「Boost技術與應用」系列文章（2）

Boost.Pool

文／侯捷

許多大型C++程式庫都有自己的記憶體管理機制，包括MFC, STL, Loki都如此，Boost自然也不例外。本文概要說明前三者的作法，然後集中火力探討Boost的作法與設計細節。

若干知名的大型C++程式庫，如MFC, STL, Loki，都有自己的記憶體管理機制，為的是能夠以最快速最節省空間的方式供應objects所需的記憶體。也就是說這些程式庫都有一套迥異於C++ new運算子的記憶體配置與歸還機制，這意味new運算子必然存在某種改善空間。本文首先說明這一值得改善的空間（Why Pooled Allocation），然後簡介MFC, STL, Loki如何改善（How Pooled Application），最後詳細介紹Boost的作法與關鍵細節。

本文相關資訊

- 讀者基礎：擁有C++ templates編程經驗和C++標準程式庫使用經驗。
- 本文測試環境：VC6, VS2003。
- 本系列文章將匯整至編寫中的《Boost運用與源碼剖析》一書。
- 本文關於STL, Loki, MFC, Boost之Pooled Allocation技術分析將匯整至編寫中的《C++記憶體管理》一書。
- 關鍵術語：Boost, Pool, Pooled Allocation, Embedded Pointers, chunks, blocks。

為什麼需要Pooled Allocation

這個題目相當於探討傳統的C malloc()或C++ new有什麼缺點。欲配置一塊記憶體，C語言用的是malloc()函式家族，例如：

```
void* p = malloc(1024);
```

欲建立一個物件，C++ 語言用的是new算式，例如：

```
Foo *p = new Foo;
```

new算式實際上分為三個步驟，其一配置記憶體，其二轉型，其三呼叫建構式：

```
//以下是 C++ 編譯器對new算式的分解動作
Foo *p;
try {
  void* mem = Foo::operator new( sizeof(Foo)); //配置記憶體
  p = static_cast<Foo*>(mem);              //轉型
( c a s t )
p->Foo::Foo;                                  //建構
}
catch( std::bad_alloc ) {
  //若配置失敗，不執行建構式
}
```

如果編譯器沒能找到上述的Foo::operator new()，就會改而呼叫全域函式::operator new()，那是C++程式庫提供的記憶體配置基本工具，功能相當於malloc()，這一點可從Borland C++源碼獲得清晰的印證：

```
inline void* _RTLENTRY
operator new (size_t size, const std::nothrow_t &)
{
  size = size ? size : 1;
  return malloc(size);   // ::operator new 呼叫malloc()
}
```

由此可見，無論C或C++，追根究底談其記憶體配置時一定得碰觸malloc()。此函式的

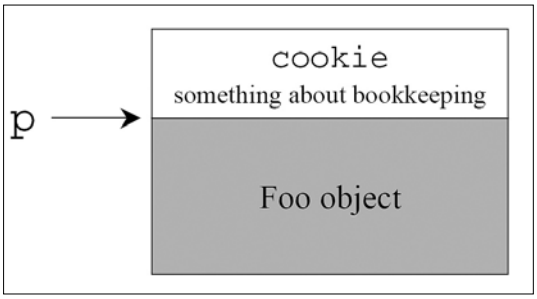

圖1 cookie示意圖。

行為可從Doug Lea所寫的知名版本窺知，此版本被各大編譯器採用，源碼見Doug Lea 的個人主頁http://gee.cs.oswego.edu/dl/。追蹤這份源碼不是件容易的差事，我手上這份源碼的.h長達679行，.c長達5621行，而且市面上找不到完整論述介紹其中的資料結構或演算法。但縱使未能完完全全了解malloc()的行為，這裡一定要讓你知道，malloc()配置出來的記憶體區塊除了滿足客戶索求的大小，另附帶了一個不為人知的區域，一般暱稱為cookie（小甜餅），見圖1。cookie的大小和其儲存的資訊及格式，視不同的實作而有所不同。我曾做過一個試驗，在剛開機的情況下（此時可用記憶體的分佈還很乾淨，未被亂切亂拼）配置若干區塊，獲得若干指標，然後將相鄰各指標相減，這便假設是cookie的大小（實驗結果不是4 就是8 ，取決於編譯器實作）。再將每一指標的前4或前8個bytes印出來，便是cookie內容。但是不知其格式該如何列印呢？瞎貓碰死耗子，以整數格式列印看看。結果發現BCB的cookie總是（並且只是）記錄著區塊大小。其他編譯器就沒有這麼「清純」。

區塊之前方帶著一塊「小甜餅」的必要性很容易理解：當我們將記憶體歸還系統，我們呼叫的是free()（或?::operator delete()）並且只給予一個指標指向先前以malloc()獲得的記憶體。這種情況下如果沒有「小甜餅」幫助，free()根本不知道客戶意欲釋放的區塊有多大。cookie甚至會記錄區塊大小以外的信息，例如該區塊和其系統中的鄰塊之間的關係…，以利系統回收後做最佳安排。這些信息的設計取決於編譯器實作。

為應付大小不一的各種需求，cookie乃是必要之「惡」。但也就是這個cookie為C++程式中常見的一種情況帶來很大浪費：如果客戶需要大量小區塊（比方說100,000個物件，每個物件16 bytes），那麼每個區塊都帶cookie就是一種浪費，此乃因為大量小區塊中的每一區塊尺寸都相同，應該可以有更好的設計，不需要cookie那份「區塊尺寸」的記錄。圍繞這個概念的相關設計幾乎全都基於一種相同的思考，就是Pooled Allocation（池式配置）。

為了拿掉cookie，通常的作法是由管理器（程式庫）挖一大塊記憶體自行細切（不帶cookie，每一塊尺寸都剛好符合客戶需求）。這些被細切的小區塊往往以linked list管理，一整串小區塊也就被稱為free list。當客戶需要這種尺寸的小區塊時，就由程式庫從free list 中給出一個，並維護linked list的正確鏈接。這種想法好似把一大塊記憶體當成一個池子（pool）來供應「個頭一致」的小東西，所以被稱為池式配置。「pool」的另一個意義是「共用物」，當動詞解則是「聯營」，Pooled Allocation的意思正是「聯營系統下的記憶體分配」。

C++ 程式中，客戶索求大量固定尺寸的小區塊，這種情況很常發生（相當於製造出大量物件，每個物件都小小的）。也非常可能客戶要的尺寸有三兩種，因此一個好的

Pooled Allocation設計，尤其是做為必須應付各種可能情況的角色而存在的程式庫，最好能同時供應（維護）多個free lists，每個list負責供應某一特定大小的區塊。Pooled Allocation不但大量節省空間，亦巨幅節省時間，因為它大量減少了向系統索求記憶體的次數。

如何設計Pooled Allocation

敏銳的你很快會想到，為了節省一個cookie（4或8個bytes）卻賠上維護linked list所需的指標（單向list需要一個指標，雙向list需要兩個指標），豈不是偷雞不著蝕把米？但事實上我們不需要為每個區塊各自準備指標，因為區塊在尚未給出之前，也就是當區塊還在配置系統（程式庫）的掌握下時，其空間可被程式庫使用，對客戶無任何影響，而這些空間（現實至少4 bytes）就足以做出至少一個指標了。這種從區塊本身挖空間建立指標的作法，在《Small Memory System, Patterns for system with limited memory》（中譯本「記憶體受限系統之設計範式」）中稱為「嵌入式指標」（Embedded Pointers），見圖2。

