第 2 章　简单的 C++垃圾回收器

回顾整个计算的历史，关于管理动态分配内存使用的最优方法一直存在争议。动态分配的内存是在运行期间从堆中获得的内存，堆是供程序使用的自由存储区域。堆通常也称为自由存储区或者动态内存。动态分配非常重要，因为它使得程序在执行期间可以获取、使用、释放，然后重用内存。因为几乎所有实际的程序都会以某种形式使用动态分配，所以管理动态分配的内存的方法对于程序的结构和性能有着深远的影响。

通常，有两种方法处理动态内存。第一种方法是手工方式，程序员必须显式地释放不再使用的内存，从而使得这块内存可以重用。第二种方法依赖于一种自动处理，通常称其为垃圾回收，当某块内存不再需要时会自动被回收器回收。两种方法各有千秋，随着时间的推移，比较好的策略是交替使用这两种方法。

C++使用手工的方法管理动态内存。Java 和 C#使用垃圾回收机制。Java 和 C#是比较新的语言，当前计算机语言设计的倾向是使用垃圾回收器。然而，这并不意味着 C++的程序员被放置在“历史的对立面”。因为 C++具有内建的强大功能，为 C++建立一个垃圾回收器是可能的，甚至是很容易的。因此，C++程序员拥有两者的优点：既可以手工管理动态内存的分配，也可以在需要时使用自动的垃圾回收器。

本章为 C++开发一个完整的垃圾回收子系统。在开始时，很重要的一点是要理解这个垃圾回收器并没有取代 C++内建的动态分配方法。而只是对它进行了补充。因此，在同一个程序中可以使用手工的管理和垃圾回收系统。

本书选用垃圾回收器作为第一个示例，是因为不仅代码本身是有用的(并且是迷人的)，而且它还显示了 C++不可超越的强大功能。通过使用模板类、运算符重载以及 C++的继承能力来处理计算机操纵的底层元素，如内存地址，可以透明地向 C++中加入核心特征。对于大多数的其他语言来说，改变处理动态分配的方式需要改变编译器本身。然而，由于 C++给予程序员的空前强大的功能，这个任务可以在源代码层次上完成。

这个垃圾回收器还显示了如何将一个新的类型定义并整合到 C++的程序设计环境中。这种类型可扩展性(type extensibility)是 C++的关键部分，经常会被忽略。最后，这个垃圾回收器证明了 C++“与机器联系紧密”的能力，因为它操纵并管理了指针。不同于其他阻止对底层细节访问的语言，C++允许程序员在需要的时候充分接近硬件。

2.1　两种内存管理方法的比较

在为 C++开发垃圾回收器之前，比较垃圾回收和内建于 C++中的手工方法是有好处的。通常，在 C++中使用动态内存需要两个步骤。首先，通过 new 从堆中分配内存。然后在不需要这块内存的时候，使用 delete 释放它。因此，每一次动态分配都要遵循下面的顺序：

```
p = new some_object;
//...
delete p;
```

通常，每一次使用 new 分配内存后，都必须有匹配的 delete 操作来释放内存。如果不使用 delete，内存就不会被释放，即使您的程序已经不再需要这块内存。

垃圾回收在一个关键方式上不同于手工方法：它自动释放不再需要的内存。因此，通过使用垃圾回收，动态分配只需要一步操作。例如，在 Java 和 C#中，使用 new 分配需要的内存，但是在程序中绝对不需要显式地释放它。相反，垃圾回收器会定期运行，查找不再有其他对象指向的内存块。当没有其他对象指向一个动态内存块时，就意味着程序的元素不再使用这块内存。当找到一块不再使用的内存时，垃圾回收器就会释放它。因此，在一个垃圾回收系统中，没有 delete 运算符，也不需要。

乍看上去，垃圾回收的内在简单性使得它成为管理动态内存显而易见的选择。事实上，人们可能会有疑问，究竟为什么要使用手工方法，特别是对于 C++这样一种成熟的语言。然而，在动态分配的时候，第一感觉是具有欺骗性的，因为这两种方法都涉及到一组权衡。哪一种方法更好是由应用程序决定的。下面部分描述了一些涉及到的问题。

2.1.1 手工内存管理的优缺点

手工管理动态内存的主要优点是效率。由于没有使用垃圾回收器，从而不需要花费时间来跟踪活动的对象或者周期性地查找不再使用的内存。而是当程序员知道分配的对象不再需要这块内存的时候，他可以显式地释放它，而不需要多余的开销。由于没有垃圾回收相关的开销，手工方法可以编写更加高效的代码。这就是 C++需要支持手工内存管理的原因之一：它能够建立高性能代码。

手工管理的另一个优点是控制。尽管要求程序员同时处理内存的分配和释放是一个负担，但这样做的好处是程序员获得了对这个过程两个方面的完全控制。您精确地知道分配内存的时刻，也精确地知道释放它的时刻。另外，当通过 delete 释放一个对象的时候，其析构函数在这个时刻执行，而不是像垃圾回收那样在后面的某个时候执行。因此，通过手工方法，可以精确地控制指定对象销毁的时刻。

尽管手工内存管理的效率高，但是它也容易导致相当恼人的一类错误：内存泄漏。由于必须手工释放内存，可能(甚至很容易)忘记这样做。忘记释放不再使用的内存意味着这块内存仍然被分配，即使不再需要它。但在垃圾回收环境中不会发生内存泄漏，因为垃圾回收器确保不再使用的对象最终会被释放。在 Windows 程序设计中，内存泄漏尤其恼人，在这个系统中忘记释放不再使用的内存会逐渐地降低系统性能。

C++的手工方法可能涉及到的其他问题包括：过早地释放了仍然在使用的内存，或者不小心将同一块内存释放了两次。这两种错误都会导致严重的问题。而且它们不会立即显示任何征兆，这就使得很难发现这类错误。

2.1.2 垃圾回收的优缺点

实现垃圾回收有多种方法，每一种方法都提供了不同的性能特征。然而，所有的垃圾回收系统都具有一个共同的、与手工方法相对的属性。垃圾回收最主要的优点是简单和安全。在垃

圾回收环境中，可以显式地使用 new 分配内存，但是不需要显式地释放内存。相反，不再使用的内存会被自动回收。因此，不可能会忘记释放对象或者过早地释放对象。这样做简化了程序设计，并且阻止了有问题的类。另外，不可能会意外地两次释放动态分配的内存。因此，垃圾回收为内存管理问题提供了一种易于使用的、不容易犯错的、可靠的解决方案。

遗憾的是，垃圾回收的简单及安全性是有代价的。第一个代价是垃圾回收机制引起的开销。所有垃圾回收的配置都会消耗一些 CPU 资源，因为这种不再使用的内存的回收并不是一个免费过程。当使用手工方法的时候，不会有这样的开销。

第二个代价是在销毁对象时容易失控。使用手工方法时，当对对象执行 delete 语句的时候，及时地销毁这个对象(和所调用的它的析构函数)，而垃圾回收没有这种切实而快速的规则。相反，当使用垃圾回收时，直到回收器运行并回收对象的时候，对象才会被销毁，而回收器只有在某个特定时刻才会运行。例如，回收器只有在自由内存的数量低于某个值的时候才会运行。另外，用户并不能总是知道垃圾回收器销毁对象的顺序和时间。在某些情况下，不能准确地知道对象销毁的时间会导致一些问题，因为这意味着程序不能准确地知道何时为动态分配的对象调用析构函数。

对于作为后台任务运行的垃圾回收系统，这种失控可能会引发某种应用程序潜在的更加严重的问题，因为这样做将某种本质上不确定的行为引入到程序中。在后台运行的垃圾回收器实际上在不可预知的某个时刻回收不再使用的内存。例如，回收器通常只有在 CPU 空闲的时候才会运行。由于可能从一个程序的运行转到下一个程序，从一台计算机转到下一台计算机，或者从一个操作系统转到另一个操作系统，因此垃圾回收器在程序中执行的确切位置是不能确定的。对于许多应用程序而言，这并不存在问题，但是对于实时应用程序这可能会引发灾难，因为在实时应用程序中对垃圾回收器不可预知的 CPU 循环的分配会导致事件的丢失。

2.1.3 两种方法都可以使用

正如前面讨论所阐述的那样，手工管理和垃圾回收都强化了一个特性而牺牲了另一个特性。手工方法强化了效率和控制，牺牲了安全性和易用性。垃圾回收强化了简单性及安全性，但是付出了运行性能降低和控制丢失的代价。因此，垃圾回收及手工内存管理本质上是相对的，每一种方法都强化了另一种方法牺牲的特性。这就是没有一种动态内存管理的方法可以适用于所有的程序设计情况的原因。

尽管这两种方法是对立的，但是它们并不互相排斥，它们可以共存。因此对于 C++程序员，这两种方法都可以使用，只需为手头的任务选择一种合适的方法。所要做的事情只是为 C++建立一个垃圾回收器，这就是本章下面部分的主题。

2.2 在 C++中创建垃圾回收器

由于 C++是一种功能丰富、强大的语言，因此有许多不同的方法可以建立垃圾回收器。一个明显但受限制的方法就是基于类建立一个垃圾回收器，想要使用垃圾回收的类可以从这个类继承。这样做可以使得一个类接一个类地实现垃圾回收。但遗憾的是，这种解决方案太有限而不能令人满意。

更好的解决方案是建立一个可以被任何动态分配的对象使用的垃圾回收器。为了提供这种

解决方案，垃圾回收器必须满足以下要求：

(1) 与内建的、C++提供的手工方法共存。

(2) 不要破坏任何预先存在的代码。更重要的是，不应该对现有的代码造成任何影响。

(3) 透明地运行，从而利用垃圾回收的分配与不使用这个方案的运行方式相似。

(4) 类似于 C++内建的方法，使用 new 来分配内存。

(5) 对所有的数据类型都有效，包括内建的类型，如 int 和 double 类型。

(6) 易于使用。

总之，垃圾回收系统必须以非常接近 C++已经使用的机制和语法来动态分配内存，并且不能影响现有的代码。乍看上去，这好像是一个令人生畏的任务，但事实并非如此。

理解这个问题

在建立垃圾回收器的时候，面临的关键问题是如何知道某一块内存不再使用。为了理解这个问题，考虑下面的序列：

```
int *p;
p = new int(99);
p = new int(100);
```

在此动态分配了两个 int 对象。第一个对象包含的值为 99，指向这个值的指针存储在 p 中。然后，分配了一个值为 100 的整型数据，其地址也存储在 p 中，从而改写了第一个地址。在此，p(或者其他对象)不再指向 int(99)的那块内存，从而可以释放这块内存。问题是，垃圾回收器如何知道 p 或者其他的对象都不指向 int(99)了呢？

