1. Structs und Referenztypen

Rationale Zahlen, Struct-Definition, Referenztypen, Operator-Überladung, Temporäre Objekte, Const-Referenzen

Rechnen mit rationalen Zahlen

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

Rechnen mit rationalen Zahlen

- Rationale Zahlen ($\mathbb{Q}$) sind von der Form 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! 🙂

Rechnen mit rationalen Zahlen

So könnte (wird) es aussehen

```
// Program: userational2.cpp
// Add two rational numbers.
#include <iostream>
#include “rational.cpp”
int main ()
{
    // 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";
    return 0;
}
```

Ein erstes Struct

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

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

Ein erstes Struct

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

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

Invariante: spezifiziert gültige Kombinationen von Werten (informell). Namen: `n` steht für *numerator* (Zähler); `d` für *denominator* (Nenner)

Ein **`struct`** definiert einen neuen Typ, dessen Wertebereich das *kartesische Produkt* der Wertebereiche existierender Typen ist; hier: **`int`** $\times$ **`int`**.

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

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

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.

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

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

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.

```
// 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 erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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.

Mitglieds-Zugriff auf die **`int`**-Objekte von **`a`**.

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

$$a + b = \texttt{result}$$
$$= \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

Der neue Typ kann wie existierende Typen benutzt werden, z.B. in

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

Der neue Typ kann wie existierende Typen benutzt werden, z.B. in Variablendeklarationen

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

Der neue Typ kann wie existierende Typen benutzt werden, z.B. in
formalen Argumentlisten

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.cpp
// Add two rational numbers.
#include <iostream>

// the 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;
}
```

Der neue Typ kann wie existierende Typen benutzt werden, z.B. in Rückgabetypen ...

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

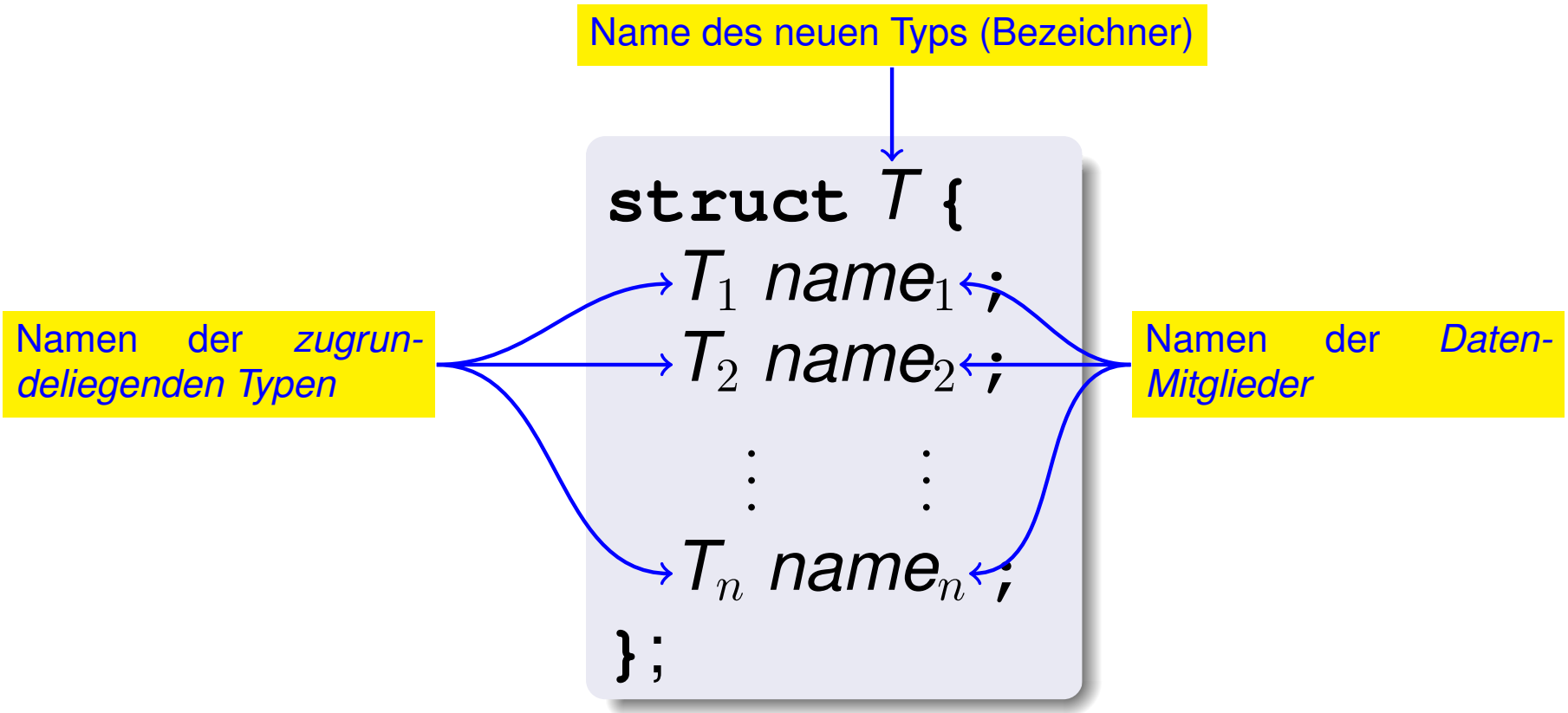
Ein erstes Struct: Benutzung

```
int main ()
{
    // input
    std::cout << "Rational number r:\n";
    rational r;
    std::cout << " numerator =? "; std::cin >> r.n;
    std::cout << " denominator =? "; std::cin >> r.d;

    std::cout << "Rational number s:\n";
    rational s;
    std::cout << " numerator =? "; std::cin >> s.n;
    std::cout << " denominator =? "; std::cin >> s.d;

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

    // output
    std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
    return 0;
}
```

