Tipps und Tricks zur Programmiersprache C++
Was sollte man tun und was besser lassen?

Dipl. Ing. Christoph Stockmayer, Stockmayer GmbH, Schwaig

Inhalt

Die Programmiersprache C++ ist eine sehr vielseite aber auch komplizierte Sprache. Man kann viel machen – aber auch viel falsch machen! Insbesondere sind einige Dinge zu beachten, ohne die Programme nicht unbedingt falsch laufen, aber viel Laufzeit oder Speicherplatz benötigen - abgesehen von nicht aufgeräumten Resourcen. Dieser Artikel soll alle Fallen aufzeigen: Was sollte man tun, was besser lassen. Und was darf keinesfalls vergessen werden – oft bekommt man dies vom Compiler nicht mitgeteilt.

1. Programmaufbau

Jede C++-Klasse ist in einem Headerfile zu deklarieren und in einem cpp-File zu implementieren. Außer Deklarationen, Inline-Definitionen, defines darf der Headerfile keine Implementationen enthalten. Der Aufbau des Headerfiles sollte sein:

```
#ifndef filename_h
#define filename_h

//weitere Includes
//Konstante
//Klassen
//Inlines
//Funktionsdeklarationen

#endif //filename_h
```

Was die Reihenfolge des Einbindens von Headferiles angeht: Erst Systemheaderfiles, dann eigene formulieren (gegebenenfalls mit Compileroptionen Pfade angeben), da möglicherweise in eigenen Headerfiles defines formuliert werden, die Konsequenzen auf die weiteren Headerfiles haben können.

```
#include <sysheader>
#include "eigenheader"
```

In der Implementierungsdatei (cpp-Datei) werden die Methoden und Funktionen dann programmiert.

2. Konstruktor / Destruktor

In jeder Klasse ist ein Defaultkonstruktor immer zur Verfügung stellen (Defaultargument zählt):Dieser wird in vielen Situationen gebraucht, so bei der Vererbung oder Mehrfachvererbung, bei Feldern aus Objekten und wenn überhaupt nichts anders angegeben wurde. Tragisch ist, daß überhaupt ohne Konstruktor ein Dummy-Defaultkonstruktor vom Compiler erzeugt wird (der nichts initialisiert).
Im Konstruktor sind alle Daten (außer static) sauber zu initialisieren: Der Anwender darf sich darauf verlassen.“
Der Copy-Konstruktor ist immer dann explizit anzugeben, wenn Pointer oder Referenzen im Objekt stehen. Er ersetzt dann ein Copy-Konstruktor-Geschenk des Compilers, der die Daten 1:1 (also flach) kopiert, also auch Pointer. Kommen keine Pointer vor, reicht oft der automatisch zur Verfügung stehende Copy-Konstruktor. Die Syntax für den Copy-Konstruktor ist (X ist der Name der Klasse):

```
X::X(const X&);
```

Möchte man das Geschenk nicht und auch verbieten, daß ein Objekt der Klasse kopiert wird, dann kann die Deklaration des Copy-Konstruktors im private oder protected-Bereich stehen. Damit merkt der Compiler bereits, wenn ein Objekt kopiert wird. Läßt man die Definition weg, würde der Binder protestieren. Situationen, in denen der Copy-Konstruktor notwendig verwendet wird, sind: Vererbung, Komposition, call by value, return by value!

Im Konstruktor besser initialisieren als zuweisen. Im Zuweisungsteil ({}) wird der Operator = verwendet, im Initalisierungsteil der Konstruktor (ansonsten werden Objekt zuerst mit dem Defaultkonstruktor initialisiert, dann mit dem Operator= überschrieben!):

```
X::X(...):basis1(wert), basis2(wert) {...;}
X::X(...):container1(wert)           {...;}
X::X(...):attr1(wert), attr2(wert)   { //attr3=wert;
                                     }
```

Die Reihenfolge erfolgt bei Container-Objekten wie in der Deklaration formuliert, nicht wie im Init-Teil angegeben (damit beim Destruktor die entgegengesetzte Reihenfolge genommen werden kann). Sofern diese Abhängigkeit überhaupt erwünscht ist!

Ein Destruktor sollt zumindest dann programmiert werden, wenn Pointer oder Referenzen verwaltet werden, da z.B. ein dynamischer Speicherplatz andere Objekte enthalten kann, die wieder mit Destruktoren aufgeräumt werden müssen.

