CHAPTER 3

Data Abstraction: The Walls

Abstract Data Types

Specifying ADTs

- The ADT List
- The ADT Sorted List
- Designing an ADT
- Axioms (Optional)

Implementing ADTs

- C++ Classes
- C++ Namespaces
- An Array-Based Implementation of the ADT List
- C++ Exceptions

Summary

Cautions

Self-Test Exercises

Exercises

Programming Problems

PREVIEW This chapter elaborates on data abstraction, which was introduced in Chapter 1 as a technique for increasing the modularity of a program—for building "walls" between a program and its data structures. During the design of a solution, you will discover that you need to support several operations on the data and therefore need to define abstract data types (ADTs). This chapter will introduce some simple abstract data types and use them to demonstrate the advantages of abstract data types in general. In Part II of this book, you will see several other important ADTs.

Only after you have clearly specified the operations of an abstract data type should you consider data structures for implementing it. This chapter explores implementation issues and introduces C++ classes as a way to hide the implementation of an ADT from its users.

ABSTRACT DATA TYPES

A modular program is easier to write, read, and modify

Modularity is a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components. You can focus on one task at a time in a modular program without other distractions. Thus, a modular program is easier to write, read, and modify. Modularity also isolates errors and eliminates redundancies.

Write specifications for each module before implementing it

You can develop modular programs by piecing together existing software components with functions that have yet to be written. In doing so, you should focus on *what* a module does and not on *how* it does it. To use existing software, you need a clear set of specifications that details how the modules behave. To write new functions, you need to decide what you would like them to do and proceed under the assumption that they exist and work. In this way you can write the functions in relative isolation from one another, knowing what each one will do but not necessarily *how* each will eventually do it. That is, you should practice **functional abstraction.**

Isolate the implementation details of a module from other modules

While writing a module's specifications, you must identify details that you can hide within the module. The principle of **information hiding** involves not only hiding these details, but also making them *inaccessible* from outside a module. One way to understand information hiding is to imagine **walls** around the various tasks a program performs. These walls prevent the tasks from becoming entangled. The wall around each task *T* prevents the other tasks from "seeing" how *T* is performed. Thus, if task *Q* uses task *T*, and if the method for performing task *T* changes, task *Q* will not be affected. As Figure 3-1 illustrates, the wall prevents task *Q*'s method of solution from depending on task *T*'s method of solution.

The isolation of the modules cannot be total, however. Although task *Q* does not know *how* task *T* is performed, it must know *what* task *T* is

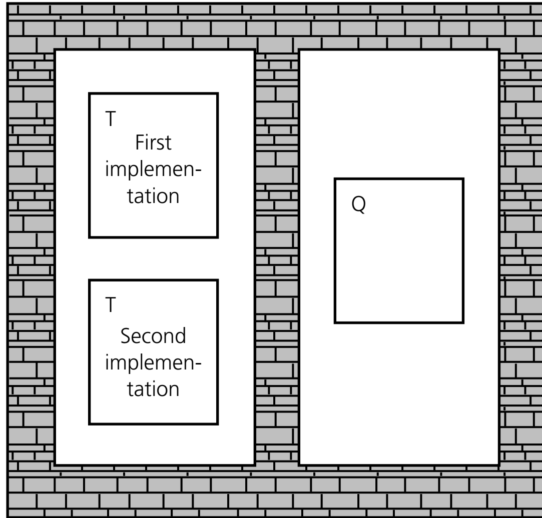


Figure 3-1

Isolated tasks: the implementation of task *T* does not affect task *Q*

and how to initiate it. For example, suppose that a program needs to operate on a sorted array of names The program may, for instance, need to search the array for a given name or display the names in alphabetical order. The program thus needs a function *S* that sorts an array of names. Although the rest of the program knows that function *S* will sort an array, it should not care how *S* accomplishes its task. Thus, imagine a tiny slit in each wall, as Figure 3-2 illustrates. The slit is not large enough to allow the outside world to see the function's inner workings, but

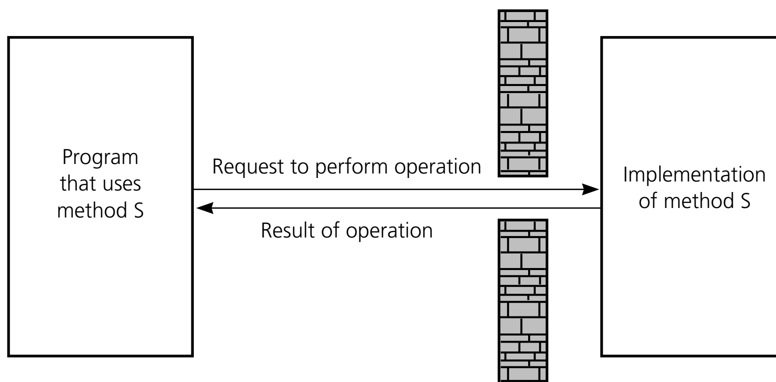


Figure 3-2

A slit in the wall

things can pass through the slit into and out of the function. For example, you can pass the array into the sort function, and the function can pass the sorted array out to you. What goes in and comes out is governed by the terms of the function's specifications, or **contract**: *If you use the function in this way, this is exactly what it will do for you.*

Often the solution to a problem requires operations on data. Such operations are broadly described in one of three ways:

Typical operations on data

- **Add** data to a data collection.
- **Remove** data from a data collection.
- **Ask questions** about the data in a data collection.

The details of the operations, of course, vary from application to application, but the overall theme is the management of data. Realize, however, that not all problems use or require these operations.

Both functional and data abstraction ask you to think "what," not "how"

Data abstraction asks that you think in terms of *what* you can do to a collection of data independently of *how* you do it. Data abstraction is a technique that allows you to develop each data structure in relative isolation from the rest of the solution. The other modules of the solution will "know" what operations they can perform on the data, but they should not depend on how the data is stored or how the operations are performed. Again, the terms of the contract are *what* and not *how*. Thus, data abstraction is a natural extension of functional abstraction.

An ADT is a collection of data and a set of operations on that data

A collection of data together with a set of operations on that data are called an **abstract data type**, or **ADT.** For example, suppose that you need to store a collection of names in a manner that allows you to search rapidly for a given name. The binary search algorithm described in Chapter 2 enables you to search an array efficiently, if the array is sorted. Thus, one solution to this problem is to store the names sorted in an array and to use a binary search algorithm to search the array for a specified name. You can view the *sorted array together with the binary search algorithm* as an ADT that solves this problem.

Specifications indicate what ADT operations do, but not how to implement them

The description of an ADT's operations must be rigorous enough to specify completely their effect on the data, yet it must not specify how to store the data nor how to carry out the operations. For example, the ADT operations should not specify whether to store the data in consecutive memory locations or in disjoint memory locations. You choose a particular **data structure** when you **implement** an ADT.

Data structures are part of an ADT's implementation

Recall that a data structure is a construct that you can define within a programming language to store a collection of data. For example, arrays and structures, which are built into C++, are data structures. However, you can invent other data structures. For example, suppose that you wanted a data structure to store both the names and salaries of a group of employees. You could use the following C++ statements:

```
const int MAX_NUMBER = 500;
string names[MAX_NUMBER];
double salaries[MAX_NUMBER];
```

Here the employee `names[i]` has a salary of `salaries[i]`. The two arrays `names` and `salaries` together form a data structure, yet C++ has no single data type to describe it.

When a program must perform data operations that are not directly supported by the language, you should first design an abstract data type and carefully specify what the ADT operations are to do (the contract). Then—*and only then*—should you implement the operations with a data structure. If you implement the operations properly, the rest of the program will be able to assume that the operations perform as specified—that is, that the terms of the contract are honored. However, the program must not depend on a particular method for supporting the operations.

Carefully specify an ADT's operations before you implement them

An abstract data type is not another name for a data structure.

KEY CONCEPTS

ADTs versus Data Structures

- An abstract data type is a collection of data and a set of operations on that data.
- A data structure is a construct within a programming language that stores a collection of data.

To give you a better idea of the conceptual difference between an ADT and a data structure, consider a refrigerator's ice dispenser, as Figure 3-3 illustrates. It has water as input and produces as output either chilled water, crushed ice, or ice cubes according to which one of three buttons you push. It also has an indicator that lights when no ice is presently available. The dispenser is analogous to an abstract data type. The water is analogous to data; the operations are *chill*, *crush*, *cube*, and *isEmpty*. At this level of design, you are not concerned with how the dispenser will perform its operations, only that it performs them. If you want crushed ice, do you really care how the dispenser accomplishes its task as long as it does so correctly? Thus, after you have specified the dispenser's functions, you can design many uses for crushed ice without knowing how the

ADTs and data structures are not the same

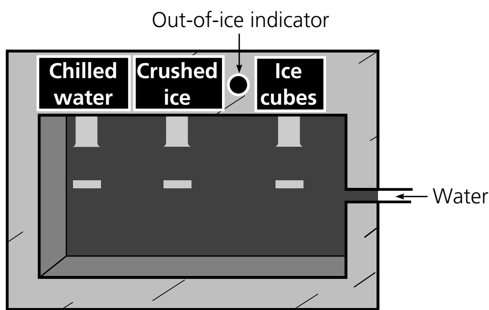


Figure 3-3

A dispenser of chilled water, crushed ice, and ice cubes

dispenser accomplishes its tasks and without the distraction of engineering details.

Eventually, however, someone must build the dispenser. Exactly how will this machine produce crushed ice, for example? It could first make ice cubes and then either crush them between two steel rollers or smash them into small pieces by using hammers. Many other techniques are possible. The internal structure of the dispenser corresponds to the implementation of the ADT in a programming language, that is, to a data structure.

Although the owner of the dispenser does not care about its inner workings, he or she does want a design that is as efficient in its operation as possible. Similarly, the dispenser's manufacturer wants a design that is as easy and cheap to build as possible. You should have these same concerns when you choose a data structure to implement an ADT in C++. Even if you do not implement the ADT yourself, but instead use an already implemented ADT, you—like the person who buys a refrigerator—should care about at least the ADT's efficiency.

Notice that the dispenser is surrounded by steel walls. The only breaks in the walls accommodate the input (water) to the machine and its output (chilled water, crushed ice, or ice cubes). Thus, the machine's interior mechanisms are not only hidden from the user but also are inaccessible. In addition, the mechanism of one operation is hidden from and inaccessible to another operation.

This modular design has benefits. For example, you can improve the operation *crush* by modifying its module without affecting the other modules. You could also add an operation by adding another module to the machine without affecting the original three operations. Thus, both abstraction and information hiding are at work here.

A program should not depend on the details of an ADT's implementation

To summarize, data abstraction results in a wall of ADT operations between data structures and the program that accesses the data within these data structures, as Figure 3-4 illustrates. If you are on the program's side of the wall, you will see an **interface** that enables you to communicate with the data structure. That is, you request the ADT operations to manipulate the data in the data structure, and they pass the results of these manipulations back to you.

Using an ADT is like using a vending machine

This process is analogous to using a vending machine. You press buttons to communicate with the machine and obtain something in return. The machine's external design dictates how you use it, much as an ADT's specifications govern what its operations are and what they do. As long as you use a vending machine according to its design, you can ignore its inner technology. As long as you agree to access data only by using ADT operations, your program can be oblivious to any change in the data structures that implement the ADT.

