

Softwareentwicklung Praktikum

Teil 6

SS 2007/08

Univ.-Prof. Dipl.-Ing. Dr. Frank Kappe

Dipl.-Ing. Christian Safran

Institut für Informationssysteme und Computer-Medien

Technische Universität Graz

Inhalt Teil 6

- Friends

- Operator Overloading

Wiederholung: Sichtbarkeit (Kapselung)

- **private:** Nur sichtbar für eigene Methoden
- **public:** Sichtbar von aussen
- **protected:** Nur sichtbar für abgeleitete Klassen

- Gelegentlich will bzw. muss man bestimmten Klassen oder Methoden, die nicht in der eigenen Ableitungshierarchie liegen, den Zugriff auf die eigenen **private** oder **protected** Attribute oder Methoden erlauben

- C++ Konstrukt: **friend**

Syntax

- Irgendwo in der Klassendeklaration:

- `friend class B; // all methods from B may access`
- `friend void B::f() // only method f() in B may access`
- `friend global_f() // global_f may access my members`

- Durch friend wird der Zugriff auf alle Members der eigenen Klasse erlaubt

- Es ist nicht möglich, nur den Zugriff auf gewisse Members freizugeben

- Sollte nur in Ausnahmefällen verwendet werden !

Operatoren in C++

- Operatoren sind "normale" Funktionen bzw. Methoden
- z.B.:

```
a = b + c;
```

ist äquivalent zu:

```
operator=(a, operator+(b,c));
```

```
a.operator=(b.operator+(c));
```

Überladen von Operatoren

- Compiler sucht automatisch einen "passenden" Operator
 - gemäß den Regeln für Funktionsüberladung

- Die folgenden Operatoren können "überladen" werden:

- **Unäre Operatoren:**

+ - * & ~ ! ++ -- -> ->*

- **Binäre Operatoren:**

+ - * / % ^ & | << >>
 += -= *= /= %= ^= &= |= <<= >>=
 < <= > >= == != && ||
 , [] ()
 new new[] delete delete[]

- Die folgenden Operatoren können nicht überladen werden:

. .* :: ?:

Besonders wichtige Anwendungsfälle:

- Zuweisung (=)
- Ein/Ausgabe von/zu Streams (>>, <<)
- Arithmetische Operatoren
- Array-Indizierung ([])
- Pointer-Zugriff (& * ->)
- new und delete

Zuweisungs-Operator (operator=)

- Ähnliche Stellung wie Copy-Constructor
 - Wird implizit aufgerufen
 - Wird implizit generiert (wenn nicht vorhanden)
- Dieselben Probleme wie beim Copy-Constructor
 - Default-Operator macht nur ein "shallow copy"
 - Ähnlich wie Copy-Constructor, nur wird keine Kopie erzeugt, sondern ein schon existierendes Objekt (der linke Operand) modifiziert
 - Return-Typ ist Referenz auf Objekt (für Verkettung: $a = b = c$)
 - Zusätzliches Problem: Zuweisung an sich selbst abfangen !
- **⇒ Faustregel: Jede Klasse muss einen = Operator haben !**
- Kann "private" deklariert werden, um implizite Aufrufe zu verhindern
- Kann überladen werden, um andere Typen in den Typ des linken Operanden zu konvertieren

Ausgabe-Operator (<<)

- Praktisch, um Inhalte einer Klasse auszugeben; z.B.:
 - `std::cerr << myobject << std::endl;`
- Kann keine Member-Funktion dieser Klasse sein
 - linker Operand ist vom Typ `std::ostream&`
 - ⇒ für Zugriff auf private Members als "friend"-Funktion deklarieren
 - Return-Typ ist ebenfalls `std::ostream&` (für Verkettung)
`operator<<(operator<<(std::cerr, myobject), std::endl);`
 - Kann nicht virtual deklariert werden
 - ⇒ `operator<<` nur in Basisklasse implementieren, und darin eine virtuelle Methode aufrufen, die das Objekt auf `ostream` ausgibt. Diese virtuelle Methode jeweils in den abgeleiteten Klassen implementieren
- Deklaration also:
`friend std::ostream &operator<<(std::ostream&, const
MyClass&);`

Beispiel: Modifizierte Klasse Person (person.h)

```
1 // person.h
2 #ifndef person_h
3 #define person_h
4
5 #include <new>
6 #include <iostream>
7
8 class Person // abstract base class
9 {
10
11     ... // as before
12
13
14
15
16     virtual void toStream(std::ostream&) const = 0;
17
18     Person &operator=(const Person&) throw(std::bad_alloc);
19     friend std::ostream &operator<<(std::ostream&, const Person&);
20 };
21 #endif
```

