5.1 Stacks

A ***stack*** is a container of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time. The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate. Perhaps an even more amusing metaphor would be a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that "pops" out the top-most candy in the stack when the top of the dispenser is lifted. (See Figure 5.1.) Stacks are a fundamental data structure. They are used in many applications, including the following.

Example 5.1: *Internet Web browsers store the addresses of recently visited sites on a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 5.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*

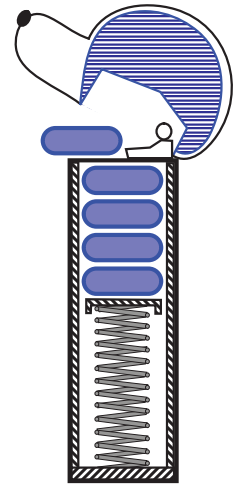

Figure 5.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

5.1.1 The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important, since they are used in a host of different applications that include many more sophisticated data structures. Formally, a stack is an abstract data type (ADT) that supports the following operations:

push(e): Insert element e at the top of the stack.

pop(): Remove the top element from the stack; an error occurs if the stack is empty.

top(): Return a reference to the top element on the stack, without removing it; an error occurs if the stack is empty.

Additionally, let us also define the following supporting functions:

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

Example 5.3: *The following table shows a series of stack operations and their effects on an initially empty stack of integers.*

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

5.1.2 The STL Stack

The Standard Template Library provides an implementation of a stack. The underlying implementation is based on the STL vector class, which is presented in Sections 1.5.5 and 6.1.4. In order to declare an object of type stack, it is necessary to first include the definition file, which is called "stack." As with the STL vector, the class stack is part of the std namespace, so it is necessary either to use "std::stack" or to provide a "**using**" statement. The stack class is templated with the class of the individual elements. For example, the code fragment below declares a stack of integers.

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

We refer to the type of individual elements as the stack's ***base type***. As with STL vectors, an STL stack dynamically resizes itself as new elements are pushed on.

The STL stack class supports the same operators as our interface. Below, we list the principal member functions. Let s be declared to be an STL vector, and let e denote a single object whose type is the same as the base type of the stack. (For example, s is a vector of integers, and e is an integer.)

size(): Return the number of elements in the stack.

empty(): Return true if the stack is empty and false otherwise.

push(e): Push e onto the top of the stack.

pop(): Pop the element at the top of the stack.

top(): Return a reference to the element at the top of the stack.

There is one significant difference between the STL implementation and our own definitions of the stack operations. In the STL implementation, the result of applying either of the operations top or pop to an empty stack is undefined. In particular, no exception is thrown. Even though no exception is thrown, it may very likely result in your program aborting. Thus, it is up to the programmer to be sure that no such illegal accesses are attempted.

5.1.3 A C++ Stack Interface

Before discussing specific implementations of the stack, let us first consider how to define an abstract data type for a stack. When defining an abstract data type, our principal concern is specifying the ***Application Programming Interface*** (API), or simply ***interface***, which describes the names of the public members that the ADT must support and how they are to be declared and used. An interface is not a complete description of all the public members. For example, it does not include

the private data members. Rather, it is a list of members that any implementation must provide. The C++ programming language does not provide a simple method for defining interfaces, and therefore, the interface defined here is not an official C++ class. It is offered principally for the purpose of illustration.

The informal interface for the stack ADT is given in Code Fragment 5.1. This interface defines a class template. Recall from Section 2.3 that such a definition implies that the base type of element being stored in the stack will be provided by the user. In Code Fragment 5.1, this element type is indicated by E. For example, E may be any fundamental type (such as **int**, **char**, **bool**, and **double**), any built-in or user-defined class (such as string), or a pointer to any of these.

```
template <typename E>
class Stack {                                    // an interface for a stack
public:
  int size() const;                              // number of items in stack
  bool empty() const;                            // is the stack empty?
  const E& top() const throw(StackEmpty);        // the top element
  void push(const E& e);                         // push x onto the stack
  void pop() throw(StackEmpty);                  // remove the top element
};
```

Code Fragment 5.1: An informal Stack interface (not a complete C++ class).

Observe that the member functions size, empty, and top are all declared to be **const**, which informs the compiler that they do not alter the contents of the stack. The member function top returns a constant reference to the top of the stack, which means that its value may be read but not written.

Note that pop does not return the element that was popped. If the user wants to know this value, it is necessary to perform a top operation first, and save the value. The member function push takes a constant reference to an object of type E as its argument. Recall from Section 1.4 that this is the most efficient way of passing objects to a function.

An error condition occurs when calling either of the functions pop or top on an empty stack. This is signaled by throwing an exception of type StackEmpty, which is defined in Code Fragment 5.2.

```
// Exception thrown on performing top or pop of an empty stack.
class StackEmpty : public RuntimeException {
public:
  StackEmpty(const string& err) : RuntimeException(err) {}
};
```

Code Fragment 5.2: Exception thrown by functions pop and top when called on an empty stack. This class is derived from RuntimeException from Section 2.4.

5.1.4 A Simple Array-Based Stack Implementation

We can implement a stack by storing its elements in an array. Specifically, the stack in this implementation consists of an N-element array S plus an integer variable t that gives the index of the top element in array S. (See Figure 5.2.)

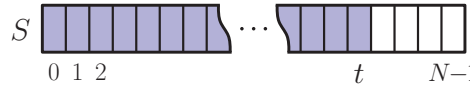

Figure 5.2: Realization of a stack by means of an array S. The top element in the stack is stored in the cell $S[t]$.

Recalling that arrays in C++ start at index 0, we initialize t to -1, and use this value for t to identify when the stack is empty. Likewise, we can use this variable to determine the number of elements in a stack $(t+1)$. We also introduce a new type of exception, called StackFull, to signal the error condition that arises if we try to insert a new element and the array S is full. Exception StackFull is specific to our implementation of a stack and is not defined in the stack ADT. Given this new exception, we can then implement the stack ADT functions as described in Code Fragment 5.3.