The following pages describe how to use an abstract data type to realize data abstraction's goal of separating the operations on data from the implementation of these operations. In doing so, we will look at several examples of ADTs.

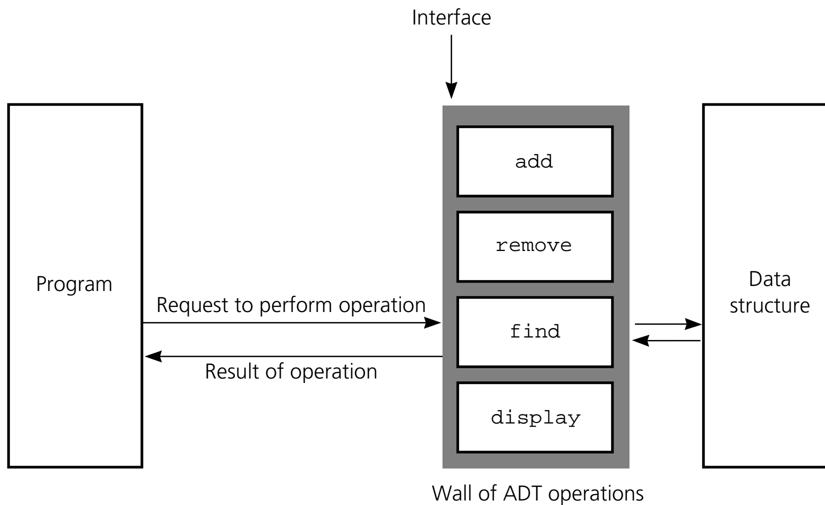


Figure 3-4

A wall of ADT operations isolates a data structure from the program that uses it

SPECIFYING ADTs

To elaborate on the notion of an abstract data type, consider a list that you might encounter, such as a list of chores, a list of important dates, a list of addresses, or the grocery list pictured in Figure 3-5. As you write a grocery list, where do you put new items? Assuming that you write a neat one-column list, you probably add new items to the end of the list. You could just as well add items to the beginning of the list or add them so that your list is sorted alphabetically. Regardless, the items on a list appear in a sequence. The list has one first item and one last item. Except for the first and last items, each item has a unique **predecessor** and a unique **successor.** The first item—the **head** or **front**

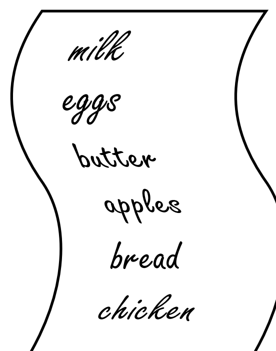


Figure 3-5

A grocery list

of the list—does not have a predecessor, and the last item—the **tail** or **end** of the list—does not have a successor.

Lists contain items of the same type: You can have a list of grocery items or a list of phone numbers. What can you do to the items on a list? You might count the items to determine the length of the list, add an item to the list, remove an item from the list, or look at (**retrieve**) an item. The items on a list, together with operations that you can perform on the items, form an abstract data type. You must specify the behavior of the ADT's operations on its data, that is, the list items. It is important that you focus only on specifying the operations and not on how you will implement them. That is, do not bring to this discussion any preconceived notion of a data structure that the term "list" might suggest.

Where do you add a new item and which item do you want to look at? The various answers to these questions lead to several kinds of lists. You might decide to add, delete, and retrieve items only at the end of the list or only at the front of the list or at both the front and end of the list. The specifications of these lists are left as an exercise; next we will discuss a more general list.

The ADT List

Once again, consider the grocery list pictured in Figure 3-5. The previously described lists, which manipulate items at one or both ends of the list, are not really adequate for an actual grocery list. You would probably want to access items anywhere on the list. That is, you might look at the item at position *i*, delete the item at position *i,* or insert an item at position *i* on the list. Such operations are part of the ADT **list.**

KEY CONCEPTS

ADT List Operations

1. Create an empty list.
2. Destroy a list.
3. Determine whether a list is empty.
4. Determine the number of items on a list.
5. Insert an item at a given position in the list.
6. Delete the item at a given position in the list.
7. Look at (retrieve) the item at a given position in the list.

Note that it is customary to include an initialization operation that creates an empty list and an operation that destroys a list. Other operations that determine whether the list is empty or the length of the list are also useful.

Although the six items on the list in Figure 3-5 have a sequential order, they are not necessarily sorted by name. Perhaps the items appear

in the order in which they occur on the grocer's shelves, but more likely they appear in the order in which they occurred to you as you wrote the list. The ADT list is simply an ordered collection of items that you reference by position number.

You reference list items by their position within the list

The following pseudocode specifies the operations for the ADT list in more detail. Figure 3-6 shows the UML diagram for this ADT.

Pseudocode for the ADT List Operations

KEY CONCEPTS

```
// ListItemType is the type of the items stored in the list.

+createList()
// Creates an empty list.

+destroyList()
// Destroys a list.

+isEmpty():boolean {query}
// Determines whether a list is empty.

+getLength():integer {query}
// Returns the number of items that are in a list.

+insert(in index:integer, in newItem:ListItemType,
        out success:boolean)
// Inserts newItem at position index of a list, if
// 1 <= index <= getLength()+1.
// If index <= getLength(), items are renumbered as
// follows: The item at index becomes the item at
// index+1, the item at index+1 becomes the
// item at index+2, and so on. The success flag indicates
// whether the insertion was successful.

+remove(in index:integer, out success:boolean)
// Removes the item at position index of a list, if
// 1 <= index <= getLength(). If index < getLength(), items
// are renumbered as follows: The item at index+1 becomes
// the item at index, the item at index+2 becomes the item
// at index+1, and so on. The success flag indicates whether
// the deletion was successful.

+retrieve(in index:integer, out dataItem:ListItemType,
          out success:boolean) {query}
// Copies the item at position index of a list into
// dataItem, if 1 <= index <= getLength(). The list is left
// unchanged by this operation. The success flag indicates
// whether the retrieval was successful.
```

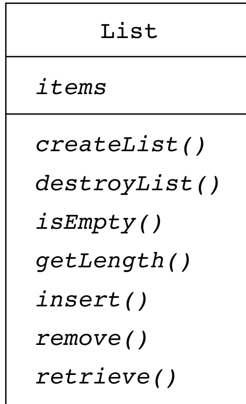


Figure 3-6

UML diagram for ADT `List`

To get a more precise idea of how the operations work, apply them to the grocery list

milk, eggs, butter, apples, bread, chicken

where milk is the first item on the list and chicken is the last item. To begin, consider how you can construct this list by using the ADT list operations. One way is first to create an empty list `aList` and then use a series of insertion operations to append successively the items to the list as follows:

```
aList.createList()
aList.insert(1, milk, success)
aList.insert(2, eggs, success)
aList.insert(3, butter, success)
aList.insert(4, apples, success)
aList.insert(5, bread, success)
aList.insert(6, chicken, success)
```

The notation[1] `aList.O` indicates that an operation `O` applies to the list `aList`.

Notice that the list's insertion operation can insert new items into any position of the list, not just at its front or end. According to `insert`'s specification, if a new item is inserted into position *i*, the position of each item that was at a position of *i* or greater is increased by 1. Thus, for example, if you start with the previous grocery list and you perform the operation

```
aList.insert(4, nuts, success)
```

the list `aList` becomes

milk, eggs, butter, nuts, apples, bread, chicken

All items that had position numbers greater than or equal to 4 before the insertion now have their position numbers increased by 1 after the insertion.

Similarly, the deletion operation specifies that if an item is deleted from position *i*, the position of each item that was at a position greater than *i* is decreased by 1. Thus, for example, if `aList` is the list

milk, eggs, butter, nuts, apples, bread, chicken

and you perform the operation

```
aList.remove(5, success)
```

the list becomes

milk, eggs, butter, nuts, bread, chicken

All items that had position numbers greater than 5 before the deletion now have their position numbers decreased by 1 after the deletion.

[1]This notation is compatible with the planned C++ implementation of the ADT.

These examples illustrate that an ADT can specify the effects of its operations without having to indicate how to store the data. The specifications of the seven operations are the sole terms of the contract for the ADT list: *If you request that these operations be performed, this is what will happen*. The specifications contain no mention of how to store the list or how to perform the operations; they tell you only what you can do to the list. It is of fundamental importance that the specification of an ADT *not* include implementation issues. This restriction on the specification of an ADT is what allows you to build a wall between an implementation of an ADT and the program that uses it. (Such a program is called a **client**.) The behavior of the operations is the only thing on which a program should depend.

An ADT specification should not include implementation issues

A program should depend only on the behavior of the ADT

Note that the insertion, deletion, and retrieval operations specify the argument `success`, which provides the ADT with a simple mechanism to communicate operation failure to its client. For example, if you try to delete the tenth item from a five-item list, `remove` can set `success` to `false`. Likewise, `insert` can set `success` to `false` if, for example, the list is full or `index` is out of range. In this way, `success` enables the client to handle error situations in an implementation-independent way.

What does the specification of the ADT list tell you about its behavior? It is apparent that the list operations fall into the three broad categories presented earlier in this chapter.

- The operation `insert` **adds** data to a data collection.
- The operation `remove` **removes** data from a data collection.
- The operations `isEmpty`, `getLength`, and `retrieve` **ask questions** about the data in a data collection.

Once you have satisfactorily specified the behavior of an ADT, you can design applications that access and manipulate the ADT's data solely in terms of its operations and without regard for its implementation. As a simple example, suppose that you want to display the items on a list. Even though the wall between the implementation of the ADT list and the rest of the program prevents you from knowing how the list is stored, you can write a function `displayList` in terms of the operations that define the ADT list. The pseudocode for such a function follows:[2]

An implementation-independent application of the ADT list

```
displayList(in aList:List)
// Displays the items on the list aList.

   for (position = 1 through aList.getLength())
   {  aList.retrieve(position, dataItem, success)
      Display dataItem
   }  // end for
```

[2]In this example, `displayList` is not an ADT operation, so a functional notation that specifies `aList` as an argument is used.

Notice that as long as the ADT list is implemented correctly, the `displayList` function will perform its task. In this case, `retrieve` successfully retrieves each list item, because `position`'s value is always valid, so `success` can be ignored.

The function `displayList` does not depend on *how* you implement the list. That is, the function will work regardless of whether you use an array or some other data structure to store the list's data. This feature is a definite advantage of abstract data types. In addition, by thinking in terms of the available ADT operations, you will not be distracted by implementation details. Figure 3-7 illustrates the wall between `displayList` and the implementation of the ADT list.

