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。

内存管理

從平地到萬丈高樓

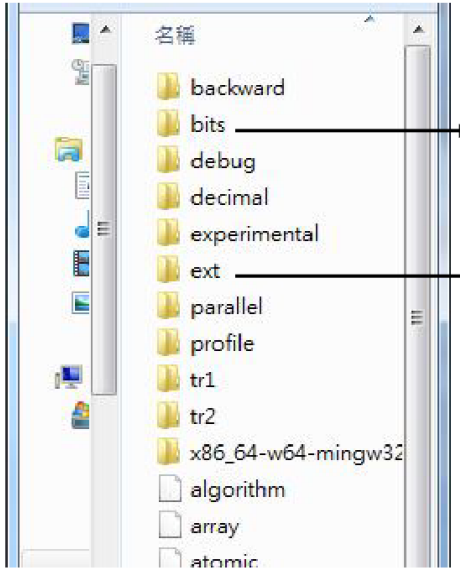
Memory Management 101

第一講 primitives
第二講 std::allocator
第三講 malloc/free
第四講 loki::allocator
第五講 the others

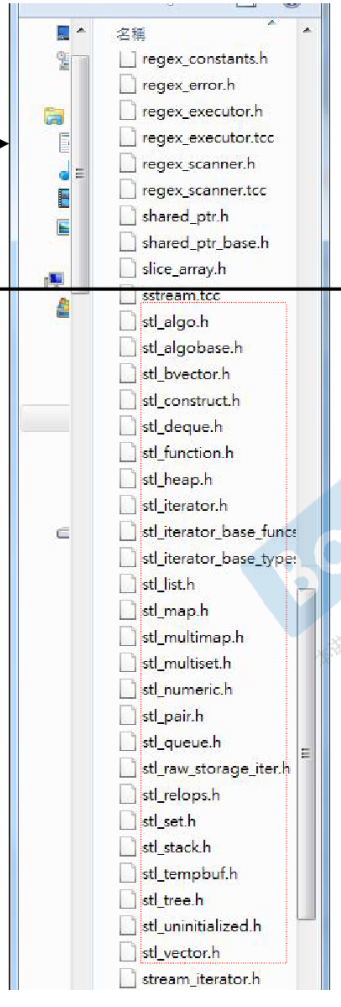
侯捷

GNU C++

…\4.9.2\include\c++



…\include\c++\bits



…\include\c++\ext

GNU C++ 對於 Allocator 的描述

當你將元素加入容器中，容器必須分配更多內存以保存這些元素，於是它們向其模板參數 **Allocator** 發出申請，該模板參數往往被另名為 (*aliased to*) allocator_type。甚至你將 chars 添加到 string class 也是如此，因為 string 也算是一個正規的 STL 容器。

每個元素類型為 T 的容器 (container-of-T) 的 Allocator 模板實參默認為 **allocator<T>.**。其接口只有大約 20 個 public 聲明，包括嵌套的 (nested) typedefs 和成員函數。最重要的兩個成員函數是：

T* **allocate** (size_type n, const void* hint = 0);
void **deallocate** (T* p, size_type n);

n 指的是客戶申請的元素個數，不是指空間總量。

這些空間是通過調用 ::operator new 獲得，但何時調用以及多麼頻繁調用，並無具體指定。

```
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;
```

GNU C++ 對於 Allocator 的描述

最容易滿足需求的作法就是每當容器需要內存就調用 operator new，每當容器釋放內存就調用 operator delete。這種作法比起分配大塊內存並緩存 (caching) 然後徐徐小塊使用當然較慢，優勢則是可以在極大範圍的硬件和操作系統上有效運作。

- **__gnu_cxx::new_allocator**

  實現出簡樸的 operator new 和 operator delete 語意。

```
template<typename _Tp>
class new_allocator
{
  .....
  pointer allocate(size_type __n, const void* = 0) {
    ……
    return static_cast<_Tp*>
        (::operator new(__n * sizeof(_Tp)));
  }
  void deallocate(pointer __p, size_type)
  { ::operator delete(__p); }
};
```

- **__gnu_cxx::malloc_allocator**

  其實現與上例唯一不同的是，
  它使用 C 函數 std::malloc 和 std::free。

```
template<typename _Tp>
class malloc_allocator
{
   pointer allocate(size_type __n, const void* = 0)   {
        ……
      pointer __ret = ……(std::malloc(__n * sizeof(_Tp)));
   }
   void deallocate(pointer __p, size_type)  {
      std::free(……(__p)); }
};
```

— 侯 捷

GNU C++ 對於 Allocator 的描述

另一種作法就是使用**智能型** allocator，將分配所得的內存加以緩存 (*cache*)。這種額外機制可以數種形式呈現：

- 可以是個 bitmap index，用以索引至一個以 2 的指數倍成長的籃子 (exponentially increasing power-of-two-sized buckets)

- 也可以是個相較之下比較簡易的 fixed-size pooling cache.

這裡所說的 cache 被程序內的所有容器共享，而 operators new 和 operator delete 不經常被調用，這可帶來速度上的優勢。使用這類技巧的 allocators 包括：

?!

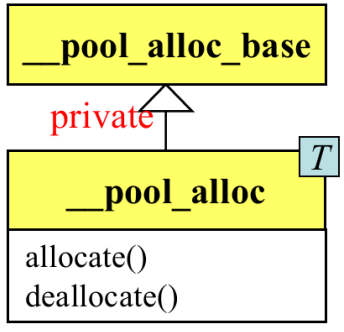
- **__gnu_cxx::bitmap_allocator**
  一個高效能 allocator, 使用 bit-map 追蹤被使用和未被使用 (used and unused) 的內存塊。

- **__gnu_cxx::pool_allocator**

- **__gnu_cxx::__mt_alloc**

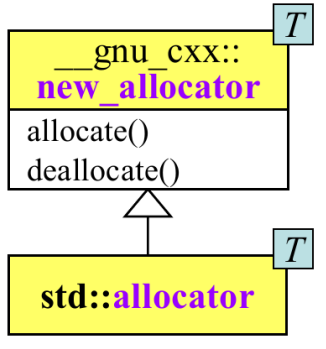
GNU C++ 對於 Allocator 的描述

Class allocator 只擁有 typedef, constructor, 和 rebind 等成員。它繼承自一個_high-speed extension allocators。也因此，所有分配和歸還 (allocation and deallocation) 都取決於該 base class，而這個 base class 也許是終端用戶無法碰觸和操控的 (user-configurable).



很難挑選出某個分配策略說它**能提供最大共同利益而不至於令某些行為過度劣勢**。事實上，就算要挑選何種典型動作以量測速度，都是一種困難。

GNU C++ 提供三項綜合測試 (three synthetic benchmarks) 用以完成 C++ allocators 之間的速度評比：



- Insertion. 經過多次 iterations 後各種 STL 容器將擁有某些極大量。分別測試循序式 (sequence) 和關聯式 (associative) 容器。

