侯捷觀點

池內春秋
Memory Pool 的設計哲學和無痛運用

北京《程序員》2002.09
台北《Run!PC》2002.09

作者簡介：侯捷，電腦技術作家，著譯評兼擅。常著文章自娛，頗示己志。
電子郵箱：jjhou@jjhou.com
侯捷網站：http://www.jjhou.com
簡體鏡站：http://jjhou.csdn.net

- 讀者基礎：有一定程度的 C++ 編程經驗
- 本文適用工具：GNU C++ 編譯器
- 本文關於 SGI STL 之剖析，部分已載於《STL 源碼剖析》第二章；嶄新內容包括 SGI STL 區塊卸除（歸還）動作分析、缺點與補強之道、無痛應用、三種編譯器之區塊配置效能比較。
- 術語：memory pool（記憶池），free list（自由串列），free block（自由區塊），allocator（配置器），heap（堆積），client（客端）。

為什麼需要記憶池

記憶體曾經是兵家必爭之地，曾經被喻爲「CPU 之外最寶貴的電腦硬體資源」。在那「640K 天塹[1]」的遠古年代裡，程式員對記憶體錙銖必較的程度可能令生活於「虛擬記憶體」環境下的當今世代瞠目結舌，千禧年（Y2K）蟲蟲危機即肇因

[1] MS-DOS 5.0 以前，PC 環境上只能開發 640K 以下的程式。640K 內必須含括作業系統本身、應用程式碼本身、以及應用程式的資料量。MS DOS 5.0 強化了 640KB 至 1024KB 之間（UMB）定址空間的運用，以及 1024K 以上少量定址空間（HMA）的運用。

於當初過份樽節記憶體[2]。當時的人們（我也屬其中之一）即便在 config.sys 中揮汗調校只省下區區數十個 bytes，都會覺得歡欣鼓舞；如果有人能夠運用 int67h 進入 EMS 記憶體或運用 int21h 進入 XMS 記憶體[3]，更可說是走路有風，呵水成凍。1991 年微軟發佈的 MS-DOS 5.0，涵蓋數個定址相關技術（UMB：Upper Memory Block，HMA：High Memory Area），大幅度提昇 MS-DOS 的定址能力，當時被譽爲「突破性的進展」。

曾幾何時，當虛擬記憶體作業系統（如 Windows、OS/2、Linux）走進群眾，苦樂俱往矣。非人道的痛苦折磨被迅速遺忘，緇銖必較的軼趣成了白頭宮女話天寶當年的回憶。我們不再被程式碼大小所限，也不再被資料量所限。所有記憶體不足的問題只要「加一條 256M 記憶體」就獲得解決。從這個角度看，程式員的生活幸福美滿。

當溫飽獲得解決，人們要求精緻。軟體開始往兩個方向發展：一是更快，一是更小。系統級軟體或特殊應用或資料量極大的軟體，要求運行極快；掌上系統或嵌入式系統則因先天硬體環境的限制而必須體積極小。於是記憶體問題又再度浮上檯面。

不論是速度問題或空間問題，都肇因於編譯器給的那些個彈性極大的記憶體配置工具帶來了一些額外開銷（overhead）。當額外開銷的比率超過你的容忍限度，你免不了要衝冠一怒尋求突破。最簡單而效果良好的一種技術就是 **memory pool**（記憶池）。在正式介紹 memory pool 技術之前，我要先帶你徹底了解 C++ 編譯器的記憶體配置策略。

---

[2] 當時的程式員爲節省記憶體用量，將年份 19xx 僅儲存爲 xx，以至於時序進入西元 2000 之後無法正常進位。

[3] EMS：Expanded Memory Spec.，XMS：eXtended Memory Spec.，兩者都是記憶體擴展（延伸）規格。詳見拙作《虛擬記憶體：觀念、設計與實作，*using EMS and XMS*》，1991，旗標出版。

C++ 平台供應的記憶體配置工具

在 C++ 平台上，你可以透過圖一所列的四種方式動態索求及釋放記憶體。其中第二組呼叫第三組，第三組功能及行爲等同於第一組。第四組的「功能」相當於第一或第三組，其最終配置動作亦需仰賴第一或第三組完成，但我們可在第四組內部設計出複雜精巧的 memory pool 機制。

配置	釋放	歸屬	可重載否	標記
malloc()	free()	C 標準函式	不可	(1)
new	delete	C++ 運算式	不可	(2)
::operator new()	::operator delete()	C++ 運算子	可	(3)
alloc::allocate()	alloc::deallocate()	C++ 標準程式庫之記憶體配置器	可自由設計並裝載於任何容器身上提供服務	(4)

圖一\ C++ 語言及程式庫供應的四種動態記憶體配置求釋放方式。表中之 alloc 代表配置器 allocator，其名稱雖有正式規範，但不同的實作品可能有不同命名（後述）。

下面是這四種記憶體配置工具的運用示範：

```
  // 四種配置方式
  void* p1 = malloc(512);
  free(p1);

  CRect* pr = new CRect;
  delete pr;

  void* p2 = ::operator new(512);
  ::operator delete(p2);

  // 以下是 C++ 標準程式庫中的配置器。其介面雖有標準規格，但各廠商多未遵守。
  // 以致下面三種型式各異（稍後另有說明）。
#ifdef _MSC_VER
  // 以下兩者都是 non-static，所以一定要藉由物件喚起之
  int* p3 = allocator<int>().allocate(512, (int*)0);
  allocator<int>().deallocate(p3,512);
#endif
#ifdef __BORLANDC__
  // 以下兩者都是 non-static，所以一定要藉由物件喚起之
  int* p3 = allocator<int>().allocate(512);
  allocator<int>().deallocate(p3,512);
```

侯捷觀點

```
#endif
#ifdef __GNUC__
  // 以下兩者都是 static，可透過全名喚起之
  void* p3 = alloc::allocate(512);
  alloc::deallocate(p3,512);
#endif
```

如果你要的不只是單純的記憶體配置，甚且希望直接在記憶體上建構物件（C++ 程式總是如此），那麼你的唯一選擇是第二組，因爲第二組不但配置（釋放）記憶體，還自動喚起物件的建構式（解構式），如圖二。

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

編譯器轉爲...

```
Complex *pc;
try {
     void* mem = ::operator new( sizeof(Complex) ); // 配置
     pc = static_cast<Complex*>(mem);                // 轉型
     pc->Complex::Complex(1,2);                      // 建構
     // 注意：只有編譯器才能夠像上面那樣直接呼叫 ctor
}
catch( std::bad_alloc ) {
     // 記憶體配置失敗，不執行建構式
}
```

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

編譯器轉爲...

```
pc->~Complex();          // 解構
::operator delete(pc);   // 釋放記憶體
```

圖二/ new/delete 運算式會呼叫 new/delete 運算子，及對應的建構式/解構式。

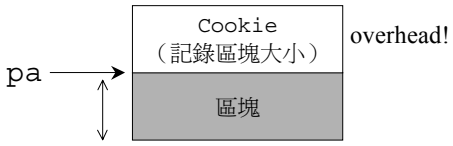
在這四組工具中，唯有第三組允許被重載、第四組允許由實作廠商（或客戶端程式員自己）發揮。因此當我們打算設計一個更高效的記憶體配置器時，關鍵便著落在第三、四兩組身上。

