BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

C/C++ für Java-Programmierer
- Klassen und Operatoren -

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

- Was fehlt uns noch für die Objektorientierung?
- Klassen, Methoden, Member-Variablen, und Co.
- Konstruktoren und Initialisierung
- `const`- und `static`-Methoden
- Umwandlung von einfachem C++-Code in C-Code
- Überladen einfacher Operatoren

In dieser SU: „einfache“ Klassen und Operatoren.
Später: Klassen mit eigener Ressourcenverwaltung.

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Wiederholung: dynamische Speicherverwaltung, `const`-Zeiger

Dynamische Allokation auf dem Heap mittels `malloc()`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Allokiere Speicher für N Objekte vom Typ T:

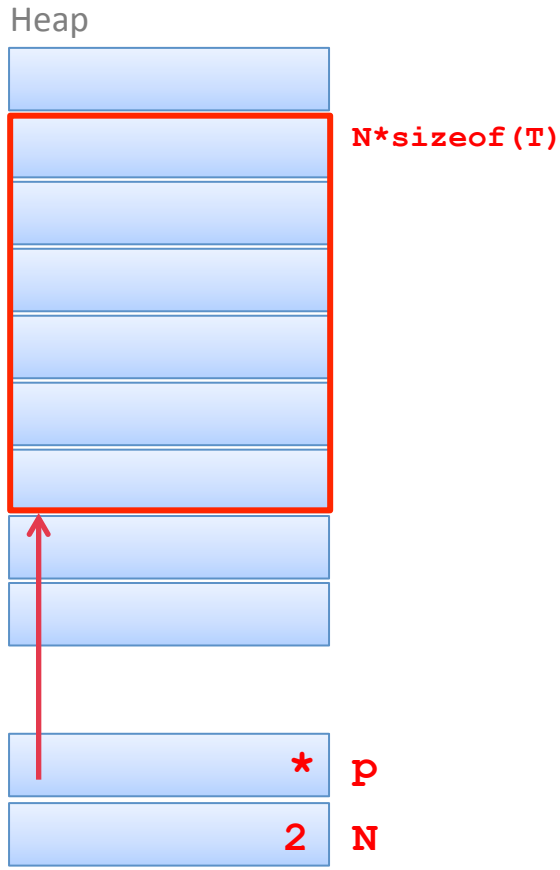
```
T* p = (T*) malloc(N*sizeof(T));
```

- Verwende danach den Zeiger wie ein Array:

```
p[0] = ...;
```

- Nach Benutzung muss der Speicher wieder freigegeben werden, sonst „Speicherloch“:

```
free(p);
```




```
struct Point3D {
  float x, y, z;
};
```

Zugriffskontrolle mittels `const`

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- Wo macht `const` was?

```
int objX(Obj* obj);
int objX(const Obj* obj);
int objX(Obj const* obj);
int objX(Obj* const obj);
int objX(Obj const* const obj);
```

Nichts const — gebräuchlich

Objekt ist const — gebräuchlich

Objekt ist const — eher ungebräuchlich

Zeiger ist const — ungebräuchlich

Zeiger ist const + Objekt ist const — ungebräuchlich

Regel:	Das, was ***links von const*** steht, ist unveränderbar.
Ausnahme:	Wenn const ***ganz links*** steht, ist das ***Objekt*** unveränderbar.

Von `struct` zu `class`

Objektorientierte (OO) Programmierung – in C?

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Grundidee der OO-Programmierung:
  - Fasse Daten und Methoden in Objekten zusammen
  - Veränderungen der Objekte nur über deren Methoden

Siehe auch http://de.wikipedia.org/wiki/Objektorientierte_Programmierung

Objektorientierte (OO) Programmierung – in C?

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- **Wichtigste OO-Prinzipien**
  (frei nach wikipedia.de)
  - Abstrakter Datentyp
  - Kapselung
  - Geheimnisprinzip
  - Module / Pakete
  - Persistenz des Objekts
  - Schnittstellen
  - Polymorphie
  - Vererbung

- **Umsetzung in C**
  (soweit bisher besprochen)
  - `struct`
  - Funktionen, die auf `struct` operieren
  - verdeckte Implementierung
  - Modul = Header + Implementierungsdatei Paket?
  - dynamische Allokation, Call by Reference
  - Schnittstellen ohne Attribute? Verdeckter Datentyp!
  - Die gleiche Methode in verschiedenen Objekten verschieden implementieren, zur Laufzeit entscheiden?
  - Wie kann ich Hierarchien von Objekt-Schablonen realisieren?

