CHAPTER 2

A Simple Garbage Collector for C++

Throughout the history of computing, there has been an ongoing debate concerning the best way to manage the use of dynamically allocated memory. Dynamically allocated memory is memory that is obtained during runtime from the *heap*, which is a region of free memory that is available for program use. The heap is also commonly referred to as *free store* or *dynamic memory*. Dynamic allocation is important because it enables a program to obtain, use, release, and then reuse memory during execution. Because nearly all real-world programs use dynamic allocation in some form, the way it is managed has a profound effect on the architecture and performance of programs.

In general, there are two ways that dynamic memory is handled. The first is the manual approach, in which the programmer must explicitly release unused memory in order to make it available for reuse. The second relies on an automated approach, commonly referred to as *garbage collection*, in which memory is automatically recycled when it is no longer needed. There are advantages and disadvantages to both approaches, and the favored strategy has shifted between the two over time.

C++ uses the manual approach to managing dynamic memory. Garbage collection is the mechanism employed by Java and C#. Given that Java and C# are newer languages, the current trend in computer language design seems to be toward garbage collection. This does not mean, however, that the C++ programmer is left on the "wrong side of history." Because of the power built into C++, it is possible—even easy—to create a garbage collector for C++. Thus, the C++ programmer can have the best of both worlds: manual control of dynamic allocation when needed and automatic garbage collection when desired.

This chapter develops a complete garbage collection subsystem for C++. At the outset, it is important to understand that the garbage collector does not replace C++'s built-in approach to dynamic allocation. Rather, it supplements it. Thus, both the manual and garbage collection systems can be used within the same program.

Aside from being a useful (and fascinating) piece of code in itself, a garbage collector was chosen for the first example in this book because it clearly shows the unsurpassed power of C++. Through the use of template classes, operator overloading, and C++'s inherent ability to handle the low-level elements upon which the computer operates, such as memory addresses, it is possible to transparently add a core feature to C++. For most other languages, changing the way that dynamic allocation is handled would require a change to the compiler itself. However, because of the unparalleled power that C++ gives the programmer, this task can be accomplished at the source code level.

The garbage collector also shows how a new type can be defined and fully integrated into the C++ programming environment. Such *type extensibility* is a key component of C++, and it's one that is often overlooked. Finally, the garbage collector testifies to C++'s ability to "get close to the machine" because it manipulates and manages pointers. Unlike some other languages which prevent access to the low-level details, C++ lets the programmer get as close to the hardware as necessary.

Comparing the Two Approaches to Memory Management

Before developing a garbage collector for C++, it is useful to compare garbage collection to the manual method that is built-in to C++. Normally, the use of dynamic memory in C++ requires a two-step process. First, memory is allocated from the heap via **new**. Second, when that memory is no longer needed, it is released by **delete**. Thus, each dynamic allocation follows this sequence:

```
p = new some_object;

// ...

delete p;
```

In general, each use of **new** must be balanced by a matching **delete**. If **delete** is not used, the memory is not released, even if that memory is no longer needed by your program.

Garbage collection differs from the manual approach in one key way: it automates the release of unused memory. Therefore, with garbage collection, dynamic allocation is a one-step operation. For example, in Java and C#, memory is allocated for use by **new**, but it is never explicitly freed by your program. Instead, the garbage collector runs periodically, looking for pieces of memory to which no other object points. When no other object points to a piece of dynamic memory, it means that there is no program element using that memory. When it finds a piece of unused memory, it frees it. Thus, in a garbage collection system, there is no **delete** operator, nor a need for one, either.

At first glance, the inherent simplicity of garbage collection makes it seem like the obvious choice for managing dynamic memory. In fact, one might question why the manual method is used at all, especially by a language as sophisticated as C++. However, in the case of dynamic allocation, first impressions prove deceptive because both approaches involve a set of trade-offs. Which approach is most appropriate is decided by the application. The following sections describe some of the issues involved.

The Pros and Cons of Manual Memory Management

The main benefit of manually managing dynamic memory is efficiency. Because there is no garbage collector, no time is spent keeping track of active objects or periodically looking for unused memory. Instead, when the programmer knows that the allocated object is no longer needed, the programmer explicitly frees it and no additional overhead is incurred. Because it has none of the overhead associated with garbage collection, the manual approach enables more efficient code to be written This is one reason why it was necessary for C++ to support manual memory management: it enabled the creation of high-performance code.

Another advantage to the manual approach is control. Although requiring the programmer to handle both the allocation and release of memory is a burden, the benefit is that the programmer gains complete control over both halves of the process. You know precisely when memory is being allocated and precisely when it is being released. Furthermore, when you release an object via **delete**, its destructor is executed at that point rather than at some later time, as can be the case with garbage collection. Thus, with the manual method you can control precisely when an allocated object is destroyed.

Although it is efficient, manual memory management is susceptible to a rather annoying type of error: the memory leak. Because memory must be freed manually, it is possible (even easy) to forget to do so. Failing to release unused memory means that the memory will remain allocated even if it is no longer needed. Memory leaks cannot occur in a garbage collection environment because the garbage collector ensures that unused objects are eventually freed. Memory leaks are a particularly troublesome problem in Windows programming, where the failure to release unused resources slowly degrades performance.

Other problems that can occur with C++'s manual approach include the premature releasing of memory that is still in use, and the accidental freeing of the same memory twice. Both of these errors can lead to serious trouble. Unfortunately, they may not show any immediate symptoms, making them hard to find.

The Pros and Cons of Garbage Collection

There are several different ways to implement garbage collection, each offering different performance characteristics. However, all garbage collection systems share a set of common attributes that can be compared against the manual approach. The main advantages to garbage collection are simplicity and safety. In a garbage collection environment, you explicitly allocate memory via **new**, but you never explicitly free it. Instead, unused memory is automatically recycled. Thus, it is not possible to forget to release an object or to release an object prematurely. This simplifies programming and prevents an entire class of problems. Furthermore, it is not possible to accidentally free dynamically allocated memory twice. Thus, garbage collection provides an easy-to-use, error-free, reliable solution to the memory management problem.

Unfortunately, the simplicity and safety of garbage collection come at a price. The first cost is the overhead incurred by the garbage collection mechanism. All garbage collection schemes consume some CPU cycles because the reclamation of unused memory is not a cost-free process. This overhead does not occur with the manual approach.