- 多線程環境中的 insertion and erasure. 這個測試展示 allocator 歸還內存的能力, 以及量測線程之間對內存的競爭。

- A threaded producer/consumer model. 分別測試循序式 (sequence) 和關聯式 (associative) 容器。

GNU C++ 對於 Allocator 的描述

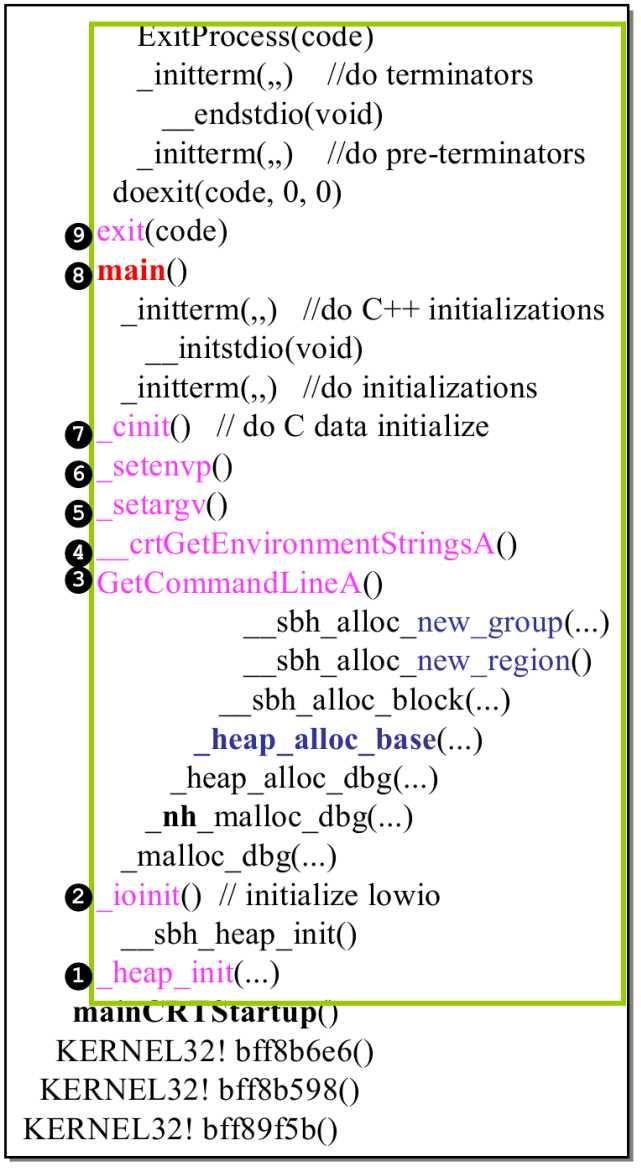
另兩個**智能型** allocator：

- **__gnu_cxx::debug_allocator**

這是一個外覆器 (wrapper)，可包覆於任何 allocator 之上。它把客戶的申請量添加一些，然後由 allocator 回應，並以那一小塊額外內存放置_size 信息。一旦 deallocate() 收到一個 pointer，就會檢查 size 並以 assert() 保證吻合。

- **__gnu_cxx::array_allocator**

允許分配一已知且固定大小 (known and fixed size) 的內存塊，內存來自 std::array objects。用上這個 allocator，大小固定的容器 (包括 std::string) 就無需再調用 ::operator new 和 ::operator delete。這就允許我們使用 STL abstractions 而無需在運行期添亂、增加開銷。甚至在 program startup 情況下也可使用。



VS2013 標準分配器 與 new_allocator

《…\xmemory0》

```
template<class _Ty>
class allocator
  : public _Allocator_base<_Ty>
{  // generic allocator for objects of class _Ty
public:
   typedef value_type *pointer;
   typedef size_t size_type;

   void deallocate(pointer _Ptr, size_type)
   {          // deallocate object at _Ptr, ignore size
      ::operator delete(_Ptr);
   }
   pointer allocate(size_type _Count)
   {          // allocate array of _Count elements
      return (_Allocate(_Count, (pointer)0));
   }
   pointer allocate(size_type _Count, const void *)
   {// allocate array of _Count elements, ignore hint
      return (allocate(_Count));
   }
…
};
```

《…\memory》

```
…
#include <xmemory>
```

《…\xmemory》

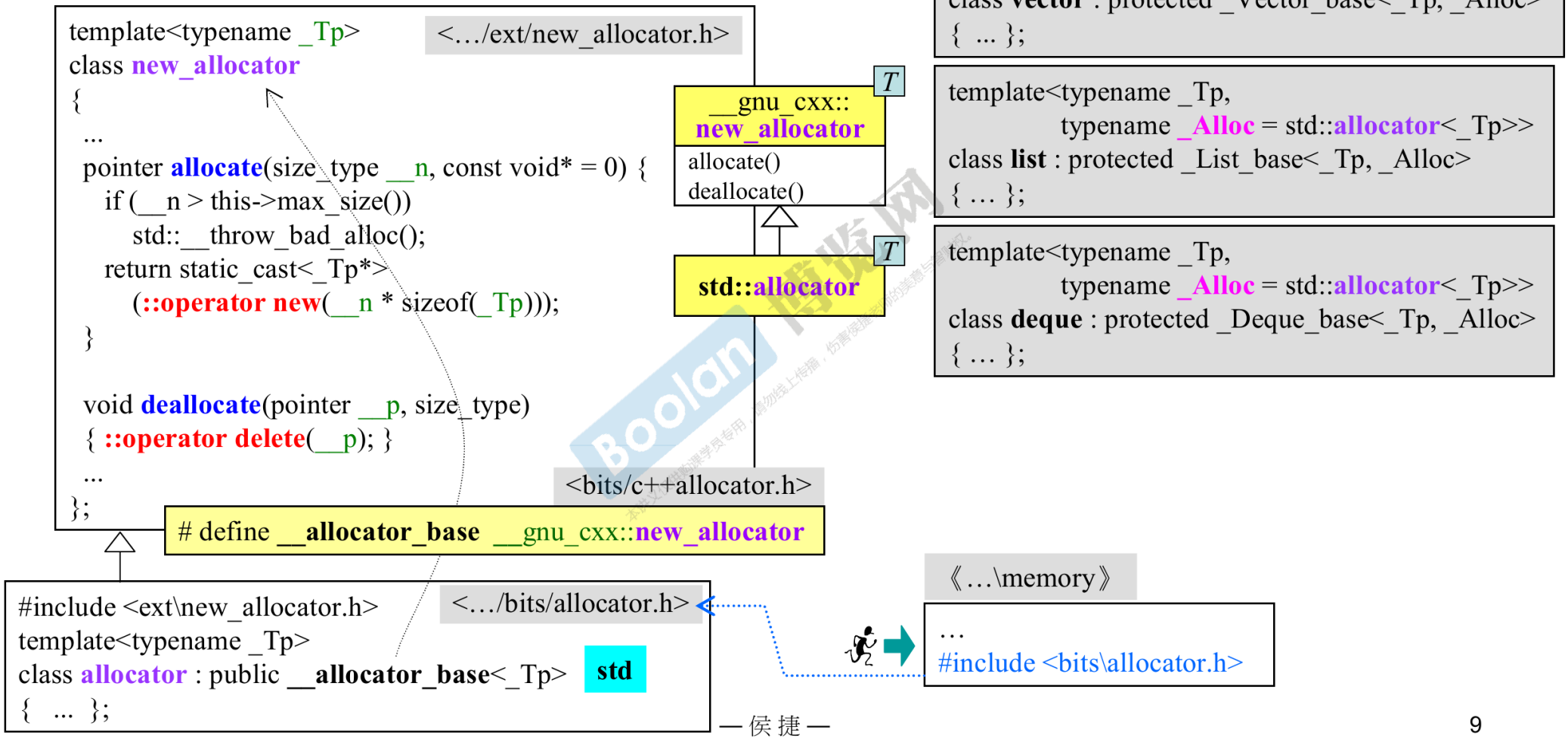
```
…
#include
<xmemory0>
```

<…/list>

```
template<class _Ty,
    class _Alloc = allocator<_Ty>>
class list
   : public _List_buy<_Ty, _Alloc>
{       // bidirectional linked list
   …
};
```

```
// TEMPLATE FUNCTION _Allocate
template<class _Ty> inline
_Ty *_Allocate(size_t _Count, _Ty *)
{    // allocate storage for _Count elements of type _Ty
   void *_Ptr = 0;
   if (_Count == 0);
   else if ((( )(-1) / sizeof (_Ty) < _Count)
            || (_Ptr = ::operator new(_Count * sizeof (_Ty))) == 0)
      _Xbad_alloc();    // report no memory
   return ((_Ty *)_Ptr);
}
```