Beispiel: Zuweisungs-Operator (person.cpp)

```
71 Person &Person::operator=(const Person& src) throw(std::bad_alloc)
72 {
73     if (this == &src)    // self-assignment?
74         return *this;    // do nothing
75
76     firstName_ = lastName_ = 0;
77
78     setFirstName(src.firstName_); // if this fails nothing has been allocated
79
80     try
81     {
82         setLastName(src.lastName_);
83         return *this;
84     }
85     catch(std::bad_alloc &e)
86     {
87         cout << "Person: allocating Last Name failed" << endl;
88         delete [] firstName_;
89         throw std::bad_alloc();
90     }
91 }
```

Beispiel: Copy Konstruktor und << Operator

```
93 Person::Person(const Person &src) throw(std::bad_alloc)
94 {
95     cout << "Person: Copy-Constructor (" << src.firstName_
96         << ', ' << src.lastName_ << ') ' << endl;
97     *this = src;    // simply use = operator
98 }
99
100 std::ostream &operator<<(std::ostream &os, const Person &p)
101 {
102     p.toStream(os);
103     return os;
104 }
```

Beispiel: Implementation von Student::toStream

```
29 void Student::toStream(std::ostream& os) const
30 {
31     os << "Student: " << getFirstName() << ' ' << getLastName()
32         << ": MatrikelNummer=" << matrikelNumber_;
33 }
```

Beispiel: assignment_demo.cpp

```
1 // assignment_demo.cpp - uses assignment operator
2 #include "student.h"
3
4 int main()
5 {
6     Student s1("Max", "Mustermann", 1234567);
7     Student s2("Josef", "Huber", 777777);
8     Student s3;
9     std::cout << "-----1-----" << std::endl;
10    s3 = s2 = s1;
11    std::cout << s3 << std::endl;
12    std::cout << "-----2-----" << std::endl;
13 }
```

Arithmetische Operatoren (inkl. Bit-Operatoren)

- **Binäre arithmetische Operatoren (z.B. + - * /):**
 - Als Funktion: 2 Parameter (1.: linker Operand, 2.: rechter Operand)
 - Als Methode: 1 Parameter (=rechter Operand; linker ist das Objekt)
 - Das Ergebnis(-objekt) wird durch den Operator erzeugt und mit return zurückübergeben (by value!)
- **Unäre arithmetische Operatoren (z.B. - ++):**
 - Als Funktion: 1 Parameter
 - Als Methode: keine Parameter (Operand ist das Objekt)
 - Das Ergebnis(-objekt) wird durch den Operator erzeugt und mit return zurückübergeben (by value!)
- **Kombinations-Operatoren (z.B. += -= *= /=):**
 - Parameter wie bei binären arithmetischen Operatoren
 - Der linke Operand wird modifiziert und per Referenz zurückübergeben

Weitere Operatoren

- **Boolesche Operatoren (z.B. == != > <):**
 - Parameter wie bei binären arithmetischen Operatoren
 - Ergebnis ist vom Typ bool und wird by value übergeben
- **Array-Indizierung []**
 - Um beliebige Objekte wie Arrays zu adressieren zu können
 - Um Index-Grenzen zu prüfen (std::range_error exception)
 - Ergebnistyp "beliebig", per Referenz übergeben
- **Zugriffs-Operatoren (& * ->)**
 - Zur Simulation von Pointern ("smart Pointer")
- **Typ-Umwandlungs-Operatoren**
 - Immer Member der Klasse, von der konvertiert werden soll (Quellentyp)
 - Syntax:
`operator TargetType();`

Beispiel: vector2d.h (1/2)

```
1 // vector2d.h - a two-dimensional vector
2 // demonstrates operator overloading
3
4 #ifndef vector2d_h
5 #define vector2d_h
6
7 #include <stdexcept>
8 #include <iostream>
9
10 enum INDEX { x=0, y=1 ,X=0, Y=1 };
11
12 class Vector2D
13 {
14 private:
15     double x_;
16     double y_;
17
18 public:
19     Vector2D() : x_(0.0), y_(0.0) {}
20     Vector2D(double x, double y) : x_(x), y_(y) {}
21     Vector2D(const Vector2D &src) : x_(src.x_), y_(src.y_) {}
22     virtual ~Vector2D() throw();
```

