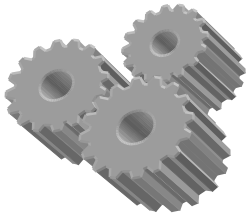
Effektiv C++

22 Tips für besseres C++

basierend auf Ideen von Scott Meyers

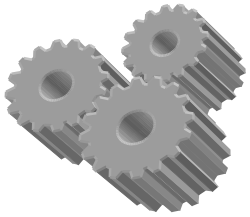


Effektiv C++

Stephan Brumme, 21.Mai 2002

Gliederung

1. Einführung
2. Der Umstieg von C nach C++
3. Konstruktoren, Destruktoren und Zuweisungsoperatoren
4. Entwurf und Deklaration von Klassen und Funktionen

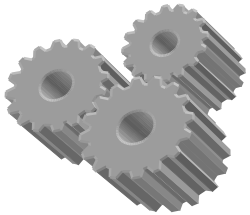


Einführung

Begriff „Deklaration“:

- Mitteilung des Namens und Typs
  - eines Objekts,
  - einer Funktion,
  - einer Klasse oder
  - eines Templates
- z.B.:

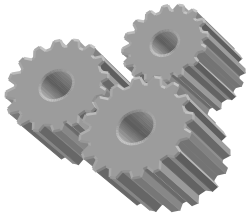
```
extern string strHostname;
string GetHostname(int nLocalIP);
class CServer;
template<class T> class CStack;
```



Einführung

Begriff „Definition“:

- Festlegung des Aufbaus und der Struktur
- der Compiler kann für ein Objekt Speicher bereitstellen
- eine Funktionsdefinition enthält den kompletten Funktionsrumpf
- eine Klasse oder ein Template zählt die Elemente auf



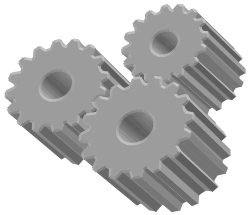
Einführung

Initialisierung:

- erstmalige Wertzuweisung an ein Objekt
- wird immer automatisch direkt nach der Erzeugung eines Objekts einer Klasse durchgeführt
- keine Initialisierung von in C++ eingebauten Typen !

Zuweisung:

- Wertzuweisung an ein bereits initialisiertes Objekt
- eventuell Freigabe von alloziierten Ressourcen notwendig



Einführung

Konstruktor:

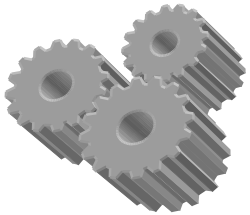
- diejenige(n) Methode(n) einer Klasse, die zur Initialisierung aufgerufen werden

Defaultkonstruktor:

- Konstruktor, der keine Parameter benötigt oder für alle Parameter jeweils einen Standardwert kennt

Copy-Konstruktor:

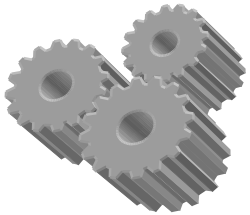
- Initialisierung eines Objekts als Kopie eines anderen, welches vom gleichen Typ ist



Einführung

Destruktor:

- diejenige Methode einer Klasse, die zur Zerstörung eines Objekts aufgerufen wird
- es existiert immer nur ein Destruktor pro Klasse



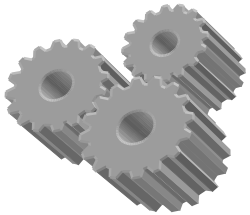
Einführung

Compiler erzeugt automatisch:

- Default-Konstruktor, wenn überhaupt kein Konstruktor vorhanden ist
- Copy-Konstruktor und Zuweisungsoperator, wenn diese notwendig sind
- Destruktor

→ diese können und sollten aber vom Programmierer selbst geschrieben werden

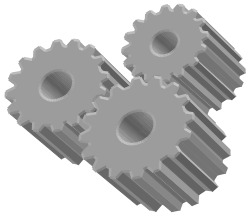
- noch einige Operatoren mehr (siehe Vortrag Effektiv C++ Teil 2)



