

17. Klassen

Klassen, Memberfunktionen, Konstruktoren, Stapel, verkettete Liste, dynamischer Speicher, Copy-Konstruktor, Zuweisungsoperator, Destruktor, Konzept Dynamischer Datentyp

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n; // error: n is private
```

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n; // error: n is private
```

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen ...

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n;  // error: n is private
```

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen ...

Anwendungscode:

... und wir auch nicht
(kein `operator+`, ...)

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n; // error: n is private
```

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of *this  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of *this  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

Memberfunktionen: Deklaration

```
class rational {
public:
    // POST: return value is the numerator of *this
    int numerator () const {
        return n;
    }
    // POST: return value is the denominator of *this
    int denominator () const {
        return d;
    }
private:
    int n;
    int d; // INV: d!= 0
};
```

öffentlicher Bereich

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of *this  
    int numerator () const { Memberfunktion  
        return n;  
    }  
    // POST: return value is the denominator of *this  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of *this  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of *this  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich


Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Memberfunktionen: Aufruf

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r; Member-Zugriff

int n = r.numerator(); // Zaehler
int d = r.denominator(); // Nenner
```



Memberfunktionen: Definition

```
// POST: returns numerator of *this
int numerator () const
{
    return n;
}
```

Memberfunktionen: Definition ???

```
// POST: returns numerator of *this
int numerator () const
{
    return n;
}
```

Memberfunktionen: Definition


```
// POST: returns numerator of *this
int numerator () const
{
    return n;                r.numerator()
}
```

- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen.

Memberfunktionen: Definition

```
// POST: returns numerator of *this  
int numerator () const  
{  
    return n;  
}
```

r.numerator()



- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen. In der Funktion: **this* ist der Name dieses impliziten Arguments.

Memberfunktionen: Definition

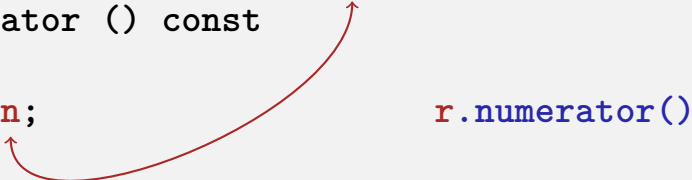
```
// POST: returns numerator of *this
int numerator () const
{
    return n;                r.numerator()
}
```

- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen. In der Funktion: `*this` ist der Name dieses impliziten Arguments.
- Das `const` bezieht sich auf `*this`

Memberfunktionen: Definition

```
// POST: returns numerator of *this
int numerator () const
{
    return n;
}
```

`r.numerator()`

A red arrow originates from the variable 'n' in the return statement 'return n;' and points to the 'r.numerator()' expression. Another red arrow originates from the 'r.numerator()' expression and points to the 'numerator of *this' text in the comment above the function signature.

- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen. In der Funktion: `*this` ist der Name dieses impliziten Arguments.
- Das `const` bezieht sich auf `*this`
- `n` ist Abkürzung für `(*this).n`

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return .n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

... ohne Memberfunktionen

```
struct bruch {
    int n;
    ...
};

int numerator (const bruch* dieser)
{
    return (*dieser).n;
}

bruch r;
..
std::cout << numerator(&r);
```

Member-Definition: In-Class

```
class rational {  
    int n;  
    ...  
  
    int numerator () const  
    {  
        return n;  
    }  
    ....  
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

Member-Definition: In-Class vs. Out-of-Class

```
class rational {  
    int n;  
    ...  
  
    int numerator () const  
    {  
        return n;  
    }  
    ....  
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

```
class rational {  
    int n;  
    ...  
  
    int numerator () const;  
    ...  
};  
  
int rational::numerator () const  
{  
    return n;  
}
```

- So geht's auch.

Initialisierung? Konstruktoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den)
    {
        assert (den != 0);
    }
    ...
};
...
rational r (2,3); // r = 2/3
```

Initialisierung? Konstruktoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den) ← Initialisierung der
                               Membervariablen
    {
        assert (den != 0); ← Funktionsrumpf.
    }
    ...
};
...
rational r (2,3); // r = 2/3
```


Initialisierung “rational = int”?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {}

    ...
};

...
rational r (2);    // Explizite Initialisierung mit 2
rational s = 2;   // Implizite Konversion
```

Initialisierung “rational = int”?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {} ← Leerer Funktionsrumpf
    ...
};
...
rational r (2);    // Explizite Initialisierung mit 2
rational s = 2;   // Implizite Konversion
```

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}

    ...
};

...
rational r;    // r = 0
```

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r;    // r = 0
```

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

```
// POST: double approximation of r
double to_double (const rational r)
{
    double result = r.numerator();
    return result / r.denominator();
}
```

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

```
// POST: double approximation of r
double to_double (const rational r)
{
    double result = r.numerator();
    return result / r.denominator();
}
```

- Wir können die Memberfunktionen zusammen mit der Repräsentation anpassen. ✓

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

```
int numerator () const  
{  
    return n;  
}
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};  
  
int numerator () const  
{  
    return n;  
}
```

nachher

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

```
int numerator () const  
{  
    return n;  
}
```

nachher

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const {  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher
- Dazu noch möglicher Überlauf

Datenkapselung noch unvollständig

Die Sicht des Kunden (rational.h):

```
class rational {  
public:  
    // POST: returns numerator of *this  
    int numerator () const;  
    ...  
private:  
    // none of my business  
};
```

Datenkapselung noch unvollständig

Die Sicht des Kunden (rational.h):

```
class rational {  
public:  
    // POST: returns numerator of *this  
    int numerator () const;  
    ...  
private:  
    // none of my business  
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.

