Constructors and the Rule of the Big 3

Steven J Zeil

September 4, 2013

Contents

Next we turn our attention to a set of issues that are often given short shrift in both introductory courses and textbooks, but that are extremely important in practical C++ programming.

As we begin to build up our own ADTS, implemented as C++ classes, we quickly come to the point where we need more than one of each kind of ADT object. Sometimes we will simply have multiple variables of our ADT types. Once we do, we will often want to copy or assign one variable to another, and *we need to understand what will happen when we do so*. Even more important, we need to be sure that what *does* happen is what we *want* to have happen, depending upon our intended behavior for our ADTs.

As we move past the simple case of multiple variables of the same ADT type, we may want to build *collections* of that ADT. The simplest case of this would be an array or linked list of our ADT type, though we will study other collections as the semester goes on. We will need to understand *what happens* when we initialize such a collection and when we copy values into and out of it, and we need to make sure that behavior is what we *want* for our ADTs.

In this lesson, we set the stage for this kind of understanding by looking at how we control initialization and copying of class values.

1 The Default Constructor

The Default Constructor

The *default constructor* is a constructor that takes no arguments. This is the constructor you are calling when you declare an object with no parameters. E.g.,

```
std::string s;
Book b;
```

.....

Declaration

A default constructor might be declared to take no parameters at all, like this:

```
class Book {
public:
   Book();
    ⋮
```

or with defaults on all of its parameters:

```
namespace std {
class string {
public:
    ⋮
   string(char* s = "");
    ⋮
```

Either way, we can *call* it with no parameters.

. . . . . .

Why Default?

Why is this called a "default" constructor? There's actually nothing particularly special about it. It's just an ordinary constructor, but it is *used* in a special way. Whenever we create an array of elements, the compiler implicitly calls this constructor to initialize each elements of the array.

For example, if we declared:

```
std::string words[5000];
```

then each of the 5000 elements of this array will be initialized using the default constructor for `string`

- In fact, if we don't *have* a default constructor for our class, then we *can't* create arrays containing that type of data. Attempting to do so will result in a compilation error.

. . . . . .

Compiler-Generated Default Constructors

Because arrays are so common, it is rare that we would actually want a class with no default constructor. The C++ compiler tries to be helpful:

> *If we create no constructors at all for a class, the compiler generates a default constructor for us.*

- That automatically generated default constructor works by initializing every data member in our class with its own default constructor.
  - In many cases (e.g., strings), this does something entirely reasonable for our purposes.
  - Be careful, though: the "default constructors" for the primitive types such as `int`, `double` and pointers work by doing nothing at all. The value is left *uninitialized*.

. . . . . .

That leaves us with an uninitialized value containing whatever bits happened to have been left at that particular memory address by earlier calculations/programs.

1.1 Default constructor examples

Book default constructor

If we haven't created a default constructor for `Books`, we could add one easily enough:

```
class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
   Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
      ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

To implement it, we need to come up with something reasonable for each data member:

```
Book::Book ()
   numAuthors(0), first(0), last(0)
{
}
```

(We'll let the strings `title` and `isbn` default to the `std::string` default of `""`, and we trust that the `Publisher` class provides a reasonable default constructor of its own.)

.....

Address default constructor

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);
```


```
  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

This class has no explicit default constructor. It does have another constructor, however, so the compiler will not generate an automatic default constructor for us.

There's no good reason why we should not have a default constructor for this class. We might want arrays of addresses sometime.

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  Address () {}
    ⋮
```

As you can see, there's not much to it. All the data members are strings, and there's really nothing much better for us to do than just rely on the default constructors for strings.

. . . . . .

2 Copy Constructors

Copy Constructors

The *copy constructor* for a class `Foo` is the constructor of the form:

```
Foo (const Foo& oldCopy);
```

- Like the default constructor, there is nothing special about the copy constructor itself.
  - It's just an ordinary constructor.
- It's special because of the number of common situations in which it gets *used*.
  - Like the default constructor, the compiler often generates implicit calls to this constructor in places where we might not expect it.

. . . . . .

Where are Copy Constructors Used?

The copy constructor gets used in 5 situations:

1. When you declare a new object as a copy of an old one:

   ```
   Book book2 (book1);
   ```

   or

   ```
   Book book2 = book1;
   ```

2. When a function call passes a parameter "by copy" (i.e., the formal parameter does not have a &):

   ```
   void foo (Book b, int k);
     ⋮

   Book text361 (0201308787, budd,
        "Data Structures in C++ Using the Standard Template Library",
        1998, 1);
   foo (text361, 0);    // foo actually gets a copy of text361
   ```

3. When a function returns an object:

   ```
   Book foo (int k);
   {
     Book b;
     ⋮
     return b; // a copy of b is placed in the caller's memory area
   }
   ```

4. When data members are initialized in a constructor's initialization list:

```
Author::Author (std::string theName,
                Address theAddress, long id)
  : name(theName),
    address(theAddress),
    identifier(id)
{
}
```

5. When an object is a data member of another class for which the compiler has generated its own copy constructor.

.....

Compiler-Generated Copy Constructors

As you can see from that list, the copy constructor gets used a *lot*. It would be very awkward to work with a class that did not provide a copy constructor.

So, again, the compiler tries to be helpful.

If we do not create a copy constructor for a class, the compiler generates one for us.

- This automatically generated version works by copying each data member via *their* individual copy constructors.
- For data members that are primitives, such as `int` or pointers, the copying is done by copying all the bits of that primitive object.
  - For things like `int` or `double`, that's just fine.
  - We'll see shortly, however, that this may or may not be what we want for pointers.

.....

2.1 Copy Constructor Examples

2.1.1 Address

Address Copy Constructor

In the case of our `Address` class, we have not provided a copy constructor, so the compiler would generate one for us.

The implicitly generated copy constructor would behave as if it had been written this way:

```
Address::Address (const Address& a)
  : street(a.street), city(a.city),
    state(a.state), zip(a.zip)
{
}
```


- This is, in fact, a perfectly good copy function for this class, so we might as well use the compiler-generated version.

. . . . . .

If our data members do not have explicit copy constructors (and *their* data members do not have explicit copy constructors, and . . .) then the compiler-provided copy constructor amounts to a bit-by-bit copy.

The compiler is all too happy to generate a copy constructor for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

