Sharing Pointers and Garbage Collection

Steven J Zeil

September 19, 2013

Contents

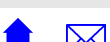
Swearing by Sharing

We've talked a lot about using pointers to share information, but mainly as something that causes problems.

- We have a pretty good idea of how to handle ourselves when we have pointers among our data members and *don't* want to share.

  In that case, we rely on implementing our own deep copying so that every "container" has distinct copies of all of its components.

```
class Catalog {
   ⋮
   Catalog (const Catalog& c);
   Catalog& operator= (const Catalog& c);
   ~Catalog();
   ⋮
private:
  Book* allBooks;      // array of books
  int numBooks;
};


Catalog::Catalog (const Catalog& c)
: numBooks(c.numBooks)
{
   allBooks = new Book[numBooks];
   copy (c.allBooks, c.allBooks+numBooks, allBoooks);
}

Catalog& Catalog::operator= (const Catalog& c)
{
  if (*this != c)
    {
      delete [] allBooks;
      numBooks = c.numBooks;
      allBooks = new Book[numBooks];
```

```
      copy (c.allBooks, c.allBooks+numBooks, allBoooks);
    }
  return *this;
}

Catalog::~Catalog()
{
  delete [] allBooks;
}
```

For some data structures, this is OK. If we are using a pointer mainly to give us access to a dynamically allocated array, we can copy the entire array as necessary. In the example shown here, we would want each catalog to get its own distinct array of books. So we would implement a deep copy for the assignment operator and copy constructor, and delete the allBooks pointer in the destructor.

- But not every data structure can be treated this way.
  - Sometimes, sharing is essential to the behavior of the data structure that we want to implement.
  - (In data structures, there is an entire class of structures and algorithms associated with "graphs" that exhibit this property.)

. . . . . .

1 Shared Structures

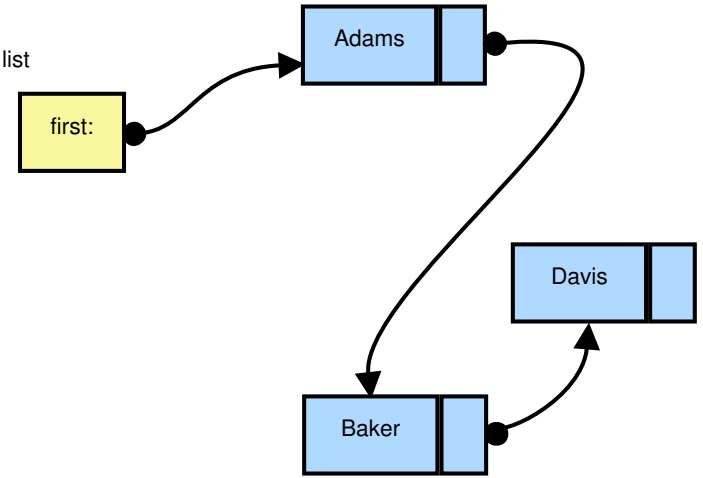
Shared Structures

In this section, we will introduce three examples that we will explore further in the remainder of the lesson. All three involve some degree of essential sharing.

. . . . . .

1.1 Singly Linked Lists

Singly Linked Lists



We'll start with a fairly prosaic example. In its simplest form, a singly linked list involves no sharing, and so we could safely treat all of its components as deep-copied.

.....

SLL Destructors

In particular, we can take a simple approach of writing the destructors - if you have a pointer, delete it:

```
struct SLNode {
   string data;
   SLNode* next;
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first;
public:
```

```
    ⋮
  ~List() {delete first;}
};
```

Problem: stack size is $O(N)$ where N is the length of the list.

. . . . . .

If a *List* object gets destroyed, its destructor will delete its *first* pointer. That node (Adams in the picture) will have its destructor called as part of the delete, and it will delete its pointer to Baker. The Baker node's destructor will delete the pointer to Davis. At then end, we have successfully recovered all on-heap memory (the nodes) with no problems.

Now, this isn't really ideal. At the time the destructor for Davis is called, there are still partially executed function activations for the Baker and Adams destructors and for the list's destructor still on the call stack, waiting to finish. That's no big deal with only three nodes in the list, but if we had a list of, say, 10000 nodes, then we might not have enough stack space for 10000 uncompleted calls. So, typically, we would actually use a more aggressive approach with the list itself:

Destroy the List, not the Nodes

```
struct SLNode {
  string data;
  SLNode* next;
    ⋮
  ~SLNode () {/* do nothing */}
};

class List {
  SLNode* first;
public:
    ⋮
  ~List()
   {
     while (first != 0)
       {
         SLNode* next = first->next;
         delete first;
```

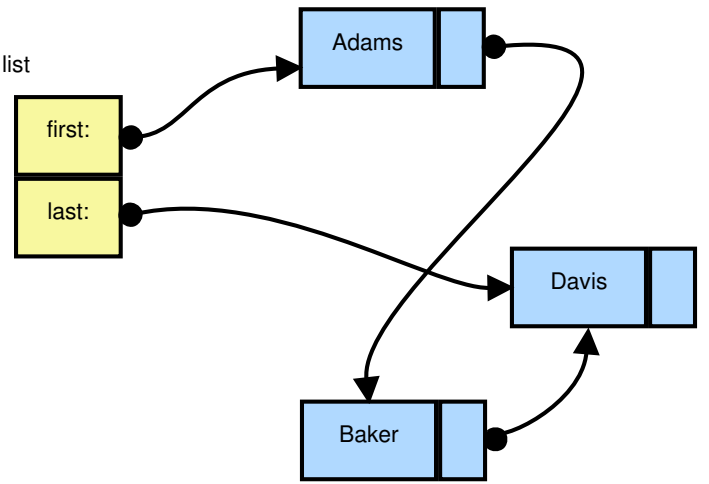
```
            first = next;
        }
    }
};
```

This avoids stacking up large numbers of recursive calls.

. . . . . .

But, if we weren't worried about stack size, then our first approach will be fine.

First-Last Headers



But now let's consider one of the more common variations on linked lists.

- If our header contains pointers to both the first and last nodes of this list, then we can do O(1) insertions at both ends of this list.

- Notice, however, that the final node in the list is now "shared" by both the list header and the next-to-last node.

. . . . . .

FLH SLL Destructor

So, if we were to extend our basic approach of writing destructors that simply delete their pointers:

```
struct SLNode {
   string data;
   SLNode* next;
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first;
   SLNode* last;
public:
     ⋮
   ~List() {delete first; delete last;}
};
```

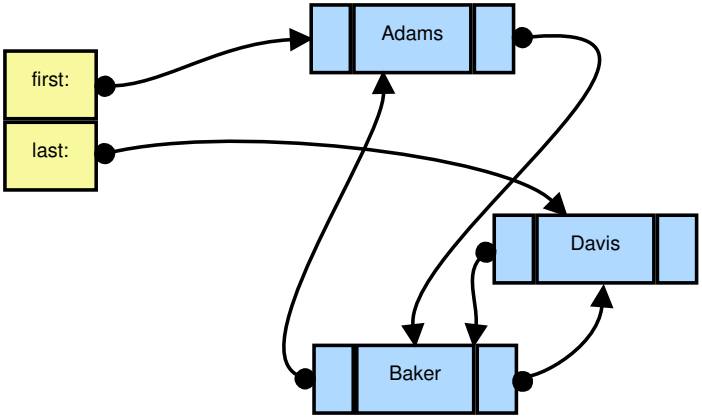
Then, when a list object is destroyed, the final node in the list will actually be deleted twice. Deleting the same block of memory twice can corrupt the heap (by breaking the structure of the free list) and eventually cause the program to fail.

. . . . . .

1.2 Doubly Linked Lists

Doubly Linked Lists

Now, let's make things just a little *more* difficult.
If we consider doubly linked lists, our straightforward approach of "delete everything" is really going to be a problem.



