Java vs. C++: Different by Design

Java

- starke Anlehnung an C++
- Deployment Schema: Interpretation
- OO ist (nahezu) zwingend
- primäres Kriterium: Komfort

  diverse (und zumeist nicht abschaltbare) implizite Overheads zu Lasten der Effizienz
  - Prüfung von Feldgrenzen
  - Reflection
  - Garbage Collection
  - Objects by Reference Semantik

Java vs. C++: Different by Design

C++

- starke Anlehnung an C
- Deployment Schema: Compilation
- OO ist möglich, nicht zwingend
- primäres Kriterium: Effizienz

  keinerlei impliziter Overhead zu Lasten der Effizienz
  - keine Prüfung von Feldgrenzen
  - (fast) kein Laufzeitabbild von Klassen
  - keine automatische Speicherverwaltung
  - Objects by Value Semantik

Objects by Reference

Java:

- Variablen vom Klassentyp sind IMMER Referenzen

```
X x;  // implizit == null !!
x = new X();
X y = x;  // ein Objekt mit zwei Referenzen!!!
```

- Objekte werden IMMER dynamisch (auf dem Heap) erzeugt

Objects by Reference

```
class A {
        private int i;
        public void foo() {
             i++;
        }
        public void out() {
             System.out.print(i);
        }
        public A() {
             i=0;
        }
        public static void bar(A a){
             a.foo();
        }
```

```
        public static void
        main(String s[]) {
            A a1 = new A();
            A a2 = a1;

            a1.foo();
            a2.foo();

            a1.out();
            a2.out();

            bar(a2);

            a1.out();
            a2.out();
        }
}
```

```
$ javac A.java
$ java A
????
```

Objects by Reference

```
class A {
        private int i;
        public void foo() {
             i++;
        }
        public void out() {
             System.out.print(i);
        }
        public A() {
             i=0;
        }
        public static void bar(A a){
             a.foo();
        }
```

```
        public static void
        main(String s[]) {
            A a1 = new A();
            A a2 = a1;

            a1.foo();
            a2.foo();

            a1.out();
            a2.out();

            bar(a2);

            a1.out();
            a2.out();
        }
}
```

```
$ javac A.java
$ java A
2233$
```

Objects by Value

C++:

- Variablen vom Klassentyp sind (primär) Werte

```
X x; // ein Objekt !
X y = x;  // ein weiteres Objekt als Kopie des ersten!!!
```

- Objekte können global, (Stack-) lokal und dynamisch erzeugt werden

- Es gibt auch Objektreferenzen und -Zeiger

Objects by Value

```
#include <iostream>

class A {
        int i;
public:
        void foo() {
            i++;
        }
        void out() {
            std::cout << i;
        }
        A() {
            i=0;
        }
        static void bar(A a) {
                a.foo();
        }
};
```

```
int main()
{
        A a1=A();
        A a2=a1;

        a1.foo();
        a2.foo();

        a1.out();
        a2.out();

        A::bar(a2);

        a1.out();
        a2.out();
}
```

```
$ g++ -o a a.cc
$ a
????
```

Objects by Value

```
#include <iostream>

class A {
        int i;
public:
        void foo() {
            i++;
        }
        void out() {
            std::cout << i;
        }
        A() {
            i=0;
        }
        static void bar(A a) {
                a.foo();
        }
};
```

```
int main()
{
        A a1=A();
        A a2=a1;

        a1.foo();
        a2.foo();

        a1.out();
        a2.out();

        A::bar(a2);

        a1.out();
        a2.out();
}
```

