BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# C/C++ für Java-Programmierer
# - Klassen und dynamische Allokation -

Bachelor Medieninformatik

Prof. Dr.-Ing. Hartmut Schirmacher
http://schirmacher.beuth-hochschule.de
hschirmacher@beuth-hochschule.de

# Lernziele dieser Unterrichtseinheit

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Wiederholung: Klassen und Operatoren
- Referenzen und const-Referenzen
- Dynamische Objekte: new und delete
- Die „Wichtigen Vier“ Operatoren
- Weitere Beispiele für Operatoren
- Operatoren und Typumwandlung

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Wiederholung

# Eine Klasse mit Operatoren in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Vec3 {
  float m_x, m_y, m_z;

public:
  void Vec3(float x=0, float y=0,
            float z=0);

  float x() const;
  float y() const;
  float z() const;

  Vec3 operator+(Vec3 rhs) const;
  Vec3 operator-(Vec3 rhs) const;
};
```

```
void useVec3() {
  Vec3 a(1,2,3), b(4,5,6);
  Vec3 c = b-a;
  cout << c.x() << "/" << c.y()
       << "/" << c.z() << endl;
}
```

# Eine Klasse mit Operatoren in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
void Vec3::Vec3(float x, float y, float z)
  : m_x(x), m_y(y), m_z(z)
{}

float Vec3::x() const { return m_x; }

Vec3 Vec3::operator+(Vec3 rhs) const {
   Vec3 r(x()+rhs.x(), y()+rhs.y(), z+rhs.z());
   return r;
};

...
```

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Referenzen und `const`-Referenzen

# Call by Reference mit Zeigern

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
float compute (const ClassA* a, const classB* b) {
  return a->xy() / b->calcF();
}

int main() {
  ClassA myA(...);
  ClassB myB(...);
  float f = compute(&myA, &myB);
  ...
}
```

- Zeiger können verwendet werden, um Kopien bei der Parameterübergabe zu vermeiden
- Unschön: der ***Aufrufer*** muss das berücksichtigen und den Adressoperator & verwenden

# Call by Reference mit Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
float compute (const ClassA& a, const classB& b) {
  return a.xy() / b.calcF();
}

int main() {
  Class A myA(...);
  Class B myB(...);
  float f = compute(myA, myB);
  ...
}
```

- Eine Referenz ist ein ***Alias*** (anderer Name) für ein Objekt
  - Alle Operationen auf dem Alias werden stattdessen auf dem referenzierten Objekt ausgeführt
- Deklaration wie ein Zeiger, mit & anstelle von *
- Der ***Aufruf*** sieht so aus wie bei *Call by Value*

# Zeiger vs. Referenzen: Deklaration, Zuweisung, Semantik

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
int i=5;
int* p = i;
int x = *p;
*p = 2;
```

```
int i=5;
int& r = i;
int x = r;
r = 2;
```

Integer `i` mit Wert 5.

Pointer bzw. Referenz auf `i`.

`x` erhält den Wert von `i`.

`i` wird der Wert 2 zugewiesen.

```
p++;
```
erhöht Zeigerwert

```
r++;
```
erhöht Wert von `i`

D.h. der Wert einer Referenz kann nach der Initialisierung nicht mehr verändert werden!

```
int* pp;
```
uninitialisierter Zeiger

```
int& rr;
```
Compilerfehler – eine Referenz muss immer initialisiert sein

# Praktische Verwendung von Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Ray {
  Vec3 m_origin, m_direction;

public:
  Ray(Vec3 orig, Vec3 dir);

  Vec3 origin()    const { return m_origin; }
  Vec3 direction() const { return m_direction; }
};
```

Was ist hier nicht optimal?

- Obwohl `Vec3` ein relativ großes Objekt ist, wird es überall *by value* übergeben.

