Implementing ADTs in C++ Classes

Steven J Zeil

August 20, 2013

Contents

1 Implementing ADTs in C++

Implementing ADTs

An ADT is *implemented* by supplying

- a *data structure* for the type name.
- coded *algorithms* for the operations.

We sometimes refer to the ADT itself as the *ADT specification* or the *ADT interface*, to distinguish it from the code of the ADT implementation.

. . . . . .

In C++, this is generally done using a C++ `class`. The class enforces the ADT contract by dividing the information about the implementation into `public` and `private` portions. The ADT designer declares all items that the application programmer can use as public, and makes the rest private. An application programmer who then tries to use the private information will be issued error messages by the compiler.

1.1 Declaring New Data Types

Declaring New Data Types

In the course of designing a program, we discover that we need a new data type. What are our options?

- Use an existing type
- Build our ADT "from scratch" by declaring a new class
- Inherit from an existing class and make slight modifications

. . . . . .

Well, to a certain degree, it depends upon whether or not we already have a data type that provides the interface (data and functions) that we want for this new one.

Use an Existing Type

On rare occasions, we may be lucky enough to have an existing type that provides *exactly* the capabilities we want for our new type. In that case, we can get by just using a `typedef` to create an "alias" for the existing type. For example, suppose that we were writing a program to look up the zip code for any given city. Obviously, one of the kinds of data we will be manipulating will be "city names". Now, a city name is pretty much just an ordinary character string, and anything we can do to a character string we could probably also do to a city name. So we might very well decide to take advantage of the existing C++ `std::string` data type and declare our new one this way:

```
typedef std::string CityName;
```

`typedef` basically gives us a way to attach a more convenient name to an existing type.

. . . . . .

Build our ADT "from scratch"

- Specify exactly what data and functions we want to associate with the new ADT.
- We do this by declaring a new class, e.g.,

```
class Book {
public:
  Book (Author)                 // for books with single authors
  Book (Author[], int nAuthors) // for books with multiple authors

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

  int getNumberOfAuthors() const;

  std::string getIsBN() const;
  void putISBN(std::string id);

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

  AuthorPosition begin();
  AuthorPosition end();

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

private:
  ⋮
};
```

.....

We may need to build our ADT "from scratch", specifying exactly what data and functions we want to associate with the new ADT. This is done by declaring a new class and listing the data and function members we need to achieve our desired abstraction. For example, in building a system to track the inventory of a library, we wanted a Book ADT. In the prior lesson, we saw how we might declare an ADT for this purpose.

Inherit from an existing class

In between those two extremes, we sometimes have an existing type that provides almost everything we want for our new type, but needs just a little more data or a few more functions to make it what we want.

- Create an extended version of the existing type using inheritance

```
class BookInSeries: public Book {
   public:
     std::string getSeriesTitle() const;
     void putSeriesTitle(std::string theSeries);

     int getVolume() const;
     void putVolume(int);
   private:
     std::string seriesTitle;
     int volume;
};
```

.....

In that case we may be able to create an extended version of the existing type by using inheritance. For example, suppose that our library has a number of books that are grouped into series (e.g., “Cookbooks of the World”, volumes 1-28). This leads to the idea (abstraction) of a "book in series", that would be identical to a regular `Book` except for providing two addition data items, a series title and a volume number. We might then write the declaration shown here, which creates a new type named BookIn-Series“ ” that is identical to the existing type Book except that it carries two additional data fields and the functions that provide access to them.

Inheritance is a powerful technique that lies at the heart of object-oriented programming. We’ll explore it in much more detail in later lessons.

1.2 Data Members

Data Members

```
class Book {
 public:
  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int maxAuthors = 12;
  Author* authors;  // array of Authors
};
```

- To be useful, an ADT must usually contain some internal data. These are declared as *data members* of the class.

To continue our prior example, what data should we associate with a book? Earlier, we said that a `Book` has a title, one or more authors, a publisher, and a unique ISBN. We can declare these as shown here. Note that it's not unusual for data members to involve still other ADTs (e.g., the `Author` data type in this example is itself presumably a class with a number of different data fields, including name, address, etc.).

Privacy

```
class Book {
 public:
    ⋮
 private:
  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int maxAuthors = 12;
  Author* authors;   // array of Authors
};
```

In fact, we seldom declare ADTs with a large number of public data members as shown on the previous page. For a variety of reasons, we usually make data members *private*, meaning that they can only be accessed by code that is itself a part of the ADT.

Of course, if that's all we did, the ADT would be of little practical use. It would be rather like building a house with sturdy walls but no exterior doors or windows. If you can't get anything in and out, what use is it? We need some way to affect the data values and to examine them.

1.3 Function Members

Function Members

In addition to data, we can associate selected functions with our ADTs. For example, the missing access to the book data in our prior example can be provided by such *function members*.

```
#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
```

- Now, any member functions that we declare must eventually be implemented as well by providing the appropriate bodies.

