16. Structs und Klassen I

Rationale Zahlen, Struct-Definition, Operator-Überladung, Datenkapselung, Klassen-Typen

Rechnen mit rationalen Zahlen

- Rationale Zahlen ($\mathbb{Q}$) sind von der Form $\frac{n}{d}$ mit n und d aus $\mathbb{Z}$
- C++hat keinen „eingebauten” Typ für rationale Zahlen

Ziel

Wir bauen uns selbst einen C++-Typ für rationale Zahlen! 🙂

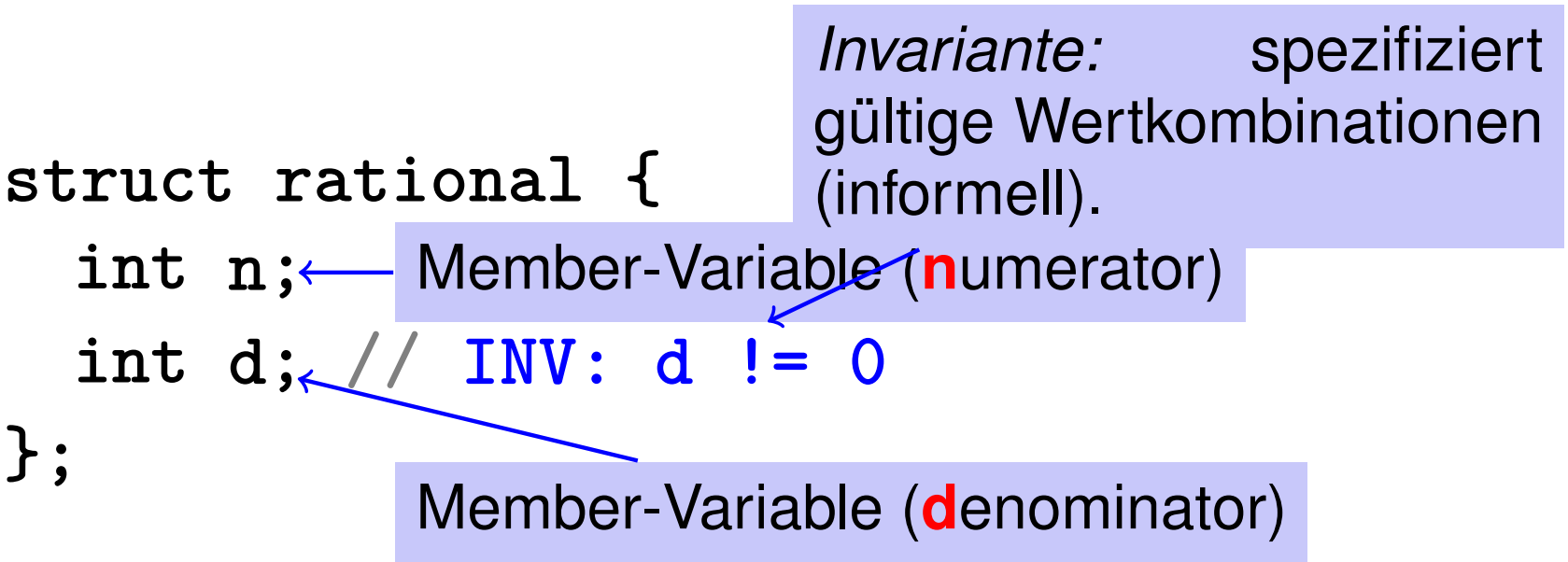
Vision

So könnte (wird) es aussehen

```
// input
std::cout << "Rational number r:\n";
rational r;
std::cin >> r;
std::cout << "Rational number s:\n";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

Ein erstes Struct



- `struct` definiert einen neuen *Typ*
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen
- Echter Wertebereich: $\texttt{rational} \subset \texttt{int} \times \texttt{int}$.

Zugriff auf Member-Variablen

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

rational add (const rational a, const rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

Ein erstes Struct: Funktionalität

```
// new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (const rational a, const rational b)
{
  rational result;
  result.n =  a.n * b.d +  a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

Ein `struct` definiert einen *Typ*, keine *Variable*!

Bedeutung: jedes Objekt des neuen Typs ist durch zwei Objekte vom Typ `int` repräsentiert, die die Namen `n` und `d` tragen.

Member-Zugriff auf die `int`-Objekte von `a`.