# Praktische Verwendung von Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Ray {
  Vec3 m_origin, m_direction;

public:
  Ray(const Vec3& orig, const Vec3& dir);

  const Vec3& origin()    const { return m_origin; }
  const Vec3& direction() const { return m_direction; }
};
```

Verbesserung durch const-Referenzen:

- Jetzt wird nur noch mit Referenzen auf die Objekte gearbeitet
  - D.h. intern werden nur Zeiger kopiert, nicht ganze Objekte
- Dennoch bleibt es die **gleiche Semantik wie bei *by-value***:
  - Konstruktor kann die übergebenen Objekte nicht verändern
  - Über Getter kann nur lesend zugegriffen werden

# Was passiert bei Rückgabe einer const-Referenz?

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Ray {
   ...
   const Vec3& origin() const { return m_origin; }
};


Ray r;

const Vec3& o1 = r.origin();
```

**Aufrufer speichert eine Referenz, hier wird kein Vektor kopiert**

```
Vec3 o2 = r.origin();
```

**Aufrufer speichert einen Vec3. Die Methode liefert eine Referenz, aber bei der Zuweisung (=) wird dann kopiert.**

# Referenzen: was man nicht tun sollte...

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
Vec3 mult_comp(Vec3 a, Vec3 b){
  Vec3 r(a.x()*b.x(),
         a.y()*b.y(),
         a.z()*b.z());
  return r;
}
```

- Dies ist eine normale Funktion, die mit `Vec3` rechnet...
- Könnte Sie genauso mittels `const&` optimiert werden?!

# Referenzen: was man nicht tun sollte...

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

OK!

```
~~const~~ Vec3~~&~~ mult_comp(const Vec3& a, const Vec3& b){
  Vec3 r(a.x()*b.x(),
         a.y()*b.y(),
         a.z()*b.z());
  return r;
}
```

**r** ist eine automatische / lokale Variable;
die Referenz zeigt nach Aufruf von `fr()` in
ungültigen Speicherbereich auf dem Stack.

- Vorsicht – für eine Referenz gilt das gleiche wie für einen Zeiger: „exportiere“ niemals Referenzen auf Stack-Objekte
- Bei Referenzen sieht man es dem Code in der Methode / Funktion nicht an, dass potentiell etwas „böses“ geschehen wird!

# Getter: eine `const`-Methode liefert auch `const`-Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Ray {
   ...
   Vec3& origin() const { return m_origin; }
   ...
};
```

binding of reference to type 'Vec3' to a value of type 'const Vec3' drops qualifiers
Vec3& origin() const { return m_origin; }
^~~~~~~~

```
const Ray r(...);
Vec& o = r.origin();
```

**Jetzt könnte ich `o = ...` schreiben!**
**Das verhindert der Compiler.**

# Getter: eine `const`-Methode liefert auch `const`-Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Ray {
   ...
   const Vec3& origin() const { return m_origin; }
   ...
};
```

```
const Ray r(...);
const Vec& o = r.origin();
Vec3 o2 = r.origin();
```

⟵ So kann nur lesend zugriffen werden!

⟵ z.B. in einen neuen Vec2 kopieren

- Wenn eine Methode `const` ist, darf Sie auch nur `const`-Referenzen auf Ihre Attribute zurückliefern

# Beides ist erlaubt: Setter vs. Getter



```
class Ray {
   ...
   const Vec3& origin() const { return m_origin; }
         Vec3& origin()       { return m_origin; }
   ...
};

Ray r(...);
Vec3 o = r.origin();
r.origin() = Vec3(0,0,0);
```

← Lesezugriff über den Getter

← Schreibzugriff über den Setter

- Es ist durchaus in einigen Fällen üblich, Accessor und Mutator (bzw. Setter und Getter) unter gleichen Namen bereitzustellen.
- Für C++ gehört das `const` zur Signatur der Funktion, d.h. Accessor und Mutator können formal unterschieden werden.

# Referenzen als Member-Variablen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class X {
  float& m_f;

public:
  X(float &f) : m_f(f) {}

};
```

- Member-Variablen können auch Referenzen sein
- Vorsicht bei der Initialisierung:
  - Einer Referenz kann man nachträglich keinen Wert zuweisen
  - Daher muss die Referenz ***direkt in der Initialisierungs-Liste*** des Konstruktors gesetzt werden