  The usual convention is to package each ADT into its own pair of appropriately named `.h` and `.cpp` files. The class declaration for `Book` would typically appear in a file named `book.h`, as shown above.

  Then, in a separate file named `book.cpp` we would place the function definitions (bodies).

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}

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

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

int Book::getNumberOfAuthors() const
{
  return numAuthors;
```

```
}

std::string Book::getISBN() const
{
  return isbn;
}

void Book::setISBN(std::string id)
{
  isbn = id;
}

Publisher Book::getPublisher() const
{
  return publisher;
}

void Book::setPublisher(const Publisher& publ)
{
  publisher = publ;
}

Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
```

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


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

. . . . . .

ADTs need not be complicated

None of the functions in that ADT are particularly complex.

- It's a common mistake to believe that an abstract idea only "needs" to be an ADT if it is complicated.
- Instead, an abstraction should become and ADT if doing so makes the code that uses it more expressive.
  - Most functions in a typical ADT are basic get/set *attribute* functions.

. . . . . .

Inline Functions

Many of the member functions in this example are simple enough that we might consider an alternate approach, declaring them as *inline functions*, in which case the `book.h` file would look something like this:

```
#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;

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

```
};

#endif
```

and the `book.cpp` file would be reduced to this:

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}


Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
```

```
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

.....

Before leaving this example, let's fill out the declarations for the other key ADTs in our library example, starting with the `Author`...

```
class Author
{
public:
  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;
```


```
};
```

...and then the `Address`.

```
class Address {
public:
  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;
};
```

Most of the member functions in these examples are awfully simple. That's not unusual. The majority of ADT member functions really are very simple. There's nothing wrong with that. ADTs do not have to be complicated to be useful.

2 Special Functions

2.1 Constructors

Initializing Data

```
class Address {
public:
  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:
  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;
};
```

- One of the weaknesses of the ADT design shown above is that there is no easy way to initialize new address and author objects.
- Of course, we could do it one data field at a time:

```
Address addr;
addr.putStreet ("21 Pennsylvania Ave.");
addr.putCity ("Washington");
addr.putState ("D.C.");
addr.putZip ("10001");
```

```
Author doe;
doe.putName ("Doe, John");
doe.putAddress (addr);
```

. . . . . .

Problems with Field-By-Field Initialization

- It would be quite easy to forget to initialize one or more of the data fields.
- As the code gets modified over the course of the project, some misguided programmer might place some additional lines of code in between these. (This is particularly likely if we need a nontrivial computation to come up with the values for these data fields.)

  That leads to a real possibility of our using `addr` before all the data fields have been initialized.
- There's no way to initialize the Author's `identifier` data.
  - We don't have a `putIdentifier` function because we did not want to allow arbitrary changes to that data.
  - Note that the `Author::identifier` field is declared as `const`.
- Last, but not least, the process just takes too long - too many lines of code written just to initialize one data object.

. . . . . .

Constructor Functions

C++ provides a special kind of member function to streamline the initialization process. It's called a *constructor*.

Constructors are functions. They can be called just like any other function, can be declared to take any parameters we think they need to get their job done, and they do return a result value, just like ordinary functions. What makes constructors unusual is that their name must be the same as their "return type", the name of the class being initialized. A constructor is called when we define a new variable, and any parameters supplied in the definition are passed as parameters to the constructor.

Suppose we add a constructor to each of our *Address* and *Author* classes.

```
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;
};
```

Declaring Variables with Constructors

Then we can create a new author object much more easily:

```
Address addr ("21 Pennsylvania Ave.",
              "Washington",
              "D.C.", "10001");

Author doe ("Doe, John", addr, 1230157);
```

or, since the `addr` variable is probably only there only for the purpose of initializing this author and is probably not used elsewhere, we can do:

```
Author doe ("Doe, John",
            Address (
             "21 Pennsylvania Ave.",
             "Washington",
             "D.C.", "10001"),
             1230157);
```

.....

Implementing Constructors

The implementation of this constructor is pretty straightforward.

```
Address::Address
  (std::string theStreet, std::string theCity,
   std::string theState, std::string theZip)
{
  street = theStreet;
  city = theCity;
  state = theState;
  zip = theZip;
}
```

.....

Initialization Lists

The implementation of the `Author` constructor has one complication.

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

- We *can't* say

  ```
  identifier = id;
  ```

  in the function body, because `identifier` was declared as being `const` and so cannot be assigned to.

.....

C++ provides a special syntax for initializing constant data members. It's called an *initialization list* and is shown in the highlighted portion of the constructor code.

Initialization lists appear only in constructors and *must* used for constants, references (which also cannot be assigned to) and to initialize data members that need elaborate constructor calls of their own. They *can* be used to initialize any data member, so an alternate implementation of this constructor would be:

Example: Alternate Constructors

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

or

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

The second constructor might run slightly faster.

.....