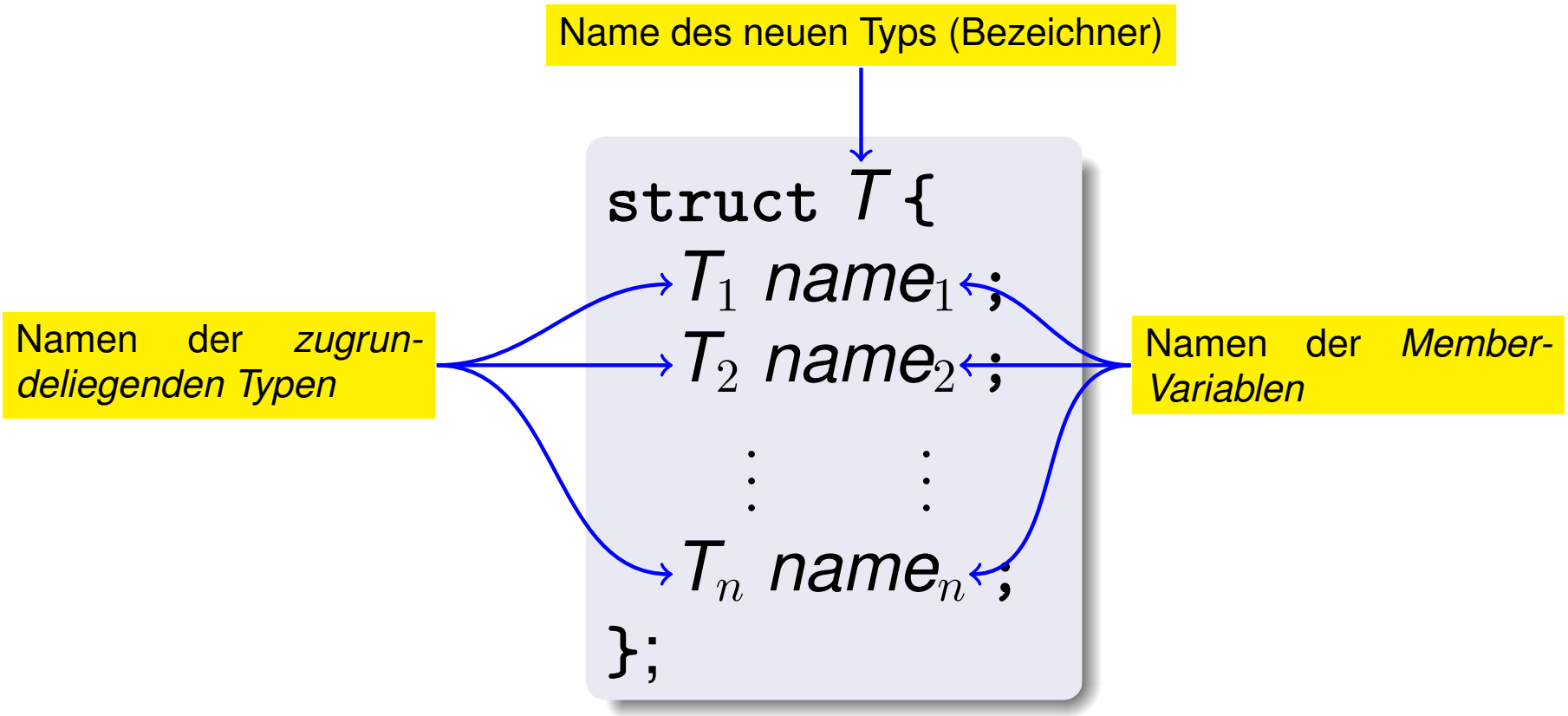
Vision in Reichweite ...

```
// Eingabe von r
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? "; std::cin >> r.n;
std::cout << " denominator =? "; std::cin >> r.d;
// Eingabe von s ebenso
rational s;
...

// computation
const rational t = add (r, s);

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
```

Struct-Definitionen



Wertebereich von T: $T_1 \times T_2 \times \ldots \times T_n$

Struct-Definitionen: Beispiele

```
struct rational_vector_3 {
  rational x;
  rational y;
  rational z;
};
```

