

Dynamische Datentypen

Listen; Funktionalität, Copy-Konstruktor, Zuweisungsoperator, Destruktor, Konzept Dynamischer Datentyp

Container für eine Folge gleichartiger Daten

- Bisher: Felder (`T[]`, `std::array<T>`, `std::vector<T>`)
- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 2 | 9 | 7 | 7 | 0 |
|---|---|---|---|---|---|---|---|

Probleme mit Feldern... ...aber nicht mit `std::vector`

- Man kann sie nicht direkt kopieren und zuweisen
- Falls Länge nicht zur Kompilierungszeit bekannt, muss Speicher explizit mit `new` geholt und mit `delete[]` wieder freigegeben werden (fehleranfällig)
- Sie können nicht wachsen/schrumpfen

Probleme mit Feldern... ... **und** auch `std::vector`

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 2 | 9 | 7 | 7 | 0 |
|---|---|---|---|---|---|---|---|

↑
8

Wollen wir hier einfügen, müssen wir alles rechts davon explizit verschieben

Probleme mit Feldern... ... **und** auch `std::vector`

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | | |
|---|---|---|---|--|---|---|---|---|
| 1 | 5 | 6 | 2 | | 9 | 7 | 7 | 0 |
|---|---|---|---|--|---|---|---|---|

↑
8

Wollen wir hier einfügen, müssen wir alles rechts davon explizit verschieben

Probleme mit Feldern... ... **und** auch `std::vector`

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 2 | 8 | 9 | 7 | 7 | 0 |
|---|---|---|---|---|---|---|---|---|

Wollen wir hier einfügen, müssen wir alles rechts davon explizit verschieben

Probleme mit Feldern... ... und auch std::vector

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 2 | 8 | 9 | 7 | 7 | 0 |
|---|---|---|---|---|---|---|---|---|

↑

Wollen wir hier löschen, müssen wir alles rechts davon explizit verschieben

Probleme mit Feldern... ... und auch std::vector

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | | |
|---|---|---|--|---|---|---|---|---|
| 1 | 5 | 6 | | 8 | 9 | 7 | 7 | 0 |
|---|---|---|--|---|---|---|---|---|

Wollen wir hier löschen, müssen wir alles rechts davon explizit verschieben

Probleme mit Feldern... ... und auch std::vector

- Man kann keine Elemente "in der Mitte" einfügen oder "in der Mitte" löschen

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 6 | 8 | 9 | 7 | 7 | 0 |
|---|---|---|---|---|---|---|---|

Wollen wir hier löschen, müssen wir alles rechts davon explizit verschieben

Lösung: Listen

- Container für eine Folge von Daten gleichen Typs
- Kein** zusammenhängender Speicherbereich, **kein** wahlfreier Zugriff

Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```

class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
    ...
private:
    Node* head_;
    ...
};

class Node {
...
private:
    int key_;
    Node* next_;
};
    
```

Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```

class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
    ...
private:
    Node* head_;
    ...
};

class Node {
...
private:
    int key_;
    Node* next_;
};
    
```

Eine Klasse für Listen: Daten-Mitglieder, Default-Konstruktor

```

class List {
public:
    // default constructor:
    // POST: *this is an empty list
    List();
private:
    Node* head_;
};

List::List()
: head_(0)
{}
    
```

MyList.h

MyList.cpp

Eine Klasse für Listen: Vorne einfügen und Ausgabe

- Gewünschter Anwendungscode:

```

#include "MyList.h"
#include<iostream>

int main() {
    List l1;
    l1.push_front(1); // l1 == 1
    l1.push_front(3); // l1 == 3 1
    l1.push_front(2); // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1
}
    
```

Eine Klasse für Listen: Vorne einfügen und Ausgabe

- Vorne einfügen:

```

// POST: key was added before first
// element of *this
void push_front(int key);

void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
    
```

MyList.h

MyList.cpp

Eine Klasse für Listen: Vorne einfügen und Ausgabe

- Vorne einfügen:

```

// POST: key was added before first
// element of *this
void push_front(int key);

void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
    
```

MyList.h

MyList.cpp

Eine Klasse für Listen: Vorne einfügen und Ausgabe

- Vorne einfügen:

```

// POST: key was added before first
// element of *this
void push_front(int key);

void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
    
```

