CHAPTER 7

STACKS

IN THIS CHAPTER, YOU WILL:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn how to use a stack to remove recursion

This chapter discusses a very useful data structure called a stack. It has numerous applications in computer science.

Stacks

Suppose that you have a program with several functions. To be specific, suppose that you have the functions A, B, C, and D in your program. Now suppose that function A calls function B, function B calls function C, and function C calls function D. When function D terminates, control goes back to function C; when function C terminates, control goes back to function B; and when function B terminates, control goes back to function A. During program execution, how do you think the computer keeps track of the function calls? What about recursive functions? How does the computer keep track of the recursive calls? In Chapter 6, we designed a recursive function to print a linked list backward. What if you want to write a nonrecursive algorithm to print a linked list backward?

This section discusses the data structure called the **stack**, which the computer uses to implement function calls. You can also use stacks to convert recursive algorithms into nonrecursive algorithms, especially recursive algorithms that are not tail recursive. Stacks have numerous other applications in computer science. After developing the tools necessary to implement a stack, we will examine some applications of stacks.

A stack is a list of homogenous elements in which the addition and deletion of elements occurs only at one end, called the **top** of the stack. For example, in a cafeteria, the second tray in a stack of trays can be removed only if the first tray has been removed. For another example, to get to your favorite computer science book, which is underneath your math and history books, you must first remove the math and history books. After removing these books, the computer science book becomes the top book—that is, the top element of the stack. Figure 7-1 shows some examples of stacks.

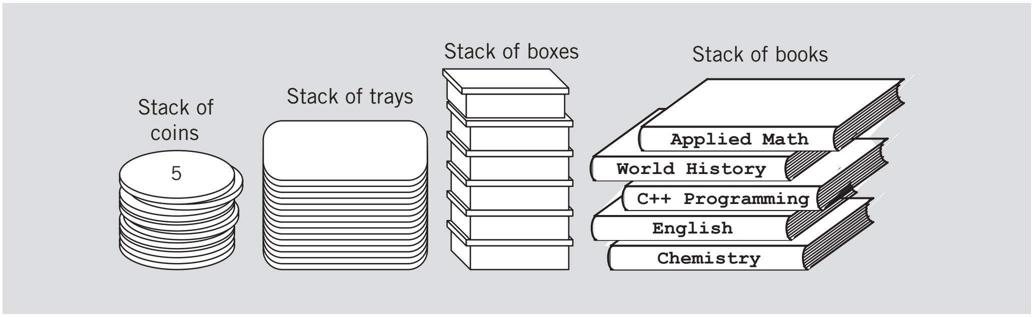


FIGURE 7-1 Various examples of stacks

The elements at the bottom of the stack have been in the stack the longest. The top element of the stack is the last element added to the stack. Because the elements are added and removed from one end (that is, the top), it follows that the item that is added last will be removed first. For this reason, a stack is also called a **Last In First Out (LIFO)** data structure.

Stack: A data structure in which the elements are added and removed from one end only; a Last In First Out (LIFO) data structure.

Now that you know what a stack is, let us see what kinds of operations can be performed on a stack. Because new items can be added to the stack, we can perform the add operation, called **push**, to add an element onto the stack. Similarly, because the top item can be retrieved and/or removed from the stack, we can perform the operation **top** to retrieve the top element of the stack, and the operation **pop** to remove the top element from the stack.

The push, top, and pop operations work as follows: Suppose there are boxes lying on the floor that need to be stacked on a table. Initially, all of the boxes are on the floor and the stack is empty. (See Figure 7-2.)

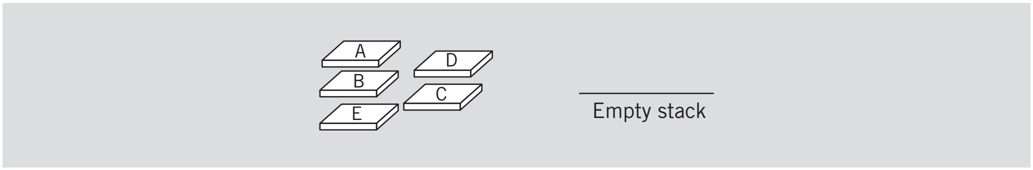


FIGURE 7-2 Empty stack

First we push box A onto the stack. After the push operation, the stack is as shown in Figure 7-3(a).

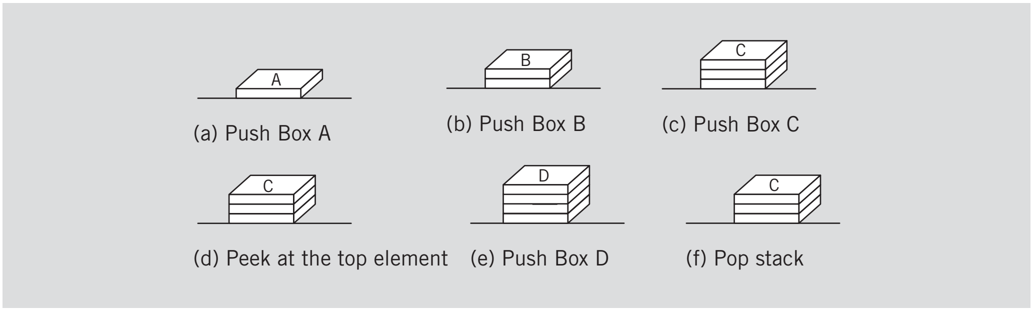


FIGURE 7-3 Stack operations

We then push box B onto the stack. After this push operation, the stack is as shown in Figure 7-3(b). Next, we push box C onto the stack. After this push operation, the stack is as shown in Figure 7-3(c). Next, we look at the top element of the stack. After this operation, the stack is unchanged and shown in Figure 7-3(d). We then push box D onto the stack. After this push operation, the stack is as shown in Figure 7-3(e). Next, we pop the stack. After the pop operation, the stack is as shown in Figure 7-3(f).

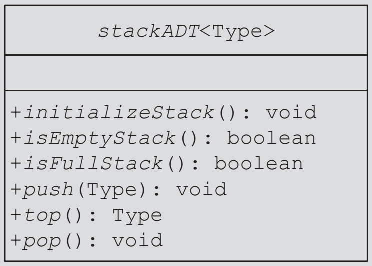
An element can be removed from the stack only if there is something in the stack, and an element can be added to the stack only if there is room. The two operations that immediately follow from push, top, and pop are **isFullStack** (checks whether the stack is full) and **isEmptyStack** (checks whether the stack is empty). Because a stack keeps changing as we add and remove elements, the stack must be empty before we first start using it. Thus, we need another operation, called **initializeStack**, which initializes the stack to an empty state. Therefore, to successfully implement a stack, we need at least these six operations, which are described next. (We might also need other operations on a stack, depending on the specific implementation.)

- **initializeStack**—Initializes the stack to an empty state.
- **isEmptyStack**—Determines whether the stack is empty. If the stack is empty, it returns the value true; otherwise, it returns the value false.
- **isFullStack**—Determines whether the stack is full. If the stack is full, it returns the value true; otherwise, it returns the value false.
- **push**—Adds a new element to the top of the stack. The input to this operation consists of the stack and the new element. Prior to this operation, the stack must exist and must not be full.
- **top**—Returns the top element of the stack. Prior to this operation, the stack must exist and must not be empty.
- **pop**—Removes the top element of the stack. Prior to this operation, the stack must exist and must not be empty.

The following abstract class stackADT defines these operations as an ADT:

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operations on a stack.
//*************************************************************

template <class Type>
class stackADT
{
public:
    virtual void initializeStack() = 0;
       //Method to initialize the stack to an empty state.
       //Postcondition: Stack is empty.

    virtual bool isEmptyStack() const = 0;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //    otherwise returns false.
    virtual bool isFullStack() const = 0;
      //Function to determine whether the stack is full.
      //Postcondition: Returns true if the stack is full,
      //    otherwise returns false.

    virtual void push(const Type& newItem) = 0;
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem is added
      //    to the top of the stack.

    virtual Type top() const = 0;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //    terminates; otherwise, the top element of the stack
      //    is returned.

    virtual void pop() = 0;
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top element
      //    is removed from the stack.
};
```

Figure 7-4 shows the UML class diagram of the `class stackADT`.



stackADT<Type>
+*initializeStack*(): void
+*isEmptyStack*(): boolean
+*isFullStack*(): boolean
+*push*(Type): void
+*top*(): Type
+*pop*(): void

FIGURE 7-4 UML class diagram of the `class stackADT`

We now consider the implementation of our abstract stack data structure. Functions such as `push` and `pop` that are required to implement a stack are not available to C++ programmers. We must write the functions to implement the stack operations.

Because all the elements of a stack are of the same type, a stack can be implemented as either an array or a linked structure. Both implementations are useful and are discussed in this chapter.

Implementation of Stacks as Arrays

Because all the elements of a stack are of the same type, you can use an array to implement a stack. The first element of the stack can be put in the first array slot, the second element of the stack in the second array slot, and so on. The top of the stack is the index of the last element added to the stack.

In this implementation of a stack, stack elements are stored in an array, and an array is a random access data structure; that is, you can directly access any element of the array. However, by definition, a stack is a data structure in which the elements are accessed (popped or pushed) at only one end—that is, a Last In First Out data structure. Thus, a stack element is accessed only through the top, not through the bottom or middle. This feature of a stack is extremely important and must be recognized in the beginning.

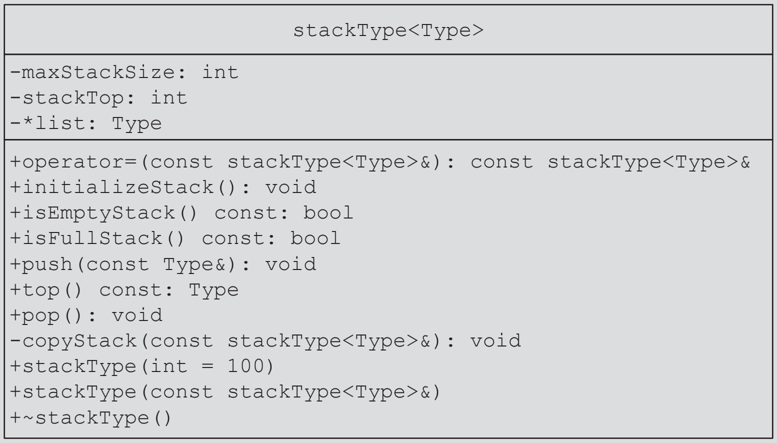
To keep track of the top position of the array, we can simply declare another variable, called **`stackTop`**.

The following `class`, `stackType`, implements the functions of the abstract `class stackADT`. By using a pointer, we can dynamically allocate arrays, so we will leave it for the user to specify the size of the array (that is, the stack size). We assume that the default stack size is 100. Because the `class stackType` has a pointer member variable (the pointer to the array to store the stack elements), we must overload the assignment operator and include the copy constructor and destructor. Moreover, we give a generic definition of the stack. Depending on the specific application, we can pass the stack element type when we declare a stack object.

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operation on a stack as an
// array.
//*************************************************************

template <class Type>
class stackType: public stackADT<Type>
{
public:
    const stackType<Type>& operator=(const stackType<Type>&);
      //Overload the assignment operator.

    void initializeStack();
      //Function to initialize the stack to an empty state.
      //Postcondition: stackTop = 0;

    bool isEmptyStack() const;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //    otherwise returns false.
    bool isFullStack() const;
      //Function to determine whether the stack is full.
      //Postcondition: Returns true if the stack is full,
      //    otherwise returns false.

    void push(const Type& newItem);
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem is
      //    added to the top of the stack.

    Type top() const;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //    terminates; otherwise, the top element of the stack
      //    is returned.

    void pop();
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top element is
      //    removed from the stack.

    stackType(int stackSize = 100);
      //Constructor
      //Create an array of the size stackSize to hold
      //the stack elements. The default stack size is 100.
      //Postcondition: The variable list contains the base address
      //   of the array, stackTop = 0, and maxStackSize = stackSize

    stackType(const stackType<Type>& otherStack);
      //Copy constructor

    ~stackType();
      //Destructor
      //Remove all the elements from the stack.
      //Postcondition: The array (list) holding the stack
      //    elements is deleted.

private:
    int maxStackSize; //variable to store the maximum stack size
    int stackTop;     //variable to point to the top of the stack
    Type *list; //pointer to the array that holds the stack elements

    void copyStack(const stackType<Type>& otherStack);
      //Function to make a copy of otherStack.
      //Postcondition: A copy of otherStack is created and assigned
      //    to this stack.
};
```

7

Figure 7-5 shows the UML class diagram of the `class stackType`.



```
stackType<Type>
-------------------------------------------------------------
-maxStackSize: int
-stackTop: int
-*list: Type
-------------------------------------------------------------
+operator=(const stackType<Type>&): const stackType<Type>&
+initializeStack(): void
+isEmptyStack() const: bool
+isFullStack() const: bool
+push(const Type&): void
+top() const: Type
+pop(): void
-copyStack(const stackType<Type>&): void
+stackType(int = 100)
+stackType(const stackType<Type>&)
+~stackType()
```

FIGURE 7-5 UML class diagram of the `class stackType`

NOTE If `stackTop` is `0`, the stack is empty. If `stackTop` is nonzero, the stack is nonempty and the top element of the stack is given by `stackTop – 1` because the first stack element is at position `0`.

NOTE The function `copyStack` is included as a `private` member. It contains the code that is common to the functions to overload the assignment operator and the copy constructor. We use this function only to implement the copy constructor and overload the assignment operator. To copy a stack into another stack, the program can use the assignment operator.

Figure 7-6 shows this data structure, wherein `stack` is an object of type `stackType`. Note that `stackTop` can range from `0` to `maxStackSize`. If `stackTop` is nonzero, then `stackTop - 1` is the index of the top element of the stack. Suppose that `maxStackSize = 100`.

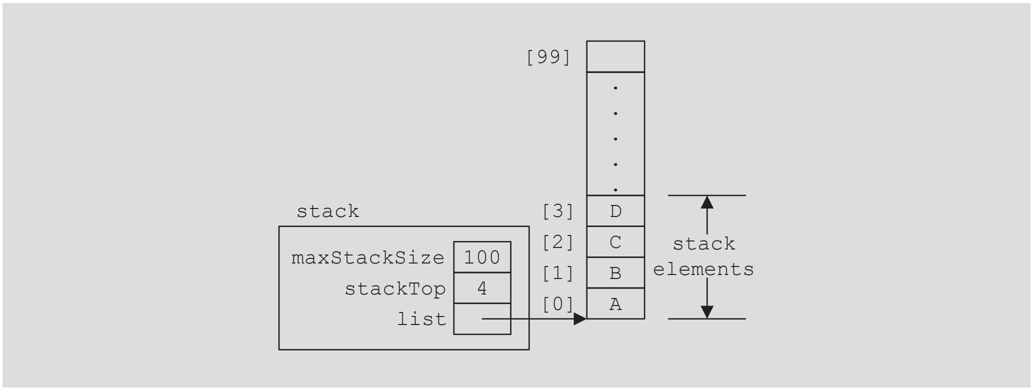


FIGURE 7-6 Example of a stack

Note that the pointer `list` contains the base address of the array (holding the stack elements)—that is, the address of the first array component. Next we discuss how to implement the member functions of the `class stackType`.

Initialize Stack

Let us consider the `initializeStack` operation. Because the value of `stackTop` indicates whether the stack is empty, we can simply set `stackTop` to `0` to initialize the stack. (See Figure 7-7.)

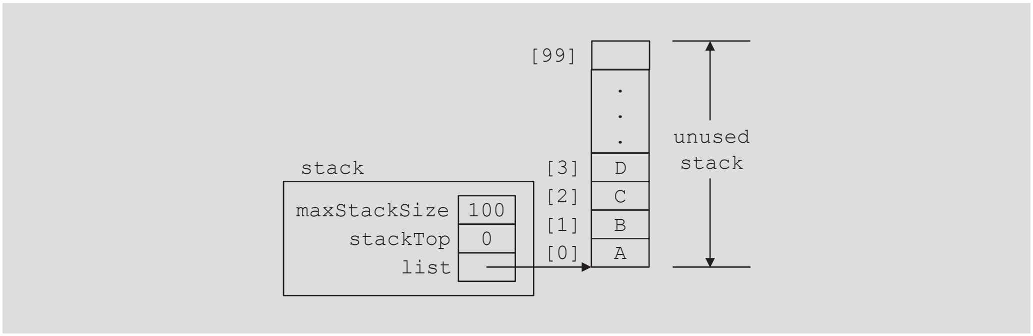


FIGURE 7-7 Empty stack

The definition of the function `initializeStack` is as follows:

```
template <class Type>
void stackType<Type>::initializeStack()
{
    stackTop = 0;
}//end initializeStack
```

Empty Stack

We have seen that the value of `stackTop` indicates whether the stack is empty. If `stackTop` is `0`, the stack is empty; otherwise, the stack is not empty. The definition of the function `isEmptyStack` is as follows:

```
template <class Type>
bool stackType<Type>::isEmptyStack() const
{
    return(stackTop == 0);
}//end isEmptyStack
```

Full Stack

Next, we consider the operation `isFullStack`. It follows that the stack is full if `stackTop` is equal to `maxStackSize`. The definition of the function `isFullStack` is as follows:

```
template <class Type>
bool stackType<Type>::isFullStack() const
{
    return(stackTop == maxStackSize);
} //end isFullStack
```

Push

Adding, or pushing, an element onto the stack is a two-step process. Recall that the value of `stackTop` indicates the number of elements in the stack, and `stackTop - 1` gives the position of the top element of the stack. Therefore, the `push` operation is as follows:

1. Store the `newItem` in the array component indicated by `stackTop`.
2. Increment `stackTop`.

Figure 7-8(a) shows the stack before pushing `'y'` into the stack. Figure 7-8(b) shows the stack after pushing `'y'` into the stack.

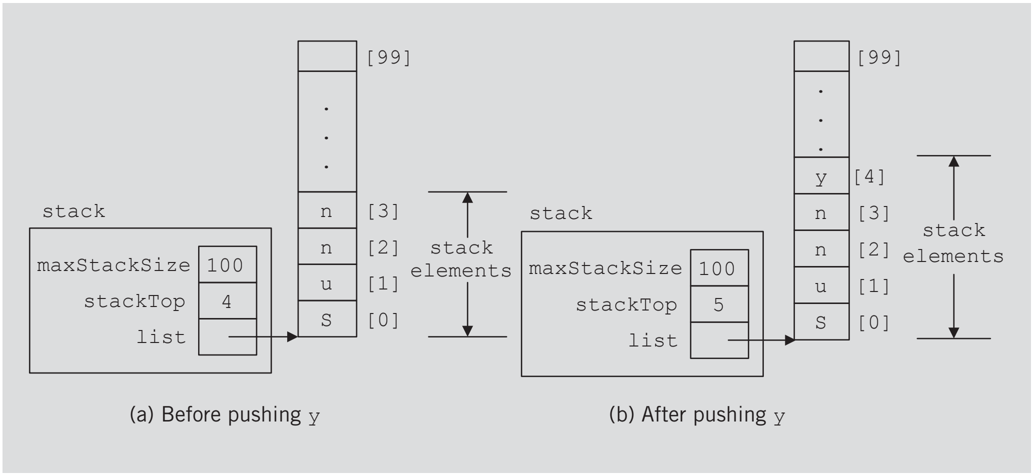


FIGURE 7-8 Stack before and after the `push` operation

The definition of the function `push` is as follows:

```
template <class Type>
void stackType<Type>::push(const Type& newItem)
{
    if (!isFullStack())
    {
        list[stackTop] = newItem;  //add newItem at the top
        stackTop++; //increment stackTop
    }
    else
        cout << "Cannot add to a full stack." << endl;
}//end push
```

If we try to add a new item to a full stack, the resulting condition is called an **overflow**. Error checking for an overflow can be handled in different ways. One way is as shown previously. Or, we can check for an overflow before calling the function `push`, as shown next (assuming `stack` is an object of type `stackType`).

```
if (!stack.isFullStack())
    stack.push(newItem);
```

Return the Top Element

The operation `top` returns the top element of the stack. Its definition is as follows:

```
template <class Type>
Type stackType<Type>::top() const
{
    assert(stackTop != 0);  //if stack is empty, terminate the
                            //program
    return list[stackTop - 1]; //return the element of the stack
                               //indicated by stackTop - 1
}//end top
```

Pop

To remove, or pop, an element from the stack, we simply decrement `stackTop` by `1`.

Figure 7-9(a) shows the stack before popping `'D'` from the stack. Figure 7-9(b) shows the stack after popping `'D'` from the stack.