Datenkapselung noch unvollständig

Die Sicht des Kunden (rational.h):

```
class rational {
public:
    // POST: returns numerator of *this
    int numerator () const;
    ...
private:
    // none of my business
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.
- Lösung: Nicht nur Daten, auch **Typen** kapseln (Handout).

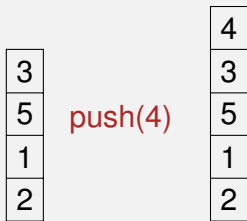
Motivation: Stapel



Motivation: Stapel

3
5
1
2

Motivation: Stapel



Motivation: Stapel

3
5
1
2

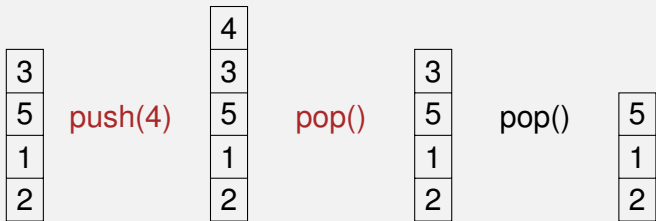
push(4)

4
3
5
1
2

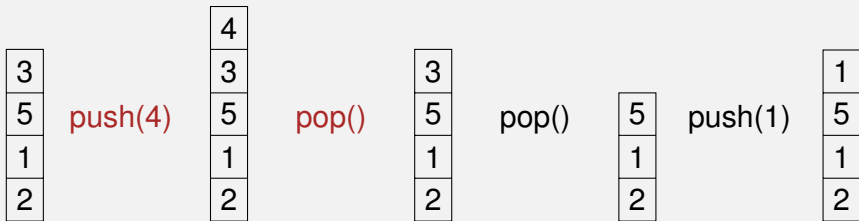
pop()

3
5
1
2

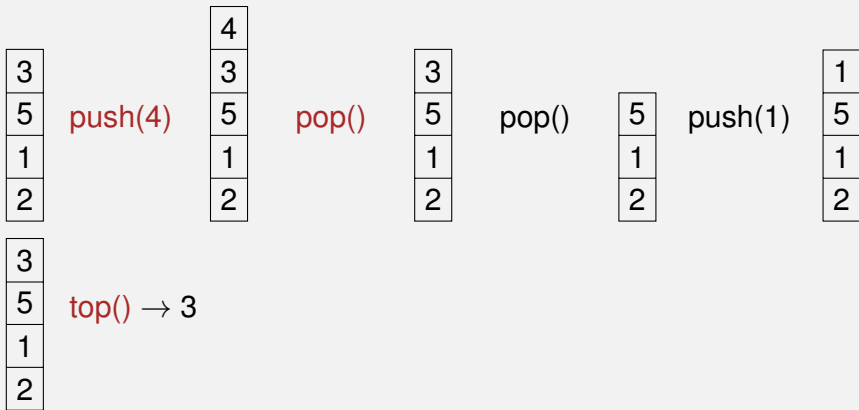
Motivation: Stapel



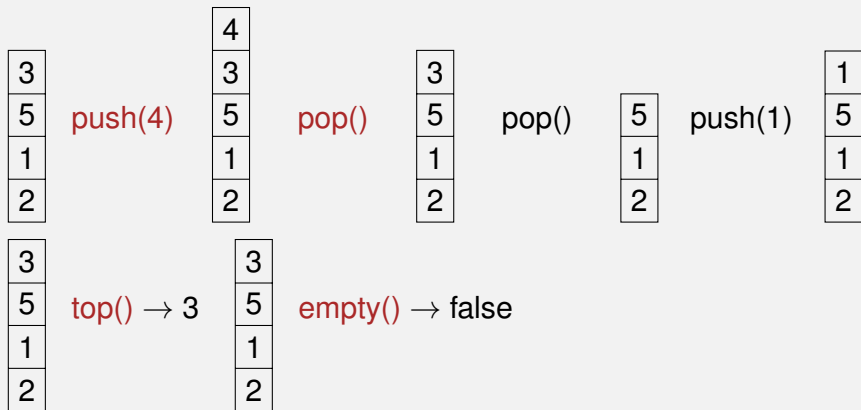
Motivation: Stapel



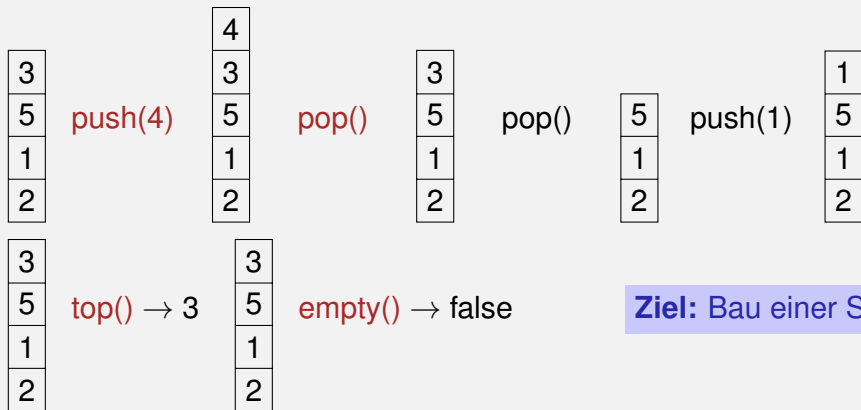
Motivation: Stapel



Motivation: Stapel (push, pop, top, empty)

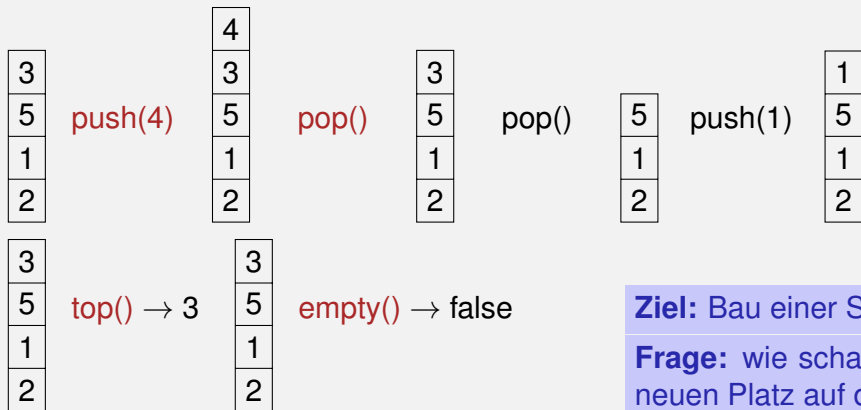


Motivation: Stapel (push, pop, top, empty)



Ziel: Bau einer Stapel-Klasse!

Motivation: Stapel (push, pop, top, empty)



Ziel: Bau einer Stapel-Klasse!

Frage: wie schaffen wir bei push neuen Platz auf dem Stapel?

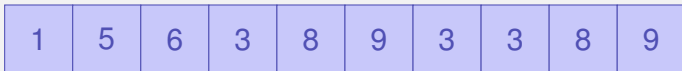
Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: `Array (T [])`

Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array ($T[]$)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i -tes Element)



Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array ($T[]$)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i -tes Element)
- Simulation eines Stapels durch ein Array?



Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array ($T[]$)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i -tes Element)
- Simulation eines Stapels durch ein Array?
- Nein, irgendwann ist das Array “voll.”



Hier kein `push(3)` möglich!