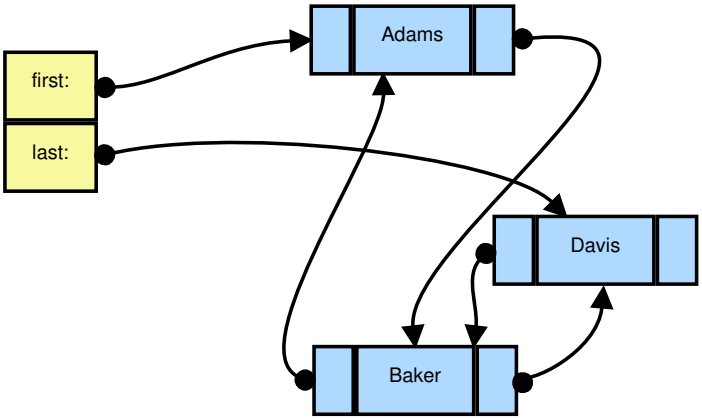
.....

DLL Aggressive Deleting

```
struct DLNode {
   string data;
   DLNode* prev;
   DLNode* next;
     ⋮
   ~DLNode () {delete prev; delete next;}
};

class List {
   DLNode* first;
   DLNode* last;
public:
     ⋮
   ~List() {delete first; delete last;}
};
```

.....

Deleting the DLL



- When a list object is destroyed, it will start by deleting the *first* pointer.
  - That node (Adams) will delete its *next* pointer (pointing to Baker).
  - That second node will delete its *prev* pointer (Adams).

- Now we've deleted the same node twice, potentially corrupting the heap.
  - But, worse, the Adam node's destructor will be invoked again.
  - It will delete its *next* pointer, and we will have deleted the Baker node a second time.
- Then the Baker node deletes its *prev* pointer *again*.

. . . . . .

Deleting and Cycles

We're now in an infinite recursion,

- which will continue running until either the heap is so badly corrupted that we crash when trying to process a delete,
- or when we finally fill up the activati0on stack to its maximum possible size.

What makes this so much nastier than the singly linked list?

- It's the fact that not only are we doing sharing via pointers, but that the various connections form *cycles*, in which we can trace a path via pointers from some object eventually back to itself.

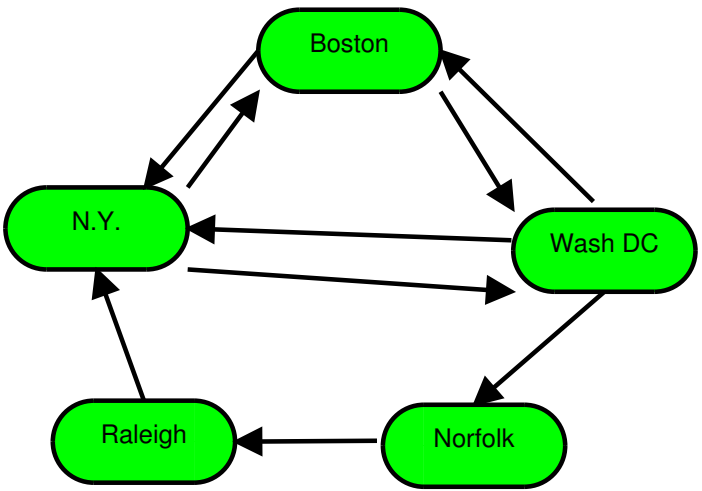
.....

1.3 Airline Connections

Airline Connections

Lest you think that this issue only arises in low-level data structures, let's consider how it might arise in programming at the application level.

This graph illustrates flight connections available from an airline.



.....

Aggressively Deleting a Graph

If we were to implement this airport graph with Big 3-style operations:

```
class Airport
{
```

```
   ⋮

private:
   vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
   for (int i = 0; i < hasFlightsTo.size(); ++i)
      delete hasFlightsTo[i];
}
```

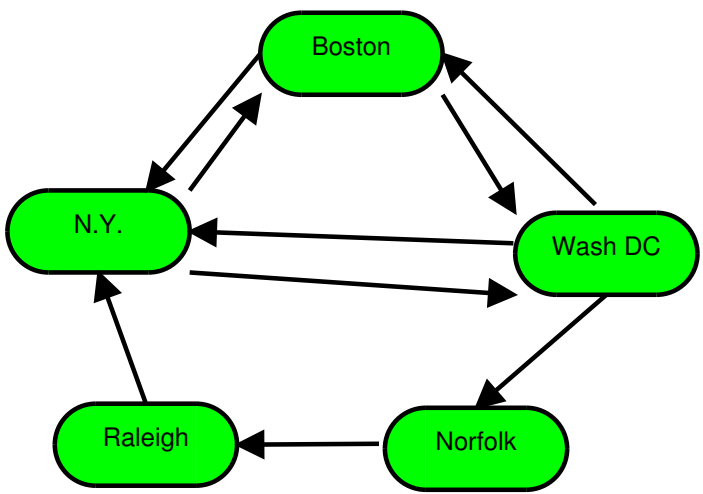
we would quickly run into a disaster.

.....

Deleting the Graph

Suppose that we delete the Boston airport.

- Its destructor would be invoked, which would delete the N.Y. airport and Wash DC airports.
  - Let's say, for the sake of example, that the NY airport is deleted first.
- The act of deleting the pointer to the NY airport causes its destructor to be invoked, which would delete the Boston and Wash DC airports.
  - But Boston has already been deleted.
    - If we don't crash right away, we will quickly wind up deleting Wash DC twice.
- In fact, we would wind up, again, in an infinite recursion among the destructors.



This should not be a big surprise. Looking at the graph, we can see that it is possible to form cycles. (In fact, if there is any node in this graph that *doesn't* participate in a cycle, there would be something very wrong with our airline. Either we would have planes piling up at some airport, unable to leave; or we would have airports that run out of planes and can't support any outgoing flights.

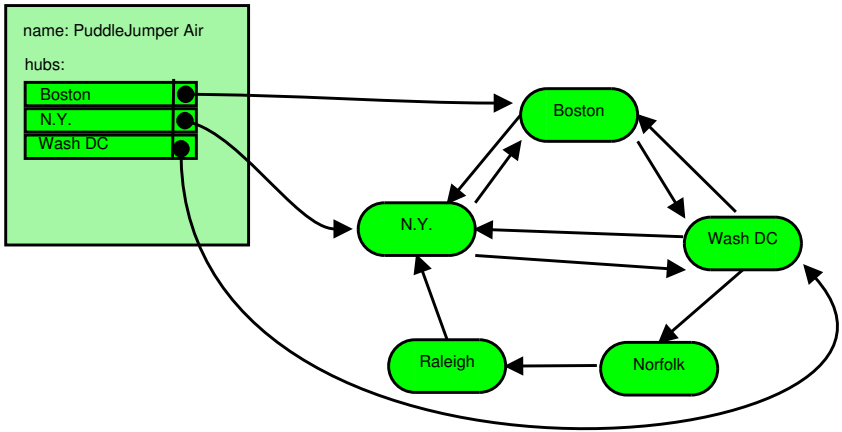
The Airline

Now, you might wonder just how or why we would have deleted that Boston pointer in the first place. So, let's add a bit of context.

• The airport graph is really a part of the description of an airline:

```
class AirLine {
   ⋮
   string name;
   map<string, Airport*> hubs;
};


AirLine::~Airline()
{
   for (map<string, Airport*>::iterator i = hubs.begin;
        i != hubs.end(); ++i)
     delete i->second;
}
```

The AirLine Structure



- The map *hubs* provides access to all those airports where planes are serviced and stored when not in flight, indexed by the airport name.
  - Not all airports are hubs.
- An airport that is not a hub is simply one where planes touch down and pick and discharge passengers while on their way to another hub.