Der Umstieg von C nach C++

Tip 1: Vermeidung von Problemen des Präprozessors

- Präprozessor führt lediglich Textersetzung durch, d.h. hat keinerlei tiefere Kenntnisse der Sprache C++
- neue C++ Techniken sind typsicher:
  - Definition von Konstanten immer mit dem Schlüsselwort `const` vornehmen
  - generische Funktionen oder Klassen als Templates implementieren



Der Umstieg von C nach C++

Tip 2: Kommentare

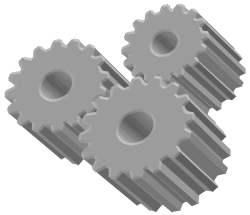
- aus C bekannte Kommentare sind zeilenübergreifend, können aber nicht verschachtelte C-Kommentare überdecken:

```
/* something */
```

- in C++ neuer Kommentartyp:

```
// something
```

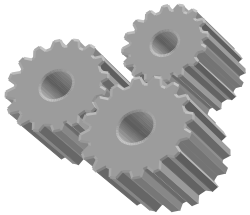
  - nur bis zum jeweiligen Zeilenende
  - kann beliebige Zeichen und Kommentare überdecken



Der Umstieg von C nach C++

Tip 3: Neue I/O-Routinen

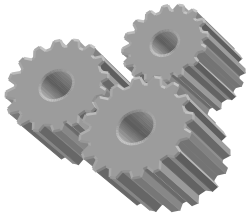
- Definition von typsicheren Eingabe-/Ausgabefunktionen in *iostream* für alle Standarddatentypen von C++
- gleiche syntaktische Form von Objekten bei Lese- und Schreibzugriffen
- deutlich weniger Fehlermöglichkeiten als bei C, da Datentyp nicht explizit angegeben werden muss
- i.d.R. ähnliche Effizienz



Der Umstieg von C nach C++

Tip 4: Geänderte Speicherverwaltung

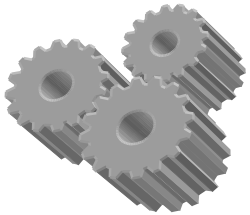
- nur `new` und `delete` rufen bei dynamisch erzeugten Objekten den Konstruktor und Destruktor auf
- als Operatoren für jede Klasse überladbar
- mit `malloc` angelegten Speicher nie mit `delete` freigeben (gleiches gilt für `new` in Zusammenspiel mit `free`)



Der Umstieg von C nach C++

Tip 4: Sichere Typkonvertierungen

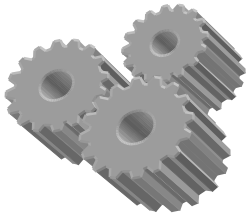
- Upcast ohne zusätzliche Syntax möglich
- sonst `pZiel = dynamic_cast<Zieltyp>(pQuelle);`
- Downcast polymorpher Objekte mit `dynamic_cast`
  - Überprüfung des Typs erfolgt zur Laufzeit
  - wenn fehlgeschlagen:
    Rückgabe von `NULL` bzw. `bad_cast`-Exception
  - Aktivierung von Runtime Type Information (RTTI) erforderlich
  - geringer zusätzlicher Laufzeitaufwand
  - sicherster Cast



Der Umstieg von C nach C++

- einfacher Cast in Klassenhierarchien mit `static_cast`
  - keine tiefe Typprüfung wie bei `dynamic_cast` möglich
  - wird während der Kompilierung vorgenommen
  - kein RTTI notwendig, kein Laufzeitoverhead
  - empfohlen, wenn Laufzeit große Rolle spielt

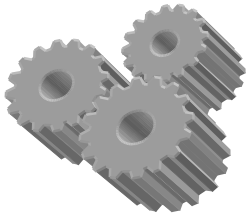
- Unterdrückung der `const`-Eigenschaft eines Objektes mit `const_cast`
  - Benutzung nur in Ausnahmefällen sinnvoll, meist in fehlerhaftem Klassenentwurf begründet



Der Umstieg von C nach C++

- Konvertierung zwischen Zeigern gleicher Bitbreite mit Hilfe von `reinterpret_cast`
  - ist i.d.R. immer möglich, da auf modernen Systemen alle Zeiger gleich breit sind
  - vollkommen sinnentstellte Typkonvertierungen sind erlaubt
  - sollte nicht benutzt werden

- C-Cast erlaubt sämtliche Typkonvertierungen ohne Überprüfung
  - Syntax: `pZiel = (Zieltyp)pQuelle;`
  - sollte nicht benutzt werden