Der Destruktor muß virtual sein, wenn mindestens eine Methode virtual ist oder Basisklassenpointer verwendet werden:

```
virtual X::~X();
```

Nur so kann erreicht werden, daß auch Destruktoren von Unterklassenobjekten zum Zug kommen.

Wie reagiert man im Konstruktor im Fehlerfall? Entweder mit Fehlerflag und Fehlermethoden - dann bekommt man es aber nicht unbedingt mit, wenn ein Fehler passiert ist; man muß schon die Methoden benutzen (ein Beispiel dafür ist die iostream-Klassenbibliothek).
Oder man wirft eine Exception, die allerdings in C++ nicht unbedingt abgefangen werden muß. Immerhin gibt es aber einen Laufzeitfehler.
Eine weitere Möglichkeit ist, die Initialisierung in eine init-Methode auszulagern und im Konstruktor nichts kritisches weiter zu tun. Allerdings entfällt dann der Automatismus!

3. dynamischer Speicher

Dynamischen Speicher immer mit `delete` freigeben, da nur dann die Destruktoren der dahinterliegenden Objekte und ev. weiterer Objekte aufgerufen werden. Zwar sorgt ein ordentliches Betriebssystem dafür, daß am Programmende alles freigegeben wird, aber es kann sich in den weiteren Objekten ja auch um andere Resourcen (shared memory, semaphore, ...) handeln.

Kommen Felder aus Objekten vor, dann `delete[]` verwenden. Leider kann ein Compiler dies nicht prüfen, da `new[]`in einer anderen Quelldatei stehen kann.

Sowieso sind nur Adressen aus `new` verwenden, Teile freigeben geht auch nicht! Vorsicht geboten ist bei `++` und `--`, man muß sich schon den Start merken oder wieder entsprechend runter/hoch zählen.

Nach dem Löschen des Speicherplatzes, den Pointer auf `NULL` setzen. Er enthält sonst noch eine Adresse, die aber nicht mehr gültig ist:

```
delete p; p = NULL;
```

Der Pointer, der von `new` geliefert wird, ist zu überprüfen (`NULL`-Pointer, `new_handler`, `exception`). Neuere Compiler erzeugen defaultmäßig eine `bad_alloc`-Exception, bei älteren Compilern bekommt man sonst nicht mit, ob es überhaupt noch Speicher gibt.

Wenn `new` überlagert wird, sollte auch `delete` überlagert werden, da sonst die Allokier-Mechanismen möglicherweise nicht mehr zusammenpassen. Entsprechendes gilt für `new[]/delete[]`.

Speicherplatzüberschreitung ist streng zu beachten (Index >= Dimension oder < 0)! Besser sollte

man die `vector`-Klasse aus der STL-Bibliothek verwenden.

```
arr[dim]; arr[-20];
```

Insbesondere ist das Überschreiben nachfolgenden Speicherplatzes in C++ ein großes Problem und kann zu Hackereinbrüchen führen.
Immer sind saubere Adressen und Speicherplätze zu verwenden:

```
char *s; *s = '\0';
```

führt möglicherweise (nicht immer, aber dann, wenn das Programm einem Kunden vorgeführt wird) zu einem Absturz!

Sofern nicht mit der `string`-Klasse gearbeitet wird, `\0` am Ende nicht vergessen!

4. Referenzen

Objekte immer per Referenz (oder als Adresse) an Funktionen übergeben, da sonst eine temporäre Kopie gemacht wird, die den Copy-Konstruktor auslöst. Dies kostet Laufzeit und Speicher und bereitet Ärger mit dem Copy-Konstruktor-Geschenk im Falle von Pointern.
Auch im Return-Fall gilt dieses Prinzip. Nur was ist zu tun im Fehlerfall? Da es in C++ eine null-Referenz nicht gibt, funktionieren nur Dummy-Objekte oder das Auslösen einer Exception.

Nie Referenzen auf lokale Daten zurückgeben!

Außer bei `void`-Methoden immer mit return etwas sinnvolles zurückgeben (ev. als Konstante `const` oder sich selber `return *this;`). Manche Compiler meckern hier einen Fehler nicht an – zur Laufzeit wird es aber problematisch.

