19. 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
};
```

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
};
```

öffentlicher Bereich

Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Gültigkeitsbereich von Membern in einer Klasse ist die ganze Klasse, unabhängig von der Deklarationsreihenfolge

Memberfunktionen: Aufruf

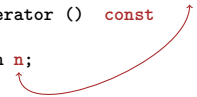
```
// Definition des Typs
class rational {
   ...
};
...
// Variable des Typs
rational r;

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

Member-Zugriff

Memberfunktionen: Definition

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



- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen. In der Funktion: `*this` ist der Name dieses impliziten Arguments. `this` selbst ist ein Zeiger darauf.
- Das const bezieht sich auf `*this`, verspricht also, dass das implizite Argument nicht im Wert verändert wird.
- `n` ist Abkürzung in der Memberfunktion für `(*this).n`

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...
public:
    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 vs. Out-of-Class

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

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

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

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

- So geht's auch.

Konstruktoren

- sind spezielle *Memberfunktionen* einer Klasse, die den Namen der Klasse tragen.
- können wie Funktionen überladen werden, also in der Klasse mehrfach, aber mit verschiedener *Signatur* vorkommen.
- werden bei der Variablendeklaration wie eine Funktion aufgerufen. Der Compiler sucht die „naheliegendste“ passende Funktion aus.
- wird kein passender Konstruktor gefunden, so gibt der Compiler eine *Fehlermeldung* aus.

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 der Membervariablen

Funktionsrumpf.

Konstruktoren: Aufruf

- direkt

```
rational r (1,2); // initialisiert r mit 1/2
```

- indirekt (Kopie)

```
rational r = rational (1,2);
```


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
```

Leerer Funktionsrumpf

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

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

Benutzerdefinierte Konversion von `int` nach `rational`. Damit wird `int` zu einem Typ, dessen Werte nach `rational` konvertierbar sind.

```
rational r = 2; // implizite Konversion
```

Der Default-Konstruktor

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

Leere Argumentliste

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Der Default-Konstruktor

- wird automatisch aufgerufen bei Deklarationen der Form `rational r;`
- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn `rational r;` kompilieren soll
- wenn in einem Struct keine Konstruktoren definiert wurden, wird der Default-Konstruktor automatisch erzeugt (wegen der Sprache C)

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;
};
```

```
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;
  }
}
```

- 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
};
```

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

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:
   using integer = int; // might change
   // POST: returns numerator of *this
   integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!
- Festlegung nur auf Funktionalität, z.B.:
  - implizite Konversion `int` → `rational::integer`
  - Funktion `double to_double (rational::integer)`

RAT PACK® Revolutions

Endlich ein Kundenprogramm, das stabil bleibt:

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

Deklaration und Definition getrennt

rational.h

```
class rational {
public:
   rational (int num, int denum);
   using integer = int;
   integer numerator () const;
   ...
private:
 ...
};
```

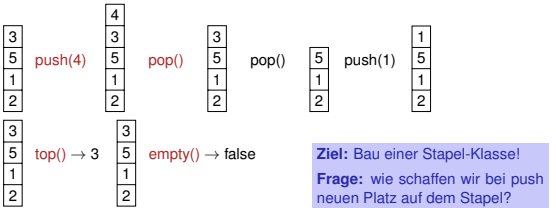
rational.cpp

```
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

Klassenname :: Membername

Motivation: Stapel

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: Feld (`T[]`)

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

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...

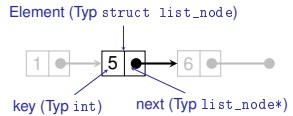
- Einfügen oder Löschen von Elementen „in der Mitte“ ist aufwändig.

Wollen wir hier löschen, müssen wir alles rechts davon verschieben

Der neue Container: Verkettete Liste

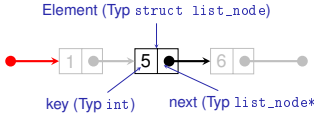
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element „kennt“ seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- ⇒ Ein Stapel kann als verkettete Liste realisiert werden

Verkettete Liste: Zoom




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

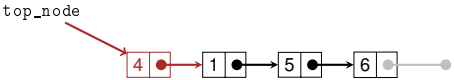
Stapel = Zeiger aufs oberste Element




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


Sneak Preview: push(4)

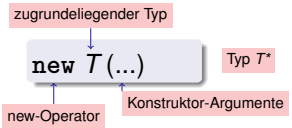
```
void stack::push (int value)
{
  top_node = new list_node (value, top_node);
}
```



Dynamischer Speicher

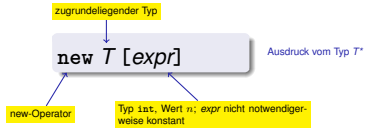
- Für dynamische Datenstrukturen wie Listen benötigt man *dynamischen Speicher*
- Bisher wurde die Grösse des Speicherplatzes für Variablen zur *Compilezeit* festgelegt
- Zeiger erlauben das Anfordern neuen Speichers zur *Laufzeit*
- Dynamische Speicherverwaltung in C++ mit Operatoren `new` und `delete`

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

Der `new`-Ausdruck für Felder

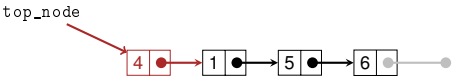


- Neuer Speicher für ein Feld der Länge n mit zugrundeliegendem Typ T wird angelegt
- Wert des Ausdrucks ist Adresse des ersten Elements des Feldes

Der `new`-Ausdruck: `push(4)`

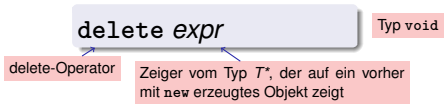
- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

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



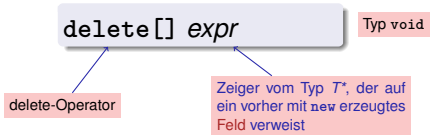
Der `delete`-Ausdruck

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

Der `delete`-Ausdruck für Felder



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

Aufpassen mit `new` und `delete`!

