Table 5.3 *Output from one run of simulation program*

Simulation					
Simulation Length	Number of Servers	Average Transaction	Time Between Arrivals	Average Wait	Jobs Left
100	1	5	3	18.0	14
500	1	5	3	106.65	58
1000	1	5	3	202.55	132
5000	1	5	3	1042.11	686
100	2	5	3	2.19	3
500	2	5	3	1.41	1
1000	2	5	3	3.89	0
5000	2	5	3	2.99	2
100	3	5	3	0.09	0
500	3	5	3	0.28	0
1000	3	5	3	0.28	1
5000	3	5	3	0.22	0
100	4	5	3	0.00	0
500	4	5	3	0.03	0
1000	4	5	3	0.01	0
5000	4	5	3	0.01	0

All times are expressed in clock units. Average wait time reflects the wait time of jobs served; it does not reflect the wait time of jobs that were not yet served. Jobs Left reflects the number of jobs that were waiting to be served when the simulation time limit was reached.

Summary

We have defined a stack at the logical level as an abstract data type, discussed its use in an application, and presented an implementation encapsulated in a class. Though our logical picture of a stack is a linear collection of data elements with the newest element (the top) at one end and the oldest element at the other end, the physical representation of the stack class does not have to recreate our mental image. The implementation of the stack class must support the last in, first out (LIFO) property; how this property is supported, however, is another matter. For instance, the Push operation could "time stamp" the stack elements and put them into an array in any order. To pop, we would have to search the array, looking for the newest time stamp. This representation is very different from the stack implementation we developed in this chapter, but to the user of the stack class they are functionally equivalent. The implementation is transparent to the program that uses the stack because the stack is encapsulated by the operations in the class that surrounds it.

We also examined the definition and operations of a queue. We discussed some of the design considerations encountered when an array is used to contain the elements of a queue. Though the array itself is a random-access structure, our logical view of the queue as a structure limits us to accessing only the elements in the front and rear positions of the queue stored in the array. We used two pointers, one to the front and one to the rear, for the linked implementation.

There usually is more than one functionally correct design for the same data structure. When multiple correct solutions exist, the requirements and specifications of the problem may determine which solution is the best design.

In the design of data structures and algorithms, you find that there are often tradeoffs. A more complex algorithm may result in more efficient execution; a solution that takes longer to execute may save memory space. As always, we must base our design decisions on what we know about the problem's requirements.

Exercises

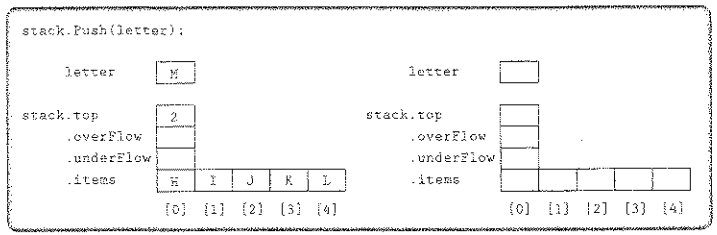
1. Indicate whether a stack would be a suitable data structure for each of the following applications.
   a. A program to evaluate arithmetic expressions according to the specific order of operators.
   b. A bank simulation of its teller operation to see how waiting times would be affected by adding another teller.
   c. A program to receive data that are to be saved and processed in the reverse order.
   d. An address book to be maintained.
   e. A word processor to have a PF key that causes the preceding command to be redisplayed. Every time the user presses the PF key, the program shows the command that preceded the one currently displayed.
   f. A dictionary of words used by a spelling checker to be built and maintained.
   g. A program to keep track of patients as they check into a medical clinic, assigning patients to doctors on a first come, first served basis.
   h. A data structure used to keep track of the return addresses for nested functions while a program is running.
2. Describe the accessing protocol of a stack at the abstract level.
3. Show what is written by the following segments of code, given that `item1`, `item2`, and `item3` are `int` variables.

a.
```
StackType<int> stack;
item1 = 1;
item2 = 0;
item3 = 4;
stack.Push(item2);
stack.Push(item1);
stack.Push(item1 + item3);
item2 = stack.Top();
stack.Pop();
stack.Push(item3*item3);
stack.Push(item2);
stack.Push(3);
item1 = stack.Top();
stack.Pop();
cout << item1 << endl << item2 << endl << item3
     << endl;
while (!stack.IsEmpty())
{
  item1 = stack.Top();
  stack.Pop();
  cout << item1 << endl;
}
```

