Implementing Linked Lists in C ++

```
/*
 * studentlist.h: Declaration of class StudentList - a list of students
 * maintained in order of year (1, 2, 3, 4, etc.).
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 */
#include <string>
using std::string;

class StudentList
{
    public:

        /* Constructor.  Create an empty list. */
        StudentList();

        /* Accessor.  Determine whether list is empty */
        bool isEmpty() const;

        /* Mutator.  Make the list empty */
        void makeEmpty();

        /* Mutator.  Insert a student at appropriate place. */
        void insert(string name, int year);

        /* Accessor.  Return name of first student.
         * Precondition: the list is not empty
         */
        string getFirst() const;

        /* Mutator.  Remove first student.
         * Precondition: the list is not empty
         */
        void removeFirst();

        /* Mutator.  Remove specific student. Returns true if student
         * was found and removed, false if not found.
         */
        bool remove(string name);

        /* Accessor.  Print out the items on the list to cout. */
        void print() const;

        // The following are necessary because we allocate memory for list
        // cells dynamically which must be freed appropriately

        /* Copy constructor. */
        StudentList(const StudentList & rhs);

        /* Assignment operator */
        StudentList & operator = (const StudentList & rhs);

        /* Destructor */
        ~StudentList();

    private:

        class Node;

        Node * _first;
};
```

```
/*
 * studentlist.cc: Implementation of StudentList
 *
 * This implementation uses a simple linked list
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 */
#include "studentlist.h"
#include <iostream>
using namespace std;

/* Local class Node */

class StudentList::Node
{
    public: // Since the class is private in StudentList, these public members
            // are only available to methods of class StudentList

        Node(string name, int year)
        : _name(name), _year(year), _next(NULL)
        { }

        // For demo purposes, print out a message when node is destroyed
        ~Node()
        {
            cout << "Destroying node containing " << _name
                 << " (" << _year << ")" << endl;
        }
        string _name;
        int _year;
        Node * _next;
};

StudentList::StudentList()
: _first(NULL)
{ }

bool StudentList::isEmpty() const
{
    return _first == NULL;
}

void StudentList::makeEmpty()
{
    // All of the nodes on the list must be deleted individually

    Node * p = _first, * d;
    while (p != NULL)
    {
        d = p;
        p = p -> _next;
        delete d;
    }
        _first = NULL;
}
```

```
void StudentList::insert(string name, int year)
{
    // Get a node, load it up

    Node * n = new Node(name, year);

    // Determine where it goes. p will point to the node that goes just AFTER the new
    // node, q to the node that goes just BEFORE it.  If q remains null, the new node
    // will be first on the list, so we will set the  external pointer to it.

    Node * p = _first, * q = NULL;
    while (p != NULL && p -> _year >= year)
    {
        q = p;
        p = p -> _next;
    }

    // Link it in

    n -> _next = p;
    if (q == NULL)
        _first = n;
    else
        q -> _next = n;
}

string StudentList::getFirst() const
{
    return _first -> _name;
}

void StudentList::removeFirst()
{
    Node * d = _first;
    _first = _first -> _next;
    delete d;
}

bool StudentList::remove(string name)
{
    // Find the node to be deleted.  p will point to the node, if it is found, or will
    // be NULL. q will point to the node BEFORE it.  If q remain NULL the node was at
    // the beginning of the list, so we must reset the external pointer.

    Node * p = _first, * q = NULL;
    while (p != NULL && p -> _name != name)
    {
        q = p;
        p = p -> _next;
    }

    // See if we found it

    if (p == NULL)
        return false;

    // Unlink it

    if (q == NULL)
        _first = p -> _next;
    else
        q -> _next = p -> _next;

    // Recycle it
```

```
    delete p;

    return true;
}

void StudentList::print() const
{
    Node * p = _first;
    while (p != NULL)
    {
        cout << p -> _name << " (" << p -> _year << ")" << endl;
        p = p -> _next;
    }
}

StudentList::StudentList(const StudentList & rhs)
: _first(NULL)
{
    * this = rhs;
}

StudentList & StudentList::operator = (const StudentList & rhs)
{
    // Don't copy on top of self

    if (this == & rhs)
        return * this;

    // Delete any existing contents

    makeEmpty();

    // If rhs has any nodes, copy them

    if (rhs._first != NULL)
    {
        _first = new Node(rhs._first -> _name, rhs._first -> _year);
        Node * p = _first; Node * prhs = rhs._first -> _next;
        while(prhs != NULL)
        {
            p -> _next = new Node(prhs -> _name, prhs -> _year);
            p = p -> _next;
            prhs = prhs -> _next;
        }
    }
    return * this;
}

StudentList::~StudentList()
{
    makeEmpty();
}
```