Lecture 28:

Data Structures

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
May 14th, 2010

20.1 Introduction

- We’ve studied fixed-size data structures such as one-dimensional arrays and two-dimensional arrays.
- This chapter introduces dynamic data structures that grow and shrink during execution.
- Linked lists are collections of data items logically “lined up in a row”—insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file-system directories and compilation of expressions into machine language.

20.2 Self-Referential Classes

- A self-referential class contains a pointer member that points to a class object of the same class type.
- Sample `Node` class definition:

  -
    ```
    class Node
    {
    public:
       Node( int ); // constructor
       void setData( int ); // set data member
       int getData() const; // get data member
       void setNextPtr( Node * ); // set pointer to next Node
       Node *getNextPtr() const; // get pointer to next Node
    private:
       int data; // data stored in this Node
       Node *nextPtr; // pointer to another object of same type
    }; // end class Node
    ```

20.2 Self-Referential Classes (cont.)

- Member `nextPtr` points to an object of type `Node`—another object of the same type as the one being declared here, hence the term “self-referential class.”
- Member `nextPtr` is referred to as a link—i.e., `nextPtr` can “tie” an object of type `Node` to another object of the same type.
- Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees.
- Figure 20.1 illustrates two self-referential class objects linked together to form a list.
- Note that a slash—representing a null (`0`) pointer—is placed in the link member of the second self-referential class object to indicate that the link does not point to another object.
- The slash is only for illustration purposes; it does not correspond to the backslash character in C++.
- A null pointer normally indicates the end of a data structure just as the null character (`'\0'`) indicates the end of a string.

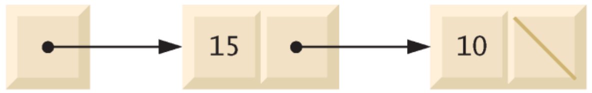


Fig. 20.1 | Two self-referential class objects linked together.

Common Programming Error 20.1

Not setting the link in the last node of a linked data structure to null (0) is a (possibly fatal) logic error.

20.3 Dynamic Memory Allocation and Data Structures

- Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes.
- When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future.
- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system.
- Often, the limits are much smaller, because available memory must be shared among many programs.

20.3 Dynamic Memory Allocation and Data Structures (cont.)

- The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type.
- For example, the following statement allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`.

  -
    ```
    // create Node with data 10
    Node *newPtr = new Node( 10 );
    ```
- If no memory is available, `new` throws a `bad_alloc` exception.
- The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`—the memory is returned to the system so that the memory can be reallocated in the future.

20.3 Dynamic Memory Allocation and Data Structures (cont.)

- To free memory dynamically allocated by the preceding `new`, use the statement
  - `delete newPtr;`
- Note that `newPtr` itself is not deleted; rather the space `newPtr` points to is deleted.
- If pointer `newPtr` has the null pointer value `0`, the preceding statement has no effect.

20.4 Linked Lists

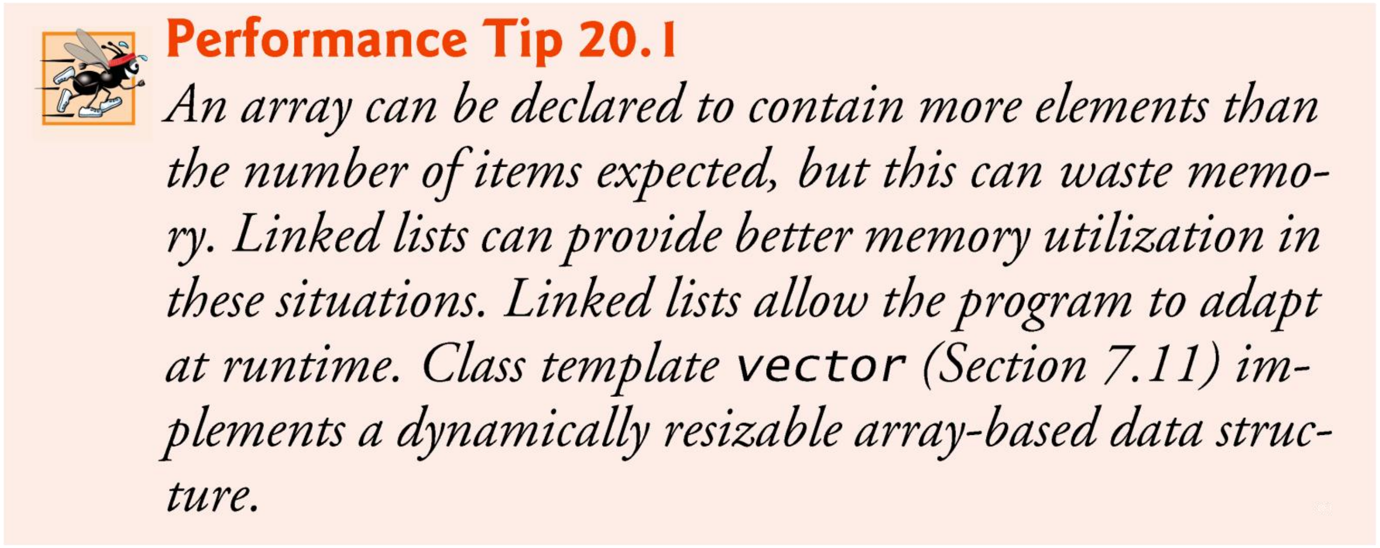
- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links—hence, the term “linked” list.
- A linked list is accessed via a pointer to the list’s first node.
- Each subsequent node is accessed via the link-pointer member stored in the previous node.
- By convention, the link pointer in the last node of a list is set to null (0) to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including objects of other classes.