Mit Zeigern geht alles... aber es wird unhandlich.

Stroustrup's *C with Classes*

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- 1980 gab es mit Simula-67 und Smalltalk bereits zwei bekannte objektorientierte Programmiersprachen mit Klassen und Vererbung
- Bjarne Stroustrup hatte als Student Simula programmiert; bei AT&T entwickelte er in C
- Er implementierte 1983 zunächst einen ***Präprozessor*** (cfront), um C um die wichtigsten OO-Konstrukte zu erweitern
  - Lange Zeit konnten alle Konstrukte von C++ mittels des Präprozessors direkt in C übersetzt werden
  - 1987 / 1988 erste C++-Compiler: gnu-c++, Oregon C++, Zortech C++
  - 1993 scheiterte das weitere Ausbauen des Präprozessor-Ansatzes (cfront 4.0) bei der Implementierung von *Exceptions*
- Stroustrup nannte die neue Sprache zunächst *C with Classes*
- Wurde später zu C++ umbenannt und standardisiert
- Wurde neben Klassen um viele andere Konzepte erweitert

Siehe auch: http://www.softwarepreservation.org/projects/c_plus_plus

Designziele bei der Einführung von Klassen in C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.

B. Stroustrup, The C++ Programming Language, Special Edition, Addison Wesley 2000

Those types are not „abstract“; they are as real as `int` and `float`.

Doug McIlroy

Ein Datentyp `Vec3` in C

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
struct Vec3 {
  float x,y,z;
};
```

freier Zugriff von außen
auf alle Attribute

```
Vec3  vec3Add(Vec3 lhs, Vec3 rhs);
Vec3  vec3Sub(Vec3 lhs, Vec3 rhs);
float Vec3Dot(Vec3 lhs, Vec3 rhs);
...
```

Funktionen operieren mit Objekten vom Typ **`Vec3`**, entweder *by value* oder *by reference* (mit Zeigern).

Anmerkung zur Notation:
`lhs` = left hand side
`rhs` = right hand side

vec3.h: Eine Datentyp **Vec3** in C++

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Attribute oder ***Member Variables**** →

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

← wenn nicht anders definiert, sind Attribute + Methoden `private`, Zugriff nur von innerhalb der Klasse*!

Freier **Zugriff** von außen →

```
public:
  float x() const { return m_x; }
  float y() const { return m_y; }
  float z() const { return m_z; }
```

Methoden oder ***Member Functions*** →

```
  Vec3  add(Vec3 rhs) const;
  Vec3  sub(Vec3 rhs) const;
  float dot(Vec3 rhs) const;
    ...
};
```

← Semikolon nicht vergessen!

*) das **m_** ist lediglich eine *Konvention* zur Kennzeichnung von Membervariablen. Es verhindert u.a Namenskonflikte zwischen Methoden und Membervariablen

Klassen: Header- vs. Implementierungsdatei

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
#inlucde "vec3.h";
```
← Klasse wird gewöhnlich in Header-Datei definiert (offen).

In der `.cpp`-Datei bzw. außerhalb der Klassendefinition muss dem Methodenname der Name der Klasse (**`Vec3::`**) vorangestellt werden → *fully qualified name.*

```
Vec3 Vec3::add(Vec3 rhs) const {
  Vec3 result;
  result.m_x = this->m_x + rhs.m_x;
  result.m_y = m_y + rhs.m_y;
  ...
  return result;
};
```

Der Bezeichner **`this`** kann (lesend) wie ein Zeiger auf das aktuelle Objekt verwendet werden.

Die Attribute (Members) des aktuellen Objekts sind auch **ohne `this`** zugreifbar.

Inline-Implementierung in der Klassendefinition

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Implementierung direkt in der Klassendefinition:
„inline“-Code

vec3.h

```
class Vec3 {
   ...
   float x() const { return m_x; }
};
```

„inline“ bedeutet noch viel mehr: der Compiler wird versuchen, den Code überall, wo `Vec3::x()` aufgerufen wird, direkt einzusetzen, anstatt einen echten Funktionsaufruf zu tätigen. Dazu später mehr.

