Programmierstile in C

Von Sebastian Rothe

Inhaltsverzeichnis

1. Motivation - "wozu Code-Formatierung?"

Beim Schreiben einer Software ist oftmals ein Autor oder ein ganzes Team beteiligt. Allerdings entfällt die meiste Lebenszeit einer Software auf die Wartung. Selten wird diese ebenfalls vom ursprünglichen Autor vorgenommen. Es müssen sich also weitere Personen mit bereits vorhandenem Code leicht und schnell auseinandersetzen können, da ansonsten bereits vorhandene Teile der Software kaum effektiv weiter genutzt und gewartet werden können.

Daher ist Code-Formatierung ein sehr wichtiger Aspekt bei der Softwareentwicklung, er kann viel Zeit und damit auch Geld sparen. Man sollte sich also innerhalb eines Entwicklungsteams auf einen bestimmten Programmierstil einigen.

2. Definition: Programmierstil

Unter dem Begriff "Programmierstil" versteht man einen Satz von Regeln, nach denen der Programmierer den Code gestaltet. Hierbei werden verschiedene Aspekte aufgegriffen, auf die später detaillierter eingegangen wird. Dazu zählen:

- Quelltextformatierung
- Namenskonventionen
- Wiederverwendbarkeit/Wartung
- Dokumentation des Codes
- Robustheit durch Fehlerbehandlung
- Modularität der Software

Auf die Modularität eines Programms wird im weiteren Verlauf nicht weiter eingegangen. Sie beschreibt kurz gesagt den Aufbau einzelner Teile der Software, sodass diese schnell und flexibel miteinander kombiniert, erweitert und im Bedarfsfall auch ersetzt werden können.

Oftmals wird auch von "gutem Programmierstil" gesprochen. Dieser Begriff ist aber häufig umstritten, da er immer subjektiv ist. An sich gibt es keine feste Definition, was zu gutem Programmierstil zählt, einige Aspekte verstehen sich jedoch von selbst. Allerdings ist dann die Einhaltung dieser Aspekte nach einem

speziellen Muster (z.B. Namenskonventionen: *Ungarische Notation*) wieder umstritten, sodass es "Verfechter" verschiedener Stile gibt.

Betroffene Aspekte sind häufig:

- Der Einrückungsstil
- Die Zeilenlänge
- Die Namenskonventionen
- "Left-hand- vs. Right-hand-comparisons"

Jeder Programmierer entwickelt mit der Zeit seinen eigenen Programmierstil. Innerhalb eines Teams ist es jedoch wichtig, sich auf bestimmte Konventionen zu einigen und diese dann auch konsequent zu verwenden.

Einige der oben angesprochenen Konventionen, die oft (in Projekten) genutzt werden, sind zum Beispiel, nur eine Anweisung pro Zeile zu verwenden. Auch die auf 80 Zeichen beschränkte Zeilenlänge wird noch oft verwendet, allerdings wird dies vermehrt als "Relikt aus vergangenen Zeiten" angesehen. Hintergrund dieser Konvention ist, dass damals nicht mehr als ungefähr 80 Zeichen pro Zeile dargestellt werden konnten, ohne dass der Arbeitsfluss durch lästiges Scrollen ständig unterbrochen wurde. Abhilfe schaffen die heute besseren Bildschirmauflösungen, aber auch Entwicklungssoftware, die die Möglichkeit besitzt, Zeilen mit einer Zeichenlänge über 80 einfach "umzuklappen".

Vor allem in einem Team ist es sehr wichtig, die festgelegte Namenskonvention konsequent zu nutzen. Gleiches gilt für die Dokumentation, bei der es oftmals Vorgaben gibt. Darunter fällt unter anderem auch, bei einem zentral verwalteten Projekt (beispielsweise einer Website auf einem Testserver) Änderungen an einzelnen Teilen der Software beim Hochladen genau zu dokumentieren.

