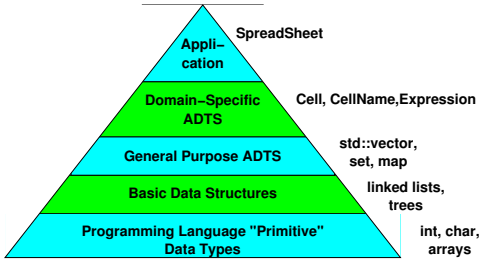
ADTs

Steven J Zeil

August 20, 2013

Contents

If we were to look at a program that is actually large enough to require a full team of programmers to implement it, you would probably not be surprised to find that it would not be organized as a single, large, monolithic unit, but instead as a large number of cooperating functions. You already know how to design and write functions in C++ . What may, however, come as a bit of a surprise if you have not worked much with programs that size is that even these functions will be further organized into higher level structure.



I've tried to illustrate that structure in the diagram that you see here. At the top we have the main application program, for example, a spell checker. The code that occurs at this level is very specific to this application and is the main thing that differentiates a spell checker from, let's say, a spreadsheet. On the other hand, at the very bottom of the hierarchy, we have all the basic primitive data types and operations, such as the int type, the char type, addition, subtraction, and so on, that are provided by our programming language, (C++ in this example). These primitive operations may very well show up in almost any kind of program.

In between those, we have all the things that we can build on top of the language primitives on our way working up towards our application program. Just above the language primitives we have the basic data structures, structures like linked lists or trees. We're going to spend a lot of time the semester looking at these kinds of structures - you may already be familiar with some of them. They are certainly very important. And yet, if we stopped at that level, we would wind up "building to order" for every application. As we move from one application to another we would often find ourselves doing the same kinds of coding, over and over again.

What's wrong with that? Most companies, and therefore most programmers, do not move from one application to a wildly different application on the next project. Programmers who been working on "accounts receivable" are unlikely to start writing compilers the next week, and programmers who have been writing compilers are not going to be writing control software for jet aircraft the week after. Instead, programmers are likely to remain within a general application domain. The people who are currently working on our spell checker may very well be assigned to work on a grammar checker next month, or on some other text processing tool. That means that any special support we can design for dealing with text, words, sentences, or other concepts natural to this application to make may prove valuable in the long run because we can share that work over the course of several projects.

And so, on top of the basic data structures, we expect to find layers of reusable libraries. Just above the basic data structures, the libraries are likely to provide fairly general purpose structures, such as support for look-up tables, user interfaces, and the like. As we move up in the hierarchy, the libraries become more specialized to the application domain in which we're working. Just below the application level, we will find support for concepts that are very close to the spell checker, such as "words", "misspellings", and "corrections".

The libraries that make up all but the topmost layer of this diagram may contain individual functions or groups of functions organized as Abstract Data Types. In this lesson, we'll review the idea of Abstract

Data Types and their implementations. Little, if any, of the material in this lesson should be entirely new to you - all of it is covered in CS 250.

1 Abstraction

Abstraction

In general,, *abstraction* is a creative process of focusing attention on the main problems by ignoring lower-level details.

In programming, we encounter two particular kinds of abstraction:

- *procedural abstraction* and
- *data abstraction*.

. . . . . .

1.1 Procedural Abstraction

Procedural Abstraction

A *procedural abstraction* is a mental model of *what* we want a subprogram to do (but not *how* to do it).

Example: if you wanted to compute the length of the a hypotenuse of a right triangle, you might write something like

```
double hypotenuse = sqrt(side1*side1 + side2*side2);
```

We can write this, understanding that the `sqrt` function is supposed to compute a square root, even if we have no idea how that square root actually gets computed.

- That's because we understand what a square root *is*.

. . . . . .

When we start actually writing the code, we *implement* a procedural abstraction by

- assigning an appropriately named "function" to represent that procedural abstraction,
- in the "main" code, calling that function, trusting that it will actually do *what* we want but not worrying about *how* it will do it, and
- finally, writing a function body using an appropriate algorithm to do what we want.

In practice, there may be many algorithms to achieve the same abstraction, and we use engineering considerations such as speed, memory requirements, and ease of implementation to choose among the possibilities.

For example, the `sqrt` function is probably implemented using a technique completely unrelated to any technique you may have learned in grade school for computing square roots. On many systems. `sqrt` doesn't compute a square root at all, but computes a polynomial function that was chosen as a good approximation to the actual square root and that can be evaluated much more quickly than an actual square root. It may then refine the accuracy of that approximation by applying Newton's method, a technique you may have learned in Calculus.