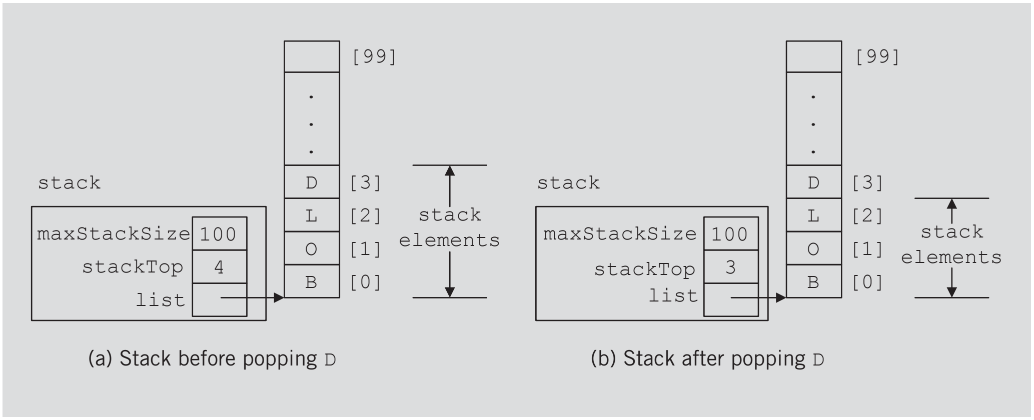


FIGURE 7-9 Stack before and after the pop operation

The definition of the function pop is as follows:

```
template <class Type>
void stackType<Type>::pop()
{
    if (!isEmptyStack())
        stackTop--;     //decrement stackTop
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```

If we try to remove an item from an empty stack, the resulting condition is called an **underflow**. Error checking for an underflow can be handled in different ways. One way is as shown previously. Or, we can check for an underflow before calling the function pop, as shown next (assuming stack is an object of type stackType).

```
if (!stack.isEmptyStack())
    stack.pop();
```

Copy Stack

The function copyStack makes a copy of a stack. The stack to be copied is passed as a parameter to the function copyStack. We will, in fact, use this function to implement the copy constructor and overload the assignment operator. The definition of this function is as follows:

```
template <class Type>
void stackType<Type>::copyStack(const stackType<Type>& otherStack)
{
    delete [] list;
    maxStackSize = otherStack.maxStackSize;
    stackTop = otherStack.stackTop;
    list = new Type[maxStackSize];

        //copy otherStack into this stack
    for (int j = 0; j < stackTop; j++)
        list[j] = otherStack.list[j];
} //end copyStack
```

Constructor and Destructor

The functions to implement the constructor and the destructor are straightforward. The constructor with parameters sets the stack size to the size specified by the user, sets `stackTop` to `0`, and creates an appropriate array in which to store the stack elements. If the user does not specify the size of the array in which to store the stack elements, the constructor uses the default value, which is `100`, to create an array of size `100`. The destructor simply deallocates the memory occupied by the array (that is, the stack) and sets `stackTop` to `0`. The definitions of the constructor and destructor are as follows:

```
template <class Type>
stackType<Type>::stackType(int stackSize)
{
    if (stackSize <= 0)
    {
        cout << "Size of the array to hold the stack must "
             << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxStackSize = 100;
    }
    else
        maxStackSize = stackSize;   //set the stack size to
                                    //the value specified by
                                    //the parameter stackSize

    stackTop = 0;                   //set stackTop to 0
    list = new Type[maxStackSize];  //create the array to
                                    //hold the stack elements
}//end constructor

template <class Type>
stackType<Type>::~stackType() //destructor
{
    delete [] list; //deallocate the memory occupied
                    //by the array
}//end destructor
```

Copy Constructor

The copy constructor is called when a stack object is passed as a (value) parameter to a function. It copies the values of the member variables of the actual parameter into the corresponding member variables of the formal parameter. Its definition is as follows:

```
template <class Type>
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
    list = NULL;

    copyStack(otherStack);
}//end copy constructor
```

Overloading the Assignment Operator (=)

Recall that for classes with pointer member variables, the assignment operator must be explicitly overloaded. The definition of the function to overload the assignment operator for the `class stackType` is as follows:

```
template <class Type>
const stackType<Type>& stackType<Type>::operator=
                          (const stackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
} //end operator=
```

Stack Header File

Now that you know how to implement the stack operations, you can put the definitions of the class and the functions to implement the stack operations together to create the stack header file. For the sake of completeness, we next describe the header file. Suppose that the name of the header file containing the definition of the `class stackType` is `myStack.h`. We will refer to this header file in any program that uses a stack.

```
//Header file: myStack.h

#ifndef H_StackType
#define H_StackType

#include <iostream>
#include <cassert>

#include "stackADT.h"

using namespace std;

//Place the definition of the class template stackType, as given
//previously in this chapter, here.

//Place the definitions of the member functions as discussed here.
#endif
```

The analysis of the stack operations is similar to the operations of the `class arrayListType` (Chapter 3). We, therefore, provide only a summary in Table 7-1.

TABLE 7-1 Time complexity of the operations of the `class stackType` on a stack with *n* elements

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(1)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(1)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

EXAMPLE 7-1

Before we give a programming example, let us first write a simple program that uses the **class** `stackType` and tests some of the stack operations. Among others, we will test the assignment operator and the copy constructor. The program and its output are as follows:

```
//*************************************************************
// Author: D.S. Malik
//
// This program tests various operations of a stack.
//*************************************************************

#include <iostream>
#include "myStack.h"

using namespace std;
```

```
void testCopyConstructor(stackType<int> otherStack);

int main()
{
    stackType<int> stack(50);
    stackType<int> copyStack(50);
    stackType<int> dummyStack(100);

    stack.initializeStack();
    stack.push(23);
    stack.push(45);
    stack.push(38);
    copyStack = stack;  //copy stack into copyStack

    cout << "The elements of copyStack: ";

    while (!copyStack.isEmptyStack())  //print copyStack
    {
        cout << copyStack.top() << " ";
        copyStack.pop();
    }
    cout << endl;

    copyStack = stack;
    testCopyConstructor(stack);  //test the copy constructor

    if (!stack.isEmptyStack())
        cout << "The original stack is not empty." << endl
             << "The top element of the original stack: "
             << copyStack.top() << endl;

    dummyStack = stack;  //copy stack into dummyStack

    cout << "The elements of dummyStack: ";

    while (!dummyStack.isEmptyStack())  //print dummyStack
    {
        cout << dummyStack.top() << " ";
        dummyStack.pop();
    }

    cout << endl;

    return 0;
}

void testCopyConstructor(stackType<int> otherStack)
{
    if (!otherStack.isEmptyStack())
        cout << "otherStack is not empty." << endl
             << "The top element of otherStack: "
             << otherStack.top() << endl;
}
```

Sample Run:

```
The elements of copyStack: 38 45 23
otherStack is not empty.
The top element of otherStack: 38
The original stack is not empty.
The top element of the original stack: 38
The elements of dummyStack: 38 45 23
```

It is recommended that you do a walk-through of this program.

PROGRAMMING EXAMPLE: Highest GPA

In this example, we write a C++ program that reads a data file consisting of each student's GPA followed by the student's name. The program then prints the highest GPA and the names of all the students who received that GPA. The program scans the input file only once. Moreover, we assume that there are a maximum of 100 students in the class.

Input The program reads an input file consisting of each student's GPA, followed by the student's name. Sample data is as follows:

```
3.5 Bill
3.6 John
2.7 Lisa
3.9 Kathy
3.4 Jason
3.9 David
3.4 Jack
```

Output The program outputs the highest GPA and all the names associated with the highest GPA. For example, for the preceding data, the highest GPA is `3.9` and the students with that GPA are `Kathy` and `David`.

PROGRAM ANALYSIS AND ALGORITHM DESIGN

We read the first GPA and the name of the student. Because this data is the first item read, it is the highest GPA so far. Next, we read the second GPA and the name of the student. We then compare this (second) GPA with the highest GPA so far. Three cases arise:

1. The new GPA is greater than the highest GPA so far. In this case, we do the following:
    a. Update the value of the highest GPA so far.
    b. Initialize the stack—that is, remove the names of the students from the stack.
    c. Save the name of the student having the highest GPA so far in the stack.

2. The new GPA is equal to the highest GPA so far. In this case, we add the name of the new student to the stack.
3. The new GPA is smaller than the highest GPA so far. In this case, we discard the name of the student having this grade.

We then read the next GPA and the name of the student, and repeat Steps 1 through 3. We continue this process until we reach the end of file.

From this discussion, it is clear that we need the following variables:

```
double GPA;          //variable to hold the current GPA
double highestGPA;   //variable to hold the highest GPA
string name;         //variable to hold the name of the student
stackType<string> stack(100); //object to implement the stack
```

The previous discussion translates into the following algorithm:

```
1. Declare the variables and initialize stack.
2. Open the input file.
3. If the input file does not exist, exit the program.
4. Set the output of the floating-point numbers to a fixed decimal
   format with a decimal point and trailing zeroes. Also, set the preci-
   sion to two decimal places.
5. Read the GPA and the student name.
6. highestGPA = GPA;
7. while (not end of file)
   {
   7.1. if (GPA > highestGPA)
        {
        7.1.1. initializeStack(stack);
        7.1.2. push(stack, student name);
        7.1.3. highestGPA = GPA;
        }
   7.2. else
           if (GPA is equal to highestGPA)
              push(stack, student name);
   7.3. Read GPA and student name;
   }
8. Output the highest GPA.
9. Output the names of the students having the highest GPA.
```

PROGRAM LISTING

```
//*************************************************************
// Author: D.S. Malik
//
// This program reads a data file consisting of students' GPAs
// followed by their names. The program then prints the highest
// GPA and the names of the students with the highest GPA.
//*************************************************************

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>

#include "myStack.h"

using namespace std;

int main()
{
        //Step 1
    double GPA;
    double highestGPA;
    string name;

    stackType<string> stack(100);

    ifstream infile;

    infile.open("HighestGPAData.txt");          //Step 2

    if (!infile)                                //Step 3
    {
        cout << "The input file does not "
             << "exist. Program terminates!" << endl;
        return 1;
    }

    cout << fixed << showpoint;                 //Step 4
    cout << setprecision(2);                    //Step 4

    infile >> GPA >> name;                      //Step 5

    highestGPA = GPA;                           //Step 6

    while (infile)                              //Step 7
    {
        if (GPA > highestGPA)                   //Step 7.1
        {
            stack.initializeStack();            //Step 7.1.1
            if (!stack.isFullStack())           //Step 7.1.2
                stack.push(name);

            highestGPA = GPA;                   //Step 7.1.3
        }
        else if (GPA == highestGPA)             //Step 7.2
            if (!stack.isFullStack())
                stack.push(name);
            else
            {
                cout << "Stack overflows. "
                     << "Program terminates!" << endl;
                return 1;  //exit program
            }

        infile >> GPA >> name;                  //Step 7.3
    }

    cout << "Highest GPA = " << highestGPA << endl;//Step 8
    cout << "The students holding the "
         << "highest GPA are:" << endl;

    while (!stack.isEmptyStack())                   //Step 9
    {
        cout << stack.top() << endl;
        stack.pop();
    }

    cout << endl;

    return 0;
}
```

Sample Run:

Input File (HighestGPAData.txt)

```
3.4 Randy
3.2 Kathy
2.5 Colt
3.4 Tom
3.8 Ron
3.8 Mickey
3.6 Peter
3.5 Donald
3.8 Cindy
3.7 Dome
3.9 Andy
3.8 Fox
3.9 Minnie
2.7 Gilda
3.9 Vinay
3.4 Danny
```

Output

```
Highest GPA = 3.90
The students holding the highest GPA are:
Vinay
Minnie
Andy
```

Note that the names of the students with the highest GPA are output in the reverse order, relative to the order they appear in the input because the top element of the stack is the last element added to the stack.

Linked Implementation of Stacks

Because an array size is fixed, in the array (linear) representation of a stack, only a fixed number of elements can be pushed onto the stack. If in a program the number of elements to be pushed exceeds the size of the array, the program might terminate in an error. We must overcome this problem.

We have seen that by using pointer variables we can dynamically allocate and deallocate memory, and by using linked lists we can dynamically organize data (such as an ordered list). Next, we will use these concepts to implement a stack dynamically.

Recall that in the linear representation of a stack, the value of `stackTop` indicates the number of elements in the stack, and the value of `stackTop - 1` points to the top item in the stack. With the help of `stackTop`, we can do several things: Find the top element, check whether the stack is empty, and so on.

Similar to the linear representation, in a linked representation `stackTop` is used to locate the top element in the stack. However, there is a slight difference. In the former case, `stackTop` gives the index of the array. In the latter case, `stackTop` gives the address (memory location) of the top element of the stack.

The following class implements the functions of the abstract `class stackADT`:

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operation on a stack as a
// linked list.
//*************************************************************

    //Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

```
template <class Type>
class linkedStackType: public stackADT<Type>
{
public:
    const linkedStackType<Type>& operator=
                              (const linkedStackType<Type>&);
      //Overload the assignment operator.

    bool isEmptyStack() const;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty;
      //    otherwise returns false.

    bool isFullStack() const;
      //Function to determine whether the stack is full.
      //Postcondition: Returns false.

    void initializeStack();
      //Function to initialize the stack to an empty state.
      //Postcondition: The stack elements are removed;
      //    stackTop = NULL;

    void push(const Type& newItem);
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem is
      //    added to the top of the stack.

    Type top() const;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //    terminates; otherwise, the top element of
      //    the stack is returned.

    void pop();
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top
      //    element is removed from the stack.

    linkedStackType();
      //Default constructor
      //Postcondition: stackTop = NULL;

    linkedStackType(const linkedStackType<Type>& otherStack);
      //Copy constructor

    ~linkedStackType();
      //Destructor
      //Postcondition: All the elements of the stack are removed.
private:
    nodeType<Type> *stackTop; //pointer to the stack

    void copyStack(const linkedStackType<Type>& otherStack);
      //Function to make a copy of otherStack.
      //Postcondition: A copy of otherStack is created and
      //    assigned to this stack.
};
```

NOTE In this linked implementation of stacks, the memory to store the stack elements is allocated dynamically. Logically, the stack is never full. The stack is full only if we run out of memory space. Therefore, in reality, the function `isFullStack` does not apply to linked implementations of stacks. However, the `class linkedStackType` must provide the definition of the function `isFullStack` because it is defined in the parent abstract `class stackADT`.

We leave the UML class diagram of the `class linkedStackType` as an exercise for you. (See Exercise 12 at the end of this chapter.)

EXAMPLE 7-2

Suppose that `stack` is an object of type `linkedStackType`. Figure 7-10(a) shows an empty stack and Figure 7-10(b) shows a nonempty stack.

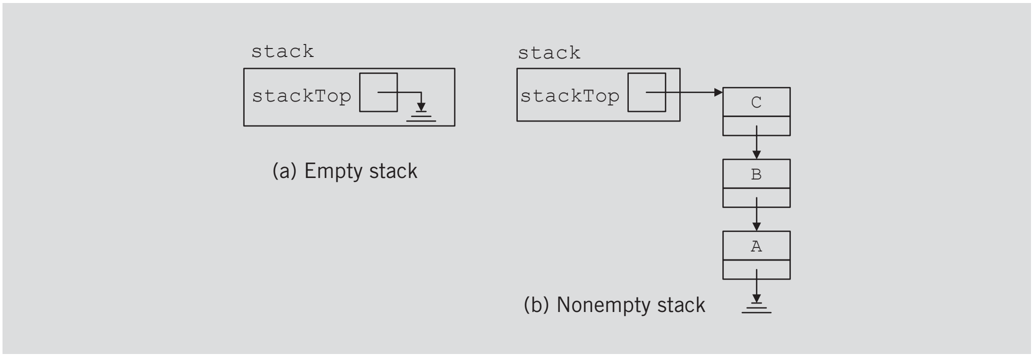


FIGURE 7-10 Empty and nonempty linked stacks

In Figure 7-10(b), the top element of the stack is `C`; that is, the last element pushed onto the stack is `C`.

Next, we discuss the definitions of the functions to implement the operations of a linked stack.

Default Constructor

The first operation that we consider is the default constructor. The default constructor initializes the stack to an empty state when a stack object is declared. Thus, this function sets `stackTop` to `NULL`. The definition of this function is as follows:

```
template  <class Type>
linkedStackType<Type>::linkedStackType()
{
    stackTop = NULL;
}
```

Empty Stack and Full Stack

The operations `isEmptyStack` and `isFullStack` are quite straightforward. The stack is empty if `stackTop` is `NULL`. Also, because the memory for a stack element is allocated and deallocated dynamically, the stack is never full. (The stack is full only if we run out of memory.) Thus, the function `isFullStack` always returns the value `false`. The definitions of the functions to implement these operations are as follows:

```
template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return(stackTop == NULL);
} //end isEmptyStack

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
} //end isFullStack
```

Recall that in the linked implementation of stacks, the function `isFullStack` does not apply because logically the stack is never full. However, you must provide its definition because it is included as an abstract function in the parent `class stackADT`.

Initialize Stack

The operation `initializeStack` reinitializes the stack to an empty state. Because the stack might contain some elements and we are using a linked implementation of a stack, we must deallocate the memory occupied by the stack elements and set `stackTop` to `NULL`. The definition of this function is as follows:

```
template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //pointer to delete the node

    while (stackTop != NULL)  //while there are elements in
                              //the stack
    {
        temp = stackTop;    //set temp to point to the
                            //current node
        stackTop = stackTop->link;  //advance stackTop to the
                                    //next node
        delete temp;    //deallocate memory occupied by temp
    }
} //end initializeStack
```

Next, we consider the push, top, and pop operations. From Figure 7-10(b), it is clear that the newElement will be added (in the case of push) at the beginning of the linked list pointed to by stackTop. In the case of pop, the node pointed to by stackTop will be removed. In both cases, the value of the pointer stackTop is updated. The operation top returns the info of the node to which stackTop is pointing.

Push

Consider the stack shown in Figure 7-11.

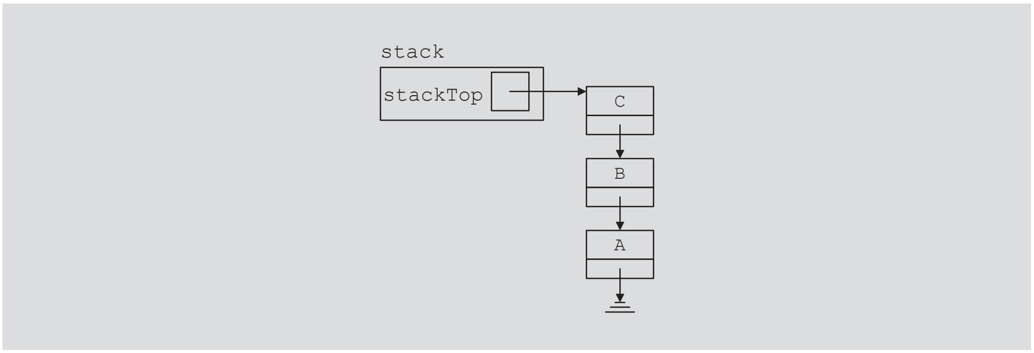


FIGURE 7-11 Stack before the push operation

Figure 7-12 shows the steps of the `push` operation. (Assume that the new element to be pushed is `'D'`.)

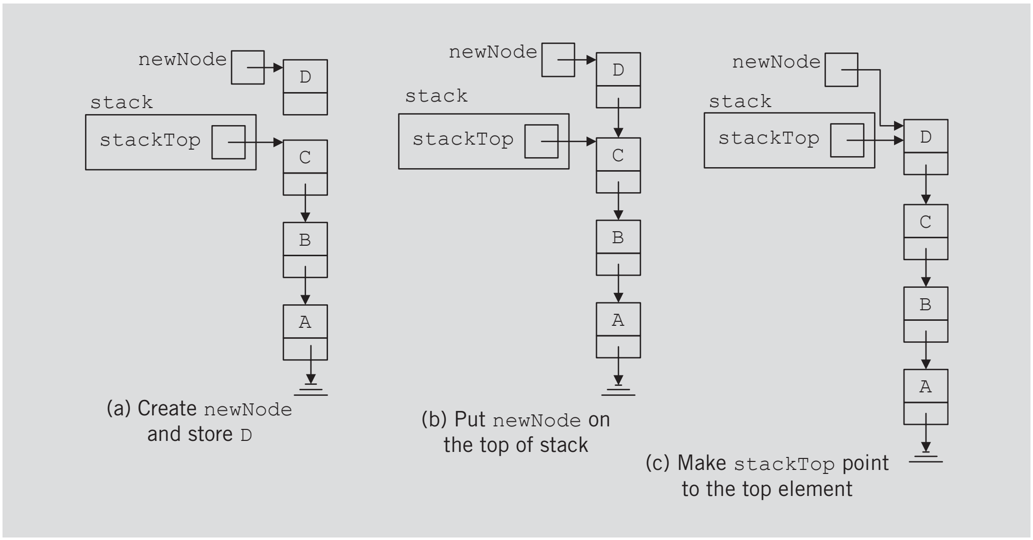

newNode
D
stack
stackTop
C
B
A
(a) Create newNode and store D
newNode
D
stack
stackTop
C
B
A
(b) Put newNode on the top of stack
newNode
stack
stackTop
D
C
B
A
(c) Make stackTop point to the top element

FIGURE 7-12 `Push` operation

As shown in Figure 7-12, to push `'D'` into the stack, first we create a new node and store `'D'` into it. Next, we put the new node on top of the stack. Finally, we make `stackTop` point to the top element of the stack. The definition of the function `push` is as follows:

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    nodeType<Type> *newNode;  //pointer to create the new node

    newNode = new nodeType<Type>; //create the node

    newNode->info = newElement; //store newElement in the node
    newNode->link = stackTop; //insert newNode before stackTop
    stackTop = newNode;       //set stackTop to point to the
                              //top node
} //end push
```

We do not need to check whether the stack is full before we push an element onto the stack because in this implementation, logically, the stack is never full.