20.4 Linked Lists (cont.)

- Stacks and queues are also linear data structures and, as we’ll see, can be viewed as constrained versions of linked lists.
- Trees are nonlinear data structures.

20.4 Linked Lists (cont.)

- Lists of data can be stored in arrays, but linked lists provide several advantages.
- A linked list is appropriate when the number of data elements to be represented at one time is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of a “conventional” C++ array, however, cannot be altered, because the array size is fixed at compile time.
- “Conventional” arrays can become full.
- Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

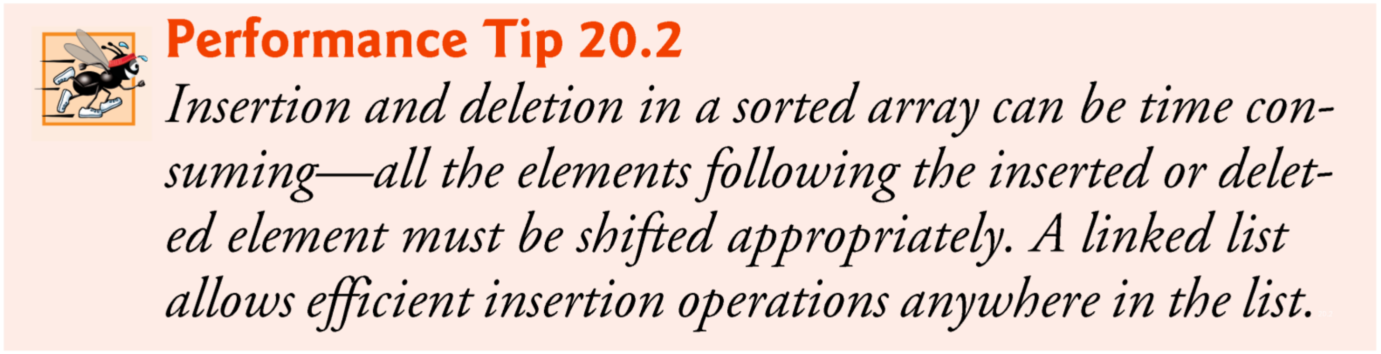


Performance Tip 20.1

An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Class template `vector` *(Section 7.11) implements a dynamically resizable array-based data structure.*

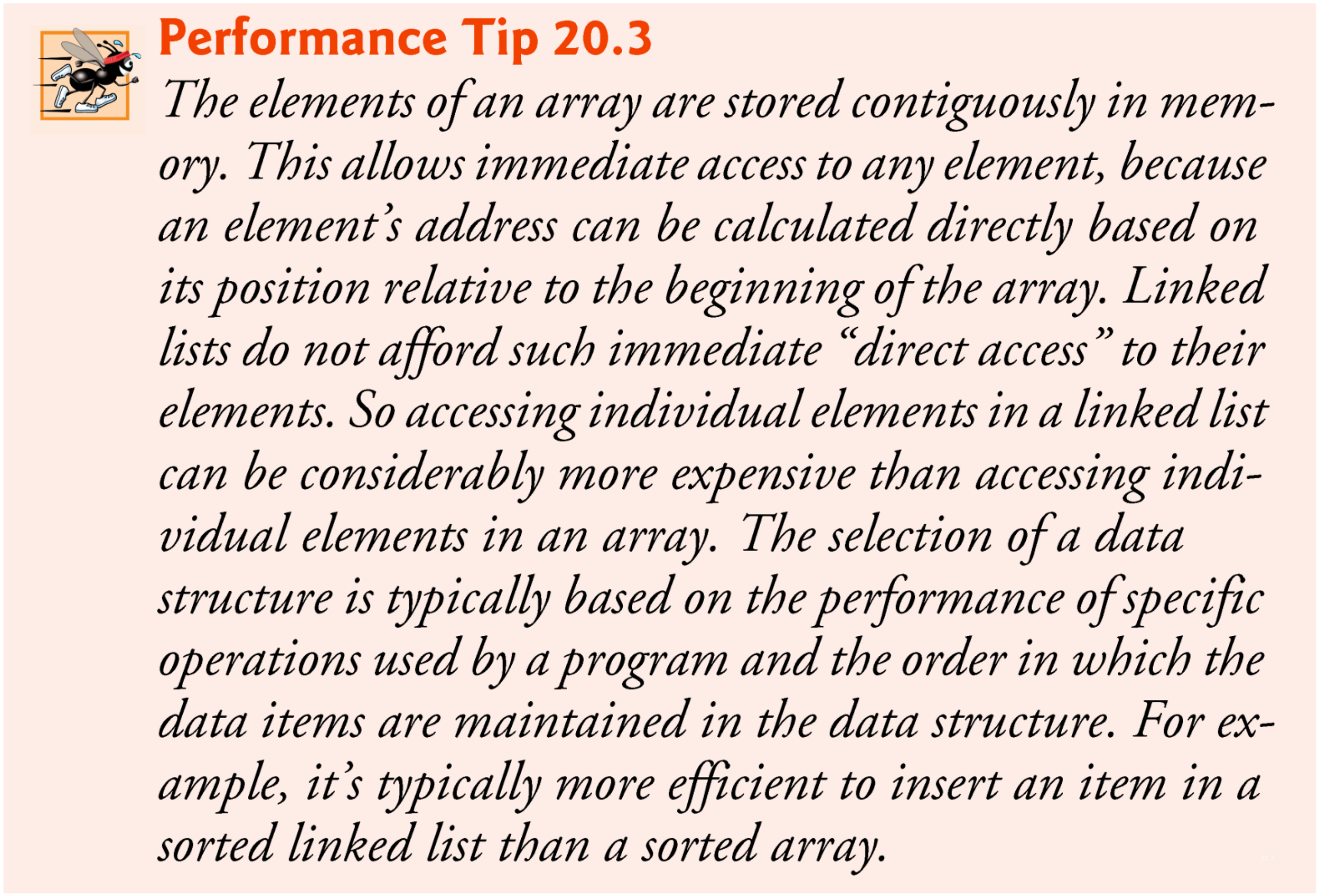
20.4 Linked Lists (cont.)