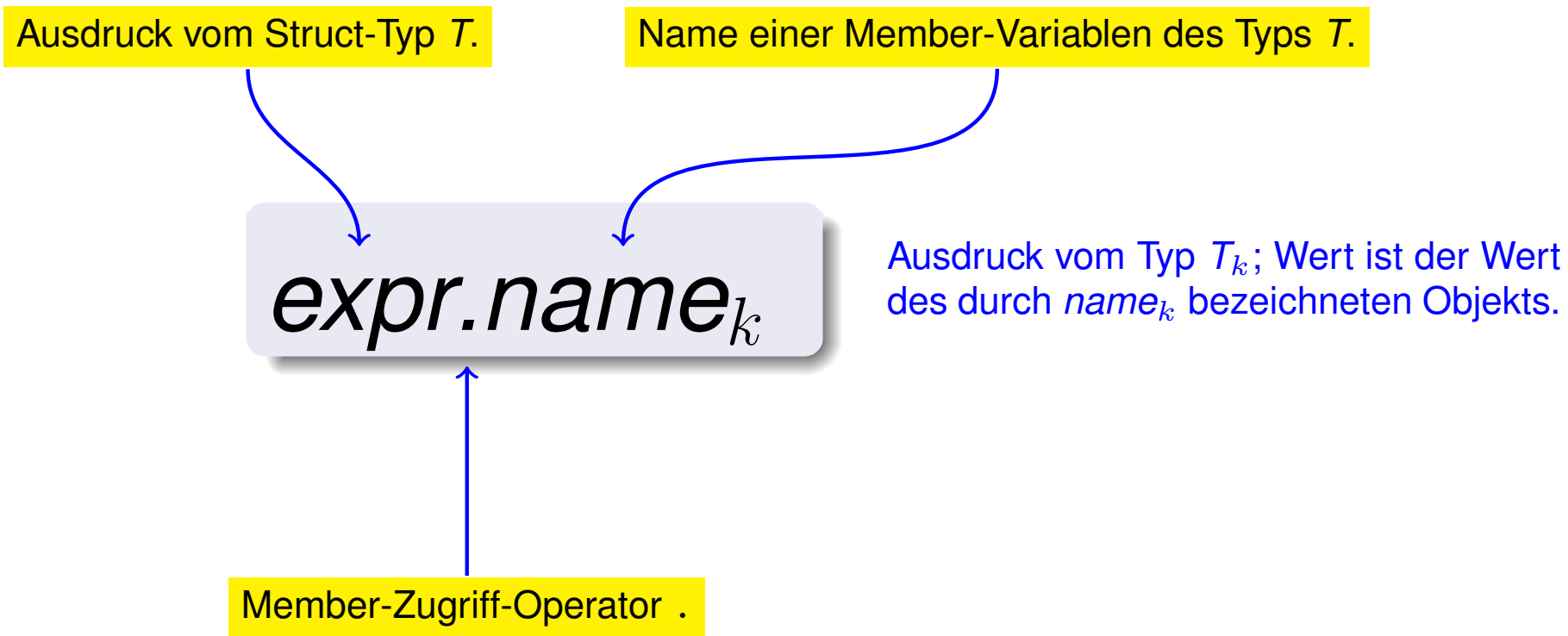
Zugrundeliegende Typen können fundamentale aber auch benutzerdefinierte Typen sein.

Struct-Definitionen: Beispiele

```
struct extended_int {
  // represents value if is_positive==true
  // and -value otherwise
  unsigned int value;
  bool is_positive;
};
```

Die zugrundeliegenden Typen können natürlich auch verschieden sein.

Structs: Member-Zugriff



Structs: Initialisierung und Zuweisung

Default-Initialisierung:

```
rational t;
```

- Member-Variablen von `t` werden default-initialisiert
- für Member-Variablen fundamentaler Typen passiert dabei nichts (Wert undefiniert)

Structs: Initialisierung und Zuweisung

Initialisierung:

```
rational t = {5, 1};
```

- Member-Variablen von `t` werden mit den Werten der Liste, entsprechend der Deklarationsreihenfolge, initialisiert.

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational s;
...
rational t = s;
```

- Den Member-Variablen von `t` werden die Werte der Datenmitglieder von `s` zugewiesen.

Structs: Initialisierung und Zuweisung

```
t.n          .n
t.d = add (r, s)  .d  ;
```

Initialisierung:

```
rational t = add(r, s);
```

- `t` wird mit dem Wert von `add(r, s)` initialisiert

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational t;
t=add(r, s);
```

- `t` wird default-initialisiert
- Der Wert von `add(r, s)` wird `t` zugewiesen

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`,...) gibt es die Vergleichsoperatoren `==` und `!=` , aber nicht für Structs! Warum?

