Chapter Sixteen

Object-Oriented Software Development

Inheritance and Polymorphism

Summing Up

The previous chapter introduced dynamic memory allocation and a linked list. The notions of indirection and dynamic memory allocation, introduced in the preceding two chapters, will be used in this chapter to store a collection of dissimilar elements.

Coming Up

This chapter uses a team-based approach to introduce the two other major features of the object-oriented paradigm:

- inheritance: the ability to derive a new class from an existing class
- polymorphism: the ability of different types of objects to respond to the same message in different ways

A case study presents another object-oriented approach to software development. Along the way, the team discovers a class hierarchy that provides experience with inheritance, polymorphism, and heterogeneous collections. After studying this chapter, you will be able to

- recognize generalization that may be implemented with inheritance
- derive new classes from old ones
- override member functions and add new features to derived classes
- apply object-oriented design heuristics for inheritance
- understand and use polymorphism

16.1 Discovery of Inheritance through Generalization

This section provides another case study in object-oriented software development. This time the problem is from the domain of a college library system. It follows the same methodology presented with the Chapter 12 cashless jukebox case study. This problem is very similar to a problem described in Nancy Wilkinson's book *Using CRC Cards* [Wilkinson 95]. Related items such as a library class hierarchy, inheritance, polymorphism, a date class, and a data structure capable of storing different classes of objects are described in *Problem Solving and Program Implementation* [Mercer 91].

THE PROBLEM STATEMENT: *College library application*

The college library has requested a system that supports a small set of library operations: students borrowing items, returning borrowed items, and paying fees. Late fees and due dates have been established at the following rates:

	Late Fee	Borrowing Period
Book:	$0.50 per day	14 days
Videotape:	$5.00 one day late plus $1.50 for each additional day late	2 days
CD-ROM:	$2.50 per day	7 days

A student with more than seven borrowed items, any one late item, or late fees greater than $25.00 may not borrow anything new.

Object-oriented software development attempts to model a real-world system as a collection of interacting objects—each with its own set of responsibilities. This helps organize the system into workable pieces. The three-step object-oriented software development strategy introduced in Chapter 12 is repeated here for your convenience:

1. Identify classes that model (shape) the system as a natural and sensible set of abstractions.
2. Determine the purpose, or main responsibility, of each class. The responsibilities of a class are what an instance of the class must be able to do

(member functions) and what each object must know about itself (data members).

3. Determine the helper classes for each. To help complete its responsibility, a class typically delegates responsibility to one or more other objects. These are called helper classes.

The team consists of the library domain expert, Deena, and five students who are analyzing the college library system as part of a computer science honors option: Jessica, Jason, Steve, Matt, and Misty. Austen is the object expert this time.

16.1.1 Identify the Classes

The team now has some experience with object-oriented software development. They plan to use their experience. The first goal, or deliverable, is a set of potential classes that model the problem statement. Each class will describe its major responsibility. The team starts by writing down all the nouns and noun phrases in the problem statement, redundant entries not recorded.

Nouns (from the problem statement)

college library	system	library operations	CD-ROM
librarian	student	item	seven borrowed books
fee	book	videotape	
late fee	day	due date	

The following list represents the set of potential classes for modeling a solution:

Potential Classes: *A first pass at finding key abstractions*

Somewhat Sure	Not Sure
librarian	system
student	operations
book	late fees
video	due date
CD-ROM	day
seven borrowed books	fine
college library	

Matt recommends keeping `librarian` as the name for a class responsible for coordinating the activities of checking books in and out.

A `student` class will be a useful key abstraction. After all, students will be checking books out, checking them in, and paying fines. The domain expert, Deena, mentions that `librarian` should also be allowed to lend books to faculty, staff, and other members of the community. After all, it's a state university paid for in part by state taxes. So the team decides to change the class name to `borrower` to reflect the general notion of someone who can borrow a book from the library.

Misty doesn't believe "seven borrowed books" should be a key abstraction. One of the programmers on the team chimes in and suggests, "This is not a problem. I can see this as a `bag` of `book` objects." The domain expert Deena remarks, "A book bag?—cool." Since the team is currently looking for classes that help observers understand what the system does rather then how the system will eventually do it, the team decides that "seven borrowed books" is not a class. Instead, this is a knowledge responsibility.

Austen uses some object-speak: "A `borrower` should know its own collection of borrowed books." Matt jumps in and asks Austen, "Aren't CD-ROMs and videotapes in the same category? They can be borrowed too!" Deena agrees. Austen states that Matt has implicitly discovered a basic concept of object-oriented analysis. Some classes have things in common. In this case, there are several categories of things that can be borrowed: books, CD-ROMs, and videotapes. The team considers the responsibilities these classes have in common. After some discussion, Jason and Jessica produce the following list of responsibilities that all three classes of objects have in common. Each `book`, `cdRom`, and `video` should:

- know its due date
- compute its due date
- determine its late fee
- know its borrower
- check itself out
- check itself in

Self-Check

16-1 Which responsibilities are the same for these three classes?

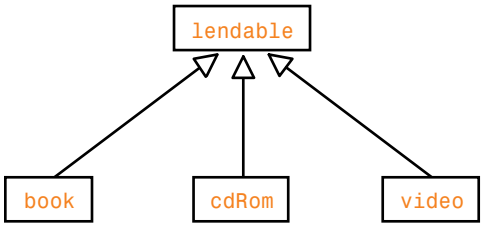
16-2 Which responsibilities should be carried out differently?

Jason notices that the term "borrowable item" has been floating around. Austen suggests there could be an abstraction named `borrowableItem` that supplies the

attributes and behavior common to all items that could be borrowed from the library. Although differences exist in the computation of late fees and the setting of due dates, each `borrowableItem` has several common responsibilities. Steve complains that `borrowableItem` is a bit of a tongue-twister. Matt suggests `lendable`: "`lendable` represents things that are, well, lendable." Deena likes this new name. `lendable` it is.

Austen points out that the team has intuitively discovered the inheritance relationship between classes. *Inheritance* is another name for generalization. Matt instinctively saw several seemingly different objects and found some common things. He generalized. Since no one has seen or heard of the inheritance relationship, Austen draws the following diagram on the chalkboard:

Figure 16.1. *The inheritance relationship in UML notation (`lendable` is the abstract class)*



Austen explains that the C++ culture would refer to `lendable` as the *base class*. The other three classes—`book`, `cdRom`, and `video`—are known as *derived classes*. The member functions and data members common to all lendables should be listed in the base class (`lendable`). Through inheritance, each derived class (`book`, `cdRom`, and `video`) inherits member functions from the base class (`lendable`). Austen then displays a design heuristic in order to explicitly encourage the team to accept the inheritance relationship between classes as good design.

Object-Oriented Design Heuristic 16.1 (Riel's 5.10)

> If two or more classes have common data and behavior, then those classes should inherit from a common base class that captures those data and methods.

The common data and operations include:

- a date for knowing the borrower and the due date
- operations such as check self out and check self in

However, without some difference amongst the derived classes, there is no reason to use the inheritance relationship. Instead, there should be just one class. So, in addi-

tion to commonalities as mentioned above, there must be enough differences to justify having more than one class. There are two, possibly three, major differences between the three classes:

1. computation of the due dates
2. computation of late fees
3. different attributes (`video` has a movie-studio attribute, for example)

Jason is confused and questions Austen. Jason doesn't understand why there is a `lendable` class in the library system. Austen decides it might prove useful if he explained the difference between an abstract class and a concrete class. An *abstract class* describes the data and operations meant to be common to all derived classes. An abstract class cannot be instantiated. Therefore, this code should be rendered illegal:

```
lendable aLendable;  // ERROR--attempt to construct abstract class
```

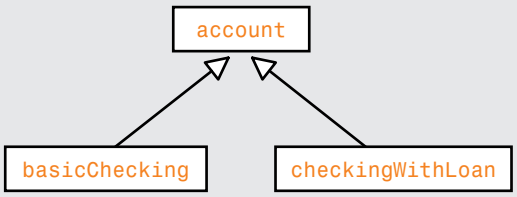
Abstract classes exist to capture common operations and data. They are not to be constructed.

A *concrete class* is one that can be instantiated. Therefore, these constructions must be legal:

```
book aBook;  // To be implemented later
cdRom aCDROM;
video aVideo;
```

Self-Check

16-3 List the abstract class in the following `account` hierarchy:



16-4 List the concrete class(es) in this inheritance hierarchy.

16-5 List an operation that would make one class different from another.

16-6 List one operation that would be the same for the concrete classes.

16-7	List one data member that both derived classes would likely have.
16-8	List one data member that would exist in one class, but not in the other.

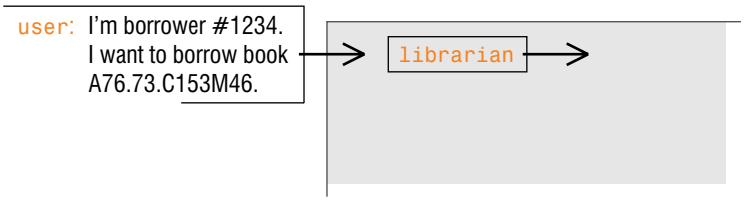
Jessica claims to understand the notion of the inheritance relationship, although the implementation is not in her head yet. Austen remarks, "I could show you how all this works, but don't worry about it for now. I'll show the implementation details in a bit." Matt articulates that the team should get back on task with analyzing the college library system.

Jason reminds the group that the following classes have been identified so far:

- `librarian`
- `borrower`
- `lendable` (and the derived classes: `book`, `cdRom`, and `video`)

With some consensus realized, Steve wants to know what the user interface will look like. Will there be text-based input and output communication between the user and the system? Is a graphical user interface desired? What about a card reader or touchscreen for input? Steve enjoyed the touchscreen ordering system recently tested at the local Taco Bell™. Misty points out that it has since been removed. Deena, the domain expert, isn't sure what the interface should look like. She believes the library will be okay with a text-based interface to the system. Text-based input and output are acceptable. Matt draws a picture so everyone can understand the relationship between a user and `librarian`:

Figure 16.2



Steve wonders whether `user` should be included as one of the classes to model the problem. Misty reminds Steve that the user is a person who approaches `librarian`, shows identification, and makes a request. `borrower` represents an object inside the system that knows everything about that user that the system will need to know. For example, `borrower` could also be responsible for knowing if its human

counterpart (the user) owes any late fees. User is not going to be a class. However, it could play a part in the scenarios. Austen confirms that this is a good decision. Matt lets Austen in on a little secret: "We went through this once before with the cashless jukebox that's jammin' in the student center."

