

# Prorammierrichtlinien

Programmierrichtlinien sind in jedem C bzw. C++ Projekt vor Beginn der Programmierung für alle Projektteilnehmer verbindlich zu regeln.

Die hier vorgestellten Richtlinien sind beispielhaft für eine solche Regelung anzusehen.

Verwiesen wird auch auf die [GNU Coding Standards](#).

## Inhalt

1	Einleitung .....	4
1.1	Anwendungsbereich .....	4
1.2	Begriffe .....	6
2	Programmierrichtlinien für C .....	9
2.1	Allgemeines .....	9
2.2	Fileorganisation.....	10
2.3	Kommentierung .....	11
2.4	Layout .....	13
2.5	Namenskonventionen.....	14

2.6	Deklarationen .....	15
2.7	Initialisierung.....	17
2.8	Kontrollfluss .....	17
2.9	Defensivprogrammierung, Fehlerbehandlung .....	19
2.10	Typumwandlungen .....	20
2.11	Sonstiges .....	21
3	Programmierrichtlinien für C++ .....	22
3.1	Allgemeines.....	22
3.2	Layout .....	23
3.3	Namenskonventionen.....	24
3.4	Deklarationen .....	24
3.5	Konstruktoren und Destruktoren .....	26
3.6	Initialisierungen .....	28
3.7	Defensivprogrammierung, Fehlerbehandlung .....	28
3.8	Typumwandlungen .....	28
3.9	Sonstiges .....	29

4	Namenskonventionen .....	30
4.1	Konstanten.....	30
4.2	Benutzerdefinierte Typen .....	30
4.3	Klassen.....	31
4.4	Funktionen.....	32
4.5	Variablen .....	32
4.5.1	Basistypen .....	32
4.5.2	Präfixe.....	34
4.5.3	Bezeichner .....	35
4.5.4	Beispiele für die ungarische Notation.....	36

# 1 Einleitung

Einen Beitrag zur Gewährleistung einer hohen **Softwarequalität** leisten firmen- bzw. projektspezifische Richtlinien für die Softwareentwicklung. Sie dienen insbesondere als

- Arbeitsanleitung,
- Kommunikationsbasis und
- Vertragsgrundlage.

Die vorliegende Programmierrichtlinie enthält Regeln für das Codieren und Kommentieren in der Programmiersprache C und C++.

## 1.1 Anwendungsbereich

Das Ziel der Richtlinie besteht darin, bei Softwareentwicklungen in der Firma <YourCompany> oder bei der Vergabe von Entwicklungsaufträgen die **Qualität** der Programme dadurch zu **verbessern**, dass

- **bekannte fehleranfällige Sprachkonstrukte vermieden** und
- **anerkannte qualitätssichernde Elemente empfohlen** werden.

Insbesondere sollen Eigenschaften wie **Zuverlässigkeit**, **Robustheit**, **Wartbarkeit** und **Testbarkeit** der Software gesichert werden.

Es ist **kein** vorrangiges **Ziel** der Richtlinie, durch ihre Anwendung die Programme **portabel** zu gestalten. Portabilität ist in vielen Fällen nicht erforderlich, und die sich unmittelbar aus ihr

ergebende Forderung nach Normkonformität widerspricht in einigen Fällen sogar den hier formulierten Regeln.

Unabhängig davon sollte sich kein Programmierer ohne zwingenden Grund von den existierenden Sprachstandards entfernen und proprietäre Sprachkonstrukte einsetzen.

Es ist ebenfalls **kein** Ziel der Richtlinie, durch Weitergabe besonderer Programmiertipps und –tricks die zu entwickelnde Software **effizient** zu machen. Solche Art von Effizienz auf der einen Seite und Zuverlässigkeit und Wartbarkeit auf der anderen Seite führen in der Regel zu einander widersprechenden Forderungen an den Code. Als Konsequenz ist die Anwendung der Richtlinie also zumindest dann nicht sinnvoll, wenn die zu entwickelnde Software unter allen Umständen vorgegebene enge Effizienzkriterien erfüllen muss, z.B. wenn harte Echtzeitforderungen bestehen. In diesen Fällen können bestimmte Regeln nicht angewendet werden und Qualitätsnachweise müssen, wenn sie erforderlich sind, gesondert geführt werden. Regeln, die sich z.B. auf die Code-Kommentierung oder die Namenskonventionen beziehen, stehen der Effizienz jedoch nicht im Wege.

Die Anwendung der Richtlinie wird empfohlen bei Softwareentwicklungen wenn

- deren Einsatz im gesetzlich geregelten oder Dienstleistungsbereich erfolgt oder vorgesehen ist ,
- die fertige Software veräußert werden soll,
- an Entwicklung, Wartung und Einsatz Personen unterschiedlicher Labore oder Projekte beteiligt sind.

Bei der Vergabe von Softwareentwicklungsaufträgen an Fremdfirmen kann die Richtlinie dann vertragswirksam angewendet werden, wenn der Auftragnehmer keine adäquate hauseigene Richtlinie zur Verfügung hat.

Darüber hinaus wird die Anwendung der Richtlinie bei jeder weiteren Softwareentwicklung empfohlen,

- wenn deren Umfang 500 Quelltextzeilen (ohne Kommentar) überschreitet oder
- wenn deren vorgesehene Anwendungsdauer mindestens 1 Jahr beträgt.

## 1.2 Begriffe

Die folgenden Begriffe werden in den Regeln mit der angegebenen Bedeutung verwendet:

Begriff	Bedeutung
Symbol	Durch den Programmierer eingeführtes, eindeutig identifizierbares, neues Programmwort (Variable, benannte Konstante, Funktion, Typ, Makro, Parameter, Register, Tag, Label, ...). Der Name muss nicht eindeutig sein und kann u.U. auch im Code fehlen.
Globales Symbol	Symbol mit dem Gültigkeitsbereich Datei (file scope).
Token	„Wort“ eines Programms (Operator, Interpunktionszeichen/Begrenzer, Symbol, Keyword, Konstante).
Modul	Sammlung von Deklarationen und Definitionen, die separat kompilierbar ist (C-File mit Includefiles).