Return the Top Element

The operation to return the top element of the stack is quite straightforward. Its definition is as follows:

```
template <class Type>
Type linkedStackType<Type>::top() const
{
    assert(stackTop != NULL); //if stack is empty,
                              //terminate the program
    return stackTop->info;    //return the top element
}//end top
```

Pop

Now we consider the pop operation, which removes the top element of the stack. Consider the stack shown in Figure 7-13.

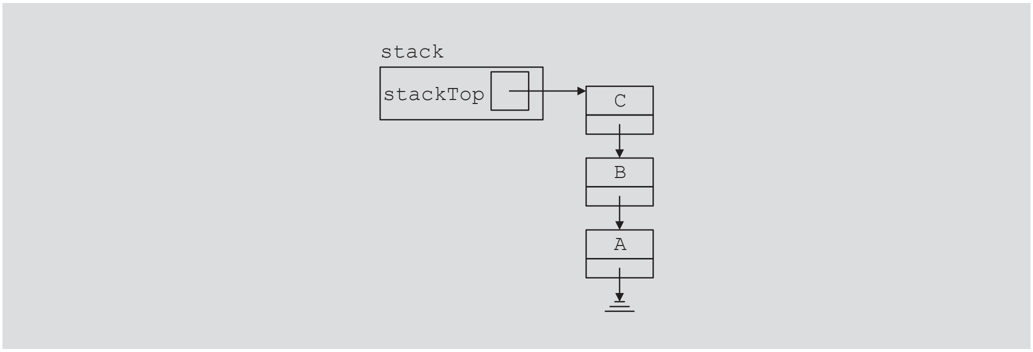


FIGURE 7-13 Stack before the pop operation

Figure 7-14 shows the pop operation.

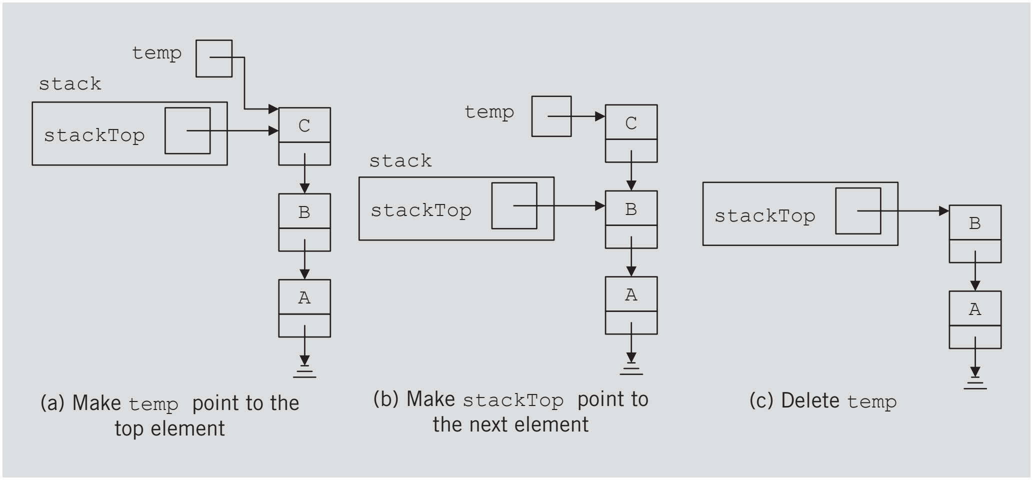

temp
stack
stackTop
C
B
A
(a) Make temp point to the top element
temp
C
stack
stackTop
B
A
(b) Make stackTop point to the next element
stackTop
B
A
(c) Delete temp

FIGURE 7-14 Pop operation

As shown in Figure 7-14, first we make a pointer temp point to the top of the stack. Next we make stackTop point to the next element of the stack, which will become the top element of the stack. Finally, we delete temp. The definition of the function pop is as follows:

```
template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;   //pointer to deallocate memory

    if (stackTop != NULL)
    {
        temp = stackTop;  //set temp to point to the top node

        stackTop = stackTop->link;  //advance stackTop to the
                                    //next node
        delete temp;    //delete the top node
    }
    else
        cout << "Cannot remove from an empty stack." << endl;
}//end pop
```

Copy Stack

The function copyStack makes an identical copy of a stack. Its definition is similar to the definition of copyList for linked lists, given in Chapter 5. The definition of the function copyStack is as follows:

```
template  <class Type>
void linkedStackType<Type>::copyStack
                     (const linkedStackType<Type>& otherStack)
{
    nodeType<Type> *newNode, *current, *last;

    if (stackTop != NULL) //if stack is nonempty, make it empty
        initializeStack();

    if (otherStack.stackTop == NULL)
        stackTop = NULL;
    else
    {
        current = otherStack.stackTop;  //set current to point
                                       //to the stack to be copied

            //copy the stackTop element of the stack
        stackTop = new nodeType<Type>;  //create the node

        stackTop->info = current->info; //copy the info
        stackTop->link = NULL;  //set the link field to NULL
        last = stackTop;        //set last to point to the node
        current = current->link; //set current to point to the
                                 //next node
            //copy the remaining stack
        while (current != NULL)
        {
            newNode = new nodeType<Type>;

            newNode->info = current->info;
            newNode->link = NULL;
            last->link = newNode;
            last = newNode;
            current = current->link;
        }//end while
    }//end else
} //end copyStack
```

Constructors and Destructors

We have already discussed the default constructor. To complete the implementation of the stack operations, next we give the definitions of the functions to implement the copy constructor and the destructor, and to overload the assignment operator. (These functions are similar to those discussed for linked lists in Chapter 5.)

```
    //copy constructor
template <class Type>
linkedStackType<Type>::linkedStackType(
                      const linkedStackType<Type>& otherStack)
{
    stackTop = NULL;
    copyStack(otherStack);
}//end copy constructor

    //destructor
template  <class Type>
linkedStackType<Type>::~linkedStackType()
{
    initializeStack();
}//end destructor
```

Overloading the Assignment Operator (=)

The definition of the function to overload the assignment operator for the `class linkedStackType` is as follows:

```
template <class Type>
const linkedStackType<Type>& linkedStackType<Type>::operator=
                      (const linkedStackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
}//end operator=
```

Table 7-2 summarizes the time complexity of the operations to implement a linked stack.

TABLE 7-2 Time complexity of the operations of the `class linkedStackType` on a stack with *n* elements

Function	Time complexity
`isEmptyStack`	$O(1)$
`isFullStack`	$O(1)$
`initializeStack`	$O(n)$
constructor	$O(1)$
`top`	$O(1)$
`push`	$O(1)$
`pop`	$O(1)$
`copyStack`	$O(n)$
destructor	$O(n)$
copy constructor	$O(n)$
Overloading the assignment operator	$O(n)$

The definition of a stack, and the functions to implement the stack operations discussed previously, are generic. Also, as in the case of an array representation of a stack, in the linked representation of a stack, we put the definition of the stack and the functions to implement the stack operations together in a (header) file. A client's program can include this header file via the `include` statement.

The program in Example 7-3 illustrates how a `linkedStack` object is used in a program.

EXAMPLE 7-3

We assume that the definition of the `class linkedStackType` and the functions to implement the stack operations are included in the header file `"linkedStack.h"`.

```
//*************************************************************
// Author: D.S. Malik
//
// This program tests various operations of a linked stack.
//*************************************************************
```

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

using namespace std;

void testCopy(linkedStackType<int> OStack);

int main()
{
    linkedStackType<int> stack;
    linkedStackType<int> otherStack;
    linkedStackType<int> newStack;

        //Add elements into stack
    stack.push(34);
    stack.push(43);
    stack.push(27);

        //Use the assignment operator to copy the elements
        //of stack into newStack
    newStack = stack;

    cout << "After the assignment operator, newStack: "
         << endl;

        //Output the elements of newStack
    while (!newStack.isEmptyStack())
    {
        cout << newStack.top() << endl;
        newStack.pop();
    }

        //Use the assignment operator to copy the elements
        //of stack into otherStack
    otherStack = stack;

    cout << "Testing the copy constructor." << endl;

    testCopy(otherStack);

    cout << "After the copy constructor, otherStack: " << endl;

    while (!otherStack.isEmptyStack())
    {
        cout << otherStack.top() << endl;
        otherStack.pop();
    }

    return 0;
}
```

```
     //Function to test the copy constructor
void testCopy(linkedStackType<int> OStack)
{
    cout << "Stack in the function testCopy:" << endl;

    while (!OStack.isEmptyStack())
    {
        cout << OStack.top() << endl;
        OStack.pop();
    }
}
```

Sample Run:

```
After the assignment operator, newStack:
27
43
34
Testing the copy constructor.
Stack in the function testCopy:
27
43
34
After the copy constructor, otherStack:
27
43
34
```

Stack as Derived from the `class unorderedLinkedList`

If we compare the `push` function of the stack with the `insertFirst` function discussed for general lists in Chapter 5, we see that the algorithms to implement these operations are similar. A comparison of other functions, such as `initializeStack` and `initializeList`, `isEmptyList` and `isEmptyStack`, and so on, suggests that the `class linkedStackType` can be derived from the `class linkedListType`. Moreover, the functions `pop` and `isFullStack` can be implemented as in the previous section. Note that the `class linkedListType` is an abstract class and does not implement all the operations. However, the `class unorderedLinkedList` is derived from the the `class linkedListType` and provides the definitions of the abstract functions of the `class linkedListType`. Therefore, we can derive the `class linkedStackType` from the `class unorderedLinkedList`.

Next, we define the `class linkedStackType` that is derived from the `class unorderedLinkedList`. The definitions of the functions to implement the stack operations are also given.

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

```
using namespace std;

template <class Type>
class linkedStackType: public unorderedLinkedList<Type>
{
public:
    void initializeStack();
    bool isEmptyStack() const;
    bool isFullStack() const;
    void push(const Type& newItem);
    Type top() const;
    void pop();
};

template <class Type>
void linkedStackType<Type>::initializeStack()
{
    unorderedLinkedList<Type>::initializeList();
}

template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return unorderedLinkedList<Type>::isEmptyList();
}

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
}

template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    unorderedLinkedList<Type>::insertFirst(newElement);
}

template <class Type>
Type linkedStackType<Type>::top() const
{
    return unorderedLinkedList<Type>::front();
}

template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;

    temp = first;
    first = first->link;
    delete temp;
}
```

Application of Stacks: Postfix Expressions Calculator

The usual notation for writing arithmetic expressions (the notation we learned in elementary school) is called **infix** notation, in which the operator is written between the operands. For example, in the expression $a + b$, the operator + is between the operands a and b. In infix notation, the operators have precedence. That is, we must evaluate expressions from left to right, and multiplication and division have higher precedence than addition and subtraction. If we want to evaluate the expression in a different order, we must include parentheses. For example, in the expression $a + b \star c$, we first evaluate ⋆ using the operands b and c, and then we evaluate + using the operand a and the result of $b \star$ c.

In the early 1920s, the Polish mathematician Jan Lukasiewicz discovered that if operators were written before the operands (**prefix** or **Polish** notation; for example, + a b), the parentheses can be omitted. In the late 1950s, the Australian philosopher and early computer scientist Charles L. Hamblin proposed a scheme in which the operators *follow* the operands (postfix operators), resulting in the **Reverse Polish** notation. This has the advantage that the operators appear in the order required for computation.

For example, the expression:

$a + b \star c$

in a postfix expression is:

$a\ b\ c \star +$

The following example shows various infix expressions and their equivalent postfix expressions.

EXAMPLE 7-4

Infix expression	**Equivalent postfix expression**
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c * +$
$a * b + c$	$a\ b * c +$
$(a + b) * c$	$a\ b + c *$
$(a - b) * (c + d)$	$a\ b - c\ d + *$
$(a + b) * (c - d / e) + f$	$a\ b + c\ d\ e / - * f +$

Shortly after Lukasiewicz's discovery, it was realized that postfix notation had important applications in computer science. In fact, many compilers use stacks to first translate infix expressions into some form of postfix notation and then translate this postfix expression into machine code. Postfix expressions can be evaluated using the following algorithm:

Scan the expression from left to right. When an operator is found, back up to get the required number of operands, perform the operation, and continue.

Consider the following postfix expression:

```
6 3 + 2 * =
```

Let us evaluate this expression using a stack and the previous algorithm. Figure 7-15 shows how this expression gets evaluated.

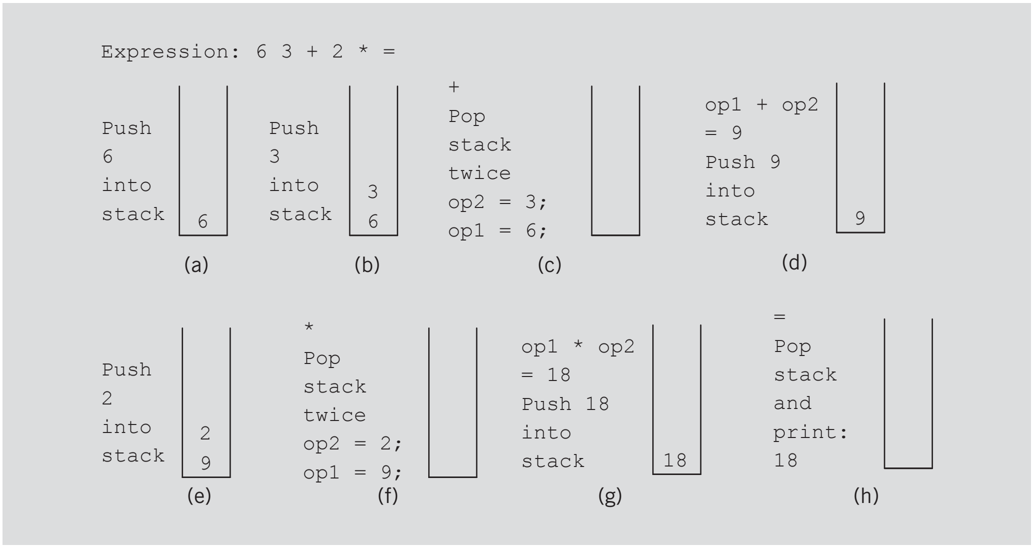


FIGURE 7-15 Evaluating the postfix expression: 6 3 + 2 * =

Read the first symbol, `6`, which is a number. Push the number onto the stack; see Figure 7-15(a). Read the next symbol, `3`, which is a number. Push the number onto the stack; see Figure 7-15(b). Read the next symbol, `+`, which is an operator. Because an operator requires two operands to be evaluated, pop the stack twice; see Figure 7-15(c). Perform the operation and put the result back onto the stack; see Figure 7-15(d).

Read the next symbol, `2`, which is a number. Push the number onto the stack; see Figure 7-15(e). Read the next symbol, `*`, which is an operator. Because an operator requires two operands to be evaluated, pop the stack twice; see Figure 7-15(f). Perform the operation and put the result back onto the stack; see Figure 7-15(g).

Scan the next symbol, `=`, which is the equal sign, indicating the end of the expression. Therefore, print the result. The result of the expression is in the stack, so pop and print; see Figure 7-15(h).

The value of the expression `6 3 + 2 * = 18`.

From this discussion, it is clear that when we read a symbol other than a number, the following cases arise:

1. The symbol we read is one of the following: `+`, `-`, `*`, `/`, or `=`.
    a. If the symbol is `+`, `-`, `*`, or `/`, the symbol is an operator and so we must evaluate it. Because an operator requires two operands, the stack must have at least two elements; otherwise, the expression has an error.
    b. If the symbol is `=` (an equal sign), the expression ends and we must print the answer. At this step, the stack must contain exactly one element; otherwise, the expression has an error.
2. The symbol we read is something other than +, -, *, /, or =. In this case, the expression contains an illegal operator.

It is also clear that when an operand (number) is encountered in an expression, it is pushed onto the stack because the operator comes after the operands.

Consider the following expressions:

i. `7 6 + 3 ; 6 - =`

ii. `14 + 2 3 * =`

iii. `14 2 3 + =`

Expression (i) has an illegal operator, expression (ii) does not have enough operands for `+`, and expression (iii) has too many operands. In the case of expression (iii), when we encounter the equal sign (=), the stack will have two elements and this error cannot be discovered until we are ready to print the value of the expression.

To make the input easier to read, we assume that the postfix expressions are in the following form:

```
#6 #3 + #2 * =
```

The symbol `#` precedes each number in the expression. If the symbol scanned is `#`, the next input is a number (that is, an operand). If the symbol scanned is not `#`, it is either an operator (might be illegal) or an equal sign (indicating the end of the expression). Furthermore, we assume that each expression contains only the `+`, `-`, `*`, and `/` operators.

This program outputs the entire postfix expression together with the answer. If the expression has an error, the expression is discarded. In this case, the program outputs the expression together with an appropriate error message. Because an expression might contain an error, we must clear the stack before processing the next expression. Also, the stack must be initialized; that is, the stack must be empty.

MAIN ALGORITHM

Pursuant to the previous discussion, the main algorithm in pseudocode is as follows:

```
Read the first character
while not the end of input data
{
    a. initialize the stack
    b. process the expression
    c. output result
    d. get the next expression
}
```

To simplify the complexity of the function `main`, we write four functions—`evaluateExpression`, `evaluateOpr`, `discardExp`, and `printResult`. The function `evaluateExpression`, if possible, evaluates the expression and leaves the result in the stack. If the postfix expression is error free, the function `printResult` outputs the result. The function `evaluateOpr` evaluates an operator, and the function `discardExp` discards the current expression if there is any error in the expression.

FUNCTION `evaluateExpression`

The function `evaluateExpression` evaluates each postfix expression. Each expression ends with the symbol =. The general algorithm in pseudocode is as follows:

```
while (ch is not = '=') //process each expression
                        //= marks the end of an expression
{
    switch (ch)
    {
    case '#':
        read a number
        output the number;
        push the number onto the stack;
        break;
     default:
        assume that ch is an operation
        evaluate the operation;
    } //end switch

    if no error was found, then
    {
        read next ch;
        output ch;
    }
    else
        Discard the expression
} //end while
```

From this algorithm, it follows that this method has five parameters—a parameter to access the input file, a parameter to access the output file, a parameter to access the stack, a parameter to pass a character of the expression, and a parameter to indicate whether there is an error in the expression. The definition of this function is as follows:

```
void evaluateExpression(ifstream& inpF, ofstream& outF,
                        stackType<double>& stack,
                        char& ch, bool& isExpOk)
```

```
{
    double num;
    outF << ch;

    while (ch != '=')
    {
        switch (ch)
        {
        case '#':
            inpF >> num;
            outF << num << " ";
            if (!stack.isFullStack())
                stack.push(num);
            else
            {
                cout << "Stack overflow. "
                     << "Program terminates!" << endl;
                exit(0);  //terminate the program
            }

            break;

        default:
            evaluateOpr(outF, stack, ch, isExpOk);
        }//end switch

        if (isExpOk) //if no error
        {
            inpF >> ch;
            outF << ch;

            if (ch != '#')
                outF << " ";
        }
        else
            discardExp(inpF, outF, ch);
    } //end while (!= '=')
} //end evaluateExpression
```

Note that the funtion `exit` terminates the program.

FUNCTION `evaluateOpr`

This function (if possible) evaluates an expression. Two operands are needed to evaluate an operation and operands are saved in the stack. Therefore, the stack must contain at least two numbers. If the stack contains fewer than two numbers, the expression has an error. In this case, the entire expression is discarded and an appropriate message is printed. This function also checks for any illegal operations. In pseudocode, this function is as follows:

```
if stack is empty
{
    error in the expression
    set expressionOk to false
}
else
{
    retrieve the top element of stack into op2
    pop stack
    if stack is empty
    {
        error in the expression
        set expressionOk to false
    }
    else
    {
        retrieve the top element of stack into op1
        pop stack

          //If the operation is legal, perform the operation and
          //push the result onto the stack;
          //otherwise, report the error.
        switch (ch)
        {
        case '+': //add the operands: op1 + op2
            stack.push(op1 + op2);
            break;
        case '-': //subtract the operands: op1 - op2
            stack.push(op1 - op2);
            break;
        case '*': //multiply the operands: op1 * op2
            stack.push(op1 * op2);
            break;
        case '/': //If (op2 != 0), op1 / op2
            stack.push(op1 / op2);
            break;
        otherwise operation is illegal
           {
               output an appropriate message;
               set expressionOk to false
           }
        } //end switch
}
```

Following this pseudocode, the definition of the function `evaluateOpr` is as follows:

```
void evaluateOpr(ofstream& out, stackType<double>& stack,
                 char& ch, bool& isExpOk)
{
    double op1, op2;
    if (stack.isEmptyStack())
    {
        out << " (Not enough operands)";
        isExpOk = false;
    }
    else
    {
        op2 = stack.top();
        stack.pop();

        if (stack.isEmptyStack())
        {
            out << " (Not enough operands)";
            isExpOk = false;
        }
        else
        {
            op1 = stack.top();
            stack.pop();

            switch (ch)
            {
            case '+':
                stack.push(op1 + op2);
                break;

            case '-':
                stack.push(op1 - op2);
                break;

            case '*':
                stack.push(op1 * op2);
                break;

            case '/':
                if (op2 != 0)
                    stack.push(op1 / op2);
                else
                {
                    out << " (Division by 0)";
                    isExpOk = false;
                }
                break;

            default:
                out << " (Illegal operator)";
                isExpOk = false;
            }//end switch
        } //end else
    } //end else
} //end evaluateOpr
```

FUNCTION `discardExp`

This function is called whenever an error is discovered in the expression. It reads and writes the input data only until the input is `'='`, the end of the expression. The definiton of this function is as follows:

```
void discardExp(ifstream& in, ofstream& out, char& ch)
{
    while (ch != '=')
    {
        in.get(ch);
        out << ch;
    }
} //end discardExp
```

FUNCTION `printResult`

If the postfix expression contains no errors, the function `printResult` prints the result; otherwise, it outputs an appropriate message. The result of the expression is in the stack and the output is sent to a file. Therefore, this function must have access to the stack and the output file. Suppose that no errors were encountered by the method `evaluateExpression`. If the stack has only one element, the expression is error free and the top element of the stack is printed. If either the stack is empty or it has more than one element, there is an error in the postfix expression. In this case, this method outputs an appropriate error message. The definition of this method is as follows:

```
void printResult(ofstream& outF, stackType<double>& stack,
                 bool isExpOk)
{
    double result;

    if (isExpOk) //if no error, print the result
    {
        if (!stack.isEmptyStack())
        {
            result = stack.top();
            stack.pop();

            if (stack.isEmptyStack())
                outF << result << endl;
            else
                outF << " (Error: Too many operands)" << endl;
        } //end if
        else
            outF << " (Error in the expression)" << endl;
    }
    else
        outF << " (Error in the expression)" << endl;

    outF << "_________________________________"
         << endl << endl;
} //end printResult
```

PROGRAM LISTING

```
//*************************************************************
// Author: D.S. Malik
//
// This program uses a stack to evaluate postfix expressions.
//*************************************************************

#include <iostream>
#include <iomanip>
#include <fstream>
#include "mystack.h"

using namespace std;

void evaluateExpression(ifstream& inpF, ofstream& outF,
                        stackType<double>& stack,
                        char& ch, bool& isExpOk);
void evaluateOpr(ofstream& out, stackType<double>& stack,
                 char& ch, bool& isExpOk);
void discardExp(ifstream& in, ofstream& out, char& ch);
void printResult(ofstream& outF, stackType<double>& stack,
                 bool isExpOk);

int main()
{
    bool expressionOk;
    char ch;
    stackType<double> stack(100);
    ifstream infile;
    ofstream outfile;

    infile.open("RpnData.txt");

    if (!infile)
    {
        cout << "Cannot open the input file. "
             << "Program terminates!" << endl;
        return 1;
    }

    outfile.open("RpnOutput.txt");

    outfile << fixed << showpoint;
    outfile << setprecision(2);

    infile >> ch;
    while (infile)
    {
        stack.initializeStack();
        expressionOk = true;
        outfile << ch;
        evaluateExpression(infile, outfile, stack, ch,
                           expressionOk);
        printResult(outfile, stack, expressionOk);
        infile >> ch; //begin processing the next expression
    } //end while

    infile.close();
    outfile.close();

    return 0;

} //end main

//Place the definitions of the function evaluateExpression,
//evaluateOpr, discardExp, and printResult as described
//previously here.
```

Sample Run:

Input File

```
#35 #27 + #3 * =
#26 #28 + #32 #2 ; - #5 / =
#23 #30 #15 * / =
#2 #3 #4 + =
#20 #29 #9 * ; =
#25 #23 - + =
#34 #24 #12 #7 / * + #23 - =
```

Output

```
#35.00 #27.00 + #3.00 * = 186.00
_________________________________

#26.00 #28.00 + #32.00 #2.00 ; (Illegal operator) - #5 / = (Error in the expression)
_________________________________

#23.00 #30.00 #15.00 * / = 0.05
_________________________________

#2.00 #3.00 #4.00 + = (Error: Too many operands)
_________________________________

#20.00 #29.00 #9.00 * ; (Illegal operator) = (Error in the expression)
_________________________________

#25.00 #23.00 - + (Not enough operands) = (Error in the expression)
_________________________________

#34.00 #24.00 #12.00 #7.00 / * + #23.00 - = 52.14
_________________________________
```

Removing Recursion: Nonrecursive Algorithm to Print a Linked List Backward

In Chapter 6, we used recursion to print a linked list backward. In this section, you will learn how a stack can be used to design a nonrecursive algorithm to print a linked list backward.

Consider the linked list shown in Figure 7-16.

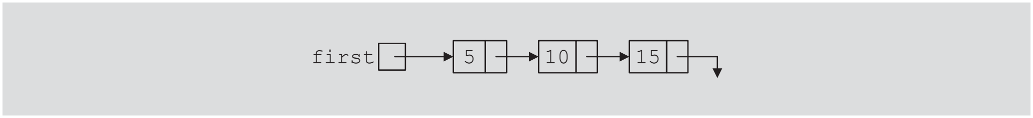


FIGURE 7-16 Linked list

To print the list backward, first we need to get to the last node of the list, which we can do by traversing the linked list starting at the first node. However, once we are at the last node, how do we get back to the previous node, especially given that links go in only one direction? You can again traverse the linked list with the appropriate loop termination condition, but this approach might waste a considerable amount of computer time, especially if the list is very large. Moreover, if we do this for every node in the list, the program might execute very slowly. Next, we show how to use a stack effectively to print the list backward.

After printing the `info` of a particular node, we need to move to the node immediately behind this node. For example, after printing `15`, we need to move to the node with `info 10`. Thus, while initially traversing the list to move to the last node, we must save a pointer to each node. For example, for the list in Figure 7-16, we must save a pointer to each of the nodes with `info 5` and `10`. After printing `15`, we go back to the node with `info 10`; after printing `10`, we go back to the node with `info 5`. From this, it follows that we must save pointers to each node in a stack, so as to implement the Last In First Out principle.

Because the number of nodes in a linked list is usually not known, we will use the linked implementation of a stack. Suppose that `stack` is an object of type `linkedListType`, and `current` is a pointer of the same type as the pointer `first`. Consider the following statements:

```
current = first;               //Line 1

while (current != NULL)        //Line 2
{                              //Line 3
    stack.push(current);       //Line 4
    current = current->link;   //Line 5
}                              //Line 6
```

After the statement in Line 1 executes, `current` points to the first node. (See Figure 7-17)

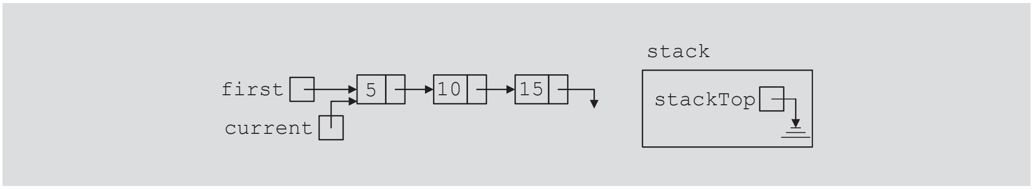


FIGURE 7-17 List after the statement `current = first;` executes

Because `current` is not `NULL`, the statements in Lines 4 and 5 execute. (See Figure 7-18.)

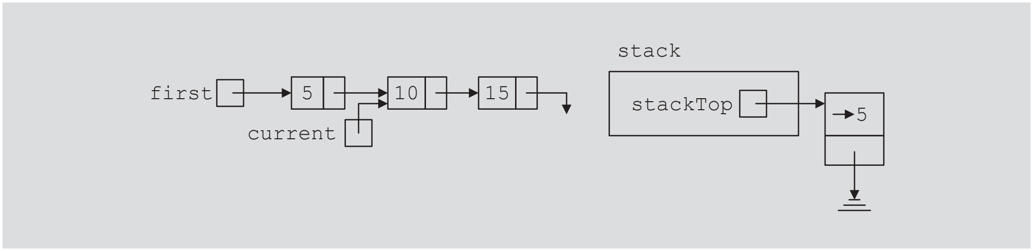


FIGURE 7-18 List and stack after the statements `stack.push(current);` and `current = current->link;` execute

Because `current` is not `NULL`, statements in Lines 4 and 5 execute. In fact, statements in Lines 4 and 5 execute until `current` beomes `NULL`. When `current` is `NULL`, Figure 7-19 results.

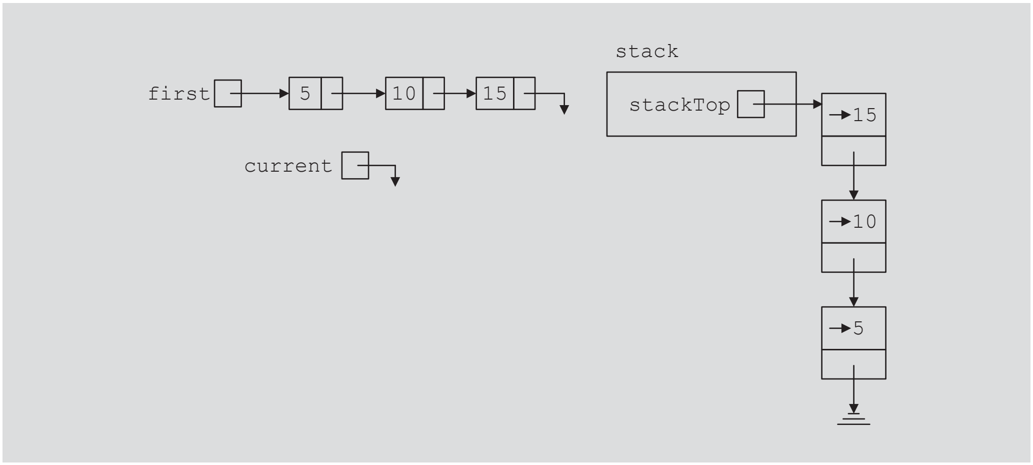


FIGURE 7-19 List and stack after the `while` statement executes

After the statement in Line 4 executes, the loop condition, in Line 2, is evaluated again. Because `current` is `NULL`, the loop condition evaluates to `false` and the `while` loop, in Line 2, terminates. From Figure 7-19, it follows that a pointer to each node in the linked list is saved in the stack. The top element of the stack contains a pointer to the last node in the list, and so on. Let us now execute the following statements:

```
while (!stack.isEmptyStack())       //Line 7
{                                   //Line 8
    current = stack.top();          //Line 9
    stack.pop();                    //Line 10
    cout << current->info << " ";   //Line 11
}                                   //Line 12
```

The loop condition in Line 7 evaluates to `true` because the stack is nonempty. Therefore, the statements in Lines 9, 10, and 11 execute. After the statement in Line 9 executes, `current` points to the last node. The statement in Line 10 removes the top element of the stack; see Figure 7-20.

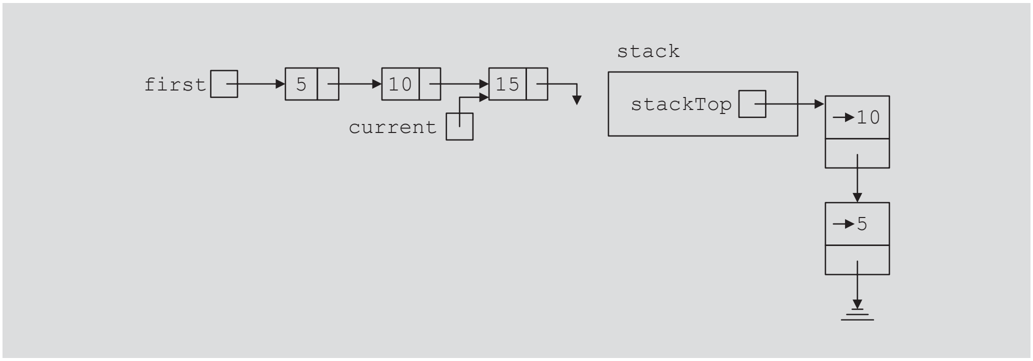


FIGURE 7-20 List and stack after the statements `current = stack.top();` and `stack.pop();` execute

The statement in Line 11 outputs `current->info`, which is `15`.

Because `stack` is nonempty, the body of the `while` loop executes again. In fact, for the linked list in Figure 7-20, the body of the `while` loop executes two more times; the first time it prints `10`, and the second time it prints `5`. Ater printing `5`, the `stack` becomes empty and the `while` loop terminates. It follows that the `while` loop in Line 7 produces the following output:

```
15 10 5
```

STL `class stack`

The previous sections discussed the data structure `stack` in detail. Because a stack is an important data structure, the Standard Template Library (STL) provides a class to implement a stack in a program. The name of the class defining a stack is `stack`; the name of the header file containing the definition of the class stack is stack. The implementation of the class stack provided by the STL is similar to the one described in this chapter. Table 7-3 defines the various operations supported by the stack container class.

TABLE 7-3 Operations on a stack object

Operation	Effect
`size`	Returns the actual number of elements in the stack.
`empty`	Returns `true` if the stack is empty, and `false` otherwise.
`push(item)`	Inserts a copy of item into the stack.
`top`	Returns the top element of the stack, but does not remove the top element from the stack. This operation is implemented as a value-returning function.
`pop`	Removes the top element of the stack.

In addition to the operations `size`, `empty`, `push`, `top`, and `pop`, the stack container class provides relational operators to compare two stacks. For example, the relational operator == can be used to determine whether two stacks are identical.

The program in Example 7-5 illustrates how to use the stack container class.

EXAMPLE 7-5

```
//*************************************************************
// Author: D.S. Malik
//
// This program illustrates how to use the STL class stack in
// a program.
//*************************************************************

#include <iostream>                                     //Line 1
#include <stack>                                        //Line 2

using namespace std;                                    //Line 3

int main()                                              //Line 4
{                                                       //Line 5
    stack<int>  intStack;                               //Line 6
    intStack.push(16);                                  //Line 7
    intStack.push(8);                                   //Line 8
    intStack.push(20);                                  //Line 9
    intStack.push(3);                                   //Line 10

    cout << "Line 11: The top element of intStack: "
         << intStack.top() << endl;                     //Line 11

    intStack.pop();                                     //Line 12

    cout << "Line 13: After the pop operation, the "
         << " top element of intStack: "
         << intStack.top() << endl;                     //Line 13

    cout << "Line 14: intStack elements: ";            //Line 14

    while (!intStack.empty())                           //Line 15
    {                                                   //Line 16
        cout << intStack.top() << " ";                 //Line 17
        intStack.pop();                                 //Line 18
    }                                                   //Line 19

    cout << endl;                                       //Line 20

    return 0;                                           //Line 21
}                                                       //Line 22
```

Sample Run:

```
Line 11: The top element of intStack: 3
Line 13: After the pop operation, the top element of intStack: 20
Line 14: intStack elements: 20 8 16
```

The preceding output is self-explanatory. The details are left as an exercise for you.

QUICK REVIEW

1. A stack is a data structure in which the items are added and deleted from one end only.
2. A stack is a Last In First Out (LIFO) data structure.
3. The basic operations on a stack are as follows: Push an item onto the stack, pop an item from the stack, retrieve the top element of the stack, initialize the stack, check whether the stack is empty, and check whether the stack is full.
4. A stack can be implemented as an array or a linked list.
5. The middle elements of a stack should not be accessed directly.
6. Stacks are restricted versions of arrays and linked lists.

7. Postfix notation does not require the use of parentheses to enforce operator precedence.
8. In postfix notation, the operators are written after the operands.
9. Postfix expressions are evaluated according to the following rules:
   a. Scan the expression from left to right.
   b. If an operator is found, back up to get the required number of operands, evaluate the operator, and continue.
10. The STL `class stack` can be used to implement a stack in a program.

EXERCISES

1. Consider the following statements:

```
stackType<int> stack;
int x, y;
```

Show what is output by the following segment of code:

```
x = 4;
y = 0;
stack.push(7);
stack.push(x);
stack.push(x + 5);
y = stack.top();
stack.pop();
stack.push(x + y);
stack.push(y - 2);
stack.push(3);
x = stack.top();
stack.pop();

cout << "x = " << x << endl;
cout << "y = " << y << endl;

while (!stack.isEmptyStack())
{
    cout << stack.top() << endl;
    stack.pop();
}
```

2. Consider the following statements:

```
stackType<int> stack;
int x;
```

Suppose that the input is:

```
14 45 34 23 10 5 -999
```

Show what is output by the following segment of code:

```
stack.push(5);

cin >> x;

while (x != -999)
{
    if (x % 2 == 0)
    {
        if (!stack.isFullStack())
            stack.push(x);
    }
    else
        cout << "x = " << x << endl;
    cin >> x;
}

cout << "Stack Elements: ";

while (!stack.isEmptyStack())
{
    cout << " " << stack.top();
    stack.pop();
}
cout << endl;
```

3. Evaluate the following postfix expressions:
   a. 8 2 + 3 * 16 4 / - =
   b. 12 25 5 1 / / * 8 7 + - =
   c. 70 14 4 5 15 3 / * - - / 6 + =
   d. 3 5 6 * + 13 - 18 2 / + =
4. Convert the following infix expressions to postfix notations:
   a. (A + B) * (C + D) - E
   b. A - (B + C) * D + E / F
   c. ((A + B) / (C - D) + E) * F - G
   d. A + B * (C + D) - E / F * G + H
5. Write the equivalent infix expression for the following postfix expressions:
   a. A B * C +
   b. A B + C D - *
   c. A B - C - D *
6. What is the output of the following program?

```
#include <iostream>
#include <string>
#include "myStack.h"

using namespace std;
```

```
template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t);

int main()
{
    stackType<string> s1;
    stackType<string> s2;

    string list[] = {"Winter", "Spring", "Summer", "Fall",
                     "Cold", "Warm", "Hot"};

    for (int i = 0; i < 7; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}

template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t)
{
    while (!s.isEmptyStack())
    {
        t.push(s.top());
        s.pop();
    }
}
```

7. What is the effect of the following statements? If a statement is invalid, explain why it is invalid. The classes `stackADT`, `stackType`, and `linkedStackType` are as defined in this chapter.
   a. `stackADT<int> newStack;`
   b. `stackType<double> sales(-10);`
   c. `stackType<string> names;`
   d. `linkedStackType<int> numStack(50);`
8. What is the output of the following program?

```
#include <iostream>
#include <string>
#include "myStack.h"

using namespace std;

void mystery(stackType<int>& s, stackType<int>& t);
```

```
int main()
{
    int list[] = {5, 10, 15, 20, 25};

    stackType<int> s1;
    stackType<int> s2;

    for (int i = 0; i < 5; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}

void mystery(stackType<int>& s, stackType<int>& t)
{
    while (!s.isEmptyStack())
    {
        t.push(2 * s.top());
        s.pop();
    }
}
```

9. What is the output of the following program segment?