Non-Inline-Code außerhalb der Klassendefinition

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

„Normale“ Implementierung in der separaten Implementierungsdatei

vec3.h

```
class Vec3 {
   ...
   float x() const;
};
```

vec3.cpp

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

Saubere Trennung von Schnittstellendeklaration und Implementierung.

Später: `inline`-Definition *ausserhalb* der Klassendefinition

const-Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

const-Methoden dürfen nicht den *Zustand des Objekts* verändern.
→ ***Accessor* bzw. *Getter***

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

public:
  float x() const { return m_x; }
  float y() const { return m_y; }
  float z() const { return m_z; }
  Vec3  add(Vec3 rhs) const;
  Vec3  sub(Vec3 rhs) const;
  float dot(Vec3 rhs) const;
    ...
};
```

```
this->m_x = 5;
```

wäre z.B. in einer solchen Methode nicht erlaubt!

Randbemerkung:
Mittels **mutable** können Members deklariert werden, die auch in const-Methoden verändert werden dürfen.

Definition von Konstruktoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Vec3 {

public:
   Vec3();
   Vec3(float x, float y, float z);
   ...
```

Konstruktoren

Konstruktoren
- initialisieren den Zustand des Objekts
- heißen so wie ihre Klasse
- werden immer ***ohne Rückgabetyp*** deklariert und definiert
- es kann mehrere ***polymorphe*** Konstruktoren geben

Wird ***kein*** Konstruktor explizit vom Entwickler definiert,
- stellt der Compiler einen „einfachen“ Default-Konstruktor zur Verfügung.
- Dieser ruft für jede Member-Variable jeweils den Default-Konstruktor auf.

Aufruf von Konstruktoren


```
class Vec3 {

public:
  Vec3();
  Vec3(float x, float y, float z);
  ...



int main() {

  Vec3 a;
  Vec3 b(1,2,3);
  Vec3 c = Vec3(4,5,6);

}
```

← **Aufruf des Default-Konstruktors**

← **Aufruf des alternativen Konstruktors**

← **Äquivalent mit dem vorherigen Aufruf**

Konstruktoren – Initialisierung von Member-Variablen

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

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

Dieser Konstruktor ist korrekt, aber nicht ganz optimal. Hier nicht ersichtlich: Alle Member-Variablen werden vor Eintritt in den Konstruktor implizit mit Ihren Default-Werten (hier 0) initialisiert. D..h. jede Member-Variable wird *zweimal* initialisiert.

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

Initialisierungs-Liste: So werden Member-Variablen direkt und nur einmal initialisiert.

Der Compiler initialisiert in der Reihenfolge der Deklaration, nicht in der der Liste – also am besten die Deklarations-Reihenfolge einhalten, um Verwirrungen zu vermeiden!

Statische Member-Variablen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Deklaration (`.h`)

```
class X {
  static int s_objCount;

public:
  X() { s_objCount++; }
};
```

Definition (`.cpp`)

```
int X::s_objCount = 0;
```

- Von statischen Klassenvariablen gibt nur eine einzige Instanz
  - nicht eine Instanz pro Objekt!
- Wie eine globale Variable, aber
  - im Namensraum der Klasse gekapselt;
  - Zugriff von außen kann kontrolliert / verhindert werden.
- Deklaration vs. Definition
  - Deklaration mit, Definition *ohne* das Wort `static`.

Statische Methoden

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class X {
public:
  // factory method
  static X* create(std::string);

  // ...
};
```

Aufruf:

```
X* obj = X::create("MyObjType");
```

- Statische Methoden sind keinem *Objekt* zugeordnet,
  - sondern lediglich dem Namensraum der Klasse.
  - In einer statischen Methode gibt es kein **`this`**
  - In einer statischen Methode kann nicht auf nicht-statische Member-Variablen zugegriffen werden

Default-Werte für Member-Variablen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Im wesentlichen konnte man in C++ bis vor kurzem ***keine*** Default-Werte für Member-Variablen direkt in der Klasse angeben.

```
class MyMath {

  static const float PI = 3.1415;
```

⟵ nicht Standard!

```
in-class initializer for static data member of type 'const float' is a GNU extension [-Wgnu-static-float-init]
    static const float PI = 3.1415;
                      ^   ~~~~~~~
```

```
  float x = 4.56;
```

⟵ Erst ab C++11, noch nicht in jedem Projekt verfügbar!