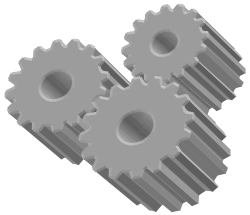
Konstruktoren, Destruktoren und Zuweisungsoperatoren

Tip 6: Initialisierung vs. Zuweisung im Konstruktor

- Attribute können initialisiert werden, bevor man den Code des Konstruktors ausgeführt:

```
class Class {
    Class(MyType Variable) : Attribut(Variable)
    { ... }
}
```

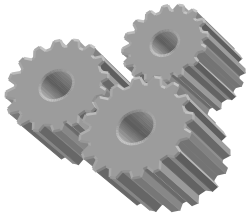
- erforderlich für `const`- und Referenzattribute
- sinnvoll, wenn Copy-Konstruktor des Attributes effizienter als Initialisierung+Zuweisung ist



Konstruktoren, Destruktoren und Zuweisungsoperatoren

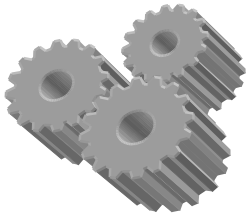
Tip 7: Reihenfolge der Initialisierung im Konstruktor

- alle Attribute werden stets in der Reihenfolge initialisiert, in der sie deklariert sind
- Reihenfolge der Aufzählung in der Inititialisierungsliste des Konstruktors ist irrelevant !
- Abhängigkeiten zwischen zu initialisierenden Attributen daher minimieren



Tip 8: Virtuelle Destruktoren

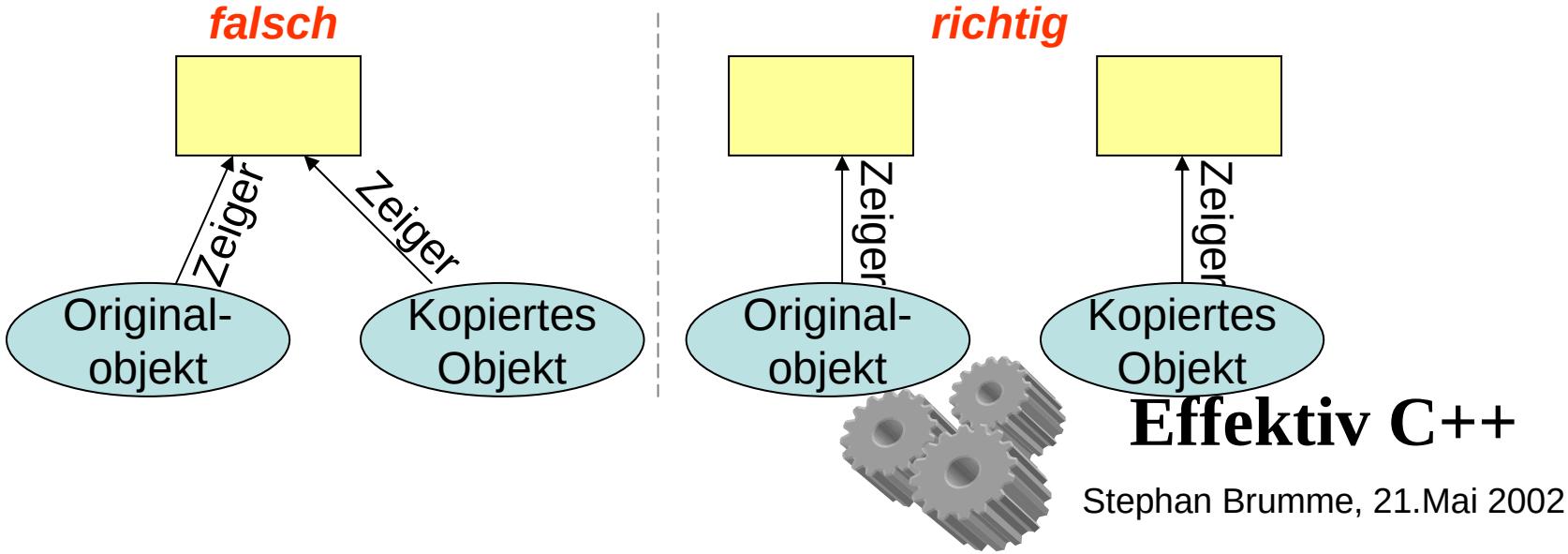
- nach einem Upcast kann Compiler nicht mehr den korrekten Destruktor finden, falls er nicht-virtuell ist, er nimmt den der momentanen Klasse, nicht den der ursprünglichen
  → nicht alle Ressourcen werden korrekt freigegeben
- Ausweg: Deklaration des Destruktors als `virtual`
- wenn Destruktor der Basisklasse `virtual` ist, dann sind es automatisch alle abgeleiteten, auch ohne explizites `virtual`
  → für jede Klasse, die als Basisklasse dienen soll, ist virtueller Destruktor empfohlen
- aber für leichtgewichtige Klassen bedeutet evtl. virtuelle Methodentabelle erheblichen Aufwand