空間上的額外開銷

C++ 平台（語言 + 程式庫）所提供的記憶體配置工具中，前三組都會帶來額外開銷，這個額外開銷被暱稱爲 cookie（小甜餅），用來標示配置所得的區塊大小，如圖三。如果沒有 cookie 的存在，一旦你想釋放先前配得的記憶體，將手上的指標傳給前三組的相應釋放函式時，釋放函式無法從指標身上看出區塊大小，也就無法做出正確的回收（納入 system heap）動作。

速度上的額外開銷

由於 C++ 系統提供的配置工具（前三組）允許你任意指定區塊大小，因此 system heap 一旦開始經歷配置和回收，將出現群雄割據的雜亂局面（但都在掌控之中）。爲了充份運用記憶體，系統配置工具必須有一套演算法，用來走訪整個 system heap，找出最適合需求的一塊完整空間。System heap 愈雜亂、區塊大小愈是參差不等，找出一個適當（夠大）區塊的時間愈可能長久[4]。



圖三/ 每個由 system heap 配置而來的記憶體區塊，前端都帶有一塊「小甜餅」，大小通常爲 4 bytes，用來記錄區塊大小。

空間額外開銷之 Proof

欲證明你所配置的每一個記憶體區塊都帶有「小甜餅」，很簡單，只要觀察動態配置得來的指標，看看其前方是否有所記錄？記錄值是否就是區塊本身的大小？

---

[4] 感謝孟岩先生對於 malloc()提供以下補充說明：Memory Pool 主要是針對 native API malloc 或 operator new 不夠高效而開發。這種情況在以前比較多見。後來很多人都開始針對這個問題進行深入研究。1986 年起 Doug Lea 潛心研究 malloc 演算法，逐漸發展出比較好的作法，被稱爲 DL Malloc，目前 Linux 的 glibc 中的 malloc 演算法就是直接來自 Doug Lea，其他平臺的 malloc 實作品多少也受到 DL 的影響。總的來說，如今的 malloc 比以前快很多，這可能是爲什麼 PJ Plauger 等直接使用 operator new 實作 allocator 的原因。Doug Lea 主頁：http://gee.cs.oswego.edu/dl/，其中可下載 DL Malloc 源碼。

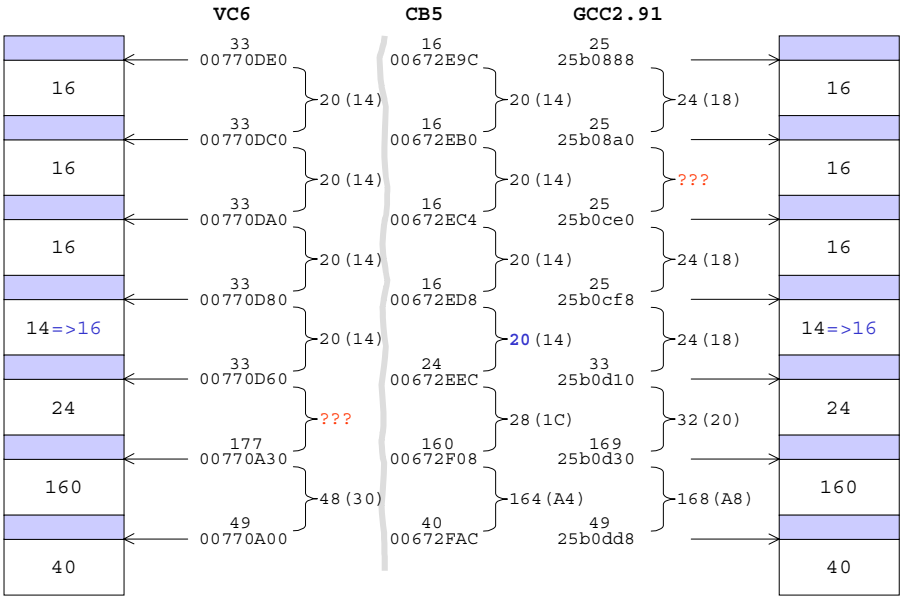
同時並觀察數個連續動態配置的區塊是否完全緊鄰？亦或中間有些縫隙？縫隙的大小是否剛好 4bytes？

下面這段程式碼刻意安排特定的物件大小，然後動態配置它們（並建構之），然後列印其前四個 bytes，觀察其內數值，印證是否和區塊的大小相同。最後並觀察這些區塊的起始位址。由於只做簡單觀察之用，所以編程手法直觀而不講究。

```
// 動態配置記憶體
C1* pc1 = new C1;          // size:16
C1* pc12 = new C1;         // size:16
C1* pc13 = new C1;         // size:16
C2* pc2 = new C2;          // size:14
C3* pc3 = new C3;          // size:24
C4* pc4 = new C4;          // size:160
int* pi = new int[10];     // size:40

// 列印每個區塊之前的四個 bytes
unsigned char* pc1c  = (unsigned char*)pc1;
unsigned char* pc12c = (unsigned char*)pc12;
unsigned char* pc13c = (unsigned char*)pc13;
unsigned char* pc2c  = (unsigned char*)pc2;
unsigned char* pc3c  = (unsigned char*)pc3;
unsigned char* pc4c  = (unsigned char*)pc4;
unsigned char* pic   = (unsigned char*)pi;
printf("%d,%d,%d,%d \n", *(pc1c-4), *(pc1c-3), *(pc1c-2), *(pc1c-1) );
printf("%d,%d,%d,%d \n", *(pc12c-4), *(pc12c-3), *(pc12c-2), *(pc12c-1) );
printf("%d,%d,%d,%d \n", *(pc13c-4), *(pc13c-3), *(pc13c-2), *(pc13c-1) );
printf("%d,%d,%d,%d \n", *(pc2c-4), *(pc2c-3), *(pc2c-2), *(pc2c-1) );
printf("%d,%d,%d,%d \n", *(pc3c-4), *(pc3c-3), *(pc3c-2), *(pc3c-1) );
printf("%d,%d,%d,%d \n", *(pc4c-4), *(pc4c-3), *(pc4c-2), *(pc4c-1) );
printf("%d,%d,%d,%d \n", *(pic -4), *(pic -3), *(pic -2), *(pic -1) );

// 列印每一區塊的起始位址
printf("%p %p %p %p %p %p %p \n", pc1,pc12,pc13,pc2,pc3,pc4,pi);
```

圖四是以上片段的某次執行結果。我們看到 C++Builder5 在這方面的表現最爲「中規中矩」，完全符合我們的預期，不僅 cookie 記錄「正確」，每個區塊的距離也恰恰是區塊大小加上 4（cookie 大小）。GCC 產生的每個 cookie 記錄的似乎都是區塊實際大小加上 1001（二進制），像是某種神秘標記；每個區塊距離則是區塊大小加上 8，但也有例外（紅色標示）。VC6 產生的每個 cookie 以及區塊的距離最難推斷規律性，並且也有十分離奇的現象（紅色標示）。這些離奇現象其實不

