Software-Technik: Vom Programmierer zur erfolgreichen …

8 Abstrakte Datentypen: Einheit von Daten und

8.1 Die Bedeutung von Schnittstellen

8.2 Klassen als abstrakte Datentypen
Operatoren

8.3 Generische Programmierung, 2. Teil

8.4 Ausnahmebehandlung, 2. Teil

8.5 Zusammenfassung

Lehrbuch: 4.1
Kompendium, 3. Auflage: 8.10, 9.1
Kompendium, 4. Auflage: 8.1 ... 8.17

Im Buch zur SW-Entwicklung im Team nicht im Detail behandelt

Folien mit gelben Punkten am oberen rechten Rand sind weniger wichtiger für das Verständnis der nachfolgenden Kapitel.

Überladen von Operatoren

Überladen von Operatoren, Einleitung

Mit Hilfe des Überladens von Operatoren können fast alle Operatoren für selbstdefinierte Klassen mit einer neuen Bedeutung versehen werden. Um nun den Mechanismus des Operator-Overloading zu verstehen, betrachte man folgende Beispiele:

```
a = b;        -->  a.operator=   (b);
a + b;        -->  a.operator+   (b);
a += b;       -->  a.operator+=  (b);
!a;           -->  a.operator!   ();
```

Das Überladen kann auch unter Verwendung von globalen Funktionen erfolgen, z.B.:

```
a = b;        -->  operator=   (a, b);
a + b;        -->  operator+   (a, b);
a += b;       -->  operator+=  (a, b);
cout << s     -->  operator<<  (cout, s)
```

Überladen von Operatoren, Einleitung

Das Überladen kann auch unter Verwendung von globalen Funktionen erfolgen, z.B.:

```
a = b;          -->  operator=   (a, b);
a + b;          -->  operator+   (a, b);
a += b;         -->  operator+=  (a, b);
cout << s       -->  operator<<  (cout, s)
```

Überladen von Operatoren, Übersicht

Überladbare Operatoren:

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

Nicht überladbare Operatoren:

- `.`
- `.*`
- `::`
- `?:`
- `sizeof`
- `static_cast`
  `const_cast`
  `reinterpret_cast`
  `dynamic_cast`
- `typeid`

Abstraktes Beispiel

Operatoren

X.h

```
class X {

  };
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Ist das Programm syntaktisch korrekt?

1. Ja
2. Standardkonstruktor fehlt (Fehler)
3. operator+ (Plus) für Klasse X nicht definiert (Fehler)
4. return in main verboten (Fehler)

Operatoren

X.h

```
class X {

   };
X operator+(const X& x1, const X& x2);
```

globale Funktion

X.cxx

```
#include "X.h"

X operator+(const X& x1, const X& x2)
{
X hlp;
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Operatoren

X.h

Methode

```
class X {

  public:
  X operator+(const X& x2) const;
  };
```

X.cxx

```
#include "X.h"

X X::operator+(const X& x2) const
{
X hlp;
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Operatoren: Klasse mit Attributen

X.h

Methode

```
class X {
   int w;
   public:
   X operator+(const X& x2) const;
   };
```

X.cxx

```
#include "X.h"

X X::operator+(const X& x2) const
{
X hlp;
hlp.w = w   +   x2.w;
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Operatoren: Globale Funktion mit Attributen

X.h

```
class X {
   private:
   int w;
   };
X operator+(const X& x1, const X& x2);
```

globale Funktion

X.cxx

```
#include "X.h"

X operator+(const X& x1, const X& x2)
{
X hlp;
hlp.w = x1.w   +   x2.w;
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Ist das Programm syntaktisch korrekt?

1. Ja
2. Standardkonstruktor fehlt (Fehler)
3. operator+ (Plus) für Klasse X nicht definiert (Fehler)
4. operator+ greift auf private-Attribute zu (Fehler)

Operatoren: Globale Funktion mit Attributen

X.h

globale Funktion

```
class X {
   int w;
   public:
   int getW() const   {return w;}
   void setW(int a) { w =a;}
   };
X operator+(const X& x1, const X& x2);
```

X.cxx

```
#include "X.h"

X operator+(const X& x1, const X& x2)
{
X hlp;
hlp.setW(x1.getW()   +   x2.getW());
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Was ist „unschön“?