Variablendefinition	Vereinbarung einer Variablen mit gleichzeitiger Allokierung von Speicher.
Variablendeklaration	Verweis auf eine Variablendefinition, Bekanntgabe eines Symbolnamens.
Funktionsdefinition	Implementierung einer Funktion (Deklarator, Parameterliste und Funktionsrumpf).
Funktionsdeklaration	Verweis auf eine Funktionsdefinition, Bekanntgabe eines Symbolnamens.
Prototyp(deklaration)	Spezielle, ausführliche Form der Funktionsdeklaration mit den Datentypen der Parameter.
Klasse	Typ, der Variablen oder Konstanten (= Datenelemente) und Funktionen (= Elementfunktionen, Methoden) zusammenfasst.
Instanz, Objekt	Variable, deren Typ eine Klasse ist.
Objektstatus	Momentaner Inhalt der Datenelemente eines Objekts.
Klassendeklaration	Verweis auf eine Definition einer Klasse, Bekanntgabe eines Symbolnamens.
Basisklasse	Klasse, die ihre Datenelemente und Funktionen vererbt.
Abgeleitete Klasse	Klasse, die Datenelemente und Funktionen erbt

Access-Funktion	Einfache Elementfunktion, die den Wert eines Datenelements zurückliefert.
Mutator-Funktion	Einfache Elementfunktion, die den Wert eines Datenelements setzt.

## 2 Programmierrichtlinien für C

Dieser Abschnitt enthält die Regeln für das Codieren und Kommentieren. Einzelne Regeln sind um kurze Erläuterungen oder Empfehlungen ergänzt. Diese Texte sind nicht Bestandteil der jeweiligen Regel.

Der Grad der Verbindlichkeit ist für alle Regeln gleich und hängt nicht davon ab, ob die Regel mittels „muss“, „sollte“ oder „ist ... zu“ gebildet wird.

### 2.1 Allgemeines

- (1) Wenn eine Regel aus einem besonderen Grund nicht eingehalten werden kann, ist dies kurz im Code zu dokumentieren und zu begründen.

Es ist möglich, dass fachliche Gründe die Einhaltung einer Regel unmöglich machen.

- (2) In reinen C-Umgebungen ist ANSI C zu verwenden. Compilerabhängige Sprachkonstrukte sind zu vermeiden. Die ‘Common Extensions’ sowie Bibliotheken können verwendet werden.

Dies setzt voraus, dass ein ANSI-konformer Compiler vorhanden ist. ANSI C ist definiert im Standard ISO/IEC 9899:1999.

Fehlerbehandlungsverfahren, die ein Betriebssystem anbietet, sind in der Regel nicht ANSI-konform, können aber immer dann verwendet werden, wenn die Software nicht portiert werden muss. Dasselbe gilt für die Ausnahmebehandlung mit `try/catch/throw`.

## 2.2 Fileorganisation

- (3) Falls das System portiert oder transportiert werden muss, sind Filenamen entsprechend den ISO-Konventionen zu wählen.

In der ISO 9660 sind folgende Regeln für File- und Verzeichnisnamen festgelegt: Für die Namen sind Buchstaben, Ziffern und Underscore '\_' zugelassen. Verzeichnisnamen sind max. 8 Zeichen lang und haben keine Namenserverweiterung. Filenamen sind max. 8 Zeichen lang und haben eine Namenserverweiterung mit max. 3 Zeichen. Name und Namenserverweiterung werden durch ein Dot '.' getrennt.

- (4) Die Namen der Standard-Headerfiles sind nicht für nutzereigene Files zu verwenden.
- (5) Die zum Export vorgesehenen Deklarationen eines \*.c-Files sind in ein analog benanntes Headerfile zu schreiben. Jede Variablen- und Konstantendeklaration in diesem Headerfile muss das Keyword `extern` aufweisen. Das Headerfile darf außer `inlines` keine Funktionsdefinitionen enthalten.
- (6) Falls das System portiert werden muss, sind nichtnormkonforme Zugriffe zu Hardware und Betriebssystem, die sich nicht vermeiden lassen, in einem separaten Modul zusammenzufassen.

Dies betrifft auch implementationsabhängige Teile, wie z.B. Operationen, die eine bestimmte byte-Ordnung voraussetzen, oder die Verwendung von wide chars.

- (7) Die Include-Anweisungen dürfen keine absoluten Pfade enthalten.

In der Regel kann jedem Compiler über Umgebungsvariablen die Lage der Include-Verzeichnisse mitgeteilt werden. Dann ist keinerlei Pfadangabe nötig. Falls das nicht möglich ist, müssen die zu includierenden Files im aktuellen Verzeichnis oder seinen Unterverzeichnissen deponiert werden.

## 2.3 Kommentierung

(8) Jedes File ist mit einem Kommentar einzuleiten. Dieser sollte Name und Zweck des Files, Autor, Erstellungsdatum oder Version sowie eine Änderungsliste enthalten.

Diese Regel bezieht sich nicht nur auf \*.c-Files, sondern auch auf Headerfiles.

Der Kommentar kann auch verschiedene andere Dinge auflisten, z.B. die im File enthaltenen Deklarationen, evtl. bekannte Restriktionen, noch fehlende Teile, eine Liste der nicht vermeidbaren Compilerwarnungen u.a.

```
/*
 * File Name:      bitcount.c
 * Purpose:       Function and test program to count the number of
 *               1 bits of a unsigned integer.
 * Author:        a.schuette@fbi.fh-darmstadt.de
 * Changes:
 *
 * 11.01.2002    initial version
 * 18.01.2002    modification to be able to handle neg. values
 */
```

(9) Funktionsdefinitionen sind mit einem Kommentar einzuleiten. Dieser sollte Name und Zweck der Funktion, Bedeutung der Parameter und des Rückkehrwertes, evtl.