# Referenzen vs. Zeiger

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
void swap(int& a, int& b) {
    int tmp = a; a=b; b=tmp;
}

int main() {
  int a=3, b=4;
  swap(a,b);
}
```

```
void swap(int* a, int* b) {
    int tmp = *a; *a=*b; *b=tmp;
}

int main() {
  int a=3, b=4;
  swap(&a,&b);
}
```

Welche Variante finden Sie besser?

# Vergleich von Objekten mittels Zeigern und Referenzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
MyObj* o = ...;
MyObj* p = ...;
...
if(o == p) {

}
```

***Zeiger*** sind Objekte, die Speicheradressen als Wert enthalten. Hier wird der Operator == auf zwei Zeigertypen angewendet. Dieser Operator kann nicht überladen werden, und vergleicht immer die Adressen der Objekte.

```
MyObj& o = ...;
MyObj& p = ...;
...
if(o == p) {

}
```

***Referenzen*** verhalten sich wie Aliase auf Objekte. Also wird der Operator **==** auf zwei Objekte vom Typ **`MyObj`** angewendet: die Werte der Objekte werden gemäß der Implementierung von `operator==()` verglichen.

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Konstruktoren & Destruktoren, `new` & `delete`

# Klassen auf dem Heap

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Bisher
  - Klassen nur als automatische Objekte auf dem Stack
  - Klassen enthalten nur Objekte, keine Zeiger

- Jetzt
  - Dynamische Allokation von Instanzen einer Klasse, Äquivalent zu `malloc()`
  - Dynamische Speicherallokation innerhalb einer Klasse
  - Zeiger in Klassen

# Konstruktoren: Anwendung und new

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Konstruktoraufruf für automatische Variable:

```
Vec3 v1;
Vec3 v2();
Vec3 v3(1,2,3);
```

← Aufruf des Konstruktors ohne Parameter

← **Achtung!**

**v2** ist eine Funktion ohne Parameter, die ein `Vec3` zurückliefert...

- Konstruktoraufruf bei dynamischem Objekt:

```
Vec3 *vp1 = new Vec3;
Vec3 *vp2 = new Vec3();
Vec3 *vp3 = new Vec3(1,2,3);
```

hier arbeiten wir stets mit Zeigern ↑

↑ Der Operator `new()` allokiert den Speicher für ein Objekt dynamisch auf dem Heap und ruft dann den entsprechenden Konstruktor auf.

# Der Operator **new**

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Das „einfache“ **new**
  - **new** *Typ-Bezeichner* **(***Konstruktor-Argumente***)**
  - `Vec3* ptr = new Vec3;`
  - `Vec3* ptr = new Vec3(1.0, 3.0, 5.0);`

- **new** erfüllt zwei Aufgaben:
  - Es reserviert Speicher für den Datentyp (hier: `Vec3`) auf dem Heap, wie zuvor mit `malloc()`
  - Es ruft immer einen Konstruktor auf:
    - entweder einen vom Benutzer definierten Konstruktor
    - oder einen vom Compiler erzeugten Default-Konstruktor (ohne Argumente)
  - Der Default-Konstruktor initialisiert alle Member-Variablen mittels deren jeweiliger Default-Konstruktoren

# Der Operator `new[]` für Arrays

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- *„array new“:* **`new[]`** für Arrays
  - `new` *Typ-Bezeichner* `[`*Anzahl*`]`
  - `Vec3* ptr = new Vec3[6];`

- **`new[]`** funktioniert wie die „einfache“ Version, aber für mehrere Elemente, die unmittelbar aufeinander im Speicher folgen.
  - Es ***reserviert Speicher*** für *(N mal Datentyp )* (hier: `6*sizeof(Vec3)`) *en bloc* auf dem Heap
  - Es ruft für jedes Element in dem resultierenden Array den ***Default-Konstruktor*** auf. D.h. primitive Datentypen werden i.d.R. ***nicht initialisiert***.
  - Andere Konstruktoren können nicht verwendet werden - es ist in ISO-C++ nicht erlaubt, bei *array new* ***Konstruktor-Argumente*** anzugeben.