MyList.h

MyList.cpp

Eine Klasse für Listen: Vorne einfügen und Ausgabe

- Vorne einfügen:

```

// Constructor
Node::Node (int key, Node* next)
: key_(key), next_(next)
{}

void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
    
```

MyNode.cpp

MyList.cpp

Eine Klasse für Listen: Vorne einfügen und Ausgabe

■ Ausgabe: MyList.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
  const Node* p = l.get_head();
  while (p != 0) {
    o << p->get_key() << " ";
    p = p->get_next();
  }
  return o;
}
```

Eine Klasse für Listen: Vorne einfügen und Ausgabe

■ Ausgabe: MyList.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
  const Node* p = l.get_head();
  while (p != 0) {
    o << p->get_key() << " ";
    p = p->get_next();
  }
  return o;
}
```

Eine Klasse für Listen: Vorne einfügen und Ausgabe

■ Ausgabe: MyList.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
  const Node* p = l.get_head();
  while (p != 0) {
    o << p->get_key() << " ";
    p = p->get_next();
  }
  return o;
}
```

Abkürzung für (*p).get_key()

Eine Klasse für Listen: Vorne einfügen und Ausgabe

■ Ausgabe: MyList.cpp

```
// POST: *this is written to std::cout
std::ostream& operator<< (std::ostream& o, const List& l) {
  const Node* p = l.get_head();
  while (p != 0) {
    o << p->get_key() << " ";
    p = p->get_next();
  }
  return o;
}
```

Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last element of *this
void push_back (int key); MyList.h

void List::push_back (int key) MyList.cpp
{
  if (head_ == 0)
    head_ = new Node (key);
  else {
    Node* p = head_;
    while (p->get_next() != 0)
      p = p->get_next();
    p->set_next (new Node (key));
  }
}
```

Eine Klasse für Listen: Hinten einfügen

```
// POST: key was added after last element of *this
void push_back (int key); MyList.h

void List::push_back (int key) MyList.cpp
{
  if (head_ == 0)
    head_ = new Node (key);
  else {
    Node* p = head_;
    while (p->get_next() != 0)
      p = p->get_next();
    p->set_next (new Node (key));
  }
}
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

MyNode.h // Constructor
Node(int key, Node* next = 0);

Zweites Argument ist 0 per Default

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Hinten einfügen

```

// POST: key was added after last
// element of *this
void push_back (int key);

void List::push_back (int key)
{
    if (head_ == 0)
        head_ = new Node (key);
    else {
        Node* p = head_;
        while (p->get_next() != 0)
            p = p->get_next();
        p->set_next (new Node (key));
    }
}
    
```

Eine Klasse für Listen: Der Copy-Konstruktor

- Was passiert hier?

```

int main() {
    List l1;
    l1.push_front (1); // l1 == 1
    l1.push_front (3); // l1 == 3 1
    l1.push_front (2); // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1

    List l2 = l1;
    std::cout << l2 << std::endl;
    l1.push_back (5);
    l1.push_back (6);
    std::cout << l2 << std::endl;
}
    
```

Eine Klasse für Listen: Der Copy-Konstruktor

■ Was passiert hier?

```
int main() {
    List l1;
    l1.push_front(1);           // l1 == 1
    l1.push_front(3);           // l1 == 3 1
    l1.push_front(2);           // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1

    List l2 = l1;                // Kopiere l1 nach l2
    std::cout << l2 << std::endl;
    l1.push_back(5);
    l1.push_back(6);
    std::cout << l2 << std::endl;
}
```

Eine Klasse für Listen: Der Copy-Konstruktor

■ Was passiert hier?

```
int main() {
    List l1;
    l1.push_front(1);           // l1 == 1
    l1.push_front(3);           // l1 == 3 1
    l1.push_front(2);           // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1

    List l2 = l1;                // Kopiere l1 nach l2
    std::cout << l2 << std::endl; // 2 3 1
    l1.push_back(5);
    l1.push_back(6);
    std::cout << l2 << std::endl;
}
```

Eine Klasse für Listen: Der Copy-Konstruktor

■ Was passiert hier?