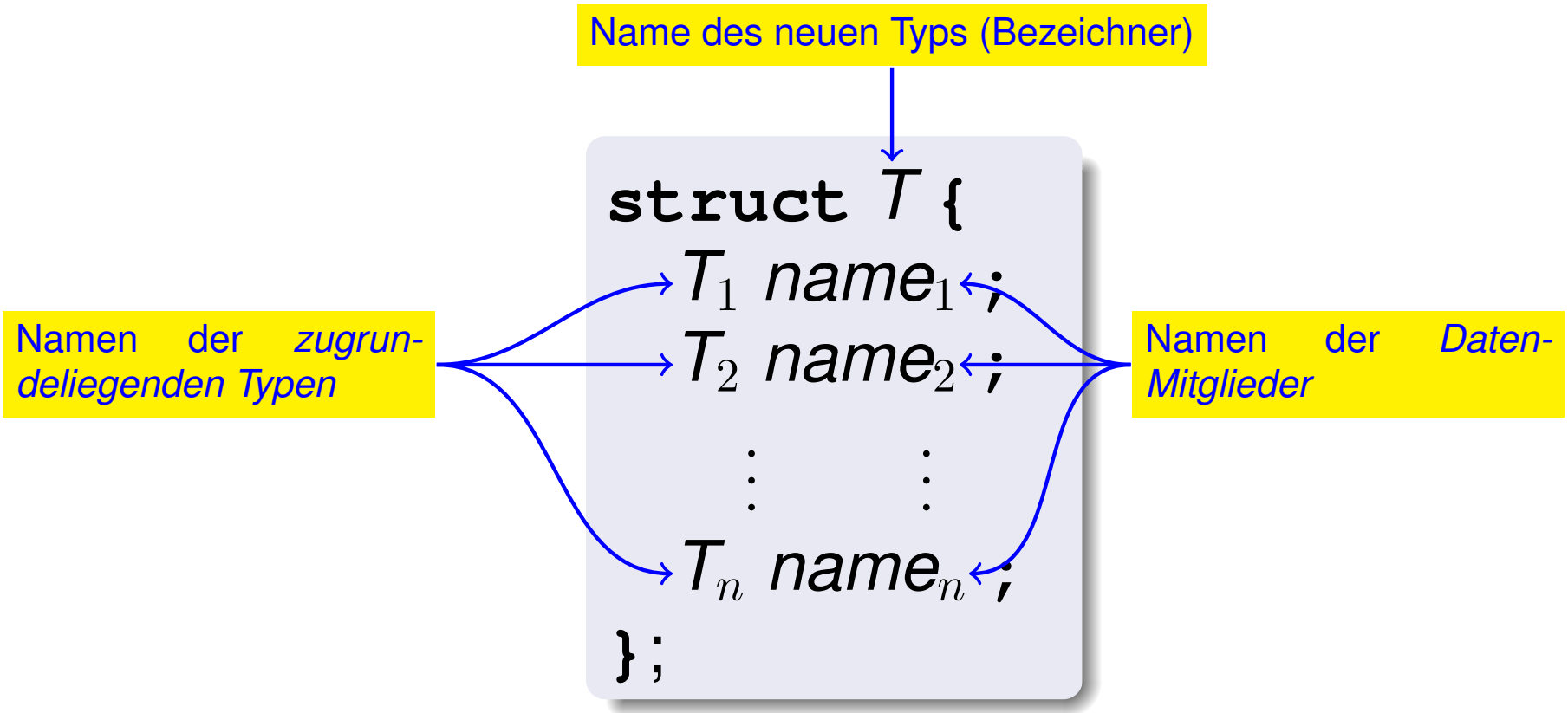
Struct-Definitionen

```
struct T {
    T_1 name_1 ;
    T_2 name_2 ;
      ⋮     ⋮
    T_n name_n ;
};
```

