# Programmierrichtlinien der TBS1, Bochum

Autor: Ralf Adams

Version 1.0 vom 6. September 2005

**Zusammenfassung**

In diesem Aufsatz werden die Richlinien und Standards für die C/C++-Programmierung an der TBS1 in Bochum festgelegt. Ziel ist es, die Qualität der von den Schülern / Studenten erstellten Quelltexte zu verbessern und schulform- und jahrgangsstufensübergreifend die gleichen Konventionen anzuwenden.

Für Anmerkungen und Änderungswünsche wenden Sie sich bitte an *adams@tbs1.de*.

**Zielgruppe:** Dieses Dokument ist in erster Linie für Lehrer und Lehrerinnen erstellt worden. Es obliegt ihrer Verantwortung die hier vorgestellten Inhalte im Unterricht zu vermitteln. Dies bedeutet nicht, dass dieses Dokument nicht den Schülern gegeben werden darf.

## Inhaltsverzeichnis

# 1 Einführung

Hier geht es um Standards und Richtlinien bei der Programmierung. Ziel ist dabei nicht das Erzeugen eines bürokratischen Mehraufwandes. Die hier vorgestellten Regeln dienen erwiesenermaßen der Reduktion von Fehler, der Verbesserung von der Wartbarkeit und Lesbarkeit von Source Code.

Leider ist es unter Schülern oder anderen *freiheitlich* Gesinnten uncool Programmierrichtlinien einzuhalten. Genau dies macht aber den Unterschied zwischen einem Angeber und einem Profi aus. Welchen Zweck ausser narzistischer Selbstgefälligkeit hat es, möglichst komplizierten trickreichen Source Code zu schreiben? Keinen. Selbst Argumente, wie *aber dafür ist C doch gebaut*, oder *das Programm ist dann schneller* sind genauer betrachtet nur Scheinbehauptungen:

- Es ist gerade der Fluch und nicht der Segen von C alle mögliche Schmutzkonstrukte zuzulassen. Mehrfachverschachtelte Anweisungen unter Ausnutzung aller Operandenprioritäten unter Auslassung von Klammern sind, wie dieser Satz, ein Qual und darüberhinaus für einen Auftraggeber nicht hinnehmbar.
- Scheinbare Tricks - wie der Shift für die Multiplikation mit 2 - sind lächerlich im Verhältnis zu den in den Compilern eingebauten sehr ausgefeilten Optimierungsverfahren. Viele Programmierer wissen gar nicht, was gute Compiler aus ihren schlechten Programmen noch raus holen. Man sollte sich auf diese Verfahren verlassen und nicht versuchen besser zu sein. Im Allgemeinen programmiert man nur einen Fehler und keine Optimierung.

**Definition 1** Standard

*Unter einem* Standard *sei eine Regel verstanden, die immer eingehalten werden soll. Die meisten hier vorgestellten Standards sind keine TBS1-Bochum Standards, sondern werden weltweit von Programmieren akzeptiert.*

**Definition 2** Richtlinie

*Eine* Richtlinie *kann bei sinvollen Randbedingungen ignoriert werden. Man sollte aber immer genau überlegen, ob dies wirklich gerechtfertigt ist.*

## 2 Source Files

Unter einem Source File wird eine Datei mit Präprozessor- und/oder C-Anweisungen verstanden.

***Richtlinie 1:***

Die Größe eines Source Files sollte 500-1000 Zeilen nicht überschreiten.

***Richtlinie 2:***

Eine Zeile sollte aus maximal 80 Zeichen bestehen.

***Standard 1:***

Jedes Source File enthält Kommentare der Art: Projekt, Autor, Version, Datum, Name der Datei etc.

***Standard 2:***

Bei Hausaufgaben, Projektarbeiten oder ähnliches enthält jedes Source File zusätzlich die Kommentare: Name der Gruppenmitglieder, Klasse, Aufgabe oder Teilaufgabe und Lehrer.