Algorithm size():
    return $t+1$
Algorithm empty():
    return $(t < 0)$
Algorithm top():
    if empty() **then**
        throw StackEmpty exception
    return $S[t]$
Algorithm push(e):
    if size() $= N$ **then**
        throw StackFull exception
    $t \leftarrow t+1$
    $S[t] \leftarrow e$
Algorithm pop():
    if empty() **then**
        throw StackEmpty exception
    $t \leftarrow t-1$

Code Fragment 5.3: Implementation of a stack by means of an array.

The correctness of the functions in the array-based implementation follows immediately from the definition of the functions themselves. Table 5.1 shows the

running times for member functions in a realization of a stack by an array. Each of the stack functions in the array realization executes a constant number of statements involving arithmetic operations, comparisons, and assignments. Thus, in this implementation of the Stack ADT, each function runs in constant time, that is, they each run in $O(1)$ time.

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Table 5.1: Performance of an array-based stack. The space usage is $O(N)$, where N is the array's size. Note that the space usage is independent from the number $n \leq N$ of elements that are actually in the stack.

A C++ Implementation of a Stack

In this section, we present a concrete C++ implementation of the above pseudo-code specification by means of a class, called ArrayStack. Our approach is to store the elements of a stack in an array. To keep the code simple, we have omitted the standard housekeeping utilities, such as a destructor, an assignment operator, and a copy constructor. We leave their implementations as an exercise.

We begin by providing the ArrayStack class definition in Code Fragment 5.4.

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);         // pop the stack
  // ...housekeeping functions omitted
private:                                // member data
  E* S;                                 // array of stack elements
  int capacity;                         // stack capacity
  int t;                                // index of the top of the stack
};
```

Code Fragment 5.4: The class ArrayStack, which implements the Stack interface.

In addition to the member functions required by the interface, we also provide a constructor, that is given the desired capacity of the stack as its only argument. If no argument is given, the default value given by DEF_CAPACITY is used. This is an example of using default arguments in function calls. We use an enumeration to define this default capacity value. This is the simplest way of defining symbolic integer constants within a C++ class. Our class is templated with the element type, denoted by E. The stack's storage, denoted S, is a dynamically allocated array of type E, that is, a pointer to E.

Next, we present the implementations of the ArrayStack member functions in Code Fragment 5.5. The constructor allocates the array storage, whose size is set to the default capacity. The members *capacity* and t are also set to their initial values. In spite of the syntactical complexities of defining templated member functions in C++, the remaining member functions are straightforward implementations of their definitions in Code 5.3. Observe that functions top and pop first check that the stack is not empty, and otherwise, they throw an exception. Similarly, push first checks that the stack is not full, and otherwise, it throws an exception.

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(-1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                     // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                     // is the stack empty?

template <typename E>                     // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                     // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                     // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --t;
}
```

Code Fragment 5.5: Implementations of the member functions of class ArrayStack (excluding housekeeping functions).

Example Output

In Code Fragment 5.6 below, we present an example of the use of our ArrayStack class. To demonstrate the flexibility of our implementation, we show two stacks of different base types. The instance A is a stack of integers of the default capacity (100). The instance B is a stack of character strings of capacity 10.

```
ArrayStack<int> A;                          // A = [], size = 0
A.push(7);                                  // A = [7*], size = 1
A.push(13);                                 // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 13
A.push(9);                                  // A = [7, 9*], size = 2
cout << A.top() << endl;                    // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();           // A = [7*], outputs: 9
ArrayStack<string> B(10);                   // B = [], size = 0
B.push("Bob");                              // B = [Bob*], size = 1
B.push("Alice");                            // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();           // B = [Bob*], outputs: Alice
B.push("Eve");                              // B = [Bob, Eve*], size = 2
```

Code Fragment 5.6: An example of the use of the ArrayStack class. The contents of the stack are shown in the comment following the operation. The top of the stack is indicated by an asterisk ("*").

Note that our implementation, while simple and efficient, could be enhanced in a number of ways. For example, it assumes a fixed upper bound N on the ultimate size of the stack. In Code Fragment 5.4, we chose the default capacity value $N = 100$ more or less arbitrarily (although the user can set the capacity in the constructor). An application may actually need much less space than the given initial size, and this would be wasteful of memory. Alternatively, an application may need more space than this, in which case our stack implementation might "crash" if too many elements are pushed onto the stack.

Fortunately, there are other implementations that do not impose an arbitrary size limitation. One such method is to use the STL stack class, which was introduced earlier in this chapter. The STL stack is also based on the STL vector class, and it offers the advantage that it is automatically expanded when the stack overflows its current storage limits. In practice, the STL stack would be the easiest and most practical way to implement an array-based stack. Later in this chapter, we see other methods that use space proportional to the actual size of the stack.

In instances where we have a good estimate on the number of items needing to go in the stack, the array-based implementation is hard to beat from the perspective of speed and simplicity. Stacks serve a vital role in a number of computing applications, so it is helpful to have a fast stack ADT implementation, such as the simple array-based implementation.

5.1.5 Implementing a Stack with a Generic Linked List

In this section, we show how to implement the stack ADT using a singly linked list. Our approach is to use the generic singly linked list, called SLinkedList, which was presented earlier in Section 3.2.4. The definition of our stack, called LinkedStack, is presented in Code Fragment 5.7.