Struct-Definitionen



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;
}
```

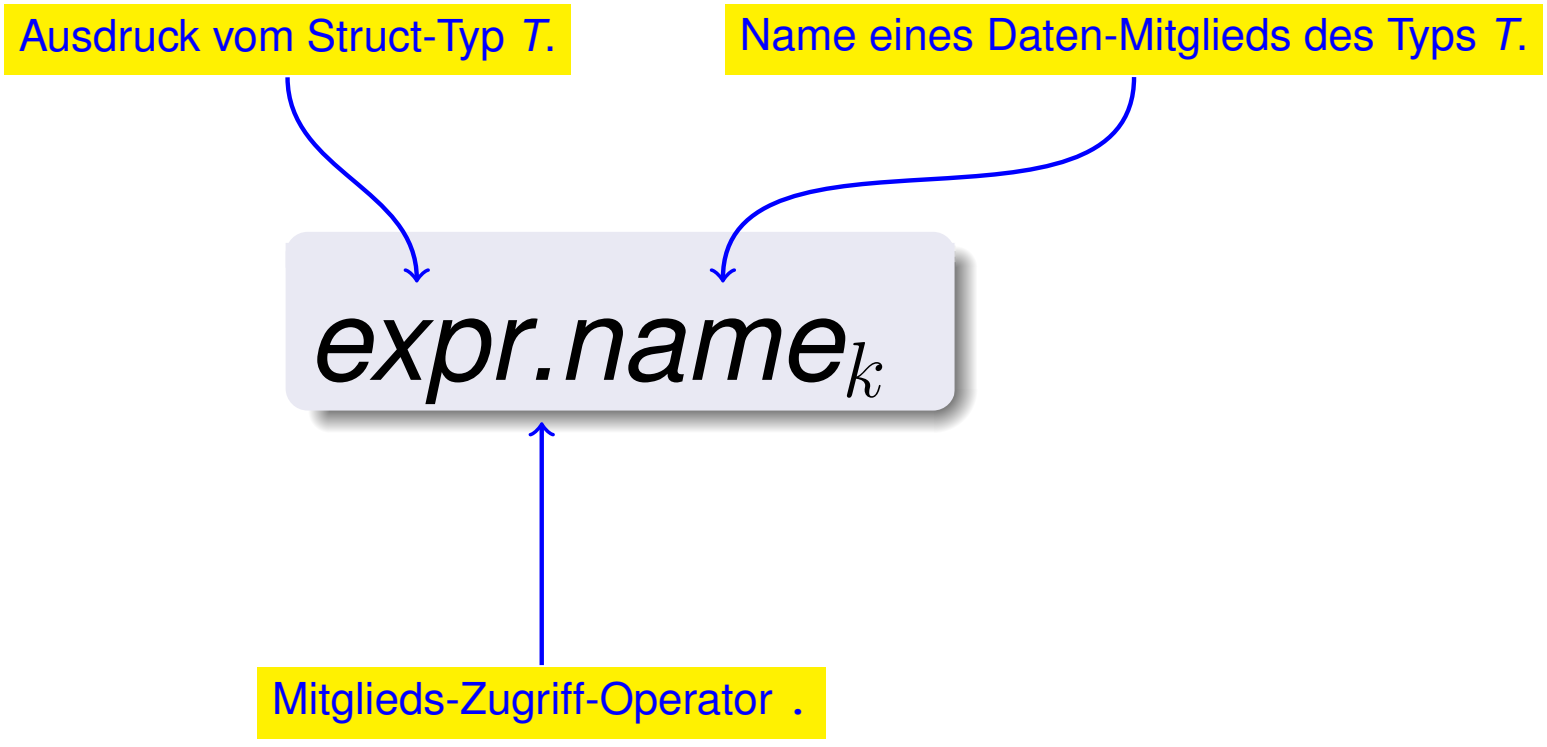
Die zugrundeliegenden Typen können fundamentale, aber auch benutzerdefinierte Typen sein.

Struct-Definitionen: Beispiele

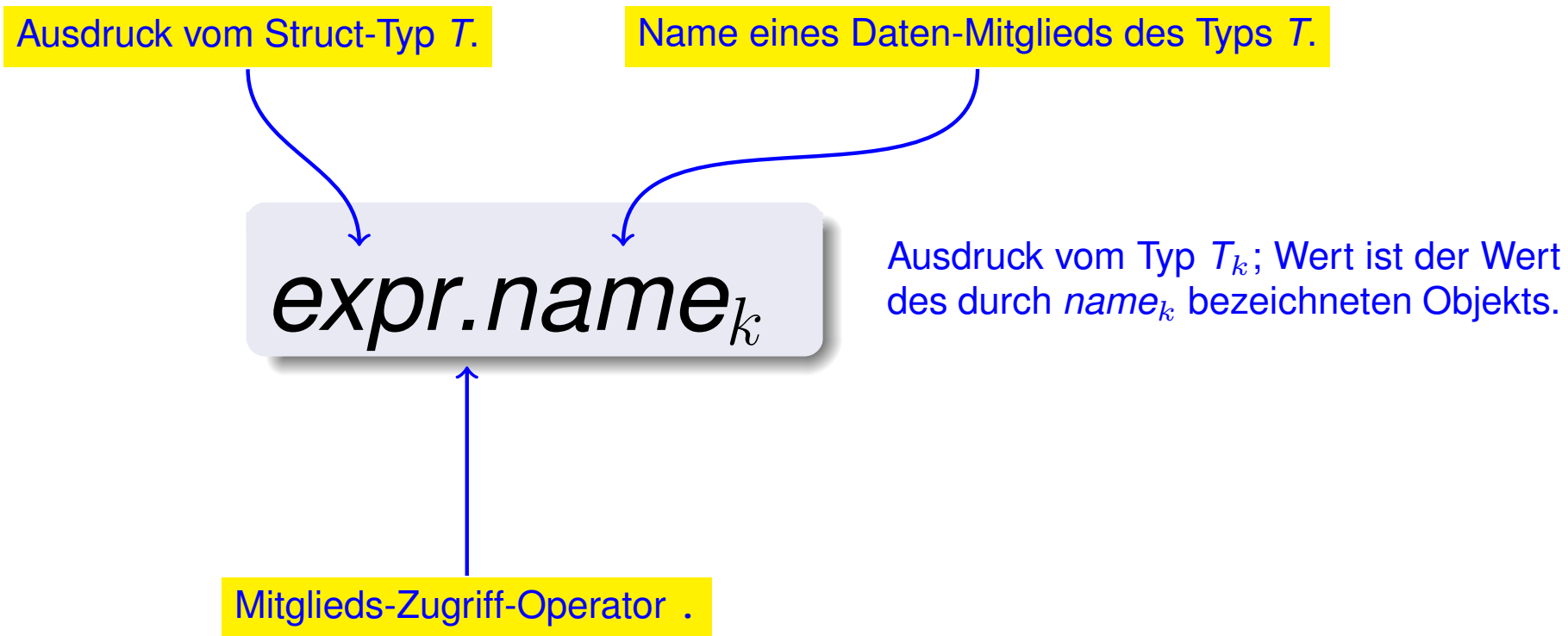
```
struct extended_int {
    // represents value if negative==false
    // and -value otherwise
    unsigned int value;
    bool negative;
}
```

Die zugrundeliegenden Typen können natürlich auch verschieden sein.

Structs: Mitglieds-Zugriff



Structs: Mitglieds-Zugriff



Structs: Initialisierung und Zuweisung

Initialisierung:

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

Structs: Initialisierung und Zuweisung

Initialisierung:

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

- `t` wird mit dem Wert von `add(r, s)` initialisiert
- Initialisierung erfolgt separat für jedes Daten-Mitglied

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
- Initialisierung erfolgt separat für jedes Daten-Mitglied

Structs: Initialisierung und Zuweisung

Daten-Initialisierung:

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

- Daten-Mitglieder von `t` werden mit den Werten der Liste, entsprechend der Deklarationsreihenfolge, initialisiert.

Structs: Initialisierung und Zuweisung

Default-Initialisierung:

```
rational t;
```

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

Structs: Initialisierung und Zuweisung

Zuweisung:

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

- `t` wird default-initialisiert
- Der Wert von `add(r, s)` wird `t` zugewiesen (wieder separat für jedes Daten-Mitglied)

Structs und Felder

- Felder können auch Mitglieder von Structs sein

Structs und Felder

- Felder können auch Mitglieder von Structs sein

```
struct rational_vector_3 {
    rational v[3];
};
```

Structs und Felder

- Felder können auch Mitglieder von Structs sein

```
struct rational_vector_3 {
    rational v[3];
};
```

- Durch Verpacken in ein Struct kann das Kopieren von Feldern erreicht werden!

Structs und Felder

```
#include<iostream>

struct point {
    double coord[2];
};

int main()
{
    point p;
    p.coord[0] = 1;
    p.coord[1] = 2;
    const point q = p; //Hier wird ein Feld mit zwei Elementen kopiert
    std::cout << q.coord[0] << " "   // 1
              << q.coord[1] << "\n"; // 2
    return 0;
}
```

Structs Gleichheitstests?

Für jeden fundamentalen Typ gibt es die Vergleichsoperatoren `==` und `!=` , aber nicht für Structs! Warum?

Structs Gleichheitstests?

Für jeden fundamentalen Typ gibt es die Vergleichsoperatoren `==` und `!=` , aber nicht für Structs! Warum?

- Mitgliedsweiser Vergleich ergibt im allgemeinen keinen Sinn,...

Structs Gleichheitstests?

Für jeden fundamentalen Typ gibt es die Vergleichsoperatoren `==` und `!=` , aber nicht für Structs! Warum?

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

Structs als Funktionsargumente

Auch Ausdrücke vom Struct-Typ werden als R-Wert an eine Funktion übergeben.

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

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

Structs als Funktionsargumente

Auch Ausdrücke vom Struct-Typ werden als R-Wert an eine Funktion übergeben.