A second cost is loss of control over when an object is destroyed. Unlike the manual approach, in which an object is destroyed (and its destructor called) at a known point in time—when a **delete** statement is executed on that object—garbage collection does not have such a hard and fast rule. Instead, when garbage collection is used, an object is not destroyed until the collector runs and recycles the object, which may not occur until some arbitrary time in the future. For example, the collector might not run until the amount of free memory drops below a certain point. Furthermore, it is not always possible to know the order in which objects will be destroyed by the garbage collector. In some cases, the inability to know precisely when an object is destroyed can cause trouble because it also means that your program can't know precisely when the destructor for a dynamically allocated object is called.

For garbage collection systems that run as a background task, this loss of control can escalate into a potentially more serious problem for some types of applications because it introduces what is essentially nondeterministic behavior into a program. A garbage collector that executes in the background reclaims unused memory at times that are, for all practical purposes, unknowable. For example, the collector will usually run only when free CPU time is available. Because this might vary from one program run to the next, from one computer to next, or from one operating system to the next, the precise point in program execution at which the garbage collector executes is effectively nondeterministic. This is not a problem for many programs, but it can cause havoc with real-time applications in which the unexpected allocation of CPU cycles to the garbage collector could cause an event to be missed.

You Can Have It Both Ways

As the preceding discussions explained, both manual management and garbage collection maximize one feature at the expense of another. The manual approach maximizes efficiency and control at the expense of safety and ease of use. Garbage collection maximizes simplicity and safety but pays for it with a loss of runtime performance and control. Thus, garbage collection and manual memory management are essentially opposites, each maximizing the traits that the other sacrifices. This is why neither approach to dynamic memory management can be optimal for all programming situations.

Although opposites, the two approaches are not mutually exclusive. They can coexist. Thus, it is possible for the C++ programmer to have access to both approaches, choosing the proper method for the task at hand. All one needs to do is create a garbage collector for C++, and this is the subject of the rest of this chapter.

Creating a Garbage Collector in C++

Because C++ is a rich and powerful language, there are many different ways to implement a garbage collector. One obvious, but limited, approach is to create a garbage collector base class, which is then inherited by classes that want to use garbage collection. This would enable you to implement garbage collection on a class-by-class basis. This solution is, unfortunately, too narrow to be satisfying.

A better solution is one in which the garbage collector can be used with any type of dynamically allocated object. To provide such a solution, the garbage collector must:

1. Coexist with the built-in, manual method provided by C++.
2. Not break any preexisting code. Moreover, it must have no impact whatsoever on existing code.
3. Work transparently so that allocations that use garbage collection are operated on in the same way as those that don't.
4. Allocate memory using **new** in the same way that C++'s built-in approach does.
5. Work with all data types, including the built-in types such as **int** and **double**.
6. Be simple to use.

In short, the garbage collection system must be able to dynamically allocate memory using a mechanism and syntax that closely resemble that already used by C++ and not affect existing code. At first thought, this might seem to be a daunting task, but it isn't.

Understanding the Problem

The key challenge that one faces when creating a garbage collector is how to know when a piece of memory is unused. To understand the problem, consider the following sequence:

```
int *p;

p = new int(99);

p = new int(100);
```

Here, two **int** objects are dynamically allocated. The first contains the value 99 and a pointer to this value is stored in **p**. Next, an integer containing the value 100 is allocated, and its address is also stored in **p**, thus overwriting the first address. At this point, the memory for **int(99)** is not pointed to by **p** (or any other object) and can be freed. The question is, how does the garbage collector know that neither **p** nor any other object points to **int(99)**?

Here is a slight variation on the problem:

```
int *p, *q;

p = new int(99);

q = p; // now, q points to same memory as p

p = new int(100);
```

In this case, **q** points to the memory that was originally allocated for **p**. Even though **p** is then pointed to a different piece of memory, the memory that it originally pointed to can't be freed because it is still in use by **q**. The question: how does the garbage collector know this fact? The precise way that these questions are answered is determined by the garbage collection algorithm employed.

Choosing a Garbage Collection Algorithm

Before implementing a garbage collector for C++, it is necessary to decide what garbage collection algorithm to use. The topic of garbage collection is a large one, having been the focus of serious academic study for many years. Because it presents an intriguing problem for which there is a variety of solutions, a number of different garbage collection algorithms have been designed. It is far beyond the scope of this book to examine each in detail. However, there are three archetypal approaches: *reference counting*, *mark and sweep*, and *copying*. Before choosing an approach, it will be useful to review these three algorithms.

Reference Counting

In reference counting, each dynamically allocated piece of memory has associated with it a reference count. This count is incremented each time a reference to the memory is added and decremented each time a reference to the memory is removed. In C++ terms, this means that each time a pointer is set to point to a piece of allocated memory, the reference count associated with that memory is incremented. When the pointer is set to point elsewhere, the reference count is decremented. When the reference count drops to zero, the memory is unused and can be released.

The main advantage of reference counting is simplicity—it is easy to understand and implement. Furthermore, it places no restrictions on the organization of the heap because the reference count is independent of an object's physical location. Reference counting adds overhead to each pointer operation, but the collection phase is relatively low cost. The main disadvantage is that circular references prevent memory that is otherwise unused from being released. A circular reference occurs when two objects point to each other, either directly or indirectly. In this situation, neither object's reference count ever drops to zero. Some solutions to the circular reference problem have been devised, but all add complexity and/or overhead.

Mark and Sweep

Mark and sweep involves two phases. In the first phase, all objects in the heap are set to their unmarked state. Then, all objects directly or indirectly accessible from program variables are marked as "in-use." In phase two, all of allocated memory is scanned (that is, a *sweep* of memory is made), and all unmarked elements are released.

There are two main advantages of mark and sweep. First, it easily handles circular references. Second, it adds virtually no runtime overhead prior to collection. It has two main disadvantages. First, a considerable amount of time might be spent collecting garbage because the entire heap must be scanned during collection. Thus, garbage collection may cause unacceptable runtime characteristics for some programs. Second, although mark and sweep is simple conceptually, it can be tricky to implement efficiently.

Copying

The copying algorithm organizes free memory into two spaces. One is active (holding the current heap), and the other is idle. During garbage collection, in-use objects from the active space are identified and copied into the idle space. Then, the roles of the two spaces are reversed, with the idle space becoming active and the active space becoming idle. Copying offers the advantage of compacting the heap in the copy process. It has the disadvantage of allowing only half of free memory to be in use at any one time.

Which Algorithm?