1. Alles gut programmiert
2. Setter und Getter stehen nun für alle Funktionen zur Verfügung
3. SetW schlechter Name, besser Set.
4. operator% (Modulo) fehlt

Operatoren: Globale Funktion mit Attributen

X.h

globale Funktion

```
class X {
   int w;
   public:
   int& getW() {return w;}
   int getW() const  {return w;}
   };
X operator+(const X& x1, const X& x2);
```

X.cxx

```
#include "X.h"

X& operator+(const X& x1, const X& x2)
{
X hlp;
hlp.getW() = x1.getW() +  x2.getW();
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Operatoren: Globale Funktion mit Attributen

X.h

```
class X {
   int w;
   friend  X operator+
              (const X& x1, const X& x2);
   };
X operator+(const X& x1, const X& x2);
```

operator+ ist zweimal deklariert.

X.cxx

```
#include "X.h"

X operator+(const X& x1, const X& x2)
{
X hlp;
hlp.w = x1.w   +   x2.w;
return hlp;
}
```

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Operatoren: Globale Funktion mit Attributen

X.h

```
class X {
  int w;
  friend  X operator+
            (const X& x1, const X& x2);
  };
```

X.cxx

```
#include "X.h"

X operator+(const X& x1, const X& x2)
{
X hlp;
hlp.w = x1.w   +   x2.w;
return hlp;
}
```

Außerhalb der Klasse kann die wiederholte Deklaration entfallen, innerhalb muss sie bleiben, damit die friend Vereinbarung erfolgt.

main.cxx

```
#include "X.h"
int main() {
X x1, x2;
x1 = x2 + x1;
return 0;
}
```

Aufgabe

```
typedef int V_BaseType;

class Vektor{
public:
   Vektor(int dim);
   Vektor(const Vektor& v2);
   ~Vektor();
   Vektor& operator=(const Vektor& v2);

   void Init(V_BaseType wert);
   void Plus(const Vektor& v2, Vektor& erg);
   void Print() const;
   int GetDim() const {return dimension;}
   V_BaseType GetDatum(int index) const {return daten[index];}
   void Resize(int dim);

private:
   V_BaseType* daten;
   int dimension;
};
```

Implementieren Sie für Ihre Vektor-Klasse verschiedene Operatoren z.B. [], <<, >>, + …

Implementieren Sie auch operator++ zur Erhöhung aller Elemente jeweils um 1.

Googlen Sie wie die Syntax für Post- und Prä-Increment sich unterscheidet.

Implementieren Sie beide Varianten und vergessen Sie nicht Ihre Test.

```
bool testPlusOp() {
    Vektor v1(4);
    const int wert1 = 81;
    v1.Init(wert1);

    Vektor v2(4);
    const int wert2 = -10;
    v2.Init(wert2);

    Vektor result(4);
    result = v1 + v2;

    for (int i = 0; i < result.GetDim(); ++i) {
        if (result.GetDatum(index: i) != (wert1 + wert2)) {
            return false;
        }
    }
    return true;
}
```

7. Vorlesung; Fr. 08.11.2024

Vorlesung

Wiederholung / Ankündigung (30.10.2023) Operatoren (20.10.2023)
Sonstiges zu Klassen (zum Selbststudium) (20.10.2023)

Übungsaufgaben WS 2023/24; Sprechfunk-Annotation

siehe vorherige Woche; mit Abgabe bis Sonntag, den 5.11.2023 23:59 Uhr

Übungsaufgaben

Klasse Vektor mit Operatoren Ausgangscode (26.10.2024) VS2022
Aufgabe Klasse Matrix mit Operatoren Ausgangscode (26.10.2024) VS2022

```
bool testPostPreInce() {
    Vektor v1(4);
    const int wert1 = 81;
    v1.Init(wert1);

    Vektor v2(v1++);
    for (int i = 0; i < v2.GetDim(); ++i) {
        if (v2.GetDatum(index:i) != wert1 ) {
            return false;
        }
    }
    for (int i = 0; i < v1.GetDim(); ++i) {
        if (v1.GetDatum(index:i) != wert1+1) {
            return false;
        }
    }
```