Konstruktoren, Destruktoren und Zuweisungsoperatoren

Tip 9: Copy-Konstruktor und Zuweisungsoperator für Klassen mit dynamisch alloziiertem Speicher

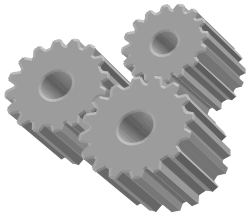
- C++ erzeugt automatisch Copy-Konstruktor und Zuweisungsoperator für jede Klasse, wenn diese nicht existieren
  - generieren bitweise Kopie mit optimalem Maschinencode
    → Adressen von Zeigern werden kopiert, aber nicht das, worauf sie verweisen



Konstruktoren, Destruktoren und Zuweisungsoperatoren

- nur wenn min. ein Attribut dynamisch Speicher alloziiert, dann sollten ein eigener Copy-Konstruktor und ein Zuweisungs-operator deklariert und definiert werden
  - Kopie der alloziierte Ressourcen durchführen
  - ansonsten Standardimplementierung nutzen, da diese optimal ist

- bei Erweiterungen der Klasse um neue Attribute den Copy-Konstruktor und Zuweisungsoperator um diese ergänzen

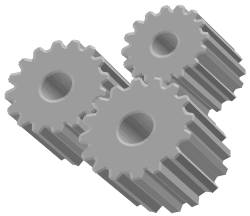


Konstruktoren, Destruktoren und Zuweisungsoperatoren

Tip 10: Copy-Konstruktor und Zuweisungsoperator in Klassenhierarchien

- Objekt kann Attribute der Basisklasse nicht kopieren, wenn diese `private` sind, da Zugriff verboten ist
- Lösung: im Copy-Konstruktor/Zuweisungsoperator Aufruf des Copy-Konstruktors/Zuweisungsoperators der Basisklasse

```
CClass(const CClass& obj) : CBaseClass(obj) ...
{ ... }
CClass& operator=(const CClass& obj) {
   ...
   CBaseClass::operator=(obj);
   ...
}
```

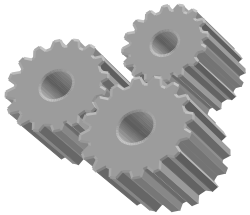


Tip 11: Rückgabewert des Zuweisunsoperators

- Verkettung soll erlaubt sein: `a = b = c;`
- Rückgabe des linken Objekts, wegen Effizienz als Referenz

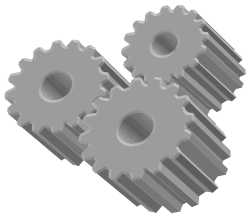
```
CClass& operator=(const CClass& obj) {
    ...
    return *this;
}
```

- Lehrmeinung ist: Rückgabewert nicht `const`, damit `(a = b) = c` zugelassen wird
  → ich empfehle trotzdem `const` !