```
linkedStackType<int> myStack;

myStack.push(10);
myStack.push(20);
myStack.pop();
cout << myStack.top() << endl;
myStack.push(25);
myStack.push(2 * myStack.top());
myStack.push(-10);
myStack.pop();

linkedStackType<int> tempStack;

tempStack = myStack;

while (!tempStack.isEmptyStack())
{
    cout << tempStack.top() << " ";
    tempStack.pop();
}

cout << endl;

cout << myStack.top() << endl;
```

10. Write the definition of the function template `printListReverse` that uses a stack to print a linked list in reverse order. Assume that this function is a member of the `class linkedListType`, designed in Chapter 5.
11. Write the definition of the function template `second` that takes as a parameter a stack object and returns the second element of the stack. The original stack remains unchanged.
12. Draw the UML class diagram of the `class linkedStackType`.
13. Write the definition of the function template `clear` that takes as a parameter a stack object of the type `stack` (STL class) and removes all the elements from the stack.

PROGRAMMING EXERCISES

1. Two stacks of the same type are the same if they have the same number of elements and their elements at the corresponding positions are the same. Overload the relational operator == for the `class stackType` that returns `true` if two stacks of the same type are the same, `false` otherwise. Also, write the definition of the function template to overload this operator.
2. Repeat Exercise 1 for the `class linkedStackType`.
3. a. Add the following operation to the `class stackType`:

   ```
   void reverseStack(stackType<Type> &otherStack);
   ```

   This operation copies the elements of a stack in reverse order onto another stack.

   Consider the following statements:

   ```
   stackType<int> stack1;
   stackType<int> stack2;
   ```

   The statement

   ```
   stack1.reverseStack(stack2);
   ```

   copies the elements of `stack1` onto `stack2` in reverse order. That is, the top element of `stack1` is the bottom element of `stack2`, and so on. The old contents of `stack2` are destroyed and `stack1` is unchanged.

   b. Write the definition of the function `template` to implement the operation `reverseStack`.
4. Repeat Exercises 3a and 3b for the `class linkedStackType`.
5. Write a program that takes as input an arithmetic expression. The program outputs whether the expression contains matching grouping symbols. For example, the arithmetic expressions `{25 + (3 – 6) * 8}` and `7 + 8 * 2` contains matching grouping symbols. However, the expression `5 + {(13 + 7) / 8 - 2 * 9` does not contain matching grouping symbols.

6. Write a program that uses a stack to print the prime factors of a positive integer in descending order.
7. **(Converting a Number from Binary to Decimal)** The language of a computer, called machine language, is a sequence of 0s and 1s. When you press the key A on the keyboard, 01000001 is stored in the computer. Also, the collating sequence of A in the ASCII character set is 65. In fact, the binary representation of A is 01000001 and the decimal representation of A is 65.

   The numbering system we use is called the decimal system, or base 10 system. The numbering system that the computer uses is called the **binary system**, or **base 2 system**. The purpose of this exercise is to write a function to convert a number from base 2 to base 10.

   To convert a number from base 2 to base 10, we first find the weight of each bit in the binary number. The weight of each bit in the binary number is assigned from right to left. The weight of the rightmost bit is 0. The weight of the bit immediately to the left of the rightmost bit is 1, the weight of the bit immediately to the left of it is 2, and so on. Consider the binary number 1001101. The weight of each bit is as follows:

   ```
   weight  6 5 4 3 2 1 0
           1 0 0 1 1 0 1
   ```

   We use the weight of each bit to find the equivalent decimal number. For each bit, we multiply the bit by 2 to the power of its weight, and then we add all of the numbers. For the binary number 1001101, the equivalent decimal number is

   $$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
   $$= 64 + 0 + 0 + 8 + 4 + 0 + 1 = 77$$

   To write a program that converts a binary number into the equivalent decimal number, we note two things: (1) The weight of each bit in the binary number must be known, and (2) the weight is assigned from right to left. Because we do not know in advance how many bits are in the binary number, we must process the bits from right to left. After processing a bit, we can add 1 to its weight, giving the weight of the bit immediately to its left. Also, each bit must be extracted from the binary number and multiplied by 2 to the power of its weight. To extract a bit, you can use the mod operator. Write a program that uses a stack to convert a binary number into an equivalent decimal number and test your function for the following values: 11000101, 10101010, 11111111, 10000000, 1111100000.
8. Chapter 6 described how to use recursion to convert a decimal number into an equivalent binary number. Write a program that uses a stack to convert a decimal number into an equivalent binary number.

9. **(Infix to Postfix)** Write a program that converts an infix expression into an equivalent postfix expression.

   The rules to convert an infix expression into an equivalent postfix expression are as follows:

   Suppose `infx` represents the infix expression and `pfx` represents the postfix expression. The rules to convert `infx` into `pfx` are as follows:

   a. Initialize `pfx` to an empty expression and also initialize the stack.
   b. Get the next symbol, `sym`, from `infx`.
      - b.1. If `sym` is an operand, append `sym` to `pfx`.
      - b.2. If `sym` is `(`, push `sym` into the stack.
      - b.3. If `sym` is `)`, pop and append all the symbols from the stack until the most recent left parenthesis. Pop and discard the left parenthesis.
      - b.4. If `sym` is an operator:
        - b.4.1. Pop and append all the operators from the stack to `pfx` that are above the most recent left parenthesis and have precedence greater than or equal to `sym`.
        - b.4.2. Push `sym` onto the stack.
   c. After processing `infx`, some operators might be left in the stack. Pop and append to `pfx` everything from the stack.

   In this program, you will consider the following (binary) arithmetic operators: `+`, `-`, `*`, and `/`. You may assume that the expressions you will process are error free.

   Design a class that stores the infix and postfix strings. The class must include the following operations:

   - **`getInfix`**—Stores the infix expression
   - **`showInfix`**—Outputs the infix expression
   - **`showPostfix`**—Outputs the postfix expression

   Some other operations that you might need are the following:

   - **`convertToPostfix`**—Converts the infix expression into a postfix expression. The resulting postfix expression is stored in `pfx`.
   - **`precedence`**—Determines the precedence between two operators. If the first operator is of higher or equal precedence than the second operator, it returns the value `true`; otherwise, it returns the value `false`.

   Include the constructors and destructors for automatic initialization and dynamic memory deallocation.

Test your program on the following five expressions:

```
A + B - C;
(A + B ) * C;
(A + B) * (C - D);
A + ((B + C) * (E - F) - G) / (H - I);
A + B * (C + D ) - E / F * G + H;
```

For each expression, your answer must be in the following form:

```
Infix Expression: A + B - C;
Postfix Expression: A B + C -
```

10. Redo the program in the section "Application of Stacks: Postfix Expressions Calculator" of this chapter so that it uses the STL `class stack` to evaluate the postfix expressions.
11. Redo Programming Exercise 9 so that it uses the STL `class stack` to convert the infix expressions to postfix expressions.

CHAPTER 8

QUEUES

IN THIS CHAPTER, YOU WILL:

- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications
- Become aware of the STL `class queue`

This chapter discusses another important data structure, called a **queue**. The notion of a queue in computer science is the same as the notion of the queues to which you are accustomed in everyday life. There are queues of customers in a bank or in a grocery store and queues of cars waiting to pass through a tollbooth. Similarly, because a computer can send a print request faster than a printer can print, a queue of documents is often waiting to be printed at a printer. The general rule to process elements in a queue is that the customer at the front of the queue is served next and that when a new customer arrives, he or she stands at the end of the queue. That is, a queue is a First In First Out data structure.

Queues have numerous applications in computer science. Whenever a system is modeled on the First In First Out principle, queues are used. At the end of this section, we will discuss one of the most widely used applications of queues, computer simulation. First, however, we need to develop the tools necessary to implement a queue. The next few sections discuss how to design classes to implement queues as an ADT.

A queue is a set of elements of the same type in which the elements are added at one end, called the **back** or **rear**, and deleted from the other end, called the **front**. For example, consider a line of customers in a bank, wherein the customers are waiting to withdraw/deposit money or to conduct some other business. Each new customer gets in the line at the rear. Whenever a teller is ready for a new customer, the customer at the front of the line is served.

The rear of the queue is accessed whenever a new element is added to the queue, and the front of the queue is accessed whenever an element is deleted from the queue. As in a stack, the middle elements of the queue are inaccessible, even if the queue elements are stored in an array.

Queue: A data structure in which the elements are added at one end, called the rear, and deleted from the other end, called the front; a First In First Out (FIFO) data structure.

Queue Operations

From the definition of queues, we see that the two key operations are add and delete. We call the add operation **`addQueue`** and the delete operation **`deleteQueue`**. Because elements can be neither deleted from an empty queue nor added to a full queue, we need two more operations to successfully implement the `addQueue` and **`deleteQueue`** operations: **`isEmptyQueue`** (checks whether the queue is empty) and **`isFullQueue`** (checks whether a queue is full).

We also need an operation, **`initializeQueue`**, to initialize the queue to an empty state. Moreover, to retrieve the first and last elements of the queue, we include the operations **`front`** and **`back`** as described in the following list. Some of the queue operations are as follows:

- **`initializeQueue`**—Initializes the queue to an empty state.
- **`isEmptyQueue`**—Determines whether the queue is empty. If the queue is empty, it returns the value `true`; otherwise, it returns the value `false`.

- `isFullQueue`—Determines whether the queue is full. If the queue is full, it returns the value `true`; otherwise, it returns the value `false`.
- `front`—Returns the front, that is, the first element of the queue. Prior to this operation, the queue must exist and must not be empty.
- `back`—Returns the last element of the queue. Prior to this operation, the queue must exist and must not be empty.
- **`addQueue`**—Adds a new element to the rear of the queue. Prior to this operation, the queue must exist and must not be full.
- **`deleteQueue`**—Removes the front element from the queue. Prior to this operation, the queue must exist and must not be empty.

As in the case of a stack, a queue can be stored in an array or in a linked structure. We will consider both implementations. Because elements are added at one end and removed from the other end, we need two pointers to keep track of the front and rear of the queue, called **`queueFront`** and **`queueRear`**.

The following abstract `class queueADT` defines these operations as an ADT:

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operations on a queue.
//*************************************************************

template <class Type>
class queueADT
{
public:
    virtual bool isEmptyQueue() const = 0;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //    otherwise returns false.

    virtual bool isFullQueue() const = 0;
      //Function to determine whether the queue is full.
      //Postcondition: Returns true if the queue is full,
      //    otherwise returns false.

    virtual void initializeQueue() = 0;
      //Function to initialize the queue to an empty state.
      //Postcondition: The queue is empty.

    virtual Type front() const = 0;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the first element of the queue
      //    is returned.
    virtual Type back() const = 0;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the last element of the queue
      //    is returned.

    virtual void addQueue(const Type& queueElement) = 0;
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and queueElement is
      //    added to the queue.

    virtual void deleteQueue() = 0;
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first element
      //    is removed from the queue.
};
```

We leave it as an exercise for you to draw the UML diagram of the `class queueADT`.

Implementation of Queues as Arrays

Before giving the definition of the class to implement a queue as an ADT, we need to decide how many member variables are needed to implement the queue. Of course, we need an array to store the queue elements, the variables `queueFront` and `queueRear` to keep track of the first and last elements of the queue, and the variable `maxQueueSize` to specify the maximum size of the queue. Thus, we need at least four member variables.

Before writing the algorithms to implement the queue operations, we need to decide how to use `queueFront` and `queueRear` to access the queue elements. How do `queueFront` and `queueRear` indicate that the queue is empty or full? Suppose that `queueFront` gives the index of the first element of the queue, and `queueRear` gives the index of the last element of the queue. To add an element to the queue, first we advance `queueRear` to the next array position and then add the element to the position that `queueRear` is pointing to. To delete an element from the queue, first we retrieve the element that `queueFront` is pointing to and then advance `queueFront` to the next element of the queue. Thus, `queueFront` changes after each `deleteQueue` operation and `queueRear` changes after each `addQueue` operation.

Let us see what happens when `queueFront` changes after a `deleteQueue` operation and `queueRear` changes after an `addQueue` operation. Assume that the array to hold the queue elements is of size 100.

Initially, the queue is empty. After the operation:

```
addQueue(Queue,'A');
```

the array is as shown in Figure 8-1.

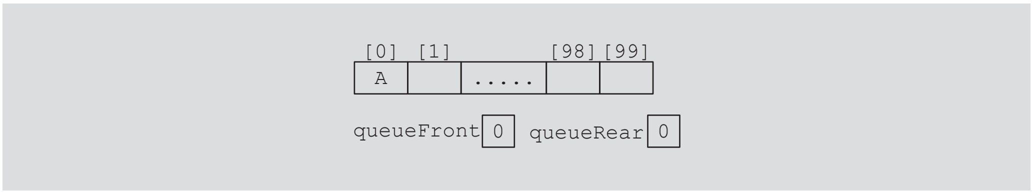


FIGURE 8-1 Queue after the first addQueue operation

After two more addQueue operations:

```
addQueue(Queue,'B');
addQueue(Queue,'C');
```

the array is as shown in Figure 8-2.

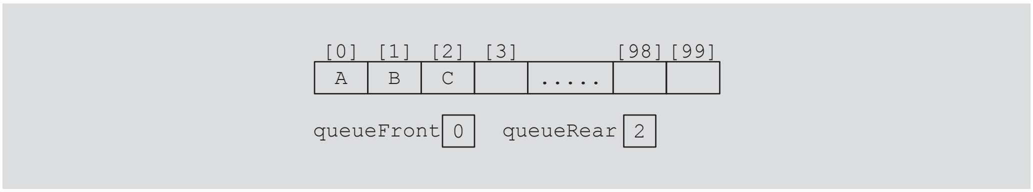


FIGURE 8-2 Queue after two more addQueue operations

Now consider the deleteQueue operation:

```
deleteQueue();
```

After this operation, the array containing the queue is as shown in Figure 8-3.

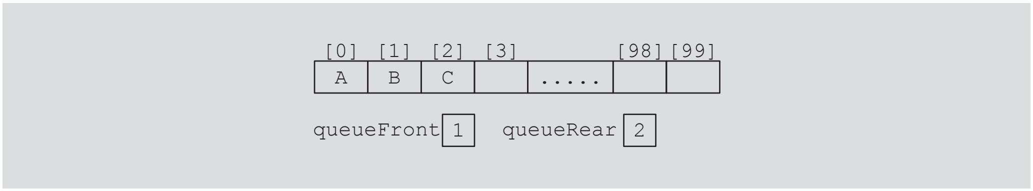


FIGURE 8-3 Queue after the deleteQueue operation

Will this queue design work? Suppose A stands for adding (that is, addQueue) an element to the queue, and D stands for deleting (that is, deleteQueue) an element from the queue. Consider the following sequence of operations:

```
AAADADADADADADADA...
```

This sequence of operations would eventually set the index `queueRear` to point to the last array position, giving the impression that the queue is full. However, the queue has only two or three elements and the front of the array is empty. (See Figure 8-4.)

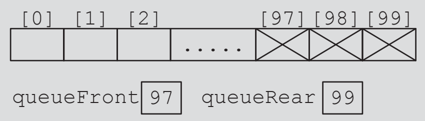


FIGURE 8-4 Queue after the sequence of operations `AAADADADADADA...`

One solution to this problem is that when the queue overflows to the rear (that is, `queueRear` points to the last array position), we can check the value of the index `queueFront`. If the value of `queueFront` indicates that there is room in the front of the array, then when `queueRear` gets to the last array position, we can slide all of the queue elements toward the first array position. This solution is good if the queue size is very small; otherwise, the program might execute more slowly.

Another solution to this problem is to assume that the array is circular—that is, the first array position immediately follows the last array position. (See Figure 8-5.)

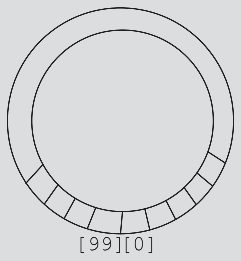


FIGURE 8-5 Circular queue

We will consider the array containing the queue to be circular, although we will draw the figures of the array holding the queue elements as before.

Suppose that we have the queue as shown in Figure 8-6(a).

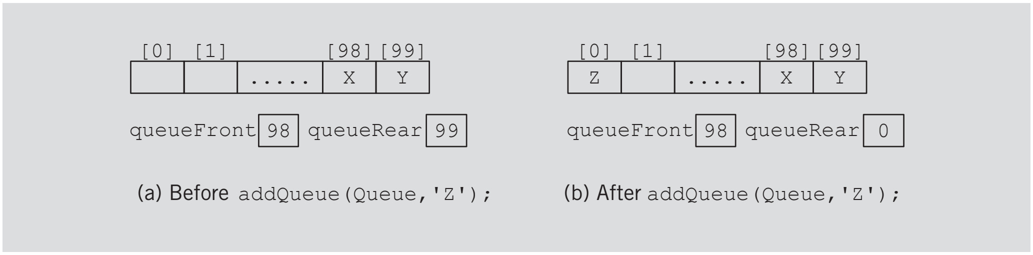

[0] [1] [98] [99]
..... X Y
queueFront 98 queueRear 99
(a) Before addQueue(Queue,'Z');
[0] [1] [98] [99]
Z ..... X Y
queueFront 98 queueRear 0
(b) After addQueue(Queue,'Z');

FIGURE 8-6 Queue before and after the add operation

After the operation `addQueue(Queue,'Z');`, the queue is as shown in Figure 8-6(b).

Because the array containing the queue is circular, we can use the following statement to advance `queueRear` (`queueFront`) to the next array position:

```
queueRear = (queueRear + 1) % maxQueueSize;
```

If `queueRear < maxQueueSize - 1`, then `queueRear + 1 <= maxQueueSize - 1`, so `(queueRear + 1) % maxQueueSize = queueRear + 1`. If `queueRear == maxQueueSize - 1` (that is, `queueRear` points to the last array position), `queueRear + 1 == maxQueueSize`, so `(queueRear + 1) % maxQueueSize = 0`. In this case, `queueRear` will be set to `0`, which is the first array position.

This queue design seems to work well. Before we write the algorithms to implement the queue operations, consider the following two cases.

Case 1: Suppose that after certain operations, the array containing the queue is as shown in Figure 8-7(a).

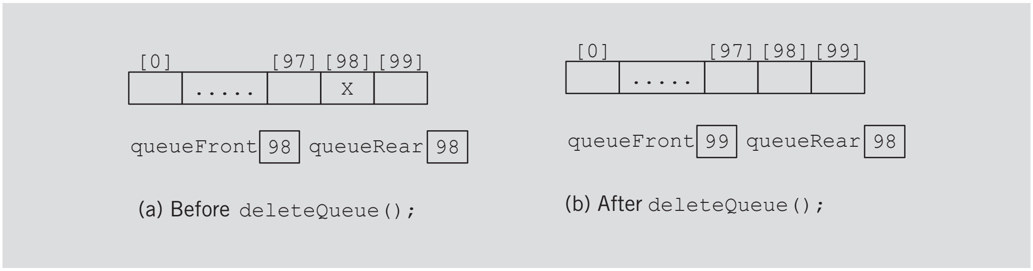

[0] [97] [98] [99]
..... X
queueFront 98 queueRear 98
(a) Before deleteQueue();
[0] [97] [98] [99]
.....
queueFront 99 queueRear 98
(b) After deleteQueue();

FIGURE 8-7 Queue before and after the delete operation

After the operation `deleteQueue();`, the resulting array is as shown in Figure 8-7(b).

Case 2: Let us now consider the queue shown in Figure 8-8(a).

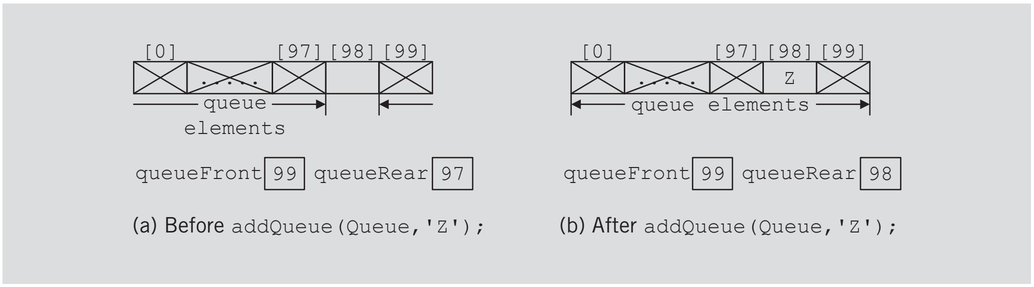


FIGURE 8-8 Queue before and after the add operation

After the operation `addQueue(Queue,'Z');`, the resulting array is as shown in Figure 8-8(b).

The arrays in Figures 8-7(b) and 8-8(b) have identical values for `queueFront` and `queueRear`. However, the resulting array in Figure 8-7(b) represents an empty queue, whereas the resulting array in Figure 8-8(b) represents a full queue. This latest queue design has brought up another problem of distinguishing between an empty and a full queue.