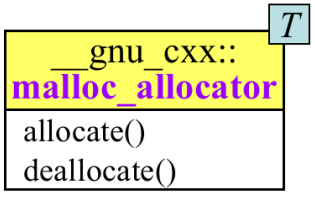
G4.9 標準分配器 與 new_allocator



G4.9 malloc_allocator

<…/ext/malloc_allocator.h>

```
template<typename _Tp>
  class malloc_allocator
  {
    // NB: __n is permitted to be 0.  The C++ standard says nothing
    // about what the return value is when __n == 0.
    pointer
    allocate(size_type __n, const void* = 0)
    {
          if (__n > this->max_size())
            std::__throw_bad_alloc();

          pointer __ret = static_cast<_Tp*>(std::malloc(__n * sizeof(_Tp)));
          if (!__ret)
            std::__throw_bad_alloc();
          return __ret;
    }

    // __p is not permitted to be a null pointer.
    void
    deallocate(pointer __p, size_type)
    { std::free(static_cast<void*>(__p)); }

    size_type
    max_size() const _GLIBCXX_USE_NOEXCEPT
    { return size_t(-1) / sizeof(_Tp); }
……
};
```

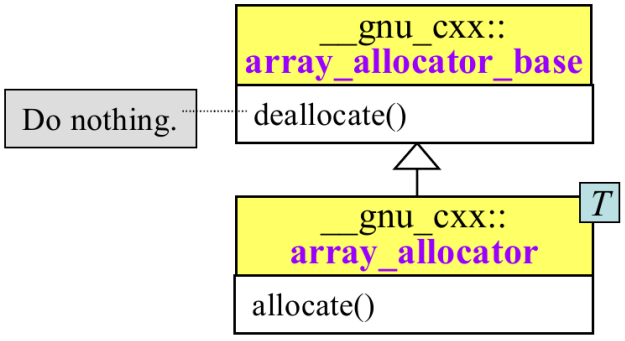


G4.9 array_allocator

<…/ext/array_allocator.h>

```
template<typename _Tp, typename _Array = std::tr1::array<_Tp, 1>>
  class array_allocator : public array_allocator_base<_Tp>
  {
  public:
    typedef size_t      size_type;
    typedef _Tp         value_type;
    typedef _Array      array_type;
    ……
  private:
    array_type* _M_array;
    size_type   _M_used;

  public:
    array_allocator(array_type* __array = NULL) throw()
    : _M_array(__array), _M_used(size_type()) { }

    ……
```

```
pointer
allocate(size_type __n, const void* = 0)
{
    if (_M_array == 0 || _M_used + __n > _M_array->size())
        std::__throw_bad_alloc();
    pointer __ret = _M_array->begin() + _M_used;
    _M_used += __n;
    return __ret;
}
};
```



G4.9 array_allocator

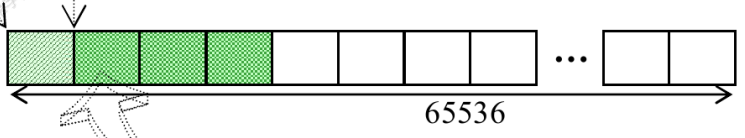
```
using namespace std;
using namespace std::tr1;
using namespace __gnu_cxx;

typedef ARRAY  std::array<int, 65536>;

ARRAY* pa = new ARRAY;

array_allocator<int, ARRAY> myalloc(pa);

int* p1 = myalloc.allocate(1);

int* p2 = myalloc.allocate(3);
myalloc.deallocate(p1);       //no-op
myalloc.deallocate(p2);       //no-op

delete pa;
```

使用 std::tr1::array 或 std::array 均可.

這種 array object
其內部將有 int _M_instance[65536];

```
pointer allocate(size_type __n, const void* = 0)
{
   if (_M_array == 0 || _M_used + __n > _M_array->size())
      std::__throw_bad_alloc();
   pointer __ret = _M_array->begin() + _M_used;
   _M_used += __n;
   return __ret;
}
```

```
template<typename _Tp, std::size_t _Nm>
   struct array
   {……
    value_type _M_instance[_Nm ? _Nm : 1];

    iterator  begin()
    { return iterator(&_M_instance[0]); }
    size_type size() const { return _Nm; }
    ……
   };
```



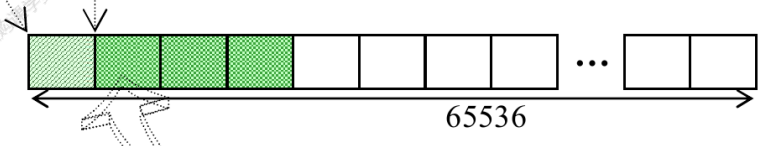
array_allocator 並不會
回收已給出的內存空間

G4.9 array_allocator

```
using namespace std;
using namespace std::tr1;
using namespace __gnu_cxx;

int my[65536];

array_allocator<int, array<int,65536>>
myalloc(&my);

int* p1 = myalloc.allocate(1);

int* p2 = myalloc.allocate(3);
myalloc.deallocate(p1);      //no-op
myalloc.deallocate(p2);      //no-op
```

使用 std::tr1::array 或 std::array 均可.

這種 array object
其內部將有 int _M_instance[65536];



array_allocator 並不會
回收已給出的內存空間

G4.9 debug_allocator

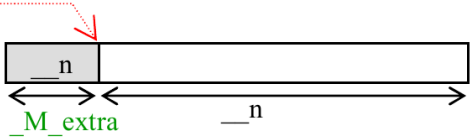
<…/ext/debug_allocator.h>

```
template<typename _Alloc>
  class debug_allocator
  {
  ……

  private:
    size_type _M_extra;  //通常是個 size_t;
    _Alloc _M_allocator;

    size_type _S_extra()  {
      const size_t __obj_size = sizeof(value_type);
      return (sizeof(size_type) + __obj_size - 1) / __obj_size;
    }

  public:
    debug_allocator(const _Alloc& __a)
      : _M_allocator(__a), _M_extra(_S_extra()) { }
```

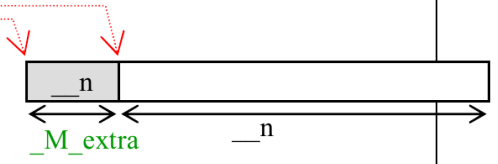
計算 extra (bytes) 相當於幾個元素

Alloc

__gnu_cxx::
debug_allocator
allocate()
deallocate()




```
pointer  allocate(size_type __n)  {
  pointer __res = _M_allocator.allocate(__n + _M_extra);
  size_type* __ps = reinterpret_cast<size_type*>(__res);
  *__ps = __n;
  return __res + _M_extra;
}

void  deallocate(pointer __p, size_type __n)  {
  using std::__throw_runtime_error;
  if (__p)  {
      pointer __real_p = __p - _M_extra;
      if (*reinterpret_cast<size_type*>(__real_p) != __n)
          __throw_runtime_error
            ("debug_allocator::deallocate wrong size");
      _M_allocator.deallocate(__real_p, __n + _M_extra);
  }
  else
      __throw_runtime_error
        ("debug_allocator::deallocate null pointer");
}
……
```