Suppose that PuddleJumper Air goes bankrupt.

- It makes sense that when an airline object is destroyed, we would delete those hub pointers.
  - But we've seen that this is dangerous.

. . . . . .

Can We Do Better?

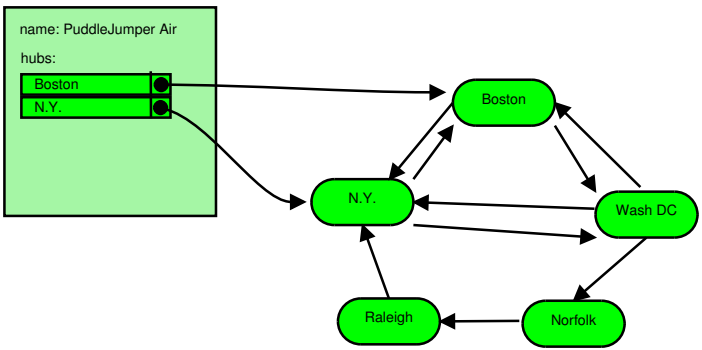
Now, that's a problem. But what makes this example particularly vexing is that it's not all that obvious what would constitute a better approach.

- Let's consider some other changes to the airline structure.

. . . . . .

Changing the Hubs

Suppose that Wash DC were to lose its status as a hub. Even though the pointer to it were removed from the *hubs* table, the Wash DC airport needs to remain in the map.

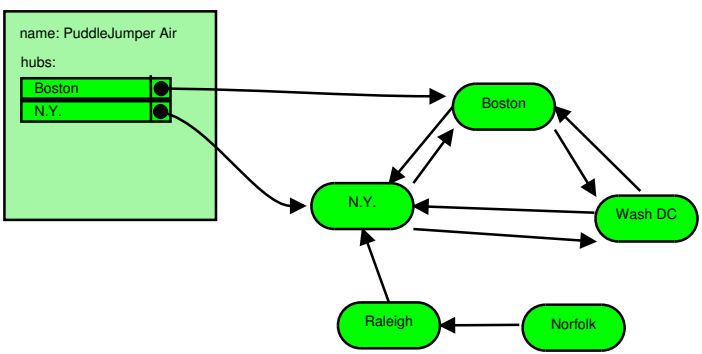


. . . . . .

Changing the Connections

On the other hand, if Wash DC were to drop its service to Norfolk, one might argue that Norfolk and Raleigh should then be deleted, as there would be no way to reach them.

- But how could you write code into your destructors and your other code that adds and removes pointers that could tell the difference between these two cases?



. . . . . .

2 Garbage Collection

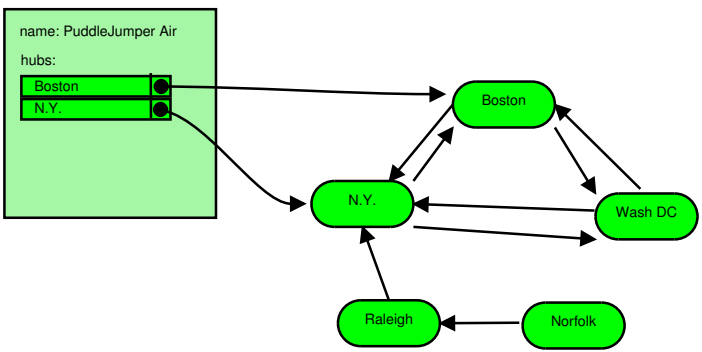
Garbage

Objects on the heap that can no longer be reached (in one or more hops) from any pointers in the activation stack (i.e., in local variables of active functions)or from any pointers in the static storage area (variables declared in C++ as "`static`")are called *garbage*.

. . . . . .

Garbage Example

- In this example, if we assume that the airline object is actually a local variable in some function, then Norfolk and Raleigh appear to be garbage.
  - Unless there's some other pointer not shown in this picture, there's no way to get to either of them.



- Being garbage is not the same thing as "nothing points to it".
  - Raleigh is garbage even though something is pointing to it. Nonetheless, there is no way to get to Raleigh from the non-heap storage.

. . . . . .

Garbage Collection

Determining when something on the heap has become garbage is sufficiently difficult that many programming languages take over this job for the programmer.

The runtime support system for these languages provides *automatic garbage collection*, a service that determines when an object on the heap has become garbage anf automatically *scavenges* (reclaims the storage of) such objects.

. . . . . .

Java has GC

In Java, for example, although Java and C++ look very similar, there is no "delete" operator.

Java programmers use lots of pointers[1], many more than the typical C++ programmer.

But Java programmers never worry about deleting anything. They just trust in the garbage collector to come along eventually and clean up the mess.

. . . . . .

C++ Does Not

Automatic garbage collection really can simplify a programmer's life. Sadly, C++ does not support automatic garbage collection.

But how is this magic accomplished (and why doesn't C++ support it)? That's the subject of the remainder of this section.

. . . . . .

2.1 Reference Counting

Reference Counting

Reference counting is one of the simplest techniques for implementing garbage collection.

- Keep a hidden counter in each object on the heap. The counter will indicate how many pointers to that object exist.

[1] Though, somewhat confusingly, they call them "references" instead of pointers. But they really are more like C++ pointers than like C++ references because you

- obtain one of these pointer/references by allocating an object on the heap via the operator `new`,
- can assign the value "null" to one of these to indicate that it isn't pointing at anything at all, and
- can assign a new address to one of these to make it point at some different object on the heap

All three of these properties are true of C++ pointers but not of C++ references. So Java "references" really are the equivalent of C++ "pointers", but by renaming them, Java advocates are able to boast that Java is a simpler language because it doesn't have pointers. That's more than a little disingenuous, IMO.

  - Each time we reassign a pointer that used to point at this object, we decrement the counter.
  - Each time we reassign a pointer so that it now points at this object, we increment the counter.

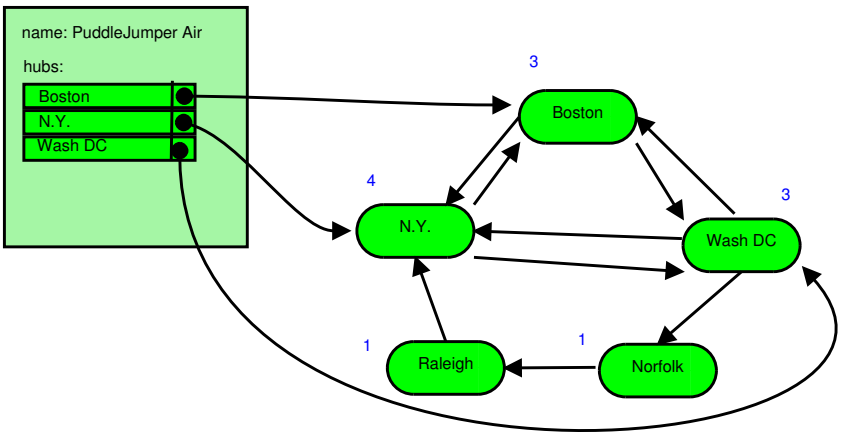
- If that counter ever reaches 0, scavenge the object.

.....

Reference Counting Example

For example, here's our airline example with reference counts. Now, suppose that Wash DC loses its hub status.

- The removal of the pointer from the airline object causes the reference count of Wash DC to decrease by one, but it's still more than zero, so we don't try to scavenge Wash DC.