Deena has a problem with the concept of a `book` class and a `borrower` class. Certainly the system must maintain many borrowers and many books. Matt relates, "One class can create many instances, or objects. Since there will be many lendables and many borrowers, we should probably add two new classes named `borrowerList` and `lendableList` to store, retrieve, and delete `borrowers` and `lendables`, respectively." The team agrees. It worked for the jukebox.

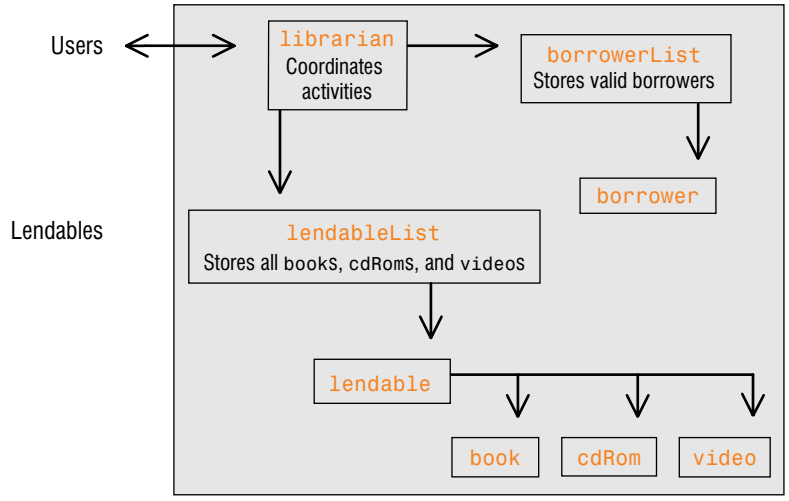
The team feels as though they have captured several key abstractions (classes) for this application. They now have a framework for analyzing the problem in more detail. There is a sense that the primary responsibility of each class has been recognized. The team documents their progress with a table that lists class names along with their primary responsibilities.

Class Name	Primary Responsibility
`librarian`	Represent object responsible for coordinating activities of checking books in and out.
`borrower`	Represent one instance of someone who can borrow a `lendable`. There may be thousands of `borrowers`.
`lendable`	Represent an abstract class from which many "borrowable" items can be derived. This abstract class captures the common member functions and data members of any item that can be borrowed from the college library.
`book`	Represent one book that can be checked in and checked out. There may be thousands of `books`.
`cdRom`	Represent one CD-ROM that can be checked in and checked out. There may be thousands of `cdRoms` stored in the database.
`video`	Represent one videotape that can be checked in and checked out. There may be thousands of `videos`.
`borrowerList`	Retrieve, delete, add, or update any `borrower` from the thousands of `borrowers` in the database of valid `borrowers`.
`lendableList`	Retrieve, delete, add, or update any `lendable` from thousands of "borrowable" items.

This following picture provides an abstract view of the system so far. It also marks the boundaries of the system. Everything in gray is in the system under

development. The users and physical items that can be borrowed are outside the system.

Figure 16.3



16.2 Refinement of Responsibilities

With primary responsibilities identified, the team now sets about the task of identifying and refining other responsibilities. The team will also identify helpers—the other classes needed to carry out a specific responsibility. The team will try to answer questions such as: What are the responsibilities? Which class should take on each of the responsibilities? and What class(es) help accomplish another class's responsibility? Responsibilities convey the purpose of a class and its role in the system. Analysis is enhanced when the team thinks that each instance of a class will be responsible for two things:

1. the knowledge each object of the class must maintain
2. the actions each object of the class can perform

Austen recommends that the team to be prepared to ask the following two questions:

1. What should an object of the class know about itself (knowledge)?
2. What should an object of the class be able to do (actions)?

Assigning a responsibility to a class means that every instance of that class will have that responsibility. This is true when there are many instances of the class—as in `lendables` and `borrowers`. It is also true when there is only one instance of the class—such as `librarian` and `lendableList`.

The next team activity involves assigning more specific responsibilities to the classes already identified. One technique involves role playing. Each team member assumes the role of one of the classes. The team members play out scenarios while adding responsibilities and helpers to CRH cards.

16.2.1 A "User Cannot Borrow" Scenario

At this point, team members assume the roles of the classes. They plan to try scenarios to see how one instance of a class interacts with other objects. A scenario is the answer to the question What happens when . . . ? The team decides the first scenario will be the response to this question:

16.2.1.1 Scenario 1: "What happens when user #1234 wants to check out a book and currently has seven borrowed `lendables`?"

`librarian`: Well, I'm the librarian so I guess I'll start. I just got a user ID. It's #1234. Now User, what do you want to do?

User: I want to check out a book.

`librarian`: Now I need to know the call number of the book.

User: The book's call number is QA76.1.

`librarian`: Okay, now let me verify that this user can borrow. I'll ask `borrowerList` to look up the `borrower` with ID #1234.

`borrowerList`: I found the `borrower` you asked for. I'm sending it back to you. I'll add `getBorrower` to my responsibility list. I must know all borrowers.

`librarian`: Thanks, `borrowerList`. Now I have the software object that represents the human user. I believe my job will be easier if I can send `borrower` a `canBorrow` message. I am now helping `borrower`. What about it, current `borrower`, can you borrow a new book?

`borrower`: Since I am responsible for knowing my borrowed `lendables`, I should be able to tell you if I have seven or more things checked out. Hey wait a minute, you didn't ask me if I had seven borrowed items. You asked me if I could borrow. So I am going to add a `canBorrow` responsibility to my list. I will return a code indicating that I can borrow. Will false or true do? To borrow, my late fine must

be $25.00 or less and I must have fewer than seven borrowed items, none of which can be overdue. It seems like I'll need to know my own borrowed `lendables`. I have seven borrowed items. No, I cannot borrow.

`librarian`: Thanks, `borrower`. Things are made easier for me because I can delegate the `canBorrow` responsibility to you. Also, remember when we were working on the jukebox. Chelsea told us that we should try to distribute system intelligence as evenly as possible. We had a `canSelect` message to simplify the jukebox. Now, I can just send a `canBorrow` message to you. I think I now must send an appropriate message to the user that borrowing a book is not an option.

The team decides they have run this particular scenario to its logical conclusion. They also feel that there are obviously many possible scenarios to role play such as successfully checking out a book and returning a lendable. The team also wonders if users should be able to look up a book by call number to see if it is in or out. Perhaps a user might want to know when a book is due back in the library. This suggests that a `lendable` should know when it is due and it should be able to tell someone the actual due date.

Deena confirms that this is certainly desirable behavior. However, the problem specification does not list these requirements. The team agrees to plan for the possibility of adding such enhancements later.

16.2.2 A Check-Out Scenario

16.2.2.1 Scenario 2: "What happens when user #1234 wants to borrow a `lendable` with call number QA76.2, has three books out, none of which are late, and has late fines of only $5.00?"

`librarian`: I'll start again; I'll get the user ID.

User: My ID is #1234.

`librarian`: Let me ask `borrowerList` for the proper borrower. Please `getBorrower` with ID #1234.

`borrowerList`: I found the `borrower` you asked for. I'm sending it back to you.

`librarian`: User, what do you want to do?

User: I want to borrow something with call number QA76.2.

`librarian`: Now I have the current `borrower` and the current `lendable` in my possession. I think I'll check with `borrower`: can you borrow?

borrower: Let me see if I have fewer than seven borrowed items. Yes, I currently am borrowing three lendables. Now, are any lendables overdue? Let me ask the first lendable in my list of borrowed items: lendable_0, are you overdue?

lendable_0: I am responsible for knowing if I am overdue. I'll have to ask date to compare my due date with today's date. I will add the responsibilities "know due date" and isOverdue. I asked date for help.

"Who's date?" asks Deena. She wants to know what a date class would be responsible for. The team decides to add date as a key abstraction and see what its responsibilities are. One of the programmers writes down date as a helper on a CRH card and agrees to play the role of date.

date: To compare the due date with today's date, I must be able to get today's date and to compare two dates. Yes lendable_0, today's date is less than or equal to the due date. I'll add <= and todaysDate to my responsibilities.

lendable_0: Thanks, date. I am not overdue.

borrower: I'll check my other lendables. lendable_1, lendable_2? None are overdue. Since I also have no late fees, I can tell librarian, "Yes, I can borrow something new."

librarian: lendableList, please get me the lendable QA76.2.

lendableList: Okay, here is the lendable.

librarian: What should I do now?

The team pauses and considers a couple of possibilities. It seems as if the current borrower and the current lendable both need to be updated somehow to record that the lendable has been checked out. It seems logical to update the lendable first and then send it to the borrower to add to its list of lendables. It also seems appropriate to update lendableList and borrowerList. That ensures that all borrowers and lendables are accurately updated.

Austen recommends that the team first ask the question, What should be done to update lendable? The team decides that the book's status should become "not available." Also, lendable's dueDate should be set to the appropriate day in the future so later on the borrower can ask the book if it is overdue. The team also believes that it is important to know who has borrowed the lendable—in case someone wants to find out who has it. The team comes up with two alternatives.

The first alternative places the responsibility of updating a `lendable` upon `librarian`. The second alternative delegates this responsibility to the `lendable` itself. The team decides to role play both alternatives. Here is the first.

16.2.2.2 Alternative 1 (updating the `lendable`)

`librarian`: `lendable`, compute and set your due date.

`lendable`: Okay, I'll `setDueDate` to either 2, 7, or 14 days from today—depending on what class of `lendable` is currently being checked out.

Austen breaks in saying, "That's polymorphism!" The class can be determined while the program is running. The particular version of `setDueDate` will depend on the class of the object. And the class of object cannot be determined until the moment the `lendable` is being checked out—at runtime.

`lendable`: I'll set the due date. I'll add a `computeDueDate` responsibility to my CRH card.

`librarian`: `lendable`, now please record #1234 as your borrower.

`lendable`: Okay, I'll set my borrower ID as user #1234.

`librarian`: Okay, now mark yourself as not available.

`lendable`: Done.

16.2.2.3 Alternative 2 (updating the `lendable`)

`librarian`: `lendable`, you could be responsible for checking yourself out. If I tell you who the `borrower` is, could you check yourself out?

`lendable`: Sure. I add these new responsibilities to my CRH card: `computeDueDate` and `checkSelfOut`.