***Standard 3:***

Nach den Kommentaren folgen `#include`, `#define`, `extern`-Deklarationen und anschließend globale Variablen.

***Richtlinie 3:***

Globale Variablen werden vermieden.

***Richtlinie 4:***

Alle Stringkonstanten werden in einem Source File gesammelt.
Grund: Die Anpassung an andere Sprachen wird dadurch erleichtert.

***Standard 4:***

Innerhalb eines Projektes gibt es ein besonderes Header File namens `local.h`, welches alle projekteinheitlichen `#include` und `#define` enthält.
Beispiel:

```
1  #ifndef LOCAL_H
2  #define LOCAL_H
3  #include <stdio.h>
4  #include <stdlib.h>
5  #define FOREVER for(;;)
6  #define YES (1 == 1)
7  #define NO (1 == 0)
8  #define TRUE YES
9  #define FALSE NO
10 #endif
```

***Standard 5:***

Konstanten mittels `#define` und Struktur-Deklarationen werden in ein eigenes Header File geschrieben, wenn sie von mehr als einem Source File verwendet werden.

***Standard 6:***

Der Inhalt von Header Files wird durch `#ifndef` und `#endif` vor Mehrfacheinbindung geschützt.
Beispiel für das Header File `wurst.h`:

```
1 #ifndef WURST_H
2 #define WURST_H
3 ...
4 #endif
```

***Standard 7:***

Lokale oder selbsterstellte Header Files werden mit Gänsefüßchen eingebunden. Standard Header Files oder Header Files der Entwicklungsumgebung mit eckigen Klammern.

***Standard 8:***

Lokale Header Files werden mit ihrem relativen und nicht ihrem absoluten Pfad angegeben.
Grund: Wird das Projekt in ein anderes Verzeichnis kopiert, stimmen die Pfade nicht mehr.

***Standard 9:***

Keine Funktionsdefinition oder Variablenzuweisungen in Header Files.
Grund: Es können Funktionen mehrfach definiert werden.

***Richtlinie 5:***

Der Name eines Source Files mit der Funktion `main()` ist der Name des Programms (oder besser der ausführbaren Datei) gefolgt von der Endung `.c` bzw. `.cpp`. Das Header Files zu diesem Source File hat den gleichen Namen mit der Endung `.h`.

# 3 Kommentare

***Standard 10:***

Kommentare stehen vor den Funktionen.

***Standard 11:***

Der Inhalt der Kommentare muss mit dem Code übereinstimmen!

***Richtlinie 6:***

Kommentare sollten neben dem *Was* auch das *Warum* beschreiben.

***Richtlinie 7:***

Kommentare sollten den Leser auf ein Problem des Programms hinweisen. Dies können bekannte Fehler oder Missverständnisse sein.

***Richtlinie 8:***

Kommentare sollten den Source Code nicht wiederholen.
Schlecht:

```
1     a++;    /* Erhöht den Inhalt der Variable a um 1 */
```

Grund: Dies versperrt die Sicht auf die wichtigen Hinweise.

***Richtlinie 9:***

Nicht überkommentieren: Zuviel Kommentar ist genauso schädlich, wie zuwenig.

***Standard 12:***

Das Ende von Anweisungsblöcken wird kommentiert.
Beispiel:

```
1     for (i = 0; i < 10; i++)
2     {
3       ...
4       if (i > 5)
5       {
6         ...
7       } // if Ende
8     } // for Ende
```

# 4 Formatierungen

***Standard 13:***

Die Operatoren `->`, `.` und `[]` werden ohne Leerzeichen umgeben.

```
1    p->m
2    s.m
3    a[i]
```

***Standard 14:***

Der öffnenden Klammer einer Funktion folgt kein Leerzeichen. Bei Ausdrücken innerhalb der Klammern steht kein Leerzeichen nach der öffnenden Klammer und kein Leerzeichen vor der schließenden Klammer.
Gut: `pow(2, x+2)`
Schlecht: `pow( 2, x+2 )`