```
in-class initialization of non-static data member is a C++11 extension [-Wc++11-extensions]
    float x = 4.56;
              ^
};
```

const-Membervariablen in der Initialisierungsliste setzen

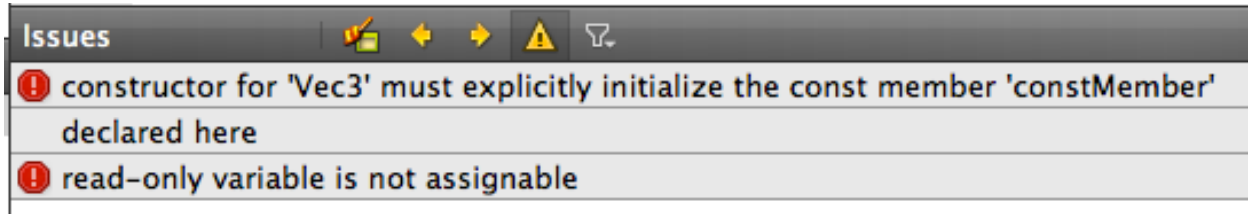
BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- const-Membervariablen können im Konstruktorcode nicht mehr verändert werden

```
class Vec3 {
  const float constMember;

public:

  Vec3() {
    constMember = 3.1415;
  };

};
```

Compilerfehler!

Bereits diese Zuweisung gilt als nachträgliche Veränderung.



const-Membervariablen in der Initialisierungsliste setzen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- const-Membervariablen können im Konstruktorcode nicht mehr verändert werden

```
class Vec3 {
  const float constMember;

public:

  Vec3()
    : constMember(3.1415)
  {}

};
```

← const-Membervariablen können nur direkt in der Initalisierungsliste initialisiert werden.

Default-Werte für Methoden-Parameter

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

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

public:

  Vec3() : m_x(0), m_y(0), m_z(0) {}

  Vec3(float x=0, float y=0, float z=0)
    : m_x(x), m_y(y), m_z(z) {}

};
```

Default-Konstruktor ohne Parameter
setzt alle Member-Variablen auf 0

Default-Werte für Parameter
definieren den Wert eines Parameters, wenn dafür kein Argument übergeben wird.
Bei der Deklaration angegeben, nicht jedoch bei separater Definition.

Weiterer Konstruktor
erlaubt das direkte Setzen aller Member-Variablen

Verwendung von Default-Parametern

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Vec3 {
  Vec3();
  Vec3(float x=0, float y=0, float z=0);
};
```

```
Vec3 a(1,2,3);
```
← **Vektor mit Werten 1,2,3**

```
Vec3 b(1,2);
```
← **Via Default-Parameter: `b.z()` == 0**

```
Vec3 c;
```
← **Welcher Konstruktor wird hier aufgerufen?**

call to constructor of 'Vec3' is ambiguous	main.cpp	43
candidate constructor	main.cpp	18
candidate constructor	main.cpp	19

Konflikt, den der Compiler nicht auflösen kann!
→ Default-Konstruktor entfernen.

Vorsicht beim Aufruf des Default-Konstruktors!

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Vec3 {
  Vec3(float x=0, float y=0, float z=0);
};
```

```
Vec3 a;
```
← OK

```
Vec3 c();
```
← Zweideutige Syntax!

```
empty parentheses interpreted as a function declaration [-Wvexing-parse]
    Vec3 c();
         ^~
```

Deklaration einer Funktion `c()` ohne Parameter, die einen `Vec3` zurückliefert...

```
float g = c.x();
```
← Hier kommt es dann zum Problem...

```
base of member reference is a function; perhaps you meant to call it with no arguments?
    float g = c.x();
              ^
```

```
Vec3 c = Vec3();
```
← Alternative, die den Konstruktoraufruf explizit / deutlicher macht.

Klassen und Namensräume

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Eine Klasse definiert für alle ihre Attribute und Methoden implizit einen Namensraum

```
class X {
  void f();
};

void X::f() {
  ...
}
```

- Namensräume werden mittels *Name*:: verwendet

Explizite Namensräume

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Deklaration (x.h)

```
namespace MySpace {
  class X {
  public:
    void A();
  };
}
```

Verwendung:

```
MySpace::X obj;
obj.A();
```

Definition (x.cpp)

```
#include "x.h"
namespace MySpace {
  void X::A() {
    ...
  }
}
```

