Von Java nach C++

Reinhard Tartler, Michael Gernoth
Fabian Scheler, Peter Ulbrich, Niko Böhm

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

11. Mai 2009

1 Primitive Datentypen

2 Komplexe Datentypen

3 Funktionen

4 Zeiger und Parameterübergabe

5 Klassen

6 Typumwandlung

7 Mehr zur Vererbung

8 Namensräume

9 Templates

Fokus

Worum geht es hier nicht?

- Standardbibliothek
- Speicherverwaltung
- Ein- und Ausgabe

Warum geht es nicht darum?

- Weil wir es hier nicht brauchen!

Worum es geht?

- *Abbildung* bzw. *Umstieg* von Java auf C++
- Was gibt es in C++, was es in Java nicht gibt?

Der bekannteste Unterschied

- Java-Programme werden auf einer virtuellen Maschine ausgeführt:
    - mächtiges Laufzeitsystem (z.B. Garbage Collection)
    - Typinformation
    - dynamische Laden von Klassen
    - Überprüfungen zur Laufzeit
    - ...
- C++-Programme werden auf der nackten Hardware ausgeführt:
    - kein Laufzeitsystem
    - keine Typinformation
    - Ausnahmen:
        - Typinformationen zur Laufzeit (RTTI)
        - Ausnahmen

Primitive Datentypen in Java und C++

- Jeder Datentyp hat eine Machinendarstellung und einen sich daraus ergebenden Wertebereich.
- Dieser ist...
  - ...in *Java* exakt spezifiziert.
  - ...in *C++* hochgradig abhängig von der Implementierung des Übersetzers und der Zielplattform.

Primitive Datentypen - Übersicht

Diese Angaben für C++-Datentypen gelten z.B. auf heutigen x86-32–Systemen mit dem GNU C Compiler (Version 3.3.5):

Java				**C++**			
Bezeichner	Größe	min	max	Bezeichner	Größe	min	max
boolean	-	-	-	bool	-	-	-
char	16 Bit	Unicode 0	Unicode $2^{16} - 1$	char	8 Bit	-128	+127
-	-	-	-	signed char	8 Bit	-128	+127
-	-	-	-	unsigned char	8 Bit	0	255
byte	8 Bit	-128	127	-	-	-	-
short	16 Bit	-2^{15}	$2^{15} - 1$	signed short	16 Bit	-2^{15}	$2^{15} - 1$
-	-	-	-	unsigned short	16 Bit	0	$2^{16} - 1$
int	32 Bit	-2^{31}	$2^{31} - 1$	signed int	32 Bit	-2^{31}	$2^{31} - 1$
-	-	-	-	unsigned int	32 Bit	0	$2^{32} - 1$
long	64 Bit	-2^{63}	$2^{63} - 1$	signed long	32 Bit	-2^{31}	$2^{31} - 1$
-	-	-	-	unsigned long	32 Bit	0	$2^{32} - 1$
float	32 Bit	IEEE754	IEEE754	float	32 Bit	-	-
double	64 Bit	IEEE754	IEEE754	double	64 Bit	-	-
-	-	-	-	long double	96 Bit	-	-
void	-	-	-	void	-	-	-

Primitive Datentypen - Übersicht

Diese Angaben für C++-Datentypen gelten z.B. auf einem H8/300 mit dem GNU C Compiler (Version 3.4):

Java				**C++**			
Bezeichner	Größe	min	max	Bezeichner	Größe	min	max
boolean	-	-	-	bool	-	-	-
char	16 Bit	Unicode 0	Unicode $2^{16}-1$	char	8 Bit	-128	+127
-	-	-	-	signed char	8 Bit	-128	+127
-	-	-	-	unsigned char	8 Bit	0	255
byte	8 Bit	-128	127	-	-	-	-
short	16 Bit	-2^{15}	$2^{15}-1$	signed short	16 Bit	-2^{15}	$2^{15}-1$
-	-	-	-	unsigned short	16 Bit	0	$2^{16}-1$
int	32 Bit	-2^{31}	$2^{31}-1$	signed int	16 Bit	-2^{15}	$2^{15}-1$
-	-	-	-	unsigned int	16 Bit	0	$2^{16}-1$
long	64 Bit	-2^{63}	$2^{63}-1$	signed long	32 Bit	-2^{31}	$2^{31}-1$
-	-	-	-	unsigned long	32 Bit	0	$2^{32}-1$
float	32 Bit	IEEE754	IEEE754	float	32 Bit	-	-
double	64 Bit	IEEE754	IEEE754	double	32 Bit	-	-
-	-	-	-	long double	32 Bit	-	-
void	-	-	-	void	-	-	-

Wie löst man dieses Problem?