```
void increment(rational dest, const rational src)
{
    dest = Add(dest, src); // verändert nur die lokale Kopie
}

int main()
{
    rational a;
    rational b;
    a.d = 1; a.n = 2;
    b = a;
    increment(b, a); // hat keinen Effekt
    std::cout << b.n << "/" << b.d;  // 1/2
}
```

Für mutierende Funktionen mussten wir bisher mit Zeigern oder Iteratoren arbeiten.

Structs als Funktionsargumente

- Um den Inhalt eines Structs zu ändern, könnten wir wieder Zeiger verwenden
- Das hat aber Nachteile:

Structs als Funktionsargumente

- Um den Inhalt eines Structs zu ändern, könnten wir wieder Zeiger verwenden
- Das hat aber Nachteile:
  - Die Deklaration und Verwendung in der Funktion wird komplizierter

```
void increment(rational* dest, const rational src)
{
    *dest = Add(*dest, src);
}
```

Structs als Funktionsargumente

- Um den Inhalt eines Structs zu ändern, könnten wir wieder Zeiger verwenden
- Das hat aber Nachteile:
  - Die Deklaration und Verwendung in der Funktion wird komplizierter

```
void increment(rational* dest, const rational src)
{
    *dest = Add(*dest, src);
}
```

  - Der Aufruf wird komplizierter und unsicherer:

```
increment(ptr, a); // ptr muss gueltige Adresse sein
```

Structs als Funktionsargumente

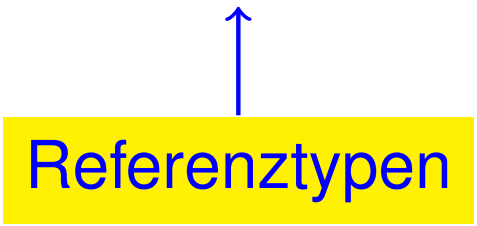
- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!

Structs als Funktionsargumente

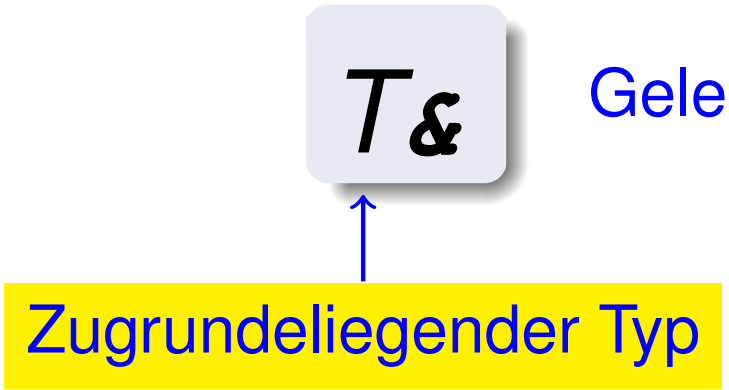
- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Interessanterweise benötigen wir kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen.

Structs als Funktionsargumente

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Interessanterweise benötigen wir kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen.



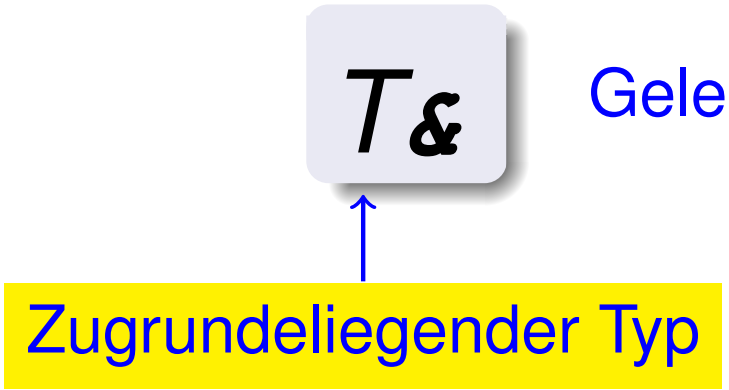
Referenztypen: Definition



Referenztypen: Definition

- *T*`&` hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...

Referenztypen: Definition



- *T*`&` hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...
- ..., nur die Initialisierungs- und Zuweisungssemantik ist anders.

Referenztypen: Initialisierung

- Eine Variable mit Referenztyp (eine *Referenz*) kann nur mit einem L-Wert initialisiert werden
- Die Variable wird dabei ein *Alias* des L-Werts (ein anderer Name für das referenzierte Objekt)

Referenztypen: Initialisierung und Zuweisung

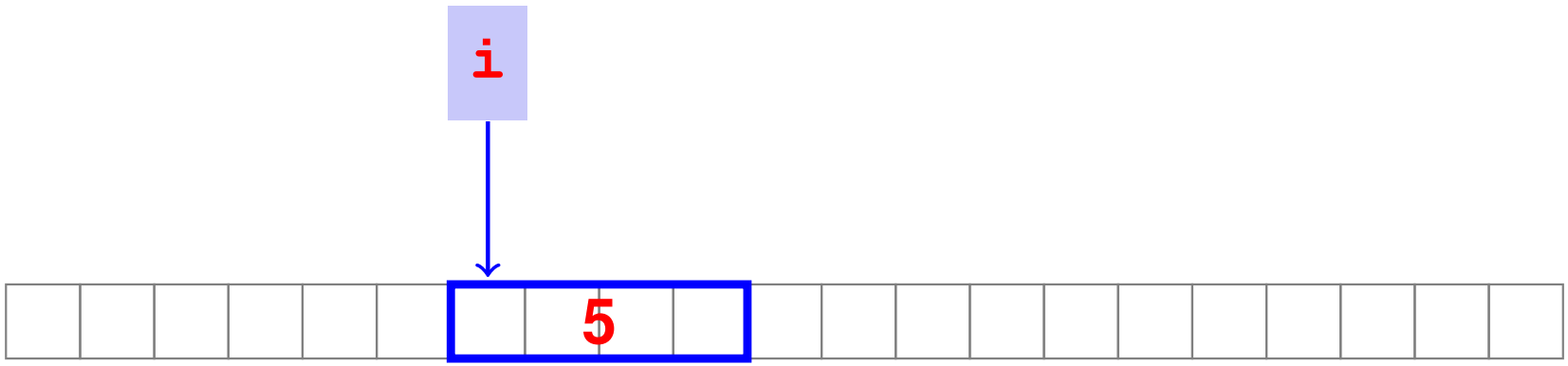
Beispiel:

```
int i = 5;
int& j = i; // j becomes an alias of i

j = 6;      // changes the value of i
std::cout << i << "\n"; // outputs 6
```

Referenztypen: Initialisierung und Zuweisung

Beispiel:

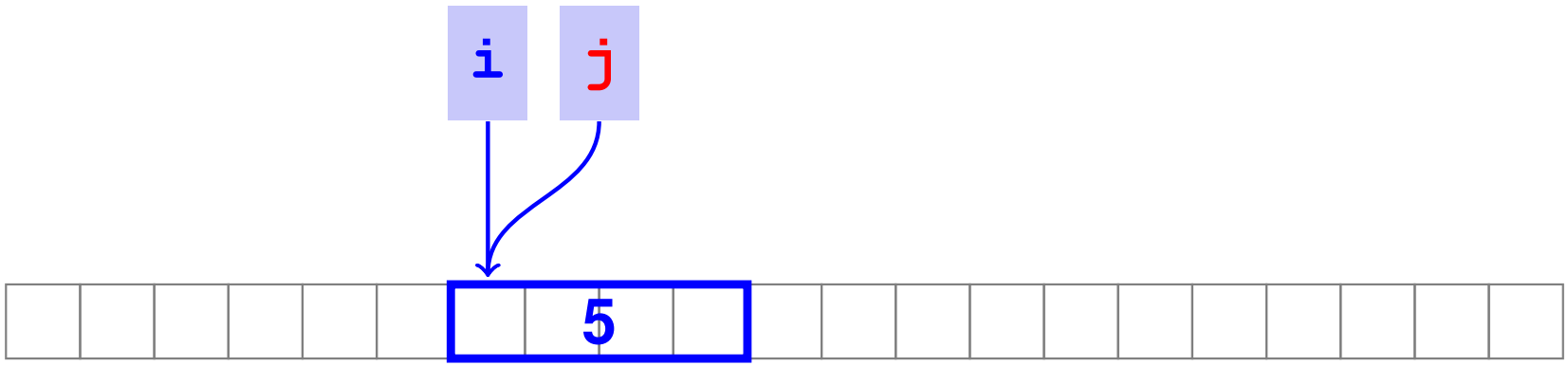
```
int i = 5;
int& j = i; // j becomes an alias of i

j = 6;      // changes the value of i
std::cout << i << "\n"; // outputs 6
```



Referenztypen: Initialisierung und Zuweisung

Beispiel:

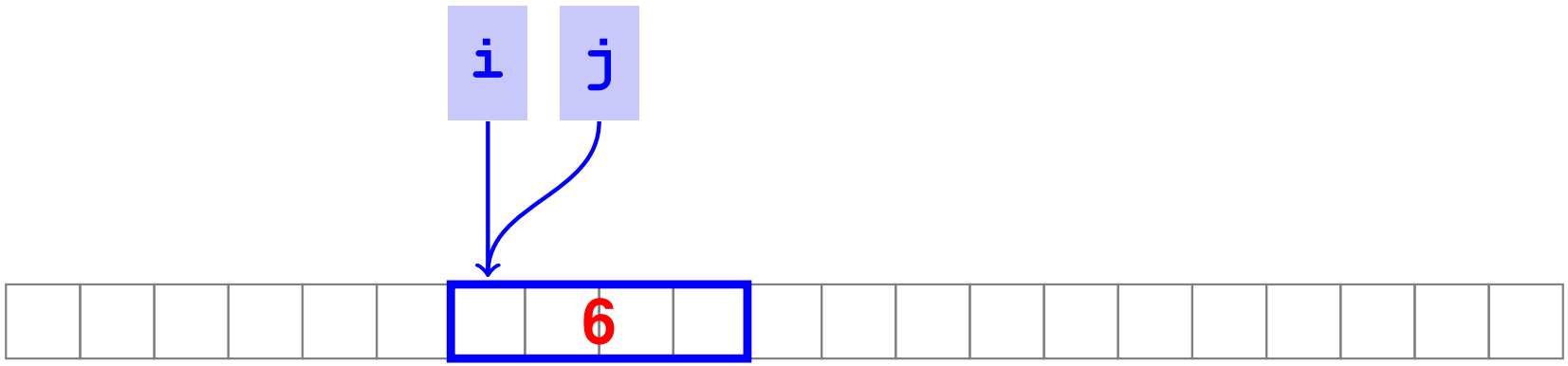
```
int i = 5;
int& j = i; // j becomes an alias of i

j = 6;      // changes the value of i
std::cout << i << "\n"; // outputs 6
```



Referenztypen: Initialisierung und Zuweisung

Beispiel:

```
int i = 5;
int& j = i; // j becomes an alias of i

j = 6;          // changes the value of i
std::cout << i << "\n"; // outputs 6
```



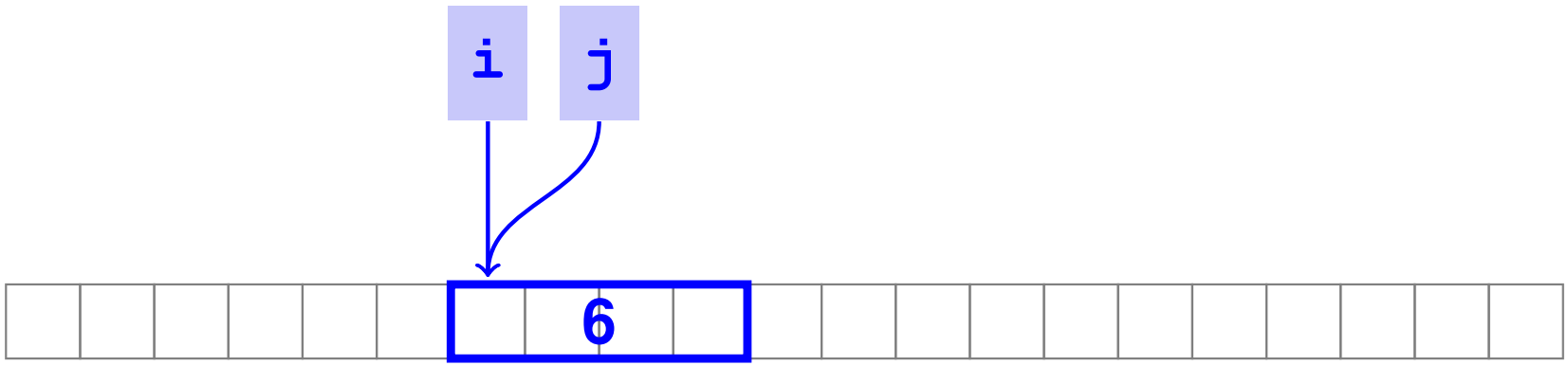
Referenztypen: Initialisierung und Zuweisung

Beispiel:

```
int i = 5;
int& j = i; // j becomes an alias of i
```

Zuweisung erfolgt an das Objekt hinter der Referenz.