難理解：我們在測試程式中連續配置數個區塊，但 malloc()或 ::operator new 並不一定就在 system heap 中找到連續空間，這時候 cookie 之中有某種「神秘」記錄是很容易想像的。就連 C++Builder 也不見得一直會有上述那麼「中規中矩」的表現。



圖四/ 觀察連續配置而得的區塊。圖中灰色即爲 cookie，白色爲配置而得的區塊，其內數值（10 進制）表示程式的需求大小。每個箭頭旁的數值代表位址，兩位址間的大括號旁的數值代表距離（bytes，10 進制），旁邊小括號內爲 16 進制表示式。緊鄰每個位址之上的數值爲實際觀察得到之 cookie 內容（10 進制）。

速度額外開銷之比較

很難像圖四那麼絕對地向大家證明速度上的額外開銷，因爲並不存在一個標準可供比較。但我們可以試著在不同編譯器上配置大量區塊，觀察其速度表現：

```
printLocalTime();
for(int i=0; i<10000000; ++i)  new C1;    //sizeof(C1) : 16
printLocalTime();
```

上述程式片段配置了一千萬個 C1 物件，動作前後分別列印出當時時間[5]。某次執行結果如圖五。我們發現，不同編譯器上的記憶體配置工具（圖一）之間存在設計上的良窳差異。

	GCC	VC6	CB5
起始時刻	3:3:54	3:1:56	3:0:32
結束時刻	3:4:7	3:2:39	3:1:1
耗時（秒）	13	43	29

圖五\ 在不同的編譯器上配置一千萬個 16bytes 區塊，所耗費的時間。**請注意，這些數值取決於環境的因素很大，本身沒有意義**，有意義的是三者之間的比較。

不僅配置需要花時間，釋放也需要花大量時間，因爲區塊必須再次被納入 system heap 的管理體系內。爲了驗證釋放花費的時間，我們可以一個大型 array 將上述所有指標記錄起來，然後再以迴圈全部釋放掉。值得一提的是，不同的編譯器所允許的 array 大小有極大的差異：

```
const int total = 7000000;
C1* ptrArray[total];
```

對 GCC 而言，7,000,000 是可以忍受的數字，超過之後效率陡降。對 VC6 和 CB5 而言，超過 200,000 個就會發生執行期異常。這其實是因爲 VC6 和 CB5 爲 local stack 預設保留的大小有限。只要透過聯結器選項，重新設定一個較大值，就可以解決問題[6]。

C++ 標準程式庫的配置器（allocator）

圖一的第四組工具，是所謂的配置器（allocator）。這是附隨 C++ 標準程式庫而來的東西，用來處理各種標準容器的記憶體需求。每個標準容器都可以在定義時

[5] printLocalTime()是我自己寫的一個函式，運用 C 標準函式的 time()，localtime()及標準結構 struct tm，獲得當時當地時間。

[6] 這些測試都在 Windows console 模式下進行（本文所有程式皆如此）。Windows console 模式做出來的同樣是 Win32 程式，同樣是 PE 可執行檔格式。如果 VC6, CB5 在其 IDE（整合開發環境）中有不同的表現，那倒是令人咄咄稱奇了。

刻接受一個配置器，此後如有需要（例如元素插入或刪除），就向該配置器索求或釋還記憶體。例如：

```
#ifdef _MSC_VER
  vector<int,allocator<int> > v;
#endif
#ifdef __BORLANDC__
  vector<int,allocator<int> > v;
#endif
#ifdef __GNUC__
  vector<int,alloc> v;      // 注意配置器的名稱及其參數（無參數）
#endif

v.push_back(10);  // 元素記憶體由指定之配置器負責配置
```

C++標準規格書明定，標準程式庫應該供應一個標準配置器，並且應該名爲 allocator。但從上段程式碼顯而易見，至少 GCC 就沒有遵循這個規定（其實並非如此，詳見稍後對 GCC 的深入討論）。C++ 標準規格書也規定，如果使用者不指定配置器，就採用預設配置器，因此我們也可以（而且通常是）這麼寫：

```
vector<int> v;
```

究竟配置器做些什麼事情呢？C++ 標準規格書中只規範了配置器的介面，實際作爲全由實作廠商自行決定。以下介紹主流的三個編譯器所附的配置器內容。

VC6 的 PJ STL 配置器

VC6 所附的標準程式庫（中的 STL）由 P.J. Plauger 發展，我常簡稱其爲 PJ STL。下面是其實作內容摘錄（都在 <xmemory> 檔案中）：

```
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); }
};
```

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))); }
```

這是一份可讀性極差的程式碼（PJ STL 處處洋溢類似風格，因此我非常不建議各位閱讀 PJ STL），但從中我們還是可以觀察到，此配置器的 allocate()和 deallocate()其實就是呼叫 operator new()和 operator delete()。

CB5 的 RW STL 配置器

CB5 所附的標準程式庫（中的 STL）是由 Rouge Wave 公司發展，我常簡稱其爲 RW STL。下面是其內容摘錄（都在 <memory.stl> 檔案中）：

```
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);
  }
  ...
};
```

這份源碼的可讀性就好些了，我們可以清楚看到，此配置器的 allocate()和 deallocate()其實就是呼叫 operator new()和 operator delete()。

GCC 的 SGI STL 配置器

GCC 所附的標準程式庫（中的 STL）是由 Sillcon Graphics 公司發展，我常簡稱其爲 SGI STL。下面是其內容摘錄（都在 <defalloc.h> 檔案中）：

```
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); }
};
```

其中用到的兩個全域函式定義如下：

```
template <class T>
inline T* allocate(ptrdiff_t size, T*) {
    set_new_handler(0);     // 稍後詳述
    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);
}
```

這份源碼乾淨易讀。我們可以清楚看到，此配置器的 allocate()和 deallocate()其實就是呼叫 operator new()和 operator delete()。

慢！先前我不是才說過，SGI STL 的配置器有點不同於標準規格嗎？上面這傢伙看起來和先前所使用的介面：