Tip 12: Prüfung auf Selbstzuweisung

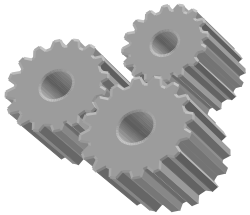
- über Aliasing-Effekte ist Zuweisung eines Objekts an sich selbst möglich
- normale Reihenfolge bei Zuweisung:
  - Freigabe der Ressourcen des zu überschreibenden Objekts
  - Kopie der neuen Ressourcen
- Kopie scheitert, wenn dasselbe Objekt bereits alle Ressourcen freigegeben hat
- Prüfung auf Objektidentität reicht i.d.R. aus
  `if (this == &obj) ...`
- Problem nicht nur auf Zuweisungsoperator beschränkt
  → alle Methoden, die die eigene Klasse als Parameter akzeptieren, sind betroffen



Entwurf & Deklaration von Klassen

Tip 13: Vollständige aber minimale Schnittstellen

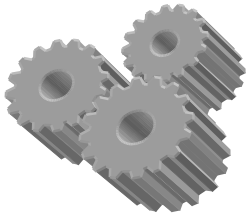
- jede Klasse soll alle sinnvollen Anforderungen an sie erfüllen
- zu große Funktionalität ist kontra-produktiv:
  - hoher Einarbeitungsaufwand
  - aufwändige Wartung
  - Gefahr von Inkonsistenzen
- minimal empfohlene Schnittstelle:
  - (Default-) Konstruktor
  - Destruktor
  - Copy-Konstruktor und Zuweisungsoperator (Tip 9)
- meist von Vorteil:
  - Serialisierungsmechanismus
  - Überprüfung der Klassen-invariante



Entwurf & Deklaration von Klassen

Tip 14: Keine öffentlichen Attribute

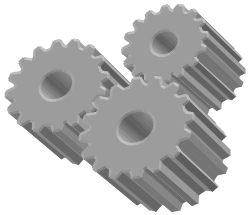
- Verlust der Kontrolle über Lese- und Schreibzugriffe
  → Gefahr von Inkonsistenzen
- keine Möglichkeit der späteren Ersetzung einer Zuweisung durch eine Berechnung/Überprüfung+Zuweisung
- funktionale Abstraktion:
  - jedes Attribut vor Zugriff von außen schützen
  - Bereitstellung von Lese- und/oder Schreibmethoden
  - Deklaration als `inline` erlaubt Compiler meist Eliminierung des Laufzeitoverheads einer Methode



Entwurf & Deklaration von Klassen

Tip 15: Methoden vs. globale Funktionen

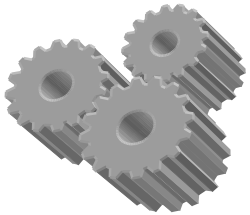
- möglichst viele Funktionen sollten Methoden sein
  - virtuelle Funktionen müssen Methoden sein
- nur globale Funktionen erlauben implizite Typumwandlung des linken Arguments
  - sinnvoll z.B. wenn linkes Argument Konstante ist
  - für Zugriff auf private Attribute ist Deklaration innerhalb der Klasse als `friend` erforderlich



Entwurf & Deklaration von Klassen

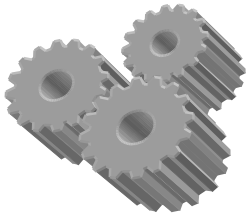
Tip 16: Benutzung von `const`

- `const` bedeutet Unveränderbarkeit des attributierten Symbols
- Wirkung auf
  - Methoden: kein Attribut des Objekts darf verändert werden
  - Objekte: nur Aufruf von const-Methoden und Lesezugriffe auf Attribute erlaubt
- Compiler überprüft bitweise Konstantheit
  → zwar nicht Zeiger selbst, aber deren Inhalt ist manipulierbar
- Ausnahme für Attribute mit Schlüsselwort `mutable`



Entwurf & Deklaration von Klassen

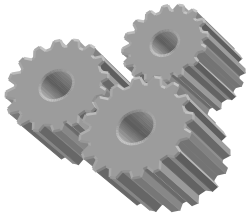
- generell existieren bei Zeigern gibt es drei Fälle von `const`:
  - das, worauf gezeigt wird, darf nicht verändert werden:
    ```
    const Typ* Variable = Wert;
    ```
  - der Zeiger selbst darf nicht verändert werden:
    ```
    Typ* const Variable = Wert;
    ```
  - weder der Zeiger noch der Inhalt darf geändert werden:
    ```
    const Typ* const Variable3 = Wert;
    ```



