定制的 new 和 delete

条款49: 了解 new-handleer 的行为

- 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。

当 operator new 无法满足某一内存分配需求时，它会抛出异常。对于旧式操作，该行为会返回一个 null 指针。我们可以通过修改错误处理函数来改变这种默认行为，从而获得旧式操作体验。

当 operator new 的行为无法被满足时，它会先调用一个客户指定的错误处理函数，所谓的 new-handler。客户可以通过调用 set_new_handler 来指定函数。

```
namespace std {
  typedef void (*new_handler) ();
  new_handler set_new_handler(new_handler p) throw(); // 不抛出任何异常
}
```

你可以这样设置 new-handler。

```
#include <new>

void outOfMem() {
  std::cerr << "Unable to satisfy request for memory\n";
  std::abort(); // 终结
}

int main() {
  std::set_new_handler(outOfMem);
  int* pBigDataArray = new int[10000000000L];
}
```

当 operator new 无法满足内存申请时，它会不断调用 new-handler 函数，直到找到足够的内存。可 以将上面的 std::abort 注释掉尝试一下。

因此一个结论是，一个设计良好的 new-handler 函数必须做到以下的事情:

- 让更多内存可以被使用: 这个策略的一个做法是，程序一开始执行就分配一大块内存，然后再 new-handler 第一次被调用时，将它们释放给程序使用。
- 安装另一个 new-handler: 如果当前这个 new-handler 无法获取更多的可用内存，或许它直到另外哪个 new-handler 有此能力，如果是这样可以对 new-handler 进行替换。
- 卸除 new-handler: 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。

- 不返回: 直接调用 abort 或者 exit。

有时，我们希望能够根据不同的 class 来执行专属的内存分配失败处理函数。c++ 并不支持 class 专 属的 new-handlers，但是可以为每一个 class 提供自己的 set_new_handler 和 operator new。

```
class NewHandlerHolder { // RAII
public:
  explicit NewHandlerHolder (std::new_handler nh) : handler(nh) {}
  ~NewHandlerHolder() {
    std::set_new_handler(handler);
  }
private:
  std::new_handler handler;
  NewHandlerHolder(const NewHandlerHolder&); // 阻止 copying 发生
  NewHandlerHolder& operator=(const NewHandlerHolder&);
};


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;
};

std::new_handler currentHandler = 0;
std::new_handler set_new_handler(std::new_handler p) throw() {
  std::new_handler oldHandler = currentHandler;
  crrentHandler = p;
  return oldHandler;
}
void* Widget::operator new(std::size_t size) throw(std::bad_alloc) {
  NewHandlerHolder h(std::set_new_handler(currentHandler)); // 返回时自动销毁，并在
析构函数中重新绑定默认的 new_handler
  return ::operator new(size); // 调用 global operator new
}
```

上述代码可以进行一般化处理，采用复合的方式来构建。简单的做法是简历一个 “mixin” 风格的 base class，这种 base class 用来允许 derived class 继承单一特定的能力。

```
template <typename T>
class NewHandlerSupport {
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;
};
template <typename T>
```

```
std::new_handler NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  crrentHandler = 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)); // 返回时自动销毁，并在
析构函数中重新绑定默认的 new_handler
  return ::operator new(size); // 调用 global operator new
}
template <typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;

class Widget : public NewHandlerSupport<Widget> { // 不再需要提供 set_new_handler
和 operator new
  ...
};
```

上面的 T 只是为了区分类型，而没有特殊含义。这种行为叫做 curiously recurring template pattern; CRTP。

现在大多数的 operator new 实现，如果空间不足都会抛出 bad_alloc 异常。但是也可以支持返回 null 的方法。

```
class Widget {};
Widget* pw1 = new Widget(); // 如果分配失败，抛出 bad_alloc
if (pw1 == 0) ... // 没必要，因为一定 pw1 不可能为0

Widget* pw2 = new (std::nothrow) Widget(); // 失败，返回0
if (pw2 == 0) ... // 可能成功
```

然而这种方式完全没必要，因为 nothrow 之后限制 operator new 抛出异常，而当 constructor 抛 出异常仍然会传播。

条款50: 了解 new 和 delete 的合理替换时机

- 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

为什么会想要替换编译器提供的 operator new 或者 operator delete ?

- 用来检测运用上的错误。
- 为了强化效能。
- 为了收集使用上的统计数据。

一个定制 operator new 的例子，用于促进并协助检查 "overruns"(写入点再分配区块之后) 或 "underruns"(写入点再分配区块之前)。

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;
// 下列代码仍然存在错误
void* operator new(std::size_t size) throw (std::bad_alloc) {
  using namespace std;
  size_t realSize = size + 2 * sizeof(int); // 增加两个空间用来存放 两个 signature

  void* pMem = malloc(realSize); // 调用 malloc 取得内存
  if (!pMem) throw bad_alloc(); // 如果为空就抛出 bad_alloc