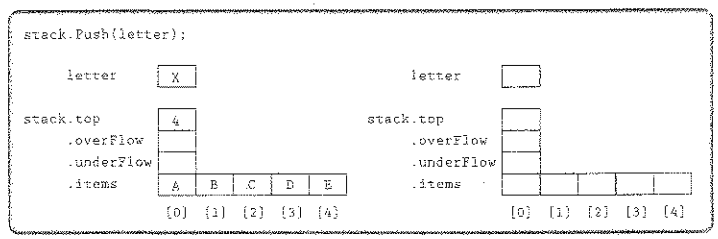
b.
```
StackType<int> stack;
item1 = 4;
item3 = 0;
item2 = item1 + 1;
stack.Push(item2);
stack.Push(item2 + 1);
stack.Push(item1);
item2 = stack.Top();
stack.Pop();
item1 = item2 + 1;
stack.Push(item1);
stack.Push(item3);
while (!stack.IsEmpty())
{
  item3 = stack.Top();
  cout << item3 << endl;
  stack.Pop();
}
cout << item1 << endl << item2 << endl << item3 << endl;
```


Use the following information for Exercises 4–7. The stack is implemented as a class containing an array of items, a data member indicating the index of the last item put on the stack (`top`), and two Boolean data members, `underFlow` and `overFlow`. The stack items are characters and `MAX_ITEM` is 5. In each exercise, show the result of the operation on the stack. Put a *T* or *F* for true or false, respectively, in the Boolean data members.

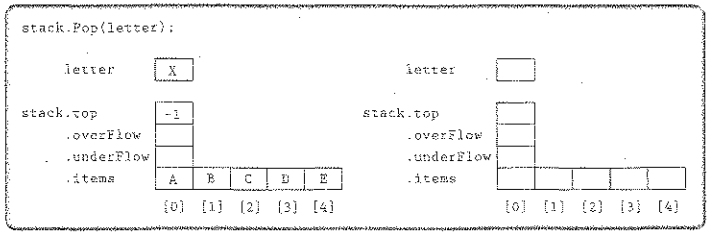
4. `stack.Push(letter);`



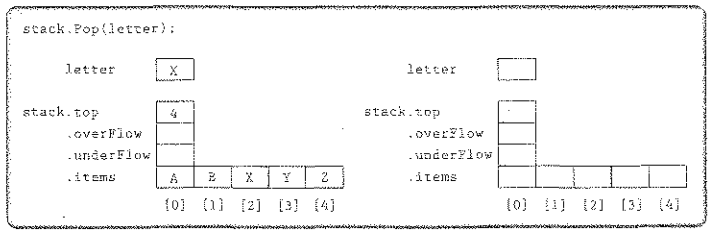
5. `stack.Push(letter);`



6.
```
(letter = stack.Top());
stack.Pop();
```



7.
```
(letter = stack.Top());
stack.Pop();
```



8. Write a segment of code to perform each of the following operations. You may call any of the member functions of `StackType`. The details of the stack type are encapsulated; you may use only the stack operations in the specification to perform the operations. (You may declare additional stack objects).
   a. Set `secondElement` to the second element in the stack, leaving the stack without its original top two elements.
   b. Set `bottom` equal to the bottom element in the stack, leaving the stack empty.
   c. Set `bottom` equal to the bottom element in the stack, leaving the stack unchanged.
   d. Make a copy of the stack, leaving the stack unchanged.
9. (Multiple choice) The statement

```
stack.items[0] = stack.items[1];
```

(setting the top element equal to the second element) in a client program of the stack class
   a. would cause a syntax error at compile time.
   b. would cause a run-time error.
   c. would not be considered an error by the computer, but would violate the encapsulation of the stack data type.
   d. would be a perfectly legal and appropriate way to accomplish the intended task.
10. (Multiple choice) The statements

```
stack.Push(item1 + 1);
stack.Pop(item1 + 1);
```

in the client program
   a. would cause a syntax error at compile time.
   b. would cause a run-time error.
   c. would be legal, but would violate the encapsulation of the stack.
   d. would be perfectly legal and appropriate.
11. Given the following specification of the `Top` operation:

ItemType Top

Function:	Returns a copy of the last item put onto the stack.
Precondition:	Stack is not empty.
Postconditions:	Function value = copy of item at top of stack.
	Stack is not changed.

