隔8  
0x3e4d00  
0x3e4d08  
0x3e4d10  
0x3e4d18  
0x3e4d20
```



```
cout << sizeof(Airplane) << endl;  
size_t const N = 100;  
Airplane* p[N];  
  
for (int i=0; i< N; ++i)  
    p[i] = new Airplane;
```

//隨機測試 object 正常否

```
p[1]->set(1000,'A');  
p[5]->set(2000,'B');  
p[9]->set(500000,'C');  
...
```

//輸出前 10 個 pointers,
//用以比較其間隔

```
for (int i=0; i< 10; ++i)  
    cout << p[i] << endl;  
  
for (int i=0; i< N; ++i)  
    delete p[i];
```

```
8  
0x3e4900 A 1000  
0x3e4930 B 2000  
0x3e4970 C 5000000  
0x3e4910  
0x3e4900  
0x3e48f0  
0x3e48e0  
0x3e4920 間隔16  
0x3e4930  
0x3e4940  
0x3e4950  
0x3e4960  
0x3e4970
```

左邊是寫了 member operator new/delete 的結果，
右邊是沒寫因而使用 global operator new/delete 的結果



static allocator

當你受困於必須為不同的 classes 重寫一遍幾乎相同的 member operator new 和 member operator delete 時，應該有方法將一個總是分配特定尺寸之區塊的 memory allocator 概念包裝起來，使它容易被重複使用。以下展示一種作法，每個 allocator object 都是個分配器，它體內維護一個 free-lists；不同的 allocator objects 維護不同的 free-lists。

```
class allocator
{
private:
    struct obj {
        struct obj* next; //embedded pointer
    };
public:
    void* allocate(size_t);
    void deallocate(void*, size_t);
private:
    obj* freeStore = nullptr;
    const int CHUNK = 5; //小一些以便觀察
};
```

```
void
allocator::deallocate(void* p, size_t)
{
    //將 *p 收回插入 free list 前端
    ((obj*)p)->next = freeStore;
    freeStore = (obj*)p;
}
```

```
void* allocator::allocate(size_t size)
{
    obj* p;
    if (!freeStore) {
        //linked list 為空，於是申請一大塊
        size_t chunk = CHUNK * size;
        freeStore = p = (obj*)malloc(chunk);
        //將分配得來的一大塊當做 linked list 般,
        //小塊小塊串接起來
        for (int i=0; i < (CHUNK-1); ++i) {
            p->next = (obj*)((char*)p + size);
            p = p->next;
        }
        p->next = nullptr; //last
    }
    p = freeStore;
    freeStore = freeStore->next;
    return p;
}
```



static allocator

寫法
十分
制式

```
class Foo {
public:
    long L;
    string str;
    static allocator myAlloc;
public:
    Foo(long l) : L(l) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Foo::myAlloc;
```

```
class Goo {
public:
    complex<double> c;
    string str;
    static allocator myAlloc;
public:
    Goo(const complex<double>& x) : c(x) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Goo::myAlloc;
```

這比先前的設計乾淨多了，application classes 不再與內存分配細節糾纏不清，所有相關細節都讓 allocator 去操心，我們的工作是讓 application classes 正確運作。



static allocator, 示例與結果

```
Foo* p[100];

cout << "sizeof(Foo)= " << sizeof(Foo) << endl;
for (int i=0; i<23; ++i) { //隨意看看結果
    p[i] = new Foo(i);
    cout << p[i] << ' ' << p[i]->L << endl;
}

for (int i=0; i<23; ++i) {
    delete p[i];
}
```

```
Goo* p[100];

cout << "sizeof(Goo)= " << sizeof(Goo) << endl;
for (int i=0; i<17; ++i) { //隨意看看結果
    p[i] = new Goo(complex<double>(i,i));
    cout << p[i] << ' ' << p[i]->c << endl;
}

for (int i=0; i<17; ++i) {
    delete p[i];
}
```



```
sizeof(Goo)= 24
0x3e6200 <0,0>
0x3e6218 <1,1>
0x3e6230 <2,2>
0x3e6248 <3,3>
0x3e6260 <4,4>
0x3e6498 <5,5>
0x3e64b0 <6,6>
0x3e64c8 <7,7>
0x3e64e0 <8,8>
0x3e64f8 <9,9>
0x3e6518 <10,10>
0x3e6530 <11,11>
0x3e6548 <12,12>
0x3e6560 <13,13>
0x3e6578 <14,14>
0x3e6598 <15,15>
0x3e65b0 <16,16>
```

```
sizeof(Foo)= 8
0x3e5f90 0
0x3e5f98 1
0x3e5fa0 2
0x3e5fa8 3
0x3e5fb0 4
0x3e5d40 5
0x3e5d48 6
0x3e5d50 7
0x3e5d58 8
0x3e5d60 9
0x3e6080 10
0x3e6088 11
0x3e6090 12
0x3e6098 13
0x3e60a0 14
0x3e6128 15
0x3e6130 16
0x3e6138 17
0x3e6140 18
0x3e6148 19
0x3e61d0 20
0x3e61d8 21
0x3e61e0 22
```



macro for static allocator

```
class Foo {
public:
    long L;
    string str;
    static allocator myAlloc;
public:
    Foo(long l) : L(l) { }
    static void* operator new(size_t size)
    { return myAlloc.allocate(size); }
    static void operator delete(void* pdead, size_t size)
    { return myAlloc.deallocate(pdead, size); }
};
allocator Foo::myAlloc;
```

寫法
十分
制式

```
// DECLARE_POOL_ALLOC -- used in class definition
#define DECLARE_POOL_ALLOC()
public:
```

```
    void* operator new(size_t size) { return myAlloc.allocate(size); } \
    void operator delete(void* p) { myAlloc.deallocate(p, 0); } \
```

protected:

```
    static allocator myAlloc;
```

```
// IMPLEMENT_POOL_ALLOC -- used in class implementation file
#define IMPLEMENT_POOL_ALLOC(class_name) \
allocator class_name::myAlloc;
```

```
class Foo {
    DECLARE_POOL_ALLOC()
public:
    long L;
    string str;
public:
    Foo(long l) : L(l) { }
};
IMPLEMENT_POOL_ALLOC(Foo)
```

```
class Goo {
    DECLARE_POOL_ALLOC()
public:
    complex<double> c;
    string str;
public:
    Goo(const complex<double>& x) : c(x) { }
};
IMPLEMENT_POOL_ALLOC(Goo)
```



macro for static allocator,示例與結果

```
Foo* pF[100];
Goo* pG[100];

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

for (int i=0; i<23; ++i) { //隨意看看結果
    pF[i] = new Foo(i);
    pG[i] = new Goo(complex<double>(i,i));

    cout << pF[i] << ' ' << pF[i]->L << '\t';
    cout << pG[i] << ' ' << pG[i]->c << '\n';
}

for (int i=0; i<23; ++i) {
    delete pF[i];
    delete pG[i];
}
```

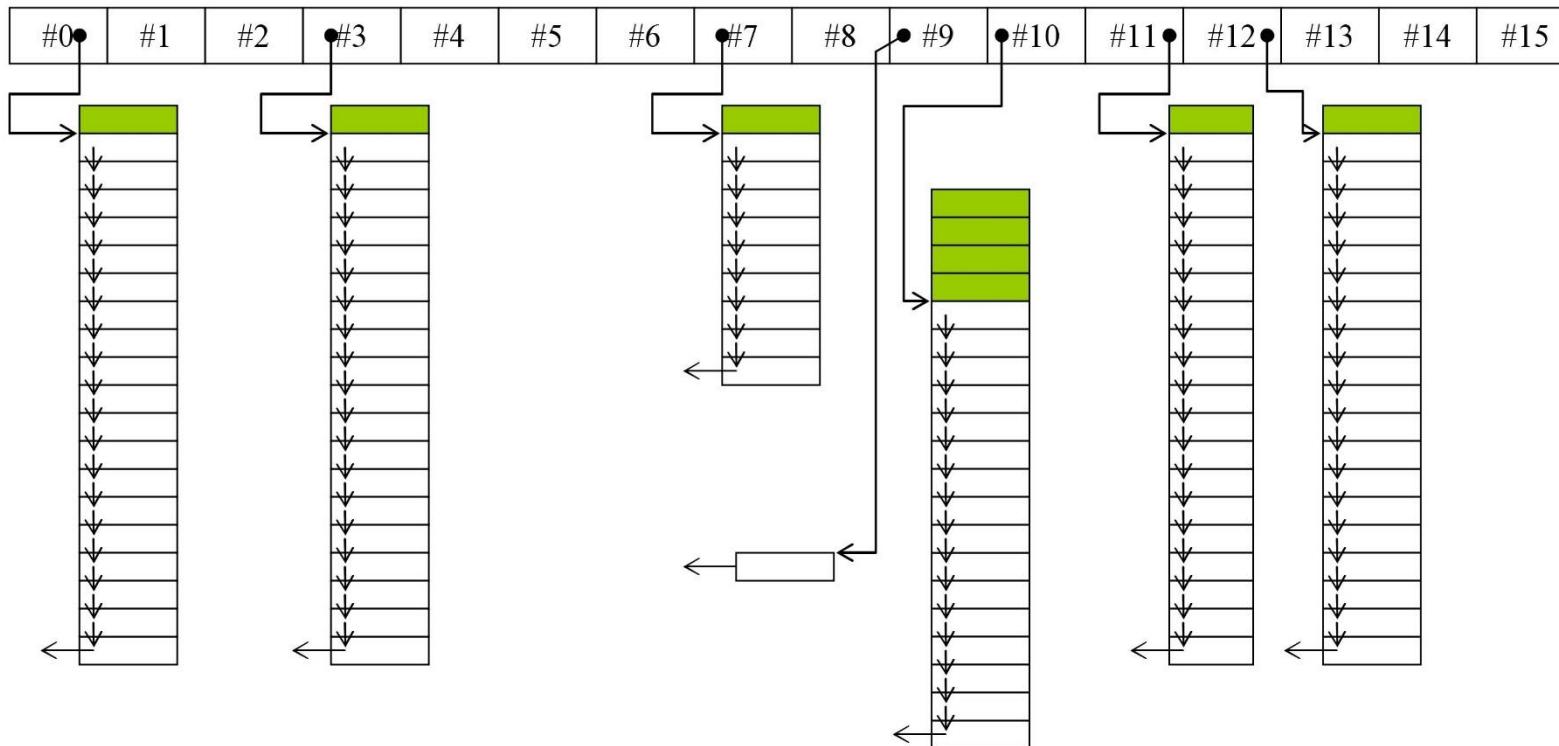


sizeof(Foo)= 8
sizeof(Goo)= 24
0x3e5fc0 0 0x3e6630 <0,0>
0x3e5fc8 1 0x3e6648 <1,1>
0x3e5fd0 2 0x3e6660 <2,2>
0x3e5fd8 3 0x3e6678 <3,3>
0x3e5fe0 4 0x3e6690 <4,4>
0x3e6158 5 0x3e66b0 <5,5>
0x3e6160 6 0x3e66c8 <6,6>
0x3e6168 7 0x3e66e0 <7,7>
0x3e6170 8 0x3e66f8 <8,8>
0x3e6178 9 0x3e6710 <9,9>
0x3e67d8 10 0x3e6808 <10,10>
0x3e67e0 11 0x3e6820 <11,11>
0x3e67e8 12 0x3e6838 <12,12>
0x3e67f0 13 0x3e6850 <13,13>
0x3e67f8 14 0x3e6868 <14,14>
0x3e6978 15 0x3e69a8 <15,15>
0x3e6980 16 0x3e69c0 <16,16>
0x3e6988 17 0x3e69d8 <17,17>
0x3e6990 18 0x3e69f0 <18,18>
0x3e6998 19 0x3e6a08 <19,19>
0x3e6b18 20 0x3e6b48 <20,20>
0x3e6b20 21 0x3e6b60 <21,21>
0x3e6b28 22 0x3e6b78 <22,22>



global allocator (with multiple free-lists)

將前述 allocator 進一步發展為具備 16 條 free-lists，並因此不再以 application classes 內的 static 呈現，而是一個 global allocator —— 這就是 G2.9 的 `std::alloc` 的雛形。



—侯捷—



new handler

當 operator new 沒能力為你分配出你所申請的 memory，會拋一個 std::bad_alloc exception。某些老舊編譯器則是返回 0—你仍然可以令編譯器那麼做：

```
new (nothrow) Foo;
```

此稱為 nothrow 形式。

拋出 exception 之前會先（不只一次）調用一個可由 client 指定的 handler，以下是 new handler 的形式和設定方法：

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

設計良好的 new handler 只有兩個選擇：

- 讓更多 memory 可用
- 調用 `abort()` 或 `exit()`

... \vc98\crt\src\newop2.cpp

```
void* operator new(size_t size, const std::nothrow_t&)
    _THROW0()
{
    // try to allocate size bytes
    void *p;
    while ((p = malloc(size)) == 0)
        { // buy more memory or return null pointer
            _TRY_BEGIN
                if (_callnewh(size) == 0) break;
            _CATCH(std::bad_alloc) return (0);
            _CATCH_END
        }
    return (p);
}
```



new handler

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

void noMoreMemory()
{
    cerr << "out of memory";
    abort();
}

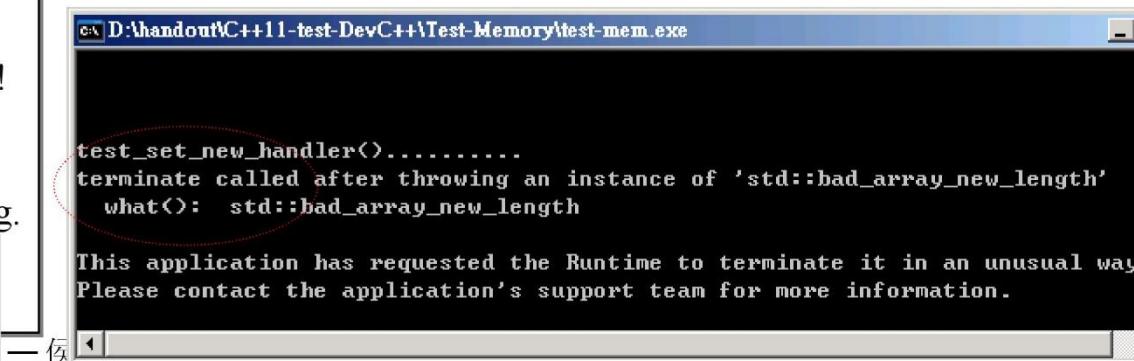
void main()
{
    set_new_handler(noMoreMemory);

    int* p = new int[1000000000000000]; //well, so BIG!
    assert(p);

    p = new int[1000000000000000]; //GCC warning.
    assert( // 執行結果：
           // BCB4 版會出現 out of memory.
           //           Abnormal program termination
           // 非常好，符合預期。);
}
```

msdev\vc98\crt\src\setnewh.cpp

```
new_handler
__cdecl set_new_handler(
    new_handler new_p)
{
    // cannot use stub to register a new handler
    assert(new_p == 0);
    // remove current handler
    _set_new_handler(0);
    return 0;
}
```



本例之 new handler 中若無調用 abort()，執行後 cerr 會不斷出現 “out of memory”，需強制中斷。這樣的表現是正確的，表示當 operator new 無法滿足申請量時，會不斷調用 new handler 直到獲得足夠 memory.



=default, =delete

```
class Foo {  
public:  
    Foo() = default;  
    Foo(const Foo&) = delete;  
    Foo& operator=(const Foo&) = delete;  
    ~Foo() = default;  
    ...  
};
```

it is not only for constructors and assignments,
but also applies to operator new/new[],operator
delete/delete[] and their overloads.

==== =default, =delete

```
class Foo {  
public:  
    long _x;  
public:  
    Foo(long x=0) : _x(x) { }  
};
```

✗ static void* operator new(size_t size) = **default**; [Error] cannot be defaulted
✗ static void operator delete(void* pdead, size_t size) = **default**;
 static void* operator new[](size_t size) = **delete**;
 static void operator delete[](void* pdead, size_t size) = **delete**;

```
class Goo {  
public:  
    long _x;  
public:  
    Goo(long x=0) : _x(x) { }  
  
    static void* operator new(size_t size) = delete;  
    static void operator delete(void* pdead, size_t size) = delete;  
};
```

[Error] use of deleted function ...

✗ Foo* p1 = new Foo(5);
 delete p1;
✗ Foo* pF = new Foo[10];
 delete [] pF;

✗ Goo* p2 = new Goo(7);
✗ delete p2;
Goo* pG = new Goo[10];
delete [] pG;



The End

內存管理

從平地到萬丈高樓

Memory Management 101

第一講 primitives

第二講 std::allocator

第三講 malloc/free

第四講 loki::allocator

第五講 other issues

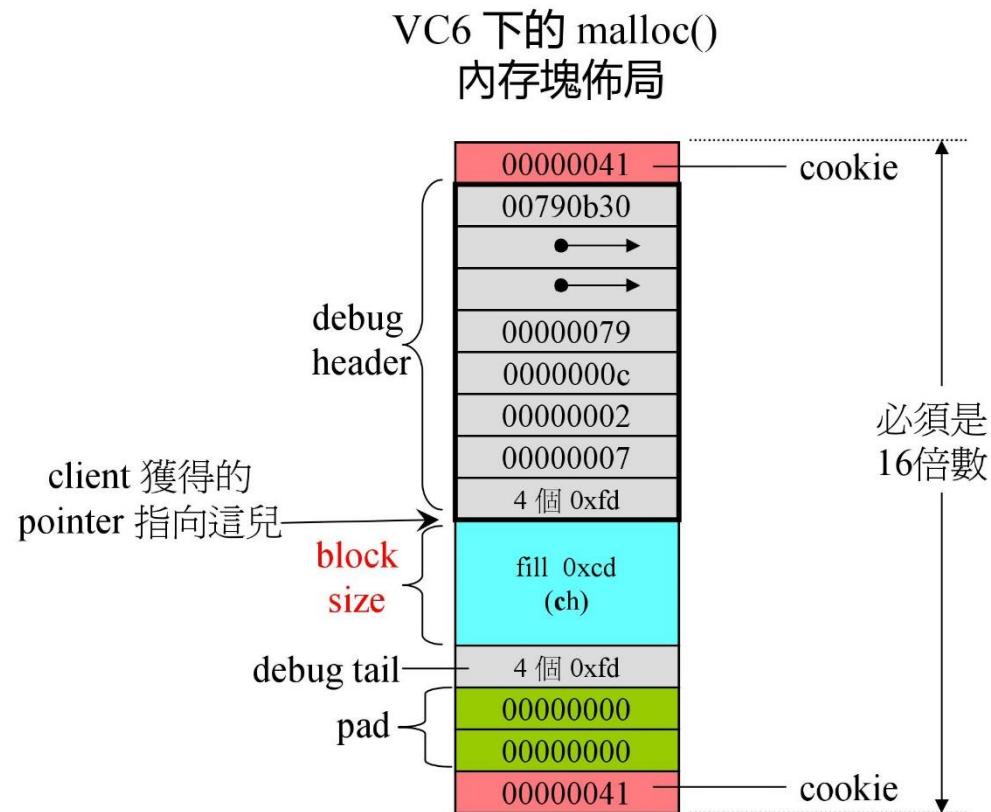


侯捷

西北有高樓
上與浮雲齊



VC6 malloc()



$$0xC + (32+4) + 4 * 2$$

→ 0x38

→ 0x40

— 侯捷 —



VC6 標準分配器之實現

VC6 所附的標準庫，其 std::allocator 實現如下 (<xmemory>)

```
template<class _Ty>
class allocator {
public:
    typedef _SIZT size_type;
    typedef _PDFT difference_type;
    typedef _Ty _FARQ *pointer;
    typedef _Ty value_type;
    pointer allocate(size_type _N, const void *)
    { return (_Allocate((difference_type)_N, (pointer)0)); }
    void deallocate(void _FARQ * _P, size_type)
    { operator delete(_P); }
};
```

```
#ifndef _FARQ
#define _FARQ
#define _PDFT ptrdiff_t
#define _SIZT size_t
#endif
```

VC6+ 的 allocator 只是以 ::operator new 和 ::operator delete 完成 allocate() 和 deallocate()，沒有任何特殊設計。

其中用到的 _Allocate() 定義如下：

```
template<class _Ty> inline
_Ty _FARQ *_Allocate(_PDFT _N, _Ty _FARQ *)
{ if (_N < 0) _N = 0;
  return ((_Ty _FARQ*) operator new((_SIZT)_N * sizeof(_Ty))); }
```

```
//分配 512 ints.
int* p = allocator<int>().allocate(512, (int*)0);
allocator<int>().deallocate(p, 512);
```

```
template<class _Ty,
        class _A = allocator<_Ty>>
class vector
{ ...};
```

```
template<class _Ty,
        class _A = allocator<_Ty>>
class list
{ ...};
```

```
template<class _Ty,
        class _A = allocator<_Ty>>
class deque
{ ...};
```

```
template<class _K,
        class _Pr = less<_K>,
        class _A = allocator<_K>>
class set { ...};
```



BC5 標準分配器之實現

BC5 所附的標準庫，其 std::allocator 實現如下 (<memory.stl>)

```
template <class T>
class allocator
{
public:
    typedef size_t           size_type;
    typedef ptrdiff_t        difference_type;
    typedef T*               pointer;
    typedef T                value_type;

    pointer allocate(size_type n,
                      allocator<void>::const_pointer = 0) {
        pointer tmp =
            _RWSTD_STATIC_CAST(pointer, (::operator new
                (_RWSTD_STATIC_CAST(size_t, (n*sizeof(value_type))))));
        _RWSTD_THROW_NO_MSG(tmp == 0, bad_alloc);
        return tmp;
    }
    void deallocate(pointer p, size_type) {
        ::operator delete(p);
    }
    ...
};
```

BC5 的 allocator 只是以 ::operator new 和 ::operator delete 完成 allocate() 和 deallocate()，沒有任何特殊設計。

```
template <class T,
          class Allocator=allocator<T>>
class vector
{ ...};

template <class T,
          class Allocator=allocator<T>>
class list
{ ...};

template <class T,
          class Allocator=allocator<T>>
class deque
{ ...};
```



//分配 512 ints.
int* p = allocator<int>().allocate(512);
allocator<int>().deallocate(p,512);

■■■ G2.9 標準分配器之實現

G2.9 所附的標準庫，其 std::allocator 實現如下 (<defalloc.h>)

G2.9 的 allocator 只是以 ::operator new 和 ::operator delete 完成 allocate() 和 deallocate()，沒有任何特殊設計。

```
template <class T>
class allocator {
public:
    typedef T      value_type;
    typedef T*     pointer;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    pointer allocate(size_type n) {
        return ::allocate((difference_type)n, (pointer)0);
    }
    void deallocate(pointer p) { ::deallocate(p); }
};
```

```
template <class T>
inline T* allocate(ptrdiff_t size, T*) {
    set_new_handler(0);
    T* tmp = (T*)
        (::operator new((size_t)(size*sizeof(T))));
    if (tmp == 0) {
        cerr << "out of memory" << endl;
        exit(1);
    }
    return tmp;
}
template <class T>
inline void deallocate(T* buffer)
    ::operator delete(buffer);
}
```

G++ <defalloc.h> 中有這樣的註釋：**DO NOT USE THIS FILE** unless you have an old container implementation that requires an allocator with the HP-style interface. **SGI STL uses a different allocator interface.** SGI-style allocators are not parametrized with respect to the object type; they traffic in void* pointers. **This file is not included by any other SGI STL header.**



G2.9 容器使用的分配器, 不是 std::allocator 而是 std::alloc



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

```
template <class T,  
         class Alloc = alloc>  
class list {  
    ...  
};
```

```
template <class T,  
         class Alloc = alloc,  
         size_t BufSiz = 0>  
class deque {  
    ...  
};
```



```
//分配 512 bytes  
void* p = alloc::allocate(512);  
//也可以這樣alloc().allocate(512);  
alloc::deallocate(p,512);
```

```
template <class Key,  
         class T,  
         class Compare = less<Key>,  
         class Alloc = alloc>  
class map {  
    ...  
};
```

```
template <class Key,  
         class Compare = less<Key>,  
         class Alloc = alloc>  
class set {  
    ...  
};
```



而今安在哉？ std::alloc vs. __pool_alloc

```

class __pool_alloc_base
{
protected:
    enum { __S_align = 8 };
    enum { __S_max_bytes = 128 };
    enum { __S_free_list_size = (size_t) __S_max_bytes / (size_t) __S_align };

    union __Obj
    {
        union __Obj* __M_free_list_link;
        char __M_client_data[1]; // The client sees this.
    };

    static __Obj* volatile __S_free_list[__S_free_list_size];
    // Chunk allocation state.

    static char* __S_start_free;
    static char* __S_end_free;
    static size_t __S_heap_size;
    ...
}

template<typename _Tp>
class __pool_alloc : private __pool_alloc_base
{ ... };

```

— 侯捷 —

G2.9

用例：
`vector<string> vec;`

G4.9

```

classDiagram
    class __pool_alloc_base {
        protected:
            enum { __S_align = 8 };
            enum { __S_max_bytes = 128 };
            enum { __S_free_list_size = (size_t) __S_max_bytes / (size_t) __S_align };

            union __Obj {
                union __Obj* __M_free_list_link;
                char __M_client_data[1]; // The client sees this.
            };

            static __Obj* volatile __S_free_list[__S_free_list_size];
            // Chunk allocation state.

            static char* __S_start_free;
            static char* __S_end_free;
            static size_t __S_heap_size;
            ...
        }

        template<typename _Tp>
        class __pool_alloc : private __pool_alloc_base
        { ... };
    }

```

G4.9 標準庫中有許多 extented allocators,
其中 `__pool_alloc` 就是 G2.9 `alloc` 的化身.

G2.9

```

enum { __ALIGN = 8 };
enum { __MAX_BYTES = 128 };
enum { __NFREELISTS = __MAX_BYTES/__ALIGN};

template <bool threads, int inst>
class __default_alloc_template
{
private:
    typedef __default_alloc_template<false,0> alloc;

    union obj {
        union obj* free_list_link;
    };

    static obj* volatile free_list[__NFREELISTS];
    ...
    // Chunk allocation state.

    static char* start_free;
    static char* end_free;
    static size_t heap_size;
    ...
}

```



而今安在哉？ std::alloc vs. __pool_alloc

```
// Allocate memory in large chunks in order to avoid fragmenting the
// heap too much. Assume that __n is properly aligned. We hold the
// allocation lock.
char*
__pool_alloc_base::__M_allocate_chunk(size_t __n, int& __nobjs)
{
    char* __result;
    size_t __total_bytes = __n * __nobjs;
    size_t __bytes_left = __S_end_free - __S_start_free;
    ...
    __try {
        __S_start_free = static_cast<char*>(__operator new(__bytes_to_get));
    }
    __catch(const std::bad_alloc&){
        // Try to make do with what we have. That can't hurt. We
        // do not try smaller requests, since that tends to result
        // in disaster on multi-process machines.
        size_t __i = __n;
        for(; __i <= (size_t) __S_max_bytes; __i += (size_t) __S_align) {
            ...
        }
        // What we have wasn't enough. Rethrow.
        __S_start_free = __S_end_free = 0; // We have no chunk.
        __throw_exception_again;
    }
    __S_heap_size += __bytes_to_get;
    __S_end_free = __S_start_free + __bytes_to_get;
    return __M_allocate_chunk(__n, __nobjs);
}
```

G4.9

G4.9 標準庫中有許多 extented allocators,
其中 `__pool_alloc` 就是 G2.9 的 `alloc` 的化身.

```
template <bool threads, int inst>
char* __default_alloc_template<threads, inst>::
chunk_alloc(size_t size, int& nobjs)
{
    char* result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free;
    ...
    start_free = (char*)malloc(bytes_to_get);
    if(0 == start_free) {
        int i;
        obj* volatile *my_free_list, *p;

        //Try to make do with what we have. That can't
        //hurt. We do not try smaller requests, since that tends
        //to result in disaster on multi-process machines.
        for(i = size; i <= __MAX_BYTES; i += __ALIGN) {
            ...
        }
        end_free = 0; //In case of exception.
        start_free = (char*)malloc_alloc::allocate(bytes_to_get);
        //This should either throw an exception or
        //remedy the situation. Thus we assume it
        //succeeded.
    }
    heap_size += bytes_to_get;
    end_free = start_free + bytes_to_get;
    return(chunk_alloc(size, nobjs));
}
```

G2.9



G4.9 標準分配器之實現

```
template<typename _Tp>      <.../bits/new_allocator.h>
class new_allocator
{
...
pointer allocate(size_type __n, const void* = 0) {
    if (__n > this->max_size())
        std::__throw_bad_alloc();
    return static_cast<_Tp*>(
        ::operator new(__n * sizeof(_Tp)));
}

void deallocate(pointer __p, size_type)
{ ::operator delete(__p); }
...
};

# define __allocator_base __gnu_cxx::new_allocator
```

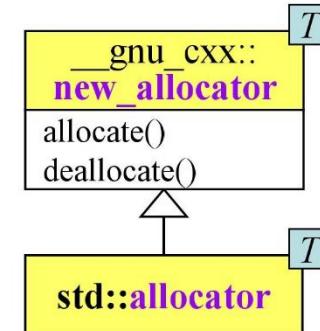
```
template<typename _Tp>      <.../bits/allocator.h>
class allocator: public __allocator_base<_Tp>
{
...
};
```

— 侯捷 —

```
template<typename _Tp,
         typename _Alloc = std::allocator<_Tp>>
class vector : protected _Vector_base<_Tp, _Alloc>
{ ... };

template<typename _Tp,
         typename _Alloc = std::allocator<_Tp>>
class list : protected _List_base<_Tp, _Alloc>
{ ... };

template<typename _Tp,
         typename _Alloc = std::allocator<_Tp>>
class deque : protected _Deque_base<_Tp, _Alloc>
{ ... };
```



■■■ G4.9 pool allocator 用例

```
//欲使用 std::allocator 以外的 allocator, 就得自行 #include <ext/...>
#include <ext/pool_allocator.h>
```

```
template<typename Alloc>
void cookie_test(Alloc alloc, size_t n)
{
    typename Alloc::value_type *p1, *p2, *p3;
    p1 = alloc.allocate(n);
    p2 = alloc.allocate(n);
    p3 = alloc.allocate(n);
```

```
cout << "p1= " << p1 << '\t' << "p2= " << p2 << '\t' << "p3= " << p3 << '\n';
```

```
alloc.deallocate(p1,sizeof(typename Alloc::value_type));
alloc.deallocate(p2,sizeof(typename Alloc::value_type));
alloc.deallocate(p3,sizeof(typename Alloc::value_type));
```

```
}
```



```
cout << sizeof(__gnu_cxx::__pool_alloc<int>) << endl; //1
vector<int, __gnu_cxx::__pool_alloc<int>> vecPool;
cookie_test(__gnu_cxx::__pool_alloc<double>(), 1);
//相距 08h (表示不帶 cookie)
```



```
p1= 0xae4138 p2= 0xae4140 p3= 0xae4148
```

```
p1= 0xae25e8 p2= 0xaddeb50 p3= 0xadd090
```