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
- Existing list elements do not need to be moved.
- Pointers merely need to be updated to point to the correct node.



Performance Tip 20.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

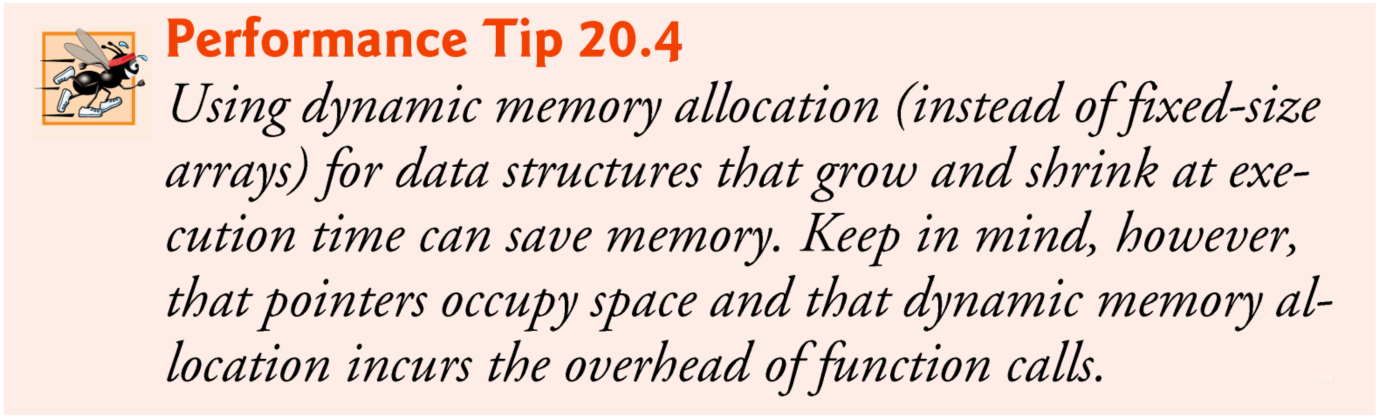


Performance Tip 20.3

The elements of an array are stored contiguously in memory. This allows immediate access to any element, because an element's address can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate "direct access" to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it's typically more efficient to insert an item in a sorted linked list than a sorted array.

20.4 Linked Lists (cont.)

- Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous.
- Figure 20.2 illustrates a linked list with several nodes.



Performance Tip 20.4

Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

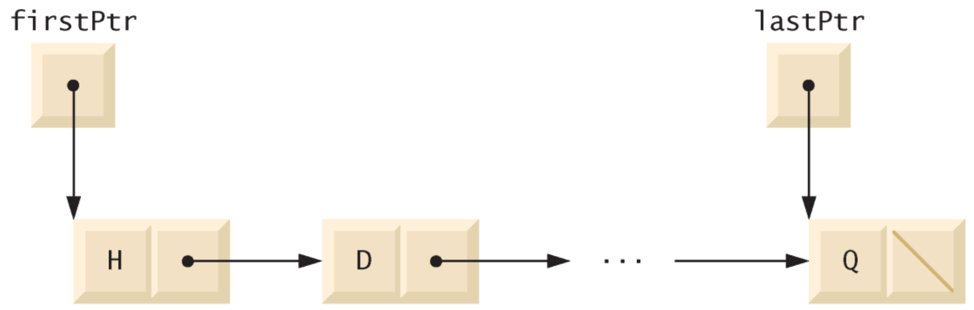


Fig. 20.2 | A graphical representation of a list.

```
 1  // Fig. 20.3: ListNode.h
 2  // Template ListNode class definition.
 3  #ifndef LISTNODE_H
 4  #define LISTNODE_H
 5
 6  // forward declaration of class List required to announce that class
 7  // List exists so it can be used in the friend declaration at line 13
 8  template< typename NODETYPE > class List;
 9
10  template< typename NODETYPE >
11  class ListNode
12  {
13     friend class List< NODETYPE >; // make List a friend
14
15  public:
16     ListNode( const NODETYPE & ); // constructor
17     NODETYPE getData() const; // return data in node
18  private:
19     NODETYPE data; // data
20     ListNode< NODETYPE > *nextPtr; // next node in list
21  }; // end class ListNode
22
```

Fig. 20.3 | `ListNode` class-template definition. (Part 1 of 2.)

```cpp
// constructor
template< typename NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 )
{
   // empty body
} // end ListNode constructor

// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData

#endif
```

Fig. 20.3 | `ListNode` class-template definition. (Part 2 of 2.)