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

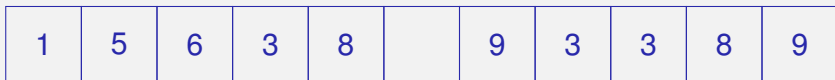
1	5	6	3	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---

↑
8

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



↑
8

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

1	5	6	3	8	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---	---

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



Wollen wir hier löschen,
müssen wir alles rechts
davon verschieben

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.



Wollen wir hier löschen,
müssen wir alles rechts
davon verschieben

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

1	5	6	8	8	9	3	3	8	9
---	---	---	---	---	---	---	---	---	---

Wollen wir hier löschen,
müssen wir alles rechts
davon verschieben

Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff



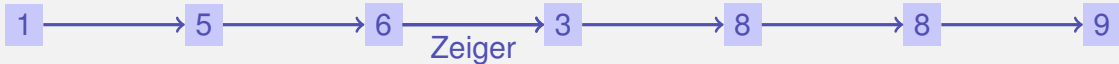
Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger



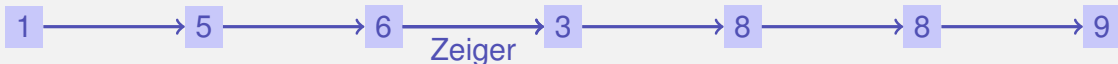
Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element "kennt" seinen Nachfolger



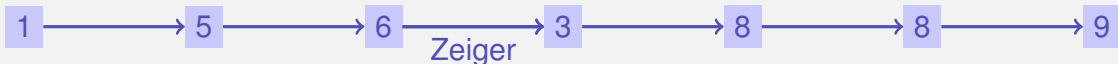
Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*



Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- ⇒ Ein Stapel kann als verkettete Liste realisiert werden

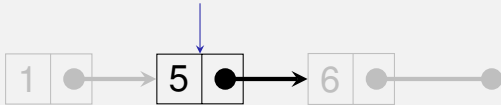


Verkettete Liste: Zoom



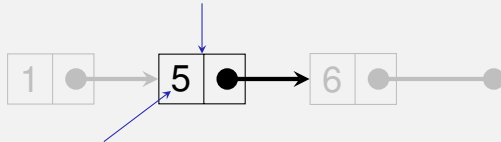
Verkettete Liste: Zoom

Element (Typ struct list_node)



Verkettete Liste: Zoom

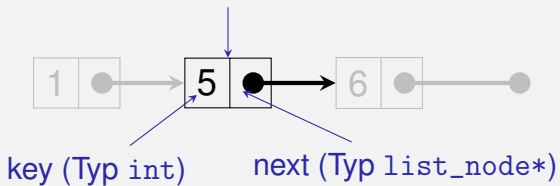
Element (Typ struct list_node)



key (Typ int)

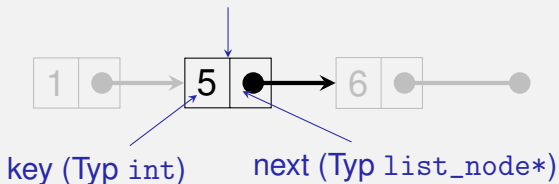
Verkettete Liste: Zoom

Element (Typ struct list_node)



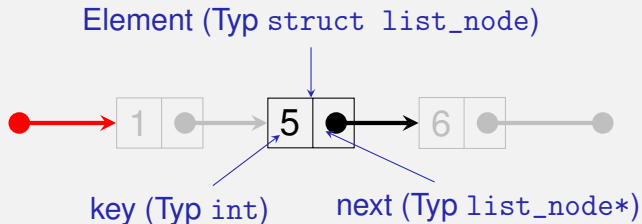
Verkettete Liste: Zoom

Element (Typ struct list_node)



