

Überladen von Funktionen

19. Klassen

Funktions- und Operatorüberladung, Datenkapselung, Klassen, Memberfunktionen, Konstruktoren

- 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

525

526

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, 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
```

527

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

528

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);
```

529

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

530

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);
```

531

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;
}
```

532

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;
}
```

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

533

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
```

534

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.

535

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.

536

Ein-/Ausgabeoperatoren

können auch überladen werden.

■ Bisher:

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

■ Neu (gewünscht):

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

537

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.

539

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

538

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 <<

540

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;
}
...
```

541

...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 <<`, ...)

542

Invarianten und Repräsentation

- Wollen für eine Datenstruktur Invarianten garantieren, ohne den Benutzercode mit Asserts zu übersähen.

Beispiel 1: Für jedes `Rational r` gilt stets $r.d \neq 0$.
Beispiel 2: Jedes `Rational r` ist stets gekürzt.

- Liefern dem Benutzer das „was“ und nicht das „wie“.

Vector `v`: `v.size()`, `v[3]`

- Die interne Darstellung sollte einfach anzupassen sein, ohne dass Benutzercode neu geschrieben werden muss.

Complex: Polarkoordinaten vs. kartesische Koordinaten

543

Idee der Datenkapselung (Information Hiding)

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

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

544

Klassen

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

Datenkapselung: public / private

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

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

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: wir können gar nichts mehr machen ...

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
```

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of this instance  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of this instance  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich

Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Gültigkeitsbereich von Mem-bern in einer Klasse ist die ganze Klasse, unabhängig von der Deklarationsreihenfolge

Memberfunktionen: Aufruf

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

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

549

const und Memberfunktionen

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

rational x;
x.set_numerator(10); // ok;
const rational y = x;
int n = y.numerator(); // ok;
y.set_numerator(10); // error;
```

Das const an einer Memberfunktion liefert das Versprechen, dass eine Instanz nicht über diese Funktion verändert wird.
const Objekte dürfen nur const Memberfunktionen aufrufen!

550

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
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 die Instanz **this**, verspricht also, dass das implizite Argument nicht im Wert verändert wird.
- **n** ist Abkürzung in der Memberfunktion für **this->n** (genaue Erklärung von „->“ nächste Woche)

551

This rational vs. dieser Bruch

So wäre es in etwa ...

```
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);
```

552

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.

553

Initialisierung? Constructoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den) ← Initialisierung der
                              Membervariablen
    {
        assert (den != 0); ← Funktionsrumpf.
    }
    ...
};
...
rational r = rational(2,3); // r = 2/3
```

554

Konstruktoren: Aufruf

- direkt

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

- indirekt (Kopie)

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

555

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r; // r = 0
// Abkürzung für rational r = rational();
⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!
```

556

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)

557

Objektinitialisierung in diesem Kurs: Hintergrund

- Ziel dieses Kurses ist nicht, Sie zu C++-Experten auszubilden, sondern Ihnen Programmiergrundlagen zu vermitteln, die allgemein relevant sind und sich – möglichst direkt – auf andere Sprachen übertragen lassen
- Wir verwenden daher drei Syntaxvarianten, jede in einer bestimmten Situation
- Dabei haben wir uns von folgenden, nicht immer gleichzeitig erreichbaren Zielen leiten lassen:
 1. Übertragbarkeit: Die Syntax sollte derer anderer populärer Sprachen, z.B. Python, Java, Go, JavaScript, möglichst ähneln
 2. Einheitlichkeit: Je weniger Syntaxen verwendet werden, desto geringer das Potenzial für Verwirrung
 3. Stabilität: Kleine Änderungen (z.B. Hinzufügen eines expliziten Konstruktors) sollte keine unerwarteten Probleme nachsichziehen (z.B. weil der Compiler nun bestimmte Konstruktoren nicht mehr erstellt)
 4. Zweckmässigkeit: Bestimmte Formen sind, in manchen Situationen, besonders intuitiv & praktisch
 5. Idiomatisch: Der resultierende Code sollte realistischer C++-Code sein

559

Objektinitialisierung in C++

- Aus technischen und historischen Gründen gibt es in C++ verschiedene Formen der Objektinitialisierung, die sich in Syntax und Semantik unterscheiden. Die Initialisierung eines neuen Objekts `t` vom Typ `T` kann daher potenziell wie folgt erfolgen:

- `T t(...);`
- `T t{...};`
- `T t = T(...);`
- `T t = T{...};`
- `T t = {...};`

- Wichtig: Bei allen Formen handelt es sich um Initialisierungen, nicht um Zuweisungen und `T t = ...;` ist nicht äquivalent zu `T t; t = ...;`
- Die Varianten mit `=` sind mit dynamischer Speicherallokation (`new`) kombinierbar
- Bei Interesse können Sie hier mehr erfahren:

- <https://en.cppreference.com/w/cpp/language/initialization>
- <https://isocpp.org/wiki/faq/cpp11-language#uniform-init>
- <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Res-list>
- <https://quuxplusone.github.io/blog/2019/02/18/knightmare-of-initialization>

558

Objektinitialisierung in diesem Kurs: Konventionen

1. Für **primitive Datentypen** (`int`, `bool`, `double`, ...) und Initialisierung mittels **Literalen** verwenden wir *Copy Initialisation*. Beispiele:

```
int i = 5;
std::string s = "I am literally a string, folks!";
```

2. Für die **komponentenweise** Initialisierung von Containern (z.B. `vector`; demnächst mehr) und simplen Datentypen mit ausschliesslich öffentlichen Membervariablen (z.B. `struct rational` auf Folie 506) nutzen wir *List Initialisation* bzw. *Aggregate Initialisation*. Beispiele:

```
std::vector<char> vowels = {'A', 'E', 'I', 'O', 'U'};
rational half = {1, 2};
```

560

Objektinitialisierung in diesem Kurs: Konventionen

3. In den **verbleibenden Fällen**, z.B. für `struct rational` mit privaten Membervariablen auf Folie 554, nutzen wir *Copy Initialisation* mittels expliziter Konstruktoraufrufe in Funktionsaufrufsyntax. Beispiele:

```
rational half = rational(1,2);
std::vector<int> empty = std::vector<int>(7, 0); // vector with seven zeroes

// 'auto' fits in nicely, and avoids having to repeat types
auto half = rational(1,2);
auto empty = std::vector<int>(7, 0);
```

561

Objektinitialisierung in diesem Kurs: Konventionen

4. **Ausnahme:** Initialisierung von Membervariablen in Konstruktord Definitionen (*member initialiser lists*). Beispiel:

```
class rational {
    ...
    rational(int n, int d): num(n), den(d) {...}
}
```

Keine der hier erlaubten Syntaxformen `t(...)` und `t{...}` – ist ideal und wir haben uns daher relativ beliebig für die runden Klammern entschieden.

562

Objektinitialisierung: Abschluss

Keine Panik!

- Sie müssen weder die Gründe für, noch die Unterschiede zwischen den einzelnen Initialisierungsformen kennen
- Auch die Namen der unterschiedlichen Initialisierungsformen müssen Sie nicht kennen
- Es reicht, wenn Sie Objektinitialisierung auf dem Niveau der Vorlesungsbeispiele und Übungsaufgaben verstehen
- Sie müssen sich nicht an unsere Konventionen halten und dürfen andere Formen der Objektinitialisierung nutzen

563

Alternative: Default-Konstruktor löschen

```
class rational
{
public:
    ...
    rational () = delete;
    ...
};
...
rational r; // error: use of deleted function 'rational::rational()'

```

⇒ Es gibt keine uninitialisierten Variablen vom Typ `rational` mehr!

564

Initialisierung "rational = int"?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {} ← Leerer Funktionsrumpf
    ...
};
...
rational r = rational(2); // Explizite Initialisierung mit 2
rational s = 2; // Implizite Konversion
```

565

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau einem Argument, das den Typ `int` nach `rational` konvertierbar sind. Damit wird `int` zu einem Typ, dessen Werte nach `rational` konvertierbar sind.

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

566

Benutzerdefinierte Konversionen

Wie kann man implizite Konversion von `rational` nach `double` realisieren?

- Problem: `double` ist kein Struct (keine Klasse), wir können dem Typ keinen Konstruktor „verpassen“ (gilt auch für alle anderen Zieltypen, die nicht „uns“ gehören)
- Lösung: wir bringen unserem Typ `rational` die Konversion nach `double` bei (als Member-Funktion):

```
struct rational{
    ...
    operator double () ← impliziter Rückgabetyt double
    {
        return double (n)/d;
    }
};

rational a(1,2);
double b = a; // implizite Konversion
```

567

Member-Definition: In-Class vs. Out-of-Class

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

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

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

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)
- So geht's auch.

568