Which alternative is better? One way to assess the design is to ask yourself what feels better. Alternative 2 feels better somehow. But wouldn't it be nice to have some design heuristics to make us feel better about feeling better? Well, it turns out that the first alternative has a higher degree of coupling. There were three different message sends versus the single message send of the second alternative. Additionally, the second alternative has better cohesion—the three responsibilities of `lendable` accomplished by a `lendable::checkSelfOut` message are closely related.

1. Know my `borrower`.
2. Compute my due date and set my due date.
3. Update my availability status.

Additionally, the first alternative requires `librarian` to know more than is necessary about the internal state of the `lendable`. Alternative 2 delegates responsibility to the more appropriate class. So the team member holding the `lendable` card adds `lendable::checkSelfOut(borrower)` to the set of responsibilities on the CRH card.

Now, has this scenario reached its logical conclusion? No. The `borrower` does not know about its new borrowed `book`. Remember that it is the responsibility of each `borrower` to know its borrowed `books`. The `borrower` must be updated. Here is one conclusion to this scenario.

`librarian`: `borrower`, let's follow the design heuristic we just talked about. I'll just send this message: `borrower.checkOut(lendable)`

`borrower`: It seems as though I should be able to add a `lendable` to my list of borrowed `lendables`. I'll add `borrower::checkOut(lendable)` as a responsibility on my CRH card.

`librarian`: Please inform the user that everything is okay—the user may take the `lendable` along. Whoops, I almost forgot. I better update `borrowerList` and `lendableList`. `borrowerList`, please put this `borrower` away.

`borrowerList`: Okay, I'll add `putBorrower(borrower)` to my CRH card.

`librarian`: `lendableList`, please but this `lendable` away.

`lendableList`: Okay, I'll add `lendableList::putLendable(lendable)` to my CRH card.

`librarian`: So User, anything else?

User: No, I'm outta here.

`librarian`: Okay, I'm ready to process another user.

The check-out scenario has reached a logical conclusion.

Self-Check

16-9 Write a check-out algorithm that sends any message you desire to any object you desire. Use any of the objects shown next as if the classes were already implemented. Add any message you like.

Remember you are designing now. You will be passing off your CRH cards and the check-out algorithm to another programmer who will have to make it all work according to your design.

```
borrowerList theBorrowerList;
lendableList theLendableList;
borrower currentBorrower;
lendable currentLendable;
```

16.2.3 A Check-In Scenario

The team knows that there are many scenarios that should be played out. Deena wants to know:

16.2.3.1 Scenario 3: "What happens when a user returns a book that is not overdue?"

`librarian`: I'll start again; I'll get the user ID.

User: My ID is #1234.

`librarian`: Let me ask `borrowerList` for the proper `borrower`. Please `getBorrower` with ID #1234.

`borrowerList`: I found the `borrower` you asked for. I'm sending it back to you.

`librarian`: User, what do you want to do?

User: I want to return something with call number QA76.2.

`librarian`: User #1234 wants to return a `lendable` with call number QA76.2. Seems like I need `checkIn` on my CRH card. Before I just thought I had to check things out. Now I know I must do both. I'll make sure I have both `checkIn` and `checkOut` on my CRH card. Let me get the current state of this `borrower` and `lendable` from their respective lists. First, get me the `borrower` with ID #1234.

`borrowerList`: Okay, here is the `borrower`.

`librarian`: `lendableList.getLendable(QA76.2)`. Wait, maybe I'll just get it from the `borrower`. No, in fact, I want my life to be easy. `borrower`, please `checkIn(QA76.2)`.

`borrower`: Okay, let's see if I can do that. I am currently borrowing that `lendable`. So `lendable`, are you overdue?

`lendable`: No.

`borrower`: Okay, I'll just remove you from my borrowed list. If you were overdue, I'd probably have to adjust my late fees. I've been updated. Back to you, `librarian`.

`librarian`: I still need to update `lendable` and `lendableList`, but first, let me put you (the `borrower`) back where you belong so the next time you try to borrow something, I will get back your current state.

`borrowerList`: Okay, I think I can handle that, but I will need to know the `borrower` you are putting back. My major responsibilities are to know all borrowers and to allow `librarian` to get `borrowers` and put updated `borrowers` back. I'll just replace the current state of the `borrower` with the updated `borrower` you send me. I need this written down so I can see it better and remember it.

Matt designs a first draft of a `borrowerList` class definition:

CRH Card

Class: `borrowerList`	**Helpers:**
Responsibilities:	`borrower`
know all borrowers	`vector`
`getBorrower(borrower)`	
`putBorrower(borrower)`	

Class Definition, First Draft

```
class borrowerList {
public:
  void getBorrower(borrower);
  void putBorrower(borrower);
private:
  vector <borrower> my_data;
  int my_size;
};
```

`librarian`: Now that the `borrower` is taken care of, I have to update `lendable` and `lendableList` to reflect the fact that the physical equivalent (a book) has been returned. So `lendable`, check yourself out.

`lendable`: I should be able to do that. Okay, I'll mark myself as available. I wonder, is there anything else I should do? Perhaps I could set my due date to today or perhaps sometime way in the past. What about 1-1-1900? No, that would make me a century overdue. What about a day in the future, say 9-9-9999. Or I could set my `borrower` to something like "?no borrower?" Let me think about it; perhaps being available is enough. No one should care about `borrower` or due date if I am available. Anyway, consider me updated.

`librarian`: Now I just need to put the book away. I send this message: `lendableList.putLendable(currentLendable);`

`lendableList`: Okay, I think I can handle that. I do need to know the `lendable`. My major responsibilities are to know all `lendables` and to allow `librarian` to get `lendables` and put them back. I'll just replace the current `lendable` state with the updated `lendable` you send me. I need this written down also.

Matt designs a first draft for the `lendableList` class definition:

CRH Card

Class: `lendableList`	**Helpers:**
Responsibilities:	`lendable`
know all `lendables`	`vector`
`getLendable(lendableID)`	
`putLendable(lendable)`	

Class Definition, First Draft

```
class lendableList {
public:
  void getLendable(string lendID);
  void putLendable(lendable);
private:
  vector<lendable> my_data;
  int my_size;
};
```

`librarian`: Everything is cool.

This scenario has reached its logical conclusion. Along the way, `librarian` also added a few major responsibilities: `checkIn` and `checkOut`. Austen claims these two scenarios might be implemented as private member functions. The team member role playing `librarian` feels it is time to review her CRH card. Once again, Matt designs a first draft of a `librarian` class definition to better visualize the class.

CRH Card

Class: `librarian`	**Helpers:**
Responsibilities:	`lendable`
coordinate activities	`borrower`
know current `borrower` and	`lendableList`
`lendable`	`borrowerList`
`checkIn`	
`checkOut`	

Class Definition, First Draft

```
class librarian {
public:
  void processOneUser();
private:
  borrower currentBorrower;
  lendable currentLendable;
  void checkOut(lendable);
  void checkIn(lendable);
};
```

Jessica and Jason are anxious to see what happens when a user returns an overdue `lendable`.

16.2.4 A "Return of an Overdue Book" Scenario

16.2.4.1 Scenario 4: "What happens when a user returns a book that is five days overdue?"

`librarian`: User #1234 wants to return a `lendable` with call number QA76.2. So I'll execute my `checkIn` algorithm. The first thing I want to know is if the `lendable` is overdue. Who can answer that?

The team debates whether `librarian` should get the software version of the borrowed item from the `borrower` or from `lendableList`. The `lendable` should be in the same state in either location—the `lendableList` or the `borrower`'s set of borrowed `lendables`. Jessica, who is role playing `borrower`, says, "I know what I have borrowed and since I may need to register a late fee, why not just ask me?" `librarian` would rather do what she did before: "I don't want to get confused. I'll get `currentLendable` from `lendableList` and `currentBorrower` from `borrowerList`. Then I can do whatever I need to. When I'm done, I'll put them both away."

`librarian`: `lendableList.getLendable(QA76.2)`.

`lendableList`: Here is that `lendable` you seek.

`librarian`: `borrowerList.getBorrower(#1234)`.

`borrowerList`: Here is that `borrower` along with all known borrowed items.

`librarian`: `borrower`, `checkIn (QA76.2)`.

`borrower`: Okay, I do have a record that I am borrowing the `lendable` with ID QA76.2. So `lendable`, are you overdue?

`lendable`: I'll ask `date`. Is `dueDate` < today's date?

`date`: Yes. The `dueDate` was some time ago. I guess there will be money to owe.

`lendable`: That's real clever rhyming, `date`. Yes `borrower`, I am overdue.

`borrower`: I'll ask `lendable` to `computeLateFee`.

`lendable`: `date`, how many days overdue? Tell me `todaysDate - dueDate`.

`date`: There is a difference of five days. I'll add "compute number of days between two dates" to my CRH card.

lendable: The late fee is 5 * (per_day_late_fee), which as a book is $2.50.

borrower: Okay, I'm supposed to know my total late fee, so I'll add that $2.50 to my late fee. Remember, I'll be implemented as software to ensure my late fees are always honest. I'll write recordLateFee on my CRH card.

"That was an awful lot of action going on to return a book," moans Steve. "I think we could simplify things by sending messages like this":

```
currentBorrower.checkIn(currentLendable);
```

Austen explains that this can and actually did happen. It's just that we observed the details from the perspective of borrower, lendable, and date. All of these details could be encapsulated into the checkIn algorithm. librarian simply sends a checkIn message to the borrower. The borrower gets help from lendable and lendable gets help from date. The borrower can do everything it needs to do to update itself. This design means less coupling (fewer message sends from librarian to borrower). "A better design," remarks Austen.

borrower: So in summary, when I receive a checkOut message, I'll adjust my late fees if necessary; I'll also remove the lendable from my list of borrowed items. So I will add checkIn to my CRH card. Might as well add checkOut also. And in case you ever want to know what my late fees are, I'll make them accessible with a lateFee accessor member function. I'm going to follow the trend and write all this down before I forget. Matt, please design a class definition for me while you're at it.