对这个问题稍做变动：

```
int *p, *q;
p = new int(99);
q = p; // now, q points to same memory as p
p = new int(100);
```

在此情况下，q 指向了原先为 p 分配的那块内存。即使 p 指向了不同的内存块，原先指向的内存也不能被释放，因为 q 仍然在使用它。问题是：垃圾回收器如何知道这一事实呢?对这个问题的准确回答取决于垃圾回收所使用的算法。

2.3 选择垃圾回收的算法

在为 C++实现垃圾回收器之前，有必要确定垃圾回收使用的算法。垃圾回收的内容很多，很多年来一直是学术研究的焦点。因为它提出了一个吸引人的问题，对于这个问题有多种解决方案，并且设计了许多不同的垃圾回收算法。详细地验证每一种方法远远超出了本书讨论的范围。然而，在此有 3 种典型方法：引用计数、标记并清除，以及复制。在确定要选择哪种方法之前，有必要先浏览一下这 3 种算法。

2.3.1 引用计数

在引用计数中，每一块动态分配的内存都与一个引用计数相关。这个计数在每次对内存的引用增加的时候增 1，在取消对内存的引用时减 1。用 C++的术语来说，这意味着每次将一个指针指向一块已分配内存的时候，与内存相关的引用计数增 1。当这个指针指向其他位置的时候，引用计数减 1。当引用计数下降为 0 的时候，内存不再被使用，从而可以释放。

引用计数的最大优点是其简单性——易于理解并实现。另外，它的位置不受堆结构的影响，因为引用计数不依赖于对象的物理位置。引用计数增加了每个指针操作的开销，但是回收阶段的开销相对较低。其主要的缺点是循环的引用阻止了其他不再使用的内存的释放。当两个对象互相指向对方的时候(无论是直接的还是间接的)，就会发生循环引用。在此情况下，对象的引用计数永不为 0。为了解决循环引用的问题，设计了一些解决方案，但是这些方案都会增加复杂程度和/或开销。

2.3.2 标记并清除

标记并清除涉及到两个阶段。在第一个阶段，堆中的所有对象都被设置为未标记状态。然后，可以由程序变量直接或者间接访问的所有对象都被标记为“正在使用”。在第二个阶段，扫描所有已分配的内存(也就是说，进行了内存的清除)，会释放所有未标记的元素。

标记并清除有两个主要优点。首先，它很容易处理循环引用。其次，在回收之前，它实际上没有增加运行时开销。它也有两个主要缺点。首先，由于在回收的时候必须扫描整个堆，因此回收垃圾可能会花费较多的时间。因此，对于某些程序，垃圾回收可能会导致程序运行效率低下。其次，尽管标记并清除在概念上很简单，但是要有效地实现它并非易事。

2.3.3 复制

复制算法将自由内存分到两个空间中。一个是活动空间(持有当前的堆)，一个是空闲空间。在垃圾回收期间，活动空间中正在使用的对象被确认，并复制到空闲空间中。然后，两个空间的角色反转，空闲空间变为活动空间，活动空间变为空闲空间。复制提供了复制过程中压缩堆的优点。它的缺点是在某个时刻只允许使用一半的自由内存。

2.3.4 采用哪种算法

三种垃圾回收的经典方法都有各自的优缺点，好像很难做出选择。然而，考虑前面列举出的限制，就会得出明显的选择：引用计数。最重要的是，引用计数可以很容易地应用于现有的 C++动态分配系统上。其次，它可以以一种直接的方式来实现，而不会影响现有的代码。第三，它不需要堆的任何特定的组织或者结构，从而不会影响 C++提供的标准分配系统。

使用引用计数的一个缺点是很难处理循环引用。这对于许多程序而言，并不是一个问题，因为有意的循环引用并不常用，并且可以避免。(即使我们所说的循环，如循环队列，也不一定要用到循环指针引用)。当然，某些情况下需要使用循环引用。也可能建立了循环引用，而您并不知道，特别是使用第三方库的时候。因此，垃圾回收器必须提供某种方法来适度地处理循环引用。

为了处理循环引用问题，本章开发的垃圾回收器在程序退出的时候，释放任何已经分配的内存。这将确保涉及到循环引用的对象被释放，并且调用它们的析构函数。通常在程序结束的

时候，不应该再有已分配的对象，理解这一点很重要。对于涉及到循环引用而不能被释放的对象，这种机制是显式的。(您可能会试验用其他的方法处理循环引用。这是一个有趣的挑战)。

2.3.5 实现垃圾回收器

为了实现引用计数的垃圾回收器，必须有某种方法来跟踪指向每块动态分配内存的指针的数量。问题在于，C++没有内建的机制来确保一个对象知道其他的对象何时指向它。幸运的是，在此有一个解决方案：可以建立一个新的支持垃圾回收的指针类型。这就是本章的垃圾回收器采用的方法。

为了支持垃圾回收，新的指针类型必须做三件事情。第一，它必须为使用中的动态分配的对象维护一个引用计数的链表。第二，它必须跟踪所有的指针运算符，每次某个指针指向一个对象时，都要使这个对象的引用计数增 1；每次某个指针重新指向其他对象的时候，都要使这个对象的引用计数减 1。第三，它必须回收那些引用计数降为 0 的对象。除了支持垃圾回收之外，这个新指针类型与普通的指针看起来一样。例如，它支持所有的指针运算，如*和->运算。

垃圾回收指针类型的建立不仅以一种简便的方法实现垃圾回收器，而且还满足不影响原始的 C++动态分配系统的限制。当需要垃圾回收的时候，使用支持垃圾回收的指针。当不需要垃圾回收的时候，使用普通的 C++指针。因此，这两种类型的指针在同一程序内都可以使用。

2.3.6 是否使用多线程

在设计 C++的垃圾回收器时，另一个考虑是应该使用单线程还是多线程。也就是说，是否应该把垃圾回收器设计为一个后台进程，在它自己的线程内运行，并且在 CPU 时间允许时回收垃圾。或者，这个垃圾回收器在使用它的进程的相同线程中运行，当满足某种程序条件的时候回收垃圾。两种方法各有优缺点。

建立多线程垃圾回收器的主要优点是效率。垃圾可以在 CPU 空闲时被回收。其缺点是，C++没有提供内建的多线程支持。这意味着为了支持多任务，任何多线程方法都依赖于操作系统是否支持多任务。这使得代码不可移植。

使用单线程垃圾回收器的主要优点是代码可以移植。在不支持多线程或者支持多线程的代价很高的情况下使用。主要缺点是当垃圾回收发生时，程序的其他部分会停止运行。

在本章使用了单线程的方法，因为在所有的 C++环境中，它都可以使用。因此，本书所有的读者都可以使用它。然而，对于那些想要使用多线程解决方案的用户，第 3 章给出了一个示例，来处理在 Windows 环境下成功地实现多线程 C++程序所需要的技术。

2.3.7 何时回收垃圾

在实现垃圾回收器之前，需要回答的最后一个问题是：什么时候开始垃圾回收？对于多线程的垃圾回收器这不成问题，因为它可以作为后台任务连续运行，并且在 CPU 空闲的时候回收垃圾。然而对于单线程的垃圾回收器，如本章开发的这个回收器，为了回收垃圾，必须停止运行程序的其他部分。

实际上，只有在有足够的理由(如内存持续降低)的时候，才会进行垃圾回收。有两个原因使得这样做有意义。首先，通过某种垃圾回收算法，如标记并清除，如果不实际执行回收，就没有办法知道不再使用的某块内存。(也就是说，如果不实际回收垃圾，某些时候没有办法知道它的存在)，其次，回收垃圾是一个耗时的过程，在不需要的时候不应该执行它。

然而，在开始垃圾回收之前等待内存持续降低不适合本章的目的，因为这样做使得几乎不可能演示回收器。为此，本章开发的垃圾回收器将更加频繁地回收垃圾，这样可以比较容易地观察到它的活动。当给回收器编码完成之后，无论什么时候指针超出了作用域，都会回收垃圾。当然，可以很容易地改变这种行为以适应您的应用程序。

最后一点要注意的是：在使用基于引用计数的垃圾回收时，只要不再使用的内存的引用计数降为 0，技术上就可以回收它，而不需要使用一个独立的垃圾回收阶段。然而，这种方法对每个指针操作都加入了额外的开销。本章所使用的方法是，在指向某块内存的指针重定向之后，简单地减小这块内存的引用计数。这样做降低了与指针运算相关的运行时开销，通常人们总是希望它越快越好。

2.3.8 关于 auto_ptr

许多读者都知道，C++定义了一个称为 auto_ptr 的库类。由于 auto_ptr 在超出作用域时，它指向的内存会自动释放，您可能会认为在开发垃圾回收器时，它会有用，或许它建立了这个垃圾回收器的基础。然而事实上并非如此。auto_ptr 类是为 ISO C++标准称为“严格所有权”的概念设计的，其中 auto_ptr“拥有”它指向的对象。这种所有权可以转换到其他的 auto_ptr，但是在任何情况下，总会有某个 auto_ptr 拥有这个对象，直到它被销毁。另外，auto_ptr 只有在初始化时，才能使用对象的地址对其赋值。在此之后，您不能够改变 auto_ptr 指向的内存，除非将一个 auto_ptr 赋值给另一个 auto_ptr。由于 auto_ptr 具有“严格所有权”的特征，在建立垃圾回收器时，它没有实际的用处，从而也不会在本书中的垃圾回收器中使用。

2.4 一个简单的 C++垃圾回收器

在此给出了一个完整的垃圾回收器。如前所述，这个垃圾回收器通过建立一个新的指针类型来运行，这个新的指针类型基于引用计数，为垃圾回收提供内在的支持。这个垃圾回收器是单线程的，这意味着它具有很好的移植性，并且不依赖于(或者对其做一些假定)执行环境。这些代码存储在文件 gc.h 中。

当您阅读代码时，需要注意两件事情。首先，大多数成员函数都相当简短，为了提高效率，都在各自的类中定义。回想一下，在类中定义的函数都自动成为内联函数，这样做消除了函数调用的开销。仅有少数函数比较长，因而需要在类的外部定义它们。