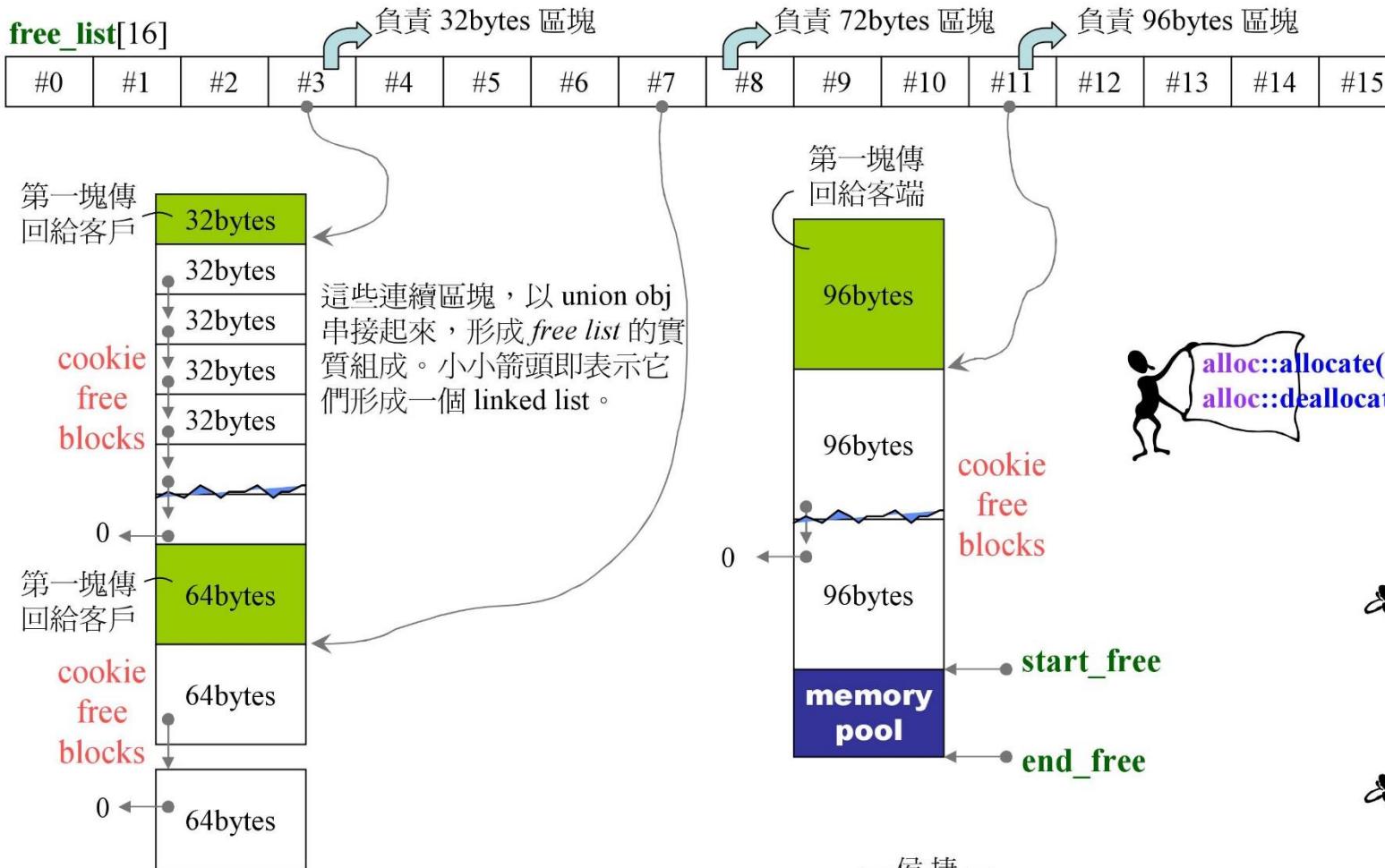


```
cout << sizeof(std::allocator<int>) << endl; //1
vector<int, std::allocator<int>> vec;
cookie_test(std::allocator<double>(), 1);
//相距 10h (表示帶 cookie)
```



```
p1= 0x3e4098 p2= 0x3e4088 p3= 0x3e4078
```

G2.9 std::alloc 運行模式



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

template <class T,
         class Alloc = alloc>
class list { ... };

template <class T,
         class Alloc = alloc,
         size_t BufSiz = 0>
class deque { ... };

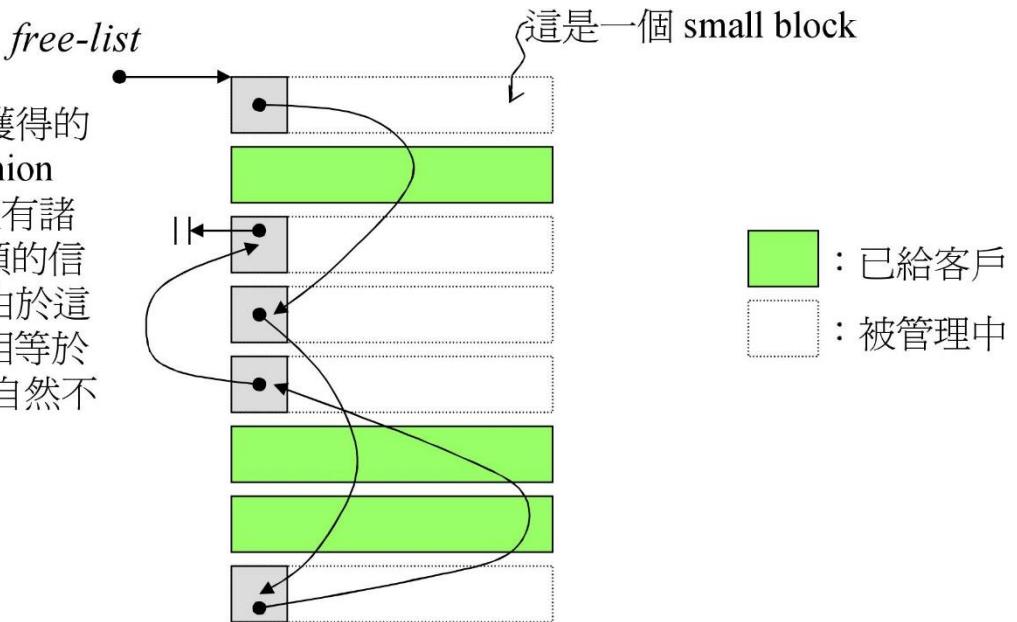
template <class Key,
         class T,
         class Compare = less<Key>,
         class Alloc = alloc>
class map { ... };

template <class Key,
         class Compare = less<Key>,
         class Alloc = alloc>
class set { ... };
```



embedded pointers

當客戶端獲得小區塊，獲得的即是 `char*`（指向某個 `union obj`）。此時雖然客戶端沒有諸如 `LString` 或 `ZString` 之類的信息可得知區塊大小，但由於這區塊是給 `object` 所用，相等於 `object` 大小，`object ctor` 自然不會逾份。



改用 struct
可

