Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Exceptions – Ausnahmebehandlung

Zur Laufzeit eines Programms können Fehlersituationen auftreten, die eine weitere Programmausführung nur bedingt oder gar nicht mehr erlauben:

- Division durch 0
- Überlauf (zu kleiner/großer Wert für einen Datentypen)
- nicht genügend Speicherplatz vorhanden (`malloc`)
- fehlerhafte Eingaben durch Benutzer
- Zugriff auf ungültige Adressen im Hauptspeicher
- keine Schreibrechte beim Zugriff auf Verzeichnisse
- zu lesende Datei ist nicht vorhanden
- Netzwerkverbindung ist unterbrochen usw.

Frage: Wie erfolgte in C eine Ausnahmebehandlung? Wie konnte in C festgestellt werden, ob eine Funktion korrekt beendet wurde oder nicht?

Die Funktion kann einen speziellen Fehlerwert zurückliefern, der im Programm mittels `if` abgefragt wird.

```
.....
int find(list_t *l, int val) {
    for (int pos = 0; pos < l->next; pos++)
        if (l->values[pos] == val)
            return pos;
    return -1;
}
.....
pos = find(myList, 10);
if (pos < 0)
    printf("Value not found\n");
else .....
```

Leider gibt es Funktionen, bei denen jeder Rückgabewert gültig ist, z.B. Funktionen aus `math.h` wie `pow` oder `ldexp`.

Bei einigen Funktionen in C wird daher nachträglich abgefragt, ob die zuletzt ausgeführte Operation erfolgreich war.

```
....
int main(void) {
    list_t *l;
    ....

    for (i = 0; i < 20; i++)
        append(l, i);

    i = getValueAt(l, 30);    // Fehlerwert ?????
    if (getError(l) == 0)     // Fehlerbehandlung
        printf("value[%2d] = %2d\n", 30, i);
    ....
}
```

Leider wird das oft vergessen und führt so zu Programmabbrüchen.

In C können Signal-Handler mittels `signal` eingerichtet werden.

```
#include <signal.h>
#include .....

void nullDiv(int sig) {
    printf("0-Division\n");
    exit(1);
}

void main(void) {
    int z, erg;

    signal(SIGFPE, nullDiv);

    scanf("%d", &z);
    erg = 123 / z;
    printf("123 / %d = %d\n", z, erg);
}
```

Exceptions – Ausnahmebehandlung

In C++ gibt es eine spezielle Fehlerbehandlung:

- Wir klammern die Anweisungen, in denen Exceptions, also Ausnahmen, evtl. Fehler auftreten können, mit einem `try`-Block.
- In den `catch`-Block, der dem `try`-Block unmittelbar folgt, schreiben wir die Anweisungen, die beim Auftreten von Exceptions ausgeführt werden sollen.
- Die `throw`-Anweisung löst eine Exception aus:
  - `const char *`-Exception: `throw "Division durch 0";`
  - `int`-Exception: `throw 4711;`

→ Der Code zur „normalen“ Programmausführung ist vom Code zur Fehlerbehandlung getrennt. Man verspricht sich davon eine bessere Lesbarkeit des Codes!

Liste mit Exceptions

exception.h

```
#ifndef _EXCEPTION_H
#define _EXCEPTION_H

#include <string>

class Exception {
private:
    std::string _error;

public:
    Exception(std::string error);
    std::string toString();
};

#endif
```

exception.cpp

```
#include "exception.h"
using namespace std;

// Konstruktor
Exception::Exception(string error) {
    _error = error;
}

// zur Ausgabe
string Exception::toString() {
    return _error;
}
```

liste.cpp

```
#include "liste.h"
#include "exception.h"

.....

int Liste::getValueAt(int idx) {
    if (idx < 0 || idx >= _next)
        throw Exception("out of bounds");
    return _values[idx];
}
```

Die Abarbeitung der Methode `getValueAt` wird durch das `throw` abgebrochen, falls ein unzulässiger Index angeben wird. Es erfolgt dann unmittelbar ein Rücksprung an die aufrufende Methode.

