Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Wintersemester 2004/2005
Software-Praktikum ‹Eihne›
Prof. Dr. Peter Thiemann, Stefan Franck

Formatierung von Quellcode

Vito Di Leo, Gunnar Ritter, Michael Schröder

26. 01. 2005

1. Teil: Formate von Textdateien

Quellcode wird in den meisten Programmiersprachensystemen in Textdateien abgelegt. Das heißt, die Dateien enthalten nur die Zeichen des Programmes, in Zeilen eingeteilt. Weitere Gestaltungsanweisungen stehen nicht zur Verfügung.

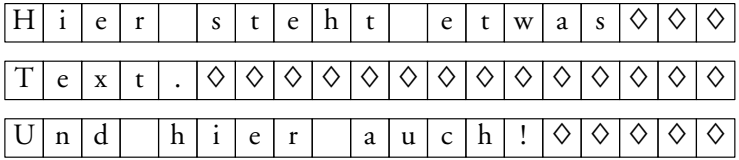
Leider gibt es aber zwischen verschiedenen Systemen kleine Unterschiede in den Formaten von Textdateien.

Historisches: Lochkarten

Textdateien haben sich aus Lochkarten entwickelt, wie sie in den sechziger Jahren des 20. Jahrhunderts gebräuchlich waren. Jede Lochkarte entspricht einer Zeile.

Auf jede Lochkarte passte eine genau bestimmte Anzahl von Zeichen, meistens 80. Deshalb waren alle Zeilen gleich lang.

Auf Großrechnersystemen gibt es noch heute Textdateiformate, die genau so organisiert sind, also aus Datensätzen von stets je 80 Zeichen bestehen.

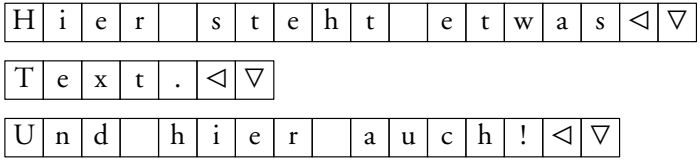


Historisches: Terminals

In den siebziger Jahren arbeitete man lieber auf Terminals, die über serielle Leitungen an den Rechner angeschlossen waren. Da diese Leitungen sehr langsam waren, hätte es störend viel Zeit verschwendet, immer ganze Datensätze zu übertragen.

Deshalb verwendete man zur Darstellung von Texten ein anderes Format: Man signalisierte das Ende einer Zeile mit speziellen Steuerzeichen. So musste man vorher nur die belegten Positionen übertragen und sparte sich so etwas Warterei.

Systeme, die auf die Arbeit mit solchen Terminals ausgerichtet sind, verwenden auch ein ähnliches Format, um Textdateien zu speichern. Das betrifft Unix genauso wie andere Minicomputer-Systeme. Auch die Windows-Konventionen haben sich über ein paar Umwege hiervon abgeleitet.



Die ASCII-Zeichencodierung

Auf Lochkarten gab es meistens oben eine Zeile für Gedrucktes und darunter Platz für die Löcher, ungefähr ein knappes Dutzend pro Spalte und damit pro Zeichen. Jeder Hersteller hatte sein eigenes Lochkartenformat, und deshalb machte es auch nicht viel aus, welcher gestanzter Code zu welchem Zeichen gehörte.

Bei Terminals wurde das wichtiger. Nicht per se, aber einfach deshalb, weil der Minicomputer-Markt vielfältiger war und die Leute die Terminals nicht zwangsläufig vom Hersteller ihres Computers kauften.

Ungefähr alle Minicomputer sowie deren Terminals haben den ASCII-Code verwendet. Deshalb verwenden auch alle Systeme, die uns heute interessieren, den ASCII-Code. Mit diesem Code lassen sich Zahlen, Zeichen von A bis Z in groß und klein sowie eine Reihe von Symbolen darstellen.

Glücklicherweise sind das auch genug Zeichen für alle Programmiersprachen, die uns interessieren.

Deutsch: Umlaute usw.

Wenn wir deutsche Texte schreiben, möchten wir aber auch gerne Umlaute verwenden.

Leider haben wir da etwas Pech. Umlaute sind im ASCII-Zeichensatz nicht enthalten. Sie wurden für Computer erst in den achtziger Jahren wichtig, und da waren die Entwicklungslinien unserer heutigen Systeme schon weiter voneinander getrennt.