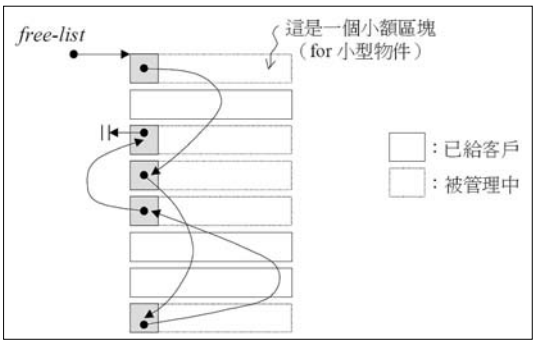

圖2 嵌入式指標。給出的區塊不再被free list管理，未給出的區塊前4 bytes被當做維護linked list所必須的指標。

這種設計可被相當精簡地實現。《C++Primer, 3/e》15.8節p.765和《Effective C++, 2/e》條款10都列有對池式配置的片段舉例，簡單地說就是為各自例中的class重載operator new和operator delete（用以接管客戶提出的申請），並在operator new中管理單一free list。兩例的唯一差別是前一例未使用嵌入式指標，後一例使用了嵌入式指標，剛好讓我們做出比較。

如果把上述功能獨立出來放在一個分離而專用的class內（假設名為Pool），由它提供介面函式（通常名為allocate()和deallocate()），讓客戶的每個classes重載operator new和operator delete並且其內不再自行管理free list也不直接呼叫malloc()和free()，而是改呼叫Pool::allocate()和Pool::deallocate()，這就把池式配置乾淨地切割了出來，又讓客戶的classes不知不覺用上它。這便是所有C++程式庫池式配置系統的雛型。

下面我要描述的每一個C++ 程式庫的記憶體配置系統，雛型都如以上描述。至於其各顯神通的差異性，可從各張示意圖中輕易感受。進行之前讓我們統一術語的用法，在池式配置系統中，每次向系統要的大區塊通常稱為chunks，chunk內切割出來尺寸相同的小區塊稱為blocks。STL, MFC, Loki皆沿用這種稱呼習慣，但Boost剛好相反。討論到Boost時我會再次提醒你。此外，將小區塊還給程式庫配置系統的動作，我會說歸還（deallocation），將大區塊還給系統的動作，我會說釋放或釋還（free）。

Pooled Allocation各顯神通

在廣被大家遵循的策略如嵌入式指標、細

切、operator new/delete重載之外，究竟Pooled Allocation還有什麼戲法可變呢？關鍵在於以下數點：

- 程式庫該提供多少個free lists（每一個代表一種特定區塊尺寸）才算合理？太少則用途不夠廣泛，而且這樣的設計必須針對不提供的尺寸給客戶一個說法。太多則過猶不及，會不會造成系統過於複雜？如何避免過於複雜？
- 每個free list是真正一根腸子到底的單一linked list，抑或是分段鏈接？如果使用單一長串linked list，將來要部分（甚或全部）釋還給系統就不可得。要知道，Pooled Allocation的策略是「先搶一大塊再慢慢消化」，而接受歸還時則是「納入自己體系以便下次再回應需求」。如果從不考慮將區塊釋還給系統，程式就有可能因為取用大量記憶體而影響多工環境下的其他行程（processes）對記憶體的使用。這不能說是記憶體洩漏（leak），因為還在程式掌控中，但它確實妨礙了其他行程。
- 欲將區塊釋放給系統，不只是上述所說將free list分段就萬事具備了。每一塊記憶體，包括現在討論的Pooled Allocation的一大塊一大塊記憶體，都必須保留cookie才能還給系統。如何能將每一大塊記憶體的cookie都記下來而不混亂？

讀者可以從下列討論的各家系統看出各家的作法，真可謂戲法人人會變，各有巧妙不同。限於篇幅和主題，惟有談到Boost時我才帶引讀者細看其體系結構和源碼，其他程式庫都只給出一個足以代表大局觀的示意圖，加上若干微言大義的文字。

SGI STL的std::alloc的作法

C++標準程式庫（或說STL）中的allocator是個記憶體配置器，這個配置器在不同版本的STL上有不同的設計和不同的底層行為。本文要說的是SGI版本，事實上也只有它做出了Pooled Allocation。此配置器雖然也可被直接使用，但多半隱身幕後負責供應STL容器之記憶體需求。可想而知在現實程式中，使用SGI STL或使用其他版本的STL，當客戶以容器放置大量元素時，記憶體用量和配置速度會有多麼大的落差。

SGI STL allocator的class名為std::alloc，它一開始就（並且只）以一個array維護16個指標，用來指向16個free lists，分別處理8,16,24,32…（以8的倍數增加）至128 bytes的尺寸需求。對於不為8倍數或高於128 bytes的區塊索求，它給的說法是，區塊大小將被自動調整為8的倍數，超過128則意味cookie與區塊尺寸之比例已經小到其浪費率可被忽略，因此改由系統內建的operator new/delete（相當於malloc/free）來服務。這個轉向服務是隱晦的，客戶並不知道（當然，現在我們都知道了），而且包裝精美，完全不影響客戶對區塊的索求操作。

圖3是SGI 的std::alloc完整示意圖。此圖非常珍貴（呵呵，我難免敝帚自珍），是歷經許多測試並追蹤源碼後的結晶。std::alloc維護的16個指向free lists的指標係以static形式存在，一開始都是null。每當客戶有所需求，std::alloc就找到對應的free list，如果其內有可用小區塊就直接拿來用。一開始當然沒有可用區塊，於是向系統要

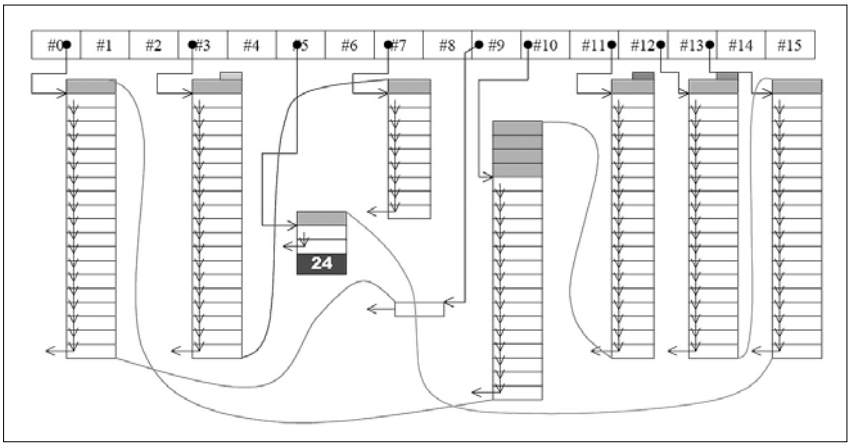

圖3 SGI STL std::alloc行為示意圖。

20*2個區塊(其實還有一個補充量，隨著配置次數增加會愈來愈大。這裡為了簡化說明只說20*2。)那麼大的空間，其中一半細切為20塊，另一半留著以備下回再需向系統要記憶體時拿出來用（我稱它為戰備池）。戰備池的存在是為了儘量減少向系統索求記憶體的次數，因為那很耗時。