This problem has several solutions. One solution is to keep a count. In addition to the member variables `queueFront` and `queueRear`, we need another variable, `count`, to implement the queue. The value of `count` is incremented whenever a new element is added to the queue, and it is decremented whenever an element is removed from the queue. In this case, the function `initializeQueue` initializes `count` to `0`. This solution is very useful if the user of the queue frequently needs to know the number of elements in the queue.

Another solution is to let `queueFront` indicate the index of the array position *preceding* the first element of the queue, rather than the index of the (actual) first element itself. In this case, assuming `queueRear` still indicates the index of the last element in the queue, the queue is empty if `queueFront == queueRear`. In this solution, the slot indicated by the index `queueFront` (that is, the slot preceding the first true element) is reserved. The queue will be full if the next available space is the special reserved slot indicated by `queueFront`. Finally, because the array position indicated by `queueFront` is to be kept empty, if the array size is, say, `100`, then 99 elements can be stored in the queue. (See Figure 8-9.)

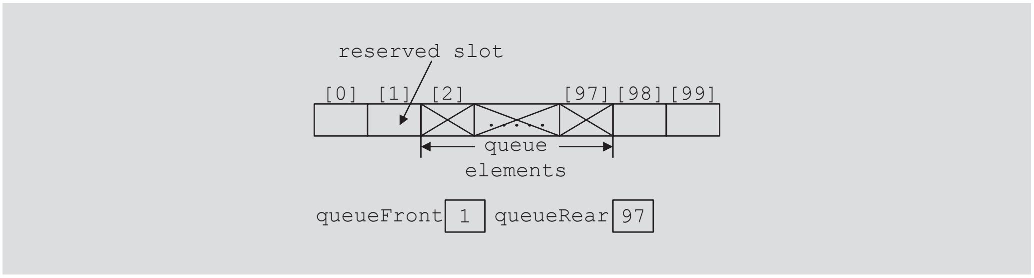


FIGURE 8-9 Array to store the queue elements with a reserved slot

Let us implement the queue using the first solution. That is, we use the variable `count` to indicate whether the queue is empty or full.

The following class implements the functions of the abstract `class queueADT`. Because arrays can be allocated dynamically, we will leave it for the user to specify the size of the array to implement the queue. The default size of the array is `100`.

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operation on a queue as an
// array.
//*************************************************************

template <class Type>
class queueType: public queueADT<Type>
{
public:
    const queueType<Type>& operator=(const queueType<Type>&);
      //Overload the assignment operator.

    bool isEmptyQueue() const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //    otherwise returns false.

    bool isFullQueue() const;
      //Function to determine whether the queue is full.
      //Postcondition: Returns true if the queue is full,
      //    otherwise returns false.

    void initializeQueue();
      //Function to initialize the queue to an empty state.
      //Postcondition: The queue is empty.

    Type front() const;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the first element of the
      //    queue is returned.

    Type back() const;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the last element of the queue
      //    is returned.

    void addQueue(const Type& queueElement);
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and queueElement is
      //    added to the queue.

    void deleteQueue();
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first element
      //    is removed from the queue.

    queueType(int queueSize = 100);
      //Constructor

    queueType(const queueType<Type>& otherQueue);
      //Copy constructor

    ~queueType();
      //Destructor

private:
    int maxQueueSize; //variable to store the maximum queue size
    int count;        //variable to store the number of
                      //elements in the queue
    int queueFront;   //variable to point to the first
                      //element of the queue
    int queueRear;    //variable to point to the last
                      //element of the queue
    Type *list;       //pointer to the array that holds
                      //the queue elements
};
```

We leave the UML diagram of the `class queueType` as an exercise for you. (See Exercise 15 at the end of this chapter.)

Next, we consider the implementation of the queue operations.

Empty Queue and Full Queue

As discussed earlier, the queue is empty if `count == 0`, and the queue is full if `count == maxQueueSize`. So the functions to implement these operations are as follows:

```
template <class Type>
bool queueType<Type>::isEmptyQueue() const
{
    return (count == 0);
} //end isEmptyQueue

template <class Type>
bool queueType<Type>::isFullQueue() const
{
    return (count == maxQueueSize);
} //end isFullQueue
```

Initialize Queue

This operation initializes a queue to an empty state. The first element is added at the first array position. Therefore, we initialize `queueFront` to `0`, `queueRear` to `maxQueueSize - 1`, and `count` to `0`. See Figure 8-10.

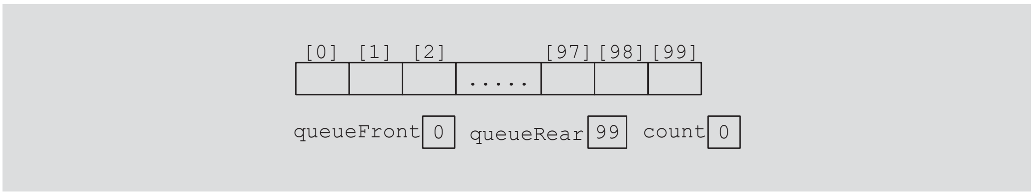


FIGURE 8-10 Empty queue

The definition of the function `initializeQueue` is as follows:

```
template <class Type>
void queueType<Type>::initializeQueue()
{
    queueFront = 0;
    queueRear = maxQueueSize - 1;
    count = 0;
} //end initializeQueue
```

Front

This operation returns the first element of the queue. If the queue is nonempty, the element of the queue indicated by the index `queueFront` is returned; otherwise, the program terminates.

```
template <class Type>
Type queueType<Type>::front() const
{
    assert(!isEmptyQueue());
    return list[queueFront];
} //end front
```

Back

This operation returns the last element of the queue. If the queue is nonempty, the element of the queue indicated by the index `queueRear` is returned; otherwise, the program terminates.

```
template <class Type>
Type queueType<Type>::back() const
{
    assert(!isEmptyQueue());
    return list[queueRear];
} //end back
```

Add Queue

Next, we implement the addQueue operation. Because queueRear points to the last element of the queue, to add a new element to the queue, we first advance queueRear to the next array position, and then add the new element to the array position indicated by queueRear. We also increment count by 1. So the function addQueue is as follows:

```
template <class Type>
void queueType<Type>::addQueue(const Type& newElement)
{
    if (!isFullQueue())
    {
        queueRear = (queueRear + 1) % maxQueueSize; //use the
                            //mod operator to advance queueRear
                            //because the array is circular
        count++;
        list[queueRear] = newElement;
    }
    else
        cout << "Cannot add to a full queue." << endl;
} //end addQueue
```

Delete Queue

To implement the deleteQueue operation, we access the index queueFront. Because queueFront points to the array position containing the first element of the queue, to remove the first queue element, we decrement count by 1 and advance queueFront to the next queue element. So the function deleteQueue is as follows:

```
template <class Type>
void queueType<Type>::deleteQueue()
{
    if (!isEmptyQueue())
    {
        count--;
        queueFront = (queueFront + 1) % maxQueueSize; //use the
                        //mod operator to advance queueFront
                        //because the array is circular
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
} //end deleteQueue
```

Constructors and Destructors

To complete the implementation of the queue operations, we next consider the implementation of the constructor and the destructor. The constructor gets the maxQueueSize from the user, sets the variable maxQueueSize to the value specified by the user, and creates an array of size maxQueueSize. If the user does not specify the queue size, the constructor uses the default value, which is 100, to create an array of size 100. The constructor also initializes `queueFront` and `queueRear` to indicate that the queue is empty. The definition of the function to implement the constructor is as follows:

```
template <class Type>
queueType<Type>::queueType(int queueSize)
{
    if (queueSize <= 0)
    {
        cout << "Size of the array to hold the queue must "
             << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxQueueSize = 100;
    }
    else
        maxQueueSize = queueSize;   //set maxQueueSize to
                                    //queueSize

    queueFront = 0;                 //initialize queueFront
    queueRear = maxQueueSize - 1;   //initialize queueRear
    count = 0;
    list = new Type[maxQueueSize];  //create the array to
                                    //hold the queue elements
} //end constructor
```

The array to store the queue elements is created dynamically. Therefore, when the queue object goes out of scope, the destructor simply deallocates the memory occupied by the array that stores the queue elements. The definition of the function to implement the destructor is as follows:

```
template <class Type>
queueType<Type>::~queueType()
{
    delete [] list;
}
```

The implementation of the copy constructor and overloading the assignment operator are left as exercises for you, (see Programming Exercise 1 at the end of this chapter). (The definitions of these functions are similar to those discussed for array-based lists and stacks.)

Linked Implementation of Queues

Because the size of the array to store the queue elements is fixed, only a finite number of queue elements can be stored in the array. Also, the array implementation of the queue requires the array to be treated in a special way together with the values of the indices `queueFront` and `queueRear`. The linked implementation of a queue simplifies many of the special cases of the array implementation and, because the memory to store a queue element is allocated dynamically, the queue is never full. This section discusses the linked implementation of a queue.

Because elements are added at one end, queueRear, and removed from the other end, queueFront, we need to know the front of the queue and the rear of the queue. Thus, we need two pointers, **queueFront** and **queueRear**, to maintain the queue. The following class implements the functions of the abstract class queueADT:

```
//*************************************************************
// Author: D.S. Malik
//
// This class specifies the basic operations on a queue as a
// linked list.
//*************************************************************

//Definition of the node
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

template <class Type>
class linkedQueueType: public queueADT<Type>
{
public:
    const linkedQueueType<Type>& operator=
                    (const linkedQueueType<Type>&);
      //Overload the assignment operator.

    bool isEmptyQueue() const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //    otherwise returns false.

    bool isFullQueue() const;
      //Function to determine whether the queue is full.
      //Postcondition: Returns true if the queue is full,
      //    otherwise returns false.

    void initializeQueue();
      //Function to initialize the queue to an empty state.
      //Postcondition: queueFront = NULL; queueRear = NULL

    Type front() const;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the first element of the
      //    queue is returned.

    Type back() const;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //    terminates; otherwise, the last element of the
      //    queue is returned.

    void addQueue(const Type& queueElement);
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and queueElement is
      //    added to the queue.

    void deleteQueue();
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first element
      //    is removed from the queue.

    linkedQueueType();
      //Default constructor

    linkedQueueType(const linkedQueueType<Type>& otherQueue);
      //Copy constructor

    ~linkedQueueType();
      //Destructor

private:
    nodeType<Type> *queueFront; //pointer to the front of the queue
    nodeType<Type> *queueRear;  //pointer to the rear of the queue
};
```

The UML diagram of the `class linkedQueueType` is left as an exercise for you. (See Exercise 16 at the end of this chapter.)

Next, we write the definitions of the functions of the `class linkedQueueType`.

Empty and Full Queue

The queue is empty if `queueFront` is `NULL`. Memory to store the queue elements is allocated dynamically. Therefore, the queue is never full and so the function to implement the `isFullQueue` operation returns the value `false`. (The queue is full only if the program runs out of memory.)

```
template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
    return(queueFront == NULL);
} //end

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{
    return false;
} //end isFullQueue
```

Note that in reality, in the linked implementation of queues, the function `isFullQueue` does not apply because logically the queue is never full. However, you must provide its definition because it is included as an abstract function in the parent `class queueADT`.

Initialize Queue

The operation `initializeQueue` initializes the queue to an empty state. The queue is empty if there are no elements in the queue. Note that the constructor initializes the queue when the queue object is declared. So this operation must remove all the elements, if any, from the queue. Therefore, this operation traverses the list containing the queue starting at the first node, and it deallocates the memory occupied by the queue elements. The definition of this function is as follows:

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
    nodeType<Type> *temp;

    while (queueFront!= NULL)  //while there are elements left
                               //in the queue
    {
        temp = queueFront;  //set temp to point to the current node
        queueFront = queueFront->link;  //advance first to
                                        //the next node
        delete temp;    //deallocate memory occupied by temp
    }

    queueRear = NULL;  //set rear to NULL
} //end initializeQueue
```

addQueue, front, back, and deleteQueue Operations

The `addQueue` operation adds a new element at the end of the queue. To implement this operation, we access the pointer `queueRear`.

If the queue is nonempty, the operation `front` returns the first element of the queue and so the element of the queue indicated by the pointer `queueFront` is returned. If the queue is empty, the function `front` terminates the program.

If the queue is nonempty, the operation `back` returns the last element of the queue and so the element of the queue indicated by the pointer `queueRear` is returned. If the queue is empty, the function `back` terminates the program. Similarly, if the queue is nonempty, the operation `deleteQueue` removes the first element of the queue, and so we access the pointer `queueFront`.

The definitions of the functions to implement these operations are as follows:

```
template <class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
```

```
{
    nodeType<Type> *newNode;

    newNode = new nodeType<Type>;   //create the node

    newNode->info = newElement; //store the info
    newNode->link = NULL;  //initialize the link field to NULL

    if (queueFront == NULL) //if initially the queue is empty
    {
        queueFront = newNode;
        queueRear = newNode;
    }
    else        //add newNode at the end
    {
        queueRear->link = newNode;
        queueRear = queueRear->link;
    }
}//end addQueue

template <class Type>
Type linkedQueueType<Type>::front() const
{
    assert(queueFront != NULL);
    return queueFront->info;
} //end front

template <class Type>
Type linkedQueueType<Type>::back() const
{
    assert(queueRear!= NULL);
    return queueRear->info;
} //end back

template <class Type>
void linkedQueueType<Type>::deleteQueue()
{
    nodeType<Type> *temp;

    if (!isEmptyQueue())
    {
        temp = queueFront;  //make temp point to the first node
        queueFront = queueFront->link; //advance queueFront

        delete temp;    //delete the first node

        if (queueFront == NULL) //if after deletion the
                                //queue is empty
            queueRear = NULL;   //set queueRear to NULL
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
}//end deleteQueue
```

The definition of the default constructor is as follows:

```
template<class Type>
linkedQueueType<Type>::linkedQueueType()
{
    queueFront = NULL; //set front to null
    queueRear = NULL;  //set rear to null
} //end default constructor
```

When the queue object goes out of scope, the destructor destroys the queue; that is, it deallocates the memory occupied by the elements of the queue. The definition of the function to implement the destructor is similar to the definition of the function `initializeQueue`. Also, the functions to implement the copy constructor and overload the assignment operators are similar to the corresponding functions for stacks. Implementing these operations is left as an exercise for you, (see Programming Exercise 2 at the end of this chapter).

EXAMPLE 8-1

The following program tests various operations on a queue. It uses the `class linkedQueueType` to implement a queue.

```
//*************************************************************
// Author: D.S. Malik
//
// This program illustrates how to use the class linkedQueueType
// in a program.
//*************************************************************

#include <iostream>
#include "linkedQueue.h"

using namespace std;

int main()
{
    linkedQueueType<int> queue;
    int x, y;

    queue.initializeQueue();
    x = 4;
    y = 5;
    queue.addQueue(x);
    queue.addQueue(y);
    x = queue.front();
    queue.deleteQueue();
    queue.addQueue(x + 5);
    queue.addQueue(16);
    queue.addQueue(x);
    queue.addQueue(y - 3);

    cout << "Queue Elements: ";
    while (!queue.isEmptyQueue())
    {
        cout << queue.front() << " ";
        queue.deleteQueue();
    }

    cout << endl;

    return 0;
}
```

Sample Run:

```
Queue Elements: 5 9 16 4 2
```

Queue Derived from the `class unorderedLinkedList`

From the definitions of the functions to implement the queue operations, it is clear that the linked implementation of a queue is similar to the implementation of a linked list created in a forward manner (see Chapter 5). The `addQueue` operation is similar to the operation `insertFirst`. Likewise, the operations `initializeQueue` and `initializeList`, `isEmptyQueue`, and `isEmptyList` are similar. The `deleteQueue` operation can be implemented as before. The pointer `queueFront` is the same as the pointer `first`, and the pointer `queueRear` is the same as the pointer `last`. This correspondence suggests that we can derive the class to implement the queue from the `class linkedListType` (see Chapter 5). Note that the `class linkedListType` is an abstract and does not implement all the operations. However, the `class unorderedLinkedList` is derived from the the `class linkedListType` and provides the definitions of the abstract functions of the the `class linkedListType`. Therefore, we can derive the `class linkedQueueType` from the `class unorderedLinkedList`.

We leave it as an exercise for you to write the definition of the `class linkedQueueType` that is derived from the `class unorderedLinkedList` (see Programming Exercise 7 at the end of this chapter).

STL **`class queue`** (Queue Container Adapter)

The preceding sections discussed the data structure queue in detail. Because a queue is an important data structure, the Standard Template Library (STL) provides a class to implement queues in a program. The name of the class defining the queue is `queue`, and the name of the header file containing the definition of the `class queue` is `queue`. The `class queue` provided by the STL is implemented similar to the classes discussed in this chapter. Table 8-1 defines various operations supported by the queue container class.

TABLE 8-1 Operations on a `queue` object

Operation	Effect
`size`	Returns the actual number of elements in the queue.
`empty`	Returns **`true`** if the queue is empty, and **`false`** otherwise.
`push(item)`	Inserts a copy of `item` into the queue.
`front`	Returns the next—that is, first—element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function.
`back`	Returns the last element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function.
`pop`	Removes the next element in the queue.

In addition to the operations `size`, `empty`, `push`, `front`, `back`, and `pop`, the queue container class provides relational operators to compare two queues. For example, the relational operator == can be used to determine whether two queues are identical.

The program in Example 8-2 illustrates how to use the queue container class.

EXAMPLE 8-2

```
//*************************************************************
// Author: D.S. Malik
//
// This program illustrates how to use the STL class queue in a
// program.
//*************************************************************

#include <iostream>                                     //Line 1
#include <queue>                                        //Line 2

using namespace std;                                    //Line 3

int main()                                              //Line 4
{                                                       //Line 5
    queue<int> intQueue;                                //Line 6

    intQueue.push(26);                                  //Line 7
    intQueue.push(18);                                  //Line 8
    intQueue.push(50);                                  //Line 9
    intQueue.push(33);                                  //Line 10
    cout << "Line 11: The front element of intQueue: "
         << intQueue.front() << endl;                    //Line 11

    cout << "Line 12: The last element of intQueue: "
         << intQueue.back() << endl;                     //Line 12

    intQueue.pop();                                      //Line 13

    cout << "Line 14: After the pop operation, the "
         << "front element of intQueue: "
         << intQueue.front() << endl;                    //Line 14

    cout << "Line 15: intQueue elements: ";              //Line 15

    while (!intQueue.empty())                            //Line 16
    {                                                    //Line 17
        cout << intQueue.front() << " ";                 //Line 18
        intQueue.pop();                                  //Line 19
    }                                                    //Line 20

    cout << endl;                                        //Line 21

    return 0;                                            //Line 22
}                                                        //Line 23
```

Sample Run:

```
Line 11: The front element of intQueue: 26
Line 12: The last element of intQueue: 33
Line 14: After the pop operation, the front element of intQueue: 18
Line 15: intQueue elements: 18 50 33
```

The preceding output is self-explanatory. The details are left as an exercise for you.

Priority Queues

The preceding sections describe how to implement a queue in a program. The use of a queue structure ensures that the items are processed in the order they are received. For example, in a banking environment, the customers who arrive first are served first. However, there are certain situations when this First In First Out rule needs to be relaxed somewhat. In a hospital environment, patients are, usually, seen in the order they arrive. Therefore, you could use a queue to ensure that the patients are seen in the order they arrive. However, if a patient arrives with severe or life-threatening symptoms, they are treated first. In other words, these patients take priority over the patients who can wait to be seen, such as those awaiting their routine annual checkup. For another example, in a shared environment, when print requests are sent to the printer, interactive programs take priority over batch-processing programs.

There are many other situations where some priority is assigned to the customers. To implement such a data structure in a program, we use a special type of queue, called **priority queues**. In a priority queue, customers or jobs with higher priority are pushed to the front of the queue.

One way to implement a priority queue is to use an ordinary linked list, which keeps the items in order from the highest to lowest priority. However, an effective way to implement a priority queue is to use a treelike structure. In Chapter 10, we discuss a special type of sorting algorithm, called the *heapsort*, which uses a treelike structure to sort a list. After describing this algorithm, we discuss how to effectively implement a priority queue.

STL `class priority_queue`

The STL provides the `class` template `priority_queue<elemType>`, where the data type of the queue elements is specified by `elemType`. This class template is contained in the STL header file `queue`. You can specify the priority of the elements of a priority queue in various ways. The default priority criteria for the queue elements uses the less-than operator, <. For example, a program that implements a priority queue of numbers could use the operator < to assign the priority to the numbers so that larger numbers are always at the front of the queue. If you design your own class to implement the queue elements, you can specify your priority rule by overloading the less-than operator, <, to compare the elements. You could also define a comparison function to specify the priority. The implementation of comparison functions is discussed in Chapter 13.