In the first constructor, the *address* will first be initialized using `Address()`, then in the body we copy over the freshly initialized value when we assign *theAddress* to it. That means that the tie spent initialliy initializing the data member is probably wasted.

2.1.1 Book Constructors

```
#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;

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

};
```

```
#endif
```

Our `Book` class needs constructors as well, but is slightly complicated by the fact that books can have multiple authors. How can we pass an arbitrary number of authors to a constructor (or to any function, for that matter)?

In this case, we'll do it by passing an array of `Authors` together with an integer indicating how many items are in the array.

To implement the constructors, we add each author, however many we are given, into the data structure we chose to hold Authors within our Book ADT. In this case, we use a simple array.

```
// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}
```

With that constructor in place, we could initialize books by first building an appropriate author array, then declaring our new `Book` object. For example, given these authors:

```
Author budd
   ("Budd, Timothy",
    Address("21 Nowhere Dr.", "Podunk", "NY", "01010"),
    1123);
Author doe
   ("Doe, John",
    Address("212 Baker St.", "Peoria", "IL", "12345"),
    1124);
Author smith
   ("Smith, Jane",
    Address("47 Scenic Ct.", "Oahu", "HA", "54321"),
    1125);
```

we can create some books like this:

```
Author textAuthors[] = {budd};
Book text361 ("Data Structures in C++", 1,
              textAuthors, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming",
             1, textAuthors, "0-201-12967");

Author recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", 2, recipeAuthors, "0-124-46821");
```

In cases were we really do have multiple authors (e.g., `recipes`), that's probably as simple as it's going to get. It's a bit awkward for books that only have a single author, however. But, just as with any other functions in C++, we can *overload* functions - we can have any number of functions of the same name as long as the formal parameter list for each constructor is different. Since single authorship is likely to be a common case, it might be convenient to declare another `Book` constructor to serve that special case.

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

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

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

We could then use either constructor, as appropriate, when creating `Book`s.

```
Book text361 ("Data Structures in C++",
              budd, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming",
             budd, "0-201-12967");

Author* recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", 2, recipeAuthors, "0-124-46821");
```

2.2 Destructors

Destructors

The flip-side of initializing new objects is cleaning up when old objects are going away. Just as C++ provides special functions, constructors, for handling initialization, it also provides special functions, destructors, for handling clean-up.

A *destructor* for the class `Foo` is a function of the form

```
~Foo();
```

Destructors are never called explicitly. Instead the compiler generates a call to an object's destructor for us.

.....

The Compiler Calls a Destructor when...

- Execution passes outside of the `{...}` within which the object was declared. For example, if we wrote

```
if (someTest)
  {
   Book b = text361;
   cout << b.getTitle() << endl;
  }
```

what the compiler would actually generate would be something along the lines of

```
if (someTest)
  {
   Book b = text361;
   cout << b.getTitle() << endl;
    b.~Book();  // implicitly generated by the compiler
   }
```

.....

The Compiler Calls a Destructor when...

- When we `delete` a pointer to an object, the object's destructor is (implicitly) called prior to actually recovering the memory occupied by the object. For example, if we were to write:

```
Book *bPointer = new Book(text361); // initialized using copy constructor
   ⋮
cout << bPointer->getTitle() << endl;
delete bPointer;
```

what the compiler would actually generate would be something along the lines of

```
Book *bPointer = new Book(text361); // initialized using copy constructor
    ⋮
cout << bPointer->getTitle() << endl;
 bPointer->~Book();
 free(bPointer);  // recover memory at the address in bPointer
```

.....

Inside a Destructor

The most common uses for destructors is to clean up allocated memory for an object that is about to disappear.

- If we have used `new` to allocate any memory on the heap for one or more data members of the object,
  - we usually `delete` that memory in the destructor.

. . . . . .

2.3 Operators

Operators

One of the truly delightful, or demonic, depending on your point of view, aspects of C++ is that: Almost every operator in the language can be declared in our own classes.

In most programming languages, we can write things like " `x+y` " or " `x<y` " only if `x` and `y` are integers, floating point numbers, or some other predefined type in the language.In C++, we can add these operators to any class we design, if we feel that they are appropriate.

The basic idea is really very simple. Almost all of the things we use as operators,

- including `+ - * / | & < > <= >= = == != ++ -- -> += -= *= /=`,
- and some things you probably don't consider as operators, such as the `[ ]` used in array indexing and the `( )` used in function calls,
- but not `.` or `::`,

are actually "syntactic sugar" for a function whose name if formed by appending the operator itself to the word "`operator`".

. . . . . .

Operators as Shorthand

For example,

- `a + b*(-c)` is actually just a shorthand for

```
operator+(a, operator*(b, operator-(c)))
```

- and if you write

```
testValue = (x <= y);
```

  is a shorthand for

```
operator=(testValue, operator<=(x, y);
```

. . . . . .