基本上std::alloc就沿著這樣的主軸進行。它的巧妙出現在當系統記憶體不足時的挖東補西能力和錙銖必較的精細程度。舉個例子，當客戶要求72 bytes，而對應的#8 free list無可用區塊供應，按規則應取戰備池來充填#8 list。如果戰備池亦空，或不足（例如圖3藍色粗體字 '24' 就是戰備池的當時大小），才向系統索求（此時戰備池的餘額會先掛到對應的free list中）。如果系統也彈盡源絕， std::alloc會依序尋找 #9~#15 lists看看有無可用區塊。如果有，拿來切割放進#8 供使用，餘下的（可能是8,16,24,32…bytes）充填到對應的free list，吃乾抹淨絲毫不浪費。

這個系統有一個不太理想的地方：它很霸道，絕不釋還記憶體給系統。事實上它無力歸還，因為每次從系統配置來的一大塊空間，並沒有（也很難）將其cookie記錄下來。這意味這一大塊記憶體只有在程式結束後才能回歸母親的懷抱了。圖3所示的#11和#12 free lists的第一個區塊上方我畫了一小塊扁扁的東西，代表cookie，這個cookie所記錄的大小有一部分被切割給其他free list用（因為戰備池發揮功效之故），因此要保持cookie和其原始對應空間的關聯，以及判斷各free lists內的所有區塊是否已全部回收，在此設計架構下是很不容易的事。

std::alloc的用法如下：

```
void* p = std::alloc::allocate(512);      //static函式
std::alloc::deallocate(p,512);    //static函式
```

你一定會驚訝，怎麼歸還時還需告知區塊大小呢？這對客戶的負擔太大了。啊，std::alloc運用template設計了一些包裝，可有效為客戶去除這層負擔。

Loki::SmallObjAllocator的作法

Loki程式庫對templates的使用簡直出神入化，它的前衛使它成為曲高和寡的踽踽先行者。不過，撇開其揭櫫的「以classes表現設計範式」的高絕理念，以及前衛而驚世駭俗的template template parameters技法，Loki的「小型物件配置器」卻是相對平易進人、功能充足而又設計輕巧。

圖4是Loki小型物件配置器的完整示意圖。再一次，此圖非常珍貴（呵呵，我又敝帚自珍了），同樣是歷經許多測試並追蹤源

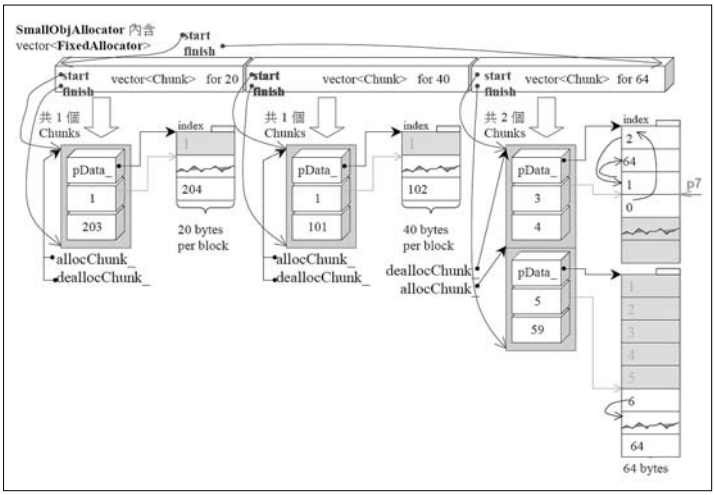

圖 4 Loki::SmallObjAllocator行為示意圖。

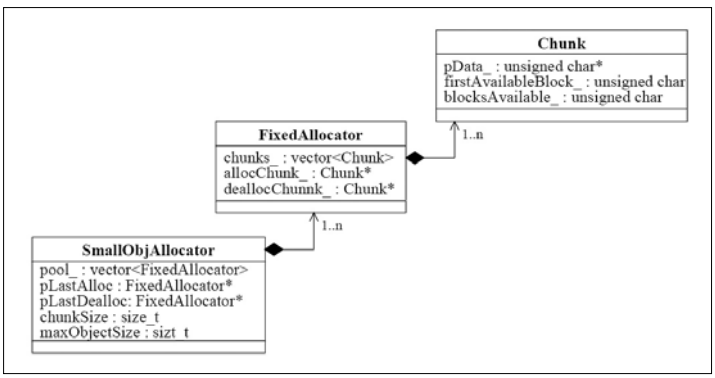

圖 5 Loki配置器的三個主要classes的關

碼後的結晶，可收一目瞭然之效。從圖中可以清楚看出，水平方向的SmallObjAllocator作用相當於前述std::alloc中維護16個free lists的array，但現在它不再是array而是個vector，從這裡可以窺知它意圖支援任意數量的free lists。只要客戶要求配置某物件，它便產生一個對應的free list專門服務。圖4顯示目前製造出三個free lists分別服務20,40,64三種尺寸。

圖4中垂直方向是身為free list的第二層管理器FixedAllocator，採「分段連續」設計，以vector管理1..n個大區塊（所謂"chunks"）。所以第二層其實不是個linked list，但我們仍可稱它為free list。大區塊箇中有特定數量的小區塊（blocks），最多255個，或最多令單一chunk達到4K。例如圖4中由左至右每一種chunk分別切割為204、102、64個blocks。每個block等同於一個指定的物件大小，以最底層管理器Chunk表現。Chunk的設計很有意思，它相當於小型free list（也就是分段連續中的「段」），但它係以array模擬linked list，並以最前一個bytes記錄下一可用block的索引，而不再採用「嵌入式指標」。這種演算法挺少見，但十分經濟有效。

這個設計的優點在於，第一，最上層和第二層都以vector管理，所以不論free lists或chunks的個數都可以彈性增加，甚至減少 — 實際有無減少端視有沒有能力將chunks釋還給系統。第二個優點是，chunks內的blocks的原個數、目前可用個數、以及每個chunk的起點位址都被記錄了下來，前二者可據以判斷某chunk內的每一個blocks是否都歸還了，第三項（起點指標）則表示其前頭必帶著cookie，那就是說它有能力將chunk釋還給系統。事實正是如此，在我所做的測試中，連續對同一尺寸要求多個blocks，導致出現4個chunks，而後一一歸還，檢測得知最後只剩2個chunks — 不全部釋放，留下一些餘地以備後用。Loki的這個配置器比較王道，不復std::alloc之霸道矣。

Loki「小型物件配置器」的相關classes

有四層：

1.Chunk：這是最底層，相當於一小段free list。實際乃以array模擬linked list。

2.FixedAllocator：這是中間層，相當於free lists，以vector串起一個個chunks。

3.SmallObjAllocator：這是最高層，以vector管理眾多free lists。

4.SmallObj：這個class架構於SmallObjAllocator之上，主要訴求是涉入Loki的ThreadingModel設計。它在記憶體配置上並無更多設計，所以本文也沒有討論它。

前三個classes的關係如圖5。

Loki小型物件配置器的用法是：

```
SmallObjAllocator myAlloc(2048,256); //可應付小於
256的各種大小配置
void* p1 = (void*)myAlloc.Allocate(20);
void* p4 = (void*)myAlloc.Allocate(64);
void* p2 = (void*)myAlloc.Allocate(40);
void* p3 = (void*)myAlloc.Allocate(300);
        //以上大於256，內部將轉由內建工具處理
myAlloc.Deallocate(p4,64);
myAlloc.Deallocate(p1,20);
myAlloc.Deallocate(p2,40);
myAlloc.Deallocate(p3,300);
```

