LOICH KAMDOUM DEAMENI =>Mat-Nr: 506520
NDAME EKOBE HUGUE BENJAMIN => Mat-Nr: 495921

<u>Uebungsblatt 4:</u>

Übung 4: Constness

- p = &i; => ist nicht zulässig, da p ein leser Pointer ist. Durch den Bedehl "const" wird es verboten, die Adresse die in dem Pointer "p" gespeichert zu ändern.
- bar(l); => ist nicht zulässig, da die Funktion "int bar(int &)" wartet auf eine Variable mit dem Typ "int &", aber "l" has den Typ "const int". Also "l" ist als konstant Referenz definiert. Der Wert auf den die Referenz verweist kann dann (mit Hilfe der Referenz) nicht geändert werden.

Übung 3: Knobelaufgabe

The reason the conversion from "int** → const int**" is dangerous is that it would let silently and accidentally modify a "const int" object without a cast. The correct answer should be simply change "const int** to const int* const*".

Übung 1 Verkettete Liste (min/max)

<u>List.h</u>

```
#ifndef LIST_H
#define LIST_H

#include "Node.h"

class List
{
public:
   List();
   ~List();
   Node *first() const;
   Node *next(const Node *n) const;
   void append(int i);
   void insert(Node *n, int i);
   void erase(Node *n);

   const Node *findMin() const;
   const Node *findMax() const;
   void testListMinMax();

private:
   Node *_first; // saves the first node of the list
   Node *_last;  // save the current node, in order to avoid iterating throug the list all time

   //cache
   mutable Node *cachedMin; // saves the current minimum
   mutable Node *cachedMax; // saves the current maximum
};

#endif // LIST_H
```

<u>List.cpp</u>

```
#include "List.h"
#include "iostream"

List::List()
{
    this->_first = 0;
    this->_last = 0;
    this->cachedMax = 0;
    this->cachedMin = 0;
}

List::~List()
{
    while (_first)
    {
        Node *tmp = _first->next;
        delete _first;
        this->_first = tmp;
    }
    //    delete _last; // no use because of the code above, might throw a segmentation fault error,
right?
    delete cachedMax;
    delete cachedMin;
    this->_first = 0;
    this->_last = 0;
    this->cachedMax = 0;
    this->cachedMin = 0;
}

Node *List::first() const
{
    return this->_first;
}

Node *List::next(const Node *n) const
{
    return n->next;
}

void List::append(int i)
{
    Node *n = new Node(i);
    if (!this->_first) // first insert into the list
    {
        this->_first = n;
        this->_last = n;
    }
    else if (this->_first && !this->_first->next) // second insert into the list
    {
        this->_last = n;
        this->_first->next = this->_last;
    }
```

```
    else // other inserts into the list
    {
        this->_last->next = n;
        this->_last = n;
    }

    // refresh the cache
    if (this->cachedMax && this->cachedMax->value < n->value)
    {
        this->cachedMax = n;
    }
    if (this->cachedMin && this->cachedMin->value > n->value)
    {
        this->cachedMin = n;
    }
}

void List::insert(Node *n, int i)
{
    Node *current = new Node(i);
    if (n == this->_first)
    {
        current->next = this->_first;
        this->_first = current;
        return;
    }
    else
    {
        Node *tmp = this->_first;
        Node *previous = new Node();
        while (tmp != n)
        {
            previous = tmp;
            tmp = tmp->next;
        }
        previous->next = current;
        current->next = tmp;
    }

    // refresh the cache
    if (this->cachedMax && this->cachedMax->value < current->value)
    {
        this->cachedMax = current;
    }
    if (this->cachedMin && this->cachedMin->value > current->value)
    {
        this->cachedMin = current;
    }
}

void List::erase(Node *n)
{
```

```
    // reset the cache according to the following conditions
    if (this->cachedMin && n->value <= this->cachedMin->value)
    {
        this->cachedMin = 0;
    }
    if (this->cachedMax && n->value >= this->cachedMax->value)
    {
        this->cachedMax = 0;
    }

    if (n == this->_last) // last element
    {
        delete _last;
        this->_last = 0;
        return;
    }

    if (n == this->_first) // first element
    {
        this->_first = this->_first->next;
        delete n;
        n = 0;
        return;
    }

    // elements inbetween
    Node *tmp = this->_first;
    Node *previous = new Node();
    while (tmp != n)
    {
        previous = tmp;
        tmp = tmp->next;
    }
    previous->next = tmp->next;
    delete tmp;
    tmp = 0;
}

const Node *List::findMin() const
{
    if (this->cachedMin)
    {
        return this->cachedMin;
    }
    int min = this->_first->value;
    Node *node = this->_first;
    Node *it = this->_first;
    while (it)
    {
        int tmp = it->value;
        if (tmp < min)
        {
```

```
            min = tmp;
            node = it;
        }
        it = it->next;
    }
    this->cachedMin = node; // one search an then cached
    return this->cachedMin;
}

const Node *List::findMax() const
{
    if (this->cachedMax)
    {
        return this->cachedMax;
    }
    int max = this->_first->value;
    Node *node = this->_first;
    Node *it = this->_first;
    while (it)
    {
        int tmp = it->value;
        if (tmp > max)
        {
            max = tmp;
            node = it;
        }
        it = it->next;
    }
    this->cachedMax = node; // one search and then cached
    return this->cachedMax;
}

void List::testListMinMax()
{
    std::cout << '\n'
         << "Der min in der Liste ist: " << this->findMin()->value;
    std::cout << '\n'
              << "Der max in der Liste ist: " << this->findMax()->value;
    std::cout << std::endl;
}
```

