

```
1  #include <iostream>
2  #include "double-linked.h"
3
4
5
6
7  ////////////////////////////////////////////
8  //          Constructor: Creates an empty list
9  ////////////////////////////////////////////
10
11 List::List(void) { first = nullptr; }
12
13
14
15
16 ////////////////////////////////////////////
17 //          Destructor
18 ////////////////////////////////////////////
19
20 // If the structure (which in C++ is equivalent as a class) has a
21 // destructor, the following can just delete the first node.
22 // That would trigger a chain of calls to destructors for every node
23 // that is linked through the pointers. This is risky though because
24 // it can initiate a very long chain of calls that can saturate the
25 // stack memory. Also, if the last node is not terminated properly
26 // (e.g. the next pointer points to itself instead of being nullptr)
27 // it causes memory leakage (the last node remains inaccessible in
28 // the heap memory.
29
30 // A for loop is not elegant but it's a simple solution to
31 // these problems.
32
33 List::~~List(void) {
34     Node * entryPoint = first;
35     int i = 0;
36     while (entryPoint) {
37         Node * temp = entryPoint;
38         entryPoint = entryPoint -> next;
39         delete temp;
40         i++;
41     }
42     first = nullptr;
43     std::cout << "Nodes eliminated: " << i << std::endl;
44 }
45
46
47
48
49
50
51
52
53
54
55
56
57
58
```

```
59
60 ///////////////////////////////////////////////////
61 //                                Insert
62 ///////////////////////////////////////////////////
63
64 // * The list must be scanned everytime because the service class only
65 // gives access to the first node... When the node with its 'next'
66 // pointer set to nullptr is found, the field 'next' is changed
67 // to the address of 'newNode'. Also the field 'prev' of the new last
68 // element must point to the previous one.
69
70 void List::insert(int n) {
71     Node * newNodePoint = new Node;
72     newNodePoint -> next = nullptr;
73     newNodePoint -> val = n;
74     // if the list is empty
75     if (!first) {
76         first = newNodePoint;
77         newNodePoint -> prev = nullptr;
78     }
79     // if it's not empty *:
80     else {
81         auto nodeP = first;
82         // The for loop is stopped at the last element 'nodeP'
83         // It doesn't have any other thing to do.
84         for ( ; nodeP -> next ; nodeP = nodeP -> next) {}
85         nodeP -> next = newNodePoint;
86         newNodePoint -> prev = nodeP;
87     }
88 }
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
```

```
117
118 ///////////////////////////////////////////////////
119 //                                Reverse
120 ///////////////////////////////////////////////////
121
122 // Also in this case, because of the limitation of the entry point,
123 // the list has to be scanned to swap the pointers to the next/previous
124 // nodes - with some additional care about the last node.
125
126 void List::reverse(void) {
127     // If the list is empty, do nothing
128     if (!first)
129         std::cout << "Empty" << std::endl;
130     else {
131         // Scan and swap
132         Node * nodePoint = first;
133         Node * temp;
134         while (nodePoint->next) {
135             temp = nodePoint->next;
136             nodePoint->next = nodePoint->prev;
137             nodePoint->prev = temp;
138             nodePoint = temp;
139         }
140
141         first = nodePoint;
142         nodePoint->next = nodePoint->prev;
143         nodePoint->prev = nullptr;
144     }
145 }
146
147
148
149 ///////////////////////////////////////////////////
150 //                                Print
151 ///////////////////////////////////////////////////
152
153 void List::print(void) {
154     // If the list is empty, do nothing
155     if (!first)
156         std::cout << "Empty" << std::endl;
157     //return;
158     for (auto nodeP = first ; nodeP ; nodeP = nodeP -> next)
159         std::cout << nodeP -> val << ' ';
160     std::cout << std::endl;
161 }
```