***Standard 15:***

Nach den unären Operatoren erfolgt kein Leerzeichen.

```
1    !p
2    ~b
3    ++i
4    --j
5    (long)m
6    *p
7    &x
```

***Standard 16:***

Vor und nach dem Zuweisungsoperator und vor und nach dem bedingten Operator steht ein Leerzeichen.

```
1    c1 = c2;
2    i += j;
3    n > 0 ? n : -n;
```

***Standard 17:***

Nach Kommata und Semikoli steht ein Leerzeichen oder ein Zeilenumbruch.

```
1    strncat(a, b, c);
2    for (i = 0; i < 10; i++)
```

***Richtlinie 10:***

Vor und nach anderen Operatoren steht ein Leerzeichen.

```
1     x + y
2     (a < b) && (b < c)
3     m + 1
```

Ausnahme: Zur Verdeutlichung einer stärkeren Bindung oder der besseren Lesbarkeit wegen, kann das Leerzeichen weggelassen werden.

```
1     x < y ? a : b-10;
2     printf(\symbol{34}\%i \%i \%i\symbol{34}, a+1, b+1, c+1);
```

***Standard 18:***

Nach den Schlüsselwörten (`if`, `while`, `do`, `switch`, `case`, `break`, `return` usw.) steht ein Leerzeichen.

***Standard 19:***

Bei Anweisungsblöcken, einzelnen Zeilen bei Kontrollanweisungen (`if`, `while`, `for`), Funktionsdefinitionen, Strukturdefinitionen, Klassendeklarationen und Uniondefinitionen werden die Zeilen eingerückt. In den meisten Fällen heißt das: *Wird eine geschweifte Klammer aufgemacht, werden alle nachfolgenden Zeilen bis zur geschlossenen geschweiften Klammer eingerückt.*

```
1     if (i < 10)
2     {
3       a = 10;
4     } // if Ende
5     else
6     {
7       a = 20;
8       switch (a)
9       {
10        case 10:
11          y = sin(x);
12          break;
13        case 20:
14          y = cos(x);
15          break;
16      } // switch Ende
17    } // else Ende
```

***Standard 20:***

Einrückungen bestehen aus 2, 3 oder 4 Leerzeichen oder einem Tabulatur-Sprung.

***Standard 21:***

Deklarationen werden durch Leerzeilen von Anweisungen getrennt.

```
1     int a = 0, b = 0;
2
3     a = b * 2;
```

***Standard 22:***

Erstreckt sich eine Einweisung über mehrere Zeilen, werden die Folgezeilen einfach oder mehrfach eingerückt.

```
1     if ((iNennwertBrutto > 9) &&
2         (iFamilienstand == 1) ||
3         (fMehrwertsteuerSatz > 15.0)
4        ) // if Ende
```

# 5 Datentypen

***Richtlinie 11:***

Es sollen nur Standarddatentypen verwendet werden.
Grund: In den meistenen Fällen müssen die selbstdefinierten Datentypen mühsam entschlüsselt werden. Leider ist es aber in IDEs üblich einen ganzen Sack voller eigener Datentypen anzubieten und zu verwenden.

***Richtlinie 12:***

Der Datentyp `int` sollte vermieden werden. Stattdessen solltem die Datentypen `short` oder `long` verwendet werden.
Grund: Je nach Compiler oder Architektur ist ein `int` gleich einem `short` oder einem `long`. Eine Vermeidung sorgt hier gleich für Klarheit.

***Standard 23:***

Über Datengrößen können folgende Annahmen gemacht werden.

| Datentyp | Länge (Standard) | Länge (Ausnahmen) |
|---|---|---|
| `char` | 8 Bits | 9, 10 Bits |
| `short` | 16 Bits | 18, 20 Bits |
| `long` | 32 Bits | 36, 40 Bits |

***Richtlinie 13:***

Neue Datentypen sollten nur eingeführt werden, wenn es die Hardware oder anderer Gegebenheiten erfordern oder wenn es die Lesbarkeit erhöht.