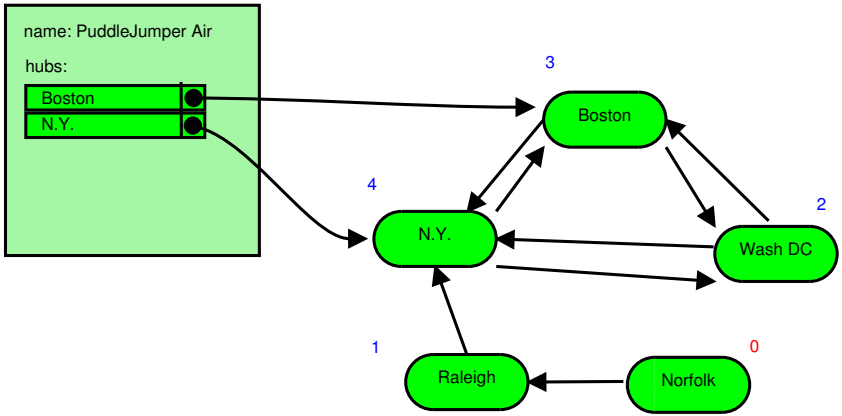


.....

Reference Counting Example II

Now, suppose that Wash DC drops its service to Norfolk

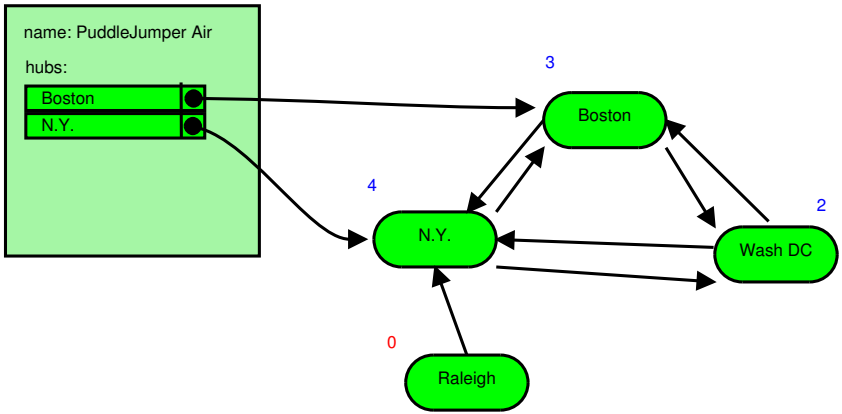
- When the pointer from Wash DC to Norfolk is removed, then the reference count of Norfolk decreases. In this case, it decreases to zero.



Reference Counting Example III

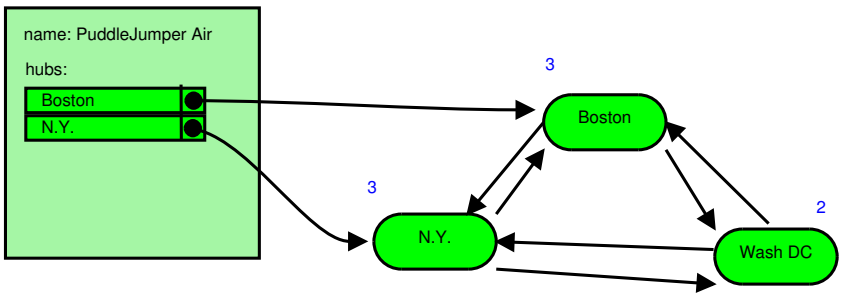
So the Norfolk object can be scavenged.

- When we do so, however, its pointer to Raleigh disappears, reducing Raleigh's reference count to zero.
  - That means that Raleigh can be scavenged.



Reference Counting Example IV

Doing that reduces N.Y.'s reference count, but the count stays above zero, so we don't try to scavenge N.Y.



Can we do this?

Implementing reference counting requires that we take control of pointers.

- To properly update reference counts, we would need to know whenever a pointer is assigned a new address (or null), whenever a pointer is created to point to a newly allocated object, and whenever a pointer is destroyed.

- Now, we can't do that for "real" pointers in C++.
  - But it is quite possible to create an ADT that looks and behaves much like a regular pointer.
  - And, by now, we know how to take control of assignment, copying, and destroying of ADT objects.

.....

A Reference Counted Pointer

Here is an (incomplete) sketch of a reference counted pointer ADT (which I will call a "smart pointer" for short).

```
template <class T>
class RefCountPointer {
  T* p;   ❶
  unsigned* count;

  void checkIfScavengable()  ❷
  {
    if (*count == 0)
      {
        delete count;
        delete p;
      }
  }

public:
  // This constructor is used to hand control of a newly
  // allocated object (*s) over to the reference count
  // system.  Example:
  //    RefCountPointer<PersonelRecord> p (new PersonelRecord());
  // It's critical that, once you create a reference counted
  // pointer to something, that you not continue playing with
  // regular pointers to the same object.
```


```
RefCountPointer (T* s)   ❸
  : p(s), count(new unsigned)
  {*count = 1;}

RefCountPointer (const RefCountPointer& rcp)
  : p(rcp.p), count(rcp.count)
  {++(*count);}   ❹

~RefCountPointer() {--(*count); checkIfScavengable();} ❺


RefCountPointer& operator= (const RefCountPointer& rcp)
  {
    ++(*(rcp.count));  ❻
    --(*count);
    checkIfScavengable();
    p = rcp.p;
    count = rcp.count;
    return *this;
  }

T& operator*() const {return *p;}  ❼
T* operator->() const {return p;}


bool operator== (const RefCountPointer<T>& ptr) const
{return ptr.p == p;}


bool operator!= (const RefCountPointer<T>& ptr) const
{return ptr.p != p;}
```


```
};
```

❶ The data stored for each of our smart pointers is p, the pointer to the real data, and count, a pointer to an integer counter for that data.

❷ This function will be called whenever we decrease the count. It checks to see if the count has reached zero and, if so, scavenges the object (and its counter).

❸ When the very first smart pointer is created for a newly allocated object, we set its counter to 1.

❹ When a smart pointer is copied, we increment its counter.

❺ When a smart pointer is destroyed, we decrement the count and see if we can scavenge the object.

❻ When a smart pointer is assigned a new value, we increment the counter of the object whose pointer is being copied, decrement the counter of the object whose pointer is being replaced, and check to see if that object can be scavenged.

❼ Other than that, the smart pointer has to behave like a real pointer, supporting the * and -> operators.

. . . . . .

Now, as I noted, this is an incomplete sketch. Right now, we have no means for dealing with null pointers, and we haven't provided the equivalent of a const pointer. Providing each of these would approximately double the amount of code to be shown (for a combined total of nearly four times what I've shown here). There's also some big issues when combining these smart pointers with inheritance and Object-Oriented programming.

But, hopefully, it's enough to serve as a proof of concept that this is really possible.

Is it worth the effort?

- Reference counting is fairly easy to implement.
  - Unlike the more advanced garbage collection techniques that we will look at shortly, it can be done in C++ because it does not require any special support from the compiler and the runtime system.

    There's a problem with reference counting, though.

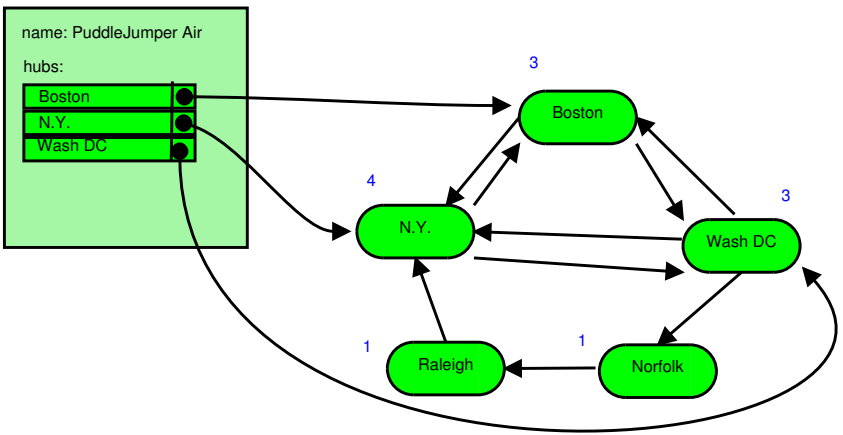
– One that's serious enough to make it unworkable in many practical situations.

