侯捷观点

池内春秋
Memory Pool的设计哲学和无痛运用（下）

* 本文介绍了Memory Pool的历史、设计思想及其在C++中的实现
* 读者基础：有一定程度的C++编程经验
* 本文适用工具：GNU C++编译器
* 本文关于SGI STL之剖析，部分已载于《STL源码剖析》第二章；崭新内容包括SGI STL区块释除（归还）动作分析、缺点与补强之道、无痛应用、三种编译器之区块配置效能比较。

撰文／侯捷

关键词 memory pool free list free block allocator heap client

记忆池的设计哲学

以下我称呼这个SGI STL预设配置器为alloc，这也是它在源码中的命名。

先前我们已经看到，为内存配置带来额外开销的，就是用以记录区块大小的那片“小甜饼”空间，以及因“区块的设置日益杂乱”而造成“寻找适当新区块”时的速度日益迟缓。如果我[illegible]小都相同，上述两个问题便可一次解[illegible]大小”的弹性来换取时间和空间优势。**但程序对区块的需求不可能永远固定大小，因此这是一种在特定情况下才能发挥功效的设计。**

如果我们一开始先向系统要求一大块内存（memory pool，记忆池，日后可扩充），并将它们视为（切割为）特定大小的区块，以某种结构（通常是list，我们称其为free list）维护之，一旦使用者需要这种特定大小的区块，就不再通过系统工具获得，而是通过这个memory pool及其接口来取得。区块释放动作也不通过系统工具完成，而是通过memory pool的接口回收到池内供下次再用。此时由于整个memory pool就是一个free list，所以在这种单一free list的情况下，两者常被混称。

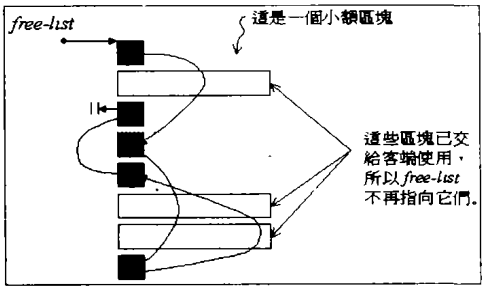
alloc有两个级次。第二级（较高级）有能力供应16种特定大小的区块，以8为单位，分别是8、16、24、32、……、128。超过128之后额外开销小于4／128=3.1%，一般认为足堪忍受，这时候就改用第一级（较初级），以malloc()配置区块。这是在功能和实现之间取其平衡的一种考量，因为我们不可能为任意大小的区块都维护free lists。为了维护（供应）16种特定大小的区块，alloc必须维护16个free lists，此时memory pool和free lists泾渭分明，观念上不能再有所含混；两者的总量称为allocHeap。以下程序片段（取自SGI STL源码）揭示16个free-lists的写法：

```
template <bool threads, int inst>
class __default_alloc_template {  // 这是 SGI STL的配置器类别名称
    ...
  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其实是个变量，为让程序易读，我直接代以16。
// 若要扩展提供更多种特定大小的区块，只需修改相关变量即可。
```