Abbildung auf wohldefinierte Typen aus dem C99 Standard

Abbildung

```
#include <cstdint>

int8_t          bst_count;
uint32_t        bst_base_addr;
uint64_t        bst_64bit_counter;
```

Konstanten in *Java* vs. *C++*

Keyword *final*

```
class Foo {
  static final int Bar = 0xff;
}
```

Keyword *const*

```
const unsigned int konstante = 5;
```

- müssen in C++ deklariert **und** definiert werden!
- Die Typüberprüfung geschieht zur Übersetzungszeit.
- Unterschied zu C: benötigt nicht unbedingt Speicherplatz.

Aufzählung

Schlüsselwort *enum* (Nur in C/C++)

```
enum Wochentage {
   Montag,
   Dienstag,
   ...
   Sonntag
};
```

- fasst eine Menge von Werten zu einem eigenständigen Typen zusammen.

Arrays

Arrays in Java

```
int[] i      = new int[5];
Object[] o = new Object[10];
```

- In *Java* sind Arrays Objekte auf dem Heap.

Arrays in C++

```
int i[10];
myClass myClassObjects[CONST_EXPRESSION];
```

- In *C++* sind Arrays nur eine Aneinanderreihung von Objekten gleichen Typs im Speicher.

Benutzerdefinierte Datentypen

Klassen in *Java*

```
class Foo {
  private int x;

  public int Bar() {
    return x;
  }
}
```

Klassen und Strukturen in *C++*

```
class Foo {
   ...
};
/* ODER */
struct Bar {
   ...
};
```

- Strukturen und Klassen sind in C++ fast äquivalent
- Unterschied: Sichtbarkeit
  - Strukturen: standardmäßig public
  - Klassen: standardmäßig private
- zu Klassen später mehr ...

Benutzerdefinierte Typen

Schlüsselwort *typedef*

```
typedef type_A type_B;
```

- `type_B` kann synonym zu `type_A` verwendet werden
- Vorsicht bei *Forward Declarations* und `typedef`

Forward Declaration != typedef

```
typedef my_Class_A my_Class_B;
...
class my_Class_B;
```

obiges Beispiel wird eine Fehlermeldung beim Übersetzen liefern!

Funktionsdefinitionen in Java

Funktionsdefinition im Klassenrumpf

```
class Foo {
  public int a;
  public int Bar(int i) {
    return  i + a;
  }
}
```

- Alle Funktionen sind Methoden genau einer Klasse
- Keine globalen Funktionen
- Funktionen werden immer im Klassenrumpf definiert

Funktionsdefinitionen in C++

Funktionsdefinition im Klassenrumpf

```
class Foo {
  int Bar(int i) { return  i + 5; }
};
```

Funktionsdefinition außerhalb des Klassenrumpfs

```
class Foo {
  int Bar(int i);
};
int Foo::Bar(int i) { return i + 5; }
```

- Funktionsdefinition innerhalb oder außerhalb des Klassenrumpfs
- auch globale Funktionen sind erlaubt

Überladen von Funktionen in Java

```
void Bar(int i) { ... }                   // OK
void Bar(long l) { ... }                  // Fehler
void Bar(Object o) { ... }                // OK
Object Bar(int i) { ... }                 // Fehler
Object Bar(Object o,int i) { ... } // Fehler
```

Überladen von Funktionen in C++

```
void Bar(int i) { ... }                   // OK
void Bar(long l) { ... }                  // Fehler
void Bar(Object o) { ... }                // OK
Object Bar(int i) { ... }                 // Fehler
Object Bar(Object o,int i) { ... } // OK
```

Einbettung von Funktionen in C++ (engl. *inlining*)

Original

```
int inc(int a) {
  return a + 1;
}
int main() {
  int a = 2;
  a = inc(a);
  a = inc(a);
  a = inc(a);
  return 0;
}
```

Eingebettet

```
int main() {
  int a = 2;
  { int a_temp = a;
    a = a_temp + 1; }
  { int a_temp = a;
    a = a_temp + 1; }
  { int a_temp = a;
    a = a_temp + 1; }
  return 0;
}
```

- Semantik der Parameterübergabe bleibt erhalten (hier *call by value*)

Einbettung von Funktionen

- Einbettung ist nicht gleich textueller Ersetzung
  - Der Präprozessor arbeitet so.
- Vorteil: Man spart den Overhead des Funktionsaufrufs
- Gefahr: Code Bloat
- Achtung: Es ist für den Compiler sehr schwer eine optimale Inlining-Strategie zu bestimmen!