客戶歸還blocks時必須告知大小。這對客戶是一種負擔。然而就像先前討論std::alloc時所說，運用template可設計出一些漂亮的包裝，為客戶去除這層負擔。

MFC's CFixedAlloc的作法

圖6是MFC CFixedAlloc的示意圖。我就不必再次強調它的珍貴性了吧呵呵。MFC的Pooled Allocation主要由兩個classes擔綱，一是CPlex一是CFixedAlloc。前者扮演「配置chunk並分段串接」的角色，後者內含前者並呼叫其函式，而後自行切割chunk成為blocks。

CPlex只開放兩個函式，Create()負責配置chunk，而chunk的大小等於block個數 * block大小 + sizeof(CPlex）。其意圖非常明顯：由chunk自身帶著一個頭部（也就是一個CPlex）負責鏈接的必要資料，而鏈接動作就在Create()完成，但不細切。另一個開放函式是FreeDataChain()，將鏈接在一起的所有chunks（其blocks尺寸都相同）以delete[] 釋還給系統。但它不像Loki會自行判斷釋還時機並自行採取行動，而是只在CFixedAlloc解構式中進行，也就是當此配置器離開作用域才進行，略遜Loki一籌。

直接面對客戶的是CFixedAlloc，它內含一個指標指向CPlex，並有一個開放函式Alloc()，其內呼叫CPlex::Create()以便獲得

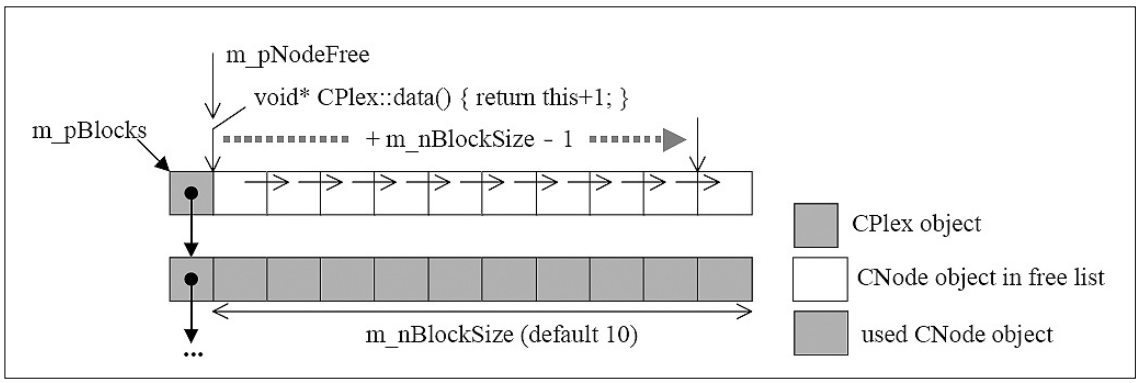

圖6 MFC CFixedAlloc行為示意圖。本圖第一個chunk的所有blocks都被取光（白色），所以再要一個chunk（灰色）。每個chunk帶有一個CPlex用來鏈接chunks。

chunk，獲得之後進行切割，採用嵌入式指標形成一個「blocks數量固定（預設為64）」的linked list，而後給出一個block，並以m_pNodeFree指向下個可用的block。細切所得的每個blocks都以CNode表現，那是定義於CFixedAlloc內部的一個struct，結構十分簡單：

```
//這裡是CFixedAlloc定義式內部
struct CNode
 {
   CNode* pNext;   // only valid when in free list
 };
```

如果某次客戶呼叫Alloc()時此配置系統無可用的blocks，Alloc()內部就再呼叫CPlex::Create()取得一個chunk並鏈接、細切、給出block，依此類推。

```
CFixedAlloc fooAlloc(sizeof(Foo)); //建立配置器時指定其服務尺寸
void* p1 = fooAlloc.Alloc();
void* p2 = fooAlloc.Alloc();
fooAlloc.Free(p1);
fooAlloc.Free(p2);
```

在這種設計架構下，MFC配置器的用法是：

建立CFixedAlloc物件並給予不同初值（代表block尺寸）的次數如果是n，就相當於建立起n個分段鏈接的free lists，每一段預設64個blocks。MFC配置器的整體設計結構和Loki比較接近。

圖7是CPlex和CFixedAlloc的關係。有趣的是雖然MFC有此配置器，但需要頻繁運用配置器的容器（MFC術語稱之為Collection classes）如各種CxxxList和 CMapxxxToxxx，都不使用它而是直接在自己class內也按著葫蘆畫瓢地做一遍CFixedAlloc的工作，白白喪失了復用價值。

順帶一提，用過MFC的程式員都知道MFC特愛提供各種巨集。由於為class設計池式配置是如此標準化而制式，不外乎就是為該class重載operator new和operator delete並在其中呼叫池式配置器提供的配置和歸還函式（如上述的Alloc()和Free()），因此MFC也為此設計了一套巨集：DECLARE_FIXED_ALLOC和IMPLEMENT_FIXED_ALLOC。MFC程式員看到這兩個名稱一定露齒微笑。這組巨集在CString及CTempXXX中被用到。

Boost.Pool的作法

圖8是Boost.Pool的示意圖。一圖解千言

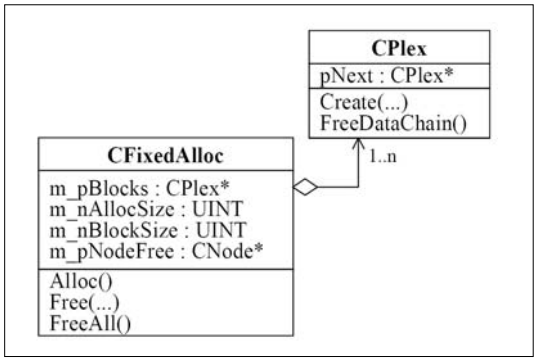

圖7 MFC配置器的二個主要classes的關係。

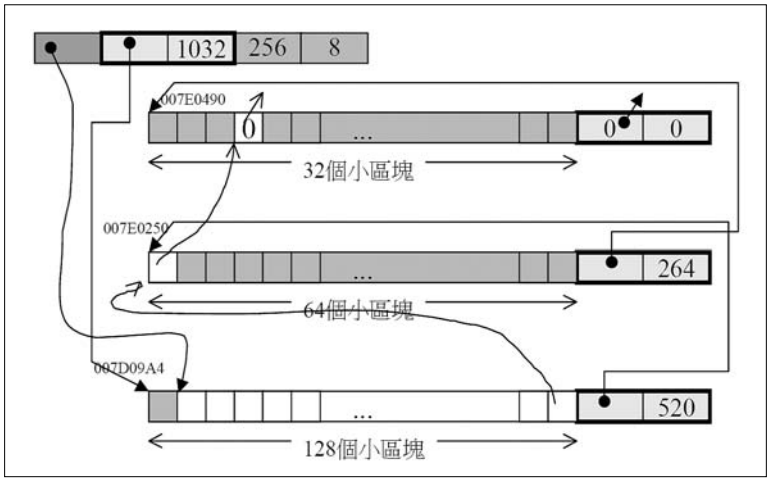

圖8 Boost.Pool的行為示意圖。

萬語，從如此精心設計的圖（well，我又老王賣瓜了。其實是要提醒你仔細看圖）便可快速而良好地了解Boost.Pool的運作。要提醒你的是，Boost對於chunk和block兩個術語的用法和其他系統的習慣都不相同，剛好相反，它稱呼小區塊為chunk，大區塊為block。我無法權宜地在本文中調整稱呼使與STL, Loki, MFC的習慣相同，因為這些術語也反應在源碼中，而我稍後要解釋Boost源碼。所以請務必記住這一變化，切切。