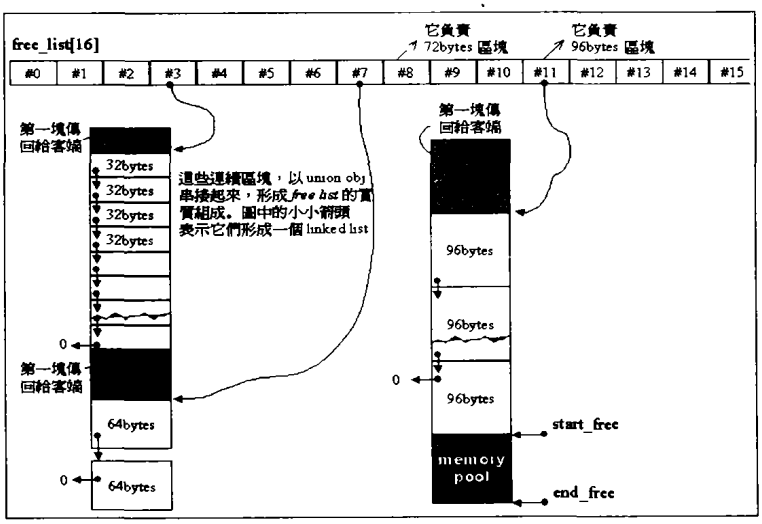
这里运用了一个重要技巧：union。要知道，memory pool的目标之一在节省内存，而如今为了维护free list，每个区块却需附加一个所谓的“next”指针做为list连接之用。东西还没到手筹码倒先流失了不少，这不是偷鸡不着蚀把米吗？幸运的是由于C++在类型检验上没有那么强硬，我们得以利用union让同一块东西有不同的解释：当它在free list手上时，被视为"next"指针；当它被配给索求者时，被视为某区块的起始地址，如图六。

有了这些基本设施后，我们看看alloc如何在memory pool

和free lists之间进行管理。下面是其管理哲学：每当使用者索求n-bytes区块，alloc首先判断n是否大于128，若是则改以系统工具处理；若否，将n调整至8的倍数，观察相应的free list有无自由区块可用。若有，调整指针指向下一区块，并因而挪出一个区块交给索求者。若相应的free list中无自由区块可用，就将pool内的内存“搬来”（其实只是指针设定）置于相应的free list中，再由后者依上述方式满足使用者索求。如果pool之中的备用内存不足以满足一个区块，首先将它“拨入”（其实只是指针设定）适当的free list内，然后向系统配置40个'区块的连续空间给pool，然后“搬移”（其实只是指针设定）其中20个放入free lists（并设好区块间的连结），另20个留置池中备用。



图六 free list结构示意。此图为一整块连续空间，若以bytes为单位，应由第一区块最左向右出发，尾端折回第二区块，依此类推。每一灰色小块代表4bytes、当区块尚在free list掌控下时（虚线框），可以它为指针（逻辑而言free list看不到虚线框那一部分）；当区块被配给使用者后，使用者即获得了一个区块（实线框）的起始地址——此时无法从中获知区块大小，但使用者应该知道自己当初索求多大区块。如果因为new一个对象而导致这里的区块配置，那么虽然使用者也许未直接知道对象大小（亦即区块大小），但因对象操作永远不可能逸出对象大小之外，所以绝无越界之虞：）



图七 memory pool和16个free lists。注意，一旦区块的配置、释放动作频繁发生后，画面上free list内的线头（代表指针）可能杂乱不堪。

图七是一个实际操作过程（稍后以实例验证）。使用者首先配置32bytes，由于一开始什么都是空的，所以alloc向系统配置32*20*2给pool，并从其中拨出20个给list #3，其中一个又拨给使用者，留下19个。接下来使用者配置64bytes，由于list #7为空，所以alloc从pool之中将刚才剩余的32*20/64=10个区块拨给list #7，其中一个并拨给使用者，留下9个。接下来使用者再配置96bytes，由于list #11为空，而且pool之内空空如也，因此alloc向系统配置96*20*2（再加上一些余裕，见注1）给pool，并从其中拨出20个给list #11，其中一个又拨给使用者，留下19个。稍后我有一个测试程序，用以监视memory pool和free lists的动态，我们可以从那儿获得更多宝贵信息。

区块回收

当使用者释放区块时，alloc将它回收纳入相应的free list中。这实际上只是指针的设定，所以速度很快。一旦区块的配置、释放动作频繁发生后，图七画面上的线头（代表指针）可能杂乱不堪，此时任何一个free list都无法确认它手上的自由区块是否连续，抑或中间有些配置出去的“漏洞”（如图六）。因此任何时候（尤其内存山穷水尽时）它都不能够将手上那些自由区块整合为较大区块供应外界索求。