Deshalb verwenden Unix-Systeme, Apple-Systeme, Microsoft-Systeme, und was es noch so gibt, alle unterschiedliche Codierungen für Umlaute.

ISO-8859-1 und Windows-1252

Wir haben aber auch ein bisschen Glück. Die meisten dieser Codierungen können wir schlicht vergessen, weil sie schon wieder fast ausgestorben sind.

Seit Mitte der neunziger Jahre haben Unix- und Windows-Systeme sogar meistens dieselbe Codierung verwendet, nämlich ISO-8859-1. Dieser Code verwendet das höchste Bit eines 8-Bit-Bytes, das für ASCII nicht benötigt wird, sodass bis zu 128 weitere Positionen für Zeichen verfügbar werden. Die werden vor allem für Umlaute und Buchstaben mit Akzenten benutzt.

Der Unterschied zwischen ISO-8859-1 und Windows-1252 besteht einfach darin, dass Microsoft die Positionen mit den Codes 128 bis 159 für weitere druckbare Zeichen verwendet. In ISO-8859-1 stehen sie hingegen für Steuerzeichen, die hierzulande zwar fast niemand braucht, die aber trotzdem für Überraschungen sorgen können, wenn ein Windows-1252-Text auf einem ISO-8859-1-Terminal ausgegeben wird.

Mittlerweile ist aber sowohl für Unix als auch für Windows eine weitere Codierung immer wichtiger geworden. Es ging manchen Leuten nämlich ziemlich auf die Nerven, dass sie verschiedene Codes benutzen müssen, wenn sie in einem Text z. B. sowohl Deutsch als auch Griechisch verwenden möchten. Deshalb haben sie Unicode erfunden, eine Zeichencodierung, die mittlerweile über hunderttausend Zeichen bestimmt. Das funktioniert natürlich nicht kompatibel mit Programmen, die erwarten, dass jedes Zeichen in acht Bit passt.

UTF-8 stellt eine Methode bereit, mit der sich Unicode ASCII-kompatibel repräsentieren lässt. Es nutzt dafür genau die Bytes mit dem höchsten Bit, die auch ISO-8859-1 und Windows-1252 nehmen. Dafür entspricht dann nicht mehr jedes einzelne dieser Bytes einem Zeichen.

Stattdessen ist festgelegt, welche Sequenzen von je zwei bis vier Bytes zu welchem Unicode-Zeichen gehören. Programme, die zeichenorientiert arbeiten – etwa Texteditoren –, müssen natürlich entsprechend überarbeitet werden.

Und wir?

Bei manchen Sprachen kann man angeben, welche Zeichencodierung im Text verwendet wird, so z. B. bei XML.

Bei den meisten richtigen Programmiersprachen gibt es diese Möglichkeit aber nicht.

Java verwendet zwar intern Unicode, überlässt das Format von Programmen aber dem jeweiligen System.

Wenn wir unseren Quellcode über Subversion austauschen wollen, müssen wir uns daher absprechen, welche Codierung wir verwenden möchten.

Wir könnten uns allerdings auch entscheiden, einfach überhaupt kein Deutsch im Quellcode zu verwenden, weder in Meldungen noch in Kommentaren.

Das wird in professionellen Umgebungen auch gerne gemacht. Wenn man trotzdem Meldungen in anderen Sprachen als Englisch ausgeben will, nimmt man dafür spezielle Lokalisierungsmechanismen, die es in den meisten modernen Programmiersprachen gibt.

Zeilenlänge

Die meisten Lochkarten hatten wie gesagt Platz für 80 Zeichen pro Zeile. Dies galt auch für die meisten seriellen Terminals.

Uns muss das heute eigentlich nicht mehr zwangsläufig beschränken. Auch konservative Programmierer, die Kommandozeilentools einer GUI-IDE vorziehen, arbeiten nicht mehr mit seriellen Terminals, sondern mit Konsolen oder Emulatoren, die wesentlich mehr Zeichen pro Zeile darstellen können oder bei denen sich die Zeilenlänge sogar dynamisch verstellen lässt.