<u>Node.h</u>

```
#ifndef NODE_H
#define NODE_H

class Node
{
public:
    int value;
    Node();
    Node(int);
    ~Node();
```

```
    Node *next;
    friend class List;
};

#endif // NODE_H
```

Node.cpp

```
#include "Node.h"
#include "iostream"

using namespace std;

Node::Node()
{
    value = 0;
    next = nullptr;
}

Node::Node(int value_)
{
    value = value_;
    next = nullptr;
}

Node::~Node()
{
    value = 0;
    next = nullptr;
}
```

Main.cpp

```
#include <iostream>
#include <stdio.h>

#include "List.h"
#include "Node.h"

using namespace std;

int main ();

int main ()
{
    // cout << "lol"<<'\n';
    List list;
    list.append(2);
    list.append(3);
    list.insert(list.first(), 1);
    for (Node *n = list.first(); n != 0; n = list.next(n))
        std::cout << n->value << std::endl;

    cout <<'\n'<<"Nach der Kopie"<<'\n';
```

```
    // Kopie von der Liste
    List list2 = list;
    for (Node *n = list2.first(); n != 0; n = list2.next(n))
        std::cout << n->value << std::endl;

    //Neue Elemente werden in der Liste hinzugefügt zur Prüfung der Methode testMinMax()
    cout <<'\n'<<"Prüfung der Methode testMinMax()"<<'\n';
    list.append(10);
    list.testListMinMax();

    return 0;
}
```

Übung 2:shared_ptr und weak_ptr

Node.h

```
#ifndef NODE_H
#define NODE_H

#include <memory>

class Node
{
public:
    int value;
    Node();
    Node(int);
    ~Node();
    shared_ptr<Node> next;
    friend class List;
};

#endif // NODE_H
```

Node.cpp

```
#include "Node.h"
#include "iostream"
#include <memory>

using namespace std;

Node::Node()
{
    value = 0;
    next = nullptr;
}

Node::Node(int value_)
{
    value = value_;
    next = nullptr;
```

```
}

Node::~Node()
{
   value = 0;
   next = nullptr;
}
```

List.h

```
#ifndef LIST_H
#define LIST_H

#include "Node.h"
#include <memory>

class List
{
public:
   List();
   ~List();
   shared_ptr<Node> first() const;
   shared_ptr<Node> next(const shared_ptr<Node> n) const;
   void append(int i);
   void insert(shared_ptr<Node> n, int i);
   void erase(shared_ptr<Node> n);

   // with the cache, the user have to call the following methods once. But if the Node
   // with the extremum is delete then he calls them once again
   const shared_ptr<Node> findMin() const;
   const shared_ptr<Node> findMax() const;
   void testListMinMax();

private:
   shared_ptr<Node> _first; // saves the first node of the list
   shared_ptr<Node> _last;  // save the current node, in order to avoid iterating throug the list all
time
private:
   //cache
   mutable shared_ptr<Node> cachedMin; // saves the current minimum
   mutable shared_ptr<Node> cachedMax; // saves the current maximum
};

#endif // LIST_H
```