```
union obj {  
    union obj* free_list_link;  
    char client_data[1]; // client sees this  
};
```



G2.9 std::alloc 運行一瞥.01

free_list[16]

#0 #1 #2 #3 #4 #5 #6 #7 #8 #9 #10 #11 #12 #13 #14 #15

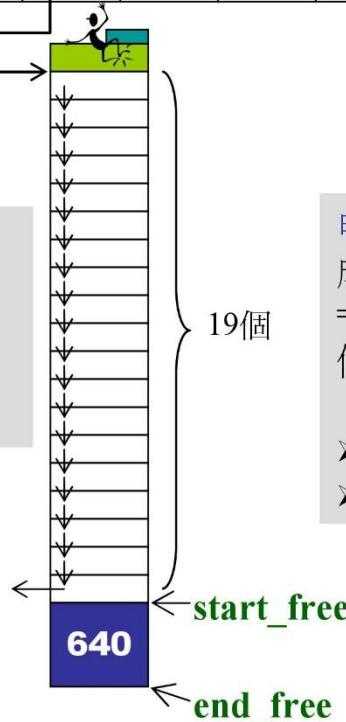


G2.9 std::alloc 運行一瞥.02

free_list[16]

#0	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15
----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

設計：
從 pool 中切割出來準備掛上 free list 的區塊，數量永遠在 1~20 之間

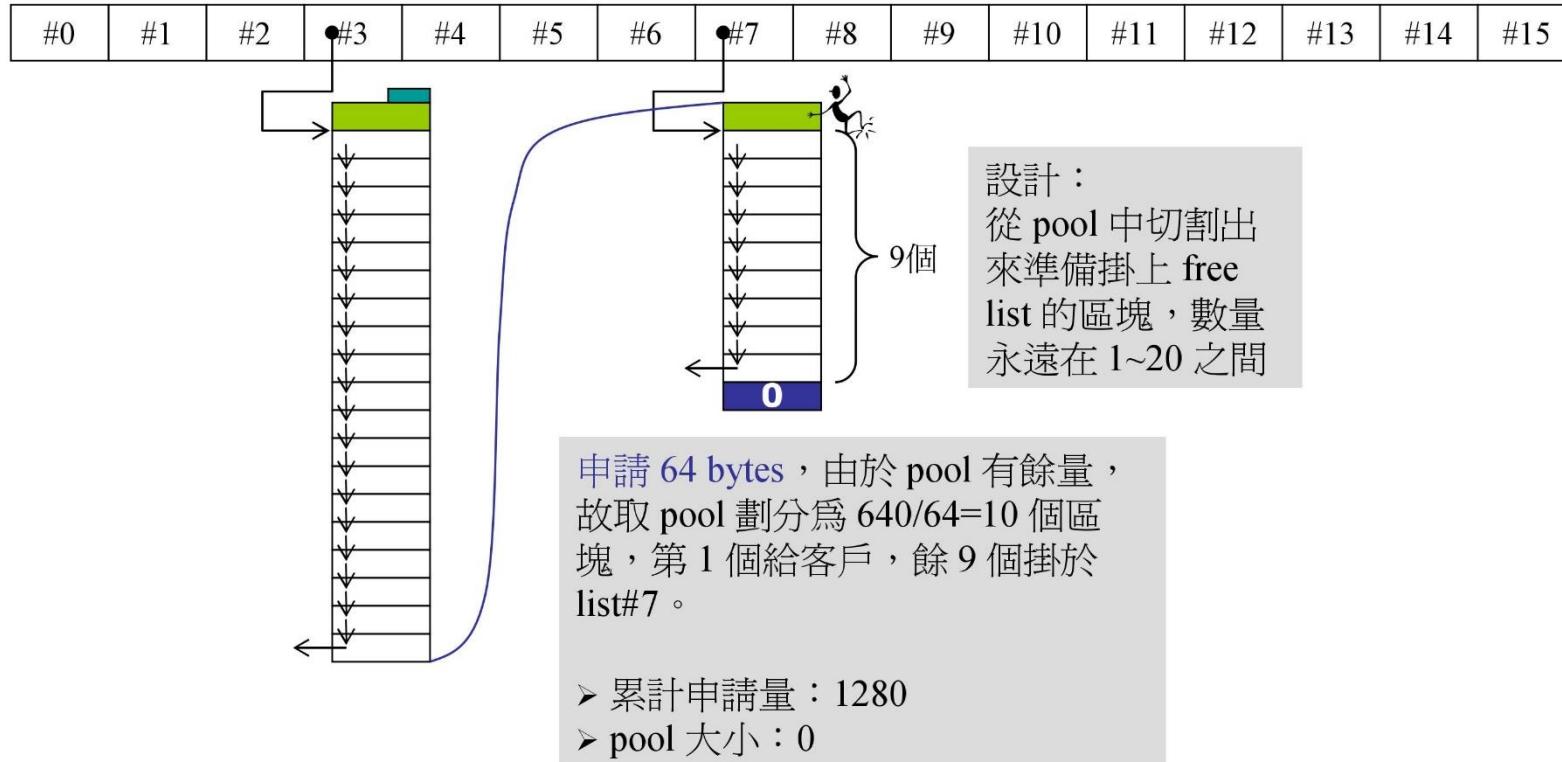


申請 32 bytes，由於 pool 為空，故索取並成功向 pool 揪注 $32*20*2+RoundUp(0>>4) = 1280$ ，從中切出 1 個區塊返給客戶，19 個區塊給 list#3，餘 640 備用。

- 累計申請量：1280
- pool 大小：640

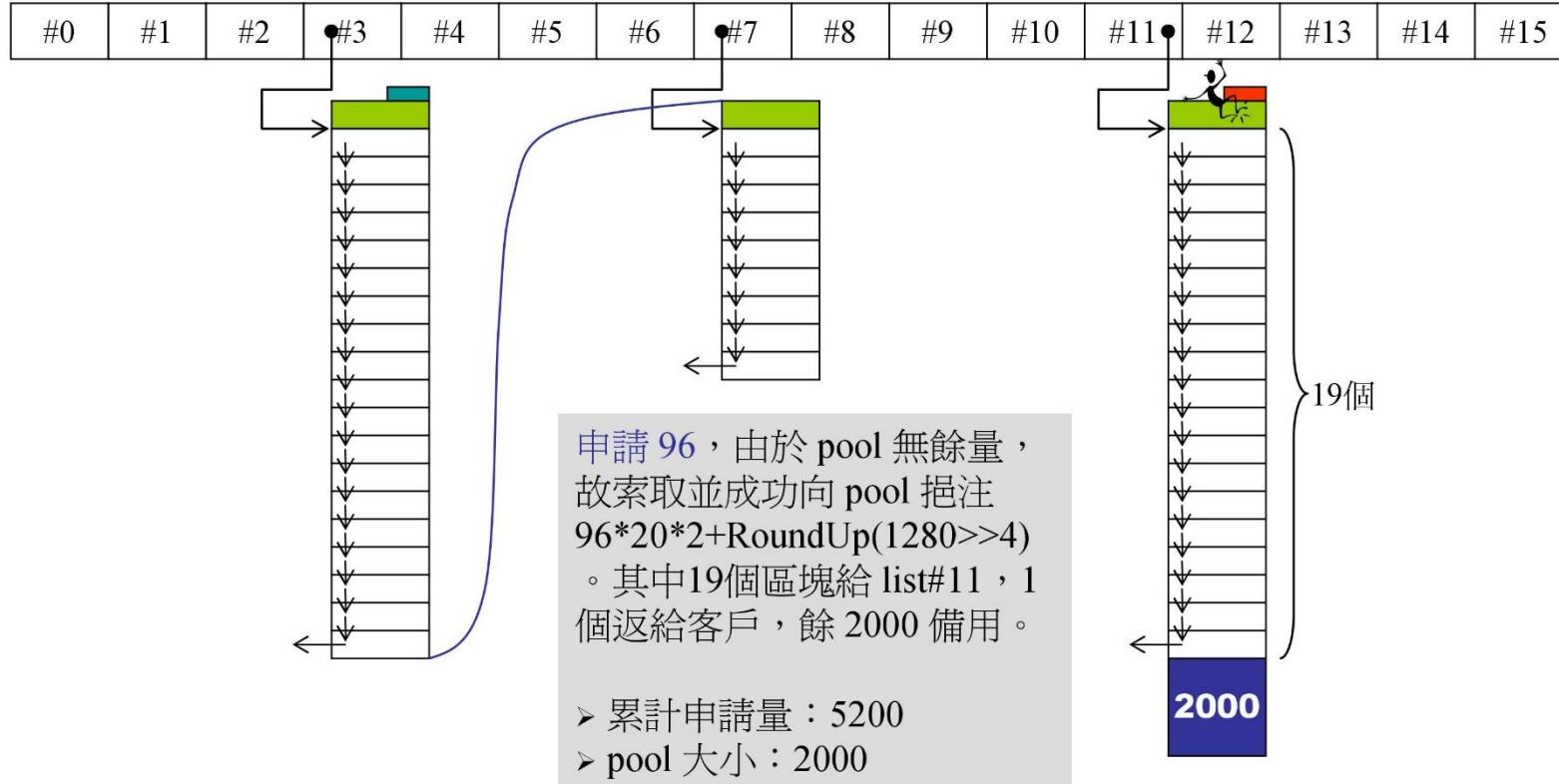


G2.9 std::alloc 運行一瞥.03



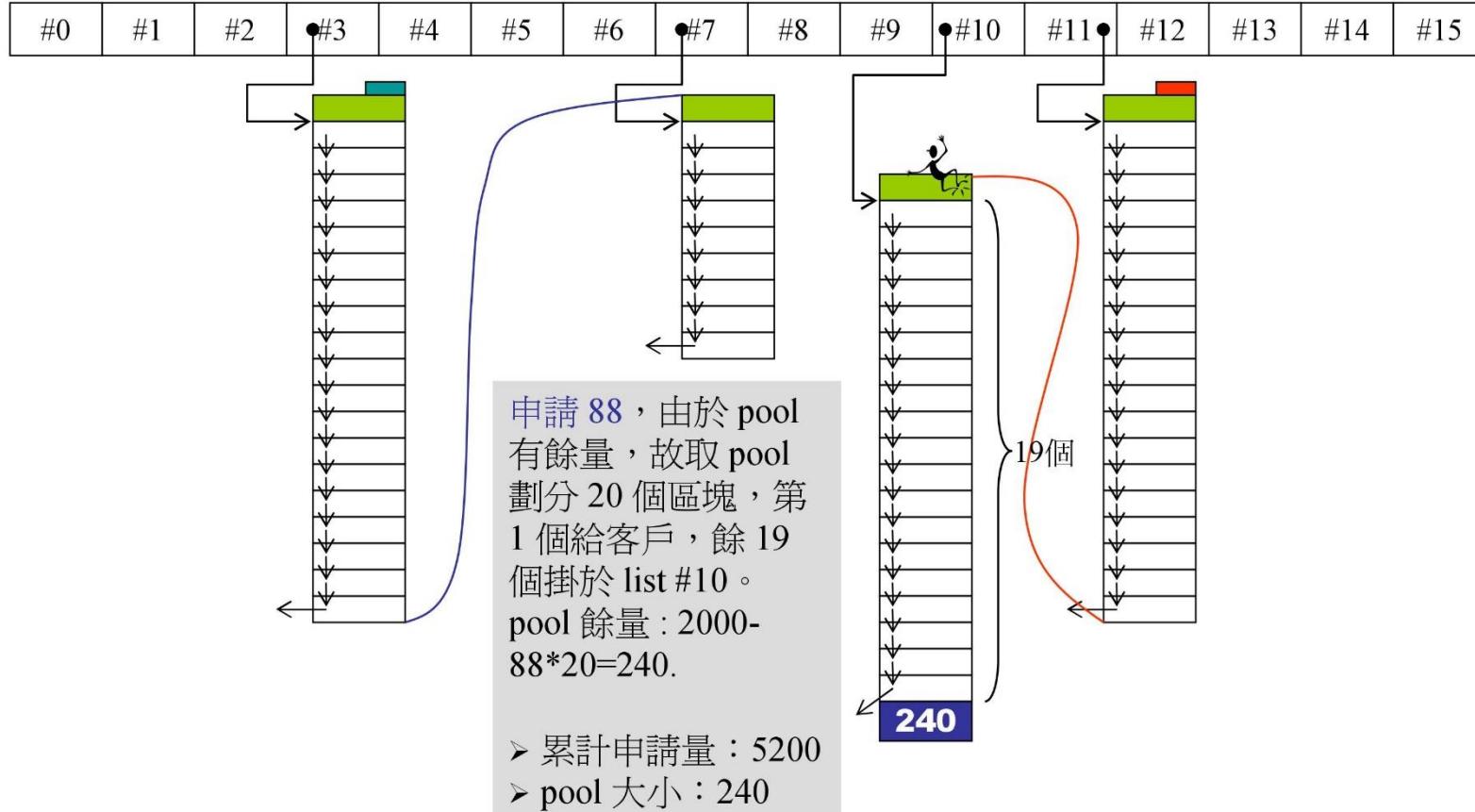


G2.9 std::alloc 運行一瞥.04



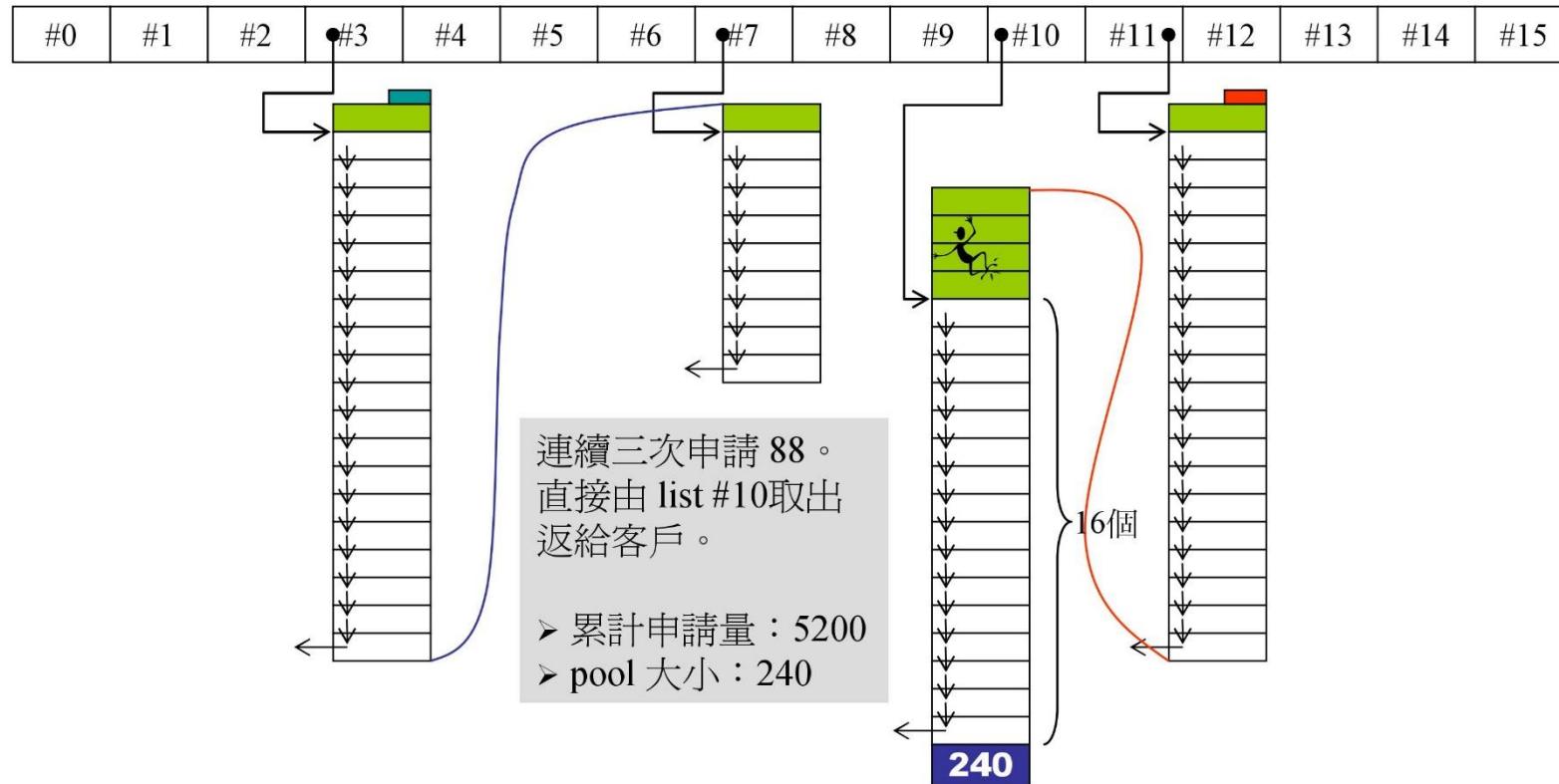


G2.9 std::alloc 運行一瞥.05



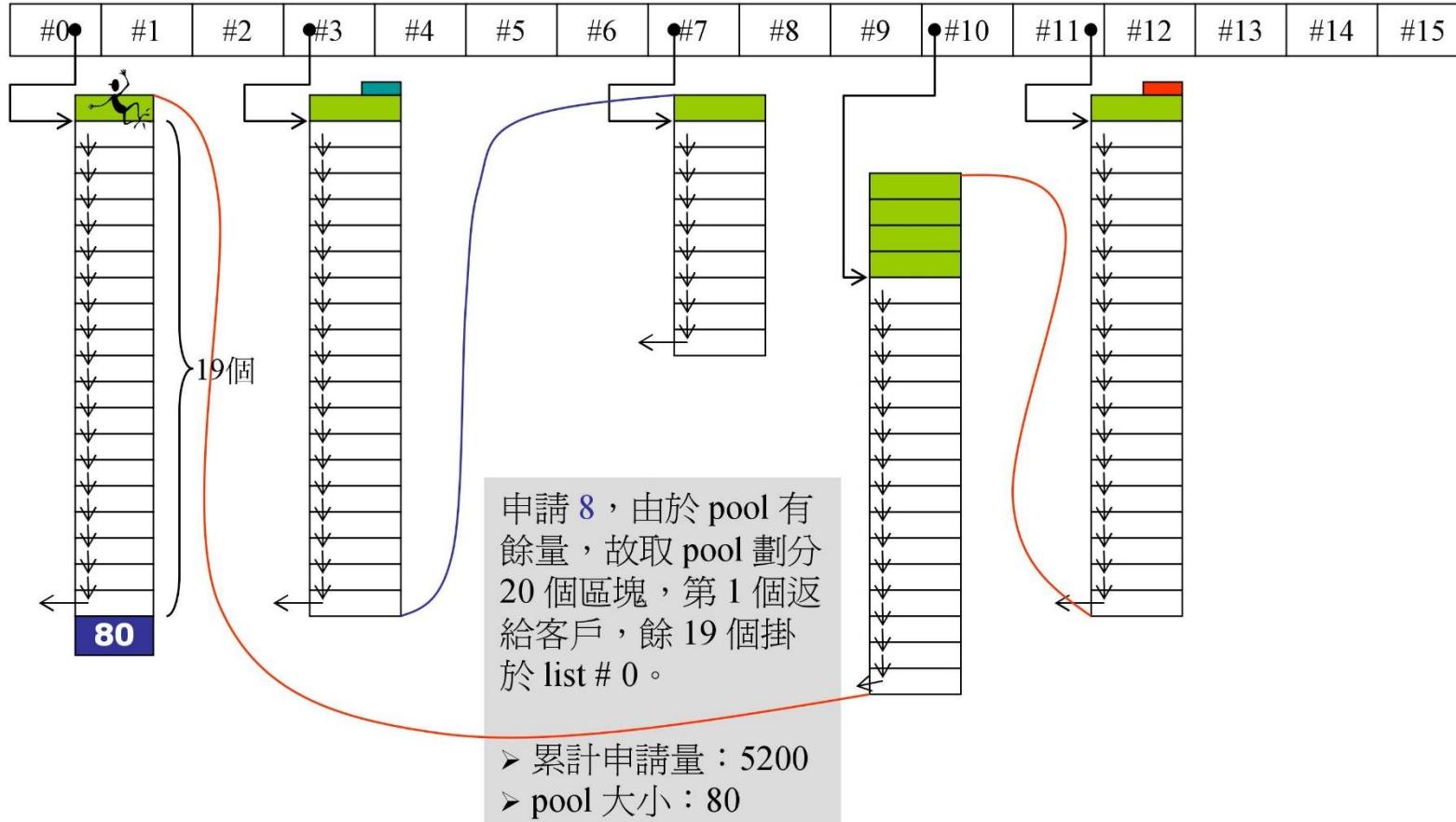


G2.9 std::alloc 運行一瞥.06



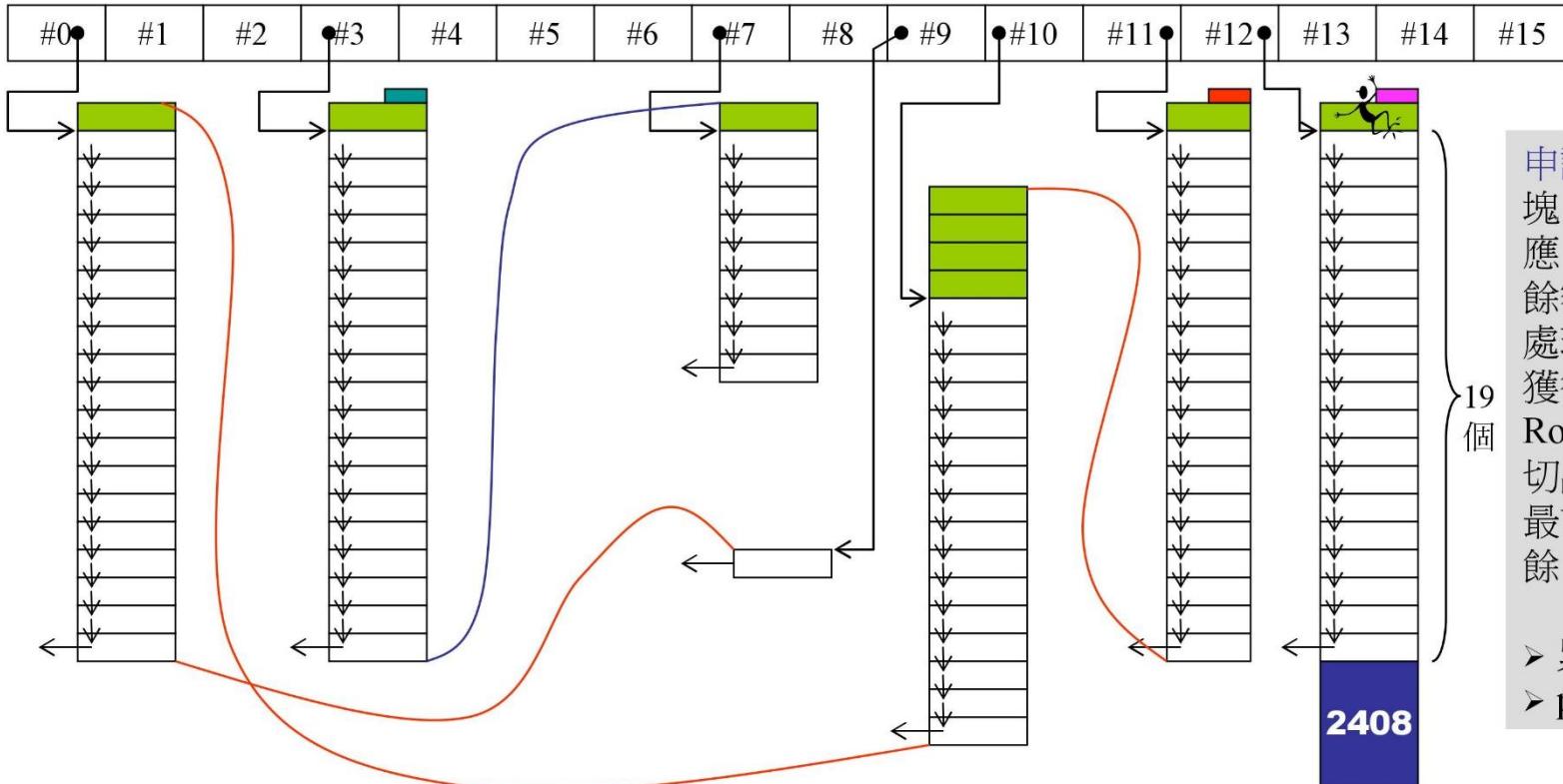


G2.9 std::alloc 運行一瞥.07





G2.9 std::alloc 運行一瞥.08

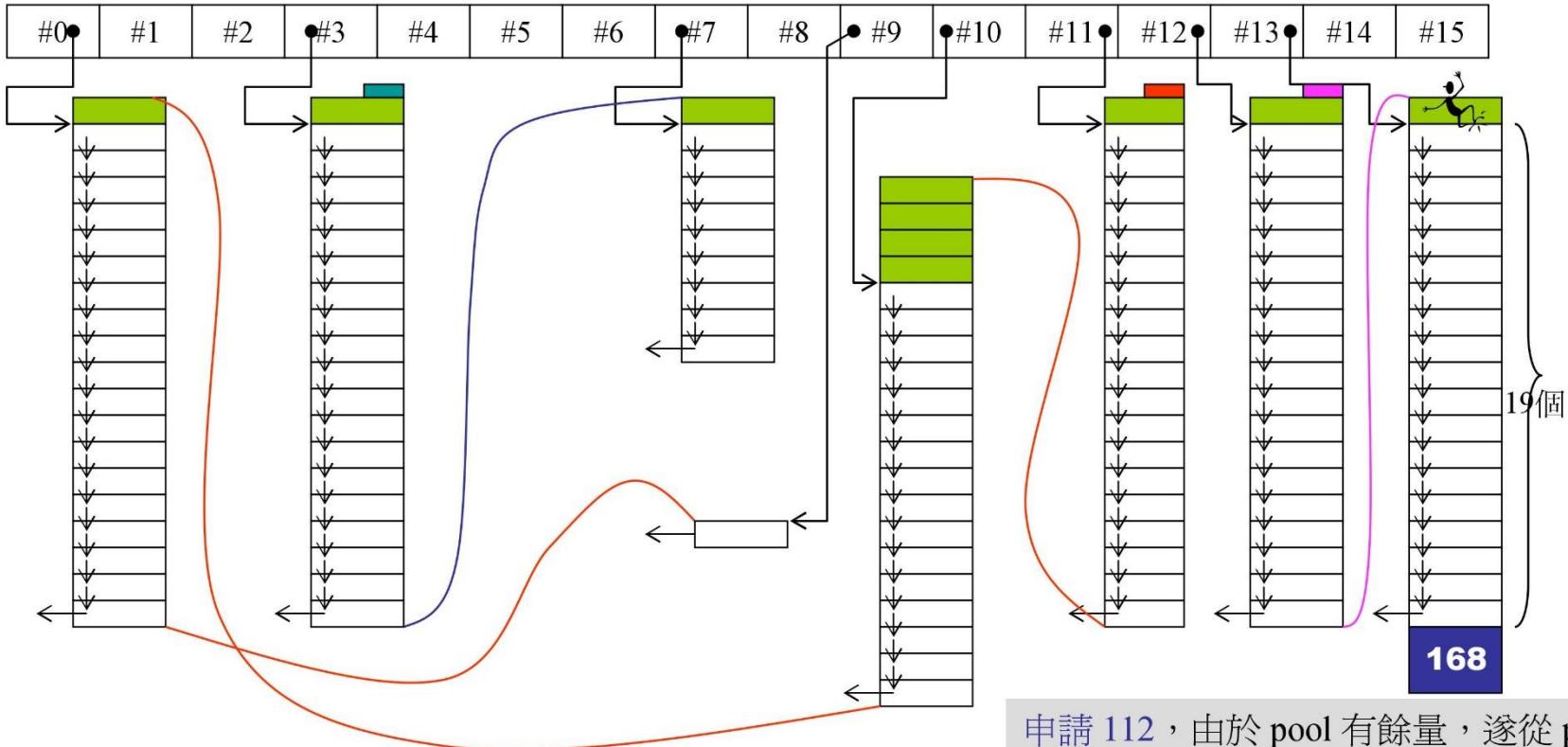


申請 104，list #12 無區塊，pool 餘量又不足供應 1 個，於是先將 pool 餘額撥給 list #9 (碎片處理)，然後索取並成功獲得挹注 $104 * 20 * 2 + \text{RoundUp}(5200 >> 4)$ ，切出 19 個給 list #12，最前頭那個返給客戶，餘 2408 備用。

- 累計申請量：9688
- pool 大小：2408



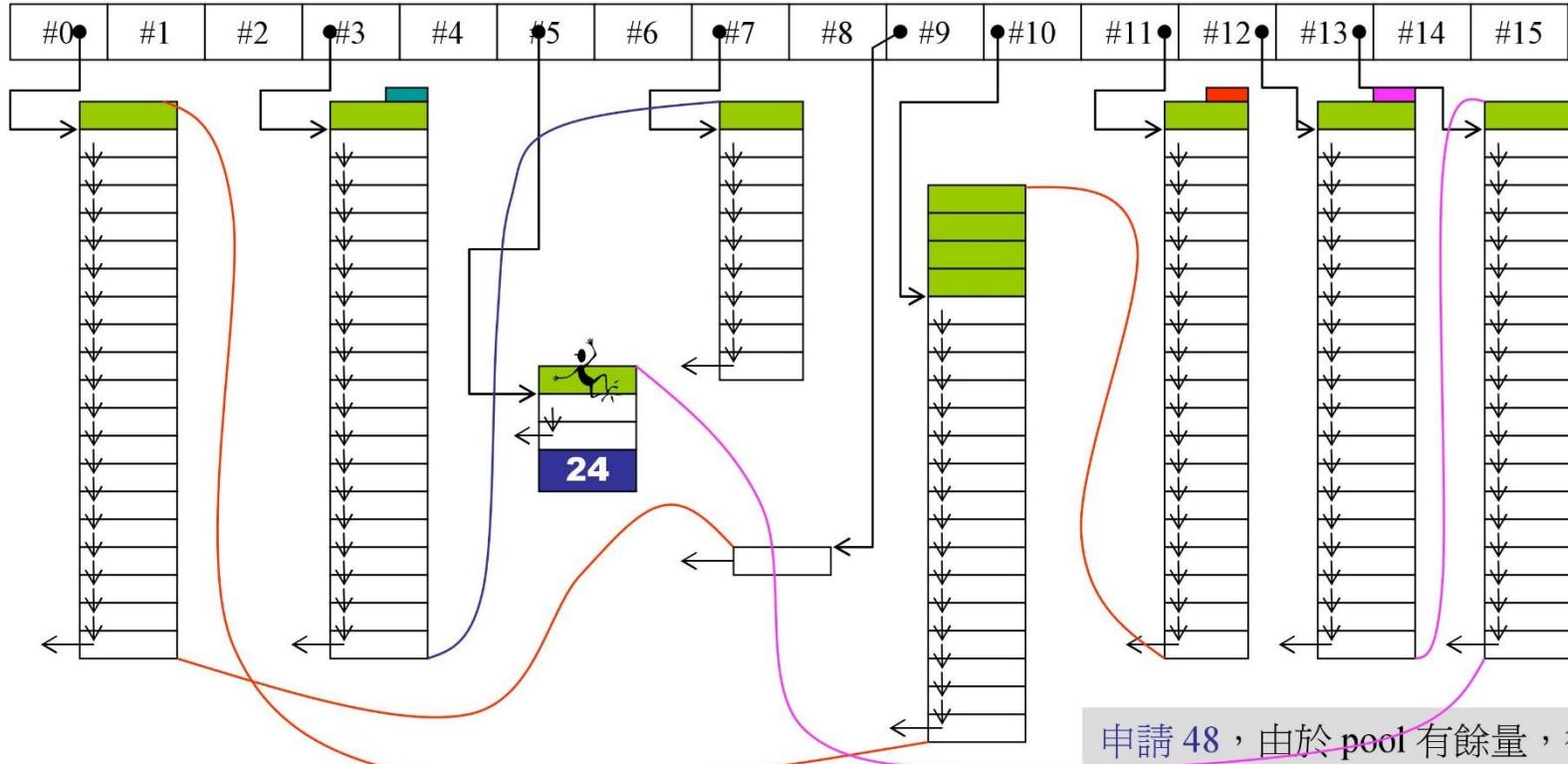
G2.9 std::alloc 運行一瞥.09



申請 112，由於 pool 有餘量，遂從 pool 取 20 個
區塊，撥 1 個返給客戶，留 19 個掛於 list #13.

- 累計申請量：9688
- pool 大小： $2408 - 112 \times 20 = 168$.

■ ■ ■ G2.9 std::alloc 運行一瞥.10

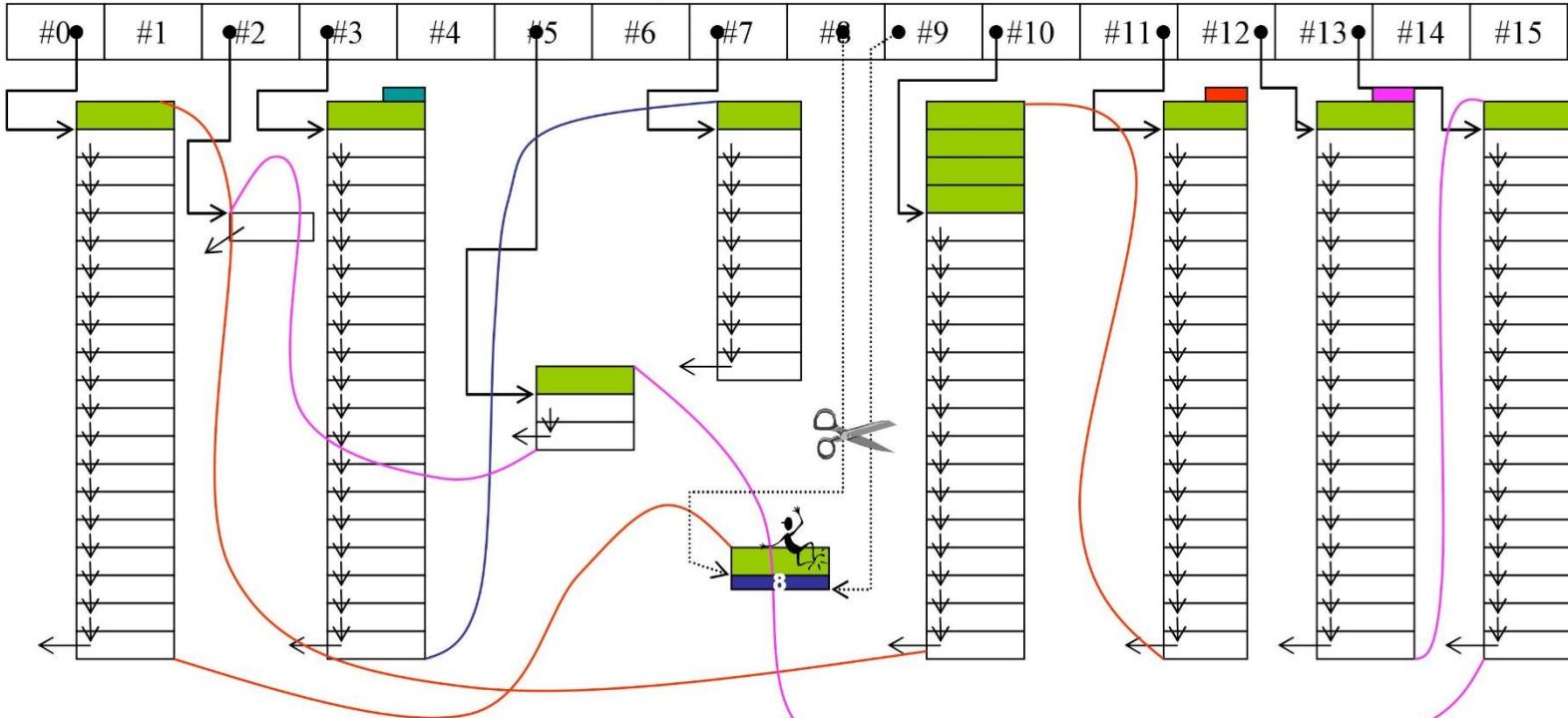


申請 48，由於 pool 有餘量，從 pool 取 3 個區塊，撥 1 個返給客戶，2 個掛於 list #5。

- 累計申請量：9688
 - pool 大小： $168 - 48 * 3 = 24$



G2.9 std::alloc 運行一瞥.11

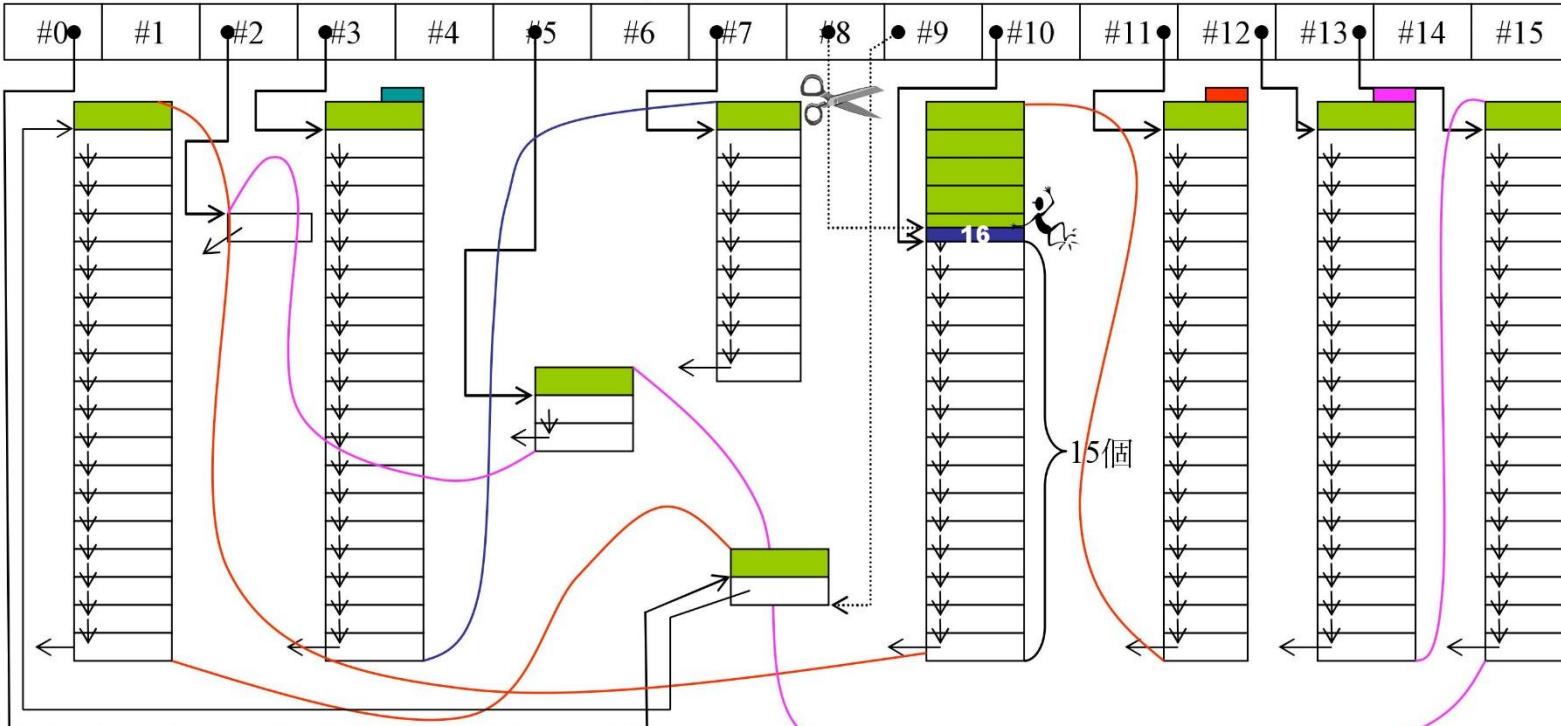


申請 72，list#8 無可用區塊，pool 餘量又不足供應 1 個，於是先將 pool 餘額撥給list#2，然後索取 $72*20*2+\text{RoundUp}(9688>>4)$ ，但為觀察系統邊界，我將 system heap 大小設為 10000，目前已索取 9688，因此無法滿足此次索取，於是 **alloc** 從手中資源取最接近之 80 (list#9) 回填 pool，再從中切出 72 返回客戶，餘 8。

- 累計申請量：9688
- pool 大小：8



G2.9 std::alloc 運行一瞥.12

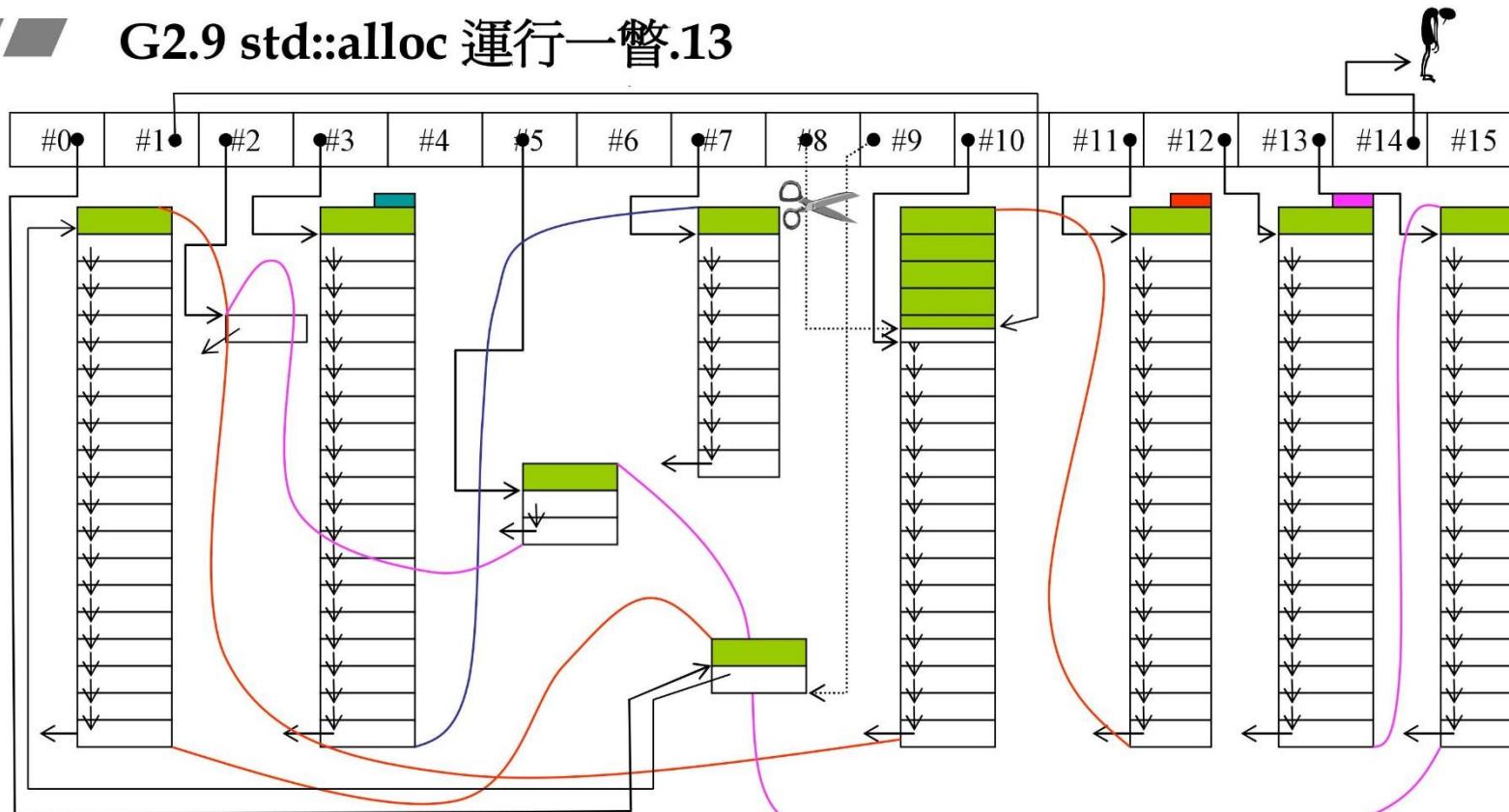


申請 72，list#8無可用區塊，pool 餘量又不足供應1個，於是先將 pool 餘額撥給 list#0，然後索取
 $72*20*2+RoundUp(9688>>4)$ ，但 system heap 大小已被我設為 10000，目前已索取 9688，因此無法滿足此次
索取，於是 alloc 從手中資源取最接近之 88 (list#10) 回填 pool，再從中切出 72 返給客戶，餘 16。

- 累計申請量：9688
- pool 大小：16



G2.9 std::alloc 運行一瞥.13

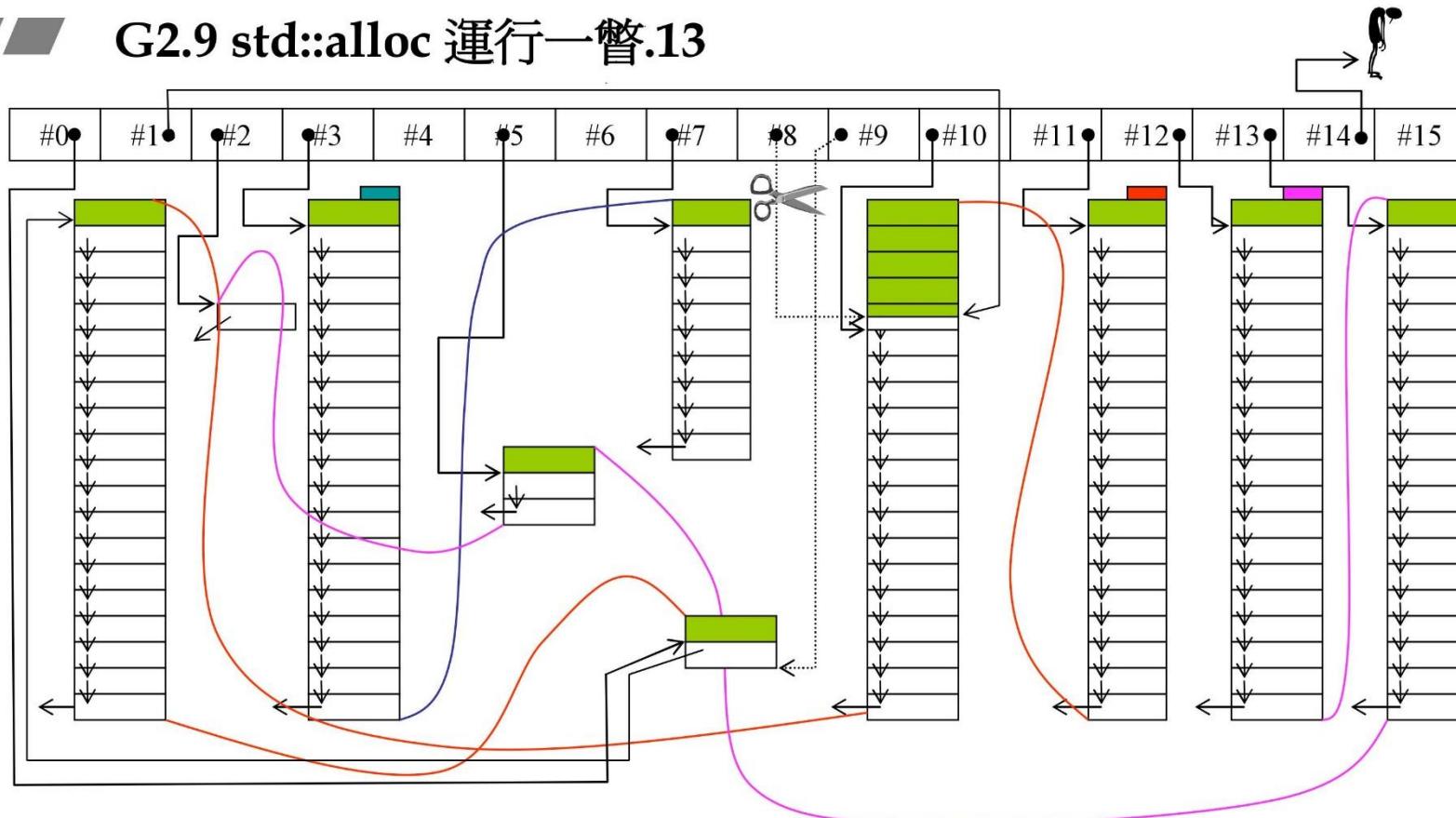


申請 120，list#14無可用區塊，pool 餘量又不足供應 1 個，於是先將 pool 餘額撥給 list#1，然後索取 $120*20*2+RoundUp(9688>>4)$ ，但 system heap 大小已被我設為 10000，目前已索取 9688，因此無法滿足此次索取，於是 alloc 從手中資源取最接近者回填 pool，但是找不到，終於窮途末路山窮水盡日薄西山。

- 累計申請量：9688
- pool 大小：0



G2.9 std::alloc 運行一瞥.13



累計量 : 9688

- 檢討**
- 檢討 1 : `alloc` 手中還有多少資源？以上白色皆是。可不可以將白色小區塊合成為較大區塊供應給客戶？唔，技術難度極高。
 - 檢討 2 : `system heap` 手中還剩多少資源？ $10000 - 9688 = 312$ 。可不可以將失敗的那次索取量折半...再折半...再折半...最終當索取量 ≤ 312 便能獲得滿足。

G2.9 std::alloc 源碼剖析, 1

```
#008 // 第一級分配器
#010 template <int inst>
#011 class __malloc_alloc_template {
#012 private:
#013     static void* oom_malloc(size_t);
#014     static void* oom_realloc(void*, size_t);
#015     static void (*__malloc_alloc_oom_handler)();
#016     //上面這個指針用來指向new-handler(if any)
#017 public:
#018     static void* allocate(size_t n)
#019     {
#020         void *result = malloc(n);    //直接使用malloc()
#021         if (0 == result) result = oom_malloc(n);
#022         return result;
#023     }
#024     static void deallocate(void *p, size_t /* n */)
#025     {
#026         free(p);                  //直接使用free()
#027     }
#028     static void* realloc(void *p, size_t /* old_sz */, size_t new_sz)
#029     {
#030         void * result = realloc(p, new_sz); //直接使用realloc()
#031         if (0 == result) result = oom_realloc(p, new_sz);
#032         return result;
#033     }
#034     static void (*set_malloc_handler(void (*f)()))()
#035     { //類似 C++ 的 set_new_handler().
#036         void (*old)() = __malloc_alloc_oom_handler; //記錄原new-handler
#037         __malloc_alloc_oom_handler = f; //把f 記起來以便爾後呼叫
#038         return(old); //把原先的handler傳回以便日後可恢復
#039     }
#040 };
```

#001 // #include <cstdlib>
#002 #include <cstddef>
#003 #include <new>
#004
#005 #define __THROW_BAD_ALLOC \
 cerr << "out of memory"; exit(1)

如果改用 operator new()，便可直接使用 C++ 的 set_new_handler()，不必寫出這般模擬。

typedef void (*H)();
H set_malloc_handler(H f);

相當於
my handler
void (f*)()

#001 // #include <cstdlib>
#002 #include <cstddef>
#003 #include <new>
#004
#005 #define __THROW_BAD_ALLOC \
 cerr << "out of memory"; exit(1)

如果改用 operator new()，便可直接使用 C++ 的 set_new_handler()，不必寫出這般模擬。

typedef void (*H)();
H set_malloc_handler(H f);

相當於
my handler
void (f*)()



G2.9 std::alloc 源碼剖析, 2

```
#042 template <int inst>
#043 void (*__malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;
#044
#045 template <int inst> //本例template參數'inst'完全沒派上用場
#046 void* __malloc_alloc_template<inst>::oom_malloc(size_t n)
#047 {
#048     void (*my_malloc_handler)();
#049     void* result;
#050
#051     for (;;) { //不斷嘗試釋放、分配、再釋放、再分配...
#052         my_malloc_handler = __malloc_alloc_oom_handler; //換個名稱
#053         if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
#054         (*my_malloc_handler)(); //呼叫handler，企圖釋放memory
#055         result = malloc(n); //再次嘗試分配memory
#056         if (result) return(result);
#057     }
#058 }
```

有可能在#037被改掉

my handler

```
#060 template <int inst>
#061 void * __malloc_alloc_template<inst>::oom_realloc(void* p, size_t n)
#062 {
#063     void (*my_malloc_handler)();
#064     void* result;
#065
#066     for (;;) { //不斷嘗試釋放、分配、再釋放、再分配...
#067         my_malloc_handler = __malloc_alloc_oom_handler; //換個名稱
#068         if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
#069         (*my_malloc_handler)(); //呼叫handler，企圖釋放memory
#070         result = realloc(p, n); //再次嘗試分配memory
#071         if (result) return(result);
#072     }
#073 }
```

是為了 multi-threading ?!



G2.9 std::alloc 源碼剖析, 3

```
#074 //-----
#075 typedef __malloc_alloc_template<0> malloc_alloc;
#076
#077 //一個換膚工程，將分配單位由 bytes 個數改為元素個數
#078 template<class T, class Alloc>
#079 class simple_alloc {
#080 public:
#081     static T* allocate(size_t n)           //一次分配n個T objects
#082     { return 0 == n? 0 : (T*)Alloc::allocate(n*sizeof(T)); }
#083     static T* allocate(void)             //一次分配1個T objects
#084     { return (T*)Alloc::allocate(sizeof(T)); }
#085     static void deallocate(T* p, size_t n) //一次歸還n個T objects
#086     { if (0 != n) Alloc::deallocate(p, n*sizeof(T)); }
#087     static void deallocate(T *p)          //一次歸還1個T objects
#088     { Alloc::deallocate(p, sizeof(T)); }
#089 };
```

用例：

```
template <class T, class Alloc = alloc>
class vector {
protected:
    // 專屬之分配器，每次分配一個元素大小
    typedef simple_alloc<T, Alloc>
        data_allocator;
    result = data_allocator::allocate(); //分配1個元素空間
    ...
    data_allocator::deallocate(result);
```

G2.9 std::alloc 源碼剖析, 4