5. Kapselung

friends sparsam nutzen, sie umgehen die Kapselung! Methoden sind immer die bessere Lösung. Allerdings geht es manchmal nicht anders (vor allem in Verbindung mit der Operatorüberlagerung). Beim Ausgabeoperator z.B. müßte `operator<<` eine Methode der Klasse von `cout` sein – dies ist aber eine fertige Bibliotheksklasse und sollte nicht geändert werden!

```
cout << obj_einer_eigenen_Klasse;
```

Möglichkeit: Zwar als `friend` deklarieren (wegen der Zugehörigkeit und besseren Lesbarkeit), aber die Freundschaft nicht ausnutzen und intern mit Methoden arbeiten.

Ist der `protected`-Teil wirklich sinnvoll? Er erlaubt allen Unterklassen den direkten Zugriff auf die Daten! Bei sensiblen Daten bitte vermeiden und wieder mit Methoden (die den Zugriff überprüfen können) zugreifen.

6. Konstante

Methoden, die nichts ändern im Objekt, sollten `const` sein (Lese-Methoden). Damit hat der Compiler bessere Prüfmöglichkeiten.

Referenzargumente, deren Inhalt sich nicht ändert: `const`

```
xxx(const yyy& arg);
```

Pointerargumente, deren Inhalt sich nicht ändert: `const`

```
xxx(const yyy* poi);
```

Ebenfalls beim Return: `const& oder const*`

Konstante Attribute müssen im Init-Teil des Konstruktors initialisiert werden. Und dann gibt es natürlich auch konstante Objekte.

7. Operatoren

Auch den Copy-Operator immer dann überlagern, wenn Pointer oder Referenzen im Objekt stehen, denn sonst wird das Copy-Operator-Geschenk verwendet, der alles 1:1 kopiert – auch Pointer!

```
const X& X::operator=(const X&);
```

Wie beim Konstruktor auch kann er eventuell im private oder protected-Bereich nur deklariert werden und der Compiler verbietet dann das Kopieren. Oder als Dummy-Operator, der eine Fehlermeldung (Exception) auslöst. Copy-Operatoren werden auch verwendet, wenn bei Vererbung oder Komposition das Unterklassen-Objekt oder das Containerobjekt kopiert wird.

Der Operator= sollte eine Referenz auf `*this` zurückliefern (`const`), um eine Kaskadierung zu erlauben:

```
a = b = c;
```

aber nicht:

```
(a = b ) = c;
```

Und der `operator=` sollte Zuweisung auf sich selber prüfen, damit nicht der Ast abgesägt wird, auf dem man sitzt:

```
a = a;
```

Kochbuchartig kann der Copy-Operator so programmiert werden:

```
const X& X::operator=(const X& obj)
{
   if(&obj != this)
```

```
   {
      i = obj.i;        // Attribute 1:1 (besser mit Methode holen)
      delete[] p;       // Speicher freigeben oder anders nutzen
      p = new typ[neueGroesse];  // und neuen Speicher besorgen
      copy(p, obj.p);  // und reinkopieren (mit Hilfsmethode)
   }
   return *this;
}
```

Wenn `+` und `=` überlagert wird, dann auch über `+=` nachdenken! Automatisch ist das nicht auch überlagert. Genauso `==` und `!=`.

Sowieso ist zu überlegen, ob überlagerte Operatoren wirklich gut und intuitiv lesbar sind oder ob besser eine normale Methoden verwendet werden sollte? Nicht daß + zu – überlagert wird, aber oft sind Operatoren wenig aussagekräftig – mit Ausnahme von mathematischen Klassen.

Explizite Typkonvertierungen vermeiden. Der Compiler weiß oft besser, ob es sinnvoll ist oder nicht. Auch die Überlagerung der Typkonvertier-Operatoren meiden, da oft Mehrdeutigkeiten entstehen. Besser ist eine echte Methode:

```
int toInt() const;
```

Dann: zwischen pre und post ++/-- unterscheiden! Die Operatoren erhalten im post-Fall ein Dummy-Argument, das niemals genutzt werden sollte:

```
typ xxx::operator++();
typ xxx::operator++(int);
```

Natürlich ist sauber zwischen `=` und `==` zu unterscheiden. Ersteres ist die Zuweisung, zweites der Vergleich, wobei

```
if(a = b){ ...; }
```

gültiger Code ist, bei dem a auf b gesetzt wird – und wenn dieser Zuweisungswert nicht 0 ist, if auch noch wahr bekommt!