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

SGI style

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

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

而今安在哉 → __pool_alloc

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

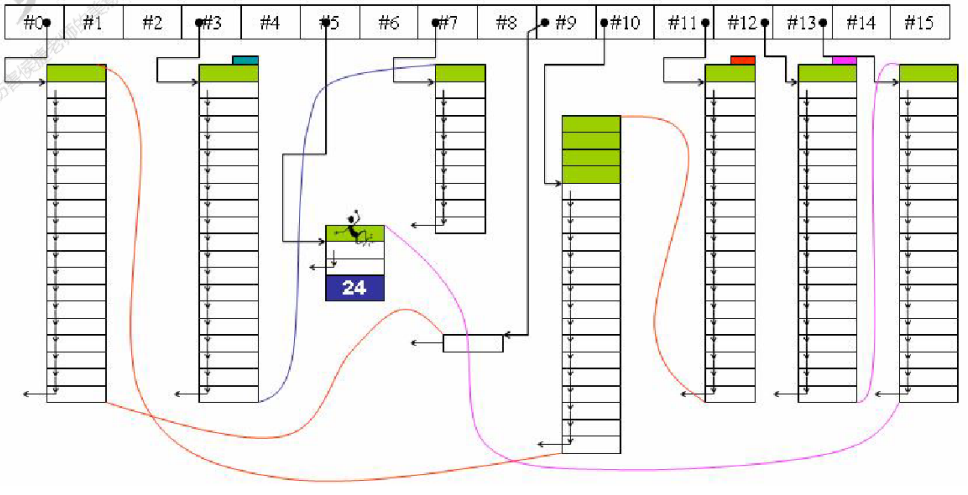
<…/ext/pool_allocator.h> **G4.9**

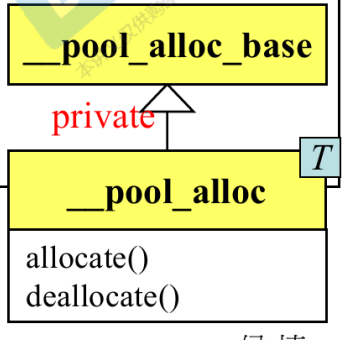
```
class __pool_alloc_base
{
protected:
  enum { _S_align = 8 };
  enum { _S_max_bytes = 128 };
  enum { _S_free_list_size = (size_t)_S_max_bytes / (size_t)_S_align };

  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;
  ...
```

STD style

用例：

```
vector<string,
       __gnu_cxx::__pool_alloc<string>>
vec;
```






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

G4.9 __pool_alloc 用例

```
//欲使用 std::allocator 以外的 allocator, 必須自行 #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);
```

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

相距 sizeof(double), 表示不帶 cookie

```
p1= 0xae25e8   p2= 0xadeb50   p3= 0xadd090
```

相距 > sizeof(double), 並不表示就帶著 cookie.

```
cout << sizeof(std::allocator<int>) << endl;   //1
vector<int, std::allocator<int>> vec;
cookie_test(std::allocator<double>(), 1);
```

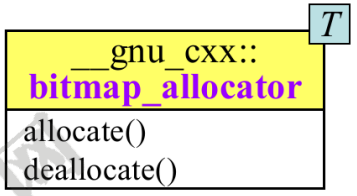
```
p1= 0x3e4098   p2= 0x3e4088   p3= 0x3e4078
```

相距(總是) == sizeof(double)=8, 可斷定帶著 cookie.

G4.9 bitmap_allocator

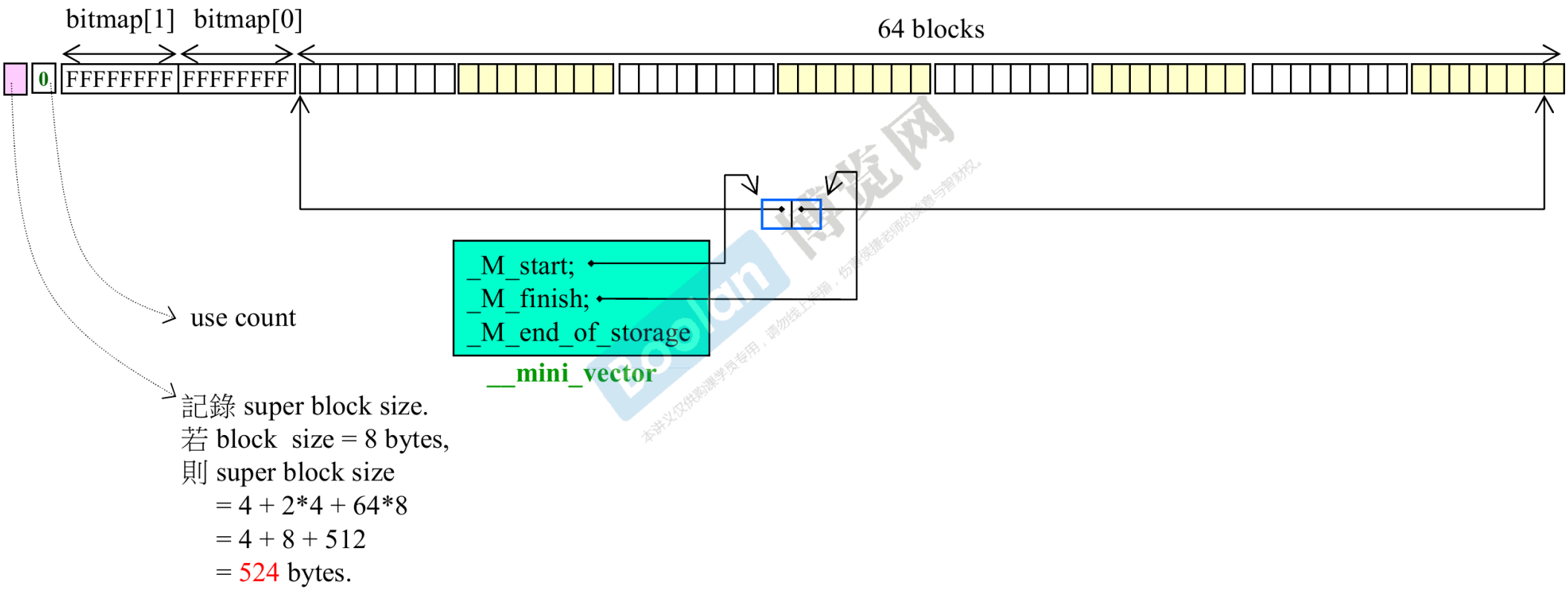
<…/ext/bitmap_allocator.h>

```
template<typename _Tp>
class bitmap_allocator : private free_list  {
…
   public:
     pointer allocate(size_type __n)   {
              if (__n > this->max_size())
                 std::__throw_bad_alloc();
              if (__builtin_expect(__n == 1, true))
                  return this->_M_allocate_single_object();
              else {
                  const size_type __b = __n * sizeof(value_type);
                  return reinterpret_cast<pointer>(::operator new(__b));
              }
     }
     void deallocate(pointer __p, size_type __n) throw()  {
              if (__builtin_expect(__p != 0, true)) {
                 if (__builtin_expect(__n == 1, true))
                    this->_M_deallocate_single_object(__p);
                 else
                    ::operator delete(__p);
              }
     }
```