***Richtlinie 14:***

Folgende Datentypen werden in der `<sys/types.h>` deklariert und können verwendet werden:

```
1    typedef unsigned char  u_char;
2    typedef unsigned short u_short;
3    typedef unsigned int   u_int;
4    typedef unsigned long  u_long;
5    typedef unsigned short ushort;
6    typedef unsigned int   uint;
```

Dieses Header File ist nicht auf allen Systemen installiert!

***Standard 24:***

Ein Ausdruck vom Typ `bool` kann nur die Werte `true` oder `false` annehmen.

***Standard 25:***

Eine Variable vom Typ `bool` kann nur mit den Werte `true` oder `false` verglichen werden.

***Standard 26:***

Strukturen und Unios werden extra deklariert und von der Variablendefinition getrennt. Die Deklaration erfolgt in einer Header-Datei, selbst wenn sie nur in einem Source File verwendet wird.
Anmerkung: Verzichten Sie auf `typedef struct` auch wenn es bequemer erscheint. Die Information, dass die Variable ein Struktur ist, geht für den Programmierer in der Anschauung verloren.

# 6 Variablen und Konstanten

***Richtlinie 15:***

Namen für Konstanten und Makros stehen in Großbuchstaben. Variablennamen beginnen mit Kleinbuchstaben.

***Richtlinie 16:***

Namen sollten selbsterklärend sein.
Schlecht:

```
1    double a = 1.0;
2    double b = 7.12;
3    double c = 0.0;
4
5    c = b / a;
```

Gut:

```
1    double dSpannung     = 1.0;
2    double dStromstaerke = 7.12;
3    double dWiderstand   = 0.0;
4
5    dWiderstand = dSpannung / dStromstaerke;
```

***Standard 27:***

Bei der Programmierung technischer oder physikalischer Formeln können anstelle sprechender Variablennamen auch die gängigen Namen der Funktionssymbole verwendet werden.

```
1    double dF     = 1.0;
2    double dAlpha = 2.0;
3    double dQ     = 3.0;
```

***Richtlinie 17:***

Abkürzungen bei der Namensgebung sollten nach einem einheitlichen Schema erfolgen. Es ist darauf zu achten, dass die Abkürzung nicht zu Missverständnissen führt (*in*put *ch*aracter -> *inch*).

***Standard 28:***

Allgemeine Variablen- und Funktionsnamen dürfen nicht mit einem Unterstrich beginnen.
Grund: Systemvariablen und -funktionen und Assembleranweisungen beginnen mit einem Unterstrich.

***Richtlinie 18:***

Innerhalb von inneren Anweisungsblöcken sollen keine Variablen deklariert werden.

```
1     if (i < 10)
2     {
3       double dMilchmannGehalt = 0.0;     // Besser nicht !
4       ...
5     } // if Ende
```

***Standard 29:***

Hilfsvariablen können folgende Namen haben:

| | |
|---|---|
| Zeichen | `c`, `d` |
| Laufvariable oder Index | `i`, `j`, `k` |
| Zähler | `m`, `n` |
| Zeiger | `p`, `q` |
| String | `s`, `t` |

***Richtlinie 19:***

Namen, die Länger als 4 Buchstaben sind, sollten sich in mindestens 2 Buchstaben unterscheiden.
Schlechtes Beispiel: `int systst, syssts t;`

***Richtlinie 20:***

Variablennamen sollten in der gleichen Spalte deklariert werden. Das gleiche gilt für Kommentare.
Schlecht:

```
1     int i; /* Zaehlvariable */
2     float fSumme; /* Gesamtsumme */
3     unsigned short uAlter; /* Alter einer Person */
```

Besser:

```
1     int            i;      /* Zaehlvariable */
2     float          fSumme; /* Gesamtsumme */
3     unsigned short uAlter; /* Alter einer Person */
```

***Standard 30:***

Globale und statische Variablen müssen explizit initialisiert werden.