20.4 Linked Lists (cont.)

- The primary `List` functions are `insertAtFront` (lines 62–74), `insertAtBack` (lines 77–89), `removeFromFront` (lines 92–110) and `removeFromBack` (lines 113–140).
- Function `isEmpty` (lines 143–147) is called a predicate function
  - it does not alter the `List`; rather, it determines whether the `List` is empty (i.e., the pointer to the first node of the `List` is null).
  - If the `List` is empty, `true` is returned; otherwise, `false` is returned.
- Function `print` (lines 158–178) displays the `List`'s contents.
- Utility function `getNewNode` (lines 150–155) returns a dynamically allocated `ListNode` object.
  - Called from functions `insertAtFront` and `insertAtBack`.

```
// Fig. 20.4: List.h
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
#include "ListNode.h" // ListNode class definition
using namespace std;

template< typename NODETYPE >
class List
{
public:
   List(); // constructor
   ~List(); // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
```

Fig. 20.4 | List class-template definition. (Part 1 of 9.)

```
private:
   ListNode< NODETYPE > *firstPtr; // pointer to first node
   ListNode< NODETYPE > *lastPtr; // pointer to last node

   // utility function to allocate new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
}; // end class List

// default constructor
template< typename NODETYPE >
List< NODETYPE >::List()
   : firstPtr( 0 ), lastPtr( 0 )
{
   // empty body
} // end List constructor

```

Fig. 20.4 | `List` class-template definition. (Part 2 of 9.)

```
// destructor
template< typename NODETYPE >
List< NODETYPE >::~List()
{
   if ( !isEmpty() ) // List is not empty
   {
      cout << "Destroying nodes ...\n";

      ListNode< NODETYPE > *currentPtr = firstPtr;
      ListNode< NODETYPE > *tempPtr;

      while ( currentPtr != 0 ) // delete remaining nodes
      {
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      } // end while
   } // end if

   cout << "All nodes destroyed\n\n";
} // end List destructor

```

Fig. 20.4 | `List` class-template definition. (Part 3 of 9.)

```
// insert node at front of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      newPtr->nextPtr = firstPtr; // point new node to previous 1st node
      firstPtr = newPtr; // aim firstPtr at new node
   } // end else
} // end function insertAtFront

```

Fig. 20.4 | `List` class-template definition. (Part 4 of 9.)

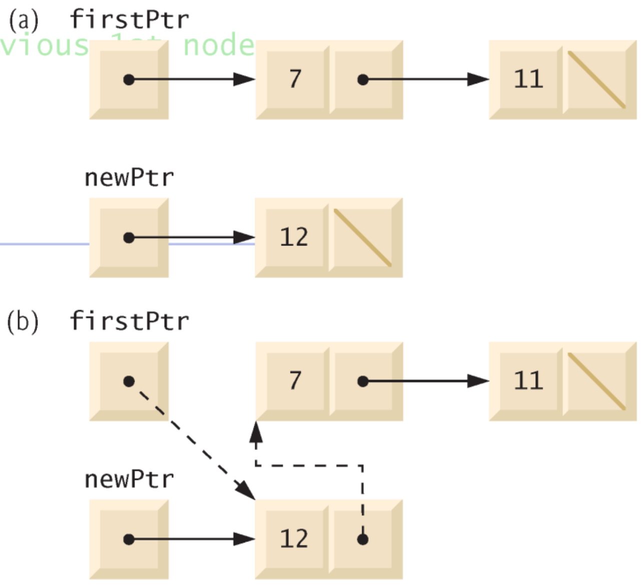


Fig. 20.6 | Operation `insertAtFront` represented graphically.

```
// insert node at back of list
template< typename NODETYPE >
void List< NODETYPE >::insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      lastPtr->nextPtr = newPtr; // update previous last node
      lastPtr = newPtr; // new last node
   } // end else
} // end function insertAtBack

```

Fig. 20.4 | `List` class-template definition. (Part 5 of 9.)

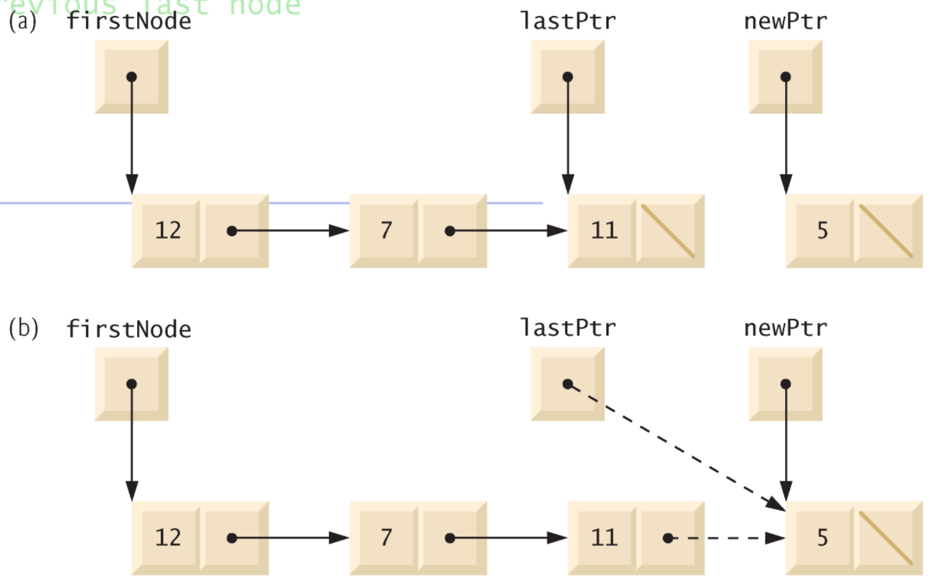


Fig. 20.7 | Operation `insertAtBack` represented graphically.

```
// delete node from front of list
template< typename NODETYPE >
bool List< NODETYPE >::removeFromFront( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
         firstPtr = firstPtr->nextPtr; // point to previous 2nd node

      value = tempPtr->data; // return data being removed
      delete tempPtr; // reclaim previous front node
      return true; // delete successful
   } // end else
} // end function removeFromFront

```

Fig. 20.4 | List class-template definition. (Part 6 of 9.)