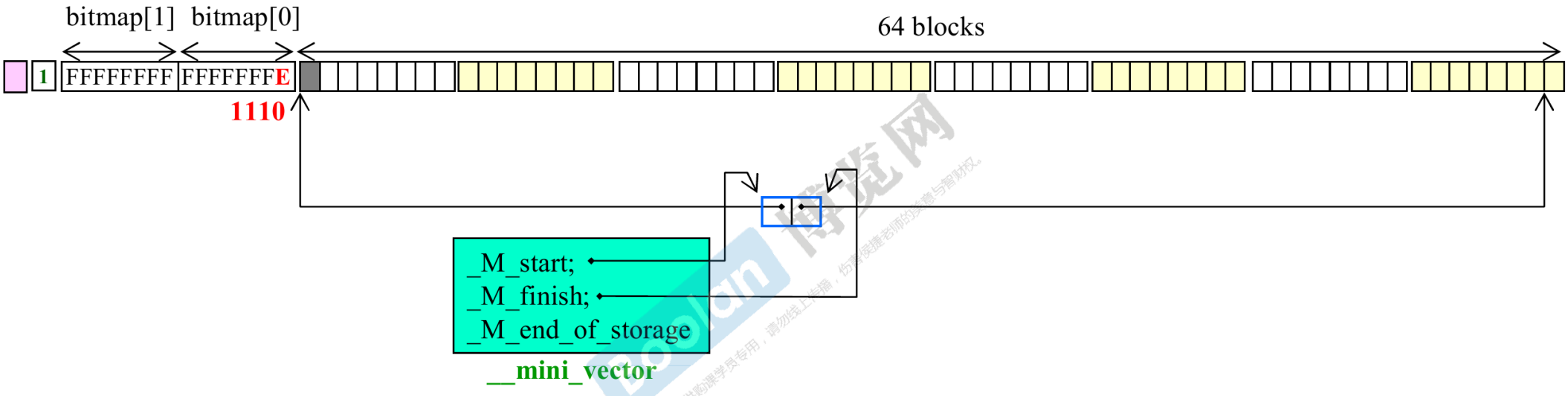
G4.9 bitmap_allocator,

關於 blocks, super-blocks, bitmap, mini-vector



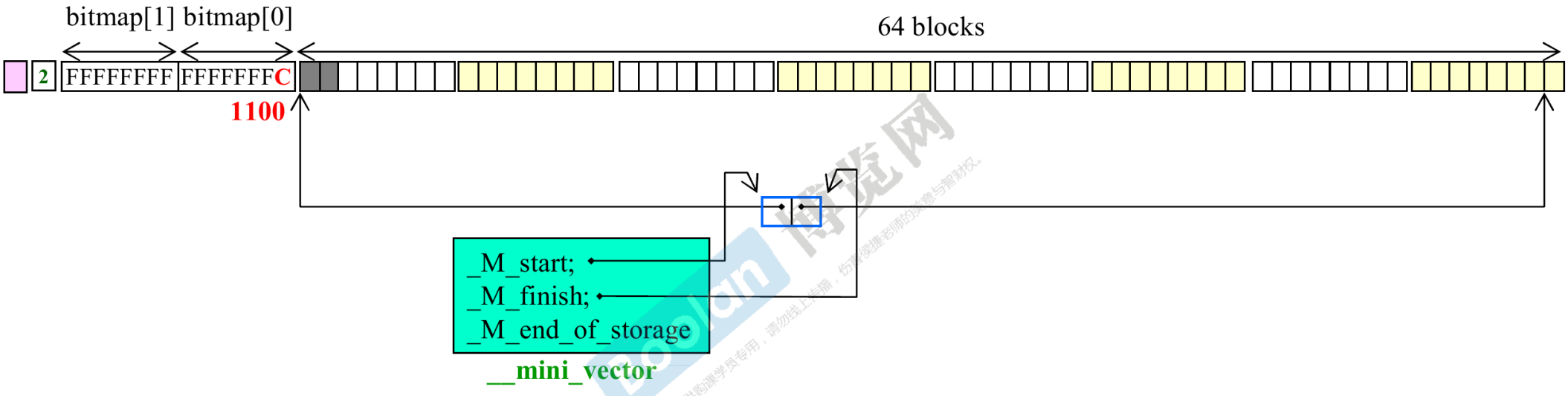
G4.9 bitmap_allocator,

關於 blocks, super-blocks, bitmap, mini-vector



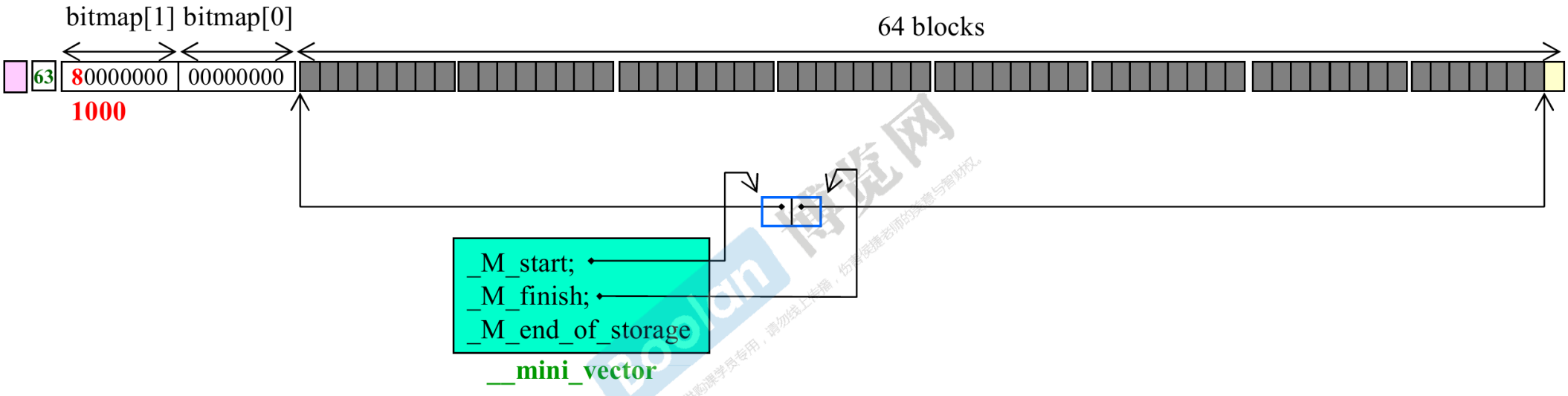
G4.9 bitmap_allocator,

關於 blocks, super-blocks, bitmap, mini-vector



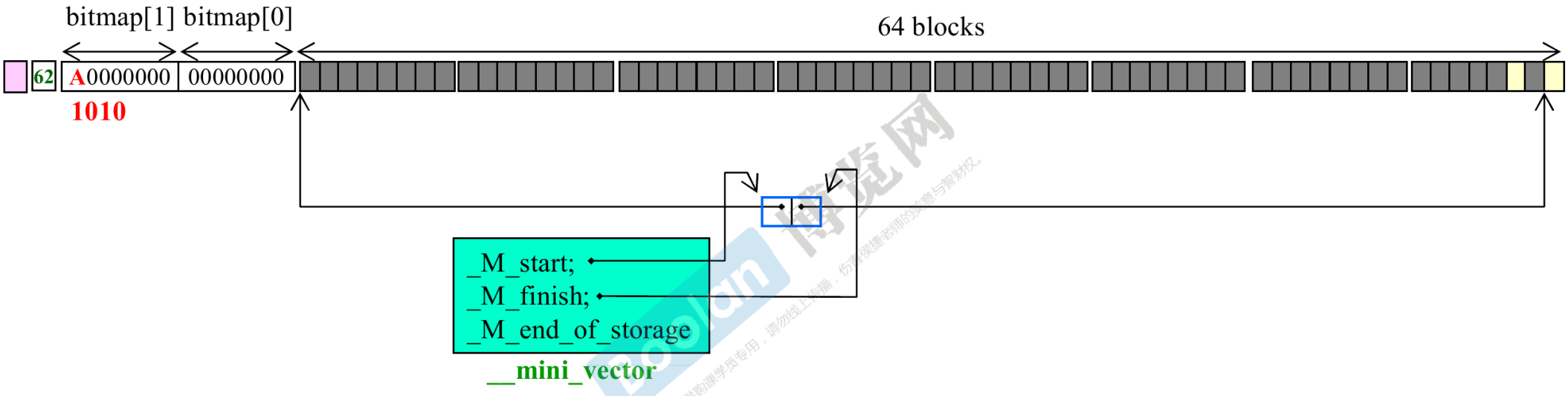
G4.9 bitmap_allocator,