其次，注意文件头的注释。如果您想要观察垃圾回收器是如何工作的，只需要简单地通过定义 DISPLAY 宏来开启显示选项。对于普通的使用，不需要定义 DISPLAY。

```
// A single-threaded garbage collector.

#include <iostream>
#include <list>
#include <typeinfo>
#include <cstdlib>

using namespace std;

// To watch the action of the garbage collector, define DISPLAY.
```

```
// #define DISPLAY

// Exception thrown when an attempt is made to
// use an Iter that exceeds the range of the
// underlying object.
//
class OutOfRangeExc {
  // Add functionality if needed by your application.
};

// An iterator-like class for cycling through arrays
// that are pointed to by GCPtrs. Iter pointers
// ** do not ** participate in or affect garbage
// collection. Thus, an Iter pointing to
// some object does not prevent that object
// from being recycled.
//
template <class T> class Iter {
  T *ptr; // current pointer value
  T *end; // points to element one past end
  T *begin; // points to start of allocated array
  unsigned length; // length of sequence
public:

  Iter() {
    ptr = end = begin = NULL;
    length = 0;
  }

  Iter(T *p, T *first, T *last) {
    ptr = p;
    end = last;
    begin = first;
    length = last - first;
  }

  // Return length of sequence to which this
  // Iter points.
  unsigned size() { return length; }

  // Return value pointed to by ptr.
  // Do not allow out-of-bounds access.
  T &operator*() {
    if( (ptr >= end) || (ptr < begin) )
      throw OutOfRangeExc();
    return *ptr;
  }

  // Return address contained in ptr.
  // Do not allow out-of-bounds access.
  T *operator->() {
    if( (ptr >= end) || (ptr < begin) )
```

```
    throw OutOfRangeExc();
  return ptr;
}

// Prefix ++.
Iter operator++() {
  ptr++;
  return *this;
}

// Prefix --.
Iter operator--() {
  ptr--;
  return *this;
}

// Postfix ++.
Iter operator++(int notused) {
  T *tmp = ptr;

  ptr++;
  return Iter<T>(tmp, begin, end);
}

// Postfix --.
Iter operator--(int notused) {
  T *tmp = ptr;

  ptr--;
  return Iter<T>(tmp, begin, end);
}

// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
  if( (i < 0) || (i >= (end-begin)) )
    throw OutOfRangeExc();
  return ptr[i];
}

// Define the relational operators.
bool operator==(Iter op2) {
  return ptr == op2.ptr;
}

bool operator!=(Iter op2) {
  return ptr != op2.ptr;
}

bool operator<(Iter op2) {
  return ptr < op2.ptr;
```

```
  }

  bool operator<=(Iter op2) {
    return ptr <= op2.ptr;
  }

  bool operator>(Iter op2) {
    return ptr > op2.ptr;
  }

  bool cperator>=(Iter op2) {
    return ptr >= op2.ptr;
  }

  // Subtract an integer from an Iter.
  Iter operator-(int n) {
    ptr -= n;
    return *this;
  }

  // Add an integer to an Iter.
  Iter operator+(int n) {
    ptr += n;
    return *this;
  }

  // Return number of elements between two Iters.
  int operator-(Iter<T> &itr2) {
    return ptr - itr2.ptr;
  }

};


// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:
  unsigned refcount; // current reference count

  T *memPtr; // pointer to allocated memory

  /* isArray is true if memPtr points
     to an allocated array. It is false
     otherwise. */
  bool isArray; // true if pointing to array

  /* If memPtr is pointing to an allocated
     array, then arraySize contains its size */
  unsigned arraySize; // size of array
```

```
  // Here, mPtr points to the allocated memory.
  // If this is an array, then size specifies
  // the size of the array.
  GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
      isArray = true;
    else
      isArray = false;

    arraySize = size;
  }
};

// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                const GCInfo<T> &ob2) {
  return (ob1.memPtr == ob2.memPtr);
}


// GCPtr implements a pointer type that uses
// garbage collection to release unused memory.
// A GCPtr must only be used to point to memory
// that was dynamically allocated using new.
// When used to refer to an allocated array,
// specify the array size.
//
template <class T, int size=0> class GCPtr {

  // gclist maintains the garbage collection list.
  static list<GCInfo<T> > gclist;

  // addr points to the allocated memory to which
  // this GCPtr pointer currently points.
  T *addr;

  /* isArray is true if this GCPtr points
     to an allocated array. It is false
     otherwise. */
  bool isArray; // true if pointing to array

  // If this GCPtr is pointing to an allocated
  // array, then arraySize contains its size.
  unsigned arraySize; // size of the array
  static bool first; // true when first GCPtr is created

  // Return an iterator to pointer info in gclist.
  typename list<GCInfo<T> >::iterator findPtrInfo(T *ptr);
```

```
public:

  // Define an iterator type for GCPtr<T>.
  typedef Iter<T> GCiterator;

  // Construct both initialized and uninitialized objects.
  GCPtr(T *t=NULL) {

    // Register shutdown() as an exit function.
    if(first) atexit(shutdown);
    first = false;

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(t);

    // If t is already in gclist, then
    // increment its reference count.
    // Otherwise, add it to the list.
    if(p != gclist.end())
      p->refcount++; // increment ref count
    else {
      // Create and store this entry.
      GCInfo<T> gcObj(t, size);
      gclist.push_front(gcObj);
    }

    addr = t;
    arraySize = size;
    if(size > 0) isArray = true;
    else isArray = false;
    #ifdef DISPLAY
      cout << "Constructing GCPtr. ";
      if(isArray)
        cout << " Size is " << arraySize << endl;
      else
        cout << endl;
    #endif
  }

  // Copy constructor.
  GCPtr(const GCPtr &ob) {
    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(ob.addr);
    p->refcount++; // increment ref count

    addr = ob.addr;
    arraySize = ob.arraySize;
    if(arraySize > 0) isArray = true;
    else isArray = false;
    #ifdef DISPLAY
```

```
    cout << "Constructing copy.";
    if(isArray)
      cout << " Size is " << arraySize << endl;
    else
      cout << endl;
  #endif
}

// Destructor for GCPTr.
~GCPtr();

// Collect garbage. Returns true if at least
// one object was freed.
static bool collect();

// Overload assignment of pointer to GCPtr.
T *operator=(T *t);

// Overload assignment of GCPtr to GCPtr.
GCPtr &operator=(GCPtr &rv);

// Return a reference to the object pointed
// to by this GCPtr.
T &operator*() {
  return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
  return addr[i];
}

// Conversion function to T *.
operator T *() { return addr; }

// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
  int size;

  if(isArray) size = arraySize;
  else size = 1;

  return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
  int size;
```

```
    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr + size, addr, addr + size);
  }

  // Return the size of gclist for this type
  // of GCPtr.
  static int gclistSize() { return gclist.size(); }

  // A utility function that displays gclist.
  static void showlist();

  // Clear gclist when program exits.
  static void shutdown();
};

// Creates storage for the static variables
template <class T, int size>
  list<GCInfo<T> > GCPtr<T, size>::gclist;

template <class T, int size>
  bool GCPtr<T, size>::first = true;

// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~GCPtr() {
  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(addr);
  if(p->refcount) p->refcount--; // decrement ref count

  #ifdef DISPLAY
    cout << "GCPtr going out of scope.\n";
  #endif

  // Collect garbage when a pointer goes out of scope.
  collect();

  // For real use, you might want to collect
  // unused memory less frequently, such as after
  // gclist has reached a certain size, after a
  // certain number of GCPtrs have gone out of scope,
  // or when memory is low.
}

// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
bool GCPtr<T, size>::collect() {
  bool memfreed = false;
```

```
#ifdef DISPLAY
  cout << "Before garbage collection for ";
  showlist();
#endif

list<GCInfo<T> >::iterator p;
do {

  // Scan gclist looking for unreferenced pointers.
  for(p = gclist.begin(); p != gclist.end(); p++) {
    // If in-use, skip.
    if(p->refcount > 0) continue;

    memfreed = true;

    // Remove unused entry from gclist.
    gclist.remove(*p);
    // Free memory unless the GCPtr is null.
    if(p->memPtr) {
      if(p->isArray) {
        #ifdef DISPLAY
          cout << "Deleting array of size "
               << p->arraySize << endl;
        #endif
        delete[] p->memPtr; // delete array
        }
        else {
          #ifdef DISPLAY
            cout << "Deleting: "
                 << *(T *) p->memPtr << "\n";
          #endif
          delete p->memPtr; // delete single element
        }
      }

      // Restart the search.
      break;
    }

  } while(p != gclist.end());

  #ifdef DISPLAY
    cout << "After garbage collection for ";
    showlist();
  #endif

  return memfreed;
}

// Overload assignment of pointer to GCPtr.
template <class T, int size>
```

```
T * GCPtr<T, size>::operator=(T *t) {
  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;

  // Next, if the new address is already
  // existent in the system, increment its
  // count. Otherwise, create a new entry
  // for gclist.
  p = findPtrInfo(t);
  if(p != gclist.end())
    p->refcount++;
  else {
    // Create and store this entry.
    GCInfo<T> gcObj(t, size);
    gclist.push_front(gcObj);
  }

  addr = t; // store the address.

  return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;

  // Next, increment the reference count of
  // the new address.
  p = findPtrInfo(rv.addr);
  p->refcount++; // increment ref count

  addr = rv.addr;// store the address.

  return rv;
}

// A utility function that displays gclist.
template <class T, int size>
void GCPtr<T, size>::showlist() {
  list<GCInfo<T> >::iterator p;

  cout << "gclist<" << typeid(T).name() << ", "
```

```
        << size << ">:\n";
    cout << "memPtr refcount value\n";

    if(gclist.begin() == gclist.end()) {
    cout << " -- Empty --\n\n";
    return;
  }

  for(p = gclist.begin(); p != gclist.end(); p++) {
    cout << "[" << (void *)p->memPtr << "]"
         << " " << p->refcount << " ";
    if(p->memPtr) cout << " " << *p->memPtr;
    else cout << " ---";
    cout << endl;
  }
  cout << endl;
}

// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
  GCPtr<T, size>::findPtrInfo(T *ptr) {

  list<GCInfo<T> >::iterator p;

  // Find ptr in gclist.
  for(p = gclist.begin(); p != gclist.end(); p++)
    if(p->memPtr == ptr)
      return p;

  return p;
}

// Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

  if(gclistSize() == 0) return; // list is empty

  list<GCInfo<T> >::iterator p;

  for(p = gclist.begin(); p != gclist.end(); p++) {
    // Set all reference counts to zero
    p->refcount = 0;
  }

  #ifdef DISPLAY
    cout << "Before collecting for shutdown() for "
         << typeid(T).name() << "\n";
  #endif

  collect();
```

```
  #ifdef DISPLAY
    cout << "After collecting for shutdown() for "
         << typeid(T).name() << "\n";
  #endif
}
```

垃圾回收器类概述

垃圾回收器使用了4个类：GCPtr、GCInfo、Iter以及OutOfRangeExc。在详细分析这些代码之前，有必要先理解每一个类所扮演的角色。

1. GCPtr

垃圾回收器的核心是GCPtr类，它实现了垃圾回收指针。GCPtr维护了一个链表，这个链表与GCPtr分配的每一块内存的引用计数相关。下面是它的运行方式。每次GCPtr指向一块内存时，这块内存的引用计数增1。如果GCPtr在分配之前指向不同的内存块，那么这块内存的引用计数减1。向某一块内存加入指针增加了这块内存的引用计数，移除指针减少了这块内存的引用计数。每一次GCPtr超出作用域，当前它所指向的内存的引用计数都会减小。当引用计数降为0时，这一块内存就被释放了。