Boost.Pool和Loki及MFC一樣維護不限個數的free lists，也就是說它為各式各樣的尺寸服務。它也和Loki和MFC一樣對每個free list都採取分段鏈接，只不過每個分段不似Loki或MFC那樣有固定的chunk（小區塊）個數，而是每次向系統要求block（大區塊）時便累加兩倍chunks數；一開始是32。這便是圖8中的32,64,128小區塊數的由來。因為chunks個數會變，所以它不可能以Loki::Chunk那種方式管理chunks，而是採用類似MFC's CPlex的作法，不過這次是以「尾部」來加持灌頂，這個尾部被設計為class PODptr（在命名空間boost::details內，只供Boost內部使用），不但必須像MFC's CPlex那般維護一個指標指向下一分段，而且必須記錄下一分段的大小（bytes數），因為每一分段大小不同嘛（大致兩倍增長）。還請注意，Boost.Pool刻意讓每一分段以起始點實際位址之高低排列於free list，如圖8三個分段起始位址，007D09A4段的尾部指向007E0250段，而後者的尾部指向最後一段007E0490，這將有利於將來接受chunks歸還時比較容易實作。

管理free list的是boost::pool。它必須擁有一個PODptr（圖8上方灰色部分）指向該free list中的第一個blocks，還需要記錄chunk大小和下次向系統要求的chunks數（圖8上方綠色部分），還要有一個指標指向目前可用的chunk（圖8上方紫色部分）。稍後我們可以從源碼中看到這些資料從何而來。

此前限於篇幅，我沒有提過STL, Loki, MFC的小區塊歸還動作（deallocation）。今天的主角是Boost，我們可得好好說事。以圖8為例，該圖表現的是已被客戶要走32（第一個block全部）+64（第二個block全部）+1個chunks（圖中灰色小區塊），之後又歸還兩個chunks（白色小區塊）的情況。歸還的兩個分別是中間block的第一個chunk和上方block的第4個chunk。由於各個分段（blocks）以位址在free list中由低而高排列，因此可輕易計算出被歸還的chunk座落於哪一個block內，從而順利完成指標的移轉和整個free list的維護。當任何（或所有）blocks中的所有chunks都回收完畢，理論上這裡記錄有足夠的信息可以把整個block（或所有blocks）釋還給系統。但Boost沒那麼做。原因呢？唔，Boost沒說，我也不好說。也許你可以為它強化這一點，這是很好的練功題。Boost唯一釋還blocks的時機，是在配置器離開作用域造成其解構式被喚起時（這和MFC很像），那有點時不我予，唔，太晚了，沒多大幫助！

在這種設計架構下，Boost.Pool的用法像這樣：

```
boost::object_pool<Foo> fooPool;
std::vector<Foo*> v;
for (int i = 0; i < 10; ++i)
```

```
  v.push_back(fooPool.construct(1,2)); //會喚起Foo
建構式並傳入(1,2)
…
for (int j = 0; j < 5; ++j)
  pool.destroy(v[j]);
```

Boost.Pool的classes結構和源碼剖析

Boost.Pool主要有6個.hpp檔案，位於\Boost\include\boost-1_33\boost\pool。各檔案的主要內容見圖9，最後兩個檔案本文不討論。

Boost.Pool 的主要檔案	每個檔案內的主要內容
poolfwd.hpp	宣告 6 個主要 classes 及其他輔助 classes（如下）
object_pool.hpp	template<T> class **object_pool**;
pool.hpp	template<T> class **pool**; struct **default_user_allocator_new_delete**; struct **default_user_allocator_malloc_free**; details:: template<T> **PODptr**
simple_segregated_storage.hpp	template<T> class **simple_segregated_storage**;
pool_alloc.hpp	class **pool_allocator**; class **fast_pool_allocator**;
singleton_pool.hpp	struct **singleton_pool**;

圖 9　Boost.Pool的主要檔案。

其中的poolfwd.hpp宣告出所有相關classes，詳見程式1。Boost的classes設計有點複雜，彼此有繼承關係(而且是罕見的protected繼承)，這在前面所談的各程式庫中是沒有的，稍後我有詳細說明。

程式1　poolfwd.hpp源碼。

```
// Copyright (C) 2000, 2001 Stephen Cleary
//
// Distributed under the Boost Software License,
Version 1.0. (See
// accompanying file LICENSE_1_0.txt or copy at
// http://www.boost.org/LICENSE_1_0.txt)

#ifndef BOOST_POOLFWD_HPP
#define BOOST_POOLFWD_HPP

#include <boost/config.hpp> // for workarounds
#include <cstddef>   // std::size_t

// boost::details::pool::default_mutex
#include <boost/pool/detail/mutex.hpp>

namespace boost {

// Location: <boost/pool/simple_segregated_storage.
hpp>
template <typename SizeType = std::size_t>
class simple_segregated_storage;

// Location: <boost/pool/pool.hpp>
struct default_user_allocator_new_delete;
struct default_user_allocator_malloc_free;

template <typename UserAllocator =
          default_user_allocator_new_delete>
class pool;

// Location: <boost/pool/object_pool.hpp>
template <typename T,
          typename UserAllocator =
            default_user_allocator_new_delete>
class object_pool;

//註：以下宣告singleton_pool, pool_allocator,
fast_pool_allocator
//    不在本文討論之列，故略。
  //…
} // namespace boost
#endif
```

剛才我們看過Boost.Pool的用法，客戶用的是最上層的boost::object_pool<T>。這個class本身不帶任何成員變數。它繼承boost::pool，因此從物件模型的角度來說其object相當於內含了一個boost::pool（亦即所謂的base class part)。這樣的雙層classes切割只是為了將上下層函式功能做個區隔，沒有其他意思(沒有虛擬函式、沒有多型作用)，所以請注意，這裡使用的繼承關係是較為罕見的protected繼承而非常見的public繼承：

```
template <typename T, typename UserAllocator>
class object_pool: protected pool<UserAllocator>
{ … }
```

相對較低階的boost::pool負責管理free list。先前說過它是利用一個個PODptr個別管理一個個blocks(大區塊)。從源碼摘錄可看到boost::pool的確內含一個PODptr(程式進行過程中隨著blocks的不敷使用，此類PODptr還

會陸續被創建出來）。boost::pool也把若干基礎工作切割給simple_segregated_storage，並再次以protected繼承方式實現上下層關係：

```
template <typename UserAllocator>
class pool: protected simple_segregated_storage<
                    typename UserAllocator::size_type>
{
 protected:
   details::PODptr<size_type> list;
   const size_type requested_size;
   size_type next_size;
 ...
};
```

前面所說的PODPtr並不對外開放，被定義於boost內層的namespace details中。稍後我會詳細說明其內的兩個成員變數：

```
//此處位於namespace boost內
namespace details {
template <typename SizeType>
class PODptr
{
 private:
   char * ptr;
   size_type sz;
 ...
};
} // namespace details
```