Given that there are advantages and disadvantages to all of the three classical approaches to garbage collection, it might seem hard to choose one over the other. However, given the constraints enumerated earlier, there is a clear choice: reference counting. First and most importantly, reference counting can be easily "layered onto" the existing C++ dynamic allocation system. Second, it can be implemented in a straightforward manner and in a way that does not affect preexisting code. Third, it does not require any specific organization or structuring of the heap, thus the standard allocation system provided by C++ is unaffected.

The one drawback to using reference counting is its difficulty in handling circular references. This isn't an issue for many programs because intentional circular references are not all that common and can usually be avoided. (Even things that we call *circular*, such as a circular queue, don't necessarily involve a circular pointer reference.) Of course, there are cases in which circular references are needed. It is also possible to create a circular reference without knowing you have done so, especially when working with third-party libraries. Therefore, the garbage collector must provide some means to gracefully handle a circular reference, should one exist.

To handle the circular reference problem, the garbage collector developed in this chapter will release any remaining allocated memory when the program exits. This ensures that objects involved in a circular reference will be freed and their destructors called. It is important to understand that normally there will be no allocated objects remaining at program termination. This mechanism is explicitly for those objects that can't be released because of a circular reference. (You might want to experiment with other means of handling the circular reference problem. It presents an interesting challenge.)

Implementing the Garbage Collector

To implement a reference counting garbage collector, there must be some way to keep track of the number of pointers that point to each piece of dynamically allocated memory. The trouble is that C++ has no built-in mechanism that enables one object to know when another object is pointing to it. Fortunately, there is a solution: you can create a new pointer type that supports garbage collection. This is the approach used by the garbage collector in this chapter.

To support garbage collection, the new pointer type must do three things. First, it must maintain a list of reference counts for active dynamically allocated objects. Second, it must keep track of all pointer operations, increasing an object's reference count each time a pointer is pointed to that object and decreasing the count each time a pointer is redirected to another object. Third, it must recycle those objects whose reference counts drop to zero. Aside from supporting garbage collection, the new pointer type will look and feel just like a normal pointer. For example, all pointer operations, such as ***** and **–>**, are supported.

In addition to being a convenient way to implement a garbage collector, the creation of a garbage collection pointer type satisfies the constraint that the original C++ dynamic allocation system must be unaffected. When garbage collection is desired, garbage collection-enabled pointers are used. When garbage collection is not desired, normal C++ pointers are available. Thus, both types of pointers can be used within the same program.

To Multithread or Not?

Another consideration when designing a garbage collector for C++ is whether it should be single-threaded or multithreaded. That is, should the garbage collector be designed as a background process, running in its own thread and collecting garbage as CPU time permits? Or, should the garbage collector run in the same thread as the process that uses it, collecting garbage when certain program conditions occur? Both approaches have advantages and disadvantages.

The main advantage to creating a multithreaded garbage collector is efficiency. Garbage can be collected when idle CPU cycles are available. The disadvantage is, of course, that C++ does not provide any built-in support for multithreading. This means that any multithreaded approach will depend upon operating system facilities to support the multitasking. This makes the code nonportable.

The main advantage to using a single-threaded garbage collector is portability. It can be used in situations that do not support multithreading or in cases in which the price of multithreading is too high. The main disadvantage is that the rest of the program stops when garbage collection takes place.

In this chapter, the single-threaded approach is used because it works in all C++ environments. Thus, it can be used by all readers of this book. However, for those readers wanting a multithreaded solution, one is given in Chapter 3, which deals with the techniques needed to successfully multithread a C++ program in a Windows environment.

When to Collect Garbage?

One final question that needs to be answered before a garbage collector can be implemented: When is garbage collected? This is less of an issue for a multithreaded garbage collector, which can run continuously as a background task, collecting garbage whenever CPU cycles are available, than it is for a single-threaded garbage collector, such as that developed in this chapter, which must stop the rest of the program to collect garbage.

In the real world, garbage collection usually takes place only when there is sufficient reason to do so, such as the case of memory running low. This makes sense for two reasons. First, with some garbage collection algorithms, such as mark and sweep, there is no way to know that a piece of memory is unused without actually performing the collection. (That is, sometimes there is no way to know that there is garbage to be collected without actually collecting it!) Second, collecting garbage is a time-consuming process which should not be performed needlessly.

However, waiting for memory to run low before initiating garbage collection is not suitable for the purposes of this chapter because it makes it next to impossible to demonstrate the collector. Instead, the garbage collector developed in this chapter will collect garbage more frequently so that its actions can easily be observed. As the collector is coded, garbage is collected whenever a pointer goes out of scope. Of course, this behavior can easily be changed to fit your applications.

One last point: when using reference-count based garbage collection, it is technically possible to recycle unused memory as soon as its reference count drops to zero, rather than using a separate garbage collection phase. However, this approach adds overhead to every pointer operation. The method used by this chapter is to simply decrement the memory's reference count each time a pointer to that memory is redirected and let the collection process handle the actual recycling of memory at more convenient times. This reduces the runtime overhead associated with pointer operations, which one typically wants to be as fast as possible.

What About auto_ptr?

As many readers will know, C++ defines a library class called **auto_ptr**. Because an **auto_ptr** automatically frees the memory to which it points when it goes out of scope, you might think that it would be of use when developing a garbage collector, perhaps forming the foundation. This is not the case, however. The **auto_ptr** class is designed for a concept that the ISO C++ Standard calls "strict ownership," in which an **auto_ptr** "owns" the object to which it points. This ownership can be transferred to another **auto_ptr**, but in all cases some **auto_ptr** will own the object until it is destroyed. Furthermore, an **auto_ptr** is assigned an address of an object only when it is initialized. After that, you can't change the memory to which an **auto_ptr** points, except by assigning one **auto_ptr** to another. Because of **auto_ptr**'s "strict ownership" feature, it is not particularly useful when creating a garbage collector, and it is not used by the garbage collectors in this book.

A Simple C++ Garbage Collector

The entire garbage collector is shown here. As explained, the garbage collector works by creating a new pointer type that provides built-in support for garbage collection based on reference counting. The garbage collector is single-threaded, which means that it is quite portable and does not rely upon (or make assumptions about) the execution environment. This code should be stored in a file called **gc.h**.

There are two things to notice as you look through the code. First, most of the member functions are quite short and are defined inside their respective classes in the interest of efficiency. Recall that a function defined within its class is automatically in-lined, which eliminates the overhead of a function call. Only a few functions are long enough to require their definition to be outside their class.