Declaring Operators

Now, this shorthand is so much easier to deal with that there's seldom any good reason to actually write out the long form of these calls. But knowing the shorthand means that we can now declare new operators in the same way we declare new functions. For example, we could declare a + operator for `Book`s this way:

```
Book operator+ (const Book& left, const Book& right);
```

and then call it like this:

```
Book b = book1 + book2;
```

but that's probably not a good idea, because it's not clear just what it means to add two books together. So any implementation we wrote for the `operator+` function for `Book`s would probably be nonintuitive and confusing to other programmers. There are, however, a few operators that *would* make sense for `Book` and for almost all ADTs.

. . . . . .

Assignment

Chief among these is the assignment operator.

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

Assignment is used so pervasively that a class without assignment would be severely limited (although sometimes designers may *want* that limitation). Consequently, if a class does not provide an explicit assignment operator, the compiler will attempt to generate one. The compiler-generated version will simply assign each data member in turn.

We'll look more closely at when and how to write our own assignment operators in the next lesson.

. . . . . .

I/O

Another, **very** common set of operators that programmers often write for their own code are the I/O operators << and >>, particularly the output operator <<. Indeed, I would argue that you should always provide an output operator for every class you write, even if you don't expect to use it in your final application.

My reason for this is quite practical. Sooner or later, you're going to discover that your program isn't working right. So how are you going to debug it? The most common thing to do is to add debugging output at all the spots where you think things might be going wrong. So you come to some statement:

```
doSomethingTo(book1);
```

and you realize that it might be really useful to know just what the value of that book was before and after the call:

```
  cerr << "Before doSomethingTo: " << book1 << endl;
  doSomethingTo(book1);
  cerr << "After doSomethingTo: " << book1 << endl;
```

Now, if you have already written that `operator<<` function for your `Book` class, you can proceed immediately. If you haven't written it already, do you really want to be writing and debugging that new function *now*, when you are already dealing with a different bug?

.....

Output Operator Example

Here's a possible output routine for our `Book` class. It simply prints the book's identifier on one line, prints the title on the next line, then prints all the authors separated by commas.

```
ostream& operator<< (ostream& out, const Book& b)
{
  out << b.getISBN() << "\n";
  out << b.getTitle() << "\n";
  for (AuthorPosition current = b.begin(); current != b.end(); ++current)
  {
    Author au = *current;
    out << au << ", ";
  }
  out << "\n  published by" << b.getPublisher();
  out << endl;
  return out;
}
```

- This assumes that we have implemented an output operator for *Author*.
  - But if you bought into the idea of providing an output operator for every class you write, that one should already exist.

.....

Output Operator Example

```
ostream& operator<< (ostream& out, const Book& b)
{
  out << b.getISBN() << "\n";
  out << b.getTitle() << "\n";
  for (AuthorPosition current = b.begin(); current != b.end(); ++current)
  {
```

```
    Author au = *current;
    out << au << ", ";
  }
  out << "\n  published by" << b.getPublisher();
  out << endl;
  return out;
}
```

- The return statement at the end returns the output stream to which we are writing. It's the fact that << is an expression (with a returned value as its result) and that it returns the very stream we are writing to that let's us write "chains" of output expressions like:

```
cout << book1 << " is better than "
     << book2 << endl;
```

which is treated by the compiler as if we had written:

```
(((cout << book1) << " is better than ")
        << book2) << endl;
```

or, if you prefer,

```
operator<<(
           operator<<(
                      operator<<(
                                 operator<<(cout, book1),
                                 " is better than "
                                ),
                      book2
                     ),
           endl
          );
```

so that each output operation, in turn, passes the stream on to the next one in line.

.....

Comparisons

After assignments and I/O, the most commonly programmed operators would be the relational operators, especially == and <. The compiler never generates these implicitly, so if we want them, we have to supply them.

```
class Address
{
   ⋮
```

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

The trickiest thing about providing these operators is making sure we understand just what they should *mean* for each individual ADT. For example, if I were to write

```
if (address1 == address2)
```

what would I expect to be true of two `Addresses` that passed this test? Probably that the two addresses would have the same street, city, state, and zip - in other words, that all the data fields should themselves be equal. In that case, . . .

. . . . . .

Equality via All Data Members Equal

This would be a reasonable implementation:

```
bool Address::operator== (const Address& right) const
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
}
```

We could do something similar for `Author`:

```
bool Author::operator== (const Author& right) const
{
  return (name       == right.name)
      && (address    == right.address)
      && (identifier == right.identifier);
}
```

(which, interestingly, makes use of the `Address::operator==` that we have just defined).

. . . . . .

Equality via "keys"

But there's actually another choice that might be reasonable. If every author has a unique, unchanging identifier, we might be able to just say:

```
bool Author::operator== (const Author& right) const
{
  return (identifier == right.identifier);
}
```

on the grounds that two `Author` objects with the same identifier value must actually be describing the same person, even if they are inconsistent in the other fields.