Einrückungsstil

Der Einrückungsstil ist für viele Softwareentwickler einer der Kernaspekte des Programmierstils. Er umfasst Elemente von der Positionierung der Klammern über die Tiefe der Einrückung bis zu der Verwendung von Leerzeichen bzw. Tabulatorzeichen und wirkt sich daher gravierend auf das "Format" des Codes aus. Über die Jahre hat sich eine beachtliche Anzahl von Einrückungsstilen entwickelt, allerdings erkennt man bei genauerem Betrachten, dass sich oftmals unterschiedliche Bezeichnungen für den (fast) gleichen Stil entwickelt haben. Einrückungsstile sind meistens von der verwendeten Programmiersprache abhängig . Im Folgenden werden ein paar der bekanntesten vorgestellt.

K&R / 1TBS / Kernel / Unix / "West Coast" / Stroustrup / Java / Sun

[Bild 1]: Variation des K&R, Funktionsdefinition nach Allman

```
int f(int x, int y, int z)
{
        if (x < foo(y, z)) {
                haha = bar[4] + 5;
        } else {
                while (z) {
                        haha += foo(z, z);
                        z--;
                }
                return ++x + bar();
        }
}
```

Der *K&R* ist einer der bekanntesten Einrückungsstile. Benannt wurde er nach *Brian Wilson Kernighan* und *Dennis Ritchie*, den Entwicklern von C. Man erkennt gut die öffnende Klammer am Ende eines Ausdrucks, wobei bei Funktionen eine neue Zeile genutzt wird. Die schließende Klammer liegt auf dem gleichen Level wie der Ausdruck. Vor allem in Java, C, C# und Perl wird der K&R oft genutzt. Je nach verwendeter Programmiersprache verwendet man zwischen 3 und 8 Leerzeichen pro Einrückungsschritt, vor allem beim *Java/Sun* (siehe Bild 2) werden nur wenige Leerzeichen verwendet.

[Bild 2]: Variation des K&R: Java/Sun

```
int f(int x, int y, int z) {
    if (x < foo(y, z)) {
        haha = bar[4] + 5;
    } else {
        while (z) {
            haha += foo(z, z);
            z--;
        }
        return ++x + bar();
    }
}
```

Allman / BSD / "East Coast" / Horstmann

Hier gibt es nur wenig Unterschiede zum *K&R*. Auffällig ist, dass nun die Klammerung immer in einer neuen Zeile beginnt. Eine kleine Ausnahme ist bei der abgeänderten Form nach Horstmann, dass die erste Anweisung innerhalb einer Funktion in der gleichen Zeile wie die öffnende Klammer steht:

[Bild 3]: Code-Beispiel nach Horstmann

```
int f(int x, int y, int z)
{   if (x < foo(y, z))
    {
        haha = bar[4] + 5;
    }
    else
    {
        while (z)
        {
            haha += foo(z, z);
            z--;
        }
        return ++x + bar();
    }
}
```

Durch diese Formatierung wird eine bessere Übersicht der umschließenden Klammern erreicht. Allerdings fallen so pro Anweisungsblock zusätzliche Zeilen allein für die Klammerung an, was den Lesefluss deutlich einschränken kann.

GNU-Stil

[Bild 4]: Code-Beispiel im GNU-Stil

```
int f (int x, int y, int z)
{
    if (x < foo (y, z))
      haha = bar[4] + 5;
    else
      {
        while (z)
          {
            haha += foo (z, z);
            z--;
          }
        return ++x + bar ();
      }
}
```

Dieser Stil wird wie der Name schon sagt vor allem, aber nicht überwiegend, in GNU-Projekten genutzt. Wesentliches Merkmal ist die Einrückung der geschweiften Klammern eines Anweisungsblocks. Das Hauptargument ist, dass dadurch die Lesbarkeit verbessert wird. Kritiker verwenden jedoch dieses Argument als Gegenargument, da ihrer Meinung nach die Lesbarkeit durch die verwirrende Einrückung nicht verbessert, sondern sogar noch verschlechtert wird.

Natürlich gibt es unzählige weitere Einrückungsstile, die sich teilweise nur im Detail von den eben gezeigten unterscheiden. Dies ist auch darauf zurückzuführen, dass es nur selten Einschränkungen durch die vorgegebene Syntax der verwendeten Sprache gibt. Gute Editoren und Entwicklungsumgebungen unterstützen den Programmierer oftmals beim Einrückungsstil. Es wird zwar immer noch viel über die Einrückungsstile diskutiert, aber schon Brian Kernighan und Dennis Ritchie sagten in ihrem Werk *The C Programming Language*:

"The position of braces is less important, although people hold passionate beliefs. We have chosen one of several popular styles. Pick a style that suits you, then use it consistently."