GCPtr是一个重载了“*”、“->”指针运算符以及数组索引运算符[]的模板类。因此，GCPtr建立了一个新的指针类型，并且将它整合到C++的程序设计环境中。从而使得能够以类似于使用普通C++指针的方式来使用GCPtr。然而在本章，为了清楚起见，GCPtr没有重载++、--以及其他为指针定义的算术运算符。因此，除了通过赋值之外，您不能改变GCPtr对象所指向的地址。这好像是一个很大的限制，但事实上并非如此，因为Iter类提供了这些操作。

为了说明问题，无论何时GCPtr对象超出作用域，垃圾回收器都会运行。这时会扫描垃圾回收链表，所有引用计数为0的内存都会被释放，即使与超出作用域的GCPtr无关的内存也是如此。如果您在前面就需要回收内存，就可以显式地要求垃圾回收。

2. GCInfo

如前所述，GCPtr维护了一个将引用计数与已分配内存链接起来的链表。这个链表中的每一项都封装在一个GCInfo类型的对象之中。GCInfo在它的refcount字段中存储了引用计数，在memPtr字段中保存了这块内存的指针。因此，GCInfo对象将引用计数与已分配内存绑定在一起。

GCInfo定义了两个其他字段：isArray和arraySize。如果memPtr指向了一个已分配的数组，那么isArray成员就为true，这个数组的长度就存储在它的arraySize字段中。

3. Iter类

前面已经说过，GCPtr对象允许使用普通的指针运算符“*”和“->”来访问它所指向的内存，但是它不支持指针运算。为了处理需要执行这些指针运算的情况，可以使用Iter类型的对象。Iter是一个功能上类似于STL迭代器的模板类，它定义了所有的指针操作，包括指针运算。Iter的主要用处是遍历动态分配的数组元素。它还提供边界检查。可以通过调用GCPtr的begin()或者end()来获取Iter对象，其运行方式与STL中的迭代器非常类似。

尽管 Iter 与 STL 中的迭代器 iterator 类型非常相似，但是它们并不相同，理解这一点很重要，它们不能互相替代使用。

4. OutOfRangeExc

如果 Iter 试图访问已分配内存范围之外的内存，则会抛出 OutOfRangeExc 异常。本章中 OutOfRangeExc 不包含成员。它只是一个可以抛出的类型。然而，如果您的应用程序需要，就可以随意向这个类中添加其他功能。

2.5 详细讨论 GCPtr

GCPtr 是垃圾回收器的核心。它实现了一个新的指针类型来保存堆中已分配对象的引用计数。它还提供了垃圾回收的功能来回收不再使用的内存。

GCPtr 是一个模板类，其声明如下：

```
template <class T, int size=0> class GCPtr {
```

GCPtr 要求您指定将被指向的数据的类型，这个类型将取代通用类型 T。如果被分配的是一个数组，就必须在 size 参数中指定数组的大小。否则，size 默认为 0，这说明它指向一个单独的对象。在此给出两个示例。

```
GCPtr<int> p; // declare a pointer to a single integer
GCPtr<int, 5> ap; // declare a pointer to an array of 5 integers
```

在此，p 可以指向一个 int 类型的对象，ap 可以指向大小为 5 的 int 数组。

注意在前面的示例中，在指定 GCPtr 对象名称时，没有使用“*”运算符。也就是说，不需要使用这样的语句来建立一个指向 int 的 GCPtr：

```
GCPtr<int> *p; // this creates a pointer to a GCPtr<int> object
```

这个声明建立了一个名为 p 的指向一个 GCPtr<int>对象的普通 C++指针，而不是建立了一个指向 int 的 GCPtr 对象。记住，GCPtr 本身就定义了一个指针类型。

在指定 GCPtr 的类型参数的时候必须谨慎。这个参数指定了 GCPtr 对象可以指向的对象类型。因此，如果您使用了这样的声明：

```
GCPtr<int *> p; // this creates a GCPtr to pointers to ints
```

那就创建了一个指向 int*指针的 GCPtr 对象，而不是指向 int 的 GCPtr 对象。

由于其重要性，在接下来的部分，详细地分析每个 GCPtr 的成员。

2.5.1 GCPtr 的数据成员

GCPtr 声明了如下的数据成员：

```
// gclist maintains the garbage collection list.
static list<GCInfo<T> > gclist;

// addr points to the allocated memory to which
```

```
// this GCPtr pointer currently points.
T *addr;

/* isArray is true if this GCPtr points
   to an allocated array. It is false
   otherwise. */
bool isArray; // true if pointing to array

// If this GCPtr is pointing to an allocated
// array, then arraySize contains its size.
unsigned arraySize; // size of the array

static bool first; // true when first GCPtr is created
```

gclist 字段包含了 GCInfo 对象的链表。(回顾一下，GCInfo 链接了引用计数与已分配的内存块。)垃圾回收器使用这个链表来判断何时不再使用已分配内存。注意 gclist 是 GCPtr 的静态成员。这意味着对于每个特定的指针类型，只会有一个 gclist。例如，所有的 GCPtr<int>类型的指针共享一个链表，所有的 GCPtr<double>类型的指针共享另一个链表。gclist 是 STL list 类的一个实例。使用 STL 极大地简化了 GCPtr 的代码，因为在此不需要创建自己的链表处理函数集。

GCPtr 在 addr 中存储了它所指向的内存的地址。如果 addr 指向一个已分配的数组，那么 isArray 为 true，数组的长度存储在 arraySize 中。

first 字段是一个静态变量，其初始值设置为 true。GCPtr 构造函数使用这个标记来了解何时建立了第一个 GCPtr 对象。在第一个 GCPtr 对象建立之后，first 被设置为 false。它用来注册一个终止函数，这个函数在程序终止时调用，从而关闭垃圾回收器。

2.5.2 函数 findPtrInfo()

GCPtr 声明了一个私有函数：findPtrInfo()。这个函数在 gclist 中查找指定的地址，并且返回该项的一个迭代器。如果没有找到这个地址，就返回这个 gclist 结尾的迭代器。这个函数由 GCPtr 在内部使用，来更新 gclist 内的对象的引用计数。其实现如下：

```
// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
  GCPtr<T, size>::findPtrInfo(T *ptr) {

  list<GCInfo<T> >::iterator p;

  // Find ptr in gclist.
  for(p = gclist.begin(); p != gclist.end(); p++)
    if(p->memPtr == ptr)
      return p;

  return p;
}
```

2.5.3 GCIterator typedef

GCPtr 的公有部分以 Iter<T>的 GCIterator typedef 开始。这个 typedef 与 GCPtr 的每个实例绑定，从而不必在每一次 Iter 需要指定 GCPtr 的版本时指定类型参数。这样做简化了迭代器的声明。例如，为了获得由特定 GCPtr 指向的内存的迭代器，可以使用下面的语句：

```
GCPtr<int>::GCiterator itr;
```

2.5.4 GCPtr 的构造函数

GCPtr 的构造函数如下所示：

```
// Construct both initialized and uninitialized objects.
GCPtr(T *t=NULL) {

// Register shutdown() as an exit function.
if(first) atexit(shutdown);
first = false;
list<GCInfo<T> >::iterator p;

p = findPtrInfo(t);

// If t is already in gclist, then
// increment its reference count.
// Otherwise, add it to the list.
if(p != gclist.end())
  p->refcount++; // increment ref count
else {
  // Create and store this entry.
  GCInfo<T> gcObj(t, size);
  gclist.push_front(gcObj);
}

addr = t;
arraySize = size;
if(size > 0) isArray = true;
else isArray = false;
#ifdef DISPLAY
  cout << "Constructing GCPtr. ";
  if(isArray)
    cout << " Size is " << arraySize << endl;
  else
    cout << endl;
  #endif
}
```

GCPtr()允许创建已初始化的实例和未初始化的实例。如果声明了已初始化的实例，那么 GCPtr 将指向的那块内存就会传递给 t。否则，t 就会是 null。让我们分析检查一下 GCPtr()的操作。

首先，如果 first 为 true，就意味着将要创建第一个 GCPtr 对象。在此情况下，通过调用 atexit()，将 shutdown()注册为终止函数。atexit()函数是 C++标准函数库的一部分，它用来注册在程序结束时调用的函数。此时，shutdown()释放任何由于循环引用而没有释放的内存。

然后对 gclist 进行搜索，查找任何与 t 中的地址匹配的先前存在的条目。如果能够找到，就会增加它的引用计数。如果没有先前存在的条目与 t 匹配，就创建包含这个地址的新 GCInfo 对象，并且将这个对象加入到 gclist 中。

随后 GCPtr()将 addr 设置为 t 指定的地址，并且正确地设置 isArray 和 arraySize 的值。记住，如果您分配了一个数组，那么当声明指向它的 GCPtr 指针时，必须显式地指定这个数组的大小。如果不这么做的话，这块内存就不会被正确地释放，在类对象数组的情况下，析构函数不会被正确地调用。

2.5.5 GCPtr 的析构函数

GCPtr 的析构函数如下：

```
// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~GCPtr() {
  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(addr);
  if(p->refcount) p->refcount--; // decrement ref count

  #ifdef DISPLAY
    cout << "GCPtr going out of scope.\n";
  #endif

  // Collect garbage when a pointer goes out of scope.
  collect();

  // For real use, you might want to collect
  // unused memory less frequently, such as after
  // gclist has reached a certain size, after a
  // certain number of GCPtrs have gone out of scope,
  // or when memory is low.
}
```

每次 GCPtr 超出作用域时，垃圾回收都会运行。这是通过~GCPtr()完成的。首先搜索 gclist，查找与将被销毁的 GCPtr 所指的地址对应的条目。一旦发现，减少其引用计数。随后，~GCPtr()调用 collect()来释放不再使用的内存(也就是其引用计数为 0 的内存)。

正如~GCPtr()结尾的注释所讲的那样，对于实际的应用程序，垃圾回收的频率少于 GCPtr 超出作用域的频率可能会更好。较小频率的回收通常效率更高。如前所述，每次 GCPtr 销毁的时候进行回收对于演示垃圾回收器是有用的，因为这样做清楚地说明了垃圾回收器的运行方式。

2.5.6 回收垃圾函数 collect()