Does it *bother* you that `sqrt` does not actually compute via a square root algorithm? Probably not. It shouldn't. As long as we trust the results, the method is something we are happy to ignore.

1.2 Data Abstraction

Data Abstraction

Data abstraction works much the same way. A *data abstraction* is a mental model of `what` can be done to a collection of data. It deliberately excludes details of `how` to do it.

. . . . . .

Example: calendar days

A day (date?) in a calendar denotes a 24-hour period, identified by a specific year, month, and day number.

That's it. That's probably all you need to know for you and I to agree that we are talking about a common idea.

. . . . . .

Example: cell names

One of the running examples I will use throughout this course is the design and implementation of a spreadsheet. I assume that you are familiar with some sort of spreadsheet program such as Microsoft's Excel or the OpenOffice Calc program. All spreadsheets present a rectangular arrangement of cells, with each cell containing a mathematical expression or formula to be evaluated.

Every cell in a spreadsheet has a unique name. The name has a column part and a row part.

- The row indicators are integer values starting at 1.
- The column indicators are case-insensitive strings of alphabetic characters as follows: `A`, `B`, . . . , `Z`, `AA`, `AB`, `AC`, . . . , `AZ`, `BA`, `BB`, . . . `ZZ`, `AAA`, `AAB`, . . . and so on.
- Optional `$` markers may appear in front of each part to "fix" the row or column during copying. For example, suppose we have a cell `B1` containing the formula `2*A1` and that we copy and paste that cell to position C3. When we look in `C3`, we would find the pasted formula was `2*B3` adjusted for the number of rows and columns over which we moved the copied value.

  But if the cell `B1` originally contained the formula `2*A$1`, the copied formula would be `2*B$1`. The `$` indicates that we are fixing the column indicator during copies. Similarly, if the cell `B1` originally

contained the formula `2*$A$1`, the copied formula would be `2*$A$1`. (If this isn't clear, fire up a spreadsheet and try it. We can't expect to share mental models (abstractions) if we don't share an experience with the subject domain.)

. . . . . .

Example: a book

How to describe a book?

- If we are implementing a card catalog and library checkout, it is probably enough to list the meta-data
  - (e.g., title, authors, publisher, date).
- If, however, we are going to be working on a project involving the full text of the document (e.g., automatic metadata extraction and indexing), then we might need all the pages and all the text.
- Of course, if we were building bookshelves, we might need more physical attributes such as size and weight!

. . . . . .

Example: positions within a container

Many of the abstractions that we work with are "containers" of arbitrary numbers of pieces of other data. This is obvious with things like arrays and lists, but is also true of more prosaic items. For example, a book is, in effect, a container of an arbitrary number of authors (and in other variations, an arbitrary number of pages). Any time you have an ordered sequence of data, you can imagine the need to look through it. That then leads to the concept of a *position within that sequence*, with notions like

- finding the first and last position,
- going forward to the next position, etc.

. . . . . .

2 Abstract Data Types

Adding Interfaces

- The *mental model* offered by a data abstraction gives us an informal understanding of how and when to use it.
  - But because it is simply a mental model, it does not tell us enough information to program with it.

- An *abstract data type* (ADT) captures this model in a programming language interface.

. . . . . .

Definition of an Abstract Data Type

Definition (traditional): An *abstract data type* (ADT) is a type name and a list of operations on that type.

It's convenient, for the purpose of this course, to modify this definition just slightly:

Definition (alternate): An *abstract data type* (ADT) is a type name and a list of members (data or function) on that type.

- an ADT corresponds, more or less, to the public portion of a typical class
- the "list of members" includes
  - names
  - data types
  - expected behavior
- a.k.a. an *ADT specification*

. . . . . . This change is not really all that significant. Traditionally, a data member X is modeled as a pair of `getX()` and `putX(x)` functions. But in practice, we will allow ADTs to include data members in their specification. This definition may make it a bit clearer that an ADT corresponds, more or less, to the public portion of a typical class.

In either case, when we talk about listing the members, this includes giving their names and their data types (for functions, their return types and the data types of their parameters).

If you search the web for the phrase "abstract data type", you'll find lots of references to stacks, queues, etc. - the "classic" data structures. Certainly, these *are* ADTs. But, just as with the abstractions introduced earlier, each application domain has certain characteristic or natural abstractions that may also need programming interfaces.