Trotzdem macht es Sinn, Zeilen nicht allzu lang werden zu lassen. Bei der Buchherstellung hat man über Jahrhunderte hin erkannt, dass Zeilen optimalerweise sogar nur 50 bis 60 Zeichen umfassen sollten. Das kommt daher, weil der Leser, der mit seinem Auge am Ende einer Zeile angekommen ist, von diesem Punkt aus wieder nach links zurückschauen und den Anfang der nächsten Zeile finden muss. Je länger eine Zeile ist, desto schlechter lässt sie sich bei diesem Prozess zurückverfolgen. Lange Zeilen erschweren daher das Lesen.

80 Zeichen als Norm

Viele Projekte entscheiden sich daher, die Zeilen doch auch ohne technische Zwänge bei höchstens 80 Zeichen zu belassen.

Neben der Lesbarkeit spricht dafür auch noch, dass man dadurch stets daran erinnert wird, Blöcke nicht zu tief zu verschachteln und lieber neue Funktionen oder Methoden einzuführen.

Code lässt sich an jeder Whitespace-Stelle umbrechen, und auch Textstrings lassen sich in *Java* leicht mit ‹+› aufteilen. Daher gibt es mit der Umsetzung einer solchen Regel kaum Probleme, wenn man sich einmal an sie gewöhnt hat.

Zeilenenden

In den Textdateiformaten, die von Minicomputer-Systemen stammen, sind wie bereits erläutert Steuerzeichen dafür zuständig, das Ende einer Zeile zu signalisieren.

Bei der Ausgabe auf ein Terminal sind das stets ein ‹\r› (Wagenrücklauf) und ein ‹\n› (Zeilenvorschub), ganz ähnlich wie bei einer Schreibmaschine. Für das Terminal ist es dabei egal, in welcher Reihenfolge man ihm die Zeichen sendet.

In Textdateien ist das aber nicht egal, sondern systemabhängig. Windows-Programme erwarten, dass das Zeilenende durch ‹\r\n›, genau in dieser Reihenfolge, angezeigt wird.

Unix-Programmen genügt hingegen ein einzelnes ‹\n›. Wenn noch ein ‹\r› vorkommt, ist das für sie ein sinnloses Zeichen, das sie entweder ignorieren oder das sie zu häufig merkwürdig aussehenden Fehlermeldungen treibt. Denn wenn das ‹\r› einfach so auf das Terminal geschrieben wird, kehrt dieses gehorsam zum Zeilenanfang zurück und überschreibt danach die Zeichen, die da schon gestanden hatten.

Für das Repository sollten wir auch hier zu einer gemeinsamen Regelung finden.

Am Dateiende

Der POSIX-Standard, an dem sich alle wichtigen Unix-Systeme seit vielen Jahren orientieren, schreibt vor, dass das letzte Zeichen einer Textdatei ein Zeilende (‹\n›) sein muss. Das tut er nicht ohne Grund; viele Unix-Tools hatten früher Probleme, wenn das nicht der Fall war.

Einigen wir uns auf die Unix-Konvention für Zeilenenden, sollten wir auch das berücksichtigen.

Leerer Raum vor dem Zeilenende

Ein lediglich kosmetisches Problem ist es, wenn nach dem letzten sichtbaren Zeichen einer Zeile noch Leerzeichen oder Tabulatoren stehen. Sowas ist aber in einigen Dateien vorgekommen, und man sollte wissen, dass es von vielen Programmierern als unsorgfältig angesehen wird.

Deshalb sollte man das nach Möglichkeit vermeiden.

Zusammenfassung: Formatierung von Textdateien

- Zeichencodierungen: Nur englische Kommentare und Meldungen in ASCII verwenden, oder Zeichensatz für deutsche Texte für das ganze Projekt festlegen
- Zeilenlänge: Beschränkung einführen, z. B. auf 80 Zeichen
- Zeilenenden: Unix- (‹\n›) oder Windows-Konvention (‹\r\n›) projektweit festlegen
- Bei Unix-Textdateien auf das ‹\n› am Dateiende achten
- Leerraum unmittelbar vor dem Zeilenende vermeiden

2. Teil: Indenting

Bei vielen alten Programmiersprachen aus der Lochkartenzeit war fest vorgegeben, auf welche Spalten einer Zeile welcher Teil eines Programmes zu stehen hatte.

Glücklicherweise sind wir da heute freier. Wir könnten unseren Code, wenn wir wollten, auch komplett auf eine riesige Zeile schreiben. Das das sinnlos ist, ist natürlich klar.