***Standard 31:***

Initialisierte Variablen werden in jeweils einer eigenen Zeile definiert.
Schlecht:

```
1    int i = 0, j = 1;
```

Gut:

```
1    int i = 0;
2    int j = 1;
```

Geht auch:

```
1    int i = 0,
2        j = 1;
```

***Richtlinie 21:***

Initialisierung für Strukturen, Unions und Felder werden mit einer Reihe pro Zeile initialisiert.

```
1    static short x[2][5] =
2    {
3      {1, 2, 3, 4, 5},
4      {6, 7, 8, 9, 10}
5    };
```

***Richtlinie 22:***

Konstanten, die eine lineare Ordnung repräsentieren, sollten als `enum`-Variable definiert werden und nicht als eine Folge von `#define`-Anweisungen.
Grund: Die richtige Belegung kann vom Compiler überprüft werden.
Schlecht:

```
1    #define ADRESSE_START  0x0100
2    #define ADRESSE_STEUER ADRESSE_START  + 1
3    #define ADRESSE_INPUT  ADRESSE_STEUER + 1
4    #define ADRESSE_OUTUT  ADRESSE_INPUT  + 1
```

Besser:

```
1    enum
2    {
3      adresseStart = 0x0100,
4      adresseSteuer,
5      adresseInput,
6      adressePutput
7    };
```

***Standard 32:***

Alle anderen Konstanten werden durch `#define` definiert. Voneinander abhängige Konstanten werden mit den schon vorhandenen Konstantennamen definiert.
Schlecht:

```
1    #define ARBEITSTAGE     5
2    #define STUNDEN_TAG     8
3    #define STUNDEN_WOCHE  40
```

Gut:

```
1    #define ARBEITSTAGE     5
2    #define STUNDEN_TAG     8
3    #define STUNDEN_WOCHE   (ARBEITSTAGE * STUNDEN_TAG)
```

***Standard 33:***

Nach Konstanten vom Typ `long` steht ein `L` in Großbuchstaben.
Schlecht:

```
1    long int i = 10000000000l;
```

Gut:

```
1    long int i = 10000000000L;
```

Grund: Das kleine `l` wird zu schnell mit einer 1 verwechselt.

***Richtlinie 23:***

Falls möglich sollte bei Bitkonstanten der Komplementoperator verwendet werden.
Schlecht:

```
1    #define MASK_FIRST_BIT  0xFFFFE
```

Gut:

```
1    #define MASK_FIRST_BIT  ~1
```

Grund: Die erste Variante arbeitet nur bei 16-Bit Zahlen korrekt. Die zweite Variante ist unabhängig von der Wortlänge, arbeitet also auch bei 32 Bit korrekt.

***Richtlinie 24:***

Alle Text- und Datenkonstanten sollten mit dem Schlüsselwort `const` deklariert werden.
Grund: So kann schon der Compiler erkennen, wenn man fälschlicherweise versucht eine Konstante zu verändern.

# 7 Zeiger

***Standard 34:***

Zeiger-Variablen müssen explizit als Zeiger deklariert werden. Sie dürfen nicht als `int` deklariert werden.
Grund: Obwohl dies ist einigen Büchern noch steht (z.B. im K&R) sind weder die Wortlänge von `int` noch die Größe eines Zeigers normiert.

***Richtlinie 25:***

Keine Typkonvertierung von Zeigern ausser bei Zuweisung des `NULL`-Zeigern oder bei der Speicherreservierung.

# 8 Funktionen

***Richtlinie 26:***

Eine Funktion sollte 150 Zeilen nicht überschreiten.

***Richtlinie 27:***

Vor der Funktionsdefinition sollte ein Kommentar stehen, der die Funktion beschreibt: Bedeutung der Aufruf-Parameter, Zweck der Funktion und mögliche Rückgaben.