- Memberweiser Vergleich ergibt im allgemeinen keinen Sinn,...
- ...denn dann wäre z.B. $\frac{2}{3} \neq \frac{4}{6}$

Structs als Funktionsargumente

```
void increment(rational dest, const rational src)
{
    dest = add(dest, src); // veraendert nur lokale Kopie
}
```

Call by Value !

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment(b, a); // kein Effekt!
std::cout << b.n << "/" << b.d; // 1 / 2
```

Structs als Funktionsargumente

```
void increment(rational & dest, const rational src)
{
    dest = add(dest, src);
}
```

Call by Reference

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment(b, a);
std::cout << b.n << "/" << b.d; // 2 / 2
```

Benutzerdefinierte Operatoren

Statt

```
rational t = add(r, s);
```

würden wir lieber

```
rational t = r + s;
```

schreiben.

Das geht mit *Operator-Überladung*.

Funktions-Überladung

Erinnerung: Verschiedene Funktionen können den gleichen Namen haben.

```
// POST: returns a * a
int square (int a);

// POST: returns a * a
rational square (rational a);
```

Der Compiler findet anhand der Aufrufargumente heraus, welche gemeint ist.

Operator-Überladung (Operator Overloading)

- Operatoren sind spezielle Funktionen und können auch überladen werden
- Name des Operators *op*:

```
operatorop
```

- Wir wissen schon, dass z.B. `operator+` für verschiedene Typen existiert

`rational` addieren, bisher

```
// POST: return value is the sum of a and b
rational add (const rational a,
              const rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = add (r, s);
```

`rational` addieren, neu

```
// POST: return value is the sum of a and b
rational operator+ (const rational a,
                    const rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
 ...
const rational t = r + s;
```

Infix-Notation

Andere binäre Operatoren für rationale Zahlen

```
// POST: return value is difference of a and b
rational operator- (rational a, rational b);

// POST: return value is the product of a and b
rational operator* (rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator/ (rational a, rational b);
```

Unäres Minus

Hat gleiches Symbol wie binäres Minus, aber nur ein Argument:

```
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```

Vergleichsoperatoren

Sind für Structs nicht eingebaut, können aber definiert werden:

```
// POST: returns true iff a == b
bool operator== (rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

$$\frac{2}{3} = \frac{4}{6} \quad \checkmark$$

Arithmetische Zuweisungen

Wir wollen z.B. schreiben

```
rational r;
r.n = 1; r.d = 2;                          // 1/2

rational s;
s.n = 1; s.d = 3;                          // 1/3

r += s;
std::cout << r.n << "/" << r.d;            // 5/6
```

Operator+= Erster Versuch

```
rational operator+= (rational a, const rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Das funktioniert nicht! Warum?

- Der Ausdruck `r += s` hat zwar den gewünschten Wert, weil die Aufrufargumente R-Werte sind (call by value!) jedoch nicht den gewünschten Effekt der Veränderung von `r`.
- Das Resultat von `r += s` stellt zudem entgegen der Konvention von C++ keinen L-Wert dar.

Operator +=

```
rational& operator+= (rational& a,
                      const rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Das funktioniert!

- Der L-Wert `a` wird um den Wert von `b` erhöht und als L-Wert zurückgegeben.

`r += s;` hat nun den gewünschten Effekt.

Ein-/Ausgabeoperatoren

können auch überladen werden.

- Bisher:

```
std::cout << "Sum is "
          << t.n << "/" << t.d << "\n";
```

- Neu (gewünscht)

```
std::cout << "Sum is "
          << t << "\n";
```

Ein-/Ausgabeoperatoren

können auch überladen werden wie folgt:

```
// POST: r has been written to out
std::ostream& operator<< (std::ostream& out,
                          const rational r)
{
    return out << r.n << "/" << r.d;
}
```

schreibt `r` auf den Ausgabestrom
und gibt diesen als L-Wert zurück

Eingabe

```
// PRE: in starts with a rational number
// of the form "n/d"
// POST: r has been read from in
std::istream& operator>> (std::istream& in,
                          rational& r)
{
    char c; // separating character '/'
    return in >> r.n >> c >> r.d;
}
```

liest `r` aus dem Eingabestrom `i`
und gibt diesen als L-Wert zurück

Zwischenziel erreicht!

```
// input
std::cout << "Rational number r:\n";
rational r;
std::cin >> r;

std::cout << "Rational number s:\n";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

operator >>

operator +

operator<<

Ein neuer Typ mit Funktionalität...

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


// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
```

...gehört in eine Bibliothek!