Definitionsbereich der Eingabeparameter, verwendete globale Symbole, Seiteneffekte oder sonstige Besonderheiten enthalten.

```
/* **** */
* int StringCopy (char *Dest, const char *Source, const int n)
* Purpose:
*   Kopiert n Bytes von einem String Source zu einem String Dest. Das
*   Endezeichen 0 wird nach dem Kopieren an Dest angefügt.
* Parameter:
*   n           Anzahl zu kopierender Bytes. Muss groesser 0 sein.
*   Source      Zeiger auf Quelle. Muss mind. n Zeichen enthalten.
*   Dest        Zeiger auf Zielbuffer. Muss mind. n+1 Zeichen aufnehmen
*               koennen.
* Return value:
*   1           Success;
*   0           Failure (Dest nicht gueltig oder innerhalb der
*               ersten n Zeichen von Source taucht das
*               Endezeichen 0 auf.)
* Global Symbols:
*   Im Fehlerfall wird MyStringError > 0 gesetzt.
/* **** */
```

(10) Jede Definition oder Deklaration einer globalen Variablen sollte in einer separaten Zeile stehen und kommentiert sein. Falls der Definitionsbereich der Variablen im Programm kleiner ist als der durch ihren Datentyp zugelassene Bereich, ist das im Kommentar zu vermerken.

- (11) Die Direktiven `#else` und `#endif` sind mit einem Kommentar zu versehen, der beschreibt, zu welchem `#if(def)` sie gehören.
- (12) Überflüssige, im Laufe der Entwicklung entstandene und in Kommentar umgewandelte Quelltextabschnitte müssen in der endgültigen Fassung entfernt werden.

## 2.4 Layout

- (13) Funktionen sollten nicht länger als eine Bildschirmseite (etwa 25 Zeilen) sein.

Die genaue Anzahl von Zeilen ist nicht wichtig. Der hinter dieser Regel stehende Gedanke ist, dass Blättern auf dem Bildschirm das schnelle, intuitive Erfassen einer Funktion behindert und lange Funktionen schlecht testbar sind. Je einfacher eine Funktion strukturiert ist (einfache Sequenz), desto länger kann sie sein, ohne ihre Testbarkeit zu verlieren.

- (14) Der Code ist durch gezielte Einrückungen und optische, blockweise Strukturierung lesbar zu gestalten.
- (15) Kein Token darf durch die Zeichenkombination Backslash+Zeilenumbruch unterbrochen werden.
- (16) Falls lange Zeilen geteilt werden müssen, ist der Zeilenumbruch vor, nicht nach einem Operator einzufügen.

Der Operator der obersten Klammerungsebene sollte die Fortsetzungszeile beginnen.

Beispiel:

```
if ((thisIsALongLine && thisIsADemonstration)
    || thisLineContainsMacros) ...
```

- (17) Keine Anweisung sollte über mehr als zwei Zeilen gehen.
- (18) Bei geschachtelten `if`-Anweisungen müssen alle Stufen außer der innersten geklammert sein.
- (19) Ein Funktionenzeiger `pfct` ist immer in der Form `(*pfct)()` zu deklarieren.

Die einfachere Variante `pfct()` wird nicht von allen Compilern unterstützt und lässt sich auch nicht auf den ersten Blick von einem Aufruf unterscheiden. Deswegen ist die explizite Schreibweise `(*pfct)()` zu bevorzugen.

## 2.5 Namenskonventionen

- (20) Vor der Implementierung sind Namenskonventionen aufzustellen, die konsequent eingehalten werden müssen.

Empfehlenswert ist, diese Namenskonventionen in einem der Filekommentare niederzulegen.

- (21) Symbolnamen sollten selbsterklärend sein. Sie sind geeignet zu untergliedern, wenn sie aus mehreren Wörtern bestehen. Kein für den Linker sichtbares Symbol darf mit einem Underscore beginnen.
- (22) Die ANSI-C-Keywords und die Namen der in den ANSI-Standard-Headerfiles deklarierten Symbole sind als nutzereigene Symbolnamen (auch in Großbuchstaben) nicht zulässig.
- (23) Namen von globalen Symbolen sollten mindestens 6 Zeichen lang sein.

- (24) Falls Konstanten wie `FALSE` und `TRUE`, `YES` und `NO` u.ä. definiert werden, ist der semantisch negativ belegten Konstanten (`FALSE`, `NO`) der Wert `0` zuzuweisen.

## 2.6 Deklarationen

- (25) Jede Variable sollte den kleinstmöglichen Gültigkeitsbereich haben.

Variablen, die nur in einer Funktion verwendet werden, sind lokal zu dieser zu deklarieren. Variablen, die nur in einem Modul verwendet werden, sind als `static` zu deklarieren.

Es ist nicht gefordert, zusätzliche Blöcke einzuführen, um das data hiding zu perfektionieren.

- (26) Neue Datentypen sind über eine `typedef`-Definition, nicht über die Präprozessordirektive `#define` zu definieren.

- (27) Datentypen sind immer vollständig anzugeben.

Statt `unsigned i;` ist `unsigned int i;` zu schreiben.

- (28) Jede Funktionsdeklaration muss explizit den Rückkehrtyp, die Parametertypen und die Parameternamen enthalten.

- (29) Die Funktion `main()` muss einen Rückkehrwert haben.

Die Funktion sollte einen der vordefinierten Werte `EXIT_SUCCESS` oder `EXIT_FAILURE` zurückgeben, falls nicht detailliertere Fehlerinformationen an das Betriebssystem oder den Elternprozess weitergegeben werden sollen.