请注意，alloc从不把区块还给system heap。因此它的free lists经过不断的来回配置／释放之后，可能拥有的区块个数难以估量。这其中并没有memory leak（内存泄漏）问题，因为所有区块都在掌控之中，无漏网之鱼。

山穷水尽疑无路

一旦内存用罄，该怎么办？这个问题可分两层次探讨，层次一是alloc如何应对？层次二是万一连alloc都双手一摊说抱歉，你（使用者）又如何应对？

如果使用者要求的区块大于128，责任已不在alloc身上，问题将跳至层次二（稍后讨论）。但如果使用者要求的区块小于128（例如96），而此时相应的free list内已无自由区块，pool的余量又不足96（例如32），此时应该将32拨给相应的free list然后向system heap求援96*20*2（再加上一些余裕，见注1）。如果此时的system heap不能够满足需求，alloc应该反求诸己，看看还可以从哪儿“挤”出96bytes满足使用者需求。最简单的想法是寻找更大的自由区块（104或112或120或128）。只要找到一个，便可拨给pool，再由pool拨到list #11中。请注意，不能将较大区块直接拨给使用者，那会造成浪费。如果先拨给pool，pool的余量便总是能做充份的运用。

以上便是alloc的行为。

柳暗花明又一村

(1)万一已无任何较大自由区块可用(例如使用者要求的是128bytes),可试着将对system heap的需求量减半,再试试能否配置成功。不成功,再减半,依此类推。

(2)如果减至最后system heap连一个区块的大小都发不出来,alloc还可以继续反求诸己,尝试从所有free lists中找出够大的连续空间(比较前后两区块的起始地址间距是否等于"区块大小"即可知道是否连续),以此拨给pool,再循规处理。

(3)如果无论如何努力,memory pool再也挤不出空间来了,这时候应该设法将控制权拉回我们手上,不要轻易任由异常状况bad_alloc发生。我们可以提供一个自己的专门处理程序(常被称为NewHandler)、做任何自己所能处理的应变措施[2]。

以上前两种作法,SGI STL alloc并未加以实现。因此alloc最大可能的浪费量有多少呢?难以估量,因为经过频繁的配置/释放之后,每个free list最终可能维护的自由区块无法估量。其中处于连续状态的区块可能不在少数,原本应该拿来再利用。此外,造成system heap供应不足的那最后一击,需求量可能很大(最大可能是128*20*2 + 余裕空间;注1告诉我们余裕空间随着allocHeapSize增加)。如果只因这样便判定内存不足,未免也太冤枉。

这都是alloc值得改善之处。但以上第三种作法alloc是支持的,稍后介绍。

实际验证

我写了一个测试程序,用来观察alloc的动态状态。这个程序一点学问都没有,纯粹只是将alloc的某些变量(如图七所示)打印出来。这些变量原本都被设计为private数据,因此必须先将它们全改为public(请记得备份;))。

我准备了17个不同大小(8、16、24、……、128、160bytes)的classes,代号分别为0~16,然后以一个循环不断要求使用者输入代号,用以new其相应对象。new动作怎么能够连接至alloc呢?这是后头"无痛运用"的主题,暂且不表。每次new出一个对象,就调用poolListing()将alloc的所有信息打印出来。这个函数接受一个output stream,所以我们调用它时可指定打印到屏幕或档案。函数中用到的MyAlloc是程序内的一个全局配置器对象:

```
    alloc MyAlloc;  // alloc 是定义于SGI STL <stl_alloc.h> 中
的一个型别

    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) {
         __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)              // 一行打印两个free-list信息
            os << endl;
      }
    }
```