2.1.2 Book - simple arrays

Book - simple arrays

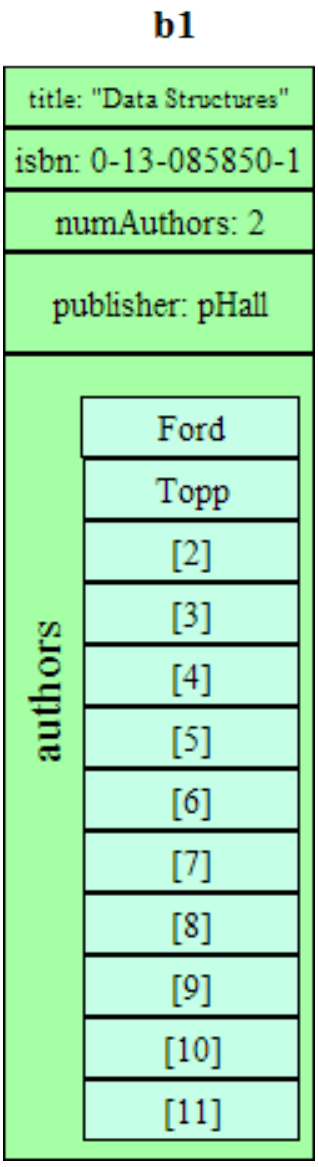
```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;
```

```
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

Consider the problem of copying a book, and for the moment we will work with our "simple" arrays version.

.....

Book - simple arrays (cont.)



If we start with a single book object, `b1`, as shown here, and then we execute

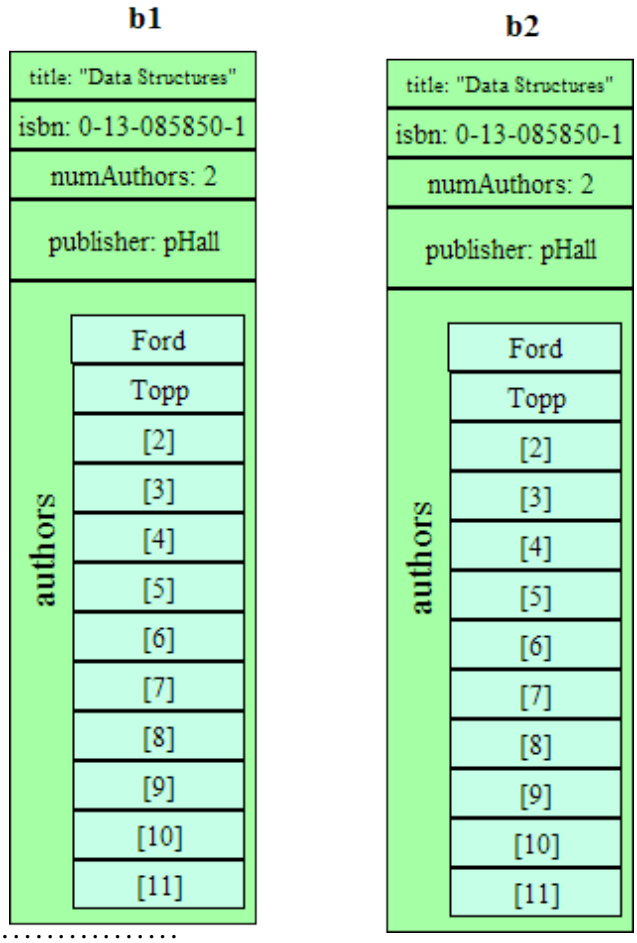
```
Book b2 = b1;
```

because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.

.....

Compiler-Generated Copy

The new result would be something like this, which looks just fine.



Of course, as we have noted before, this fixed-length array design has a lot of drawbacks, so let's look at another one of our implementations.

2.1.3 Book - dynamic arrays

Book - dynamic arrays

Now let's consider the problem of copying a book implemented using dynamically allocated arrays.

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
```

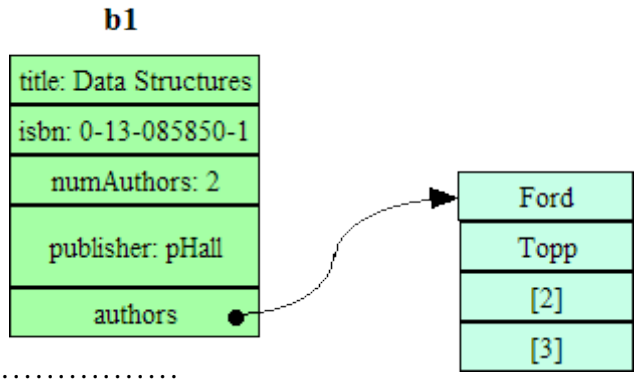
```
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

Copying dynamic arrays

If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

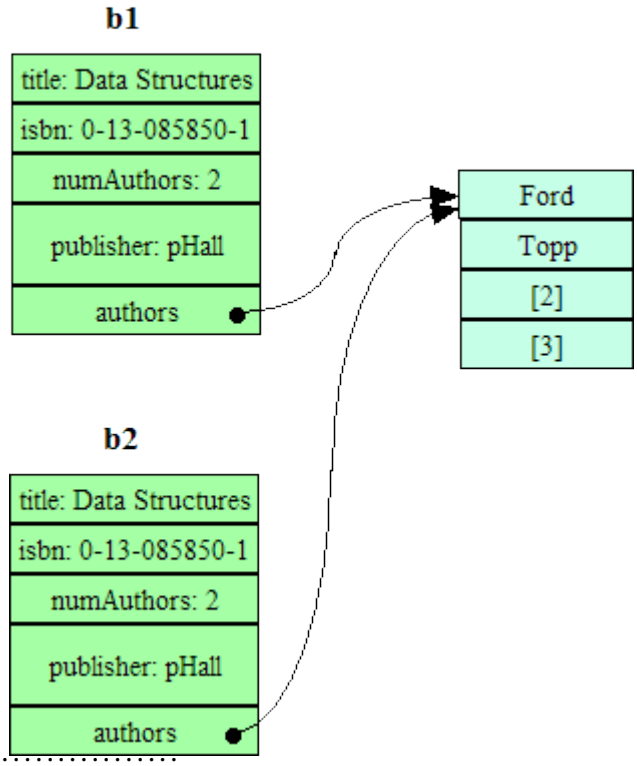
because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.



Compiler-Generated Copy

The new result would be something like this.

- The *authors* pointer is copied bit-by-bit
- two book objects now share the same author array



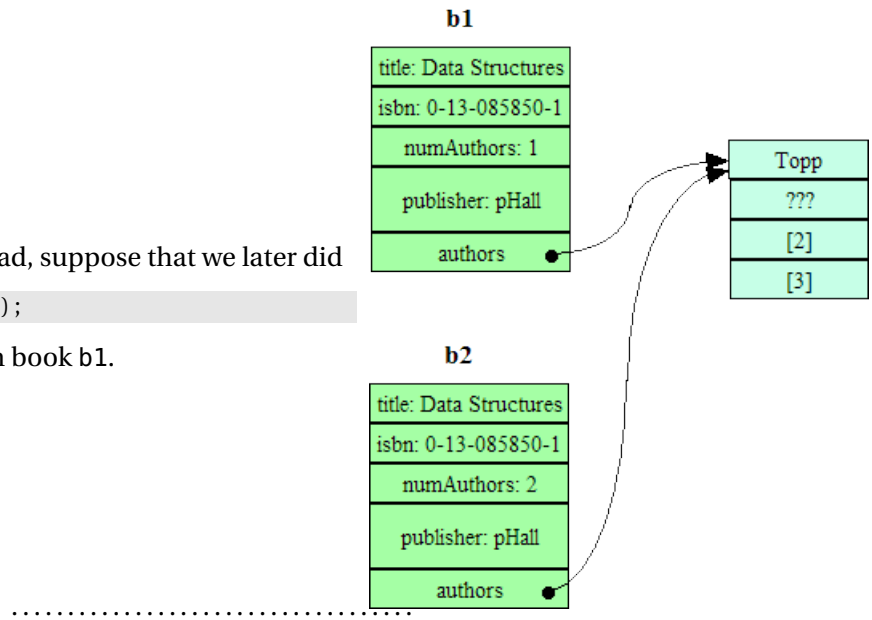
The key factor here is that the `authors` data member, of data type `AuthorNode*` is copied using the default copy procedure for pointers, which is to simply copy the bits of the pointer. That has the result of placing the same array address in both book objects. That's a really, really, bad idea!

Co-authors Should Not Share

To understand why this is so bad, suppose that we later did