```

#098 //本例兩個template參數完全沒派上用場
#099 template <bool threads, int inst>
#100 class _default_alloc_template {
#101 private:
#102     //實際上應使用 static const int x = N
#103     //取代 #094~#096 的 enum { x = N }, 但目前支援該性質的編譯器不多
#104
#105     static size_t ROUND_UP(size_t bytes) {
#106         return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1)); → 若bytes為13，則(13+7) & ~7
#107     }
#108     union obj { //type definition
#109         union obj* free_list_link;
#110     }; //改用 struct 亦可
#111
#112 private:
#113     static obj* volatile free_list[__NFREELISTS];
#114
#115     static size_t FREELIST_INDEX(size_t bytes) {
#116         return (((bytes) + __ALIGN-1)/__ALIGN - 1); → 若bytes為8，則(8+7)/8-1=0
#117     }
#118
#119     // Returns an object of size n, and optionally adds to size n free list.
#120     static void *refill(size_t n);
#121
#122     // Allocates a chunk for nobjs of size "size". nobjs may be reduced
#123     // if it is inconvenient to allocate the requested number.
#124     static char* chunk_alloc(size_t size, int &nobjs);
#125
#126     // Chunk allocation state.
#127     static char* start_free; //指向'pool'的頭
#128     static char* end_free; //指向'pool'的尾
#129     static size_t heap_size; //分配累計量
#130

```

若bytes為13，則(13+7) & ~7
即 10100 & 11000，得 10000即 16

#090 //---
#091 //第二級分配器
#092 //---
#094 enum { __ALIGN = 8}; //小區塊的上調邊界
#095 enum { __MAX_BYTES = 128}; //小區塊的上限
#096 enum { __NFREELISTS = __MAX_BYTES / __ALIGN}; //free-list個數

若bytes為8，則(8+7)/8-1=0
若bytes為16，則(16+7)/8-1=1
若bytes為20，則(20+7)/8-1=2
若bytes為24，則(24+7)/8-1=2

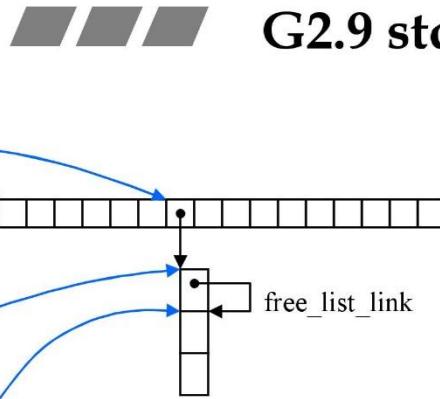
G2.9 std::alloc 源碼剖析, 5

```

#132 public:
#133
#134     static void* allocate(size_t n) //n must be > 0
#135     {
#136         obj* volatile *my_free_list; //obj**
#137         obj* result;
#138
#139         if (n > (size_t) __MAX_BYTES) { //改用第一級
#140             return (malloc_alloc::allocate(n));
#141         }
#142
#143         my_free_list = free_list + FREELIST_INDEX(n);
#144         result = *my_free_list;
#145         if (result == 0) { //list為空
#146             void* r = refill(ROUND_UP(n)); //挹注之
#147             return r;
#148         } //若往下進行表示list內已有可用區塊
#149         *my_free_list =
#150             result->free_list_link;
#151         return (result);
#152     }

```

refill() 會充填 free list 並
返回一個（其實就是第
一個）區塊的起始地址



如果這 p 並非當初從 **alloc** 取得，仍可併入 **alloc** 內 (這不好).
如果 p 所指大小不是 8 倍數，甚至會帶來災難

```

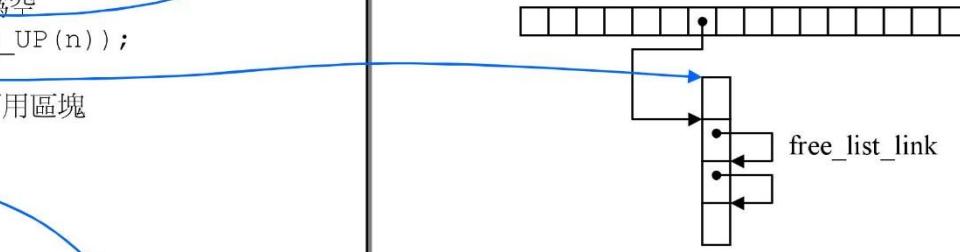
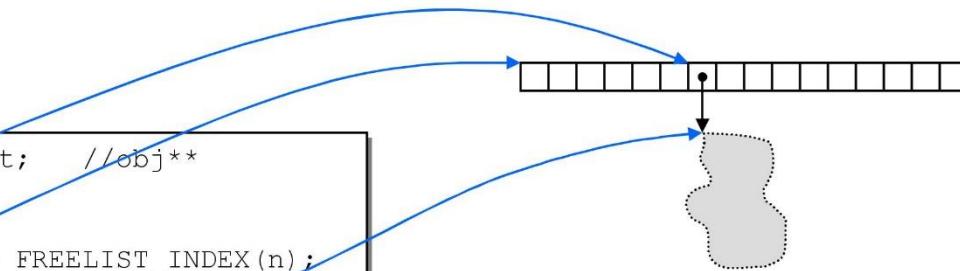
#154     static void deallocate(void* p, size_t n) //p 不為 0
#155     {
#156         obj* q = (obj*)p;
#157         obj* volatile *my_free_list; //obj**
#158
#159         if (n > (size_t) __MAX_BYTES) {
#160             malloc_alloc::deallocate(p, n); //改用第一級
#161             return;
#162         }
#163         my_free_list = free_list + FREELIST_INDEX(n);
#164         q->free_list_link = *my_free_list;
#165         *my_free_list = q;
#166     }
#167
#168     static void* reallocate(void* p, size_t old_sz,
#169                             size_t new_sz); //此處略列
#170 }

```

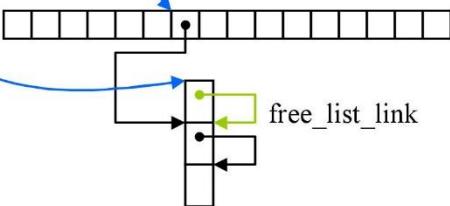


G2.9 std::alloc 源碼剖析, 5

```
#136     obj* volatile *my_free_list;    //obj**
#137     obj* result;
...
#143     my_free_list = free_list + FREELIST_INDEX(n);
#144     result = *my_free_list;
#145     if (result == 0) { //list為空
#146         void* r = refill(ROUND_UP(n));
#147         return r;
#148     } //若往下進行表示list內已有可用區塊
#149     *my_free_list =
#150         result->free_list_link;
#151     return (result);
#152 }
```



這個 list 是
refill() 所得結果



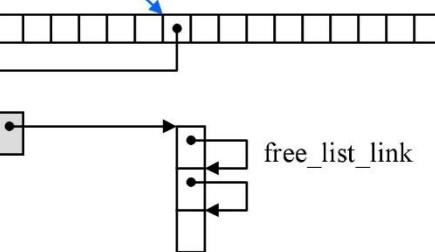
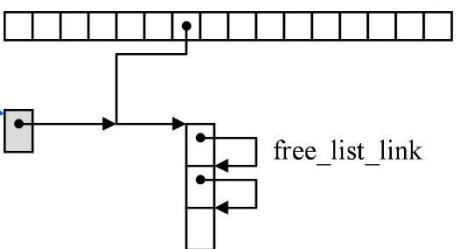
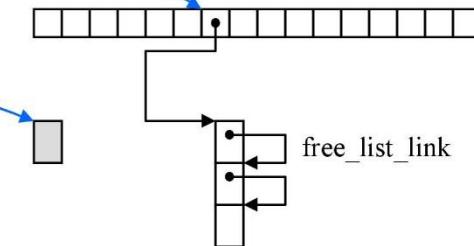
這個 list 是
原本就有的

G2.9 std::alloc 源碼剖析, 5

```
#154     static void deallocate(void* p, size_t n) //p 不為 0
#155     {
#156         obj* q = (obj*)p;
#157         obj* volatile *my_free_list; //obj**
...
#163         my_free_list = free_list + FREELIST_INDEX(n);
```

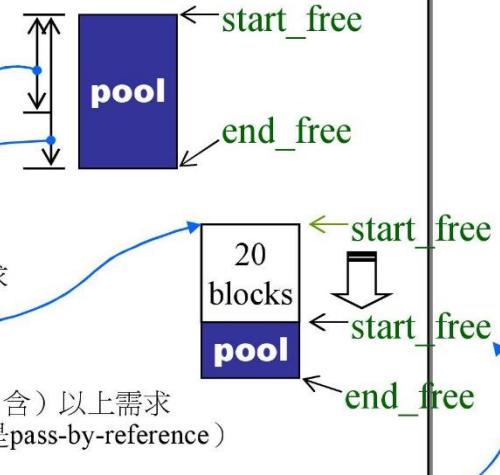
```
#164     q->free_list_link = *my_free_list;
```

```
#165     *my_free_list = q;
#166 }
```



G2.9 std::alloc 源碼剖析, 6

```
#172 // -----
#173 // We allocate memory in large chunks in order to avoid fragmenting the
#174 // malloc heap too much. We assume that size is properly aligned.
#175 // We hold the allocation lock.
#176 //-----
#177 template <bool threads, int inst>
#178 char*
#179 __default_alloc_template<threads, inst>::
#180 chunk_alloc(size_t size, int& nobjs)
#181 {
#182     char* result;
#183     size_t total_bytes = size * nobjs;
#184     size_t bytes_left = end_free - start_free;
#185
#186     if (bytes_left >= total_bytes) { //pool空間足以滿足20塊需求
#187         result = start_free;
#188         start_free += total_bytes; //調整(降低)pool水位
#189         return(result);
#190     } else if (bytes_left >= size) { //pool空間只足以滿足一塊(含)以上需求
#191         nobjs = bytes_left / size; //改變需求數(注意nobjs是pass-by-reference)
#192         total_bytes = size * nobjs; //改變需求總量(bytes)
#193         result = start_free;
#194         start_free += total_bytes; //調整(降低)pool水位
#195         return(result);
#196     } else { //pool空間不足以滿足一塊需求
#197         size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4); //現在打算從system free-store取這麼多來挹注
#198
#199         //首先嘗試將 pool 做充份運用
#200         if (bytes_left > 0) { //如果pool還有空間,
#201             obj* volatile *my_free_list = free_list + FREELIST_INDEX(bytes_left); //找出應移轉至第#號free-list(區塊儘可能大)
#202             free_list += FREELIST_INDEX(bytes_left);
#203             //將pool空間編入第#號free-list(肯定只成1區塊)
#204             ((obj*)start_free)->free_list_link = *my_free_list;
#205             *my_free_list = (obj*)start_free;
#206     }
```



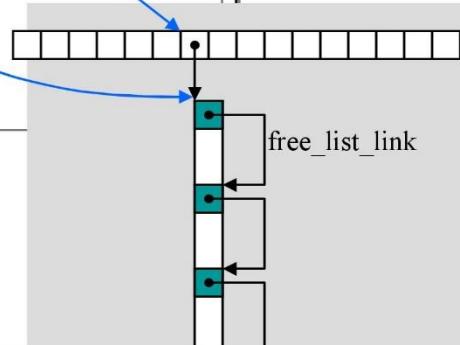
G2.9 std::alloc 源碼剖析, 7

```

#207     start_free = (char*)malloc(bytes_to_get); //從system free-store取這麼多,注入pool
#208     if (0 == start_free) {                      //失敗!以下試從free-list找區塊
#209         int i;
#210         obj* volatile *my_free_list, *p;
#211
#212         //Try to make do with what we have. That can't hurt.
#213         //We do not try smaller requests, since that tends
#214         //to result in disaster on multi-process machines.
#215         for (i = size; i <= _MAX_BYTES; i += _ALIGN) { //例88,96,104,112...
#216             my_free_list = free_list + FREELIST_INDEX(i);
#217             p = *my_free_list;
#218             if (0 != p) { //該free-list內有可用區塊,以下釋出一塊(only)給pool
#219                 *my_free_list = p -> free_list_link;
#220                 start_free = (char*)p;
#221                 end_free = start_free + i;
#222                 return(chunk_alloc(size, nobjs)); //遞歸再試一次
#223                 //此時的pool定夠供應至少一個區塊
#224                 //而任何殘餘零頭終將被編入適當free-list
#225             }
#226         }
#227         end_free = 0; //至此,表示 memory 已山窮水盡.
#228         //改用第一級,看看 oom-handler 能否盡點力
#229         start_free = (char*)malloc_alloc::allocate(bytes_to_get);
#230         //這會導致拋出異常,或導致memory不足的情況得到改善
#231     }
#232     //至此,表示已從system free-store成功取得很多 memory
#233     heap_size += bytes_to_get; //累計總分配量
#234     end_free = start_free + bytes_to_get; //挹注pool(調整尾端)
#235     return(chunk_alloc(size, nobjs)); //遞歸再試一次
#236 }
#237 }
```

→ p易讓人誤解,實為
obj** my_free_list;
obj* p;

i 從 size + _ALIGN
開始迭代更佳.



G2.9 std::alloc 源碼剖析, 8

```
#240 // Returns an object of size n, and optionally adds to size n free list.
#241 // We assume that n is properly aligned. We hold the allocation lock.
#242 //-----
#243 template <bool threads, int inst>
#244 void* __default_alloc_template<threads, inst>::
#245 refill(size_t n) //n 已調整至 8 的倍數
#246 {
#247     int nobjs = 20; //預設取 20 個區塊 (但不一定能夠)
#248     char* chunk = chunk_alloc(n, nobjs); //nobjs 是 pass-by-reference
#249     obj* volatile *my_free_list; //obj**
#250     obj* result;
#251     obj* current_obj;
#252     obj* next_obj;
#253     int i;
#254 }
```

```
#255     if (1 == nobjs) return(chunk); //實際得1，交給客戶
#256     //以下開始將所得區塊掛上 free-list
#257     my_free_list = free_list + FREELIST_INDEX(n);
#258     //在 chunk 內建立 free list
#259     result = (obj*)chunk;
#260     *my_free_list = next_obj = (obj*)((char*)chunk + n);
#261     for (i=1; ; ++i) {
#262         current_obj = next_obj;
#263         next_obj = ((obj*)((char*)next_obj + n));
#264         if (nobjs-1 == i) { //最後一個
#265             current_obj->free_list_link = 0;
#266             break;
#267         } else {
#268             current_obj->free_list_link = next_obj;
#269         }
#270     }
#271     return(result);
#272 }
```

所謂切割就是把指針所指處轉型為obj，再取其next_obj指針繼續行事，一而再三...

G2.9 std::alloc 源碼剖析, 9

■■■ G2.9 std::alloc 觀念大整理

```
list<Foo> c; //假設 sizeof(Foo)+(4*2) <= 128  
c.push_back( Foo(1) );
```

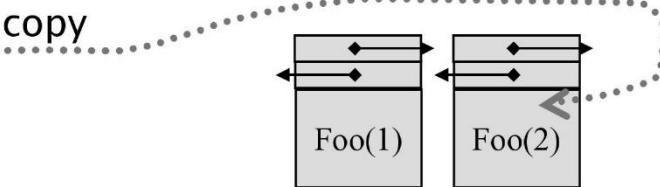
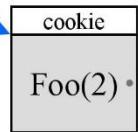


這一塊是通過 **alloc** 取得，
不帶cookie

~~c.push_back(new Foo(2)); //錯誤（元素型別不符）~~

Foo* p = new Foo(2);

c.push_back(*p);



這一塊是通過 **alloc** 取得，
不帶cookie

delete p;

■■■ G2.9 std::alloc 批鬥大會

```
#136 obj* volatile *my_free_list;
```

```
#210 obj* volatile *my_free_list, *p; ➔
```

```
#208 if (0 == start_free) {
```

```
#218 if (0 != p) {
```

```
#255 if (1 == nobjs) return(chunk);
```

```
obj* *p1, *p2;
```

➔ ~~obj** p1, *p2;~~

➔ ~~obj **p1, *p2; ➔ obj** p1; obj* p2;~~

➔ ~~obj** p1, **p2;~~

➔ ~~obj** p1; obj** p2;~~

```
#197 size_t bytes_to_get =
```

```
...  
#207 start_free = (char*)malloc(bytes_to_get);
```

```
typedef void (*new_handler)();  
new_handler set_new_handler(new_handler p) throw();
```

|||

```
typedef void (*H)();  
static H set_malloc_handler(H f);
```

```
#212 //Try to make do with what we have. That can't hurt.
```

```
#213 //We do not try smaller requests, since that tends
```

```
#214 //to result in disaster on multi-process machines.
```

deallocate() 完全沒有調用 free() or delete
(源於其設計上的先天缺陷)

■■■ G4.9 pool allocator 運行觀察

//我要一個可累計總分配量和總釋放量的 operator new/delete.

//除非 user 直接使用 malloc/free,
//否則都避不開它們, 這就可以累計總量.
static long long **countNew** = 0;
static long long **timesNew** = 0;

//小心，這影響無遠弗屆

//它們不可被聲明於 namespace 內

```
inline void* operator new(size_t size) {  
    //cout << "jjhou global new(), \t" << size << "\t";  
    countNew += size;  
    ++timesNew;  
    return myAlloc( size );  
}
```

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

... 繢右

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

■■■ 重載 ::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); }
```

... 接左

//天啊, 以下(1)(2)可並存並由(2)抓住流程 (但它對我這兒測試無用).

//若只存在 (1), 抓不住流程. 為什麼?

//在 class members 中必須二擇一 (任一均可), 否則編譯報錯.

```
inline void operator delete(void* ptr, size_t size) { //(1)  
    myFree( ptr );  
}
```

```
inline void operator delete(void* ptr) { //(2)  
    myFree( ptr );  
}
```

```
inline void operator delete[](void* ptr, size_t size) { //(1)  
    myFree( ptr );  
}
```

```
inline void operator delete[](void* ptr) { //(2)  
    myFree( ptr );  
}
```



G4.9 pool allocator 運行觀察

Q: 我能不能觀察到 malloc() 真正分配出去的總量 (含所有 overhead) ?

A: **不能**; 除非...



```
//C++/11 alias template
template <typename T>
using listPool = list<T, __gnu_cxx::__pool_alloc<T>>;
```

```
//reset countNew
countNew = 0;
timesNew = 0;

listPool<double> lst;
for (int i=0; i< 1000000; ++i)
    lst.push_back(i);
cout << "::countNew= " << ::countNew << endl;
        //16752832 (注意, node 都不帶 cookie)
cout << "::timesNew= " << ::timesNew << endl; //122
```



```
//reset countNew
countNew = 0;
timesNew = 0;

list<double> lst;
for (int i=0; i< 1000000; ++i)
    lst.push_back(i);
cout << "::countNew= " << ::countNew << endl;
        //16000000 (注意, node 都帶 cookie)
cout << "::timesNew= " << ::timesNew << endl; //1000000
```

```
list<double, __gnu_cxx::__pool_alloc<double>> lst;
```

■■■■ G2.9 std::alloc 移植至 C

std::alloc 整個由 static data members 和 static member functions 構成，整體只一個 class 而無分枝，因此移植至 C 很容易。需注意：

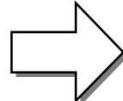
`chunk_alloc(size_t size, int& nobjs);`

pass by reference 移至 C 時或需改為 pass by pointer.

移植步驟

- 去除 templates
- 將 union obj 移至 _default_alloc_template class 外部獨立
- 將所有 static data 移為 global
- 將 __malloc_alloc 改名為 malloc_alloc, 將 __default_alloc 改名為 alloc
- 將 malloc_alloc 的所有 static functions 移為 global
- 將 alloc 的所有 static functions 移為 global
- 將 .cpp 改為 .c，將上述 pass by reference 改為 pass by pointer，再將：

```
union obj {  
    union obj* free_list_link;  
};
```



```
typedef union __obj {  
    union __obj* free_list_link;  
} obj;
```

或：

```
typedef struct __obj {  
    struct __obj* free_list_link;  
} obj;
```



The End

內存管理

從平地到萬丈高樓

Memory Management 101

第一講 primitives

第二講 std::allocator

第三講 malloc/free

第四講 loki::allocator

第五講 other issues



侯捷

胸中自有丘壑

觸類旁通

VC6 內存分配

```
ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc_new_group(...)
    __sbh_alloc_new_region()
    sbh_alloc_block(...)
    heap_alloc_base(...)
    heap_alloc_dbg(...)
    nh_malloc_dbg(...)
    malloc_dbg(...)
② __ioinit() // initialize lowio
    __sbh_heap_init()
① heap_init(...)
mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()
```

SBH: Small Block Heap

```
if (size <= __sbh_threshold) { //3F8, i.e. 1016
    pvReturn = __sbh_alloc_block(size);
    if (pvReturn) return pvReturn;
}
if (size == 0) size = 1;
size = (size + ...) & ~(...);
return HeapAlloc(crtheap, 0, size);
```

VC10 內存分配

```

    graph TD
        A[_heap_init(...)] --> B[_initterm(..)]
        A --> C[mainCRTStartup()]
        C --> D[_initterm(..)]
        D --> E[_initstdio()]
        E --> F[_initterm(..)]
        F --> G[_doexit(code, 0, 0)]
        G --> H[_exit(code)]
        H --> I[main()]
        I --> J[_initterm(..)]
        J --> K[_initstdio()]
        K --> L[_initterm(..)]
        L --> M[_cinit()]
        M --> N[_setenvp()]
        N --> O[_setargv()]
        O --> P[_crtGetEnvironmentStringsA()]
        P --> Q[GetCommandLineA()]
        Q --> R[_heap_alloc_base(size_t size)]
        R --> S[_nh_malloc_dbg(...)]
        S --> T[_malloc_dbg(...)]
        T --> U[_iointi()]
        U --> V[_heap_alloc_base(...)]
        V --> W[_nh_malloc_dbg(...)]
        W --> X[_malloc_dbg(...)]
        X --> Y[_iointi()]
        Y --> Z[_heap_alloc_base(...)]
        Z --> AA[_forceinline void * __cdecl _heap_alloc (size_t size)]
        AA --> BB[_if (_crtheap == 0) { ... }]
        BB --> CC[_return HeapAlloc(_crtheap, 0, size ? size : 1);]
    
```

Call graph illustrating the flow of control from the CRT startup code through various initialization routines to the final memory allocation code in `_heap_alloc_base()`.

The graph shows the following sequence of calls:

- `_heap_init()` calls `_initterm()`, `mainCRTStartup()`, and `_iointi()`.
- `mainCRTStartup()` calls `_initterm()`, `_initstdio()`, `_iointi()`, and `_cinit()`.
- `_iointi()` calls `_heap_alloc_base()`.
- `_heap_alloc_base()` calls `_nh_malloc_dbg()`.
- `_nh_malloc_dbg()` calls `_malloc_dbg()`.
- `_malloc_dbg()` calls `_iointi()`.
- `_iointi()` calls `_heap_alloc_base()`.
- `_heap_alloc_base()` calls `_nh_malloc_dbg()`.
- `_nh_malloc_dbg()` calls `_malloc_dbg()`.
- `_malloc_dbg()` calls `_iointi()`.
- `_iointi()` calls `_heap_alloc_base()`.
- `_heap_alloc_base()` calls `_forceinline void * __cdecl _heap_alloc (size_t size)`.
- `_forceinline void * __cdecl _heap_alloc (size_t size)` checks if `_crtheap == 0`.
- If `_crtheap == 0`, it performs error handling: `_FF_MSGBANNER()`, `_NMSG_WRITE(_RT_CRT_NOTINIT)`, and `_crtExitProcess(255)`.
- Finally, it returns `HeapAlloc(_crtheap, 0, size ? size : 1)`.

所有內存管理
機制都繼續存
在，只不過，
深埋到 O.S. 層
面去了 (詳見
[Windows Heap
課程](#))



SBH 之始 – `_heap_init()` 和 `_sbh_heap_init()`

```
int __cdecl _heap_init (
    int mtflag
)
{
    // Initialize the "big-block" heap first.
    if ( __crtheap = HeapCreate( mtflag ? 0 : HEAP_NO_SERIALIZE,
                                BYTES_PER_PAGE, 0 ) == NULL )
        return 0; 4096

    // Initialize the small-block heap
    if ( __sbh_heap_init() == 0 )
    {
        ←
        HeapDestroy( __crtheap );
        return 0;
    }
    return 1;
}
不論 big-block heap or
small-block heap ,
只要失敗就 return 0 ,
別玩了。
```

heapinit.c

CRT 會先為自己建立一個 `_crtheap`，然後從中配置 SBH 所需的 headers, regions 做為管理之用。App. 動態配置時若 size > threshold 就以 `HeapAlloc()` 從 `_crtheap` 取。若 size <= threshold 就從 SBH 取 (實際區塊來自 `VirtualAlloc()`)



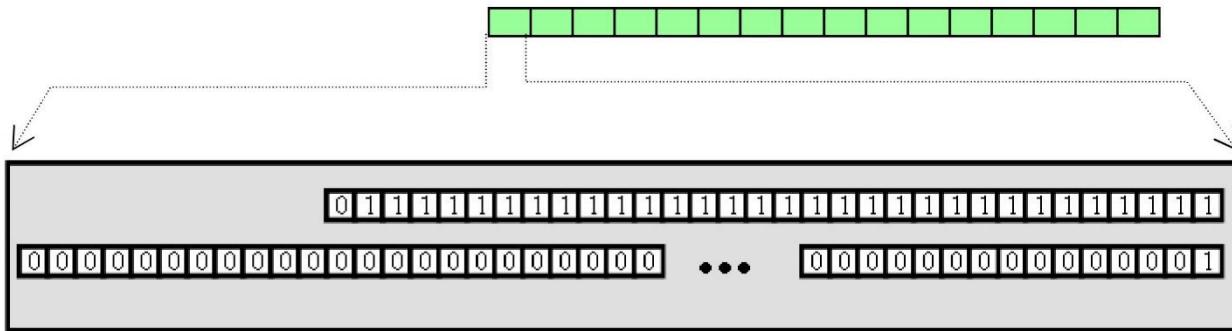
```
int __cdecl __sbh_heap_init (void)
{
    if ( !(__sbh_pHeaderList =
           HeapAlloc( __crtheap, 0, 16 * sizeof(HEADER) ) ) )
        return FALSE;

    __sbh_pHeaderScan = __sbh_pHeaderList;
    __sbh_pHeaderDefer = NULL;
    __sbh_cntHeaderList = 0;
__sbh_sizeHeaderList = 16;

    return TRUE;
}
```

sbheap.c

SBH 之始 – `_heap_init()` 和 `_sbh_heap_init()`



```
typedef unsigned int BITVEC;  
  
typedef struct tagHeader  
{  
    BITVEC      bitvEntryHi;  
    BITVEC      bitvEntryLo;  
    BITVEC      bitvCommit;  
    void *      pHeapData;  
    struct tagRegion * pRegion;  
}  
HEADER, *PHEADER;
```

VC6 內存分配

```

    ExitProcess(code)
    _initterm(,,) //do terminators
    __endstdio(void)
    _initterm(,,) //do pre-terminators
    doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
    __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    {
        __sbh_alloc_new_group(...)
        __sbh_alloc_new_region()
        sbh_alloc_block(...)

        __heap_alloc_base(...)
        __heap_alloc_dbg(...)
        __nh_malloc_dbg(...)
        malloc_dbg(...)

        ② __ioinit() // initialize lowio .....
        __sbh_heap_init()

        ① __heap_init(...)

        mainCRTStartup()
        KERNEL32! bff8b6e6()
        KERNEL32! bff8b598()
        KERNEL32! bff89f5b()
    }

```

```

/* Memory block identification */
#define _FREE_BLOCK      0
#define _NORMAL_BLOCK    1
#define _CRT_BLOCK        2
#define _IGNORE_BLOCK     3
#define _CLIENT_BLOCK     4
#define _MAX_BLOCKS       5

#ifndef _DEBUG
#define _malloc_crt  malloc
...
#else /* _DEBUG */
#define _THISFILE _FILE_
#define _malloc_crt(s) _malloc_dbg
(s, _CRT_BLOCK,
 _THISFILE, _LINE_)
...
#endif /* _DEBUG */

```

```

void __cdecl __ioinit (void)
{
    ...
    if ( (pio = _malloc_crt( IOINFO_ARRAY_ELTS * sizeof(ioinfo) ))
        == NULL )
    ...
}

```

```

typedef struct {
    long osfhnd;
    char osfile;
    char pipech;
} ioinfo;

```

32

8

100

VC6 內存分配

```

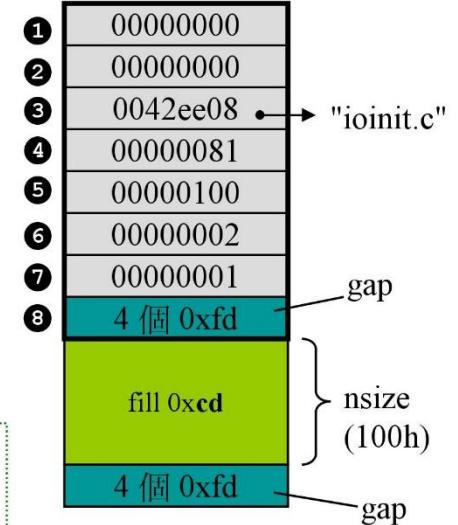
ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    [sbh_alloc_new_group()]
    [sbh_alloc_new_region()]
    sbh_alloc_block(...)
    _heap_alloc_base(...)
        heap_alloc_dbg(...)
            nh_malloc_dbg(...)
            malloc_dbg(...)
② _ioinit() // initialize lowio
    sbh_heap_init()
① _heap_init(...)
mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()

```

```

#define nNoMansLandSize 4
typedef struct _CrtMemBlockHeader
{
    struct _CrtMemBlockHeader * pBlockHeaderNext;
    struct _CrtMemBlockHeader * pBlockHeaderPrev;
    char * szFileName;
    int nLine;
    size_t nDataSize;
    int nBlockUse;
    long lRequest;
    unsigned char gap[nNoMansLandSize];
    /* followed by:
     * unsigned char data[nDataSize];
     * unsigned char anotherGap[nNoMansLandSize];
     */
} _CrtMemBlockHeader;

```



```

...
blockSize = sizeof(_CrtMemBlockHeader) + nSize + nNoMansLandSize;
...
pHead = (_CrtMemBlockHeader *) _heap_alloc_base(blockSize);
... (續下頁)

```

VC6 內存分配

```

ExitProcess(code)
    _initterm(,,) //do terminators
    __endstdio(void)
    _initterm(,,) //do pre-terminators
doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
    __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ _crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc_new_group(...)
    __sbh_alloc_new_region()
    sbh_alloc_block(...)
    heap_alloc_base(...)
        heap_alloc_dbg(...)  

        nh_malloc_dbg(...)
        malloc_dbg(...)
② _ioinit() // initialize lowio
    __sbh_heap_init()
① heap_init(...)
mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()

```

... (承上頁)

```

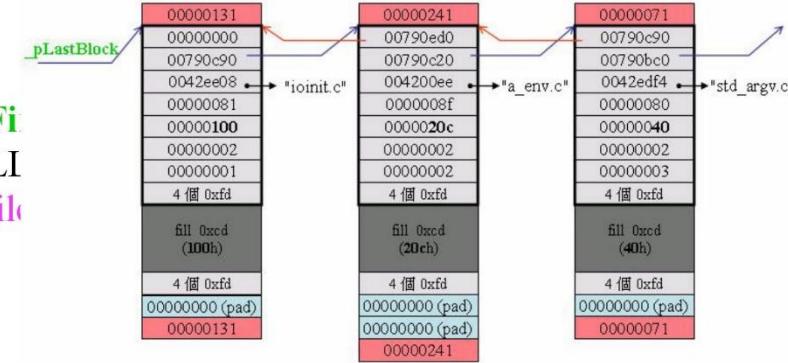
if (_pFirstBlock)
    _pFirstBlock->pBlockHeaderPrev = pHead;
else
    _pLastBlock = pHead;

pHead->pBlockHeaderNext = _pFi
pHead->pBlockHeaderPrev = NULI
pHead->szFileName = (char *)szFile
pHead->nLine = nLine;
pHead->nDataSize = nSize;
pHead->nBlockUse = nBlockUse;
pHead->lRequest = lRequest;

/* link blocks together */
_pFirstBlock = pHead;

/* fill in gap before and after real block */
memset((void *)pHead->gap, _bNoMansLandFill, nNoMansLandSize);
memset((void *)(pData(pHead) + nSize), _bNoMansLandFill, nNoMansLandSize);
/* fill data with silly value (but non-zero) */
memset((void *)pData(pHead), _bCleanLandFill, nSize);
return (void *)pData(pHead);