# Placement **new** und Exceptions

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- Placement **new**
  - **new**(*Placement-Argumente*) *Typ-Bezeichner* (*Konstruktor-Argumente*)
  - `Vec3* ptr = new(MyArena) Vec3;`

- Implementieren eigener **new**-Varianten
  - Das Definieren eigener Placement-`new`-Operatoren erlaubt es, explizit zu kontrollieren, in welchen Speicherbereichen die Allokation für bestimmte Typen stattfindet (→ für Fortgeschrittene)

- Exceptions
  - Standardmäßig werfen `new` und `new[]` eine `bad_alloc`-Exception, wenn der Speicher nicht allokierbar ist
  - Der Placement-Operaor **new(nothrow_t)** ist eine Exception-freie Variante
  - `Vec3* ptr = new(std::nothrow) Vec3;`

http://en.wikipedia.org/wiki/Placement_syntax
http://stackoverflow.com/questions/222557/cs-placement-new

# Initialisierung und Default-Konstruktor (1)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Der vom Compiler bereitgestellte Default-Konstruktor von ***primitiven Typen*** (wie `int`, `float`, `char`, `T*`) initialisiert *nicht* den Wert des Objekts.

- `int i; int j(3);`
  - Der Wert von **`i`** ist undefiniert; der von **`j`** ist 3.
- `int i[5];`
  - Fünf aufeinanderfolgende uninitialisierte Werte
- `int* ip = new int;`
  - Potentiell uninitialisiertes **`int`**-Objekt auf dem Heap.
- `int* ip = new int[size];`
  - **`size`** aufeinanderfolgende uninitialisierte Integer auf dem Heap

http://stackoverflow.com/questions/563221/is-there-an-implicit-default-constructor-in-c
http://stackoverflow.com/questions/1613341/what-do-the-following-phrases-mean-in-c-zero-default-and-value-initializat

# Initialisierung und Default-Konstruktor (2)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Der vom Compiler bereitgestellte Default-Konstruktor von ***Klassen*** initialisiert jede Member-Variable einzeln mittels des jeweiligen Default-Konstruktors.

```
class X {
 int i, j;
public:
  X() : i(5), j(7) {}
};
```

```
class Y {
  X x;
  float a;
public:
};
```

- `X myX;`
  - Der Default-Konstruktor wurde vom Entwickler überschrieben. Daher **`myX.i=5`** und **`myX.j=7`**.
- `Y myY;`
  - **`myY.x`** wird initialisiert, **`myY.a`** nicht.

# Konstruktoren und `new` für primitive Datentypen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

## Weitere Punkte bzgl. Initialisierung von Objekten

- Der Bereich für ***statische und globale Daten*** wird mit dem Bitmuster 0 initialisiert.
- C++98 und C++03 haben unterschiedliche Definitionen für die Initialisierung von Daten.
- Compiler implementieren unterschiedliche Modelle der Initialisierung.

→ Guter Stil = ***Attribut-Werte immer explizit initialisieren***.

http://stackoverflow.com/questions/1613341/what-do-the-following-phrases-mean-in-c-zero-default-and-value-initializat

# Destruktoren und `delete`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Analog zum ***Konstruktor*** hat jedes Objekt auch einen ***Destruktor***.
  Er wird am Ende der Lebenszeit des Objekts aufgerufen:
  - Für globale Objekte bei der Terminierung des Prozesses
  - Für automatische Objekte am Ende ihres Code-Blocks
  - Für dynamische Objekte durch den expliziten Aufruf von **`delete`**

Live-Programmierbeispiel

```
int main()
{
    cout << "Hello World!" << endl;

    MyClass* p = new MyClass();
    MyClass m;
    f();
    return 0;
}
```

# Aufgaben von Destruktoren, `delete[]`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Der Destruktor dient vor allen Dingen dazu, Ressourcen wieder freizugeben, die von der Klasse allokiert wurden

```
class X {
  float* m_data;
  int m_size
public:
  X(int size) {
    m_data = new float[size];
  }
  ~X() {
    delete[] m_data;
  }
};
```

← **`new[]` allokiert ein Array**

← **`delete[]` gibt ein Array frei.**
**Die Größe hat sich der Heap gemerkt.**

# `new` und `delete`, `new[]` und `delete[]`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- `new` und `delete` treten immer in Paaren auf
- Jedes mittels `new` allokierte Objekt muss auch irgendwann mit `delete` wieder freigegeben werden.
- Wird `new[]` für die Allokation verwendet, muss auch `delete[]` für die Freigabe verwendet werden (nicht das `delete` ohne Klammern!)