Secondly, notice the comment near the top of the file. If you want to watch the action of the garbage collector, simply turn on the display option by defining the macro called **DISPLAY**. For normal use, leave **DISPLAY** undefined.

```
// 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 operator>=(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
}
```

An Overview of the Garbage Collector Classes

The garbage collector uses four classes: **GCPtr**, **GCInfo**, **Iter**, and **OutOfRangeExc**. Before examining the code in detail, it will be helpful to understand the role each class plays.

GCPtr

At the core of the garbage collector is the class **GCPtr**, which implements a garbage-collection pointer. **GCPtr** maintains a list that associates a reference count with each piece of memory allocated for use by a **GCPtr**. In general, here is how it works. Each time a **GCPtr** is pointed at a piece of memory, the reference count for that memory is incremented. If the **GCPtr** was previously pointing at a different piece of memory prior to the assignment, the reference count for that memory is decremented. Thus, adding a pointer to a piece of memory increases the memory's reference count. Removing a pointer decreases the memory's reference count. Each time a **GCPtr** goes out of scope, the reference count associated with the memory to which it currently points is decremented. When a reference count drops to zero, that piece of memory can be released.

GCPtr is a template class that overloads the ***** and **–>** pointer operators and the array indexing operator **[ ]**. Thus, **GCPtr** creates a new pointer type and integrates it into the C++ programming environment. This allows a **GCPtr** to be used in much the same way that you use a normal C++ pointer. However, for reasons that will be made clear later in this chapter, **GCPtr** does not overload the **++**, **– –**, or the other arithmetic operations defined for pointers. Thus, except through assignment, you cannot change the address to which a **GCPtr** object points. This may seems like a significant restriction, but it isn't because the **Iter** class provides these operations.

For the sake of illustration, the garbage collector runs whenever a **GCPtr** object goes out of scope. At that time, the garbage collection list is scanned and all memory with a reference count of zero is released, even if it was not originally associated with the **GCPtr** that went out of scope. Your program can also explicitly request garbage collection if you need to recycle memory earlier.

GCInfo

As explained, **GCPtr** maintains a list that links reference counts with allocated memory. Each entry in this list is encapsulated in an object of type **GCInfo**. **GCInfo** stores the reference count in its **refcount** field and a pointer to the memory in its **memPtr** field. Thus, a **GCInfo** object binds a reference count to a piece of allocated memory.

GCInfo defines two other fields: **isArray** and **arraySize**. If **memPtr** points to an allocated array, then its **isArray** member will be **true** and the length of the array will be stored in its **arraySize** field.

The Iter Class

As explained, a **GCPtr** object allows you to access the memory to which it points by using the normal pointer operators * and –>, but it does not support pointer arithmetic. To handle situations in which you need to perform pointer arithmetic, you will use an object of type **Iter**. **Iter** is a template class similar in function to an STL iterator, and it defines all pointer operations, including pointer arithmetic. The main use of **Iter** is to enable you to cycle through the elements of a dynamically allocated array. It also provides bounds checking. You obtain an **Iter** from **GCPtr** by calling either **begin( )** or **end( )**, which work much like their equivalents in the STL.

It is important to understand that although **Iter** and the STL **iterator** type are similar, they are not the same, and you cannot use one in place of the other.

OutOfRangeExc

If an **Iter** encounters an attempt to access memory outside the range of the allocated memory, an **OutOfRangeExc** exception is thrown. For the purposes of this chapter, **OutOfRangeExc** contains no members. It is simply a type that can be thrown. However, you are free to add functionality to this class as your own applications dictate.

GCPtr In Detail

GCPtr is the heart of the garbage collector. It implements a new pointer type that keeps a reference count for objects allocated on the heap. It also provides the garbage collection functionality that recycles unused memory.

GCptr is a template class with this declaration:

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

GCPtr requires that you specify the type of the data that will be pointed to, which will be substituted for the generic type **T**. If an array is being allocated, you must specify the size of the array in the **size** parameter. Otherwise, **size** defaults to zero, which indicates that a single object is being pointed to. Here are two examples.

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

Here, **p** can point to single objects of type **int**, and **ap** can point to an array of 5 **int**s.

In the preceding examples, notice that you do not use the * operator when specifying the name of the **GCPtr** object. That is, to create a **GCPtr** to **int**, you *do not* use a statement like this:

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

This declaration creates a normal C++ pointer called **p** that can point to a **GCPtr<int>** object. It does not create a **GCPtr** object that can point to **int**. Remember, **GCPtr** defines a pointer type by itself.

Be careful when specifying the type parameter to **GCPtr**. It specifies the type of the object to which the **GCPtr** object can point. Therefore, if you write a declaration like this:

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

you are creating a **GCPtr** object that points to **int *** pointers, not a **GCPtr** to **int**s.

Because of its importance, each member of **GCPtr** is examined in detail in the following sections.

GCPtr Data Members

GCPtr declares the following data members:

```
// 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
```

The **gclist** field contains a list of **GCInfo** objects. (Recall that **GCInfo** links a reference count with a piece of allocated memory.) This list is used by the garbage collector to determine when allocated memory is unused. Notice that **gclist** is a **static** member of **GCPtr**. This means that for each specific pointer type, there is only one **gclist**. For example, all pointers of type **GCPtr<int>** share one list, and all pointers of type **GCPtr<double>** share a different list. **gclist** is an instance of the STL **list** class. Using the STL substantially simplifies the code for **GCPtr** because there is no need for it to create its own set of list-handling functions.

GCPtr stores the address of the memory to which it points in **addr**. If **addr** points to an allocated array, then **isArray** will be **true**, and the length of the array will be stored in **arraySize**.

The **first** field is a **static** variable that is initially set to **true**. It is a flag that the **GCPtr** constructor uses to know when the first **GCPtr** object is created. After the first **GCPtr** object is constructed, **first** is set to false. It is used to register a termination function that will be called to shut down the garbage collector when the program ends.