Vertikale Anordnung

Ein weiterer interessanter Aspekt ist die vertikale Anordnung von Codeteilen, beispielsweise Variablen gleichen Typs, die eine ähnliche Aussage besitzen.

[Bild 5]: Keine besondere Anordnung

```
1 int einArray[5] = {1, 2, 3, 4, 5};
2 int nochEinArray[5] = {6, 7, 8, 9, 0};
3
4 int einWert = 0;
5 int nochEinWert = 1;
6 int nochEinAndererWert = 2;
```

[Bild 6]: Vertikale Anordnung zur Gruppierung einzelner Variablen

```
1 int einArray[5]     = {1, 2, 3, 4, 5};
2 int nochEinArray[5] = {6, 7, 8, 9, 0};
3
4 int einWert             = 0;
5 int nochEinWert         = 1;
6 int nochEinAndererWert  = 2;
```

Durch diese abgeänderte Anordnung kann man einen besseren Bezug der Variablen zueinander schaffen und somit die zugewiesenen Werte besser vergleichen oder überprüfen. Beispielsweise lässt sich so schnell ein Array mit ungültiger Anzahl von Werten erkennen, wenn ein zweites mit gleicher Länge in der nächsten Zeile als direkter Vergleich dient.

Eine alternative Anordnung, die in der Skriptsprache PHP gerne genutzt wird, wäre folgende:

[Bild 7]: Alternative Anordnung in PHP

```
1 <?php
2             $einWert = 0;
3     $einAndererWert = 1;
4 $nochEinAndererWert = 2;
5 ?>
```

Gerade das letzte Beispiel lässt aber einen entscheidenden Nachteil erkennen: Beim Schreiben des Codes hat der Programmierer durch die spezielle Formatierung einen größeren Aufwand. Auch das nachträgliche Abändern vorgegebener Werte wie beispielsweise die Tabulatoreinrückweite kann im Code unerwünschte Ergebnisse mit sich bringen.

Leerzeichen & Tabulatoren

Leerzeichen werden beim Schreiben einer Software sehr oft verwendet, obwohl sie in vielen Sprachen an sich keine Auswirkung auf die Funktionalität des Endproduktes haben. Der "Whitespace" dient also erstrangig der Lesbarkeit. Auch hier gibt es wieder verschiedene Arten, Leerzeichen zu verwenden:

[Bild 8, 9 & 10]: Verschiedene Möglichkeiten der Nutzung von Leerzeichen

```
1 int i;
2 for(i=42;i>0;--i){
3   printf("%d \n",i);
4 }
```

```
1 int i;
2 for(i=42; i>0; --i){
3   printf("%d \n", i);
4 }
```

```
1 int i;
2 for(i = 42; i > 0; --i){
3   printf ("%d \n", i);
4 }
```

Auch hier hat jeder Programmierer seinen eigenen bevorzugten Stil. So nutzen manche nur sehr wenig Leerzeichen (Bild 8), einige fügen Leerzeichen bei Anweisungen mit mehreren Argumenten ein (Bild 9) und einige trennen strikt jede einzelne Anweisung und jedes Argument (Bild 10). Syntaktisch bedeuten die Beispiele jedoch alle das Gleiche. Trotzdem sollte man sich im Team auf einen Stil einigen.