Wie im aufrufenden Programmteil eine solche Ausnahme abgefangen werden kann, zeigt die nächste Folie:

main.cpp

```
#include .....
using namespace std;

int main(void) {
    Liste l(10);

    for (int value = 3; value < 8; value++)
        l.append(value);

    try {
        for (int i = 0; i < 8; i++)
            cout << i + 1 << ":  " << l.getValueAt(i) << endl;
    } catch (Exception e) {
        cout << e.toString() << endl;
    }
}
```

Eine Schachtelung von `try`-Blöcken ist erlaubt.

Wenn eine Exception außerhalb eines `try`-Blocks auftritt, gilt:

- Es wird die Funktion `terminate` aufgerufen, die standardmäßig die Funktion `abort` aufruft, die das Programm beendet.
- Wenn unser Programm vor dem Halten noch etwas anderes tun soll, können wir eine andere Funktion als `abort` hinterlegen.

  Dazu rufen wir die Funktion `set_terminate` auf, der wir einen Zeiger auf eine Funktion übergeben.

Für jeden möglichen Typ, der geworfen werden kann, müssen wir einen entsprechenden `catch`-Block angeben:

```
try {
    // some statements
} catch (invalid_argument e) {
    // other statements
} catch (out_of_range e) {
    // more statements
} catch (bad_alloc e) {
    // even more statements
} catch (...) {  // Syntax korrekt!!!!!
    // handle any other exception!!!!!!
}
```

- Die Syntax ist analog zu einer Funktionsdeklaration.
- Der Parameter `e` kann weggelassen werden, falls der Wert innerhalb der geschweiften Klammern nicht benötigt wird.

Wird die catch-all-Klausel `catch (...)` verwendet, muss es die letzte Klausel in der Sequenz sein:

> *The catch-all clause* `catch (...)` *matches exceptions of any type. If present, it has to be the last catch clause in the handler-sequence. Catch-all block may be used to ensure that no uncaught exceptions can possibly escape from a function that offers nothrow exception guarantee.*[1]

Der Compiler liefert sonst eine Fehlermeldung wie diese:

```
main.cpp: In function ‘int main()’:
main.cpp:13: error: ‘...’ handler must be the last handler for its
    try block [-fpermissive]
   13 |     } catch (...) {
      |
```

[1] https://en.cppreference.com/w/cpp/language/try_catch

Exceptions – Ausnahmebehandlung

In C++ sind einige Klassen zur Ausnahmebehandlung bereits vorhanden:

- `#include <exception>` stellt die Basisklasse `exception` bereit. (Basisklasse: siehe Kapitel Vererbung)
- `#include <new>` stellt die Klasse `bad_alloc` bereit, die geworfen wird, wenn kein Speicher bereit gestellt werden konnte.
- `#include <typeinfo>` stellt die Klassen `bad_cast`, `bad_typeid` bereit.
- `#include <stdexcept>` stellt die Klassen `domain_error`, `invalid_argument`, `length_error` und `out_of_range` bereit.
- Jede dieser Klassen implementiert die Methode `what()`, die ein `const char*` liefert, also eine C-Zeichenkette, die den Fehler näher beschreibt.

```
try {
    ....
} catch (std::out_of_range e) {
    std::cout << e.what() << std::endl;
}
```

```
#include <iostream>
using namespace std;

class InnerC {
public:
    InnerC() {
        cout << "constructor InnerC() called" << endl;
    }

    ~InnerC() {
        cout << "destructor ~InnerC() called" << endl;
    }
};
```

Ein Objekt dieser Klasse `InnerC` wird als Attribut in der folgenden Klasse `OuterC` im Konstruktor erzeugt und im Destruktor zerstört.

```
class OuterC {
    InnerC *_i;

public:
    OuterC() : _i(new InnerC()) {
        cout << "constructor OuterC() called" << endl;
        throw "exception occurred";
    }

    ~OuterC() {
        delete _i;
        cout << "destructor ~OuterC() called" << endl;
    }
};
```