Einbettung von Funktionen in Java

Schlüsselwort *final*

```
class Bar {
  final int Foo() { ... }
}
```

- Es gibt kein explizites Inlining in Java!
- Schlüsselwort *final* signalisiert dem Compiler: diese Methode wird nicht mehr überschrieben!
- Nur solche Methoden kann der Compiler einbetten.

Einbettung von Funktionen in C++

- durch Definition im Klassenrumpf
- durch entsprechende Deklaration

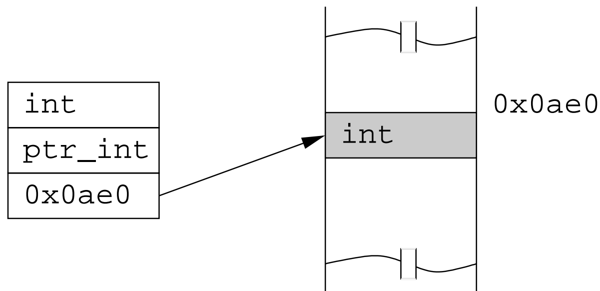
Schlüsselwort *inline*

```
class Foo {
  inline int Bar(int i);
};
int Foo::Bar(int i) { return i + 5; }
```

- Definition muss beim Aufruf für den Compiler sichtbar sein.
- → Funktioniert nur in der selben Übersetzungseinheit.
- Letztendliche Einbettung hängt im Allgemeinen von den Compileroptionen ab!

Was sind Zeiger?

- Ein Zeiger in C/C++ ist ein Tripel (*Typ*, *Bezeichner*, *Wert*)
- Wert eines Zeigers: eine *Adresse*.
- Semantik eines Zeigers: an der *Adresse* findet man ein Objekt vom Typ *Typ*.
- Java: **alle** komplexen Datentypen sind Zeiger
- C++: wählbar



Verwendung von Zeigern

Deklaration

```
int* i;
MyClass* foo;
```

Zugriff auf den referenzierten Wert: Operator *

```
*i = 5;
```

- Wo kommen Zeiger her?

Operator &

```
int i;
int* ptr_to_i = &i;
```

Operator new

```
Foo* f = new Foo();
```

Referenzen

- im Prinzip ein (konstanter) Zeiger, der automatisch dereferenziert wird.
- Referenzen müssen explizit initialisiert werden.

Verwendung

```
int i = 6;
int &c = i;
Foo bar;
Foo &foobar = bar;

c = 5;  // i == 5!
foobar.bar_method()
```

- Java
  - primitive Datentypen: *call by value*
  - komplexe Datentypen: *call by reference*
- C++
  - *call by value*: ein Objekt übergeben
  - *call by reference*: einen Zeiger oder eine Referenz übergeben

```
void by_value(int i,Foo f);
void by_reference(int *i, Foo &f);

int i = 5;
Foo f;

int main() {
  by_value(i,f);          // call by value
  by_reference(&i,f); // call by reference
  return 0;
}
```

Klassen in Java und C++

... sind im Großen und Ganzen sehr ähnlich, vor allem, was das Konzept betrifft

Sichtbarkeit von Klassenfeldern

Java

```java
class Foo {
  public int a;
  protected int b;
  private int c;
  public int Bar() {
    ...
  }
}
```

- Angabe der Sichtbarkeit für jedes Feld einer Klasse einzeln

C++

```cpp
class Foo {
public:
  int a;
  int Bar() { ... }
private:
  int c;
protected:
  int b;
};
```

- Angabe der Sichtbarkeit für eine Gruppe von Feldern einer Klasse

Klassenvariablen - statische Felder

- Deklaration wie in Java

Schlüsselwort *static*

```
class Foo {
  static int bar;
  static void foobar();
};
```

- In C++ muss aber Speicher *angefordert* werden:

```
int Foo::bar = 0;
```

- Vorsicht: `static` für globale Variablen/Funktionen existiert, hat aber eine andere Bedeutung!