The findPtrInfo( ) Function

GCPtr declares one private function: **findPtrInfo( )**. This function searches **gclist** for a specified address and returns an iterator to its entry. If the address is not found, an iterator to the end of **gclist** is returned. This function is used internally by **GCPtr** to update the reference counts of the objects in **gclist**. It is implemented as shown here:

```
// 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;
}
```

The GCIterator typedef

At the start of the public section of **GCPtr** is the **typedef** of **Iter<T>** to **GCiterator**. This **typedef** is bound to each instance of **GCPtr**, thus eliminating the need to specify the type parameter each time an **Iter** is needed for a specific version of **GCPtr**. This simplifies the declaration of an iterator. For example, to obtain an iterator to the memory pointed to by a specific **GCPtr**, you can use a statement like this:

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

The GCPtr Constructor

The constructor for **GCPtr** is shown here:

```
// 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( ) allows both initialized and uninitialized instances to be created. If an initialized instance is declared, then the memory to which this **GCPtr** will point is passed through **t**. Otherwise, **t** will be null. Let's examine the operation of **GCPtr( )** in detail.

First, if **first** is true, it means that this is the first **GCPtr** object to be created. If this is the case, then **shutdown( )** is registered as a termination function by calling **atexit( )**. The **atexit( )** function is part of the standard C++ function library, and it registers a function that will be called when a program terminates. In this case, **shutdown( )** releases any memory that was prevented from being released because of a circular reference.

Next, a search of **gclist** is made, looking for any preexisting entry that matches the address in **t**. If one is found, then its reference count is incremented. If no preexising entry matches **t**, a new **GCInfo** object is created that contains this address, and this object is added to **gclist**.

GCPtr( ) then sets **addr** to the address specified by **t** and sets the values of **isArray** and **arraySize** appropriately. Remember, if you are allocating an array, you must explicitly specify the size of the array when you declare the **GCPtr** pointer that will point to it. If you don't, the memory won't be released correctly, and, in the case of an array of class objects, the destructors won't be called properly.

The GCPtr Destructor

The destructor for **GCPtr** is shown here:

```
// 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.
}
```

Garbage collection takes place each time a **GCPtr** goes out of scope. This is handled by **~GCPtr( )**. First, a search of **gclist** is made, looking for the entry that corresponds to the address pointed to by the **GCPtr** being destroyed. Once found, its **refcount** field is decremented. Next, **~GCptr( )** calls **collect( )** to release any unused memory (that is, memory whose reference count is zero).

As the comment at the end of **~GCPtr( )** states, for real applications, it is probably better to collect garbage less often than each time a single **GCPtr** goes out of scope. Collecting less frequently will usually be more efficient. As explained earlier, collecting each time a **GCPtr** is destroyed is useful for demonstrating the garbage collector because it clearly illustrates the garbage collector's operation.

Collect Garbage with collect( )

The **collect( )** function is where garbage collection takes place. It is shown here:

```
// 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;
}
```

The **collect( )** function works by scanning the contents of **gclist**, looking for entries that have a **refcount** of zero. When such an entry is found, it is removed from **gclist** by calling the **remove( )** function, which is a member of the STL **list** class. Then the memory associated with that entry is freed.

Recall that in C++, single objects are freed by **delete**, but arrays of objects are freed via **delete[ ]**. Thus, the value of the entry's **isArray** field determines whether **delete** or **delete[ ]** is used to free the memory. This is one reason you must specify the size of an allocated array for any **GCPtr** that will point to one: it causes **isArray** to be set to **true**. If **isArray** is not set correctly, it is impossible to properly release the allocated memory.

Although the point of garbage collection is to recycle unused memory automatically, you can take a measure of manual control if necessary. The **collect( )** function can be called directly by user code to request that garbage collection take place. Notice that it is declared as a **static** function within **GCPtr**. This means that it can be invoked without reference to any object. For example:

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

This causes **gclist<int>** to be collected. Because there is a different **gclist** for each type of pointer, you will need to call **collect( )** for each list that you want to collect. Frankly, if you need to closely manage the release of dynamically allocated objects, you are better off using the manual allocation system provided by C++. Directly calling **collect( )** is best reserved for specialized situations, such as when free memory is running unexpectedly low.

The Overloaded Assignment Operators

GCPtr overloads **operator=( )** twice: once for the assignment of a new address to a **GCPtr** pointer, and once for the assignment of one **GCPtr** pointer to another. Both versions are shown here:

```
// 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;
}
```

The first overload of **operator=( )** handles assignments in which a **GCPtr** pointer is on the left and an address is on the right. For example, it handles cases like the one shown here:

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

Here, the address returned by **new** is assigned to **p**. When this happens, **operator=(T *t)** is called with the new address passed to **t**. First, the entry in **gclist** for the memory that is currently being pointed to is found, and its reference count is decremented. Next, a search of **gclist** is made for the new address. If it is found, its reference count is incremented. Otherwise, a new **GCInfo** object is created for the new address, and this object is added to **gclist**. Finally, the new address is stored in the invoking object's **addr**, and this address is returned.

The second overload of the assignment operator, **operator=(GCPtr &rv)**, handles the following type of assignment:

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

Here, both **p** and **q** are **GCPtr** pointers, and **p** is assigned to **q**. This version of the assignment operator works much like the other. First, the entry in **gclist** for the memory that is currently being pointed to is found, and its reference count is decremented. Next, a search of **gclist** is made for the new address, which is contained in **rv.addr**, and its reference count is incremented. Then the invoking object's **addr** field is set to the address contained in **rv.addr**. Finally, the right-hand object is returned. This allows a chain of assignments to take place, such as:

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

There is one other important point to make about the way that the assignment operators work. As mentioned earlier in this chapter, it is technically possible to recycle memory as soon as its reference count drops to zero, but doing so puts an extra overhead on each pointer operation. This is why, in the overloaded assignment operators, the reference count for the memory previously pointed to by the left-hand operand is simply decremented, and no further action is taken. Thus, the management overhead associated with actually releasing memory and performing maintenance on **gclist** is avoided. These actions are deferred to later, when the garbage collector runs. This approach makes for faster runtimes for the code that uses a **GCPtr**. It also lets garbage collection take place at times that are (potentially) more convenient in terms of runtime performance.

The GCPtr Copy Constructor

Because of the need to keep track of each pointer to allocated memory, the default copy constructor (which makes a bitwise identical copy) cannot be used. Instead, **GCPtr** must define is own copy constructor, which is shown here:

```
// 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
}
```

Recall that a class' copy constructor is invoked when a copy of an object is required, such as when an object is passed as an argument to a function, when an object is returned from a function, or when one object is used to initialize another. **GCPtr**'s copy constructor duplicates the information contained in the original object. It also increments the reference count associated with the memory pointed to by the original object. When the copy goes out of scope, this reference count will be decremented.

Actually, the extra work performed by the copy constructor is not usually necessary because the overloaded assignment operators properly maintain the garbage collection list in most cases. However, there are a small number of cases in which the copy constructor is needed, such as when memory is allocated inside a function and a **GCPtr** to that memory is returned.

The Pointer Operators and Conversion Function

Because **GCPtr** is a pointer type, it must overload the pointer operators ***** and **–>**, plus the indexing operator **[ ]**. This is done by the functions shown here. Given that it is the ability to overload these operators that makes it possible to create a new pointer type, it is amazing how simple they are.