```
#ifdef __GNUC__
  vector<int,alloc> v;
```

```
#endif
```

徹頭徹尾不像，倒是很像 PJ STL 和 RW STL 所提供者（但也只是像而已，注意其參數列並不相同）。這是怎麼回事？

是這樣的，SGI STL 定義的這個符合 C++ 標準規格的配置器，只是聊備一格，你可以用它，但 SGI 不建議你用，而且 SGI STL 本身從未使用過它。主要原因是 SGI 早就開發出一套更具威力、帶有 memory pool 功能的配置器，因此 SGI STL 的所有容器都以後者爲預設配置器。

接下來我要探討的，便是這個功能強大的 SGI STL 預設配置器。

記憶池的設計哲學

以下我稱呼這個 SGI STL 預設配置器爲 **alloc**，這也是它在源碼中的命名。

先前我們已經看到，爲記憶體配置帶來額外開銷的，就是用以記錄區塊大小的那片「小甜餅」空間，以及因「區塊的設置日益雜亂」而造成「尋找適當新區塊」時的速度日益遲緩。如果我們能夠讓每個區塊大小都相同，上述兩個問題便可一次解決。這是犧牲「萬用大小」的彈性來換取時間和空間優勢。但程式對區塊的需求不可能永遠固定大小，因此這是一種在特定情況下才能發揮功效的設計。

如果我們一開始先向系統要求一大塊記憶體（**memory pool**，記憶池，日後並可擴充），並將它們視爲（切割爲）特定大小的區塊，以某種結構（通常是 list，我們稱其爲 **free list**）維護之，一旦使用者需要這種特定大小的區塊，就不再透過系統工具（圖一）獲得，而是透過這個 memory pool 及其介面來取得。區塊釋放動作也不透過系統工具（圖一）完成，而是透過 memory pool 的介面回收到池內供下次再用。此時由於整個 memory pool 就是一個 free list，所以在這種單一 free list 的情況下，兩者常被混稱。

alloc 有兩個級次。第二級（較高級）有能力供應 16 種特定大小的區塊，以 8 爲單位，分別是 8,16,24,32,...,128。超過 128 之後額外開銷小於 4/128=3.1%，一般認爲足堪忍受，這時候就改用第一級（較初級），以 `malloc()` 配置區塊。這是在功

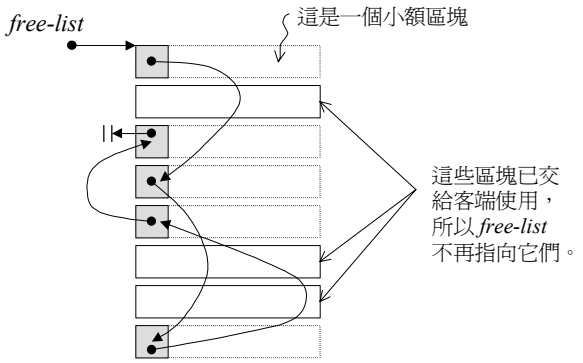
能和實現之間取其平衡的一種考量，因爲我們不可能爲任意大小的區塊都維護 free lists。爲了維護（供應）16 種特定大小的區塊，**alloc** 必須維護 16 個 free lists，此時 memory pool 和 free lists 涇渭分明，觀念上不能再有所含混；兩者的總量稱爲 **allocHeap**。以下程式片段（取自 SGI STL 源碼）揭示 16 個 free-lists 的寫法：

```
template <bool threads, int inst>
class __default_alloc_template { // 這是 SGI STL 的配置器類別名稱
    ...
  union obj {
        union obj* free_list_link;
        char client_data[1];    // The client sees this.
  };
  static obj * volatile free_list[16];
};

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj * volatile
__default_alloc_template<threads, inst> ::free_list[16]
 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