```
struct list_node {  
    int          key;  
    list_node*  next;  
    // constructor  
    list_node (int k, list_node* n)  
        : key (k), next (n) {}  
};
```

Stapel = Zeiger aufs oberste Element

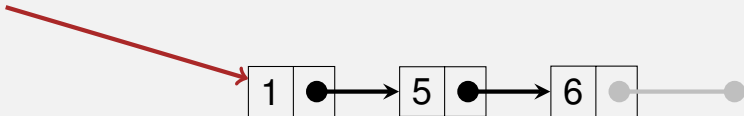


```
class stack {  
public:  
    void push (int value) {...}  
    ...  
private:  
    list_node* top_node;  
};
```

Sneak Preview: push(4)

```
void push (int value)
{
    top_node = new list_node (value, top_node);
}
```

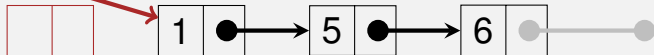
top_node



Sneak Preview: push(4)

```
void push (int value)
{
    top_node = new list_node (value, top_node);
}
```

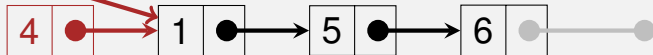
top_node



Sneak Preview: push(4)

```
void push (int value)
{
    top_node = new list_node (value, top_node);
}
```

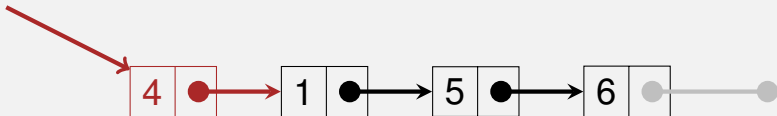
top_node



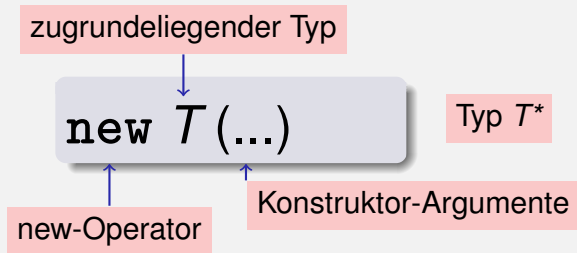
Sneak Preview: push(4)

```
void push (int value)
{
    top_node = new list_node (value, top_node);
}
```

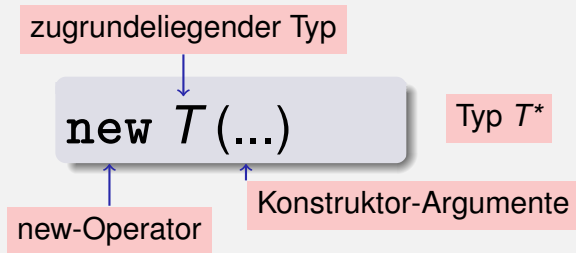
top_node



Der new-Ausdruck

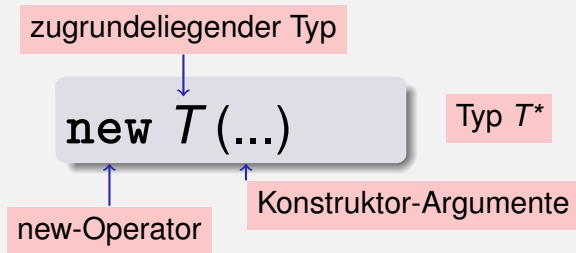


Der new-Ausdruck



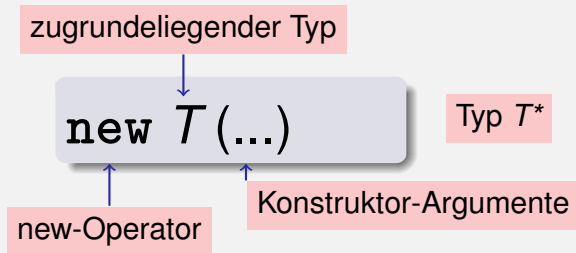
- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...

Der new-Ausdruck



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.

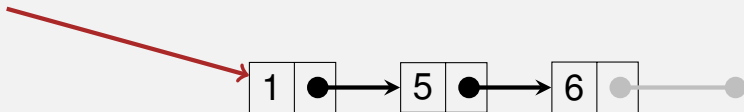
Der new-Ausdruck



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
top_node = new list_node (value, top_node);
```

