of the List type. A data type that is designed to hold other objects is called a *container* or *collection type*. Moving up a level, we might see the abstract view of List as the implementation level of another programmer-defined data type, `ProductInventory`, and so on.

Perspectives on Data

Application or user view	Logical or abstract view	Implementation view
Product inventory	List	Array
List	Array	Row major access function
Array	Row major access function	32-Bit words on IBM Power PC

What do we gain by separating these views of the data? First, we reduce complexity at the higher levels of the design, making the program easier to understand. Second, we make the program more easily modifiable: The implementation can be completely changed without affecting the program that uses the data structure. We take advantage of this ability in this text, developing various implementations of the same objects in different chapters. Third, we develop software that is *reusable*: The structure and its accessing operations can be used by other programs, for completely different applications, as long as the correct interfaces are maintained. You saw in Chapter 1 that the design, implementation, and verification of high-quality computer software is a very laborious process. Being able to reuse pieces that are already designed, coded, and tested cuts down on the amount of work required.

In the chapters that follow, we extend these ideas to build other container classes that C++ does not provide: lists, stacks, queues, priority queues, trees, graphs, and sets. We consider these data structures from the logical view: What is our abstract picture of the data, and what accessing operations can we use to create, assign, and manipulate elements in the data structure? We express our logical view as an abstract data type (ADT) and record its description in a data specification.

Next, we take the application view of the data, using an instance of the data type in a short example.

Finally, we change hats and turn to the implementation view of the data type. We consider the C++ type declarations that represent the data structure as well as the design of the functions that implement the specifications of the abstract view. Data structures can be implemented in more than one way, so we often look at alternative representations and methods for comparing them. In some of the chapters, we include a longer Case Study in which instances of the data type are used to solve a problem.

Exercises

1. Explain what we mean by "data abstraction."
2. What is data encapsulation? Explain the programming goal "to protect our data abstraction through encapsulation."
3. Name three perspectives from which we can view data. Using the logical data structure "a list of student academic records," give examples of what each perspective might tell us about the data.
4. Consider the abstract data type GroceryStore.
   a. At the application level, describe GroceryStore.
   b. At the logical level, what grocery store operations might be defined for the customer?
   c. Specify (at the logical level) the operation CheckOut.
   d. Write an algorithm (at the implementation level) for the operation CheckOut.
   e. Explain how parts (c) and (d) represent information hiding.
5. What composite types are predefined in the C++ language?
6. Describe the component selectors for structs and classes at the logical level.
7. Describe the accessing functions for structs and classes at the implementation level.
8. Describe the component selectors for one-dimensional arrays at the logical level.
9. Describe the accessing functions for one-dimensional arrays at the implementation level.
10. a. Declare a one-dimensional array, `name`, that contains 20 characters.
    b. If each character occupies one "cell" in memory, and the base address of `name` is 1000, what is the address of the cell referenced in the following statement?

```
name[9] = 'A';
```

Use the following declarations for Exercises 11 and 12:

```
enum MonthType {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
                OCT, NOV, DEC};
struct WeatherType
{
  int avgHiTemp;
  int avgLoTemp;
  float actualRain;
  float recordRain;
};
```

Assume that an `int` requires one cell in memory, that a `float` number requires two cells, and that the struct members are found in contiguous memory locations with no gaps.

11. a. Declare a one-dimensional array type, `weatherListType`, of `WeatherType` components, to be indexed by values of type `MonthType`. Declare a variable, `yearlyWeather`, of `weatherListType`.
    b. Assign the value 1.05 to the actual rainfall member of the July record in `yearlyWeather`.
    c. If the base address of `yearlyWeather` is 200, what is the address of the member that you assigned in part (b)?
12. a. Declare a two-dimensional array, `decadeWeather`, of `WeatherType` components, to be indexed by values of type `MonthType` in the first dimension.
    b. Draw a picture of `decadeWeather`.
    c. Assign the value 26 to the `avgLoTemp` member of the March 2006 entry.
13. a. Define a three-dimensional array at the logical level.
    b. Suggest some applications for three-dimensional arrays.