Beispiel: vector2d.h (1/2)

```
24 virtual Vector2D &operator=(const Vector2D&);
25 virtual double &operator[](const INDEX) throw(range_error);
26
27 virtual Vector2D &operator+=(const Vector2D&);
28 virtual Vector2D &operator-=(const Vector2D&);
29 virtual Vector2D &operator*=(double);
30
31 virtual Vector2D operator+(const Vector2D&) const;
32 virtual Vector2D operator-(const Vector2D&) const;
33 virtual Vector2D operator-() const;
34 virtual double operator*(const Vector2D&) const; // scalar product
35 virtual Vector2D operator*(double) const;
36 friend Vector2D operator*(double, const Vector2D&);
37
38 virtual bool operator==(const Vector2D&) const;
39 virtual bool operator!=(const Vector2D&) const;
40
41 virtual operator bool() const; // true if non-zero
42
43 friend istream& operator>>(istream&, Vector2D&);
44 friend ostream& operator<<(ostream&, const Vector2D&);
45 };
46 #endif
```

Implementation (vector2d.cpp 1/5)

```
8 Vector2D &Vector2D::operator=(const Vector2D &v)
9 {
10     x_ = v.x_;
11     y_ = v.y_;
12     return *this;
13 }
14
15 double &Vector2D::operator[](const INDEX index) throw(std::range_error)
16 {
17     switch (index)
18     {
19         case X :
20             return x_;
21
22         case Y :
23             return y_;
24
25         default:
26             throw std::range_error("Vector2D: index not in [x,y]");
27     }
28 }
```

Implementation (vector2d.cpp 2/5)

```
38 Vector2D &Vector2D::operator-=(const Vector2D &v)
39 {
40     x_ -= v.x_;
41     y_ -= v.y_;
42     return *this;
43 }

52 Vector2D Vector2D::operator+(const Vector2D &v) const
53 {
54     Vector2D temp;
55     temp.x_ = x_ + v.x_;
56     temp.y_ = y_ + v.y_;
57     return temp; // by value!
58 }
59
60 Vector2D Vector2D::operator-(const Vector2D &v) const
61 {
62     Vector2D temp(*this);
63     temp -= v;
64     return temp; // by value!
65 }
```

Implementation (vector2d.cpp 3/5)

```
67 Vector2D Vector2D::operator-() const // unary -
68 {
69     return Vector2D(-x_, -y_); // by value
70 }
71
72 double Vector2D::operator*(const Vector2D &v) const
73 {
74     return (x_ * v.x_ + y_ * v.y_); // scalar product
75 }
76
77 Vector2D Vector2D::operator*(double factor) const
78 {
79     return Vector2D (x_ * factor, y_ * factor);
80 }
81
82 Vector2D operator*(double factor, const Vector2D &v)
83 {
84     return Vector2D(factor * v.x_, factor * v.y_);
85 }
```

Implementation (vector2d.cpp 4/5)

```
87 bool Vector2D::operator==(const Vector2D &v) const
88 {
89     return (x_ == v.x_ && y_ == v.y_);
90 }
91
92 bool Vector2D::operator!=(const Vector2D &v) const
93 {
94     return !(*this == v);
95 }
96
97 Vector2D::operator bool() const // true if non-zero
98 {
99     return (x_ != 0.0 || y_ != 0.0);
100 }
```

Implementation (vector2d.cpp 5/5)

```
102 std::istream& operator>>(std::istream& in, Vector2D& v)
103 {
104     double x, y;
105     in >> x >> y;
106     if (in.good())
107     {
108         v.x_ = x;
109         v.y_ = y;
110     }
111     return in;
112 }
113
114 std::ostream& operator<<(std::ostream& out, const Vector2D& v)
115 {
116     out << "(" << v.x_ << "/" << v.y_ << ")";
117     return out;
118 }
```

Beispiel: operator_demo.cpp

```
1 // operator_demo.cpp - uses overloaded operators
2 #include "vector2d.h"
3 using std::cout;
4 using std::endl;
5 using std::cin;
6
7 int main()
8 {
9     Vector2D p1, p2, p3;    // initialized to zero
10    cout << "P1=" << p1 << " P2=" << p2 << " P3=" << p3 << endl;
11
12    cin >> p3;              // read new vector
13    p1 = p2 = p3;          // p1.operator=(p2.operator=(p3));
14    cout << "P1=" << p1 << " P2=" << p2 << " P3=" << p3 << endl;
15
16    if (p1) // non-zero (using bool operator)
17    {
18        p1 = -p2 + p3 * 0.5;
19        p2[x] = 1.0;
20        p2[Y] = 2.0;
21        cout << "P1=" << p1 << " P2=" << p2 << " P3=" << p3 << endl;
22        cout << "P1 * P2 = " << p1 * p2 << endl;
23        cout << "3.0 * P1 = " << 3.0 * p1 << endl;
24    }
25 }
```

Experimente / Übungen

- Was passiert, wenn Sie den = Operator der Klasse Person undefiniert lassen?