

## 17. Rekursion 2

---

Bau eines Taschenrechners, Formale Grammatiken, Extended Backus Naur Form (EBNF), Parsen von Ausdrücken

# Motivation: Taschenrechner

Eingabe: 3 + 5

Ausgabe: 8

- Binäre Operatoren +, -, \*, / und Zahlen

# Motivation: Taschenrechner

Eingabe: 3 / 5

Ausgabe: 0.6

- Binäre Operatoren +, -, \*, / und Zahlen
- Fließkommaarithmetik

# Motivation: Taschenrechner

Eingabe:  $3 + 5 * 20$

Ausgabe: 103

- Binäre Operatoren  $+$ ,  $-$ ,  $*$ ,  $/$  und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++

# Motivation: Taschenrechner

Eingabe:  $(3 + 5) * 20$

Ausgabe: **160**

- Binäre Operatoren  $+$ ,  $-$ ,  $*$ ,  $/$  und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung

# Motivation: Taschenrechner

Eingabe:  $-(3 + 5) + 20$

Ausgabe: **12**

- Binäre Operatoren  $+$ ,  $-$ ,  $*$ ,  $/$  und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung
- Unärer Operator  $-$

# Naiver Versuch (ohne Klammern)

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

# Scheint zu klappen...

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

Eingabe 1 \* 2 \* 3 \* 4 =  
Ergebnis 24

# Oops, Strich- vor Punktrechnung...

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

Eingabe 2 + 3 \* 3 =  
Ergebnis 15

# Analyse des Problems

Eingabe:

13 + ...

# Analyse des Problems

Eingabe:

$$13 + 4 * \dots$$

# Analyse des Problems

Eingabe:

$$13 + 4 * (15 - ...$$

# Analyse des Problems

Eingabe:

$$13 + 4 * (15 - 7 * ...$$

# Analyse des Problems

Eingabe:

$$13 + 4 * (15 - 7 * 3) =$$

Muss gespeichert bleiben,  
damit jetzt ausgewertet  
werden kann!

# Analyse des Problems

Ergebnis:

$$13 + 4*(15 - 21)$$

# Analyse des Problems

Ergebnis:

$$13 + 4 * (-6)$$

# Analyse des Problems

Ergebnis:

$$13 + (-24)$$

# Analyse des Problems

Ergebnis:

-11

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt recht

# Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt recht rekursiv.

# Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

# Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

# Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

# Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

**Wir brauchen ein neues formales (von C++ unabhängiges) Handwerkszeug.**

# Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

# Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

# Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Zur Beschreibung der Grammatik verwenden wir:

**Extended Backus Naur Form (EBNF)**

## What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth  
Federal Institute of Technology (ETH), Zürich, and  
Xerox Palo Alto Research Center

**Key Words and Phrases:** syntactic description  
language, extended BNF  
**CR Categories:** 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietary circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasympols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as (empty) or  $\epsilon$ ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax      = {production}.
production  = identifier "=" expression " ".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | literal | "(" expression ")" |
              "[" expression "]" | "{" expression "}".
literal     = " " " " character {character} " " " " .
```

Repetition is denoted by curly brackets, i.e. {a} stands for  $\epsilon$  | a | aa | aaa | . . . . Optionality is expressed by square brackets, i.e. [a] stands for  $\epsilon$  | a. Parentheses merely serve for grouping, e.g. (a|b|c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Received January 1977; revised February 1977

# Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer

2

# Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer , oder
- eine Ziffer gefolgt von einer Folge von Ziffern

2

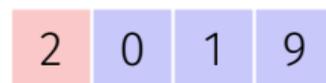
2 0 1 9

# Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer , oder 2

■ eine Ziffer gefolgt von einer Folge von Ziffern



`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

# Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer , oder
- eine Ziffer gefolgt von einer Folge von Ziffern

2

2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

Alternative

Terminales Symbol

Nicht-terminales Symbol

# Zahl (nicht-rekursiv)

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer, oder
- eine Ziffer gefolgt von **beliebig vielen Ziffern**

2

2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit { digit } .`

# Zahl (nicht-rekursiv)

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer, oder
- eine Ziffer gefolgt von **beliebig vielen Ziffern**

2

2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit { digit } .`

**Optionale Repetition**

$$-(3-(4-5))*(3+4*5)/6$$

Was benötigen wir in einer Grammatik?

# Ausdrücke

$$-(\underline{3} - (\underline{4} - \underline{5})) * (\underline{3} + \underline{4} * \underline{5}) / \underline{6}$$

Was benötigen wir in einer Grammatik?

- Zahl

# Ausdrücke

$$- \underline{(3 - (4 - 5))} * \underline{(3 + 4 * 5)} / \underline{6}$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )

$$\underline{-} (3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )
- Zahl, -( ? )

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- ? \* ?, ? / ?, ...

$$-(3-(4-5)) * (3+4*5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- ? \* ?, ? / ?, ...
- ? - ?, ? + ?, ...

$$-(3-(4-5)) * (3+4*5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- ? \* ?, ? / ?, ...
- ? - ?, ? + ?, ...

Faktor

$$-(3-(4-5)) * (3+4*5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor,  
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor,  
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor, Faktor  
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor, Faktor  
Faktor / Faktor, ...
- Term + Term,  
Term - Term, ...

Faktor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor, Faktor  
Faktor / Faktor, ...
- Term + Term,  
Term - Term, ...

Faktor

Term

Ausdruck

$$-(3-(4-5))*(3+4*5)/6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( ? )  
-Zahl, -( ? )
- Faktor \* Faktor, Faktor  
Faktor / Faktor, ...
- Term + Term, **Term**  
Term - Term, ...

Faktor

Term

Ausdruck

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, ( Ausdruck )  
-Zahl, -( Ausdruck )
- Faktor \* Faktor, Faktor  
Faktor / Faktor, ...
- Term + Term, Term  
Term - Term, ...

Faktor

Term

Ausdruck

# Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,

# Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck

# Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck

# Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor.
```

# Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor.
```

**Nicht-terminales Symbol**

**Terminales Symbol**

**Alternative**

# Die EBNF für Ausdrücke

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor .
```

Implikation: Ein Faktor beginnt mit

- einer Ziffer, oder
- mit „(“ , oder
- mit „-“.

# Die EBNF für Ausdrücke

Ein Term ist

- Faktor,

# Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor \* Faktor, Faktor / Faktor,

# Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor \* Faktor, Faktor / Faktor,
- Faktor \* Faktor \* Faktor, Faktor / Faktor \* Faktor, ...
- ...

# Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor \* Faktor, Faktor / Faktor,
- Faktor \* Faktor \* Faktor, Faktor / Faktor \* Faktor, ...
- ...

term = factor { "\*" factor | "/" factor }.

# Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor \* Faktor, Faktor / Faktor,
- Faktor \* Faktor \* Faktor, Faktor / Faktor \* Faktor, ...
- ...

term = factor { "\*" factor | "/" factor } .

**Optionale Repetition**

# Die EBNF für Ausdrücke

factor = unsigned\_number  
| "(" expression ")"  
| "-" factor.

term = factor { "\*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.

# Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der EBNF kann (fast) automatisch ein Parser generiert werden:
  - Regeln werden zu Funktionen
  - Alternativen und Optionen werden zu **if**-Anweisungen
  - Nichtterminale Symbole auf der rechten Seite werden zu Funktionsaufrufen
  - Optionale Repetitionen werden zu **while**-Anweisungen

# Regeln

factor = unsigned\_number  
| "(" expression ")"  
| "-" factor.

term = factor { "\*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

Ausdruck wird aus einem [Eingabestrom](#) gelesen.

```
// POST: returns true if and only if in_stream = factor ...  
//       and in this case extracts factor from in_stream
```

```
bool factor (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = term ...,  
//       and in this case extracts all factors from in_stream
```

```
bool term (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = expression ...,  
//       and in this case extracts all terms from in_stream
```

```
bool expression (std::istream& in_stream);
```

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: extracts a factor from in_stream  
//       and returns its value  
double factor (std::istream& in_stream);
```

```
// POST: extracts a term from in_stream  
//       and returns its value  
double term (std::istream& in_stream);
```

```
// POST: extracts an expression from in_stream  
//       and returns its value  
double expression (std::istream& in_stream);
```

## Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: the next character at the stream is returned  
//       without being consumed. returns 0 if stream ends.
```

```
char peek (std::istream& input){  
    if (input.eof()) return 0; // end of stream  
    return input.peek(); // next character in input  
}
```

```
// POST: leading whitespace characters are extracted from input  
//       and the first non-whitespace character on input returned
```

```
char lookahead (std::istream& input) {  
    input >> std::ws; // skip whitespaces  
    return peek(input);  
}
```

# Zahlen parsen

```
bool isDigit(char ch){
    return ch >= '0' && ch <= '9';
}
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int unsigned_number (std::istream& input){
    char ch = lookahead(input);
    assert(isDigit(ch));
    unsigned int num = 0;
    while(isDigit(ch) && input >> ch){ // read remaining digits
        num = num * 10 + ch - '0';
        ch = peek(input);
    }
    return num;
}
unsigned_number =digit { digit }.
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.
```

# Rosinenpickerei

...um jeweils nur das gewünschte Zeichen zu extrahieren.

```
// POST: if expected matches the next lookahead then consume it
//       and return true; return false otherwise
bool consume (std::istream& in_stream, char expected)
{
    if (lookahead(in_stream) == expected){
        in_stream >> expected; // consume one character
        return true;
    }
    return false;
}
```

# Faktoren auswerten

```
double factor (std::istream& in_stream)
{
    double value;
    if (consume(in_stream, '(')) {
        value = expression (in_stream);
        consume(in_stream, ')');
    } else if (consume(in_stream, '-')) {
        value = -factor (in_stream);
    } else {
        value = unsigned_number(in_stream);
    }
    return value;
}
```

factor = "(" expression ")"  
| "-" factor  
| unsigned\_number.

