```
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
23 // constructor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // empty body
29 } // end ListNode constructor
```

Declare class **`List< NODETYPE >`** as a **`friend`**

Member **`data`** stores a value of type parameter **`NODETYPE`**

Member **`nextPtr`** stores a pointer to the next **`ListNode`** object in the linked list

```
30
31 // return copy of data in node
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35    return data;
36 } // end function getData
37
38 #endif
```

```
1  // Fig. 20.4: List.h
2  // Template List class definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  using std::cout;
8
9  #include "listnode.h" // ListNode class definition
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15    List(); // constructor
16    ~List(); // destructor
17    void insertAtFront( const NODETYPE & );
18    void insertAtBack( const NODETYPE & );
19    bool removeFromFront( NODETYPE & );
20    bool removeFromBack( NODETYPE & );
21    bool isEmpty() const;
22    void print() const;
23 private:
24    ListNode< NODETYPE > *firstPtr; // pointer to first node
25    ListNode< NODETYPE > *lastPtr; // pointer to last node
26
27    // utility function to allocate new node
28    ListNode< NODETYPE > *getNewNode( const NODETYPE & );
29 }; // end class List
30
```

`private` data members **`firsrtPtr`** (a pointer to the first **`ListNode`** in a **`List`**) and **`lastPtr`** (a pointer to the last **`ListNode`** in a **`List`**)

This private utility function is the key to the 'friendship'.

```
31 // default constructor
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34    : firstPtr( 0 ), lastPtr( 0 )
35 {
36    // empty body
37 } // end List constructor
38
39 // destructor
40 template< typename NODETYPE >
41 List< NODETYPE >::~List()
42 {
43    if ( !isEmpty() ) // List is not empty
44    {
45       cout << "Destroying nodes ...\n";
46
47       ListNode< NODETYPE > *currentPtr = firstPtr;
48       ListNode< NODETYPE > *tempPtr;
49
50       while ( currentPtr != 0 ) // delete remaining nodes
51       {
52          tempPtr = currentPtr;
53          cout << tempPtr->data << '\n';
54          currentPtr = currentPtr->nextPtr;
55          delete tempPtr;
56       } // end while
57    } // end if
58
59    cout << "All nodes destroyed\n\n";
60 } // end List destructor
```

Initialize both pointers to **0** (null)

Ensure that all **`ListNode`** objects in a **`List`** object are destroyed when that **`List`** object is destroyed

```
61
62 // insert node at front of list
63 template< typename NODETYPE >
64 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
65 {
66    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
67
68    if ( isEmpty() ) // List is empty
69       firstPtr = lastPtr = newPtr; // new list has only one node
70    else // List is not empty
71    {
72       newPtr->nextPtr = firstPtr; // point new node to previous 1st n
73       firstPtr = newPtr; // aim firstPtr at new node
74    } // end else
75 } // end function insertAtFront
76
77 // insert node at back of list
78 template< typename NODETYPE >
79 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
80 {
81    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
82
83    if ( isEmpty() ) // List is empty
84       firstPtr = lastPtr = newPtr; // new list has only one node
85    else // List is not empty
86    {
87       lastPtr->nextPtr = newPtr; // update previous last node
88       lastPtr = newPtr; // new last node
89    } // end else
90 } // end function insertAtBack
```

Places a new node at the front of the list

Use function **getNewNode** to allocate a new **ListNode** containing **value** and assign it to **newPtr**

If the list is empty, then both **firstPtr** and **lastPtr** are set to **newPtr**

Thread the new node into the list so that the new node points to the old first node
...nts to the new node

Places a new node at the back of the list

Use function **getNewNode** to allocate a new **listNode** containing **value** and assign it to **newPtr**

If the list is empty, then both **firstPtr** and **lastPtr** are set to **newPtr**

Thread the new node into the list so that the old last node points to the new node and **lastPtr** points to the new node

```
91
92 // delete node from front of list
93 template< typename NODETYPE >
94 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
95 {
96    if ( isEmpty() ) // List is empty
97       return false; // delete unsuccessful
98    else
99    {
100      ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete
101
102      if ( firstPtr == lastPtr )
103         firstPtr = lastPtr = 0; // no nodes remain after removal
104      else
105         firstPtr = firstPtr->nextPtr; // point to previous 2nd node
106
107      value = tempPtr->data; // return data being removed
108      delete tempPtr; // reclaim previous front node
109      return true; // delete successful
110   } // end else
111} // end function removeFromFront
112
```

Removes the front node of the list and copies the node value to the reference parameter

Return **`false`** if an attempt is made to remove a node from an empty list

Save a pointer to the first node, which will be removed

If the list has only one element, leave the list empty

Set **`firstPtr`** to point to the second node (the new first node)