```
b1.removeAuthor(b1.begin());
```

to remove the first author from book `b1`.

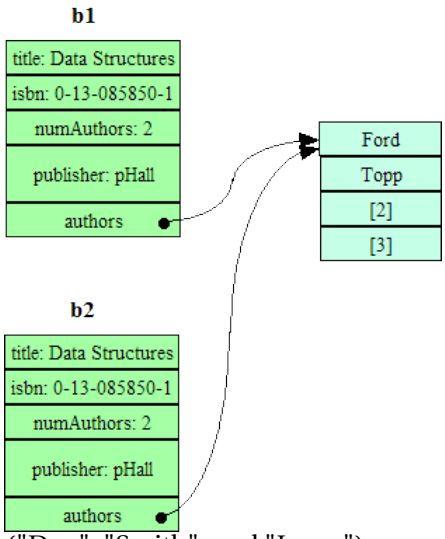


Afterwards, we would have something like this. Notice that the change to `b1` has, in effect, corrupted `b2`. The book object `b2` still believes it has two authors (`numAuthors` == 2) but there is only one in the array.

Unplanned Sharing Leads to Corruption

```
void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  if (numAuthors >= MAXAUTHORS)
    {
      Author* newAuthors = new Author[2*MAXAUTHORS];
      for (int i = 0; i < MAXAUTHORS; ++i)
    newAuthors[i] = authors[i];
      MAXAUTHORS *= 2;
      delete [] authors;
      authors = newAuthors;
    }
  int i = numAuthors;
  int atk = at - authors;
  while (i > atk)
    {
      authors[i] = authors[i-1];
```

```
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}
```



That's not even the worst possible scenario. Recall that our code for adding authors works by

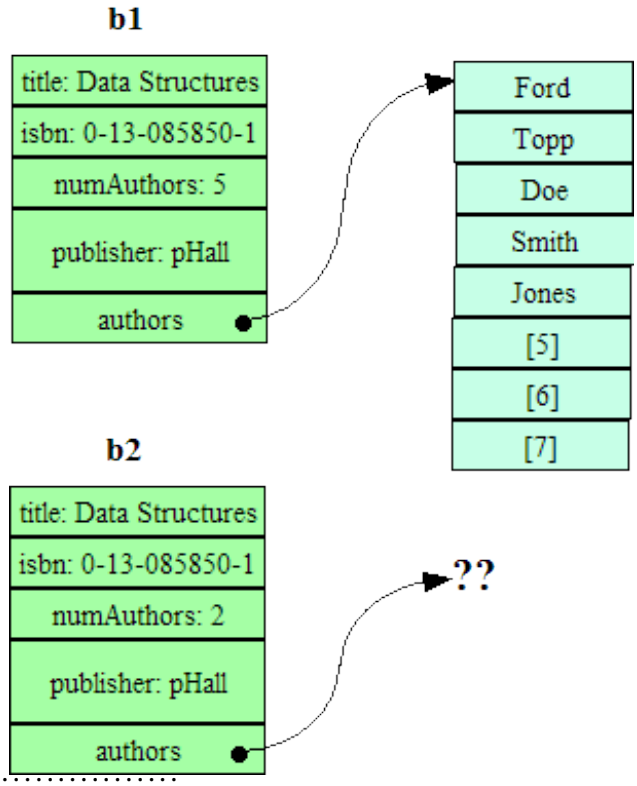
- Checking to see if there was room in the array.
- if not,

1. allocating a larger array,
2. copying the data, and
3. deleting the old array:

So if we start from this data state, and then add 3 authors to book 1 ("Doe", "Smith", and "Jones"), . . .

. . . . . .

Corrupted Data



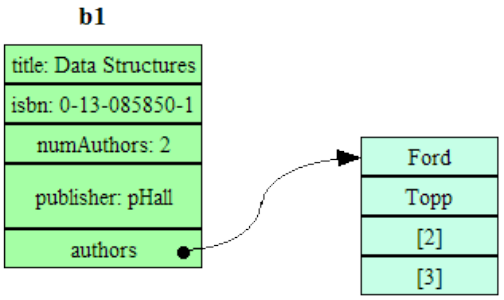
. . . we would end up with the state shown here.

- `b1` deleted the old array,
- but `b2` still has the old array address in its data member.

So any attempt to access `b2`'s authors is now essentially a throw of the dice - nobody can really predict what would happen.

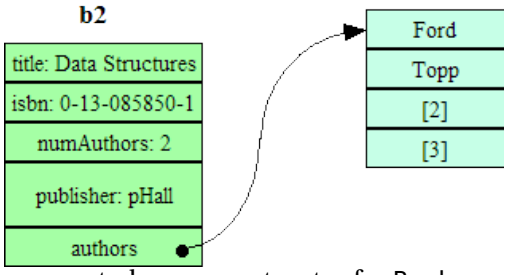
Sometime it's Better to Have 2 Copies

What we really wanted, after the copy:




```
Book b2 = b1;
```

is something more like this:



But to get that, we will not be able to rely on the automatically generated copy constructor for `Books`.

. . . . . .

2.2 Shallow vs Deep Copy

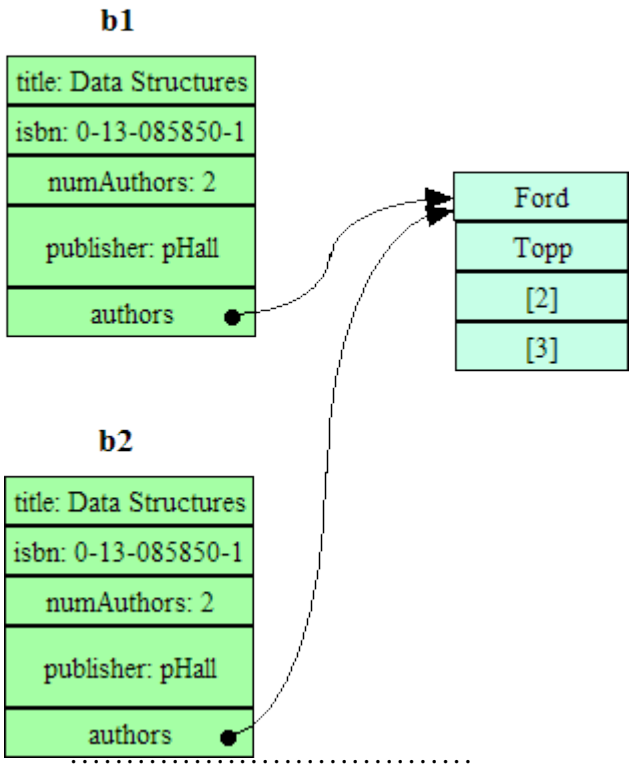
Shallow vs Deep Copy

Copy operations are distinguished by how they treat pointers:

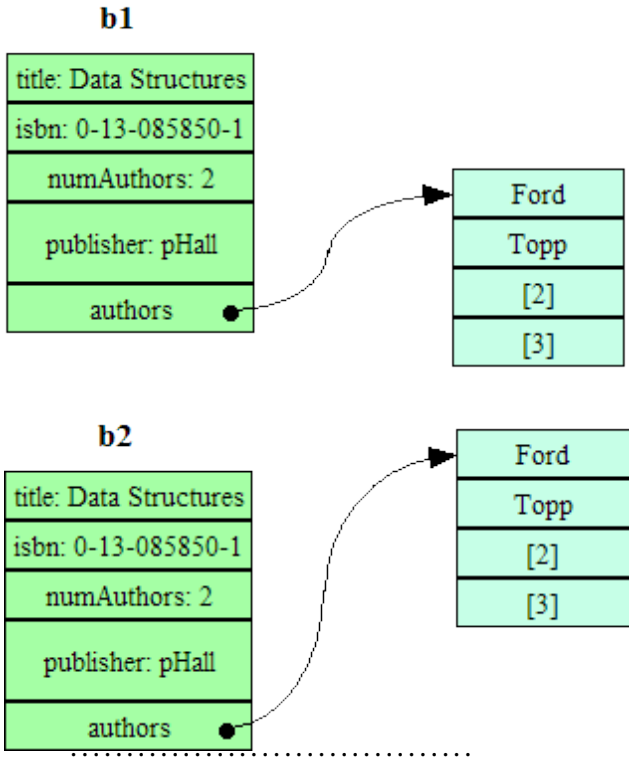
- In a *shallow copy*, all pointers are copied.
  - Leads to shared data on the heap.
- In a *deep copy*, objects pointed to are copied, then the new pointer set to the address of the copied object.
  - Copied objects keep exclusive access to the things they point to.

. . . . . .

This was a Shallow Copy



This was a Deep Copy



Shallow versus Deep

- In this example, deep is preferred.
- That's not always the case.
  - When we design an ADT we have to *think* about what is right for the abstraction we want to provide
- "Shallow" and "deep" are two extremes of a range of possible copies

For any ADT, we must decide whether the things it points to are things we want to *share* with other objects or whether we want to *own* them exclusively. That's a matter of just how we want our ADTs to behave, which depends in turn on what we expect to do with them.

In this context, it should be noted that "shallow" and "deep" are actually two extremes of a range of possible copy depths - sometimes our ADTs call for a behavior that has us treat some pointers shallowly and others deeply.

2.3 Deep Copy Examples

Book - dynamic arrays

If we want to implement a proper deep copy for our books, we start by adding the constructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

Then we supply a function body for this constructor.