Application of Queues: Simulation

A technique in which one system models the behavior of another system is called **simulation**. For example, physical simulators include wind tunnels used to experiment with the design of car bodies and flight simulators are used to train airline pilots. Simulation techniques are used when it is too expensive or dangerous to experiment with real systems. You can also design computer models to study the behavior of real systems. (We will describe some real systems modeled by computers shortly.) Simulating the behavior of an expensive or dangerous experiment using a computer model is usually less expensive than using the real system, and a good way to gain insight without putting human life in danger. Moreover, computer simulations are particularly useful for complex systems where it is difficult to construct a mathematical model. For such systems, computer models can retain descriptive accuracy. In mathematical simulations, the steps of a program are used to model the behavior of a real system. Let us consider one such problem.

The manager of a local movie theater is hearing complaints from customers about the time they have to wait in line to buy tickets. The theater currently has only one cashier. Another theater is preparing to open in the neighborhood and the manager is afraid of losing customers. The manager wants to hire enough cashiers so that a customer does not have to wait too long to buy a ticket, but does not want to hire extra cashiers on a trial basis and potentially waste time and money. One thing that the manager would like to know is the average time a customer has to wait for service. The manager wants someone to write a program to simulate the behavior of the theater.

In computer simulation, the objects being studied are usually represented as data. For the theater problem, some of the objects are the customers and the cashier. The cashier serves the customers and we want to determine a customer's average waiting time. Actions are implemented by writing algorithms, which in a programming language are implemented with the help of functions. Thus, functions are used to implement the actions of the objects. In C++, we can combine the data and the operations on that data into a single unit with the help of classes. Thus, objects can be represented as classes. The member variables of the class describe the properties of the objects, and the function members describe the actions on that data. This change in simulation results can also occur if we change the values of the data or modify the definitions of the functions (that is, modify the algorithms implementing the actions). The main goal of a computer simulation is to either generate results showing the performance of an existing system or predict the performance of a proposed system.

In the theater problem, when the cashier is serving a customer, the other customers must wait. Because customers are served on a first-come, first-served basis and queues are an effective way to implement a First In First Out system, queues are important data structures for use in computer simulations. This section examines computer simulations in which queues are the basic data structure. These simulations model the behavior of systems, called **queuing systems**, in which queues of objects are waiting to be served by various servers. In other words, a queuing system consists of servers and queues of objects waiting to be served. We deal with a variety of queuing systems on a daily basis. For example, a grocery store and a banking system are both queuing systems. Furthermore, when you send a print request to a networked printer that is shared by many people, your print request goes in a queue. Print requests that arrived before your print request are usually completed before yours. Thus, the printer acts as the server when a queue of documents is waiting to be printed.

Designing a Queuing System

In this section, we describe a queuing system that can be used in a variety of applications, such as a bank, grocery store, movie theater, printer, or mainframe environment in which several people are trying to use the same processors to execute their programs. To describe a queuing system, we use the term **server** for the object that provides the service. For example, in a bank, a teller is a server; in a grocery store or movie theater, a cashier is a server. We will call the object receiving the service the **customer**, and the service time—the time it takes to serve a customer—the **transaction time**.

Because a queuing system consists of servers and a queue of waiting objects, we will model a system that consists of a list of servers and a waiting queue holding the customers to be served. The customer at the front of the queue waits for the next available server. When a server becomes free, the customer at the front of the queue moves to the free server to be served.

When the first customer arrives, all servers are free and the customer moves to the first server. When the next customer arrives, if a server is available, the customer immediately moves to the available server; otherwise, the customer waits in the queue. To model a queuing system, we need to know the number of servers, the expected arrival time of a customer, the time between the arrivals of customers, and the number of events affecting the system.

Let us again consider the movie theater system. The performance of the system depends on how many servers are available, how long it takes to serve a customer, and how often a customer arrives. If it takes too long to serve a customer and customers arrive frequently, then more servers are needed. This system can be modeled as a time-driven simulation. In a **time-driven simulation**, the clock is implemented as a counter and the passage of, say, 1 minute can be implemented by incrementing the counter by 1. The simulation is run for a fixed amount of time. If the simulation needs to be run for 100 minutes, the counter starts at 1 and goes up to 100, which can be implemented by using a loop.

For the simulation described in this section, we want to determine the average wait time for a customer. To calculate the average wait time for a customer, we need to add the waiting time of each customer, and then divide the sum by the number of customers who have arrived. When a customer arrives, he or she goes to the end of the queue and the customer's waiting time starts. If the queue is empty and a server is free, the customer is served immediately and so this customer's waiting time is zero. On the other hand, if when the customer arrives and either the queue is nonempty or all the servers are busy, the customer must wait for the next available server and, therefore, this customer's waiting time starts. We can keep track of the customer's waiting time by using a timer for each customer. When a customer arrives, the timer is set to 0, which is incremented after each clock unit.

Suppose that, on average, it takes five minutes for a server to serve a customer. When a server becomes free and the waiting customer's queue is nonempty, the customer at the front of the queue proceeds to begin the transaction. Thus, we must keep track of the time a customer is with a server. When the customer arrives at a server, the transaction time is set to five and is decremented after each clock unit. When the transaction time becomes zero, the server is marked free. Hence, the two objects needed to implement a time-driven computer simulation of a queuing system are the customer and the server.

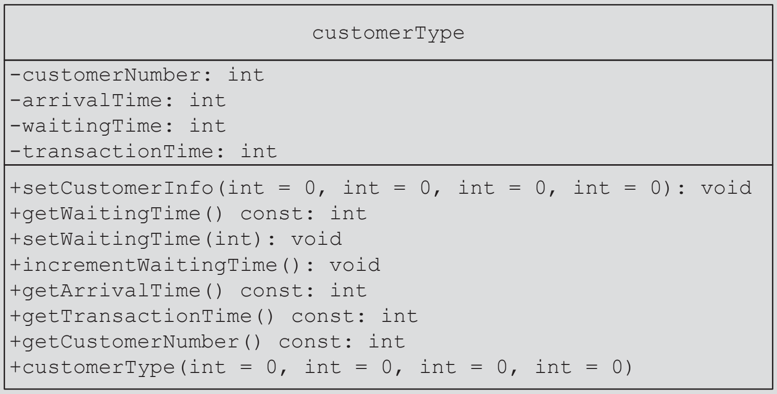
Before designing the main algorithm to implement the simulation, we design classes to implement each of the two objects: *customer* and *server*.

Customer

Every customer has a customer number, arrival time, waiting time, transaction time, and departure time. If we know the arrival time, waiting time, and transaction time, we can determine the departure time by adding these three times. Let us call the class to implement the customer object `customerType`. It follows that the `class customerType` has four member variables: the `customerNumber`, `arrivalTime`, `waitingTime`, and `transactionTime`, each of type int. The basic operations that must be performed on an object of type customerType are as follows: Set the customer's number, arrival time, and waiting time; increment the waiting time by one clock unit; return the waiting time; return the arrival time; return the transaction time; and return the customer number. The following class, customerType, implements the customer as an ADT:

```
//*************************************************************
// Author: D.S. Malik
//
// class customerType
// This class specifies the members to implement a customer.
//*************************************************************

class customerType
{
public:
    customerType(int cN = 0, int arrvTime = 0, int wTime = 0,
                 int tTime = 0);
      //Constructor to initialize the instance variables
      //according to the parameters
      //If no value is specified in the object declaration,
      //the default values are assigned.
      //Postcondition: customerNumber = cN; arrivalTime = arrvTime;
      //    waitingTime = wTime; transactionTime = tTime

    void setCustomerInfo(int cN = 0, int inTime = 0,
                         int wTime = 0, int tTime = 0);
      //Function to initialize the instance variables.
      //Instance variables are set according to the parameters.
      //Postcondition: customerNumber = cN; arrivalTime = arrvTime;
      //    waitingTime = wTime; transactionTime = tTime;

    int getWaitingTime() const;
      //Function to return the waiting time of a customer.
      //Postcondition: The value of waitingTime is returned.

    void setWaitingTime(int time);
      //Function to set the waiting time of a customer.
      //Postcondition: waitingTime = time;

    void incrementWaitingTime();
      //Function to increment the waiting time by one time unit.
      //Postcondition: waitingTime++;

    int getArrivalTime() const;
      //Function to return the arrival time of a customer.
      //Postcondition: The value of arrivalTime is returned.

    int getTransactionTime() const;
      //Function to return the transaction time of a customer.
      //Postcondition: The value of transactionTime is returned.
    int getCustomerNumber() const;
      //Function to return the customer number.
      //Postcondition: The value of customerNumber is returned.

private:
    int customerNumber;
    int arrivalTime;
    int waitingTime;
    int transactionTime;
};
```

Figure 8-11 shows the UML diagram of the `class customerType`.



```
customerType
-customerNumber: int
-arrivalTime: int
-waitingTime: int
-transactionTime: int
+setCustomerInfo(int = 0, int = 0, int = 0, int = 0): void
+getWaitingTime() const: int
+setWaitingTime(int): void
+incrementWaitingTime(): void
+getArrivalTime() const: int
+getTransactionTime() const: int
+getCustomerNumber() const: int
+customerType(int = 0, int = 0, int = 0, int = 0)
```

FIGURE 8-11 UML diagram of the `class customerType`

The definitions of the member functions of the `class customerType` follow easily from their descriptions. Next, we give the definitions of the member functions of the `class customerType`.

The function `setCustomerInfo` uses the values of the parameters to initialize `customerNumber`, `arrivalTime`, `waitingTime`, and `transactionTime`. Its definition is as follows:

```
void customerType::setCustomerInfo(int cN, int arrvTime,
                                   int wTime, int tTime)
{
    customerNumber = cN;
    arrivalTime = arrvTime;
    waitingTime = wTime;
    transactionTime = tTime;
}
```

The definition of the constructor is similar to the definition of the function `setCustomerInfo`. It uses the values of the parameters to initialize `customerNumber`, arrivalTime, waitingTime, and transactionTime. To make debugging easier, we use the function setCustomerInfo to write the definition of the constructor, which is given next.

```
customerType::customerType(int cN, int arrvTime,
                           int wTime, int tTime)
{
    setCustomerInfo(cN, arrvTime, wTime, tTime);
}
```

The function getWaitingTime returns the current waiting time. The definition of the function getWaitingTime is as follows:

```
int customerType::getWaitingTime() const
{
    return waitingTime;
}
```

The function incrementWaitingTime increments the value of waitingTime. Its definition is as follows:

```
void customerType::incrementWaitingTime()
{
    waitingTime++;
}
```

The definitions of the functions setWaitingTime, getArrivalTime, getTransactionTime, and getCustomerNumber are left as an exercise for you, (see Programming Exercise 8 a the end of this chapter).

Server

At any given time unit, the server is either busy serving a customer or is free. We use a string variable to set the status of the server. Every server has a timer and because the program might need to know which customer is served by which server, the server also stores the information of the customer being served. Thus, three member variables are associated with a server: the status, the transactionTime, and the currentCustomer. Some of the basic operations that must be performed on a server are as follows: Check whether the server is free; set the server as free; set the server as busy; set the transaction time (that is, how long it takes to serve the customer); return the remaining transaction time (to determine whether the server should be set to free); if the server is busy after each time unit, decrement the transaction time by one time unit; and so on. The following class, serverType, implements the server as an ADT:

```
//*************************************************************
// Author: D.S. Malik
//
// class serverType
// This class specifies the members to implement a server.
//*************************************************************
```

```
class serverType
{
public:
    serverType();
      //Default constructor
      //Sets the values of the instance variables to their default
      //values.
      //Postcondition: currentCustomer is initialized by its
      //    default constructor; status = "free"; and the
      //    transaction time is initialized to 0.

    bool isFree() const;
      //Function to determine if the server is free.
      //Postcondition: Returns true if the server is free,
      //    otherwise returns false.

    void setBusy();
      //Function to set the status of the server to busy.
      //Postcondition: status = "busy";

    void setFree();
      //Function to set the status of the server to "free."
      //Postcondition: status = "free";

    void setTransactionTime(int t);
      //Function to set the transaction time according to the
      //parameter t.
      //Postcondition: transactionTime = t;

    void setTransactionTime();
      //Function to set the transaction time according to
      //the transaction time of the current customer.
      //Postcondition:
      //   transactionTime = currentCustomer.transactionTime;

    int getRemainingTransactionTime() const;
      //Function to return the remaining transaction time.
      //Postcondition: The value of transactionTime is returned.

    void decreaseTransactionTime();
      //Function to decrease the transactionTime by 1 unit.
      //Postcondition: transactionTime--;

    void setCurrentCustomer(customerType cCustomer);
      //Function to set the info of the current customer
      //according to the parameter cCustomer.
      //Postcondition: currentCustomer = cCustomer;

    int getCurrentCustomerNumber() const;
      //Function to return the customer number of the current
      //customer.
      //Postcondition: The value of customerNumber of the
      //    current customer is returned.
    int getCurrentCustomerArrivalTime() const;
      //Function to return the arrival time of the current
      //customer.
      //Postcondition: The value of arrivalTime of the current
      //    customer is returned.

    int getCurrentCustomerWaitingTime() const;
      //Function to return the current waiting time of the
      //current customer.
      //Postcondition: The value of transactionTime is returned.

    int getCurrentCustomerTransactionTime() const;
      //Function to return the transaction time of the
      //current customer.
      //Postcondition: The value of transactionTime of the
      //    current customer is returned.

private:
    customerType currentCustomer;
    string status;
    int transactionTime;
};
```

Figure 8-12 shows the UML diagram of the `class serverType`.

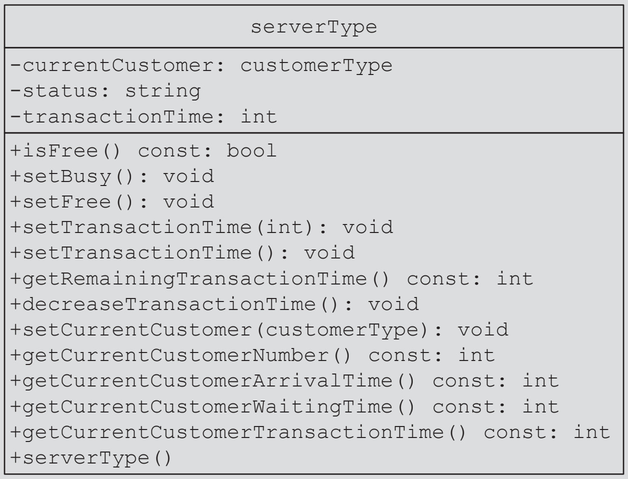

serverType
-currentCustomer: customerType
-status: string
-transactionTime: int
+isFree() const: bool
+setBusy(): void
+setFree(): void
+setTransactionTime(int): void
+setTransactionTime(): void
+getRemainingTransactionTime() const: int
+decreaseTransactionTime(): void
+setCurrentCustomer(customerType): void
+getCurrentCustomerNumber() const: int
+getCurrentCustomerArrivalTime() const: int
+getCurrentCustomerWaitingTime() const: int
+getCurrentCustomerTransactionTime() const: int
+serverType()

FIGURE 8-12 UML diagram of the `class serverType`

The definitions of some of the member functions of the **`class`** `serverType` are as follows:

```
serverType::serverType()
{
    status = "free";
    transactionTime = 0;
}

bool serverType::isFree() const
{
    return (status == "free");
}

void serverType::setBusy()
{
    status = "busy";
}

void serverType::setFree()
{
    status = "free";
}

void serverType::setTransactionTime(int t)
{
    transactionTime = t;
}

void serverType::setTransactionTime()
{
    int time;

    time = currentCustomer.getTransactionTime();

    transactionTime = time;
}

void serverType::decreaseTransactionTime()
{
    transactionTime--;
}
```

We leave the definitions of the functions `getRemainingTransactionTime`, `setCurrentCustomer`, `getCurrentCustomerNumber`, `getCurrentCustomerArrivalTime`, `getCurrentCustomerWaitingTime`, and `getCurrentCustomerTransactionTime` as an exercise for you, (see Programming Exercise 8 at the end of this chapter).

Because we are designing a simulation program that can be used in a variety of applications, we need to design two more classes: a class to create and process a list of servers and a class to create and process a queue of waiting customers. The next two sections describe each of these classes.

Server List

A server list is a set of servers. At any given time, a server is either free or busy. For the customer at the front of the queue, we need to find a server in the list that is free. If all the servers are busy, the customer must wait until one of the servers becomes free. Thus, the class that implements a list of servers has two member variables: one to store the number of servers and one to maintain a list of servers. Using dynamic arrays, depending on the number of servers specified by the user, a list of servers is created during program execution. Some of the operations that must be performed on a server list are as follows: Return the server number of a free server; when a customer gets ready to do business and a server is available, set the server to busy; when the simulation ends, some of the servers might still be busy, so return the number of busy servers; after each time unit, reduce the `transactionTime` of each busy server by one time unit; and if the `transactionTime` of a server becomes zero, set the server to free. The following `class`, `serverListType`, implements the list of servers as an ADT:

```
//*************************************************************
// Author: D.S. Malik
//
// class serverListType
// This class specifies the members to implement a list of
// servers.
//*************************************************************

class serverListType
{
public:
    serverListType(int num = 1);
      //Constructor to initialize a list of servers
      //Postcondition: numOfServers = num
      //    A list of servers, specified by num, is created and
      //    each server is initialized to "free".

    ~serverListType();
      //Destructor
      //Postcondition: The list of servers is destroyed.

    int getFreeServerID() const;
      //Function to search the list of servers.
      //Postcondition: If a free server is found, returns its ID;
      //    otherwise, returns -1.

    int getNumberOfBusyServers() const;
      //Function to return the number of busy servers.
      //Postcondition: The number of busy servers is returned.

    void setServerBusy(int serverID, customerType cCustomer,
                       int tTime);
      //Function to set a server busy.
      //Postcondition: The server specified by serverID is set to
      //    "busy", to serve the customer specified by cCustomer,
      //    and the transaction time is set according to the
      //    parameter tTime.

    void setServerBusy(int serverID, customerType cCustomer);
      //Function to set a server busy.
      //Postcondition: The server specified by serverID is set to
      //    "busy", to serve the customer specified by cCustomer.

    void updateServers(ostream& outFile);
      //Function to update the status of a server.
      //Postcondition: The transaction time of each busy server
      //    is decremented by one unit. If the transaction time of
      //    a busy server is reduced to zero, the server is set to
      //    "free". Moreover, if the actual parameter corresponding
      //    to outFile is cout, a message indicating which customer
      //    has been served is printed on the screen, together with the
      //    customer's departing time. Otherwise, the output is sent
      //    to a file specified by the user.

private:
    int numOfServers;
    serverType *servers;
};
```

Figure 8-13 shows the UML diagram of the `class serverListType`.