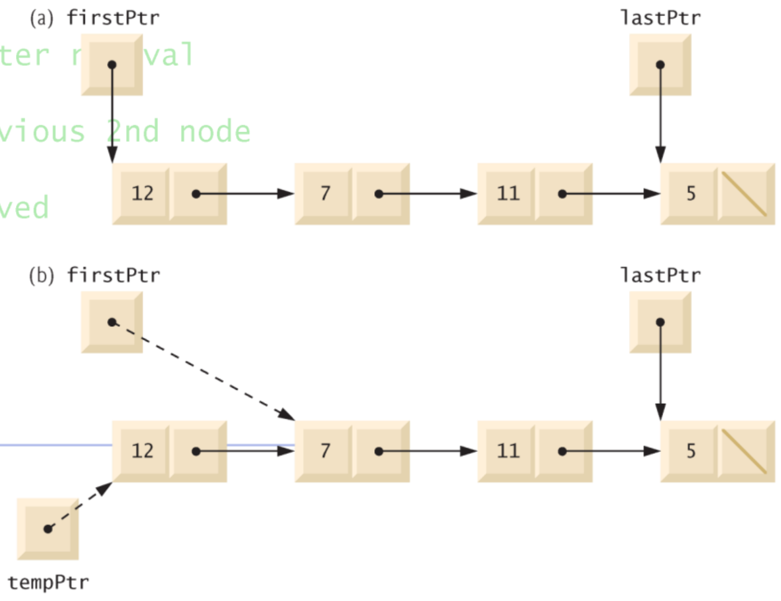


Fig. 20.8 | Operation removeFromFront represented graphically.

```
// delete node from back of list
template< typename NODETYPE >
bool List< NODETYPE >::removeFromBack( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete

      if ( firstPtr == lastPtr ) // List has one element
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
      {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node

         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = 0; // this is now the last node
      } // end else

```

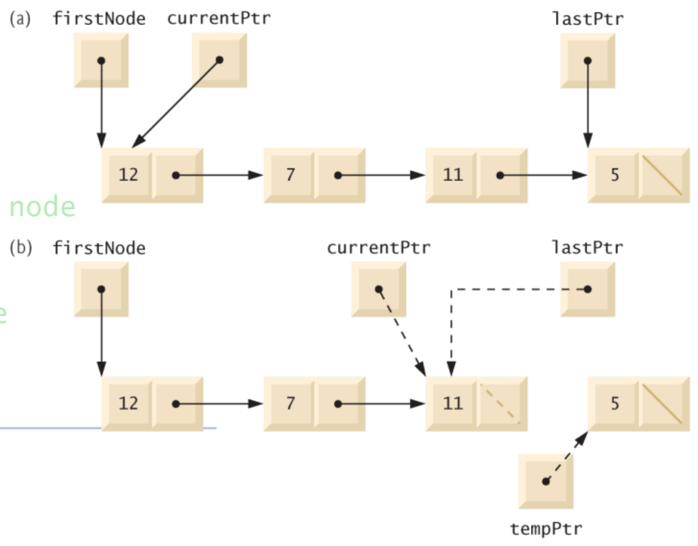


Fig. 20.9 | Operation `removeFromBack` represented graphically.

Fig. 20.4 | `List` class-template definition. (Part 7 of 9.)

```
         value = tempPtr->data; // return value from old last node
         delete tempPtr; // reclaim former last node
         return true; // delete successful
      } // end else
   } // end function removeFromBack

   // is List empty?
   template< typename NODETYPE >
   bool List< NODETYPE >::isEmpty() const
   {
      return firstPtr == 0;
   } // end function isEmpty

   // return pointer to newly allocated node
   template< typename NODETYPE >
   ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
      const NODETYPE &value )
   {
      return new ListNode< NODETYPE >( value );
   } // end function getNewNode

```

Fig. 20.4 | `List` class-template definition. (Part 8 of 9.)

```
// display contents of List
template< typename NODETYPE >
void List< NODETYPE >::print() const
{
   if ( isEmpty() ) // List is empty
   {
      cout << "The list is empty\n\n";
      return;
   } // end if

   ListNode< NODETYPE > *currentPtr = firstPtr;

   cout << "The list is: ";

   while ( currentPtr != 0 ) // get element data
   {
      cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   } // end while

   cout << "\n\n";
} // end function print

#endif
```

Fig. 20.4 | `List` class-template definition. (Part 9 of 9.)