Vorsicht: Was passiert mit dem Objekt `*_i` vom Typ `InnerC` in der Klasse `OuterC`, wenn im Konstruktor von `OuterC` die Ausnahme geworfen wird und in `main` in die Fehlerbehandlung verzweigt wird?

```
1   int main(void) {
2       try {
3           OuterC *f = new OuterC();
4           ....
5           delete f;
6       } catch (const char *e) {
7           cout << "first: " << e << endl;
8       }
9
10      try {
11          OuterC f;
12          ....
13      } catch (const char *e) {
14          cout << "second: " << e << endl;
15      }
16  }
```

Das `delete` in Zeile 5 und der automatische Aufruf des Destruktors in Zeile 13 am Ende des Blocks wird nicht aufgerufen! Es bleibt Müll im Speicher zurück.

Ausgabe:

```
constructor InnerC() called
constructor OuterC() called
first: exception occurred
constructor InnerC() called
constructor OuterC() called
second: exception occurred
```

Weder das `delete` in Zeile 5 noch der automatische Aufruf des Destruktors von `OuterC` in Zeile 13 am Ende des Blocks wird aufgerufen!

In solchen Fällen können Smart-Pointer wie `unique_ptr` oder `shared_ptr` eine große Hilfe sein. Dazu später mehr.

Referenzen

Referenzen erübrigen die Übergabe von Zeigern, wenn Werte innerhalb einer Funktion geändert werden sollen:

```
void swap(int &x, int &y) {
    int t = x;

    x = y;
    y = t;
}

int main(void) {
    int a, b;
    ...
    swap(a, b);
    ...
}
```

Pointer sind klarer und sollten vorgezogen werden: Bei Verwendung von Pointern ist für den *Anwender* einer Funktion ersichtlich, dass die Werte in der Funktion verändert werden können, vergleiche dazu die Aufrufe `swap(a, b)` und `swap(&a, &b)`.

Bjarne Stroustrup, the designer and original implementor of C++: Consequently "plain" reference arguments should be used only when the name of the function gives a strong hint that the reference argument is modified.

Referenzen sind implementiert über die Verwendung von konstanten Zeigern!

Übung: Voriges `swap(int &x, int &y)` entspricht welcher der folgenden Implementierungen?

```
void swap1(const int *x, const int *y) {
    ...           // Implementierung siehe swap3
}

void swap2(int const *x, int const *y) {
    ...           // Implementierung siehe swap3
}

void swap3(int * const x, int * const y) {
    int t = *x;

    *x = *y;
    *y = t;
}
```

Referenzen

Eine Referenz ist ein alternativer Name für eine Variable, Konstante oder Funktion.

```
// typische Deklaration:
T & Bezeichner = Ausdruck;
// Beispiel:
const int &x = 4711;
```

Einige Aspekte von Referenzen:

- Referenzen müssen bei ihrer Deklaration initialisiert werden.
- Referenzen können nicht durch `new` erzeugt werden.
- Zeiger auf Referenzen sind nicht möglich.
- Felder von Referenzen sind nicht möglich.
- Es gibt keine Referenzen auf Referenzen.
- Es gibt keine Referenzen auf `void`.

Lernkontrolle:

- Ist das folgende Programm korrekt?
- Falls ja, welche Ausgabe erzeugt es?

```
#include <iostream>
using namespace std;

int summe(int &a, int &b) {
    return a + b;
}

int main() {
    cout << summe(1, 2) << endl;
}
```

Das vorige Programm ist nicht korrekt: 1 und 2 sind Konstanten und könnten unter dem Alias `a` bzw. `b` verändert werden!

besser:

```
#include <iostream>
using namespace std;

int summe(const int &a, const int &b) {
    return a + b;
}

int main() {
    cout << summe(1, 2) << endl;
}
```

Wenn ein Objekt obj als konstante Referenz an eine Funktion oder Methode übergeben wird, können nur als const deklarierte Methoden des Objekts aufgerufen werden:

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>
4  using namespace std;
5
6  class Date {
7      int _day, _month, _year;
8  public:
9      Date(int d, int m, int y) : _day(d), _month(m), _year(y) {}
10
11     string toString() const {     // !!!!!!!!!!!!!!!
12         ostringstream os;
13         os << _day << "." << _month << "." << _year;
14         return os.str();
15     }
16 };
```

```
void output(const Date& date) {    // !!!!!!!!!!!!!!!!
    cout << "output: " << date.toString() << endl;
}

int main(void) {
    Date d(4, 2, 2020);
    output(d);
}
```

Das Objekt `date`, das der Funktion `output` als Parameter übergeben wird, darf in der Funktion `output` nicht geändert werden.

Daher muss sichergestellt sein, dass der Aufruf `date.toString()` keine Änderung am Objekt `date` bewirkt.

Dies wird dadurch erreicht, dass bei der Methode `toString` der Klasse `Date` das Schlüsselwort `const` in Zeile 11 hinter der Parameterliste angegeben wird.

Kopieren von Objekten

In folgenden Fällen werden die Attributwerte eines Objekts in ein anderes Objekt derselben Klasse kopiert:

- Initialisieren eines Objekts mit einem anderen Objekt derselben Klasse: Zuweisung bei Deklaration.
- Übergabe eines Objekts als Argument bei einem Funktionsaufruf, nicht bei Zeigern oder Referenzen!
- Rückgabe eines Objekts als Funktionswert.

Dabei wird nicht der Konstruktor aufgerufen, sondern der *Copy-Konstruktor*!

Syntax:

```
Klasse(const Klasse& zuKopierendesObjekt)
```

Kopieren von Objekten

Wird kein Copy-Konstruktor programmiert, dann erzeugt der Compiler einen *default copy constructor*:

- Alle Werte des Objekts werden elementweise kopiert, unabhängig davon, ob es Adressen sind. $\rightarrow$ Verweise im Original und in der Kopie sind identisch!
- Eine solche Kopie wird als *flache Kopie* bezeichnet.

```
Liste::Liste(const Liste& l) {    // default
    _size = l._size;
    _next = l._next;
    _values = l._values;
}
```

Soll eine echte Kopie unserer Liste erzeugt werden, muss das Array, das die eigentlichen Werte enthält, kopiert werden.

Schauen wir uns an, was ohne expliziten Kopier-Konstruktor passiert:

```
#include <iostream>
#include "liste.h"
using namespace std;

void fct(Liste cp, int n) {  // call by value
    for (int i = 0; i < n; i++)
        cp.erase(i);

    cout << "(2) cp: ";
    cp.toScreen();

    for (int i = 0; i < n; i++)
        cp.append(2*i+1);

    cout << "(3) cp: ";
    cp.toScreen();
} // hier wird die lokale Kopie cp zerstört
```

Kopieren von Objekten

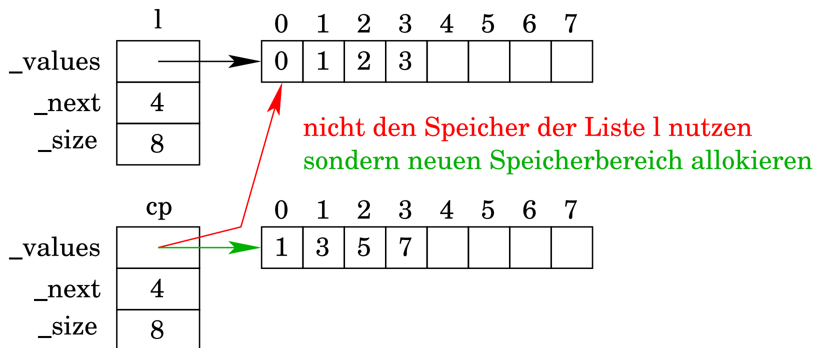
```
int main(void) {
    Liste l(4);
    for (int i = 0; i < 4; i++)
        l.append(i);
    cout << "(1) l: ";
    l.toScreen();

    fct(l, 4);  // call by value
    cout << "(4) l: ";
    l.toScreen();
}
```

Ausgabe:

```
(1) l: 0, 1, 2, 3
(2) cp:
(3) cp: 1, 3, 5, 7
(4) l: 0, 0, 5, 7
```

Wird nur eine flache Kopie erstellt, nutzt die Kopie und das Original den gleichen Speicherbereich für die Daten.

Sobald die Kopie nach dem Ende der Funktion `fct` zerstört wird, wird durch das `delete[]` im Destruktor auch der Speicherbereich freigegeben.



Das Original greift aber noch auf diesen bereits freigegebenen Speicher zu, in dem noch einige der überschriebenen Werte stehen!

Wir fügen unserer Liste also einen Kopier-Konstruktor hinzu:

```
Liste::Liste(const Liste& l) {
    _size = l._size;
    _next = l._next;
    _values = new int[_size];

    for (int i = 0; i < _next; i++)
        _values[i] = l._values[i];
}
```

Jede automatisch erzeugte Kopie wird am Ende seiner Lebensdauer auch automatisch durch den Aufruf des Destruktors zerstört.

Welche Ausgabe erzeugt das Programm?

```
#include <iostream>

class CTest {
public:
    CTest() { std::cout << "K\n"; }
    ~CTest() { std::cout << "D\n"; }
    CTest(const CTest &c) { std::cout << "C\n"; }
};

void down(CTest t) { return; }

int main(void) {
    CTest u;
    CTest v = u;
    down(u);
}
```

Anmerkungen:

- Ein Copy-Konstruktor kann auch explizit aufgerufen werden:

```
CTest *t = new CTest(orig);
```

- Weitere Parameter sind möglich, sie müssen aber alle default-Werte haben.
- Der Copy-Konstruktor kann wie andere Konstruktoren eine Initialisierungsliste haben.

Wie war es bisher?

Rationale Zahlen in C mit modularer Programmierung:

rational.h

```
// incomplete data type
typedef struct rat rat_t;

// interface
rat_t *createRat(int zaehler, int nenner);
void destroyRat(rat_t *r);

rat_t *addRat(rat_t *a, rat_t *b);    // Addition +
rat_t *subRat(rat_t *a, rat_t *b);    // Subtraktion -
rat_t *mulRat(rat_t *a, rat_t *b);    // Multiplikation *
rat_t *divRat(rat_t *a, rat_t *b);    // Division /
.....
void printRat(rat_t *a);
```

Aufruf mit:

```
rat_t *a, *b, *c; *d;
a = createRat(1, 2);
b = createRat(3, 7);

c = addRat(a, b);
d = mulRat(a, b);
printRat(c);
if (isGreater(c, d))
    ...
else ...
.....
```

Schöner wäre:

```
Rational a(1, 2);
Rational b(3, 7);
Rational c, d;

c = a + b;
d = a * b;
cout << c << endl;
if (c > d)
    ...
else ...
.....
```

rational.h

```
#include <iostream>

class Rational {
friend std::ostream& operator<<(std::ostream& os,
                                const Rational& x) {
    os << x.zaehler << "/" << x.nenner;
    return os;
}
private:
    int zaehler, nenner;

    Rational add(const Rational& x) const;  // Addition
    Rational sub(const Rational& x) const;  // Subtraktion
    Rational mul(const Rational& x) const;  // Multiplikation
    Rational div(const Rational& x) const;  // Division

    void kehrwert();
    void kuerzen();
```

```
public:
    Rational(int z = 1, int n = 1);

    Rational operator-() const;          // Vorzeichen -
    Rational operator+(const Rational& x) const;   // +
    Rational operator-(const Rational& x) const;   // -
    Rational operator*(const Rational& x) const;   // *
    Rational operator/(const Rational& x) const;   // /

    bool operator<(const Rational& x) const;
    bool operator>(const Rational& x) const;
    bool operator==(const Rational& x) const;
    bool operator<=(const Rational& x) const;

    int getZaehler() const;
    int getNenner() const;
};
```

rational.cpp

```
#include "rational.h"