(30) Definitionen von strukturierten Typen mit `struct`, `union` oder `enum` innerhalb einer anderen Typdefinition sind nicht zulässig.

Diese Regel wird wegen der schlechten Lesbarkeit von geschachtelten Definitionen aufgestellt, auch wenn sie dem Prinzip der Datenkapselung widerspricht. Hiermit sind auch anonyme Typen mit `struct` oder `union` nicht möglich.

(31) Zahlenwerte sollten nicht direkt in den Code geschrieben werden. Sie sind vorher über eine `const`-Variablendefinition oder über eine `enum`-Definition festzulegen.

Für die vorherige Vereinbarung von Konstanten gibt es einige Ausnahmen:

- die Zahlen `-1`, `0` und `1`, wenn sie im Sinne boolescher Werte verwendet werden,
- einzelne Zeichen wie `'\n'`, leere Zeichen und leere Strings,
- der zweite Operand von Bit-Operationen (Shifts, Maskierungen).

Die Direktive `#define` sollte sparsam verwendet werden, da die entsprechenden Bezeichner für die meisten Debugger unsichtbar sind. Sie kann aber verwendet werden, wenn

- Umgebungsvariablen für die bedingte Kompilierung definiert werden sollen (`#define MSDOS`),
- der Compiler die Verwendung einer `const`-Konstanten in einem Initialisierungsausdruck nicht zulässt,
- Konstanten definiert werden müssen, die unter keinen Umständen während der Laufzeit geändert werden dürfen.

- (32) Jeder Makroausdruck als Ganzes ist in Klammern zu setzen. Bei funktionalen Makros ist außerdem jedes Auftreten jedes Parameters ebenfalls in Klammern zu setzen.

Beispiel: `#define abs(x) ( (x) >= 0 ? (x) : -(x) )`

- (33) Makroaufrufe dürfen keine Seiteneffekte enthalten.

Im obigen Beispiel hätte der Aufruf `abs(i++);` undefiniertes Verhalten zur Folge.

## 2.7 Initialisierung

- (34) Die erste Konstante einer `enum`-Deklaration muss entweder als „Nullelement“ vorgesehen werden oder einen Wert größer 0 zugewiesen bekommen.

Die Einführung eines Nullelements dient der besseren Fehlererkennung. Es sollte also entweder `enum tagColor { NOCOLOR, WHITE, YELLOW, BLACK };` oder `enum tagColor { WHITE = 1, YELLOW, BLACK };` vereinbart werden.

- (35) Für die Initialisierung von globalen Symbolen dürfen nicht globale Symbole aus anderen Modulen verwendet werden.

Die Reihenfolge, in der globale Symbole definiert werden, ist nur innerhalb eines Moduls festgelegt. Sind die Symbole in unterschiedlichen Modulen definiert, ist ihre Definitionsreihenfolge nicht bestimmt.

## 2.8 Kontrollfluss

- (36) Die `goto`-Anweisung und die Funktionsaufrufe `setjmp()` und `longjmp()` sollten nicht verwendet werden.

Die einzigen zugelassenen Sprünge sind Sprünge aus einer Schleife heraus an das Schleifenende (`break`, `continue`) und Sprünge zur Ausnahmebehandlung (`try/catch`).

(37) Falls ein Schleifenkörper nur aus einer Nullanweisung besteht, ist ein Kommentar einzufügen.

Falls alle in der Schleife benötigten Berechnungen schon in der Schleifenbedingung vorgenommen werden, kann als Kommentar `/* do nothing */` oder `/* NOP */` eingefügt werden. Eine andere übliche Variante der Kennzeichnung ist die Form `while (cond) continue;`.

(38) Jeder `case`-Zweig einer `switch`-Anweisung muss mit `break` oder mit dem Kommentar `/* fall through */` beendet werden.

(39) Jedes `switch`-Konstrukt muss mit einem `default`-Zweig beendet werden.

(40) In aktuellen Parameterlisten sind keine Seiteneffekte erlaubt.

Die Reihenfolge der Berechnung der einzelnen Parameter ist nicht festgelegt. Das Komma in der aktuellen Parameterliste ist **nicht** der Komma-Operator.

Im Aufruf `a = 4; func(a, ++a);` ist z.B. nicht festgelegt, ob `func` mit `(4, 5)` oder mit `(5, 5)` aufgerufen wird.

(41) Jede Variable, die in einer eingebetteten Zuweisung gesetzt wird, darf in derselben Anweisung nicht noch einmal vorkommen.

Eingebettete Zuweisungen sind neben expliziten geschachtelten Zuweisungen auch Inkrementierung/Dekrementierung und das Auftreten als Ausgabeparameter eines Funktionsaufrufs.

Der Sprachstandard legt nicht fest, in welcher Reihenfolge die Glieder eines Ausdrucks berechnet werden. Im Ausdruck  $(x++ + 2*x)$  werden zwar die Einzeltermine von links nach rechts addiert (Auswertungsreihenfolge des  $op+$ ), aber es ist nicht festgelegt, dass die Einzeltermine vor der Addition auch in dieser Reihenfolge ermittelt werden. Ähnliche Probleme liefern Zuweisungen wie  $c = b + f(\&b);$  oder  $x[i] = i++;$

- (42) Falls das Programm auf ein bestimmtes Ereignis wartet (Beendigung einer Berechnung, Interrupt, Nutzereingabe, ...) muss dies für den Benutzer erkennbar sein.

## 2.9 Defensivprogrammierung, Fehlerbehandlung

- (43) Vor der Implementierung ist ein System der Fehlerbehandlung festzulegen, das konsequent eingehalten wird.

Die Bibliotheksfunktionen bieten hier kein einheitliches System.

