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区块卸除（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。

关键词：memory pool free list free block allocator heap client

池畔春秋

——Memory Pool的设计哲学和无痛运用（上）

撰文／侯捷

为什么需要记忆池

内存曾经是兵家必争之地，曾经被喻为“CPU之外最宝贵的计算机硬件资源”。在那“640K天堑[1]”的远古年代里，程序员对内存缁铢必较的程度可能令生活于“虚拟内存”环境下的当今世代瞠目结舌，千禧年（Y2K）虫虫危机即肇因于当初过份樽节内存[2]。当时的人们（我也属其中之一）即便在config.sys中挥汗调校只省下区区数十个bytes，都会觉得欢欣鼓舞；如果有人能够运用int67h进入EMS内存或运用int21h进入XMS内存[3]，更可说是走路有风，呵水成冻。1991年微软发布的MS-DOS5.0，涵盖数个寻址相关技术（UMB: Upper Memory Block，HMA: High Memory Area），大幅度提升MS-DOS的寻址能力，当时被誉为“突破性的进展”。

曾几何时，当虚拟内存操作系统（如Windows、OS/2、Linux）走进群众，苦乐俱往矣。非人道的痛苦折磨被迅速遗忘，缁铢必较的轶趣成了白头宫女话天宝当年的回忆。我们不再被程序代码大小所限，也不再被数据量所限。所有内存不足的问题只要“加一条256M内存”就获得解决。从这个角度看，程序员的生活幸福美满。

当温饱获得解决，人们要求精致。软件开始往两个方向发展：一是更快，一是更小。系统级软件或特殊应用或数据量极大的软件，要求运行极快；掌上系统或嵌入式系统则因先天硬件环境的限制而必须体积极小。于是内存问题又再度浮上台面。

不论是速度问题或空间问题，都肇因于编译器给的那些个弹性极大的内存配置工具带来了一些额外开销（overhead）。当额外开销的比率超过你的容忍限度，你免不了要冲冠一怒寻求突破。最简单而效果良好的一种技术就是memory pool（记忆池）。在正式介绍memory pool技术之前，我要先带你彻底了解C++编译器的内存配置策略。

C++ 平台提供的内存配置工具

在C++平台上，你可以透过图一所列的四种方式动态索求及释放内存。其中第二组调用第三组，第三组功能及行为等同于第一组。第四组的“功能”相当于第一或第三组，其最终配置动作亦需仰赖第一或第三组完成，但我们可在第四组内部设计出复杂精巧的memory pool机制。

配置	释放	归属	可重载否	标记
malloc()	free()	C标准函式	不可	(1)
new	delete	C++ 表达式	不可	(2)
::operator new()	::operator delete()	C++ 运算子	可	(3)
alloc::allocate()	alloc::deallocate()	C++ 标准链接库之内存配置器	可自由设计并装载于任何容器身上提供服务	(4)

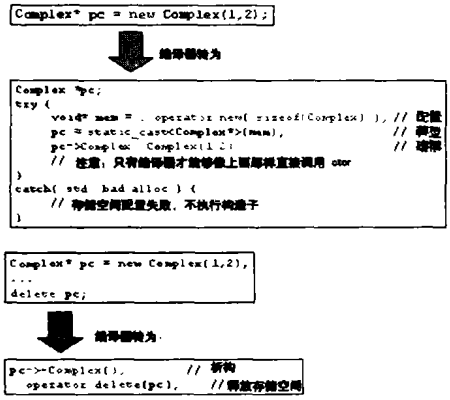
图一：C++语言及链接库供应的四种动态内存配置及释放方式。表中之alloc代表配置器allocator，其名称虽有正式规范，但不同的实现品可能有不同命名（后述）。

下面是这四种内存配置工具的运用示范：

```
// 四种配置方式
void* p1 = malloc(512);
free(p1);

CRect* pr = new CRect;
delete pr;

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

// 以下是 C++ 标准链接库中的配置器。其接口虽有标准规格，但各厂商多未遵守。
// 以致下面三种型式各异（稍后另有说明）。
#ifdef _MSC_VER
// 以下两者都是non-static，所以一定要藉由对象调用之
int* p3 = allocator<int>().allocate(512, (int*)0);
allocator<int>().deallocate(p3,512);
#endif
#ifdef __BORLANDC__
// 以下两者都是non-static，所以一定要藉由对象调用之
int* p3 = allocator<int>().allocate(512);
allocator<int>().deallocate(p3,512);
#endif
#ifdef __GNUC__
```

```
// 以下两者都是static，可透过全名调用之
void* p3 = alloc::allocate(512);
alloc::deallocate(p3,512);
#endif
```

如果你要的不只是单纯的内存配置，甚且希望直接在内存上建构对象（C++程序总是如此），那么你的唯一选择是第二组，因为第二组不但配置（释放）内存，还自动调用对象的构造子（析构子），如图二。