```
j = 6;          // changes the value of i
std::cout << i << "\n"; // outputs 6
```



Referenzen sind implizte Zeiger ...

... aber sicherer und einfacher:

Mit Referenzen

```
int i = 5;
int& j = i;

j = 6;
std::cout << i
          << "\n";
std::cout << j
          << "\n";
```

Mit Zeigern

```
int i = 5;
int* j = &i;

*j = 6;
std::cout << i
          << "\n";
std::cout << *j
          << "\n";
```

Referenztypen: Realisierung

Intern wird ein Wert vom Typ *T&* durch die Adresse eines Objekts vom Typ *T* repräsentiert.

```
int& j;    // error: j must be an alias of something
int& k= 5; // error: the literal 5 has no address
```

Call by Reference

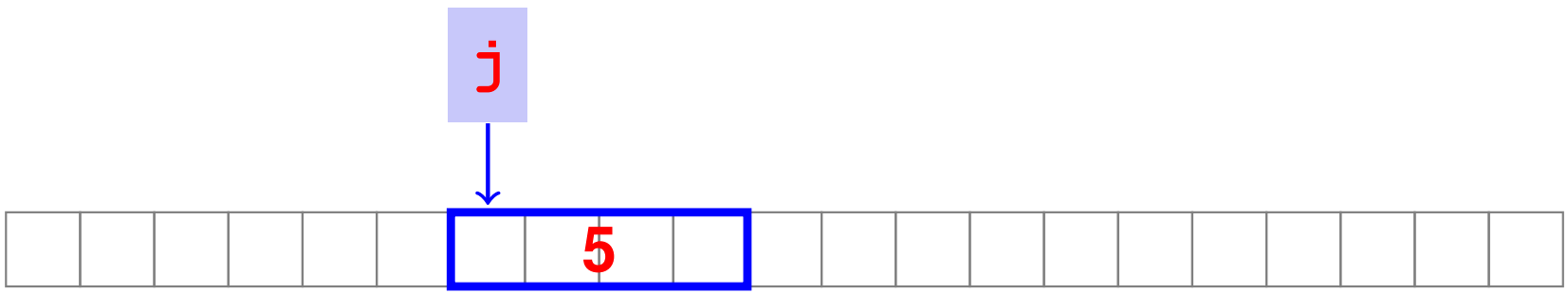
Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i)
{
    ++i;
}

int main ()
{
    int j = 5;
    increment (j);
    std::cout << j << "\n"; // outputs 6
    return 0;
}
```

Call by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i)
{
    ++i;
}

int main ()
{
    int j = 5;
    increment (j);
    std::cout << j << "\n"; // outputs 6
    return 0;
}
```

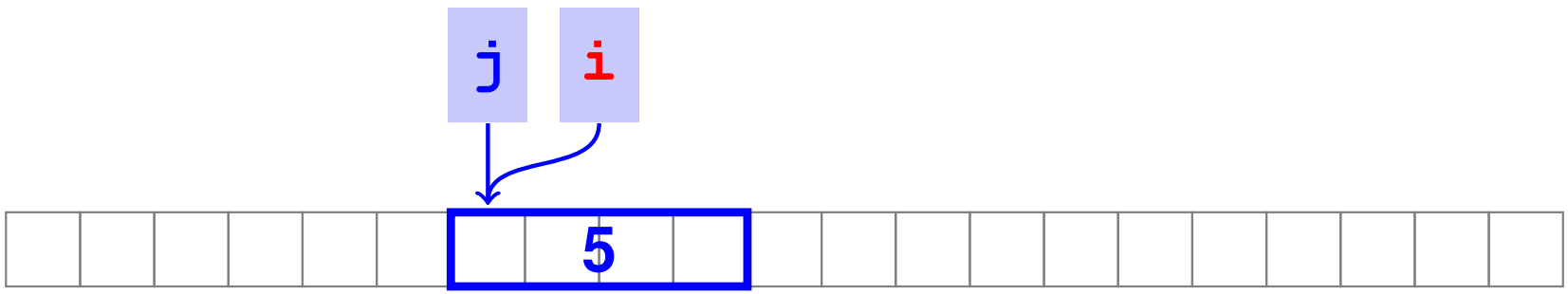


Call by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i)
{ // i becomes alias of call argument
    ++i;
}

int main ()
{
    int j = 5;
    increment (j);
    std::cout << j << "\n"; // outputs 6
    return 0;
}
```

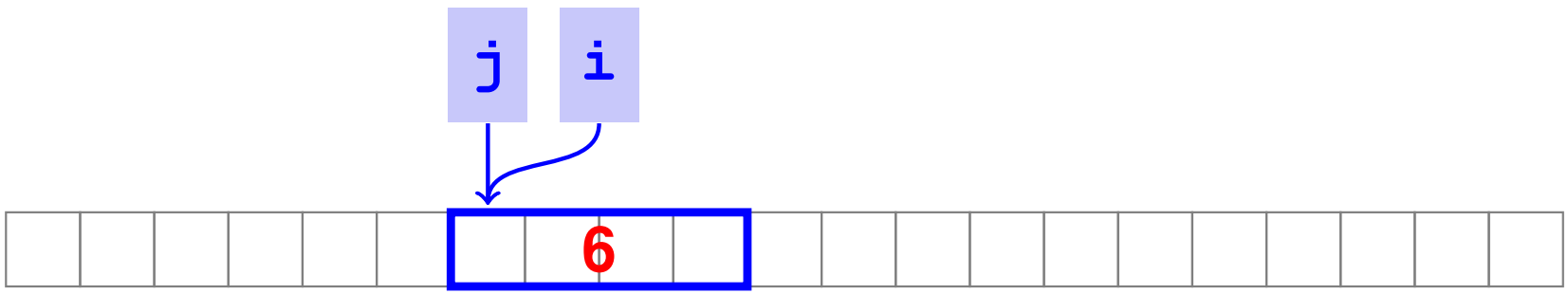
← Initialisierung der formalen Argumente.