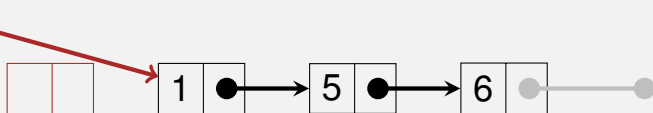
top_node



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...

```
top_node = new list_node (value, top_node);
```

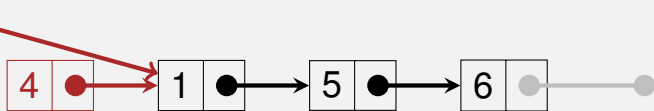
top_node



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.

```
top_node = new list_node (value, top_node);
```

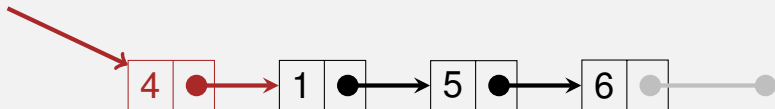
top_node



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
top_node = new list_node (value, top_node);
```

top_node

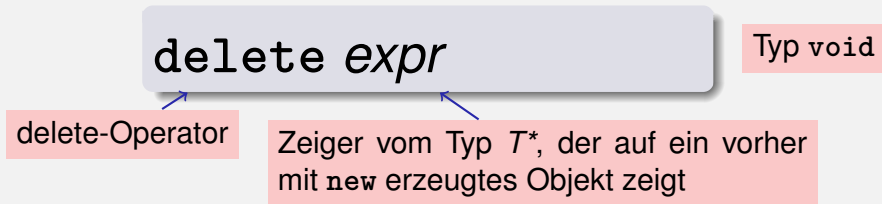


Der delete-Ausdruck

Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer*: sie “leben”, bis sie explizit *gelöscht* werden.

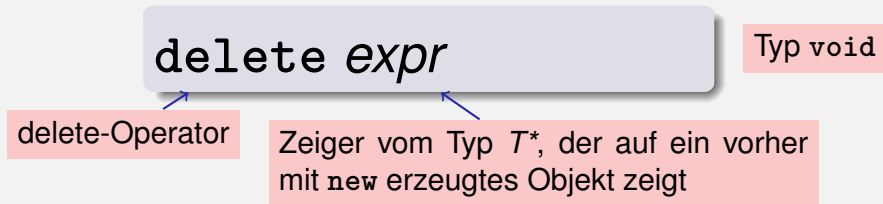
Der delete-Ausdruck

Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer*: sie “leben”, bis sie explizit *gelöscht* werden.



Der delete-Ausdruck

Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer*: sie “leben”, bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird gelöscht, Speicher wird wieder freigegeben

Wer geboren wird, muss sterben...

Richtlinie "Dynamischer Speicher"

Zu jedem `new` gibt es ein passendes `delete`!

Wer geboren wird, muss sterben...

Richtlinie "Dynamischer Speicher"

Zu jedem `new` gibt es ein passendes `delete`!

Nichtbeachtung führt zu *Speicherlecks*:

Wer geboren wird, muss sterben...

Richtlinie "Dynamischer Speicher"

Zu jedem `new` gibt es ein passendes `delete`!

Nichtbeachtung führt zu *Speicherlecks*:

- "Alte" Objekte, die den Speicher blockieren...

Wer geboren wird, muss sterben...

Richtlinie "Dynamischer Speicher"

Zu jedem `new` gibt es ein passendes `delete`!

Nichtbeachtung führt zu *Speicherlecks*:

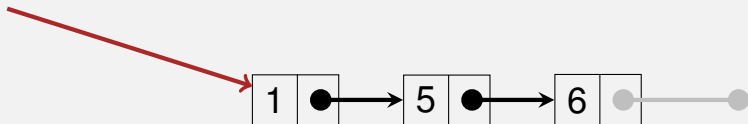
- "Alte" Objekte, die den Speicher blockieren. . .
- . . . bis er irgendwann voll ist (**heap overflow**)

Weiter mit dem Stapel:

pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node



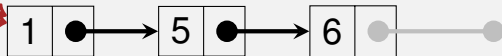
Weiter mit dem Stapel:

pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

p



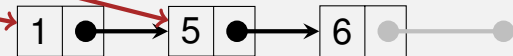
Weiter mit dem Stapel:

pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

p



Weiter mit dem Stapel:

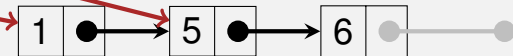
pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

Abkürzung für `(*top_node).next`

top_node

p



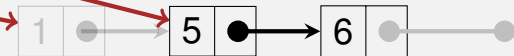
Weiter mit dem Stapel:

pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

p



Stapel traversieren:

print()

```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " ";
        p = p->next;
    }
}
```

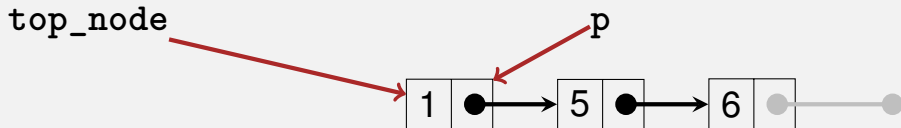
top_node



Stapel traversieren:

print()

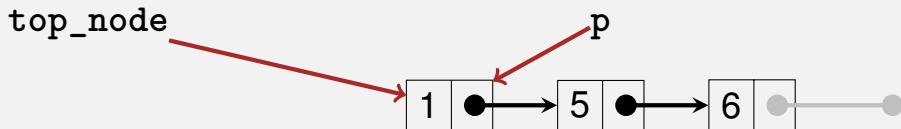
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " ";
        p = p->next;
    }
}
```