ADT Members: attributes and operations

Commonly divided into

- *attributes*: the things that we think of as being data stored inthe ADT
  - Actual interface is often through `get`*`Attr`*`()` and `set`*`Attr`*`()` functions.
  - which, in turn, might or might not actually involve direct access to a "data member"
- operations: the functions or behaviors or the ADT

– the "type" of a function consists of its return type and an ordered list of of its parameters' types

2.1 Examples

Calendar Days

Nothing in the definition of ADT that says that the interface *has* to be written out in a programming language.

UML diagrams present classes as a 3-part box: name, attributes, & operations

Day
day: int month: int year: int
+(#days: int): Day -(: Day): int <, ==

Calendar Days: alternative

But we can use a more programming-style interface:

```
class Day {
public:
   // Attributes
   int getDay();
   void setDay (int);
   int getMonth();
   void setMonth(int);
   int getYear();
   void setYear(int);

   // Operations
   Day operator+ (int numDays);
   int operator- (Day);
   bool operator< (Day);
   bool operator== (Day);
     ⋮
```

See also the interface developed in sections 3.1 and 3.2 of your text (Horstmann).

- It's essentially the same ADT, but lots of details are different

Notations

Day
day: int month: int year: int
+(#days: int): Day -(: Day): int <, ==

```
class Day {
public:
   // Attributes
   int getDay();
   void setDay (int);
  ⋮
```

- Disadvantages of moving early to programming-style interfaces:
  - getting lost in language details
  - prematurely committing to those details

.....

Cell Names

Here is a possible interface for our cell name abstraction.

```
class CellName
{
public:
  CellName (std::string column, int row,
            bool fixTheColumn = false,
            bool fixTheRow=false);
  //pre: column.size() > 0 && all characters in column are alphabetic
  //     row > 0

  CellName (std::string cellname);
  //pre: exists j, 0<=j<cellname.size()-1,
  //        cellname.substr(0,j) is all alphabetic (except for a
  //             possible cellname[0]=='$')
  //        && cellname.substr(j) is all numeric (except for a
  //             possible cellname[j]=='$') with at least one non-zero
  //             digit

  CellName (unsigned columnNumber = 0, unsigned rowNumber = 0,
            bool fixTheColumn = false,
            bool fixTheRow=false);
```

```
  std::string toString() const;
  // render the entire CellName as a string

  // Get components in spreadsheet notation
  std::string getColumn() const;
  int getRow() const;

  bool isRowFixed() const;
  bool isColumnFixed() const;


  // Get components as integer indices in range 0..
  int getColumnNumber() const;
  int getRowNumber() const;


  bool operator== (const CellName& r) const
     ⋮
private:
     ⋮
```

CellName
row: int column: string columnNumber: int isRowFixed: bool isColumnFixed: bool
toString(): string == (CellName): bool

Arguably, the diagram on the left presents much the same information.

.....

Example: a book

If we were to try to capture our book abstraction (concentrating on the metadata), we might come up with something like:

```

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:
  ⋮
};
```

- What are `Author` and `Publisher` in this interface?
  - They are simply other ADTs in this library world, and will need to have designed interfaces of their own.

.....

2.1.1 Example: positions within a container

Example: positions within a container

Coming up with a good interface for our position abstraction is a problem that has challenged many an ADT designer.

- A look at our `Book` interface may suggest why.

```
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);

  typedef int AuthorPosition;
  <+1>Author getAuthor (AuthorPosition authorNum) const;  <-1>

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

private:
  ⋮
};
```

- One intuitive idea might be to simply number the authors and treat the number as a position indicator, as shown here.

..... A problem with this is that the `getAuthor` function could then be done efficiently only if the authors inside each book were stored in an array or array-like data structure. And then `addAuthor` and `removeAuthor` cannot be implemented efficiently. Arrays also pose a difficulty – how large an array should be allocated for this purpose? If we allocate too few, the program crashes. So programmers usually wind up allocating an array large enough to contain as many items as possible – in this case as many authors as have ever collaborated on a single book. This means a lot of wasted storage for most books.

Both C++ and Java provide "expandable" array types, `std::vector` and `java.util.ArrayList`, respectively, that grow to accommodate however much data you actually insert into them. These resolve the storage issue, at a slight cost in speed, but still do not permit efficient implementation of `addAuthor` and `removeAuthor`.

Iterators

The solution adapted by the C++ community is to have every ADT that is a "container" of sequences of other data to provide a special type for positions within that sequence.

- The container itself provides functions to return
  - the beginning position in the sequence and
  - the position just *after* the last data item in the
    (these are typically called `begin()` and `end()`
- The position ADT must provide, at a minimum:
  - A function to fetch the data item at the given position.
  - A function to advance from the current position to the next position in the sequence.
  - A function to compare two positions to see if they are the same.