`rational.h:`

- Definition des Structs `rational`
- Funktionsdeklarationen

`rational.cpp:`

- Arithmetische Operatoren (`operator+`, `operator+=`, ...)
- Relationale Operatoren (`operator==`, `operator>`, ...)
- Ein-/Ausgabe (`operator >>`, `operator <<`, ...)

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung
- Lehre
- Technologietransfer

Wir gründen die Startup-Firma RAT PACK®!

- Verkauf der `rational`-Bibliothek an Kunden
- Weiterentwicklung nach Kundenwünschen

Der Kunde ist zufrieden

...und programmiert fleissig mit `rational`.

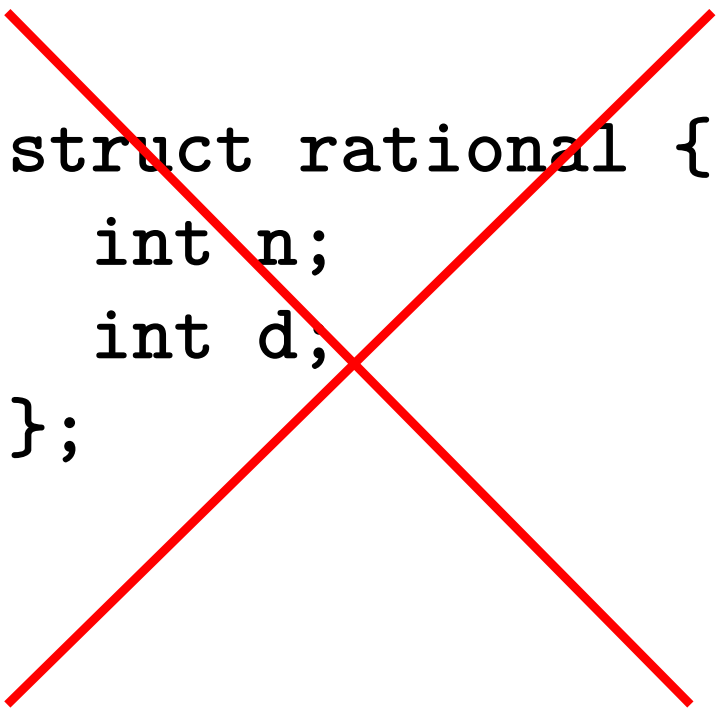
- Ausgabe als `double`-Wert ($\frac{3}{5} \rightarrow 0.6$)

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

Der Kunde will mehr

“Können wir rationale Zahlen mit erweitertem Wertebereich bekommen?”

- Klar, kein Problem, z.B.:



```
struct rational {
  int n;
  int d;
};
```

$\Rightarrow$

```
struct rational {
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

Neue Version von RAT PACK®

Nichts geht mehr!

- Was ist denn das Problem?

$-\frac{3}{5}$ ist jetzt manchmal 0.6, das kann doch nicht sein!

- Daran ist wohl Ihre Konversion nach `double` schuld, denn unsere Bibliothek ist korrekt.

Bisher funktionierte es aber, also ist die neue Version schuld!

Schuldanalyse

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

`r.is_positive` und `result.is_positive` kommen nicht vor.

Korrekt mit...

```
struct rational {
  int n;
  int d;
};
```

...aber nicht mit

```
struct rational {
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

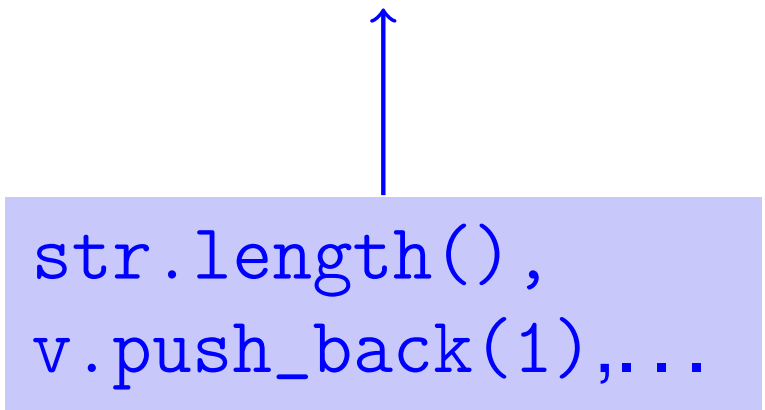
Wir sind schuld!

