[**彻底学习STL中的Allocator**](http://cissco.iteye.com/blog/379093)

**博客分类：**

* [Linux/Win C＋＋](http://cissco.iteye.com/category/46085)

[IT厂商](http://www.iteye.com/blogs/tag/IT%E5%8E%82%E5%95%86)[C](http://www.iteye.com/blogs/tag/C)[C++](http://www.iteye.com/blogs/tag/C++)[C#](http://www.iteye.com/blogs/tag/C%23)[活动](http://www.iteye.com/blogs/tag/%E6%B4%BB%E5%8A%A8)

Allocator是C++语言标准库中最神秘的部分之一。它们很少被显式使用，标准也没有明确出它们应该在什么时候被使用。今天的allocator与最初的STL建议非常不同，在此过程中还存在着另外两个设计－－这两个都依赖于语言的一些特性，而直到最近才在很少的几个编译器上可用。对allocator的功能，标准似乎在一些方面追加了承诺，而在另外一些方面撤销了承诺。

    这篇专栏文章将讨论你能用allocator来做什么以及如何定义一个自己的版本。我只会讨论C++标准所定义的allocator：引入准标准时代的设计，或试图绕过有缺陷的编译器，只会增加混乱。

**什么时候不使用Allocator**

    C++标准中的Allocator分成两块：一个通用需求集（描述于§ 20.1.5(表 32)），和叫std::allocator的class（描述于§20.4.1）。如果一个class满足表32的需求，我们就称它为一个allocator。std::allocator类满足那些需求，因此它是一个allocator。它是标准程序库中的唯一一个预先定义allocator类。

    每个 C++程序员都已经知道动态内存分配：写下new X来分配内存和创建一个X类型的新对象，写下delete p来销毁p所指的对象并归还其内存。你有理由认为allocator会使用new和delete－－但它们没有。（C++标准将::operator new()描述为“allocation function”，但很奇怪，allocator并不是这样的。）

    有关allocator的最重要的事实是它们只是为了一个目的：封装STL容器在内存管理上的低层细节。你不应该在自己的代码中直接调用allocator的成员函数，除非正在写一个自己的STL容器。你不应该试图使用allocator来实现operator new[]；这不是allocator该做的。 如果你不确定是否需要使用allocator，那就不要用。

    allocator是一个类,有着叫allocate()和deallocate()成员函数（相当于malloc和free）。它还有用于维护所分配的内存的辅助函数和指示如何使用这些内存的typedef（指针或引用类型的名字）。如果一个STL容器使用用户提供的allocator来分配它所需的所有内存(预定义的STL容器全都能这么做；他们都有一个模板参数，其默认值是std::allocator)，你就能通过提供自己的allocator来控制它的内存管理。

    这种柔性是有限制的：仍然由容器自己决定它将要申请多少内存以及如何使用它们。在容器申请更多的内存时，你能控制它调用那个低层函数，但是你不能通过使用allocator来让一个vector行动起来像一个deque一样。虽然如此，有时候，这个受限的柔性也很有用。比如，假设你有一个特殊的fast\_allocator，能快速分配和释放内存（也许通过放弃线程安全性，或使用一个小的局部堆），你能通过写下std::list<T, fast\_allocator<T> >而不是简单的std::list<T>来让标准的list使用它。

    如果这看起来对你很陌生，那就对了。没有理由在常规代码中使用allocator的。

**定义一个Allocator**

    关于allocator的这一点你已经看到了：它们是模板。Allocator，和容器一样，有value\_type，而且allocator的value\_type必须要匹配于使用它的容器的value\_type。这有时会比较丑陋：map的value\_type相当复杂，所以显式调用allocator的map看起来象这样的，std::map<K，V, fast\_allocator<std::pair<const K, V> > >。(像往常一样，typedef会对此有帮助。)

    以一个简单的例子开始。根据C++标准，std::allocator构建在::operator new()上。如果你正在使用一个自动化内存使用跟踪工具，让std::allocator更简单些会更方便。我们可以用malloc()代替::operator new()，而且我们也不考虑（在好的std::allocator实作中可以找到的）复杂的性能优化措施。我们将这个简单的allocator叫作malloc\_allocator 。