Call by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i)
{
    ++i;
}

int main ()
{
    int j = 5;
    increment (j);
    std::cout << j << "\n"; // outputs 6
    return 0;
}
```



Call by Value / Call by Reference

- formales Argument hat Referenztyp:
  $\Rightarrow$ **Call by Reference**

formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*

Call by Value / Call by Reference

- formales Argument hat Referenztyp:
  $\Rightarrow$ **Call by Reference**

  formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*

- formales Argument hat keinen Referenztyp:
  $\Rightarrow$ **Call by Value**

  formales Argument wird mit dem *Wert* des Aufrufarguments (R-Wert) initialisiert und wird damit zu einer *Kopie*

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst einen L-Wert

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst einen L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst einen L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```

Exakt die Semantik des Prä-Inkrements

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.

Dies geht mit *Operator-Überladung*.

Funktions- und Operator-Überladung

Zur Erinnerung: Verschiedene Funktionen können den gleichen Namen haben.

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

// POST: returns a * a
extended_int square (extended_int 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

Additionsoperator für rationale Zahlen

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);
```

Additionsoperator für rationale Zahlen

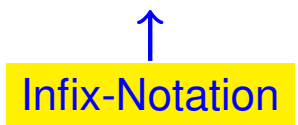
Neu:

```
// 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;
```

Additionsoperator für rationale Zahlen

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;
```



Additionsoperator für rationale Zahlen

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 = operator+ (r, s);
```

Äquivalent, aber unpraktisch: funktionale Notation.

Andere binäre Operatoren für rationale Zahlen

```
// POST: return value is the 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;
}
```

Relationale Operatoren

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;
}
```

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 << "\n"; // 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?

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`**.

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!

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` inkrementiert 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.

Das kann wie folgt erreicht werden

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

Ein-/Ausgabeoperatoren

können auch überladen werden.

Das kann wie folgt erreicht werden

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

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

Ein-/Ausgabeoperatoren

können auch überladen werden.

```
// PRE: i starts with a rational number
// of the form "n/d"
// POST: r has been read from i

std::istream& operator>> (std::istream& i,
                          rational& r)
{
    char c; // separating character '/'
    return i >> r.n >> c >> r.d;
}
```

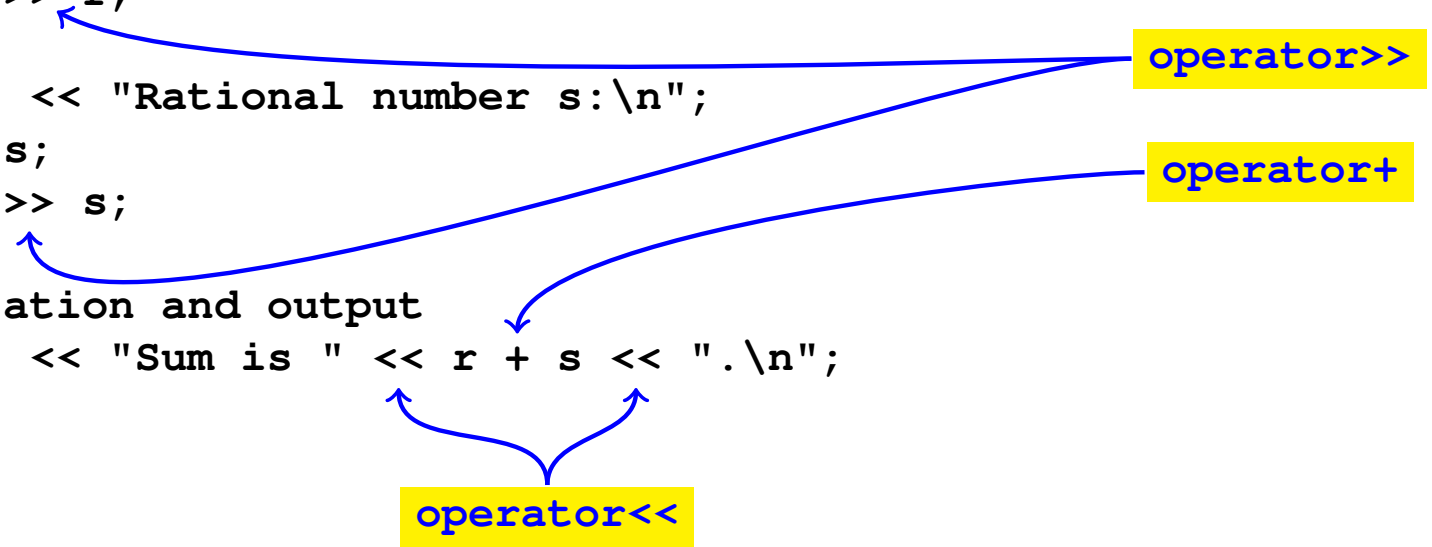
liest `r` aus dem Eingabestrom `i`
und gibt diesen als L-Wert zurück

Zwischenziel erreicht!

```
int main ()
{
    // 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";
    return 0;
}
```

Zwischenziel erreicht!

```
int main ()
{
    // 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";
    return 0;
}
```



Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Argu-ments, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Aufrufstapel 

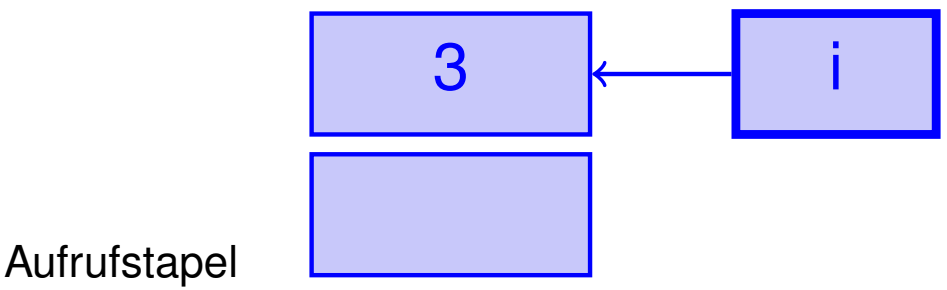
```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

Wert des Aufrufarguments kommt auf den Aufrufstapel

```
int& foo (int i)
{
    return i;
}
```



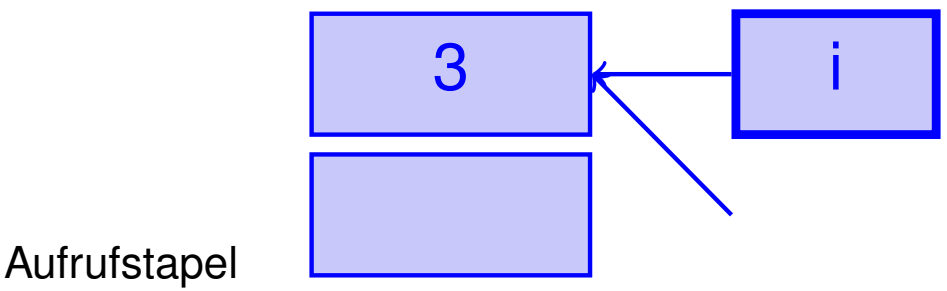

```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

`i` wird als Referenz zurück-gegeben