被boost::pool切割出去的更基礎工作由父類別simple_segregated_storage完成：

```
template <typename SizeType>
class simple_segregated_storage
{
 protected:
   void * first;
 ...
};
```

圖10顯示上述四個classes的關係。圖11是object_pool的物件模型（記憶體佈局），其中紫色、灰色、綠色分別由圖10同色class的成員變數貢獻。稍後我要解釋這四個classes的主要行為並帶領大家看源碼。

有了以上大局觀，現在我們可以任意討論任何一層而不致迷失方向了。

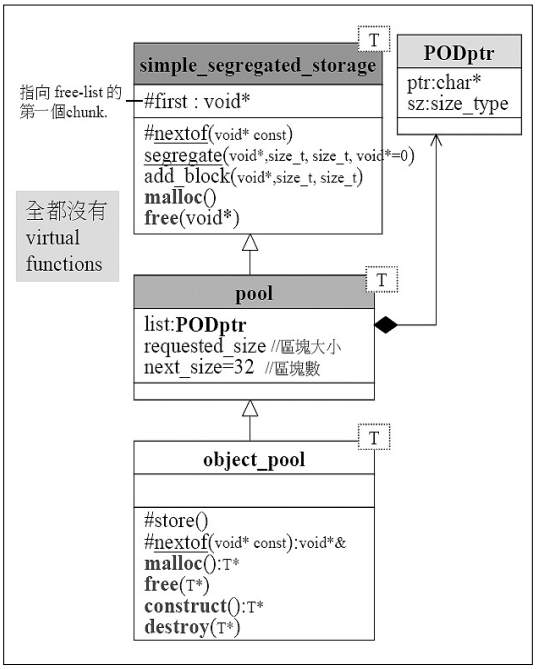

圖10 Boost.Pool的四個主要classes的關係。

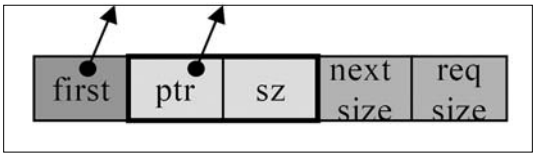

圖11 boost::object_pool的物件模型（記憶體佈局）。

參數UserAllocator決定實際用上的配置／釋放工具

UserAllocator是一個template參數，主要出現在下面這程式2與程式3幾個地方：

程式2

```
template <typename T, typename UserAllocator>
class object_pool: protected pool<UserAllocator>
{ … }
```

程式3

```
template <typename UserAllocator>
class pool: protected simple_segregated_storage<
                    typename UserAllocator::
size_type>
{
 …
};
```

顧名思義UserAllocator是一個可由用戶指

定的配置器，換句話說Boost允許用戶自行決定最底層的配置／釋放動作採用哪一套基本工具。先前例中並沒有指定這個參數：

```
boost::object_pool<Foo> fooPool;  //先前是這麼寫的
```

那是因為它有預設值，定義於poolfwd.hpp，見圖10。甚至連boost::pool也在其宣告式中指定了自己的這個預設參數值，亦見圖10。

程式4與程式5都是可做為UserAllocator用途的兩個classes。Boost預設使用程式4。

程式4

```
struct default_user_allocator_new_delete
{
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;

  static char * malloc(const size_type bytes)
  { return new (std::nothrow) char[bytes]; }
  static void free(char * const block)
  { delete [] block; }
};
```

程式5

```
struct default_user_allocator_malloc_free
{
  typedef std::size_t size_type;
  typedef std::ptrdiff_t difference_type;

  static char * malloc(const size_type bytes)
  { return reinterpret_cast<char *>(std::malloc(bytes)); }
  static void free(char * const block)
  { std::free(block); }
};
```

兩者都提供malloc()和free()，唯一差別在於其一內部用的是new[] 配置記憶體並以delete[] 釋放，另一個以std::malloc() 配置並以std::free()釋放。其實兩者在常見的情況下效果相等。

boost::simple_segregated_storage

這個class的命名意義是「經過簡易切割處理後的儲存空間」。它維護一個指標first，指向第一個可用小區塊。

```
template <typename SizeType>
class simple_segregated_storage
{
  protected:
    void * first;
  ...
};
```

它的建構式將指標first初始化為null。

```
simple_segregated_storage()
    : first(0) { }
```

將來用戶要求分配小區塊時，經過層層函式呼叫，只要這個first不是null，就直接拿first所指的小區塊來交差。

如果first指向null，就表示這個「經過簡易分割處理後的儲存空間」已經耗盡（或從來沒有過）：

```
bool empty() const { return (first == 0); }
```

在「非空」前提下，小區塊的分配是這麼做的：

```
// pre: !empty()
 void * malloc()
 {
   void * const ret = first;

   // Increment the "first" pointer to point to the next
chunk
   first = nextof(first);
   return ret;
 }
```

是的，在傳回可用小區塊的位址之前，當然要先調動first指標，令它指向下一個可用小區塊。但為什麼竟是將它設為nextof(first)的返回值呢？這裡有兩件事需要討論，第一，可用小區塊是如何佈署的？第二，nextof()做了什麼？

先回答第一個問題：每一個可用小區塊（chunk）都是在向系統要一個大區塊

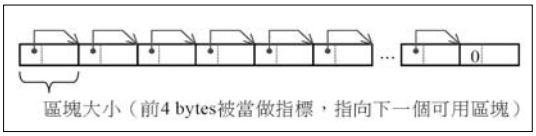

圖12

（block）之後按給定尺寸切割出來的，採用嵌入式指標完成linked list，像圖12這樣（這些動作在segregate()內完成，限於篇幅不列源碼）。

因此我們必須調整first令它指向下一個可用小區塊。也就是將first的內容設為「目前first所指物之前4 bytes的內容」。問題是first本身型別是void*，不能允許提領內容。如果先將first轉型為void**（只是轉型唷，其值仍是原本的那個指標值），對它取值就會取到一個void*（4 bytes），那正符合上圖顯示的佈局。那就是nextof()的意義：

```
static void * & nextof(void * const ptr)
   { return *(static_cast<void **>(ptr)); }
```

這個函式主要用來處理從first指標一脈所指的指標。

boost::details::PODptr

PODptr的命名表示，這是一個被當做指標的東西，用來指向一個POD。什麼是POD呢？常常出現於C++文獻的一個字眼，意思是「Plain Old Data」，簡樸的舊式資料型態，你也可以認知它是C++ class以外的任何型式的資料組合，其大小就是其成員的單純總和。C struct就是一種POD。

Boost.Pool利用不對外公開的這個PODptr來鏈接一個個blocks，形成一個free list。POD代表的是「向系統索求一大塊block再加上必要的管理元素」，而所謂「管理」就是「當block內的每個chunks都已被用掉，就鏈接另一個blocks以便供應更多chunks」。PODPtr結構如下：

```
//此處在namespace boost之內
namespace details {
template <typename SizeType>
class PODptr
{
 private:
   char * ptr;
   size_type sz;
 ...
};
} // namespace details
```

它一開始可能是圖13這樣（以下數字皆為某種情況下的實例），然後是圖14這樣。當第一個block的所有chunks統統被取光後，再變成圖15這樣。當第二個block的所有chunks統統被用光後，又變成圖16這樣。