```



```

static unsigned char _bNoMansLandFill = 0xFD;
static unsigned char _bDeadLandFill = 0xDD;
static unsigned char _bCleanLandFill = 0xCD;

static _CrtMemBlockHeader * _pFirstBlock;
static _CrtMemBlockHeader * _pLastBlock;

```

VC6 內存分配

```
ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
    doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc_new_group(...)
    __sbh_alloc_new_region()
    sbh_alloc_block(...)
    __heap_alloc_base(...)

    __heap_alloc_dbg(...)
    __nh_malloc_dbg(...)
    malloc_dbg(...)

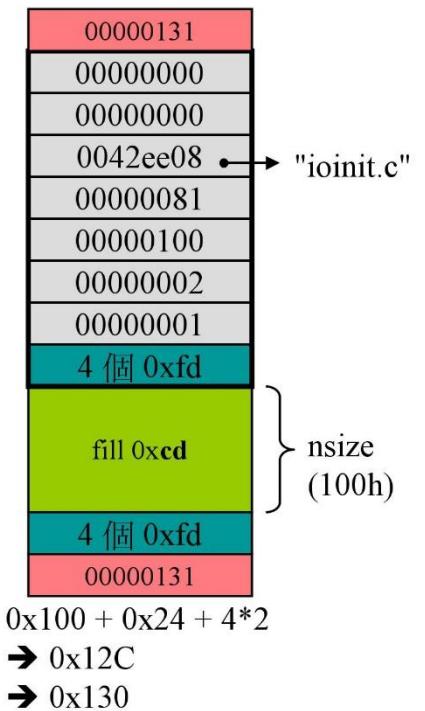
② __ioinit() // initialize lowio
    __sbh_heap_init()

① __heap_init(...)

mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()
```

```
if (size <= __sbh_threshold) { //3F8, i.e. 1016
    pvReturn = __sbh_alloc_block(size);
    if (pvReturn) return pvReturn;
}
if (size == 0) size = 1;
size = (size + ...) & ~(...);
return HeapAlloc(__crtheap, 0, size);
```

VC6 內存分配



```

ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
    doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc_new_group(...)
    __sbh_alloc_new_region()
    sbh_alloc_block(...)
    heap_alloc_base(...)
    heap_alloc_dbg(...)
    nh_malloc_dbg(...)
    malloc_dbg(...)
② _ioinit() // initialize lowio
    __sbh_heap_init()
① heap_init(...)
mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()

```

```

// add 8 bytes entry overhead and round up to next para size
sizeEntry = (intSize + 2 * sizeof(int) 16
            + (BYTES_PER_PARA - 1))
& ~(BYTES_PER_PARA - 1);

```

VC6 內存分配

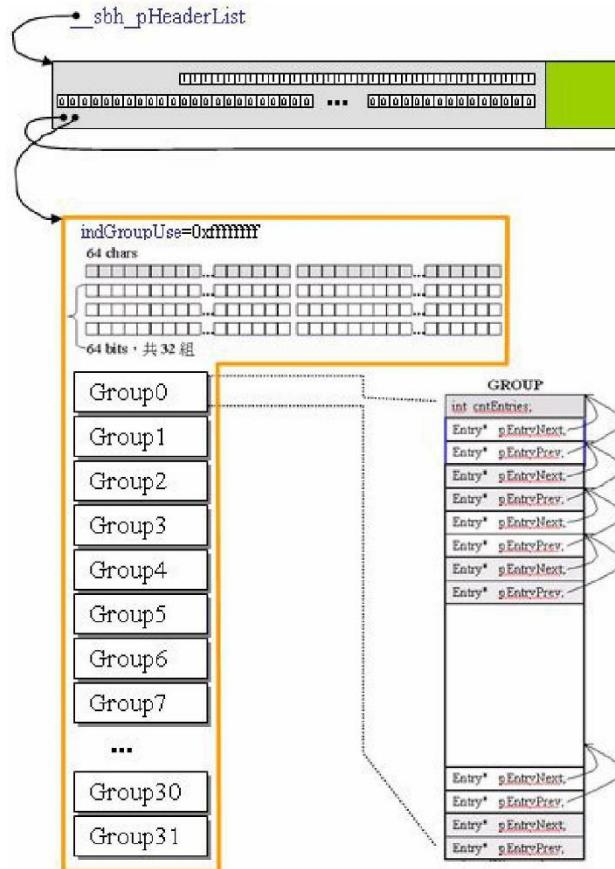
虛擬地址空間

```

ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ __crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc new_group(...)
    __sbh_alloc new_region()
    sbh_alloc block(...)
        __heap_alloc_base(...)
        __heap_alloc_dbg(...)
        __nh_malloc_dbg(...)
        malloc_dbg(...)
    ② __ioinit() // initialize lowio
        __sbh_heap_init()
    ① __heap_init(...)

mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()

```



```

typedef struct tagRegion
{
    int           indGroupUse;
    char          cntRegionSize[64];
    BITVEC        bitvGroupHi[32];
    BITVEC        bitvGroupLo[32];
    struct tagGroup grpHeadList[32];
} REGION, *PREGION;

```

```

typedef struct tagGroup
{
    int           cntEntries;
    struct tagListHead listHead[64];
} GROUP, *PGROUP;

```

```

typedef struct tagListHead
{
    struct tagEntry * pEntryNext;
    struct tagEntry * pEntryPrev;
} LISTHEAD, *PLISTHEAD;

```

VC6 內存分配

```

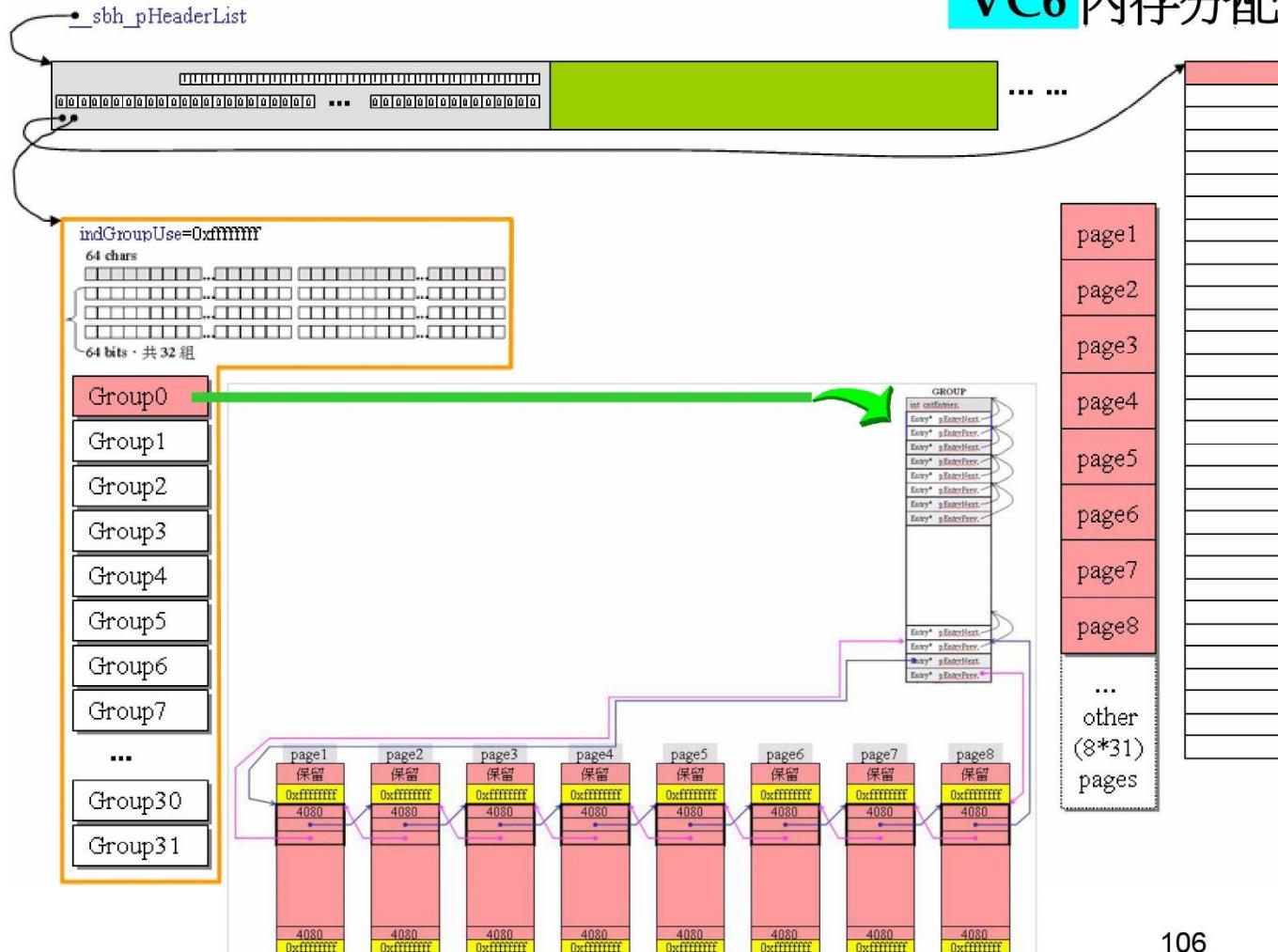
ExitProcess(code)
    _initterm(,,) //do terminators
        __endstdio(void)
    _initterm(,,) //do pre-terminators
doexit(code, 0, 0)
⑨ exit(code)
⑧ main()
    _initterm(,,) //do C++ initializations
        __initstdio(void)
    _initterm(,,) //do initializations
⑦ _cinit() // do C data initialize
⑥ _setenvp()
⑤ _setargv()
④ _crtGetEnvironmentStringsA()
③ GetCommandLineA()
    __sbh_alloc_new_group(...)
    __sbh_alloc_new_region()
    sbh_alloc_block(...)

    heap_alloc_base(...)
    heap_alloc_dbg(...)
    nh_malloc_dbg(...)
    malloc_dbg(...)

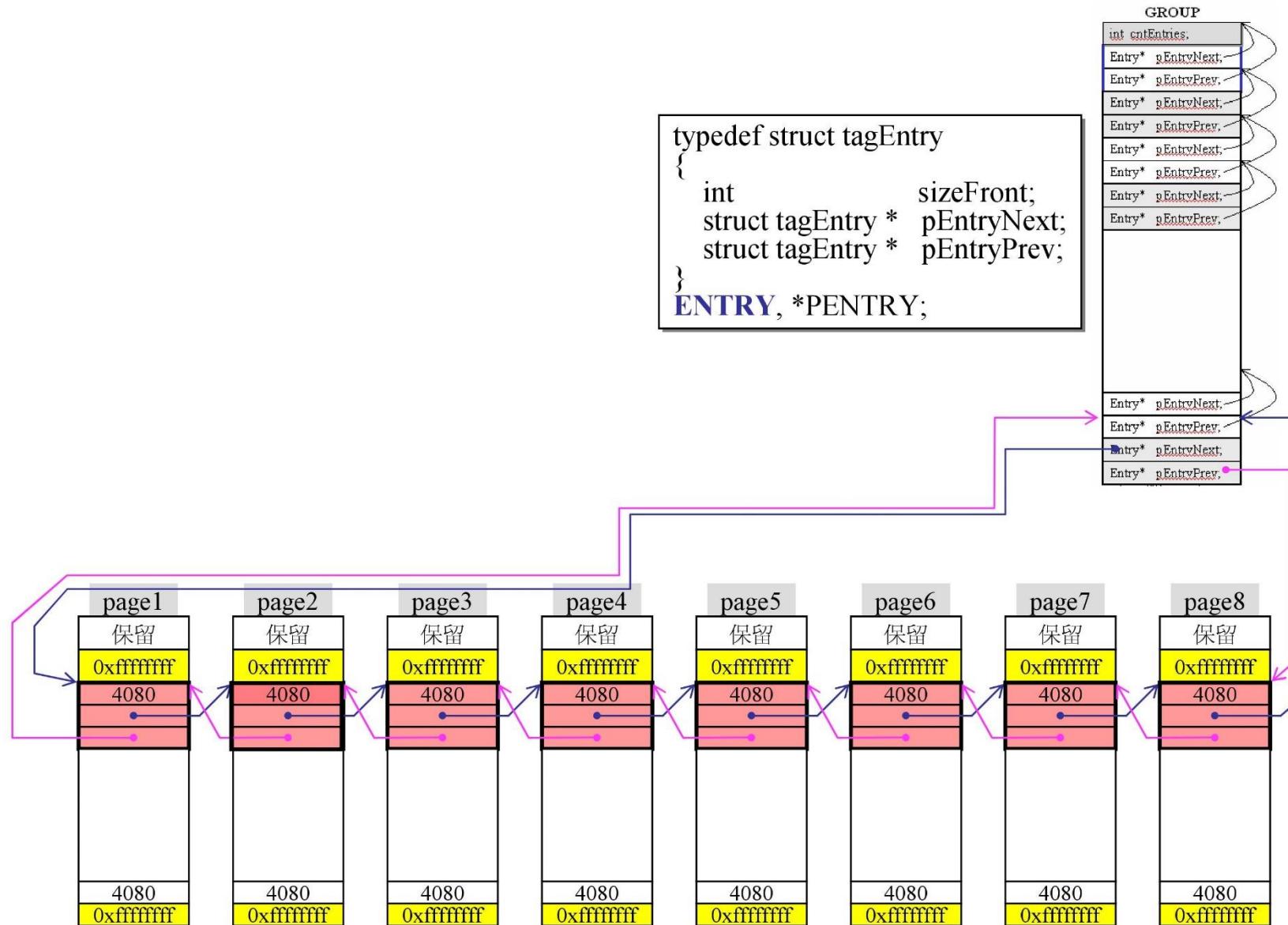
② _ioinit() // initialize lowio
    __sbh_heap_init()

① heap_init(...)
mainCRTStartup()
KERNEL32! bff8b6e6()
KERNEL32! bff8b598()
KERNEL32! bff89f5b()

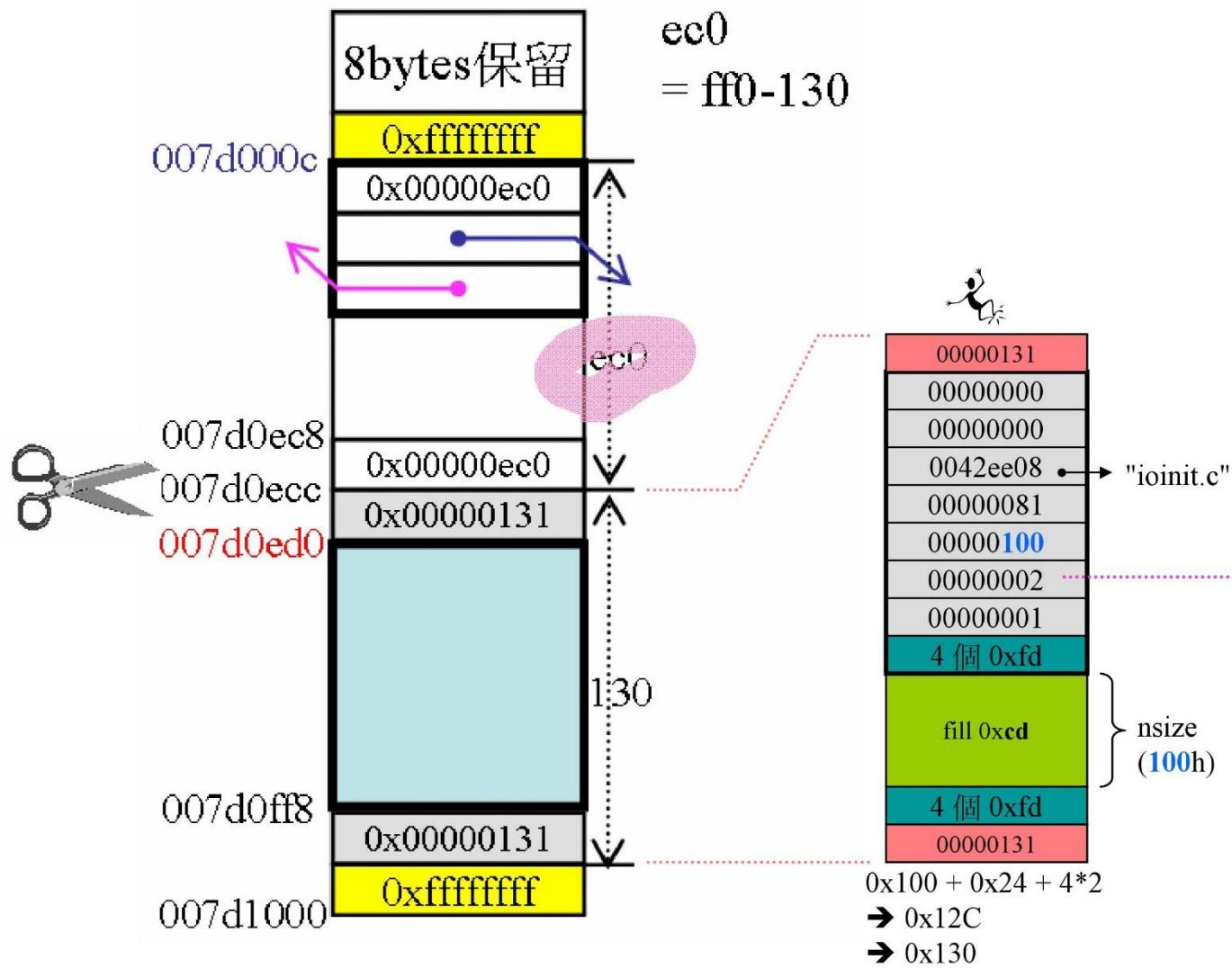
```



VC6 內存分配



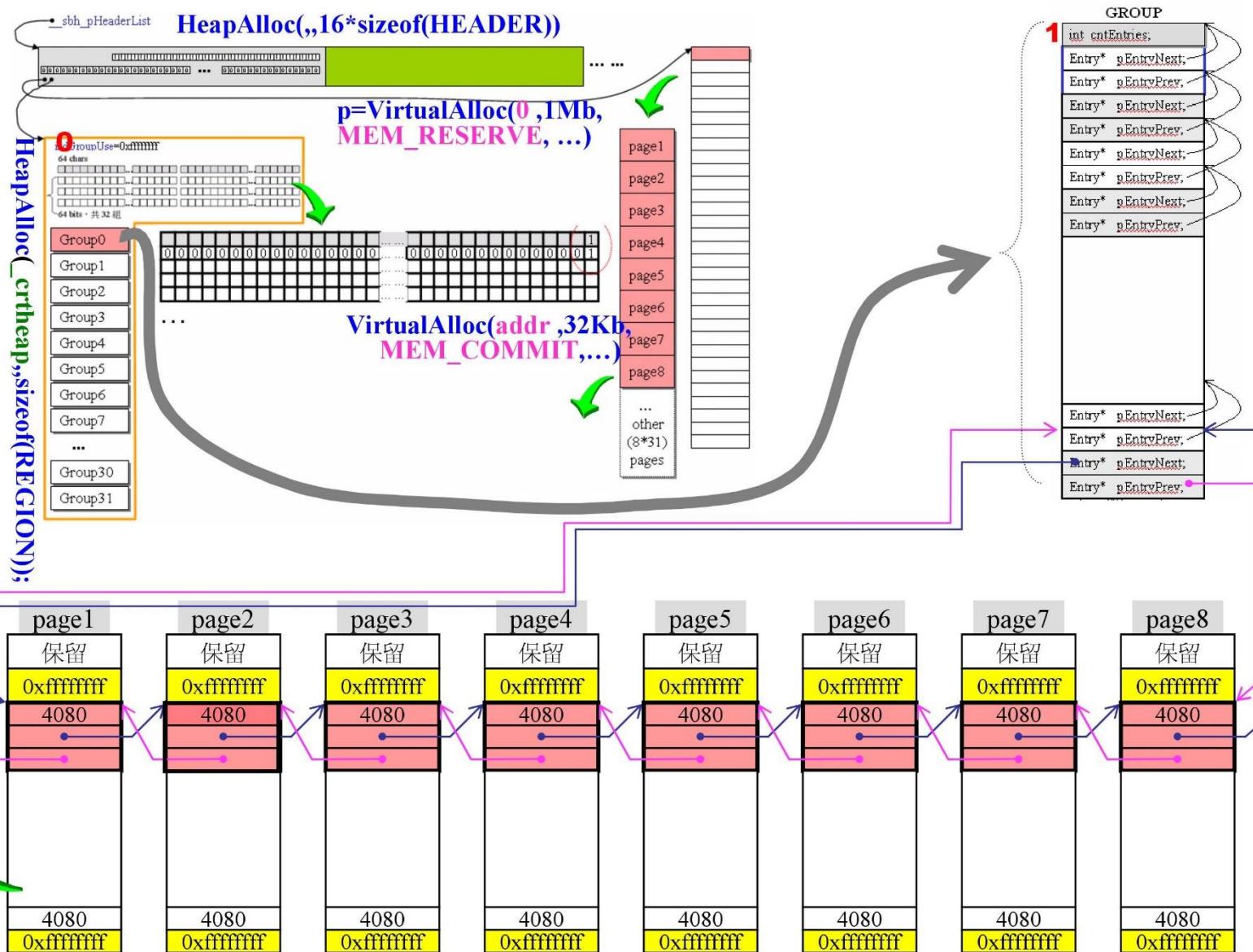
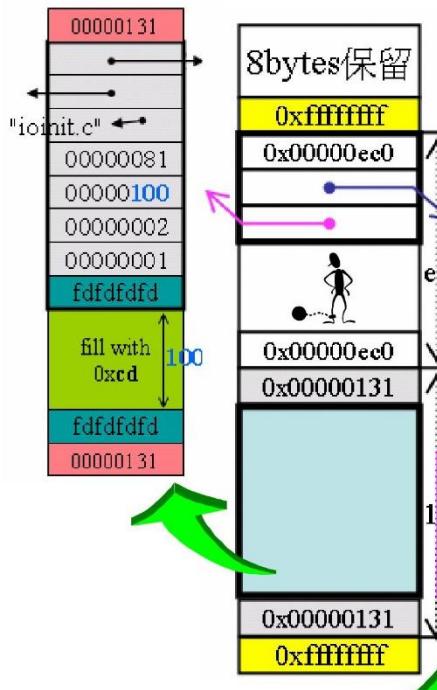
VC6 內存分配



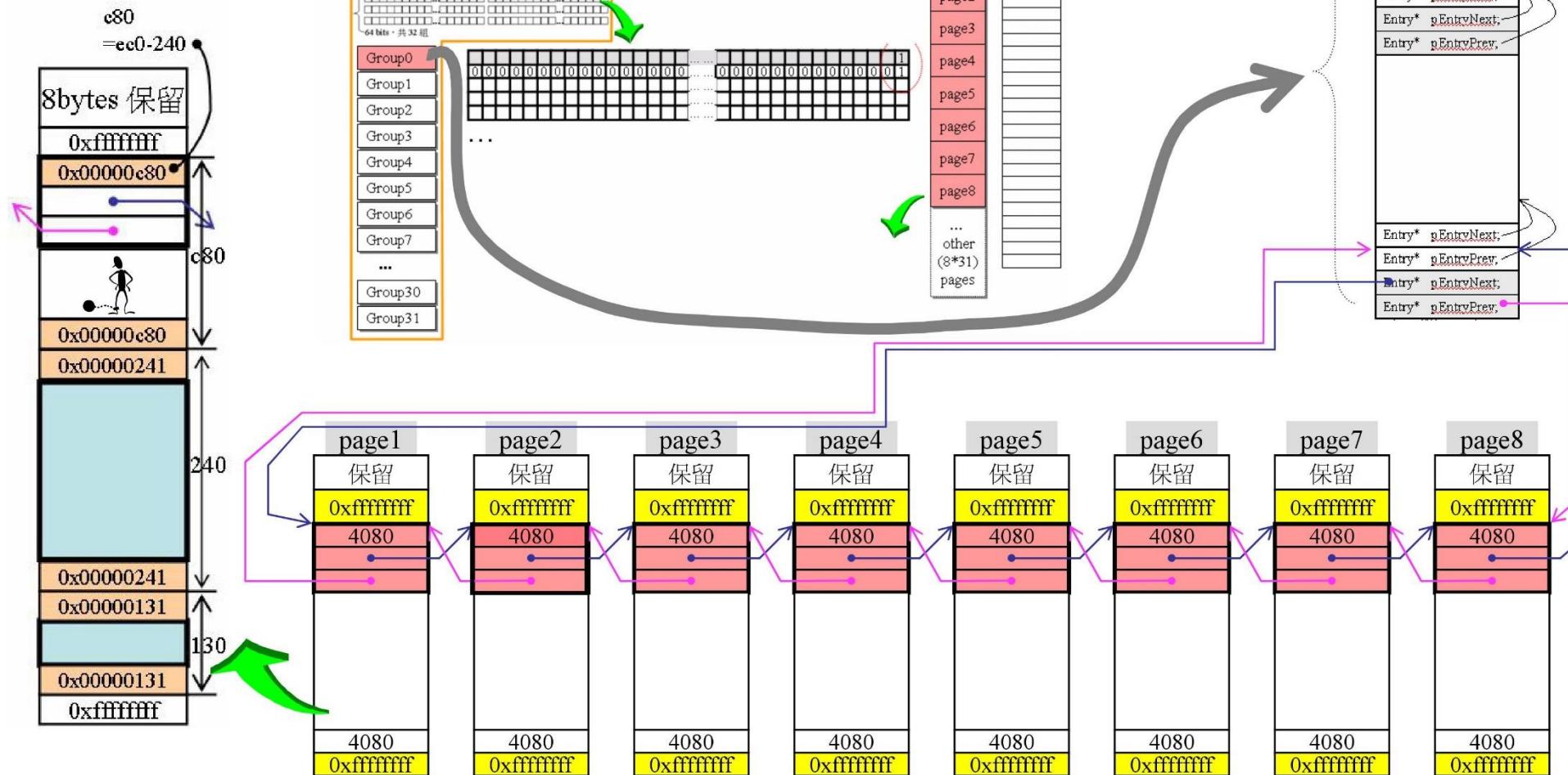
```
/* Memory block identification */
#define _FREE_BLOCK 0
#define _NORMAL_BLOCK 1
#define _CRT_BLOCK 2
#define _IGNORE_BLOCK 3
#define _CLIENT_BLOCK 4
#define _MAX_BLOCKS 5
```

VC6 內存管理 首次分配

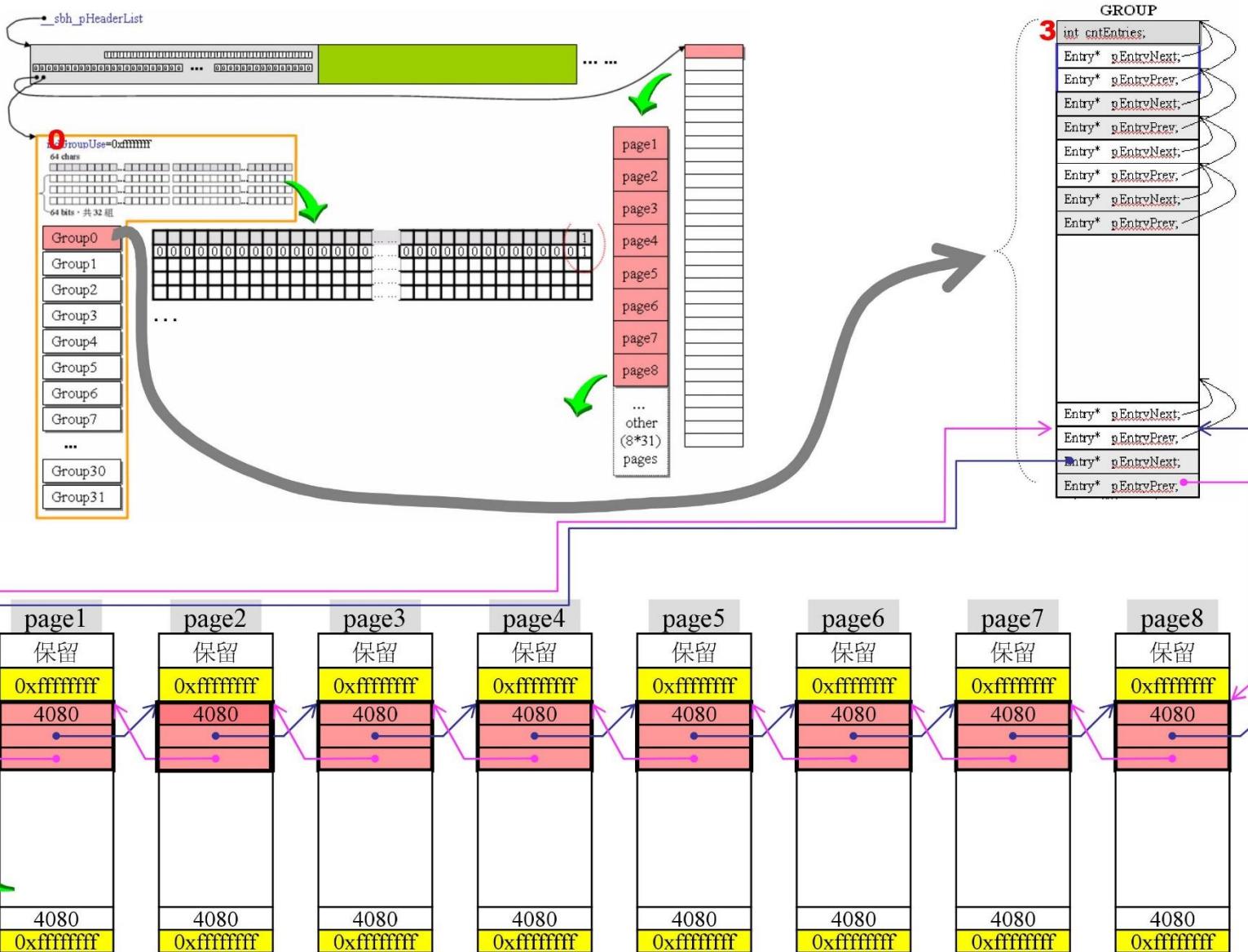
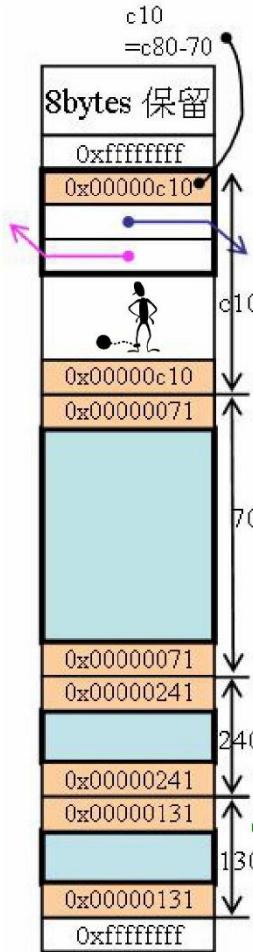
由 ioinit.c, line#81 申請
100h, 區塊大小130h,
理應由 #18 lists 供應