使用 collect()函数进行垃圾回收。其代码如下：

```
// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
```

```
bool GCPtr<T, size>::collect() {
  bool memfreed = false;

  #ifdef DISPLAY
    cout << "Before garbage collection for ";
    showlist();
  #endif

  list<GCInfo<T> >::iterator p;
  do {

    // Scan gclist looking for unreferenced pointers.
    for(p = gclist.begin(); p != gclist.end(); p++) {
      // If in-use, skip.
      if(p->refcount > 0) continue;

      memfreed = true;

      // Remove unused entry from gclist.
      gclist.remove(*p);

      // Free memory unless the GCPtr is null.
      if(p->memPtr) {
        if(p->isArray) {
          #ifdef DISPLAY
            cout << "Deleting array of size "
                 << p->arraySize << endl;
          #endif
          delete[] p->memPtr; // delete array
        }
        else {
          #ifdef DISPLAY
            cout << "Deleting: "
                 << *(T *) p->memPtr << "\n";
          #endif
          delete p->memPtr; // delete single element
        }
      }

      // Restart the search.
      break;
    }

  } while(p != gclist.end());

  #ifdef DISPLAY
    cout << "After garbage collection for ";
    showlist();
  #endif
  return memfreed;
}
```

collect()函数通过扫描 gclist 的内容，查找是否有 refcount 为 0 的条目。当找到这种条目时，通过调用 remove()函数删除它，remove()函数是 STL list 类的一个成员。然后释放与这个条目相关的内存。

回忆一下，在 C++中，单个对象通过 delete 释放，而对象数组通过 delete[]释放。因此，条目的 isArray 字段的值决定了是使用 delete 还是 delete[]来释放内存。这就是您必须为任何指向数组的 GCPtr 指定已分配数组的大小的原因之一：这样做将会使得 isArray 为 true。如果没有正确地设置 isArray，就不可能正确地释放已分配的内存。

尽管垃圾回收的主要目的是为了自动回收不再使用的内存，但是在需要的时候也可以采用手工控制的方法。用户代码可以直接调用 collect()函数进行垃圾回收。注意这个函数声明为 GCPtr 中的静态函数，这意味着不引用任何对象就可以调用它。

例如：

```
GCPtr<int>::collect(); // collect all unused int pointers
```

这会导致 gclist<int>被回收。因为对于每个不同类型的指针，都会有不同的 gclist，所以您需要为每个想要回收的链表调用 collect()。坦白地讲，如果想要更加明确地管理动态分配内存的释放，最好使用 C++提供的手工分配系统。对于特定的情况，如自由内存出乎意料地持续降低时，最好直接调用 collect()。

2.5.7 重载赋值运算符

GCPtr 重载了两个 operator=()：一个是为了将新的地址赋给 GCPtr，另一个是为了将一个 GCPtr 指针赋给另一个 GCPtr。下面给出了这两个版本：

```
// Overload assignment of pointer to GCPtr.
template <class T, int size>
T * GCPtr<T, size>::operator=(T *t) {
  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;
  // Next, if the new address is already
  // existent in the system, increment its
  // count. Otherwise, create a new entry
  // for gclist.
  p = findPtrInfo(t);
  if(p != gclist.end()}
    p->refcount++;
  else {
    // Create and store this entry.
    GCInfo<T> gcObj(t, size);
    gclist.push_front(gcObj);
  }

  addr = t; // store the address.
```

```
  return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;

  // Next, increment the reference count of
  // the new address.
  p = findPtrInfo(rv.addr);
  p->refcount++; // increment ref count

  addr = rv.addr;// store the address.

  return rv;
}
```

第一个重载的 operator=()处理了 GCPtr 指针在左、地址在右的赋值情况。例如，它可以处理如下的情况：

```
GCPtr<int> p;
// ...
p = new int(18);
```

在此，new 返回的地址赋给了 p。当这个操作发生时，就会调用 operator=(T*t)，并且将新地址传递给 t。首先，找到 gclist 中关于当前指向的内存的条目，并且其引用计数减 1。随后，搜索 gclist 查找新地址。如果能够找到的话，它的引用计数就会增 1。否则，就会为这个新地址建立新 GCInfo 对象，并且将这个对象加入到 gclist。最后，在调用对象的 addr 中存储这个新地址，并且返回这个地址。

赋值运算符的第二个重载，operator=(GCPtr&rv)，处理下面的赋值类型：

```
GCPtr<int> p;
GCPtr<int> q;
// ...
p = new int(88);
q = p;
```

在此，p 和 q 都是 GCPtr 指针，p 的值赋给了 q。这个版本与另一个版本的赋值运算符的运行方式很类似。首先，在 gclist 中找到了当前指向的内存的条目，其引用计数减 1。随后，搜索 gclist 来查找新地址，这个地址包含在 rv.addr 中，并且它的引用计数增 1。然后调用对象的 addr 字段被设置为包含在 rv.addr 中的地址。最后，返回右边的对象。从而允许进行连续的赋值，例如：

```
p = q = w = z;
```

关于赋值运算符的工作方式，还有很重要的一点需要说明。在本章的前面部分已经提到过，在技术上，可以在内存的引用计数减为 0 的时候，立即回收这块内存，但是这样做对每个指针操作都加入了额外的开销。这就是在赋值运算符的重载中，左操作数先前指向的内存的引用计数只是简单地减小，而没有执行另外动作的原因。因此，避免了与实际的内存释放以及对 gclist 的维护相关的开销。这些动作被推迟到垃圾回收器运行的时刻。这种方法使得使用了 GCPtr 的代码运行效率较高。同时还使得垃圾回收在尽量不影响系统性能的情况下进行。

2.5.8 GCPtr 的复制构造函数

由于必须通过跟踪每一个指针来分配内存，因此不能使用系统默认的复制构造函数。取而代之的是 GCPtr 必须定义它自己的复制构造函数，如下所示：

```
// Copy constructor.
GCPtr(const GCPtr &ob) {
  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(ob.addr);
  p->refcount++; // increment ref count

  addr = ob.addr;
  arraySize = ob.arraySize;
  if(arraySize > 0) isArray = true;
  else isArray = false;
  #ifdef DISPLAY
    cout << "Constructing copy.";
    if(isArray)
      cout << " Size is " << arraySize << endl;
    else
      cout << endl;
  #endif
}
```

当需要对象的副本时，如对象作为参数传递给函数时、对象作为函数的返回值时、或者使用一个对象来初始化另一个对象时，就会调用类的复制构造函数。GCPtr 的复制构造函数复制保存在原始对象中的信息。并且它还增加了原始对象所指向的内存的引用计数。当这个副本超出作用域时，引用计数会减少。

实际上，通常并不需要由复制构造函数所执行的额外的工作，因为重载的赋值运算符在大多数情况下会恰当地维护垃圾回收链表。然而，在少数情况下还是需要复制构造函数，如在函数中分配内存并且返回指向这块内存的 GCPtr 的时候。

2.5.9 指针运算符和转换函数

因为 GCPtr 是一个指针类型，所以它必须重载指针运算符*和->，还有索引运算符[ ]。这些工作由这里显示的函数完成。尽管正是这些被重载的运算符的功能使得建立新的指针类型成为可能，但在此它们却非常简单：

```
// Return a reference to the object pointed
// to by this GCPtr.
```

```
T &operator*() {
  return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
  return addr[i];
}
```

operator*()函数返回一个对象的引用，由调用 GCPtr 的 addr 字段指向这个对象。operator->()返回了包含在 addr 中的地址，operator[]返回了对索引指定的元素的引用。operator[]只能在指向已分配数组的 GCPtr 上使用。

如前所述，在此不支持指针运算。例如，没有为 GCPtr 重载“++”或者“--”运算符。这样做的原因是垃圾回收机制假定 GCPtr 指向已分配内存的开始。如果 GCPtr 可以增加，那么当这个指针被垃圾回收的时候，delete 使用的地址就会变得无效。

如果需要执行涉及到指针运算的操作，将有两个选择。首先，如果 GCPtr 指向一个已分配的数组，那么可以建立 Iter 来遍历数组。稍后将讲述这种方法。其次，可以使用 GCPtr 定义的 T*转换函数将 GCPtr 转换成一个普通的指针。这个函数如下所示：

```
// Conversion function to T *.
operator T*() { return addr; }
```

这个函数返回一个普通指针，这个指针指向存储在 addr 中的地址。这个函数可以如下使用：

```
GCPtr<double> gcp = new double(99.2);
double *p;

p = gcp; // now, p points to same memory as gcp
p++; // because p is a normal pointer, it can be incremented
```

在前面的示例中，由于 p 是一个普通指针，从而可以使用对其他指针使用的任何方法来操作它。当然，这个操作是否产生了有意义的结果，取决于您的应用程序。

对 T*的转换最主要的优点是，当与现有的、需要这种指针的代码一起工作时，可以使用 GCPtr 来取代普通的 C++指针。例如，考虑下面的代码：

```
GCPtr<char> str = new char[80];
strcpy(str, "this is a test");
cout << str << endl;
```

在此，str 是一个指向 char 的 GCPtr 指针，在 strcpy()的调用中用到了它。由于 strcpy()需要 char*类型的参数，因此自动调用 GCPtr 中对 T*的转换，因为在此情况下，T 是 char。在 cout 语句中使用 str 的时候，也会自动调用相同的转换。因此，转换函数使得 GCPtr 与现有的 C++函数和类无缝地整合在一起。

记住，这种转换返回的 T*指针既不参与也不影响垃圾回收。因此，即使常规的 C++指针

仍然指向它，已分配的内存也可以被释放。因此，应该明智且尽量少使用对 T*的转换。

2.5.10 begin()和 end()函数

下面所示的 begin()和 end()函数与 STL 中对应的函数很相似：

```
// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
  int size;

  if(isArray) size = arraySize;
  else size = 1;

  return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
  int size;

  if(isArray) size = arraySize;
  else size = 1;

  return Iter<T>(addr + size, addr, addr + size);
}
```

begin()函数返回了一个 Iter，这个 Iter 指向 addr 指向的已分配数组的起始位置。end()函数返回一个 Iter，这个 Iter 指向这个数组结尾的后一个位置。尽管在此并没有阻止指向单个对象的 GCPtr 调用这两个函数，但它们的目的是为了支持对已分配数组的操作。(为单个对象获取一个 Iter 并没有坏处，只是没有意义)。

2.5.11 shutdown()函数

如果一个程序建立了 GCPtr 的循环引用，那么当这个程序结束时，仍然存在需要被释放的动态分配的对象。这很重要，因为这些对象可能有需要调用的析构函数。shutdown()函数用来处理这种情况。当建立第一个 GCPtr 时，这个函数就由 atexit()注册，如前所述。这意味着当程序结束时，会调用这个函数。

shutdown()函数如下所示：

```
 // Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

  if(gclistSize() == 0) return; // list is empty

  list<GCInfo<T> >::iterator p;

  for(p = gclist.begin(); p != gclist.end(); p++) {
  // Set all reference counts to zero
  p->refcount = 0;
```

```
}

#ifdef DISPLAY
  cout << "Before collecting for shutdown() for "
       << typeid(T).name() << "\n";
#endif

collect();

#ifdef DISPLAY
  cout << "After collecting for shutdown() for "
       << typeid(T).name() << "\n";
  #endif
}
```

