# 8

# 定制 new 和 delete

## Customizing new and delete

当计算环境（例如 Java 和 .NET）夸耀自己内置“垃圾回收能力”的当今，C++ 对内存管理的纯手工法也许看起来有点老气。但是许多苛刻的系统程序开发人员之所以选择 C++，就是因为它允许他们手工管理内存。这样的开发人员研究并学习他们的软件使用内存的行为特征，然后修改分配和归还工作，以求获得其所建置的系统的最佳效率（包括时间和空间）。

这样做的前提是，了解 C++ 内存管理例程的行为。这正是本章焦点。这场游戏的两个主角是分配例程和归还例程（allocation and deallocation routines，也就是 operator new 和 operator delete），配角是 new-handler，这是当 operator new 无法满足客户的内存需求时所调用的函数。

多线程环境下的内存管理，遭受单线程系统不曾有过的挑战。由于 heap 是一个可被改动的全局性资源，因此多线程系统充斥着发狂访问这一类资源的 race conditions（竞速状态）出现机会。本章多个条款提及使用可改动之 static 数据，这总是会令线程感知（thread-aware）程序员高度警戒如坐针毡。如果没有适当的同步控制（synchronization），一旦使用无锁（lock-free）算法或精心防止并发访问（concurrent access）时，调用内存例程可能很容易导致管理 heap 的数据结构内容败坏。我不想一再提醒你这些危险，我只打算在这里提一下，然后假设你会牢记在心。

另外要记住的是，operator new 和 operator delete 只适合用来分配单一对象。Arrays 所用的内存由 operator new[]分配出来，并由 operator delete[]归还（注意两个函数名称中的[]）。除非特别表示，我所写的每一件关于 operator new 和 operator delete 的事也都适用于 operator new[]和 operator delete[]。

最后请注意，STL 容器所使用的 heap 内存是由容器所拥有的分配器对象（allocator objects）管理，不是被 new 和 delete 直接管理。本章并不讨论 STL 分配器。

## 条款 49：了解 new-handler 的行为

Understand the behavior of the new-handler.

当 operator new 无法满足某一内存分配需求时，它会抛出异常。以前它会返回一个 null 指针，某些旧式编译器目前也还那么做。你还是可以取得旧行为（有那么几分像啦），但本条款最后才会进行这项讨论。

当 operator new 抛出异常以反映一个未获满足的内存需求之前，它会先调用一个客户指定的错误处理函数，一个所谓的 *new-handler*。（这其实并非全部事实。operator new 真正做的事情稍微更复杂些。详见条款 51。）为了指定这个“用以处理内存不足”的函数，客户必须调用 set_new_handler，那是声明于 <new>的一个标准程序库函数：

```
namespace std {
   typedef void (*new_handler)( );
   new_handler set_new_handler(new_handler p) throw();
}
```

如你所见，new_handler 是个 typedef，定义出一个指针指向函数，该函数没有参数也不返回任何东西。set_new_handler 则是“获得一个 new_handler 并返回一个 new_handler”的函数。set_new_handler 声明式尾端的 "throw()" 是一份异常明细，表示该函数不抛出任何异常——虽然事实更有趣些，详见条款 29。

set_new_handler 的参数是个指针，指向 operator new 无法分配足够内存时该被调用的函数。其返回值也是个指针，指向 set_new_handler 被调用前正在执行（但马上就要被替换）的那个 new-handler 函数。

你可以这样使用 set_new_handler：

```
//以下是当 operator new 无法分配足够内存时，该被调用的函数
void outOfMem()
{
   std::cerr << "Unable to satisfy request for memory\n";
   std::abort( );
}
```

```
int main()
{
   std::set_new_handler(outOfMem);
   int* pBigDataArray = new int[100000000L];
   ...
}
```

就本例而言，如果 operator new 无法为 100,000,000 个整数分配足够空间，outOfMem 会被调用，于是程序在发出一个信息之后夭折（abort）。（顺带一提，如果在写出错误信息至 cerr 过程期间必须动态分配内存，考虑会发生什么事……）

当 operator new 无法满足内存申请时，它会不断调用 new-handler 函数，直到找到足够内存。引起反复调用的代码显示于条款 51，这里的高级描述已足够获得一个结论，那就是一个设计良好的 new-handler 函数必须做以下事情：

- **让更多内存可被使用**。这便造成 operator new 内的下一次内存分配动作可能成功。实现此策略的一个做法是，程序一开始执行就分配一大块内存，而后当 new-handler 第一次被调用，将它们释还给程序使用。
- **安装另一个 new-handler**。如果目前这个 new-handler 无法取得更多可用内存，或许它知道另外哪个 new-handler 有此能力。果真如此，目前这个 new-handler 就可以安装另外那个 new-handler 以替换自己（只要调用 set_new_handler）。下次当 operator new 调用 new-handler，调用的将是最新安装的那个。（这个旋律的变奏之一是让 new-handler 修改自己的行为，于是当它下次被调用，就会做某些不同的事。为达此目的，做法之一是令 new-handler 修改“会影响 new-handler 行为”的 static 数据、namespace 数据或 global 数据。）
- **卸除 new-handler**，也就是将 null 指针传给 set_new_handler。一旦没有安装任何 new-handler，operator new 会在内存分配不成功时抛出异常。
- **抛出 bad_alloc（或派生自 bad_alloc）的异常**。这样的异常不会被 operator new 捕捉，因此会被传播到内存索求处。
- **不返回**，通常调用 abort 或 exit。

这些选择让你在实现 new-handler 函数时拥有很大弹性。

有时候你或许希望以不同的方式处理内存分配失败情况，你希望视被分配物属于哪个 class 而定：

```
class X {
   public:
   static void outOfMemory();
   ...
};
class Y {
   public:
   static void outOfMemory();
   ...
};
X * p1 = new X;    //如果分配不成功，
                   //调用 X::outOfMemory
Y * p2 = new Y;    //如果分配不成功，
                   //调用 Y::outOfMemory
```

