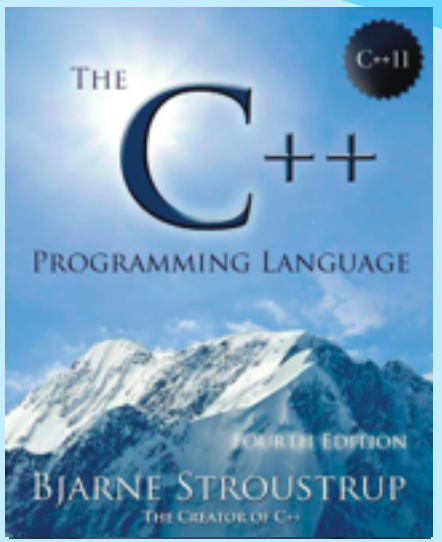
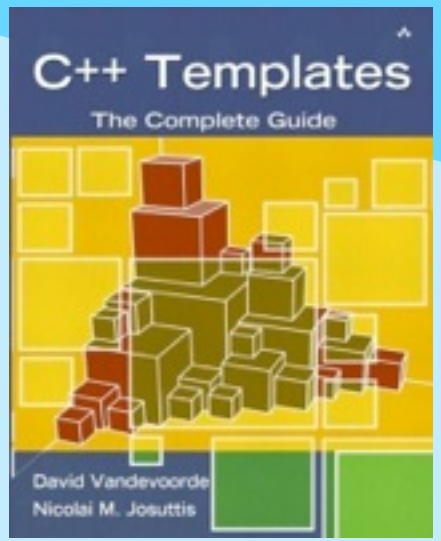
BTW: C++ Literaturempfehlungen beginners level

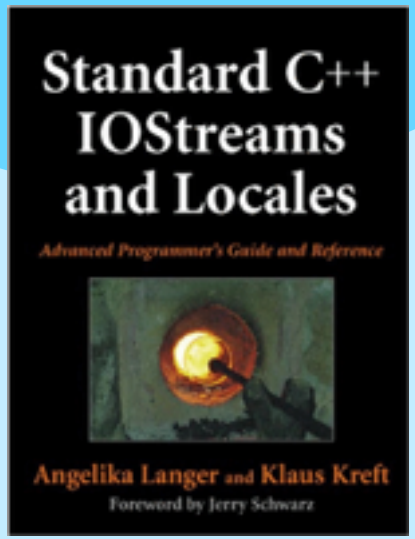
Kaufen: „Bafög-Ausgabe" 19,95 €

Ausleihen: im Handel leider vergriffen ☹

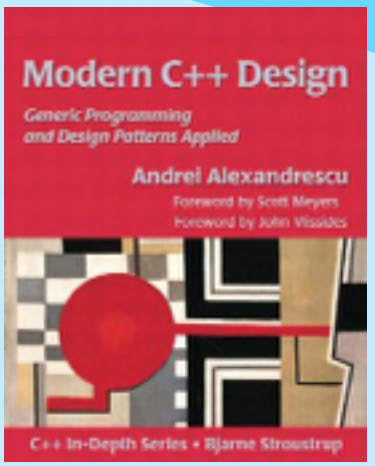
BTW: C++ Literaturempfehlungen expert level