// Konstruktor wirft ggf. NaN-Ausnahme
Rational::Rational(int z, int n) {
    if (n == 0)
        throw "NaN exception";
    zaehler = z;
    nenner = n;
    kuerzen();
}

void Rational::kehrwert() {
    int t = zaehler;

    zaehler = nenner;
    nenner = t;
}
```

```
void Rational::kuerzen() {
    // Algorithmus nach Euklid
    int r;
    int p = zaehler;
    int q = nenner;

    do {
        r = p % q;
        p = q;
        q = r;
    } while (r != 0);

    zaehler /= p;
    nenner /= p;
}
```

Rationale Zahlen in C++

```
Rational Rational::add(const Rational& x) const {
    Rational r;

    r.zaehler = zaehler * x.nenner + nenner * x.zaehler;
    r.nenner = nenner * x.nenner;
    r.kuerzen();

    return r;
}

Rational Rational::sub(const Rational& x) const {
    Rational r = *this;
    return r.add(-x);
}

Rational Rational::mul(const Rational& x) const {
    return Rational(zaehler * x.zaehler, nenner * x.nenner);
}
```

Bevor es mit dem Programm weiter geht, wollen wir uns zunächst klar machen, warum die Methoden `add`, `sub` und `mul` nur einen Parameter haben, obwohl bei der Addition, Subtraktion und Multiplikation doch immer zwei Operanden benötigt werden.

Da die Methoden zur Klasse `Rational` gehören, und die Methoden immer anhand eines Objekts der Klasse aufgerufen werden, ist der implizite Objektzeiger `this` immer der erste Operand, der einzige Parameter der Methoden ist der zweite Operand.

```
Rational Rational::add(const Rational& other) const {
      // ^^^^^^^^^^ add gehört zur Klasse Rational !!!
    Rational r;

    r.zaehler = this->zaehler * other.nenner
              + this->nenner * other.zaehler;
    r.nenner = this->nenner * other.nenner;
    r.kuerzen();     // r = *this + other

    return r;
}
```

Würden wir add, sub und mul nicht als Methoden der Klasse Rational, sondern als globale Funktionen definieren, müssten wir zwei Parameter angeben, da dann der implizite Objektzeiger this nicht existiert.

```
Rational add_function(const Rational& a, const Rational& b) {
    Rational r(a.getZaehler() * b.getNenner()
                         + a.getNenner() * b.getZaehler,
               a.getNenner() * b.getNenner());

    return r;
}
```

Da die Attribute zaehler und nenner in der Klasse Rational als private deklariert sind, kann die globale Funktion darauf nicht zugreifen und muss die entsprechenden Werte über die get-Methoden der Klasse ermitteln.

Entsprechendes gilt natürlich für die Methode div, für die überladenen Operatoren +, −, ∗ und / als auch für die Operatoren ==, <, > usw.

```
Rational Rational::div(const Rational& x) const {
    Rational r = x;    // x.kehrwert() würde const verletzen !!
    r.kehrwert();
    return mul(r);
}

int Rational::getZaehler() const {
    return zaehler;
}

int Rational::getNenner() const {
    return nenner;
}

Rational Rational::operator-() const {    // Vorzeichen -
    return Rational(-zaehler, nenner);
}
```

```
Rational Rational::operator+(const Rational& x) const {
    return add(x);
}

Rational Rational::operator-(const Rational& x) const {
    return sub(x);
}

Rational Rational::operator*(const Rational& x) const {
    return mul(x);
}

Rational Rational::operator/(const Rational& x) const {
    return div(x);
}
```

```
bool Rational::operator<(const Rational& x) const {
    return zaehler * x.nenner < nenner * x.zaehler;
}

bool Rational::operator>(const Rational& x) const {
    return zaehler * x.nenner > nenner * x.zaehler;
}

bool Rational::operator==(const Rational& x) const {
    return zaehler * x.nenner == nenner * x.zaehler;
}

bool Rational::operator<=(const Rational& x) const {
    return *this < x || *this == x;
}
```

main.cpp