Auch ist die Initialisierung von der Zuweisung zu unterscheiden: Initialisieren wird beim Anlegen des Speicherplatzes gemacht (idR vom Konstruktor), Zuweisen kann man jederzeit später auf ein existierendes Objekt:

```
int a = 10; a = 20;
Klasse obj(10); obj = 20;
```

8. Polymorphie

Zum Erkennen der Objekte nicht abfragen: „Bin ich nun ein X-Objekt, dann mach ich's so; bin ich aber jemand anders, mach ich's eben anders“! Dieses Erkennen ist Aufgabe des Schlüsselworts `virtual`, alle dynamisch polymorphen Methoden müssen in der Basisklasse so deklariert werden.

Im Gegenzug dazu nicht-`virtual`-Methoden in Unterklassen nicht überlagern, da im Zweifelsfall nur die Basisklassen-Methode zum Zuge kommt!

Beim Überladen auf eine gleiche Schnittstelle achten: Das Interface muß in der Basisklasse (oder in einer eigenen Interfaceklasse) bereits deklariert sein!

```
int f(int);    und
int f(double);
```

sind unterschiedliche Methoden!

Dies alles setzt natürlich ein sauberes Klassen-Konzept voraus!

9. Hautprogramm

Das Hauptprogramm immer mit return beenden, denn sonst werden lokale Objekte nicht mit deren Destruktoren aufgeräumt ! Abhilfe schafft

- ➔ Destruktor für Objekte explizit aufrufen
  `obj.klasse::~klasse();`
- ➔ Objekte static deklarieren

Sowieso ist immer ein definierter Exit-Code ans Betriebssystem zurückzugeben

```
return 0;
exit(0);
```

um in möglichen Skripten den Erfolg des Programms sehen zu können (0 gilt als ok, 1-255 als Fehler).

10.Vererbung

Soll in der Klassenhierarchie Mehrfachvererbung verwendet werden, dann ist bereits ganz oben `virtual` vererben, ansonsten werden Methoden oder Attribute leicht mehrdeutig. Nachträgliches `virtual`-machen ist nicht ratsam, da der gesamte Klassen-Strukturaufbau anders wird. Mit Laufzeit-Nachteilen muß bei `virtual`-Vererbung gerechnet werden, da möglicherweise zusätzliche Pointer innerhalb der Klassen-Struktur verwendet werden.

Die oberste Klasse (Superbasisklasse) sollte als abstrakte Klasse formuliert werden (also nur die Schnittstellen bekanntgeben, noch keine Implementierungen besitzen). Dies fördert die Wiederverwendbarkeit.

Sauber zwischen Lese-, Setze- und Arbeits-Methoden unterscheiden, auch dies dient der Wiederverwendung!

Im Normalfall `public` erben. Die Möglichkeiten des `protected` und `private` erbens sind für Spezialfälle reserviert.

11.Templates

Alles `inline` im Header unterbringen! Eine Trennung Deklaration/Definition wird von vielen Compilern nicht akzeptiert! Eine `friend` und `template` Kombination ist für einige Compiler ein Problem, was bis zu Compilerabstürzen führen kann. Neuere Compiler erwarten, daß `friend`-Funktionen eigene Templateparameter haben müssen:

```
template <class T>
class XXX
{
   template <class T1, class T2>
   friend T1 funktion(T1, T2, XXX<T1>);
};
```

Somit können solche Freunde auch eine Freundschaft zu anderen Klassen (mit anderen Template-Parametern) eingehen.

Vorsicht ist geboten bei der Überlagerung von Template- und Normal-Methoden :

```
int f(int);
Typ f(Typ);
```

Sollte der Template-Typ `int` werden, wäre die Methode doppelt vorhanden! Auf der anderen Seite können Template-Methoden durchaus mehrfach für spezielle Typen definiert sein und werden dann für diesen Typ bevorzugt verwendet.

12.Exceptions

Exceptions sind in C++ zwar implementiert, der Programmierer wird aber nicht gezwungen, sie zu verwenden oder auch nur sauber zu deklarieren. Dies ist historisch dadurch bedingt, daß Exceptions erst sehr spät in die Sprache C++ aufgenommen wurden und eine Vielzahl an Bibliotheksklassen sie somit (noch) nicht verwendet. Als Empfehlung: Exceptions nutzen, sauber deklarieren und Exceptions immer abfangen oder nach oben weiterreichen.

Wenn eine Exception im Konstruktor ausgeworfen wird, muß bisheriges selber freigeräumt werden. Der Destruktor wird nicht aufgerufen.

Eine Exception im Oberklassenkonstruktor ausgelöst, kann in Unterklasse nicht abgefangen werden.