```
Book::Book (const Book& b)
  : title(b.title), isbn(b.isbn), publisher(b.publisher),
    numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS)
{
  authors = new Author[numAuthors+1];
  for (int i = 0; i < numAuthors; ++i)
    authors[i] = b.authors[i];
}
```

.....

Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a

new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

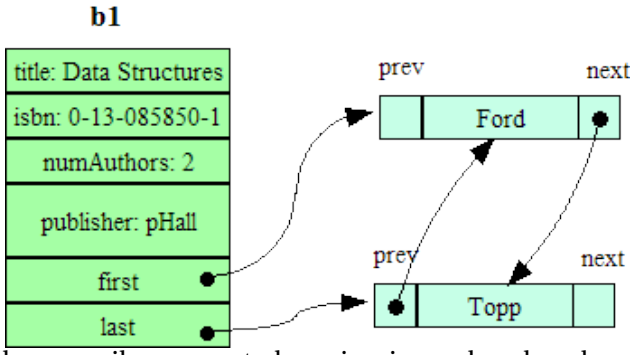
Book - linked list

```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
```

```
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};

#endif
```

Now let's consider the problem of copying a book implemented using linked lists.
If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

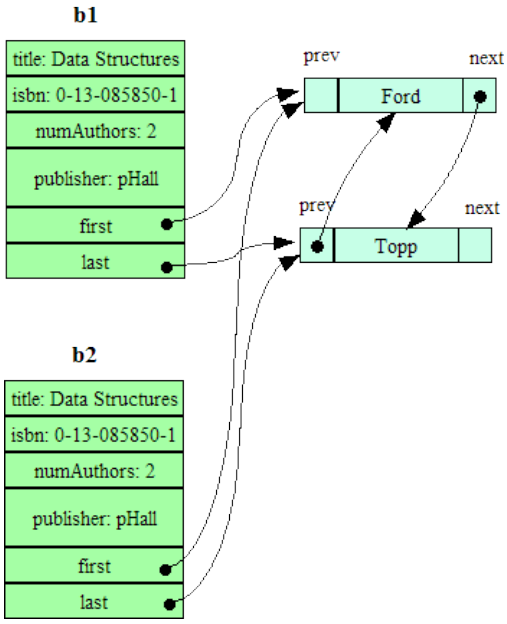


because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.

. . . . . .

Shallow Copy of Linked List

The new result would be something like this.



• Again, the pointer data members (`first` and `last`) are copied using the default copy procedure for pointers, which is to simply copy the bits of the pointer.

• That has the result of placing the same node addresses in both book objects. In effect, there is one collection of nodes being shared between both books.

• Once again, that's a really, really, bad idea!

. . . . . .

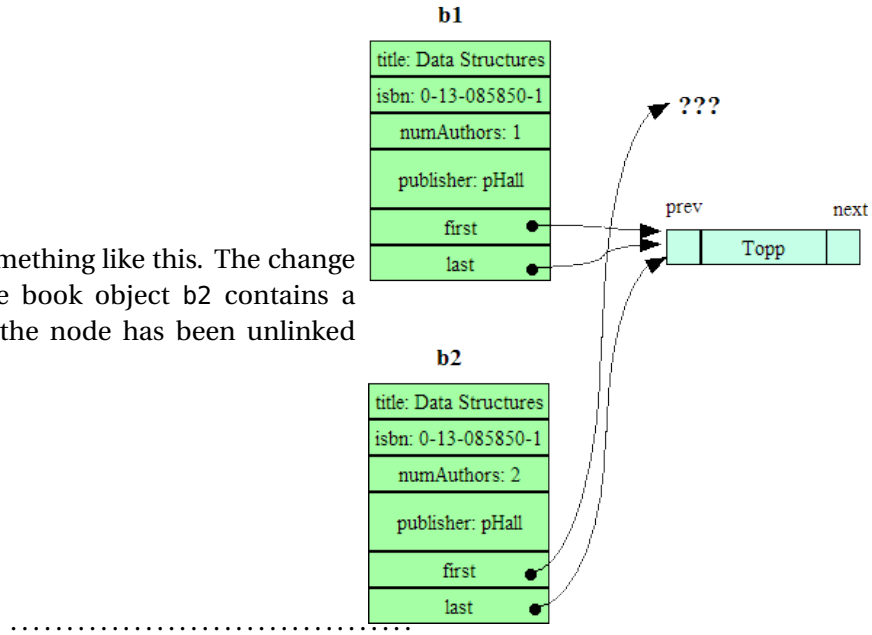
Corrupted List

For example, suppose that we later did

```
b1.removeAuthor(b1.begin());
```

to remove the first author from book `b1`.

Afterwards, we would have something like this. The change to `b1` has corrupted `b2`. The book object `b2` contains a pointer to an address where the node has been unlinked from the list and deleted.



List-Based Copy Constructor

If we want to implement a proper deep copy for our books, we start by adding the constructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     :
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;
```

```
  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

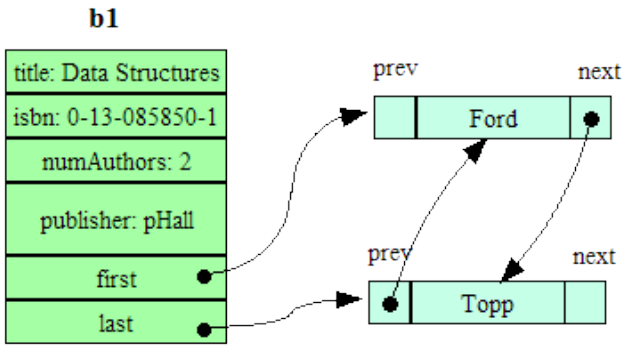
Then we supply a function body for this constructor.