. . . . . .

Disappearing Airline

Let's return to our original airline example, with reference counts.

- Assume that
  - the airline object itself is a local variable in a function and that
  - we are about to return from that function.



- That object will therefore be destroyed, and its reference counted pointers to the three hubs will disappear.

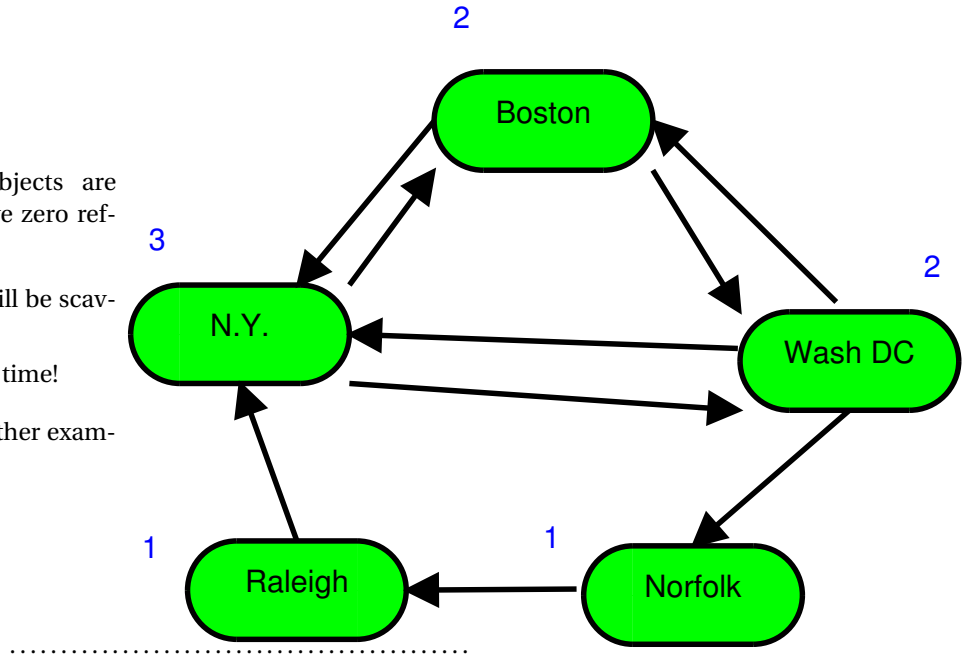
. . . . . .

Leaky Airports

Here is the result, with the updated reference counts.

- Now, all of these airport objects are garbage, but none of them have zero reference counts.
  - Therefore none of them will be scavenged.
  - We're leaking memory, big time!

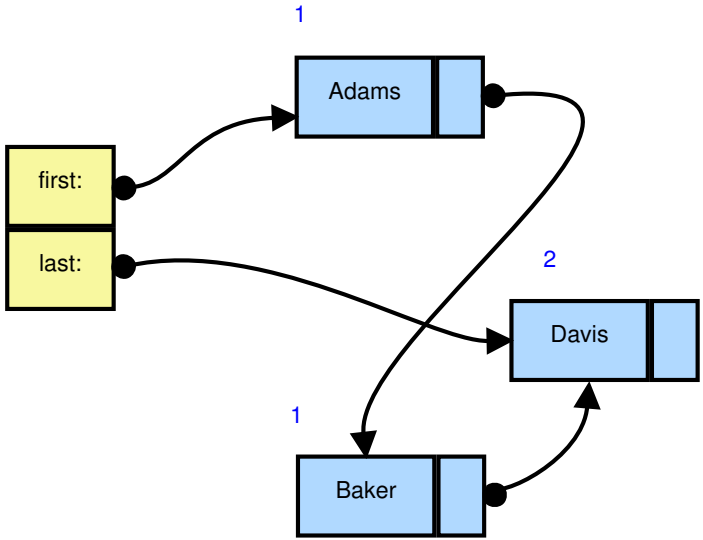
What went wrong? Let's look at our other examples.



Ref Counted SLL

Here is our singly linked list with reference counts.

Assume that the list header itself is a local variable that is about to be destroyed.

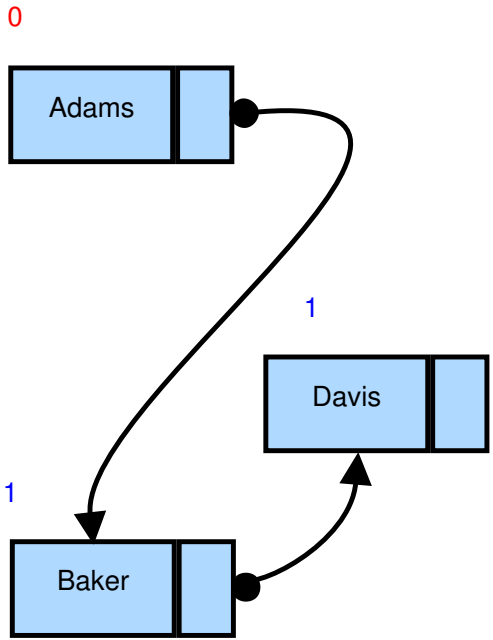


. . . . . .

Ref Counted SLL II

- When the first and last pointers disappear, the reference count for Adams goes to zero.
  - So Adams can be scavenged.
- When it is, that will cause Baker's reference count to drop to zero.
  - When Baker is scavenged, Davis' count will drop to zero
- and it too will be scavenged.

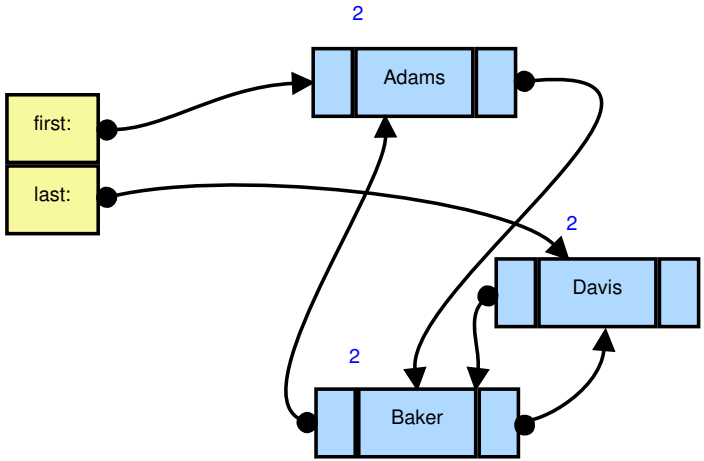
So that works just fine!



Ref Counted DLL

Now let's look at our doubly linked list.

Again, let's assume that the list header itself is a local variable that is about to be destroyed.

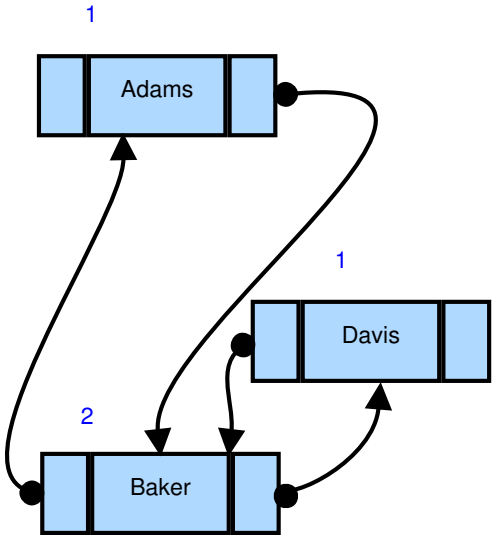


.....