To avoid the syntactic messiness inherent in C++ templated classes, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the stack's elements, called Elem. In this example, we define Elem to be of type string. We leave the task of producing a truly generic implementation as an exercise. (See Exercise R-5.7.)

```
typedef string Elem;                          // stack element type
class LinkedStack {                           // stack as a linked list
public:
  LinkedStack();                              // constructor
  int size() const;                           // number of items in the stack
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:                                      // member data
  SLinkedList<Elem> S;                        // linked list of elements
  int n;                                      // number of elements
};
```

Code Fragment 5.7: The class LinkedStack, a linked list implementation of a stack.

The principal data member of the class is the generic linked list of type Elem, called S. Since the SLinkedList class does not provide a member function size, we store the current size in a member variable, n.

In Code Fragment 5.8, we present the implementations of the constructor and the size and empty functions. Our constructor creates the initial stack and initializes n to zero. We do not provide an explicit destructor, relying instead on the SLinkedList destructor to deallocate the linked list S.

```
LinkedStack::LinkedStack()
  : S(), n(0) { }                             // constructor

int LinkedStack::size() const
  { return n; }                               // number of items in the stack

bool LinkedStack::empty() const
  { return n == 0; }                          // is the stack empty?
```

Code Fragment 5.8: Constructor and size functions for the LinkedStack class.

The definitions of the stack operations, top, push, and pop, are presented in Code Fragment 5.9. Which side of the list, head or tail, should we chose for the top of the stack? Since SLinkedList can insert and delete elements in constant time only at the head, the head is clearly the better choice. Therefore, the member function top returns S.front(). The functions push and pop invoke the functions addFront and removeFront, respectively, and update the number of elements.

```
                                          // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {   // push element onto stack
  ++n;
  S.addFront(e);
}
                                          // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

Code Fragment 5.9: Principal operations for the LinkedStack class.

5.1.6 Reversing a Vector Using a Stack

We can use a stack to reverse the elements in a vector, thereby producing a nonrecursive algorithm for the array-reversal problem introduced in Section 3.5.1. The basic idea is to push all the elements of the vector in order into a stack and then fill the vector back up again by popping the elements off of the stack. In Code Fragment 5.10, we give a C++ implementation of this algorithm.

```
template <typename E>
void reverse(vector<E>& V) {              // reverse a vector
  ArrayStack<E> S(V.size());
  for (int i = 0; i < V.size(); i++)      // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) {    // pop them in reverse order
    V[i] = S.top();  S.pop();
  }
}
```

Code Fragment 5.10: A generic function that uses a stack to reverse a vector.

For example, if the input vector to function reverse contained the five strings [Jack, Kate, Hurley, Jin, Michael], then on returning from the function, the vector would contain [Michael, Jin, Hurley, Kate, Jack].

5.1.7 Matching Parentheses and HTML Tags

In this section, we explore two related applications of stacks. The first is matching parentheses and grouping symbols in arithmetic expressions. Arithmetic expressions can contain various pairs of grouping symbols, such as

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"
- Floor function symbols: "$\lfloor$" and "$\rfloor$"
- Ceiling function symbols: "$\lceil$" and "$\rceil$,"

and each opening symbol must match with its corresponding closing symbol. For example, a left bracket symbol ("[") must match with a corresponding right bracket ("]") as in the following expression:

- Correct: $()(())\{([()])\}$
- Correct: $(((())(()))\{([()])\}))$
- Incorrect: $)(())\{([()])\}$
- Incorrect: $(\{[])\}$
- Incorrect: $($

We leave the precise definition of matching of grouping symbols to Exercise R-5.8.

An Algorithm for Parentheses Matching

An important problem in processing arithmetic expressions is to make sure their grouping symbols match up correctly. We can use a stack S to perform the matching of grouping symbols in an arithmetic expression with a single left-to-right scan. The algorithm tests that left and right symbols match up and also that the left and right symbols are both of the same type.

Suppose we are given a sequence $X = x_0x_1x_2\ldots x_{n-1}$, where each x_i is a ***token*** that can be a grouping symbol, a variable name, an arithmetic operator, or a number. The basic idea behind checking that the grouping symbols in S match correctly, is to process the tokens in X in order. Each time we encounter an opening symbol, we push that symbol onto S, and each time we encounter a closing symbol, we pop the top symbol from the stack S (assuming S is not empty) and we check that these two symbols are of corresponding types. (For example, if the symbol "(" was pushed, the symbol ")" should be its match.) If the stack is empty after we have processed the whole sequence, then the symbols in X match.

Assuming that the push and pop operations are implemented to run in constant time, this algorithm runs in $O(n)$ total time. We give a pseudo-code description of this algorithm in Code Fragment 5.11.

Algorithm ParenMatch(X,n):
Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number
Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
  if X[i] is an opening grouping symbol then
    S.push(X[i])
  else if X[i] is a closing grouping symbol then
    if S.empty() then
      return false          {nothing to match with}
    if S.top() does not match the type of X[i] then
      return false          {wrong type}
    S.pop()
if S.empty() then
  return true               {every symbol matched}
else
  return false              {some symbols were never matched}
```

Code Fragment 5.11: Algorithm for matching grouping symbols in an arithmetic expression.

Matching Tags in an HTML Document

Another application in which matching is important is in the validation of HTML documents. HTML is the standard format for hyperlinked documents on the Internet. In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>." Commonly used HTML tags include:

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

We show a sample HTML document and a possible rendering in Figure 5.3. Our goal is to write a program to check that the tags properly match.

A very similar approach to that given in Code Fragment 5.11 can be used to match the tags in an HTML document. We push each opening tag on a stack, and when we encounter a closing tag, we pop the stack and verify that the two tags match.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even
as a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

Figure 5.3: HTML tags: (a) an HTML document; (b) its rendering.

In Code Fragments 5.12 through 5.14, we present a C++ program for matching tags in an HTML document read from the standard input stream. For simplicity, we assume that all tags are syntactically well formed.