關於 blocks, super-blocks, bitmap, mini-vector



G4.9 bitmap_allocator,

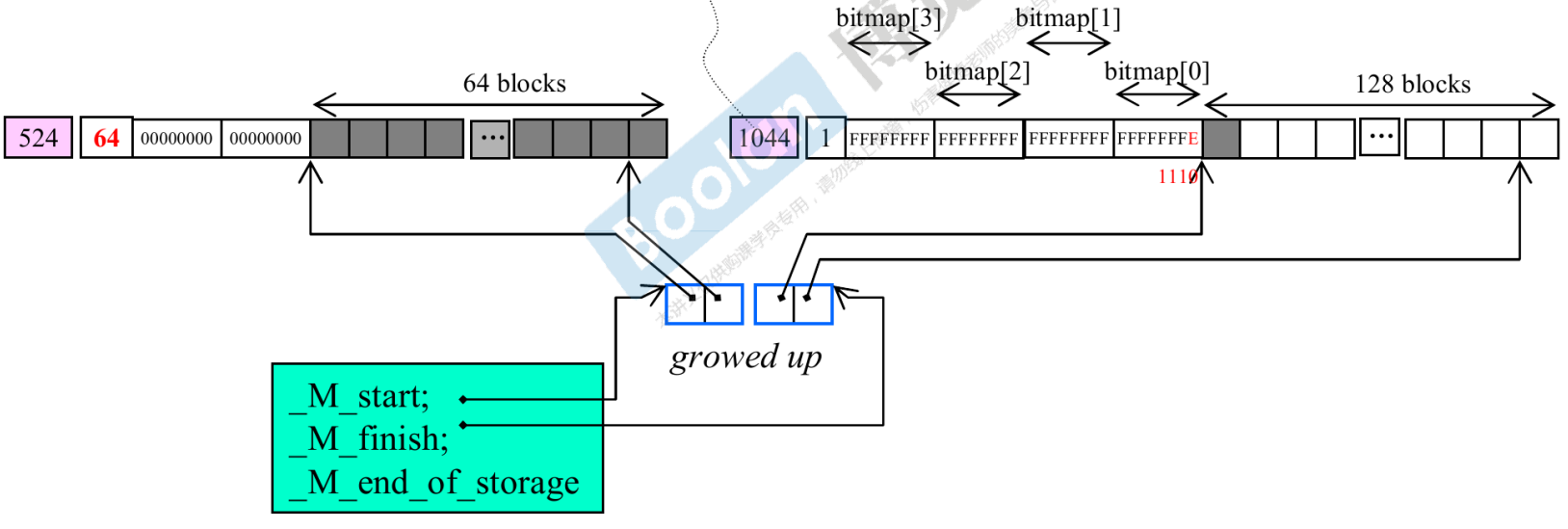
關於 blocks, super-blocks, bitmap, mini-vector



G4.9 bitmap_allocator,

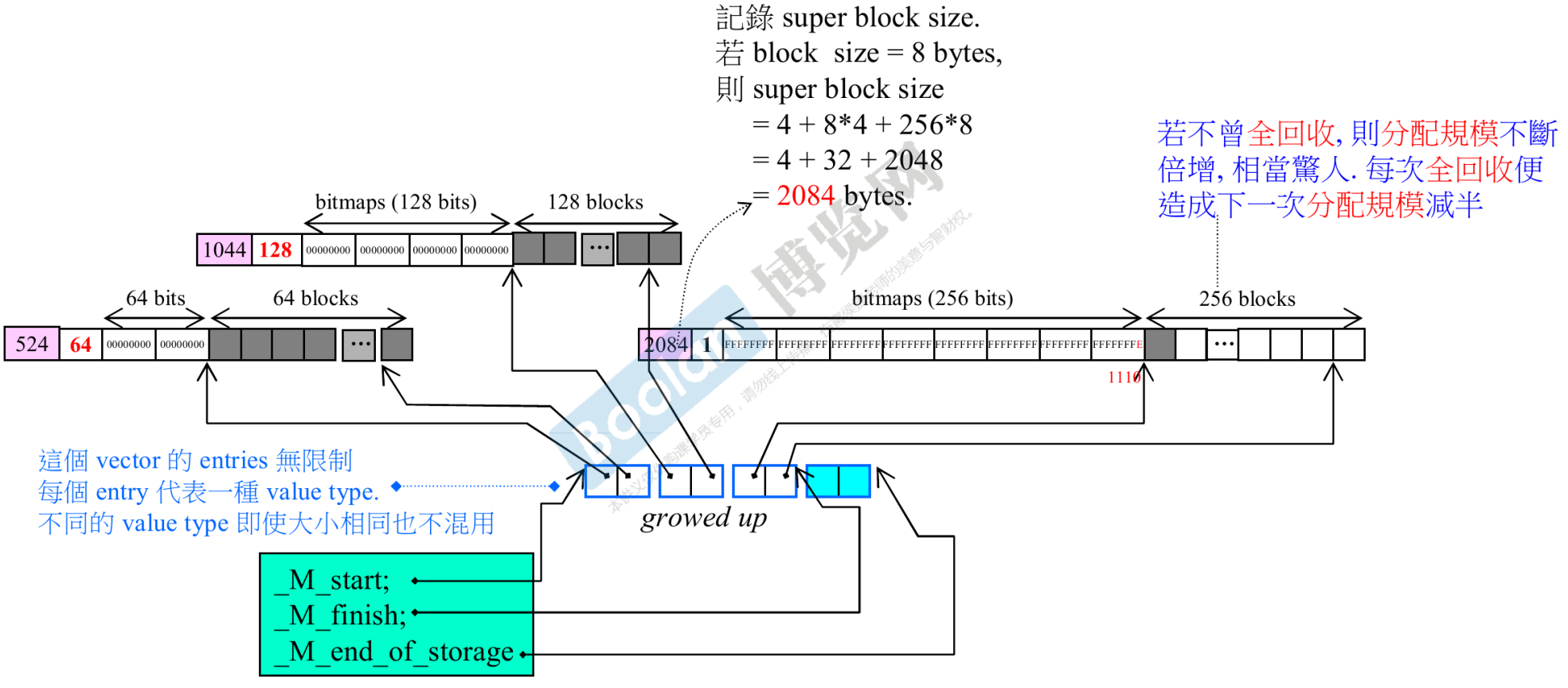
1st super-block 用罄, 啟動 2nd super-block

記錄 super block size.
若 block size = 8 bytes,
則 super block size
= 4 + 4*4 + 128*8
= 4 + 16 + 1024
= 1044 bytes.