```
int main() {
    List l1;
    l1.push_front(1);           // l1 == 1
    l1.push_front(3);           // l1 == 3 1
    l1.push_front(2);           // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1

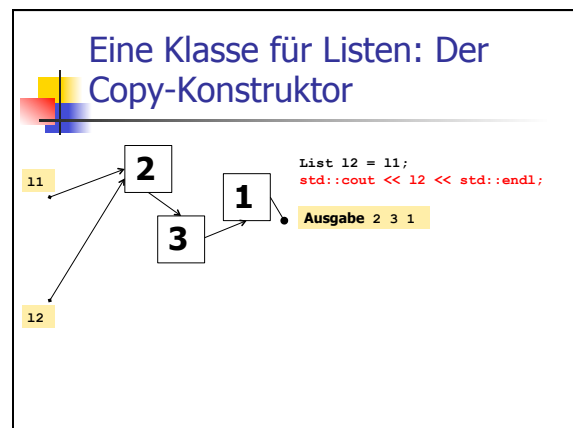
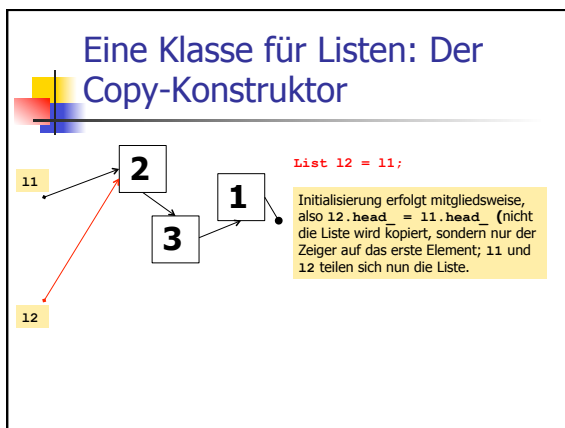
    List l2 = l1;                // Kopiere l1 nach l2
    std::cout << l2 << std::endl; // 2 3 1
    l1.push_back(5);            // l1 == 2 3 1 5
    l1.push_back(6);            // l1 == 2 3 1 5 6
    std::cout << l2 << std::endl;
}
```

Eine Klasse für Listen: Der Copy-Konstruktor

■ Was passiert hier? l1 wurde verändert, aber die gleichen Änderungen passieren auch mit l2. Warum?

```
int main() {
    List l1;
    l1.push_front(1);           // l1 == 1
    l1.push_front(3);           // l1 == 3 1
    l1.push_front(2);           // l1 == 2 3 1
    std::cout << l1 << std::endl; // 2 3 1

    List l2 = l1;                // Kopiere l1 nach l2
    std::cout << l2 << std::endl; // 2 3 1
    l1.push_back(5);            // l1 == 2 3 1 5
    l1.push_back(6);            // l1 == 2 3 1 5 6
    std::cout << l2 << std::endl; // 2 3 1 5 6
}
```



Eine Klasse für Listen: Der Copy-Konstruktor

```

List l2 = l1;
std::cout << l2 << std::endl;

l1.push_back(5);
l1.push_back(6);
    
```

Eine Klasse für Listen: Der Copy-Konstruktor

```

List l2 = l1;
std::cout << l2 << std::endl;

l1.push_back(5);
l1.push_back(6);
std::cout << l2 << std::endl;
    
```

Ausgabe 2 3 1 5 6

Eine Klasse für Listen: Der Copy-Konstruktor

- Wir wollen eine „echte“ Kopie, wenn wir `List l2 = l1` schreiben!

Eine Klasse für Listen: Der Copy-Konstruktor

- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration


```
T(const T& x);
```
- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T* initialisiert werden
 - `T x = t; // t vom Typ T`

Eine Klasse für Listen: Der Copy-Konstruktor

- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration


```
T(const T& x);
```
- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T* initialisiert werden
 - `T x (t); // t vom Typ T`

Eine Klasse für Listen: Der Copy-Konstruktor

- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration


```
T(const T& x);
```
- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T* initialisiert werden
 - Initialisierung von formalen Funktions-Argumenten und Rückgabewerten

Eine Klasse für Listen: Der Copy-Konstruktor

```

// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}

public: MyList.h
    List (const List& l);
...
private:
    // PRE: *this is empty
    // POST: list starting at from was copied to *this
    void copy (const Node* from);
    