First, the procedure getHtmlTags reads the input line by line, extracts all the tags as strings, and stores them in a vector, which it returns.

```
vector<string> getHtmlTags() {                  // store tags in a vector
  vector<string> tags;                          // vector of html tags
  while (cin) {                                 // read until end of file
    string line;
    getline(cin, line);                         // input a full line of text
    int pos = 0;                                // current scan position
    int ts = line.find("<", pos);               // possible tag start
    while (ts != string::npos) {                // repeat until end of string
      int te = line.find(">", ts+1);            // scan for tag end
      tags.push_back(line.substr(ts, te-ts+1)); // append tag to the vector
      pos = te + 1;                             // advance our position
      ts = line.find("<", pos);
    }
  }
  return tags;                                  // return vector of tags
}
```

Code Fragment 5.12: Get a vector of HTML tags from the input, and store them in a vector of strings.

Given the example shown in Figure 5.3(a), this procedure would return the following vector:

<body>, , <h1>, </h1>, , ..., </body>

In Code Fragment 5.12, we employ a variable *pos*, which maintains the current position in the input line. We use the built-in string member function find to locate the first occurrence of "<" that follows the current position. (Recall the discussion of string operations from Section 1.5.5.) This tag start position is stored in the variable *ts*. We then find the next occurrence of ">," and store this tag end position in *te*. The tag itself consists of the substring of length $te - ts + 1$ starting at position *ts*. This is pushed onto the vector *tags*. We then update the current position to be $te + 1$ and repeat until we find no further occurrences of "<." This occurs when the find function returns the special value string::npos.

Next, the procedure isHtmlMatched, shown in Code Fragments 5.12, implements the process of matching the tags.

```
                                                  // check for matching tags
bool isHtmlMatched(const vector<string>& tags) {
  LinkedStack S;                                  // stack for opening tags
  typedef vector<string>::const_iterator Iter;// iterator type
                                                  // iterate through vector
  for (Iter p = tags.begin(); p != tags.end(); ++p) {
    if (p->at(1) != '/')                          // opening tag?
      S.push(*p);                                 // push it on the stack
    else {                                        // else must be closing tag
      if (S.empty()) return false;                // nothing to match - failure
      string open = S.top().substr(1);            // opening tag excluding '<'
      string close = p->substr(2);                // closing tag excluding '</'
      if (open.compare(close) != 0) return false; // fail to match
      else S.pop();                               // pop matched element
    }
  }
  if (S.empty()) return true;                     // everything matched - good
  else return false;                              // some unmatched - bad
}
```

Code Fragment 5.13: Check whether HTML tags stored in the vector tags are matched.

We create a stack, called *S*, in which we store the opening tags. We then iterate through the vector of tags. If the second character tag string is not "/," then this is an opening tag, and it is pushed onto the stack. Otherwise, it is a closing tag, and we check that it matches the tag on top of the stack. To compare the opening and closing tags, we use the string substr member function to strip the first character off the opening tag (thus removing the "<") and the first two characters off the closing tag (thus removing the "</"). We check that these two substrings are equal, using

the built-in string function compare. When the loop terminates, we know that every closing tag matches its corresponding opening tag. To finish the job, we need to check that there were no unmatched opening tags. We test this by checking that the stack is now empty.

Finally, the main program is presented in Code Fragment 5.14. It invokes the function getHtmlTags to read the tags, and then it passes these to isHtmlMatched to test them.

```
int main() {                                    // main HTML tester
  if (isHtmlMatched(getHtmlTags()))             // get tags and test them
    cout << "The input file is a matched HTML document." << endl;
  else
    cout << "The input file is not a matched HTML document." << endl;
}
```

Code Fragment 5.14: The main program to test whether the input file consists of matching HTML tags.

5.2 Queues

Another fundamental data structure is the ***queue***, which is a close relative of the stack. A queue is a container of elements that are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle. Elements can be inserted in a queue at any time, but only the element that has been in the queue the longest can be removed at any time. We usually say that elements enter the queue at the ***rear*** and are removed from the ***front***. The metaphor for this terminology is a line of people waiting to get on an amusement park ride. People enter at the rear of the line and get on the ride from the front of the line.

5.2.1 The Queue Abstract Data Type

Formally, the queue abstract data type defines a container that keeps elements in a sequence, where element access and deletion are restricted to the first element in the sequence, which is called the ***front*** of the queue, and element insertion is restricted to the end of the sequence, which is called the ***rear*** of the queue. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (FIFO) principle.

The ***queue*** abstract data type (ADT) supports the following operations:

enqueue(e): Insert element e at the rear of the queue.

dequeue(): Remove element at the front of the queue; an error occurs if the queue is empty.

front(): Return, but do not remove, a reference to the front element in the queue; an error occurs if the queue is empty.

The queue ADT also includes the following supporting member functions:

size(): Return the number of elements in the queue.

empty(): Return true if the queue is empty and false otherwise.

We illustrate the operations in the queue ADT in the following example.

Example 5.4: *The following table shows a series of queue operations and their effects on an initially empty queue, Q, of integers.*

Operation	***Output***	*front* $\leftarrow Q \leftarrow$ *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*"error"*	()
empty()	*true*	()

5.2.2 The STL Queue

The Standard Template Library provides an implementation of a queue. As with the STL stack, the underlying implementation is based on the STL vector class (Sections 1.5.5 and 6.1.4). In order to declare an object of type queue, it is necessary to first include the definition file, which is called "queue." As with the STL vector, the class queue is part of the std namespace, so it is necessary either to use "std::queue" or to provide an appropriate "**using**" statement. The queue class is templated with the base type of the individual elements. For example, the code fragment below declares a queue of floats.

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

As with instances of STL vectors and stacks, an STL queue dynamically resizes itself as new elements are added.