```
Book::Book (const Book& b)
  : title(b.title), isbn(b.isbn),
    publisher(b.publisher),
    numAuthors(0), first(0), last(0)
{
  for (AuthorPosition p = b.begin(); p != b.end();
       ++p)
    addAuthor(end(), *p);
}
```

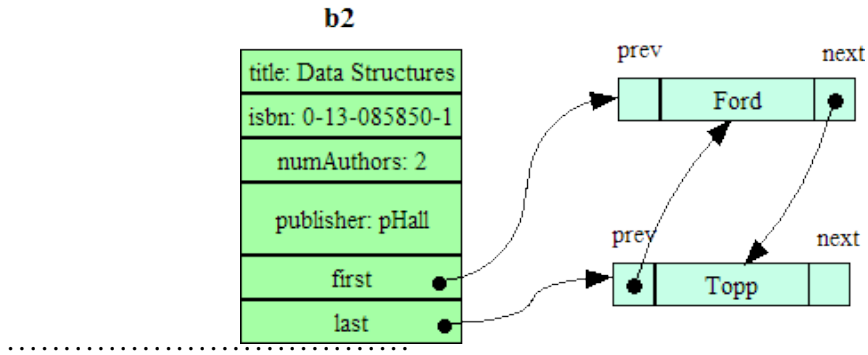
. . . . . .

Most of the data members can be copied easily. But the `first` and `last` pointers are copied by setting the list up first as an empty list, then copying each author from the existing book into the new one.

Linked-List Deep Copy



The end result after the copy should be this.



Book - std::list

```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
```

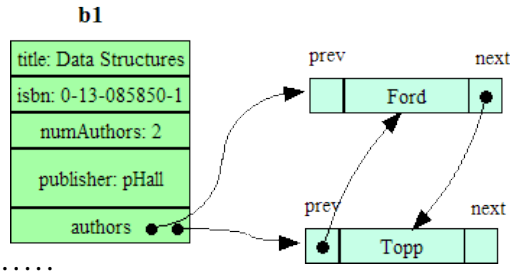
```
  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

Now let's consider the problem of copying a book implemented using `std::list`.

If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.

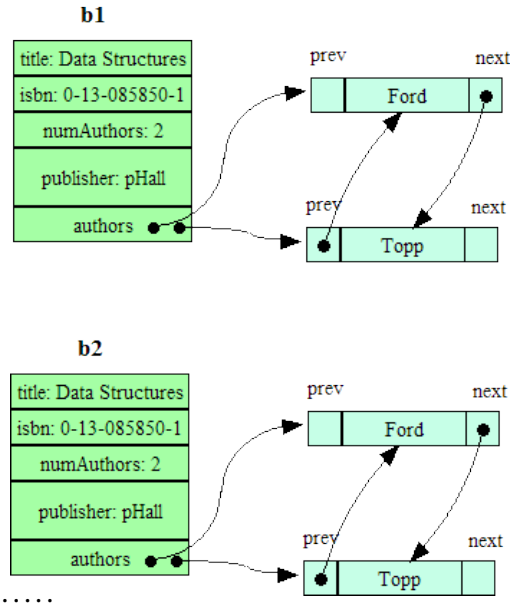


. . . . . .

Copy of std::list

The new result would be something like this. This looks just fine.

We don't need to override the compiler-generated copy constructor in this case.



. . . . . .

But what's so different between this case and the last one where we implemented our own linked list? Look at the `authors` data member. This is not a pointer. It's an object of type `std::list`. Now, we don't really know what's in there, and we don't need to (or particularly want to) know. But the documentation for `std::list` guarantees us that this class provides its own deep copy for the copy constructor, so we trust it. It's certainly not the `Book` class's job to know how to copy arbitrary other data structures. (For that matter, we have no idea at the moment whether the `Publisher` class contains internal pointers, but we trust that class to also provide a proper copy constructor.)

2.4 Trusting the Compiler-Generated Copy Constructor

Trusting the Compiler-Generated Copy Constructor

By now, you may have perceived a pattern.

Shallow copy is wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

And it follows that:
Compiler-generated copy constructors are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . . . .

3 Assignment

Assignment

When we write `book1 = book2`, that's shorthand for `book1.operator=(book2)`.

We tend to do a lot of assignment in typical programming, so, once more, the compiler tries to be helpful:

> **If you don't provide your own assignment operator for a class, the compiler generates one automatically.**

- The automatically generated assignment operator works by assigning each data member in turn.
- If none of the members have programmer-supplied assignment ops, then this is a *shallow copy*

. . . . . .

A Compiler-Generated Assignment Op

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
```

```
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

For example, we have not provided an assignment operator for `Address` class. Therefore the compiler will attempt to generate one, just as if we had written

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);


   Address& operator= (const Address&);
     ⋮
```

.....

A Compiler-Generated Assignment Op (cont.)

The automatically generated body for this assignment operator will be the equivalent of

```
Address& Address::operator= (const Address& a)
{
  street = a.street;
  city = a.city;
  state = a.state;
  zip = a.zip;
  return *this;
}
```

And that automatically generated assignment is just fine for `Address`.

.....

Return values in Asst Ops

```
Address& Address::operator= (const Address& a)
{
  street = a.street;
  city = a.city;
  state = a.state;
  zip = a.zip;
  return *this;
}
```

The return value returns the value just assigned, allowing programmers to chain assignments together:

```
addr3 = addr2 = addt1;
```

This can simplify code where a computed value needs to be tested and then maybe used again if it passes the test, e.g.,

```
while ((x = foo(y)) > 0) {
  do_something_useful_with(x);
}
```

.....

The compiler is all too happy to generate an assignment operator for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

3.1 Assignment Examples

3.1.1 Book - Simple Arrays

Book - Simple Arrays

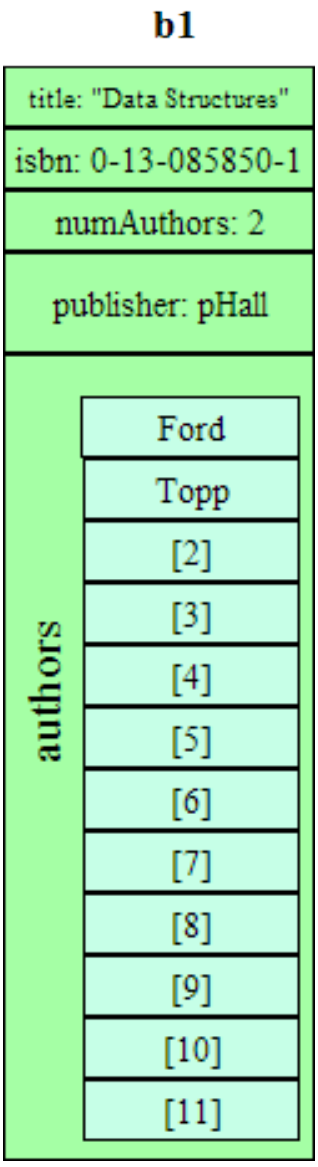
```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
```

```
  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

Consider the problem of copying a book, and for the moment we will work with our "simple" arrays version.

.....

Book - Simple Arrays (cont.)