Neben den gängigen Methoden (Rückkehrwerte und globale Fehlervariablen oder terminierende Fehlerbehandlungsfunktionen) können auch Assertions oder Exception Handling genutzt werden. Günstig ist es, das Fehlerbehandlungsprinzip in einem der Filekommentare niederzulegen.

- (44) Jede Möglichkeit zum Testen von Rückkehrwerten oder Statusvariablen auf Erfolg oder Misserfolg muss genutzt werden.

Das betrifft auch die Rückkehrwerte von Standardfunktionen, besonders bei Allokierungsfunktionen.

- (45) Jede Division muss durch einen vorherigen Null-Test des Nenners geschützt werden.

Statt `result = a/b;` ist zu schreiben `if (b != 0) result = a/b; else ..`

```
oder assert (b != 0); result = a/b;
```

- (46) Jedes Wurzelziehen muss durch einen vorherigen Test des Radikanden geschützt werden.
- (47) Jedes Logarithmieren muss durch einen vorherigen Test des Arguments geschützt werden.
- (48) Falls ein Ausdruck auf Wahrheit getestet werden soll, ist er immer mit falsch (`false`, `0`, `..`) zu vergleichen.

Da ANSI C keine Aussage über den Wert eines wahren Ausdrucks trifft, ist statt `if (result == 1)` zu schreiben `if (result != 0)`. Der Ausdruck `if (result)` muss laut Standard vom Compiler als `if (result != 0)` interpretiert werden.

## 2.10 Typumwandlungen

- (49) Die Verwendung von Varianten (`union`) ist zu minimieren.
- (50) Typumwandlungen zum Maskieren einer Zahl sind nicht zulässig.
- (51) Ausdrücke, die `signed`- und `unsigned`-Variablen mischen, sind explizit nach `signed` oder `unsigned` zu konvertieren.

Das ist als Hinweis für den Programmierer gedacht, nicht für den Compiler.

- (52) Falls Variablen des Typs `char` in arithmetischen Ausdrücken verwendet werden, müssen die Variablen nach `unsigned char/int` oder `signed char/int` gecastet werden.

Die default-Einstellung für den Typ `char` ist maschinenabhängig.

## 2.11 Sonstiges

(53) Falls bei ganzzahligen Divisionen Dividend und Divisor unterschiedliche Vorzeichen haben können, sind statt dem Divisionsoperator `/` die Standardfunktionen `div()` oder `ldiv()` zu verwenden.

Ob das gebrochene negative Divisionsergebnis auf- oder abgerundet wird, ist implementation-sabhängig.

(54) Zwei gebrochene Zahlen oder eine gebrochene und eine ganze Zahl dürfen nur über eine Toleranz (`eps`) auf Gleichheit oder Ungleichheit überprüft werden.

(55) Über ein und denselben Filepointer darf nicht gleichzeitig sowohl sequentiell als auch wahlfrei auf ein File zugegriffen werden.

Die Verfahren sind nicht voll konsistent. Laut ANSI-Standard ändert z.B. der Aufruf von `ungetc()` nicht den Filepointer.

### 3 Programmierrichtlinien für C++

Ergänzend zu den C++ Richtlinien sind die C Richtlinien anzuwenden, insofern sie nicht durch eine der folgenden C++ Regeln ersetzt werden, wie z.B. die Verwendung des Sprachstandards (56).

#### 3.1 Allgemeines

(56) Es ist nach dem ISO C++-Standard zu programmieren.

Der Standard ISO/IEC 14882:1998 kann über das ANSI bezogen werden.

(57) Jede Klasse muss am Ort ihrer Deklaration mit einem Kommentar eingeleitet werden, der Name und Zweck der Klasse, Autor, Erstellungsdatum oder Version sowie eine Änderungsliste enthält.

Falls ein File nur Klassendeklarationen und -definitionen enthält, die ausführlich kommentiert sind, braucht das File selbst nicht kommentiert zu werden.

(58) Jede Definition einer Elementfunktion ist mit einem Kommentar einzuleiten. Dieser sollte Name und Zweck der Funktion, durch die Funktion verursachte Änderungen des Objektstatus, Bedeutung der Parameter und des Rückkehrwertes, evtl. Definitionsbereich der Eingabeparameter, Seiteneffekte oder sonstige Besonderheiten enthalten.

Der Definitionsbereich der Eingabeparameter ist nur dann anzugeben, wenn er gegenüber dem Datentyp eingeschränkt ist.

Beispiel:

```

/*****
* int SourceFile::SetCursorToLine (long NewLine)
*
* Setzt den Cursor in dem mit dem Objekt SourceFile
* verknuepften Dokument auf die Zeile mit der Nummer
* NewLine.
*
* Status:
*   Aendert SourceFile::CursorCurrentLine auf
*   NewLine, falls NewLine gueltig ist, bzw.
*   auf die Nummer der letzten Zeile des Dokuments,
*   falls NewLine zu gross ist.
* Parameter:
*   NewLine muss groesser 0 sein.
* Return value:
*   1   Success;
*   0   Failure (NewLine ungueltig oder Dokument hat
*         weniger als NewLine Zeilen)
* Globale Variables:
*****/

```

Access- und Mutator-Funktionen brauchen nicht kommentiert werden.

## 3.2 Layout

- (59) Die Zugriffs-Spezifizierer `public`, `protected` und `private` sind explizit in jeder Klassendeklaration anzugeben.
- (60) Klassendeklarationen dürfen keine Funktionsdefinitionen enthalten.

Inline-Funktionen werden mit Hilfe des Spezifizierers `inline` außerhalb der Klassendeklaration definiert.

### 3.3 Namenskonventionen

(61) Die C++-Keywords und die Keywords, die als Alternativen für bestimmte Operatoren dienen, sind als Symbolnamen (auch in Großbuchstaben) nicht zulässig.