- That's a different *meaning* for ==.
  - Neither of these two possible meanings is obviously better or "more correct" than the other.
- In a real project, we would need to look hard at how we will use these objects to decide what meaning is appropriate for ==.

. . . . . .

How Many Relational Ops do we Need?

- In C++ we usually provide only operators == and <
  - Many `std::` functions use these two, but none rely on the other 4 (`!= > >= <=`).
- The standard library contains functions defining each of these 4 additional relational operators in terms of == and <.
  - We get these additional operators by simply saying

```
using namespace std::rel_ops;
```

. . . . . .

We might or might not choose to define an `operator!=`. If we do, it's pretty simple, as shown here.

```
class Address
{
   <t>\smvdots</t>
  bool operator== (const Address&) const;

  /*+1*/bool operator!= (const Address& a) const {return !operator==(a);} /*-1*/
   ⋮
};
```

On the other hand, if we don't provide this operator, it's easy enough for programmers using our classes to write tests like:

```
if (!(address1 == address2))
```

Similarly, if we provide any one of the remaining relational operators (`< > <= >=`) then we can derive the others from a combination of the one we provide and `operator==` and `!`. For example, if we have == and <, we can define the others in term of those two:

```
bool operator> (const WordSet& left, const WordSet& right)
{
  return (right < left);
}

bool operator>= (const WordSet& left, const WordSet& right)
{
  return !(right < left);
}
   ⋮
```

In fact, the standard library already contains functions defining each of these 4 additional relational operators in terms of the == and <, so we don't need to write these. We get the additional operators by simply saying

```
using namespace std::rel_ops;
```

Designing a Good <

In C++, we traditionally provide `operator<` whenever possible (and many code libraries assume that this operator is available).

Again, just what this should *mean* depends upon the abstraction we are trying to support, but there are a few hard and fast rules. We should always design `operator<` so that

- If `x < y` is true, then `y < x` and `x == y` must be false.
- If `x == y` is true, then `x < y` and `y < x` must be false.
- If `x == y` is false, then either `x < y` or `y < x` (but not both) must be true.

In other words, given any two values *x* and *y*, one and exactly one of the relations `x < y`, `y < x`, and `x == y` should be true.

.....

Example: Comparing Authors

For example, this would be a reasonable pair of comparison operators for `Authors`:

```
bool Author::operator== (const Author& right) const
{
  return (identifier == right.identifier);
}

bool Author::operator< (const Author& right) const
{
  return (identifier < right.identifier);
}
```

.....

Example: Comparing Addresses

Here is a reasonable pair for `Address`:

```
bool Address::operator== (const Address& right) const
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
}

bool Address::operator< (const Address& right) const
{
  if (street < right.street)
    return true;
  else if (street == right.street)
    {
     if (city < right.city)
       return true;
     else if (city == right.city)
       {
        if (state < right.state)
          return true;
        else if (state == right.state)
          {
           if (zip < right.zip)
            return true;
          }
       }
    }
  return false;
}
```

.....

On the other hand, this pair does *not* work.

```
bool Address::operator== (const Address& right) const
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
```

```
}

bool Address::operator< (const Address& right) const
{
  return (street < right.street)
      || (city   < right.city)
      || (state  < right.state)
      || (zip    < right.zip);
}
```

Can you explain why?

3 Example: Multiple Implementations of Book

Example: Multiple Implementations of Book

One of the characteristics we expect when working with ADTs is that the implementing data structures and algorithms can be altered without changing the interface.

.....

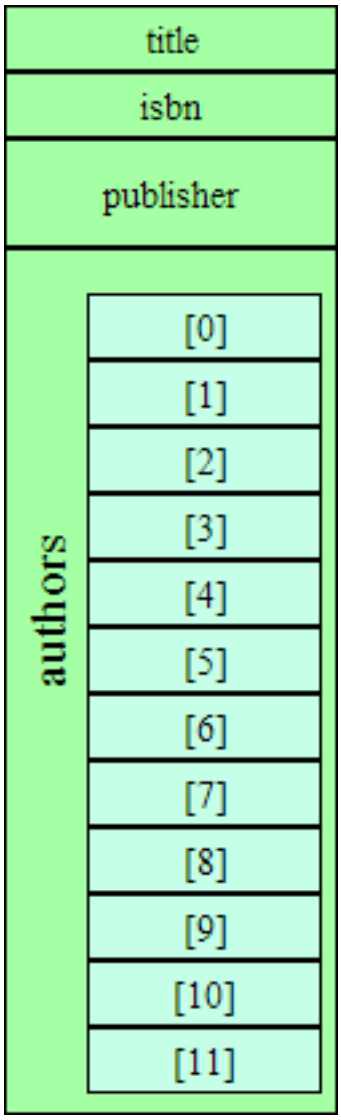
As an example of this, I will present 4 different implementations of the `Book` ADT. The first three should be well within the grasp of everyone in this course. The last will take advantage of a data structure familiar to most students who have taken a C++ -based data structures course, such as our own CS361.