.....

A Possible Position Interface

In theory, we could satisfy this requirement with an ADT like this:

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

   // get data at this position
   Author getData() const;

   // get the position just after this one
   AuthorPosition next() const;

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

};
```

which in turn would allow us to access authors like this:

```
void listAllAuthors(Book& b)
{
   for (AuthorPosition p = b.begin(); p != b.end();
        p = p.next())
     cout << "author: " << p.getData() << endl;
}
```

.....

The Iterator ADT

For historical reasons (and brevity), however, C++ programmers use overloaded operators for the `getData()` and `next()` 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;

};
```

so that code to access authors would look like this:

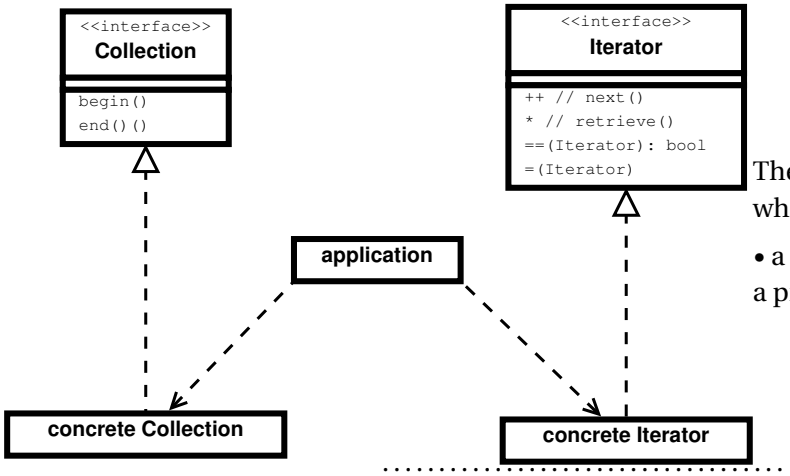
```
void listAllAuthors(Book& b)
{
   for (AuthorPosition p = b.begin(); p != b.end();
        ++p)
     cout << "author: " << *p << endl;
}
```

This ADT for positions is called an *iterator* (because it lets us iterate over a collection of data).

. Java has similar ADTs for positions within a container, called `Enumeration` and `Iterator`, which we will see in a later lesson.

2.2 Design Patterns

Iterator as a Design Pattern



The idea of an iterator is an instance of what we call a *design pattern*:

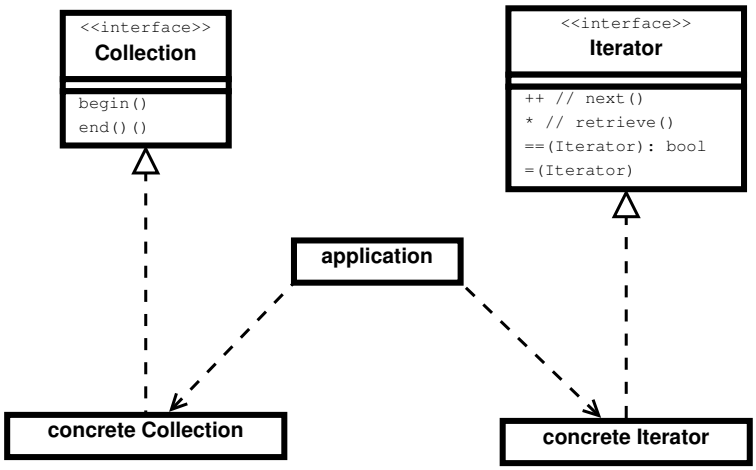
• a reusable concept that is not so much a piece of code as it is a design idea.

Pattern, not ADT

In C++, our application code does not actually work with an actual ADT named "Iterator".

- Instead, we typically have a lot of different ADTs, all of which share the common *pattern* of supporting an operator ++ to move forward, an operator $*$ for fetching a value at the position, etc.
- Each of these iterator ADTs is related to some kind of collection of data.
  - These collections have nothing to do with one another except that they share the common idea of supplying `begin()` and `end()` functions to provide the beginning and ending position within that collection. Again, there's probably no class in our application code actually named "Collection".

Realizing a Design Pattern



- *Iterator* and *Collection* are general patterns for interfaces.
- We will have many actual classes that are unrelated to one another but that *implement* or *realize* those patterns.
  - Such classes are called *concrete* realizations of the general patterns.
  - It's these concrete classes that our application actually works with.