首先，如果这个链表是空的(通常是这样的)，shutdown()只是简单地返回。否则，将仍然存在于 gclist 中的条目的引用计数设置为 0，然后调用 collect()。collect()释放任何引用计数为 0 的对象。因此，将引用计数设置为 0 可以确保释放所有对象。

2.5.12 两个实用函数

最后，GCPtr 定义了两个实用函数。第一个是 gclistSize()函数，这个函数返回当前 gclist 中拥有的条目数量。第二个是 showlist()函数，这个函数显示 gclist 的内容。这两个函数对于实现垃圾回收指针类型都不是必须的，但是，如果您想要观察垃圾回收器的运行情况，它们就有用了。

2.6 GCInfo

在 gclist 中的垃圾回收列表包含了 GCInfo 类型的对象。GCInfo 类如下所示：

```
// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:
  unsigned refcount; // current reference count

  T *memPtr; // pointer to allocated memory

  /* isArray is true if memPtr points
     to an allocated array. It is false
     otherwise. */
  bool isArray; // true if pointing to array

  /* If memPtr is pointing to an allocated
     array, then arraySize contains its size */
  unsigned arraySize; // size of array

  // Here, mPtr points to the allocated memory.
```

```
  // If this is an array, then size specifies
  // the size of the array.
  GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
      isArray = true;
    else
      isArray = false;

    arraySize = size;
  }
};
```

如前所述，每一个 GCInfo 对象都在 memPtr 中存储了一个指向已分配内存的指针，并在 refcount 中存储了与这块内存相关的引用计数。如果 memPtr 指向的内存包含了数组，那么当 GCInfo 对象建立时，必须指定这个数组的长度。在此情况下，isArray 被设置为 true，数组的长度将存储在 arraySize 中。

GCInfo 对象存储在一个 STL list 中。为了能够搜索这个链表，必须定义 operator==(), 如下所示：

```
// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                const GCInfo<T> &ob2) {
  return (ob1.memPtr == ob2.memPtr);
}
```

只有在两个对象的 memPtr 字段相同时，它们才会相等。为了使得 GCInfo 存储在 STL 列表中，可能还需要重载其他运算符，这取决于您所使用的编译器。

2.7 Iter

Iter 类实现了一个类似于迭代器的对象，可以使用它遍历已分配数组的元素。在技术上 Iter 并非必须存在，因为 GCPtr 可以转换为一个基本类型的普通指针，但是 Iter 具有两个优点。首先，它可以使用遍历 STL 容器内容的方式来遍历已分配的数组。因此，使用 Iter 的语法是为人熟知的。其次，Iter 不允许越界访问。因此，相对于普通的指针，使用 Iter 比较安全。然而需要知道，Iter 没有参与垃圾回收。因此，如果将 Iter 作为基础的 GCPtr 超出了作用域，那么 GCPtr 所指向的内存就会被释放，无论 Iter 是否还需要它。

Iter 是一个模板类，其定义如下：

```
template <class T> class Iter {
```

Iter 指向的数据类型通过 T 传递。

Iter 定义了如下的实例变量：

```
T *ptr;    // current pointer value
```

```
T *end;    // points to element one past end
T *begin;  // points to start of allocated array
unsigned length; // length of sequence
```

Iter 当前所指的地址保存在 ptr 中。数组的起始地址保存在 begin 中，数组结尾的后一个元素的地址保存在 end 中。动态数组的长度保存在 length 中。

Iter 定义了这里所显示的两个构造函数。第一个是默认的构造函数。第二个建立了一个 Iter，并给出了 ptr 的初始值，另外还有指向数组起始位置和结束位置的指针。

```
Iter() {
ptr = end = begin = NULL;
  length = 0;
}

Iter(T *p, T *first, T *last) {
  ptr = p;
  end = last;
  begin = first;
  length = last - first;
}
```

为了被本章给出的垃圾回收器代码使用，ptr 的初始值总是等于 begin。然而，可以随意地建立 Iter，在其中 ptr 的初始值是不同的值。

为了使得 Iter 像普通的指针那样，它重载了*和->指针运算符，以及数组索引运算符[ ]，如下所示：

```
// Return value pointed to by ptr.
// Do not allow out-of-bounds access.
T &operator*() {
  if( (ptr >= end) || (ptr < begin) )
    throw OutOfRangeExc();
  return *ptr;
}

// Return address contained in ptr.
// Do not allow out-of-bounds access.
T *operator->() {
  if( (ptr >= end) || (ptr < begin) )
    throw OutOfRangeExc();
  return ptr;
}
// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
  if( (i < 0) || (i >= (end-begin)) )
    throw OutOfRangeExc();
  return ptr[i];
}
```

*运算符返回动态数组中当前指向元素的引用。->运算符返回当前指向元素的地址。[ ]返回

指定索引处的元素的引用。注意这些操作都不允许越界访问。如果尝试这样做，就会抛出OutOfRangeExc异常。

Iter定义了不同的指针算术运算符，如++、--等，用来递增或者递减Iter。使用这些运算符可以遍历一个动态数组。考虑到速度的因素，这些算术运算符本身没有执行范围检查。然而，任何访问边界外元素的尝试都会导致异常，从而阻止了越界错误。Iter定义了关系运算符。指针算术运算和关系运算的函数都相当直接并且易于理解。

Iter还定义了一个名为size()的实用函数，它返回Iter所指数组的长度。

如前所述，在GCPtr中，对于每个GCPtr的实例，Iter<T>都被类型定义为GCiterator，从而简化了迭代器的声明。这意味着您可以使用类型名GCiterator来为任何GCPtr获取Iter。

2.8 如何使用GCPtr

GCPtr的使用相当容易。首先，包含文件gc.h。然后，声明一个GCPtr对象，并且指定它所指向的数据的类型。例如，为了声明一个名为p的、指向int的GCPtr对象，可以使用下面的声明：

```
GCPtr<int> p; // p can point to int objects
```

随后，使用new动态分配内存，并将new返回的指针赋给p，如下所示：

```
p = new int; // assign p the address of an int
```

可以使用这样的赋值操作来给已分配内存赋一个值：

```
*p = 88; // give that int a value
```

当然，可以将前面的3个语句合并在一起：

```
GCPtr<int> p = new int(88); // declare and initialize
```

可以获取p所指位置的内存的值，如下所示：

```
int k = *p;
```

正如前面示例所显示的那样，通常使用GCPtr的方式类似于使用C++的普通指针的方式。惟一区别就是当您不再需要此指针时，不需要删除它。当为这个指针分配的内存不再需要的时候，会自动释放它。

下面是组合了前面所示片断的完整程序：

```
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
  GCPtr<int> p;

  try {
```

```
    p = new int;
  } catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
  }

  *p = 88;

  cout << "Value at p is: " << *p << endl;

  int k = *p;

  cout << "k is " << k << endl;

  return 0;
}
```

当开启显示选项时，这个程序的输出如下所示。(可以在 gc.h 中定义 DISPLAY 来观察垃圾回收器的运行)：

```
Constructing GCPtr.
Value at p is: 88
k is 88
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr       refcount      value
[002F12C0]       0          88
[00000000]       0          ---

Deleting: 88
After garbage collection for gclist<int, 0>:
memPtr       refcount      value
            -- Empty --
```

当程序结束时，p 超出作用域。这将调用它的析构函数，从而使得 p 所指内存的引用计数递减。由于 p 是这块内存惟一的指针，因此这个操作使得引用计数变为 0。随后，p 的析构函数调用 collect()，collect 扫描 gclist，查找是否有引用计数为 0 的条目。由于先前与 p 相关条目的引用计数为 0，因此它所指的内存就会被释放。

2.8.1 处理分配异常

正如前面程序所显示的那样，由于垃圾回收器没有改变通过 new 分配内存的方式，因此可以用通常的方法捕获 bad_alloc 异常来处理分配失败。(当 new 失败时，就会抛出 bad_alloc 类型的异常)。当然，前面的程序不会用尽内存，从而实际上不需要 try/catch 块，但是实际的程序可能会耗尽堆。因此，您应该检测这种异常。

通常，在使用垃圾回收时，对 bad_alloc 异常最佳的响应方式是调用 collect()来回收任何不再使用的内存，然后重新尝试分配内存。在本章稍后显示的加载-测试程序中采用了这种技术。也可以在您的程序中使用这种技术。

2.8.2 一个更有趣的示例

在此有一个更有趣的示例，显示了在程序结束之前 GCPtr 超出作用域的效果：

```
// Show a GCPtr going out of scope prior to the end
// of the program.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
  GCPtr<int> p;
  GCPtr<int> q;

  try {
    p = new int(10);
    q = new int(11);

    cout << "Value at p is: " << *p << endl;
    cout << "Value at q is: " << *q << endl;

    cout << "Before entering block.\n";

    // Now, create a local object.
    { // start a block
      GCPtr<int> r = new int(12);
      cout << "Value at r is: " << *r << endl;
    } // end the block, causing r to go out of scope

    cout << "After exiting block.\n";

  } catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
  }

  cout << "Done\n";

  return 0;
}
```

当开启显示选项时，整个程序的输出如下：

```
Constructing GCPtr.
Constructing GCPtr.
Value at p is: 10
Value at q is: 11
Before entering block.
Constructing GCPtr.
Value at r is: 12
```

```
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F31D8]      0          12
[002F12F0]      1          11
[002F12C0]      1          10
[00000000]      0          ---

Deleting: 12
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F12F0]      1          11
[002F12C0]      1          10

After exiting block.
Done
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount     value
[002F12F0]      0          11
[002F12C0]      1          10

Deleting: 11
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F12C0]      1          10

GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F12C0]      0          10

Deleting: 10
After garbage collection for gclist<int, 0>:
memPtr      refcount       value
          -- Empty --
```

下面仔细地分析这个程序及其输出。首先，注意在 main()的开始创建了 p 和 q，但是直到输入 r 的代码块才创建 r。在 C++中，直到进入局部变量所在的代码块时，才会创建局部变量。当创建 r 时，它所指的内存被赋予了初始值 12。然后显示这个值，代码块结束。这使得 r 超出了作用域，意味着调用了它的析构函数。从而 gclist 中 r 的引用计数减小到 0。然后调用 collect()来回收垃圾。

