Algoritmen I LABO: Introductie van unique pointers in lijsten

1. Gegeven de klasse `Lijst`. Lijsten worden aangemaakt met nul of één knoop; langere lijsten worden verkregen door bestaande lijsten aan elkaar te plakken. Lees de code grondig, en compileer. Haal er de fout(en) uit. (Tip: let op de constructoren.)

```
#include <iostream>
using namespace std;

typedef int T;

class Knoop;

class Lijst{
    public:
        Lijst();
        Lijst(const T &);
        void append(Lijst & andereLijst);
        string to_string() const;  // handig voor testen
        ~Lijst();
    private:
        Knoop* eerste;
        Knoop* laatste;
        void schrijf(ostream & out)const;

    friend ostream & operator<<(ostream & out, const Lijst & l){
        l.schrijf(out);
        return out;
    }

};

class Knoop{
    friend class Lijst;
    private:
        T sl;
        Knoop* volgende;
    public:
        Knoop(const T & _sl):sl(_sl){};
};

Lijst::Lijst(){
    eerste = 0;
    laatste = 0;
}

Lijst::Lijst(const T & t){
    eerste = new Knoop(t);
    laatste = eerste;
}

void Lijst::append(Lijst & andereLijst){
    laatste->volgende = andereLijst.eerste;
    laatste = andereLijst.laatste;
    andereLijst.laatste = 0;
    andereLijst.eerste = 0;
}
```

```
void Lijst::schrijf(ostream & out)const{
    out<<to_string();
}

string Lijst::to_string()const{ // eigen to_string-functie
    string str="";
    Knoop* loper = eerste;
    while(loper!=0){
        str = str + std::to_string(loper->sl) + "-";  // to_string(int) is nieuw voor C++11
        loper = loper->volgende;
    }
    str += "X";
    return str;
}

void test_append(Lijst & a, Lijst & b, const string & naam_a, const string & naam_b, const string & verwachte_uitko
    cout<<endl<<"    Voor append: \n    "<<naam_a<<": "<<a<<"\n    "<<naam_b<<": "<<b<<endl;
    a.append(b);
    cout<<endl<<"    Na append: \n    "<<naam_a<<": "<<a<<"\n    "<<naam_b<<": "<<b<<endl;
    if(a.to_string() == verwachte_uitkomst_a && b.to_string() == verwachte_uitkomst_b){
        cout<<"ok"<<endl;
    }
    else{
        cout<<"niet ok"<<endl;
    }
}

void test_gelijkheid(const Lijst & a, const string & verwachte_voorstelling){
    if(a.to_string() == verwachte_voorstelling){
        cout<<"ok"<<endl;
    }
    else{
        cout<<"niet ok"<<endl;
    }
}

int main(){

    Lijst a(5);
    Lijst b(10);
    test_append(a,b,"a","b","5-10-X","X");

    Lijst x(0);
    for(int i=1; i<=5; i++){
        Lijst y(i);
        x.append(y);
    }
    test_gelijkheid(x,"0-1-2-3-4-5-X");

    return 0;
}
```

2. Nu willen we dit herwerken naar een `Lijst` die unique pointers gebruikt. Dit geeft een extra controle bij het manipuleren van knopen. Immers, stel dat we extra lidfuncties zouden schrijven die knopen van plaats verhuizen. Een veel voorkomend probleem is dat bij dit verhuizen van een knoop naar een andere plaats in dezelfde lijst of naar een andere lijst, deze knoop per ongeluk vanuit twee locaties

bereikbaar blijft. Dat willen we vermijden, of alleszins opspoorbaar maken door de knopen enkel via unique pointers aan elkaar te hangen.
Eerste opdracht: geef aan welke pointers (type `Knoop*`) je best door unique pointers vervangt, en welke niet.
Tweede opdracht: pas daarna de hele code aan, volgens de zonet voorgestelde wijzigingen.

3. De belangrijkste informatie die de klasse `Lijst` bewaart, is de pointer `eerste`. Al de rest is bijzaak. Een klasse `A` die één heel belangrijk datalid van type `X` bevat, kunnen we ook anders ontwerpen. We kunnen die klasse `A` afleiden van de klasse `X`. Waarom zouden we voor dit ontwerp kiezen in plaats van voor het behoud van het datalid?

   (a) Als we willen verbergen wat het type van het belangrijkste datalid is, dan behouden we dit best als datalid. Een voorbeeld: stel dat je een priority queue implementeert aan de hand van een rood-zwarte boom. Dan heeft de gebruiker aan die boom en zijn knopen geen boodschap; hij mag toch alleen aan het bovenste element kunnen.

   (b) Als we alle functionaliteiten/lidfuncties die de klasse X voorziet ook willen voorzien voor de klasse `A`, dan leiden we de klasse best af van `X`.

   Opdracht: laat de klasse `Lijst` overerven van `unique_ptr<Knoop>` en pas de code aan. Merk op: we leiden af van de klasse `unique_ptr<Knoop>` en volgen dus ook de interface van deze klasse. Een unique pointer kan je niet kopiëren (enkel 'verplaatsen' aan de hand van `move`), en dat is dan ook wat er met de afgeleide klasse `Lijst` zal gebeuren: vanaf nu kopieer je geen lijsten meer (ook geen deep copy), maar behandel je ze als unique pointers - dat zijn het in deze oefening ook.