```
// 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];
}
```

The **operator*( )** function returns a reference to the object pointed to by the **addr** field of the invoking **GCPtr**, **operator–>( )** returns the address contained in **addr**, and **operator[ ]** returns a reference to the element specified by the index. **operator[ ]** should be used only on **GCPtr**s that point to allocated arrays.

As mentioned earlier, no pointer arithmetic is supported. For example, neither the **++** nor the **– –** operator is overloaded for **GCPtr**. The reason is that the garbage collection mechanism assumes that a **GCPtr** is pointing to *the start of* allocated memory. If a **GCPtr** could be incremented, for example, then when that pointer was garbage collected, the address used with **delete** would be invalid.

You have two options if you need to perform operations that involve pointer arithmetic. First, if the **GCPtr** is pointing to an allocated array, you can create an **Iter** that will let you cycle through that array. This approach is described later. Second, you can convert a **GCPtr** into a normal pointer by use of the **T *** conversion function defined by **GCPtr**. This function is shown here:

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

This function returns a normal pointer that points to the address stored in **addr**. It can be used like this.

```
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
```

In the preceding example, because **p** is a normal pointer, it can be manipulated in any way that any other pointer can. Of course, whether such manipulations yield meaningful results is dependent upon your application.

The main advantage of the conversion to **T *** is that it lets you use **GCPtr**s in place of normal C++ pointers when working with preexisting code that requires such pointers. For example, consider this sequence:

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

Here, **str** is a **GCPtr** pointer to **char** that is used in a call to **strcpy( )**. Because **strcpy( )** is expecting its arguments to be of type **char ***, the conversion to **T *** inside **GCPtr** is automatically invoked because, in this case, **T** is **char**. The same conversion is automatically invoked when **str** is used in the **cout** statement. Thus, the conversion function enables **GCPtr**s to seamlessly integrate with existing C++ functions and classes.

Keep in mind that the **T *** pointer returned by this conversion does not participate in or affect garbage collection. Thus, it is possible for the allocated memory to be freed even if a regular C++ pointer is still pointing to it. So, use the conversion to **T *** wisely—and infrequently.

The begin( ) and end( ) Functions

The **begin( )** and **end( )** functions, shown next, are similar to their counterparts in the STL:

```
// 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);
}
```

The **begin( )** function returns an **Iter** to the start of the allocated array pointed to by **addr**. The **end( )** function returns an **Iter** to one past the end of the array. Although there is nothing that stops these functions from being called on a **GCPtr** that points to a single object, their purpose is to support operations on allocated arrays. (Obtaining an **Iter** to a single object is not harmful, just pointless.)

The shutdown( ) Function

If a program creates a circular reference of **GCPtr**s, then when the program ends there will still be dynamically allocated objects that need to be released. This is important because these objects might have destructors that need to be called. The **shutdown( )** function handles this task. This function is registered by **atexit( )** when the first **GCPtr** is created, as described earlier. This means that it will be called when the program terminates.

The **shutdown( )** function is shown here:

```
// 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
}
```

First, if the list is empty, which it will normally be, then **shutdown( )** simply returns. Otherwise, it sets to zero the reference counts of the entries that still exist in **gclist**, and then it calls **collect( )**. Recall that **collect( )** releases any object that has a reference count of zero. Thus, setting the reference counts to zero ensures that all objects will be freed.

Two Utility Functions

GCPtr ends by defining two utility functions. The first is **gclistSize( )**, which returns the number of entries currently held in **gclist**. The second is **showlist( )**, which displays the contents of **gclist**. Neither of these are necessary for the implementation of a garbage collection pointer type, but they are useful when you want to watch the operation of the garbage collector.

GCInfo

The garbage collection list in **gclist** holds objects of type **GCInfo**. The **GCInfo** class is shown here:

```
// 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;
  }
};
```

As mentioned earlier, each **GCInfo** object stores a pointer to allocated memory in **memPtr** and the reference count associated with that memory in **refcount**. If the memory pointed to by **memPtr** contains an array, then the length of that array must be specified when the **GCInfo** object is created. In this case, **isArray** is set to **true**, and the length of the array will be stored in **arraySize**.

GCInfo objects are stored in an STL **list**. To enable searches on this list, it is necessary to define **operator==( )**, as shown here:

```
// 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);
}
```

Two objects are equal only if both their **memPtr** fields are identical. Depending upon the compiler you are using, other operators may need to be overloaded to enable **GCInfo**s to be stored in an STL **list**.

Iter

The **Iter** class implements an iterator-like object that can be used to cycle through the elements of an allocated array. **Iter** is not technically necessary because a **GCPtr** can be converted to a normal pointer of its base type, but **Iter** offers two advantages. First, it lets you cycle through an allocated array in a fashion similar to the way in which you cycle through the contents of an STL container. Thus, the syntax for using an **Iter** is familiar. Second, **Iter** will not allow out-of-range accesses. Thus, an **Iter** is a safe alternative to using a normal pointer. Understand, however, that **Iter** does not participate in garbage collection. Thus, if the underlying **GCPtr** on which an **Iter** is based goes out of scope, the memory to which it points will be freed whether or not it is still needed by that **Iter**.

Iter is a template class defined like this:

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

The type of data to which the **Iter** points is passed through **T**.

Iter defines these instance variables:

```
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
```

The address to which the **Iter** currently points is held in **ptr**. The address to the start of the array is stored in **begin**, and the address of an element one past the end of the array is stored in **end**. The length of the dynamic array is stored in **length**.

Iter defines the two constructors shown here. The first is the default constructor. The second constructs an **Iter**, given an initial value for **ptr**, and pointers to the beginning and end of the array.

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

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

For use by the garbage collector code shown in this chapter, the initial value of **ptr** will always equal **begin**. However, you are free to construct **Iter**s in which the initial value of **ptr** is a different value.

To enable **Iter**'s pointer-like nature, it overloads the ***** and **–>** pointer operators, and the array indexing operator **[ ]**, as shown here:

```
// 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];
}
```

The * operator returns a reference to the element currently being pointed to in the dynamic array. The **–>** returns the address of the element currently being pointed to. The **[ ]** returns a reference to the element at the specified index. Notice that these operations do not allow an out-of-bounds access. If one is attempted, an **OutOfRangeExc** exception is thrown.

Iter defines the various pointer arithmetic operators, such as ++, – –, and so on, which increment or decrement an **Iter**. These operators enable you to cycle through a dynamic array. In the interest of speed, none of the arithmetic operators perform range checks themselves. However, any attempt to access an out-of-bounds element will cause an exception, which prevents a boundary error. **Iter** also defines the relational operators. Both the pointer arithmetic and relational functions are straightforward and easy to understand.

Iter also defines a utility function called **size( )**, which returns the length of the array to which the **Iter** points.

As mentioned earlier, inside **GCPtr**, **Iter<T>** is **typedef**ed to **GCiterator** for each instance of **GCPtr**, which simplifies the declaration of an iterator. This means that you can use the type name **GCiterator** to obtain the **Iter** for any **GCPtr**.

How to Use GCPtr

Using a **GCPtr** is quite easy. First, include the file **gc.h**. Then, declare a **GCPtr** object, specifying the type of the data to which it will point. For example, to declare a **GCPtr** object called **p** that can point to an **int**, use this kind of declaration:

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

Next, dynamically allocate the memory using **new** and assign the pointer returned by **new** to **p**, as shown here:

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

You can assign a value to the allocated memory using an assignment operation like this:

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

Of course, you can combine the three preceding statements like this:

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

You can obtain the value of the memory at the location pointed to by **p**, as shown here:

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

As these examples show, in general you use a **GCPtr** just like a normal C++ pointer. The only difference is that you don't need to delete the pointer when you are through with it. The memory allocated to that pointer will be automatically released when it is no longer needed.

Here is an entire program that assembles the pieces just shown:

```
#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;
}
```

The output from this program with the display option turned on is shown here. (Recall that you can watch the operation of the garbage collector by defining **DISPLAY** within **gc.h**.)

```
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 --
```

When the program ends, **p** goes out of scope. This causes its destructor to be called, which causes the reference count for the memory pointed to by **p** to be decremented. Because **p** was the only pointer to this memory, this operation sets the reference count to zero. Next, **p**'s destructor calls **collect( )**, which scans **gclist**, looking for entries that have a reference count of zero. Because the entry previously associated with **p** has a reference count of zero, its memory is freed.

One other point: notice that prior to garbage collection, a null pointer entry is also in **gclist**. This null pointer was created when **p** was constructed. Recall that if a **GCPtr** is not given an initial address, the null address (which is zero) is used. Although it is not technically necessary to store a null pointer in **gclist** (because it is never freed), doing so simplifies other parts of **GCPtr** because it ensures that every **GCPtr** has a corresponding entry in **gclist**.

Handling Allocation Exceptions

As the preceding program shows, because the garbage collector does not change the way that memory is allocated via **new**, you handle allocation failures in the same way as usual, by catching the **bad_alloc** exception. (Recall that when **new** fails, it throws an exception of type **bad_alloc**.) Of course, the preceding program won't run out of memory, and the **try**/**catch** block isn't really needed, but a real-world program might exhaust the heap. Thus, you should always check for this possibility.

In general, the best way to respond to a **bad_alloc** exception when using garbage collection is to call **collect( )** to recycle any unused memory and then retry the allocation that failed. This technique is employed by the load-testing program shown later in this chapter. You can use the same basic technique in your own programs.

A More Interesting Example

Here is a more interesting example that shows the effect of a **GCPtr** going out of scope before the end of the program:

```
// 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;
}
```

This program produces the following output when the display option is turned on:

```
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 --
```

Examine this program and its output closely. First, notice that **p** and **q** are created at the start of **main( )**, but **r** is not created until its block is entered. As you know, in C++, local variables are not created until their block is entered. When **r** is created, the memory to which it points is given an initial value of 12. This value is then displayed, and the block ends. This causes **r** to go out of scope, which means that its destructor is called. This causes **r**'s reference count in **gclist** to be decremented to zero. Then **collect( )** is called to collect garbage.

Because the display option is on, when **collect( )** begins, it displays the contents of **gclist**. Notice that it has four entries. The first is the one that was previously linked to **r**. Notice that its **refcount** field is zero, indicating that the memory pointed to by the **memPtr** field is no longer in use by any program element. The next two entries are still active, and they are linked to **p** and **q**. Because they are still in use, the memory they point to is not freed at this time. The final entry represents the null pointer to which **p** and **q** originally pointed when they were created. Because it is no longer in use, it will be removed from the list by **collect( )**. (Of course, no memory is freed when the null pointer is removed.)

Because no other **GCPtr** points to the same memory as **r**, its memory can be released, as the **Deleting: 12** line confirms. Once this is done, program execution continues after the block. Finally, **p** and **q** go out of scope when the program ends and their memory is released. In this case, **q**'s destructor is called first, meaning that it is collected first. Finally, **p** is destroyed and **gclist** is empty.

Allocating and Discarding Objects

It is important to understand that memory becomes subject to garbage collection as soon as its reference count drops to zero (which means that no **GCPtr** is pointing to it). It is not necessary for the **GCPtr** that originally pointed to that object to go out of scope. Thus, you can use a single **GCPtr** object to point to any number of allocated objects by simply assigning that **GCPtr** a new value. The discarded memory will eventually be collected. For example:

```
// 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;
}
```

The output from this program, with the display option turned on, is shown here:

```
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 --
```

In the program, **p**, a **GCPtr** to **int**, is assigned a pointer to four separate chunks of dynamic memory, each being initialized with a different value. Next, a call is made to **collect( )**, which forces garbage collection to take place. Notice the contents of **gclist**: three of the entries are marked inactive, and only the entry that points to the memory that was allocated last is still in use. Next, the unused entries are deleted. Finally, the program ends, **p** goes out of scope, and the final entry is removed.

Notice that the first three chunks of dynamic memory to which **p** pointed have reference counts of zero. This is because of the way the overloaded assignment operator works. Recall that when a **GCPtr** is assigned a new address, the reference count for its original value is decremented. Thus, each time **p** is assigned the address of a new integer, the reference count for the old address is reduced.

One other point: because **p** was initialized when it was declared, no null-pointer entry was generated and put on **gclist**. Remember, a null-pointer entry is created only when a **GCPtr** is declared without an initial value.

Allocating Arrays

If you are allocating an array using **new**, then you must tell **GCPtr** this fact by specifying its size when the **GCPtr** pointer to that array is declared. For example, here is the way to allocate an array of five **double**s:

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

The size must be specified for two reasons. First, it tells the **GCPtr** constructor that this object will point to an allocated array, which causes the **isArray** field to be set to **true**. When **isArray** is **true**, the **collect( )** function frees memory by using **delete[ ]**, which releases a dynamically allocated array, rather than **delete**, which releases only a single object. Therefore, in this example, when **pda** goes out of scope, **delete[ ]** is used and all five elements of **pda** are freed. Ensuring that the correct number of objects are freed is especially important when arrays of class objects are allocated. Only by using **delete[ ]** can you know that the destructor for each object will be called.

The second reason that the size must be specified is to prevent an out-of-bounds element from being accessed when an **Iter** is used to cycle through an allocated array. Recall that the size of the array (stored in **arraySize**) is passed by **GCPtr** to **Iter**'s constructor whenever an **Iter** is needed.

Be aware that nothing enforces the rule that an allocated array be operated on only through a **GCPtr** that has been specified as pointing to an array. This is solely your responsibility.

Once you have allocated an array, there are two ways you can access its elements. First, you can index the **GCPtr** that points to it. Second, you can use an iterator. Both methods are shown here.

Using Array Indexing

The following program creates a **GCPtr** to a 10-element array of **int**s. It then allocates that array and initializes it to the values 0 through 9. Finally, it displays those values. It performs these actions by indexing the **GCPtr**.