由于开启了显示选项，当 collect()开始时，就会显示 gclist 的内容。注意它有 4 个条目。第一个条目先前与 r 链接。注意其引用计数为 0，说明由 memPtr 字段指的内存不再被任何程序元素使用。后面的两个条目仍然是活动的，并且与 p 和 q 链接。由于仍然使用它们，因此在此时不会释放它们所指的内存。最后一个条目代表了一个空指针，指向 p 和 q 创建时最初指向的条目。由于不再使用它，因此，collect()将会把它从链表中移除。(当然，当移除 *null* 指针时，不会释放内存)。

由于没有其他的 GCPtr 指向与 r 相同的内存，这块内存就可以释放了，Deleting: 12 行证

明了这一点。当完成这些动作的时候，这个代码块之后的程序继续执行。最后，p 和 q 在程序结束时超出了作用域，释放它们所指的内存。在此情况下，首先调用 q 的析构函数，意味着首先将它回收。最后销毁 p，gclist 为空。

2.8.3 对象的分配和丢弃

当某块内存的引用计数变为 0 的时候(意味着没有 GCPtr 指向它)，这块内存就要被回收，理解这一点很重要。并不需要原先指向这个对象的 GCPtr 一定要超出作用域。因此，可以使用一个 GCPtr 对象指向任意数量的已分配的对象，只需要将这个 GCPtr 赋予一个新的值。被丢弃的内存最终会被回收。例如：

```
// Allocate and discard objects.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
  try {
    // Allocate and discard objects.
    GCPtr<int> p = new int(1);
    p = new int(2);
    p = new int(3);
    p = new int(4);

    // Manually collect unused objects for
    // demonstration purposes.
    GCPtr<int>::collect();

    cout << "*p: " << *p << endl;
  } catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
  }


  return 0;
}
```

当开启显示选项时，程序的输出如下：

```
Constructing GCPtr.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F1310]      1          4
[002F1300]      0          3
[002F12D0]      0          2
[002F12A0]      0          1

Deleting: 3
```

```
Deleting: 2
Deleting: 1
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F1310]      1          4

*p: 4
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F1310]      0          4

Deleting: 4
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
          -- Empty --
```

在这个程序中，p 是指向 int 的 GCPtr，被赋予了 4 个指向独立动态内存块的指针，每一块内存都使用不同的值进行了初始化。然后调用了 collect()，强制进行垃圾回收。注意 gclist 的内容：3 个条目被标记为不活跃，只有指向最后分配的内存的条目仍然在使用。随后，删除不再使用的条目。最终，程序结束，p 超出了作用域，最后的条目被删除。

注意，p 指向的前三块内存的引用计数为 0。这是由重载的赋值运算符的运行方式决定的。当给 GCPtr 赋予一个新地址时，它的原始值的引用计数减小。因此，每次给 p 赋予新整型数的地址时，旧地址的引用计数就会减小。

另外，由于 p 在声明的时候就被初始化，因此没有产生 null 指针条目，也没有将其放入到 gclist 中。记住，只有在声明 GCPtr 而不使用初始值时，才会建立 null 指针条目。

2.8.4 分配数组

如果使用 new 分配数组，那么必须在声明 GCPtr 所指的数组时指定它的大小，以通知 GCPtr 这一事实。例如，下面是分配 5 个 double 数组的方法：

```
GCPtr<double, 5> pda = new double[5];
```

有两个原因使得必须指定这个大小。首先，这样做会通知 GCPtr 构造函数，这个对象将指向一个已分配的数组，从而使得 isArray 字段为 true。当 isArray 为 true 的时候，collect()函数使用 delete[]释放内存，从而释放了动态分配的数组，而不是使用 delete 仅释放一个对象。因此，在示例中，当 pda 超出作用域时，使用了 delete[]，pad 的 5 个元素全部被释放。当分配类对象的数组时，确保释放正确数量的对象尤其重要。只有使用 delete[]，才能够确保调用每个对象的析构函数。

必须指定数组大小的第二个原因是为了在使用 Iter 遍历已分配数组时，阻止对数组元素的越界访问。当需要 Iter 时，数组的大小(存储在 arraySize 中)由 GCPtr 传递给 Iter 的构造函数。

要知道，在此并没有强制要求只能通过指向数组时指定的 GCPtr 来操作已分配的数组。您自己应该承担这个责任。

在您想要分配数组时，有两种方法可以访问它的元素。首先，可以给指向它的 GCPtr 做索引。其次，可以使用迭代器。以下介绍这两种方法。

1. 使用数组索引

下面程序创建的 GCPtr 指向具有 10 个元素的 int 数组。然后分配了这个数组，并将其初始化为值 0~9。最后通过给 GCPtr 做索引，显示这些值：

```
// Demonstrate indexing a GCPtr.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {

  try {
    // Create a GCPtr to an allocated array of 10 ints.
    GCPtr<int, 10> ap = new int[10];

    // Give the array some values using array indexing.
    for(int i=0; i < 10; i++)
      ap[i] = i;

    // Now, show the contents of the array.
    for(int i=0; i < 10; i++)
      cout << ap[i] << " ";

    cout << endl;

  } catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
  }


  return 0;
}
```

当关闭显示选项时，输出如下：

```
0 1 2 3 4 5 6 7 8 9
```

由于 GCPtr 模拟了普通的 C++指针，没有执行数组边界的检查，因此有可能会越界访问动态分配的数组。因此，在使用 GCPtr 访问数组时，要像使用普通的 C++指针访问数组那样小心。

2. 使用迭代器

尽管数组索引是遍历已分配的数组的方便方法，但它并不是您处理问题的惟一方法。对于许多应用程序，使用迭代器将是更好的选择，因为这样做可以防止越界的错误。对于 GCPtr，迭代器是 Iter 类型的对象。Iter 支持全部的指针运算，如++。它还允许迭代器像数组那样被索引。

在此使用迭代器修改前面的程序示例。获取 GCPtr 的迭代器的最方便方法是使用

GCiterator，这是在 GCPtr 内的 typedef，GCPtr 自动绑定到通用类型 T：

```
// Demonstrate an iterator.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {

try {
  // Create a GCPtr to an allocated array of 10 ints.
    GCPtr<int, 10> ap = new int[10];

    // Declare an int iterator.
    GCPtr<int>::GCiterator itr;

    // Assign itr a pointer to the start of the array.
    itr = ap.begin();

    // Give the array some values using array indexing.
    for(unsigned i=0; i < itr.size(); i++)
      itr[i] = i;

    // Now, cycle through array using the iterator.
    for(itr = ap.begin(); itr != ap.end(); itr++)
      cout << *itr << " ";

    cout << endl;

  } catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
  } catch(OutOfRangeExc exc) {
    cout << "Out of range access!\n";
    return 1;
  }


  return 0;
}
```

您可能会试图递增 itr，以使得它所指的位置越过已分配数组的边界，然后会试图在这个位置访问值。您将会看到，这样做会抛出 OutOfRangeExc 异常。通常，可以使用您喜欢的任意方式增加或者减小迭代器而不会导致异常。然而，如果迭代器所指向的位置不在数组范围之内的话，尝试在这个位置获取或者设置值都会导致越界错误。

2.8.5 使用具有类类型的 GCPtr

使用具有类类型的 GCPtr 与使用具有内建类型的 GCPtr 没有区别。例如，在此由一个简短

的程序分配了 MyClass 的对象：

```
// Use GCPtr with a class type.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

class MyClass {
  int a, b;
public:
  double val;

  MyClass() { a = b = 0; }

  MyClass(int x, int y) {
    a = x;
    b = y;
    val = 0.0;
  }

  ~MyClass() {
    cout << "Destructing MyClass(" <<
         a << ", " << b << ")\n";
  }

  int sum() {
    return a + b;
  }

  friend ostream &operator<<(ostream &strm, MyClass &obj);
};

// An overloaded inserter to display MyClass.
ostream &operator<<(ostream &strm, MyClass &obj) {
  strm << "(" << obj.a << " " << obj.b << ")";
  return strm;
}

int main() {
  try {
    GCPtr<MyClass> ob = new MyClass(10, 20);

    // Show value via overloaded inserter.
    cout << *ob << endl;

    // Change object pointed to by ob.
    ob = new MyClass(11, 21);
    cout << *ob << endl;

    // Call a member function through a GCPtr.
```

```
    cout << "Sum is : " << ob->sum() << endl;

    // Assign a value to a class member through a GCPtr.
    ob->val = 98.6;
    cout << "ob->val: " << ob->val << endl;

    cout << "ob is now " << *ob << endl;
  } catch(bad_alloc exc) {
    cout << "Allocation error!\n";
    return 1;
  }

  return 0;
}
```

注意使用->运算符访问 MyClass 的成员的方式。记住，GCPtr 定义了一个指针类型，因此，对 GCPtr 执行的操作与其他任何类型指针的操作方式完全相似。

当关闭显示选项时，程序的输出如下：

```
(10 20)
(11 21)
Sum is : 32
ob->val: 98.6
ob is now (11 21)
Destructing MyClass(11, 21)
Destructing MyClass(10, 20)
```

特别注意最后两行，这是在垃圾回收时~MyClass()的输出。虽然只建立了一个 GCPtr 指针，然而分配了两个 MyClass 对象。垃圾回收链表中的条目表示了这两个对象。当销毁 ob 时，扫描 gclist 以找到引用计数为 0 的条目。在此情况下，会找到这样的两个条目，并删除它们所指的内存。

