

# 16. Structs und Klassen I

Rationale Zahlen, Struct-Definition, Funktions- und Operatorüberladung, Const-Referenzen, Datenkapselung

# 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 =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

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

# Ein erstes Struct

```
struct rational {  
    int n; ← Member-Variable (numerator)  
    int d; ← // INV: d != 0  
};  
           ← Member-Variable (denominator)
```

*Invariante:* spezifiziert gültige Wertkombinationen (informell).

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

# Zugriff auf Member-Variablen

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

rational add (rational a, 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

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

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

```
// 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;
}
```

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

# Eingabe

```
// 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;
...
```

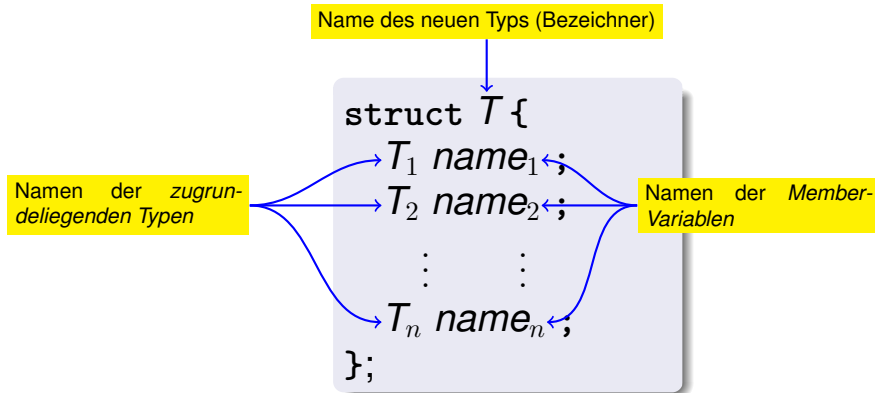
# Vision in Reichweite ...

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

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



# Struct-Definitionen



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

# 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

Ausdruck vom Struct-Typ  $T$ .

Name einer Member-Variablen des Typs  $T$ .

$expr.name_k$

Ausdruck vom Typ  $T_k$ ; Wert ist der Wert des durch  $name_k$  bezeichneten Objekts.

Member-Zugriff-Operator  $.$

# 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    = add (r, s) .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*.

# Überladen von Funktionen

- Funktionen sind durch Ihren Namen im Gültigkeitsbereich ansprechbar
- Es ist sogar möglich, mehrere Funktionen des gleichen Namens zu definieren und zu deklarieren
- Die „richtige“ Version wird aufgrund der *Signatur* der Funktion ausgewählt

# Funktionsüberladung

- Eine Funktion ist bestimmt durch Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... }           // f1
int sq (int x) { ... }                 // f2
int pow (int b, int e) { ... }        // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“ (wir vertiefen das nicht)

```
std::cout << sq (3);           // Compiler wählt f2
std::cout << sq (1.414);       // Compiler wählt f1
std::cout << pow (2);          // Compiler wählt f4
std::cout << pow (3,3);        // Compiler wählt f3
```

# Überladen von Funktionen

Mit dem Überladen von Funktionen lassen sich also

- verschiedene Aufrufvarianten des gleichen Algorithmus realisieren und / oder
- verschiedene Algorithmen für verschiedene Datentypen mit dem gleichen Namen `sq(int)` vs. `sq(double)` verbinden.

Funktioniert ein Algorithmus für verschiedene Datentypen identisch, so verwendet man in C++ *Templates*.



# 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 (rational a, 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+ (rational a, 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, 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, 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,
                          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  
und gibt diesen als L-Wert zurück

# Ziel erreicht!

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

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

operator >>

operator +

operator <<

# Grosse Objekte ...

```
struct SimulatedCPU {  
    unsigned int pc;  
    unsigned int registers[4];  
    unsigned int memory[65536];  
};
```

Call by value: mehr als 256k werden kopiert!

```
void outputState (SimulatedCPU p) {  
    std::cout << p.pc;  
    for (int i = 0; i <4; ++i)  
        std::cout << p.registers[i];  
}
```

# ... übergibt man als Referenz

```
struct SimulatedCPU{  
    unsigned int pc;  
    unsigned int registers[4];  
    unsigned int memory[65536];  
};
```

Call by reference: nur eine Adresse wird kopiert.

```
void outputState (SimulatedCPU& p) {  
    std::cout << p.pc;  
    for (int i = 0; i < 4; ++i)  
        std::cout << p.registers[i];  
}
```

“Und was ist mit Const-Correctness?”

# So ist es richtig!

```
struct SimulatedCPU {
    unsigned int pc;
    unsigned int registers[4];
    unsigned int memory[65536];
};

void outputState (const SimulatedCPU& p) {
    std::cout << p.pc;
    for (int i = 0; i < 4; ++i)
        std::cout << p.registers[i];
}
```



# Const-Referenzen

- haben Typ `const T &` (= `const (T &)`)
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

```
const T& r = lvalue;
```

r wird mit der Adresse von *lvalue* initialisiert (effizient)

```
const T& r = rvalue;
```

r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)


# Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`

- Fall 1:  $T$  ist kein Referenztyp

Dann ist der L-Wert eine **Konstante**.

```
const int n = 5;  
int& i = n; // error: const-qualification is discarded  
i = 6;
```



Der Schummelversuch wird vom Compiler erkannt

# Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`

## ■ Fall 2: $T$ ist Referenztyp

Dann ist der L-Wert ein Lese-Alias, **durch den der Wert dahinter nicht verändert werden darf.**

```
int n = 5;
const int& i = n; // i: Lese-Alias von n
int& j = n;      // j: Lese-Schreib-Alias
i = 6;          // Fehler: i ist Lese-Alias
j = 6;          // ok: n bekommt Wert 6
```

# Wann `const T&` ?

## Regel

Argumenttyp `const T&` (call by *read-only* reference) wird aus Effizienzgründen anstatt `T` (call by value) benutzt, wenn der Typ `T` grossen Speicherbedarf hat. Für fundamentale Typen und kleine Structs (`rational`) lohnt es sich aber nicht.

# 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 (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;  
};
```

⇒

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

# Neue Version von RAT PACK<sup>®</sup>



*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 (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.

⇒ RAT PACK<sup>®</sup> ist Geschichte...

# Idee der Datenkapselung

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



```
str.length(),  
v.push_back(1),...
```

# Klassen

- sind das Konzept zur Datenkapselung in C++
- sind eine Variante von Structs
- gibt es in vielen **objektorientierten Programmiersprachen**

# Datenkapselung: public / private

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

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

*Einzig*er Unterschied:

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