The STL queue supports roughly the same operators as our interface, but the syntax and semantics are slightly different. Below, we list the principal member

functions. Let q be declared to be an STL queue, and let e denote a single object whose type is the same as the base type of the queue. (For example, q is a queue of floats, and e is a float.)

size(): Return the number of elements in the queue.
empty(): Return true if the queue is empty and false otherwise.
push(e): Enqueue e at the rear of the queue.
pop(): Dequeue the element at the front of the queue.
front(): Return a reference to the element at the queue's front.
back(): Return a reference to the element at the queue's rear.

Unlike our queue interface, the STL queue provides access to both the front and back of the queue. Similar to the STL stack, the result of applying any of the operations front, back, or pop to an empty STL queue is undefined. Unlike our interface, no exception is thrown, but it may very likely result in the program aborting. It is up to the programmer to be sure that no such illegal accesses are attempted.

5.2.3 A C++ Queue Interface

Our interface for the queue ADT is given in Code Fragment 5.15. As with the stack ADT, the class is templated. The queue's base element type E is provided by the user.

```
template <typename E>
class Queue {                                   // an interface for a queue
public:
  int size() const;                             // number of items in queue
  bool empty() const;                           // is the queue empty?
  const E& front() const throw(QueueEmpty);     // the front element
  void enqueue (const E& e);                    // enqueue element at rear
  void dequeue() throw(QueueEmpty);             // dequeue element at front
};
```

Code Fragment 5.15: An informal Queue interface (not a complete C++ class).

Note that the size and empty functions have the same meaning as their counterparts in the Stack ADT. These two member functions and front are known as ***accessor*** functions, for they return a value and do not change the contents of the data structure. Also note the use of the exception QueueEmpty to indicate the error state of an empty queue.

The member functions size, empty, and front are all declared to be **const**, which informs the compiler that they do not alter the contents of the queue. Note

that the member function front returns a constant reference to the top of the queue.

An error condition occurs when calling either of the functions front or dequeue on an empty queue. This is signaled by throwing an exception QueueEmpty, which is defined in Code Fragment 5.16.

```
class QueueEmpty : public RuntimeException {
public:
  QueueEmpty(const string& err) : RuntimeException(err) { }
};
```

Code Fragment 5.16: Exception thrown by functions front or dequeue when called on an empty queue. This class is derived from RuntimeException from Section 2.4.

5.2.4 A Simple Array-Based Implementation

We present a simple realization of a queue by means of an array, Q, with capacity N, for storing its elements. The main issue with this implementation is deciding how to keep track of the front and rear of the queue.

One possibility is to adapt the approach we used for the stack implementation. In particular, let $Q[0]$ be the front of the queue and have the queue grow from there. This is not an efficient solution, however, for it requires that we move all the elements forward one array cell each time we perform a dequeue operation. Such an implementation would therefore require $\Theta(n)$ time to perform the dequeue function, where n is the current number of elements in the queue. If we want to achieve constant time for each queue function, we need a different approach.

Using an Array in a Circular Way

To avoid moving objects once they are placed in Q, we define three variables, f, r, n, which have the following meanings:

- f is the index of the cell of Q storing the front of the queue. If the queue is nonempty, this is the index of the element to be removed by dequeue.
- r is an index of the cell of Q following the rear of the queue. If the queue is not full, this is the index where the element is inserted by enqueue.
- n is the current number of elements in the queue.

Initially, we set $n = 0$ and $f = r = 0$, indicating an empty queue. When we dequeue an element from the front of the queue, we decrement n and increment f to the next cell in Q. Likewise, when we enqueue an element, we increment r and increment n. This allows us to implement the enqueue and dequeue functions in constant time

Nonetheless, there is still a problem with this approach. Consider, for example, what happens if we repeatedly enqueue and dequeue a single element N different times. We would have $f = r = N$. If we were then to try to insert the element just one more time, we would get an array-out-of-bounds error, even though there is plenty of room in the queue in this case. To avoid this problem and be able to utilize all of the array Q, we let the f and r indices "wrap around" the end of Q. That is, we now view Q as a "circular array" that goes from $Q[0]$ to $Q[N-1]$ and then immediately back to $Q[0]$ again. (See Figure 5.4.)

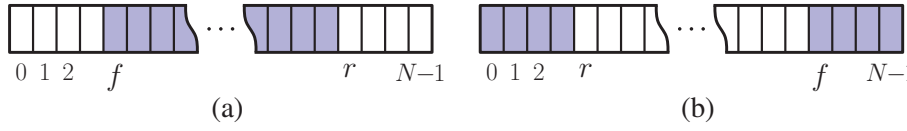

Figure 5.4: Using array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are shaded.

Using the Modulo Operator to Implement a Circular Array

Implementing this circular view of Q is actually pretty easy. Each time we increment f or r, we simply need to compute this increment as "$(f+1) \bmod N$" or "$(r+1) \bmod N$," respectively, where the operator "mod" is the ***modulo*** operator. This operator is computed for a positive number by taking the remainder after an integral division. For example, 48 divided by 5 is 9 with remainder 3, so $48 \bmod 5 = 3$. Specifically, given integers x and y, such that $x \geq 0$ and $y > 0$, $x \bmod y$ is the unique integer $0 \leq r < y$ such that $x = qy + r$, for some integer q. Recall that C++ uses "%" to denote the modulo operator.

We present our implementation in Code Fragment 5.17. Note that we have introduced a new exception, called QueueFull, to signal that no more elements can be inserted in the queue. Our implementation of a queue by means of an array is similar to that of a stack, and is left as an exercise.

The array-based queue implementation is quite efficient. All of the operations of the queue ADT are performed in $O(1)$ time. The space usage is $O(N)$, where N is the size of the array, determined at the time the queue is created. Note that the space usage is independent from the number $n < N$ of elements that are actually in the queue.