# Terme auswerten

```
double term (std::istream& in_stream)
{
    double value = factor (in_stream);
    while(true){
        if (consume(in_stream, '*'))
            value *= factor(in_stream);
        else if (consume(in_stream, '/'))
            value /= factor(in_stream)
        else
            return value;
    }
}
```

term = factor { "\*" factor | "/" factor }.

# Ausdrücke auswerten

```
double expression (std::istream& in_stream)
{
    double value = term(in_stream);
    while(true){
        if (consume(in_stream, '+'))
            value += term (in_stream);
        else if (consume(in_stream, '-'))
            value -= term(in_stream)
        else
            return value;
    }
}
```

expression = term { "+" term | "-" term }.

# Rekursion!

Factor

Term

Expression

# Rekursion!

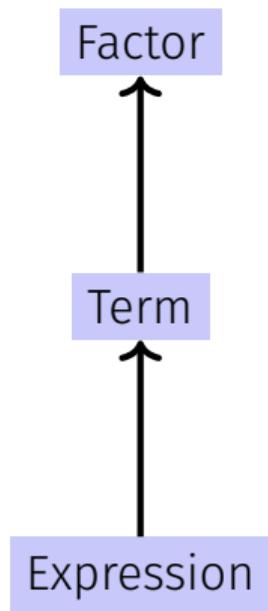
Factor

Term

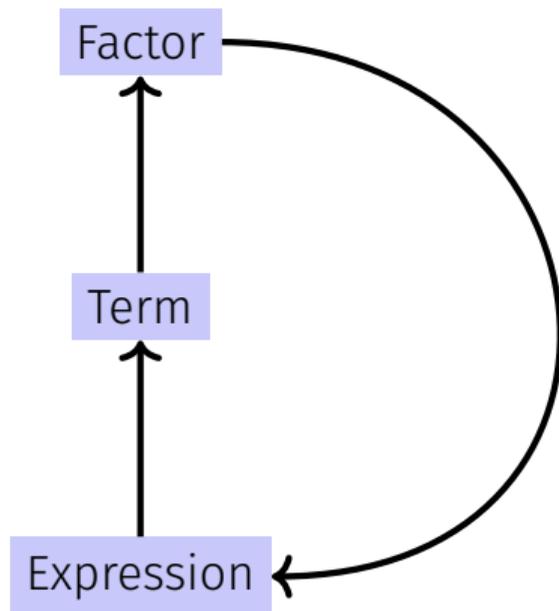
Expression



# Rekursion!



# Rekursion!



# EBNF – Und es funktioniert!

EBNF (calculator.cpp, Auswertung von links nach rechts):

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor.  
  
term        = factor { "*" factor | "/" factor }.  
  
expression = term { "+" term | "-" term }.
```

```
std::stringstream input ("1-2-3");  
std::cout << expression (input) << "\n"; // -4
```

# 18. Structs

---

Rationale Zahlen, Struct-Definition

# Rechnen mit rationalen Zahlen

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

# Rechnen mit rationalen Zahlen

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

## Ziel

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



# Vision

```
// 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;  
    int d; // INV: d != 0  
};
```

# Ein erstes Struct

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

Member-Variable (**n**umerator)

Member-Variable (**d**enominator)

# Ein erstes Struct

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

- **struct** definiert einen neuen **Typ**

# Ein erstes Struct

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

- **struct** definiert einen neuen **Typ**
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen

# Ein erstes Struct

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

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

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

# Eingabe

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

// Input s the same way
rational s;
...
```

## Vision in Reichweite ...

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

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\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: Initialisierung und Zuweisung

```
rational s; ← Member-Variablen uninitialisiert (wird sich bald ändern)
```

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

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialisierung und Zuweisung

```
rational s;
```

```
rational t = {1,5}; ← Memberweise Initialisierung:  
t.n = 1, t.d = 5
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialisierung und Zuweisung

```
rational s;
```

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

```
rational u = t; ← Memberweise Kopie
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialisierung und Zuweisung

```
rational s;
```

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

```
rational u = t;
```

```
t = u; ← Memberweise Kopie
```

```
rational v = add (u,t);
```

# Structs: Initialisierung und Zuweisung

```
rational s;
```

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

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t); ← Memberweise Kopie
```

# Structs vergleichen?

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

# 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,...

# 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.  $\frac{2}{3} \neq \frac{4}{6}$

# Benutzerdefinierte Operatoren

Statt

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

würden wir lieber

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

schreiben.

# Benutzerdefinierte Operatoren

Statt

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

würden wir lieber

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

schreiben.

Das geht mit *Überladen von Operatoren* ( $\rightarrow$  nächste Woche).