Stapel traversieren:

print()

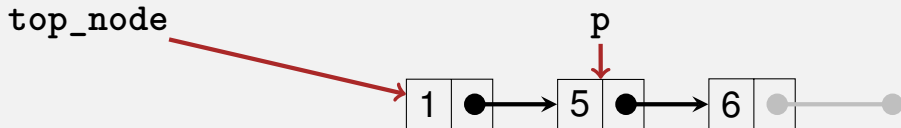
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1
        p = p->next;
    }
}
```



Stapel traversieren:

print()

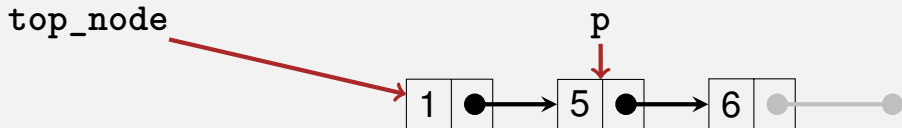
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1
        p = p->next;
    }
}
```



Stapel traversieren:

print()

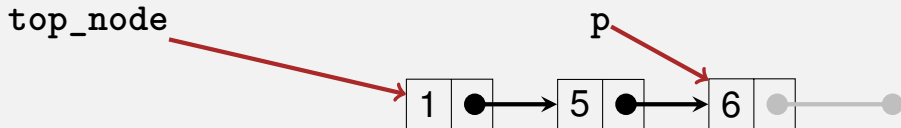
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1 5
        p = p->next;
    }
}
```



Stapel traversieren:

print()

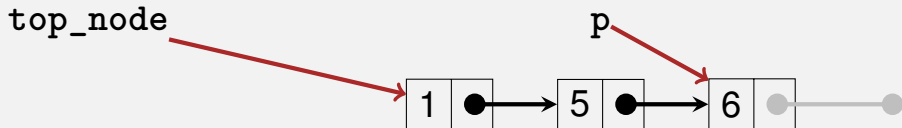
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1 5
        p = p->next;
    }
}
```



Stapel traversieren:

print()

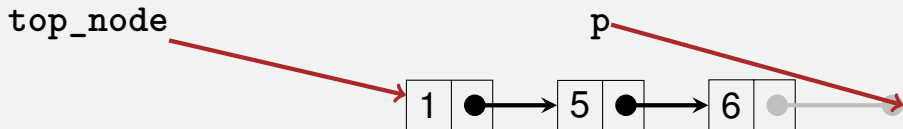
```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1 5 6
        p = p->next;
    }
}
```



Stapel traversieren:

print()

```
void print (std::ostream& o) const
{
    const list_node* p = top_node;
    while (p != 0) {
        o << p->key << " "; // 1 5 6
        p = p->next;
    }
}
```



Stapel ausgeben:

operator<<

```
class stack {  
public:  
    void push (int value) {...}  
    ...  
    void print (std::ostream& o) const {...}  
private:  
    list_node* top_node;  
};
```


Stapel ausgeben:

operator<<

```
class stack {
public:
    void push (int value) {...}
    ...
    void print (std::ostream& o) const {...}
private:
    list_node* top_node;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s)
{
    s.print (o);
    return o;
}
```

Leerer Stapel

```
stack()      // default constructor  
    : top_node (0)  
{}
```

Leerer Stapel , empty()

```
stack()      // default constructor
    : top_node (0)
{}

bool empty () const
{
    return top_node == 0;
}
```

Leerer Stapel , empty(), top()

```
stack()      // default constructor
    : top_node (0)
{}

```

```
bool empty () const
{
    return top_node == 0;
}

```

```
int top () const
{
    assert (!empty());
    return top_node->key;
}

```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```


Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```

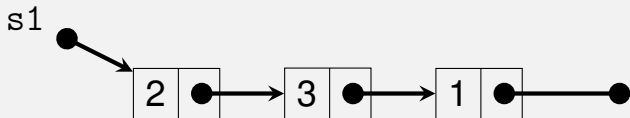
```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```


Was ist hier schiefgegangen?



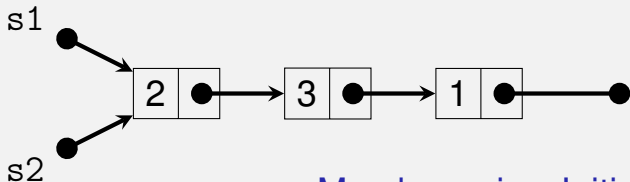
...

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



Memberweise Initialisierung: kopiert
nur den top_node-Zeiger

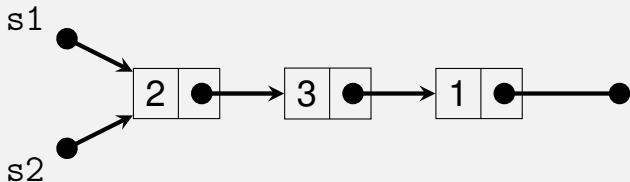
...