Ref Counted DLL II

Here's the result.

Alas, we can see that none of the reference counters have gone to zero, so nothing will be scavenged, even though all three nodes are garbage.



Reference Counting's Achilles Heel

What's the common factor between the failures in the first and third examples?

- It's the *cycles*. If the pointers form a cycle, then the objects in that cycle can never get a zero reference count, and reference counting will fail.
- Reference counting won't work if our data can form cycles of pointers.
  - And, as the examples discussed here have shown, such cycles aren't particularly unusual or difficult to find in practical structures.

So a more general approach is needed.

2.2 Mark and Sweep

Mark and Sweep

Mark and sweep is one of the earliest and best-known garbage collection algorithms.

- It works perfectly well with cycles, but
- requires some significant support from the compiler and run-time support system.

. . . . . .

Assumptions

The core assumptions of mark and sweep are:

- Each object on the heap has a hidden "mark" bit.
- We can find all pointers outside the heap (i.e., in the activation stack and static area)
- For each data object on the heap, we can find all pointers within that object.
- We can iterate over all objects on the heap

. . . . . .

The Mark and Sweep Algorithm

With those assumptions, the mark and sweep garbage collector is pretty simple:

```
void markAndSweep()
{
 // mark
 for (all pointers P on the run-time stack or
   in the static data area )
  {
    mark *P;
```


```
  }

 //sweep
 for (all objects *P on the heap)
   {
     if *P is not marked then
        delete P
     else
        unmark *P
   }
}

template <class T>
void mark(T* p)
{
  if *p is not already marked
    {
      mark *p;
      for (all pointers q inside *p)
        {
          mark *q;
        }
     }
}
```

The algorithm works in two stages.

- In the first stage, we start from every pointer outside the heap and recursively mark each object reachable via that pointer. (In graph terms, this is a depth-first traversal of objects on the heap.)
- In the second stage, we look at each item on the heap.
  - If it's marked, then we have demonstrated that it's possible to reach that object from a pointer outside the heap.

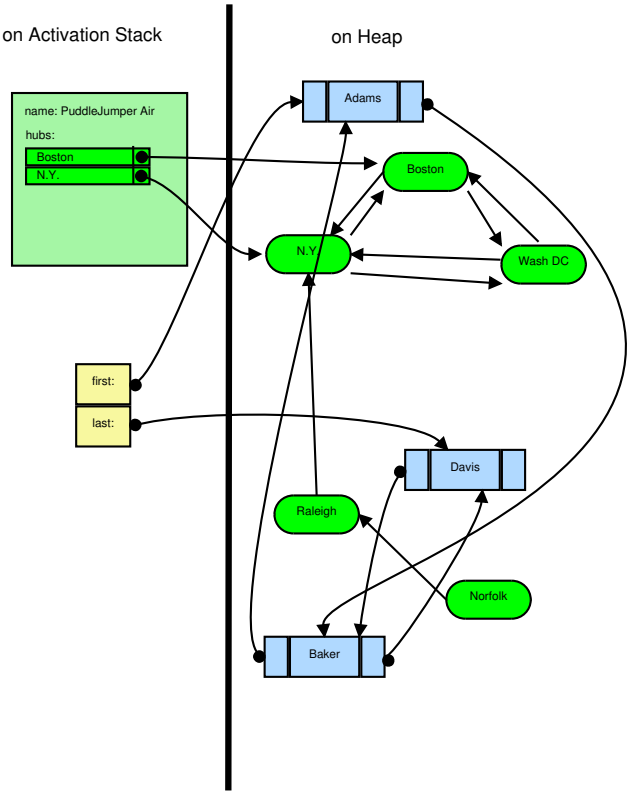
    - It isn't garbage, so we leave it alone (but clear the mark so we're ready to repeat the whole process at some time in the future).
  - If the object on the heap is not marked, then it's garbage and we scavenge it.

. . . . . .

Mark and Sweep Example

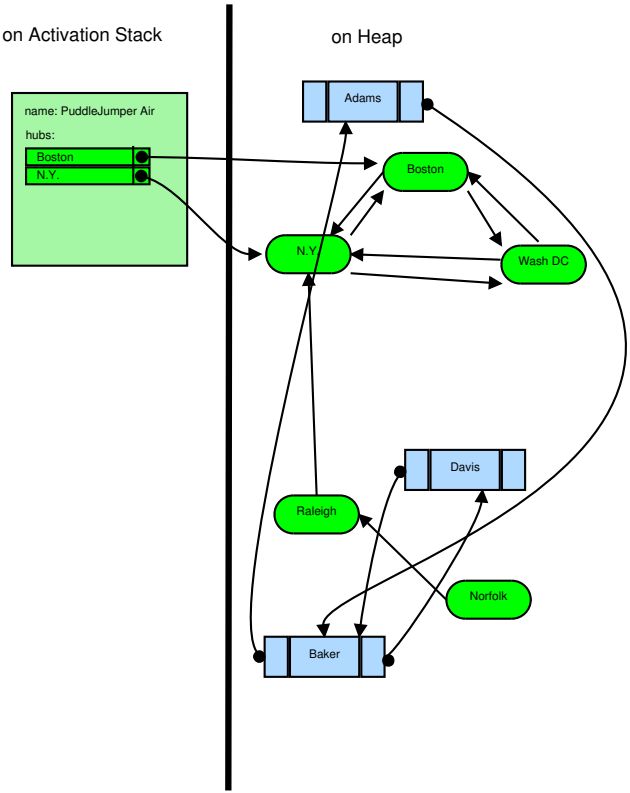
As an example, suppose that we start with this data.

Then, let's assume that the local variable holding the list header is destroyed.



Mark and Sweep Example II

- At some point later in time, the mark and sweep algorithm is started (typically in response to later code of ours trying to allocate something new in memory and the run-time system has discovered that we are running low on available storage.).
- The main algorithm begins the marking phase, looping through the pointers in the activation stack.
  - We have two. The first points to the Boston node. So we invoke the `mark()` function on the pointer to Boston.
    - The Boston node has not been marked yet, so we mark it.
  - Then the mark() function iterates over the pointers in the Boston object. It first looks at the N.Y. pointer and recursively invokes itself on that.
  - The N.Y. object has not been marked yet, so we mark it and then iterate over the pointers in N.Y.,



We first come to the pointer to Boston, and recursively invoke mark() on that. But Boston is already marked, so we return immediately to the N.Y. call. Continuing on, we find a pointer to Wash DC. and invoke mark() on that.

The Wash DC object has not been marked yet, so we mark it and then iterate over the pointers in Wash DC. We first come to the pointer to Boston, and recursively invoke mark() on that. But Boston is already marked, so we return immediately to the N.Y. call. Again, that object is already marked so we immediately return to the earlier N.Y. call. That one has now visited all of its pointers, so it returns to the first Boston call.

The Boston call resumes iterating over its pointers, and finds a pointer to Wash DC. It calls mark() on that pointer, but Wash DC has already been marked, so we return immediately. The Boston call has now iterated over all of its pointers, so we return to the main mark and sweep algorithm.

That algorithm continues looking at pointers on the activation stack. We have a pointer to N.Y., and call mark() on that. But N.Y. is already marked, so we return immediately.

Mark and Sweep Example III

Once the mark phase of the main algorithm is complete,

- We have marked the Boston, N.Y., and Wash DC objects.
- The Norfolk, Raleigh, Adams, Baker, and Davis objects are unmarked.

. . . . . .

The Sweep Phase

In the sweep phrase, we visit each object on the heap.

- The three marked hubs will be kept, but their marks will be cleared in preparation for running the algorithm again at some time in the future.
- All of the other objects will be scavenged.

. . . . . .

Assessing Mark and Sweep

In practice, the recursive form of mark-and-sweep requires too much stack space.