Auch das Tabulatorzeichen wird oft genutzt. Sinnvoll ist es unter anderem bei einer speziellen vertikalen Anordnung. So lassen sich, wie im Beispiel gezeigt wird, bei der Deklaration von mehreren Variablen der Variablentyp und der Bezeichner gut trennen:

[Bild 11]: Keine Nutzung von Tabs

```
1 int i;
2 const int x;
3 double j;
4 char c;
5 char s[];
```

[Bild 12]: Anordnung durch Tabs

```
1 int       i;
2 const int x;
3 double    j;
4 char      c;
5 char      s[];
```

Neben der Frage nach der Nutzung von Leerzeichen und Tabulatorzeichen gibt es auch die Diskussion, ob Einrückungen durch Leerzeichen *oder* Tabs vorgenommen werden sollten. Dies greift noch einmal auf den Aspekt der Einrückungsstile zurück. Beide Methoden haben ihre Vorteile. Zum Einrücken genutzte Leerzeichen werden unabhängig von den Anzeigeoptionen des jeweiligen Editors konstant dargestellt. Dafür kann der Programmierer bei der Nutzung des Tabulatorzeichens die Einrückungstiefe selbst wählen und auch

ohne Probleme nachträglich ändern. Auch hier gilt wieder, dass man die beiden Varianten außer in speziellen Situationen nicht mischen sollte.

Left-hand- vs. Right-hand-comparisons

Zunächst ein kleines Codebeispiel in Perl zur Verdeutlichung des Problems:

[Bild 13]: Einfacher Vergleich

```
# Vergleich:
if($einWert == 42) {...} # Right-hand
if(42 == $einWert) {...} # Left-hand
```

[Bild 14]: if-Abfrage mit Zuweisung

```
# Achtung! Zuweisung:
if($einWert = 42) {...} # Right-hand
if(42 = $einWert) {...} # Left-hand
```

Bei der Left-hand-comparison steht der zu vergleichende Ausdruck auf der linken Seite des Vergleichsoperators, die Right-hand-comparison ist dementsprechend das Gegenteil. Für gewöhnlich nutzen die meisten Programmierer die Right-hand-comparison.

Bei dem oben gezeigten Codebeispiel tritt nun jedoch folgendes Problem auf: Verwendet ein Programmierer wie in Bild 14 gezeigt aus Versehen den Zuweisungs- statt einem Vergleichsoperator, so wird bei der allgemein genutzten Right-hand-comparison der Variable $einWert der Wert 42 zugewiesen und dann der folgende Anweisungsblock ausgeführt. Bei der Left-hand-comparison bricht das Programm jedoch mit einem Fehler ab, der Bug würde also schnell erkannt werden.

Dieses Problem ist ein gutes Beispiel dafür, dass sich die Lesbarkeit (im Sinne von Gewohnheit, da überwiegend die Right-hand-comparison genutzt wird) auch auf die Funktionalität eines Programms auswirken kann.

Kommentare

Kommentare sind ein wichtiges und oftmals unterschätztes Element. Sie sollten ausgiebig genutzt werden und sind für Team-Projekte unerlässlich, um komplexe Anweisungsblöcke für andere Teammitglieder zugänglich zu machen. Aber auch bei Kommentaren gibt es unterschiedliche "Richtlinien". Der Code soll nicht erklärt oder noch einmal wiederholt werden. Vielmehr geht es darum, die Absicht von Codezeilen deutlich zu machen. Zusätzlich lassen sich Kommentare auch sehr gut als Platzhalter verwenden, um für andere oder sich selbst zu notieren, dass noch gewisse Aufgaben erfüllt werden müssen. Auch zum Debuggen sind Kommentare sehr praktisch. Dieses Verfahren nennt sich "Auskommentieren". Dabei werden einzelne Anweisungen oder ganze Blöcke als Kommentar gekennzeichnet, sodass diese vom Compiler nicht mehr als Code angesehen werden und unterschiedliches Verhalten der Software überprüft werden kann.

In C gibt es zwei verschiedene Arten von Kommentaren: Block-Kommentare, die sich über mehrere Zeilen erstrecken können, und einzeilige Kommentare. Letztere zählen jedoch erst ab 1999 zum C-Standard, sodass es mit einzeiligen Kommentaren bei alten Compilern vereinzelt noch zu Schwierigkeiten kommen kann.