C++ 并不支持 class 专属之 new-handlers，但其实也不需要。你可以自己实现出这种行为。只需令每一个 class 提供自己的 set_new_handler 和 operator new 即可。其中 set_new_handler 使客户得以指定 class 专属的 new-handler（就像标准的 set_new_handler 允许客户指定 global new-handler），至于 operator new 则确保在分配 class 对象内存的过程中以 class 专属之 new-handler 替换 global new-handler。

现在，假设你打算处理 Widget class 的内存分配失败情况。首先你必须登录“当 operator new 无法为一个 Widget 对象分配足够内存时”调用的函数，所以你需要声明一个类型为 new_handler 的 static 成员，用以指向 class Widget 的 new-handler。看起来像这样：

```
class Widget {
public:
   static std::new_handler set_new_handler(std::new_handler p) throw();
   static void* operator new(std::size_t size) throw(std::bad_alloc);
private:
   static std::new_handler currentHandler;
};
```

Static 成员必须在 class 定义式之外被定义（除非它们是 const 而且是整数型，见条款 2），所以需要这么写：

```
std::new_handler Widget::currentHandler = 0;
                                   //在 class 实现文件内初始化为 null
```

Widget 内的 set_new_handler 函数会将它获得的指针存储起来，然后返回先前（在此调用之前）存储的指针，这也正是标准版 set_new_handler 的作为：

```
std::new_handler Widget::set_new_handler(std::new_handler p) throw()
{
   std::new_handler oldHandler = currentHandler;
   currentHandler = p;
   return oldHandler;
}
```

最后，Widget 的 operator new 做以下事情：

1. 调用标准 set_new_handler，告知 Widget 的错误处理函数。这会将 Widget 的 new-handler 安装为 global new-handler。
2. 调用 global operator new，执行实际之内存分配。如果分配失败，global operator new 会调用 Widge 的 new-handler，因为那个函数才刚被安装为 global new-handler。如果 global operator new 最终无法分配足够内存，会抛出一个 bad_alloc 异常。在此情况下 Widget 的 operator new 必须恢复原本的 global new-handler，然后再传播该异常。为确保原本的 new-handler 总是能够被重新安装回去，Widget 将 global new-handler 视为资源并遵守条款 13 的忠告，运用资源管理对象（resource-managing objects）防止资源泄漏。
3. 如果 global operator new 能够分配足够一个 Widget 对象所用的内存，Widget 的 operator new 会返回一个指针，指向分配所得。Widget 析构函数会管理 global new-handler，它会自动将 Widget's operator new 被调用前的那个 global new-handler 恢复回来。

下面以 C++ 代码再阐述一次。我将从资源处理类（resource-handling class）开始，那里面只有基础性 RAII 操作，在构造过程中获得一笔资源，并在析构过程中释还（见条款 13）：

```
class NewHandlerHolder {
public:
   explicit NewHandlerHolder(std::new_handler nh)      //取得目前的
     : handler(nh) {}                                  //new-handler。
   ~NewHandlerHolder()                                 //释放它
   { std::set_new_handler(handler); }
private:
   std::new_handler handler;                           //记录下来.
   NewHandlerHolder(const NewHandlerHolder&);          //阻止 copying
   NewHandlerHolder&                                   //（见条款 14）
     operator=(const NewHandlerHolder&);
};
```

这就使得 Widget's operator new 的实现相当简单：

```
void* Widget::operator new(std::size_t size) throw(std::bad_alloc)
{
   NewHandlerHolder                                    //安装 Widget 的
     h(std::set_new_handler(currentHandler));          //new-handler.
   return ::operator new(size);                        //分配内存或抛出异常.
}                                                      //恢复 global new-handler.
```

Widget 的客户应该类似这样使用其 new-handling：

```
void outOfMem();                                       //函数声明。此函数在
                                                       //Widget 对象分配失败时被调用.

Widget::set_new_handler(outOfMem);                     //设定 outOfMem 为 Widget 的
                                                       // new-handling 函数.

Widget* pw1 = new Widget;                              //如果内存分配失败，
                                                       //调用 outOfMem.

std::string* ps = new std::string;                     //如果内存分配失败，
                                                       //调用 global new-handling 函数
                                                       //（如果有的话）.

Widget::set_new_handler(0);                            //设定 Widget 专属的
                                                       //new-handling 函数为 null.

Widget* pw2 = new Widget;                              //如果内存分配失败，
                                                       //立刻抛出异常.
                                                       //（class Widget 并没有专属的
                                                       //new-handling 函数）.
```

实现这一方案的代码并不因 class 的不同而不同，因此在它处加以复用是个合理的构想。一个简单的做法是建立起一个 "mixin" 风格的 base class，这种 base class 用来允许 derived classes 继承单一特定能力——在本例中是“设定 class 专属之 new-handler”的能力。然后将这个 base class 转换为 template，如此一来每个 derived class 将获得实体互异的 class data 复件。

这个设计的 base class 部分让 derived classes 继承它们所需的 set_new_handler 和 operator new，而 template 部分则确保每一个 derived class 获得一个实体互异的 currentHandler 成员变量。听起来似乎有点复杂，但代码非常近似前个版本。实际上，唯一真正意义上的不同是，它现在可被任何有所需要的 class 使用：

```
template<typename T>                    //"mixin" 风格的 base class，用以支持
class NewHandlerSupport {               //class 专属的 set_new_handler
public:
   static std::new_handler set_new_handler(std::new_handler p) throw();
   static void* operator new(std::size_t size) throw(std::bad_alloc);
   ...                                  //其他的 operator new 版本——见条款 52
private:
   static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
   std::new_handler oldHandler = currentHandler;
   currentHandler = p;
   return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size)
throw(std::bad_alloc)
{
   NewHandlerHolder h(std::set_new_handler(currentHandler));
   return ::operator new(size);
}

//以下将每一个 currentHandler 初始化为 null
template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;
```