Copy the removed node's **`data`** to reference parameter **`value`**

`delete` the removed node

```
113// delete node from back of list
114template< typename NODETYPE >
115bool List< NODETYPE >::removeFromBack( NODETYPE &value )
116{
117   if ( isEmpty() ) // List is empty
118      return false; // delete unsuccessful
119   else
120   {
121      ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delet
122
123      if ( firstPtr == lastPtr ) // List has one element
124         firstPtr = lastPtr = 0; // no nodes remain after removal
125      else
126      {
127         ListNode< NODETYPE > *currentPtr = firstPtr;
128
129         // locate second-to-last element
130         while ( currentPtr->nextPtr != lastPtr )
131            currentPtr = currentPtr->nextPtr; // move to nex
132
133         lastPtr = currentPtr; // remove last node
134         currentPtr->nextPtr = 0; // this is now the last node
135      } // end else
136
137      value = tempPtr->data; // return value from old last node
138      delete tempPtr; // reclaim former last node
139      return true; // delete successful
140   } // end else
```

Removes the back node of the list and copies the node value to the reference parameter

Return **`false`** if an attempt is made to remove a node from an empty list

Save a pointer to the last node, which will be removed

If the list has only one element, leave the list empty

Assign **`currentPtr`** the address of the first node to prepare to "walk the list"

"Walk the list" until **`currentPtr`** points to the node before the last node, which will be the new last node

Make the **`currentPtr`** node the new last node

Copy the removed node's **`data`** to reference parameter **`value`**

`delete` the removed node

```
141} // end function removeFromBack
142
143// is List empty?
144template< typename NODETYPE >
145bool List< NODETYPE >::isEmpty() const
146{
147   return firstPtr == 0;
148} // end function isEmpty
149
150// return pointer to newly allocated node
151template< typename NODETYPE >
152ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
153   const NODETYPE &value )
154{
155   return new ListNode< NODETYPE >( value );
156} // end function getNewNode
157
158// display contents of List
159template< typename NODETYPE >
160void List< NODETYPE >::print() const
161{
162   if ( isEmpty() ) // List is empty
163   {
164      cout << "The list is empty\n\n";
165      return;
166   } // end if
```

Determine whether the **`List`** is empty

Return a dynamically allocated **`ListNode`** object

```
167
168   ListNode< NODETYPE > *currentPtr = firstPtr;
169
170   cout << "The list is: ";
171
172   while ( currentPtr != 0 ) // get element data
173   {
174      cout << currentPtr->data << ' ';
175      currentPtr = currentPtr->nextPtr;
176   } // end while
177
178   cout << "\n\n";
179} // end function print
180
181#endif
```

Iterate through the list and output the value in each node

```
1  // Fig. 20.5: Fig20_05.cpp
2  // List class test program.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;
10
11 #include "List.h" // List class definition
12
13 // function to test a List
14 template< typename T >
15 void testList( List< T > &listObject, const string &typeName )
16 {
17    cout << "Testing a List of " << typeName << " values\n";
18    instructions(); // display instructions
19
20    int choice; // store user choice
21    T value; // store input value
22
23    do // perform user-selected actions
24    {
25       cout << "? ";
26       cin >> choice;
27
```

```
28       switch ( choice )
29       {
30          case 1: // insert at beginning
31             cout << "Enter " << typeName << ": ";
32             cin >> value;
33             listObject.insertAtFront( value );
34             listObject.print();
35             break;
36          case 2: // insert at end
37             cout << "Enter " << typeName << ": ";
38             cin >> value;
39             listObject.insertAtBack( value );
40             listObject.print();
41             break;
42          case 3: // remove from beginning
43             if ( listObject.removeFromFront( value ) )
44                cout << value << " removed from list\n";
45
46             listObject.print();
47             break;
48          case 4: // remove from end
49             if ( listObject.removeFromBack( value ) )
50                cout << value << " removed from list\n";
51
52             listObject.print();
53             break;
54       } // end switch
55    } while ( choice != 5 ); // end do...while
56
```

```
57    cout << "End list test\n\n";
58 } // end function testList
59
60 // display program instructions to user
61 void instructions()
62 {
63    cout << "Enter one of the following:\n"
64       << "  1 to insert at beginning of list\n"
65       << "  2 to insert at end of list\n"
66       << "  3 to delete from beginning of list\n"
67       << "  4 to delete from end of list\n"
68       << "  5 to end list processing\n";
69 } // end function instructions
70
71 int main()
72 {
73    // test List of int values
74    List< int > integerList;
75    testList( integerList, "integer" );
76
77    // test List of double values
78    List< double > doubleList;
79    testList( doubleList, "double" );
80    return 0;
81 } // 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

? 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
```

(continued at top of next slide...)

(...continued from bottom of previous slide)

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

? 4
4 removed from list
The list is: 3

? 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
```

(continued at top of next slide...)

(...continued from bottom of previous slide)

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

? 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

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```