20.4 Linked Lists (cont.)

- In Fig. 20.5, Lines 69 and 73 create `List` objects for types `int` and `double`, respectively.
- Lines 70 and 74 invoke the `testList` function template to manipulate objects.

```cpp
// Fig. 20.5: Fig21_05.cpp
// List class test program.
#include <iostream>
#include <string>
#include "List.h" // List class definition
using namespace std;

// display program instructions to user
void instructions()
{
   cout << "Enter one of the following:\n"
      << "  1 to insert at beginning of list\n"
      << "  2 to insert at end of list\n"
      << "  3 to delete from beginning of list\n"
      << "  4 to delete from end of list\n"
      << "  5 to end list processing\n";
} // end function instructions

```

Fig. 20.5 | Manipulating a linked list. (Part 1 of 8.)

```
// function to test a List
template< typename T >
void testList( List< T > &listObject, const string &typeName )
{
   cout << "Testing a List of " << typeName << " values\n";
   instructions(); // display instructions

   int choice; // store user choice
   T value; // store input value

   do // perform user-selected actions
   {
      cout << "? ";
      cin >> choice;

      switch ( choice )
      {
         case 1: // insert at beginning
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
            break;
```

Fig. 20.5 | Manipulating a linked list. (Part 2 of 8.)

```
            case 2: // insert at end
               cout << "Enter " << typeName << ": ";
               cin >> value;
               listObject.insertAtBack( value );
               listObject.print();
               break;
            case 3: // remove from beginning
               if ( listObject.removeFromFront( value ) )
                  cout << value << " removed from list\n";

               listObject.print();
               break;
            case 4: // remove from end
               if ( listObject.removeFromBack( value ) )
                  cout << value << " removed from list\n";

               listObject.print();
               break;
         } // end switch
      } while ( choice < 5 ); // end do...while

      cout << "End list test\n\n";
   } // end function testList

```

Fig. 20.5 | Manipulating a linked list. (Part 3 of 8.)

```cpp
int main()
{
   // test List of int values
   List< int > integerList;
   testList( integerList, "integer" );

   // test List of double values
   List< double > doubleList;
   testList( doubleList, "double" );
} // end main
```

```
Testing a List of integer values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter integer: 1
The list is: 1
```

Fig. 20.5 | Manipulating a linked list. (Part 4 of 8.)

```
? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3
```

Fig. 20.5 | Manipulating a linked list. (Part 5 of 8.)

```
? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
  1 to insert at beginning of list
  2 to insert at end of list
  3 to delete from beginning of list
  4 to delete from end of list
  5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3
```

Fig. 20.5 | Manipulating a linked list. (Part 6 of 8.)

```
? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3
```