下面是某次执行结果。虽然这份结果看似庞大太占篇幅,但其中的状态演变以及注释,都足以让你对alloc的实际运作有充分的体会,很有价值。

```
// 说明: 一开始什么都是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

// 说明: 以下配置32,pool获得32*20*2=1280的把注。
// 其中20个区块给list#3(并拨一个给使用者),余640备用。
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

// 说明: 以下配置64,取pool划分为640/64=10个区块,
// 拨一个给使用者,留9个于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

// 说明：以下配置96，pool获得96•20•2+RoundUp(1280>>4) 的挹注。
// 其中20个区块给list#11（并拨一个给使用者），余2000备用。

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

// 说明：以下配置88，取pool划分为20个区块，拨一个给使用者，
// 留19个于list#10。Pool剩余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

// 说明：以下连续三次配置88。直接由list取出拨给使用者。
// 连续三次后，得以下结果。

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

// 说明：以下配置8，取pool划分为20个区块，拨一个给使用者，
// 留19个于list#0。Pool剩余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

// 说明：以下配置104，list#12之中无可用区块，pool余额又
//不足供应一个，于是先将pool余额拨给list#9，[illegible]
//104•20•2+RoundUp(5200>>4) 的挹注，其中20个[illegible]#12
//（并又拨一个给使用者），余2408备用。

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

// 说明：以下配置112，取pool划分为20个区块，拨一个给使用者，
// 留19个于list#13。Pool剩余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

// 说明：以下配置48，取pool划分为3个区块，拨一个给使用者，
// 留2个于list#5。Pool剩余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

```
// 说明: 以下配置72，list#8中无可用区块，pool余量又不足
// 供应一个，于是先将pool余额拨给list#9，然后争取
// 72*20*2+RoundUp(9688>>4) 的挹注，但此要求已超越
// system heap大小（我将它设定为10000，后述），因此内存
// 不足，于是反求诸己取88区块回填pool，再以之当做72区块
// 给使用者，余8备用。
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

// 说明: 以下配置72，list#8中无可用区块，pool余量又不足
// 供应一个，于是先将pool余额拨给list#0，然后争取
// 72*20*2+RoundUp(9688>>4) 的挹注，但此要求已超越
// system heap大小（我将它设定为10000，后述），因此内存不足，
// 于是反求诸己取88区块回填pool，再以之当做72区块给使用者，
// 余8备用。
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

// 说明: 以下配置120，list#14中无可用区块，pool余量又不足
// 供应一个，于是先将pool余额拨给list#0，然后争取
// 120*20*2+RoundUp(9688>>4) 的挹注，但此要求已超越
// system heap大小（我将它设定为10000，后述），因此内存不足，
// 但反求诸己后仍得不到自由区块，于是失败。
// 检讨: 此时其实尚有可用内存，包括system heap还有
// 10000-9688=312，各个free lists内亦可能有些连续自由区块。
select (0~16, -1 to end): 14
out-of-memory -- jjhou simulation.
```

效率加快的明证

图八是在三种不同的编译器中分别以new和allocator配置一千万个16bytes区块，所耗费的时间。其中new的结果与图五虽有大异，但因配置速度本就和执行当时的system heap状态有关，所以也还可以理解。我们的关切重点是，当改以配置器负责区块配置工作时，GCC由于运用了memory pool手法（当区块小于128bytes），速度有飞升现象；VC6和CB5的速度则较之以new完成者有着严重的落后[3]。

	GCC (new)	GCC (allocator)	VC6 (new)	VC6 (allocator)	CB5 (new)	CB5 (allocator)
起始时刻	21:29:25	21:30:52	21:31 26	21:32.57	21:37:06	21.38:44
结束时刻	21 29 52	21 30 55	21:32:13	21.36.03	21:37 12	21:41.24
耗时（秒）	27	3	47	186	6	160

图八 在不同的编译器上配置一千万个16bytes区块，所耗费的时间。请注意，这些数值取决于环境的因素很大，本身没有意义，有意义的是彼此之间的比较。

```
    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();
```

如何设定system heap大小（用以模拟内存不足）

先前的测试程序曾经观察到内存不足时alloc的memory-pool/free-lists的变化。欲在程序中耗尽system heap，其实并不困难：