Einfache Vererbung

Java

```
class Foo extends Bar {
   ...
}
```

- Felder und Methoden der Basisklasse werden *public* vererbt.

C++

```
class Foo : public Bar {
   ...
};
```

- Sichtbarkeit von Feldern und Methoden der Basisklasse muss spezifiziert werden.

Polymorphismus

Auswahl der Methode zur Laufzeit

```
Unterklasse1 u1;
Unterklasse2 u2;

int main() {
  Oberklasse* o;

  // Unterklasse1::foo()
  o = &u1;
  o->foo();

  // Unterklasse2::foo()
  o = &u2;
  o->foo();
}
```

Virtuelle Methoden

- Polymorphismus nur bei virtuellen Methoden
- in Java sind alle Methoden virtuell
- in C++ müssen Methoden als virtuell deklariert werden
- Vorsicht: Virtuelle Methoden sind teuer, weil zur Laufzeit ein Funktionszeiger nachgeschlagen werden muss.

Schlüsselwort *virtual*

```
class Foo {
  virtual void bar();
};
```

Schnittstellenbeschreibungen

- Interfaces in Java

Schlüsselwort *interface*

```
interface Comparable {
  public int compare(Object o);
}
```

- Abstrakte Klassen in C++: rein virtuelle Methoden

```
class Comparable {
public:
  virtual int compare(Object o) = 0;
};
```

Konstruktoren

- Konzeptionell sehr ähnlich wie in Java.
- Reihenfolge bei Vererbungsketten:
  1. Konstruktor der Basisklasse
  2. Konstruktor der eigentlichen Klasse
- In C++ gibt es kein `super`.

Initialisierungsliste

- C++ Konstruktoren kennen *Initialisierungslisten*
- Basisklassenkonstruktor können hier aufgerufen werden.
- nicht notwendig bei Default-Konstruktoren

```
class Thread : public Coroutine {
  public:
  Thread(void *sp)
    : Coroutine(sp), counter_(0) {
  private:
  int counter_;
  }
};
```

Destruktoren

- Ähnlich `finalize` aus Java
- Räumt ein Objekt auf, wenn seine Lebenszeit endet
- Reihenfolge:
  1. Destruktor der eigentlichen Klasse
  2. Destruktor der Basisklasse
- Es gibt für jede Klasse **genau einen** Destruktor!

Syntax

```
class Foo {
  Foo() { .. }    // Konstruktor
  ~Foo() { .. }   // Destruktor
};
```

Explizite Erzeugung

- Dynamische Erzeugung von Objekten mit dem Operator `new`

```
MyClass* mc = new MyClass();
```

- `new` Liefert einen Zeiger auf das erzeuge Objekt
- Erfordert eine dynamische Speicherverwaltung
- Vorsicht: Speicher muss mit `delete` wieder freigegeben werden. Es gibt in C++ keine Garbage Collection![1]

```
delete mc;
```

[1] zumindest nicht von sich auch...

Implizite Erzeugung - lokale Variablen

Lokale Variablen

```
int main() {
  myClass a;       // default Konstruktor
  yourClass b(3); // allgemeiner Konstruktor
  ...
  return 0;
}
```

- Werden auf dem Stapel angelegt
- Beim Betreten des Sichtbarkeitsbereichs: *Konstruktor*
- Beim Verlassen des Sichtbarkeitsbereichs: *Destruktor*

Implizite Erzeugung - Felder

Felder einer Klasse

```
class Foo {
  public:
  Foo() : b(3),c("Hallo") {...}

  private:
  myClass a;        // default Konstruktor
  yourClass b;      // benutzerdefinierter Konstruktor
  ourClass c;       // benutzerdefinierter Konstruktor
};
```

- *Kind*-Objekte werden implizit mit dem *Eltern*-Objekt erzeugt
- Aufruf des Default-Konstruktors durch den Compiler
- andere Konstruktoren in der Initializer-Liste

Implizite Erzeugung - global

Globale Variablen

```
myClass a;         // Default-Konstruktor
yourClass b(3);    // benutzerdefinierter Konstruktor
```

- globaler Sichtbarkeitsbereich wird beim
  - Start der Anwendung betreten
  - Beenden der Anwendung verlassen
- Diese Konstruktoren werden also noch vor `main` ausgeführt.
- Compiler erzeugt entsprechende Konstruktor/Destruktor-Aufrufe
- Vorsicht: Reihenfolge nicht spezifiziert!