Use the following declarations for Exercises 14–16.

```
typedef char String[10];
struct StudentRecord
{
  String firstName;
  String lastName;
  int id;
  float gpa;
  int currentHours;
  int totalHours;
};
StudentRecord student;
StudentRecord students[100];
```

Assume that an `int` requires one cell in memory, that a `float` number requires two cells, and that the struct members are found in contiguous memory locations with no gaps.

14. Construct a member-length-offset table for `StudentRecord`.
15. If the base address of `student` is 100, what address does the compiler generate as the target of the following assignment statement?

```
student.gpa = 3.87;
```

16. How much space does the compiler set aside for `students`?

17. Indicate which predefined C++ types would most appropriately model each of the following (more than one may be appropriate for each):
    a. a chessboard
    b. information about a single product in an inventory-control program
    c. a list of famous quotations
    d. the casualty figures (number of deaths per year) for highway accidents in Texas from 1995 to 2005
    e. the casualty figures for highway accidents in each of the states from 1995 to 2005
    f. the casualty figures for highway accidents in each of the states from 1995 to 2005, subdivided by month
    g. an electronic address book (name, address, and phone information for all your friends)
    h. a collection of hourly temperatures for a 24-hour period
18. What C++ construct is used to represent abstract data types?
19. Explain the difference between a C++ struct and class.
20. How is the client prevented from directly accessing the details of an instance of a class?
21. a. The details of a private member can be *seen* by the user of a class. (True or False?)
    b. The details of a private member may be *accessed* by a client program. (True or False?)
22. Why is it good practice to put a class declaration in one file and the implementation in another?
23. Name three ways that classes can relate to each other.
24. Distinguish between composition and inheritance.
25. Distinguish between a base class and a derived class.
26. Does a derived class have access to the private data members of the base class?
27. Does a derived class have access to the public member functions of the base class?
28. a. Write the specification for an ADT SquareMatrix. (A square matrix can be represented by a two-dimensional array with *N* rows and *N* columns.) You may assume a maximum size of 50 rows and columns. Include the following operations:

    `MakeEmpty(n)`, which sets the first `n` rows and columns to zero

    `StoreValue(i, j, value)`, which stores value into the [i, j] position

    `Add`, which adds two matrices together

    `Subtract`, which subtracts one matrix from another

    `Copy`, which copies one matrix into another

b. Convert your specification to a C++ class declaration.

c. Implement the member functions.

d. Write a test plan for your class.

29. `DateType` keeps only the integer representation of the month, day, and year. When a month is wanted in string form, the string is calculated. An alternate approach would be to add a string field to the date and calculate and store the string representation in the `Initialize` function. Which methods would have to be changed? Would this change make the use of an *if* statement to find the appropriate string more or less attractive? Write the code for this *if* statement.

30. What changes would be necessary if the number of days in the month were carried as a data field in class `DateType` rather than being looked up when necessary? Would this change make the use of a *switch* statement to find the number of days in the month more attractive? Write the code for this `switch` statement.

31. Compare and contrast the implementation of class `DateType` used in the case study and the solution proposed in Exercises 29 and 30. These two approaches represent the classic trade-off between space and algorithm complexity. Please comment.

chapter 3

ADT Unsorted List

After studying this chapter, you should be able to

- Describe the Abstract Data Type Unsorted List from three perspectives
- Use the Unsorted List operations to implement utility routines to do the following application-level tasks:
  - Print the list of elements
  - Create a list of elements from a file
- Implement the following Unsorted List operations using an array-based implementation
  - Create and destroy a list
  - Determine whether the list is full
  - Insert an element
  - Retrieve an element
  - Delete an element
- Write and execute a test plan for an abstract data type
- Declare variables of pointer types
- Access the variables to which pointers point
- Implement the list operations outlined above using a linked implementation
- Compare the two implementations of the ADT Unsorted List in terms of Big-O approximations