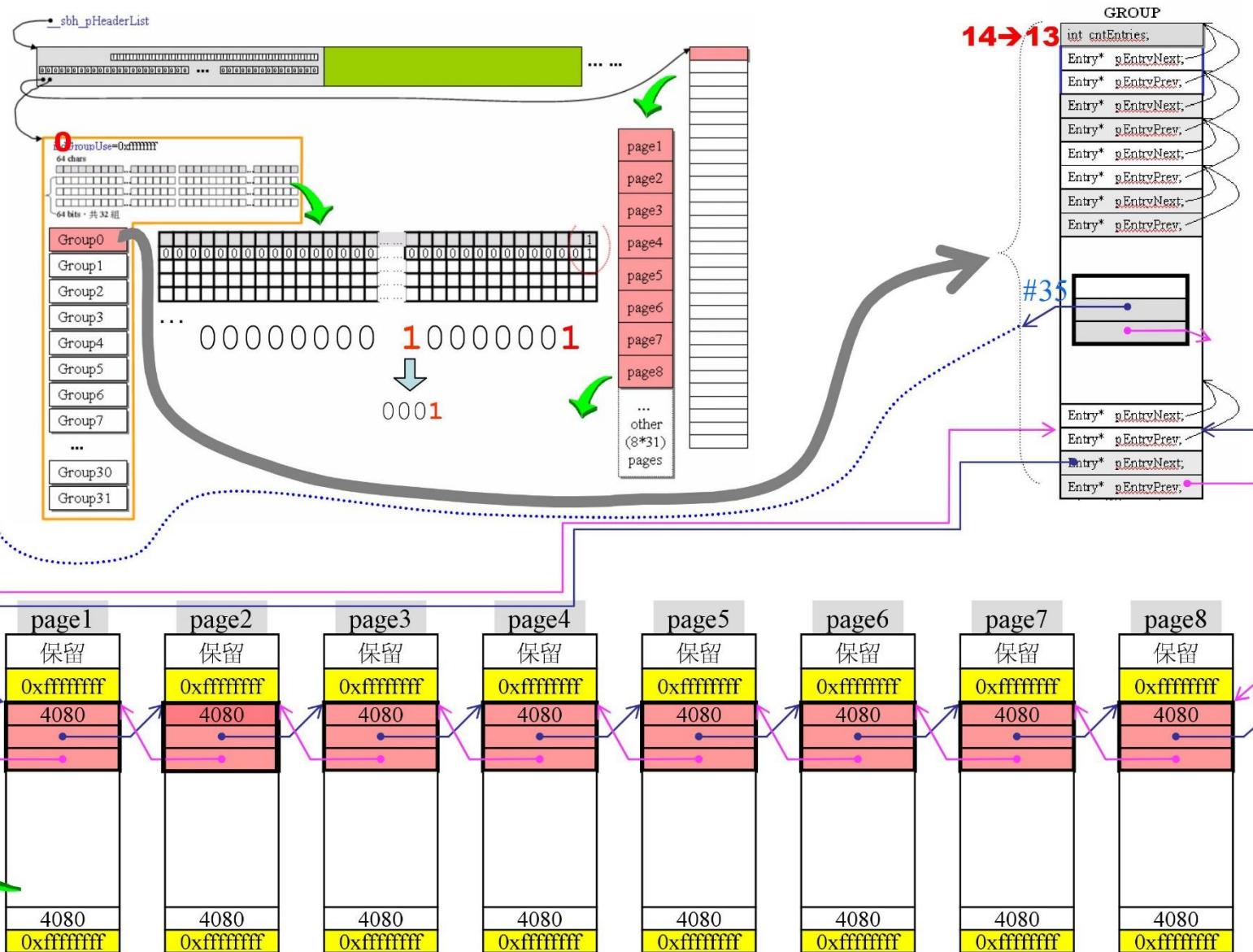
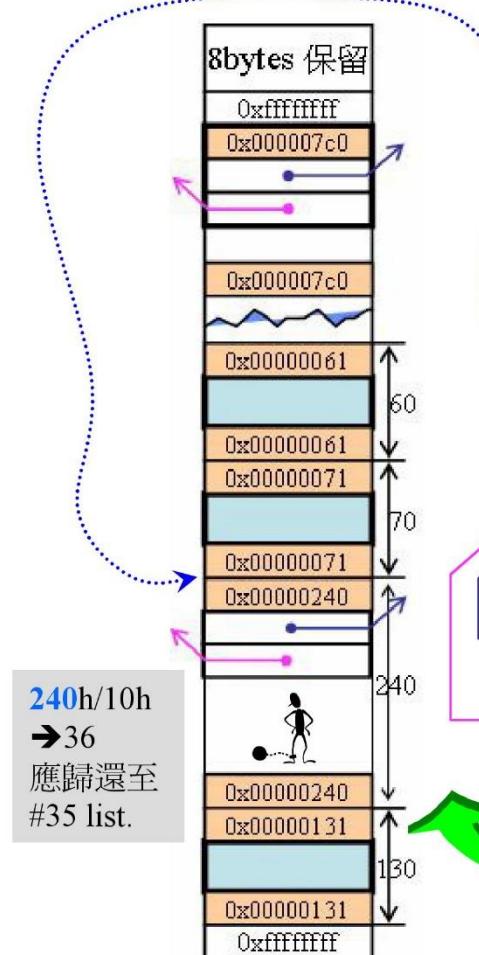
VC6 内存管理 第2次, 分配



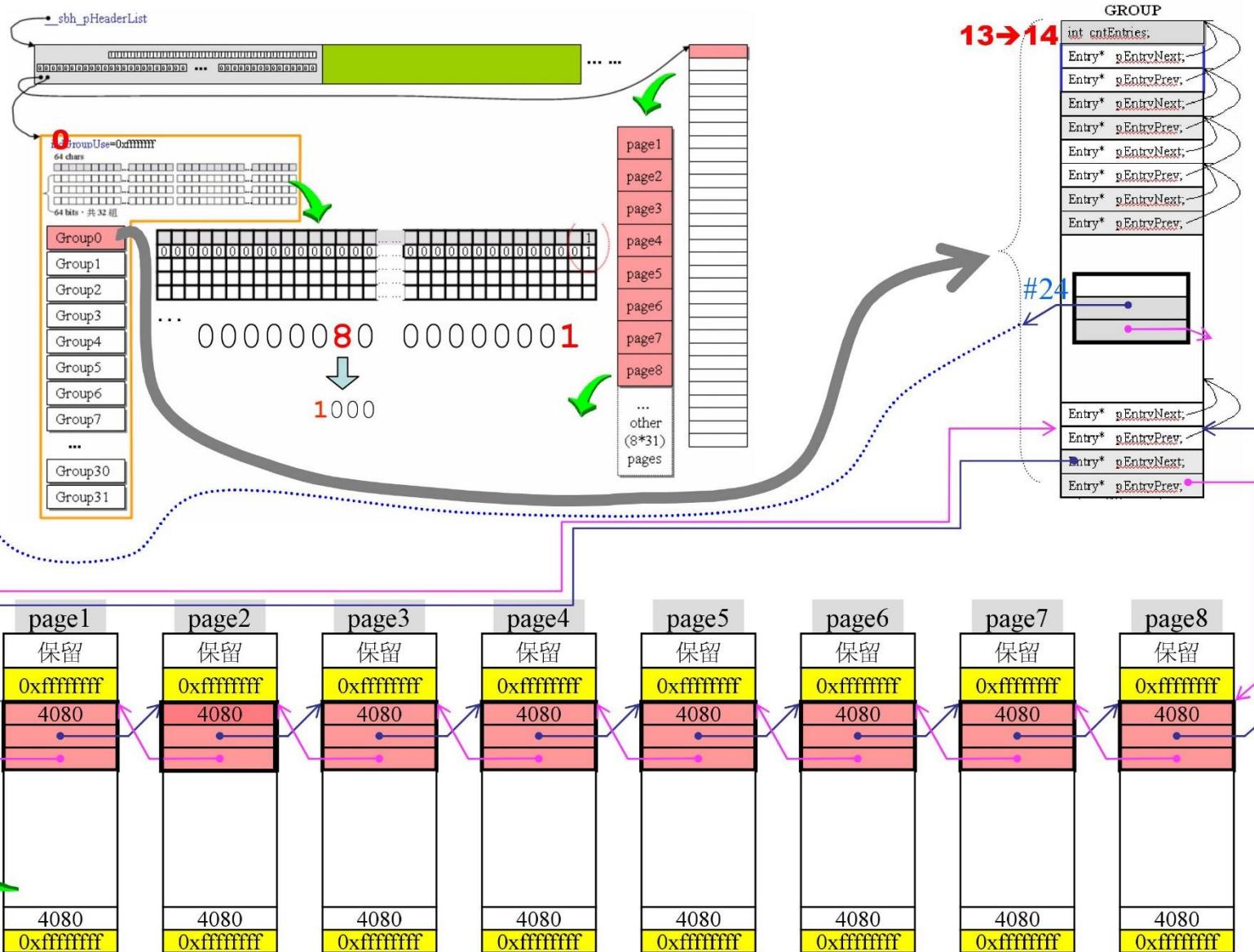
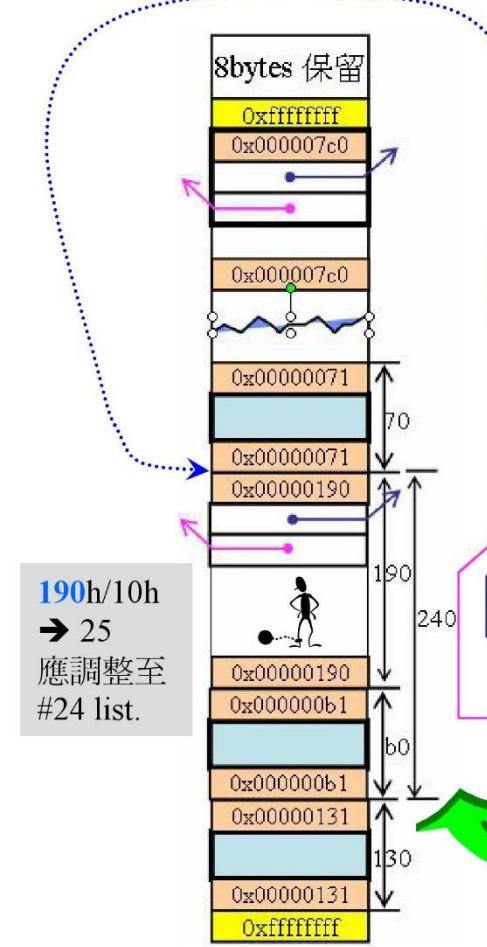
VC6 内存管理 第3次, 分配



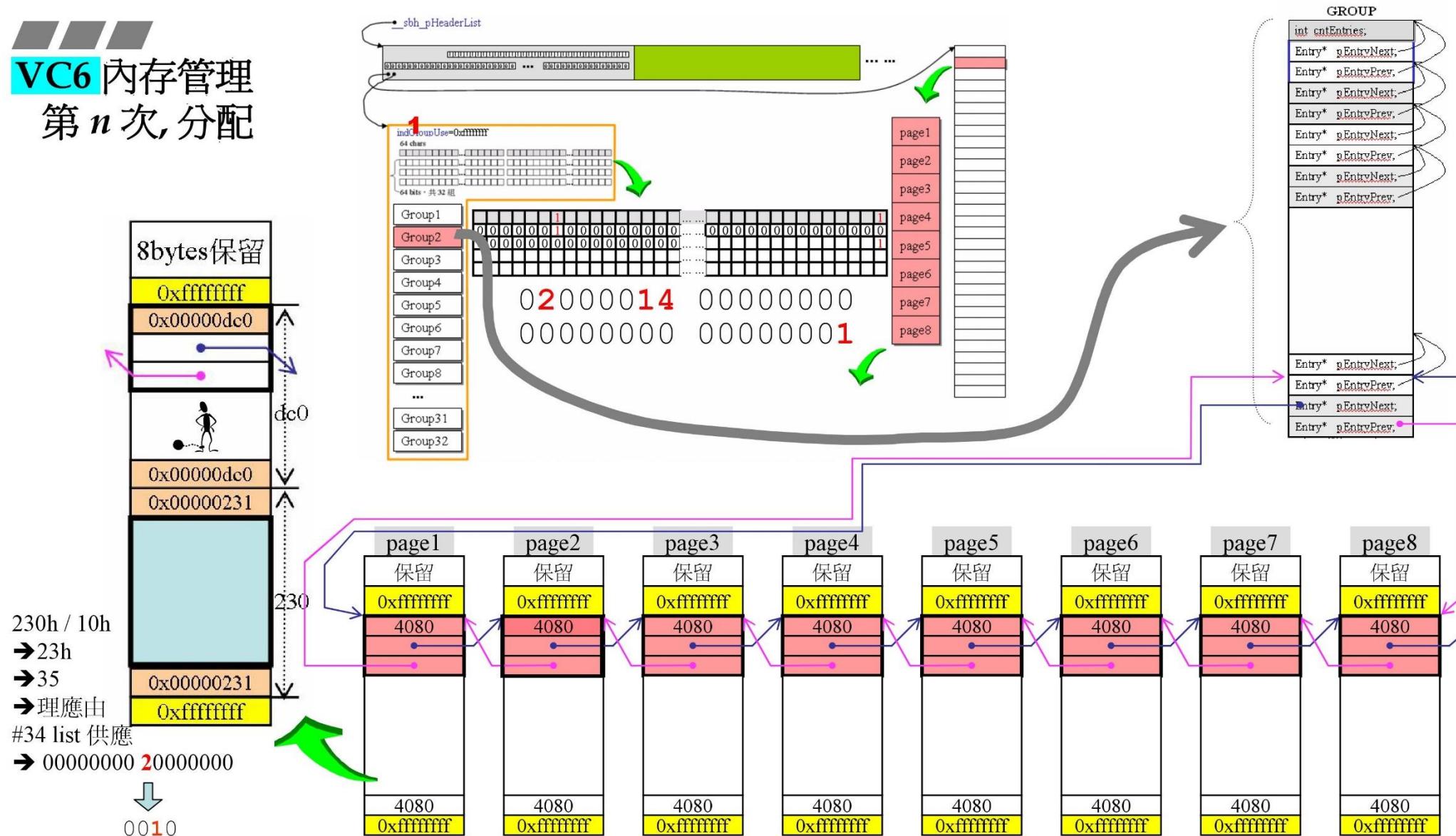
VC6 內存管理 第15次，釋還



VC6 内存管理 第16次, 分配



VC6 內存管理 第 n 次, 分配

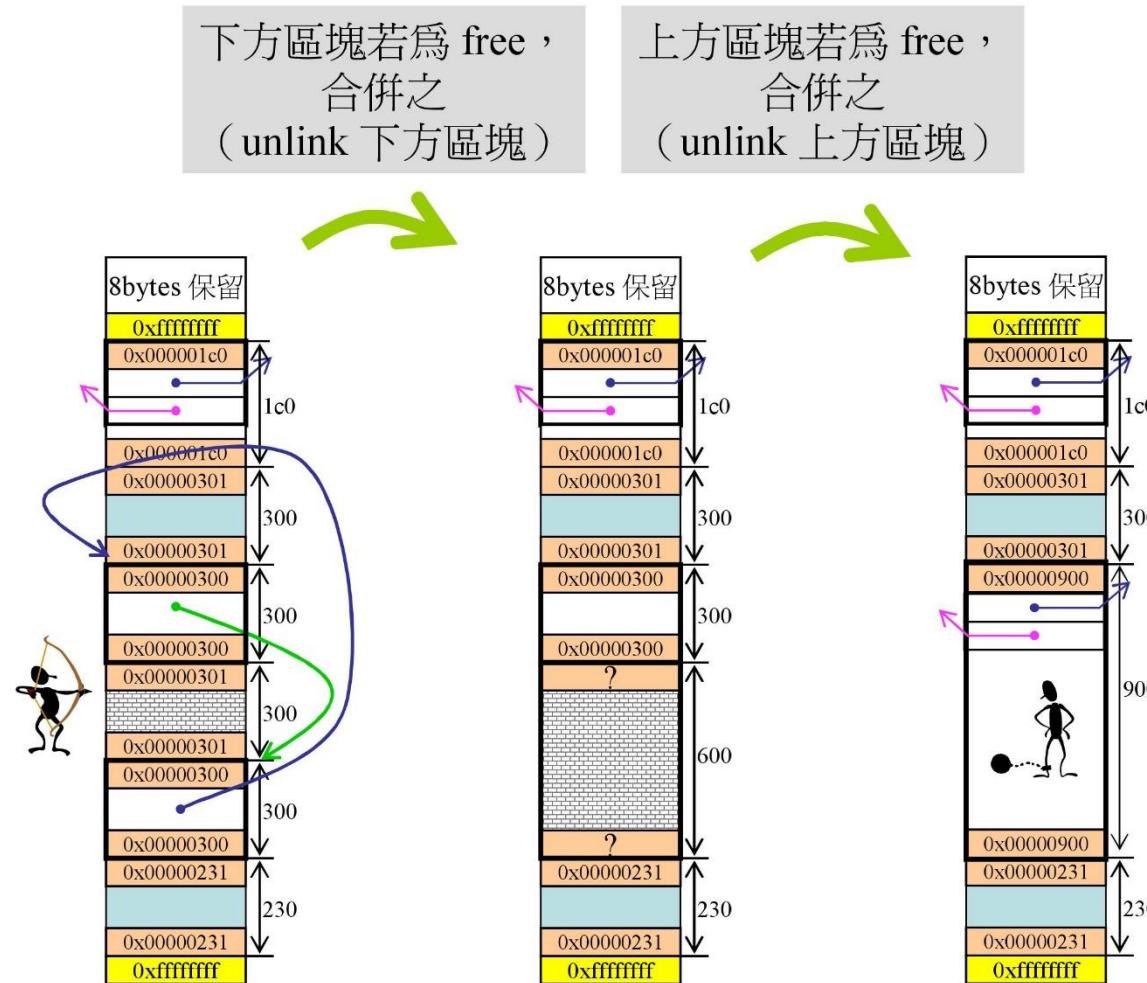




VC6 內存管理

區塊之合併

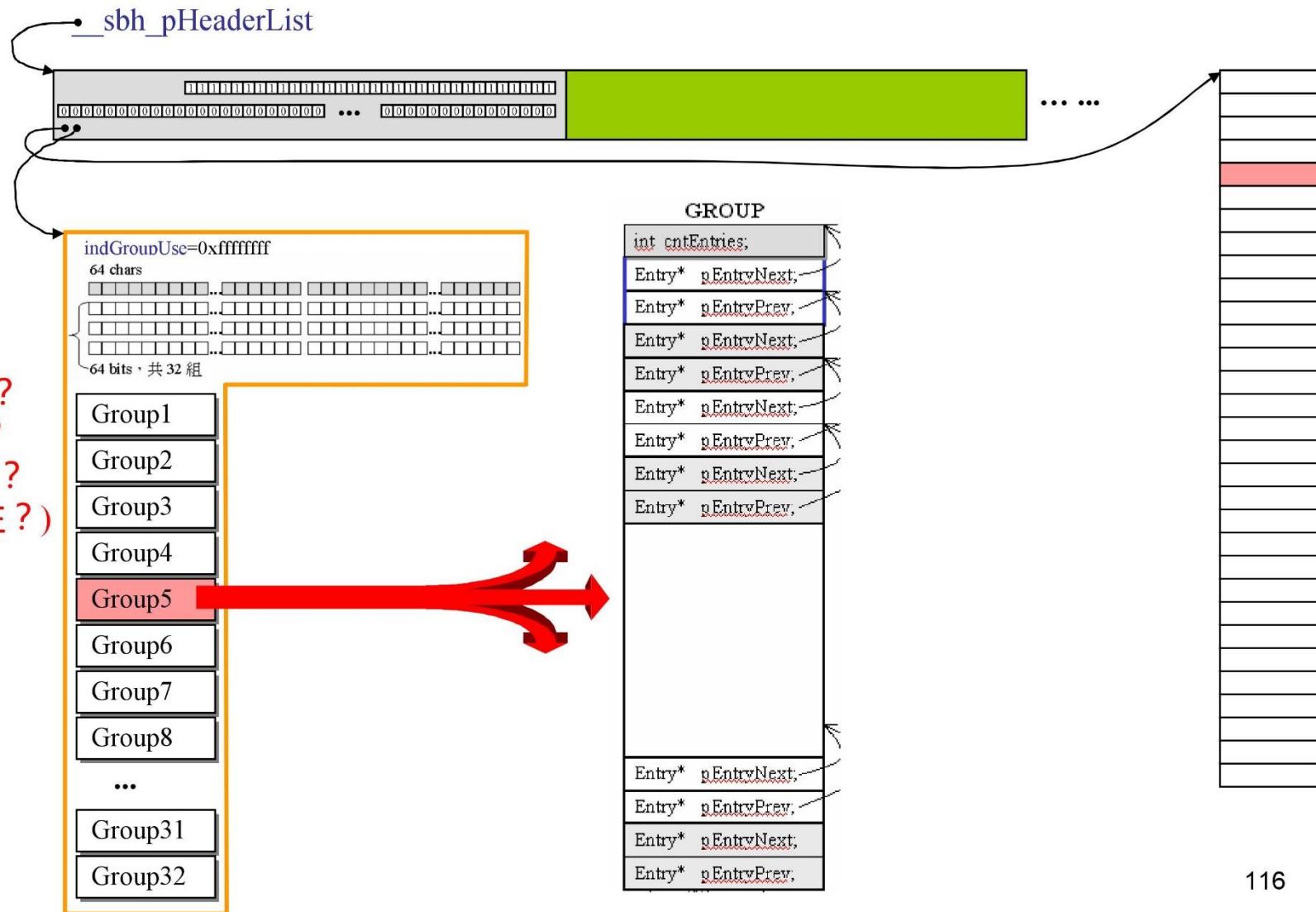
上 cookie
與
下 cookie
的作用



VC6 內存管理 free(p)

p 花落誰家？

落在哪個 Header 內？
落在哪個 Group 內？
落在哪個 free-list 內？
(被哪個 free-list 鏈住？)

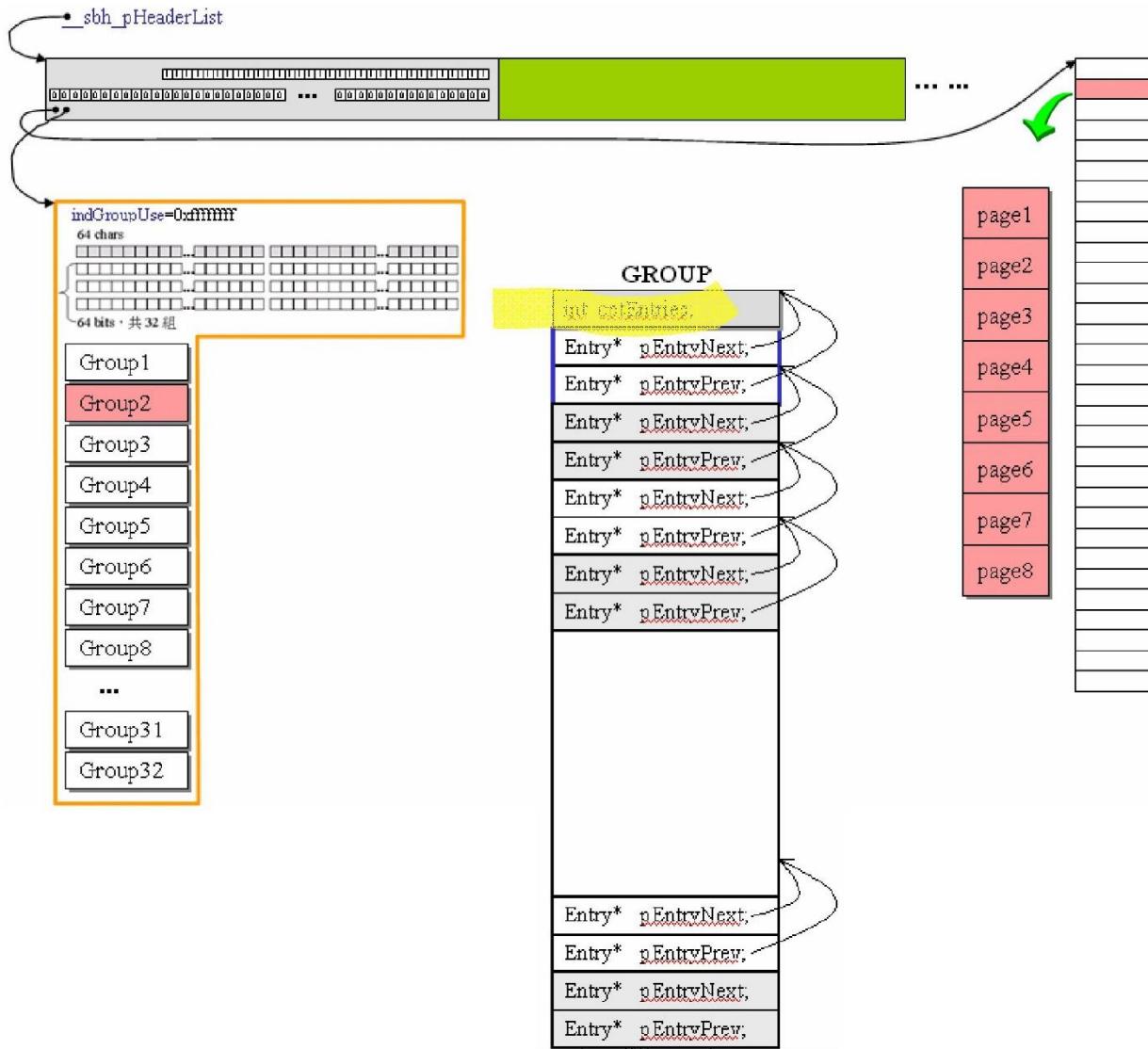




VC6 內存管理

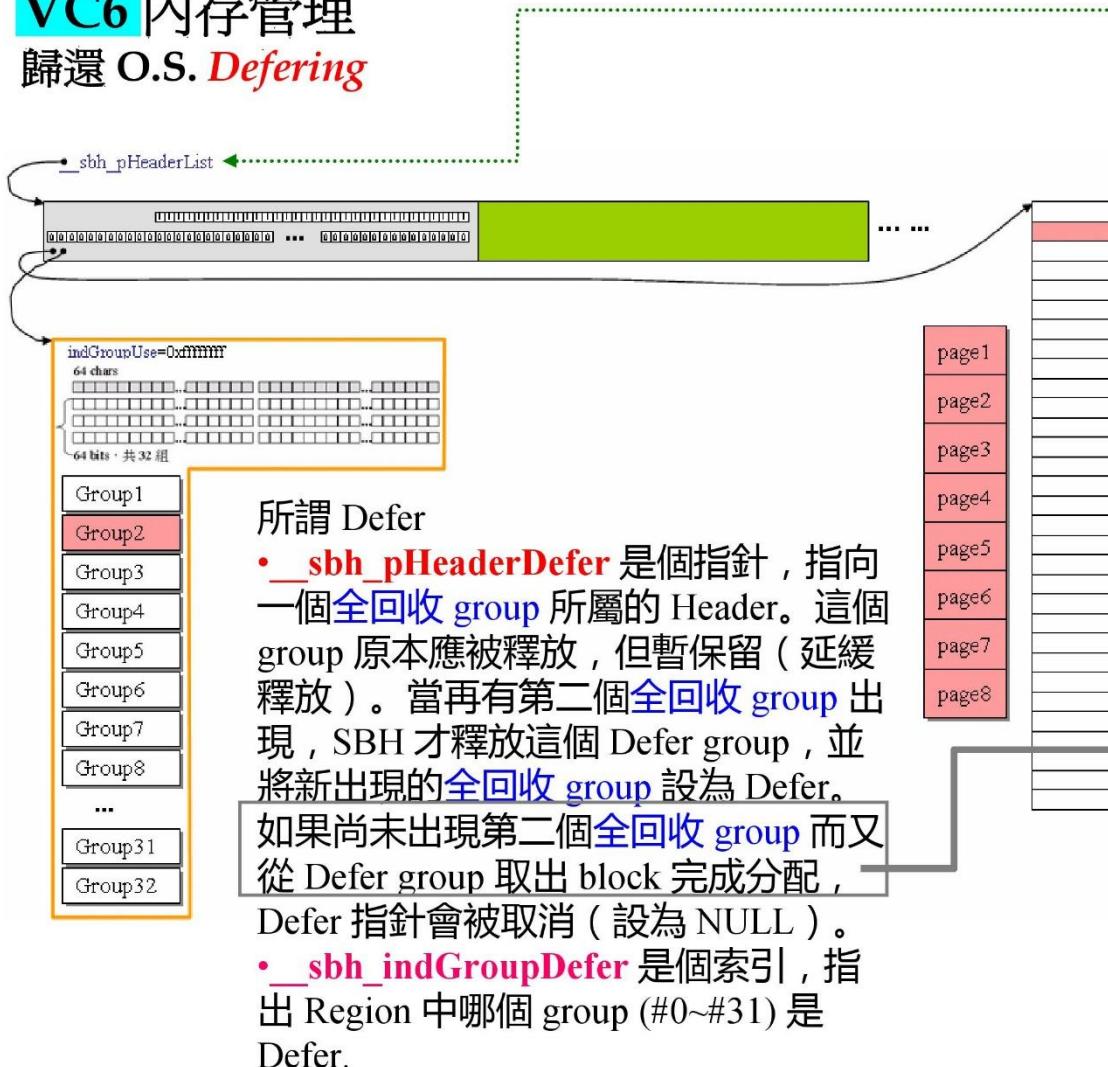
分段管理之妙 利於歸還O.S.