- Namensräume kapseln alle enthaltenen Bezeichner
- Namensräume können hierarchisch geschachtelt werden.
- Im Nutzer-Code kann `using namespace` verwendet werden, um alle Namen aus einem Namensraum direkt verfügbar zu machen:

```
#include "x.h"
using namespace MySpace;
X obj1, obj2;
```

Nur für die Nutzer des Moduls, **nicht für die Definition im .cpp des Moduls selbst!**

Äquivalenz von C und C++

`struct` vs. `class`

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- In C ist die `struct` ein einfacher Container für mehrere Attribute ohne Zugriffs-Steuerung und Methoden

- In C++ wurde die `struct` „befördert“ und unterscheidet sich technisch nicht mehr von der `class`, bis auf eine formale Ausnahme:
  - Bei einer `class` sind Members standardmäßig `private`,
    bei der `struct` jedoch `public`

- Man könnte also anstelle von `class` einfach `struct` schreiben

- Konvention: Verwende `struct` nur für passive Container und ***Plain Old Datatypes*** *(POD)*
  - D.h. bei Verwendung von `struct` wird in der Regel direkt und
    ohne eigene Methoden auf die Attribute zugegriffen

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- Wie funktioniert (prinzipiell) ein C++-Präprozessor?

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

entsprechender Code in reinem C ?

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- `class` wird zu `struct`, nur Attribute

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

```
struct Vec3 {
  float m_x, m_y, m_z;
};
```

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Methoden werden zu Funktionen, `this` als Parameter

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

```
struct Vec3 {
  float m_x, m_y, m_z;
};

void Vec3_Vec3(Vec3* this, float x, ...) {
    this->m_x = x;
    ...
}
```

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Methoden werden zu Funktionen, `this` als Parameter

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

```
struct Vec3 {
  float m_x, m_y, m_z;
};

void Vec3_Vec3(Vec3* this, float x, ...) {
    this->m_x = x;
    ...
}
float Vec3_x(const Vec3* this) {
    return this->m_x;
}
```

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Konstruktor in C explizit nach Anlegen des Objekts aufrufen

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

```
struct Vec3 {
  float m_x, m_y, m_z;
};

void Vec3_Vec3(Vec3* this, float x, ...) {
    this->m_x = x;
    ...
}
float Vec3_x(const Vec3* this) {
    return this->m_x;
}


int main() {
  Vec3 v;
  Vec3_Vec3(&v, 1,2,3);

}
```

Von C++ zu C mittels Präprozessor

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Bei Methodenaufrufen immer `this` mitgeben

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

public:
  void Vec3(float x, ...) {
    this->m_x = x;
    ...
  }
  float x() const {
    return this->m_x;
  }
};

int main() {
  Vec3 v(1,2,3);

  float f = v.x();
}
```

```
struct Vec3 {
  float m_x, m_y, m_z;
};

void Vec3_Vec3(Vec3* this, float x, ...) {
    this->m_x = x;
    ...
}
float Vec3_x(const Vec3* this) {
    return this->m_x;
}


int main() {
  Vec3 v;
  Vec3_Vec3(&v, 1,2,3);
  float f = Vec_x(&v);
}
```

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Operatoren und Operator Overloading (1)

Operator Overloading: Motivation

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Wichtiges Designziel von C++: selbst definierte Objekte sollen sich möglichst so verhalten können wie primitive Datentypen

```
int  i1 = 3, i2 = 5;
int i3 = i1 * i2;
```

```
RationalNumber t1(1,2);
RationalNumber t2(3,4);
RationalNumber t3 = t1 * t2;
```

- D.h. die für primitive Typen vordefinierten *Operatoren* sollten auch für selbst definierte Objekt-Typen „funktionieren“
- Das ist in C++ auf einfache Weise möglich – für jeden Operator kann eine entsprechende Methode überladen werden.