If we start with a single book object, `b1`, as shown here, and then we execute

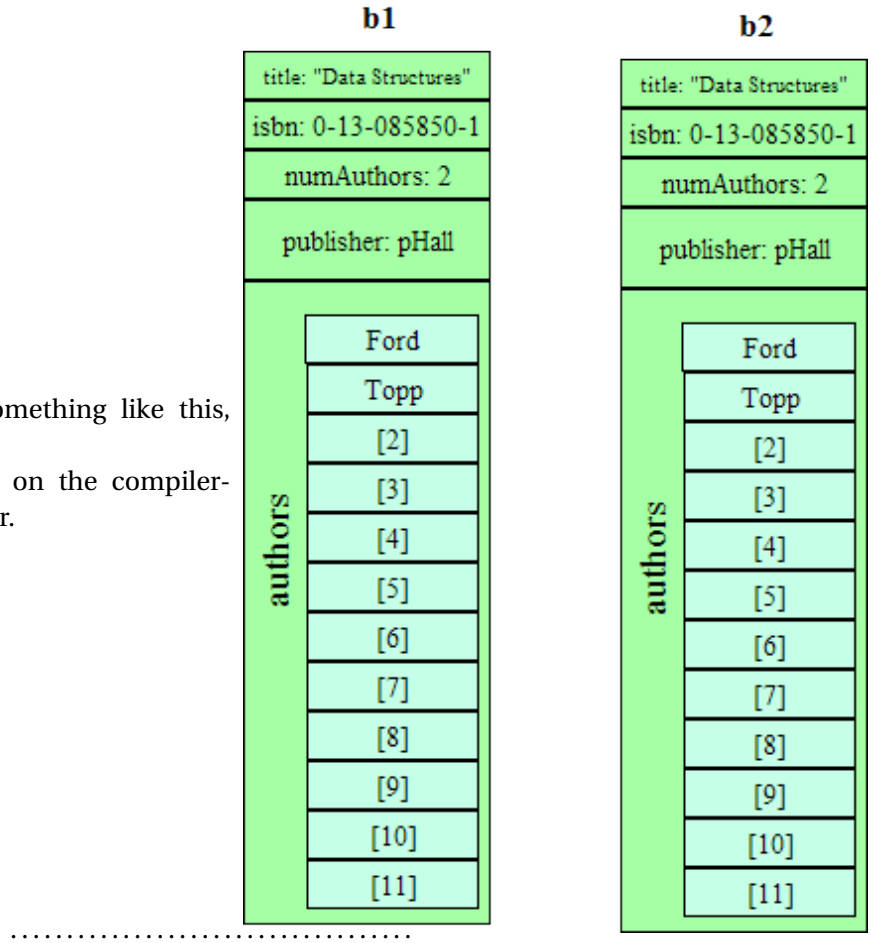
```
b2 = b1;
```

because we have provided no assignment operator, the compiler-generated version is used and each data member is copied.

.....

Compiler-Generated Assignment

The new result would be something like this, which looks just fine.
So in this case, we can rely on the compiler-generated assignment operator.



3.1.2 Book - dynamic arrays

Book - dynamic arrays

Now let's consider the problem of copying a book implemented using dynamically allocated arrays.

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;
```

```
  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

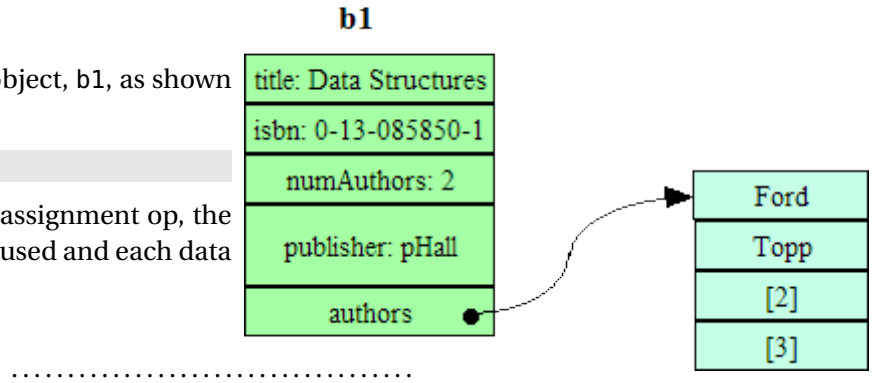
.....

Assigning with dynamic arrays

If we start with a single book object, `b1`, as shown here, and then we execute

```
b2 = b1;
```

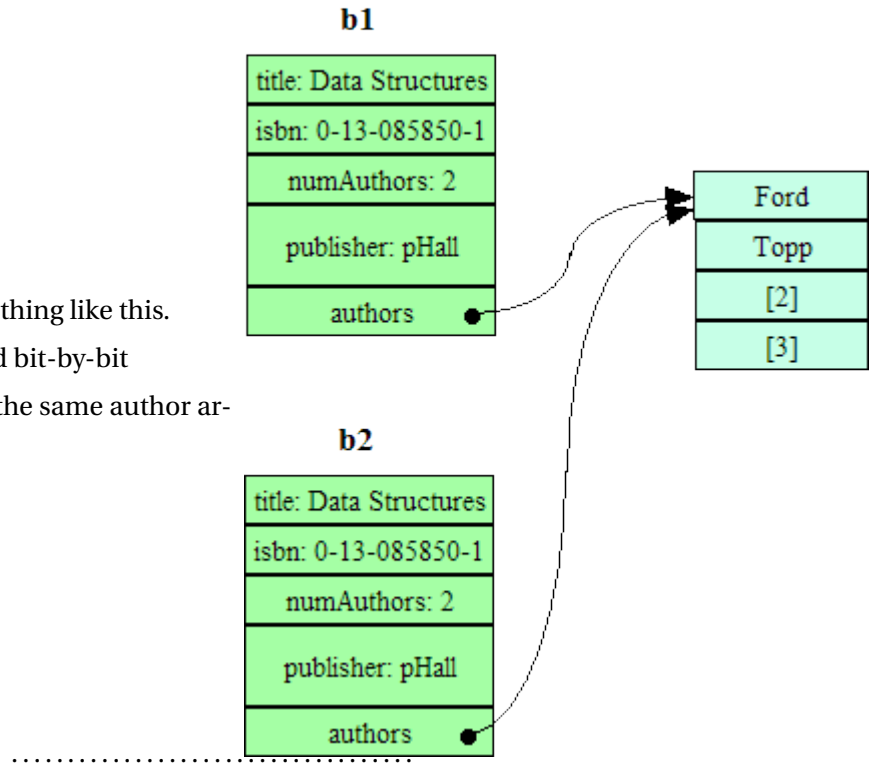
because we have provided no assignment op, the compiler-generated version is used and each data member is copied.



Compiler-Generated Asst

The new result would be something like this.

- The *authors* pointer is copied bit-by-bit
- two book objects now share the same author array



This Seems Familiar...

- We saw earlier that having two books sharing the same array was a bad idea.

- In this case, we now have the additional disadvantage that we have lost all contact with one of the allocated arrays on the heap,
  - a "memory leak" that will never be recovered.
- The compiler-generated assignment operator implements a shallow copy, and that is not what we want.

. . . . . .

Implementing Deep-Copy Assignment

If we want to implement a proper deep copy for our books, we start by adding the operator declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

Then we supply a function body for this operator.

```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = b.numAuthors;
  if (b.numAuthors > MAXAUTHORS)
    {
      MAXAUTHORS = b.MAXAUTHORS;
      delete [] authors;
      authors = new Author[MAXAUTHORS];
    }
  for (int i = 0; i < numAuthors; ++i)
    authors[i] = b.authors[i];
  return *this;
}
```

. . . . . .