CRH Card

Class: borrower	**Helpers:**
Responsibilities:	lendable
know borrowed lendables	bag or vector?
recordLateFee	librarian
canBorrow	
checkIn(lendable)	
checkOut(lendable)	
double lateFee()	

Class Definition, First Draft

```
class borrower {
public:
  checkOut(lendable aLendable);
  checkIn(lendable aLendable);
  double lateFee() const;
  bool canBorrow() const;
private:
  double my_fines;
  vector <lendable> my_borrowedItems;
  int my_numberOfBorrowedItems;
};
```

librarian: Okay, now that the borrower is updated, put it away:

```
borrowerList.putBorrower(currentBorrower)
```

borrowerList: Done. I replaced the old state of the borrower with the updated version you just sent me.

librarian: Now I have to deal with the lendable. Say lendable, could you do a checkOut also?

lendable: Let me see, I'll mark myself as available, set my due date to a special date way in the future, and then set my borrower to some special value also. I'll call this algorithm lendable::checkSelfOut to distinguish it from borrower::checkOut. I have a lot to remember. Let me review my CRH card too.

Matt says, "Sure, I'll sketch a design of the class definition."

CRH Card

Class: lendable		**Helpers:**
Responsibilities:		date
know due date		
know borrower		
computeDueDate	isOverdue	
computeLateFee	isAvailable	
checkSelfIn	setDueDate	
checkSelfOut(borrower)		

Class Definition, First Draft

```
class lendable {
public:
  void checkSelfIn();
  void checkSelfOut(borrower);
  bool isOverdue() const;
  bool isAvailable() const;

private:
  date my_dueDate;
  string my_borrowersID;
};
```

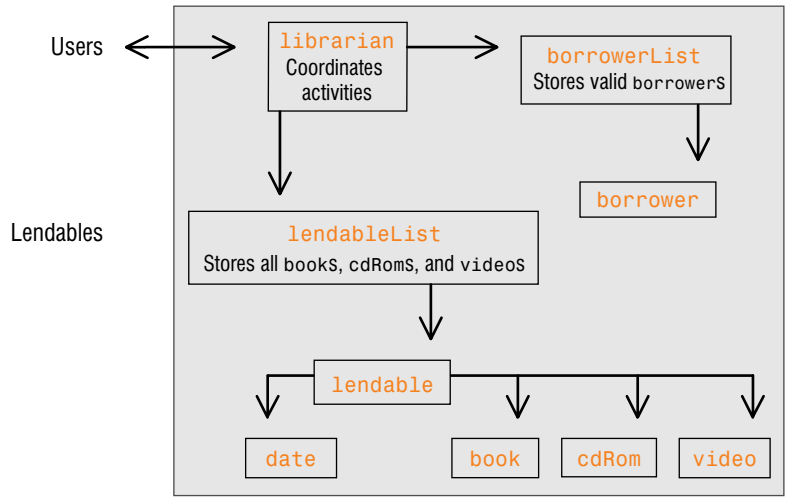
librarian: lendableList, please put away this updated lendable.

lendableList: I'll replace my current but outdated version of lendable with the updated version you just sent me.

librarian: We're done. I can now get another user request or get the next user.

Austen congratulates the team. So far, there is a reasonable set of classes with clearly defined responsibilities. Austen remarks, "The design feels right. Whether you knew it or not, you were actually using many of the object-oriented design heuristics. Even if you weren't thinking of them. Also notice that the first selection of classes held up."

Figure 16.4



There are a few new lines of helpers. `librarian` gets help not only from two lists but now also from individual instances of `borrower` and `lendable`—`borrower::checkIn` and `lendable::checkSelfIn`, for example.

The new class is `date`. In some ways, `date` is like classes such as `string` and `vector`. It could be considered a utility like `string`, `double`, `bool`, `int`, and `vector`—ready to serve. The picture above shows only domain-specific classes such as `lendable`, `librarian`, and `lendableList`. So why is `date` there? First, the `date` class played a role in helping `lendable`. `date` is present because the team will certainly have to do something about a `date` class during design and implementation. Here is the current state of `date`'s CRH card to remind the team of some of the things a `date` object should be able to do. Matt says he will not design a class definition for `date` until he checks with his teacher who told him he knew about a `Date` class on the Internet that is available for free from a gracious computer science professor.

Class: `date`	**Helpers:**
Responsibilities:	
compute number of days between two dates	
less than or equal <=	
`todaysDate`	

Self-Check

With a team, run the following additional scenarios:

16-10 What happens when a user enters an ID number that is not found in `borrowerList`?

16-11 What happens when a user wants to pay a late fee?

16-12 What happens when a user want to check on the availability of a `lendable`?

16-13 Write a `checkIn` algorithm that sends any message you desire to any object you desire. Use any of the objects shown next as if the classes were already implemented. Add any message you like.

Remember you are designing now. You will be passing off your CRH cards and the `checkIn` algorithm to another programmer who will have to make it all work according to your design.

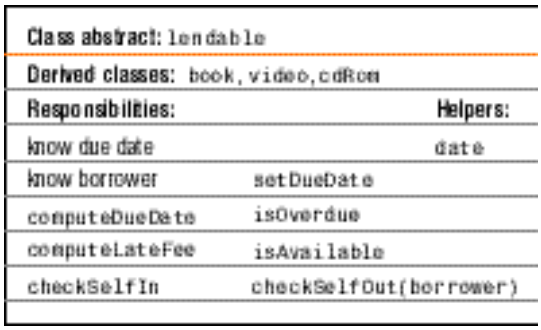
```
borrowerList theBorrowerList;
lendableList theLendableList;
borrower currentBorrower;
lendable currentLendable;
```

16.3 Design

Now that there is some understanding of the system, the programming team—Charlie and Matt—turns its focus to designing the class definitions. This programming team already knows there is a relationship between the responsibilities on the CRH cards and C++ class definitions. The things that each instance of a class must do could be

listed as the public member functions. The things that each instance of the class must know can become the private data members.

Austen suggests that the team first cope with the new class relationship of inheritance. The first thing to do is ensure that the `lendable` class captures all the knowledge and action responsibilities that will be common to the derived classes. The CRH card ends up looking like this with an additional mention of the derived classes that make up the current inheritance hierarchy:



Austen tells the team, "In a short time, you will see action responsibilities inherited by the derived classes (`book`, `video`, and `cdRom`) that have the same name, but execute differently." For example, each class will `computeDueDate`, but each class will do it differently. A book can be borrowed for 14 days but a video for only 2. At the same time, the base class can define and implement member functions that all derived classes will usefully inherit—`checkSelfIn`, for example. That behavior is the same for all derived classes.

16.3.1 The `lendable` Action Responsibilities (member functions)

The `lendable` card currently lists the names of seven possible messages. `isOverdue` and `isAvailable` seem to be the easiest to consider. Neither requires arguments. Both return either true or false indicating the state of the `lendable`. The check-out operation needs to know the `borrower`, so `checkSelfOut` requires the borrower's ID, which could be a `string`.

The team next considers the `checkSelfIn` responsibility. Matt believes the function needs no arguments. It is a `void` function. `librarian` simply sends a `checkSelfIn` message and `book` or `video` will update itself. This leads to a refined design of the base `lendable`.

```
class lendable {  // Second draft
public:
  // . . .
  bool isOverdue() const;
  bool isAvailable() const;
  void checkSelfIn();
  void checkSelfOut(string borrowersID);

private:
  // TBA
};
```

The first four member functions are common to all derived classes and do not vary between those derived classes. However, the two other responsibilities—`computeDueDate` and `computeLateFee`—have different meanings to the derived classes. They will be implemented differently for each of the three classes. For example, `book` will compute due date differently than `video`. The late fees also vary amongst all three classes to be derived from `lendable`.

Misty wonders if these two operations should be part of the interface. Austen proposes that it can be decided by answering the question, Who will send the `computeLateFee` and `computeDueDate` messages? Charlie claims the `computeLateFee` message might be sent from within the `checkSelfIn` message. To do that, the object must also `getDueDate`. Neither function is part of the public interface. Instead they are part of the hidden details. And the actual code to do each of these differs amongst the derived classes.

Austen points out that the public interface should be kept as simple as possible. There is even a design heuristic for this.

OBJECT-ORIENTED DESIGN HEURISTIC 16.2 (RIEL'S 2.5)

Do not put implementation details into the public interface of a class.

No one outside of the `lendable` hierarchy has to send a `computeLateFee` or `computeDueDate` message. And it seems safe to assume that no one will send a `computeDueDate` or `computeLateFee` message. One of the objects might want to ask a `lendable` for its due date or late fee, but probably no one will directly ask `lendable` to compute them. These are implementation details. Therefore these two operations should not be made public.

"Make them private," proclaims Matt. "Better to make them protected," claims Austen. He goes on to explain that derived classes inherit both public and protected members. However, protected members are invisible to users. The protected-access

mode allows access only to member functions of the base class and to any class derived from the base class. These protected members will not be accessible to `librarian` or anyone else outside the `lendable` hierarchy. Also, because they do not change anything (`checkSelfOut` changes the state, not `computeDueDate`), they are declared const. The team now recognizes that two other accessors should have the `const` tag.

```
class lendable {  // Third draft
public:
  // . . .
  void checkSelfIn();
  void checkSelfOut(string borrowersID);
  bool isOverdue() const;
  bool isAvailable() const;

protected:
  date computeDueDate() const;
  double computeLateFee() const;

private:
  // TBA
};
```

Austen points out that these messages will be called from the base class's `checkSelfIn` and `checkSelfOut` messages. `checkSelfOut` will send a `computeDueDate` message. `checkSelfIn` will send a `computeLateFee` message. Jessica wonders, "But how will `lendable::checkSelfIn` know which of the three `computeDueDate` messages to send?" Matt recommends that the team consider applying multiple selection.

EXPLICIT CASE ANALYSIS

```
if (lendableIsBook())
    book::computeDueDate()
else if (lendableIsVideo())
     video::computeDueDate()
else
     cdRom::computeDueDate()
```

Austen explains that the inheritance relationship is an attempt to avoid such explicit multiple selection. Otherwise at some later point, when more `lendables` are added, the `checkSelfOut` and `checkSelfIn` member functions will have to be changed. Additionally, each derived class would have to carry around some data member value indicating the class.

OBJECT-ORIENTED DESIGN HEURISTIC 16.3

Don't add a data member that indicates a class's type. If you have to make a decision based on different classes of objects, implement an inheritance hierarchy.

This redundancy is not necessary. Each class knows what type it is. It will be easier to add a few `lendables` later by planning for them now. "How is this possible?" asks Matt. "Through polymorphism," replies Austen.

16.3.2 Polymorphism

Polymorphism makes it possible to have a collection of heterogeneous objects (e.g., `video`, `book`, and `cdRom`) that appears to be a container of homogenous objects (e.g., a lists of `lendables`). This works because the program can distinguish the class of objects at runtime. This means a `vector` could store `books`, `videos`, `cdRoms`, and, as it turns out, any other class of object derived from the same base class—`lendable`.