图二：new/delete表达式会调用new/delete运算子，及对应的构造子/析构子。

在这四组工具中，唯有第三组允许被重载、第四组允许由实现厂商（或客户端程序员自己）发挥。因此当我们打算设计一个更高效的内存配置器时，关键便着落在第三、四两组身上。

空间上的额外开销

C++平台（语言+链接库）所提供的内存配置工具中，前三组都会带来额外开销，这个额外开销被昵称为cookie（小甜饼），用来标示配置所得的区块大小，如图三。如果没有cookie的存在，一旦你想释放先前配得的内存，将手上的指针传给前三组的相应释放函数时，释放函数无法从指针身上看出区块大小，也就无法做出正确的回收（纳入system heap）动作。

速度上的额外开销

由于C++系统提供的配置工具（前三组）允许你任意指定区块大小，因此system heap一旦开始经历配置和回收，将出现群雄割据的杂乱局面（但都在掌控之中）。为了充分运用内存，系统配置工具必须有一套算法，用来遍历整个system heap，找出最适合需求的一块完整空间。System heap愈杂乱、区块大小愈是参差不等，找出一个适当（够大）区块的时间愈可能长久[4]。

图三：每个由system heap配置而来的内存区块，前端都带有一块"小甜饼"，大小通常为4 bytes，用来记录区块大小。

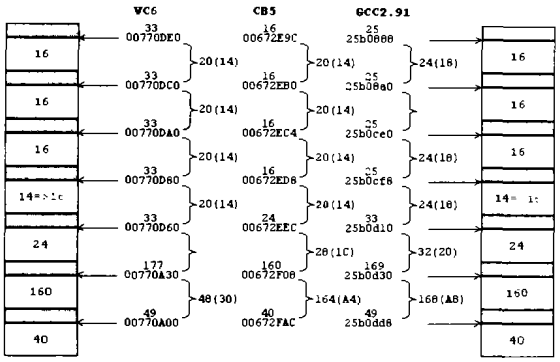
空间额外开销之明证

欲证明你所配置的每一个内存区块都带有"小甜饼"，很简单，只要观察动态配置得来的指针，看看其前方是否有所记录？记录值是否就是区块本身的大小？同时并观察数个连续动态配置的区块是否完全紧邻？亦或中间有些缝隙？缝隙的大小是否刚好4bytes?

下面这段程序代码刻意安排特定的对象大小，然后动态配置它们（并建构之），然后打印其前四个bytes，观察其内数值，印证是否和区块的大小相同。最后并观察这些区块的起始地址。由于只做简单观察之用，所以编程手法直观而不讲究。

```
// 动态配置内存
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

// 打印每个区块之前的四个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) );

// 打印每一区块的起始地址
printf("%p %p %p %p %p %p %p \n", pc1,pc12,pc13,pc2,pc3,pc4,pi);
```

图四是以上片段的某次执行结果。我们看到C++Builder5在这方面的表现最为"中规中矩"，完全符合我们的预期，不仅cookie记录"正确"，每个区块的距离也恰恰是区块大小加上4（cookie大小）。GCC产生的每个cookie记录的似乎都是区块实际大小加上1001（二进制），像是某种神秘标记；每个区块距离则是区块大小加上8，但也有例外（红色标示）。VC6产生的每个cookie以及区块的距离最难推断规律性，并且也有十分离奇的现象（红色标示）。这些离奇现象其实不难理解：我们在测试程序中连续配置数个区块，但malloc()或::operator new并不一定就在system heap中找到连续空间，这时候cookie之中有某种"神秘"记录是很容易想象的。就连C++Builder也不见得一直会有上述那么"中规中矩"的表现。



图四：观察连续配置而得的区块。图中灰色即为cookie，白色为配置而得的区块，其内数值（10进制）表示程序的需求大小。每个箭头旁的数值代表地址，两地址间的大括号旁的数值代表距离（bytes，10进制），旁边小括号内为16进制表示式。紧邻每个地址之上的数值为实际观察得到之cookie内容（10进制）。

速度额外开销之明证

很难像图四那么绝对地向大家证明速度上的额外开销，因为并不存在一个标准可供比较。但我们可以试着在不同编译器上配置大量区块，观察其速度表现：

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

上述程序片段配置了一千万个C1对象，动作前后分别打印出当时时间[5]。某次执行结果如图五。我们发现，不同编译器上的内存配置工具（图一）之间存在设计上的良莠差异。

	GCC	VC6	CB5
起始时刻	3:3:54	3:1:56	3:0:32
结束时刻	3:4:7	3:2:39	3:1:1
耗时（秒）	13	43	29

图五：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，有意义的是三者之间的比较。

不仅配置需要花时间，释放也需要花大量时间，因为区块必须再次被纳入system heap的管理体系内。为了验证释放花费的时间，我们可以一个大型array将上述所有指针记录起来，然后再以循环全部释放掉。值得一提的是，不同的编译器所允许的array大小有极大的差异：

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

对GCC而言，7,000,000是可以忍受的数字，超过之后效率陡降。对VC6和CB5而言，超过200,000个就会发生执行期异常。这其实是因为VC6和CB5为local stack预设保留的大小有限。只要透过联结器选项，重新设定一个较大值，就可以解决问题[6]。