Assume `Top` is not a stack operation and `Pop` returns the item removed. Write this function as client code, using operations from the nontemplate version of the `StackType` class. Remember—the client code has no access to the private members of the class.

12. Two stacks of positive integers are needed, one containing elements with values less than or equal to 1,000 and the other containing elements with values larger than 1,000. The total number of elements in the small-value stack and the large-value stack combined are not more than 200 at any time, but we cannot predict how many are in each stack. (All of the elements could be in the small-value stack, they could be evenly divided, both stacks could be empty, and so on.) Can you think of a way to implement both stacks in one array?
   a. Draw a diagram of how the stack might look.
   b. Write the definitions for such a double-stack structure.
   c. Implement the `Push` operation; it should store the new item into the correct stack according to its value (compared to 1,000).

13. A stack of integer elements is implemented as an array. The index of the top element is kept in position 0 in the array, and the stack elements are stored in `stack[1]..stack[stack[0]]`.
    a. How does this implementation fare when assessed against the idea of an array as a homogeneous collection of data elements?
    b. How would this implementation change the stack specifications? How would it change the implementations of the functions?
14. Using one or more stacks, write a code segment to read in a string of characters and determine whether it forms a palindrome. A palindrome is a sequence of characters that reads the same both forward and backward—for example: ABLE WAS I ERE I SAW ELBA.

    The character '.' ends the string. Write a message indicating whether the string is a palindrome. You may assume that the data are correct and that the maximum number of characters is 80.
15. Write the body for a function that replaces each copy of an item in a stack with another item. Use the following specification. (This function is in the *client* program.)

ReplaceItem(StackType& stack, ItemType oldItem, ItemType newItem)

Function:	Replaces all occurrences of oldItem with newItem.
Precondition:	stack has been initialized.
Postcondition:	Each occurrence of oldItem in stack has been replaced by newItem.

You may use any of the member functions of the nontemplate version of `StackType`, but you may not assume any knowledge of the stack's implementation.

16. In each plastic container of Pez candy, the colors are stored in random order. Your little brother likes only the yellow ones, so he painstakingly takes out all the candies, one by one, eats the yellow ones, and keeps the others in order, so that he can return them to the container in exactly the same order as before—minus the yellow candies, of course. Write the algorithm to simulate this process. You may use any of the stack operations defined in the Stack ADT, but may not assume any knowledge of the stack's implementation.
17. The specifications for the Stack ADT have been changed. The class representing the stack must now check for overflow and underflow and sets an error flag (a parameter) to true if either occurs.
    a. Rewrite the specifications incorporating this change.
    b. What new data members must you add to the class?

18. Implement the following specification for a client Boolean function that returns true if two stacks are identical and false otherwise.

Boolean Identical(StackType stack1, StackType stack2)

Function:	Determines if two stacks are identical.
Preconditions:	stack1 and stack2 have been initialized.
Postconditions:	stack1 and stack2 are unchanged.
	Function value = (stack1 and stack2 are identical)

You may use any of the member functions of `StackType`, but you may not assume any knowledge of the stack's implementation.

The following code segment (used for Exercises 19 and 20) is a count-controlled loop going from 1 through 5. At each iteration, the loop counter is either printed or put on a stack depending on the result of the Boolean function `RanFun()`. (The behavior of `RanFun()` is immaterial.) At the end of the loop, the items on the stack are popped and printed. Because of the logical properties of a stack, this code segment cannot print certain sequences of the values of the loop counter. You are given an output and asked to determine whether the code segment could generate the output.

```
for (count = 1; count <= 5; count++)
  if (RanFun())
    cout << count;
  else
    stack.Push(count);
while (!stack.IsEmpty())
{
   number = stack.Top();
   stack.Pop();
   cout << number;
}
```

19. The following output is possible using a stack: `1 3 5 2 4`
    a. True
    b. False
    c. Not enough information to determine
20. The following output is possible using a stack: `1 3 5 4 2`
    a. True
    b. False

21. Describe the accessing protocol of a queue at the abstract level.
22. Show what is written by the following segments of code, given that `item1`, `item2`, and `item3` are `int` variables.