Polymorphism allows the same message to be sent to every object in a container, even though those objects are instances of different classes. However, the same message will activate different member functions. For example, if the current object is a book, `book::computeLateFee` will be called. However, if the current object is a CD-ROM, `cdRom::computeLateFee` will be called. So imagine a container (`vector` or `list`) that has two `books`, followed by a `video`, followed by a `cdRom`, followed by another `book`.

Message	Class	Days Overdue	Message Returns
`item[0].computeLateFee()`	`book`	2	2 * 0.50
`item[1].computeLateFee()`	`book`	2	2 * 0.50
`item[2].computeLateFee()`	`video`	2	5.00 + 1 * 1.50
`item[3].computeLateFee()`	`cdRom`	2	2 * 2.50
`item[4].computeLateFee()`	`book`	2	2 * 0.50

These five messages call three distinct functions. When the container item is a `book`, `book::computeLateFee` applies the $0.50 per day fine. When the `lendable` is a `video`, `video::computeLateFee` applies the $5.00 first late day fee plus 1 * the $1.50 additional late day fee. When the container item is a `cdRom`, `cdRom::computeLateFee` applies the $2.50 per day fine.

Matt remarks to Austen, "This is what you meant when you said earlier one name can have different meanings." "Yes," says Austen. "Now how do you get this to work?" questions Charlie.

First, place the common action responsibilities in the base class. That has already been done. Then identify the responsibilities that have different meanings for each of the derived classes. In this inheritance hierarchy, they are the following two:

1. `computeLateFee`
2. `computeDueDate`

These member functions should be made into *pure virtual functions*. That means that every derived class must implement that function in a manner appropriate to the particular `lendable`. It also means no programmer can ever instantiate `lendable` because it has a pure virtual function. Here is what pure virtual functions look like in the `lendable` class.

```
class lendable {  // Fourth draft
public:
  // . . .
  void checkSelfIn();
  void checkSelfOut(string borrowersID);
  bool isOverdue() const;
  bool isAvailable() const;

protected:
  virtual Date computeDueDate() const = 0;    // Implement in
  virtual double computeLateFee() const = 0;  // derived classes

private:
  // TBA
};
```

Virtual functions are necessary for implementing polymorphism. Once declared as virtual in the base class, the runtime system will search for the appropriate function of that name. Because the runtime system knows the class of object that sent the message, it can call the proper implementation.

A virtual function implies the implementation will vary amongst the derived classes. You specify a function as polymorphic by preceding it with the C++ keyword `virtual`:

```
virtual Date computeDueDate() const = 0;
virtual double computeLateFee() const = 0;
```

The strange ending `= 0` specifies the functions as pure virtual.

```
virtual Date computeDueDate() const = 0;
virtual double computeLateFee() const = 0;
```

Every derived class must implement every pure virtual function in its ancestor. Errors result when a programmer forgets to implement the function in any one of the derived classes. So declaring a function pure virtual is an antibugging technique.

Additionally, it is impossible to instantiate a class that has one or more pure virtual functions with `= 0`. This ensures that the class is abstract, rather than concrete. The keyword `virtual` signifies that the implementation will depend on whether the object is a `book`, `video`, or `cdRom`. And finally, `const` is added because these two functions are accessors.

Since `Date` just showed up again, Matt begins to explain where the `Date` class came from. It is the current return type for `computeDueDate`.

```
virtual Date computeDueDate() const = 0;
```

Austen replies that a `date` class is referred to even though it isn't implemented yet. But now we have a `Date` class as seen in Owen Astrachan's book *A Computer Science Tapestry* [Astrachan 97]. He says, "When I asked Owen if we could use his `Date` class to save ourselves a lot of time, he graciously offered to give it to us or to anyone else who wanted it. We'll come back to that class definition later. For now I can assure you that it does everything we need it to do such as compare dates, get the current date, add days to a date, subtract days from a date, and find the difference between two dates. I should also tell you that Owen, like many other computer scientists, likes to capitalize the first letter of his classes—so he named it `Date` rather than `date`."

Now let us consider the data members we'll need for all `lendables`.

16.3.3 The Knowledge Responsibilities (data members)

The team examines the `lendable` CRH card and notices `date` is often written as a helper. `computeDueDate` returns a `Date` object, but does `lendable` need a `Date` data member?

Consider that every derived class must maintain its due date. Therefore, this common knowledge responsibility should be declared in the base class. Austen informs Matt and Charlie that private data members are not inherited by derived classes. The common solution is to place a due date object in the `lendable` private section and then provide an accessor to that data member in the `protected:` access mode (this has been done). The fifth draft of the `lendable` class definition declares these newest considerations.

```
class lendable {  // Fifth draft
public:
//-- modifiers
  void checkSelfIn();
  void checkSelfOut(string borrowersID);

//--accessors
  bool isOverdue() const;
  bool isAvailable() const;
  string lendableID() const;
  Date dueDate() const;

protected:
  virtual Date computeDueDate() const = 0;    // Implement in
  virtual double computeLateFee() const = 0;  // derived classes

private:
  Date my_dueDate;
  string my_ID;
};
```

The data member `my_ID` and an accessor were added because there is a need to search for `lendables`. `my_ID` should make each `lendable` unique. And now, it appears that other information about a book's author or a CD-ROM's artist or software vendor name should be added later in the derived classes.

So now that the data members are established, the constructors can be considered. One constructor parameter can be used to initialize `my_ID`. An accessor to `my_ID` should also be added. By reviewing the `lendable` CRH card, Charlie also noticed the borrower's responsibility was still not part of the class definition. Instead of storing the entire `borrower` however, it seems as though the borrower's ID number will suffice. The availability knowledge responsibility was missing too. And finally, Matt recalled that someone asked the `lendable` for its late fee. So that too should be added as an accessor. The final version of the `lendable` class summarizes all of the above considerations.

```
class lendable {
public:
//--constructors
  lendable(string initID);

//--accessors
  bool isOverdue() const;
  bool isAvailable() const;
  string lendableID() const;
```

```
    string borrowersID() const;
    double lateFee() const;
    Date dueDate() const;

  //--modifiers
    void checkSelfIn();
    void checkSelfOut(string borrowersID);

  // Pure virtual functions to be implemented
  protected:
    virtual Date computeDueDate() const = 0;
    virtual double computeLateFee() const = 0;

  private:
    Date my_dueDate;
    string my_ID;
    bool my_availability;
    string my_borrowersID;
  };
```

Now the `lendable` member functions must be implemented, all but the pure virtual functions that is. First, a detail is added that has nothing to do with inheritance.

While testing the classes derived from `lendable`, it was found that a `book` and a `video` had a `dueDate` in the future, even after they were returned. Matt and Charlie decide to establish an "empty" value for a due date. When a `lendable` gets checked back in, the due date is set to something in the future, 9-September-9999, to be precise. Any `lendable` with this `emptyDueDate` value cannot be considered overdue. It also helps to set the `borrower` to someone other than the user who had already checked it in—an "empty" `borrower`. Here are the two global constants:

```
// Use two special values to indicate irrelevant dueDate and
// borrowersID
const Date emptyDate = Date(9,9,9999);
const string emptyID = "?";
```

16.3.3.1 The `lendable` Constructor Has an Initializer List

The following familiar pattern for constructors could be followed to implement the constructor `lendable::lendable`.

```
lendable::lendable(string initID)
{ // Less efficient, and inadequate when this is a derived class
  my_ID = initID;
  my_dueDate = emptyDate;
  my_availability = true;
```

```
  my_borrowersID = emptyID;
}
```

However, a different method of initialization must now be employed. Initialization lists, like the one you are about to see, have been avoided as unnecessary syntax details. They are introduced now because they are absolutely, positively needed to implement the derived class's constructors. Additionally, using initialization lists makes the program run faster. Here is the initialization list for `lendable` (not really necessary in a base class):

EXAMPLE OF INITIALIZATION LIST

```
lendable::lendable(string initID)
         : my_ID(initID),
           my_dueDate(emptyDate),
           my_availability(true),
           my_borrowersID(emptyID)
{
  // More efficient initialization already occurred
}
```

An *initialization list* begins immediately after the function heading with a colon (`:`) followed by each data member (initial value) pair separated by commas. The effect of both implementations of `lendable::lendable`—with four assignments or one long initialization list—is the same. Both adequately initialize all data members of `lendable`.

Although an initialization list isn't necessary to implement the base class constructor, the initialization list must be used by all derived classes. For only in an initialization list can the base class constructor be called. Observe the call to `lendable`'s constructor here in the `book` constructor:

```
book::book(string initID, string initAuthor, string initTitle)
     : lendable (initID),      // Call base class constructor
       my_author(initAuthor),  // Could have used less efficient
       my_title(initTitle)     // assignment
{
  // The initialization list took care of everything. Remember, the
  // lendable constructor was also called to initialize the data
  // members that are common to all derived classes.
}
```

The highlighted part of the initialization list calls the `lendable` constructor with one argument, which in turn, initializes all the other common data members (see `lendable::lendable` above).

16.3.3.2 The Accessors

The `lendable` accessors require no additional explanation other than perhaps the `Date` member functions that were used. Astrachan's `Date::Absolute` function returns a number that can be compared with <= and == as in the first guarded action that checks to see if the `dueDate` is equal to the `emptyDate`.

```
bool lendable::isOverdue() const
{
  if(my_dueDate.Absolute() == emptyDate.Absolute())
     return false;

  Date today;
  // assert: today stores today's date
  return my_dueDate.Absolute() <= today.Absolute();
}

bool lendable::isAvailable() const
{
  return my_availability;
}

string lendable::lendableID() const
{
  return my_ID;
}

string lendable::borrowersID() const
{
  return my_borrowersID;
}

Date lendable::dueDate() const
{
  return my_dueDate;
}

double lendable::lateFee() const
{
  return computeLateFee();
}
```

16.3.3.3 The Modifiers

The `lendable` modifiers show polymorphism in action. A `checkSelfOut` message sends a `computeDueDate` message.

```
void lendable::checkSelfOut(string borrowersID)
{
  my_dueDate = computeDueDate();
  // Polymorphism in action. At runtime, the system will know
  // which computeDueDate implementation to use.
  my_availability = false;
  my_borrowersID = borrowersID;
}
```

Since there were three `computeDueDate` functions, which one will get called?