```
int* p = new int[100000000]; // 配置一亿个int，亦即四亿个bytes。
```

还没用光吗（可能有虚拟内存）？再加一个0试试：）

但我希望我的测试过程不是try-and-error的暴力法，我希望一切都在我的掌控之下。为此我修改了SGI STL源码，在原本“直接配置system heap以求挹注pool”之处（《STL源码剖析》p67中央）：

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

加上一个判断式：

```
      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是从system heap配置而来的内存的累

计总量（原本就已经维护着这么一个变量）。如果将它加上此次配置量后，超过system heap大小（本处仿真为10,000 bytes），就令配置动作失败。这个小小计俩纯粹是为了测试并观察SGI STL配置器面临内存不足威胁时的反应，我们没有必要为PJ STL和RW STL配置器也思考这个问题。

如何注册“内存不足处理程序”（New Handler）

C++ 提供一个全局函数set_new_handler()，允许我们注册自己的NewHandler（内存不足处理程序）。下面是其形式：

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

根据这个形式，我们可以在应用程序的任何地点这么做：

```
// 下面这个函数准备用来在 operator new 无法配置足够内存时被调用
void jjNewHandler() {
   // 这里可以尝试许多努力，设法挤出一些内存，例如释放不用的空间等等。
   // 本例只是单纯地秀出讯息而后结束程序
   cout << "out of memory -- jjhou Simulation" << endl;
   abort();
}
set_new_handler(jjNewHandler);
```

由于历史因素，SGI STL配置器并未支持这个标准规格，而是以C型式模拟之，因此我们必须遵循其规格，如下设定NewHandler：

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

其中malloc_alloc是第一级配置器——当alloc遇到128bytes以上的区块需求，就会把执行权交到这个第一级配置器来，它提供了如上的static函数，允许我们设定自己的NewHandler。

SGI STL的记忆池实现细节

SGI STL的alloc设计于<stl_alloc.h>内。本文先前对alloc动作细节的描述，其实便已经描述了其完整的算法，剩下只是编程工作而已。详细源码及说明可参考《STL源码剖析》第二章，此处不多赘述。

无痛运用的包装技巧

有了这么棒的一个memory pool配置器，虽然它可以无缝接合于标准容器，但我们该如何让它也无缝接合于C++ 对象生成动作呢？

从上一期中的图二可以看出，new表达式被编译器分解为(1)调用::operator new() 以配置内存，(2)调用构造子以构造对象。因此如果我们能够重载operator new，便可将配置器无缝衔接到new表达式中。当然，这么做的同时我们也必须重载operator delete。

重载 ::operator new() 和 ::operator delete()

C++极为强大的一个特性就是：允许你对几乎所有运算子做重载动作，重新定义其行为。下面是对全局性operator new和operator delete的重载动作：

```
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
}
```

现在，使用者再次于GCC中执行原先曾经做过的10,000,000个对象（大小为16bytes）的动态生成测试。程序写法完全不变，速度却提升了许多（仅4秒），证明的确通过精巧的alloc完成了配置任务：

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

但是这种手法还需更广泛的测试，因为有时候会在delete动作中出现STATUS_ACCESS_VIOLATION 异常。

针对特定类别重载其operator new() 和 operator delete()

重载全局性运算子的好处是釜底抽薪，缺点则是过于勇猛。是的，一旦你重载了全局运算子，那么除非你在某个class中又再重载之，否则该运算子的任何运用场合里用上的便都是那“经过重载的全局运算子”。就内存配置而言这本是好事，但若你所要求的大量区块都大于128bytes，造成每次配置都经由重载后的::operator new()转调用配置器的allocate()，而后由于区块大于128之故又再转给第一级配置器调用malloc()进行实际配置动作，这比起直接调用“未经重载”的::operator new()实在是绕了一大圈；次数一多便严重影响效率。

为此，我们或许并不希望太过于釜底抽薪，我们或许希望只针对某些“对象体积小于等于128bytes”的classes进行改装。作法如下：

```
class MyClass      // 如果体积小于等于 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
  }
...                  // 其它成员函数，及数据成员
};
```