- Kunde sieht und benutzt unsere Repräsentation rationaler Zahlen (zu Beginn `r.n, r.d`)
- Ändern wir sie (`r.n, r.d, r.is_positive`), funktionieren Kunden-Programme nicht mehr.
- Kein Kunde ist bereit, bei jeder neuen Version der Bibliothek seine Programme anzupassen.

$\Rightarrow$ RAT PACK® ist Geschichte...

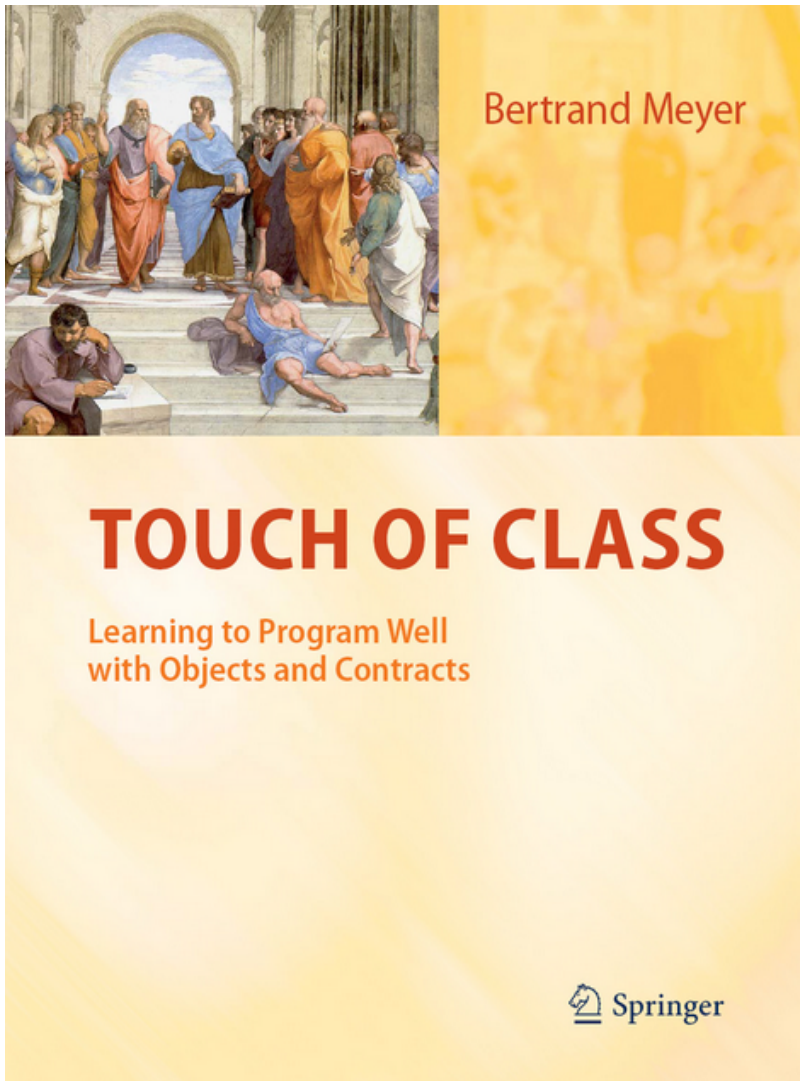
Idee der Datenkapselung

- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.
- Die Repräsentation soll nicht sichtbar sein.
- $\Rightarrow$ Dem Kunden wird keine Repräsentation, sondern Funktionalität angeboten.



Klassen

- sind das Konzept zur Datenkapselung in C++
- verallgemeinern Structs
- sind Bestandteil jeder objektorientierten Programmiersprache
- C++: Objektorientierung unvollständig (nicht alle Typen sind Klassen)



Datenkapselung: `public` / `private`

Wird statt `struct` verwendet, wenn überhaupt etwas "versteckt" werden soll.

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

Einziger Unterschied:

- `struct`: standardmässig wird *nichts* versteckt
- `class` : standardmässig wird *alles* versteckt

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...

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

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
```