As another application of the ADT operations, suppose that you want a function `replace` that replaces the item in position `i` with a new item. If the ith item exists, `replace` deletes the item and inserts the new item at position `i`, as follows:

```
replace(in aList:List, in i:integer,
        in newItem:ListItemType, out success:boolean)
// Replaces the ith item on the list aList with newItem.
// The success flag indicates whether the replacement
// was successful.

   aList.remove(i, success)
   if (success)
      aList.insert(i, newItem, success)
```

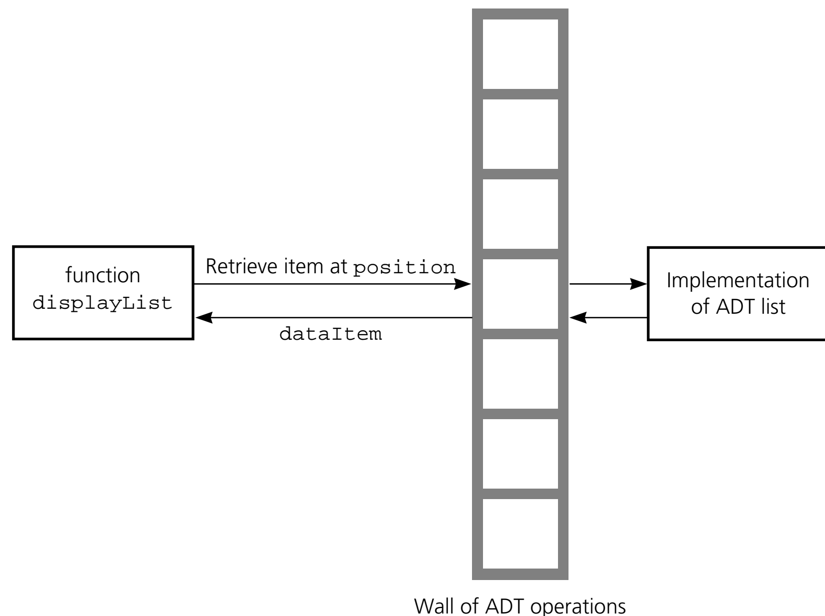


Figure 3-7

The wall between `displayList` and the implementation of the ADT list

If `remove` is successful it sets `success` to `true`. By testing `success`, `replace` will attempt the insertion only if the deletion actually occurred. Then `insert` sets `success`, which `replace` returns to the function that called it. If `remove` is unsuccessful for any reason, including an incorrect value of `i`, it sets `success` to `false`. The `replace` function then ignores the insertion and returns `success`.

In both of the preceding examples, notice how you can focus on the task at hand without the distraction of implementation details such as arrays. With less to worry about, you are less likely to make an error in your logic when you use the ADT operations in applications such as `displayList` and `replace`. Likewise, when you finally implement the ADT operations in C++, you will not be distracted by these applications. In addition, because `displayList` and `replace` do not depend on any implementation decisions that you make, they are not altered by your decisions. These assertions assume that you do not change the specifications of the ADT operations when you implement them. However, as Chapter 1 pointed out, developing software is not a linear process. You may realize during implementation that you need to refine your specifications. Clearly, changes to the specification of any module affect any already-designed uses of that module.

You can use ADT operations in an application without the distraction of implementation details

To summarize, you can specify the behavior of an ADT independently of its implementation. Given such a specification, and without any knowledge of how the ADT will be implemented, you can design applications that utilize the ADT's operations to access its data.

The ADT Sorted List

One of the most frequently performed computing tasks is the maintenance, in some *specified* order, of a collection of data. Many examples immediately come to mind: students placed in order by their names, baseball players listed in order by their batting averages, and corporations listed in order by their assets. These orders are called **sorted.** In contrast, the items on a grocery list might be ordered—the order in which they appear on the grocer's shelves, for example—but they are probably not sorted by name.

The problem of *maintaining* sorted data requires more than simply sorting the data. Often you need to insert some new data item into its proper, sorted place. Similarly, you often need to delete some data item. For example, suppose your university maintains an alphabetical list of the students who are currently enrolled. The registrar must insert names into and delete names from this list because students constantly enroll in and leave school. These operations should preserve the sorted order of the data.

The ADT sorted list maintains items in sorted order

The following specifications define the operations for the ADT **sorted list.**

KEY CONCEPTS

Pseudocode for the ADT Sorted List Operations

```
// ListItemType is the type of the items stored in the list.

+createSortedList()
// Creates an empty sorted list.

+destroySortedList()
// Destroys a sorted list.

+sortedIsEmpty():boolean {query}
// Determines whether a sorted list is empty.

+sortedGetLength():integer {query}
// Returns the number of items that are in a sorted list.

+sortedInsert(in newItem:ListItemType, out success:boolean)
// Inserts newItem into its proper sorted position in a
// sorted list. The success flag indicates whether the
// insertion was successful.

+sortedRemove(in anItem:ListItemType, out success:boolean)
// Deletes anItem from a sorted list. The success flag
// indicates whether the deletion was successful.

+sortedRetrieve(in index:integer, out dataItem:ListItemType,
                out success:boolean) {query}
// Sets dataItem to the item at position index of a sorted
// list, if 1 <= index <= sortedGetLength(). The list is left
// unchanged by this operation. The success flag indicates
// whether the retrieval was successful.

+locatePosition(in anItem:ListItemType,
             out isPresent:boolean):integer {query}
// Returns the position where the item belongs or exists
// in a sorted list. The isPresent flag indicates whether
// anItem is currently in the list. The item anItem and the
// list are unchanged.
```

The ADT sorted list differs from the ADT list in that a sorted list inserts and deletes items by their values and not by their positions. For example, `sortedInsert` determines the proper position for `newItem` according to its value. Also, `locatePosition`—which determines the position of any item, given its value—is a sorted list operation but not a list operation. However, `sortedRetrieve` is like list's `retrieve`: Both operations retrieve an item, given its position. The function `sortedRetrieve` enables you, for

example, to write another function to retrieve and then display each item in a sorted list.

Designing an ADT

The design of an abstract data type should evolve naturally during the problem-solving process. As an example of how this process might occur, suppose that you want to determine the dates of all the holidays in a given year. One way to do this is to examine a calendar. That is, you could consider each day in the year and ascertain whether that day is a holiday. The following pseudocode is thus a possible solution to this problem:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

   date = date of first day of year
   while (date is before the first day of year+1)
   {   if (date is a holiday)
         write (date," is a holiday")

       date = date of next day
   }  // end while
```

What data are involved here? Clearly, this problem operates on dates, where a date consists of a month, day, and year. What operations will you need to solve the holiday problem? Your ADT must specify and restrict the legal operations on the dates just as the fundamental data type `int` restricts you to operations such as addition and comparison. You can see from the previous pseudocode that you must

What data does a problem require?

- Determine the date of the first day of a given year
- Determine whether a date is before another date
- Determine whether a date is a holiday
- Determine the date of the day that follows a given date

What operations does a problem require?

Thus, you could define the following operations for your ADT:

```
+firstDay(in year:integer):Date {query}
// Returns the date of the first day of a given year.

+isBefore(in date1:Date,
          in date2:Date) : boolean {query}
// Returns true if date1 is before date2,
// otherwise returns false.

+isHoliday(in aDate:Date) : boolean {query}
// Returns true if date is a holiday,
// otherwise returns false.

+nextDay(in aDate:Date) : Date {query}
// Returns the date of the day after a given date.
```

The `listHolidays` pseudocode now appears as follows:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

   date = firstDay(year)
   while (isBefore(date, firstDay(year+1)))
   {  if (isHoliday(date))
         write (date," is a holiday ")
      date = nextDay(date)
   }  // end while
```

Thus, you can design an ADT by identifying data and choosing operations that are suitable to your problem. After specifying the operations, you use them to solve your problem independently of the implementation details of the ADT.

An appointment book. As another example of an ADT design, imagine that you want to create a computerized appointment book that spans a one-year period. Suppose that you make appointments only on the hour and half hour between 8 a.m. and 5 p.m. You want your system to store a brief notation about the nature of each appointment along with the date and time.

To solve this problem, you can define an ADT appointment book. The data items in this ADT are the appointments, where an appointment consists of a date, time, and purpose. What are the operations? Two obvious operations are

- Make an appointment for a certain date, time, and purpose. (You will want to be careful that you do not make an appointment at an already occupied time.)
- Cancel the appointment for a certain date and time.

In addition to these operations, it is likely that you will want to

- Ask whether you have an appointment at a given time.
- Determine the nature of your appointment at a given time.

Finally, ADTs typically have initialization and destruction operations.

Thus, the ADT appointment book can have the following operations:

```
+createAppointmentBook()
// Creates an empty appointment book.

+isAppointment(in apptDate:Date,
               in apptTime:Time) : boolean {query}
// Returns true if an appointment exists for the date
// and time specified; otherwise returns false.
```

```
+makeAppointment(in apptDate:Date, in apptTime:Time,
                 in purpose:string) : boolean
// Inserts the appointment for the apptDate, apptTime, and
// purpose specified as long as it does not conflict with
// an existing appointment.
// Returns true if successful, false otherwise.

+cancelAppointment(in apptDate:Date,
                   in apptTime:Time) : boolean
// Deletes the appointment for the apptDate and apptTime
// specified.
// Returns true if successful, false otherwise

+checkAppointment(in apptDate:Date, in apptTime:Time,
                  out purpose:string) {query}
// Retrieves the purpose of the appointment at the given
// apptDate/apptTime, if one exists. Otherwise, purpose
// is set equal to a null string
```

You can use these ADT operations to design other operations on the appointments. For example, suppose that you want to change the date or time of a particular appointment within the existing appointment book *apptBook*. The following pseudocode indicates how to accomplish this task by using the previous ADT operations:

```
// change the date or time of an appointment

read (oldDate, oldTime, newDate, newTime)
// get purpose of appointment

apptBook.checkAppointment(oldDate, oldTime, oldPurpose)

if (oldPurpose not null)
{  // see if new date/time is available
   if (apptBook.isAppointment(newDate, newTime))
      // new date/time is booked
      write ("You already have an appointment at ", newTime,
             " on ", newDate)

   else  // new date/time is available
   {  apptBook.cancelAppointment(oldDate, oldTime)
      if (apptBook.makeAppointment(newDate, newTime, oldPurpose))
         write ("Your appointment has been rescheduled to ",
             newTime, " on ", newDate)
   }  // end if
}  // end if
else
   write ("You do not have an appointment at ", oldTime,
          " on ", oldDate)
```

You can use an ADT without knowledge of its implementation

Again notice that you can design applications of ADT operations without knowing how the ADT is implemented. The exercises at the end of this chapter provide examples of other tasks that you can perform with this ADT.

ADTs that suggest other ADTs. Both of the previous examples require you to represent a date; the appointment book example also requires you to represent the time. C++ has a date-time *struct* specified in *time.h* that you can use to represent the date and the time. You can also design ADTs to represent these items in a more object-oriented way. It is not unusual for the design of one ADT to suggest other ADTs. In fact, you can use one ADT to implement another ADT. The programming problems at the end of this chapter ask you to design and implement the simple ADTs date and time.

You can use an ADT to implement another ADT

This final example also describes an ADT that suggests other ADTs for its implementation. Suppose that you want to design a database of recipes. You could think of this database as an ADT: The recipes are the data items, and some typical operations on the recipes could include the following:

```
+insertRecipe(in aRecipe:Recipe, out success:boolean)
// Inserts recipe into the database.

+deleteRecipe(in aRecipe:Recipe, out success:boolean)
// Deletes recipe from the database.

+retrieveRecipe(in name:string, out aRecipe:Recipe,
                out success:boolean) {query}
// Retrieves the named recipe from the database.
```

This level of the design does not indicate such details as where *insertRecipe* will place a recipe into the database.