a.

```
QueType queue;

item1 = 1;
item2 = 0;
item3 = 4;
queue.Enqueue(item2);
queue.Enqueue(item1);
queue.Enqueue(item1 + item3);
queue.Dequeue(item2);
queue.Enqueue(item3*item3);
queue.Enqueue(item2);
queue.Enqueue(3);
queue.Dequeue(item1);
cout  << item1  << endl  << item2  << endl  << item3
      << endl;
while (!queue.IsEmpty())
{
   queue.Dequeue(item1);
   cout << item1 << endl;
}
```

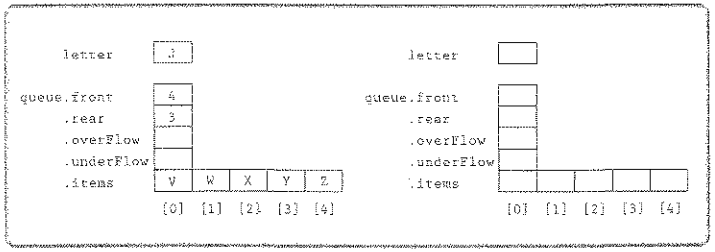
b.

```
QueType queue;
item1 = 4;
item3 = 0;
item2 = item1 + 1;
queue.Enqueue(item2);
queue.Enqueue(item2 + 1);
queue.Enqueue(item1);
queue.Dequeue(item2);
item1 = item2 + 1;
queue.Enqueue(item1);
queue.Enqueue(item3);
while (!queue.IsEmpty())
{
  queue.Dequeue(item3);
  cout << item3  << endl;
}
cout << item1 << endl  << item2  << endl  << item3  << endl;
```

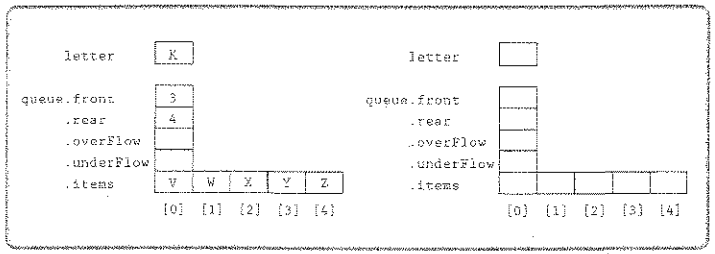

23. The specifications for the Queue ADT have been changed. The class representing the queue must now check for overflow and underflow and set an error flag (a parameter) to true if either occurs.
    a. Rewrite the specifications incorporating this change.
    b. What new data members must you add to the class?
    c. What new member functions must you add to the class?

Use the following information for Exercises 24–29. The queue is implemented as a class containing an array of items, a data member indicating the index of the last item put on the queue (`rear`), a data member indicating the index of the location before the first item put on the queue (`front`), and two Boolean data members, `underFlow` and `overFlow`, as discussed in this chapter. The item type is `char` and `maxQue` is 5. For each exercise, show the result of the operation on the queue. Put a *T* or *F* for true or false, respectively, in the Boolean data members.