Die sogenannten 'alternative keywords' sind die folgenden: `and`, `and_eq`, `or`, `or_eq`, `xor`, `xor_eq`, `not`, `not_eq`, `bitand`, `bitor`, `compl`.

(62) Access-Funktionen sollten mit `Get` oder `get`, Mutator-Funktionen mit `Set` oder `set` beginnen.

(63) Funktionen, die ein Objekt dynamisch erzeugen, das die aufrufende Funktion löschen muss, müssen mit `Create` oder `create` beginnen. Funktionen, die ein übernommenes dynamisches Objekt evtl. löschen, müssen mit `Adopt` oder `adopt` beginnen.

### 3.4 Deklarationen

(64) Innerhalb einer Klassendeklaration ist keine andere Klassendeklaration erlaubt.

Diese Regel wird wegen der schlechten Lesbarkeit geschachtelter Deklarationen aufgestellt, auch wenn sie dem Prinzip der Datenkapselung widerspricht.

(65) Als `public` deklarierte Datenelemente sind nicht erlaubt.

(66) Keine Klasse darf `friend` einer anderen Klasse sein.

Nur einzelne Elementfunktionen einer Klasse dürfen `friend` einer anderen Klasse sein.

(67) Falls eine Elementfunktion oder eine `friend`-Funktion eine Referenz auf ein Datenelement zurückliefert, muss es eine `const`-Referenz sein. Falls eine Elementfunktion oder eine `friend`-Funktion einen Pointer auf ein Datenelement zurückliefert, muss es ein Pointer auf `const` sein.

(68) Variablen oder Konstanten, die allen Instanzen einer Klasse gemeinsam sind, sind nicht global, sondern als statische Datenelemente zu definieren.

Das betrifft z.B. Instanzenzähler. Ein statisches Element hat für alle Instanzen einer Klasse denselben Wert.

(69) Überladene Operatoren und Funktionen sollten konservativ definiert werden.

Die Bedeutung, die einem Operator wie `+` oder einer Funktion wie `abs` intuitiv beigelegt wird, sollte sinngemäß erhalten bleiben. Dazu gehört auch, dass der Ausdruck `(a + b)` ein neues Objekt darstellt, das sowohl von `a` als auch von `b` verschieden ist. Der Operator `+` sollte also ein neues Objekt zurückliefern, statt das Objekt `this` oder den/die Parameter zu ändern.

(70) Bei paarweise existierenden Operatoren wie `==` und `!=` muss, falls einer überladen wird, auch der andere überladen werden.

Auch die Operatoren `->` und `[]` gehören in gewisser Weise zusammen, da die Ausdrücke `foo->mem` und `foo[0].mem` dieselbe Bedeutung haben.

(71) Wird ein binärer arithmetischer Operator `op` überladen, muss auch der Operator `op=` definiert werden.

(72) Der Dereferenzoperator `*`, der Komma-Operator und die logischen Operatoren `&&` und `||` sollten nicht überladen werden.

Komma-Operator, `&&` und `||` definieren sogenannte 'sequence points'. Sie begrenzen Teilausdrücke, die vollständig evaluiert werden, bevor mit der Berechnung des Rests fortgefahren wird. Der logische Und-Operator garantiert zum Beispiel, dass der linke Operand vollständig berechnet ist, bevor mit der Berechnung des rechten Operanden begonnen wird. Die sequence points werden beim Überladen zerstört, so dass die gewohnte, intuitive Verwendung des überladenen Operators zu Fehlern führen kann.

(73) Beim Überschreiben virtueller Funktionen dürfen der Definitionsbereich und der Wertebereich nicht geändert werden.

Anderenfalls müsste das aufrufende Objekt eine dynamische Typprüfung durchführen.

(74) Beim Überschreiben virtueller Funktionen dürfen die Default-Werte der Parameter nicht geändert werden.

Die Default-Werte orientieren sich immer am statischen Typ.

(75) Variablen mit dem Spezifizierer `static` dürfen nicht in `inline`-Funktionen definiert werden.

### 3.5 Konstruktoren und Destruktoren

(76) Klassen, die dynamisch allokierten Speicher verwenden, müssen selbst einen Kopierkonstruktor und einen Zuweisungsoperator definieren.

Der Zuweisungsoperator wird generell nicht vererbt.

Wird der Kopierkonstruktor durch Aufruf des Zuweisungsoperators definiert, dann ist sichergestellt, dass das Gleichheitszeichen in der Definition `<type> y = x;` (Kopierkonstruktor) genauso arbeitet wie in der Zuweisung `y = x;` (Zuweisungsoperator).

(77) Zuweisungsoperatoren müssen eine `const`-Referenz auf das Objekt `this` zurückliefern.

Damit wird die Semantik der Zuweisung erhalten. Die Zuweisung als Ganzes hat einen Wert (d.h. der Ausdruck:

`x = y = z;` ist korrekt), kann aber nicht als lvalue verwendet werden (d.h. der Ausdruck: `(x = y) = z;` ist nicht korrekt).

(78) Die Argumente von Zuweisungsoperator und Copy-Konstruktor müssen als `const`-Referenzen deklariert sein.

(79) Basisklassen müssen einen virtuellen Destruktor haben.

Falls der Destruktor die einzige virtuelle Funktion ist und auf den damit verbundenen Overhead verzichtet werden soll, kann diese Regel vernachlässigt werden, wenn

- niemals im Programm Objekte zwischen den Typen `(BaseClass *)` oder `(BaseClass &)` und `(DerivedClass *)` oder `(DerivedClass &)` konvertiert werden, oder
- der Destruktor von `DerivedClass` leer ist.

## 3.6 Initialisierungen

(80) Initialisierer müssen in der Initialisiererliste eines Konstruktors in der Reihenfolge stehen, in der sie vom Compiler berechnet werden.