Namenskonventionen

Namenskonventionen sind vor allem in Team-Projekten ebenfalls ein wichtiger Aspekt eines guten Programmierstils. Oft wird die sogenannte *Ungarische* Notation verwendet. Durch diese Konvention kann man beispielsweise schnell erkennen, um was für einen Variablentyp es sich bei einer Variable handelt:

[Bild 15]: Ungarische Notation

```
1 int iSumme;
2
3 long lGrosseZahl;
4
5 char chZeichen;
6
7 char stString[];
8
9 int const iUMSATZSTEUER;
```

Es gibt auch oftmals sprachenspezifische Namenskonventionen. Typisch für C sind zum Beispiel:

- Konstanten werden in Großbuchstaben deklariert,
- Funktionen, Typendefinitionen und Variablen in Kleinbuchstaben
- Bezeichner mit führendem Unterstrich sollten nicht verwendet werden
- Namen vermeiden, die sich in nur einem Buchstaben unterscheiden
- Namen vermeiden, die ähnlich aussehen (kleines L, großes I, 1)

3. GLib als Beispiel

GLib ist eine Bibliothek für C, welche verschiedene, sonst nur schwer zu realisierende Funktionen enthält. Dementsprechend ist es ziemlich wichtig, auf guten Programmierstil zu achten, damit diese Bibliothek weiterhin gewartet und erweitert werden kann. Sie beinhaltet eine Vielzahl von komplexen Funktionen, unter anderem betreffen diese die Themen Makros, Typumwandlungen, diverse Funktionen rund um Timer, verschiedene Datenstrukturen, Threads und die Möglichkeit, Strings auf verschiedene Arten zu verarbeiten.

Man erkennt in GLib verschiedene Aspekte der angesprochenen Konventionen. Vor allem der Schwerpunkt, der auf der vertikalen Anordnung liegt, wird sichtbar. Ebenfalls erkennbar sind die Kommentare für große if-Anweisungen, damit Programmierer schnell Übergänge zu weiteren else-Blöcken finden und auch wissen, welche Fälle diese abdecken.

[Bild 16]: Auszug aus einem Skript der GLib-Bibliothek

```
311 /**
312  * g_time_val_add:
313  * @time_: a #GTimeVal
314  * @microseconds: number of microseconds to add to @time
315  *
316  * Adds the given number of microseconds to @time_. @microseconds can
317  * also be negative to decrease the value of @time_.
318  **/
319 void
320 g_time_val_add (GTimeVal *time_, glong microseconds)
321 {
322   g_return_if_fail (time_->tv_usec >= 0 && time_->tv_usec < G_USEC_PER_SEC);
323
324   if (microseconds >= 0)
325     {
326       time_->tv_usec += microseconds % G_USEC_PER_SEC;
327       time_->tv_sec += microseconds / G_USEC_PER_SEC;
328       if (time_->tv_usec >= G_USEC_PER_SEC)
329         {
330           time_->tv_usec -= G_USEC_PER_SEC;
331           time_->tv_sec++;
332         }
333     }
334   else
335     {
336       microseconds *= -1;
337       time_->tv_usec -= microseconds % G_USEC_PER_SEC;
338       time_->tv_sec -= microseconds / G_USEC_PER_SEC;
339       if (time_->tv_usec < 0)
340         {
341           time_->tv_usec += G_USEC_PER_SEC;
342           time_->tv_sec--;
343         }
344     }
345 }
```

Deutlich wird, wie wichtig in einer großen Bibliothek wie GLib die richtige Wahl der Variablennamen ist. So werden lange Bezeichner mit Unterstrichen unterteilt. Auch auf Kommentare, zum Beispiel zur Umschreibung einer Funktion, wird in GLib sehr viel wert gelegt. Man erkennt im Beispiel das strikte Einrücken der Anweisungsblöcke, um die Übersicht gewährleisten zu können. Bei "kleinen" Blöcken wie oben gezeigt wird jedoch auf zusätzliche Kommentare verzichtet. Beinhaltet eine Funktion mehrere oder lange Parameter, wird auch oftmals der Rückgabetyp der Funktion in eine eigene Zeile gestellt.

4. Codeformatierungssoftware

Der Begriff Codeformatierungssoftware erklärt sich eigentlich von selbst, jedoch kann man zwischen verschiedenen Möglichkeiten der Codeformatierung unterscheiden. Zunächst gibt es einige Programme, die den geschriebenen Code nur analysieren. Dann gibt es aber auch Programme, die beispielsweise über übergebene Parameter den Code auch formatieren. Und schlussendlich gibt es diverse Editoren, die den Programmierer bereits beim Schreiben der Software unterstützen.