Vorsicht mit einer Exception im Destruktor, da der Destruktor auch durch eine Exception aufgerufen werden kann.

Unterscheiden zwischen 'catch by reference' und 'catch by value'. Eventuell muß die Speicherverwaltung beachtet werden; Auch der Copy-Konstruktor kann ein Problem darstellen.

Exceptions immer so deklarieren:

```
void f(...) throw(Typ);
```

13.Diverses

Keine Zuweisung an `this` verwenden. Dies war früher ein Weg zur dynamischen Speicherverwaltung. Besser new überlagern.

Auch `delete this` ist problematisch, da man ja noch im Objekt drin ist, was gerade weggeworfen werden soll.
Keine #`define`-Makros und #`define`-Konstanten mehr verwenden. `Inline`-Funktionen und echte Konstante `const` sind sicherer!

Gemieden werden sollte:
- Mehrfachvererbung
- intensive `Operator`-Überlagerung
- `struct` (da alles öffentlich)

Voreinstellungen sollten nicht ausgenutzt werden, besser immer schreiben:
- `int`-Retruntyp
- `private`-Vererbung
- Default-Konstruktor
- Copy-Konstruktor
- Copy-Operator

Objekte nicht zu `NULL` setzen (wie Pointer), sie überbügeln möglicherweise das gesamte Objekt:
```
string s = NULL;
```

Compiler-Warnungen immer ernst nehmen!

Bei der Funktionsüberlagerung beachten:
- Mehrdeutigkeiten
- automatische Typkonvertierung
- Defaultargumentenwerte (und nur ein Mal angeben)
- Templategeneration
- Vererbung

Bei gleichen Typen ist dies auch der Ergebnis-Typ! Ganzzahl durch Ganzzahl ist Ganzzahl:
`10 / 3` == 3 nicht 3.333

Die Typerweiterung zum Namen setzen, nicht zum Grundtyp:
```
int*    p1, p2;
```

Dies wird leicht gelesen: es sind beides Pointer. Was aber nicht stimmt! Besser:
```
int    *p1, p2;
```

Einem `NULL`-Pointer niemals einen Inhalt zuweisen. `NULL` ist keine gültige Adresse!

Ein möglicher Over/Under-flow von Ganzzahldatentypen ist rechtzeitig zu beachten, da sonst der Wert ungültig wird, ohne dies wieder feststellen zu können.

16bit/32bit/64bit-Abhängikeiten beachten!

Pointer niemals in `int` wandeln (und eventuell wieder zurück)! Wenn doch einmal notwendig, `reinterpret_cast<>()` verwenden.

Globalen Variablen nicht mehr verwenden. Globale Objekt sind durch deren Kapsel allerdings möglich.

Kein goto verwenden – es geht wirklich ohne!

Nebeneffekte nicht ausnutzen, da compilerabhängig:
```
a[i] = i++;
f(i, ++i);
```

Einer static-Variablen nicht `this` zuweisen. Außerdem immer einmal außerhalb der Klasse ein `static`-Member deklarieren und initialisieren (auch wenn manche Compiler dies implizit selber tun).

`iostream.h`-Klassen verwenden statt `stdio.h`-Funktionen, da die Klassen ein typsicheres Konzept anbieten, das mit `printf/scanf` nicht funktioniert. Vor allem beide Möglichkeiten nicht mischen, da unterschiedliche Puffer- und Flush-Mechanismen verwendet werden.

Genauso `new/delete` statt `malloc/free` verwenden, da diese den Typ kennen und Konstruktor/Destruktor automatisch aufrufen. Auch hier nicht mischen!

`iostream` mit `namespace` ist besser (erweiterbarer). Erreicht wird dies bei Standardheaderfiles zunächst durch Weglassen der Endung .h (C-Headerfiles mit vorgestelltem c):
```
#include <header_ohne_h>
#include <iostream>
#include <cstdlib>
```
und mit expliziter Nennung des Namensraums (manche Compiler verwenden `std` standardmäßig):
```
using namespace std;
```

Keinen Kommentar zum Ausblenden von Code-Stücken verwenden, sondern (Vorteil: Einschalten über Compileroptionen):
```
#ifdef xxx
...
#endif
```

14.Konventionen

Über Konventionen können ganze Bücher gefüllt werden. Hier sollen nur wichtige oder problematische Konventionen angesprochen werden.