C++ 标准链接库的配置器（allocator）

图一的第四组工具，是所谓的配置器（allocator）。这是附随C++标准链接库而来的东西，用来处理各种标准容器的内存需求。每个标准容器都可以在定义时刻接受一个配置器，此后如有需要（例如元素插入或删除），就向该配置器索求或释还内存。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;   // 注意配置器的名称及其参数（无参数）
#endif

v.push_back(10);  // 元素内存由指定之配置器负责配置
```

C++标准规格书明定，标准链接库应该提供一个标准配置器，并且应该名为allocator。但从上段程序代码显而易见，至少GCC就没有遵循这个规定（其实并非如此，详见稍后对GCC的深入讨论）。C++标准规格书也规定，如果使用者不指定配置器，就采用预设配置器，因此我们也可以（而且通常是）这么写：

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

究竟配置器做些什么事情呢？C++标准规格书中只规范了配置器的接口，实际作为全由实现厂商自行决定。以下介绍主流的三个编译器所附的配置器内容。

VC6的PJ STL配置器

VC6所附的标准链接库（中的STL）由P.J. Plauger开发，我常简称其为PJ STL。下面是其实现内容摘录（都在<xmemory> 文件中）：

```
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()定义如下：

```
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))); }
```

这是一份可读性极差的程序代码（PJ STL处处洋溢类似风格，因此我非常不建议各位阅读PJ STL），但从中我们还是可以观察到，此配置器的allocate()和deallocate()其实就是调用operator new()和operator delete()。

CB5的RW STL配置器

CB5所附的标准链接库（中的STL）是由Rouge Wave公司发展，我常简称其为RW STL。下面是其内容摘录（都在<memory.stl> 文件中）：

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

这份源码的可读性就好些了，我们可以清楚看到，此配置器的allocate()和deallocate()其实就是调用operator new()和operator delete()。

GCC的SGI STL配置器

GCC所附的标准链接库（中的STL）是由Sillcon Graphics公司开发，我常简称其为SGI STL。下面是其内容摘录（都在<defalloc.h> 文件中）：

```
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); }
};
```

其中用到的两个全局函数定义如下：

```
template <class T>
inline T* allocate(ptrdiff_t size, T*) {
    set_new_handler(0);  // 稍后详述
    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);
}
```

这份源码干净易读。我们可以清楚看到，此配置器的allocate()和deallocate()其实就是调用operator new()和operator delete()。

慢！先前我不是才说过，SGI STL的配置器有点不同于标准规格吗？上面这家伙看起来和先前所使用的接口：

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

彻头彻尾不像，倒是很像PJ STL和RW STL所提供者（但也只是像而已，注意其参数列并不相同）。这是怎么回事？

是这样的，SGI STL定义的这个符合C++标准规格的配置器，只是聊备一格，你可以用它，但SGI不建议你用，而且SGI STL本身从未使用过它。主要原因是SGI早就开发出一套更具威力、带有memory pool功能的配置器，因此SGI STL的所有容器都以后者为预设配置器。

在下个月的文章中，我要探讨的，便是这个功能强大的SGI STL预设配置器。

注 解

1.MS-DOS 5.0 以前，PC环境上只能开发640K以下的程序。640K内必须包含操作系统本身、应用程序代码本身、以及应用程序的数据量。MS DOS 5.0强化了640KB至1024KB之间（UMB）寻址空间的运用，以及1024K以上少量寻址空间（HMA）的运用。

2. 当时的程序员为节省内存用量，将年份19xx仅储存为xx，以至于时序进入公元2000之后无法正常进位。

3.EMS：Expanded Memory Spec.，XMS：eXtended Memory Spec.，两者都是内存扩展（延伸）规格。详见拙作《虚拟内存：观念、设计与实作，using EMS and XMS》，1991，旗标出版。

4. 感谢孟岩先生对于malloc()提供以下补充说明：Memory Pool主要是针对native API malloc或operator new不够高效而开发。这种情况在以前比较多见。后来很多人都开始针对这个问题进行深入研究。1986年起Doug Lea潜心研究malloc算法，逐渐发展出比较好的作法，被称为DL Malloc，目前Linux的glibc中的malloc算法就是直接来自Doug Lea，其它平台的malloc实作品多少也受到DL的影响。总的来说，如今的malloc比以前快很多，这可能是为什么PJ Plauger等直接使用operator new实作allocator的原因。Doug Lea主页：http://gee.cs.oswego.edu/dl/ ，其中可下载DL Malloc源码。

5.printLocalTime()是我自己写的一个函数，运用C标准函数的time()、localtime()及标准结构struct tm，获得当时当地时间。

6. 这些测试都在Windows console模式下进行（本文所有程序皆如此）。Windows console模式做出来的同样是Win32程序，同样是PE可执行文件格式。如果VC6和CB5在其IDE（整合开发环境）中有不同的表现，那倒是令人咄咄称奇了。