***Richtlinie 28:***

Die Funktionsbeschreibung sollte nicht beschreiben, *wie* die Aufgabe gelöst wurde, sondern welche Aufgabe gelöst wurde. Das *Wie* sollte dem Source Code entnommen werden können. Der Kommentar sollte keine Liste der verwendeten Funktionen enthalten.
Grund: Diese Liste wird i.d.R. nicht gepflegt und neigt dazu nicht mehr zum Source code zu passen. Auffrufhierarchien können von Tools erstellt werden.

***Standard 35:***

Funktionen werden vor ihrer Verwendung durch einen Prototypen deklariert.

***Standard 36:***

Eine Funktion ohne einen Rückgabewert ist vom Rückgabetyp `void`.

***Standard 37:***

Keine Sprünge mit `goto`.

***Standard 38:***

Um Konflikte zu vermeiden, darf ein Funktionsname nicht mit dem Namen einer Bibliotheksfunktion übereinstimmen.

# 9 Makros

***Standard 39:***

Parameter von Makros müssen geklammert werden.
Schlecht:

```
1    #define ABS(x) x > 0 ? x -x
```

Gut:

```
1    #define ABS(x) (x) > 0 ? (x) -(x)
```

Grund: Das Ergbnis kann u.U. nicht den Erwartungen entsprechen. Beispiel: `ABS(3-4)`

***Standard 40:***

Zwischen Makroname und Klammer darf kein Leerzeichen stehen.
Grund: Einige Präprozessoren erlauben das nicht.

# 10 Operatoren

***Standard 41:***

Typ-Konvertierungen müssen explizit durch den *cast*-Operator erfolgen.
Schlecht:

```
1    int    i;
2    double x = 1.2;
3
4    i = x;
```

Gut:

```
1    int    i;
2    double x = 1.2;
3
4    i = (int)x;
```

***Standard 42:***

Für Zeiger Konvertierungen muss ein *cast*-Operator verwendet werden.
Schlecht:

```
1    long int    *p;
2
3    p = malloc(sizeof(long int) * 1000);
```

Gut:

```
1    long int    *p;
2
3    p = (long *)malloc(sizeof(long int) * 1000);
```

***Richtlinie 29:***

Typkonvertierungen sollte auf ein Minimum reduziert werden.

***Standard 43:***

Beim bedingten Operator werden die ?-Ausdrücke nicht geschachtelt.

***Standard 44:***

Beim bedingten Operator stehen vor und hinter dem `:` die gleichen Datentypen.

***Standard 45:***

Operanden für die logischen Operatoren `!`, `&&` und `||` dürfen nur vom Typ `bool` sein.

***Richtlinie 30:***

Um Seiteneffekte zu vermeiden, darf der zweite Operand bei Und- und Oder-Verknüpfungen keine Zuweisung o.ä. enthalten.
Grund: Sprunglogik!

***Standard 46:***

Alle bitweisen Operationen (`&`, `|`, `~`, `^`, « und») werden geklammert.
Grund: Folgender Fehler soll vermieden werden:
Schlecht:

```
1     if(status & BITMASK != SET)
```

Gut:

```
1     if((status & BITMASK) != SET)
```

***Standard 47:***

Der Rechts-Shift Operator » darf nur auf vorzeichenlose ganze Zahlen angewendet werden.
Grund: Die Vorzeichenexpandierung ist maschinenabhängig.

***Standard 48:***

Variablen, die inkrementiert oder dekrementiert werden, dürfen in einem Ausdruck nur einmal vorkommen.
Grund: Das Ergebnis von `a[i] = i++;` ist undefiniert.

***Standard 49:***

Bei zusammengesetzten Anweisungsoperatoren sollten Klammern verwendet werden.
Schlecht:

```
1     x *= y + 1;
```

Gut:

```
1     x *= (y + 1);
```

***Standard 50:***

Der Kommaoperator sollte nur bei einer `for`-Schleife oder einer Variablendeklaration verwendet werden.

***Richtlinie 31:***

Wenn die Abarbeitungsreihenfolge eines Ausdrucks nicht ohne weiteres erkennbar ist, sollte dies durch Klammerung verdeutlicht werden.