有了这个 class template，为 Widget 添加 set_new_handler 支持能力就轻而易举了：只要令 Widget 继承自 NewHandlerSupport<Widget>就好，像下面这样。看起来似乎很奇妙，稍后我将更详细解释它的精确意义。

```
class Widget: public NewHandlerSupport<Widget> {
   ...                  //和先前一样，但不必声明
};                      //set_new_handler 或 operator new
```

这就是 Widget 为了提供“class 专属之 set_new_handler”所需做的全部动作。

但或许你还是对 Widget 继承 NewHandlerSupport<Widget>感到心慌意乱。果真如此，你的焦虑还可能因为注意到 NewHandlerSupport template 从未使用其类型参数 T 而更放大数倍。实际上 T 的确不需被使用。我们只是希望，继承自 NewHandlerSupport 的每一个 class，拥有实体互异的 NewHandlerSupport 复件（更明确地说是其 static 成员变量 currentHandler）。类型参数 T 只是用来区分不同的

derived class。Template 机制会自动为每一个 T（NewHandlerSupport 赖以具现化的根据）生成一份 currentHandler。

至于说到 Widget 继承自一个模板化的（templatized）base class，而后者又以 Widget 作为类型参数，如果你对此头昏眼花，不要觉得惭愧。每个人一开始都有那种反应。由于它被证明是一个有用的技术，因此甚至拥有自己的名称："怪异的循环模板模式"（*curiously recurring template pattern*; CRTP）。有些人认为这个名称给人的第一眼印象很不自然。嗯，确实如此。

我曾发表过一篇文章，建议给它一个比较好的名称，像是 Do It For Me，因为当 Widget 继承 NewHandlerSupport<Widget> 时它其实并不是说"我是 Widget，我要针对 Widget class 继承 NewHandlerSupport"。但是，哎，没人采用我建议的名称（甚至我自己也不），但如果你看到 CRTP 会联想到它说的是 "do it for me"，或许可以帮助你了解这一模板化继承（templatized inheritance）到底用意为何。

像 NewHandlerSupport 这样的 templates，使得"为任何 class 添加一个它们专属的 new-handler"成为易事。然而 "mixin" 风格的继承肯定导致多重继承的争议，而在开始那条路之前，你需要先阅读条款 40。

直至 1993 年，C++ 都还要求 operator new 必须在无法分配足够内存时返回 null。新一代的 operator new 则应该抛出 bad_alloc 异常，但很多 C++ 程序是在编译器开始支持新修规范前写出来的。C++ 标准委员会不想抛弃那些"侦测 null"的族群，于是提供另一形式的 operator new，负责供应传统的"分配失败便返回 null"行为。这个形式被称为 "nothrow" 形式——某种程度上是因为他们在 new 的使用场合用了 nothrow 对象（定义于头文件 <new>）：

```
class Widget { ... };
Widget* pw1 = new Widget;                       //如果分配失败，抛出 bad_alloc.
if (pw1 == 0) ...                               //这个测试一定失败.
Widget* pw2 = new (std::nothrow) Widget;        //如果分配 Widget 失败，返回 0.
if (pw2 == 0) ...                               //这个测试可能成功
```

Nothrow new 对异常的强制保证性并不高。要知道，表达式 "new (std::nothrow) Widget" 发生两件事，第一，nothrow 版的 operator new 被调用，用以分配足够内存给 Widget 对象。如果分配失败便返回 null 指针，一如文档所言。如果分配成功，接下来 Widget 构造函数会被调用，而在那一点上所有的筹码便都耗尽，因为

Widget 构造函数可以做它想做的任何事。它有可能又 new 一些内存，而没人可以强迫它再次使用 nothrow new。因此虽然 "new (std::nothrow) Widget" 调用的 operator new 并不抛掷异常，但 Widget 构造函数却可能会。如果它真那么做，该异常会一如往常地传播。需要结论吗？结论就是：使用 nothrow new 只能保证 operator new 不抛掷异常，不保证像 "new (std::nothrow) Widget" 这样的表达式绝不导致异常。因此你其实没有运用 nothrow new 的需要。

无论使用正常（会抛出异常）的 new，或是其多少有点发育不良的 nothrow 兄弟，重要的是你需要了解 new-handler 的行为，因为两种形式都使用它。

### 请记住

- set_new_handler 允许客户指定一个函数，在内存分配无法获得满足时被调用。
- Nothrow new 是一个颇为局限的工具，因为它只适用于内存分配；后继的构造函数调用还是可能抛出异常。

## 条款 50：了解 new 和 delete 的合理替换时机

**Understand when it makes sense to replace new and delete.**

让我们暂时回到根本原理。首先，怎么会有人想要替换编译器提供的 operator new 或 operator delete 呢？下面是三个最常见的理由：

- **用来检测运用上的错误。**如果将“new 所得内存”delete 掉却不幸失败，会导致内存泄漏（memory leaks）。如果在“new 所得内存”身上多次 delete 则会导致不确定行为。如果 operator new 持有一串动态分配所得地址，而 operator delete 将地址从中移走，倒是很容易检测出上述错误用法。此外各式各样的编程错误可能导致数据 "overruns"（写入点在分配区块尾端之后）或 "underruns"（写入点在分配区块起点之前）。如果我们自行定义一个 operator news，便可超额分配内存，以额外空间（位于客户所得区块之前或后）放置特定的 byte patterns（即签名，signatures）。operator deletes 便得以检查上述签名是否原封不动，若否就表示在分配区的某个生命时间点发生了 overrun 或 underrun，这时候 operator delete 可以志记（log）那个事实以及那个惹是生非的指针。