    既然malloc\_allocator的内存管理很简单，我们就能将重点集中在所有STL的allocator所共有的样板上。首先，一些类型：allocator是一个类模板，它的实例专为某个类型T分配内存。我们提供了一序列的typedef，以描述该如何使用此类型的对象：value\_type指T本身，其它的则是有各种修饰字的指针和引用。

Cpp代码  [收藏代码](javascript:void())

1. template <class T> class malloc\_allocator
3. {
5. public:
7. typedef T                 value\_type;
9. typedef value\_type\*       pointer;
11. typedef const value\_type\* const\_pointer;
13. typedef value\_type&       reference;
15. typedef const value\_type& const\_reference;
17. typedef std::size\_t       size\_type;
19. typedef std::ptrdiff\_t    difference\_type;
21. ...
23. };

 这些类型与STL容器中的很相似，这不是巧合：容器类常常直接从它的allocator提取这些类型。

    为什么有这么多的typedef？你可能认为pointer是多余的：它就是value\_type \*。绝大部份时候这是对的，但你可能有时候想定义非传统的allocator，它的pointer是一个pointer-like的class，或非标的厂商特定类型value\_type \_\_far \*；allocator是为非标扩展提供的标准hook。不寻常的pointer类型也是存在address()成员函数的理由，它在malloc\_allocator中只是operator &()的另外一种写法：

Cpp代码  [收藏代码](javascript:void())

1. template <class T> class malloc\_allocator
3. {
5. public:
7. pointer address(reference x) const { return &x; }
9. const\_pointer address(const\_reference x) const {
11. return &x;
13. }
15. ...
17. };

 现在我们能开始真正的工作：allocate()和deallocate()。它们很简单，但并不十分象malloc()和free()。我们传给allocate()两个参数：我们正在为其分派空间的对象的数目(max\_size返回可能成功的最大请求值)，以及可选的一个地址值（可以被用作一个位置提示）。象malloc\_allocator这样的简单的allocator没有利用那个提示，但为高性能而设计的allocator可能会利用它。返回值是一个指向内存块的指针，它足以容纳n个value\_type类型的对象并有正确的对齐。

    我们也传给deallocate()两个参数：当然一个是指针，但同样还有一个元素计数值。容器必须自己掌握大小信息；传给allocate和deallocate的大小参数必须匹配。同样，这个额外的参数是为效率而存在的，而同样，malloc\_allocator不使用它。

Cpp代码  [收藏代码](javascript:void())

1. template <class T> class malloc\_allocator
3. {
5. public:
7. pointer allocate(size\_type n, const\_pointer = 0) {
9. void\* p = std::malloc(n \* sizeof(T));
11. if (!p)
13. throw std::bad\_alloc();
15. return static\_cast<pointer>(p);
17. }
19. void deallocate(pointer p, size\_type) {
21. std::free(p);
23. }
25. size\_type max\_size() const {
27. return static\_cast<size\_type>(-1) / sizeof(value\_type);
29. }
31. ...
33. };

 allocate()和deallocate()成员函数处理的是未初始化的内存，它们不构造和销毁对象。语句a.allocate(1)更象malloc(sizeof(int))而不是new int。在使用从allocate()获得的内存前，你必须在这块内存上创建对象；在通过deallocate()归还内存前，你需要销毁那些对象。

    C++语言提供一个机制以在特定的内存位置创建对象：placement new。如果你写下new(p) T(a, b)，那么你正在调用T的构造函数产生一个新的对象，一如你写的new T(a, b)或 T t(a, b)。区别是当你写new(p) T(a, b)，你指定了对象被创建的位置：p所指向的地址。(自然，p必须指向一块足够大的内存，而且必须是未被使用的内存；你不能在相同的地址构建两个不同的对象。)。你也可以调用对象的析构函数，而不释放内存，只要写p->~T()。

    这些特性很少被使用，因为通常内存的分配和初始化是一起进行的：使用未初始化的内存是不方便的和危险的。你需要如此低层的技巧的很少几处之一就是你在写一个容器类，于是allocator将内存的分配与初始化解耦。成员函数construct()调用placement new，而且成员函数destory()调用析构函数。