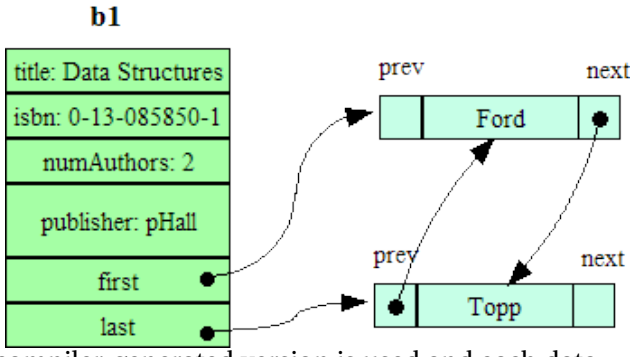
Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

3.1.3 Book - linked list

Book - linked list

Now let's consider the problem of copying a book implemented using linked lists.
If we start with a single book object, `b1`, as shown here, and then we execute

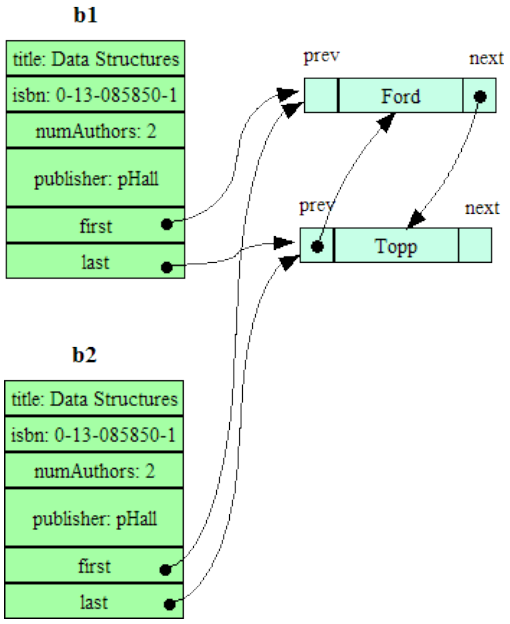
```
b2 = b1;
```



because we have provided no assignment op, the compiler-generated version is used and each data member is copied.

. . . . . .

Shallow Copy of Linked List



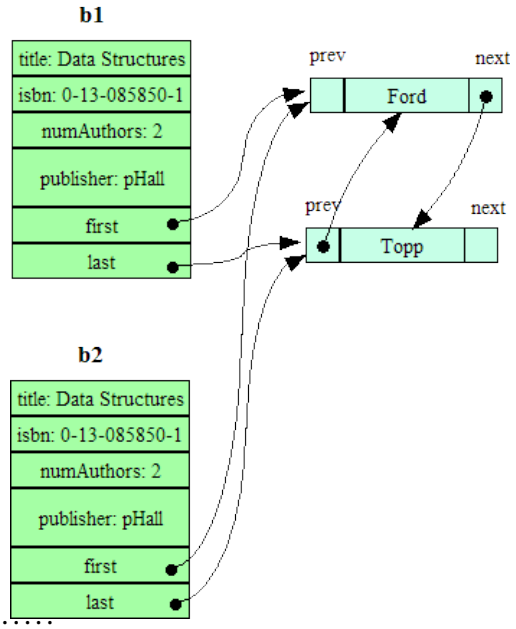
The new result would be something like this.

• Again, the pointer data members (`first` and `last`) are assigned using the default assignment procedure for pointers, which is to simply copy the bits of the pointer.

• That has the result of placing the same node addresses in both book objects. In effect, there is one collection of nodes being shared between both books.

. . . . . .

Shallow Copy of Linked List



• Once again, that's a really, really, bad idea!

• And, once again, the original nodes in `b2` are now unreachable memory leaks.

. . . . . .

Assignment via Linked Lists

If we want to implement a proper deep copy for our books, we start by adding the operator declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

Then we supply a function body for this operator.

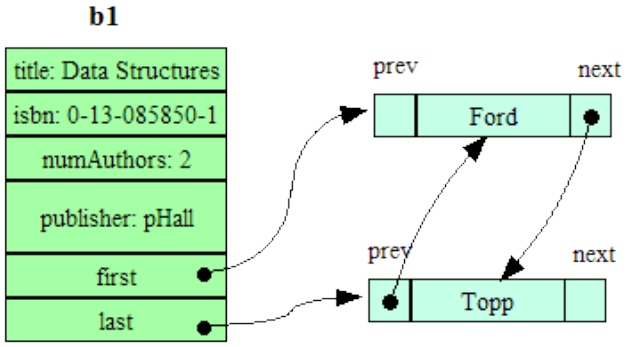
```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = 0;
  for (AuthorPosition p = begin(); p != end();)
    {
      AuthorPosition nxt = p;
```

```
      ++nxt;
      delete p;
      p = nxt;
    }
  first =  last = 0;
  for (AuthorPosition p = b.begin(); p != b.end(); ++p)
    addAuthor(end(), *p);
  return *this;
}
```

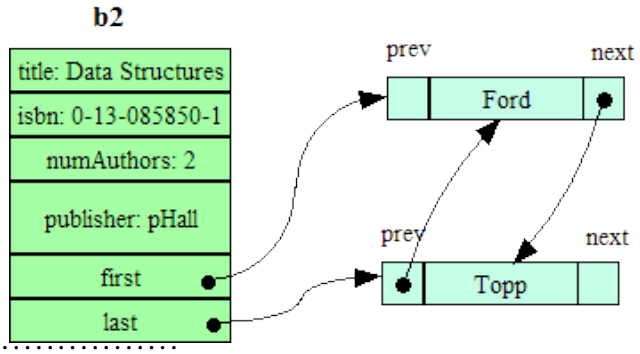
. . . . . .

Most of the data members can be copied easily. But the `first` and `last` pointers are copied by first emptying out any elements already in the list, then copying each author from the source book into the new one.

Assignment Result



The end result after the assignment should be this:



. . . . . .

Self-Assignment

If we assign something to itself:

```
x = x;
```

we normally expect that nothing really happens.

But when we are writing our own assignment operators, that's not always the case. Sometimes assignment of an object to itself is a nasty special case that breaks thing badly.

```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = 0;
  for (AuthorPosition p = begin(); p != end();)
    {
      AuthorPosition nxt = p;
      ++nxt;
      delete p;
      p = nxt;
    }
  first =  last = 0;
  for (AuthorPosition p = b.begin(); p != b.end(); ++p)
    addAuthor(end(), *p);
  return *this;
}
```

What happens if we do `b1 = b1;`?

.....

Self-Assignment Can Corrupt

- The first loop removes all nodes from the destination list.
- The second loop copies all nodes in the source list.
- But if the source and destination are the same, then by the time we reach the second loop, there won't be any nodes left to copy.

So, instead of `b1 = b1;` leaving `b1` unchanged, it would actually destroy `b1`.

.....

Checking for Self-Assignment

```
const Book& Book::operator= (const Book& b)
{
  if (this != &b)
    {
      title = b.title;
      isbn = b.isbn;
      publisher = b.publisher;
      numAuthors = 0;
      for (AuthorPosition p = begin(); p != end();)
        {
          AuthorPosition nxt = p;
          ++nxt;
          delete p;
          p = nxt;
        }
      first =  last = 0;
      for (AuthorPosition p = b.begin(); p != b.end(); ++p)
         addAuthor(end(), *p);
    }
  return *this;
}
```

This is safer.

- We check to see if the object we are assigning *to* (`this`) is at the same address as the one we are assigning *from*
  - the & in the expression `&b` is the C++ *address-of* operator).
- If the two are the same, we leave them alone.
- Only if the two addresses are different do we carry on with the assignment.

.....

You might think that self-assignment is so rare that we wouldn't need to worry about it. But, in practice, you might have lots of ways to reach the same object.

For example, we might have passed the same object as two different parameters of a function call `foo(b1,b1)`. If the function body of `foo` were to assign one parameter to another, we would then have a self-assignment that would likely not have been anticipated by the author of `foo` and that would be very hard to detect in the code that called `foo`.

As another example, algorithms for sorting arrays often contain statements like

```
array[i] = array[j];
```

with a very real possibility that, on occasion, `i` and `j` might be equal.

So self-assignment does occur in practice, and it's a good idea to check for this whenever you write your own assignment operators.