  // 将 signature 写入内存的最前和最后
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem) + realSize - sizeof(int))) =
signature;

  return static_cast<Byte*>(pMem) + sizeof(int); // 返回 signature 之后的内存位置
}
```

对齐

这里强调一下 对齐 问题。在计算机体系结构中，如果对齐条件满足，通常是效率较高的。比如一个 double 如果是 8-byte，当它正好与地址的 8-byte 位置对齐时效率时最高的。通常使用 malloc 获得的地址 是满足对齐的。但是上述代码中，我们显然返回的不是原始 pointer，而是一个后移了 4-byte 的指针。 这时就可能获得的是一个没有适当对齐的指针，那么可能会造成程序崩溃或者执行速度缓慢。

有时好的效率是必要的，但是很多时候这也没有太大影响。

那么究竟何时需要替换缺省的 new 和 delete 呢？

- 为了检测运用错误。
- 为了收集动态分配内存之使用统计信息。
- 为了增加分配和归还的速度。
- 为了降低缺省内存管理器带来的空间额外开销。
- 为了弥补缺省分配器中的非最佳对齐。
- 为了将相关对象成簇集中。
- 为了获得非传统行为。

条款51: 编写 new 和 delete 时需要固守常规

- 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。它在应该有能力处理 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 “比正确大小更大的申请”。

编写自己的 operator new 和 operator delete 时应该遵守哪些规矩。

operator new

实现一致性的 operator new 1. 必须得返回正确的值，2. 内存不足时必须得调用 new-handling 函 数，3. 必须有能够对付零内存需求的准备，4. 避免不慎掩盖正常形式的 new。

operator new 的返回值非常简单，如果有能力就返回内存指针，没有则抛出 bad_alloc 异常。

一个 operator new 的伪代码:

```
void* operator new(std::size_t size) throw(std::bad_alloc) {
  using namespace std;
  if (size == 0) { // 用来处理 申请 0-bytes 内存的行为
    size = 1;
  }
  while (true) {
    尝试分配 size bytpes;
    if (成功)
      return pointer;

    // 分配失败
    new_handler globalHandler = set_new_handler(0); // 获取当前的 new-handling 函数
    set_new_handler(globalHandler);

    if (globalHandler) // 如果 globalHandler 不为空就执行
      (*gloablHandler)();
    else // 否则默认抛出异常
      throw std::bad_alloc();
  }
}
```

但是上述代码有一个小缺点，就是会影响 derived class。而通常我们重写 operator new 实际上是 为了能够针对某个特定 class 进行优化，而非其它 derived class。一种好的方法是，在operator new 中进行判断大小。

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc) {
  if(size != sizeof(Base))
    return ::operator new(size);
}
```

这不仅包括了对类型的判断，也包括了对 size == 0 的判断，因为 c++ 中独立对象必须有大小，因此 sizeof(Base) 不可能为0。

对于 operator new[] 来分配数组，唯一需要做的就是分配一块未加工的内存，其他计算空间大小等工 作都不应该在此处进行。

operator delete

operator delete 情况十分简单，只需要记住的唯一一件事情，就是保证 “删除 null 指针永远安全”。

```
void operator delete(void* rawMemory) throw() {
  if (rawMemory == 0) return;
```

```
  // 归还 rawMemory 内存。
}
```

member 版本也十分简单，和 operator new 一样进行类型判断即可。

```
void* Base::operator delete(void* rawMemory, std::size_t size) throw() {
  if (rawMemory == 0) return;
  if(size != sizeof(Base)) {
    ::operator delete(rawMemory);
    return;
  }
  ...
}
```

另外，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。

条款52: 写了 placement new 也要写 placement delete

- 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时断时续的内存泄露。
- 当你声明 placement new 和 placement delet，请确定不要无疑是的遮掩他们的正常版本。

---

new operator 总是分为两步进行的，首先调用 operator new 进行空间申请，而第二步则是执行 constructor。如果后者出错，那么 operator new 所分配的地址空间应该怎样做？答案必然是被释放以防止内存泄露。 但是有个问题，就是编译器怎么找到对应的 operator delete 呢？

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

{
  Widget* pw = new (std::cerr) Widget;
}
```

实际上编译器会选择同调用的 operator new 最相像的 operator delete 进行调用，也就是说寻找 参数个数和类型都与 operator new 相同的某个 operator delete。例如上面这段代码，当 Widget constructor 发生异常后就会调用 operator delete(std::size_t size, std::ostream& logStream); 而上面没有定义就会产生另一个异常最终导致 abort。

而在正常的主动删除情况下，可以直接 delete 即可，而此时并不会去调用与 operator new 对应版本 的 operator delete。

所以为了保证内存安全，你应该随时虽可提供相应形式的 delete。这和 placement new/delete 成对 设计的道理相同。

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

{
  Widget* pw = new (std::cerr) Widget;
}
```

需要注意的一点是，重新定义 operator new/delete 是会进行名称掩盖的。

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

Base* pb = new Base; // 错误，global 被掩盖了
Base* pb = new (std::cerr) Base; // 正确

Derived* pd = new (std::clog) Derived; // 错误，Base 被掩盖掉了
Derived* pd = new Derived; // 正确
```

如果你不希望这些默认行为发生掩盖呢？你可以按照下面这样进行设计。

```
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(std::bad_alloc) {
    return ::operator new(size, ptr);
  }
  static void operator delete(void* pMemory, void* ptr) throw() {
    ::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& nt) throw() {
    ::operator delete(pMemory);
  }
};


class Widget : public StandardNewDeleteForms {
public:
  using StandardNewDeleteForms::operator new;
  using StandardNewDeleteForms::operator delete;
  static void* operator new(std::size_t size, std::ostream& logStream)
throw(std::bad_alloc);
  static void operator delete(void* pMemory, std::ostream& logStream) throw();
};
```

利用继承机制和 using 声明式来取得标准形式。