List.cpp

```
#include "List.h"
#include "iostream"
#include <memory>

List::List()
{
   this->_first = 0;
```

```
    this->_last = 0;
    this->cachedMax = 0;
    this->cachedMin = 0;
}

List::~List()
{
    while (_first)
    {
        shared_ptr<Node> tmp = _first->next;
        delete _first;
        this->_first = tmp;
    }
    //    delete _last; // no use because of the code above, might throw a segmentation fault error,
right?
    delete cachedMax;
    delete cachedMin;
    this->_first = 0;
    this->_last = 0;
    this->cachedMax = 0;
    this->cachedMin = 0;
}

shared_ptr<Node> List::first() const
{
    return this->_first;
}

shared_ptr<Node> List::next(const shared_ptr<Node> n) const
{
    return n->next;
}

void List::append(int i)
{
    shared_ptr<Node> n = new Node(i);
    if (!this->_first) // first insert into the list
    {
        this->_first = n;
        this->_last = n;
    }
    else if (this->_first && !this->_first->next) // second insert into the list
    {
        this->_last = n;
        this->_first->next = this->_last;
    }
    else // other inserts into the list
    {
        this->_last->next = n;
        this->_last = n;
    }
```

```
    // refresh the cache
    if (this->cachedMax && this->cachedMax->value < n->value)
    {
        this->cachedMax = n;
    }
    if (this->cachedMin && this->cachedMin->value > n->value)
    {
        this->cachedMin = n;
    }
}

void List::insert(shared_ptr<Node> n, int i)
{
    shared_ptr<Node> current = new Node(i);
    if (n == this->_first)
    {
        current->next = this->_first;
        this->_first = current;
        return;
    }
    else
    {
        shared_ptr<Node> tmp = this->_first;
        shared_ptr<Node> previous = new Node();
        while (tmp != n)
        {
            previous = tmp;
            tmp = tmp->next;
        }
        previous->next = current;
        current->next = tmp;
    }

    // refresh the cache
    if (this->cachedMax && this->cachedMax->value < current->value)
    {
        this->cachedMax = current;
    }
    if (this->cachedMin && this->cachedMin->value > current->value)
    {
        this->cachedMin = current;
    }
}

void List::erase(shared_ptr<Node> n) // remove n from the list
{
    // reset the cache according to the following conditions
    if (this->cachedMin && n->value <= this->cachedMin->value)
    {
        this->cachedMin = 0;
    }
    if (this->cachedMax && n->value >= this->cachedMax->value)
```

```
    {
        this->cachedMax = 0;
    }

    if (n == this->_last) // last element
    {
        delete _last;
        this->_last = 0;
        return;
    }

    if (n == this->_first) // first element
    {
        this->_first = this->_first->next;
        delete n;
        n = 0;
        return;
    }

    // elements inbetween
    shared_ptr<Node> tmp = this->_first;
    shared_ptr<Node> previous = new Node();
    while (tmp != n)
    {
        previous = tmp;
        tmp = tmp->next;
    }
    previous->next = tmp->next;
    delete tmp;
    tmp = 0;
}

const shared_ptr<Node> List::findMin() const
{
    if (this->cachedMin)
    {
        return this->cachedMin;
    }
    int min = this->_first->value;
    shared_ptr<Node> node = this->_first;
    shared_ptr<Node> it = this->_first;
    while (it)
    {
        int tmp = it->value;
        if (tmp < min)
        {
            min = tmp;
            node = it;
        }
        it = it->next;
    }
    this->cachedMin = node; // one search an then cached
```

```
    return this->cachedMin;
}

const shared_ptr<Node> List::findMax() const
{
    if (this->cachedMax)
    {
        return this->cachedMax;
    }
    int max = this->_first->value;
    shared_ptr<Node> node = this->_first;
    shared_ptr<Node> it = this->_first;
    while (it)
    {
        int tmp = it->value;
        if (tmp > max)
        {
            max = tmp;
            node = it;
        }
        it = it->next;
    }
    this->cachedMax = node; // one search and then cached
    return this->cachedMax;
}

void List::testListMinMax()
{
    std::cout << '\n'
         << "Der min in der Liste ist: " << this->findMin()->value;
    std::cout << '\n'
             << "Der max in der Liste ist: " << this->findMax()->value;
    std::cout << std::endl;
}
```

Main.cpp

Diese Funktion ist die gleiche wie in der Übung 1.

2)

- Der virtuelle Speicher der Smart Zeiger wird freigegeben sobald der letzte shared_ptr verschwindet.
- Der Destruktor dieser Klasse wird zunächst aufgerufen

3)

- Es entstehen zyklen und deswegen werden die shared_ptr Objekte nicht gelöscht, da der doppelt genutzt wird.
- weak_ptr<Node> kann das Problem lösen, in dem man im Destruktor die Funktion lock um den Zeiger zu übergeben und danach reset, um den Speicher freizugeben