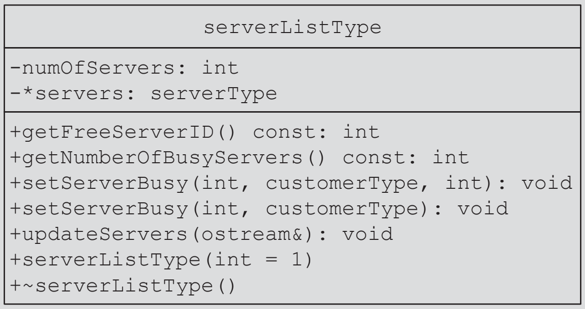

serverListType
-numOfServers: int
-*servers: serverType
+getFreeServerID() const: int
+getNumberOfBusyServers() const: int
+setServerBusy(int, customerType, int): void
+setServerBusy(int, customerType): void
+updateServers(ostream&): void
+serverListType(int = 1)
+~serverListType()

FIGURE 8-13 UML diagram of the `class serverListType`

Following are the definitions of the member functions of the `class serverListType`. The definitions of the constructor and destructor are straightforward.

```
serverListType::serverListType(int num)
{
    numOfServers = num;
    servers = new serverType[num];
}
```

```
serverListType::~serverListType()
{
    delete [] servers;
}
```

The function getFreeServerID searches the list of servers. If a free server is found, it returns the server's ID; otherwise, the value -1 is returned, which indicates that all the servers are busy. The definition of this function is as follows:

```
int serverListType::getFreeServerID() const
{
    int serverID = -1;

    for (int i = 0; i < numOfServers; i++)
        if (servers[i].isFree())
        {
            serverID = i;
            break;
        }

    return serverID;
}
```

The function getNumberOfBusyServers searches the list of servers and determines the number of busy servers. The number of busy servers is returned. The definition of this function is as follows:

```
int serverListType::getNumberOfBusyServers() const
{
    int busyServers = 0;

    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
            busyServers++;

    return busyServers;
}
```

The function setServerBusy sets a server to busy. This function is overloaded. The serverID of the server that is set to busy is passed as a parameter to this function. One function sets the server's transaction time according to the parameter tTime; the other function sets it by using the transaction time stored in the object cCustomer. The transaction time is later needed to determine the average waiting time. The definitions of these functions are as follows:

```
void serverListType::setServerBusy(int serverID,
                            customerType cCustomer, int tTime)
{
    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(tTime);
    servers[serverID].setCurrentCustomer(cCustomer);
}
```

```
void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer)
{
    int time = cCustomer.getTransactionTime();

    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(time);
    servers[serverID].setCurrentCustomer(cCustomer);
}
```

The definition of the function `updateServers` is quite straightforward. Starting at the first server, it searches the list of servers for busy servers. When a busy server is found, its `transactionTime` is decremented by 1. If the `transactionTime` reduces to zero, the server is set to free. If the `transactionTime` of a busy server reduces to zero, the transaction of the customer being served by the server is complete. If the actual parameter corresponding to `outFile` is `cout`, a message indicating which customer has been served is printed on the screen, together with the customer's departing time. Otherwise, the output is sent to a file specified by the user. The definition of this function is as follows:

```
void serverListType::updateServers(ostream& outF)
{
   for (int i = 0; i < numOfServers; i++)
       if (!servers[i].isFree())
       {
          servers[i].decreaseTransactionTime();

          if (servers[i].getRemainingTransactionTime() == 0)
          {
              outF << "From server number " << (i + 1)
                   << " customer number "
                   << servers[i].getCurrentCustomerNumber()
                   << "\n     departed at clock unit "
                   << servers[i].getCurrentCustomerArrivalTime()
                   + servers[i].getCurrentCustomerWaitingTime()
                   + servers[i].getCurrentCustomerTransactionTime()
                   << endl;
              servers[i].setFree();
          }
       }
}
```

Waiting Customers Queue

When a customer arrives, he or she goes to the end of the queue. When a server becomes available, the customer at the front of the queue leaves to conduct the transaction. After each time unit, the waiting time of each customer in the queue is incremented by 1. The ADT `queueType` designed in this chapter has all the operations needed to implement a queue, except the operation of incrementing the waiting time of each customer in the queue by one time unit. We will derive a `class`, `waitingCustomerQueueType`, from the `class queueType` and add the additional operations to implement the customer queue. The definition of the `class waitingCustomerQueueType` is as follows:

```
//*************************************************************
// Author: D.S. Malik
//
// class waitingCustomerQueueType
// This class extends the class queueType to implement a list
// of waiting customers.
//*************************************************************

class waitingCustomerQueueType: public queueType<customerType>
{
public:
    waitingCustomerQueueType(int size = 100);
      //Constructor
      //Postcondition: The queue is initialized according to the
      //    parameter size. The value of size is passed to the
      //    constructor of queueType.

    void updateWaitingQueue();
      //Function to increment the waiting time of each
      //customer in the queue by one time unit.
};
```

NOTE Notice that the `class waitingCustomerQueueType` is derived from the `class queueType`, which implements the queue in an array. You can also derive it from the `class linkedQueueType`, which implements the queue in a linked list. We leave the details as an exercise for you.

The definitions of the member functions are given next. The definition of the constructor is as follows:

```
waitingCustomerQueueType::waitingCustomerQueueType(int size)
                          :queueType<customerType>(size)
{
}
```

The function `updateWaitingQueue` increments the waiting time of each customer in the queue by one time unit. The `class waitingCustomerQueueType` is derived from the `class queueType`. Because the member variables of `queueType` are `private`, the function `updateWaitingQueue` cannot directly access the elements of the queue. The only way to access the elements of the queue is to use the `deleteQueue` operation. After incrementing the waiting time, the element can be put back into the queue by using the `addQueue` operation.

The `addQueue` operation inserts the element at the end of the queue. If we perform the `deleteQueue` operation followed by the `addQueue` operation for each element of the queue, eventually the front element again becomes the front element. Given that each `deleteQueue` operation is followed by an `addQueue` operation, how do we determine that all the elements of the queue have been processed? We cannot use the `isEmptyQueue` or `isFullQueue` operations on the queue because the queue will never be empty or full.

One solution to this problem is to create a temporary queue. Every element of the original queue is removed, processed, and inserted into the temporary queue. When the original queue becomes empty, all of the elements in the queue are processed. We can then copy the elements from the temporary queue back into the original queue. However, this solution requires us to use extra memory space, which could be significant. Also, if the queue is large, extra computer time is needed to copy the elements from the temporary queue back into the original queue. Let us look into another solution.

In the second solution, before starting to update the elements of the queue, we can insert a dummy customer with a waiting time of, say `-1`. During the update process, when we arrive at the customer with the waiting time of `-1`, we can stop the update process without processing the customer with the waiting time of `-1`. If we do not process the customer with the waiting time `-1`, this customer is removed from the queue and after processing all the elements of the queue, the queue will contain no extra elements. This solution does not require us to create a temporary queue, so we do not need extra computer time to copy the elements back into the original queue. We will use this solution to update the queue. Therefore, the definition of the function `updateWaitingQueue` is as follows:

```
void waitingCustomerQueueType::updateWaitingQueue()
{
    customerType cust;

    cust.setWaitingTime(-1);
    int wTime = 0;

    addQueue(cust);

    while (wTime != -1)
    {
        cust = front();
        deleteQueue();

        wTime = cust.getWaitingTime();
        if (wTime == -1)
            break;
        cust.incrementWaitingTime();
        addQueue(cust);
    }
}
```

Main Program

To run the simulation, we first need to get the following information:

- The number of time units the simulation should run. Assume that each time unit is one minute.
- The number of servers.
- The amount of time it takes to serve a customer—that is, the transaction time.
- The approximate time between customer arrivals.

These pieces of information are called simulation parameters. By changing the values of these parameters, we can observe the changes in the performance of the system. We can write a function, `setSimulationParameters`, to prompt the user to specify these values. The definition of this function is as follows:

```
void setSimulationParameters(int& sTime, int& numOfServers,
                             int& transTime, int& tBetweenCArrival)
{
    cout << "Enter the simulation time: ";
    cin >> sTime;
    cout << endl;

    cout << "Enter the number of servers: ";
    cin >> numOfServers;
    cout << endl;

    cout << "Enter the transaction time: ";
    cin >> transTime;
    cout << endl;

    cout << "Enter the time between customers arrival: ";
    cin >> tBetweenCArrival;
    cout << endl;
}
```

When a server becomes free and the customer queue is nonempty, we can move the customer at the front of the queue to the free server to be served. Moreover, when a customer starts the transaction, the waiting time ends. The waiting time of the customer is added to the total waiting time. The general algorithm to start the transaction (supposing that `serverID` denotes the ID of the free server) is as follows:

1. Remove the customer from the front of the queue.

   ```
   customer = customerQueue.front();
   customerQueue.deleteQueue();
   ```

2. Update the total waiting time by adding the current customer's waiting time to the previous total waiting time.

   ```
   totalWait = totalWait + customer.getWaitingTime();
   ```

3. Set the free server to begin the transaction.

   ```
   serverList.setServerBusy(serverID, customer, transTime);
   ```

To run the simulation, we need to know the number of customers arriving at a given time unit and how long it takes to serve the customer. We use the Poisson distribution from statistics, which says that the probability of y events occurring at a given time is given by the formula:

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \ldots,$$

where λ is the expected value that y events occur at that time. Suppose that, on average, a customer arrives every four minutes. During this four-minute period, the customer can arrive at any one of the four minutes. Assuming an equal likelihood of each of the four minutes, the expected value that a customer arrives in each of the four minutes is, therefore, $1 / 4 = 0.25$. Next, we need to determine whether the customer actually arrives at a given minute.

Now $P(0) = e^{-\lambda}$ is the probability that no event occurs at a given time. One of the basic assumptions of the Poisson distribution is that the probability of more than one outcome occurring in a short time interval is negligible. For simplicity, we assume that only one customer arrives at a given time unit. Thus, we use $e^{-\lambda}$ as the cutoff point to determine whether a customer arrives at a given time unit. Suppose that, on average, a customer arrives every four minutes. Then $\lambda = 0.25$. We can use an algorithm to generate a number between 0 and 1. If the value of the number generated is $> e^{-0.25}$, we can assume that the customer arrived at a particular time unit. For example, suppose that *rNum* is a random number such that $0 \leq rNum \leq 1$. If $rNum > e^{-0.25}$, the customer arrived at the given time unit.

We now describe the function `runSimulation` to implement the simulation. Suppose that we run the simulation for 100 time units and customers arrive at time units 93, 96, and 100. The average transaction time is 5 minutes—that is, 5 time units. For simplicity, assume that we have only one server and the server becomes free at time unit 97, and that all customers arriving before time unit 93 have been served. When the server becomes free at time unit 97, the customer arriving at time unit 93 starts the transaction. Because the transaction of the customer arriving at time unit 93 starts at time unit 97 and it takes 5 minutes to complete a transaction, when the simulation loop ends, the customer arriving at time unit 93 is still at the server. Moreover, customers arriving at time units 96 and 100 are in the queue. For simplicity, we assume that when the simulation loop ends, the customers at the servers are considered served. The general algorithm for this function is as follows:

1. Declare and initialize the variables such as the simulation parameters, customer number, clock, total and average waiting times, number of customers arrived, number of customers served, number of customers left in the waiting queue, number of customers left with the servers, `waitingCustomersQueue`, and a list of servers.
2. The main loop is as follows:

```
for (clock = 1; clock <= simulationTime; clock++)
{
    2.1. Update the server list to decrement the transaction time of
         each busy server by one time unit.
    2.2. If the customer's queue is nonempty, increment the waiting
         time of each customer by one time unit.
    2.3. If a customer arrives, increment the number of customers by 1
         and add the new customer to the queue.
    2.4. If a server is free and the customer's queue is nonempty,
         remove a customer from the front of the queue and send
         the customer to the free server.
}
```

3. Print the appropriate results. Your results must include the number of customers left in the queue, the number of customers still with servers, the number of customers arrived, and the number of customers who actually completed a transaction.

Once you have designed the function `runSimulation`, the definition of the function `main` is simple and straightforward because the function `main` calls only the function `runSimulation`. (See Programming Exercise 8 at the end of this chapter.)

When we tested our version of the simulation program, we generated the following results. We assumed that the average transaction time is 5 minutes and that on average a customer arrives every 4 minutes, and we used a random number generator to generate a number between `0` and `1` to decide whether a customer arrived at a given time unit.

Sample Run:

```
Customer number 1 arrived at time unit 4
Customer number 2 arrived at time unit 8
From server number 1 customer number 1
     departed at clock unit 9
Customer number 3 arrived at time unit 9
Customer number 4 arrived at time unit 12
From server number 1 customer number 2
     departed at clock unit 14
From server number 1 customer number 3
     departed at clock unit 19
Customer number 5 arrived at time unit 21
From server number 1 customer number 4
     departed at clock unit 24
From server number 1 customer number 5
     departed at clock unit 29
Customer number 6 arrived at time unit 37
Customer number 7 arrived at time unit 38
Customer number 8 arrived at time unit 41
From server number 1 customer number 6
     departed at clock unit 42
Customer number 9 arrived at time unit 43
Customer number 10 arrived at time unit 44
From server number 1 customer number 7
     departed at clock unit 47
Customer number 11 arrived at time unit 49
Customer number 12 arrived at time unit 51
From server number 1 customer number 8
     departed at clock unit 52
Customer number 13 arrived at time unit 52
Customer number 14 arrived at time unit 53
Customer number 15 arrived at time unit 54
From server number 1 customer number 9
     departed at clock unit 57
Customer number 16 arrived at time unit 59
From server number 1 customer number 10
     departed at clock unit 62
```

```
Customer number 17 arrived at time unit 66
From server number 1 customer number 11
     departed at clock unit 67
Customer number 18 arrived at time unit 71
From server number 1 customer number 12
     departed at clock unit 72
From server number 1 customer number 13
     departed at clock unit 77
Customer number 19 arrived at time unit 78
From server number 1 customer number 14
     departed at clock unit 82
From server number 1 customer number 15
     departed at clock unit 87
Customer number 20 arrived at time unit 90
From server number 1 customer number 16
     departed at clock unit 92
Customer number 21 arrived at time unit 92
From server number 1 customer number 17
     departed at clock unit 97

The simulation ran for 100 time units
Number of servers: 1
Average transaction time: 5
Average arrival time difference between customers: 4
Total waiting time: 269
Number of customers that completed a transaction: 17
Number of customers left in the servers: 1
The number of customers left in queue: 3
Average waiting time: 12.81
************** END SIMULATION *************
```

QUICK REVIEW

1. A queue is a data structure in which the items are added at one end and removed from the other end.
2. A queue is a First In First Out (FIFO) data structure.
3. The basic operations on a queue are as follows: Add an item to the queue, remove an item from the queue, retrieve the first and last element of the queue, initialize the queue, check whether the queue is empty, and check whether the queue is full.
4. A queue can be implemented as an array or a linked list.
5. The middle elements of a queue should not be accessed directly.
6. If the queue is nonempty, the function `front` returns the front element of the queue and the function `back` returns the last element in the queue.
7. Queues are restricted versions of arrays and linked lists.

EXERCISES

1. Consider the following statements:

```
queueType<int> queue;
int x, y;
```

   Show what is output by the following segment of code:

```
x = 4;
y = 5;
queue.addQueue(x);
queue.addQueue(y);
x = queue.front();
queue.deleteQueue();
queue.addQueue(x + 5);
queue.addQueue(16);
queue.addQueue(x);
queue.addQueue(y - 3);

cout << "Queue Elements: ";
while (!queue.isEmptyQueue())
{
    cout << queue.front() << " ";
    queue.deleteQueue();
}
cout << endl;
```

2. Consider the following statements:

```
stackType<int> stack;
queueType<int> queue;
int x;
```

   Suppose the input is:

```
15 28 14 22 64 35 19 32 7 11 13 30 -999
```

   Show what is written by the following segment of code:

```
stack.push(0);
queue.addQueue(0);
cin >> x;

while (x != -999)
{
    switch (x % 4)
    {
    case 0:
        stack.push(x);
        break;
    case 1:
        if (!stack.isEmptyStack())
        {
            cout << "Stack Element = " << stack.top()
                 << endl;
            stack.pop();
        }
```

```
            else
                cout << "Sorry, the stack is empty." << endl;
                break;
        case 2:
            queue.addQueue(x);
            break;
        case 3:
            if (!queue.isEmptyQueue())
            {
                cout << "Queue Element = " << queue.front()
                     << endl;
                queue.deleteQueue();
            }
            else
                cout << "Sorry, the queue is empty." << endl;
            break;
        } //end switch

        cin >> x;
    } //end while

    cout << "Stack Elements: ";
    while (!stack.isEmptyStack())
    {
        cout << stack.top() << " ";
        stack.pop();
    }

    cout << endl;

    cout << "Queue Elements: ";
    while (!queue.isEmptyQueue())
    {
        cout << queue.front() << " ";
        queue.deleteQueue();
    }
    cout << endl;
```

3. What does the following function do?

```
void mystery(queueType<int>& q)
{
    stackType<int> s;

    while (!q.isEmptyQueue())
    {
        s.push(q.front());
        q.deleteQueue();
    }

    while (!s.isEmptyStack())
    {
        q.addQueue(2 * s.top());
        s.pop();
    }
}
```

4. What is the effect of the following statements? If a statement is invalid, explain why it is invalid. The classes queueADT, queueType, and linkedQueueType are as defined in this chapter.
   a. `queueADT<int> newQueue;`
   b. `queueType <double> sales(-10);`
   c. `queueType <string> names;`
   d. `linkedQueueType <int> numQueue(50);`
5. What is the output of the following program segment?

```
linkedQueueType<int> queue;

queue.addQueue(10);
queue.addQueue(20);
cout << queue.front() << endl;
queue.deleteQueue();
queue.addQueue(2 * queue.back());
queue.addQueue(queue.front());
queue.addQueue(5);
queue.addQueue(queue.back() - 2);

linkedQueueType<int> tempQueue;

tempQueue = queue;

while (!tempQueue.isEmptyQueue())
{
    cout << tempQueue.front() << " ";
    tempQueue.deleteQueue();
}

cout << endl;

cout << queue.front() << " " << queue.back() << endl;
```

6. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 50 and the value of queueRear is 99.
   a. What are the values of queueFront and queueRear after adding an element to queue?
   b. What are the values of queueFront and queueRear after removing an element from queue?
7. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 25.

a. What are the values of queueFront and queueRear after adding an element to queue?
b. What are the values of queueFront and queueRear after removing an element from queue?

8. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 25 and the value of queueRear is 75.
   a. What are the values of queueFront and queueRear after adding an element to queue?
   b. What are the values of queueFront and queueRear after removing an element from queue?
9. Suppose that queue is a queueType object and the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 99 and the value of queueRear is 99.
   a. What are the values of queueFront and queueRear after adding an element to queue?
   b. What are the values of queueFront and queueRear after removing an element from queue?
10. Suppose that queue is implemented as an array with the special reserved slot, as described in this chapter. Also, suppose that the size of the array implementing queue is 100. If the value of queueFront is 50, what is the position of the first queue element?
11. Suppose that queue is implemented as an array with the special reserved slot, as described in this chapter. Suppose that the size of the array implementing queue is 100. Also, suppose that the value of queueFront is 74 and the value of queueRear is 99.
    a. What are the values of queueFront and queueRear after adding an element to queue?
    b. What are the values of queueFront and queueRear after removing an element from queue? Also, what is the position of the removed queue element?
12. Write a function template, reverseQueue, that takes as a parameter a queue object and uses a stack object to reverse the elements of the queue.
13. Add the operation queueCount to the class queueType (the array implementation of queues), which returns the number of elements in the queue. Write the definition of the function template to implement this operation.
14. Draw the UML diagram of the class queueADT.
15. Draw the UML diagram of the class queueType.
16. Draw the UML diagram of the class linkedQueueType.

PROGRAMMING EXERCISES

1. Write the definitions of the functions to overload the assignment operator and copy constructor for the `class queueType`. Also, write a program to test these operations.
2. Write the definitions of the functions to overload the assignment operator and copy constructor for the `class linkedQueueType`. Also, write a program to test these operations.
3. This chapter described the array implementation of queues that use a special array slot, called the reserved slot, to distinguish between an empty and a full queue. Write the definition of the class and the definitions of the function members of this queue design. Also, write a test program to test various operations on a queue.
4. Write the definition of the function `moveNthFront` that takes as a parameter a positive integer, *n*. The function moves the *n*th element of the queue to the front. The order of the remaining elements remains unchanged. For example, suppose

   `queue = {5, 11, 34, 67, 43, 55} and` *n* `= 3.`

   After a call to the function `moveNthFront`,

   `queue = {34, 5, 11, 67, 43, 55}.`

   Add this function to the `class queueType`. Also write a program to test your method.
5. Write a program that reads a line of text, changes each uppercase letter to lowercase, and places each letter both in a queue and onto a stack. The program should then verify whether the line of text is a palindrome (a set of letters or numbers that is the same whether read forward or backward).
6. The implementation of a queue in an array, as given in this chapter, uses the variable `count` to determine whether the queue is empty or full. You can also use the variable `count` to return the number of elements in the `queue`. (See Exercise 13.) On the other hand, `class linkedQueueType` does not use such a variable to keep track of the number of elements in the queue. Redefine the `class linkedQueueType` by addding the variable `count` to keep track of the number of elements in the queue. Modify the definitions of the functions `addQueue` and `deleteQueue` as necessary. Add the function `queueCount` to return the number of elements in the queue. Also, write a program to test various operations of the class you defined.
7. Write the definition of the `class linkedQueueType`, which is derived from the `class unorderedLinkedList`, as explained in this chapter. Also write a program to test various operations of this class.

8. a. Write the definitions of the functions `setWaitingTime`, `getArrivalTime`, `getTransactionTime`, and `getCustomerNumber` of the `class customerType` defined in the section, "Application of Queues: Simulation."

   b. Write the definitions of the functions `getRemainingTransactionTime`, `setCurrentCustomer`, `getCurrentCustomerNumber`, `getCurrentCustomerArrivalTime`, `getCurrentCustomerWaitingTime`, and `getCurrentCustomerTransactionTime` of the `class serverType` defined in the section, "Application of Queues: Simulation."

   c. Write the definition of the function `runSimulation` to complete the design of the computer simulation program (see the section, "Application of Queues: Simulation"). Test run your program for a variety of data. Moreover, use a random number generator to decide whether a customer arrived at a given time unit.

9. Redo the simulation program of this chapter so that it uses the STL `class queue` to maintain the list of waiting customers.