3.1 Simple Arrays

The first of these is the one we have used so far in this lesson.

Simple Arrays

The authors are stored in an array of fixed size, statically allocated as a part of the `Book` object.




```
#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;

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

};

#endif
```

In this approach, each book object can be illustrated as a single block of memory, within which can be found the individual data members such as title and ISBN, but also can be found a fixed number of slots for authors.

Adding and Removing

- The code to add and remove authors is rather typical array manipulation code.

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}


Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
```

```
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

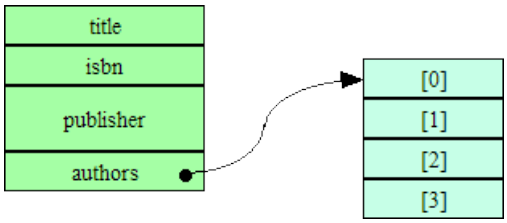
This approach has the advantage of simplicity - it's probably what most novice programmers would attempt. Its disadvantages include speed when adding or removing authors from very long lists (though, realistically, very few books are likely to have long enough lists of authors for this to really matter). More important, the choice of value for `MAXAUTHORS` is somewhat critical. Too small and we risk a program crash when a book is created with more authors than can fit in the array. Too large, and we waste a lot of storage for the vast majority of books that only have one or two authors.

3.2 Dynamically Allocated Arrays

Dynamically Allocated Arrays

A more flexible approach can be obtained by allocating the array of authors on the heap.

In this approach, each book object occupies two distinct blocks of memory, one of them an array allocated on the heap.




```
#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
```

In the .h file, the statically sized array `authors` is replaced by a pointer to an (array of) `Author`.

.....

Dynamically Allocated Arrays (cont.)

```
#include "book2.h"

  // for books with single authors
Book::Book (Author a)
{
  MAXAUTHORS = 4;
  authors = new Author[MAXAUTHORS];
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  MAXAUTHORS = 4;
  authors = new Author[MAXAUTHORS];
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}
```

```
Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


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;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
```

```
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

The code is still pretty straightforwardly array-based.

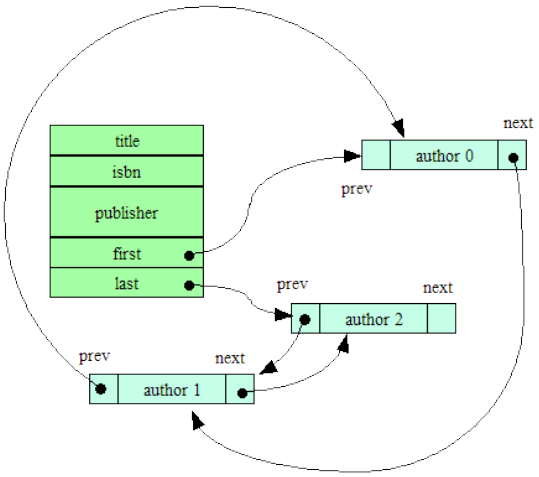
- The constructors are changed to now allocate the array on the heap.
- The array size allocated is smaller, because of a more sophisticated approach used in `addAuthor`.
  - There, before adding a new author to the book, we make a check to see if the array is already full.
  - If it is, we allocate a new, larger array, copy the former list of authors from the old array into the new one, discard the old array, and then proceed to add the new author into the new, larger array.
    That way we need not worry about choosing a fixed bound on the number of authors that can be accommodated in any book. (Some of you may recognize this expandable array approach as being typical of the `std::vector` class in the C++ standard library.)

. . . . . .

3.3 Linked Lists

Linked Lists

A still more flexible approach can be obtained by using a linked list of authors. In this case, I have chosen a doubly-linked list (pointers going both forward and backward from each node).



In this approach, each book object occupies many blocks of memory, most of them being linked list nodes 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


  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
```

The Book itself holds pointers to the first and the last node in the chain.

.....

Linked Lists impl

```
#include "authoriterator.h"
#include "book3.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  first = last = new AuthorNode (a, 0, 0);
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = 0;
  first = last = 0;
  for (int i = 0; i < nAuthors; ++i)
    {
      addAuthor(end(), au[i]);
    }
}


Book::AuthorPosition Book::begin() const
{
  return AuthorPosition(first);
}

Book::AuthorPosition Book::end() const
```

```
{
  return AuthorPosition(0);
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  if (first == 0)
     first = last = new AuthorNode (author, 0, 0);
  else if (at.pos == 0)
    {
      last->next = new AuthorNode (author, last, 0);
      last = last->next;
    }
  else
    {
      AuthorNode* newNode = new AuthorNode(author, at.pos->prev, at.pos);
      at.pos->prev = newNode;
      if (at.pos == first)
    first = newNode;
      else
    newNode->prev->next = newNode;
    }
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  if (at.pos == first)
    {
      if (first == last)
        first = last = 0;
      else
    {
      first = first->next;;
      first->prev = 0;
    }
    }
  else if (at.pos == last)
```

```
    {
      last = last->prev;
      last->next = 0;
    }
  else
    {
      AuthorNode* prv = at.pos->prev;
      AuthorNode* nxt = at.pos->next;
      prv->next = nxt;
      nxt->prev = prv;
    }
  delete at.pos;
  --numAuthors;
}
```

The code is considerably messier - pointer manipulation can be tricky, no matter how many times you do it.