# 11 Kontrollfluss

***Standard 51:***

Testausdrücke in `while`-, `for`-, `do/while`-Schleifen oder `if`-Anweisungen sollten als expliziter Vergleich geschrieben werden.
Hinweis: Weit verbreitete Ausnahmen sind Vergleiche auf 0, ungleich 0, dem Nullzeichen und dem NULL-Zeiger.
Schlecht:

```
1     FILE *fp = NULL;
2
3     fp = fopen(szDateiname, szModus);
4     if (fp)
5     {
6       /* Tue was */
7     } // if Ende
```

Gut:

```
1     FILE *fp = NULL;
2
3     fp = fopen(szDateiname, szModus);
4     if (NULL == fp)
5     {
6       /* Tue was */
7     } // if Ende
```

***Standard 52:***

Um eine Verwechslung des Zuweisungsoperators `=` mit dem Vergleichsoperator `==` auszuschließen, wird bei der Verwendung des Zuweisungsoperators zur Bildung von Testausdrucken der Wert der Zuweisung explizit verglichen.
Schlecht:

```
1     while ('\0' != (*s++ = *t++));
```

Gut:

```
1     for ( ; '\0' != *t; s++, t++) {
2       *s = *t;
3     } // for Ende
```

***Standard 53:***

Bei Vergleichen mit konstanten Werten steht die Konstante immer auf der linken Seite.

```
1     1 == m;
```

***Richtlinie 32:***

Bedingungen sollten verständlich formuliert sein.

***Richtlinie 33:***

Vergleiche sollten - falls fachlich sinnvoll - nicht auf Gleichheit, sondern auf größer gleich oder kleiner gleich durchgeführt werden (defensives Programmieren).

***Richtlinie 34:***

Gleitkommazahlen sollten nicht auf Gleichheit getestet werden, sondern, ob sie in einem bestimmten Abstand vom Verrgleichswert liegen.
Schlecht:

```
1     double x = 0.0;
2     double y = 0.0;
3
4     /* Hier passiert was mit x und y */
5
6     if (x == y)
7     {
8       /* Tue was */
9     } // if Ende
```

Gut:

```
1     double x = 0.0;
2     double y = 0.0;
3
4     /* Hier passiert was mit x und y */
5
6     if (abs(x - y) < 0.5e-15)
7     {
8       /* Tue was */
9     } // if Ende
```

Besser:

```
1     #define EPS (0.5e-15)
2     [...]
3     double x = 0.0;
4     double y = 0.0;
5
6     /* Hier passiert was mit x und y */
7
8     if (abs(x - y) < EPS)
9     {
10      /* Tue was */
11    } // if Ende
```

***Standard 54:***

Die `switch`-Anweisung wird wie folgt eingerückt:

```
1     switch(Ausdruck)
2     {
3         case Konstante:
4                 Anweisungen
5                 break;
6         case Konstante:
7         case Konstante:
8                 Anweisungen
9                 break;
10        default:
11                Anweisungen
12                break;
13    } // case Ende
```

***Standard 55:***

Der letzte `case`-Zweig der `switch`-Anweisung ist der `default`-Zweig.

***Richtlinie 35:***

Ein `case`-Zweig kann leer sein.

***Richtlinie 36:***

Die letzte Anweisung innerhalb eines `case`-Zweigs ist die `break`-Anweisung.

***Standard 56:***

Falls die letzte Anweisung innerhalb eines `case`-Zweigs nicht die `break`-Anweisung ist, ist dies zu kommentieren.
Grund: Bei Wartungsarbeiten soll dokumentiert werden, warum das break fehlt, damit man nicht fälschlicherweise von einem Fehler ausgeht.