The small `lendable` hierarchy presents three possibilities: If the `lendable` is a book, the system will send a `book::computeDueDate` message. If the `lendable` is a video, the system will send a `video::computeDueDate` message. And if the `lendable` is a CD-ROM, the system will send a `cdRom::computeDueDate` message. Each of these three classes implements its own `computeDueDate` function. Because the `lendable` class definition was designed with inheritance in mind, `computeDueDate` was declared as a pure virtual function. This forces all derived classes to implement their own `computeDueDate` member function.

A `checkSelfIn` message performs three actions. They are summarized first as discovered during the scenario and then as the algorithm for `lendable::checkSelfIn`.

16.3.3.4 From an Earlier Scenario

`lendable`: Okay, I'll mark myself as available, set my due date to a special date that is not a valid due date, and then set my borrower to some special value also.

```
void lendable::checkSelfIn()
{
  my_availability = true;
  my_borrowersID = emptyID;
  my_dueDate = emptyDate;
}
```

The algorithms for the `checkSelfIn` and `checkSelfOut` messages are the same for any derived class with one exception. During `checkSelfOut`, the `computeDueDate` message will call one of three different operations; it all depends on the sender's class.

```
my_dueDate = this->computeDueDate();
// Call either 1. book::computeDueDate
//          or 2. video::computeDueDate
//          or 3. cdRom::computeDueDate
```

If a fourth class gets added to the `lendable` hierarchy (`artWork`, for example) by inheriting from `lendable`, it too will have to implement a `computeDueDate` member

function (`artWork::computeDueDate`, for example). Let's now look at how inheritance gets done in C++.

16.4 The Derived Classes

The general form for deriving one class from another is a class definition with the base class listed after the derived class name and a colon (`:`) to indicate inheritance.

GENERAL FORM 16.1. *Defining a derived class*

```
class derived-class-name : public ancestor-class-name {
public:
   new-function-heading-1 ;
   overridden-function-heading -1 ;
   new-function-heading-2 ;
   overridden-function-heading -2 ;
private:
   additional-data-members
};
```

The colon (`:`) followed by `public` could be read as "inherits public and protected things from." The ancestor's public and protected member functions are passed on to derived classes (the descendants). The *derived-class* inherits the operations of the ancestor. Member functions and data members can be added to the derived class. Finally, the *ancestor-class* member functions can be overridden (given new meaning).

For example, the `book` class adds a constructor, `book`, and it overrides the `computeDueDate` and `computeLateFee` member functions. `book` also adds two private data members, `my_author` and `my_title`, and accessors to this data.

```
class book : public lendable {
public:
// A new constructor
   book(string initID, string initAuthor, string initTitle);

// The two virtual functions that must be implemented by all
// derived classes
   Date computeDueDate() const;
   double computeLateFee() const;
```

```
// Additional accessors
   string author();
   string title();

private:
// Additional data members
   string my_author;
   string my_title;
};
```

The `video` class also adds a constructor, overrides `computeDueDate` and `computeLateFee`, and adds one private data member.

```
class video : public lendable {
public:
// A new constructor
  video(string initID, string initTitle);

// The virtual functions to be implemented by all derived classes
  Date computeDueDate() const;
  double computeLateFee() const;

// Additional accessors
  string title();

private:
  string my_title;
};
```

16.4.1 Implementing the Derived Classes

First, the data relating to overdue `lendables` are summarized as a collection of global constant objects at the top of the `lendable.h` file.

```
const int BOOK_BORROW_DAYS = 14;
const int VIDEO_BORROW_DAYS = 2;
const int CDROM_BORROW_DAYS = 7;
const double BOOK_LATE_FEE = 0.50;
const double VIDEO_LATE_FEE = 1.50;
const double FLAT_VIDEO_LATE_FEE = 5.00;
const double CDROM_LATE_FEE = 2.50;
```

The `book` constructor also uses an initialization list.

```
book::book(string initID, string initAuthor, string initTitle)
     : lendable (initID),
       my_author(initAuthor),
```

```
        my_title(initTitle)
{
  // Initialization list took care of everything
}
```

The `book` constructor could initialize its newly added private data members. However, the initialization list is necessary because it allows the base class constructor to be called from the derived class. Calling the base class constructor guarantees that the same thing will be done for every single derived class. Besides, this is the only way to call the `lendable` constructor—in an initialization list. This call to the base class constructor:

```
: lendable (initID),
```

passes `initID` along to initialize the private data member declared in `lendable`. However, a `set_ID` member function could have avoided this. So why bother with the base class constructor and initialization list?

The real purpose for calling the base class constructor is to guarantee that whatever initialization is important to all derived classes will in fact occur. The programmer adheres to this by ensuring that all derived classes use initialization lists. The base class constructor is called via an initialization list. This guarantees that anything the base class does will be done for any and all derived classes.

In this next example, `lendable::lendable(string initID)` initializes all `lendable` objects as available (`my_availability(true)`) and sets `my_dueDate` and `my_borrowersID` to a global named constant recognized as being meaningless.

```
lendable::lendable(string initID)
          : my_ID(initID),
            my_dueDate(emptyDate),
            my_availability(true),
            my_borrowersID(emptyID)
{
  // More efficient initialization already occurred
}
```

The remaining `book` member function implementations have a few new items. First, while testing, it was discovered that nothing was ever overdue. Rather than running the program over a several week period, a conditional compilation was put in place to allow testing of due dates and late fines. The `computeDueDate` function gets called from `lendable::checkSelfOut`.

```
Date book::computeDueDate() const
{
  Date today;
#ifdef DebuggingLateFee
  return today - BOOK_BORROW_DAYS;  // Compile only when defined
#else
  return today + BOOK_BORROW_DAYS;  // Otherwise only compile this

#endif
}
```

If the test driver has this compiler directive:

```
#defines DebuggingLateFee
```

then the due date gets set to 14 days in the past. An immediate call to `lendable::dueDate` finds the `book` was due two weeks ago. A `lendable::lateFee` message returns 0.50 * 14 or $7.00 (see the test driver below).

16.4.2 Astrachan's `Date` Class

The following `Date` operations were used from Astrachan's `Date` class:

```
// a class for manipulating dates
// written 2/2/94, Owen Astrachan
// Date()                    --- construct default date (today)
// Date(int m, int d, int y) --- constructor requires three
//                               parameters:
//                               month, day, year, e.g.,
//                               Date d(4,8,1956); initializes d to
//                               represent the date April 8, 1956.
//                               Full year is required
//
// long int Absolute() --- returns absolute # of date assuming that
//                         Jan 1, 1 AD is day 1.  Has property that
//                         Absolute() % 7 = k, where k = 0 is sunday
//                         k = 1 is monday, ... k = 6 is saturday
//
// string ToString()    -- returns string version of date, e.g.,
//                      -- d.SetDate(11,23,1963); then d.ToString()
//                         returns string "November 23 1963"
// ***********************************************
//         arithmetic operators for dates
// ***********************************************
// dates support some addition and subtraction operations
// Date d(1,1,1960);        // 1960 is a leap year
// Date d2 = d + 1;         // d2 is January 2, 1960
// Date d4 = d - 1;         // d4 is December 31, 1959
```

The `Date::Absolute` function returns the number of days since 1-1-1 AD. This allows dates to be compared with <, >, <=, and so on. The following function used == to return 0.00 as the late fee when the book gets checked in:

```
double book::computeLateFee() const
{
  Date today;
  int daysLate;
  daysLate = today - dueDate();  // Call protected base member
  // daysLate will be negative unless the book is overdue
  if(daysLate > 0)
    return daysLate * BOOK_LATE_FEE;
  else
    return 0.00;
}
```

The other member functions are typical accessors.

```
string book::author()
{
 return my_author;
}

string book::title()
{
  return my_title;
}
```

The `video` class looks almost exactly the same, except a `video` adds a `my_title` data member only. If you want to see the code, visit the class definitions for `lendable`, `book`, and `video` stored in `lendable.h`. You can also see the member function implementations in `lendable.cpp`. Both are on this textbook's disk and at this textbook's Web site.

16.4.3 Testing the Derived Classes

The following program test drives `video` and `book` in a `DebuggingLateFee` mode. Notice that the `void show` function takes a reference to a `lendable` object as an argument. There is no `lendable` object passed! Remember that an abstract class cannot be constructed. So without the little `&`, this function heading would be an error:

```
void show(lendable aLendable)  // Error, can't instantiate lendable
```

This test driver passes two different classes of arguments: `aBook` and `aVideo`.

```
// File name: testlend.cpp
//
#include <iostream>
using namespace std;

#define DebuggingLateFee  // lendable.h now sets dueDate in the past
#include "lendable"  // For the lendable, book, and video classes
#include "compfun"   // For decimals(cout, 2)

void show(const lendable & aLendable)
{
  cout << "The lendable " << aLendable.lendableID();
  if(aLendable.isOverdue())
    cout << " is overdue. ";
  else
    cout << " is not overdue. ";

  cout << "Late fee = $" << aLendable.lateFee() << endl;

  cout << "Due date: " << aLendable.dueDate().ToString() << endl;

  if(aLendable.isAvailable())
    cout << "It is available. " << endl;
  else
    cout << "It is not available. " << aLendable.borrowersID()
         << " has it." << endl;

   cout << "-------------------------------------------" << endl;
}

int main()
{ // Test drive video and book

  decimals(cout, 2);  // To show late fees nicely

  cout << "TEST BOOK: " << endl;
  book aBook("QA76.1M46", "Rick Mercer", "Computing Fun.");
  show(aBook);

  aBook.checkSelfOut("555-55-5555");
  show(aBook);

  aBook.checkSelfIn();
  show(aBook);

  cout << "\nTEST VIDEO: " << endl;
  video aVideo("MGM10023", "Spartacus");
  show(aVideo);
```

```
    aVideo.checkSelfOut("555-55-5555");
    show(aVideo);

    aVideo.checkSelfIn();
    show(aVideo);

    return 0;
}
```

OUTPUT (WITH `DebuggingLateFee` DEFINED; WITHOUT IT, NOTHING IS OVERDUE, FINES ARE 0)

```
TEST BOOK:
The lendable QA76.1M46 is not overdue. Late fee = $0.00
Due date: September 09 9999
It is available.
-------------------------------------------
The lendable QA76.1M46 is overdue. Late fee = $7.00
Due date: April 27 1998
It is not available. 555-55-5555 has it.
-------------------------------------------
The lendable QA76.1M46 is not overdue. Late fee = $0.00
Due date: September 09 9999
It is available.
-------------------------------------------

TEST VIDEO:
The lendable MGM10023 is not overdue. Late fee = $0.00
Due date: September 09 9999
It is available.
-------------------------------------------
The lendable MGM10023 is overdue. Late fee = $6.50
Due date: May 09 1998
It is not available. 555-55-5555 has it.
-------------------------------------------
The lendable MGM10023 is not overdue. Late fee = $0.00
Due date: September 09 9999
It is available.
-------------------------------------------
```

16.5 `lendableList`: A Heterogeneous Container

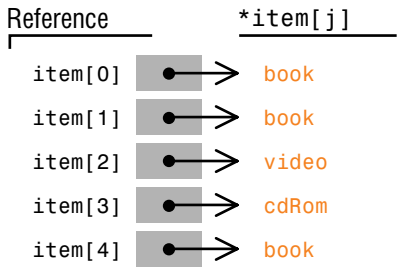
The power of inheritance and polymorphism comes in very handy when you need to store a collection of objects that are not of the same type. Consider the `lendableList` class. Its knowledge responsibility is to know all `lendables`. This implies that there

could be books, videos, cdRoms, or any other new class of objects added to the lendable hierarchy. Actually, lendableList will hold all three types of objects. When a container holds a collection of dissimilar objects, it is said to be a heterogeneous container.