```

MyList.cpp

Warum ist **const List& l** hier zwingend notwendig, während **List l** nicht geht?

Eine Klasse für Listen: Der Copy-Konstruktor

```

// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}

public: MyList.h
    List (const List& l);
...
private:
    // PRE: *this is empty
    // POST: list starting at from was copied to *this
    void copy (const Node* from);
    
```

MyList.cpp

Aufruf eines Copy-Konstruktors mit Deklaration **List (List l)**; müsste zuerst den Copy-Konstruktor aufrufen (Initialisierung des formalen Arguments!); unendliche Rekursion!

Eine Klasse für Listen: Der Copy-Konstruktor

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert mitgliedswise)

Das ist für unsere Klasse **List** genau nicht das, was wir wollen (es wird nur der Zeiger **head_** kopiert, *nicht* jedoch die dahinterliegende Liste – wir erhalten Alias-Semantik!)

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

MyList.cpp

```

// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
    
```

MyList.h

***this.head** →

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

MyList.cpp

```

// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
    
```

MyList.h

***this.head** →

from → **2**

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

MyList.cpp

```

// copy constructor
List::List (const List& l)
: head_ (0)
{
    copy (l.head_);
}
    
```

MyList.h

***this.head** →

from → **2**

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

```

// copy constructor
List::List (const List& l)
: head_(0)
{
    copy (l.head_);
}
    
```

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

```

// copy constructor
List::List (const List& l)
: head_(0)
{
    copy (l.head_);
}
    
```

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

```

// copy constructor
List::List (const List& l)
: head_(0)
{
    copy (l.head_);
}
    
```

Eine Klasse für Listen: Die (private) Copy-Funktion

```

// private copy method
void List::copy (const Node* from)
{
    assert (head_ == 0);
    if (from != 0) {
        // copy first element to head
        head_ = new Node (from->get_key());
        Node* p = from->get_next();
        Node* q = head_;
        // copy remaining elements
        while (p != 0) {
            q->set_next (new Node (p->get_key()));
            p = p->get_next();
            q = q->get_next();
        }
    }
}
    
```

```

// copy constructor
List::List (const List& l)
: head_(0)
{
    copy (l.head_);
}
    
```

und so weiter...

Eine Klasse für Listen: Echtes Kopieren

```

List l1;
l1.push_front (1);
l1.push_front (3);
l1.push_front (2);
std::cout << l1 << std::endl; // 2 3 1

List l2 = l1; // Aufruf „unseres“ Copy-Konstruktors
std::cout << l2 << std::endl; // 2 3 1

l1.push_back (5);
l1.push_back (6);
std::cout << l1 << std::endl; // 2 3 1 5 6
std::cout << l2 << std::endl; // 2 3 1
    
```

Eine Klasse für Listen: Der Zuweisungsoperator

```

List l3;
l3 = l2;
    
```

- Überladung von `operator=` als Mitglieds-Funktion
- ähnlich wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
 - Freigabe des Speichers für den "alten" Wert
 - Prüfen auf Selbstzuweisungen (`l3 = l3`), die keinen Effekt haben sollten

Eine Klasse für Listen: Der Zuweisungsoperator

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) { Prüfen auf Selbstzuweisungen
        clear();
        copy (l.head_);
    }
    return *this;
}
```

Eine Klasse für Listen: Der Zuweisungsoperator

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) { Freigabe des Speichers für den
        clear(); "alten" Wert
        copy (l.head_);
    }
    return *this;
}
```

Eine Klasse für Listen: Der Zuweisungsoperator

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_); Das eigentliche Kopieren
    }
    return *this;
}
```

Eine Klasse für Listen: Der Zuweisungsoperator

```
List& List::operator= (const List& l)
{
    if (head_ != l.head_) {
        clear();
        copy (l.head_);
    }
    return *this; Konvention: Zuweisungsoperator gibt
} den neuen Wert als L-Wert zurück.
```

Eine Klasse für Listen: Der Zuweisungsoperator

- o Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist mitgliedswise zu)

Das ist für unsere Klasse `List` wiederum nicht das, was wir wollen (es wird nur der Zeiger `head_` kopiert, *nicht* jedoch die dahinterliegende Liste – wir erhalten Alias-Semantik)!