- Diese Anforderungen werden in hervorragender Weise durch die Paarung Konstruktor + Destruktor unterstützt. Eine Klasse kann *Ownership* über eine Speicherressource übernehmen.

**Hierzu später mehr (RAII Ressourcenverwaltung mit Klassen)!**

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Die Wichtigen Vier Operatoren einer Klasse

# Die Wichtigen Vier

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Der Compiler stellt Operatoren ***automatisch*** zu Verfügung:
  - Default-Konstruktor
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

Diese Standard-Operatoren funktionieren „member by member“: der entsprechende Operator wird hintereinander für jede Member-Variable ausgeführt.

# Wo die Default-Operatoren versagen (1)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class MyString {
    char* m_ptr;
    int   m_len;
  public:
    MyString(const char *s = "") {
      m_len=strlen(s);
      m_ptr = new char[m_len+1];
      ...
    }
};
```

```
{
  MyString a = new MyString("Hallo");
}
```

hier wird der Destruktor von a aufgerufen.
Aber er gibt den Speicher nicht frei.

# Wo die Default-Operatoren versagen (2)

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

```
class MyString {
    char* m_ptr;
    int   m_len;
  public:
    MyString(const char *s = "") {
      m_len=strlen(s);
      m_ptr = new char[m_len+1];
      ...
    }
    ~MyString() { delete[] m_ptr; }
};
```

```
MyString a = new MyString("Hallo");
{
  MyString b;
  b=a;
}
```

Zuweisungsoperator: hier wird `m_ptr` einfach kopiert

hier wird der Destruktor von b aufgerufen.
Er gibt den Speicher frei – aber leider auch den von a!

# Zuweisung und Copy-Konstruktor (1)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Zuweisung mittels =

```
MyString& MyString::operator=(const MyString& rhs)
{
  if(ptr)
    delete[] m_ptr;

  m_len = rhs.m_len;
  m_ptr = new char[m_len+1];
  memcpy(m_ptr, rhs.m_ptr, m_len+1);

  return *this;
}
```

Nicht vergessen: bei Zuweisung immer erst die eigenen (alten) Ressourcen freigeben!

Einfache Möglichkeit, Speicher zu kopieren*

Für Verkettung von Operatoren: Zuweisungs-Operationen sollten eine Referenz auf das Objekt zurückliefern.

*) `memcpy()` ist in stdlib.h deklariert

# Zuweisung und Copy-Konstruktor (2)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- **Copy-Konstruktor**
  Erstelle ein Objekt als Kopie eines anderen

```
MyString a("Hallo");
MyString b(a);
```

```
MyString::MyString(const MyString& rhs)
  : m_len(rhs.m_len), m_ptr(new char[m_len+1])
{
  memcpy(m_ptr, rhs.m_ptr, m_len+1);
}
```

- Schlaue / alternative Implementierung:

```
MyString::MyString(const MyString& rhs)
  : m_len(0), m_ptr(0)
{
  *this = rhs;
}
```

← Verwendet den Zuweisungsoperator

# Zuweisung und Copy-Konstruktor – Hinweise (2)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Best Practices bzgl. der Standard-Operatoren:
  - Sobald eine Klasse Zeiger oder Ressourcen-Handles beinhaltet, definiere eigene Operatoren für Copy und Assignment
  - Oder mache die entsprechenden Operatoren `private`, so dass sie nicht aus Versehen angewendet werde

- Compiler können einige Operatoren-Aufrufe wegoptimieren

```
Vec2 i(1,3);
Vec2 j = i;
```

← hier steht eigentlich:

```
Vec2 j(); j = i;
```

← Der Compiler macht daraus:

```
Vec2 j(i);
```

1. Default-Konstruktor
2. Zuweisungsoperator

Reduziert auf Copy-Konstruktor.

# Index-Operator / Subscript-Operator (`operator[]`)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Der Index-Operator wird z.B. bei dynamischen Arrays und ähnlichen Containertypen überladen
- Auch bei Vektoren wird der Index-Operator oft für die Adressierung der einzelnen Elemente verwendet
  - x = Element 0, y = Element 1, ...
  - Verallgemeinerung für Vektoren beliebiger Dimension / Länge
- Unterscheide Lesezugriff von Lese- und Schreibzugriff