The lendable class name represents any class of lendable. For this collection, the trick is to have a collection of *pointers* to lendable. The following vector can store 10 instances of any class derived from lendable:

```
vector <lendable*> item(5);
```

The elements in item are not lendables; instead the elements are pointers to any class derived from lendable. A snapshot of memory could look like this:



Here is a program that constructs a vector of pointers to lendables. Notice that the assignment statements assign different classes of objects to the same vector.

```
// File name: testleli.cpp
#include <vector>
using namespace std;

#define DebuggingLateFee  // Sets due dates 2, 7, or 14 days ago
#include "lendable"       // For the lendable class
#include "compfun"

void show(const lendable & aLendable)
{
  cout << "The lendable " << aLendable.lendableID();
  if(aLendable.isOverdue())
    cout << " is overdue. ";
  else
    cout << " is not overdue. ";

  cout << "Late fee = $" << aLendable.lateFee() << endl;

  cout << "Due date: " << aLendable.dueDate().ToString() << endl;
```

```
  if(aLendable.isAvailable())
    cout << "It is available. " << endl;
  else
    cout << "It is not available. " << aLendable.borrowersID()
         << " has it." << endl;

   cout << "-------------------------------------------" << endl;
}

int main()
{
  decimals(cout, 2);  // To show late fees nicely

  vector<lendable*> item(10);
  item[0] = new book("BOOK 1", "Author One", "Title One");
  item[1] = new video("VIDEO 1", "Video Title One");
  item[2] = new book("BOOK 2", "Author Two", "Title Two");
  item[3] = new video("VIDEO 2", "Video Title Two");

  // Check out four lendables for borrower 444-44-4444; show them
  int j;
  for(j = 0; j < 4; j++)
  {
    item[j]->checkSelfOut("444-44-4444");
    show( *item[j] );  // Pass the object pointed to by item[j]
  }
  return 0;
}
```

OUTPUT

```
The lendable BOOK 1 is overdue. Late fee = $7.00
Due date: April 27 1998
It is not available. 444-44-4444 has it.
-------------------------------------------
The lendable VIDEO 1 is overdue. Late fee = $6.50
Due date: May 09 1998
It is not available. 444-44-4444 has it.
-------------------------------------------
The lendable BOOK 2 is overdue. Late fee = $7.00
Due date: April 27 1998
It is not available. 444-44-4444 has it.
-------------------------------------------
The lendable VIDEO 2 is overdue. Late fee = $6.50
Due date: May 09 1998
It is not available. 444-44-4444 has it.
-------------------------------------------
```

The new operator allocates memory for the object and returns the address to that object. Later on, because item is a vector of pointers, operations are performed on the items in the vector by dereferencing them with ->, the arrow operator. For example, during testing, all lendables can be checked back in with this loop:

```
// Check 'em all back in, no matter what class they are
for(j = 0; j < 4; j++)
{
  item[j]->checkSelfIn();  // Polymorphic message
}
```

16.5.1 The lendableList Class

The vector of pointers to lendables can be written as a data member.

CRH Card

Class: lendableList	**Helpers:**
Responsibilities:	lendable
know all lendables	vector
getLendable(lendableID)	
putLendable(lendable)	

Class Definition, First Draft

```
class lendableList {
public:
  void getLendable(string lendID);
  void putLendable(lendable);
private:
  vector<lendable*> my_data;
  int my_size;
};
```

Here is the final version of lendableList after it has undergone further design, testing, and a few changes—especially related to the need for indirection (pointers) in the parameter lists. Also, three member functions were added, and another removed (putLendable was not necessary).

```
class lendableList {
public:
// Default constructor initializes the list of lendables
  lendableList();

//--destructor
  ~lendableList();

//--modifiers
  void addLendable(lendable* lendPtr);
```

```
    // Add the lendable to the lendable list

    void removeLendable(string lendableID);
    // Add the lendable to the lendable list

  //--accessors
    bool getLendable(string searchID, lendable* & lendPtr) const;
    // If found return true, set second argument to point to lendable
    // inside this lendableList. The client can update me indirectly.

  private:
    int my_size;
    int my_index;
    vector <lendable* > my_data;  // vector of pointers to any class
  };                              // derived from the lendable class
```

Because the `lendableList` now returns a pointer to an element, `librarian` can update the `lendable` indirectly like this:

```
lendableList lendList;
lendable* lendPtr;
if( lendList.getLendable("QA76.2", lendPtr) )
{ // assert: lendPtr is a pointer to lendable with ID "QA76.2"
  //         inside the lendable list
  lendPtr->checkSelfOut("Robert Evans");
  // assert: The lendable list has been updated
}
```

The `addLendable` and `removeLendable` member functions were added because it seems that eventually some maintenance program will have to be able to add and remove `lendables`. Besides, you'll see that `addLendable` proves useful in the constructor. A destructor was added because there are many pointers in any `lendableList` (this one starts with 2,000 chunks of allocated memory). When the program terminates, the destructor returns memory. More importantly, however, the destructor also updates the files that store the `lendables`. This makes `lendableList` somewhat *persistent*. The objects will remain intact until the next time the program gets called (assuming the power stays on, that is). To be truly persistent, each object should be stored to a disk as soon as a change is made. This could be done with any number of database management systems, but since we haven't discussed this, the `lendableList` class will instead will get help from the `ifstream` class to maintain the data.

16.5.2 A Heterogeneous Collection

The lendableList class is heterogeneous. The elements stored in a lendableList object can be of any class derived from lendable. This has several implications. First, the container that stores the elements is a vector of pointers to the base class. So you see this data member:

```
vector <lendable*> my_data;
```

The next thing you might notice is parameters with *. Now that the underlying data structure is a vector of pointers, there will be a lot of argument/parameter association where the value being passed is a *pointer* to a lendable:

```
void addLendable(lendable* lendPtr); // Can't pass lendable, need *
bool getLendable(string searchID, lendable* & lendPtr) const;
```

The getLendable function heading was changed during testing because it suddenly seemed important to indicate whether or not a lendable's ID was actually found. The return type is now bool. This also means that the pointer has to be passed back to librarian as a reference parameter lendPtr.

```
bool getLendable(string searchID, lendable* & lendPtr) const;
```

The & was needed because getLendable now returns two values: either true or false and a pointer to the object if found. A getLendable message now looks like this (shown earlier):

```
if( lendList.getLendable("QA76.2", lendPtr) )
{
  lendPtr->checkSelfOut(currentBorrower->borrowersID());
}
```

This guarded action also protects against dereferencing a pointer that points to nothing when lendList is empty, for example, when the input files are not found.

The constructor initializes the lendableList by reading from two different input files: books.dat and video.dat. Maintaining the cdRoms is left as a programming project.

lendableList::addLendable should look familiar to those of you who studied the bag and set classes.

```
void lendableList::addLendable(lendable* lendPtr)
{
```

```
    if(my_size >= my_data.capacity())
     { // Avoid running out of room and out-of-range subscripts
      my_size = my_size + sizeIncrement;
      my_data.resize(my_size);
    }
    my_data[my_size] = lendPtr;
    my_size++;
  }
```

Once again, sequential search is employed to find a `lendable`.

```
bool lendableList::getLendable(string searchID,
                               lendable* & lendPtr) const
{
  int subscript;
  string nextID;

  // Perform a sequential search
  for(subscript = 0; subscript < my_size; subscript++)
  { // Search all items or break out of the loop when found
    nextID = my_data[subscript]->lendableID();
    if(nextID == searchID)
    { // Found it
      break;
    }
  }

  if (subscript < my_size)
  { // Found it
    lendPtr = my_data[subscript];  // Assign a pointer
    return true;
  }
  else
  { // Have to return something, so let it be the first
    lendPtr = my_data[0];  // Return a pointer that hopefully
    return false;          // will never be used by the client!
  }
}
```

Here is a test driver that traverses the entire `lendableList` and then searches for a particular `lendable`. The `librarian` object would often send `getLendable` messages.

```
#include <iostream>
using namespace std;
#include "lendlist"  // For lendableList, lendable, book, video, cdRom
#include "date"      // For Date::ToString
```

```
int main()
{
  lendableList lendList;
  lendable* lendPtr;  // Store a reference to any lendable object

  string searchID = "QA76.2";
  if( lendList.getLendable(searchID, lendPtr) )
  {
    if( lendPtr->isAvailable() )
    { // Don't check out something that is unavailable
      cout << "Check out " << lendPtr->lendableID() << endl;
      lendPtr->checkSelfOut("Robert Evans");
    }
    else
    {
      cout << lendPtr->lendableID() << " unavailable" << endl;
    }
  }
  else
  {
    cout << searchID << " not found." << endl;
    cout << "Please recheck the lendable ID" << endl;
  }

  // Show updated status to indicate lendable list was in fact updated
  if(lendList.getLendable(searchID, lendPtr))
  {
    cout << "Borrower: " << lendPtr->borrowersID() << endl;
    cout << "Due: " << lendPtr->dueDate().ToString() << endl;
  }

  return 0;
}
```

OUTPUT (PROGRAM EXECUTED ON 11-MAY-1998)

```
Check out QA76.2
Borrower: Robert Evans
Due: May 25 1998
```

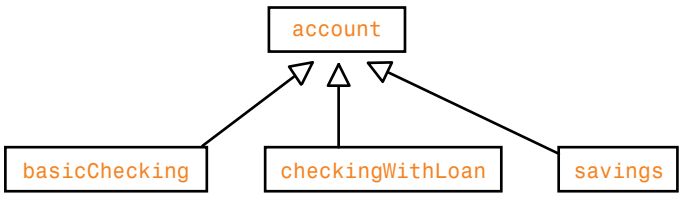
Chapter Summary

- This chapter presented another case study for object-oriented software development to reinforce the object-oriented design strategy introduced in Chapters 12 and 13.
- The notion of inheritance was discovered when several classes were found that had common behavior, common data, and at least one difference, such as different behavior for the same message.