- **为了强化效能。**编译器所带的 operator new 和 operator delete 主要用于一般目的，它们不但可被长时间执行的程序（例如网页服务器，web servers）接受，也可被执行时间少于一秒的程序接受。它们必须处理一系列需求，包括大块内存、小块内存、大小混合型内存。它们必须接纳各种分配形态，范围从程序存活期间的少量区块动态分配，到大数量短命对象的持续分配和归还。它们必须考虑破碎问题（fragmentation），这最终会导致程序无法满足大区块内存要求，即使彼时有总量足够但分散为许多小区块的自由内存。

  现实存在这么些个对内存管理器的要求，因此编译器所带的 operatornews 和 operator deletes 采取中庸之道也就不令人惊讶了。它们的工作对每个人都是适度地好，但不对特定任何人有最佳表现。如果你对你的程序的动态内存运用型态有深刻的了解，通常可以发现，定制版之 operator new 和 operator delete 性能胜过缺省版本。说到胜过，我的意思是它们比较快，有时甚至快很多，而且它们需要的内存比较少，最高可省 50%。对某些（虽然不是所有）应用程序而言，将旧有的（编译器自带的）new 和 delete 替换为定制版本，是获得重大效能提升的办法之一。

- **为了收集使用上的统计数据。**在一头栽进定制型 news 和定制型 deletes 之前，理当先收集你的软件如何使用其动态内存。分配区块的大小分布如何？寿命分布如何？它们倾向于以 FIFO（先进先出）次序或 LIFO（后进先出）次序或随机次序来分配和归还？它们的运用型态是否随时间改变，也就是说你的软件在不同的执行阶段有不同的分配/归还形态吗？任何时刻所使用的最大动态分配量（高水位）是多少？自行定义 operator new 和 operator delete 使我们得以轻松收集到这些信息。

观念上，写一个定制型 operator new 十分简单。举个例子，下面是个快速发展得出的初阶段 global operator new，促进并协助检测 "overruns"（写入点在分配区块尾端之后）或 "underruns"（写入点在分配区块起点之前）。其中还存在不少小错误，稍后我会完善它。

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;
//这段代码还有若干小错误，详下。
void* operator new(std::size_t size) throw(std::bad_alloc)
{
   using namespace std;
   size_t realSize = size + 2 * sizeof(int);     //增加大小，使能够
                                                 //塞入两个 signatures.
   void* pMem = malloc(realSize);                //调用 malloc 取得内存.
   if (!pMem) throw bad_alloc();