```
#include <iostream>
#include "rational.h"
using namespace std;

int main(void) {
    Rational a(2), b(1, 3), c;

    c = a + b;    // c = a.add(b)
    cout << "a + b = " << c << endl;

    c = a - b;    // c = a.sub(b)
    cout << "a - b = " << c << endl;

    c = a * b;    // c = a.mul(b)
    cout << "a * b = " << c << endl;

    c = a / b;    // c = a.div(b)
    cout << "a / b = " << c << endl;
```

```
    if (a < b)
        cout << "a < b" << endl;

    if (a > b)
        cout << "a > b" << endl;

    if (a == b)
        cout << "a == b" << endl;

    cout << "c.zaehler = " << c.getZaehler();
    cout << "c.nenner  = " << c.getNenner();

    return 0;
}
```

Überladen von Operatoren

Der Operator `<<` in der Klasse `ostream` ist vielfach überladen, um beliebige Datentypen in ein `ostream`-Objekt schreiben zu können:

```
class ostream {
public:
    ostream& operator<< (bool& wert);
    ostream& operator<< (short& wert);
    ostream& operator<< (unsigned short& wert);
    ostream& operator<< (int& wert);
    ostream& operator<< (unsigned int& wert);
    ostream& operator<< (long& wert);
    ostream& operator<< (unsigned long& wert);
    .....
};
```

in C++: `cout` ist ein vordefiniertes Objekt der Klasse `ostream`
in C: `stdout` ist ein vordefinierter Zeiger vom Typ `FILE *`

```
cout << 1;
```

wird vom Compiler übersetzt nach

```
cout.operator<<(1);
```

Alle `<<`-Operatorfunktionen liefern eine Referenz auf das aktuelle Stream-Objekt als Rückgabewert, wodurch eine Verkettung mehrerer Ausgaben möglich wird:

```
cout << 1 << ", " << 2 << endl;
```

Überladen von Operatoren

Der Operator `<<` wird von links nach rechts abgearbeitet:

```
cout << 1 << ", " << 2 << endl;
```

entspricht also

```
(((cout << 1) << ", ") << 2) << endl;
```

und daher

```
cout.operator<<(1).operator<<(", ")
    .operator<<(2).operator<<(endl);
```

Auch die Priorität kann nicht geändert werden:

```
cout << a + b;      // Klammern unnötig
cout << (a & b);    // Klammern notwendig
```

Überladen von Operatoren

Wenn Sie den Operator `<<` auf eigene Klassen erweitern wollen, müssen Sie die globale Operatorfunktion `operator<<()` überladen:

```
ostream& operator<< (ostream& os, const Rational& r) {
    os << r.getZaehler();
    os << "/";
    os << r.getNenner();
    return os;
}
```

Die Getter-Methoden `getZaehler` und `getNenner` sind notwendig, da der Zugriff auf die privaten Variablen `zaehler` und `nenner` nicht möglich ist.

Es sei denn ...

Überladen von Operatoren

... die Operatorfunktion `operator<<()` wird als Freund der Klasse `Rational` definiert. Freunde dürfen die Inhalte von privaten Variablen sehen!

```
class Rational {
friend ostream& operator<< (ostream& os, const Rational& r) {
    os << r.zaehler << "/" << r.nenner;
    return os;
}
.....
};
```

Anmerkungen:

- Zur besseren Lesbarkeit stehen `friend`-Deklarationen am Anfang der Klasse.
- Es können ganze Klassen als Freund definiert werden.
- Das Konzept der Freunde gibt es nicht in allen objektorientierten Programmiersprachen.

Überladen von Operatoren

Operatoren können wie Funktionen überladen werden. Jede Operatoranwendung kann aufgefasst werden als Aufruf einer Operatorfunktion der Form:

```
<Typ> operator<Zeichen>(<Formalparameter>)
Rational operator+(Rational a)
```

Zum Überladen muss die Operatorfunktion entweder definiert werden als

- Klassenfunktion, dann besitzt sie einen impliziten Objektparameter.