- An abstract class captures common operations and data amongst the classes derived from that base class. The derived classes capture the differences.
- Role playing produced design decisions.
- `lendable` was the major class definition discussed in this chapter. It was designed slowly to indicate the new consideration in designing an abstract class intended to have derived classes.
- Initialization lists were introduced as a way to allow derived classes to call the base class constructor.
- A heterogeneous container was implemented as a `vector` of pointers to the base class. This allows for a collection of objects, where the elements may be constructed from different classes.

Exercises

1. Provide a first-draft design of the base class for an `account` hierarchy. You should be familiar enough with bank accounts to know what might be common. If you are not, consult a domain expert—someone at a bank for example. The difference between `basicChecking` and `checkingWithLoan` is this: `basicChecking` does not allow withdrawals more than the balance but `checkingWithLoan` does. A `checkingWithLoan` object also maintains the amount of money that has been "loaned" to the account. A savings account earns interest; the other two do not.



2. Provide a first-draft design of the three derived classes in the `account` hierarchy. Do not worry about the constructors yet.
3. Write the constructors for one of the derived classes above.
4. Rewrite the `track` class constructor using an initialization list (see `track.cpp`).
5. What action requires an initialization list?

6. Provide a first-draft design of the base class for a United States `employee` hierarchy to capture the operations common to all employees that are paid on an hourly basis with hours over 40 paid at 1.5 times the hourly rate. Then, the only difference amongst the derived classes is in the way the U.S. federal income tax is computed for withholding from the paycheck. For a start, see `weekemp.h` and programming project 7N, "Maintain `weeklyEmp`." Also check out **http://www.irs.ustreas.gov/prod/forms_pubs/pubs.html** for the different tax tables for the current year or use the tax tables given below for 1998. (*Note:* A complete design for all classes of employees would look quite different.)

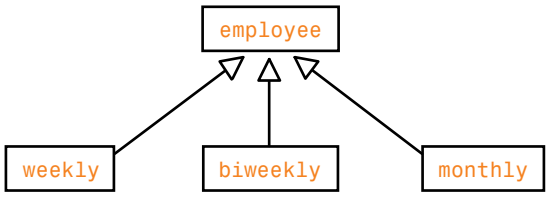


TABLE 16.1. *Weekly payroll period (1998)*
One withholding allowance = $51.92

(a) SINGLE person (including head of household)

If the amount of wages (after subtracting withholding allowances) is:		The amount of income tax to withhold is:
Not over $51 . . .		$0
Over–	**But not over–**	**Of excess over–**
$51	$517 . . 15%	$51
$517	$1,105 . . $69.90 plus 28%	$517
$1,105	$2,493 . . $234.54 plus 31%	$1,105
$2,493	$5,385 . . $664.82 plus 36%	$2,493
$5,385	. . . . . .$1,705.94 plus 39.6%	$5,385

(b) MARRIED person

If the amount of wages (after subtracting withholding allowances) is:		The amount of income tax to withhold is:
Not over $124 . . .		$0
Over–	**But not over–**	**Of excess over–**
$124	$899 . . 15%	$124
$899	$1,855 . . $116.25 plus 28%	$899
$1,855	$3,084 . . $383.93 plus 31%	$1,855
$3,084	$5,439 . . $764.92 plus 36%	$3,084
$5,439	. . . . . .$1,612.72 plus 39.6%	$5,439

TABLE 16.2. *Biweekly payroll period (1998)*
One withholding allowance = $103.85

(a) SINGLE person (including head of household)

If the amount of wages (after subtracting withholding allowances) is:	The amount of income tax to withhold is:
Not over $51 . . .	$0

Over–	But not over–	Of excess over–
$102	$1,035 . . 15%	$102
$1,035	$2,210 . . $139.95 plus 28%	$1,035
$2,210	$4,987 . . $468.95 plus 31%	$2,210
$4,987	$10,769 . . $1,329.82 plus 36%	$4,987
$10,769	. . . . . .$3,411.34 plus 39.6%	$10,769

(b) MARRIED person

If the amount of wages (after subtracting withholding allowances) is:	The amount of income tax to withhold is:
Not over $51 . . .	$0

Over–	But not over–	Of excess over–
$248	$1,798 . . 15%	$248
$1,798	$3,710 . . $232.50 plus 28%	$1,798
$3,710	$6,167 . . $767.86 plus 31%	$3,710
$6,167	$10,879 . . $1,529.53 plus 36%	$6,167
$10,879	. . . . . .$3,225.85 plus 39.6%	$10,879

TABLE 16.3. *Monthly payroll period (1998)*
One withholding allowance = $225.00

(a) SINGLE person (including head of household)

If the amount of wages (after subtracting withholding allowances) is:	The amount of income tax to withhold is:
Not over $51 . . .	$0

Over–	But not over–	Of excess over–
$221	$2,242 . . 15%	$221
$2,242	$4,788 . . $303.15 plus 28%	$2,242
$4,788	$10,804 . . $1,016.03 plus 31%	$4,788
$10,804	$23,333 . . $2,880.99 plus 36%	$10,804
$23,333	. . . . . .$7,391.43 plus 39.6%	$23,333

(b) MARRIED person

If the amount of wages (after subtracting withholding allowances) is:	The amount of income tax to withhold is:
Not over $51 . . .	$0

Over–	But not over–	Of excess over–
$538	$3,896 . . 15%	$538
$3,896	$8,038 . . $503.70 plus 28%	$3,896
$8,038	$13,363 . . $1,663.46 plus 31%	$8,038
$13,363	$23,571 . . $3,314.21 plus 36%	$13,363
$23,571	. . . . . .$6,989.09 plus 39.6%	$23,571

7. Design the class definitions for the derived classes `weekly`, `biWeekly`, and `monthly`.

Analysis/Design/Programming Tips

1. Inheritance and polymorphism are part of object-oriented analysis, design, and programming. But they are not the only things.

Object-oriented (OO) thinking also involves encapsulation—a public interface to private data. Additionally, OO software development involves making analysis and design decisions with the object as the major architectural structure. If there is inheritance, so be it. If there is not, you are still doing things in an object-oriented fashion.

2. Use the delegation model to help with your design.

Object-oriented software often has one object delegating responsibility to another object that may in turn delegate responsibility to another object. Try to think that way. Users of the class can then have a clearer understanding of the system. For example, `librarian` may send a `checkSelfIn` message that in turn fires off messages to other objects. This is analogous to making a call to a free function, that may in turn call another function behind the scenes. For example, the `find` function from `<algorithm>` most likely makes several function calls and/or message sends behind the scenes.

3. Role playing helps because three or four minds are better than one.

Any project can benefit from many people. Most software is developed in a team setting. Anyone can have ideas. Someone with no knowledge of programming can help simply by asking questions that others may be too shy to ask, feeling they are supposed to know it already. The team approach also helps set up a common vocabulary among the stakeholders. Besides, it can be more fun working in groups, even though it can also be challenging to deal with a diversity of opinions.

4. Use inheritance when you can generalize about two or more objects.

If you recognize that several classes have some common behaviors and attributes but there is some distinguishing behavior, there is a chance that inheritance might

prove useful. However, with all the special considerations necessary to plan for adding new derived classes, it might not seem worth it. Sometimes inheritance is the best way to go. For example, consider a window on your computer screen. It has a collection of other windows, buttons, menus, selector lists, icons, and so on. Think of that window as a heterogeneous list of graphic objects (kind of like `lendableList`). All graphic objects can be "drawn" in a screen, but they are drawn in different ways. When the window draws itself, the polymorphic draw operation is applied to all the graphics in the window collection. There are some commonalities between the graphic objects; however, there are enough differences to justify many derived classes. This has proven to be an effective use of inheritance.

5. Put only public messages in the public section.

Don't clutter the public interface with member functions best kept private. If you have an inheritance hierarchy, place the utility functions (those not called by any client) in the `protected:` access sections. With public inheritance, all derived classes inherit the ancestor's `public:` and `protected:` members.

6. This has been only a brief introduction to inheritance.

This chapter did not attempt to demonstrate all concepts related to inheritance. It was only an introduction. Proper use of inheritance is still being debated and opinions vary widely.

7. There is a heavy use of indirection in `lendableList`. Here is a summary.

First, the container that stores the elements is a `vector` of pointers to the base class. So `lendableList` has this data member:

```
vector<lendable*> my_data;  // Elements can point to derived objects
```

Another new thing was pointer parameters (with `*` after them). These parameters are used to communicate addresses of objects rather than objects themselves. This is part of the syntax required for implementing polymorphism in C++.

```
void addLendable(lendable* lendPtr);  // Need *; cannot pass a lendable
```

A new kind of parameter was shown in `lendableList::getLendable`.

```
bool getLendable(string searchID, lendable* & lendPtr) const;
```

The second parameter is a reference to a pointer. The `&` is added so a change to `lendPtr` in the function also changes the pointer argument in the message, which in the following call is `lendPtr`.

```
lendList.getLendable("QA76.2", aPointerToALendable)
```

Programming Projects

16A Implement `cdRom`

Completely implement and test the `cdRom` class as a class derived from `lendable`. You will need these files: `lendable.h` and `lendable.cpp`.

16B Implement the `account` Hierarchy

Completely implement and test the `account` hierarchy described in exercise 1.

16C Implement the `employee` Hierarchy

Completely implement and test the `employee` hierarchy described in exercise 6.

16D Complete the College Library System

Find two or three others and

- walk through some scenarios with the college library system
- develop your own CRH cards or modify the ones that exist in this chapter

When you have a full understanding of the system (have practiced those scenarios)

- design all class interfaces (the `lendable` hierarchy has mostly been completed in `lendable.h` and `lendable.cpp`)
- separately test all classes
- integrate the classes and test the entire system