Now imagine that you want to write a function that scales a recipe retrieved from the database: If the recipe is for n people, you want to revise it so that it will serve m people. Suppose that the recipe contains measurements such as 2½ cups, 1 tablespoon, and ¼ teaspoon. That is, the quantities are given as mixed numbers—integers and fractions—in units of cups, tablespoons, and teaspoons.

This problem suggests another ADT—measurement—with the following operations:

```
+getMeasure() : Measurement {query}
// Returns the measure.
```

```
+setMeasure(in m: Measurement)
// Sets a measure.

+scaleMeasure(out newMeasure:Measurement,
              in scaleFactor:float)
// Multiplies measure by a fractional scaleFactor, which
// has no units, to obtain newMeasure.

+convertMeasure(in oldUnits:MeasureUnit,
                out newMeasure:Measurement,
                in newUnits:MeasureUnit) {query}
// Converts measure from its old units to newMeasure in
// new units.
```

Suppose that you want the ADT measurement to perform exact fractional arithmetic. Because our planned implementation language C++ does not have a data type for fractions and floating-point arithmetic is not exact, another ADT called fraction is in order. Its operations could include addition, subtraction, multiplication, and division of fractions. For example, you could specify addition as

```
+addFractions(in first:Fraction,
              in second:Fraction) : Fraction
// Adds two fractions and returns the sum reduced to lowest
// terms.
```

Moreover, you could include operations to convert a mixed number to a fraction and to convert a fraction to a mixed number when feasible.

When you finally implement the ADT measurement, you can use the ADT fraction. That is, you can use one ADT to implement another ADT.

Axioms (Optional)

The previous specifications for ADT operations have been stated rather informally. For example, they rely on your intuition to know the meaning of "an item is at position *i*" in an ADT list. This notion is simple, and most people will understand its intentions. However, some abstract data types are much more complex and less intuitive than a list. For such ADTs, you should use a more rigorous method of defining the behavior of their operations: You must supply a set of mathematical rules—called **axioms**—that precisely specify the behavior of each ADT operation.

An axiom is a mathematical rule

An axiom is actually an invariant—a true statement—for an ADT operation. For example, you are familiar with axioms for algebraic operations; in particular, you know the following rules for multiplication:

Axioms for multiplication

$$(a \times b) \times c = a \times (b \times c)$$
$$a \times b = b \times a$$
$$a \times 1 = a$$
$$a \times 0 = 0$$

These rules, or axioms, are true for any numeric values of a, b, and c, and describe the behavior of the multiplication operator $\times$.

Axioms specify the behavior of an ADT

In a similar fashion, you can write a set of axioms that completely describes the behavior of the operations for the ADT list. For example,

A newly created list is empty

is an axiom since it is true for all newly created lists. You can state this axiom succinctly in terms of the ADT list operations as follows:

`(aList.createList()).isEmpty()` is true

That is, the list `aList` is empty.

The statement

If you insert an item x into the ith position of an ADT list, retrieving the ith item will result in x

is true for all lists, and so it is an axiom. You can state this axiom in terms of the ADT list operations, as follows:[3]

```
(aList.insert(i, x)).retrieve(x) = x
```

That is, `retrieve` retrieves from position `i` of list `aList` the item `x` that `insert` has put there. To simplify the notation, the `success` arguments are omitted and `retrieve` is treated as if it were a valued function.

The following axioms formally define the ADT list:

KEY CONCEPTS

Axioms for the ADT List

```
1.  (aList.createList()).getLength() = 0
2.  (aList.insert(i, x)).getLength() =
       aList.getLength() + 1
3.  (aList.remove(i)).getLength() =
       aList.getLength() - 1
4.  (aList.createList()).isEmpty() = true
5.  (aList.insert(i, item)).isEmpty() = false
6.  (aList.createList()).remove(i) = error
7.  (aList.insert(i, x)).remove(i) = aList
8.  (aList.createList()).retrieve(i) = error
9.  (aList.insert(i, x)).retrieve(i) = x
10. aList.retrieve(i) =
       (aList.insert(i, x)).retrieve(i+1)
11. aList.retrieve(i+1) = (aList.remove(i)).retrieve(i)
```

[3]The = notation within these axioms denotes algebraic equality.

A set of axioms does not make the pre- and postconditions for an ADT's operations unnecessary. For example, the previous axioms do not describe `insert`'s behavior when you try to insert an item into position 50 of a list of 2 items. One way to handle this situation is to include the restriction

```
1 <= index <= getLength()+1
```

in `insert`'s precondition. Another way—which you will see when we implement the ADT list later in this chapter—does not restrict `index`, but rather sets an argument `success` to `false` if `index` is outside the previous range. Thus, you need both a set of axioms and a set of pre- and postconditions to define the behavior of an ADT's operations completely.

Use axioms to determine the effect of a sequence of ADT operations

You can use axioms to determine the outcome of a sequence of ADT operations. For example, if `aList` is a list of characters, how does the sequence of operations

```
aList.insert(1, b)
aList.insert(1, a)
```

affect `aList`? We will show that `a` is the first item in this list and that `b` is the second item by using `retrieve` to retrieve these items.

You can write the previous sequence of operations in another way as

```
(aList.insert(1, b)).insert(1, a)
```

or

```
tempList.insert(1, a)
```

where `tempList` represents `aList.insert(1, b)`. Now retrieve the first and second items in the list `tempList.insert(1, a)`, as follows:

```
(tempList.insert(1, a)).retrieve(1) = a    by axiom 9
```

and

```
(tempList.insert(1, a)).retrieve(2)
  = tempList.retrieve(1)                   by axiom 10
  = (aList.insert(1, b)).retrieve(1)       by definition of tempList
  = b                                      by axiom 9
```

Thus, `a` is first item in the list and `b` is the second item.

Axioms are treated further in exercises in the rest of the book.

IMPLEMENTING ADTS

The previous sections emphasized the specification of an abstract data type. When you design an ADT, you concentrate on what its operations do, but you ignore how you will implement them. The result should be a set of clearly specified ADT operations.

How do you implement an ADT once its operations are clearly specified? That is, how do you store the ADT's data and carry out its operations? Earlier in this chapter you learned that when implementing an ADT, you choose data structures to represent the ADT's data. Thus, your first reaction to the implementation question might be to choose a data structure and then to write functions that access it in accordance with the ADT operations. Although this point of view is not incorrect, hopefully you have learned not to jump right into code. In general, you should refine an ADT through successive levels of abstraction. That is, you should use a top-down approach to designing an algorithm for each of the ADT operations. You can view each of the successively more concrete descriptions of the ADT as implementing its more abstract predecessors. The refinement process stops when you reach data structures that are available in your programming language. The more primitive your language, the more levels of implementation you will require.

The choices that you make at each level of the implementation can affect its efficiency. For now, our analyses will be intuitive, but Chapter 9 will introduce you to quantitative techniques that you can use to weigh the trade-offs involved.

Recall that the program that uses the ADT should see only a wall of available operations that act on data. Figure 3-8 illustrates this wall once again. Both the data structure that you choose to contain the data and the implementations of the ADT operations are hidden behind the wall. By now, you should realize the advantage of this wall.

In a non-object-oriented implementation, both the data structure and the ADT operations are distinct pieces. The client agrees to honor the wall by using only the ADT operations to access the data structure. Unfortunately, the data structure is hidden only if the client does not look over the wall! Thus, the client can violate the wall—either inten-

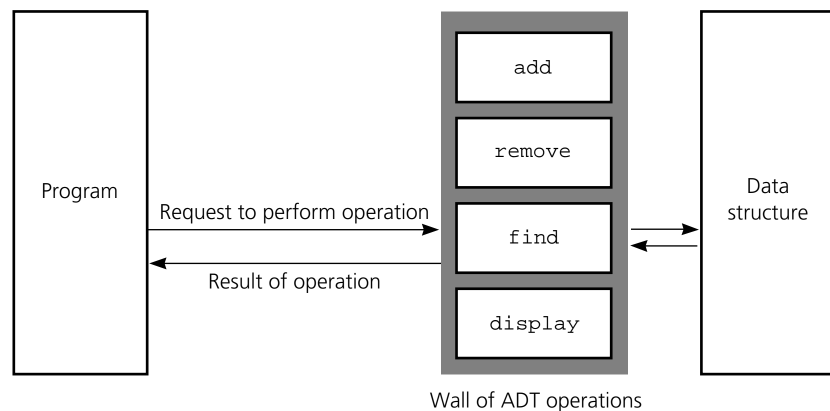


Figure 3-8

ADT operations provide access to a data structure

tionally or accidentally—by accessing the data structure directly, as Figure 3-9 illustrates. Why is such an action undesirable? Later, this chapter will use an array `items` to store an ADT list's items. In a program that uses such a list, you might, for example, accidentally access the first item in the list by writing

```
firstItem = items[0];
```

instead of by invoking `retrieve`. If you changed to another implementation of the list, your program would be incorrect. To correct your program, you would need to locate and change all occurrences of `items[0]`—but first you would have to realize that `items[0]` is in error!

Object-oriented languages such as C++ provide a way for you to enforce the wall of an ADT, thereby preventing access of the data structure in any way other than by using the ADT operations. We will spend some time now exploring this aspect of C++ by discussing classes, namespaces, and exceptions.

C++ Classes

Recall from Chapter 1 that object-oriented programming, or OOP, views a program not as a sequence of actions but as a collection of components called objects. Encapsulation—one of OOP's three fundamental principles[4]—enables you to enforce the walls of an ADT. It is, therefore, essential to an ADT's implementation and our main focus here.

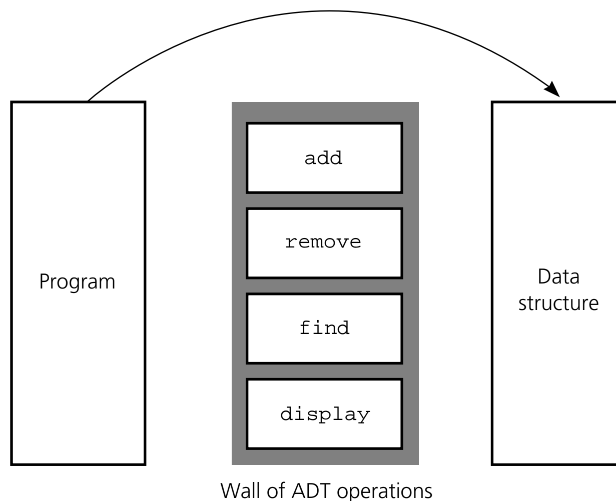


Figure 3-9

Violating the wall of ADT operations

[4] The other principles are inheritance and polymorphism, which Chapter 8 will discuss.

Encapsulation hides implementation details

Encapsulation combines an ADT's data with its operations—called **methods**—to form an **object.** Rather than thinking of the many components of the ADT in Figure 3-8, you can think at a higher level of abstraction when you consider the object in Figure 3-10 because it is a single entity. The object hides its inner detail from the programmer who uses it. Thus, an ADT's operations become an object's behaviors.

We could use a ball as an example of an object. Because thinking of a basketball, volleyball, tennis ball, or soccer ball probably suggests images of the game rather than the object itself, let's abstract the notion of a ball by picturing a sphere. A sphere of a given radius has attributes such as volume and surface area. A sphere as an object should be able to report its radius, volume, surface area, and so on. That is, the sphere object has methods that return such values. This section will develop the notion of a sphere as an object. Later, in Chapter 8, you will see how to derive a ball from a sphere.