As with the array-based stack implementation, the only real disadvantage of the array-based queue implementation is that we artificially set the capacity of the queue to be some number N. In a real application, we may actually need more or less queue capacity than this, but if we have a good estimate of the number of

```
Algorithm size():
    return n
Algorithm empty():
    return (n = 0)
Algorithm front():
    if empty() then
        throw QueueEmpty exception
    return Q[f]
Algorithm dequeue():
    if empty() then
        throw QueueEmpty exception
    f ← (f + 1) mod N
    n = n − 1
Algorithm enqueue(e):
    if size() = N then
        throw QueueFull exception
    Q[r] ← e
    r ← (r + 1) mod N
    n = n + 1
```

Code Fragment 5.17: Implementation of a queue using a circular array.

elements that will be in the queue at the same time, then the array-based implementation is quite efficient. One such possible application of a queue is dynamic memory allocation in C++, which is discussed in Chapter 14.

5.2.5 Implementing a Queue with a Circularly Linked List

In this section, we present a C++ implementation of the queue ADT using a linked representation. Recall that we delete from the head of the queue and insert at the rear. Thus, unlike our linked stack of Code Fragment 5.7, we cannot use our singly linked list class, since it provides efficient access only to one side of the list. Instead, our approach is to use the circularly linked list, called CircleList, which was introduced earlier in Section 3.4.1.

Recall that CircleList maintains a pointer, called the ***cursor***, which points to one node of the list. Also recall that CircleList provides two member functions, back and front. The function back returns a reference to the element to which the cursor points, and the function front returns a reference to the element that immediately follows it in the circular list. In order to implement a queue, the element referenced by back will be the rear of the queue and the element referenced by front will be the front. (Why would it not work to reverse matters using the back of the circular list

as the front of the queue and the front of the circular list as the rear of the queue?)

Also recall that CircleList supports the following modifier functions. The function add inserts a new node just after the cursor, the function remove removes the node immediately following the cursor, and the function advance moves the cursor forward to the next node of the circular list.

In order to implement the queue operation enqueue, we first invoke the function add, which inserts a new element just after the cursor, that is, just after the rear of the queue. We then invoke advance, which advances the cursor to this new element, thus making the new node the rear of the queue. The process is illustrated in Figure 5.5.

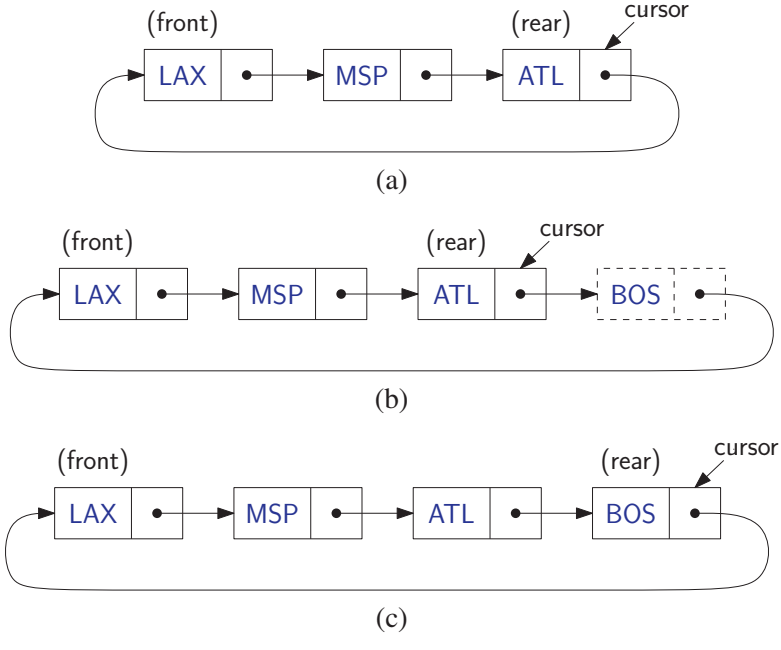

Figure 5.5: Enqueueing "BOS" into a queue represented as a circularly linked list: (a) before the operation; (b) after adding the new node; (c) after advancing the cursor.

In order to implement the queue operation dequeue, we invoke the function remove, thus removing the node just after the cursor, that is, the front of the queue. The process is illustrated in Figure 5.6.

The class structure for the resulting class, called LinkedQueue, is shown in Code Fragment 5.18. To avoid the syntactic messiness inherent in C++ templated classes, we have chosen not to implement a fully generic templated class. Instead, we have opted to define a type for the queue's elements, called Elem. In this example, we define Elem to be of type string. The queue is stored in the circular list

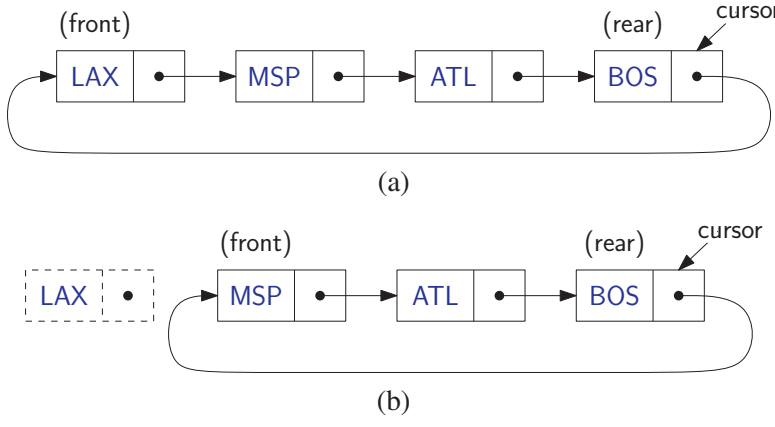

Figure 5.6: Dequeueing an element (in this case "LAX") from the front queue represented as a circularly linked list: (a) before the operation; (b) after removing the node immediately following the cursor.