- Klammerung:
  {
  } // immer untereinanderschreiben ist lesbarer, dazwischen einrücken (TAB oder Blanks)

  {
  }; // kein Semikolon nach Klammer-zu mit einer Ausnahme: nach der Klassendeklaration
  // da sost Probleme z.B.: `if() {}; else xxx;`
- Kontrollstrukturen immer mit Klammer:
  ```
  while(...)
  {
  }
  ```
  nicht:
  ```
  while()
     xxx;
     yyy;
  if()
    if()
      xxx;
  else yyy;
  ```
- `break` bei `switch` nicht vergessen
- Klassen-Name groß beginnen, objekt-Name klein, macheMethode() gemischt. Dies erhöht wesentlich die Lesbarkeit
- pxx-Pointer, rxx-Referenz, xx-Variable: Im Namen der Variable sollte bereits auf den Typ geschlossen werden können.
- in Oberklasse nicht zuviel unterbringen (schlecht wiederverwendbar)
- Interface-, Abstrakte-Klasse verwenden als oberste Oberklasse. In C++ kann eine Interfaceklasse dadurch gebaut werden, indem dort nur pure-virtual-Methoden stehen und keine Daten
- keine Daten im `public`-Bereich, da keine Kapselung
- `protected`-Teil ist eventuell problematisch (durch Zugriff aus Unterklassen)
- keinen Pointer oder Verweis auf private Daten (über Returnwerte bei Methoden) herausgeben, da diese dann (trotz `private`) geändert werden können
- keine Referenz auf ein lokal dynamisch angelegtes Objekt zurückgeben (wer soll aufräumen?),
- jedes Objekt sollte seinen Speicher selber verwalten inklusive `new` und `delete`
- Vererbung implementiert die "ist"-Relation, nicht die enthält-Beziehung!
- eine nicht virtual-Methode in einer Unterklasse nicht überlagern!
- down-casts vermeiden! Wenn notwendig, dann `dynamic_cast<typ>` verwenden
- private/protected-Vererbung meiden, da dies nicht der normale Vererbungsweg ist.
- Vererbung/Template: Unterschied beachten!
- Immer Konstruktor programmieren(da sonst ein Geschenk)
- neue Cast-Operatoren verwenden, da sicherer!
- eventuell Handle-Klasse zur sauberen Speicherverwaltung nutzen (reference counting)
- overload, um Typkonversion zu vermeiden
- Zusatzkosten bei virtual, RTTI und virtual Vererbung beachten

- SmartPointer statt native Pointer benutzen (größere Sicherheit)
- dynamische Pointer in Objekten auf dem Stack verstecken (die dann automatisch abgebaut werden und durch den Destruktor den Speicher freigeben)

15.OO-Tools

Die Analyse und das Design von OO-Programmen sollte immer mit Hilfe von UML durchgeführt werden. Nur so ist ein Überblick möglich, der spätere Änderungen einfach erlaubt. Die UML-Diagramme erlauben auch bzw. sind schon eine Dokumentation und ermöglichen weitgehend eine Generierung des Quell-Codes. Mit Hilfe dieser Diagramme wird die zu programmierende Software auch besser verstanden und Rückfragen ermöglicht.

UML stellt eine Vielzahl an Diagrammen zur Verfügung, nicht alle müssen benutzt werden. In der Praxis haben sich bewährt:

- ✔ Use-Case-Diagramm
- ✔ Klassendiagramm
- ✔ Dynamikdiagramme (Sequencechart, Zustandsdiagramm)
- ✔ Komponenten-Diagramm

16.Literatur

Effective C++, Scott Meyers, Addison Wesley, 0-201-56364-9

More Effective C++, Scott Meyers, Addison Wesley, 0-201-63371-X

Advanced C++, James O. Coplien, Addison Wesley, 0-201-54855-0

C++ Primer, Stanley B. Lippman, Addison Wesley, 0-201-54848-8

C++ Strategies and Tactics,
Robert B. Murray, Addison Wesley, 0-201-56382-7

Tips und Tricks zu UNIX, C und C++, Stockmayer, Hanser

zur Person:

Christoph Stockmayer ist seit 20 Jahren freiberuflicher Trainer in den Gebieten Programmierung, C/C++/Java, OOA/OOD und im gesamten UNIX/Linux-Sektor. Er ist SuSE Certified Linux-Trainer und Lehrbeauftragter an der FH Nürnberg. Er betreut außerdem Programmier- und UNIX/Linux-Projekte.