// 上述 16 其實是個變數，爲讓程式易讀，我直接代以 16。
// 若要擴展提供更多種特定大小的區塊，只需修改相關變數即可。
```

這裡運用了一個重要技巧：**union**。要知道，memory pool 的目標之一在節省記憶體，而如今爲了維護 free list，每個區塊卻需附加一個所謂的 "next" 指標做爲 list 連接之用。東西還沒到手籌碼倒先流失了不少，這不是偷雞不著蝕把米嗎？幸運的是由於 C++ 在型態檢驗上沒有那麼強硬，我們得以利用 union 讓同一塊東西有不同的解釋：當它在 free list 手上時，被視爲 "next" 指標；當它被配給索求者時，被視爲某區塊的起始位址，如圖六。

有了這些基本設施後，我們看看 **alloc** 如何在 memory pool 和 free lists 之間進行管理。下面是其管理哲學：每當客端索求 n-bytes 區塊，**alloc** 首先判斷 n 是否大於 128，若是則改以系統工具（圖一）處理，若否，將 n 調整至 8 的倍數，觀察相應的 free list 有無自由區塊可用。若有，調整指標指向下一區塊，並因而挪出一個區塊交給索求者。若相應的 free list 中無自由區塊可用，就將 pool 內的記憶體「搬來」（其實只是指標設定）置於相應的 free list 中，再由後者依上述方式滿足客端索求。如果 pool 之中的備用記憶體不足以滿足一個區塊，首先將它「撥入」（其實只是指

標設定）適當的 free list 內，然後向系統配置 40 個[7]區塊的**連續空間**給 pool，然後「搬移」（其實只是指標設定）其中 20 個放入 free lists（並設好區塊間的連結），另 20 個留置池中備用。

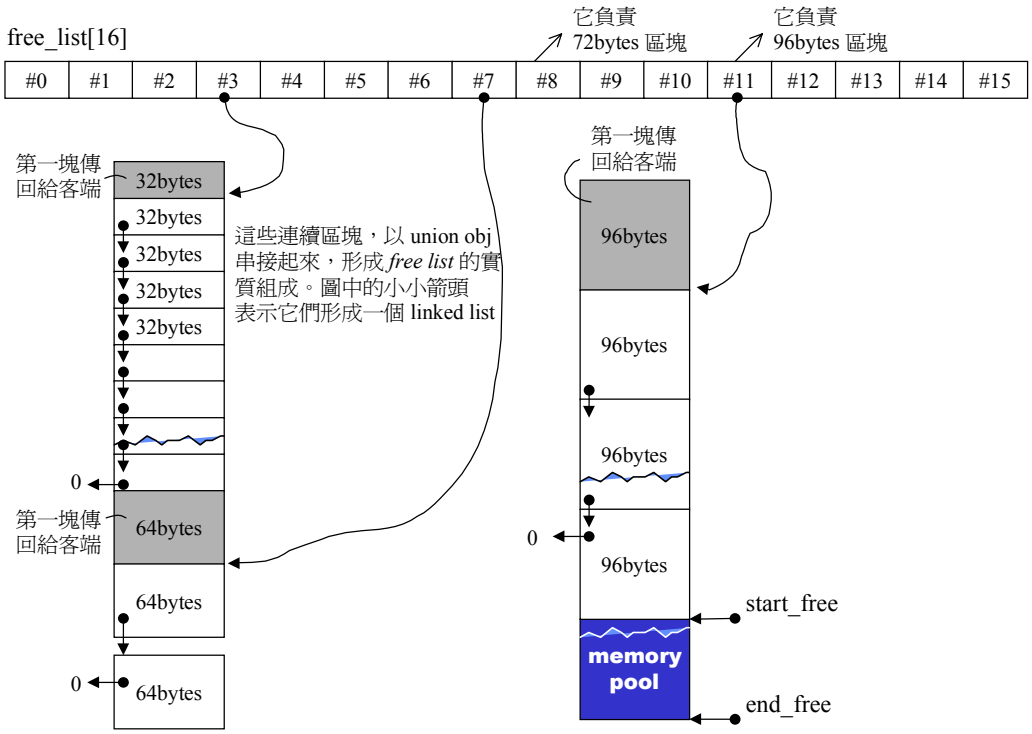


圖六\ free list 結構示意。此圖爲一整塊連續空間，若以 bytes 爲單位，應由第一區塊最左向右出發，尾端折回第二區塊，依此類推。每一灰色小塊代表 4bytes，當區塊尚在 free list 掌控下時（虛線框），可以它爲指標（邏輯而言 free list 看不到虛線框那一部分）；當區塊被配給客端後，客端即獲得了一個區塊（實線框）的起始位址 — 此時無法從中獲知區塊大小，但客端應該知道自己當初索求多大區塊。如果因爲 `new` 一個物件而導致這裡的區塊配置，那麼雖然客端也許未直接知道物件大小（亦即區塊大小），但因物件操作永遠不可能逸出物件大小之外，所以絕無越界之虞☺

圖七是一個實際操作過程（稍後以實例驗證）。客端首先配置 32bytes，由於一開始什麼都是空的，所以 **alloc** 向系統配置 32*20*2 給 pool，並從其中撥出 20 個給 list #3，其中一個又撥給客端，留下 19 個。接下來客端配置 64bytes，由於 list #7 爲空，所以 **alloc** 從 pool 之中將剛才剩餘的 32*20/64=10 個區塊撥給 list #7，其中一個並撥給客端，留下 9 個。接下來客端再配置 96bytes，由於 list #11 爲空，而且 pool 之內空空如也，因此 **alloc** 向系統配置 96*20*2（再加上一些餘裕，見註解 7）給 pool，

[7] SGI STL 的實際作法是隨著 allocHeap 的大小而有一些成長，實作手法是：`(allocHeapSize)>>4`，之後再上調至 8 的邊界。但我無法理解爲什麼要這麼做，可能是考慮到「heap 愈大代表區塊用量愈多愈頻繁，因此每次對記憶池的挹注量也應該大些」。稍後對 pool 的解釋中我將不提這塊小額空間。

並從其中撥出 20 個給 list #11，其中一個又撥給客端，留下 19 個。稍後我有一個測試程式，用以監視 memory pool 和 free lists 的動態，我們可以從那兒獲得更多寶貴資訊。



圖七\ memory pool 和 16 個 free lists。注意，一旦區塊的配置、釋放動作頻繁發生後，畫面上 free list 內的線頭（代表指標）可能雜亂不堪。

區塊回收

當客端釋放區塊，**alloc** 將它回收納入相應的 free list 中。實際上只是指標的設定，所以速度很快。一旦區塊的配置、釋放動作頻繁發生後，圖七畫面上的線頭（代表指標）可能雜亂不堪，此時任何一個 free list 都無法確認它手上的自由區塊是否連續，抑或中間有些配置出去的「漏洞」（如圖六）。因此任何時候（尤其記憶體山窮水盡時）它都不能夠將手上那些自由區塊整合爲較大區塊供應外界索求。

請注意，**alloc** 從不把區塊還給 system heap。因此它的 free lists 經過不斷的來回配

置/釋放之後，可能擁有的區塊個數難以估量。這其中並沒有 memory leak（記憶體洩漏）問題，因爲所有區塊都在掌控之中，無漏網之魚。

山窮水盡疑無路

一旦記憶體用罄，該怎麼辦？這個問題可分兩層次探討，層次一是 **alloc** 如何應對？層次二是萬一連 **alloc** 都雙手一攤說抱歉，你（客端）又如何應對？

如果客端要求的區塊大於 128，責任已不在 **alloc** 身上，問題將跳至層次二（稍後討論）。但如果客端要求的區塊小於 128（例如 96），而此時相應的 free list 內已無自由區塊，pool 的餘量又不足 96（例如 32），此時應該將 32 撥給相應的 free list 然後向 system heap 求援 96*20*2（再加上一些餘裕，見註解 7）。如果此時的 system heap 不能夠滿足需求，**alloc** 應該反求諸己，看看還可以從哪兒「擠」出 96bytes 滿足客端需求。最簡單的想法是尋找更大的自由區塊（104 或 112 或 120 或 128）。只要找到一個，便可撥給 pool，再由 pool 撥到 list #11 中。請注意，不能將較大區塊直接撥給客端，那會造成浪費。如果先撥給 pool，pool 的餘量便總是能做充份的運用。

以上便是 **alloc** 的行爲。

柳暗花明又一村

(1) 萬一已無任何較大自由區塊可用（例如客端要求的是 128bytes），可試著將對 system heap 的需求量減半，再試試能否配置成功。不成功，再減半，依此類推。

(2) 如果減至最後 system heap 連一個區塊的大小都發不出來，**alloc** 還可以繼續反求諸己，嘗試從所有 free lists 中找出夠大的連續空間（比較前後兩區塊的起始位址間距是否等於「區塊大小」即可知道是否連續），以此撥給 pool，再循規處理。

(3) 如果無論如何努力，memory pool 再也擠不出空間來了，這時候應該設法將控制權拉回我們手上，不要輕易任由異常狀況 *bad_alloc* 發生。我們可以提供一個自

己的專門處理常式（常被稱爲 NewHandler），做任何自己所能處理的應變措施[8]。

以上前兩種作法，SGI STL **alloc** 並未加以實現。因此 **alloc** 最大可能的浪費量有多少呢？難以估量，因爲經過頻繁的配置/釋放之後，每個 free list 最終可能維護的自由區塊無法估量。其中處於連續狀態的區塊可能不在少數，原本應該拿來再利用。此外，造成 system heap 供應不足的那最後一擊，需求量可能很大（最大可能是 128*20*2 + 餘裕空間；註解 7 告訴我們餘裕空間隨著 allocHeapSize 增加）。如果只因這樣便判定記憶體不足，未免也太冤枉。

這都是 **alloc** 值得改善之處。但以上第三種作法 **alloc** 是支援的，稍後介紹。

實際驗證

我寫了一個測試程式，用來觀察 **alloc** 的動態狀態。這個程式一點學問都沒有，純粹只是將 **alloc** 的某些變數(如圖七所示)列印出來。這些變數原本都被設計爲 private 資料，因此必須先將它們全改爲 public（請記得備份☺）。

我準備了 17 個不同大小（8,16,24,…,128,160bytes）的 classes，代號分別爲 0~16，然後以一個迴圈不斷要求使用者輸入代號，用以 new 其相應物件。new 動作怎麼能夠連接至 **alloc** 呢？這是後頭「無痛運用」的主題，暫且不表。每次 new 出一個物件，就呼叫 poolListing() 將 **alloc** 的所有資訊列印出來。這個函式接受一個 output stream，所以我們呼叫它時可指定列印到螢幕或檔案。函式中用到的 MyAlloc 是程式內的一個全域配置器物件：

```
alloc MyAlloc;  // alloc 是定義於 SGI STL <stl_alloc.h> 中的一個型別

void poolListing(ostream& os)
{
  os << (void*)MyAlloc.start_free << ' '
     << (void*)MyAlloc.end_free << ' '
     << "poolSize:"  << MyAlloc.end_free - MyAlloc.start_free << ' '
     << "heapSize: " << MyAlloc.heap_size
     << endl;

  for(int i=0; i< MyAlloc.__NFREELISTS; ++i) {
```

[8] 請參考《*Effective C++*》2e, 條款 7：爲記憶體不足的狀況預做準備。