```
const float& Vec2::operator[](int i) const {
    return m_data[i];
}
float& Vec2::operator[](int i) {
    return m_data[i];
}
```

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Noch mehr Operatoren

# Arithmetische und kombinierte Zuweisungsoperatoren (1)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Subtraktion als ***Methode***

```
Vec2 Vec2::operator-(const Vec2 &rhs) const {
  return Vec2(x()-rhs.x(), y()-rhs.y());
}
```

```
Vec2 l(1,2);
Vec2 r(3,4);
Vec2 i = l - r;
i -= l;
i = l + (i-=r);
```

- Subtraktion als ***freie Funktion***

```
Vec2 operator-(const Vec2& lhs, const Vec2& rhs) {
  return Vec2(lhs.x()-rhs.x(), lhs.y()-rhs.y());
}
```

- Zuweisung ***kombiniert*** mit Subtraktion

```
Vec2& Vec2::operator-=(const Vec2& rhs) {
  x() -= rhs.x();
  y() -= rhs.y();
  return *this;
}
```

**Dies ist die zweitbeste Lösung!**

Kombinierte Ausdrücke (mit Zuweisung) ***verändern den Wert*** des „linken“ Objekts und ***liefern `*this` zurück***, um Verkettung von Ausdrücken zu ermöglichen.

# Arithmetische und kombinierte Zuweisungsoperatoren (2)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Zuweisung ***kombiniert*** mit Subtraktion

```
Vec2& Vec2::operator-=(const Vec2& rhs) {
  x() -= rhs.x();
  y() -= rhs.y();
  return *this;
}
```

```
Vec2 l(1,2);
Vec2 r(3,4);
Vec2 i = l - r;
i -= l;
i = l + (i-=r);
```

- Subtraktion ***verwendet*** kombinierten Operator

```
Vec2 Vec2::operator-(const Vec2 &rhs) const {
  Vec2 result(*this);
  return result -= rhs;
}
```

`operator-=()` wird ***auf ein temporäres Objekt angewendet***, dessen Wert dann zurückgeliefert wird.

- Bei komplexeren Operationen ist so auch bei Code-Änderungen sichergestellt, dass `-` und `-=` sich konsistent verhalten.
- Genauso effizient wie die erste Lösung

# Inkrement- und Dekrement-Operatoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Inkrement und Dekrement gibt es jeweils als Präfix- und Postfix-Operator:
  `A++, A--, ++A, --A`

- Wie unterscheide sich die Methoden für Prä- und Postfix?

```
MyFloat& MyFloat::operator++() {
    m_value += 1.0f;
    return *this;
}
```

← Präfix-Operator

← Verändert das Objekt, ***liefert das veränderte Objekt zurück***.

```
MyFloat MyFloat::operator++(int) {
    MyFloat f(*this);
    m_value += 1.0f;
    return f;
}
```

← Postfix-Operator wird durch ***Dummy-Parameter vom Typ*** ***`int`*** gekennzeichnet.

Liefert den ***vorherigen Wert*** des Objekts zurück.

# Index-Operator / Subscript-Operator (`operator[]`)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Als Parameter für den Index-Operator können beliebige Typen verwendet werden (z.B. auch Strings)
- Damit werden z.B. gerne assoziative Arrays implementiert

```
const ValueT& ObjT::operator[](const KeyT key) const {
    return searchFor(key);
}
```

# Shift-Operatoren (operator<< / operator>>)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Die Shift-Operatoren werden bei den primitiven Typen für Bitmuster-Manipulationen verwendet (alle Bits werden um N Stellen nach links / rechts geschoben).
- In der C++ - Standardbibliothek wurde die Konvention entwickelt, << und >> für das Schreiben und Lesen in/aus Streams zu verwenden.
- Der Stream ist das „linke“ Objekt. Wir können dem Stream aber keine Methode hinzufügen!
  - → Operator als freie Funktion implementieren

```
#include <iostream>

std::ostream&
operator<<(std::ostream &lhs, const RationalNumber &rhs) {
    lhs << "(" << rhs.x() << "," << rhs.y() << ")";
    return lhs;
}
```

Auch bei Streams immer eine ***Referenz auf den Stream zurückliefern***, um verkettete Ausdrücke zu erlauben.