Fig. 20.5 | Manipulating a linked list. (Part 7 of 8.)

```
? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Fig. 20.5 | Manipulating a linked list. (Part 8 of 8.)

20.4 Linked Lists (cont.)

- The kind of linked list we've been discussing is a singly linked list—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence."
- This list terminates with a node whose pointer member has the value 0.
- A singly linked list may be traversed in only one direction.
- A circular, singly linked list (Fig. 20.10) begins with a pointer to the first node, and each node contains a pointer to the next node.
- The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

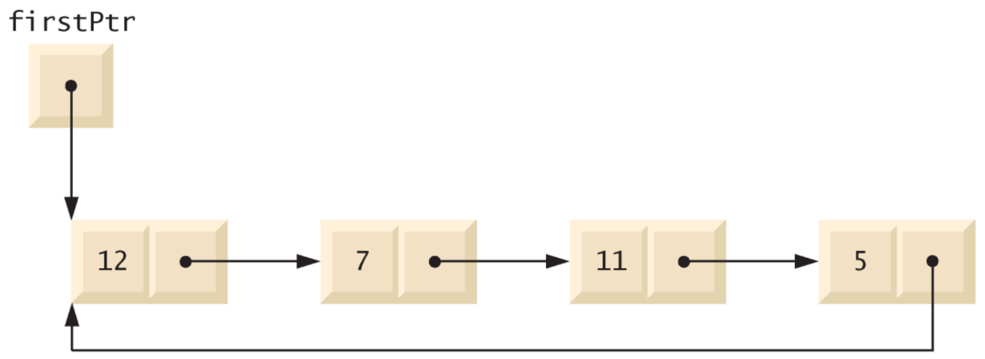


Fig. 20.10 | Circular, singly linked list.

20.4 Linked Lists (cont.)

- A doubly linked list (Fig. 20.11) allows traversals both forward and backward.
- Such a list is often implemented with two “start pointers”—one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal.
- Each node has both a forward pointer to the next node in the list in the forward direction and a backward pointer to the next node in the list in the backward direction.
- If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list.
- Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

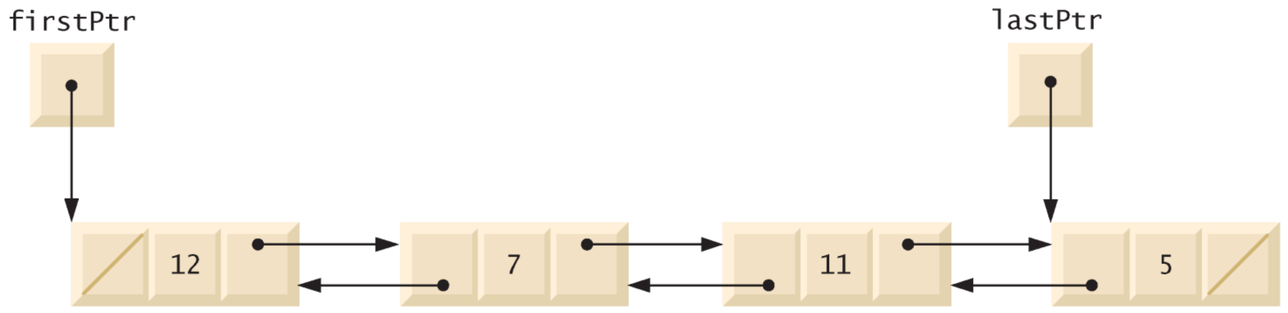


Fig. 20.11 | Doubly linked list.

20.4 Linked Lists (cont.)

- In a circular, doubly linked list (Fig. 20.12), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the “circle.”

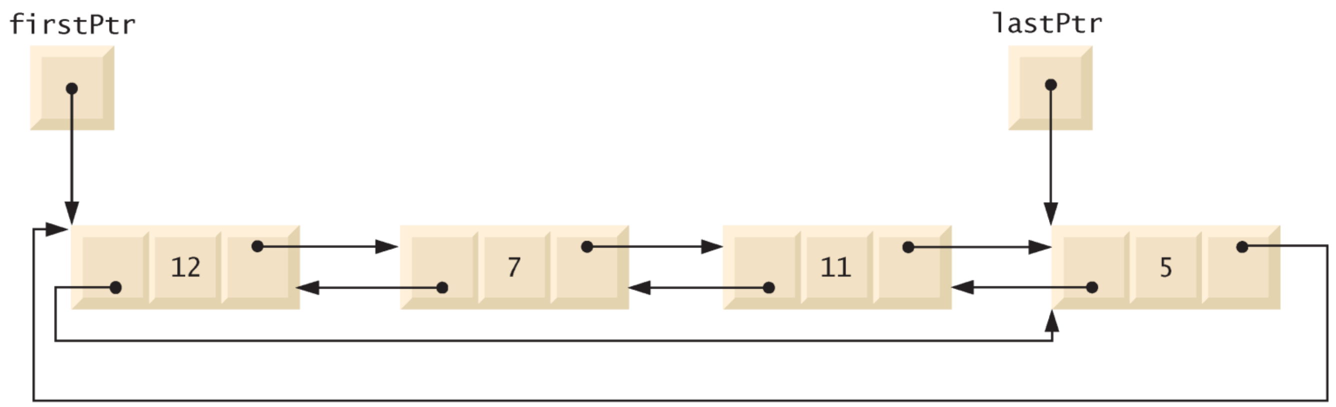


Fig. 20.12 | Circular, doubly linked list.

20.5 Stacks

- Chapter 14, Templates, explained the notion of a stack class template with an underlying array implementation.
- In this section, we use an underlying pointer-based linked-list implementation.
- A stack data structure allows nodes to be added to the stack and removed from the stack only at the top.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.

20.5 Stacks (cont.)

- One way to implement a stack is as a constrained version of a linked list.
- In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.
- The primary member functions used to manipulate a stack are `push` and `pop`.
- Function `push` inserts a new node at the top of the stack.
- Function `pop` removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

20.5 Stacks (cont.)

- Stacks have many interesting applications.
- For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller.
- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.
- Section 6.11 discusses the function call stack in detail.
- Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function.
- When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function’s automatic variables is popped off the stack and those variables are no longer known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

20.5 Stacks (cont.)

- We’ll take advantage of the close relationship between lists and stacks to implement a stack class primarily by reusing a list class.
- First, we implement the stack class through `private` inheritance of the list class.
- Then we implement an identically performing stack class through composition by including a list object as a `private` member of a stack class.
- All of the data structures in this chapter, including these two stack classes, are implemented as templates to encourage further reusability.

20.5 Stacks (cont.)

- The program of Figs. 20.13–20.14 creates a `Stack` class template (Fig. 20.13) primarily through `private` inheritance (line 9) of the `List` class template of Fig. 20.4.
- We want the `Stack` to have member functions `push` (lines 13–16), `pop` (lines 19–22), `isStackEmpty` (lines 25–28) and `printStack` (lines 31–34).
  - These are essentially the `insertAtFront`, `removeFromFront`, `isEmpty` and `print` functions of the `List` class template.

20.5 Stacks (cont.)

- Of course, the `List` class template contains other member functions (i.e., `insertAtBack` and `removeFromBack`) that we would not want to make accessible through the `public` interface to the `Stack` class.
- So when we indicate that the `Stack` class template is to inherit from the `List` class template, we specify `private` inheritance.
- This makes all the `List` class template's member functions `private` in the `Stack` class template.
- When we implement the `Stack`'s member functions, we then have each of these call the appropriate member function of the `List` class—`push` calls `insertAtFront` (line 15), `pop` calls `removeFromFront` (line 21), `isStackEmpty` calls `isEmpty` (line 27) and `printStack` calls `print` (line 33)—this is referred to as delegation.

```cpp
// Fig. 20.13: Stack.h
// Template Stack class definition derived from class List.
#ifndef STACK_H
#define STACK_H

#include "List.h" // List class definition

template< typename STACKTYPE >
class Stack : private List< STACKTYPE >
{
public:
   // push calls the List function insertAtFront
   void push( const STACKTYPE &data )
   {
      insertAtFront( data );
   } // end function push

   // pop calls the List function removeFromFront
   bool pop( STACKTYPE &data )
   {
      return removeFromFront( data );
   } // end function pop

```

Fig. 20.13 | `Stack` class-template definition. (Part 1 of 2.)