```
    __default_alloc_template<0,0>::obj* ptr = MyAlloc.free_list[i];
    int num = 0;

    while(ptr) {
       ++num;                          // counting
       ptr = ptr->free_list_link;      // next
    }

    os << '#' << i << ' '
       << '(' << (i+1)*8 << ")" << ' '
       << MyAlloc.free_list[i] << ' '
       << num << "\t\t";

    if ((i%2)==1)      // 一行列印兩個 free-list 資訊
       os << endl;
  }
}
```

下面是某次執行結果。雖然這份結果看似龐大太佔篇幅，但其中的狀態演變以及註釋，都足以讓你對 **alloc** 的實際運作有充份的體會，很有價值。

```
// 說明：一開始什麼都是 0
0x0 0x0 poolSize:0 heapSize: 0
#0 (8)       0x0 0          #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x0 0
#4 (40)      0x0 0          #5 (48)       0x0 0
#6 (56)      0x0 0          #7 (64)       0x0 0
#8 (72)      0x0 0          #9 (80)       0x0 0
#10 (88)     0x0 0          #11 (96)      0x0 0
#12 (104)    0x0 0          #13 (112)     0x0 0
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 32，pool 獲得 32*20*2=1280 的挹注。
// 其中 20 個區塊給 list#3（並撥一個給客端），餘 640 備用。
select (0~16, -1 to end): 3
0x25c1918 0x25c1b98 poolSize:640 heapSize: 1280
#0 (8)       0x0 0          #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x0 0
#6 (56)      0x0 0          #7 (64)       0x0 0
#8 (72)      0x0 0          #9 (80)       0x0 0
#10 (88)     0x0 0          #11 (96)      0x0 0
#12 (104)    0x0 0          #13 (112)     0x0 0
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 64，取 pool 劃分爲 640/64=10 個區塊，
// 撥一個給客端，留 9 個於 list#7。
select (0~16, -1 to end): 7
0x25c1b98 0x25c1b98 poolSize:0 heapSize: 1280
#0 (8)       0x0 0          #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x25c16b8 19
```

```
#4 (40)      0x0 0          #5 (48)        0x0 0
#6 (56)      0x0 0          #7 (64)        0x25c1958 9
#8 (72)      0x0 0          #9 (80)        0x0 0
#10 (88)     0x0 0          #11 (96)       0x0 0
#12 (104)    0x0 0          #13 (112)      0x0 0
#14 (120)    0x0 0          #15 (128)      0x0 0
// 說明：以下配置 96，pool 獲得 96*20*2+RoundUp(1280>>4)的挹注。
// 其中 20 個區塊給 list#11（並撥一個給客端），餘 2000 備用。
select (0~16, -1 to end): 11
0x25c2320 0x25c2af0 poolSize:2000 heapSize: 5200
#0 (8)       0x0 0          #1 (16)        0x0 0
#2 (24)      0x0 0          #3 (32)        0x25c16b8 19
#4 (40)      0x0 0          #5 (48)        0x0 0
#6 (56)      0x0 0          #7 (64)        0x25c1958 9
#8 (72)      0x0 0          #9 (80)        0x0 0
#10 (88)     0x0 0          #11 (96)       0x25c1c00 19
#12 (104)    0x0 0          #13 (112)      0x0 0
#14 (120)    0x0 0          #15 (128)      0x0 0
// 說明：以下配置 88，取 pool 劃分爲 20 個區塊，撥一個給客端，
// 留 19 個於 list#10。Pool 剩餘 2000-88*20=240.
select (0~16, -1 to end): 10
0x25c2a00 0x25c2af0 poolSize:240 heapSize: 5200
#0 (8)       0x0 0          #1 (16)        0x0 0
#2 (24)      0x0 0          #3 (32)        0x25c16b8 19
#4 (40)      0x0 0          #5 (48)        0x0 0
#6 (56)      0x0 0          #7 (64)        0x25c1958 9
#8 (72)      0x0 0          #9 (80)        0x0 0
#10 (88)     0x25c2378 19   #11 (96)       0x25c1c00 19
#12 (104)    0x0 0          #13 (112)      0x0 0
#14 (120)    0x0 0          #15 (128)      0x0 0
// 說明：以下連續三次配置 88。直接由 list 取出撥給客端。
// 連續三次後，得以下結果。
select (0~16, -1 to end): 10
0x25c2a00 0x25c2af0 poolSize:240 heapSize: 5200
#0 (8)       0x0 0          #1 (16)        0x0 0
#2 (24)      0x0 0          #3 (32)        0x25c16b8 19
#4 (40)      0x0 0          #5 (48)        0x0 0
#6 (56)      0x0 0          #7 (64)        0x25c1958 9
#8 (72)      0x0 0          #9 (80)        0x0 0
#10 (88)     0x25c2480 16   #11 (96)       0x25c1c00 19
#12 (104)    0x0 0          #13 (112)      0x0 0
#14 (120)    0x0 0          #15 (128)      0x0 0
// 說明：以下配置 8，取 pool 劃分爲 20 個區塊，撥一個給客端，
// 留 19 個於 list#0。Pool 剩餘 240-8*20=80.
select (0~16, -1 to end): 0
0x25c2aa0 0x25c2af0 poolSize:80 heapSize: 5200
#0 (8)       0x25c2a08 19   #1 (16)        0x0 0
#2 (24)      0x0 0          #3 (32)        0x25c16b8 19
#4 (40)      0x0 0          #5 (48)        0x0 0
```

```
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x0 0
#10 (88)     0x25c2480 16   #11 (96)      0x25c1c00 19
#12 (104)    0x0 0          #13 (112)     0x0 0
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 104，list#12 之中無可用區塊，pool 餘量又不足供應一個，
// 於是先將 pool 餘額撥給 list#9，然後獲得 104*20*2+RoundUp(5200>>4)
// 的挹注，其中 20 個撥給 list#12（並又撥一個給客端），餘 2408 備用。
select (0~16, -1 to end): 12
0x25c3318 0x25c3c80 poolSize:2408 heapSize: 9688
#0 (8)       0x25c2a08 19   #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x0 0
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x25c2aa0 1
#10 (88)     0x25c2480 16   #11 (96)      0x25c1c00 19
#12 (104)    0x25c2b60 19   #13 (112)     0x0 0
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 112，取 pool 劃分爲 20 個區塊，撥一個給客端，
// 留 19 個於 list#13。Pool 剩餘 2408-112*20=168.
select (0~16, -1 to end): 13
0x25c3bd8 0x25c3c80 poolSize:168 heapSize: 9688
#0 (8)       0x25c2a08 19   #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x0 0
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x25c2aa0 1
#10 (88)     0x25c2480 16   #11 (96)      0x25c1c00 19
#12 (104)    0x25c2b60 19   #13 (112)     0x25c3388 19
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 48，取 pool 劃分爲 3 個區塊，撥一個給客端，
// 留 2 個於 list#5。Pool 剩餘 168-48*3=24.
select (0~16, -1 to end): 5
0x25c3c68 0x25c3c80 poolSize:24 heapSize: 9688
#0 (8)       0x25c2a08 19   #1 (16)       0x0 0
#2 (24)      0x0 0          #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x25c3c08 2
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x25c2aa0 1
#10 (88)     0x25c2480 16   #11 (96)      0x25c1c00 19
#12 (104)    0x25c2b60 19   #13 (112)     0x25c3388 19
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 72，list#8 中無可用區塊，pool 餘量又不足供應一個，
// 於是先將 pool 餘額撥給 list#9，然後爭取 72*20*2+RoundUp(9688>>4)
// 的挹注，但此要求已超越 system heap 大小（我將它設定爲 10000，後述），
// 因此記憶體不足，於是反求諸己取 88 區塊回填 pool，再以之當做 72 區塊給客端，
// 餘 8 備用。
select (0~16, -1 to end): 8
0x25c2ae8 0x25c2af0 poolSize:8 heapSize: 9688
```