Ein Programm zur C-Codeformatierung ist *Indent*. Über verschiedene Parameter können diverse Formatierungen vorgenommen werden. Das folgende Beispiel zeigt eine zum Linux-Stil äquivalente Formatierung:

[Bild 17]: Indent-Formatierung im Linux-Stil

```
indent -nbad -bap -nbc -bbo -hnl -br -brs -c33 -cd33 -ncdb -ce -ci4
       -cli0 -d0 -di1 -nfc1 -i8 -ip0 -l80 -lp -npcs -nprs -npsl -sai
       -saf -saw -ncs -nsc -sob -nfca -cp33 -ss -ts8 -il1
       code.c -o code_formatiert.c
```

Jeder Parameter steht für verschiedene Formatierungsschritte, so zum Beispiel:

- -bap: blank line after procedure body
- -br: braces on if-line
- -brs: braces on struct declaration line
- -saf: space after for

Editoren, die direkt Unterstützung bei der Code-Formatierung geben können, gibt es sehr viele. Einige der hier aufgeführten Programme sind IDEs, liefern also weitere Unterstützungsmöglichkeiten, die sich nicht nur auf Formatierung beschränken.

- Bloodshed Dev-C++ (auch für C geeignet)
- Notepad++
- PSPad
- Eclipse IDE for C/C++ Developers
- Microsoft Visual Studio
- NetBeans IDE

5. Zusammenfassung

Es zeigt sich zusammenfassend, dass es zahlreiche und auch unterschiedlichste Programmierstile gibt. Schlussendlich liegt das natürlich auch daran, dass jeder Programmierer seine eigene Auffassung von gutem Programmierstil hat. Wichtig ist eigentlich eher, dass man seinen eigenen Stil findet und diesen dann auch für sich persönlich konsequent nutzt.

Im Team hingegen sollte ein einheitlicher Stil genutzt werden, auch wenn man einen anderen Stil gewohnt ist. Programmierstile stellen an sich aber auch keine Regeln, sondern eher Richtlinien auf. Sollte es also in Einzelfällen Schwierigkeiten geben, kann man diese Richtlinien gut begründet auch umgehen.

6. Quellen

- *http://de.wikipedia.org/wiki/Programmierstandard*
- *http://en.wikipedia.org/wiki/Indent_style*
- *http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml*
- *http://en.wikipedia.org/wiki/Programming_style*
- *http://en.wikipedia.org/wiki/Coding_conventions*
- *http://en.wikipedia.org/wiki/Naming_convention_(programming)*
- *http://golang.org/cmd/gofmt/*
- *http://de.wikipedia.org/wiki/C_%28Programmiersprache%29*
- *http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#C*
- *http://www.dmoz.org/Computers/Programming/Development_Tools/Source_Code_Formatters/*
- *http://www.gnu.org/software/indent/*
- *http://stackoverflow.com/questions/411249/coding-style-checker-for-c*
- *http://wr.informatik.uni-hamburg.de/teaching/sommersemester_2011/c-programmierung*
- *http://wr.informatik.uni-hamburg.de/teaching/organisatorische_hinweise#proseminare*
- *http://www.inf.udec.cl/~leo/cstyguid.pdf*
- *http://www.psgd.org/paul/docs/cstyle/cstyle11.htm*
- *http://pronix.linuxdelta.de/C/standard_C/c_programmierung_5.shtml*

- Codebeispiele [1] –[4]: *http://de.wikipedia.org/wiki/Einrückungsstil*
- Codebeispiele [5] –[7]: *http://en.wikipedia.org/wiki/Programming_style*
- Codebeispiel [13] + [14]: *http://en.wikipedia.org/wiki/Coding_conventions#Left-hand_comparisons*
- Codebeispiel [16]: GLibVersion 2.26.1 (Windows),*glib**gtimer.c*
- Codebeispiel [17]: *http://www.gnu.org/software/indent/manual/indent.html#SEC4*
- Alle Codebeispiele wurden mit *PSPad* noch einmal überarbeitet