***Standard 57:***

Sowohl der `if`-Zweig als auch der `else`-Zweig sind unabhängig von der Anzahl der Anweisungen des Zweig mit geschweifte Klammern zu umgeben.
Schlecht:

```
1     if (i < 10)
2       j = 12;
3     i = 0;
```

Gut:

```
1     if (i < 10)
2     {
3       j = 12;
4     } // if Ende
5     i = 0;
```

***Richtlinie 37:***

Bei verschachtelten `if`s auf der gleichen Variablen kann folgendes Format verwendet werden:

```
1     if (ausdruck)
2     {
3       ...
4     } // if Ende
5     else if (ausdruck)
6     {
7       ...
8     } // if Ende
9     else if (ausdruck)
10    {
11      ...
12    } // if Ende
13    else
14    {
15      ...
16    } // else Ende
```

***Richtlinie 38:***

Bei jeder Schleife sind unabhängig von der Anzahl der Anweisungen innerhalb der Schleife die Anweisungen mit geschweifte Klammern zu umgeben.
Schlecht:

```
1     while (i < 10)
2       printf("\%d\n", i++);
3     i = 0;
```

Gut:

```
1     while (i < 10)
2     {
3       printf("\%d\n", i++);
4     } // while Ende
5     i = 0;
```

***Standard 58:***

Die Leeranweisung einer Schleife steht immer in einer eigenen Zeile.
Schlecht:

```
1     for (i = 0; i < 1000000; i++) {;}
```

Gut:

```
1     for (i = 0; i < 1000000; i++)
2     {
3       ;
4     } // for Ende
```

***Standard 59:***

Die `do/while`-Schleife muss immer mindestens eine nicht leere Anweisung enthalten.

## 12 Namenskonvention für Variablen

***Standard 60:***

In der Tabelle 1 ist eine Liste von Abkürzungen zu finden, die man bei der Bennenung von Variablen verwenden kann. In vielen Firmen gibt es Namenkonventionen für Funktionen und Variablen. Die ungarische Notation und Verwandte findet dabei immer mehr Zuspruch. Ziel der Namenkonvention ist es, Fehler durch Typenproblem zu erkennen und zu vermeiden.
Die Tabelle 1 ist von keine Seite *normiert*. Selbst in Microsoft zertifizierter Literatur findet man unterschiedliche Angaben. Die hier angegebene Tabelle stellt den für uns gültigen Standard dar. Dieser Standard ist so angelegt, dass er mit den gängigsten Namenskonventionen kompatibel ist.

| **Präfix** | **Bedeutung** |
|---|---|
| `i` | int |
| `s` | short int |
| `l` | long int |
| `ui` | unsigend int |
| `us` | unsigend short |
| `ul` | unsigned long |
| `f` | float |
| `d` | double |
| `ld` | long double |
| `c` | char |
| `sz` | (nullterminierter) String |
| `str` | String Objekt |
| `v` | void |
| `b` | bool |
| `fl` | flag |
| `p` | Zeiger |
| `a` | Array |
| `m_` | Attribut einer Klasse |

Tabelle 1: Präfix nach der ungarischen Notation

Schlecht:

```
1  int zaehler = 1, nenner = 3;
2  float quotient = 0.0;
3
4  /* 30000 Zeilen Source Code */
5
6  quotient = zaehler / nenner;
```

Das Ergebnis ist vermutlich nicht das, was der Programmierer erwartet. C führt hier eine ganzahlige Division durch, unabhängig davon, welcher Datentyp die Variable `ergebnis` hat.

Gut:

```
1 int   nZaehler  = 1;
2 int   nNenner   = 3;
3 float fQuotient = 0.0;
4
5  /* 30000 Zeilen Source Code */
6
7  fQuotient = nZaehler / nNenner;
```

Der Programmierer sieht sofort, dass es sich um eine ganzzahlige Division handelt.

Es ist auch üblich den Variablennamen durch einen Unterstrich vom Präfix zu trennen.

Auch Gut:

```
1 int i_zaehler = 1, i_nenner = 3;
2 float f_quotient = 0.0;
3
4 [30000 Zeilen Source Code]
5
6 f_quotient = i_zaehler / i_nenner;
```