```
rational* t = new rational;
rational* s = t;
delete s;
int nominator =  (*t).denominator();
```

Speicher für t wird angelegt

Auch andere Zeiger können auf das Objekt zeigen..

... und zur Freigabe verwendet werden..

Fehler: Speicher freigegeben!

Dereferenzieren eines „dangling pointers“

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit `delete` ist ein ähnlicher schwerer Fehler.
- `delete` kann leicht vergessen werden: Folge sind Speicherlecks (*memory leaks*). Kann auf Dauer zu Speicherüberlauf führen.

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)

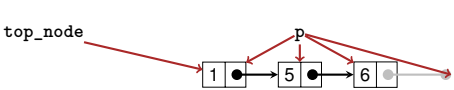
Weiter mit dem Stapel: `pop()`

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

Abkürzung für `(*top_node).next`

Stapel traversieren: `print()`

```
void stack::print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != nullptr) {
    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;
};

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


Leerer Stapel , empty(), top()

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

bool stack::empty () const
{
  return top_node == nullptr;
}

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


Stapel fertig? Offenbar noch nicht...

```
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?

Memberweise Initialisierung: kopiert nur den `top_node`-Zeiger

```
...
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"; // 2 3 1

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

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


Der Copy-Konstruktor

- Der Copy-Konstruktor einer 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

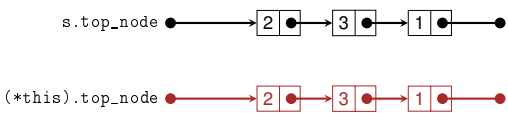
  T `x = t;` (t vom Typ T)

  T `x (t);`
- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

Mit dem Copy-Konstruktor klappt's!

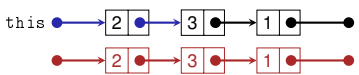
Hier wird eine Kopierfunktion des `list_node` benutzt:

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s)
  : top_node (nullptr)
{
  if (s.top_node != nullptr)
    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* list_node::copy () const
{
  if (next != nullptr)
    return new list_node (key, next->copy());
  else
    return new list_node (key, nullptr);
}
```



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!
```


Der Zuweisungsoperator

- Überladung von `operator=` als Memberfunktion
- Wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
  - Freigabe des Speichers für den „alten“ Wert
  - Prüfen auf Selbstzuweisungen (`s1=s1`), die keinen Effekt haben sollen
- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist memberweise zu – Grund für obiges Problem)

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& stack::operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbstzuweisung!
    if (top_node != nullptr) {
      top_node->clear(); // loesche Listenknoten
      top_node = nullptr;
    }
    if (s.top_node != nullptr)
      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 list_node::clear ()
{
  if (next != nullptr)
    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)...
```

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

Der Destruktor

- Der Destruktor einer Klasse T ist die eindeutige Memberfunktion mit Deklaration

$$\sim T\,();$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts endet
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger `top_node`, kein Effekt – Grund für Zombie-Elemente)

Mit dem Destruktor klappt's!

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

- löscht automatisch alle Stapelemente, 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)
- Andere Anwendungen:
  - Listen (mit Einfügen und Löschen „in der Mitte")
  - Bäume (nächste Woche)
  - Warteschlangen
  - Graphen
- Mindestfunktionalität:
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

  } Dreierregel: definiert eine Klasse eines davon, so sollte sie auch die anderen zwei definieren!