- The code is quite efficient however.
- Adding and removing nodes remains quick no matter how many authors are actually in the list.

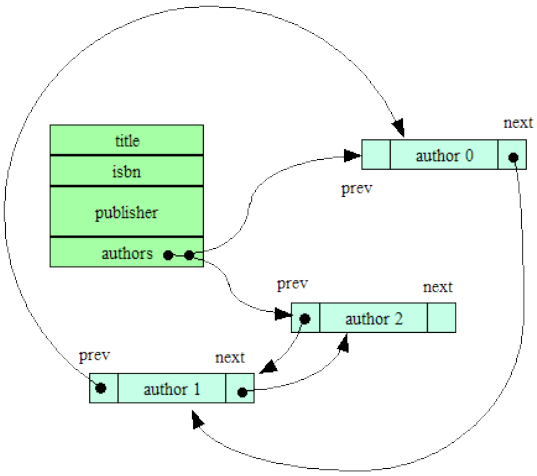
.....

3.4 Standard List

Those of you who have taken CS361 or a similar course might guess that we could save ourselves a lot of programming effort by exploiting one of the standard container data structured provided in the C++ std library. So, for our final implementation, I present an implementation based upon `std::list`.

Standard List

We presume that the data in this approach is arranged pretty much as before, though we don't really *know* that for sure because we trust the abstraction provided by the `std::list` ADT.




```
#ifndef BOOK_H

#include <list>

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


class Book {
public:
  typedef std::list<Author>::const_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
```

```
#include "authoriterator.h"
#include "book4.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors.push_back(a);
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
```

```
      authors.push_back(au[i]);
    }
}


Book::const_AuthorPosition Book::begin() const
{
  return authors.begin();
}

Book::const_AuthorPosition Book::end() const
{
  return authors.end();
}


Book::AuthorPosition Book::begin()
{
  return authors.begin();
}

Book::AuthorPosition Book::end()
{
  return authors.end();
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  authors.insert (at, author);
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  authors.erase (at);
  --numAuthors;
}
```


The code is quite simple, at least for anyone already familiar with the `std::list` ADT.

3.5 Implementing Book::AuthorPosition

Implementing Book::AuthorPosition

Our `Book` ADT calls for a "helper" ADT, the `AuthorPosition`, to represent the position of a particular author within the author list for the book.

Although we have looked at the desired interface (10) for iterators such as `AuthorPosition`, we skipped past its implementation while we were looking at implementing `Book`.

Iterator Interface Review

To review, our iterator must supply the following operations:

```
class AuthorPosition {
public:
   AuthorPosition();

   // get data at this position
   Author operator*() const;

   // get a data/function member at this position
   Author* operator->() const;

   // move forward to the position just after this one
   AuthorPosition operator++();

   // Is this the same position as pos?
   bool operator== (const AuthorPosition& pos) const;
   bool operator!= (const AuthorPosition& pos) const;

};
```

A little thought will lead to the realization that, to "get data at this position", this ADT will need to access the underlying data structure used to implement `Book`. This may seem a bit odd, since we have gone to great lengths to hide that data structure from the rest of the world. But if you look back at our various declarations of the `Book` class, you can see that we declare the `AuthorPosition` *inside* the `Book` class. It's a part of `Book` and contributes to the support of the `Book` abstraction, giving it something of a privileged position.

Another consequence of this close relationship between the two classes is that the implementation of `AuthorPosition` will be different for each of the different implementations of `Book` that we have considered.

3.5.1 Simple Arrays

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        { 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;

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

};

#endif
```

If we implement `Book` using simple arrays, the implementation of `AuthorPosition` is trivial.

It takes advantage of the fact that the conventional interface for iterators in C++ is chosen to mimic the behavior of pointers to array elements.

. . . . . .

- Suppose we have an array `a`

```
T a[N];
  ⋮
for (int i = 0; i < N; i++)
   a[i] = foo(a[i]);
```

  - In C++, an array name without an index is actually a pointer to the first element:

```
*a == a[0]
```

  - adding 1 to a pointer is equivalent to moving one element forward in an array:

```
(a+1) == a[1], *(a+2) == a[2], etc.
```

So here is another way to iterate over an array:

```
T a[N];
 ⋮
for (T* p = a; p != a+N; p++)
   *p = foo(*p);
```

It does the exact same thing as the first form of iteration:

```
for (int i = 0; i < N; i++)
    a[i] = foo(a[i]);
```

C++ iterators are designed to mimic this behavior:

```
Container c;
 ⋮
for (iterator p = c.begin(); p != c.end(); p++)
    *p = foo(*p);
```

Iterators and Arrays

The only differences lie in how one gets access to the starting and ending positions.

	`T a[N];`	`Container c;`
get element at position	`*p p->`	`*it it->`
move forward 1 position	`++p`	`++it`
compare positions	`p1 == p2` `p1!= p2`	`it1 == it2` `it1 != it2`
position of first element	`a`	`c.begin()`
position just after last element	`a+N`	`c.end()`

Consequently, we can actually implement the `AuthorPosition` ADT as a simple pointer to one of the array elements in `Book`'s array of authors.

Book (Array) Iterator

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

Book::AuthorPosition Book::begin() const
{
 return authors;
}

Book::AuthorPosition Book::end() const
{
 return authors + numAuthors;
}
```

The pointer type already provides the `*`, `->`, `++`, `==`, and `!=` operations we need.

So we just needed to implement the `begin()` and `end()` functions for `Book`.

3.5.2 Dynamically Allocated Arrays

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
```

Everything we have said about iterators over simple arrays applies equally well to dynamic arrays. So the implementation of AuthorPostion is identical:

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

.....

3.5.3 Linked Lists

Linked Lists

```
#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
```

If the authors of a book are kept as a linked list, the iterator type requires a bit more work. The logical first thought for a way to designate a position within a linked list is to use a pointer to a linked list node. But, unlike the pointers to array elements, pointers to linked list nodes cannot serve as implementations of iterators. If `p` is a pointer to a linked list node, then `*p` returns the whole node, *not* the data stored inside that node, which is what we want from `*iterator`. Also, the operation `++p`, although it will compile, is technically illegal in C++ as increment of pointers is only legal within an array. What we *really* want `++iterator` to do is to follow the `next` link from one linked list node to another.

To get the desired behaviors for our iterator ADT, we will need to implement it as a class in its own right. The declaration shown here provides our desired interface.

```
class AuthorIterator {
public:
  AuthorIterator ();

  Author operator*() const;
  const Author* operator->() const;
```

```
  const AuthorIterator& operator++(); // prefix form ++i;
  AuthorIterator operator++(int); // postfix form i++;

  bool operator== (const AuthorIterator& ai) const;
  bool operator!= (const AuthorIterator& ai) const;

private:
  AuthorNode* pos;
  AuthorIterator (AuthorNode* p)
    : pos(p)
  {}

  friend class Book;
};
```

Inside the `Book` class, we write

```
class Book {
public:
   typedef AuthorIterator AuthorPosition;
     ⋮
 friend class AuthorIterator;
};
```

which gives `AuthorPosition` access to all private members of `Book`.

. . . . . .

Our `AuthorIterator` has one (hidden) data member, a pointer to a linked list node, which we use to indicate a position within the author list. But the operations we provide will interpret this pointer in a way that gives it iterator-appropriate behavior.

Implementing the Iterator Ops

For example, we implement the * operator like this:

```
Author AuthorIterator::operator*() const
{
  return pos->au;
}
```

returning the data field (only) from the linked list node.

. . . . . .

Implementing the Iterator Ops (cont.)

The other particularly interesting operator is ++:

```
// prefix form ++i;
const AuthorIterator& AuthorIterator::operator++()
{
  pos = pos->next;
  return *this;
}

// postfix form i++;
AuthorIterator AuthorIterator::operator++(int)
{
  AuthorIterator oldValue = *this;
  pos = pos->next; return oldValue;
}
```

The main thing that happens here is simple advancing the *pos* pointer to the next linked list node.

.....

3.5.4 Standard List

Standard Lists

```
#ifndef BOOK_H

#include <list>

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


class Book {
public:
  typedef std::list<Author>::const_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
```

If we use a `std::list` to keep our authors, the implementation of our iterator becomes fairly simple again.

- That's because all the `std` containers provide their own iterators
  - (which, obviously, will follow the `std` interface), and so
  - we can simply implement our own iterator in terms of that one:

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

```
typedef std::list<Author>::const_iterator const_AuthorPosition;
```

. . . . . .

The one complication is that we will actually need two iterator types, one for use with `const Book` objects and one for use with non-const `Book`s. This is a common idiom in C++, which we will discuss later when we look at "const correctness".

Implementing the Iterator Ops

Implementing the `begin()` and `end()` functions is pretty straightforward.

```
Book::const_AuthorPosition Book::begin() const
{
  return authors.begin();
}

Book::const_AuthorPosition Book::end() const
{
  return authors.end();
}

Book::AuthorPosition Book::begin()
{
  return authors.begin();
}

Book::AuthorPosition Book::end()
{
  return authors.end();
}
```

. . . . . .