2.8.6 一个比较大的演示程序

下面的程序是一个比较大的示例，它体现了 GCPtr 的所有特性：

```
// Demonstrating GCPtr.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

// A simple class for testing GCPtr with class types.
class MyClass {
  int a, b;
public:
  double val;

  MyClass() { a = b = 0; }
```

```
  MyClass(int x, int y) {
    a = x;
    b = y;
    val = 0.0;
  }

  ~MyClass() {
    cout << "Destructing MyClass(" <<
         a << ", " << b << ")\n";
  }

  int sum() {
    return a + b;
  }

  friend ostream &operator<<(ostream &strm, MyClass &obj);
};

// Create an inserter for MyClass.
ostream &operator<<(ostream &strm, MyClass &obj) {
  strm << "(" << obj.a << " " << obj.b << ")";
  return strm;
}

// Pass a normal pointer to a function.
void passPtr(int *p) {
  cout << "Inside passPtr(): "
       << *p << endl;
  }

// Pass a GCPtr to a function.
void passGCPtr(GCPtr<int, 0> p) {
  cout << "Inside passGCPtr(): "
       << *p << endl;
}

int main() {

  try {
    // Declare an int GCPtr.
    GCPtr<int> ip;

    // Allocate an int and assign its address to ip.
    ip = new int(22);

    // Display its value.
    cout << "Value at *ip: " << *ip << "\n\n";

    // Pass ip to a function
    passGCPtr(ip);

    // ip2 is created and then goes out of scope
```

```
{
  GCPtr<int> ip2 = ip;
}

int *p = ip; // convert to int * pointer'

passPtr(p); // pass int * to passPtr()

*ip = 100; // Assign new value to ip

// Now, use implicit conversion to int *
passPtr(ip);
cout << endl;

// Create a GCPtr to an array of ints
GCPtr<int, 5> iap = new int[5];

// Initialize dynamic array.
for(int i=0; i < 5; i++)
  iap[i] = i;

// Display contents of array.
cout << "Contents of iap via array indexing.\n";
for(int i=0; i < 5; i++)
  cout << iap[i] << " ";
cout << "\n\n";

// Create an int GCiterator.
GCPtr<int>::GCiterator itr;

// Now, use iterator to access dynamic array.
cout << "Contents of iap via iterator.\n";
for(itr = iap.begin(); itr != iap.end(); itr++)
  cout << *itr << " ";
cout << "\n\n";

// Generate and discard many objects
for(int i=0; i < 10; i++)
  ip = new int(i+10);

// Now, manually garbage collect GCPtr<int> list.
// Keep in mind that GCPtr<int, 5> pointers
// will not be collected by this call.
cout << "Requesting collection on GCPtr<int> list.\n";
GCPtr<int>::collect();

// Now, use GCPtr with class type.
GCPtr<MyClass> ob = new MyClass(10, 20);

// Show value via overloaded insertor.
cout << "ob points to " << *ob << endl;
```

```
// Change object pointed to by ob.
ob = new MyClass(11, 21);
cout << "ob now points to " << *ob << endl;

// Call a member function through a GCPtr.
cout << "Sum is : " << ob->sum() << endl;

// Assign a value to a class member through a GCPtr.
ob->val = 19.21;
cout << "ob->val: " << ob->val << "\n\n";

cout << "Now work with pointers to class objects.\n";

// Declare a GCPtr to a 5-element array
// of MyClass objects.
GCPtr<MyClass, 5> v;

// Allocate the array.
v = new MyClass[5];

// Get a MyClass GCiterator.
GCPtr<MyClass>::GCiterator mcItr;

// Initialize the MyClass array.
for(int i=0; i<5; i++) {
  v[i] = MyClass(i, 2*i);
}

// Display contents of MyClass array using indexing.
cout << "Cycle through array via array indexing.\n";
for(int i=0; i<5; i++) {
  cout << v[i] << " ";
}
cout << "\n\n";

// Display contents of MyClass array using iterator.
cout << "Cycle through array through an iterator.\n";
for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
  cout << *mcItr << " ";
}
cout << "\n\n";

// Here is another way to write the preceding loop.
cout << "Cycle through array using a while loop.\n";
mcItr = v.begin();
while(mcItr != v.end()) {
  cout << *mcItr << " ";
  mcItr++;
}
cout << "\n\n";

cout << "mcItr points to an array that is "
```

```
      << mcItr.size() << " objects long.\n";

  // Find number of elements between two iterators.
  GCPtr<MyClass>::GCiterator mcItr2 = v.end()-2;
  mcItr = v.begin();
  cout << "The difference between mcItr2 and mcItr is "
       << mcItr2 - mcItr;
  cout << "\n\n";

  // Can also cycle through loop like this.
  cout << "Dynamically compute length of array.\n";
  mcItr = v.begin();
  mcItr2 = v.end();
  for(int i=0; i < mcItr2 - mcItr; i++) {
    cout << v[i] << " ";
  }
  cout << "\n\n";

  // Now, display the array backwards.
  cout << "Cycle through array backwards.\n";
  for(mcItr = v.end()-1; mcItr >= v.begin(); mcItr--)
    cout << *mcItr << " ";
  cout << "\n\n";

  // Of course, can use "normal" pointer to
  // cycle through array.
  cout << "Cycle through array using 'normal' pointer\n";
  MyClass *ptr = v;
  for(int i=0; i < 5; i++)
    cout << *ptr++ << " ";
  cout << "\n\n";

  // Can access members through a GCiterator.
  cout << "Access class members through an iterator.\n";
  for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
    cout << mcItr->sum() << " ";
  }
  cout << "\n\n";

  // Can allocate and delete a pointer to a GCPtr
  // normally, just like any other pointer.
  cout << "Use a pointer to a GCPtr.\n";
  GCPtr<int> *pp = new GCPtr<int>();
  *pp = new int(100);
  cout << "Value at **pp is: " << **pp;
  cout << "\n\n";

  // Because pp is not a garbage collected pointer,
  // it must be deleted manually.
  delete pp;
} catch(bad_alloc exc) {
  // A real application could attempt to free
```

```
    // memory by collect() when an allocation
    // error occurs.
    cout << "Allocation error.\n";
  }

  return 0;
}
```

当关闭显示选项时，程序输出如下：

```
Value at *ip: 22

Inside passGCPtr(): 22
Inside passPtr(): 22
Inside passPtr(): 100

Contents of iap via array indexing.
0 1 2 3 4

Contents of iap via iterator.
0 1 2 3 4

Requesting collection on GCPtr<int> list.
ob points to (10 20)
ob now points to (11 21)
Sum is : 32
ob->val: 19.21

Now work with pointers to class objects.
Destructing MyClass(0, 0)
Destructing MyClass(1, 2)
Destructing MyClass(2, 4)
Destructing MyClass(3, 6)
Destructing MyClass(4, 8)
Cycle through array via array indexing.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array through an iterator.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array using a while loop.
(0 0) (1 2) (2 4) (3 6) (4 8)

mcItr points to an array that is 5 objects long.
The difference between mcItr2 and mcItr is 3

Dynamically compute length of array.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array backwards.
(4 8) (3 6) (2 4) (1 2) (0 0)
```

```
Cycle through array using 'normal' pointer
(0 0) (1 2) (2 4) (3 6) (4 8)

Access class members through an iterator.
0 3 6 9 12

Use a pointer to a GCPtr.
Value at **pp is: 100

Destructing MyClass(4, 8)
Destructing MyClass(3, 6)
Destructing MyClass(2, 4)
Destructing MyClass(1, 2)
Destructing MyClass(0, 0)
Destructing MyClass(11, 21)
Destructing MyClass(10, 20)
```

开启显示选项(也就是在 gc.h 中定义 DISPLAY)，试着编译并运行这个程序。然后浏览程序，将输出与每条语句匹配。从而您会对垃圾回收器的运行方式有明确的认识。记住，无论什么时候 GCPtr 超出作用域，都会发生垃圾回收。这在程序的不同时刻发生，例如，当接收了 GCPtr 的副本的函数返回时，在此情况下，这个副本超出作用域，从而发生垃圾回收。还要记住每一个类型的 GCPtr 都维护着自身的 gclist。因此，从一个链表进行垃圾回收不会导致它从其他类型的链表回收。

2.8.7 加载测试

下面的程序通过重复地分配并丢弃对象直到自由内存耗尽来加载测试 GCPtr。当这些发生时，new 会抛出 bad_alloc 异常。在异常处理程序中，显式地调用了 collect()来回收不再使用的内存，这个过程继续。可以在您自己的程序中使用相同的技术：

```
// Load test GCPtr by creating and discarding
// thousands of objects.
#include <iostream>
#include <new>
#include <limits>
#include "gc.h"

using namespace std;

// A simple class for load testing GCPtr.
class LoadTest {
  int a, b;
public:
  double n[100000]; // just to take up memory
  double val;
  LoadTest() { a = b = 0; }

  LoadTest(int x, int y) {
    a = x;
    b = y;
```

```
    val = 0.0;
  }

  friend ostream &operator<<(ostream &strm, LoadTest &obj);
};

// Create an inserter for LoadTest.
ostream &operator<<(ostream &strm, LoadTest &obj) {
  strm << "(" << obj.a << " " << obj.b << ")";
  return strm;
}

int main() {
  GCPtr<LoadTest> mp;
  int i;

  for(i = 1; i < 20000; i++) {
    try {
      mp = new LoadTest(i, i);
    } catch(bad_alloc xa) {
      // When an allocation error occurs, recycle
      // garbage by calling collect().
      cout << "Last object: " << *mp << endl;
      cout << "Length of gclist before calling collect(): "
           << mp.gclistSize() << endl;
      GCPtr<LoadTest>::collect();
      cout << "Length after calling collect(): "
           << mp.gclistSize() << endl;
    }
  }

  return 0;
}
```

这个程序的部分输出(关闭显示选项)如下。当然，您看到的确切输出可能会有所不同，因为您系统可用内存的数量与您使用的编译器可能不同。

```
Last object: (518 518)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (1036 1036)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (1554 1554)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2072 2072)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2590 2590)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
```

```
Last object: (3108 3108)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (3626 3626)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
```

2.8.8 一些限制

下面是使用 GCPtr 的一些限制：

(1) 不能创建全局的 GCPtr。程序的其余部分结束之后，全局对象才会超出作用域。当全局的 GCPtr 超出作用域时，GCPtr 的析构函数调用 collect()来试图释放不再使用的内存。问题是，根据您的 C++编译器的实现方式，gclist 可能已经被销毁。在此情况下，执行 collect()会导致运行时错误。因此，GCPtr 只能在创建局部对象时使用。

(2) 当使用动态分配的数组时，您在声明指向它的 GCPtr 时必须指定这个数组的大小。然而并没有机制强制这么做，因此要小心。

(3) 不能通过显式地使用 delete 来释放 GCPtr 所指的内存。如果需要立即释放一个对象，就可以调用 collect()。

(4) GCPtr 对象只能指向由 new 动态分配的内存。将任何其他类型的指针赋给 GCPtr 都会在 GCPtr 对象超出作用域时引发错误，因为尝试释放从来没有分配过的内存。

(5) 最好避免循环指针引用，原因在本章的前面部分已经描述过。尽管所有已分配的内存最终会被释放，然而包含了循环引用的对象直到程序结束才会被释放，而不是在其他程序元素不再使用它们的时候被释放。

2.9 试着完成下面的任务

可以容易地通过修改 GCPtr 来适应您的应用程序的需求。如前所述，您想要尝试的改变之一是在达到某个标准时才试图回收垃圾，如 gclist 达到了某个尺寸，或者一定数量的 GCPtr 超出了作用域。

对 GCPtr 功能的一个有趣增强是重载 new，从而可以在分配失败时自动回收垃圾。也可以在为 GCPtr 分配内存的时候不使用 new，而使用 GCPtr 定义的工厂函数来代替。这样做可以让您更仔细地控制动态内存的分配，但是使得分配过程从根本上不同于 C++内建的方法。

您可能会尝试使用其他方法来处理循环引用的问题。方法之一是实现弱引用的概念，它不会阻止垃圾回收的发生。可以在需要循环引用的时候使用弱引用。

可能关于 GCPtr 最有趣的版本在第 3 章。在那里创建了一个多线程的版本，当 CPU 空闲时，垃圾回收就会自动进行。