```
class RationalNumber {
...
   RationalNumber operator+(const RationalNumber& other) const;
   RationalNumber operator*(const RationalNumber& other) const;
...
};
```

Anwendung von Operator Overloading (OpOv)

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- OpOv ist im wesentlichen eine syntaktische Vereinfachung.
  - Code wird kompakter
  - Kann helfen, Code sehr viel schneller zu verstehen

- Wann sollte man OpOv anwenden?
  - Man sollte intuitiv vom Symbol auf die Semantik der Methode schließen können.
  - Goldene Regel: „**do as `int` does**“ - „**mach es so wie der Typ `int`**“.

- Gegenbeispiele
  - `a + b` → Multiplikation von a mit b.
  - `GameCharacter PlayerA = PlayerB + PlayerC;` → ???

- Kontroverse Beispiele:
  - Hinzufügen von Elementen zu Containern mittels „+“. Heisst „+“ hinten, geordnet, ...? `append`, `push_back`, `insert`, etc. sind *sprechender*.

Grundsätzliches

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Zu einem Operator gibt es eine entsprechende Methode, die überladen werden kann
  - Die Methode für einen Operator **<X>** heisst **`operator<X>()`**, wenn **<X>** ein Sonderzeichen ist.
  - Für textuelle Operatoren wie **`new`** oder **`delete`** wird der Methodenname mit einem Leerzeichen notiert: **`operator new()`**

- Methoden vs. freie Funktionen
  - Fast alle Operatoren können entweder als Methoden oder als „freie“ Funktionen implementiert werden:
  - **`A::operator+(B)`** vs. **`operator+(A,B)`**

- Einschränkungen
  - Mindestens ein Operand muss ein nutzerdefinierter Datentyp sein
  - Die Anzahl der Operanden, die Priorität und Assoziativität der einzelnen Operatoren ist in der Sprache festgelegt und kann nicht verändert werden.

http://www.c-plusplus.de/forum/232010

Überblick der überladbaren Operatoren

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Arithmetische Operatoren

- `+A, -A`
- `A+B, A-B, A*B, A/B, A%B`

Logische Operatoren

- `!A`
- `A&&B, A||B`

Inkrement/Dekrement (Präfix + Postfix)

- `A++, A--`
- `++A, --A`

Zuweisung und Veränderung

- `A=B`
- `A+=B, A-=B, A*=B, A/=B`
- `A&=B, A|=B, A^=B, A<<=B, A>>=B`

Vergleichsoperatoren

- `A==B, A!=B, A<B, A>B`
- `A<=B, A>=B`

Bitweise Operatoren

- `~A`
- `A&B, A|B, A^B, A<<B, A>>B`

Sonstige

- `A[B], A(`<Argument-Liste>`)`
- `&A, *A, A->`
- `A,B`
- `new, new[], delete, delete[]`
- `(int)A, (float)A, (MyType)A, …`

http://en.wikibooks.org/wiki/C%2B%2B_Programming/Operators/Operator_Overloading

Beispiele anhand von 2D-Vektorklassen

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

```
class Vec2 {
    float m_x, m_y;

  public:
    // constructor
    Vec2(float x=0, float y=0)
      : m_x(x), m_y(y) {}

    // accessors
    float x() const { return m_x; }
    float y() const { return m_y; }

};
```

Unäre arithmetische Operatoren

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

- Negation (unäres Minus) als ***Methode***

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

- Negation als ***freie Funktion***

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

```
Vec2 i(1,2);
Vec2 j = -i;
```

Arithmetische Operatoren (ohne Zuweisung) liefern typischerweise das Ergebnis als ein ***neues temporäres / automatisches Objekt*** zurück.

Häufige Konvention für die Bennennung der Argumente:
`lhs` = "left hand side" – das Argument links vom Operator, bzw. das einzige Argument
`rhs` = "right hand side" – Argument rechts vom Operator (nur bei binären Operatoren)

Binäre arithmetische Operatoren

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;
```

- Subtraktion als ***freie Funktion***

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

Vergleichsoperatoren

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

- Prüfung auf Gleichheit

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

```
Vec2 l(1,2);
Vec2 r(3,4);
if(l==r) {...}
if(l!=r) {...}
```

- Das Gegenstück `operator!=()` wird ***nicht*** automatisch vom Compiler hergeleitet

- Hier ist es sinnvoll, für maximale Konsistenz den bereits definierten `operator==()` zu verwenden

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

- Analog: <, >, <=, >=, ...

Vergleich von Objekten – Java vs. C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Wichtiger Unterschied in der Semantik von Java und C++ !

Java:

der Operator == testet, ob `o` und `p` auf ***die gleiche Objektinstanz*** verweisen.

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

}
```

C++:

der Operator == testet, was der Entwickler im `operator==` implementiert hat. Die ***Erwartung*** ist, dass die ***Werte*** von o und p verglichen werden.

Vergleich von Objekten – Java vs. C++

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

Java

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

}
```

C++

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

}
```

Vergleich der
Speicheradressen

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

}
```

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

}
```

Vergleich der
Werte der Objekte