   //将 signature 写入内存的最前段落和最后段落.
   *(static_cast<int*>(pMem)) = signature;
   *(reinterpret_cast<int*>(static_cast<Byte*>(pMem)
       +realSize-sizeof(int))) = signature;
   //返回指针，指向恰位于第一个 signature 之后的内存位置.
   return static_cast<Byte*>(pMem) + sizeof(int);
}
```

这个 operator new 的缺点主要在于它疏忽了身为这个特殊函数所应该具备的“坚持 C++ 规矩”的态度。举个例子，条款 51 说所有 operator news 都应该内含一个循环，反复调用某个 new-handling 函数，这里却没有。由于条款 51 就是专门为此协议而写，所以这儿我暂且忽略之。我现在只想专注于一个比较微妙的主题：齐位（*alignment*）。

许多计算机体系结构（computer architectures）要求特定的类型必须放在特定的内存地址上。例如它可能会要求指针的地址必须是 4 倍数（*four-byte aligned*）或 doubles 的地址必须是 8 倍数（*eight-byte aligned*）。如果没有奉行这个约束条件，可能导致运行期硬件异常。有些体系结构比较慈悲，没有那么霹雳，而是宣称如果齐位条件获得满足，便提供较佳效率。例如 Intel x86 体系结构上的 doubles 可被对齐于任何 byte 边界，但如果它是 8-byte 齐位，其访问速度会快许多。

在我们目前这个主题中，齐位（alignment）意义重大，因为 C++ 要求所有 operator news 返回的指针都有适当的对齐（取决于数据类型）。malloc 就是在这样的要求下工作，所以令 operator new 返回一个得自 malloc 的指针是安全的。然而上述 operator new 中我并未返回一个得自 malloc 的指针，而是返回一个得自 malloc 且偏移一个 int 大小的指针。没人能够保证它的安全！如果客户端调用 operator new 企图获取足够给一个 double 所用的内存（或如果我们写个 operator new[]，元素类型是 doubles），而我们在一部“ints 为 4 bytes 且 doubles 必须 8-byte

齐位”的机器上跑，我们可能会获得一个未有适当齐位的指针。那可能会造成程序崩溃或执行速度变慢。不论哪种情况都非我们所乐见。

像齐位（alignment）这一类技术细节，正可以在那种“因其他纷扰因素而被程序员不断抛出异常”的内存管理器中区分出专业质量的管理器。写一个总是能够运作的内存管理器并不难，难的是它能够优良地运作。一般而言我建议你在必要时才试着写写看。

很多时候是非必要的！某些编译器已经在它们的内存管理函数中切换至调试状态（enable debugging）和志记状态（logging）。快速浏览一下你的编译器文档，很可能就此消除自行撰写 new 和 delete 的需要。许多平台上已有商业产品可以替代编译器自带的内存管理器。如果需要它们来为你的程序提高机能和改善效能，你唯一需要做的就是重新连接（relink）。当然啦，首先你得花点钱买下它们。

另一个选择是开放源码（open source）领域中的内存管理器。它们对许多平台都可用，你可以下载并试试。Boost 程序库（见条款 55）的 Pool 就是这样一个分配器，它对于最常见的“分配大量小型对象”很有帮助。许多 C++ 书籍，包括本书早期版本，都曾展示高效率的小型对象分配器源码，但它们往往漏掉可移植性和齐位考虑、线程安全性……等等令人生厌的麻烦细节。真正称得上程序库者，必然稳健强固。即使你还是执意写一个自己的 news 和 deletes，看一看开放源码版本也可能对若干容易被漠视的细节（它们用来区分“几乎行得通”和“真正行得通”的制品）取得深刻的理解。齐位就是这样一个细节，TR1（见条款 54）支持各类型特定的对齐条件，很值得注意。

本条款的主题是，了解何时可在“全局性的”或“class 专属的”基础上合理替换缺省的 new 和 delete。挖掘更多细节之前，让我先对答案做一些摘要。

- **为了检测运用错误**（如前所述）。
- **为了收集动态分配内存之使用统计信息**（如前所述）。

- **为了增加分配和归还的速度**。泛用型分配器往往（虽然并不总是）比定制型分配器慢，特别是当定制型分配器专门针对某特定类型之对象而设计时。Class 专属分配器是“区块尺寸固定”之分配器实例，例如 Boost 提供的 Pool 程序库便是。如果你的程序是个单线程程序，但你的编译器所带的内存管理器具备线程安全，你或许可以写个不具线程安全的分配器而大幅改善速度。当然，在获得“operator new 和 operator delete 有加快程序速度的价值”这个结论之前，首先请分析你的程序，确认程序瓶颈的确发生在那些内存函数身上。

- **为了降低缺省内存管理器带来的空间额外开销**。泛用型内存管理器往往（虽然并非总是）不只比定制型慢，它们往往还使用更多内存，那是因为它们常常在每一个分配区块身上招引某些额外开销。针对小型对象而开发的分配器（例如 Boost 的 Pool 程序库）本质上消除了这样的额外开销。

- **为了弥补缺省分配器中的非最佳齐位**（suboptimal alignment）。一如先前所说，在 x86 体系结构上 doubles 的访问最是快速——如果它们都是 8-byte 齐位。但是编译器自带的 operator news 并不保证对动态分配而得的 doubles 采取 8-byte 齐位。这种情况下，将缺省的 operator new 替换为一个 8-byte 齐位保证版，可导致程序效率大幅提升。

- **为了将相关对象成簇集中**。如果你知道特定之某个数据结构往往被一起使用，而你又希望在处理这些数据时将“内存页错误”（page faults）的频率降至最低，那么为此数据结构创建另一个 heap 就有意义，这么一来它们就可以被成簇集中在尽可能少的内存页（pages）上。new 和 delete 的“*placement* 版本”（见条款 52）有可能完成这样的集簇行为。

- **为了获得非传统的行为**。有时候你会希望 operators new 和 delete 做编译器附带版没做的某些事情。例如你可能会希望分配和归还共享内存（shared memory）内的区块，但唯一能够管理该内存的只有 C API 函数，那么写下一个定制版 new 和 delete（很可能是 *placement* 版本，见条款 52），你便得以为 C API 穿上一件 C++ 外套。你也可以写一个自定的 operator delete，在其中将所有归还内存内容覆盖为 0，藉此增加应用程序的数据安全性。

### 请记住

- 有许多理由需要写个自定的 new 和 delete，包括改善效能、对 heap 运用错误进行调试、收集 heap 使用信息。

## 条款 51：编写 new 和 delete 时需固守常规

Adhere to convention when writing new and delete.

条款 50 已解释什么时候你会想要写个自己的 operator new 和 operator delete，但并没有解释当你那么做时必须遵守什么规则。这些规则不难奉行，但其中一些并不直观，所以知道它们究竟是些什么很重要。

让我们从 operator new 开始。实现一致性 operator new 必得返回正确的值，内存不足时必得调用 new-handling 函数（见条款 49），必须有对付零内存需求的准备，还需避免不慎掩盖正常形式的 new ——虽然这比较偏近 class 的接口要求而非实现要求。正常形式的 new 描述于条款 52。

operator new 的返回值十分单纯。如果它有能力供应客户申请的内存，就返回一个指针指向那块内存。如果没有那个能力，就遵循条款 49 描述的规则，并抛出一个 bad_alloc 异常。

然而其实也不是非常单纯，因为 operator new 实际上不只一次尝试分配内存，并在每次失败后调用 new-handling 函数。这里假设 new-handling 函数也许能够做某些动作将某些内存释放出来。只有当指向 new-handling 函数的指针是 null，operator new 才会抛出异常。

奇怪的是 C++ 规定，即使客户要求 0 bytes，operator new 也得返回一个合法指针。这种看似诡异的行为其实是为了简化语言其他部分。下面是个 non-member operator new 伪码（pseudocode）：

```
void* operator new(std::size_t size) throw(std::bad_alloc)
{                                    //你的 operator new 可能接受额外参数.
   using namespace std;
   if (size == 0) {                  //处理 0-byte 申请.
      size = 1;                      //将它视为 1-byte 申请.
   }
   while (true) {
      尝试分配 size bytes;
```

```
        if (分配成功)
        return (一个指针，指向分配得来的内存);