A C++ class defines a new data type

An object is an instance of a class

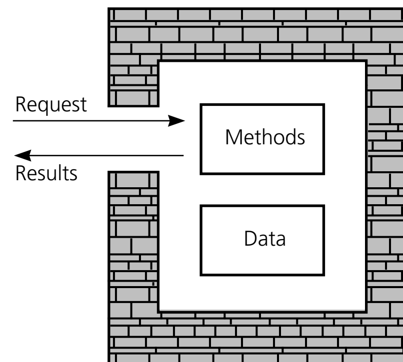
How do you actually define an object in C++? Recall that Chapter 1 introduced the **class** as a set of objects of the same kind. In C++, a class is actually a new data type whose instances are objects. The syntax for a class is like that for a C++ structure. Like a structure, a class can contain **data members.** You reference these data members in the same way that you reference the data members in a structure: You qualify a data member's name with an instance of the class. A class can also contain **member functions**, collectively known as class members which typically act on the data members. You invoke a member function by qualifying its name with the instance of the class in the same way that you reference data members.

By default, all members in a class are **private**—they are not directly accessible by your program—unless you designate them as **public.** The implementations of a class's member functions, however, can use any private members. In contrast, all members of a structure are public unless you designate them as private.

Since a C++ structure can contain member functions, a class and a structure have the same effect, if you explicitly designate both their public and private members. Although you could use `struct` and `class` interchangeably, you should not do so. A structure is appropriate when

Figure 3-10

An object's data and methods are encapsulated



you want to group data of different types, but you do not want to create a new data type. The structures in this book contain only data members and possibly one member function to initialize them. You should use a class to define a new data type when you implement an ADT. This book uses *class* only to define object types and will always indicate the public and private portions explicitly.

The ADTs that you saw earlier had operations for their creation and destruction. Classes have such methods, called **constructors** and **destructors.** A constructor creates and initializes new instances of a class. A destructor destroys an instance of a class, when the object's lifetime ends. A typical class has several constructors but only one destructor. For many classes, you can omit the destructor. In such cases, the compiler will generate a destructor for you. For the classes in this chapter, the **compiler-generated destructor** is sufficient. Chapter 4 will discuss how and why you would write your own destructor.

A constructor creates and initializes an object

A destructor destroys an object

In C++, a constructor has the same name as the class. Constructors have no return type—not even *void*—and cannot use *return* to return a value. Constructors can have arguments. We will discuss constructors in more detail shortly, after we look at an example of a class definition.

The header file. You should place each class definition in its own **header file** or **specification file**—whose name by convention ends in *.h*.[5] The following header file, *Sphere.h*, contains a class definition for sphere objects:

```
// *************************************************
// Header file Sphere.h for the class Sphere.
// *************************************************
const double PI = 3.14159;
class Sphere
{
public:
   Sphere();
   // Default constructor: Creates a sphere and
   // initializes its radius to a default value.
   // Precondition: None.
   // Postcondition: A sphere of radius 1 exists.

   Sphere(double initialRadius);
   // Constructor: Creates a sphere and initializes
   // its radius.
   // Precondition: initialRadius is the desired
   // radius.
   // Postcondition: A sphere of radius initialRadius
   // exists.
```

[5]Other conventions, such as *.hpp* or *.hxx*, exist.

```
   void setRadius(double newRadius);
   // Sets (alters) the radius of an existing sphere.
   // Precondition: newRadius is the desired radius.
   // Postcondition: The sphere's radius is newRadius.

   double getRadius() const;
   // Determines a sphere's radius.
   // Precondition: None.
   // Postcondition: Returns the radius.

   double getDiameter() const;
   // Determines a sphere's diameter.
   // Precondition: None.
   // Postcondition: Returns the diameter.

   double getCircumference() const;
   // Determines a sphere's circumference.
   // Precondition: PI is a named constant.
   // Postcondition: Returns the circumference.

   double getArea() const;
   // Determines a sphere's surface area.
   // Precondition: PI is a named constant.
   // Postcondition: Returns the surface area.

   double getVolume() const;
   // Determines a sphere's volume.
   // Precondition: PI is a named constant.
   // Postcondition: Returns the volume.

   void displayStatistics() const;
   // Displays statistics of a sphere.
   // Precondition: None.
   // Postcondition: Displays the radius, diameter,
   // circumference, area, and volume.

private:
   double theRadius;  // the sphere's radius
};  // end class
// End of header file.
```

A class's data members should be private

You should almost always place a class's data members within its private section. Typically, you provide methods—such as *setRadius* and *getRadius*—to access the data members. In this way, you control how and if the rest of the program can access the data members. This design principle should lead to programs that not only are easier to debug, but also have fewer logical errors from the beginning.

Some function declarations, such as

```
double getRadius() const;
```

are tagged with *const*. Such functions cannot alter the data members of the class. Making *getRadius* a *const* function is a fail-safe technique that ensures that it will only return the current value of the sphere's radius, without changing it.

***const* functions cannot change a class's data members**

A programmer who uses your class in a program usually sees only the header file, as you will soon learn. For this reason, your author prefers to place the documentation for the member functions in the header file and to place the class's public section before its private section.

Comments in the header file specify the member functions

Let's begin implementing the class *Sphere* by examining its constructors.

Constructors. A constructor allocates memory for an object and can initialize the object's data to particular values. A class can have more than one constructor, as is the case for the class *Sphere*.

The first constructor in *Sphere* is the **default constructor**

A default constructor has no arguments

```
Sphere();
```

A default constructor by definition has no arguments. Typically, a default constructor initializes data members to values that the class implementation chooses. For example, the implementation

```
Sphere::Sphere(): theRadius(1.0)
{
}  // end default constructor
```

sets *theRadius* to 1.0.

Notice the qualifier *Sphere::* that precedes the constructor's name. When you implement any member function, you qualify its name with its class type followed by the **scope resolution operator *::*** to distinguish it from other functions that might have the same name.

Although you could simply use an assignment statement to assign a value to *theRadius*, it is preferable to use an **initializer**—*theRadius(1.0)* in this case. Each initializer uses a functional notation that consists of a data member name followed by its initial value enclosed in parentheses. If you write more than one initializer,[6] you separate them with commas. A colon precedes the first (or only) initializer. Often the implementation of a constructor consists only of initializers, so its body is empty, as is the case here. Note that you can use these initializers with constructors but not with other member functions.

Use initializers within a constructor's implementation to set data members to initial values

Use initializers only within constructors

[6]When a class has several data members, the constructor initializes them in the order in which they appear in the class definition instead of the order in which the initializers appear in the constructor definition. You should use the same order in both cases to avoid confusion, even if the initialization order does not make a difference.

When you declare an instance of the class, a constructor is invoked implicitly. For example, the statement

```
Sphere unitSphere;
```

invokes the default constructor, which creates the object *unitSphere* and sets its radius to 1.0. Notice that you do not include parentheses after *unitSphere*.

The next constructor in *Sphere* is

```
Sphere(double initialRadius);
```

It creates a sphere object of radius *initialRadius*. This constructor needs only to initialize the private data member *theRadius* to *initialRadius*. Its implementation[7] is

```
Sphere::Sphere(double initialRadius):
                                   theRadius(initialRadius)
{
}  // end constructor
```

You implicitly invoke this constructor by writing a declaration such as

```
Sphere mySphere(5.1);
```

In this case, the object *mySphere* has a radius of 5.1.

If you omit all constructors from your class, the compiler will generate a default constructor—that is, one with no arguments—for you. A **compiler-generated default constructor**, however, might not initialize data members to values that you will find suitable.

If you define a constructor that has arguments, but you omit the default constructor, the compiler will not generate one for you. Thus, you will not be able to write statements such as

```
Sphere defaultSphere;
```

The implementation file. Typically, you place the implementation of a class's member functions in an **implementation file** whose name ends in *.cpp*.[8] An implementation file for *Sphere* follows. Notice within the definition of a member function, you can reference the class's data member or invoke its other member functions without preceding the member names with *Sphere::*

The implementation file contains the definitions of the class's member functions

```
// *********************************************************
// Implementation file Sphere.cpp for the class Sphere.
// *********************************************************
#include "Sphere.h"   // header file
#include <iostream.h>
```

[7]This implementation will be improved shortly to guard against a negative radius.

[8]Other conventions, such as *.c*, *.cpp*, and *.cxx*, exist.

```
Sphere::Sphere(): theRadius(1.0)
{
}  // end default constructor

Sphere::Sphere(double initialRadius)
{
   if (initialRadius > 0)
      theRadius = initialRadius;
   else
      theRadius = 1.0;
}  // end constructor

void Sphere::setRadius(double newRadius)
{
   if (newRadius > 0)
      theRadius = newRadius;
   else
      theRadius = 1.0;
}  // end setRadius

double Sphere::getRadius() const
{
   return theRadius;
}  // end getRadius

double Sphere::getDiameter() const
{
   return 2.0 * theRadius;
}  // end getDiameter

double Sphere::getCircumference() const
{
   return PI * getDiameter();
}  // end getCircumference

double Sphere::getArea() const
{
   return 4.0 * PI * theRadius * theRadius;
}  // end getArea

double Sphere::getVolume() const
{
   double radiusCubed = theRadius * theRadius * theRadius;
   return (4.0 * PI * radiusCubed)/3.0;
}  // end getVolume
```

A local variable such as `radiusCubed` should not be a data member

From within `display-Statistics`, you can invoke the member function `getRadius` or access the private data member `theRadius`

```
void Sphere::displayStatistics() const
{
   cout << "\nRadius = " << getRadius()
        << "\nDiameter = " << getDiameter()
        << "\nCircumference = " << getCircumference()
        << "\nArea = " << getArea()
        << "\nVolume = " << getVolume() << endl;
}  // end displayStatistics
// End of implementation file.
```

You should distinguish between a class's data members and any local variables that the implementation of a member function requires. It is inappropriate for such local variables to be data members of the class.

Using the Class *Sphere*. The following simple program demonstrates the use of the class *Sphere*:

```
#include <iostream.h>
#include "Sphere.h"

int main()
{
   Sphere unitSphere;          // radius is 1.0
   Sphere mySphere(5.1);       // radius is 5.0

   unitSphere.displayStatistics();
   mySphere.setRadius(4.2);    // resets radius to 4.2
   cout << mySphere.getDiameter() << endl;

   return 0;
}  // end main
```

An object such as *mySphere* can, upon request, reset the value of its radius; return its radius; compute its diameter, surface area, circumference, and volume; and display these statistics. These requests to an object are called **messages** and are simply calls to functions. Thus, an object responds to a message by acting on its data. To invoke an object's member function, you qualify the function's name—such as *setRadius*—with the object variable—such as *mySphere*.

Notice that the previous program included the header file *Sphere.h*, but did not include the implementation file *Sphere.cpp*.[9] You compile a class's implementation file separately from the program that uses the class. The way in which you tell the operating system where to locate the compiled implementation depends on the particular system.

[9]Appendix A provides more information about header and implementation files in the section "Libraries."

The previous program is an example of a **client** of a class. A client of a particular class is simply a program or module that uses the class. We will reserve the term **user** for the person who uses a program.

Inheritance. A brief discussion of inheritance is provided here, since it is a common way to create new classes in C++. A more complete discussion of inheritance appears in Chapter 8.