- It can frequently result in recursive calls of the mark() function running thousands deep.
  - Since we call this algorithm precisely because we are running out of space, that's not a good idea.

    Practical implementations of mark-and-sweep have countered this problem with an iterative version of the mark function that "reverses" the pointers it is exploring so that they leave a trace behind it of where to return to.

- Even with that improvement, systems that use mark and sweep are often criticized as slow. The fact is, tracing *every* object on the heap can be quite time-consuming. On virtual memory systems, it can result in an extraordinary number of page faults. The net effect is that mark-and-sweep systems often appear to freeze up for seconds to minutes at a time when the garbage collector is running. There are a couple of ways to improve performance.

. . . . . .

2.3 Generation-Based Collectors

Old versus New Garbage

In many programs, people have observed that object lifetime tends toward the extreme possibilities.

- temporary objects that are created, used, and become garbage almost immediately
- long-lived objects that do not become garbage until program termination

. . . . . .

Generational GC

Generational collectors take advantage of this behavior by dividing the heap into "generations".

- The area holding the older generation is scanned only occasionally.
- The area holding the youngest generation is scanned frequently for possible garbage.
  - an object in the young generation area that survives a few garbage collection passes is moved to the older generation area

. . . . . .

The actual scanning process is a modified mark and sweep. But because relatively few objects are scanned on each pass, the passes are short and the overall cost of GC is low.

To keep the cost of a pass low, we need to avoid scanning the old objects on the heap. The problem is that some of those objects may have pointers to the newer ones. Most generational schemes use traps in the virtual memory system to detect pointers from "old" pages to "new" ones to avoid having to explicitly scan the old area on each pass.

2.4 Incremental Collection

Incremental GC

Another way to avoid the appearance that garbage collection is locking up the system is to modify the algorithm so that it can be run one small piece at a time.

- Conceptually, every time a program tries to allocate a new object, we run just a few mark steps or a few sweep steps,
- By dividing the effort into small pieces, we give the illusion that garbage collection is without a major cost.

  There is a difficuty here, though. Because the program might be modifying the heap while we are marking objects, we have to take extra care to be sure that we don't improperly flag something as garbage just because all the pointers to it have suddenly been moved into some other data structure that we had already swept.
- In languages like Java where parallel processes/threads are built in to the language capabilities, systems can take the incremental approach event further by running the garbage collector in parallel with the main calculation.

  Again, special care has to be taken so that the continuously running garbage collector and the main calculation don't interfere with one another.

. . . . . .

3 Strong and Weak Pointers

Doing Without

OK, garbage collection is great if you can get it.

- But C++ does not provide it, and C++ compilers don't really provide the kind of support necessary to implement mark ans sweep or the even more advanced forms of GC.
- So what can we, as C++ programmers do, when faced with data structures that need to share heap data with one another?

. . . . . .

Ownership

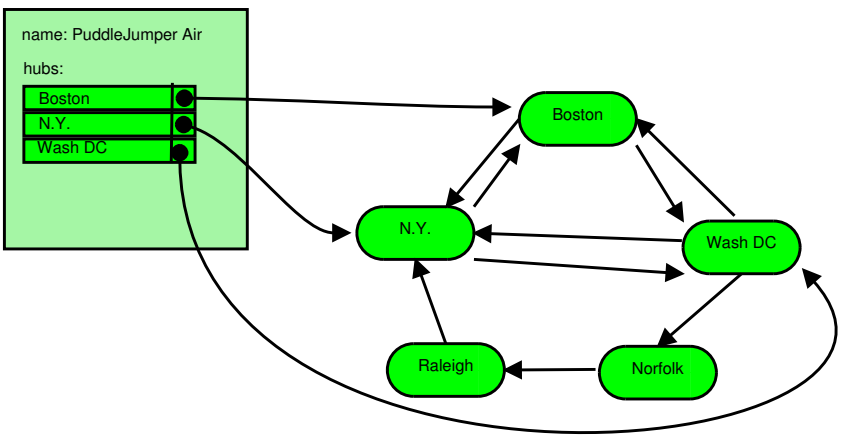
One approach that works in many cases is to try to identify which ADTs are the *owners* of the shared data, and which ones merely use the data.

- The owner of a collection of shared data has the responsibility for creating it, sharing out pointers to it, and deleting it.
- Other ADTs that share the data without owning it should never create or delete new instances of that data.

. . . . . .

Ownership Example

In this example that we looked at earlier, we saw that if both the Airline object on the left and the Airport objects on the right deleted their own pointers when destroyed, our program would crash.



. . . . . .

Ownership Example

We could improve this situation by deciding that the Airline *owns* the Airport descriptors that it uses. So the Airline object would delete the pointers it has, but the Airports would never do so.

```
class Airport
{
   ⋮
```

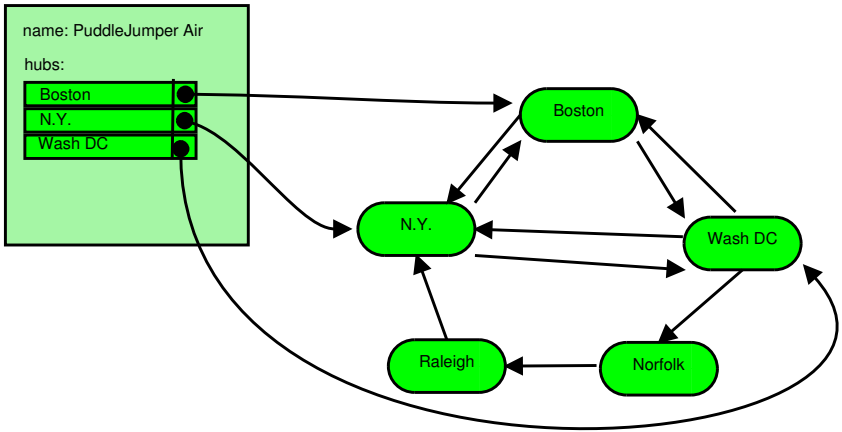
```
private:
   vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
  /* for (int i = 0; i < hasFlightsTo.size(); ++i)
       delete hasFlightsTo[i]; */
}

class AirLine {
   ⋮
   string name;
   map<string, Airport*> hubs;
};


AirLine::~Airline()
{
   for (map<string, Airport*>::iterator i = hubs.begin;
        i != hubs.end(); ++i)
     delete i->second;
}
```

.....

Ownership Example

Thus, when the airline object on the left is destroyed, it will delete the Boston, N.Y., and Wash DC objects.

- Each of those will be deleted exactly once, so our program should no longer crash.



- This solution isn't perfect. The Norfolk and Raleigh objects are never reclaimed, so we do wind up leaking memory. The problem is that, having decided that the Airline owns the Airport descriptors, we have some Airport objects with no owner at all.

. . . . . .

Asserting Ownership

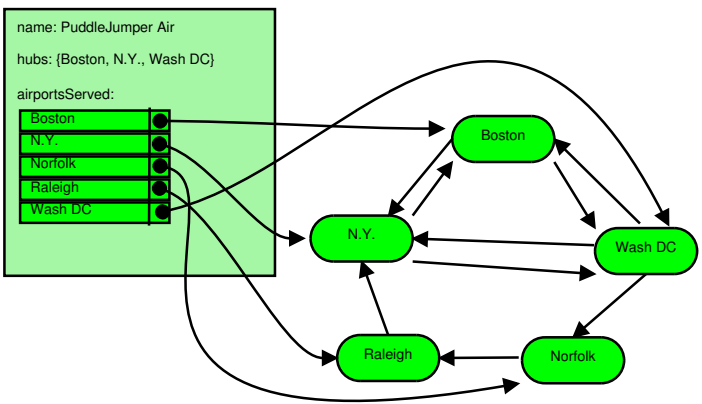
I would probably resolve this by modifying the Airline class to keep better track of its Airports.

```
class AirLine {
   ⋮
   string name;
   set<string> hubs;
   map<string, Airport*> airportsServed;
};
```

```
AirLine::~Airline()
{
   for (map<string, Airport*>::iterator i = airportsServed.begin;
        i != airportsServed.end(); ++i)
     delete i->second;
}
```

Asserting Ownership (cont.)