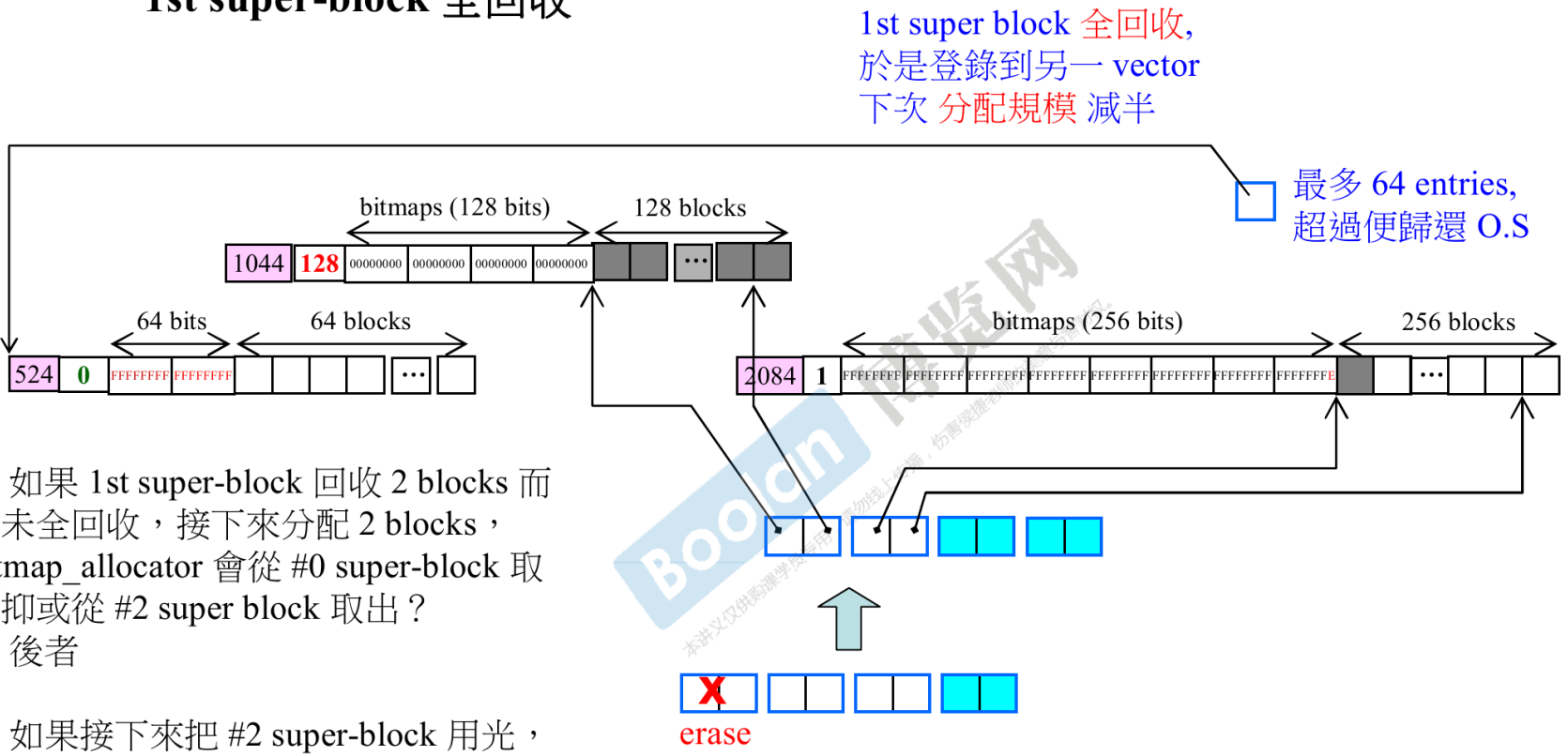
G4.9 bitmap_allocator,

2nd super-block 用罄, 啟動 3rd super-block



G4.9 bitmap_allocator,

1st super-block 全回收



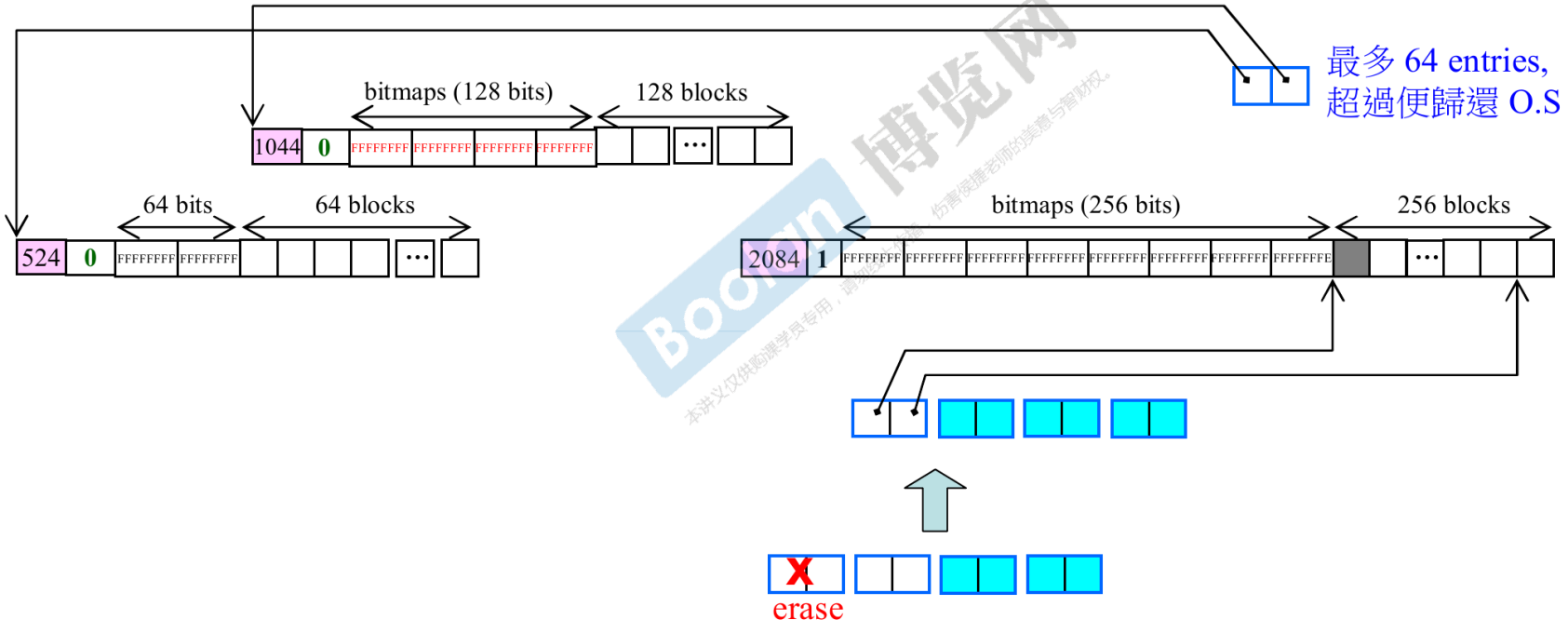
Q: 如果 1st super-block 回收 2 blocks 而尚未全回收，接下來分配 2 blocks，bitmap_allocator 會從 #0 super-block 取出抑或從 #2 super block 取出？
A: 後者

Q; 如果接下來把 #2 super-block 用光，然後分配 2 blocks，bitmap_allocator 會從 #0 super-block 取出抑或新建一個 #3 super block 並從中取出？
A: 前者