Entwurf & Deklaration von Klassen

Tip 17: Übergabe via Wert vs. Referenz

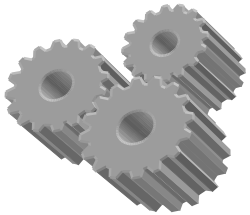
- Referenz vermeidet Aufruf des Copy-Konstruktors
- `const`, wenn Argument nicht verändert werden braucht/darf
- Wertübergabe nimmt ggf. Typkonvertierung auf Basisklasse vor
  → oft unerwünscht
- Referenz arbeitet intern mit Zeigern, d.h. wenn Datentyp sehr klein ist, dann sind Wertübergaben effizienter
  → für alle C++ Basistypen (außer STL) ist Wertübergabe sinnvoll
  → sonst immer (konstante) Referenz benutzen



Entwurf & Deklaration von Klassen

Tip 18: Rückgabe einer Referenz

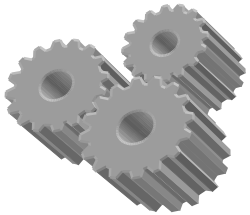
- zwar effizient für große Objekte, aber meist keine vollständige Kontrolle:
  - Veränderungen durch andere Objekte möglich
  - Zerstörung des Objektes denkbar
- nur sinnvoll für langlebige Objekte, `const` idealerweise benutzen
- Einsatz in der Praxis oft auf mathematische Operatoren beschränkt
- dynamisch erzeugte Objekte immer per Wert zurückgeben



Entwurf & Deklaration von Klassen

Tip 19: Überladung vs. Default-Parameter

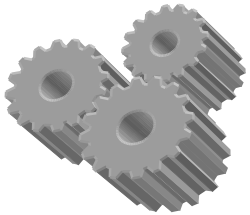
- Methoden, die unter gleichem Namen in einer Klasse auftauchen, müssen diese Entscheidung treffen
- Default-Parameter optimal, wenn der verwendete Algorithmus der Methoden ähnlich oder identisch ist
- sonst Überladung einsetzen



Entwurf & Deklaration von Klassen

Tip 20: Mehrdeutigkeiten

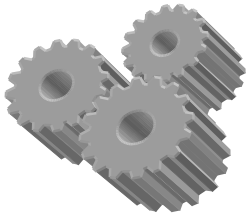
- implizite Typkonvertierungen können zu Mehrdeutigkeiten führen
  - treten oft unbewusst auf
- signaturgleiche Methoden in Basisklassen sorgen bei Mehrfachvererbung auch für Mehrdeutigkeiten
- zwei Reaktionen möglich:
  - Compiler kann keine Entscheidung treffen → Fehlermeldung
  - fehlerhaftes Kompilat
- als `explicit` deklarierter Konstruktor verhindert implizite Konvertierungen



Entwurf & Deklaration von Klassen

Tip 21: Verbot automatisch generierter Methoden

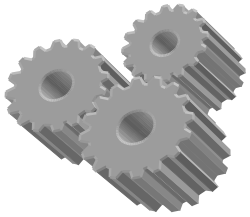
- in einigen Fällen ist Copy-Konstruktor bzw. Zuweisungsoperator nicht erwünscht
- Deklaration als `private` ohne anschließende Definition



Entwurf & Deklaration von Klassen

Tip 22: Unterteilung des Namensraums

- in großen Projekten oftmals Symbole mehrfach verwendet
- Namenskollisionen durch Compiler / Linker nicht auflösbar
- eigene Namensräume mit `namespace` schaffen
- Benutzung von Symbolen des Namensraumes durch `Namensraum::Symbol`
- Einbindung eines Symbols in den eigenen Namensraum mit `using Namensraum::Symbol`, danach per `Symbol` verfügbar
- kompletten Namensraum mit `using namespace Namensraum` inkludieren



Entwurf & Deklaration von Klassen

Literaturverzeichnis

- Scott Meyers: Effektiv C++ programmieren, 3.Auflage, Addison-Wesley Longman, Bonn, 1998
  Hinweis: Die Nummerierung in diesem Buch unterscheidet sich von meiner, die grobe Reihenfolge der Tips ist aber ähnlich

- Weiter empfehlenswert:
  - Bjarne Stroustrup: Die C++ Programmiersprache, 3.Auflage, Addison-Wesley Longman, Bonn, 1998
  - Bruce Eckel: Thinking in C++ Band 1&2, 2.Auflage, Prentice Hall, Upper Saddle River, 2000
    → kostenlos im PDF-Format erhältlich unter www.bruceeckel.com