```
Vektor v3(++v2);
for (int i = 0; i < v3.GetDim(); ++i) {
    if (v3.GetDatum(index:i) != wert1+1) {
        return false;
    }
}
for (int i = 0; i < v2.GetDim(); ++i) {
    if (v2.GetDatum(index:i) != wert1 + 1) {
        return false;
    }
}
return true;
```

Konkretes Beispiel
Klasse Complex

Operator- Überladen am Beispiel der Klasse *Complex* (1)

H-Datei

```
class complex {
      double re, im;
   public:
      // Kopierkonstruktor, Destruktor und Zuweisungsoperator per Default
      complex(double r=0, double i=0);  // ist auch Standardkonstruktor !

      friend complex operator+ (const complex& a, const complex& b);  // gut
   //  complex operator+ (const complex& a);  → ungünstig
      . . .
};
```

- Symmetrische Operatoren, wie z.B. „operator+“, sollten als „globale-Funktion“ mit Rückgabe des Resultats als Wert definiert werden (meist friend-Vereinbarung erforderlich)! Die rote Lösung führt zu Problemen!

Globale Funktionen ← → Methode

H-Datei

```
class complex {
      double re, im;
   public:
   friend complex operator+ (const complex& a, const complex& b);  // gut
  //  complex operator+ (const complex& a);  → ungünstig
};
. . .
complex x, y;
. . .
y = 5.0 + x;  //entspricht: y = operator+ (complex(5.0), x);  → in Ordnung !
                // Eine Konvertierung von double in complex ist möglich

// y = 5.0 + x;  //  entspricht: y = (5.0).operator+ (x);  → Syntaxfehler !
              // Die folgende Konvertierung findet nicht statt.
              y = (complex(5.0)).operator+ (x);  // kein Syntaxfehler wäre diese
                                                 // Schreibweise
```

Operator- Überladen am Beispiel der Klasse *Complex* (2)

H-Datei

```
class complex {
      double re, im;
   public:
      // Kopierkonstruktor, Destruktor und Zuweisungsoperator per Default
      complex(double r=0, double i=0);  // ist auch Standardkonstruktor !

      complex& operator+= (const complex& a);
      . . .
};

. . .
complex x1, x2, y;
. . .
y += 5.0;          //  →  y.operator+= (5.0);  →  y.operator+= (complex(5.0));

y += x1 += x2;  //  →  y.operator+= (x1.operator+=(x2));
```

Im Gegensatz zu „operator+“ wird „operator+=“ wie alle Zuweisungsoperatoren als Elementfunktion mit Rückgabe einer Referenz auf das eigene Objekt deklariert

Operator- Überladen am Beispiel der Klasse *Complex* (3)

H-Datei

```
class complex {
    double re, im;
  public:
    // Kopierkonstruktor, Destruktor und Zuweisungsoperator per Default
    complex(double r=0, double i=0);  // ist auch Standardkonstruktor !

    friend ostream& operator<< (ostream& os, const complex& c);
    . . .
};

. . .
complex x1, x2, y;
. . .
cout << y;              // → operator<< (cout, y);

cout << x1 << x2;       // → operator<< (operator<< (cout, x1), x2);
```

ostream ist eine Klasse der C++-Standardbibliothek, die Ausgabeobjekte, wie z.B. *cout*, definiert. Die hier angegebene Syntax ist die einzig mögliche, um den Operator wie bei der Ausgabe von vordefinierten Typen verwenden zu können.

Operator- Überladen am Beispiel der Klasse *Complex* (4)

H-Datei

```
class complex {
    double re, im;
  public:
    // Kopierkonstruktor, Destruktor und Zuweisungsoperator per Default
    complex(double r=0, double i=0);  // ist auch Standardkonstruktor !

    friend complex operator+ (const complex& a, const complex& b);
    complex& operator+= (const complex& a);

    . . .

    friend ostream& operator<< (ostream& os, const complex& c);
};
```

Operator- Überladen am Beispiel der Klasse *Complex* (5)

CPP-Datei

```
complex::complex(double r, double i) { re = r; im = i; }

complex operator+ (const complex& a, const complex& b) {
   complex temp;
   temp.re = a.re + b.re; temp.im = a.im + b.im; return temp;
   // effizienter:
   // return complex(a.re + b.re, a.im + b.im);
}

complex& complex::operator+= (const complex& a) {
   re += a.re; im += a.im; return *this;
}

// . . .