```
    // isStackEmpty calls the List function isEmpty
    bool isStackEmpty() const
    {
       return this->isEmpty();
    } // end function isStackEmpty

    // printStack calls the List function print
    void printStack() const
    {
       this->print();
    } // end function print
}; // end class Stack

#endif
```

Fig. 20.13 | `Stack` class-template definition. (Part 2 of 2.)

20.5 Stacks (cont.)

- The explicit use of the `this` pointer on lines 27 and 33 is required so the compiler can resolve identifiers in template definitions properly.
- A dependent name is an identifier that depends on a template parameter.
- For example, the call to `removeFromFront` (line 21) depends on the argument `data` which has a type that is dependent on the template parameter `STACKTYPE`.
- Resolution of dependent names occurs when the template is instantiated.

20.5 Stacks (cont.)

- In contrast, the identifier for a function that takes no arguments like `isEmpty` or `print` in the `List` superclass is a non-dependent name.
- Such identifiers are normally resolved at the point where the template is defined.
- If the template has not yet been instantiated, then the code for the function with the non-dependent name does not yet exist and some compilers will generate compilation errors.
- Adding the explicit use of `this->` in lines 27 and 33 makes the calls to the base class's member functions dependent on the template parameter and ensures that the code will compile properly.

```cpp
// Fig. 20.14: Fig21_14.cpp
// Template Stack class test program.
#include <iostream>
#include "Stack.h" // Stack class definition
using namespace std;

int main()
{
   Stack< int > intStack; // create Stack of ints

   cout << "processing an integer Stack" << endl;

   // push integers onto intStack
   for ( int i = 0; i < 3; i++ )
   {
      intStack.push( i );
      intStack.printStack();
   } // end for

   int popInteger; // store int popped from stack

```

Fig. 20.14 | A simple stack program. (Part 1 of 5.)

```
   // pop integers from intStack
   while ( !intStack.isStackEmpty() )
   {
      intStack.pop( popInteger );
      cout << popInteger << " popped from stack" << endl;
      intStack.printStack();
   } // end while

   Stack< double > doubleStack; // create Stack of doubles
   double value = 1.1;

   cout << "processing a double Stack" << endl;

   // push floating-point values onto doubleStack
   for ( int j = 0; j < 3; j++ )
   {
      doubleStack.push( value );
      doubleStack.printStack();
      value += 1.1;
   } // end for

```

Fig. 20.14 | A simple stack program. (Part 2 of 5.)

```
   double popDouble; // store double popped from stack

   // pop floating-point values from doubleStack
   while ( !doubleStack.isStackEmpty() )
   {
      doubleStack.pop( popDouble );
      cout << popDouble << " popped from stack" << endl;
      doubleStack.printStack();
   } // end while
} // end main
```

Fig. 20.14 | A simple stack program. (Part 3 of 5.)

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1
```

Fig. 20.14 | A simple stack program. (Part 4 of 5.)

```
The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

Fig. 20.14 | A simple stack program. (Part 5 of 5.)

20.5 Stacks (cont.)

- Another way to implement a `Stack` class template is by reusing the `List` class template through composition.
- Figure 20.15 is a new implementation of the `Stack` class template that contains a `List< STACKTYPE >` object called `stackList` (line 38).
- This version of the `Stack` class template uses class `List` from Fig. 20.4.
- To test this class, use the driver program in Fig. 20.14, but include the new header file—`Stackcomposition.h` in line 6 of that file.
- The output of the program is identical for both versions of class `Stack`.

```
// Fig. 20.15: Stackcomposition.h
// Template Stack class definition with composed List object.
#ifndef STACKCOMPOSITION_H
#define STACKCOMPOSITION_H

#include "List.h" // List class definition

template< typename STACKTYPE >
class Stack
{
public:
   // no constructor; List constructor does initialization

   // push calls stackList object's insertAtFront member function
   void push( const STACKTYPE &data )
   {
      stackList.insertAtFront( data );
   } // end function push

   // pop calls stackList object's removeFromFront member function
   bool pop( STACKTYPE &data )
   {
      return stackList.removeFromFront( data );
   } // end function pop
```

Fig. 20.15 | `Stack` class template with a composed `List` object. (Part 1 of 2.)

```

   // isStackEmpty calls stackList object's isEmpty member function
   bool isStackEmpty() const
   {
      return stackList.isEmpty();
   } // end function isStackEmpty

   // printStack calls stackList object's print member function
   void printStack() const
   {
      stackList.print();
   } // end function printStack
private:
   List< STACKTYPE > stackList; // composed List object
}; // end class Stack

#endif
```

Fig. 20.15 | `Stack` class template with a composed `List` object. (Part 2 of 2.)

20.6 Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
- For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The insert and remove operations are known as enqueue and dequeue.

20.6 Queues (cont.)

- Queues have many applications in computer systems.
- Computers that have a single processor can service only one user at a time.
- Entries for the other users are placed in a queue.
- Each entry gradually advances to the front of the queue as users receive service.
- The entry at the front of the queue is the next to receive service.

20.6 Queues (cont.)

- Queues are also used to support print spooling.
- For example, a single printer might be shared by all users of a network.
- Many users can send print jobs to the printer, even when the printer is already busy.
- These print jobs are placed in a queue until the printer becomes available.
- A program called a spooler manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.

20.6 Queues (cont.)

- Information packets also wait in queues in computer networks.
- Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination.
- The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- A file server in a computer network handles file access requests from many clients throughout the network.
- Servers have a limited capacity to service requests from clients.
- When that capacity is exceeded, client requests wait in queues.

20.6 Queues (cont.)

- Queues are used in distributed systems, such as clusters, clouds, grids, and supercomputers
  - Local resource managers (e.g. Condor / LSF / SGE / PBS / Cobalt / Falkon) have queues for jobs, which are then dispatched out to remote compute resources for processing
  - Amazon (a cloud example) has the Simple Queuing Service
- Ideal to load balance across many threads/processes/nodes/clusters
  - Decouples producers from consummers

Questions

?