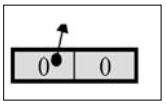

圖13

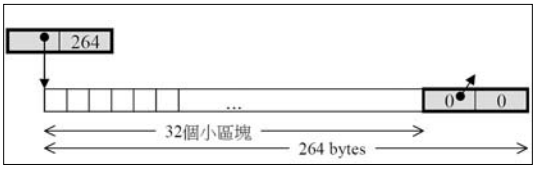

圖14

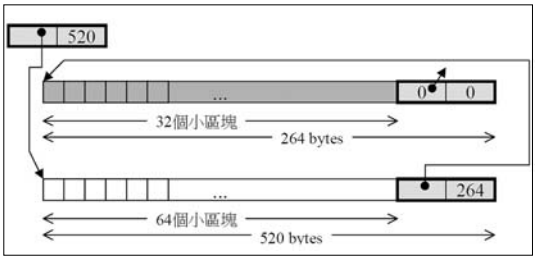

圖15

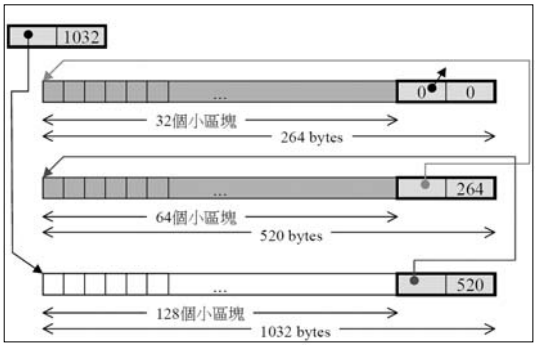

圖16

三階段初始化

當用戶這麼寫：

```
boost::object_pool<Foo> fooPool;
```

首先喚起建構式：

```
explicit
object_pool(const size_type next_size = 32)
  : pool<UserAllocator>(sizeof(T), next_size)
{ }
```

更早一步又先喚起子類別boost::pool的建構式，並且T是Foo，UserAllocator是預設值default_user_allocator_new_delete：

```
explicit
  pool(const size_type nrequested_size,
       const size_type nnext_size = 32)
    : list(0,0),requested_size(nrequested_size),
      next_size(nnext_size)
  { }
```

這便設定了三個成員變數的初值：區塊大小（nrequested_size）被設為sizeof(Foo)，下次配置區塊數（next_size）被設為32，「特形指標」PODptr的sz被設為0，ptr被設為0。在此之前還會先喚起base class建構式：

```
simple_segregated_storage()
  : first(0) { }
```

設定指標first為0。因此，初初建立一個為Foo服務的object_pool獲得的結果如圖17。

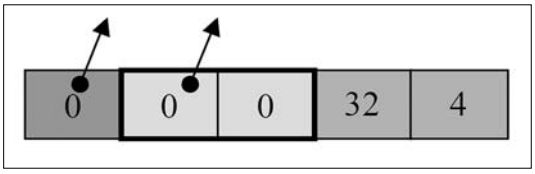

圖17

三階段建構／配置（construct()/malloc()）

欲從上述的fooPool配置區塊，可呼叫construct()或malloc()。前者會呼叫後者並於配置點建構一個元素，後者只是單純配置記憶體。先看前者，參考程式6。

程式6

```
//註：這裡是object_pool class 定義式內部
//因此以下出現的 element_type在先前例中就是Foo

   element_type * construct()
   {
     element_type * const ret = malloc();
     if (ret == 0)
       return ret;
     try { new (ret) element_type(); }
     catch (...) { free(ret); throw; }
     return ret;
   }
#ifndef
BOOST_NO_TEMPLATE_CV_REF_OVERLOADS
#   include <boost/pool/detail/pool_construct.inc>
#else
#   include <boost/pool/detail/pool_construct_simple.
inc>
#endif
```

construct()另有許多版本，全都定義於上述最後出現的兩個 .inc 之一（上述最後的條件編譯用來判斷編譯器是否允許template參數擁有const和volatile，即所謂CV modifiler）。如果用戶這麼寫：

```
boost::object_pool<Foo> fooPool;
fooPool.construct('a', 10);
//假設已知Foo建構式第一參數為 type,第二參數為
value
```

construct()相當於：

```
Foo * const ret = malloc();
 if (ret == 0)
   return ret;
 try { new (ret) Foo(a0, a1); }
 catch (...) { free(ret); throw; }
 return ret;
```

於是我們追蹤上述第一行的boost::object_pool::malloc()：

```
// Returns 0 if out-of-memory
   element_type * malloc()
   { return static_cast<element_type *>(store().
ordered_malloc()); }
```

這裡最後一行呼叫了一個奇怪的函式store

()。那是什麼東東？

在Boost.Pool的三層繼承體系中，由於存在許多簽名式（signature）相同的non-virtual函式，例如：

- boost::object_pool::malloc()；
- boost::pool::malloc()；
- boost::simple_segregated_storage::malloc()；

以及：

- boost::pool::free(void* const chunk)；
- boost::simple_segregated_storage::free(void* const chunk)；

以及：

- boost::pool::order_free(void* const chunk)；
- boost::simple_segregated_storage::ordered_free(void* const chunk)；

以及：

- boost::pool::nextof(void* const ptr)；
- boost::simple_segregated_storage::nextof(void* const ptr)；

這些上下層提供的「簽名式相同的non-virtual函式」彼此遮蔽（hide），因此derived class若欲呼叫其base class中的這類函式，辦法之一是先將自己轉型為base class（這是up-cast，一定安全）。而store()就做這件事情：

```
//註：這裡是boost::object_pool定義式內部
  protected:
    pool<UserAllocator> & store() { return *this; }
    const pool<UserAllocator> & store() const { return
*this; }
//註：這裡是boost::pool定義式內部
  protected:
    simple_segregated_storage<size_type> & store()
    { return *this; }
    const simple_segregated_storage<size_type> &
store() const
    { return *this; }
```

這些函式都是將自己（'this' object）傳出去，並強制改頭換面成為另一種type。由於以by reference方式傳出，傳的是本尊而不是複件。改頭換面之後可被拿來呼叫新身份之任何函式而不受遮蔽（hide）之苦了。

回到先前討論的：

```
// Returns 0 if out-of-memory
  element_type * malloc()
  { return static_cast<element_type *>
            (store().ordered_malloc()); }
```

我們已知store()的意義，因此最後一行喚起boost::pool::ordered_malloc()，如圖18。

顯然是要看看first指標是否「所指有物」。我們知道一開始first指標為0，所以判斷之後喚起的函式應該是pool的ordered_malloc_need_resize()。這個函式可就複雜些了，作用相當於配置一大區塊、切割為若干小區塊並鏈接為一個linked list。源碼分析太佔篇幅，不適合在雜誌上呈現。總之，獲得的成果如圖19。