You may have noticed that your textbook is titled "Objected-Oriented Design *& Patterns*". Keep an eye out, as we move through the semester, for more instances of common design patterns. (You might want to compare this diagram to the more Java-oriented version of this pattern on page 178 - in Java there really *is* something in the library named Iterator and our application works directly with that and only indirectly with the concrete realization.)

3 ADTs as contracts

ADTs as contracts

An ADT represents a contract between the ADT developer and the users (application programmers).

The Contract

- Application writers[a] are expected to alter/examine values of this type only via the operations and members provided.
- The creator of the ADT promises to leave the operation specifications unchanged.
- The creator of the ADT is allowed to change the code of the operations at any time, as long as it continues to satisfy the specifications.
- The creator of the ADT is also allowed to change the data structure actually used to implement the type.

[a] the "users" of the ADT

Why the Contract?

What do we gain by holding ourselves to this contract?

- Application programmers can be designing and even implementing the application before the details of the ADT implementation have been worked out. This helps in
  - top-down design
  - development by teams
- The ADT implementors knows exactly what they must provide and what they are allowed to change.
- ADTs designed in this manner are often re-usable. By reusing code, we save time in
  - implementation
  - debugging
- We gain the flexibility to try/substitute different data structures to actually implement the ADT, *without needing to alter the application code.*
- By encouraging modularity, application code becomes more readable.

. . . . . .

Look back at the sample ADTs from the previous sections. Note that, although none of them contain data structures or algorithms to actually provide the required functions, all of them provide enough information that you could start writing application code using them.

3.1 Information Hiding

Information Hiding

Every design can be viewed as a collection of "design decisions".

- David Parnas formulated the principle: "Every module [procedure] should be designed so as to hide one design decision from the rest of the program."
  - He argued that such information hiding made future changes more economical.

. . . . . .

Encapsulation

Although ADTs can be designed without language support, they rely on programmers' self-discipline for enforcement of information hiding.

- Like "airline food" and "military intelligence", some would argue that "programmers' self-discipline" is an oxymoron.
- So programming languages evolved to provide enforcement of the ADT contract.

Encapsulation is the enforcement of information hiding by programming language constructs. In C++, this is accomplished by allowing ADT implementors to put some declarations into a `private:` area. Any application code attempting to use those private names will fail to compile.

. . . . . .

4 ADT Implementations

ADT Implementations

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++, implementation is generally done using a C++ `class`.

- Uses public/private to enforce the ADT contract

. . . . . .

4.1 Examples

Calendar Day Implementations

Read section 3.3 of your text (Horstmann) for a discussion of three different implementations of the `Day` ADT. notice how we can choose among implementations for performance reasons, without breaking any application code that relies on the `Day` interface.

. . . . . .

CellName implementation

```
class CellName
{
public:
  CellName (std::string column, int row,
            bool fixTheColumn = false,
            bool fixTheRow=false);
  //pre: column.size() > 0 && all characters in column are alphabetic
  //     row > 0
```

```
  CellName (std::string cellname);
  //pre: exists j, 0<=j<cellname.size()-1,
  //        cellname.substr(0,j) is all alphabetic (except for a
  //             possible cellname[0]=='$')
  //        && cellname.substr(j) is all numeric (except for a
  //             possible cellname[j]=='$') with at least one non-zero
  //             digit

  CellName (unsigned columnNumber = 0, unsigned rowNumber = 0,
            bool fixTheColumn = false,
            bool fixTheRow=false);

  std::string toString() const;
  // render the entire CellName as a string

  // Get components in spreadsheet notation
  std::string getColumn() const;
  int getRow() const;

  bool isRowFixed() const;
  bool isColumnFixed() const;


  // Get components as integer indices in range 0..
  int getColumnNumber() const;
  int getRowNumber() const;


  bool operator== (const CellName& r) const
    {return (columnNumber == r.columnNumber &&
             rowNumber == r.rowNumber &&
             theColIsFixed == r.theColIsFixed &&
             theRowIsFixed == r.theRowIsFixed);}

private:
  ⋮
  int rowNumber;
  bool theRowIsFixed;
  bool theColIsFixed;
```


```
  int CellName::alphaToInt (std::string columnIndicator) const;
  std::string CellName::intToAlpha (int columnIndex) const;

};


inline
bool CellName::isRowFixed() const {return theRowIsFixed;}

inline
bool CellName::isColumnFixed() const {return theColIsFixed;}



#endif
```

There are some options here the have not been explored:

- Do we want the column info stored as a number, a string, or both?

.....

Book implementation

We can implement Book in `book.h`:

```
#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
```

and in book.cpp:

```
#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;
}
```

We'll explore some of the details and alternatives of these implementations in the next lesson.

. . . . . .