G4.9 bitmap_allocator,

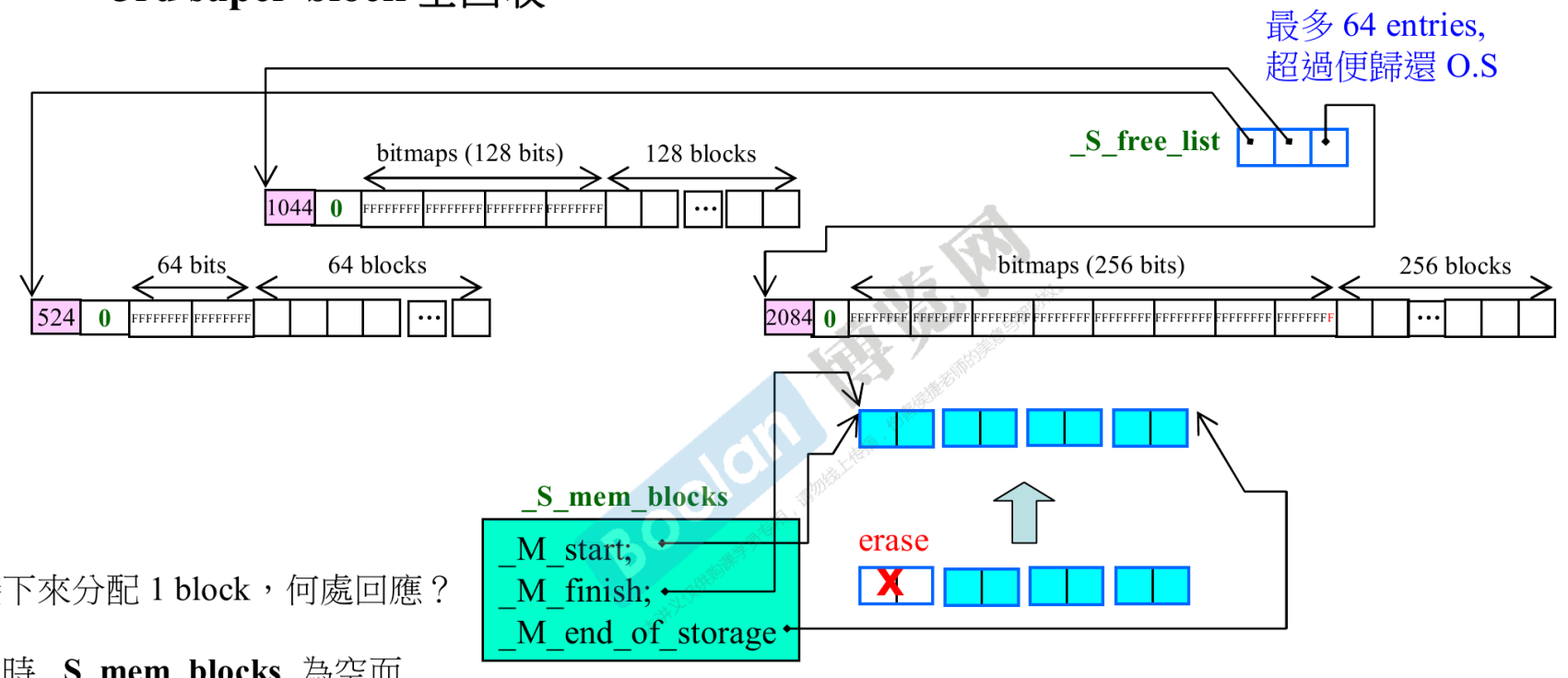
2nd super-block 全回收

這個 vector 的元素排列將以 super block size 為依據. 因此當觸及 threshold (64)，新進者若大於最末者便直接 delete 新進者, 否則 delete 最末者然後再 insert 新進者. 若未觸及 64 則無條件 insert 至適當位置.



G4.9 bitmap_allocator,

3rd super-block 全回收



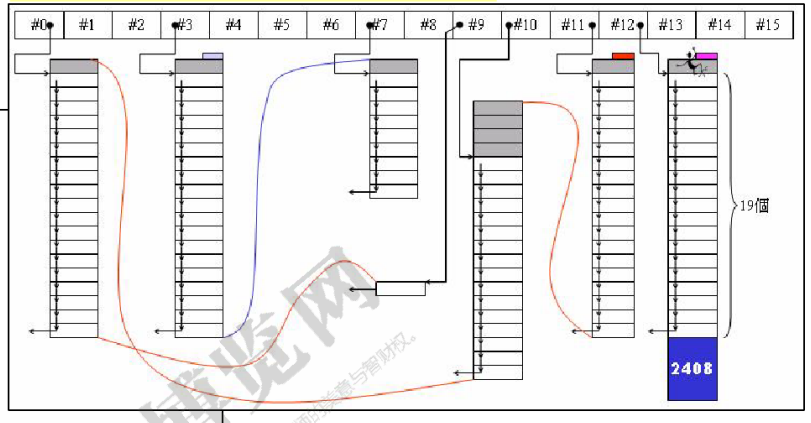
Q: 接下來分配 1 block，何處回應？

A: 此時 **_S_mem_blocks** 為空而 **_S_free_list** 有三個 super-blocks，於是取一個放進 **_S_mem_blocks**，然後遵循先前法則完成分配。

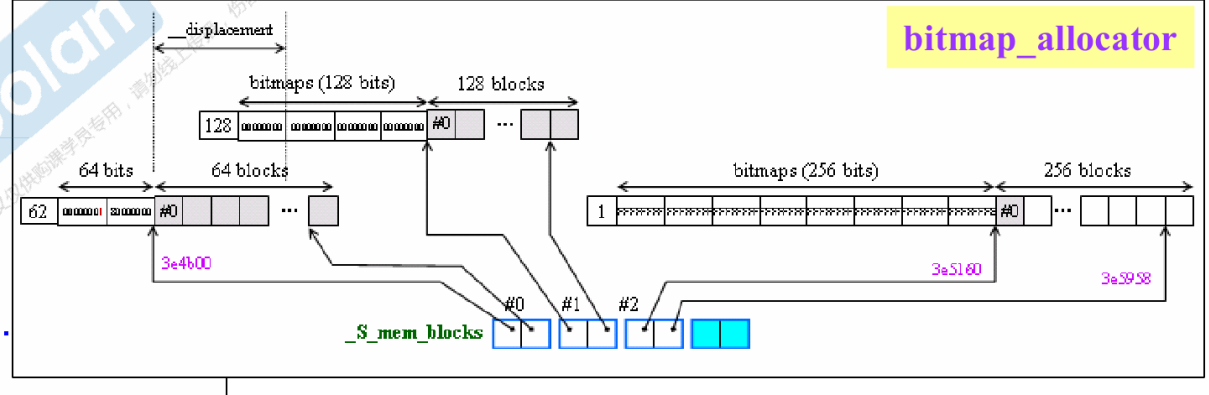
使用 G4.9 分配器

```
916  #include <list>
917  #include <stdexcept>
918  #include <string>
919  #include <cstdlib>      //abort()
920  #include <cstdio>       //snprintf()
921  #include <algorithm>    //find()
922  #include <iostream>
923  #include <ctime>
924
925  #include <cstddef>
926  #include <memory>       //內含 std::allocator
927      //欲使用 std::allocator 以外的 allocator, 得自行 #include <ext\...>
928  #include <ext\array_allocator.h>
929  #include <ext\mt_allocator.h>
930  #include <ext\debug_allocator.h>
931  #include <ext\pool_allocator.h>
932  #include <ext\bitmap_allocator.h>
933  #include <ext\malloc_allocator.h>
934  #include <ext\new_allocator.h>
935  namespace jj20
936  {
937  void test_list_with_special_allocator()
938  {
939      cout << "\ntest_list_with_special_allocator().....
940
941      list<string, allocator<string>> c1;
942      list<string, __gnu_cxx::malloc_allocator<string>> c2;
943      list<string, __gnu_cxx::new_allocator<string>> c3;
944      list<string, __gnu_cxx::__pool_alloc<string>> c4;
945      list<string, __gnu_cxx::__mt_alloc<string>> c5;
946      list<string, __gnu_cxx::bitmap_allocator<string>> c6;
```

__pool_alloc (pool allocator)



bitmap_allocator



使用 G4.9 分配器

```
948  int choice;
949  long value;
950
951      cout << "select: ";
952      cin >> choice;
953      if ( choice != 0 ) {
954          cout << "how many elements: ";
955          cin >> value;
956      }
957
958  char buf[10];
959  clock_t timeStart = clock();
960      for(long i=0; i< value; ++i)
961      {
962          try {
963              snprintf(buf, 10, "%d", i);
964              switch (choice)
965              {
966                  case 1 :    c1.push_back(string(buf));
967                              break;
968                  case 2 :    c2.push_back(string(buf));
969                              break;
970                  case 3 :    c3.push_back(string(buf));
971                              break;
972                  case 4 :    c4.push_back(string(buf));
973                              break;
974                  case 5 :    c5.push_back(string(buf));
975                              break;
976                  case 6 :    c6.push_back(string(buf));
977                              break;
978                  default:
979                      break;
980              }
981          }
982          catch(exception& p) {
983              cout << "i=" << i << " " << p.what() << en
984              abort();
985          }
986      }
987      cout << "a lot of push_back(), milli-seconds : "
988           << (clock()-timeStart) << endl;
```

```
989
990
991      //test all allocators' allocate() & deallocate();
992      int* p;
993      allocator<int> alloc1;
994      p = alloc1.allocate(1);
995      alloc1.deallocate(p,1);
996
997      __gnu_cxx::malloc_allocator<int> alloc2;
998      p = alloc2.allocate(1);
999      alloc2.deallocate(p,1);
1000
1001     __gnu_cxx::new_allocator<int> alloc3;
1002     p = alloc3.allocate(1);
1003     alloc3.deallocate(p,1);
1004
1005     __gnu_cxx::__pool_alloc<int> alloc4;
1006     p = alloc4.allocate(2);
1007     alloc4.deallocate(p,2);
1008
1009     __gnu_cxx::__mt_alloc<int> alloc5;
1010     p = alloc5.allocate(1);
1011     alloc5.deallocate(p,1);
1012
1013     __gnu_cxx::bitmap_allocator<int> alloc6;
1014     p = alloc6.allocate(3);
1015     alloc6.deallocate(p,3);
1016 }
1017 }
```

The End