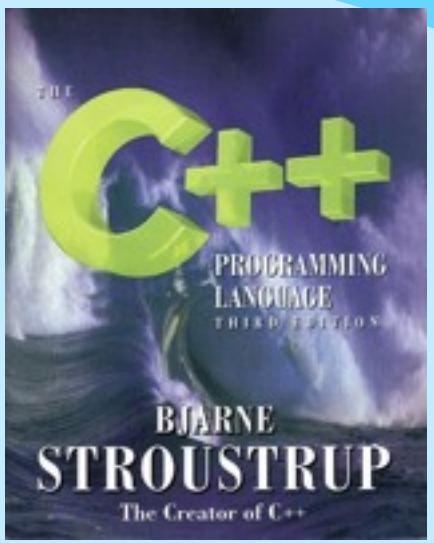
```
$ g++ -o a a.cc
$ a
1111$
```

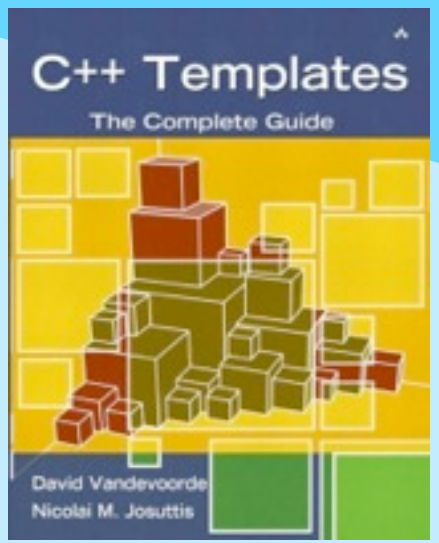
BTW: C++ Literaturempfehlungen beginners level

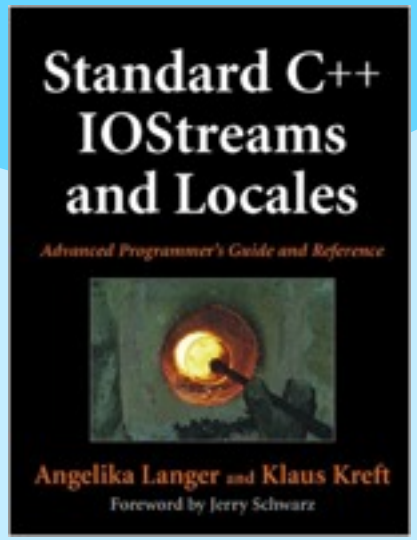
Kaufen: „Bafög-Ausgabe" 19,95 €

Ausleihen: im Handel leider vergriffen ☹

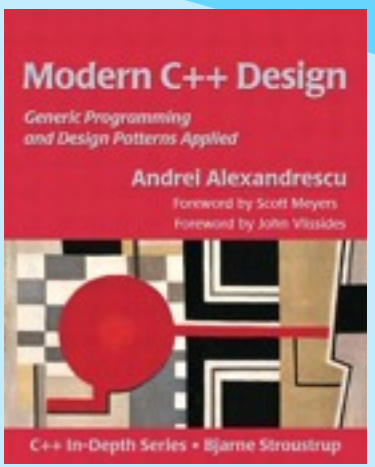
BTW: C++ Literaturempfehlungen expert level

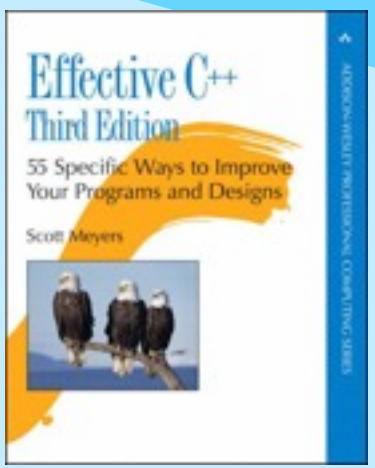


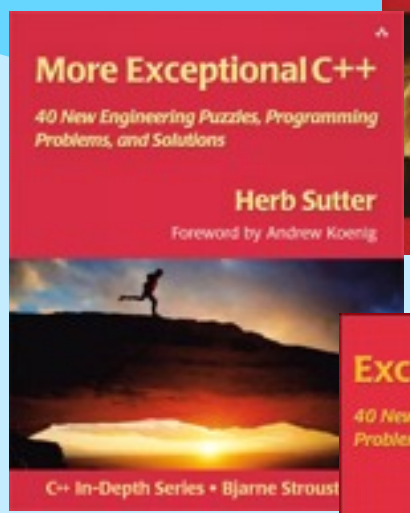


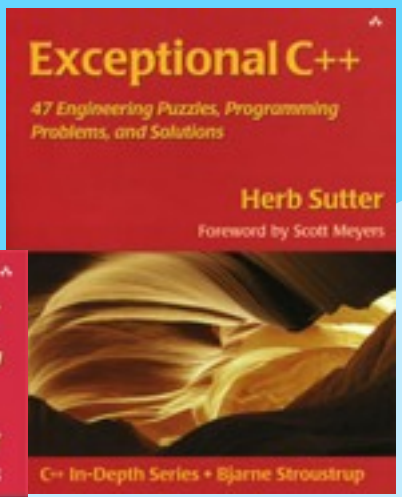


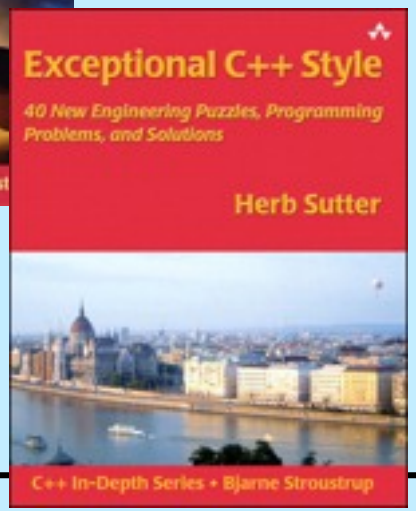
BTW: C++ Literaturempfehlungen guru level