ostream& operator<< (ostream& os, const complex& c) {
   os << '(' << c.re << ',' << c.im << "); ";  return os;
}
```

Operator- Überladen am Beispiel der Klasse *Complex* (5)

```
istream& operator>> (istream& is, complex& c) {
   char dummy;
    is >> dummy >> c.re;
    is >> dummy;
    is >> c.im;
    is >> dummy;
    return is;
}
```

Operator- Überladen am Beispiel der Klasse *Complex* (6)

Anwendung

```
#include "complex.h"

int main() {
  complex x1(1, 2), x2(3, 4), x3, y1, y2, y3;

  x3 = complex(5, 6);

  y1 = x1 + x2;
  y2 = 5 + x1;
  y3 = x1 + 5;
  y3 += y2;

  cout << x1 << x2 << x3 << y1 << y2 << y3 << endl;

  x1 += x2 += x3;

  cout << x1 << x2 << x3 << endl;
}
```

Aufgabe

Implementieren Sie für Ihre Matrix-Klasse verschiedene Operatoren z.B. [], <<, >>, + …

Implementieren Sie auch operator++ zur Erhöhung aller Elemente jeweils um 1.

Googlen Sie wie die Syntax für Post- und Prä-Increment sich unterscheidet.

Implementieren Sie beide Varianten und vergessen Sie nicht Ihre Test.

```
typedef double BasisTyp;
class Matrix {
public:
   Matrix(int z, int sp);
   Matrix(const Matrix& m2);
   ~Matrix();

   void Resize(int z, int sp);
   void Setze(int z, int sp, BasisTyp wert);
   BasisTyp Lese(int z, int sp) const;
   int GetSpAnz() const { return spanz; }
   int GetZeAnz() const { return zanz; }

   Matrix& operator=(const Matrix& m);

private:
   int GetPos(int z, int sp) const;
   BasisTyp* arr;
   int zanz;
   int spanz;
   };
```

Mehrfaches Überladen des gleichen Operators

Der operator+ und auch die meisten anderen können innerhalb (und / oder außerhalb) einer Klasse auch mehrfach überladen werden:

```
class complex
{
...
complex operator+ (const complex &a);
complex operator+ (const float b);
float operator+ (int d);
...
};
```

Operatoren Überladung, Zusammenfassung (1)

Die allgemeine Form zum Überladen von *binären Operatoren* sieht wie folgt aus:

```
friend X  operator op  (const X& x1, const X& x2);
```

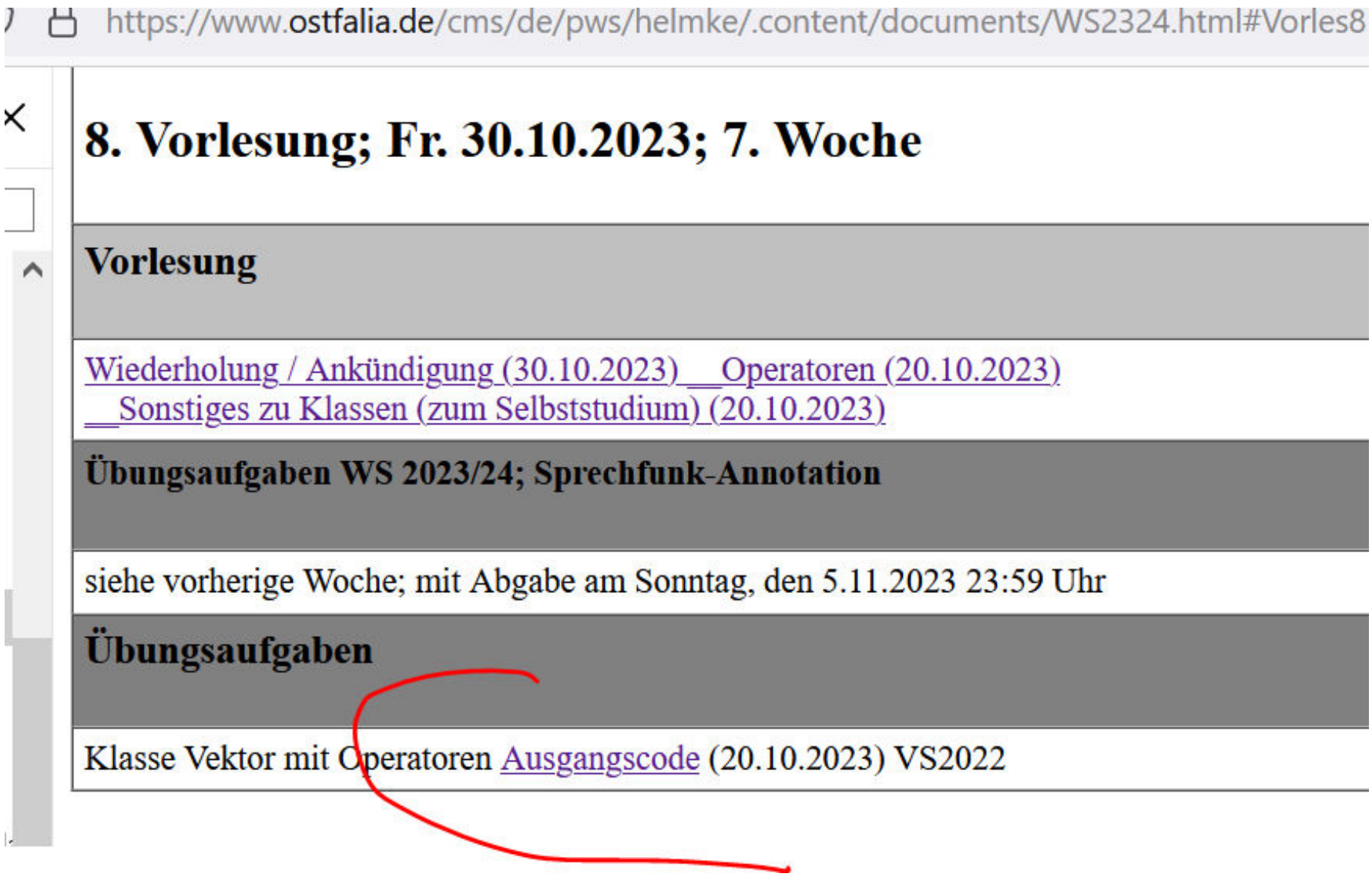
Die allgemeine Form für das Überladen von Zuweisungsoperatoren (z.B. +=, -=, = /= usw ..) sieht wie folgt aus:

```
X&  operator op=  (const X& x);
```

Die allgemeine Form für das Überladen von Ausgabeoperatoren sieht wie folgt aus:

```
friend ostream&  operator <<  (ostream& os, const X& x2);
```

Code für Übung heute



Operatoren überladen, Zusammenfassung (2)

Hinweise zum Überladen von Operatoren

Durch das Überladen können die Bedeutungen der Operatoren verändert werden, ihre Prioritäten bleiben aber erhalten.

Die Bedeutung eines zusammengesetzten Operators, wie z.B. **+=**, passt sich nicht automatisch der Bedeutung seiner einzelnen Komponenten **+** und **=** an, sondern er muss separat definiert werden, siehe dazu das vorangehende Beispiel.

Beim Überladen von Operatoren muss **zumindest ein** Argument den Typ einer (selbstdefinierten) Klasse oder einer Referenz darauf haben, d.h. es ist z.B. nicht möglich die Operatoren für die vordefinierten elementaren Datentypen auf diese Weise neu zu definieren.

Funktionsobjekte (1)

```
class Func {
  public:
    double operator()(double x)
          { return x*x; }
};
int main() {
Func f;
   cout << f(5) << endl;}
```

```
void Print(double x, Func f)
{
 cout << f(x) << endl;
}
. . .
Print(5, f);
```

Funktionsobjekte sind Objekte, die wie Funktionen benutzt werden. Sie zeichnen sich dadurch aus, dass sie eine Elementfunktion haben, die den Funktionsoperator „()“ überlädt. Funktionsobjekte können natürlich auch als Parameter von Funktionen verwendet werden, z.B. wie folgt:

Funktionsobjekte (2)

```
class Func {
    double a, b, c; // Polynominalkoeffizienten
  public:
    Func(double a1, double b1=0, double c1=0) {
      a = a1; b = b1; c = c1;
    }
    double operator()(double x) const { return a*x*x + b*x + c; }
};
```