        //分配失败；找出目前的 new-handling 函数（见下）
        new_handler globalHandler = set_new_handler(0);
        set_new_handler(globalHandler);

        if (globalHandler) (*globalHandler)();
        else throw std::bad_alloc();
    }
}
```

这里的伎俩是把 0 bytes 申请量视为 1 byte 申请量。看起来有点黏搭搭地令人厌恶，但做法简单、合法、可行，而且毕竟客户多久才会发出一个 0 bytes 申请呢？

你也可能带着怀疑的眼光斜睨这份伪码（pseudocode），因为其中将 new-handling 函数指针设为 null 而后又立刻恢复原样。那是因为我们很不幸地没有任何办法可以直接取得 new-handling 函数指针，所以必须调用 set_new_handler 找出它来。拙劣，但有效——至少对单线程程序而言。若在多线程环境中你或许需要某种机锁（lock）以便安全处置 new-handling 函数背后的（global）数据结构。

条款 49 谈到 operator new 内含一个无穷循环，而上述伪码明白表明出这个循环；"while (true)" 就是那个无穷循环。退出此循环的唯一办法是：内存被成功分配或 new-handling 函数做了一件描述于条款 49 的事情：让更多内存可用、安装另一个 new-handler、卸除 new-handler、抛出 bad_alloc 异常（或其派生物），或是承认失败而直接 return。现在，对于 new-handler 为什么必须做出其中某些事你应该很清楚了。如果不那么做，operator new 内的 while 循环永远不会结束。

许多人没有意识到 operator new 成员函数会被 derived classes 继承。这会导致某些有趣的复杂度。注意上述 operator new 伪码中，函数尝试分配 size bytes（除非 size 是 0）。那非常合理，因为 size 是函数接受的实参。然而就像条款 50 所言，写出定制型内存管理器的一个最常见理由是为针对某特定 class 的对象分配行为提供最优化，却不是为了该 class 的任何 derived classes。也就是说，针对 class X 而设计的 operator new，其行为很典型地只为大小刚好为 sizeof(X) 的对象而设计。然而一旦被继承下去，有可能 base class 的 operator new 被调用用以分配 derived class 对象：

```
class Base {
public:
   static void* operator new(std::size_t size) throw(std::bad_alloc);
   ...
};

class Derived: public Base  //假设Derived未声明operator new
{ ... };

Derived* p = new Derived;   //这里调用的是Base::operator new
```

如果 Base class 专属的 operator new 并非被设计用来对付上述情况（实际上往往如此），处理此情势的最佳做法是将“内存申请量错误”的调用行为改采标准 operator new，像这样：

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc)
{
   if (size != sizeof(Base))          //如果大小错误，
        return ::operator new(size);  //令标准的operator new起而处理。
   ...                                //否则在这里处理。
}
```

“等一下！”我听到你大叫，“你忘了检验 size 等于 0 这种病态**但是**可能出现的情况！”。是的，我没检验，但请你收回你的**但是**。测试依然存在，只不过它和上述的“size 与 sizeof(Base)的检测”融合一起了。是的，C++ 在某种秘境中运行，而其中一个秘境就是它裁定所有非附属（独立式）对象必须有非零大小（见条款 39）。因此 sizeof(Base)无论如何不能为零，所以如果 size 是 0，这份申请会被转交到::operator new 手上，后者有责任以某种合理方式对待这份申请。

> 译注：这里所谓非附属/独立式（freestanding）对象，指的是不以“某对象之 base class 成分”存在的对象。此处所言的这个规定，可参考《*Inside the C++ Object*》by Stanly Lippman, Addison Wesley, 1996。

如果你打算控制 class 专属之“arrays 内存分配行为”，那么你需要实现 operator new 的 array 兄弟版：operator new[]。这个函数通常被称为 "array new"，因为很难想出如何发音 "operator new[]"。如果你决定写个 operator new[]，记住，唯一需要做的一件事就是分配一块未加工内存（raw memory），因为你无法对 array 之内迄今尚未存在的元素对象做任何事情。实际上你甚至无法计算这个 array 将含多少个元素对象。首先你不知道每个对象多大，毕竟 base class 的 operator new[]有可能经由继承被调用，将内存分配给“元素为 derived class 对象”的 array 使用，而你当然知道，derived class 对象通常比其 base class 对象大。

因此，你不能在 Base::operator new[]内假设 array 的每个元素对象的大小是 sizeof(Base)，这也就意味你不能假设 array 的元素对象个数是(bytes 申请数)/sizeof(Base)。此外，传递给 operator new[]的 size_t 参数，其值有可能比“将被填以对象”的内存数量更多，因为条款 16 说过，动态分配的 arrays 可能包含额外空间用来存放元素个数。

这就是撰写 operator new 时你需要奉行的规矩。operator delete 情况更简单，你需要记住的唯一事情就是 C++ 保证“删除 null 指针永远安全”，所以你必须兑现这项保证。下面是 non-member operator delete 的伪码（pseudocode）：

```
void operator delete(void * rawMemory) throw()
{
   if (rawMemory == 0) return;    //如果将被删除的是个 null 指针，
                                  //那就什么都不做。
   现在，归还 rawMemory 所指的内存；
}
```

这个函数的 member 版本也很简单，只需要多加一个动作检查删除数量。万一你的 class 专属的 operator new 将大小有误的分配行为转交::operator new 执行，你也必须将大小有误的删除行为转交::operator delete 执行：

```
class Base {        //一如以往，但此刻重点在 operator delete
public:
   static void* operator new(std::size_t size) throw(std::bad_alloc);
   static void operator delete(void* rawMemory,std::size_t size) throw();
   ...
};
void Base::operator delete(void* rawMemory, std::size_t size) throw()
{
   if (rawMemory == 0) return;              //检查 null 指针。
   if (size != sizeof(Base)) {              //如果大小错误，令标准版
       ::operator delete(rawMemory);        //operator delete 处理此一申请。
       return;
   }
   现在，归还 rawMemory 所指的内存；
   return;
}
```

有趣的是，如果即将被删除的对象派生自某个 base class 而后者欠缺 virtual 析构函数，那么 C++ 传给 operator delete 的 size_t 数值可能不正确。这是“让你的 base classes 拥有 virtual 析构函数”的一个够好的理由；条款 7 还提过一个更好

的理由。我就不岔开话题了，此刻只要你提高警觉，如果你的 base classes 遗漏 virtual 析构函数，operator delete 可能无法正确运作。

### 请记住

- operator new 应该内含一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用 new-handler。它也应该有能力处理 0 bytes 申请。Class 专属版本则还应该处理“比正确大小更大的（错误）申请”。
- operator delete 应该在收到 null 指针时不做任何事。Class 专属版本则还应该处理“比正确大小更大的（错误）申请”。

## 条款 52：写了 *placement* new 也要写 *placement* delete

Write placement delete if you write placement new.