Zugriff auf nicht-statische Felder

Zugriff auf nicht-statische Felder bei Objekten

Operator .

```
class Foo {
public:
  int bar_var;
  int bar_method();
};
```

```
int main() {
  int a,b;
  Foo foo;

  a = foo.bar_var;
  b = foo.bar_method();

  return 0;
}
```

Zugriff auf nicht-statische Felder

Zugriff auf nicht-statische Felder bei Zeigern auf Objekte

Operator ->

```
class Foo {
public:
  int bar_var;
  int bar_method();
};
```

```
int main() {
  int a,b;
  Foo* foo = new Foo();

  a = foo->bar_var;
  b = foo->bar_method();

  return 0;
}
```

- `foo->a` ist Kurzform für `(*foo).a`

Zugriff auf statische Felder

Operator :: (scope-Operator)

```
class Foo {
public:
  static int bar_var;
  static int bar_method();
};
```

```
int main() {
  int a,b;

  a = Foo::bar_var;
  b = Foo::bar_method();

  return 0;
}
```

Konstante Klassen und Methoden

```
class Person {
        unsigned int age;
        Person : age(0) {}
        unsigned int getAge() const;
        void birthday();
};

int main () {
        const Person p;
        printf("Alter:␣%d\n", p.getAge());
        p.birthday(); // Compiler Fehler!
}
```

Typumwandlung in C++

```
type var = const_cast< type >(param);
```

- *entfernt* `const`-Qualifizierung
- besser: `mutable`-Qualifizierung für Instanzvariablen
- Ausnahme: existierende API, die kein `const` unterstützt

```
void foo(const char* str) {
  legacy_func(const_cast< char* >(str));
}
```

- Achtung: `legacy_func` darf `str` nicht verändern!

Typumwandlung in C++

```
type var = static_cast< type >(param);
```

- Konvertierung zwischen verschiedenen Zahlenformaten (z.B. `int` und `float`)
- Konvertierung zwischen verwandten Zeiger- und Referenztypen (die Typen werden statisch während der Übersetzung aufgelöst)
- nur möglich wenn die Typdefinition bekannt ist (eine Deklaration alleine reicht nicht aus), ansonsten $\mapsto$ Übersetzungsfehler
- `this`-Zeiger werden bei Bedarf angepasst (z.B. im Falle der Mehrfachvererbung)

Typumwandlung in C++

```
type var = dynamic_cast< type >(param);
```

- Konvertierung zwischen verwandten Zeiger- und Referenztypen (wird dynamisch während der Laufzeit aufgelöst)
- Im Fehlerfall wird `0` zurück gegeben (Zeiger) oder eine Ausnahme `bad_cast` geworfen (Referenzen).
- Ist nur in Verbindung mit polymorphen Typen möglich (virtuelle Methoden!)
- Alternative: *dynamic dispatch* durch virtuelle Funktionen

Typumwandlung in C++

```
type var = reinterpret_cast< type >(param);
```

- erzwungene Reinterpretation einer bestimmten Speicherstelle
- ähnlich zu Typumwandlungen für Zeiger in C
  ($\neq$ Typumwandlung im C-Stil)
- keine Überprüfung - weder zur Übersetzungs- noch zur Laufzeit
- in der Regel: Finger weg!

Typumwandlung im C-Stil

```
class A { ... };
class B : public class A { ...};

A* a = new A(); // OK
B* b = new B(); // OK
A* pa = b;      // OK
B* b1 = a;      // Fehler
B* b2 = pa;     // Fehler
B* b3 = (B*)pa; // OK (mit einfacher Vererbung)
B* b4 = (B*)a;  // Fehler (nicht erkennbar)
```

Typumwandlung im C-Stil (considered harmful)