- 1, 如何判斷全回收？
- 2, 不要躁進！



VC6 內存管理

歸還 O.S. *Defering*



```

int __cdecl __sbh_heap_init (void)
{
    if (!(_sbh_pHeaderList =
        HeapAlloc(_crtheap, 0, 16 * sizeof(H HEADER)))
        return FALSE;

    _sbh_pHeaderScan = _sbh_pHeaderList;
    _sbh_pHeaderDefer = NULL;
    _sbh_cntHeaderList = 0;
    _sbh_sizeHeaderList = 16;

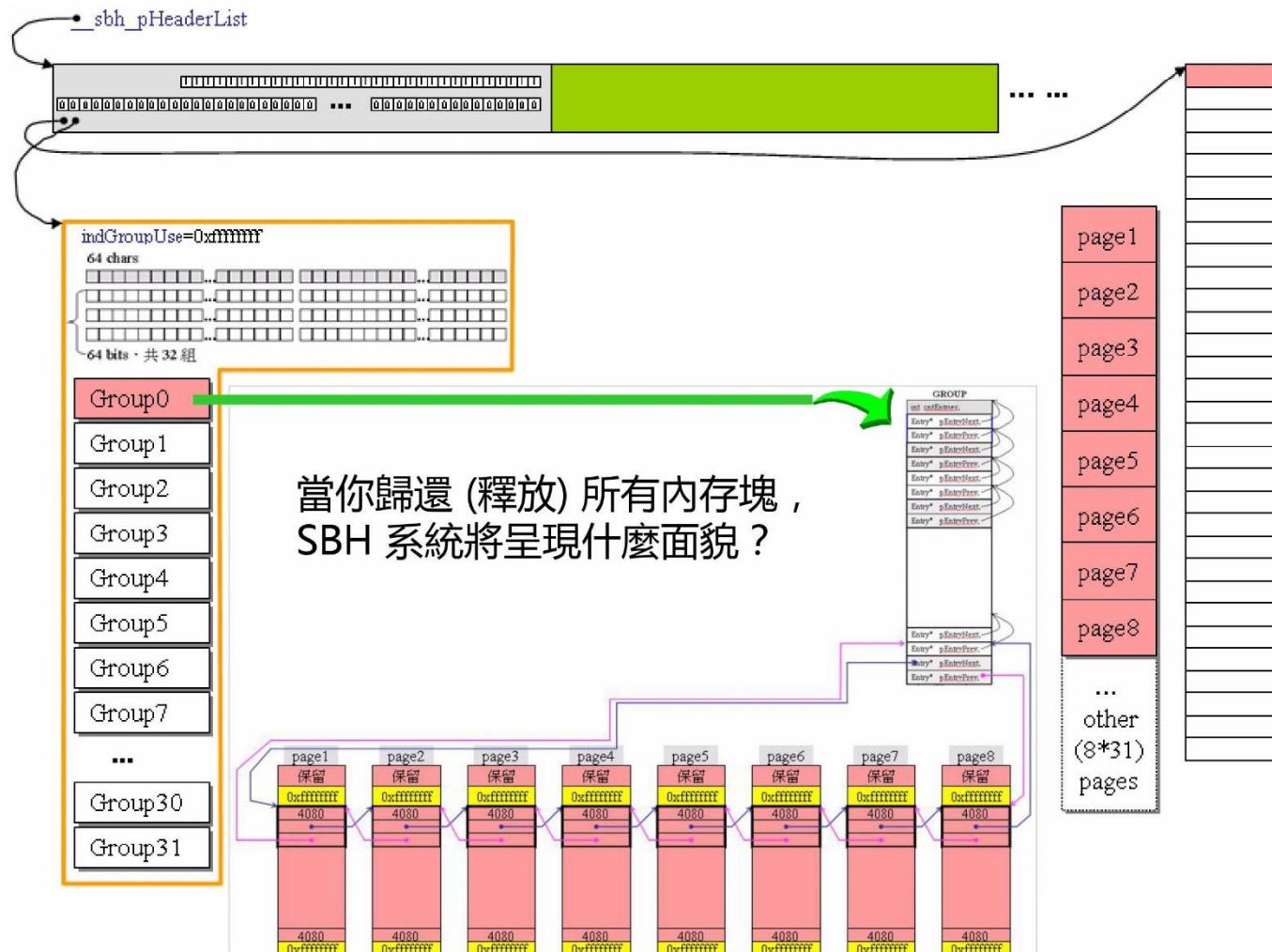
    return TRUE;
}

```

SBH 分配好 block 後，最終準備將 cntEntries 累加 1，但會先檢查它原本是否為 0；若成立，再看此次所在之 Header 是否 Defer 且此次所用之 Group 是否 Defer？都吻合就取消其 Defer 身份（令 **_sbh_pHeaderDefer = NULL**）。

此意味 Defer 必定是個全回收 Group。全新 Group 不會是 Defer，因為 Defer 指針不可能指向全新 Group。

VC6 內存管理



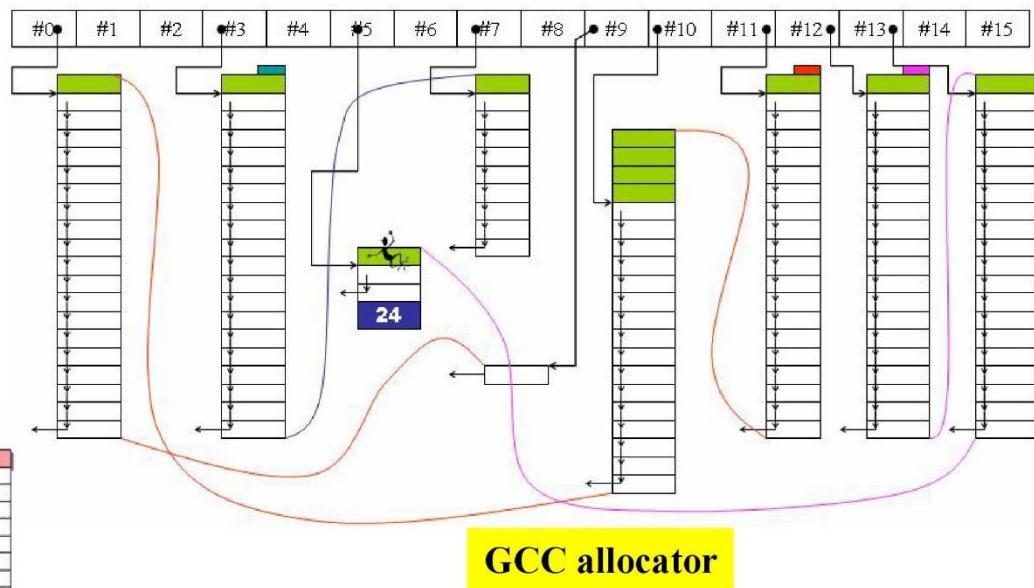
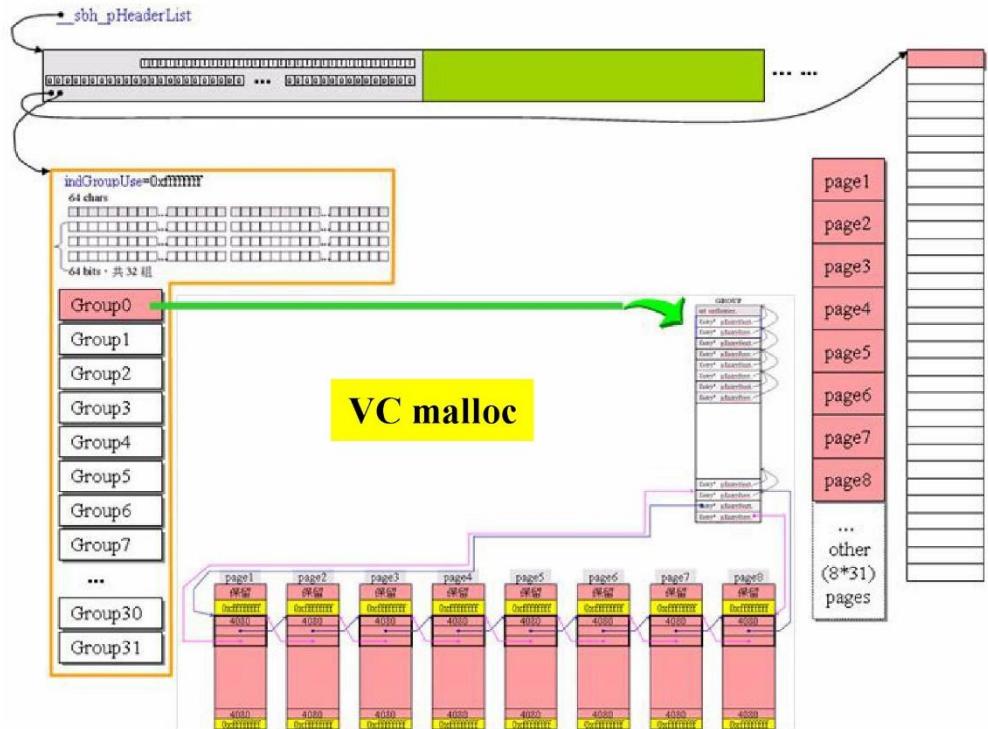
VC6, Heap State Reporting Functions

The following functions report the state and contents of the heap, and use the information to help detect memory leaks and other problems:

Function	Description
_CrtMemCheckpoint	Saves a snapshot of the heap in a _CrtMemState structure supplied by the application.
_CrtMemDifference	Compares two memory state structures, saves the difference between them in a third state structure, and returns TRUE if the two states are different.
_CrtMemDumpStatistics	Dumps a given _CrtMemState structure. The structure may contain a snapshot of the state of the debug heap at a given moment, or the difference between two snapshots. "Dumping" means reporting the data in a form that a person can understand.
_CrtMemDumpAllObjectsSince	Dumps information about all objects allocated since a given snapshot was taken of the heap, or from the start of execution. Every time it dumps a _CLIENT_BLOCK block, it calls a hook function supplied by the application, if one has been installed using _CrtSetDumpClient .
_CrtDumpMemoryLeaks	Determines whether any memory leaks occurred since the start of program execution, and if so, it dumps all allocated objects. Every time it dumps a _CLIENT_BLOCK block, it calls a hook function supplied by the application, if one has been installed using _CrtSetDumpClient .

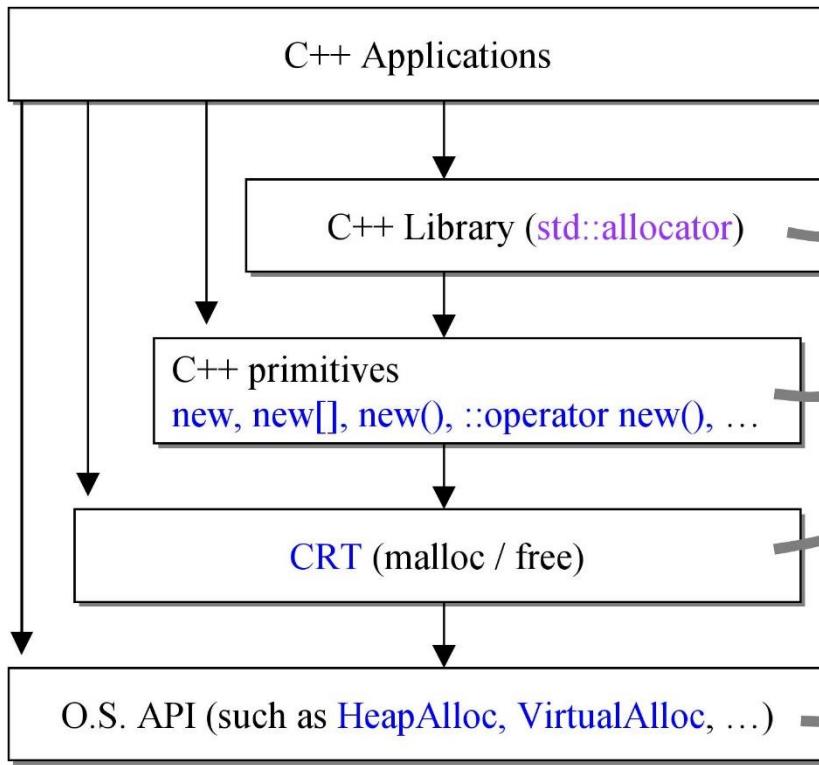


VC malloc + GCC allocator 亂點鴛鴦譜？

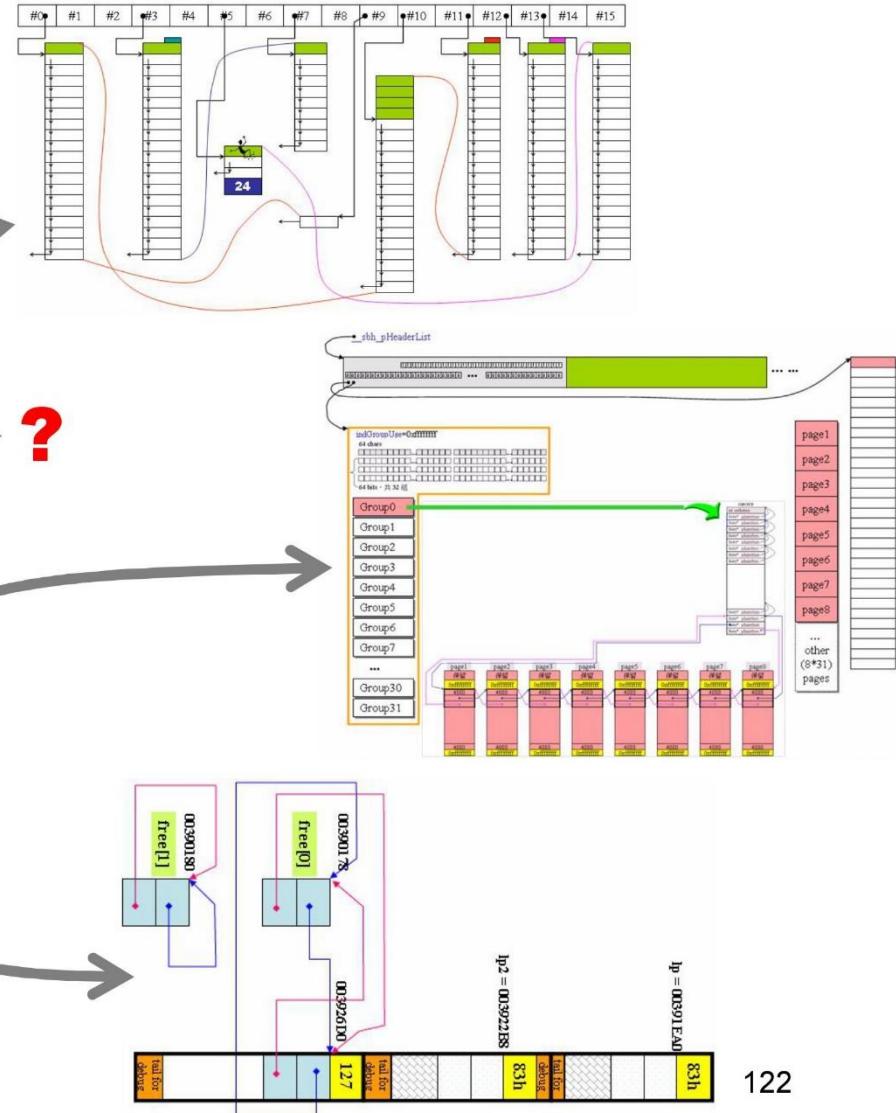




疊床架屋,有必要嗎?



—侯捷—





The End

內存管理

從平地到萬丈高樓

Memory Management 101

第一講 primitives

第二講 std::allocator

第三講 malloc/free

第四講 loki::allocator

第五講 other issues



侯捷

成竹在胸



Loki Library

sourceforge.net/projects/loki-lib/

SOURCEFORGE Search Browse Enterprise Blog Jobs

SOLUTION CENTERS Go Parallel Resources Newsletters

Home / Browse / Development / Design / Loki

Loki Beta

Brought to you by: [aandrei](#), [rich_sposato](#), [syntheticpp](#)

Summary Files Reviews Support Wiki Mailing Lists Tickets News Discussion Code

★ 5.0 Stars (6)
↓ 187 Downloads (This Week)
Last Update: 2013-04-08

sf Download loki-0.1.7.exe

Browse All Files

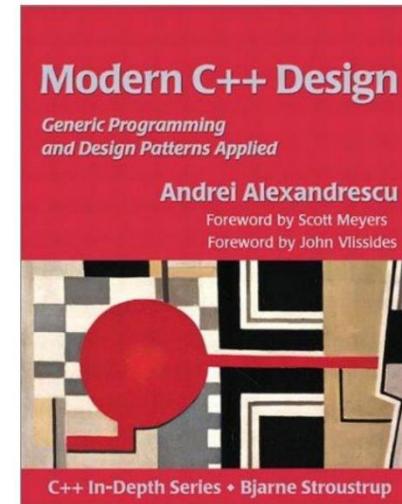
Description

A C++ library of designs, containing flexible implementations of common design patterns and idioms.

[Loki Web Site >](#)

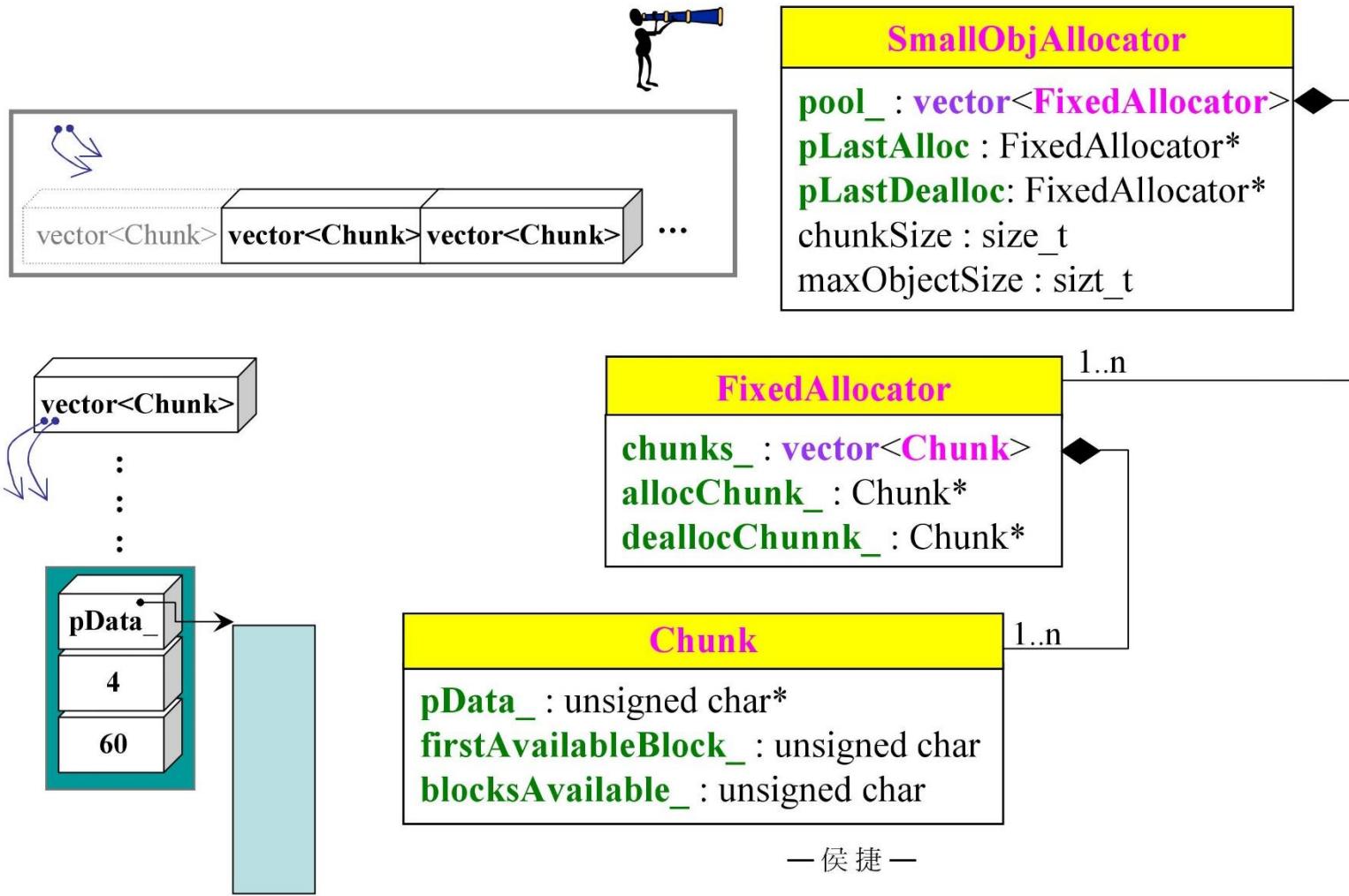
Categories License
Design MIT License

126



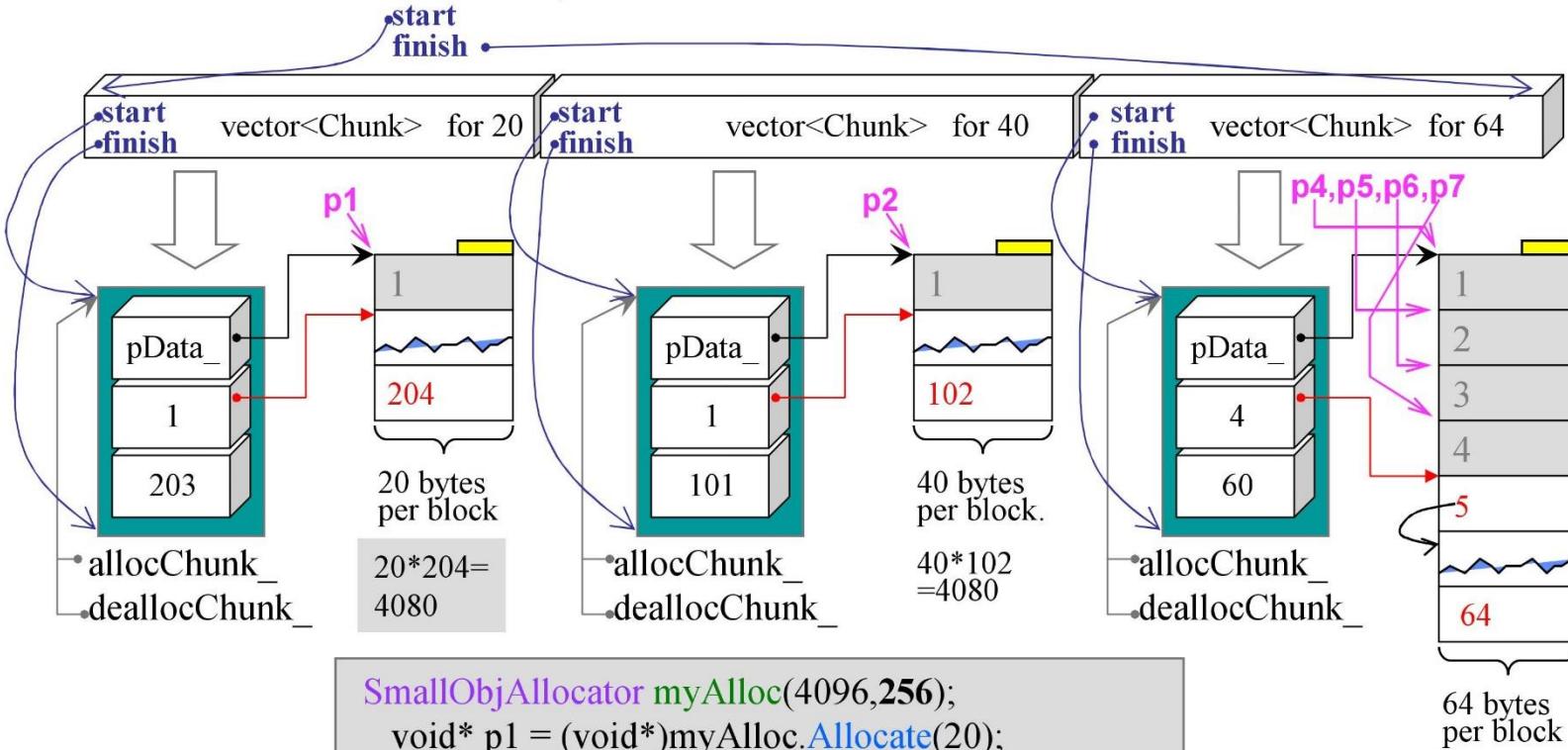


Loki allocator, 3 classes





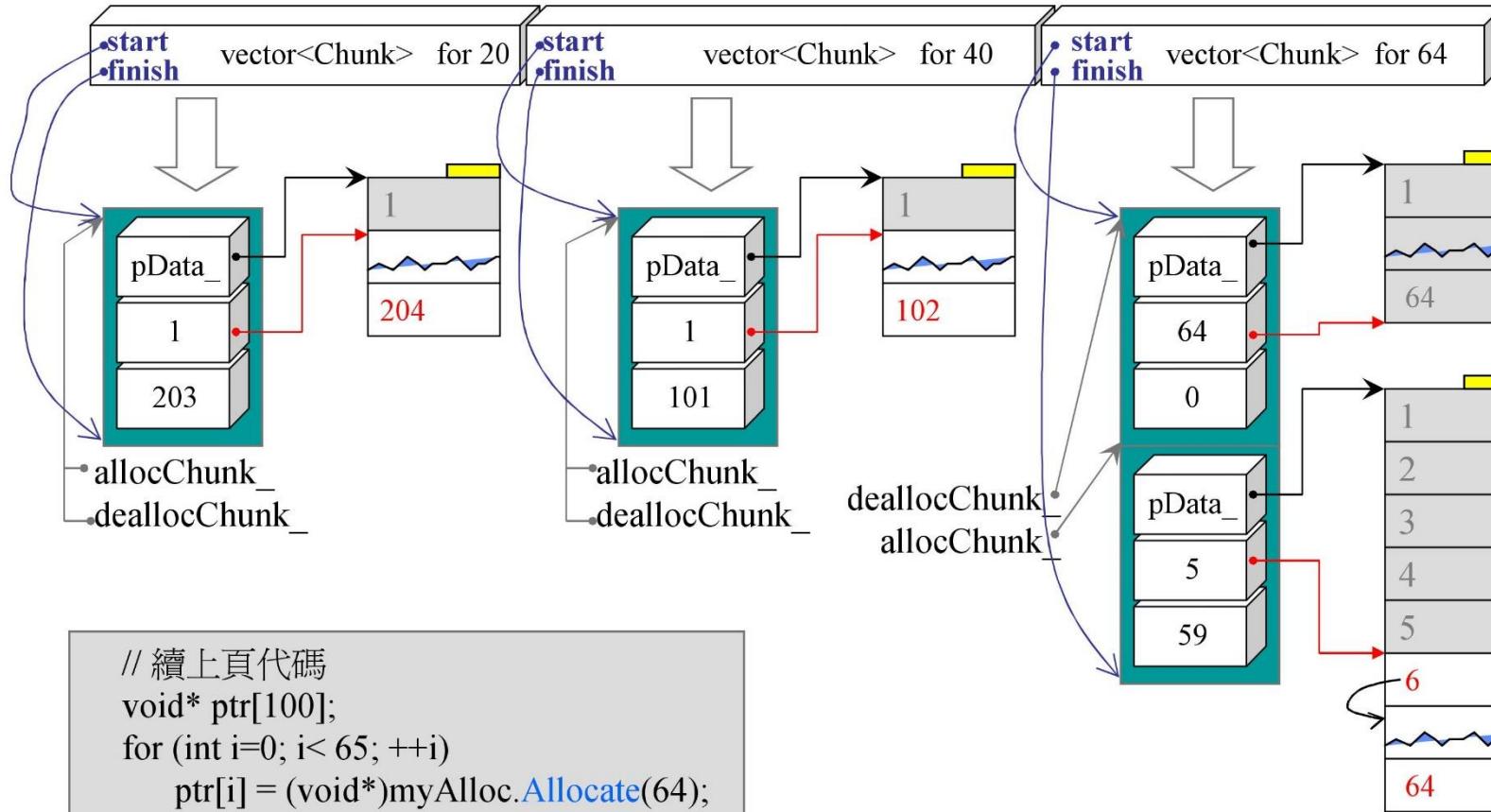
Loki allocator, 1



```
SmallObjAllocator myAlloc(4096,256);
void* p1 = (void*)myAlloc.Allocate(20);
void* p4 = (void*)myAlloc.Allocate(64);
void* p2 = (void*)myAlloc.Allocate(40);
void* p3 = (void*)myAlloc.Allocate(300); // > 256
void* p5 = (void*)myAlloc.Allocate(64);
void* p6 = (void*)myAlloc.Allocate(64);
void* p7 = (void*)myAlloc.Allocate(64);
```