```
// 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;
}
```

The output, with the display option off, is shown here:

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

Because a **GCPtr** emulates a normal C++ pointer, no array bounds checking is performed, and it is possible to overrun or under run the dynamically allocated array. So, use the same care when accessing an array through a **GCPtr** as you do when accessing an array through a normal C++ pointer.

Using Iterators

Although array indexing is certainly a convenient method of cycling through an allocated array, it is not the only method at your disposal. For many applications, the use of an iterator will be a better choice because it has the advantage of preventing boundary errors. Recall that for **GCPtr**, iterators are objects of type **Iter**. **Iter** supports the full complement of pointer operations, such as **++**. It also allows an iterator to be indexed like an array.

Here is the previous program reworked to use an iterator. Recall that the easiest way to obtain an iterator to a **GCPtr** is to use **GCiterator**, which is a **typedef** inside **GCPtr** that is automatically bound to the generic type **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;
}
```

On your own, you might want to try incrementing **itr** so that it points beyond the boundary of the allocated array. Then try accessing the value at that location. As you will see, an **OutOfRangeExc** is thrown. In general, you can increment or decrement an iterator any way you like without causing an exception. However, if it is not pointing within the underlying array, attempting to obtain or set the value at that location will cause a boundary error.

Using GCPtr with Class Types

GCPtr is used with class types in just the same way it is used with built-in types. For example, here is a short program that allocates objects of **MyClass**:

```
// 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;
}
```

Notice how the members of **MyClass** are accessed through the use of the **–>** operator. Remember, **GCPtr** defines a pointer type. Thus, operations through a **GCPtr** are performed in exactly the same fashion that they are with any other pointer.

The output from the program, with the display option turned off, is shown here:

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

Pay special attention to the last two lines. These are output by **~MyClass( )** when garbage is collected. Even though only one **GCPtr** pointer was created, two **MyClass** objects were allocated. Both of these objects are represented by entries in the garbage collection list. When **ob** is destroyed, **gclist** is scanned for entries having a reference count of zero. In this case, two such entries are found, and the memory to which they point is deleted.

A Larger Demonstration Program

The following program shows a larger example that exercises all of the features of **GCPtr**:

```
// 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;
}
```

Here is the output with the display option turned off:

```
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)
```

On your own, try compiling and running this program with the display option turned on. (That is, define **DISPLAY** in **gc.h**.) Next, walk through the program, matching the output against each statement. This will give you a good feel for the way the garbage collector works. Remember, garbage collection occurs whenever a **GCPtr** goes out of scope. This happens at various points in the program, such as when a function that receives a copy of a **GCPtr** returns. In this case, the copy goes out of scope and garbage collection takes place. Also remember that each type of **GCPtr** maintains its own **gclist**. Thus, collecting garbage from one list does not cause it to be collected from other types of lists.

Load Testing

The following program load tests **GCPtr** by repeatedly allocating and discarding objects until free memory is exhausted. When this occurs, a **bad_alloc** exception is thrown by **new**. Inside the exception handler, **collect( )** is explicitly called to reclaim the unused memory, and the process continues. You can use this same technique in your own programs.

```
// 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;
}
```

A portion of the output from the program (with the display option off) is shown here. Of course, the precise output you see may differ because of the amount of memory available in your system and the compiler that you are using.

```
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
```

Some Restrictions

There are a few restrictions to using **GCPtr**:

1. You cannot create global **GCPtr**s. Recall that a global object goes out of scope after the rest of the program ends. When a global **GCPtr** goes out of scope, the **GCPtr** destructor calls **collect( )** to try to release the unused memory. The trouble is that, depending on how your C++ compiler is implemented, **gclist** may have already been destroyed. In this case, executing **collect( )** will cause a runtime error. Therefore, **GCPtr** should be used only when creating local objects.
2. When using dynamically allocated arrays, you must specify the size of the array when you declare a **GCPtr** that will point to it. There is no mechanism that enforces this, however, so be careful.
3. You must not attempt to release the memory pointed to by a **GCPtr** by explicitly using **delete**. If you need to immediately release an object, call **collect( )**.
4. A **GCPtr** object must point only to memory that is dynamically allocated via **new**. Assigning to a **GCPtr** object a pointer to any other memory will cause an error when the **GCPtr** object goes out of scope because an attempt will be made to free memory that was never allocated.
5. It is best to avoid circular pointer references for reasons described earlier in this chapter. Although all allocated memory is eventually released, objects containing circular references remain allocated until the program ends, rather than being released when they are no longer used by another program element.

Some Things to Try

It is easy to tailor **GCPtr** to the needs of your applications. As explained earlier, one of the changes that you might want to try is collecting garbage only after some metric has been reached, such as **gclist** reaching a certain size, or after a certain number of **GCPtr**s have gone out of scope.

An interesting enhancement to **GCPtr** is to overload **new** so that it automatically collects garbage when an allocation failure occurs. It is also possible to bypass the use of **new** when allocating memory for a **GCPtr** and use factory functions defined by **GCPtr** instead. Doing this lets you carefully control the dynamic allocation of memory, but it makes the allocation process fundamentally different than it is for C++'s built-in approach.

You might want to experiment with other solutions to the circular reference problem. One way is to implement the concept of a *weak reference*, which does not prevent garbage collection from occurring. You would then use a weak reference whenever a circular reference was needed.

Perhaps the most interesting variation on **GCPtr** is found in Chapter 3. There, a multithreaded version is created in which garbage collection takes place automatically, when free CPU time exists.