data structure C. In order to support the size function (which CircleList does not provide), we also maintain the queue size in the member n.

```
typedef string Elem;                                  // queue element type
class LinkedQueue {                                   // queue as doubly linked list
public:
  LinkedQueue();                                      // constructor
  int size() const;                                   // number of items in the queue
  bool empty() const;                                 // is the queue empty?
  const Elem& front() const throw(QueueEmpty); // the front element
  void enqueue(const Elem& e);                        // enqueue element at rear
  void dequeue() throw(QueueEmpty);                   // dequeue element at front
private:                                              // member data
  CircleList C;                                       // circular list of elements
  int n;                                              // number of elements
};
```

Code Fragment 5.18: The class LinkedQueue, an implementation of a queue based on a circularly linked list.

In Code Fragment 5.19, we present the implementations of the constructor and the basic accessor functions, size, empty, and front. Our constructor creates the initial queue and initializes n to zero. We do not provide an explicit destructor, relying instead on the destructor provided by CircleList. Observe that the function front throws an exception if an attempt is made to access the first element of an empty queue. Otherwise, it returns the element referenced by the front of the circular list, which, by our convention, is also the front element of the queue.

```
LinkedQueue::LinkedQueue()                    // constructor
  : C(), n(0) { }

int LinkedQueue::size() const                 // number of items in the queue
  { return n; }

bool LinkedQueue::empty() const               // is the queue empty?
  { return n == 0; }
                                              // get the front element
const Elem& LinkedQueue::front() const throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("front of empty queue");
  return C.front();                           // list front is queue front
}
```

Code Fragment 5.19: Constructor and accessor functions for the LinkedQueue class.

The definition of the queue operations, enqueue and dequeue are presented in Code Fragment 5.20. Recall that enqueuing involves invoking the add function to insert the new item immediately following the cursor and then advancing the cursor. Before dequeuing, we check whether the queue is empty, and, if so, we throw an exception. Otherwise, dequeuing involves removing the element that immediately follows the cursor. In either case, we update the number of elements in the queue.

```
                                              // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                   // insert after cursor
  C.advance();                                // ...and advance
  n++;
}
                                              // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                 // remove from list front
  n--;
}
```

Code Fragment 5.20: The enqueue and dequeue functions for LinkedQueue.

Observe that, all the operations of the queue ADT are implemented in $O(1)$ time. Therefore this implementation is quite efficient. Unlike the array-based implementation, by expanding and contracting dynamically, this implementation uses space proportional to the number of elements that are present in the queue at any time.

5.3 Double-Ended Queues

Consider now a queue-like data structure that supports insertion and deletion at both the front and the rear of the queue. Such an extension of a queue is called a ***double-ended queue***, or ***deque***, which is usually pronounced "deck" to avoid confusion with the dequeue function of the regular queue ADT, which is pronounced like the abbreviation "D.Q." An easy way to remember the "deck" pronunciation is to observe that a deque is like a deck of cards in the hands of a crooked card dealer—it is possible to deal off both the top and the bottom.

5.3.1 The Deque Abstract Data Type

The functions of the deque ADT are as follows, where D denotes the deque:

insertFront(e): Insert a new element e at the beginning of the deque.

insertBack(e): Insert a new element e at the end of the deque.

eraseFront(): Remove the first element of the deque; an error occurs if the deque is empty.

eraseBack(): Remove the last element of the deque; an error occurs if the deque is empty.

Additionally, the deque includes the following support functions:

front(): Return the first element of the deque; an error occurs if the deque is empty.

back(): Return the last element of the deque; an error occurs if the deque is empty.

size(): Return the number of elements of the deque.

empty(): Return true if the deque is empty and false otherwise.

Example 5.5: *The following example shows a series of operations and their effects on an initially empty deque, D, of integers.*

Operation	***Output***	D
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	5	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	7	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

5.3.2 The STL Deque

As with the stack and queue, the Standard Template Library provides an implementation of a deque. The underlying implementation is based on the STL vector class (Sections 1.5.5 and 6.1.4). The pattern of usage is similar to that of the STL stack and STL queue. First, we need to include the definition file "deque." Since it is a member of the std namespace, we need to either preface each usage "std::deque" or provide an appropriate "**using**" statement. The deque class is templated with the base type of the individual elements. For example, the code fragment below declares a deque of strings.

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

As with STL stacks and queues, an STL deque dynamically resizes itself as new elements are added.

With minor differences, the STL deque class supports the same operators as our interface. Here is a list of the principal operations.

size():	Return the number of elements in the deque.
empty():	Return true if the deque is empty and false otherwise.
push_front(e):	Insert e at the beginning the deque.
push_back(e):	Insert e at the end of the deque.
pop_front():	Remove the first element of the deque.
pop_back():	Remove the last element of the deque.
front():	Return a reference to the deque's first element.
back():	Return a reference to the deque's last element.

Similar to STL stacks and queues, the result of applying any of the operations front, back, push_front, or push_back to an empty STL queue is undefined. Thus, no exception is thrown, but the program may abort.

5.3.3 Implementing a Deque with a Doubly Linked List

In this section, we show how to implement the deque ADT using a linked representation. As with the queue, a deque supports efficient access at both ends of the list, so our implementation is based on the use of a doubly linked list. Again, we use the doubly linked list class, called DLinkedList, which was presented earlier in Section 3.3.3. We place the front of the deque at the head of the linked list and the rear of the queue at the tail. An illustration is provided in Figure 5.7.