***placement*** new 和 ***placement*** delete 并非 C++ 兽栏中最常见的动物，如果你不熟悉它们，不要感到挫折或忧虑。回忆条款 16 和 17，当你写一个 new 表达式像这样：

```
Widget* pw = new Widget;
```

共有两个函数被调用：一个是用以分配内存的 operator new，一个是 Widget 的 ***default*** 构造函数。

假设其中第一个函数调用成功，第二个函数却抛出异常。既然那样，步骤一的内存分配所得必须取消并恢复旧观，否则会造成内存泄漏（memory leak）。在这个时候，客户没有能力归还内存，因为如果 Widget 构造函数抛出异常，pw 尚未被赋值，客户手上也就没有指针指向该被归还的内存。取消步骤一并恢复旧观的责任因此落到 C++ 运行期系统身上。

运行期系统会高高兴兴地调用步骤一所调用的 operator new 的相应 operator delete 版本，前提当然是它必须知道哪一个（因为可能有许多个）operator delete 该被调用。如果目前面对的是拥有正常签名式（signature）的 new 和 delete，这并不是问题，因为正常的 operator new：

```
void* operator new(std::size_t) throw(std::bad_alloc);
```

对应于正常的 operator delete：

```
void operator delete(void* rawMemory) throw();
                                          //global 作用域中的正常签名式.
void operator delete(void* rawMemory, std::size_t size) throw();
                                          //class 作用域中典型的签名式.
```

因此，当你只使用正常形式的 new 和 delete，运行期系统毫无问题可以找出那个“知道如何取消 new 所作所为并恢复旧观”的 delete。然而当你开始声明非正常形式的 operator new，也就是带有附加参数的 operator new，“究竟哪一个 delete 伴随这个 new”的问题便浮现了。

举个例子，假设你写了一个 class 专属的 operator new，要求接受一个 ostream，用来志记（logged）相关分配信息，同时又写了一个正常形式的 class 专属 operator delete：

```
class Widget {
public:
   ...
   static void* operator new(std::size_t size, std::ostream& logStream)
      throw(std::bad_alloc);                    //非正常形式的 new
   static void operator delete(void* pMemory std::size_t size)
      throw();                                  //正常的 class 专属 delete
   ...
};
```

这个设计有问题，但在探讨原因之前，我们需要先绕道，扼要讨论若干术语。

如果 operator new 接受的参数除了一定会有的那个 size_t 之外还有其他，这便是个所谓的 ***placement*** new。因此，上述的 operator new 是个 ***placement*** 版本。众多 ***placement*** new 版本中特别有用的一个是“接受一个指针指向对象该被构造之处”，那样的 operator new 长相如下：

```
void* operator new(std::size_t, void* pMemory) throw();
                                          //placement new
```

这个版本的 new 已被纳入 C++ 标准程序库，你只要 #include <new> 就可以取用它。这个 new 的用途之一是负责在 vector 的未使用空间上创建对象。它同时也是最早的 ***placement*** new 版本。实际上它正是这个函数的命名根据：一个特定位置上的 new。以上说明意味术语 ***placement*** new 有多重定义。当人们谈到 ***placement*** new，大多数时候他们谈的是此一特定版本，也就是“唯一额外实参是个 void*”，少数时候才是指接受任意额外实参之 operator new。上下文语境往往也能够使意义不明确的含糊话语清晰起来，但了解这一点相当重要：一般性术语 "***placement***

new" 意味带任意额外参数的 new，因为另一个术语 "*placement* delete" 直接派生自它。稍后我们即将遭遇后者。

现在让我们回到 Widget class 的声明式，也就是先前我说设计有问题的那个。这里的技术困难在于，那个 class 将引起微妙的内存泄漏。考虑以下客户代码，它在动态创建一个 Widget 时将相关的分配信息志记（logs）于 cerr：

```
Widget* pw = new (std::cerr) Widget; //调用 operator new 并传递 cerr 为其
                                     //ostream 实参；这个动作会在 Widget
                                     //构造函数抛出异常时泄漏内存
```

再说一次，如果内存分配成功，而 Widget 构造函数抛出异常，运行期系统有责任取消 operator new 的分配并恢复旧观。然而运行期系统无法知道真正被调用的那个 operator new 如何运作，因此它无法取消分配并恢复旧观，所以上述做法行不通。取而代之的是，运行期系统寻找“参数个数和类型都与 operator new 相同”的某个 operator delete。如果找到，那就是它的调用对象。既然这里的 operator new 接受类型为 ostream& 的额外实参，所以对应的 operator delete 就应该是：

```
void operator delete(void*, std::ostream&) throw();
```

类似于 new 的 ***placement*** 版本，operator delete 如果接受额外参数，便称为 ***placement*** deletes。现在，既然 Widget 没有声明 ***placement*** 版本的 operator delete，所以运行期系统不知道如何取消并恢复原先对 ***placement*** new 的调用。于是什么也不做。本例之中如果 Widget 构造函数抛出异常，不会有任何 operator delete 被调用（那当然不妙）。