Cpp代码  [收藏代码](javascript:void())

1. template <class T> class malloc\_allocator
3. {
5. public:
7. void construct(pointer p, const value\_type& x) {
9. new(p) value\_type(x);
11. }
13. void destroy(pointer p) { p->~value\_type(); }
15. ...
17. };

 (为什么allocator有那些成员函数，什么时候容器可以直接使用placement new？一个理由是要隐藏笨拙的语法，而另一个是如果写一个更复杂的allocator时你可能想在构造和销毁对象时construct()和destroy()还有其它一些副作用。比如，allocator可能维护一个所有当前活动对象的日志。)

    这些成员函数没有一个是static的，因此，容器在使用allocator前做的第一件事就是创建一个allocator对象－－也就是说我们应该定义一些构造函数。但是，我们不需要赋值运算：一旦容器创建了它的allocator，这个allocator就从没想过会被改变。表32中的对allocator的需求没有包括赋值。只是基于安全，为了确保没人偶然使用了赋值运算，我们将禁止掉这个可能自动生成的函数。

Cpp代码  [收藏代码](javascript:void())

1. template <class T> class malloc\_allocator
3. {
5. public:
7. malloc\_allocator() {}
9. malloc\_allocator(const malloc\_allocator&) {}
11. ~malloc\_allocator() {}
13. private:
15. void operator=(const malloc\_allocator&);
17. ...
19. };

 这些构造函数实际上没有做任何事，因为这个allocator不需要初始化任何成员变量。基于同样的理由，任意两个malloc\_allocator都是可互换的；如果a1和a2的类型都是malloc\_allocator<int>，我们可以自由地通过a1来allocate()内存然后通过a2来deallocate()它。我们因此定义一个比较操作以表明所有的malloc\_allocator对象是等价的：

Cpp代码  [收藏代码](javascript:void())

1. template <class T>
3. inline bool operator==(const malloc\_allocator<T>&,
5. const malloc\_allocator<T>&) {
7. return true;
9. }
11. template <class T>
13. inline bool operator!=(const malloc\_allocator<T>&,
15. const malloc\_allocator<T>&) {
17. return false;
19. }

 你会期望一个allocator，它的不同对象是不可替换的吗？当然－－但很难提供一个简单而有用的例子。一种明显的可能性是内存池。它对大型的C程序很常见，从几个不同的位置（“池”）分配内存，而不是什么东西都直接使用malloc()。这样做有几个好处，其一是it only takes a single function call to reclaim all of the memory associated with a particular phase of the program。 使用内存池的程序可能定义诸如mempool\_Alloc和mempool\_Free这样的工具函数，mempol\_Alloc(n, p)从池p中分配n个字节。很容易写出一个mmepool\_alocator以匹配这样的架构：每个mempool\_allocator对象有一个成员变量以指明它绑定在哪个池上，而mempool\_allocator::allocate()将调用mempool\_Alloc()从相应的池中获取内存。[注1]

    最后，我们到了allocator的定义体中一个微妙的部份：在不同的类型之间映射。问题是，一个allocator类，比如malloc\_allocator<int>，全部是围绕着单个value\_type构建的：malloc\_allocator<int>::pointer是int\*，malloc\_allocator<int>().allocate(1)返回足够容纳一个int对象的内存，等等。然而，通常，容器类使用malloc\_allocator可能必须处理超过一个类型。比如，一个list类，不分配int对象；实际上，它分配list node对象。(我们将在下一段落研究细节。)于是，当你创建一个std::list<int, malloc\_allocator<int> >时，list必须将malloc\_allocator<int>转变成为处理list\_node类型的malloc\_allocator。

    这个机制称为重绑定，它有二个部份。首先，对于给定的一个value\_type是X1的allocator类型A1，你必须能够写出一个allocator类型A2，它与A1完全相同，除了value\_type是X2。其次，对于给定的A1类型的对象a1，你必须能够创建一个等价的A2类型对象a2。这两部分都使用了成员模板，这也就是allocator不能被老的编译器支持，或支持得很差的原因。