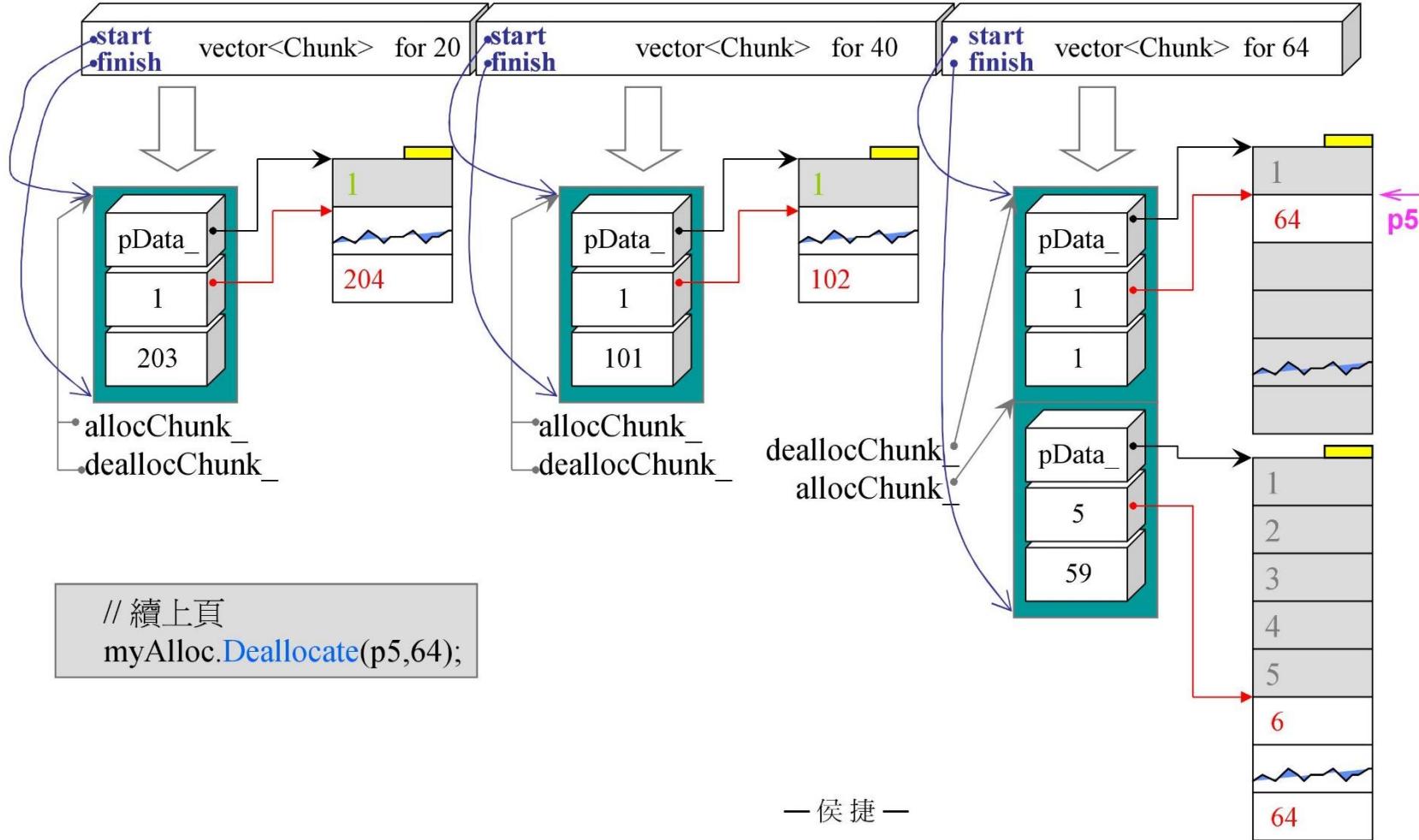


Loki allocator, 2





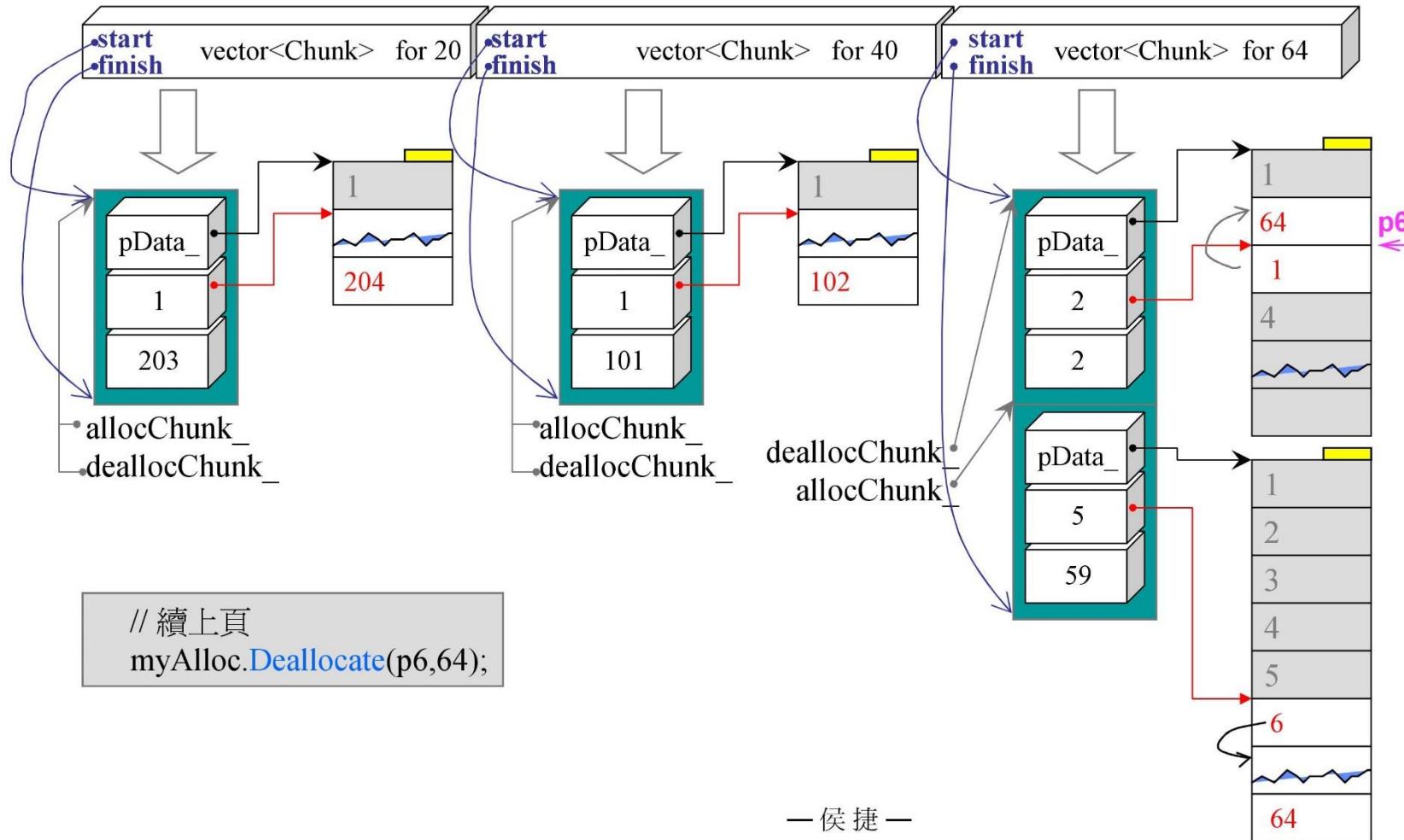
Loki allocator, 3



—侯捷—



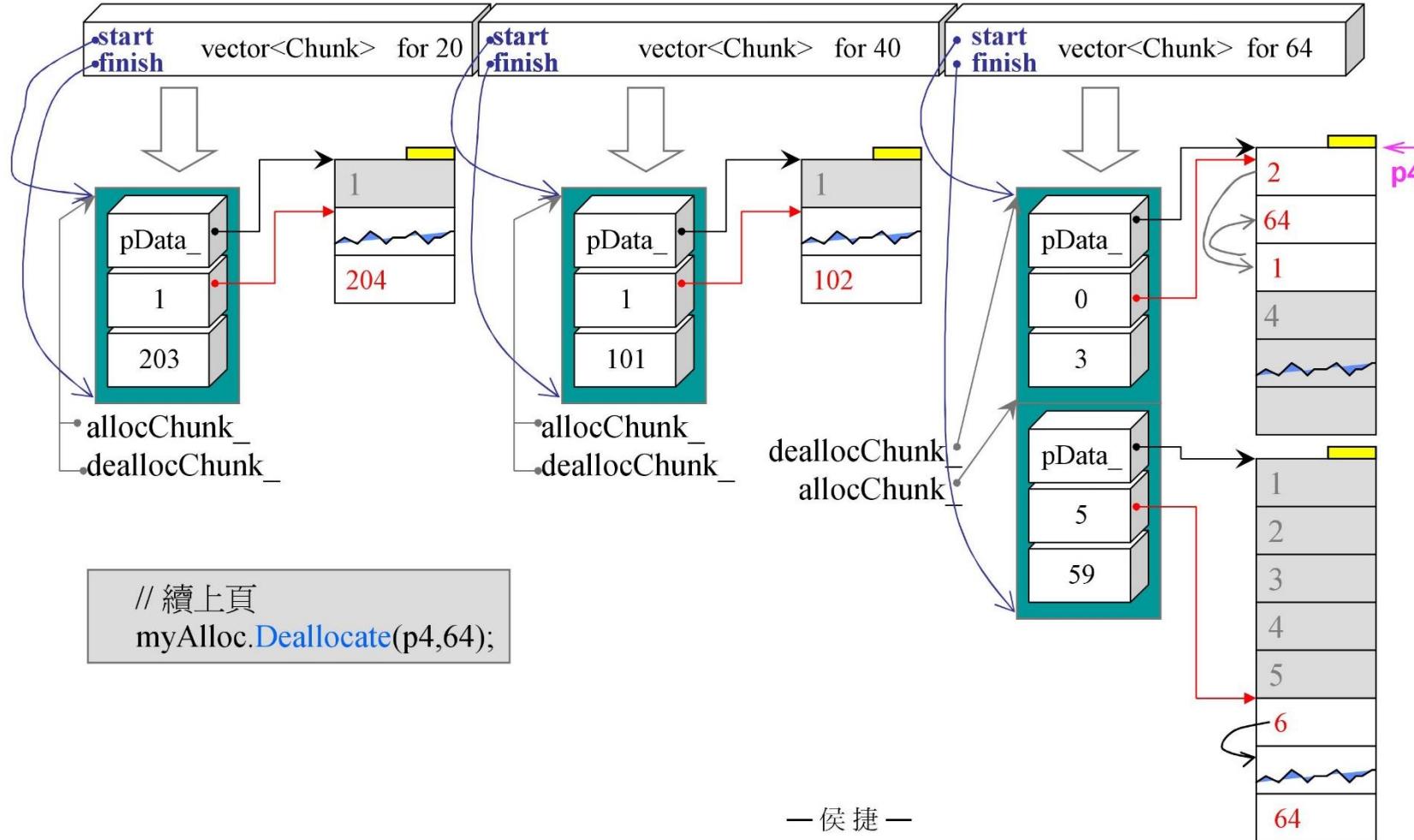
Loki allocator, 4



—侯捷—



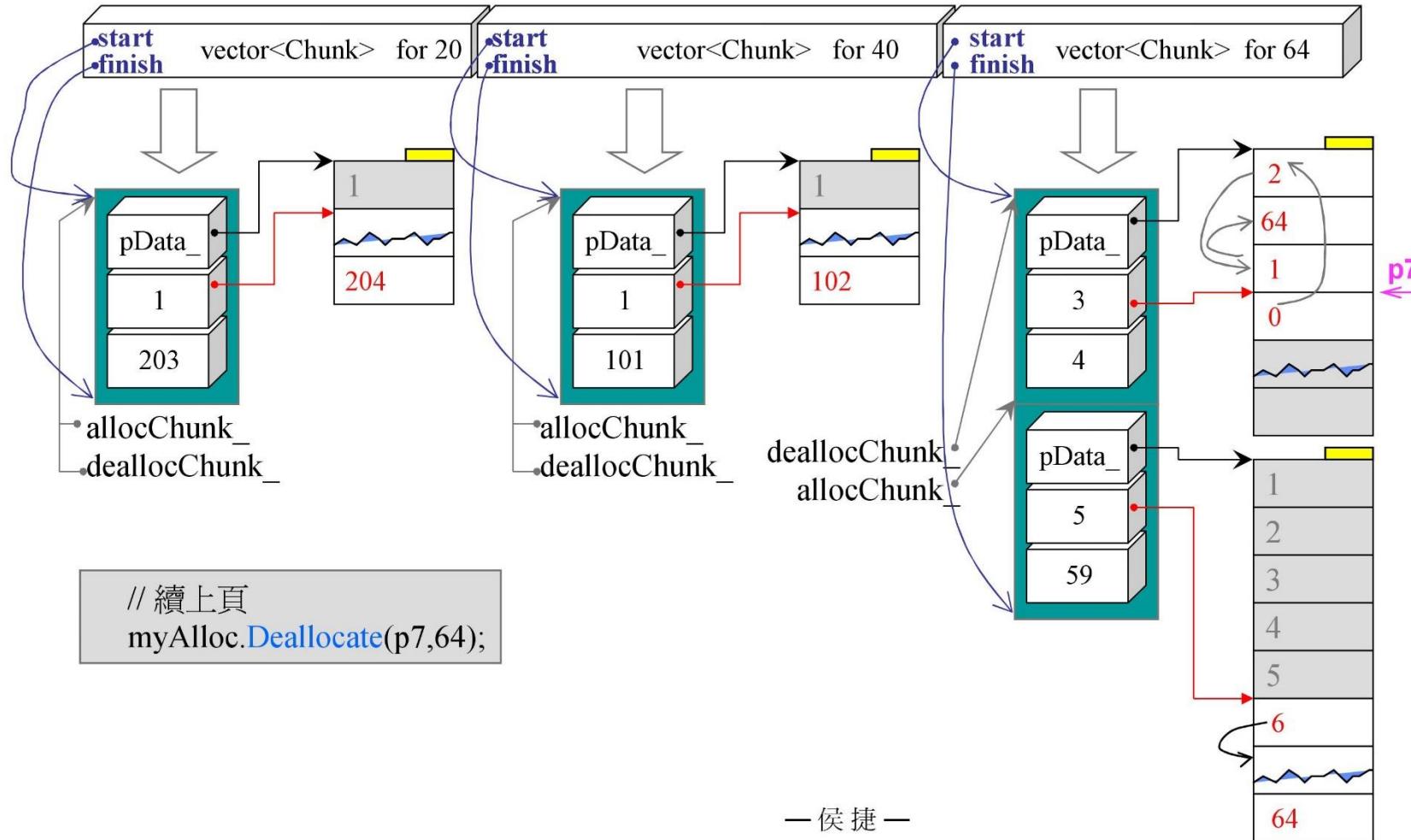
Loki allocator, 5



—侯捷—



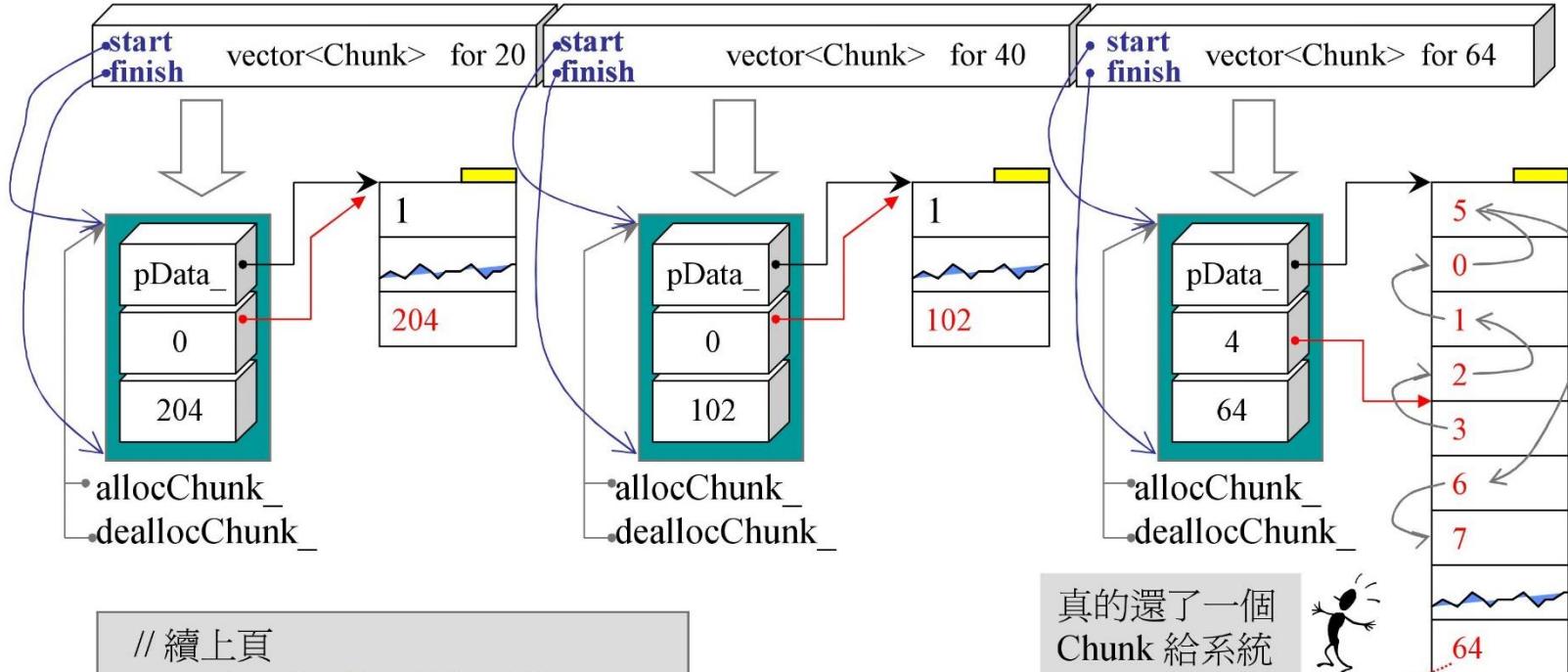
Loki allocator, 6



—侯捷—



Loki allocator, 7



```
// 續上頁
myAlloc.Deallocate(p1,20);
myAlloc.Deallocate(p2,40);
myAlloc.Deallocate(p3,300);
for (int i=0; i< 65; ++i)
    myAlloc.Deallocate(ptr[i], 64);
```

真的還了一個
Chunk 給系統

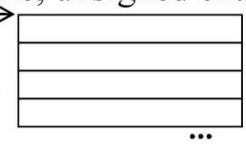


64 對本系統而言是錯誤數字，但它永遠不會被拿取（因為當它被拿取必是 blocksAvailable_ 為 0 的狀態，那就放棄拿取了）



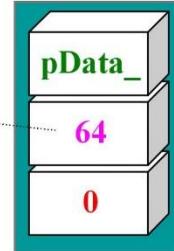
Loki allocator, Chunk

```
void FixedAllocator::Chunk::Init(std::size_t blockSize, unsigned char blocks)
{
    pData_ = new unsigned char[blockSize * blocks];
    Reset(blockSize, blocks);
}
```

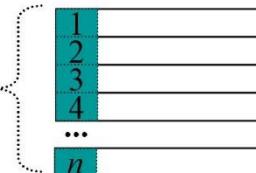


```
void FixedAllocator::Chunk::Reset(std::size_t blockSize,
                                  unsigned char blocks)
{
    firstAvailableBlock_ = 0;
    blocksAvailable_ = blocks;

    unsigned char i = 0;
    unsigned char* p = pData_;
    for (; i != blocks; p += blockSize) //流水號標示索引
        *p = ++i;
}
```

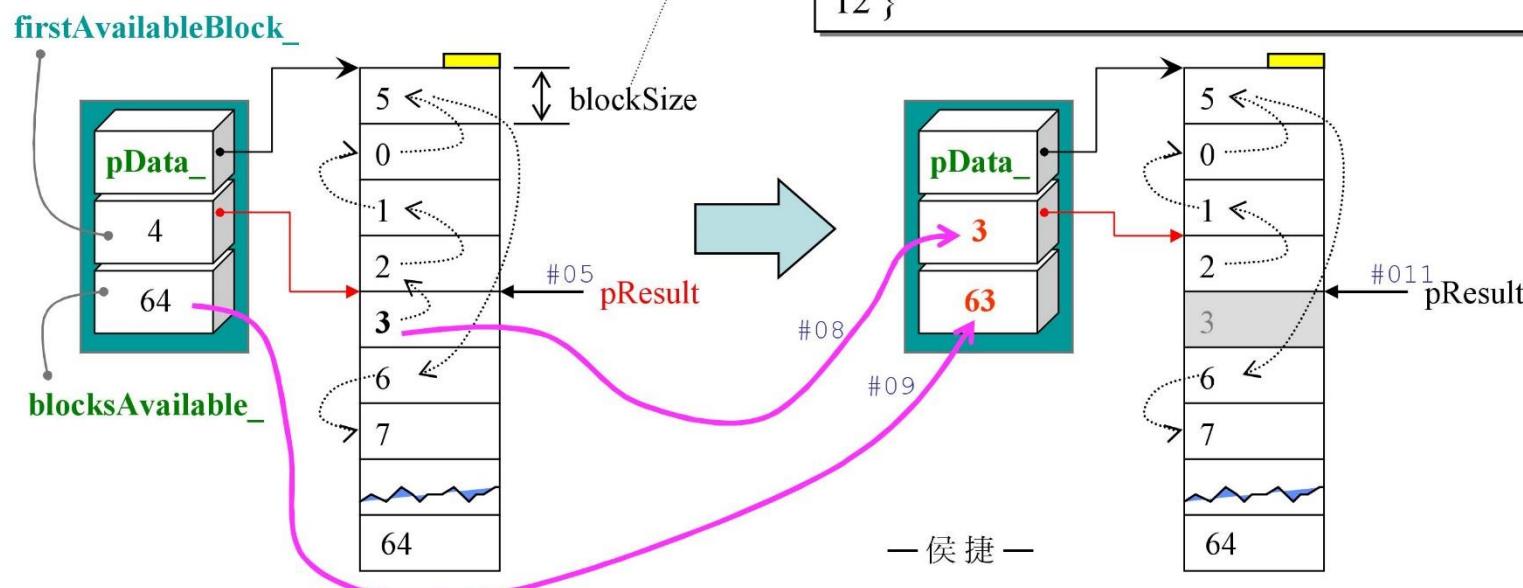


```
void FixedAllocator::Chunk::Release()
{
    delete[] pData_; //釋放自己
} //此函數被上一層調用
```





Chunk::Allocate()

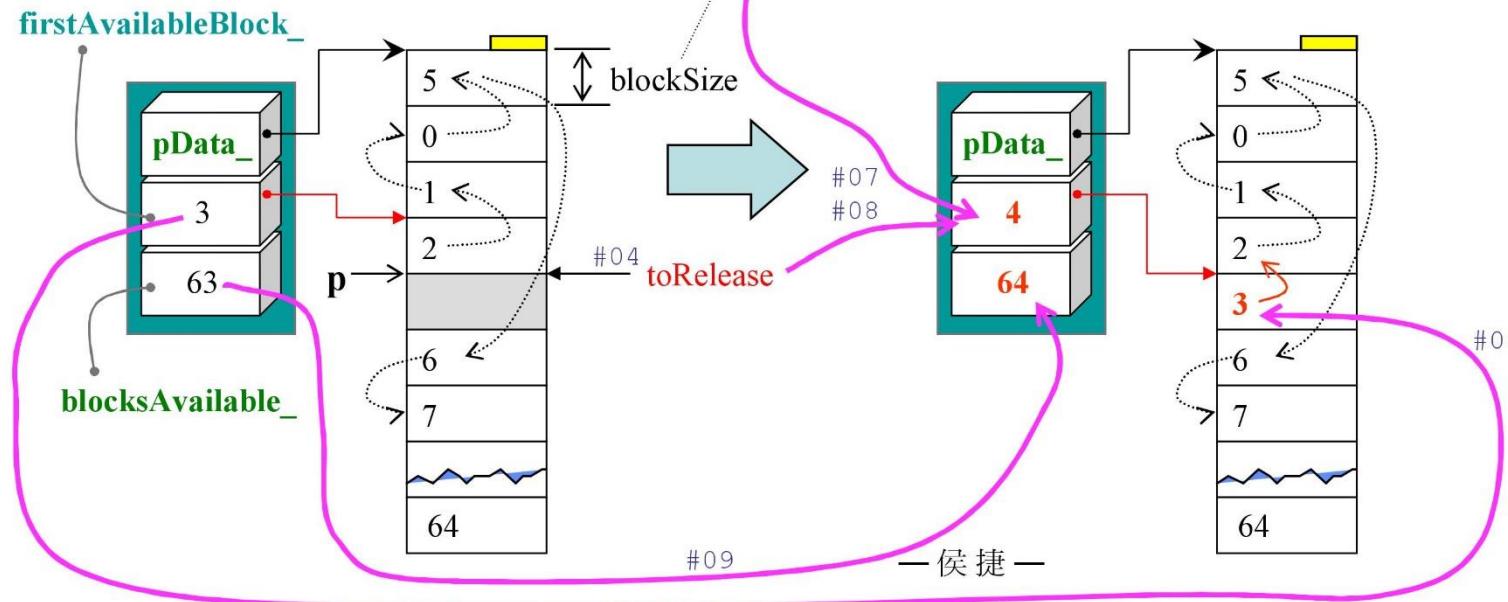


```
01 void* FixedAllocator::Chunk::Allocate(std::size_t blockSize)
02 {
03     if (!blocksAvailable_) return 0; //此地無銀
04
05     unsigned char* pResult = //指向第一個可用區塊
06     pData_ + (firstAvailableBlock_ * blockSize);
07
08     firstAvailableBlock_ = *pResult; //右側索引便是下一個可用區塊
09     --blocksAvailable_;
10
11     return pResult;
12 }
```



Chunk::Deallocate()

```
01 void FixedAllocator::Chunk::Deallocate(void* p,
02                                         std::size_t blockSize)
03 {
04     unsigned char* toRelease = static_cast<unsigned char*>(p);
05
06     *toRelease = firstAvailableBlock_;
07     firstAvailableBlock_ = static_cast<unsigned char>(
08         (toRelease - pData_) / blockSize);
09     ++blocksAvailable_; //可用區塊數加 1
10 }
```



FixedAllocator::Allocate()

```
#01 void* FixedAllocator::Allocate()
#02 {
#03     if (allocChunk_ == 0 || allocChunk_->blocksAvailable_ == 0)
#04     { //目前沒有標定chunk或該chunk已無可用區塊
#05         Chunks::iterator i = chunks_.begin(); //打算從頭找起
#06         for (; ++i) //找遍每個chunk 直至找到擁有可用區塊者.
#07     {
#08         if (i == chunks_.end()) //到達尾端，都沒找著
#09         {
#10             // Initialize 創建 temp object 拷貝至容器然後結束生命
#11             chunks_.push_back(Chunk()); //產生 a new chunk 掛於末端
#12             Chunk& newChunk = chunks_.back(); //指向末端 chunk
#13             newChunk.Init(blockSize_, numBlocks_); //設好索引
#14             allocChunk_ = &newChunk; //標定,稍後將對此 chunk 取區塊
#15             deallocChunk_ = &chunks_.front(); //另一標定
#16             break;
#17         }
#18         if (i->blocksAvailable_ > 0)
#19         { //current chunk 有可用區塊
#20             allocChunk_ = &*i; //取其位址
#21             break; //不找了,退出 for-loop
#22         }
#23     }
#24 }
#25 return allocChunk_->Allocate(blockSize_); //向這個chunk 取區塊
#26 }
```

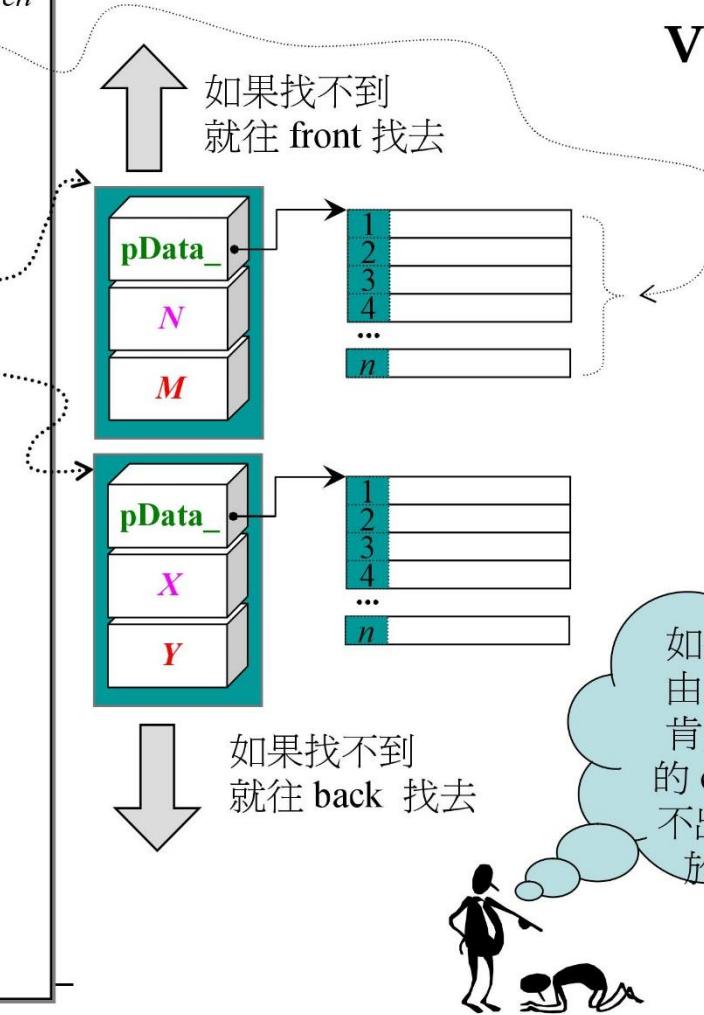
在此 chunk 找到可用區塊，
下次就優先由此找起

FixedAllocator::Deallocate()

```
void FixedAllocator::Deallocate(void* p)
{
    deallocChunk_ = VicinityFind(p);
    DoDeallocate(p);
}
```

FixedAllocator::VicinityFind()

```
#01 // FixedAllocator::VicinityFind (internal)
#02 // Finds the chunk corresponding to a pointer, using an efficient search
#03 FixedAllocator::Chunk* FixedAllocator::VicinityFind(void* p)
#04 {
#05     const std::size_t chunkLength = numBlocks_ * blockSize_;
#06
#07     Chunk* lo = deallocChunk_;
#08     Chunk* hi = deallocChunk_ + 1;
#09     Chunk* loBound = &chunks_.front();
#10    Chunk* hiBound = &chunks_.back() + 1;
#11
#12    for (;;) { //兵分兩路
#13        if (lo) { //上路未到盡頭
#14            if (p >= lo->pData_ && p < lo->pData_ + chunkLength)
#15                return lo; //p 落於 lo 區間內
#16            if (lo == loBound) lo = 0; //到頂了，讓 loop 結束
#17            else --lo; //否則往 lo 的 front 繼續找
#18        }
#19
#20        if (hi) { //下路未到盡頭
#21            if (p >= hi->pData_ && p < hi->pData_ + chunkLength)
#22                return hi; //p 落於 hi 區間內
#23            if (++hi == hiBound) hi = 0; //往 hi 的 back 繼續找
#24            //若到最尾則讓 loop 結束
#25        }
#26    }
#27 }
```



FixedAllocator:: DoDeallocate()

```

#0001 void FixedAllocator::DoDeallocate(void* p)
#0002 {
#0003     //
#0004     deallocChunk_->Deallocate(p, blockSize_);
#0005
#0006     if (deallocChunk_->blocksAvailable_ == numBlocks_) {
#0007         // 確認全回收, 那麼, 該 release 它嗎?
#0008         // 注意, 以下三種情況都重新設定了allocChunk_
#0009         Chunk& lastChunk = chunks_.back(); //標出最後一個
#0010         //如果“最後一個”就是“當前 chunk”
#0011         if (&lastChunk == deallocChunk_) {
#0012             ① // 檢查是否擁有兩個 last chunks empty
#0013                 // 這只需往 front 方向看前一個 chunk 便知.
#0014                 if (chunks_.size() > 1 &&
#0015                     deallocChunk_[-1].blocksAvailable_ == numBlocks_) {
#0016                     // 是的, 有 2 free chunks, 拋棄最後一個
#0017                     lastChunk.Release();
#0018                     chunks_.pop_back();
#0019                     allocChunk_ = deallocChunk_ = &chunks_.front();
#0020                 }
#0021             return;
#0022         }
#0023     }

```



```

#0024     if (lastChunk.blocksAvailable_ == numBlocks_) {
#0025         ② //兩個 free chunks, 拋棄最後一個
#0026         lastChunk.Release();
#0027         chunks_.pop_back();
#0028         allocChunk_ = deallocChunk_;
#0029     }
#0030     else {
#0031         ③ //將free chunk 與最後一個chunk 對調
#0032         std::swap(*deallocChunk_, lastChunk);
#0033         allocChunk_ = &chunks_.back();
#0034     }
#0035 }
#0036 }

```



Loki allocator 檢討

- 曾經有兩個 bugs, 新版已修正.
- 精簡強悍; 手段暴力 (關於 for-loop).
- 使用「以 array 取代 list, 以 index 取代 pointer」的特殊實現手法.
- 能夠以很簡單的方式判斷「chunk 全回收」進而將 memory 歸還給操作系統.
- 有 Deferring (暫緩歸還) 能力
- 這是個 allocator , 用來分配大量小塊不帶 cookie 的 memory blocks , 它的最佳客戶是容器 , 但它本身卻使用 vector , What happens ? 雞生蛋 蛋生雞 ?



The End

內存管理

從平地到萬丈高樓

Memory Management 101

第一講 primitives

第二講 std::allocator

第三講 malloc/free

第四講 loki::allocator

第五講 other issues



侯捷