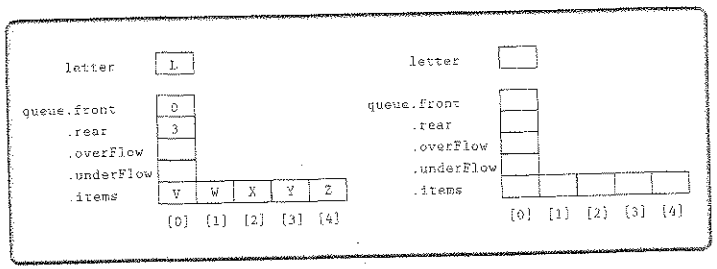
24. `queue.Enqueue(letter);`



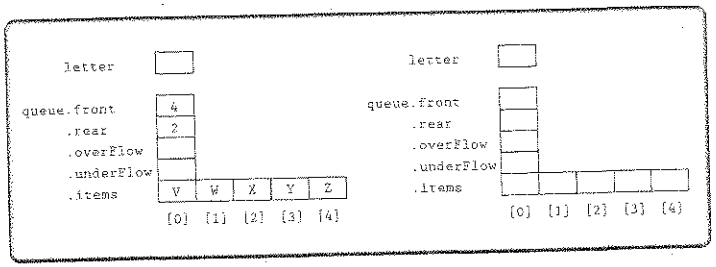
25. `queue.Enqueue(letter);`



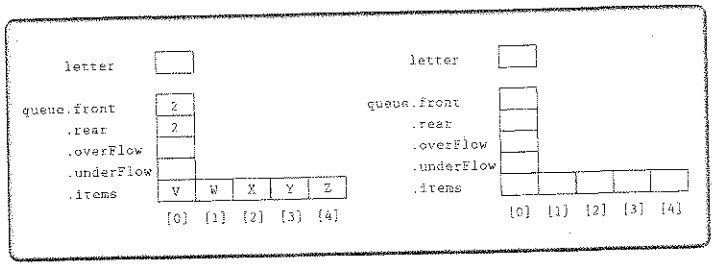
26. `queue.Enqueue(letter);`



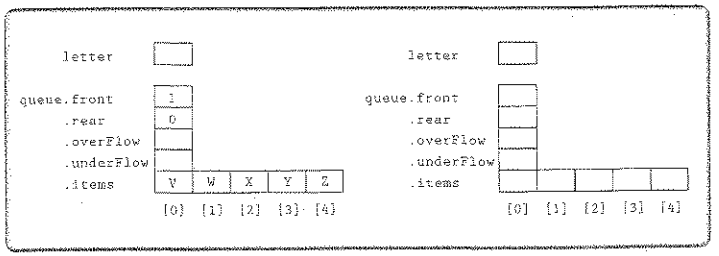
27. `queue.Dequeue(letter);`



28. `queue.Dequeue(letter);`



29. `queue.Dequeue(letter);`



30. Write a segment of code to perform each of the following operations. You may call any of the member functions of `QueType`. The details of the queue are encapsulated; you may use only the queue operations in the specification to perform the operations. (You may declare additional queue objects.)

   a. Set `secondElement` to the second element in the queue, leaving the queue without its original front two elements.
   b. Set `last` equal to the rear element in the queue, leaving the queue empty.
   c. Set `last` equal to the rear element in the queue, leaving the queue unchanged.
   d. Make a copy of the queue, leaving the queue unchanged.

31. (Multiple choice) The statement

```
queue.items[1] = queue.items[2];
```

(setting one element equal to the next element) in a client program of the queue class

   a. would cause a syntax error at compile time.
   b. would cause a run-time error.
   c. would not be considered an error by the computer, but would violate the encapsulation of the queue data type.
   d. would be a perfectly legal and appropriate way to accomplish the intended task.

32. (Multiple choice) The statements

```
queue.Enqueue(item1 + 1);
queue.Dequeue(item1 + 1);
```

in the client program

   a. would cause a syntax error at compile time.
   b. would cause a run-time error.
   c. would be legal, but would violate the encapsulation of the queue.
   d. would be perfectly legal and appropriate.

33. Given the following specification of a `Front` operation:

ItemType Front

Function:	Returns a copy of the front item on the queue.
Precondition:	Queue is not empty.
Postconditions:	Function value = copy of the front item on the queue.
	Queue is not changed.

a. Write this function as client code, using operations from the `QueType` class. (Remember—the client code has no access to the private members of the class.)

b. Write this function as a new member function of the `QueType` class.

34. Write the body for a function that replaces each copy of an item in a queue with another item. Use the following specification. (This function is in the client program.)

ReplaceItem(QueType queue, int oldItem, int newItem)

Function:	Replaces all occurrences of oldItem with newItem.
Precondition:	queue has been initialized.
Postcondition:	Each occurrence of oldItem in queue has been replaced by newItem.

You may use any of the member functions of `QueType`, but you may not assume any knowledge of the queue's implementation.

35. Indicate whether each of the following applications would be suitable for a queue.

a. An ailing company wants to evaluate employee records so as to lay off some workers on the basis of service time (the most recently hired employees are laid off first).

b. A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first come, first served basis.

c. A program to solve a maze is to backtrack to an earlier position (the last place where a choice was made) when a dead-end position is reached.

d. An inventory of parts is to be processed by part number.

e. An operating system is to process requests for computer resources by allocating the resources in the order in which they are requested.

f. A grocery chain wants to run a simulation to see how the average customer wait time would be affected by changing the number of checkout lines in its stores.

g. A dictionary of words used by a spelling checker is to be initialized.

h. Customers are to take numbers at a bakery and be served in order when their numbers come up.

i. Gamblers are to take numbers in the lottery and win if their numbers are picked.