  ```
  Rational Rational::operator+(Rational b);          // binär
  Rational Rational::operator-();        // unär (Vorzeichen)
  ```

- Funktion mit mindestens einem Parameter vom Typ Klasse oder Referenz auf Klasse.

  ```
  Rational operator+(Rational a, Rational b);        // binär
  Rational operator-(Rational a);        // unär (Vorzeichen)
  ```

Überladen von Operatoren

Anmerkungen:

- Operatoren für Standardtypen können nicht überladen werden.
- Operatorfunktionen können explizit aufgerufen werden.
- Priorität und Assoziativität eines Operators wird nicht verändert.
- Operandenzahl eines Operators kann nicht verändert werden.
- Es können keine neuen Operatorzeichen wie ** für Potenzieren definiert werden.

  Dazu müssten auch Priorität und Assoziativität definiert werden, was gar nicht möglich ist.

Übung

Warum ist beim Einlesen eines Wertes von der Tastatur kein Adress-Operator notwendig, wenn doch das Einlesen mittels >> in einen Funktionsaufruf umgewandelt wird?

```
int a;
cin >> a;    // entspricht: cin.operator>>(a);
```

Überladen von Operatoren

Wir hatten bereits festgestellt: Programmierer legen fest, wann zwei Objekte als gleich angesehen werden. In C++ wird dazu der ==-Operator überladen.

- `Konto`: gleiche Kontonummer und Bankleitzahl
- `Student`: gleiche Hochschule und Matrikelnummer

Wunscheigenschaften an den ==-Operator:

- *Reflexivität:* Ein Objekt ist gleich zu sich selbst. Es sollte also `obj == obj` gelten.
- *Symmetrie:* Wenn `obj1 == obj2` gilt, dann sollte auch `obj2 == obj1` gelten.
- *Transitivität:* Gleichheit kann verkettet werden und gilt für mehrere Objekte. Wenn `ob1 == obj2` und `obj2 == obj3` gilt, dann sollte auch `obj1 == obj3` gelten.

Überladen von Operatoren

Frage: Gibt es Probleme, wenn wir folgendes schreiben?

```
#include <iostream>
#include "rational.h"
using namespace std;

int main(void) {
    int x = 5;
    Rational b(1, 3), c;

    c = x + b;                  // ?????????
    cout << "a + b = " << c << endl;

    c = b - x;                  // ?????????
    cout << "a - b = " << c << endl;
}
```

Überladen von Operatoren

Im folgenden Programm wird nicht der Kopier-Konstruktor aufgerufen:

```
#include <iostream>
#include "liste.h"

int main(void) {
    Liste l(10), m;     // Konstruktoraufruf !!!
    for (int i = 1; i <= 10; i++)
        l.append(i);

    m = l;     // Zuweisungsoperator operator= !!!
    for (int i = 11; i <= 20; i++)
        m.append(i);

    std::cout << "Liste m:" << std::endl;
    m.toScreen();
    std::cout << "Liste l:" << std::endl;
    l.toScreen();
}
```

Die Ausgabe sieht nicht korrekt aus:

```
Liste m:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
Liste l:
0, 0, 3, 4, 5, 6, 7, 8, 9, 10
```

Der Zuweisungsoperator, der durch den Compiler zur Verfügung gestellt wird, kopiert lediglich die Attributwerte, siehe auch Abschnitt „Kopieren von Objekten“.

- Es entstehen Speicherlecks, weil der alte Speicherbereich, auf den `m._values` gezeigt hat, nicht freigegeben wird. Unser Zuweisungsoperator auf der folgenden Seite verhindert in Zeile 3 und 4 solche Speicherlecks.
- Da beim Einfügen des 17. Elements der Speicherbereich vergrößert und der alte Bereich mittels `delete[]` gelöscht wird, ist der Zeiger `l._values` nicht mehr gültig und `toScreen` gibt falsche Werte aus.