规则很简单：如果一个带额外参数的 operator new 没有“带相同额外参数”的对应版 operator delete，那么当 new 的内存分配动作需要取消并恢复旧观时就没有任何 operator delete 会被调用。因此，为了消弭稍早代码中的内存泄漏，Widget 有必要声明一个 ***placement*** delete，对应于那个有志记功能（logging）的 ***placement*** new：

```
class Widget {
public:
   ...
   static void* operator new(std::size_t size, std::ostream& logStream)
      throw(std::bad_alloc);
```

```
    static void operator delete(void* pMemory) throw();
    static void operator delete(void* pMemory, std::ostream& logStream)
      throw();
    ...
};
```

这样改变之后，如果以下语句引发 Widget 构造函数抛出异常：

```
Widget* pw = new (std::cerr) Widget;          //一如以往，但这次不再泄漏
```

对应的 ***placement*** delete 会被自动调用，让 Widget 有机会确保不泄漏任何内存。

然而如果没有抛出异常（通常如此），而客户代码中有个对应的 delete，会发生什么事：

```
delete pw;                          //调用正常的 operator delete
```

就如上一行注释所言，调用的是正常形式的 operator delete，而非其 ***placement*** 版本。***placement*** delete 只有在“伴随 ***placement*** new 调用而触发的构造函数”出现异常时才会被调用。对着一个指针（例如上述的 pw）施行 delete 绝不会导致调用 ***placement*** delete。不，绝对不会。

这意味如果要对所有与 ***placement*** new 相关的内存泄漏宣战，我们必须同时提供一个正常的 operator delete（用于构造期间无任何异常被抛出）和一个 ***placement*** 版本（用于构造期间有异常被抛出）。后者的额外参数必须和 operator new 一样。只要这样做，你就再也不会因为难以察觉的内存泄漏而失眠。唔，至少不是本条款所说的这些难以察觉的内存泄漏。

附带一提，由于成员函数的名称会掩盖其外围作用域中的相同名称（见条款 33），你必须小心避免让 class 专属的 news 掩盖客户期望的其他 news（包括正常版本）。假设你有一个 base class，其中声明唯一一个 ***placement*** operator new，客户端会发现他们无法使用正常形式的 new：

```
class Base {
public:
   ...
   static void* operator new(std::size_t size,
                             std::ostream& logStream)
     throw(std::bad_alloc);          //这个 new 会遮掩正常的 global 形式
   ...
};
```

```
Base* pb = new Base;                    //错误! 因为正常形式的operator new 被掩盖.
Base* pb = new (std::cerr) Base;        //正确，调用Base 的 placement new.
```

同样道理，derived classes 中的 operator news 会掩盖 global 版本和继承而得的 operator new 版本：

```
class Derived: public Base {                        //继承自先前的 Base
public:
   ...
   static void* operator new(std::size_t size) //重新声明正常形式的
      throw(std::bad_alloc);                        //形式的 new
   ...
};

Derived* pd = new (std::clog) Derived;      //错误! 因为 Base 的
                                            // placement new 被掩盖了。
Derived* pd = new Derived;                  //没问题，调用 Derived 的
                                            // operator new.
```

条款 33 更详细地讨论了这种名称遮掩问题。对于撰写内存分配函数，你需要记住的是，缺省情况下 C++ 在 global 作用域内提供以下形式的 operator new：

```
void* operator new(std::size_t) throw(std::bad_alloc); //normal new.
void* operator new(std::size_t, void*) throw();        //placement new.
void* operator new(std::size_t,                        //nothrow new,
                   const std::nothrow_t&) throw(); //见条款 49。
```

如果你在 class 内声明任何 operator news，它会遮掩上述这些标准形式。除非你的意思就是要阻止 class 的客户使用这些形式，否则请确保它们在你所生成的任何定制型 operator new 之外还可用。对于每一个可用的 operator new 也请确定提供对应的 operator delete。如果你希望这些函数有着平常的行为，只要令你的 class 专属版本调用 global 版本即可。

完成以上所言的一个简单做法是，建立一个 base class，内含所有正常形式的 new 和 delete：

```
class StandardNewDeleteForms {
public:
   // normal new/delete
   static void* operator new(std::size_t size) throw(std::bad_alloc)
   { return ::operator new(size); }

   static void operator delete(void* pMemory) throw()
   { ::operator delete(pMemory); }
```

```
    // placement new/delete
    static void* operator new(std::size_t size, void* ptr) throw()
    { return ::operator new(size, ptr); }

    static void operator delete(void* pMemory, void* ptr) throw()
    { return ::operator delete(pMemory, ptr); }

    // nothrow new/delete
    static void* operator new(std::size_t size, const std::nothrow_t& nt) throw()
    { return ::operator new(size, nt); }

    static void operator delete(void *pMemory, const std::nothrow_t&) throw()
    { ::operator delete(pMemory); }
};
```

凡是想以自定形式扩充标准形式的客户，可利用继承机制及 using 声明式（见条款 33）取得标准形式：

```
class Widget: public StandardNewDeleteForms {          //继承标准形式
public:
    using StandardNewDeleteForms::operator new;         //让这些形式可见
    using StandardNewDeleteForms::operator delete;

    static void* operator new(std::size_t size,         //添加一个自定的
                              std::ostream& logStream) //placement new
      throw(std::bad_alloc);

    static void operator delete(void* pMemory,          //添加一个对应的
                                std::ostream& logStream) //placement delete
      throw();
    ...
};
```

## 请记住

- 当你写一个 *placement* operator new，请确定也写出了对应的 *placement* operator delete。如果没有这样做，你的程序可能会发生隐微而时断时续的内存泄漏。
- 当你声明 *placement* new 和 *placement* delete，请确定不要无意识（非故意）地遮掩了它们的正常版本。