36. Implement the following specification for a Boolean function in the client program that returns true if two queues are identical and false otherwise.

Boolean Identical(QueType queue1, QueType queue2)

Function:	Determines if two queues are identical.
Precondition:	queue1 and queue2 have been initialized.
Postconditions:	Queues are unchanged.
	Function value = (queue1 and queue2 are identical).

You may use any of the member functions of `QueType`, but you may not assume any knowledge of the queue's implementation.

37. Implement the following specification for an integer function in the client program that returns the number of items in a queue. The queue is unchanged.

int Length(QueType queue)

Function:	Determines the number of items in the queue.
Precondition:	queue has been initialized.
Postconditions:	queue is unchanged.
	Function value = number of items in queue.

38. One queue implementation discussed in this chapter dedicated an unused cell before the front of the queue to distinguish between a full queue and an empty queue. Write another queue implementation that keeps track of the length of the queue in a data member `length`.

b. Implement the member functions for this implementation. (Which of the member functions have to be changed and which do not?)

c. Compare this new implementation with the previous one in terms of Big-O notation.

39. Write a queue application that determines if two files are the same.

40. Discuss the difference between the `MakeEmpty` operation in the specification and a class constructor.

The following code segment (used for Exercises 43 and 44) is a count-controlled loop going from 1 through 5. At each iteration, the loop counter is either printed or put on a queue depending on the result of the Boolean function `RanFun()`. (The behavior of `RanFun()` is immaterial.) At the end of the loop, the items on the queue are dequeued and printed. Because of the logical properties of a queue, this code segment cannot print certain sequences of the values of the loop counter. You are given an output and asked to determine whether the code segment could generate the output.

```
for (count = 1; count <= 5; count++)
  if (RanFun())
    cout << count;
  else
    queue.Enqueue(count);
while (!queue.IsEmpty())
{
  queue.Dequeue(number);
  cout << number;
}
```

41. The following output is possible using a queue: `1 2 3 4 5`

a. True

b. False

c. Not enough information to determine

42. The following output is possible using a queue: `1 3 5 4 2`

a. True

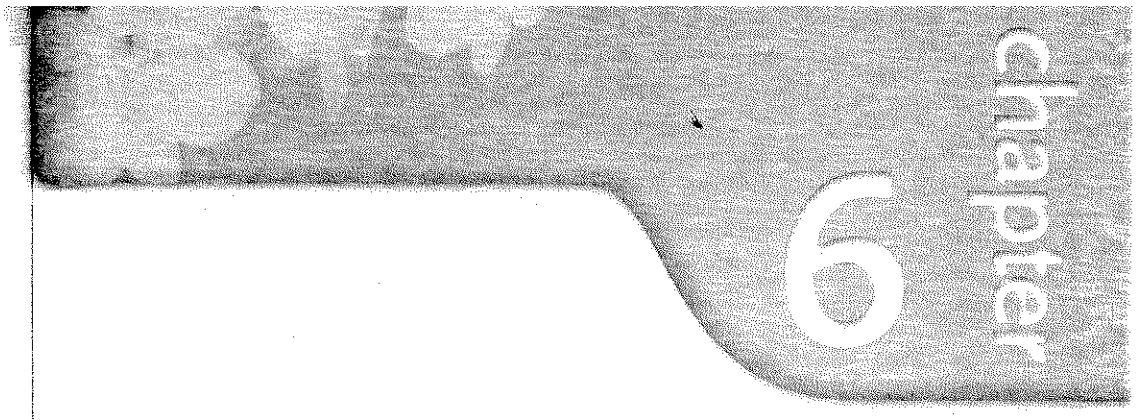
b. False

c. Not enough information to determine

43. Change the code for the program `Balanced` so that the calls to `Push`, `Pop`, and `Top` are embedded within a *try* clause. The *catch* clause should print an appropriate error message and halt.

44. Define and implement a counted stack, which inherits from `StackType`.

45. Create UML diagrams for the ADTs implemented in this chapter.



Chapter 6

Lists Plus

After studying this chapter, you should be able to

- Use the C++ template mechanism for defining generic data types
- Implement a circular linked list
- Implement a linked list with a header node or a trailer node or both
- Implement a doubly linked list
- Distinguish between shallow copying and deep copying
- Overload C++ operators
- Implement a linked list as an array of records
- Implement dynamic binding with virtual functions