```
stack s2 = s1; ←  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



...

```
stack s2 = s1;
```

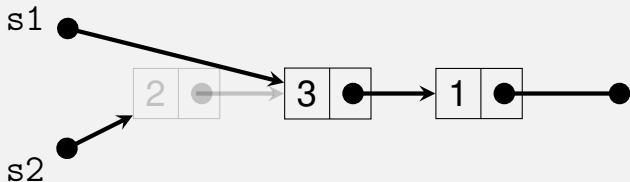
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



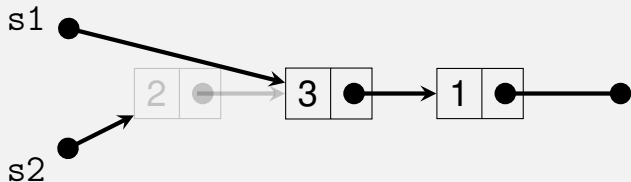
...

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



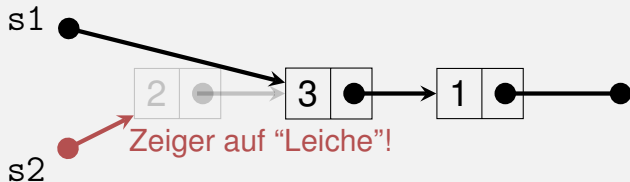
...

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



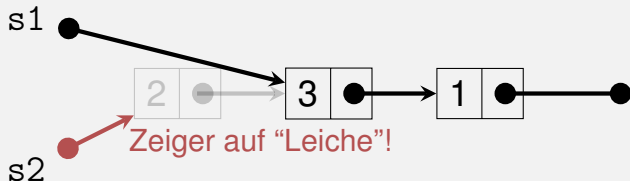
...

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Was ist hier schiefgegangen?



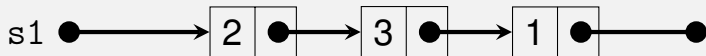
...

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```

Wir brauchen eine echte Kopie!



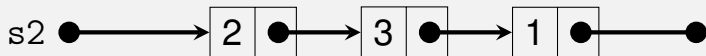
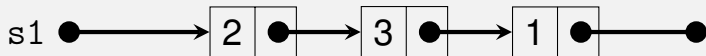
...

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```


Wir brauchen eine echte Kopie!



...

```
stack s2 = s1;
```

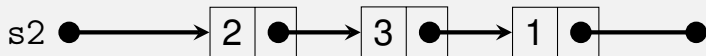
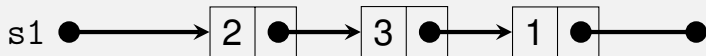
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Wir brauchen eine echte Kopie!



...

```
stack s2 = s1;
```

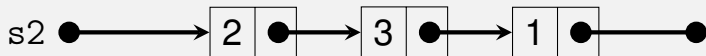
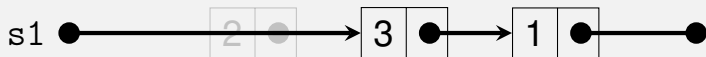
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Wir brauchen eine echte Kopie!



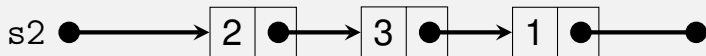
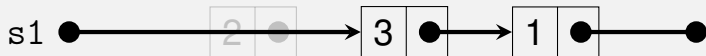
...

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Wir brauchen eine echte Kopie!



...

```
stack s2 = s1;
```

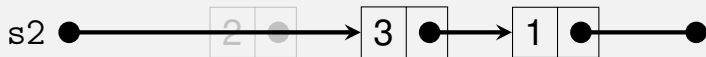
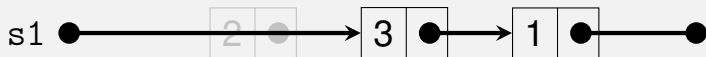
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Wir brauchen eine echte Kopie!



...

```
stack s2 = s1;
```

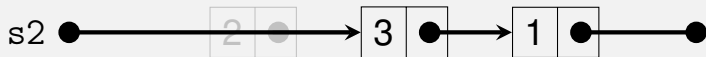
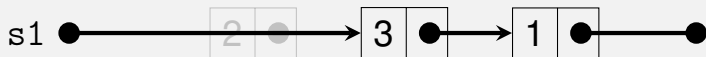
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Wir brauchen eine echte Kopie!



...

```
stack s2 = s1;
```

```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

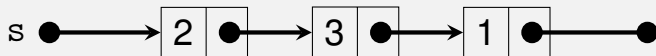
```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // ok
```

Mit dem Copy-Konstruktor klappt's!

Hier wird eine Kopierfunktion des `list_node` benutzt:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```

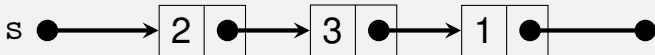


*this

Mit dem Copy-Konstruktor klappt's!

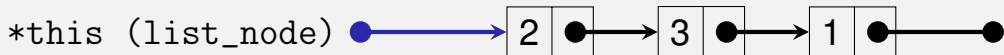
Hier wird eine Kopierfunktion des `list_node` benutzt:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



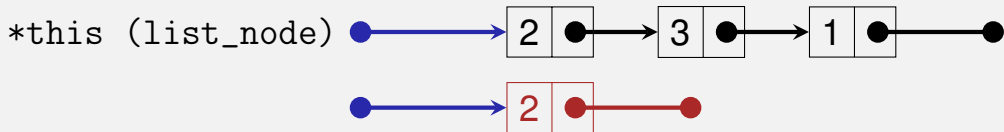
Die (rekursive) Kopierfunktion von list_node

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
    if (next != 0)
        return new list_node (key, next->copy());
    else
        return new list_node (key, 0);
}
```



