```cpp
#include <iostream>
#include "double-linked.h"




////////////////////////////////////////////////////////////////////
//            Constructor: Creates an empty list
////////////////////////////////////////////////////////////////////

List::List(void) { first = nullptr; }




/////////////////////////////////////////////////////////////////////
//                       Destructor
/////////////////////////////////////////////////////////////////////

// If the structure (which in C++ is equivalent as a class) has a
// destructor, the following can just delete the first node.
// That would trigger a chain of calls to destructors for every node
// that is linked through the pointers. This is risky though because
// it can initiate a very long chain of calls that can saturate the
// stack memory. Also, if the last node is not terminated properly
// (e.g. the next pointer points to itself instead of being nullptr)
// it causes memory leakage (the last node remains inaccessible in
// the heap memory.

// A for loop is not elegant but it's a simple solution to
// these problems.

List::~List(void) {
    Node * entryPoint = first;
    int i = 0;
    while (entryPoint) {
        Node * temp = entryPoint;
        entryPoint = entryPoint -> next;
        delete temp;
        i++;
    }
    first = nullptr;
    std::cout << "Nodes eliminated: " << i << std::endl;
}
```

```cpp

/////////////////////////////////////////////////////////////////////
//                            Insert
/////////////////////////////////////////////////////////////////////

// * The list must be scanned everytime because the service class only
// gives access to the first node... When the node with its 'next'
// pointer set to nullptr is found, the field 'next' is changed
// to the address of 'newNode'. Also the field 'prev' of the new last
// element must point to the previous one.

void List::insert(int n) {
    Node * newNodePoint  = new Node;
    newNodePoint -> next = nullptr;
    newNodePoint -> val  = n;
    // if the list is empty
    if (!first) {
        first = newNodePoint;
        newNodePoint -> prev = nullptr;
    }
    // if it's not empty *:
    else {
        auto nodeP = first;
        // The for loop is stopped at the last element 'nodeP'
        // It doesn't have any other thing to do.
        for ( ; nodeP -> next ; nodeP = nodeP -> next) {}
        nodeP -> next = newNodePoint;
        newNodePoint -> prev = nodeP;
    }
}
```

```cpp

///////////////////////////////////////////////////////////////////////
//                          Reverse
///////////////////////////////////////////////////////////////////////

// Also in this case, because of the limitation of the entry point,
// the list has to be scanned to swap the pointers to the next/previous
// nodes - with some additional care about the last node.

void List::reverse(void) {
    // If the list is empty, do nothing
    if (!first)
        std::cout << "Empty" << std::endl;
    else {
        // Scan and swap
        Node * nodePoint = first;
        Node * temp;
        while (nodePoint->next) {
            temp = nodePoint->next;
            nodePoint->next = nodePoint->prev;
            nodePoint->prev = temp;
            nodePoint = temp;
        }

        first = nodePoint;
        nodePoint->next = nodePoint->prev;
        nodePoint->prev = nullptr;
    }
}



/////////////////////////////////////////////////////////////////////////////
//                             Print
/////////////////////////////////////////////////////////////////////////////

void List::print(void) {
    // If the list is empty, do nothing
    if (!first)
        std::cout << "Empty" << std::endl;
        //return;
    for (auto nodeP = first ; nodeP ; nodeP = nodeP -> next)
        std::cout << nodeP -> val << ' ';
    std::cout << std::endl;
}
```