```
#0 (8)       0x25c2a08 19   #1 (16)       0x0 0
#2 (24)      0x25c3c68 1    #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x25c3c08 2
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x0 0
#10 (88)     0x25c2480 16   #11 (96)      0x25c1c00 19
#12 (104)    0x25c2b60 19   #13 (112)     0x25c3388 19
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 72，list#8 中無可用區塊，pool 餘量又不足供應一個，
// 於是先將 pool 餘額撥給 list#0，然後爭取 72*20*2+RoundUp(9688>>4)
// 的挹注，但此要求已超越 system heap 大小（我將它設定爲 10000，後述），
// 因此記憶體不足，於是反求諸己取 88 區塊回填 pool，再以之當做 72 區塊給客端，
// 餘 8 備用。
select (0~16, -1 to end): 8
0x25c24c8 0x25c24d8 poolSize:16 heapSize: 9688
#0 (8)       0x25c2ae8 20   #1 (16)       0x0 0
#2 (24)      0x25c3c68 1    #3 (32)       0x25c16b8 19
#4 (40)      0x0 0          #5 (48)       0x25c3c08 2
#6 (56)      0x0 0          #7 (64)       0x25c1958 9
#8 (72)      0x0 0          #9 (80)       0x0 0
#10 (88)     0x25c24d8 15   #11 (96)      0x25c1c00 19
#12 (104)    0x25c2b60 19   #13 (112)     0x25c3388 19
#14 (120)    0x0 0          #15 (128)     0x0 0
// 說明：以下配置 120，list#14 中無可用區塊，pool 餘量又不足供應一個，
// 於是先將 pool 餘額撥給 list#0，然後爭取 120*20*2+RoundUp(9688>>4)
// 的挹注，但此要求已超越 system heap 大小（我將它設定爲 10000，後述），
// 因此記憶體不足，但反求諸己後仍得不到自由區塊，於是失敗。
// 檢討：此時其實尚有可用記憶體，包括 system heap 還有 10000-9688=312，
// 各個 free lists 內亦可能有些連續自由區塊。
select (0~16, -1 to end): 14
out-of-memory -- jjhou simulation.
```

效率和快的用語

圖八是在三種不同的編譯器中分別以 new 和 allocator 配置一千萬個 16bytes 區塊，所耗費的時間。其中 new 的結果與圖五雖有大異，但因配置速度本就和執行當時的 system heap 狀態有關，所以也還可以理解。我們的關切重點是，當改以配置器負責區塊配置工作時，GCC 由於運用了 memory pool 手法（當區塊小於 128bytes），速度有飛昇現象；VC6 和 CB5 的速度則較之以 new 完成者有著嚴重的落後[9]。

```
    int size = sizeof(C1);  // 16 bytes
    int i;

    printLocalTime();

#ifdef _MSC_VER
    allocator<int> MyAlloc;
    for(i=0; i<10000000; ++i)  MyAlloc.allocate(size, (int*)0);
#endif
#ifdef __BORLANDC__
    allocator<int> MyAlloc;
    for(i=0; i<10000000; ++i)  MyAlloc.allocate(size);
#endif
#ifdef __GNUC__
    for(i=0; i<10000000; ++i)  alloc::allocate(size);
#endif

    printLocalTime();
```

[9] 按理說，透過配置器，也只不過比直接使用 new 多了一層函式呼叫和回返動作。10,000,000 次呼叫和回返似乎不應該造成速度延遲那麼多。至今我尚未能夠理解造成巨大延遲的原因。當然，我必得再強調一次，每次執行時的系統當下狀態，都可能影響配置速度。另請注意，如果區塊大於 128bytes，GCC 配置器會交給 malloc()處理，等同於 ::operator new()，效果和直接使用 new 差不多（只多一點點時間，可理解爲用於額外的函式呼叫和回返動作上）。

	GCC (new)	GCC (allocator)	VC6 (new)	VC6 (allocator)	CB5 (new)	CB5 (allocator)
起始時刻	21:29:25	21:30:52	21:31:26	21:32:57	21:37:06	21:38:44
結束時刻	21:29:52	21:30:55	21:32:13	21:36:03	21:37:12	21:41:24
耗時（秒）	27	3	47	186	6	160

圖八\ 在不同的編譯器上配置一千萬個 16bytes 區塊，所耗費的時間。請注意，**這些數值取決於環境的因素很大，本身沒有意義**，有意義的是彼此之間的比較。

如何設定 system heap 大小（用以模擬記憶體不足）

先前的測試程式曾經觀察到記憶體不足時 **alloc** 的 memory-pool/free-lists 的變化。欲在程式中耗盡 system heap，其實並不困難：

```
int* p = new int[100000000]; // 配置一億個 int，亦即四億個 bytes。
```

還沒用光嗎（可能有虛擬記憶體）？再加一個 0 試試☺

但我希望我的測試過程不是 try-and-error 的暴力法，我希望一切都在我的掌控之下。爲此我修改了 SGI STL 源碼，在原本「直接配置 system heap 以求挹注 pool」之處（《STL 源碼剖析》p67 中央）：

```
        start_free = (char *)malloc(bytes_to_get);
```

加上一個判斷式：

```
    if (heap_size + bytes_to_get > 10000)
        start_free = 0;     // jjhou's out-of-memory simulation
    else
        start_free = (char *)malloc(bytes_to_get);
```

其中 `heap_size` 是從 system heap 配置而來的記憶體的累計總量（原本就已經維護著這麼一個變數）。如果將它加上此次配置量後，超過 system heap 大小（本處模擬爲 10,000 bytes），就令配置動作失敗。 這個小小計倆純粹是爲了測試並觀察 SGI STL 配置器面臨記憶體不足威脅時的反應，我們沒有必要爲 PJ STL 和 RW STL 配置器也思考這個問題。

如何註冊「記憶體不足處理常式」（New Handler）

C++ 提供一個全域函式 `set_new_handler()`，允許我們註冊自己的 NewHandler（記憶體不足處理常式）。下面是其形式：

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

根據這個型式，我們可以在應用程式的任何地點這麼做：