Eine Klasse für Listen: Der Zuweisungsoperator

```
...
std::cout << l1 << std::endl; // 2 3 1 5 6
std::cout << l2 << std::endl; // 2 3 1

List l3;
l3 = l1; Aufruf „unseres“ Zuweisungsoperators
std::cout << l3 << std::endl; // 2 3 1 5 6
```

Eine Klasse für Listen: Der Destruktor

- Problem: Quelltext enthält jede Menge `new`'s, aber keine `delete`'s, das führt zu Speicherlecks
- Die `delete`'s sollen passieren, wenn wir die Liste nicht mehr brauchen.

```
void List::push_front (int key)
{
    head_ = new Node (key, head_);
}
```

Eine Klasse für Listen: Der Destruktor

- Problem: Quelltext enthält jede Menge `new`'s, aber keine `delete`'s, das führt zu Speicherlecks
- Die `delete`'s sollen passieren, wenn wir die Liste nicht mehr brauchen....
- Nämlich genau dann, wenn ihre automatische Speicherdauer endet!

Eine Klasse für Listen: Der Destruktor

- Nämlich genau dann, wenn ihre automatische Speicherdauer endet!

```
int main() {
    List l1;
    l1.push_front (1);
    l1.push_front (3);
    l1.push_front (2);
    std::cout << l1 << std::endl; // 2 3 1
    return 0;
}
```

Ziel genau wie bei fundamentalen Typen: Ende des Gültigkeitsbereichs, der Speicher wird wieder freigegeben.

Eine Klasse für Listen: Der Destruktor

- Der *Destruktor* ist eine spezielle Mitgliedsfunktion, die **automatisch** aufgerufen wird, wenn die Speicherdauer eines Klassenobjekts endet.
- Wird kein Destruktor deklariert, so wird er automatisch erzeugt und ruft die Destruktoren für die Datenmitglieder auf (hier: Zeiger `head_`, kein Effekt)

Eine Klasse für Listen: Der Destruktor

```
List::~List()
{
    clear();
}
```

Destruktor: Name wie die Klasse, mit `~` davor, und stets ohne Argumente

Eine Klasse für Listen: Der Destruktor

```
List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Eine Klasse für Listen: Der Destruktor

```

List::~List()
{
    clear();
}
void List::clear ()
{
    Node* p = head_;
    while (p != 0) {
        p = p->get_next();
        delete head_;
        head_ = p;
    }
    assert (head_ == 0);
}
    
```

Entfernt alle Elemente aus der Liste und gibt ihren Speicher frei.

Dynamischer Datentyp

- o Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Listen)
- o andere typische Anwendungen:
 - o Stapel
 - o Bäume
 - o Graphen

Dynamischer Datentyp

- o sollte immer mindestens
 - o Konstruktoren
 - o Destruktor
 - o Copy-Konstruktor
 - o Zuweisungsoperator
 haben.

Listen: Elemente entfernen

- Zur Erinnerung: damit haben wir Listen motiviert!
- Gewünschte Funktionalität:

```

// POST: the first occurrence of key was removed from *this
//       if *this does not contain key, nothing happened
void remove (int key);
    
```

Listen: Elemente entfernen

```

void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_->get_next();
            delete p;
        } else {
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else {
                    p = q;
                }
            }
        }
    }
}
    
```

Fall 1: erstes Element wird entfernt (key == 2).

Listen: Elemente entfernen

```

void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_->get_next();
            delete p;
        } else {
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                } else {
                    p = q;
                }
            }
        }
    }
}
    
```

Fall 1: erstes Element wird entfernt (key == 2).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 1: erstes Element wird entfernt (key == 2).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 2: anderes Element wird entfernt (key == 3).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 2: anderes Element wird entfernt (key == 3).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 2: anderes Element wird entfernt (key == 3).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 2: anderes Element wird entfernt (key == 3).

Listen: Elemente entfernen

```
void List::remove (int key)
{
    if (head_ != 0) {
        Node* p = head_;
        if (p->get_key() == key) {
            head_ = head_>get_next();
            delete p;
        }
        else
            while (p->get_next() != 0) {
                Node* q = p->get_next();
                if (q->get_key() == key) {
                    p->set_next(q->get_next());
                    delete q;
                    break;
                }
                else
                    p = q;
            }
    }
}
```

Fall 2: anderes Element wird entfernt (key == 3).

Fertig!