- $\neq$ Typumwandlung in C
- *probiert* verschiedene C++-Typumwandlungen:
  1. `const_cast`
  2. `static_cast`
  3. `static_cast`, dann `const_cast`
  4. `reinterpret_cast`
  5. `reinterpret_cast`, dann `const_cast`
- das Ergebnis unterscheidet sich, je nachdem ob
  - alle Header eingebunden wurden $\mapsto$ `static_cast` oder
  - nur eine Deklaration verfügbar ist $\mapsto$ `reinterpret_cast`
- solange es irgendwie geht: Finger weg!
  - Problem: Verwendung von C-Bibliotheken in C++-Programmen

Sichtbarkeit und Vererbung

Basisklasse	Vererbung	Klasse
public	public	public
	protected	protected
	private	private
protected	public	protected
	protected	protected
	private	private
private	public	private
	protected	private
	private	private

- Vorsicht: `protected`/`private`-Vererbung ändert die Schnittstelle
- *Interface*- vs. *Implementation*-Inheritance
- default: *private*

Wozu ist das gut?

- Definition von Benutzerschnittstellen
- Beispiel: Unterbrechungssynchronisation

Schnittstelle

```
// not synchronized
class MyClass {
public:
  myMethod();
};
// synchronized
class MyGuardedClass
  : protected myClass {
public:
  myMethod();
};
```

Verwendung

```
MyGuardedClass myObject;

// OK
myObject.myMethod();

// Compiler Fehler
myObject.
  MyClass::myMethod();
```

Mehrfache Vererbung

Mehrfachvererbung

```
class Foo
  : public Bar1, protected Bar2, ... private Barn
{
  ...
};
```

- eine Klasse kann mehr als eine Basisklasse haben
- Vorsicht: Konflikte können entstehen
  - Klasse `Bar1` definiert Methode `void Bar1::FooBar()`
  - Klasse `Bar2` definiert Methode `void Bar2::FooBar()`

  in Klasse `Foo`: welche Methode ist gemeint?
- Konflikte müssen vom Programmierer aufgelöst werden!

- Java: Packages

Schlüsselwort *package*

```
package Bar;

public class Foo { ... }
```

- Vollständiger Name der Klasse "`Foo`": `Bar.Foo`
- C++: Namensräume

Schlüsselwort *namespace*

```
namespace Bar {

  class Foo { ... };
}
```

- Vollständiger Name der Klasse "`Foo`": `Bar::Foo`

Funktionstemplates

- Zweck: generische Funktion für verschiedene Parameter- und Rückgabetypen
- Syntax:

```
template < class class_name > declaration
template < type type_name > declaration
```

- Beispiel:

```
template < type T > T max(T a,T b) {
  return a > b ? a : b;
}
```

Klassentemplates

- Zweck: parametrisierbare Klassen
- Syntax:

```
template < class class_name > declaration
template < type type_name > declaration
```

- Beispiel:

```
template <type T> class Tuple {
  T content[20];
public:
  T get(int i) { return content[i]; }
  void set(T val, int i) { content[i] = val; }
};
```

Klassentemplates (fort.)

- Templates koennen sowohl auf Klassen als auch auf den primitiven Datentyp `int` parametriert werden:

```
template < type T, int N > class Tuple {
  T content[N];
public:
  T get(int i) { return content[i]; }
  void set(T val, int i) { content[i] = val; }
};
```

Template Beispiel

- Fakultäts-Template (rekursive Expansion mit Abbruchbedingung):

```
template <int X>
struct Faculty {
  static const int result = X *
    Faculty<X-1>::result;
};
template <>
struct Faculty<1> {
  static const int result = 1;
};

int main() {
  int fak = Faculty<10>::result;
}
```

Template Beispiel

- Fakultäts-Template - Ergebnis:

```
$ g++ -o fak fak.cpp
$ objdump -D fak
...
_main:
...
00001ff0        movl    $0x00375f00,0xf4(%ebp)
...
```

- 1*2*3*4*5*6*7*8*9*10=3628800 (375F00h)
- Fakultät wird zur Übersetzungszeit berechnet

Operatoren entsprechen *normalen* Funktionen

Beispiele

```
class Foo {
  // Zuweisungsoperator
  Foo operator=(Foo val);
  // Typkonvertierung
  operator Bar ();
};
```

- Syntactic Sugar
- keine echten benutzerdefinierten Operatoren
- gewisse Operatoren können nicht überladen werden
- Präzedenz kann nicht beeinflusst werden