Suppose we want to create a class for colored spheres, knowing that we have already developed the class *Sphere*. We could write an entirely new class for the colored spheres, but if colored spheres are actually like spheres in the class *Sphere*, we can reuse the *Sphere* implementation and add color operations and characteristics by using inheritance. Here is a specification of the class *ColoredSphere* that uses inheritance:

*A class derived from the class **Sphere***

```
#include "Sphere.h"
enum Color {RED, BLUE, GREEN, YELLOW};
class ColoredSphere: public Sphere
{
public:
   ColoredSphere(Color initialColor);
   ColoredSphere(Color initialColor,
                 double initialRadius);
   void setColor(Color newColor);
   Color getColor();
private:
   Color c;
};  //end ColoredSphere class
```

The class *Sphere* is called the **base class** or **superclass**, and *ColoredSphere* is called the **derived class** or **subclass** of the class *Sphere*.

Any instance of the derived class is also considered to be an instance of the base class and can be used in a program anywhere that an instance of the base class can be used. Also, when the clause *public* is used with the base class, any of the publicly defined functions or data members that can be used with instances of the base class can be used with instances of the derived class. The derived class instances also have the functions and data members that are publicly defined in the derived class definition.

The implementation of the functions for the class *ColoredSphere* is as follows:

```
ColoredSphere::ColoredSphere(Color initialColor)
                            :Sphere()
{
   c = initialColor;
}  // end constructor
```

```
ColoredSphere::ColoredSphere(Color initialColor,
                             double initialRadius)
                            :Sphere(initialRadius)
{
   c = initialColor;
}  // end constructor

void ColoredSphere::setColor(Color newColor)
{
   c = newColor;
}  // end setColor

Color ColoredSphere::getColor()
{
   return c;
}  // end getColor
```

In the constructors for the `ColoredSphere` class, notice the use of the constructors `Sphere()` and `Sphere(initialRadius)`. Often, the derived class implementation will use the base class constructor in this manner and then add initializations that are specific to the derived class.

Here is a function that uses the `ColoredSphere` class:

An instance of a derived class can invoke public methods of the base class

```
void useColoredSphere()
{
   ColoredSphere ball(RED);
   ball.setRadius(5.0);
   cout << "The ball diameter is " << ball.getDiameter();
   ball.setColor(BLUE);
   ...
}  // end useColoredSphere
```

This function uses the constructor and the method `setColor` from the derived class `ColoredSphere`. It also uses the methods `setRadius` and `getDiameter` that are defined in the base class `Sphere`.

C++ Namespaces

Often, a solution to a problem will have groups of related classes and other declarations, such as functions, variables, types, and constants. C++ provides a mechanism for logically grouping these declarations and definitions into a common declarative region known as a **namespace**. You declare a namespace as follows:

```
namespace namespaceName
{
  // Place declarations here
}
```

The contents of the namespace can be accessed by code inside or outside the namespace. Inside the namespace, code can access elements directly. But accessing the same elements from outside the namespace requires special syntax. For example, suppose you have the following namespace, called *smallNamespace*:

```
namespace smallNamespace
{
  int count = 0;
  void abc();
} // end smallNamespace
```

A function declared in the namespace can be implemented directly in the namespace or can have its implementation appear elsewhere with the scope resolution operator specified. For example, here is the implementation of the function *abc*:

```
void smallNamespace::abc()
{
  // implementation
  . . .
} // end abc
```

You can access elements from outside the namespace *smallNamespace* by using the scope resolution operator. For example,

```
smallNamespace::count += 1;
smallNamespace::abc();
```

Since this syntax can become cumbersome when you're using many elements of the namespace, C++ also provides the ***using* declaration**. The *using* declaration allows the names of the elements to be used directly, without the scope resolution operator. For example, the previous code becomes:

```
using namespace smallNamespace;
count += 1;
abc();
```

A second form of the *using* declaration allows you to target specific elements in the namespace for the shortened notation. For example,

```
using smallNamespace::abc;
smallNamespace::count += 1;
abc();
```

This *using* declaration specifies that only the function *abc* can be accessed using the shortened notation. Access to the *count* variable still requires the scope resolution operator.

Items declared in the C++ Standard Library are declared in the namespace called *std*. When you wish to use elements in the Standard

Library with the shortened notation, you must include the following *using* declaration in your code:

```
using namespace std;
```

Most of the C++ *include* files have been updated in the latest version of the standard. Usually, the newer version has the same name as the older version, but without the *.h* extension. For example, to include the older specification in a program that contains many of the C++ input and output functions, you would use the following *include* statement:

```
#include <iostream.h>
```

To use the newer version of this file, you would write

```
#include <iostream>
using namespace std;
```

The *using namespace* statement indicates that you want to use the shortened notation.

When declarations are made outside of a namespace, they are said to exist in the **global namespace.** Most of the classes created in this text are declared in the global namespace for simplicity.

An Array-Based Implementation of the ADT List

We will now implement the ADT list as a class. Recall that the ADT list operations are

```
+createList()
+destroyList()
+isEmpty():boolean
+getLength():integer
+insert(in index:integer, in newItem:ListItemType,
       out success:boolean)
+remove(in index:integer, out success:boolean)
+retrieve(in index:integer,
         out dataItem:ListItemType,
         out success:boolean)
```

You need to represent the items in the ADT list and its length. Your first thought is probably to store the list's items in an array *items*. In fact, you might believe that the list is simply a fancy name for an array. This belief is not quite true, however. An **array-based implementation** is a natural choice because both an array and a list identify their items by number. However, the ADT list has operations such as *getLength* that an array does not. In the next chapter you will also see another implementation of the ADT list that does not use an array.

In any case, you can store a list's k^{th} item in *items[k-1]*. How much of the array will the list occupy? Possibly all of the array, but probably

not. That is, you need to keep track of the array elements that you have assigned to the list and those that are available for use in the future. The maximum length of the array—its **physical size**—is a known, fixed value such as *MAX_LIST*. You can keep track of the current number of items on the list—that is, the list's length or **logical size**—in a variable *size*. An obvious benefit of this approach is that implementing the operation *getLength* will be easy. Thus, we could use the following statements for the implementation:

```
const int MAX_LIST = 100;       // max length of list
typedef int ListItemType;       // data type of list items
ListItemType items[MAX_LIST];   // array of list items
int size;                       // length of list
```

Shift array elements to insert an item

Figure 3-11 illustrates the data members for an array-based implementation of an ADT list of integers. To insert a new item at a given position in the array of list items, you must shift to the right the items from this position on, and insert the new item in the newly created opening. Figure 3-12 depicts this insertion.

Now consider how to delete an item from the list. You could blank it out, but this strategy can lead to gaps in the array, as Figure 3-13a (page 146) illustrates. An array that is full of gaps has three significant problems:

- *size* – *1* is no longer the index of the last item in the array. You need another variable, *lastPosition*, to contain this index.
- Because the items are spread out, the function *retrieve* might have to look at every cell of the array even when only a few items are present.
- When *items[MAX_LIST - 1]* is occupied, the list could appear full, even when fewer than *MAX_LIST* items are present.

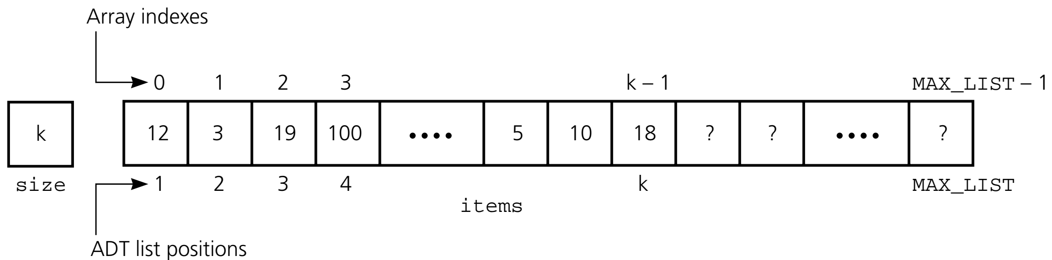


Figure 3-11

An array-based implementation of the ADT list

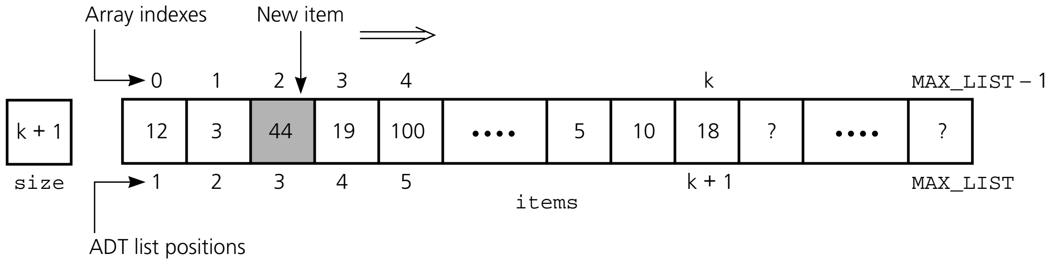


Figure 3-12

Shifting items for insertion at position 3

Shift array elements to delete an item

Implement the ADT list as a class

`items` and `size` are private data members

Thus, what you really need to do is shift the elements of the array to fill the gap left by the deleted item, as shown in Figure 3-13b.

You should implement each ADT operation as a function of a class. Each operation will require access to both the array *`items`* and the list's length *`size`*, so make *`items`* and *`size`* data members of the class. To hide *`items`* and *`size`* from the clients of the class, make these data members private. Although it is not obvious why at this point, it will be convenient to define the function *`translate(position)`*, which returns the index of the array element that contains the list item at posi-

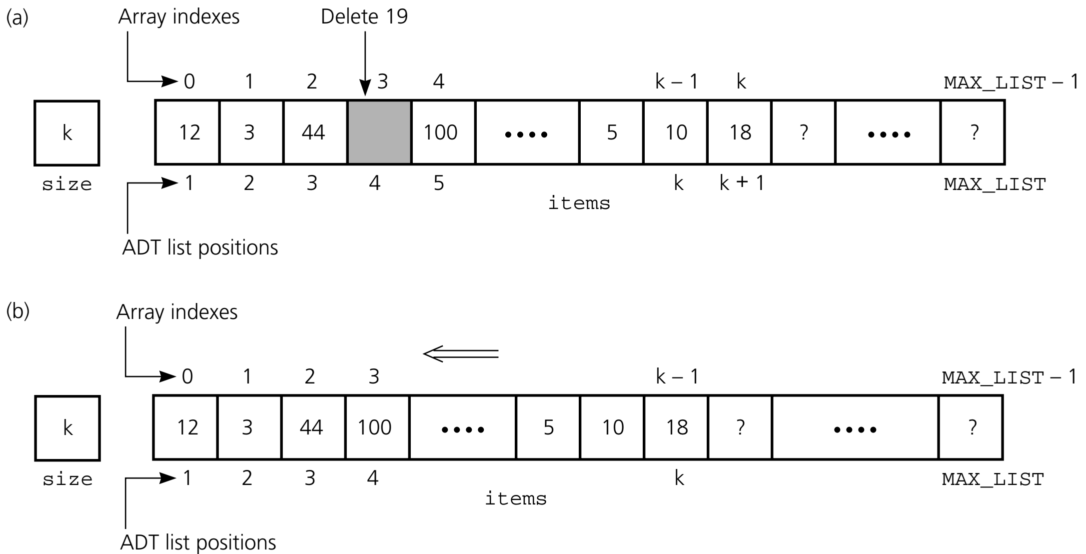


Figure 3-13

(a) Deletion causes a gap; (b) fill gap by shifting

tion *position*. That is, *translate(position)* returns the index value *position* – 1. Such a function is not one of the ADT operations and should not be available to the client. It simply makes the implementer's task easier. Thus, you should hide *translate* from the client by defining it within the class's private section.