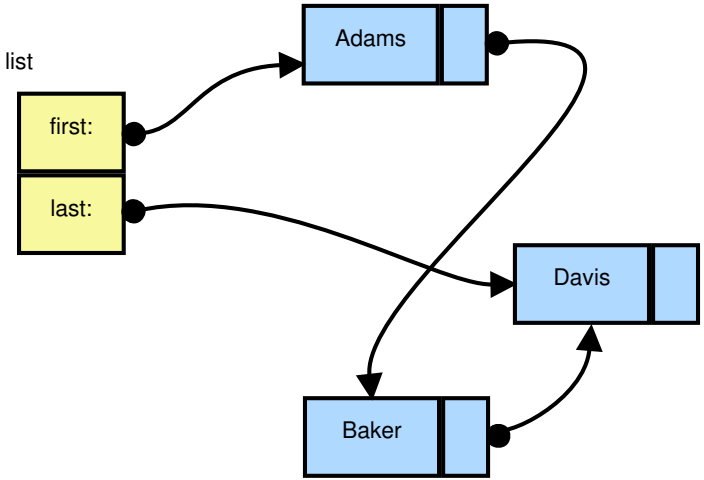
The new map tracks all of the airports served by this airline, and we use a separate data structure to indicate which of those airports are hubs.
Now, when an airline object is destroyed, all of its airport descriptors will be reclaimed as well.



Ownership Can Be Too Strong

Ownership is sometimes a bit too strong a relation to be useful.

- In this example, if we simply say that the list header owns the nodes it points to, then we would delete the first and last node and would leave Baker on the heap.



- And if we say that the nodes owned the other nodes that they point to *and* that the list header owns the ones it points to, we would delete the last node twice.

. . . . . .

Strong and Weak Pointers

We can generalize the notion of ownership by characterizing the various pointer data members as strong or weak.

- A *strong pointer* is a pointer data member that indicates that the object pointed to must remain in memory.
- A *weak pointer* is a pointer data member that is allowed to point to data that might have been deleted.
  - (Obviously, we never want to follow a weak pointer unless we are sure that the data has not, in fact, been deleted.)

When an object containing pointer data members is destroyed, it deletes its strong pointer members and leaves its weak ones alone.

. . . . . .

Strong and Weak SLL

In this example, if we characterize the pointers as shown:

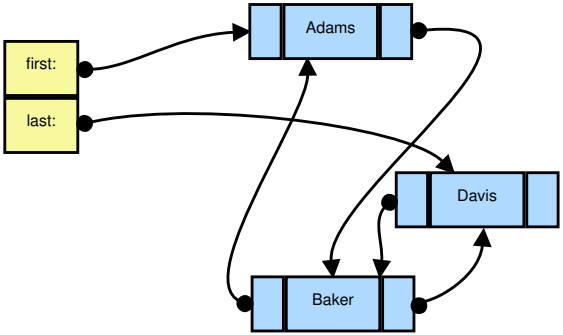
```
struct SLNode {
   string data;
   SLNode* next; // strong
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first; // strong
   SLNode* last;  // weak
public:
     ⋮
   ~List()
    {
      delete first;  // OK, because this is strong
      /*delete last;*/ // Don't delete. last is weak.
    }
};
```

then our program will run correctly.

. . . . . .

Picking the Strong Ones

The key idea is to select the smallest set of pointer data members that would connect together all of the allocated objects, while giving you exactly one path to each such object.



. . . . . .

Strong and Weak DLL

Similarly, in a doubly linked list, we can designate the pointers as follows:

```
struct DLNode {
   string data;
   DLNode* prev; // weak
   DLNode* next; // strong
     ⋮
   ~DLNode () {delete next;}
};

class List {
   DLNode* first; // strong
   DLNode* last;  // weak
public:
     ⋮
   ~List() {delete first;}
};
```

and so achieve a program that recovers all garbage without deleting anything twice.

. . . . . .

4 Coming Soon to C++: std Reference Counting

C++11 Adds Reference Counting

The new C++11 standard (endorced by ISO in Sept 2011) contains proposals for smart pointer templates, quite similar in concept to the *RefCountPointer* discussed earlier.

- Some compilers may already offer versions.

. . . . . .

shared and weak ptrs

There are two primary class templates involved

- *shared_ptr<T>* provides a strong pointer with an associated reference counter.
  - Shared pointers may be freely assigned to one another.
  - If all shared pointers to an object on the heap are destroyed or are reassigned to point elsewhere, then that reference count will drop to zero and the object on the heap will be deleted.
- *weak_ptr<T>* provides a weak pointer to an object that has an associated reference counter, but
  - creating, copying, assigning, or destroying weak pointers does not affect the reference counter value.
- Strong and weak pointers may be copied to one another.

. . . . . .

We can illustrate these with our doubly linked list. (I would not do this in real practice, because the prior version with informally identified strong and weak pointers is simpler and probably slightly faster.)

```
struct DLNode {
   string data;
   weak_ptr<DLNode> prev;
   shared_ptr<DLNode> next;
     ⋮
```


```
   ~DLNode ()
     {
       // do nothing - the reference counting will take care of it
      }
};

class List {
   shared_ptr<DLNode> first;
   weak_ptr<DLNode> last;
public:
     ⋮
   ~List() {/* do nothing */}
};
```

Once this is set up, we use those four data members largely as if they were regular pointers. The only difference is when constructing new smart pointers. Some sample code:

```
void addToFront (List& list, string newData)
{
   shared_ptr<DLNode> newNode (new DLNode()); ❶
   newNode->data = newData;           ❷
   if (first == shared_ptr<DLNode>()) ❸
     {
       // list is empty
       first = last = newNode;    ❹
     }
   else
     {
      newNode->next = first;
      first->prev = newNode;
      first = newNode;
     }
}
```

❶ This line allocates a new object and immediately hands it off to the reference counting system by creating a shared pointer to it (and losing the "real" pointer so that we aren't even tempted to play around with it directly).

❷ Looking at this line, there's nothing to tell us that we're working with a smart pointer instead of a regular pointer.

❸ The default constructor for shared and weak pointers presents the equivalent of a null pointer.

❹ Notice that, in this statement, we assign a shared pointer to a weak pointer, then assign that weak pointer to a different shared pointer.

Once you figure out which data members should be strong (shared) and which should be weak, you can pretty much forget the difference afterwards.

5 Java Programmers Have it Easy

Java Programmers Have it Easy

Java has included automatic garbage collection since its beginning.

- From a practical point of view, sharing in Java is actually easier (and more common) than deep copying.
- Java programmers typically are unconcerned with many of the memory management errors that C++ programmers must strive to avoid.

. . . . . .

C++ programmers may sometimes sneer at the slowdown caused by garbage collection. The collector implementations, however, continue to evolve. In fact, current versions of Java commonly offer multiple garbage collectors, one which can be selected at run-time in an attempt to find one whose run-time characteristics (i.e., how aggressively it tries to collect garbage and how much of the time it can block the main program threads while it is working) that matches your program's needs.

Java programmers sometimes face an issue of running out of memory because they have inadvertently kept pointers to data that they no longer need. This is a particular problem in implementing algorithms that use caches or memoization to keep the answers to prior computations in case the same result is needed again in the future. Because of this, Java added a concept of a weak reference (pointer) that can be ignored when checking to see if an object is garbage and that gets set to null if the object it points to gets collected.