3.1.4 Book - std::list

Book - std::list

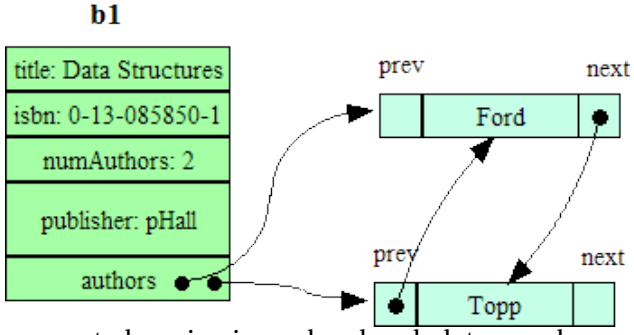
```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;
```

```
  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

Now let's consider the problem of assigning books implemented using `std::list`.

If we start with a single book object, `b1`, as shown here, and then we execute

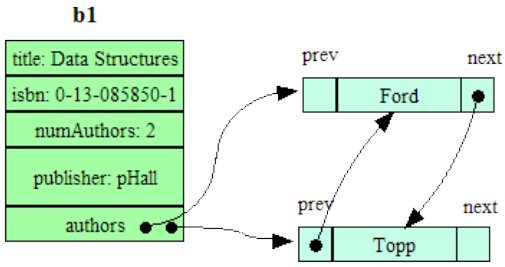
```
b2 = b1;
```



because we have provided no asst op, the compiler-generated version is used and each data member is copied.

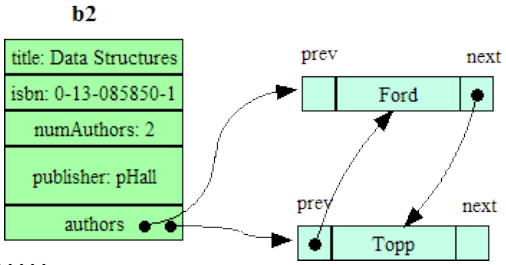
. . . . . .

Assigned std::list



The new result would be something like this. This looks just fine.

We don't need to override the compiler-generated assignment operator in this case.



. . . . . .

3.2 Trusting the Compiler-Generated Assignment Operator

Trusting the Compiler-Generated Assignment Operator

By now, you may have perceived a pattern.

Shallow copy is wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

And it follows that:

Compiler-generated assignment operators are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . . . .

4 Destructors

Destructors

Destructors are used to clean up objects that are no longer in use.

If you don't provide a destructor for a class, the compiler generates one for you automatically.

- The automatically generated destructor simply invokes the destructors for any data member objects.
- If none of the members have programmer-supplied destructors, does nothing.

.....

A Compiler-Generated Destructor

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
   std::string street;
   std::string city;
   std::string state;
   std::string zip;
};


class Author
{
public:
```

```
  Author (std::string theName, Address theAddress, long id);

  std::string getName() const        {return name;}
  void putName (std::string theName) {name = theName;}

  const Address& getAddress() const   {return address;}
  void putAddress (const Address& addr) {address = addr;}

  long getIdentifier() const     {return identifier;}

private:
   std::string name;
   Address address;
   const long identifier;
};
```

We have not declared or implemented a destructor for any of our classes. For `Address` and `Author`, that's OK.

- Note that the *string*s probably do contain pointers, but we trust the `std::string` to handle its own cleanup.

.....

4.1 Destructor Examples

4.1.1 Book - simple arrays

Book - simple arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
```

```
  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

This version of the book has all of its data in a single block of memory. Assuming that each data member knows how to clean up its own internal storage, there's really nothing we would have to do when this book gets destroyed.

We can rely on the compiler-provided destructor.

. . . . . .

4.1.2 Book - dynamic arrays

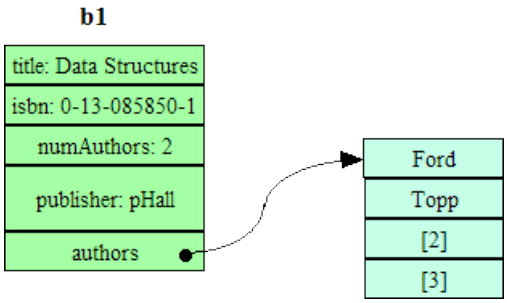
Book - dynamic arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  ~Book();
  const Book& operator= (const Book& b);


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:
```

```
  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

In this version of the Book class, a portion of the data is kept on the heap, in a number of linked list nodes allocated on the heap.

• If this object were destroyed, we would need to be sure that the storage allocated for the array is recovered.



• That won't happen in the compiler-generated destructor, because the default action on pointers is to do nothing.

.....

Implementing the Dyn. ArrayDestructor

We start by adding the destructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);
  ~Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
```

```
    :
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

Then we supply a function body for this destructor.

```
Book::~Book()
{
  delete [] authors;
}
```

Not much needs to be done - just delete the pointer to the array of authors.

.....

4.1.3 Book - linked list

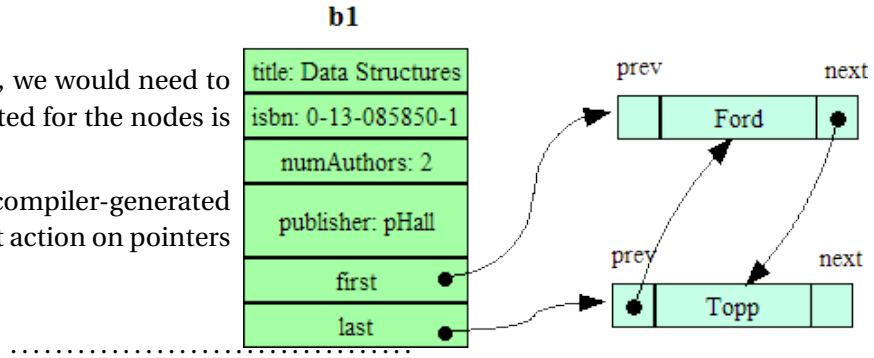
Book - linked list

In this version of the Book class, a portion of the data is kept in an array allocated on the heap.

```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                      // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  ~Book();
```

```
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};

#endif
```

- If this object were destroyed, we would need to be sure that the storage allocated for the nodes is recovered.
- That won't happen in the compiler-generated destructor, because the default action on pointers is to do nothing.



Implementing a Linked List Destructor

We start by adding the destructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);
  ~Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

Then we supply a function body for this destructor.

```
Book::~Book()
{
  AuthorPosition nxt;
  for (AuthorPosition current = begin(); current != end(); current = nxt)
    {
      nxt = current;
      ++nxt;
      delete current.pos;
    }
}
```

This one is more elaborate. We need to walk the entire list, deleting each node as we come to it.

.....

4.1.4 Book - std::list

Book - std::list

- This version of the book has no pointers among its data members.

```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }
```

```
  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

  - It's entirely likely that the `std::list` has pointers inside it, but we trust its destructor to clean up its own internal data structures.

- With that in mind, there's really nothing we would have to do when this book gets destroyed.

  We can rely on the compiler-provided destructor.

.....

4.2 Trusting the Compiler-Generated Destructor

Trusting the Compiler-Generated Destructor

By now, you may have perceived a pattern.

Compiler-generated destructors are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . . . .

5 The Rule of the Big 3

The Big 3

The "*Big 3*" are the

- copy constructor
- assignment operator, and
- destructor

By now, you may have noticed a pattern with these.

. . . . . .

The Rule of the Big 3

The *rule of the big 3* states that,

> *if you provide your own version of any one of the big 3, you should provide your own version of all 3.*

Why? Because we don't trust the compiler-generated...

- copy constructor if our data members include pointers to data we don't share
- assignment operator if our data members include pointers to data we don't share
- destructor if our data members include pointers to data we don't share

So if we don't trust one, we don't trust any of them.

. . . . . .