***translate* is a private member function**

In the following header file *ListA.h* for the class of lists, the constructor corresponds to the ADT operation *createList*. A compiler-generated destructor—which corresponds to the ADT operation *destroyList*—is sufficient for this class, so we did not write one of our own.

```
// *********************************************************
// Header file ListA.h for the ADT list
// Array-based implementation
// *********************************************************
const int MAX_LIST = maximum-size-of-list;
typedef desired-type-of-list-item ListItemType;

class List
{
public:
   List();  // default constructor
            // destructor is supplied by compiler

// list operations:
   bool isEmpty() const;
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty;
   // otherwise returns false.

   int getLength() const;
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items
   // that are currently in the list.

   void insert(int index, ListItemType newItem,
               bool& success);
   // Inserts an item into the list at position index.
   // Precondition: index indicates the position at which
   // the item should be inserted in the list.
   // Postcondition: If insertion is successful, newItem is
   // at position index in the list, and other items are
   // renumbered accordingly, and success is true;
   // otherwise success is false.
```

```
   // Note: Insertion will not be successful if
   // index < 1 or index > getLength()+1.

   void remove(int index, bool& success);
   // Deletes an item from the list at a given position.
   // Precondition: index indicates where the deletion
   // should occur.
   // Postcondition: If 1 <= index <= getLength(),
   // the item at position index in the list is
   // deleted, other items are renumbered accordingly,
   // and success is true; otherwise success is false.

   void retrieve(int index, ListItemType& dataItem,
                 bool& success) const;
   // Retrieves a list item by position.
   // Precondition: index is the number of the item to
   // be retrieved.
   // Postcondition: If 1 <= index <= getLength(),
   // dataItem is the value of the desired item and
   // success is true; otherwise success is false.

private:
   ListItemType items[MAX_LIST];  // array of list items
   int          size;             // number of items in list

   int translate(int index) const;
   // Converts the position of an item in a list to the
   // correct index within its array representation.
};  // end List class
// End of header file.
```

The implementations of the functions that the previous header file declares appear in the following file *ListA.cpp*:

Implementation file

```
// *********************************************************
// Implementation file ListA.cpp for the ADT list
// Array-based implementation
// *********************************************************
#include "ListA.h"  //header file
List::List() : size(0)
{
}   // end default constructor
```

```
bool List::isEmpty() const
{
   return bool(size == 0);
}  // end isEmpty

int List::getLength() const
{
   return size;
} // end getLength

void List::insert(int index, ListItemType newItem,
                  bool& success)
{
   success = bool( (index >= 1) &&
                   (index <= size+1) &&
                   (size < MAX_LIST) );

   if (success)
   {  // make room for new item by shifting all items at
      // positions >= index toward the end of the
      // list (no shift if index == size+1)
      for (int pos = size; pos >= index; --pos)
         items[translate(pos+1)] = items[translate(pos)];

      // insert new item
      items[translate(index)] = newItem;
      ++size;  // increase the size of the list by one
   }  // end if
} // end insert

void List::remove(int index, bool& success)
{
   success = bool( (index >= 1) && (index <= size) );

   if (success)
   {  // delete item by shifting all items at positions >
      // index toward the beginning of the list
      // (no shift if index == size)
      for (int fromPosition = index+1;
               fromPosition <= size; ++fromPosition)
         items[translate(fromPosition-1)] =
                             items[translate(fromPosition)];
      --size;  // decrease the size of the list by one
   }  // end if

}  // end remove
```

```
void List::retrieve(int index, ListItemType& dataItem,
                    bool& success) const
{
   success = bool( (index >= 1) &&
                   (index <= size) );

   if (success)
      dataItem = items[translate(index)];
} // end retrieve

int List::translate(int index) const
{
   return index-1;
} // end translate
//  End of implementation file.
```

The following program segment demonstrates the use of the class *List*:

```
#include "ListA.h"    // ADT list operations
int main()
{
   List         aList;
   ListItemType dataItem;
   bool         success;

   aList.insert(1, 20, success);
   . . .
   aList.retrieve(1, dataItem, success);
   . . .
```

A client of the class cannot access the class's private members directly

Note that references within this program, such as *aList.size*, *aList.items[4]*, and *aList.translate(6)*, would be illegal because *size*, *items*, and *translate* are within the private portion of the class.

In summary, to implement an ADT, given implementation-independent specifications of the ADT operations, you first must choose a data structure to contain the data. Next, you define and implement a class within a header file. The ADT operations are public member functions within the class, and the ADT data are class members that are typically private. You then implement the class's member functions within an implementation file. The program that uses the class will be able to access the data only by using the ADT operations.

C++ Exceptions

An exception is a mechanism for handling an error during execution

Many programming languages, including C++, support **exceptions**, which are a mechanism for handling errors. If you detect an error during execution, you can **throw** an exception. The code that deals with the exception is said to **catch** or **handle** it.

Catching exceptions. To catch an exception, C++ provides ***try-catch* blocks**. You place a statement that might cause an exception within a *try* block. The *try* block must be followed by one or more *catch* blocks. Each *catch* block indicates a type of exception you want to handle. A *try* block can have many *catch* blocks associated with it, since even a single statement might cause more than one type of exception. In addition, the *try* block can contain many statements, any of which might cause an exception. Here is the general syntax for a *try* block:

Use a ***try*** *block for statements that can throw an exception*

```
try
{
  statement(s);
}
```

and here is the syntax for a *catch* block:

Use a ***catch*** *block for each type of exception that you handle*

```
catch (ExceptionClass identifier)
{
  statement(s);
}
```

When a statement in a *try* block causes an exception, the remainder of the *try* block is abandoned, and control passes to the statements in the *catch* block that correspond to the type of exception thrown. The statements in the *catch* block are then executed and, upon completion of the *catch* block, execution resumes at the point following the last *catch* block. If there is no applicable *catch* block for an exception, abnormal program termination usually occurs.

Note that if an exception occurs in the middle of a *try* block, the destructors of all objects local to that block are called. This ensures that all resources allocated in that block are released, even if the block is not completely executed.

Throwing exceptions. When you detect an error within a function, you can throw an exception by executing a statement with the following form:

Use a ***throw*** *statement to throw an exception*

```
throw ExceptionClass(stringArgument);
```

Here *ExceptionClass* is the type of exception you want to throw, and *stringArgument* is an argument to the *ExceptionClass* constructor that provides a more detailed description of what may have caused the exception. When a *throw* statement executes, the remaining code in the function does not execute, and the exception is propagated back to the point where the function was called. See Appendix A for a more complete description.

You can define your own exception class

You may find that the C++ Standard Library has an exception class already defined that suits the exception needs of your program. You may also want to define your own exception class. Usually, the C++ exception class *exception*, or one of its derived classes, is used as the base class for

the exception. This provides a standardized interface for working with exceptions. In particular, all of the exceptions in the C++ Standard Library have a member function *what* that returns a message describing the exception. You will need to use the *std* namespace if you base your class on the C++ exception class *exception*.

To indicate the exceptions that will be thrown by a function, you include a *throw* clause in the function's header as follows:

A function whose code can throw an exception

```
void myMethod(int x) throw(BadArgException, MyException)
{
  if (x == MAX)
      throw BadArgException("BadArgException: reason");
   // some code here
    ...
   throw MyException("MyException: reason");
}  // end myMethod
```

Including a *throw* statement in the function specification ensures that the function can throw only those exceptions. An attempt to throw any other exception will result in a runtime error.

An Implementation of the ADT List Using Exceptions

We will now implement the ADT list using exceptions. In the original implementation, the *success* flag was used to indicate whether or not the operation had succeeded. This implementation will use exceptions to indicate when an operation is unsuccessful.

The *List* class has two types of error scenarios that we will respond to by throwing an exception: an out-of-bounds list index and an attempt to insert into a full list. Attempting to delete or retrieve from an empty list will be treated as an out-of-bounds list index error.

The following defines the exception *ListIndexOutOfRangeException*, which will be used for the out-of-bounds list index error. It is based upon the more general *out_of_range* exception from the C++ Standard Library.

```
#include <stdexcept>
#include <string>
using namespace std;
class ListIndexOutOfRangeException: public out_of_range
{
public:
   ListIndexOutOfRangeException(const string & message = "")
                        : out_of_range(message.c_str())
   { }
};  // end ListIndexOutOfRangeException
```

The following defines the exception *ListException*, which will be used when the array storing the list becomes full:

```
#include <exception>
#include <string>
using namespace std;
class ListException: public exception
{
public:
   ListException(const string & message = "")
           : exception(message.c_str())
   { }
};  // end ListException
```

We can now specify the *List* class presented earlier in the chapter with these exceptions as follows:

```
// *********************************************************
// Header file ListAexcept.h for the ADT list
// Array-based implementation with exceptions
// *********************************************************
#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
const int MAX_LIST = maximum-size-of-list;
typedef desired-type-of-list-item ListItemType;

class List
{
public:
   List();  // default constructor
            // destructor is supplied by compiler

// list operations:
   bool isEmpty() const;
   // Exception: None.

   int getLength() const;
   // Exception: None.

   void insert(int index, ListItemType newItem)
        throw(ListIndexOutOfRangeException, ListException);
   // Exception: Throws ListIndexOutOfRangeException if
   // index < 1 or index > getLength()+1.
   // Exception: Throws ListException if newItem cannot be
   // placed in the list because the array is full.

   void remove(int index)
        throw(ListIndexOutOfRangeException);
   // Exception: Throws ListIndexOutOfRangeException if
   // index < 1 or index > getLength().
```

```
   void retrieve(int index, ListItemType& dataItem) const
        throw(ListIndexOutOfRangeException);
   // Exception: ListIndexOutOfRangeException if
   // index < 1 or index > getLength().
private:
   ListItemType items[MAX_LIST];  // array of list items
   int          size;             // number of items in list
   int translate(int index) const;
};  // end List class
// End of header file.
```

The implementation of *insert* is shown below. The implementation (with exceptions) of the *List* functions *remove* and *retrieve* is left as an exercise.

```
void List::insert(int index, ListItemType newItem)
{
   if (size >= MAX_LIST)
      throw ListException(
         "ListException: List full on insert");

   if (index >= 1 && index <= size+1)
   {
       for (int pos = size; pos >= index; --pos)
         items[translate(pos+1)] = items[translate(pos)];
      // insert new item
      items[translate(index)] = newItem;
      ++size;  // increase the size of the list by one
   }
   else  // index out of range
      throw ListIndexOutOfRangeException(
       "ListIndexOutOfRangeException: Bad index on insert");
   // end if
} // end insert
```