```
// 下面這個函式準備用來在 operator new 無法配置足夠記憶體時被喚起
void jjNewHandler() {
   // 這裡可以嘗試許多努力，設法擠出一些記憶體，例如釋放不用的空間等等。
   // 本例只是單純地秀出訊息而後結束程式
   cout << "out of memory -- jjhou Simulation" << endl;
   abort();
}
set_new_handler(jjNewHandler);
```

由於歷史因素，SGI STL 配置器並未支援這個標準規格，而是以 C 型式模擬之，因此我們必須遵循其規格，如下設定 NewHandler：

```
malloc_alloc::set_malloc_handler(jjNewHandler);
```

其中 `malloc_alloc` 是第一級配置器 — 當 **alloc** 遇到 128bytes 以上的區塊需求，就會把執行權交到這個第一級配置器來，它提供了如上的 static 函式，允許我們設定自己的 NewHandler。

SGI STL 的記憶池實作細節

SGI STL 的 **alloc** 設計於 `<stl_alloc.h>` 內。本文先前對 **alloc** 動作細節的描述，其實便已經描述了其完整的演算法，剩下只是編程工作而已。詳細源碼及說明可參考《STL 源碼剖析》第二章，此處不多贅述。

無痛運用的包裝技巧

有了這麼棒的一個 memory pool 配置器，雖然它可以無縫接合於標準容器，但我們該如何讓它也無縫接合於 C++ 物件生成動作呢？從圖二可以看出，`new` 運算式被編譯器分解爲 (1) 呼叫 `::operator new()` 以配置記憶體，(2) 呼叫建構式以

建構物件。因此如果我們能夠重載 operator new，便可將配置器無縫銜接到 new 運算式中。當然，這麼做的同時我們也必須重載 operator delete。

重載 ::operator new() 和 ::operator delete()

C++ 極爲強大的一個特性就是：允許你對幾乎所有運算子做重載動作，重新定義其行爲。下面是對全域性 operator new 和 operator delete 的重載動作：

```
void* operator new(size_t size) {
   return alloc::allocate(size);  // get from free-list
}

void operator delete(void* p, size_t size) {
   alloc::deallocate(p, size);    // return to free-list
}
```

現在，客端再次於 GCC 中執行原先曾經做過的 10,000,000 個物件（大小爲 16bytes）的動態生成測試。程式寫法完全不變，速度卻提昇了許多（僅 4 秒），證明的確透過精巧的 **alloc** 完成了配置任務：

```
printLocalTime();
for(int i=0; i<10000000; ++i)  new C1;    // sizeof(C1): 16
printLocalTime();
```

但是這種手法還需更廣泛的測試，因爲有時候會在 delete 動作中出現 *STATUS_ACCESS_VIOLATION* 異常。

針對特定類別重載其 operator new() 和 operator delete()

重載全域性運算子的好處是釜底抽薪，缺點則是過於勇猛。是的，一旦你重載了全域運算子，那麼除非你在某個 class 中又再重載之，否則該運算子的任何運用場合裡用上的便都是那「經過重載的全域運算子」。就記憶體配置而言這本是好事，但若你所要求的大量區塊都大於 128bytes，造成每次配置都經由重載後的 ::operator new()轉呼叫配置器的 allocate()，而後由於區塊大於 128 之故又再轉給第一級配置器喚起 malloc()進行實際配置動作，這比起直接呼叫「未經重載」的 ::operator new()實在是繞了一大圈；次數一多便嚴重影響效率。

爲此，我們或許並不希望太過於釜底抽薪，我們或許希望只針對某些「物件體積小於等於 128bytes」的 classes 進行改裝。作法如下：

```
class MyClass     // 如果體積小於等於 128bytes
{
public:
  static void* operator new(size_t size) {
     return alloc::allocate(size);  // get from free-list
  }
  static void operator delete(void* p, size_t size) {
     alloc::deallocate(p, size);    // return to free-list
  }
...      // 其他成員函式，及資料成員
};
```

由於 operator new()在物件建構之前被喚起，operator delete()在物件解構之後被喚起，所以這兩者以成員函式出現的重載運算子都必須是 static。正因它們一定必須是 static，編譯器允許你不必爲它們加上關鍵字 static。

operator new 和 operator delete 可被繼承

如果你的 classes 不單單只是一或二個，而是一個家族繼承體系，那麼一一爲每個 classes 重載 operator new()/operator delete()（如上）實在太過麻煩又易出錯。幸運的是這兩個重載運算子可被繼承。因此我們可以設計一個虛擬基礎類別，專只用來提供對此二個運算子的重載能力：

```
class BaseAlloc
{
public:
  virtual ~BaseAlloc() { } // 基礎類別總是應該提供 virtual dtor.

  static void* operator new(size_t size) {
     return alloc::allocate(size);  // get from free-list
  }
  static void operator delete(void* p, size_t size) {
     alloc::deallocate(p, size);    // return to free-list
  }
}; // 此類別之物件大小爲 4
```

並令其他所有 classes 都衍生自此基礎類別，如此便唾手可得我們所希望的功能：

```
// 第一案。本案已經測試（作法是看看其中的虛擬函式在多型狀態下是否運行正常）
class MyB : public BaseAlloc    { ... };
class MyD : public MyB          { ... };
```

採用多重繼承也可得到相同利益：

```
// 第二案。本案已經測試（作法是看看其中的虛擬函式在多型狀態下是否運行正常）
```

```
class MyB_Root { ... };
class MyD_MI : public MyB_Root, public BaseAlloc  { ... };
```

上述兩種作法各有缺點：每一個物件的體積都增加了！第一案使每個物件增加 4bytes，第二案增加的大小更是驚人，數量視編譯器而異（多重繼承的編譯器底層作法向來沒有太多統一，詳見《深度探索 C++ 物件模型》第三章）。我們必須時刻記住，只有在物件體積小於或等於 128bytes（可調整）時，這裡的一切努力才有意義。

由於各有優劣，所以應該視你的實際專案決定採用哪一個方案。

一個帶有 memory pool 系統的可攜性配置器

將 SGI STL **alloc** 改裝爲一個可移植的配置器，並不是一件太困難的事。當然，你必須對 `<stl_alloc.h>` 中的各種環境組態（configuration）都有相當程度的理解。至於 SGI STL 程式碼本身，可讀性和移植性都非常高，可參考《STL 源碼剖析》。

致謝

本文草稿獲得孟岩先生的許多寶貴意見，特此致謝。

更多討論

你可以從以下書籍或網站獲得更多與本文主題相關的討論。這些資訊可以彌補因文章篇幅限制而帶來的不足，或帶給你更多視野。

- 《C++ Primer 3/e》15.8 節，介紹 operator new/delete 的重載作法。
- 《Effective C++ 2e》條款 5~10，介紹記憶體不足時的應對策略及相關主題。
- 《STL 源碼剖析》第 2 章，介紹 SGI STL 配置器的設計和源碼。
- 《C++ 標準程式庫》第 15 章，介紹配置器的構想和概念。
- 《Small Memory Software》，全書介紹記憶體受限系統（例如嵌入式系統或掌上型系統）中的軟體求存之道。
- Doug Lea 個人網頁 http://gee.cs.oswego.edu/dl/，其中有 DL Malloc 源碼可自由下載。