```
x << a << b << "!" << endl;
```

# Weiterführende Themen beim Operator Overloading

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Funktionsaufruf-Operator `()`
  - später bei den C++ - Standardbibliotheken
- Dereferenzierung und Member-Zugriff (`*` und `->`)
  - später bei Smart Pointers!
- `new` und `delete` → eigene Speicherverwaltung
  - wahrscheinlich nicht mehr in diesem Kurs ☺

http://courses.cms.caltech.edu/cs11/material/cpp/donnie/cpp-ops.html
http://www.parashift.com/c++-faq-lite/operator-overloading.html

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

# Operatoren und Typumwandlung

# Typumwandung: Konstruktoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Umwandlung `float` → `FuzzyFloat`

- implementiere `FuzzyFloat`-Konstruktor, der einen `float`-Wert als Argument akzeptiert
- 
```
FuzzyFloat::FuzzyFloat(float v)
   : m_value(v), m_min(v), m_max(v)
{ }
```

- Damit werden die folgenden Ausdrücke ermöglicht:

```
FuzzyFloat f(3.0f);
```
← expliziter Aufruf des Konstruktors

```
FuzzyFloat f = 3.0f;
```
← Es wird rechts implizit ein temporäres `FuzzyFloat`-Objekt konstruiert, dann Zuweisung zu `f`.

```
FuzzyFloat f = 3.0;
```

```
FuzzyFloat f = 3;
```
← Es zunächst von `double`/`int` nach `float` umgewandelt, dann ein temporäres `FuzzyFloat`-Objekt..., dann Zuweisung zu `f`.

# Typumwandung: Probleme mit Konstruktoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Vorsicht mit diesen „Cast-Konstruktoren“!
- Beispiel: eine String-Klasse, deren Größe man initial angeben kann

```
class String {
public:
  // empty string, but pre-allocate n bytes
  String (int n);
  // initialize string from C-string ptr
  String(const char* ptr);
};
```

- `String s1 = "hello world!";`

  Der zweite Konstruktor wird aufgerufen, der C-String kopiert.

- `String s = 'x';` ← was passiert hier?

  `'x'` ist vom Typ `char`, wird implizit nach `int` umgewandelt. Daher wird der erste Konstruktor aufgerufen, und `s2` ist ein leerer String anstelle von `"x"`.

# Typumwandung: `explicit`-Konstruktoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Konstruktoren mit einem Argument können wegen der impliziten Typumwandlung schnell zu Verwirrung führen.
- Man möchte i.d.R. verhindern, dass jeder `int` „einfach so“ in ein Objekt umgewandelt wird
- Lösung: `explicit`

```
class String {
  public:
    explicit String(int n);
};
```

- Wie kann man den Konstruktor nun aufrufen?

```
String s = 'x';
```
← Compiler-Fehler!

```
String s = 42;
```
← Compiler-Fehler!

```
String s = String(42);
```
← man muss den Konstruktor ***explizit*** aufrufen

```
String s(42);
```
← noch besser / direkter

# Typumwandlung: Cast-Operator

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Umwandlung `FuzzyFloat` → `float`

- Wir können dem Typen `float` keinen neuen Konstruktor hinzufügen.
- Aber es gibt die Typumwandlungs-Operatoren:

```
FuzzyFloat::operator float() const {
   return m_value;
}
```

- Funktioniert für beliebige Typen, auch selbstdefinierte.
- Wie bei Konstruktoren wird kein Rückgabetyp angegeben.
- Wie bei Konstruktoren findet die Typumwandlung potentiell implizit statt, aber es gibt leider für diesen Fall kein `explicit`...

```
FuzzyFloat ff(3.0,2.9.3.1);
float f = ff;
```

← `FuzzyFloat` wird implizit in `float` umgewandelt. Viele Entwickler bevorzugen daher eine explizite „sprechende“ Methode **`toFloat()`**.