```
int& foo (int i)
{
    return i;
}
```



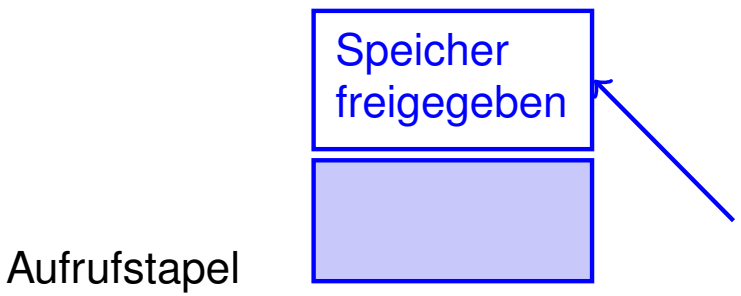

```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

... und verschwindet vom Aufrufstapel




```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

`j` wird mit Rückgaberef-erenz initialisiert

```
int& foo (int i)
{
    return i;
}
```


```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Wert von `j` wird ausgegeben


```
int k = 3;
int& j = foo(k); // j refers to expired object
std::cout << j << "\n";  // undefined behavior
```

Die Referenz-Richtlinie

Referenz-Richtlinie

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben” wie die Referenz selbst.

Const-Referenzen

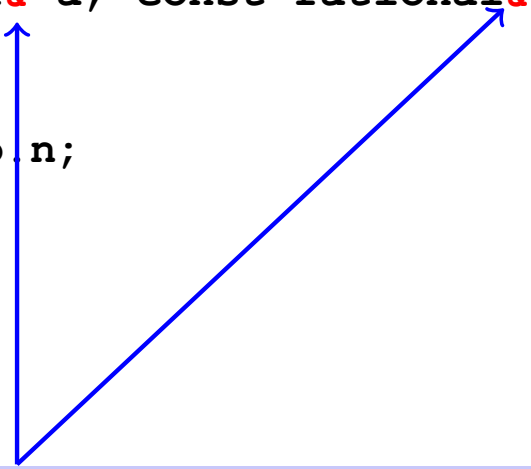
Vorschlag des Effizienzfanatikers

```
// 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-Referenzen

Vorschlag des Effizienzfanatikers

```
// 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;
}
```

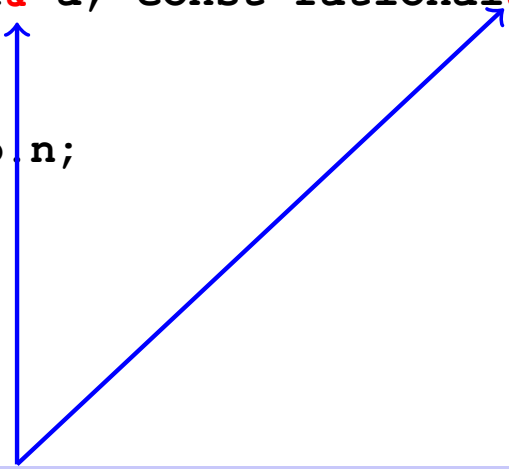


Beim Aufruf muss jeweils nur eine Addresse kopiert werden statt zwei ints (Einsparung wird dramatisch bei grossen Structs)

Const-Referenzen

Vorschlag des Effizienzfanatikers

```
// 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;
}
```

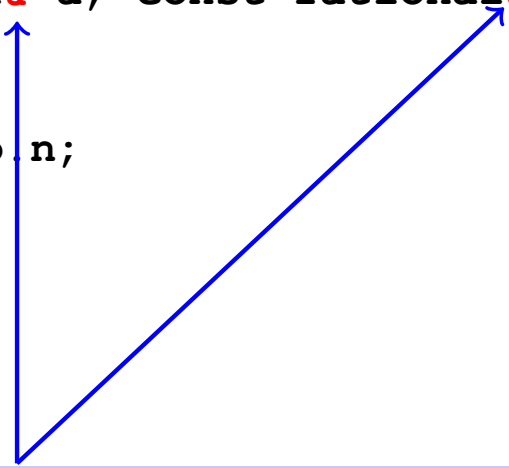


Scheinbarer Nachteil: Operator kann nur mit L-Werten aufgerufen werden.

Const-Referenzen

Vorschlag des Effizienzfanatikers

```
// 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;
}
```



Scheinbarer Nachteil: Operator kann nur mit L-Werten aufgerufen werden. **Die Wahrheit:** *für const-Referenzen gehen auch R-Werte*

Was genau ist konstant?

Betrachte L-Wert vom Typ **`const`** *T* (R-Werte sind ohnehin konstant)

- Fall 1: *T* ist kein Referenztyp

Dann verweist der L-Wert auf ein konstantes Objekt

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

Was genau ist konstant?

Betrachte L-Wert vom Typ `const` *T* (R-Werte sind ohnehin konstant)

- Fall 1: *T* ist kein Referenztyp

Dann verweist der L-Wert auf ein konstantes Objekt

```
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* (R-Werte sind ohnehin konstant)

- Fall 2: *T* ist Referenztyp

  Dann verweist der L-Wert auf ein Objekt, dessen Wert zwar veränderbar ist, aber nicht durch diesen Alias.

Was genau ist konstant?

Betrachte L-Wert vom Typ **`const`** *T* (R-Werte sind ohnehin konstant)

- Fall 2: *T* ist Referenztyp

  Dann verweist der L-Wert auf ein Objekt, dessen Wert zwar veränderbar ist, aber nicht durch diesen Alias.

```
int n = 5;
const int& i = n;// i becomes a non-modifiable alias of n
int& j = n;      // j becomes a modifiable alias of n
i = 6;           // error: n is modified through const reference
j = 6;           // ok: n receives value 6
```

Const-Referenzen

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

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-Referenzen

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

`const` *T***`&`** **`r`** = *rvalue*;
r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)

`const T` vs. `const T&`

```
// 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;
}
```

Regel

`const` *T* kann als Argumenttyp immer durch **`const`** *T* **`&`** ersetzt werden; dies lohnt sich aber nicht für fundamentale Typen oder „kleine” Structs.

T vs. *T*&

```
// 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;
}
```

Regel

`const` *T* kann als Argumenttyp immer durch **`const`** *T* **`&`** ersetzt werden; dies lohnt sich aber nicht für fundamentale Typen oder „kleine" Structs.

Achtung!

T kann als Argumenttyp nicht immer durch *T*`&` ersetzt werden (Gegenbeispiel: unärer **`operator-`** für rationale Zahlen ändert dadurch sein Verhalten)