爾後用戶如果再呼叫31次fooPool.construct()或fooPool.malloc()，情況如圖20，

圖18

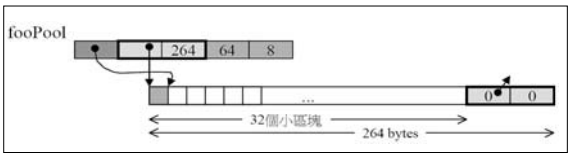

圖19

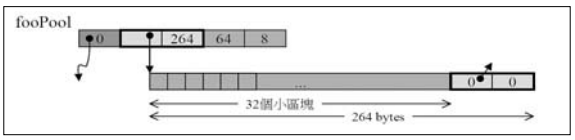

圖20

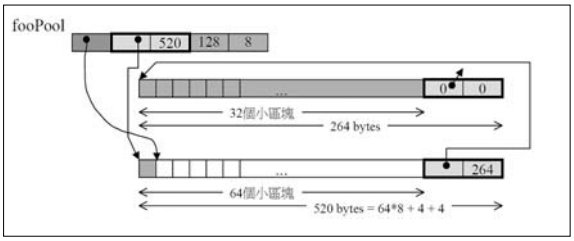

圖2 1

注意所有32個chunks都被取用了，first指標又變成null。

當用戶再次呼叫fooPool.construct()或fooPool.malloc()，情況如圖21（又一次配置、細切、鏈接整合、給出一個chunk）。

三階段銷毀／釋放（destroy()/free()）

當客戶欲歸還來自fooPool的chunks，可呼叫destroy()或free()。前者會先呼叫chunk位址上的那個object的解構式然後呼叫後者：

```
//註：在 boost::object_pool定義式內
void destroy(element_type * const chunk)
{
  chunk->~T();
  free(chunk);
}

void free(element_type * const chunk)
{ store().ordered_free(chunk); }
```

繼續呼叫下去：

```
//註：在 boost::pool定義式內
void ordered_free(void * const chunk)
{ store().ordered_free(chunk); }
```

再繼續呼叫下去：

```
//註：在 boost::simple_segregated_storage定義式內
// pre: chunk was previously returned from a malloc()
referring
//  to the same free list
// post: !empty()
void ordered_free(void * const chunk)
{
// This (slower) implementation of 'free' places
//  the memory back in the list in its proper order.

// Find where "chunk" goes in the free list
    void * const loc = find_prev(chunk);

// Place either at beginning or in middle/end
    if (loc == 0)
      free(chunk);
    else {
      nextof(chunk) = nextof(loc);
      nextof(loc) = chunk;
    }
}

void free(void * const chunk)
{
  nextof(chunk) = first;
  first = chunk;
}
```

其中呼叫的find_prev()我就不列源碼了，只以圖片說明情況的變化。

假設目前有一個block已分配5 chunks出去，指標first指向第6個chunk，如圖22。歸還p2指標時，結果如圖23。歸還p4指標時，結果如圖24。

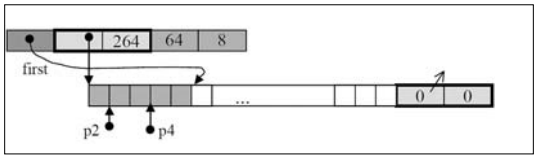

圖2 2

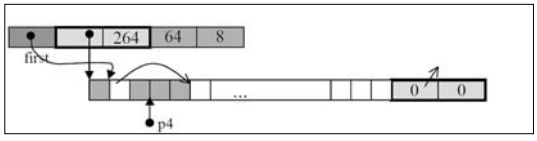

圖2 3

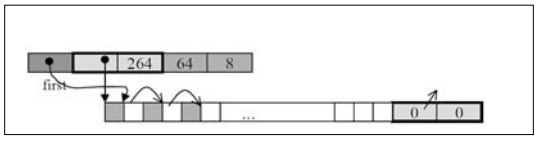

圖2 4

換個例子。假設目前有3個blocks如圖25（注意free lists的每個blocks都按其實際位址高低鏈接起來，由低而高，例如圖中blocks的起始點先是007D09A4然後是007E0250再然後是007E0490）。歸還p4，造成如圖26。再歸還pp1，造成如圖27。

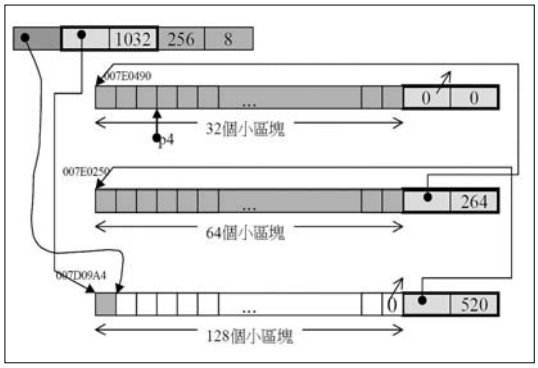

圖25

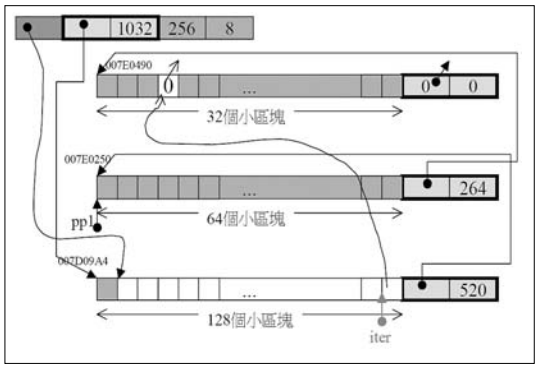

圖26

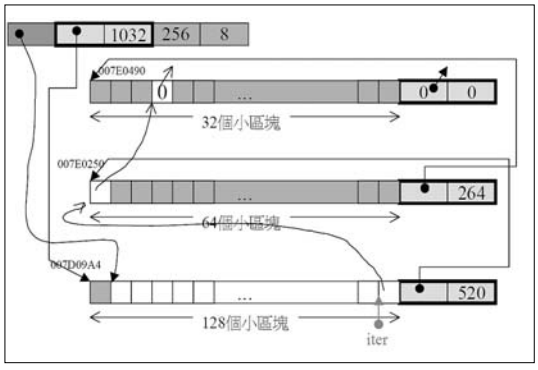

圖27

不論chunks被歸還多少，這些blocks不會被釋還給系統。只要記憶體被Boost.Pool拿去，在boost::object_pool object結束生命前不會釋放記憶體。這一點很像std::alloc，後者更是直到程式結束才釋放記憶體。

離開作用域時的清理動作

object_pool的解構式會以迴圈逐一找出每個POD，並對其內每一個allocated chunks呼叫其上的物件的解構式，完畢後將該POD釋還給系統。

pool的解構式亦是逐一找出每一個POD，將該POD釋還給系統。它和object_pool的解構式很像，唯一差別是它不會逐一呼叫每個區塊上的物件的解構式。換句話說object_pool解構式和pool解構式各自獨立，不存在誰呼叫誰。

以上就是Boost.Pool的故事。同時我們也一日看盡長安花，對STL, Loki, MFC的Pooled Allocation也有了底。我為多家（尤其是嵌入式需求為主的）軟體公司開「C++記憶體管理」課程時，這些程式庫的作法是重要的參考對象。下一期我們談Boost的Any, Array和Tuple。

責任編輯／洪羿漣

更多資訊

以下是與本文相關的讀物或網上資源。

- 《Effective C++》3/e, by Scott Meyers，chap8 "Customizing new and delete".
- 《STL源碼剖析》by侯捷，第二章「空間配置器」。
- 《Modern C++ Design》by Andrei Alexandrescu，chap4 "Small Object Allocation".
- 《深入淺出MFC》2/e by侯捷。
- Boost Libraries Documentation, from http://www.boost.org.

作者介紹

侯捷

資訊顧問、專欄主筆、大學執教。常著文章自娛，頗示己志。
侯捷網站：http://www.jjhou.com（繁體）
北京鏡站：http://jjhou.csdn.net（簡體）
永久郵箱：jjhou@jjhou.com