Erst werden virtuelle Basisklassen in der Reihenfolge ihrer Deklarationen initialisiert, danach nichtvirtuelle Basisklassen in der Reihenfolge ihrer Deklarationen und zum Schluss die einfachen Datenelemente.

(81) Konstruktoren und Initialisierer von globalen Objekten dürfen keine Symbole aus anderen Modulen enthalten.

Zwar werden alle globalen Symbole vor Beginn der Funktion `main()` angelegt und initialisiert, die Initialisierungsreihenfolge ist jedoch nur für Symbole festgelegt, die ein und demselben Modul angehören.

## 3.7 Defensivprogrammierung, Fehlerbehandlung

(82) Jeder Rückkehrwert von `new` muss vor der ersten Verwendung geprüft werden.

(83) Jeder Zuweisungsoperator muss einen Test enthalten, der verhindert, dass ein Objekt sich selbst zugewiesen wird.

## 3.8 Typumwandlungen

(84) Pointer auf Funktionen dürfen nicht in Pointer auf Objekte umgewandelt werden und umgekehrt.

(85) Pointer auf Funktionen dürfen nicht in Pointer auf Elementfunktionen umgewandelt werden und umgekehrt.

### 3.9 Sonstiges

(86) Statt der Funktionen `alloc()` und `free()` sind die Operatoren `new` und `delete` zu verwenden.

(87) Die Ein-/Ausgabefunktionen aus dem Headerfile `stdio.h` dürfen nicht mit der Ein-/Ausgabe über Streams gemischt werden.

(88) Für jedes Datenelement, dem mit `new` im Konstruktor (oder an einer anderen Stelle) Speicher zugewiesen wird, muss ein `delete` im Destruktor (oder an einer anderen Stelle) implementiert sein.

(89) Der Zuweisungsoperator und die Konstruktoren müssen alle nichtstatischen Datenelemente zuweisen.

(90) Die Verwendung des Bezeichners `this` ist zu minimieren.

(91) Die Funktionen `memcpy()`, `memmove()` u.ä. dürfen nicht auf Objekte angewendet werden.

## 4 Namenskonventionen

### 4.1 Konstanten

Konstanten und Aufzählungen sollen groß geschrieben werden, einzelne Wortteile getrennt durch den Unterstrich.

Adjektive wie MAX und MIN werden hinten an den Namen angehängt. Dabei bezeichnet MAX oft die Anzahl der Elemente in einem Array, also eins mehr als den größten zulässigen Index:

```
const MODULES_MAX = 64;
const MODULE_NAME_MAX = 32;
MODULE_DATA g_Modules[MODULES_MAX];
for (int i = 0; i < MODULES_MAX; i++)
    ...
```

### 4.2 Benutzerdefinierte Typen

Benutzerdefinierte Typen (`typedef`, `struct`, `union`, `enum`) werden, genau wie benannte Konstanten, in Großbuchstaben mit Unterstrich geschrieben. Typen die einen Pointer darstellen, enthalten als ersten Buchstaben ein großes "P".

Ein Zeiger auf eine Funktion (Callback - Zeiger) beginnt mit "PF":

```
typedef const char* LPCSTR;
typedef char* LPSTR;
typedef void (WINAPI *PFPRINT) (LPCSTR pszText);
enum MODULE_TYPE {
    MDT_UNKNOWN,
```

```

    MDT_SIMPLE,
    MDT_SMART
};
struct MODULE_DATA {
    char szName [MODULE_NAME_MAX];
    MODULE_TYPE Type;
};

```

### 4.3 Klassen

Klassen (`class`) werden in Groß- Kleinschreibung ohne Unterstrich geschrieben. Dem Namen wird außerdem ein großes "C" vorangestellt.

Die Namen von abstrakten Basisklassen enden mit einem großen A. Klassen gelten hier als abstrakt, wenn sie im wesentlichen eine Schnittstelle definieren und es entweder nicht möglich oder nicht sinnvoll ist Instanzen von ihnen zu erzeugen.

```

class CTrashObj : public CNodeA {
    CTrashObj ();
    virtual SetData (LPVOID pData);
    ...
};
class IClassFactory : public IUnknown {
    HRESULT QueryInterface (REFIID riid, void * * ppvObj) ;
    unsigned long AddRef ();
    unsigned long Release ();
    ...
};

```

## 4.4 Funktionen

Funktionsnamen beginnen mit einem Großbuchstaben, die einzelnen Wortteile werden ebenfalls durch Großbuchstaben getrennt. Ein Funktionsname ist immer nach dem Schema Operation-Objekt[-Attribut] aufgebaut. Beispiele sind: SetThreadPriority (), GetCurrentWindow (), GetFileSecurity (), CreateProcess (), EnterCriticalSection () etc.

Selektoren (Funktionen, die ein Attribut eines Objektes ermitteln) beginnen immer mit "Get", Modifizierer (Funktionen, die ein Attribut eines Objektes verändern) immer mit "Set".

## 4.5 Variablen

Variablen werden durch die folgende (ungarische) Notation dargestellt.

Namen bestehen aus drei Teilen:

- Basistyp,
- Präfix und
- Bezeichner

Mit dem Aufbau: „Präfix Basistyp Bezeichner“.

### 4.5.1 Basistypen

Basistypen bezeichnen den **Datentyp** der Variable. Dabei werden sowohl die Standardtypen als auch abstrakte Datentypen durch Kürzel repräsentiert. Einige übliche Kürzel für die Standardtypen:

Basistyp	Bedeutung
n	allgemeiner numerischer Wert (int, oft auch UINT oder LONG).
u	numerischer Wert ohne Vorzeichen (UINT). Diese werden in der Praxis jedoch oft auch mit "n" bezeichnet.
l	LONG, wird oft auch mit "n" bezeichnet.
b	BOOL
f	Float
ch	einzelnes Zeichen (char).
by	einzelnes Byte (BYTE).
sz	0-terminierter String als Charakter-Array (char []).
s	String bei Verwendung einer Stringklasse (string).