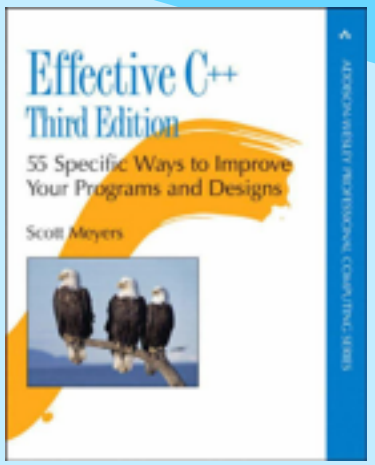


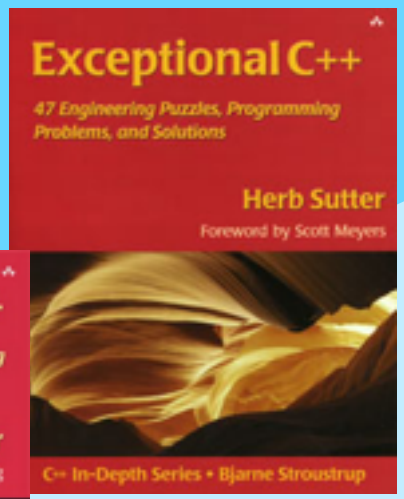


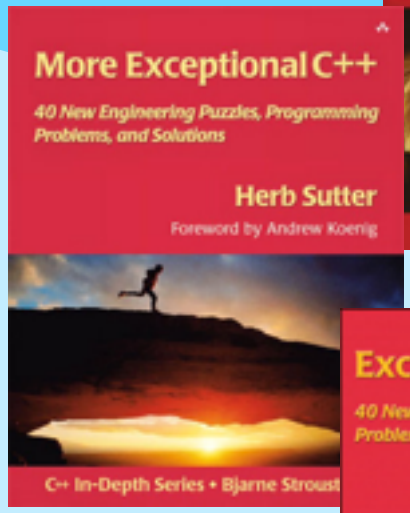


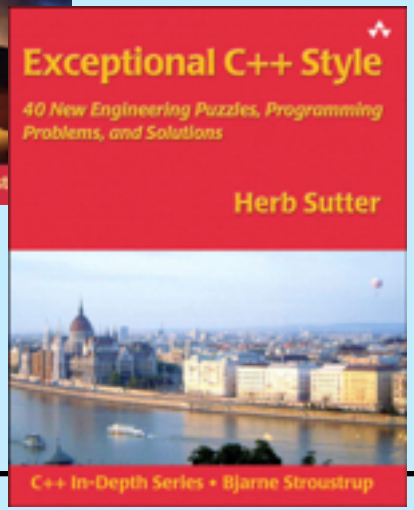
BTW: C++ Literaturempfehlungen guru level











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, ohne $)
Groß-/Kleinschreibung wird unterschieden

übliche Konventionen:

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

_ oder __ am Anfang vermeiden (für Implementation reserviert)

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
  // ...
}
```

1. Elementares C++

C - **enum**s mit Problemen:

- konvertierbar nach int
- exportieren ihre Aufzählungsbezeichner in den umgebenden Bereich (name clashes)
- schwach typisiert (z.B. keine forward Deklaration möglich)

enum classes ("strong enums") sind stark typisiert und 'scoped':

```
enum Alert { green, yellow, election, red }; // traditional enum
enum class Color { red, blue };              // scoped and strongly typed enum
                 // no export of enumerator names into enclosing scope
                 // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;             // error (as ever in C++)
Color c = 7;             // error: no int->Color conversion
int a2 = red;            // ok: Alert->int conversion
int a3 = Alert::red;     // error in C++98; ok in C++0x
int a4 = blue;           // error: blue not in scope
int a5 = Color::blue;    // error: not Color->int conversion
Color a6 = Color::blue; // ok
```

Typ der Repräsentation kann spezifiziert werden

```
enum class Color : char { red, blue }; // compact representation enum class

TrafficLight { red, yellow, green };
    // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
// how big is an E?
// (whatever the old rules say;
// i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
// now we can be specific
```

C++11

forward Deklaration möglich

```
enum class Color_code : char; // (forward) declaration
void foobar(Color_code* p); // use of forward declaration
// ...
enum class Color_code : char { red, yellow, green, blue };
// definition
```

1.2. Datentypen (Felder)

```
int p[];
```

nur in Argumentlisten von Funktionen:

int-Feld unbekannter == beliebiger Länge, Größeninformation ist separat bereitzustellen

```
double m[3][4];
```

12 doubles hintereinander !

Typedefs: Synonyme für (u.U.) komplexe Typkonstrukte

```
typedef double V4[4];
V4 m[3]; // entspricht obigem Feld
```

1.2. Datentypen (Felder)

es gibt keine Prüfung auf Einhaltung der Feldgrenzen bei Zugriffen:

```
T f [10];
```