1. Elementares C++

1.1. Lexik

- Kommentare wie Java

```
//  this line
/*   no
     nesting
     allowed */
```

- kein spezielles doc-Kommentarformat, aber von einigen tools unterstützt (z.b. doxygen)

- free format: whitespaces (space, newline, comment) beliebig zur Trennung von Token: `int a;  <---> inta;`

1.1. Lexik

Schlüsselwörter:

```
alignof asm auto bool break case catch char char16_t
char32_t class const constexpr const_cast continue
decltype default delete do double dynamic_cast else
enum explicit export extern false float for friend
goto if inline int long mutable namespace new noexcept
nullptr operator private protected public register
reinterpret_cast return short signed sizeof static
static_assert static_cast struct switch template this
thread_local throw true try typedef typeid typename
union unsigned using virtual void volatile wchar_t while
```

(C: 32) (Δ C++98: 31) (Δ C++11: 9)

1.1. Lexik

Operatoren:

`+ - * / % < <= > >= == != && || !` wie üblich (Java)

`<< >> & ^ | ~` bitweise left-, right-Shift, and, xor, or, Komplement

`= *= /= %= += -= <<= >>= &= ^= |=` x?=y <--> x = x ? y

`++ --` als Prefix und Postfix

`sizeof(Typname)` oder

`sizeof(Expression)` **`oder`** Größe in Bytes

`sizeof Expression`

`,` Kommaoperator: Gruppierung von Ausdrücken, der letzte Teilausdruck legt den Wert des Gesamtausdruchs fest!

ACHTUNG: foo(1,2,3) vs. foo((1,2,3))

1.1. Lexik

Bezeichner: wie in Java (incl. _ als Buchstabe)

Groß-/Kleinschreibung wird unterschieden

übliche Konventionen:

sog. Macros durchgängig groß:	#define A_MACRO
nutzerdef. Typnamen beginnen groß:	MyType
Variablen durchweg klein:	MyType myvar;

1.2. Datentypen

build-in Typen:

```
char, int, short (int), long (int), (un)(signed)(long)
  int, void, float, double, bool (!)
```

- ACHTUNG: long ist kein eigener Typ, sondern Kürzel für long int
- ACHTUNG: es gibt KEINE Vorgaben zur Größe von Variablen dieser Typen: 1==sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
  sizeof(float)<=sizeof(double)
- literale Werte dieser Typen nach den »üblichen« Regeln:

```
'A'     '\n'   '\\'   '\000'   '\0x12'
123     -45    0123   0x123    0XCDEF
12U     23u    123L   0l       0x12345L
1.234   .5f    45.    1.1e12   -2.3E-5
true    false
```

1.2. Datentypen

Enumerations: Aufzählungstypen == benannte Werte

```
enum Season {spring, sommer, fall, winter}; //unscoped
enum class Direction {left, right, up, down}; //scoped
Season now = spring; ... if (now == winter) ...
Direction where = Direction::up;
```

Felder: mehrere Objekte (Variablen) direkt hintereinander im Speicher,
ein Feld ist selbst KEIN Objekt, --> KEIN Längenattribut

```
int f [n];
```

f zeigt auf den Beginn eines Feldes von n int's, n muss eine vom Compiler errechenbare Konstante sein !

neu in C++11: Typdeduktion

```
auto x = 7;
```

x ist von Typ int wegen des Typs des Literals.

```
auto x = expression;
```

x ist vom Typ des Resultats von expression.

(erlangt erst im Zusammenhang mit Templates seine volle Bedeutung)

neu in C++11: Typdeduktion

```
template<class T> void printall(const vector<T>& v) {
  for (auto p = begin(v); p!=end(v); ++p) cout << *p << "\n";
}
statt C++98:
template<class T> void printall(const vector<T>& v) {
  for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
    cout << *p << "\n";
}
```

C++11

```
template<class T,class U> void f(const vector<T>& vt, const vector<U>& vu){
  // ...
  auto tmp = vt[i]*vu[i]; // whatever T*U yields
  // ...
}
```