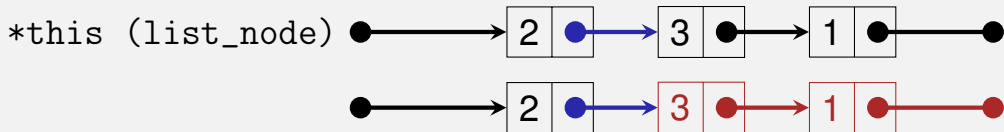
Die (rekursive) Kopierfunktion von list_node

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
    if (next != 0)
        return new list_node (key, next->copy());
    else
        return new list_node (key, 0);
}
```



Die (rekursive) Kopierfunktion von list_node

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
    if (next != 0)
        return new list_node (key, next->copy());
    else
        return new list_node (key, 0);
}
```



Initialisierung \neq Zuweisung!

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1; // Initialisierung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // ok: Copy-Konstruktor!
```

Initialisierung \neq Zuweisung!

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2;  
s2 = s1; // Zuweisung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // Oops, Programmabsturz!
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
    if (top_node != s.top_node) { // keine Selbstzuweisung!
        if (top_node != 0) {
            top_node->clear(); // loesche Knoten in *this
            top_node = 0;
        }
        if (s.top_node != 0)
            top_node = s.top_node->copy(); // kopiere s nach *this
    }
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
    if (top_node != s.top_node) { // keine Selbstzuweisung!
        if (top_node != 0) {
            top_node->clear(); // loesche Knoten in *this
            top_node = 0;
        }
        if (s.top_node != 0)
            top_node = s.top_node->copy(); // kopiere s nach *this
    }
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
    if (top_node != s.top_node) { // keine Selbstzuweisung!
        if (top_node != 0) {
            top_node->clear(); // loesche Knoten in *this
            top_node = 0;
        }
        if (s.top_node != 0)
            top_node = s.top_node->copy(); // kopiere s nach *this
    }
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```


Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
    if (top_node != s.top_node) { // keine Selbstzuweisung!
        if (top_node != 0) {
            top_node->clear(); // loesche Knoten in *this
            top_node = 0;
        }
        if (s.top_node != 0)
            top_node = s.top_node->copy(); // kopiere s nach *this
    }
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

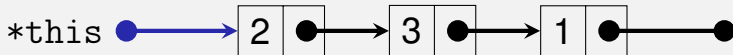
Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
    if (top_node != s.top_node) { // keine Selbstzuweisung!
        if (top_node != 0) {
            top_node->clear(); // loesche Knoten in *this
            top_node = 0;
        }
        if (s.top_node != 0)
            top_node = s.top_node->copy(); // kopiere s nach *this
    }
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Die (rekursive) Aufräumfunktion des list_node

```
// POST: the list starting at *this is deleted
void clear ()
{
    if (next != 0)
        next->clear();
    delete this;
}
```



Die (rekursive) Aufräumfunktion des list_node

```
// POST: the list starting at *this is deleted
void clear ()
{
    if (next != 0)
        next->clear();
    delete this;
}
```



Die (rekursive) Aufräumfunktion des list_node

```
// POST: the list starting at *this is deleted
void clear ()
{
    if (next != 0)
        next->clear();
    delete this;
}
```



Zombie-Elemente

```
{
    stack s1; // local variable
    s1.push (1);
    s1.push (3);
    s1.push (2);
    std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

Zombie-Elemente

```
{  
    stack s1; // local variable  
    s1.push (1);  
    s1.push (3);  
    s1.push (2);  
    std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 has died (become invalid)...
```

- ... aber die drei *Elemente* des Stapels s1 leben weiter (Speicherleck)!

Zombie-Elemente

```
{  
    stack s1; // local variable  
    s1.push (1);  
    s1.push (3);  
    s1.push (2);  
    std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 has died (become invalid)...
```

- ... aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!
- Sie sollten zusammen mit `s1` aufgeräumt werden!

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
    if (top_node != 0)
        top_node->clear();
}
```

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
    if (top_node != 0)
        top_node->clear();
}
```

- löscht automatisch alle Stapелеlemente, wenn der Stapel ungültig wird

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
    if (top_node != 0)
        top_node->clear();
}
```

- löscht automatisch alle Stapелеlemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse befolgt jetzt die Richtlinie “Dynamischer Speicher”!

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Andere Anwendungen:
 - Listen (mit Einfügen und Löschen “in der Mitte”)
 - Bäume (nächste Woche)
 - Warteschlangen
 - Graphen

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
 - Mindestfunktionalität:
 - Konstruktoren
 - Destruktor
 - Copy-Konstruktor
 - Zuweisungsoperator
- } Dreierregel: definiert eine Klasse eines davon, so sollte sie auch die anderen zwei definieren!