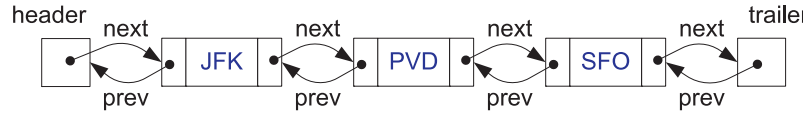

Figure 5.7: A doubly linked list with sentinels, *header* and *trailer*. The front of our deque is stored just after the header ("JFK"), and the back of our deque is stored just before the trailer ("SFO").

The definition of the resulting class, called LinkedDeque, is shown in Code Fragment 5.21. The deque is stored in the data member D. In order to support the size function, we also maintain the queue size in the member n. As in some of our earlier implementations, we avoid the syntactic messiness inherent in C++ templated classes, and instead use just a type definition to define the deque's base element type.

```
typedef string Elem;                          // deque element type
class LinkedDeque {                           // deque as doubly linked list
public:
  LinkedDeque();                              // constructor
  int size() const;                           // number of items in the deque
  bool empty() const;                         // is the deque empty?
  const Elem& front() const throw(DequeEmpty); // the first element
  const Elem& back() const throw(DequeEmpty); // the last element
  void insertFront(const Elem& e);            // insert new first element
  void insertBack(const Elem& e);             // insert new last element
  void removeFront() throw(DequeEmpty);       // remove first element
  void removeBack() throw(DequeEmpty);        // remove last element
private:                                      // member data
  DLinkedList D;                              // linked list of elements
  int n;                                      // number of elements
};
```

Code Fragment 5.21: The class structure for class LinkedDeque.

We have not bothered to provide an explicit destructor, because the DLinkedList class provides its own destructor, which is automatically invoked when our LinkedDeque structure is destroyed.

Most of the member functions for the LinkedDeque class are straightforward generalizations of the corresponding functions of the LinkedQueue class, so we have omitted them. In Code Fragment 5.22, we present the implementations of the member functions for performing insertions and removals of elements from the deque. Observe that, in each case, we simply invoke the appropriate operation from the underlying DLinkedList object.

```
                                                    // insert new first element
void LinkedDeque::insertFront(const Elem& e) {
  D.addFront(e);
  n++;
}
                                                    // insert new last element
void LinkedDeque::insertBack(const Elem& e) {
  D.addBack(e);
  n++;
}
                                                    // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeFront of empty deque");
  D.removeFront();
  n--;
}
                                                    // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeBack of empty deque");
  D.removeBack();
  n--;
}
```

Code Fragment 5.22: The insertion and removal functions for LinkedDeque.

Table 5.2 shows the running times of functions in a realization of a deque by a doubly linked list. Note that every function of the deque ADT runs in $O(1)$ time.

Operation	*Time*
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

Table 5.2: Performance of a deque realized by a doubly linked list. The space usage is $O(n)$, where n is number of elements in the deque.

5.3.4 Adapters and the Adapter Design Pattern

An inspection of code fragments of Sections 5.1.5, 5.2.5, and 5.3.3, reveals a common pattern. In each case, we have taken an existing data structure and ***adapted*** it

to be used for a special purpose. For example, in Section 5.3.3, we showed how the DLinkedList class of Section 3.3.3 could be adapted to implement a deque. Except for the additional feature of keeping track of the number of elements, we have simply mapped each deque operation (such as insertFront) to the corresponding operation of DLinkedList (such as the addFront).

An ***adapter*** (also called a ***wrapper***) is a data structure, for example, a class in C++, that translates one interface to another. You can think of an adapter as the software analogue to electric power plug adapters, which are often needed when you want to plug your electric appliances into electric wall sockets in different countries.

As an example of adaptation, observe that it is possible to implement the stack ADT by means of a deque data structure. That is, we can translate each stack operation to a functionally equivalent deque operation. Such a mapping is presented in Table 5.3.

Stack Method	*Deque Implementation*
size()	size()
empty()	empty()
top()	front()
push(o)	insertFront(o)
pop()	eraseFront()

Table 5.3: Implementing a stack with a deque.

Note that, because of the deque's symmetry, performing insertions and removals from the rear of the deque would have been equally efficient.

Likewise, we can develop the correspondences for the queue ADT, as shown in Table 5.4.

Queue Method	*Deque Implementation*
size()	size()
empty()	empty()
front()	front()
enqueue(e)	insertBack(e)
dequeue()	eraseFront()

Table 5.4: Implementing a queue with a deque.

As a more concrete example of the adapter design pattern, consider the code fragment shown in Code Fragment 5.23. In this code fragment, we present a class DequeStack, which implements the stack ADT. It's implementation is based on translating each stack operation to the corresponding operation on a LinkedDeque, which was introduced in Section 5.3.3.

```
typedef string Elem;                          // element type
class DequeStack {                            // stack as a deque
public:
  DequeStack();                               // constructor
  int size() const;                           // number of elements
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:
  LinkedDeque D;                              // deque of elements
};
```

Code Fragment 5.23: Implementation of the Stack interface by means of a deque.

The implementations of the various member functions are presented in Code Fragment 5.24. In each case, we translate some stack operation into the corresponding deque operation.

```
DequeStack::DequeStack()                      // constructor
  : D() { }

                                              // number of elements
int DequeStack::size() const
  { return D.size(); }

                                              // is the stack empty?
bool DequeStack::empty() const
  { return D.empty(); }

                                              // the top element
const Elem& DequeStack::top() const throw(StackEmpty) {
  if (empty())
    throw StackEmpty("top of empty stack");
  return D.front();
}

                                              // push element onto stack
void DequeStack::push(const Elem& e)
  { D.insertFront(e); }

                                              // pop the stack
void DequeStack::pop() throw(StackEmpty)
{
  if (empty())
    throw StackEmpty("pop of empty stack");
  D.removeFront();
}
```

Code Fragment 5.24: Implementation of the Stack interface by means of a deque.