

17. Klassen

Datenkapselung, Klassen, Memberfunktionen, Konstruktoren

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;
}
...
```

560

561

...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 <<`, ...)

562

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

563

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;
}
```

564

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;
};
```

565

Neue Version von RAT PACK[®]



Nichts geht mehr!

- Was ist denn das Problem?



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

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



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



566

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;
};
```

567

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[®] ist Geschichte...

Idee der Datenkapselung (Information Hiding)

- 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),...`

568

569

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

570

571

Datenkapselung: public/private

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

Gute Nachricht: r.d = 0 aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen ...

Anwendungscode:

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

```
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

572

573

Memberfunktionen: Aufruf

```
// Definition des Typs  
class rational {  
    ...  
};  
...  
// Variable des Typs  
rational r;  
  
int n = r.numerator(); // Zaehler  
int d = r.denominator(); // Nenner
```

Member-Zugriff

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)

574

575

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!

576

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);
```

577

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

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

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

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

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

- So geht's auch.

578

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.

579

Initialisierung? Konstruktoren!

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

580

Konstruktoren: Aufruf

■ direkt

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

■ indirekt (Kopie)

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

581

Initialisierung “rational = int”?

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

582

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

```
rational (int num) ← Benutzerdefinierte Konversion von int
                    nach rational. Damit wird int zu einem
                    Typ, dessen Werte nach rational kon-
                    vertierbar sind.
    {}

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

583

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r;    // r = 0
```

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

584

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!

585

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)

586

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

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

- Wir können die Memberfunktionen zusammen mit der Repräsentation anpassen. ✓

587

RAT PACK® Reloaded ...

vorher

```
class rational {
...
private:
    int n;
    int d;
};

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

nachher

```
class rational {
...
private:
    unsigned int n;
    unsigned int d;
    bool is_positive;
};

int numerator () const{
    if (is_positive)
        return n;
    else {
        int result = n;
        return -result;
    }
}
```

588

RAT PACK® Reloaded ?

```
class rational {
...
private:
    unsigned int n;
    unsigned int d;
    bool is_positive;
};

int numerator () const
{
    if (is_positive)
        return n;
    else {
        int result = n;
        return -result;
    }
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher
- Dazu noch möglicher Überlauf

589

Datenkapselung noch unvollständig

Die Sicht des Kunden (rational.h):

```
class rational {
public:
    // POST: returns numerator of *this
    int numerator () const;
    ...
private:
    // none of my business
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.
- Lösung: Nicht nur Daten, auch **Typen** kapseln.

590

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (rational.h):

```
public:
    using integer = long int; // might change
    // POST: returns numerator of *this
    integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!
- Festlegung nur auf **Funktionalität**, z.B.:
 - implizite Konversion `int` → `rational::integer`
 - Funktion `double to_double (rational::integer)`

591

Endlich ein Kundenprogramm, das stabil bleibt:

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

592

```
class rational {
public:
    rational (int num, int denum);
    using integer = long int;
    integer numerator () const;
    ...
private:
    ...
};
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

rational.h

rational.cpp

Klassenname :: Membername

593