由于operator new()在对象构造之前被调用，operator delete()在对象析构之后被调用，所以这两者以成员函数出现的重载运算子都必须是static。正因它们一定必须是static，编译器允许你不必为它们加上关键词static。

operator new和operator delete可被继承

如果你的classes不单单只是一或二个，而是一个家族继承体系，那么一一为每个classes重载operator new()/operator delete()（如上）实在太过麻烦又易出错。幸运的是这两个重载运算子可被继承。因此我们可以设计一个虚拟基础类别，专只用来提供对此二个运算子的重载能力：

```
class BaseAlloc
{
public:
  virtual ~BaseAlloc() { }  // 基础类别总是应该提供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
  }
}; // 此类别之对象大小为4
```

并令其它所有classes都衍生自此基础类别，如此便唾手可得我们所希望的功能：

```
// 第一案。本案已经测试（作法是看看其中的虚拟函数在多型状态下是否运行正常）
class MyB : public BaseAlloc     { ... };
class MyD : public MyB           { ... };
```

采用多重继承也可得到相同利益：

```
// 第二案。本案已经测试（作法是看看其中的虚拟函数在多型状态下是否运行正常）
class MyB_Root { ... };
class MyD_MI : public MyB_Root, public BaseAlloc  { ... };
```

上述两种作法各有缺点：每一个对象的体积都增加了！第一案使每个对象增加4 bytes，第二案增加的大小更是惊人，数量视编译器而异（多重继承的编译器底层作法向来没有太多统一，详见《深度探索C++对象模型》第三章）。我们必须时刻记住，只有在对象体积小于或等于128bytes（可调整）时，这里的一切努力才有意义。

由于各有优劣，所以应该视你的实际项目决定采用哪一个方案。

一个带有memory pool系统的可移植性配置器

将SGI STL alloc改装为一个可移植的配置器，并不是一件太困难的事。当然，你必须对<stl_alloc.h>中的各种环境配置（configuration）都有相当程度的理解。至于SGI STL程序代码本身，可读性和移植性都非常高，可参考《STL源码剖析》。

致谢

本文草稿获得孟岩先生的许多宝贵意见，特此致谢。

更多信息

你可以从以下书籍或网站获得更多与本文主题相关的讨论。这些信息可以弥补因文章篇幅限制而带来的不足，或带给你更多视野。

- 《C++ Primer 3/e》15.8节，介绍operator new/delete的重载作法。
- 《Effective C++ 2e》条款5~10，介绍内存不足时的应对策略及相关主题。
- 《STL源码剖析》第2章，介绍SGI STL配置器的设计和源码。
- 《C++ 标准链接库》第15章，介绍配置器的构想和概念。
- 《Small Memory Software》，全书介绍内存受限系统（例如嵌入式系统或掌上型系统）中的软件求存之道。
- Doug Lea个人网页http://gee.cs.oswego.edu/dl/ ，其中有 DL Malloc源码可自由下载。

注：

1 SGI STL的实际作法是随着allocHeap的大小而有一些成长，实作手法是：(allocHeapSize)>>4，之后再上调至8的边界。但我无法理解为什么要这么做，可能是考虑到“heap愈大代表区块用量愈多愈频繁，因此每次对记忆池的挹注量也应该大些”。稍后对pool的解释中我将不提这块小额空间。

2 请参考《Effective C++》2e，条款7：为内存不足的状况预做准备。

3 按理说，透过配置器，也只不过比直接使用new多了一层函数调用和返回动作。10,000,000次调用和返回似乎不应该造成速度延迟那么多。至今我尚未能够理解造成巨大延迟的原因。当然，我必得再强调一次，每次执行时的系统当下状态，都可能影响配置速度。另请注意，如果区块大于128bytes，GCC配置器会交给malloc()处理，等同于:operator new()，效果和直接使用 new差不多（只多一点点时间，可理解为用于额外的函数调用和返回动作上）。