```
int main() {
  Func f(3, -2, -7);
  cout << f(5) << endl;
}
```

Ein wesentlicher Vorteil von Funktionsobjekten gegenüber Funktionen liegt darin, dass sie über ein *Gedächtnis* in Form von internen Variablen verfügen können, die über einen Konstruktor initialisierbar sind.

Funktionsobjekte sind die objektorientierte Alternative zu globalen Funktionen!

Problem bei globalen Operatoren als friend

```
template <typename T>
class Vektor{
public:
   Vektor(int dim);
…
friend operator+(const Vektor<T>& v1,
            const Vektor<T>& v2);
friend inline ostream&
            operator<<(ostream& str, const
            Vektor<T>& v);
};
```

Geht nicht.

```
template <typename T>
class Vektor{
public:
   Vektor(int dim);
…
 template <typename U>
 friend Vektor<U> operator+(
 const Vektor<U> & v1, const Vektor<U>& v2);

 template <typename V>
 friend inline ostream& operator<<(ostream & str,
              const Vektor<V>& v);

};
```

Geht

Problem bei globalen Operatoren als friend

```
template <typename T>
class Vektor{
public:
   Vektor(int dim);
…
};
```

```
template <typename U>
Vektor<U> operator+(
const Vektor<U> & v1, const Vektor<U>& v2);

template <typename V>
friend inline ostream& operator<<(ostream & str, const Vektor<V>& v);
```

Übung

Erweitern Sie das Matrixbeispiel, indem Sie für Matrix einen operator+ (oder operator-) implementieren.

Schreiben Sie einen Test, der die Funktionalität nachweist und nicht vergessen: „erst mal Text, dann Code“

Zusatzübung:
Implementieren Sie anschließend den operator<<

Warum gibt es Probleme beim Index-Operator operator[].
z.B. als double operator[](int index) { /*code */ }
Hiermit wäre nur ein lesender Zugriff möglich.

Unterbinden von Operationen

C++-11

```
class  NewVec {
public:
NewVec(int s, double d) :
                    size(s), defWert(d) {};
double GetDefWert(){ return defWert; }
void* operator new(std::size_t) = delete;
private:
   int size; double defWert;
};
```

```
int main() {
NewVec n1(1, 1.8);
NewVec* p = &n1;
cout << p->GetDefWert() << endl;
NewVec* p = new NewVec(16, 16.4);
cout << p->GetDefWert() << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

C++-11

Unterbinden von Operationen

```
class  NewVec {
public:
NewVec(int s, double d) :
                    size(s), defWert(d) {};
double GetDefWert(){ return defWert; }
void* operator new(std::size_t) = delete;
private:
   int size; double defWert;
};
```

```
int main() {
NewVec n1(1, 1.8);
NewVec* p = &n1;
cout << p->GetDefWert() << endl;
NewVec* p = new NewVec(16, 16.4);
cout << p->GetDefWert() << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler, new verboten für diese Klasse
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

Unterbinden von Operationen

C++-11

```
class  FVec {
public:
  FVec(double d) : w(d) {};
   double Wert(int k) {return k*w;}
   double Wert(double d) = delete;
private:
   double w;
};
```

```
int main() {
FVec f(1.1);
cout << f.Wert(8) << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

C++-11

Unterbinden von Operationen

```
class  FVec {
public:
  FVec(double d) : w(d) {};
   double Wert(int k) {return k*w;}
   double Wert(double d) = delete;
private:
   double w;
};
```

```
int main() {
FVec f(1.1);
cout << f.Wert(8) << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

41

Unterbinden von Operationen

C++-11

```
class  FVec {
public:
  FVec(double d) : w(d) {};
   double Wert(int k) {return k*w;}
   double Wert(double d) = delete;
private:
   double w;
};
```

```
int main() {
FVec f(1.1);
cout << f.Wert(8.0) << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Unterbinden von Operationen

C++-11

```
class  FVec {
public:
  FVec(double d) : w(d) {};
   double Wert(int k) {return k*w;}
   double Wert(double d) = delete;
private:
   double w;
};
```

```
int main() {
FVec f(1.1);
cout << f.Wert(8.0) << endl;
}
```

Bildschirmausgabe ?

1. Compilerfehler; darf nicht mit double aufgerufen werden
2. Compiler Warnung
3. 8.0
4. 8.8

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Programmierparadigmen von C++