SUMMARY

1. **Data abstraction is a technique for controlling the interaction between a program and its data structures. It builds walls around a program's data structures, just as other aspects of modularity build walls around a program's algorithms. Such walls make programs easier to design, implement, read, and modify.**
2. **The specification of a set of data-management operations together with the data values upon which they operate define an abstract data type (ADT).**

3. The formal mathematical study of ADTs uses systems of axioms to specify the behavior of ADT operations.
4. Only after you have fully defined an ADT should you think about how to implement it. The proper choice of a data structure to implement an ADT depends both on the details of the ADT operations and on the context in which you will use the operations.
5. Even after you have selected a data structure as an implementation for an ADT, the remainder of the program should not depend on your particular choice. That is, you should access the data structure by using only the ADT operations. Thus, you hide the implementation behind a wall of ADT operations. To enforce the wall within C++, you define the ADT as a class, thus hiding the ADT's implementation from the program that uses the ADT.
6. An object encapsulates both data and operations on that data. In C++, objects are instances of a class, which is a programmer-defined data type.
7. A C++ class contains at least one constructor, which is an initialization method, and a destructor, which is a cleanup method that destroys an object when its lifetime ends.
8. If you do not define a constructor for a class, the compiler will generate a default constructor—that is, one without arguments—for you. If you do not define a destructor, the compiler will generate one for you. Although a compiler-generated destructor is sufficient for the classes in this chapter, Chapter 4 will tell you when you need to write your own.
9. Members of a class are private unless you designate them as public. The client of the class, that is, the program that uses the class, cannot use members that are private. However, the implementations of functions can use them. Typically, you should make the data members of a class private, and provide public functions to access some or all of the data members.
10. Because certain classes have applications in many programs, you should take steps to facilitate their use. You can define and implement a class within header and implementation files, which a program can include when it needs to use the class.
11. A namespace provides a mechanism for logically grouping related classes, functions, variables, types, and constants.
12. If you detect an error during execution, you can throw an exception. The code that deals with the exception is said to catch or handle it.

CAUTIONS

1. **After you design a class, try writing some code that uses your class before you commit to your design. Not only will you see whether your design works for the problem at hand, but also you will test your understanding of your own design and check the comments that document your specifications.**
2. **When you implement a class, you might discover problems with either your class design or your specifications. If these problems occur, change your design and specifications, try using the class again, and continue implementing. These comments are consistent with the discussion of software life cycle in Chapter 1.**
3. **A program should not depend upon the particular implementations of its ADTs. By using a class to implement an ADT, you encapsulate the ADT's data and operations. In this way, you can hide implementation details from the program that uses the ADT. In particular, by making the class's data members private, you can change the class's implementation without affecting the client.**
4. **By making a class's data members private, you make it easier to locate errors in a program's logic. An ADT—and hence a class—is responsible for maintaining its data. If an error occurs, you look at the class's implementation for the source of the error. If the client could manipulate this data directly because the data was public, you would not know where to look for errors.**
5. **When a member function does not alter the class's data members, make it a *`const`* function as a safeguard against an implementation error.**
6. **Variables that are local to a function's implementation should not be data members of the class.**
7. **If you define a constructor for a class but do not also define a default constructor, the compiler will not generate one for you. In this case, a statement such as**

   ```
   List myList;
   ```

   is illegal.
8. **An array-based implementation of an ADT restricts the number of items that you can store. Thus, the implementation should check whether the data structure has space available before inserting a new item, and the client should take appropriate action if the insertion is impossible.**
9. **An exception that is not handled by using a try-catch block may cause abnormal program termination.**

SELF-TEST EXERCISES

1. What is the significance of "wall" and "contract"? Why do these notions help you to become a better problem solver?
2. Write a pseudocode function *swap(aList, i, j)* that interchanges the items currently in positions *i* and *j* of a list. Define the function in terms of the ADT list operations, so that it is independent of any particular implementation of the list. Assume that the list, in fact, has items at positions *i* and *j*. What impact does this assumption have on your solution? (See Exercise 2.)
3. What grocery list results from the following sequence of ADT list operations?

```
aList.createList()
aList.insert(1, butter, success)
aList.insert(1, eggs, success)
aList.insert(1, milk, success)
```

4. Write specifications for a list whose insertion, deletion, and retrieval operations are at the end of the list.
5. Write preconditions and postconditions for each of the ADT sorted list operations.
6. Write a pseudocode function that creates a sorted list *sortedList* from the list *aList* by using the operations of the ADTs list and sorted list.
7. The specifications of the ADTs list and sorted list do not mention the case in which two or more items have the same value. Are these specifications sufficient to cover this case, or must they be revised?

EXERCISES

1. Consider an ADT list of integers. Write a function that computes the sum of the integers in the list *aList*. The definition of your function should be independent of the list's implementation.
2. Implement the function *swap*, as described in Self-Test Exercise 2, but remove the assumption that the ith and j^{th} items on the list exist. Add an argument *success* that indicates whether the swap was successful.
3. Use the function *swap* that you wrote in Exercise 2 to write a function that reverses the order of the items in a list *aList*.
4. The section "The ADT List" describes the functions *displayList* and *replace*. As given in this chapter, these operations exist outside of the ADT, that is, they are not ADT list operations. Instead, their implementations are written in terms of the ADT list operations.
   a. What is an advantage and a disadvantage of the way that *displayList* and *replace* are implemented?

b. What is an advantage and a disadvantage of adding the operations `displayList` and `replace` to the ADT list?

5. In mathematics, a **set** is a group of distinct items. Specify operations such as equality, subset, union, and intersection as a part of the ADT set.
6. Specify operations that are a part of the ADT character string. Include typical operations such as length computation and concatenation (appending one string to another).
7. Write a pseudocode function in terms of the ADT appointment book, described in the section "Designing an ADT," for each of the following tasks:

   a. Change the purpose of the appointment at a given date and time.

   b. Display all the appointments for a given date.

   Do you need to add operations to the ADT to perform these tasks?
8. Consider the ADT polynomial—in a single variable *x*—whose operations include the following:

```
degree()
// Returns the degree of a polynomial.
coefficient(power)
// Returns the coefficient of the x^power term.
changeCoefficient(newCoefficient, power)
// Replaces the coefficient of the x^power term
// with newCoefficient.
```

For this problem, consider only polynomials whose exponents are nonnegative integers. For example,

$p = 4x^5 + 7x^3 - x^2 + 9$

The following examples demonstrate the ADT operations on this polynomial.

`p.degree()` is 5 (the highest power of a term with a nonzero coefficient)

`p.coefficient(3)` is 7 (the coefficient of the x^3 term)

`p.coefficient(4)` is 0 (the coefficient of a missing term is implicitly 0)

`p.changeCoefficient(-3, 7)` produces the polynomial

$p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$

Using these ADT operations, write statements to perform the following tasks:

a. Display the coefficient of the term that has the highest power.

b. Increase the coefficient of the x^3 term by 8.

c. Compute the sum of two polynomials.

9. Write pseudocode implementations of the ADT polynomial operations, as defined in Exercise 8, in terms of the ADT list operations.
10. Imagine an unknown implementation of an ADT sorted list of integers. This ADT organizes its items into ascending order. Suppose that you have just read *n* integers into a one-dimensional array of integers called `data`. Write some C++ statements that use the ADT sorted list operations to sort the array into ascending order.
11. Use the axioms for the ADT list, as given in this chapter in the section "Axioms," to prove that the sequence of operations

    ```
    Insert A into position 2
    Insert B into position 2
    Insert C into position 2
    ```

    has the same effect on a nonempty list of characters as the sequence

    ```
    Insert C into position 2
    Insert B into position 3
    Insert A into position 4
    ```

12. Define a set of axioms for the ADT sorted list and use them to prove that the sorted list of characters, which is defined by the sequence of operations

    ```
    Create an empty sorted list
    Insert S
    Insert T
    Insert R
    Delete T
    ```

    is exactly the same as the sorted list defined by the sequence

    ```
    Create an empty sorted list
    Insert T
    Insert R
    Delete T
    Insert S
    ```

13. Repeat Exercise 17 in Chapter 2, using a variation of the ADT list to implement the function $f(n)$.
14. Write pseudocode that merges two sorted lists into a new third sorted list by using only ADT sorted list operations.
15. Implement the `List` functions `retrieve` and `remove` to use exceptions.

PROGRAMMING PROBLEMS

1. Design and implement an ADT that represents a triangle. The data for the ADT should include the three sides of the triangle but could also include the triangle's three angles. This data should be in the private section of the class that implements the ADT.

   Include at least two initialization operations: One that provides default values for the ADT's data, and another that sets this data to client-supplied values. These operations are the class's constructors.

   The ADT also should include operations that look at the values of the ADT's data; change the values of the ADT's data; compute the triangle's area; and determine whether the triangle is a right triangle, an equilateral triangle, or an isosceles triangle.

2. Design and implement an ADT that represents the time of day. Represent the time as hours and minutes on a 24-hour clock. The hours and minutes are the private data members of the class that implements the ADT.

   Include at least two initialization operations: One that provides a default value for the time, and another that sets the time to a client-supplied value. These operations are the class's constructors.

   Include operations that set the time, increase the present time by a number of minutes, and display the time in 12-hour and 24-hour notations.

3. Design and implement an ADT that represents a calendar date. You can represent a date's month, day, and year as integers (for example, 4/1/2002). Include operations that advance the date by one day and display the date by using either numbers or words for the months. As an enhancement, include the name of the day.

4. Design and implement an ADT that represents a price in U.S. currency as dollars and cents. After you complete the implementation, write a client function that computes the change due a customer who pays *x* for an item whose price is *y*.

5. Define a class for an array-based implementation of the ADT sorted list. Consider a recursive implementation for `locatePosition`. Should `sortedInsert` and `sortedRemove` call `locatePosition`?

6. Write recursive array-based implementations of the insertion, deletion, and retrieval operations for the ADTs list and sorted list.

7. Implement the ADT set that you specified in Exercise 5 by using only arrays and simple variables.

8. Implement the ADT character string that you specified in Exercise 6.

9. Implement the ADT polynomial that Exercise 8 describes.

10. Implement the ADT appointment book, described in the section "Designing an ADT." Add operations as necessary. For example, you should add operations to read and write appointments.

11. **a.** Specify and implement an ADT for rational numbers. Provide operations that read, write, add, subtract, multiply, and divide fractions. The results of all arithmetic operations should be in lowest terms, so include a private function *reduceToLowestTerms*. Exercise 20 in Chapter 2 will help you with the details of this function. (Should your read and write operations call *reduceToLowestTerms*?) To simplify the determination of a fraction's sign, you can assume that the denominator of the fraction is positive.

    b. Specify and implement an ADT for mixed numbers, each of which contains an integer portion and a fractional portion in lowest terms. Assume the existence of the ADT fraction (see part a). Provide operations that read, write, add, subtract, multiply, and divide mixed numbers. The results of all arithmetic operations should have fractional portions that are in lowest terms. Also include an operation that converts a fraction to a mixed number.

    c. Implement the ADT recipe book as described in the section "Designing an ADT" and, in doing so, implement the ADT measurement. Add operations as necessary. For example, you should add operations to recipe book to read, write, and scale recipes.

12. Repeat Programming Problem 2 of Chapter 1 in light of your knowledge of ADTs and classes.

13. Repeat Programming Problem 3 of Chapter 1 in light of your knowledge of ADTs and classes.