Festzustellen ist, dass die Abkürzung der Basistypen nicht immer eindeutig ist. Das liegt bei den Standardtypen daran, daß sie äußerst wenig über den Verwendungszweck der Variable aussagen. So ist es meistens nur von Interesse, daß eine Variable einen numerischen Typ hat ("n") und es ist unwesentlich, ob dieser nun ein int, LONG, UINT oder DWORD ist.

## 4.5.2 Präfixe

Präfixe gehen über den Basistyp hinaus und beschreiben den **Zweck** einer Variablen. Im Gegensatz zu den Basistypen sind Präfixe eindeutig.

Präfix	Bedeutung
a	Array.
c	Zähler (Counter), z.B. Anzahl der Elemente eines Arrays.
d	Differenz zwischen zwei Variablen desselben Typs.
e	Element eines Arrays.
h	Handle (Kennziffer, Objekte des Betriebssystems werden häufig durch Handles repräsentiert).
i	Index eines Arrayelementes.
p (lp, np)	Zeiger (Pointer). (Oft findet man auch noch die 16-Bit Relikte "lp" und "np" für Long Pointer und Near Pointer).
r (C++)	Referenz, wird in Parameterlisten für Variablen verwendet, die zurückgegeben werden sollen, also NICHT für konstante Referenzen.
g_	globale Variable.
m_ (C++)	Membervariable.
tl_	threadlokale Variable (Thread Local Storage). Threadlokale Variablen sind "global" innerhalb eines Threads.
pf	Funktion, die als Parameter übergeben wird (Callback-Funktion). Wird oft auch mit "pfn" bezeichnet.

**Präfixe** erscheinen im Namen **vor** dem **Basistyp**. Sie können beliebig mit weiteren Präfixen oder Basistypen kombiniert werden. Präfix und Basistyp sind zusammen der **kleingeschriebene** Teil, mit dem jeder Variablenname beginnt.

Beispiele:

Variablenanfang	Bedeutung
psz	Zeiger auf ein Charakter-Array (char []), das einen 0-terminierten String representiert.
ach	allgemeines Charakter-Array (char []), z.B. für einen Puffer
cch	Anzahl der Zeichen in einem Charakter-Array
ich	Index des gerade bearbeiteten Zeichens
ppsz	Zeiger auf einen Zeiger auf einen String
apsz	Array von Zeigern auf Strings
g_sz	Globaler String
hwnd	Handle eines Fensters

Die ungarische Notation hebt ausdrücklich den semantischen Unterschied zwischen Arrays und Zeigern hervor. Diese werden in C ja bekannterweise syntaktisch gleich behandelt, was häufig zu Verwirrungen führt.

### 4.5.3 Bezeichner

Der dritte Anteil eines ungarischen Namens ist der Bezeichner. Außerhalb der ungarischen Notation bestehen Variablennamen meistens nur aus dem Bezeichner. Der Bezeichner ist der eigentliche Name, aus dem der Sinn und Zweck der Variablen im Kontext des Codes hervorgehen sollte. (Stichwort "**Sinnvolle Variablennamen**")

Es ist üblich, Bezeichner für Variablen (und Funktionen) mit einem Großbuchstaben zu beginnen und durch Groß-/Kleinschreibung zu unterteilen.

Zusätzlich zu den selbst vergebenen Bezeichnern kennt die ungarische Notation einige Standardbezeichner:

Bezeichner	Bedeutung
Min	Absolutes, erstes Element eines Arrays oder einer Liste. Untere Grenze eines Wertebereiches.
Max	Absolutes, letztes Element eines Arrays. Obere Grenze eines Wertebereiches.
First	Das erste relevante Element eines Arrays bezogen auf eine bestimmte Operation.
Last	Letztes relevantes Element eines Arrays. Gegenstück zu "First".

#### 4.5.4 Beispiele für die ungarische Notation

Die folgenden Quelltextbeispiele sollen die ungarische Notation nochmal verdeutlichen:

```
BOOL GetCommandLine (char* pszDest, int cchDest);  
BOOL SplitCommandLine (const char* pszCommandLine,  
const char* rapszArgs[], int cpszArgs);  
BOOL GetScreenContext (SCREEN_CONTEXT& rscDest);  
BOOL SetScreenContext (const SCREEN_CONTEXT& scNew);  
BOOL EnumWindows (PFENUMWND pfEnumWndCB);
```

Bezeichner	Bedeutung
pszDest	Zeiger auf einen Puffer, der mit dem Ergebnis gefüllt wird.
cchDest	Größe des Puffers in Zeichen.
pszCommandLine	Zeiger auf den String, der von GetCommandLine () ermittelt wurde.
rapszArgs[]	(Referenz) eines Arrays von Zeigern auf konstante Strings, in denen die Adressen der Strings der einzelnen Argumente zurückgegeben werden. (OK, dieses Beispiel ist wirklich heavy.)
cpszArgs	Größe des Arrays.
rscDest	Referenz auf einen SCREEN_CONTEXT, der das Ergebnis erhalten soll.
scNew	SCREEN_CONTEXT, der gesetzt werden soll. Dieser ist kein Rückgabewert und wird nur aus Speicherplatzgründen als Referenz übergeben, was an dem "const" leicht zu erkennen ist.
pfEnumWndCB	Callbackfunktion für die Auflistung von Fenstern
g_achArgs	globaler Puffer zum Speichern von Argumenten
g_hwndMain	globales Handle des Hauptfensters
m_btnStart	Membervariable für den Startknopf in einem Dialog.
chAct	aktuelles Zeichen
nInstances	Anzahl der Instanzen
bSuccess	Ergebniswert einer boolschen Funktion
ichFirst	Index des ersten zu bearbeiten Zeichen eines Arrays oder Strings.