Trotzdem gibt es eine ganze Reihe von Möglichkeiten, Code auch auf kürzere Zeilen gezielt zu verteilen, indem man die kleineren Ansammlungen von Leerraum *(Whitespace)* zwischen den Token des Programms geregelt organisiert.

Warum das wichtig ist

Es ist natürlich eine berechtigte Frage, warum man sich die Mühe macht.

Das Code irgendwie aufgeräumt aussehen sollte, ist zwar fast unmittelbar einsichtig.

Dass man dafür aber Regeln braucht und es nicht einfach nach Geschmack machen kann, ist es vielleicht nicht.

Code ist natürlich in sich bereits strukturiert. Diese Strukturen erkennt der Compiler, indem er Zeichen scannt und die erkannten Token mit einer Grammatik vergleicht.

Menschen lesen Code aber nicht so. Es wäre nämlich äußerst mühsam, z. B. die ‹if›-Statements so zu zählen, wie man an ihnen beim Lesen vorbeikommt.

Deshalb kann man sich und anderen das Lesen erleichtern, indem man die Strukturen, die der Quellcode aufweist, auch durch die visuelle Organisation erkennbar macht. Je genauer man dabei vorgeht, desto mehr Einzelheiten kann man dann auch beim Lesen unterscheiden.

diff

Es gibt aber noch handfestere Gründe, strikte Regeln für die Formatierung von Code zu haben.

Eines der nützlichsten Hilfswerkzeuge beim Programmieren auf Unix-Systemen ist *diff*. Es erzeugt eine Liste der Änderungen zweier Fassungen von Textdateien, die man entweder lesen und selbst interpretieren oder an ein Programm namens *patch* weitergeben kann, das die eine Fassung in die andere umwandelt. Jedes Unix-Entwicklungsprojekt setzt dieses Werkzeug ein, um Änderungen zwischen Programmierern auszutauschen. Es ist aber auch nützlich, um sich selbst einen Überblick über das zu verschaffen, was man etwa nach ein paar Stunden Debugging tatsächlich geändert hat.

Versionskontrollsysteme ermöglichen es meist, *diff* direkt auf verschiedene Fassungen einer Datei anzuwenden Mit dem Subversion-Kommando ‹svn help diff› kann man sich hierzu eine Übersicht verschaffen.

unified diff

diff kann die Änderungen auf unterschiedliche Weise darstellen.
Die heute gebräuchlichste Form ist das *unified diff:*

```
@@ -255,8 +255,11 @@
 {
        char *charset;

-       if (isclean & MIME_HIGHBIT) {
-               charset = value("charset");
+       if (isclean & (MIME_CTRLCHAR|MIME_HASNUL))
+               charset = NULL;
+       else if (isclean & MIME_HIGHBIT) {
+               charset = wantcharset ? wantcharset :
+                       value("charset");
                if (charset == NULL) {
                        charset = defcharset;
                }
```

diff und Formatierung

Natürlich kann *diff* nicht alle Programmiersprachen verstehen und Programme so auf semantisch relevante Änderungen untersuchen – schon deshalb nicht, weil man ja u. U. mit der Änderung einen syntaktischen Fehler behoben hat.

Deshalb vergleicht *diff* nur einzelne Zeilen der Dateien miteinander und gibt diejenigen aus, in denen es Abweichungen gefunden hat.

diff kann dabei optional Leerraum innerhalb von Zeilen ignorieren, aber es kann schon nicht mehr erkennen, dass Code nur über Zeilengrenzen hinweg verschoben wurde.

```
callFunction(firstArgument, secondArgument,
        thirdArgument);

callFunction(firstArgument,
        secondArgument,
        thirdArgument);
```

Erzwungene Formatierung

Für manche Projekte ist *diff* so nützlich, dass es eine Vorschrift gibt, Code automatisiert formatieren zu lassen, bevor er ins Repository gelangt.

Das ist besonders bei Sprachen interessant, die für den Menschen schlecht zu übersehen sind und daher meist nicht direkt mit Texteditoren bearbeitet werden.

Ein Beispiel dafür ist XML. Programme zur XML-Bearbeitung implementieren i. d. R. keine, schlechte oder unterschiedliche Formatierungen ihrer Ausgabe. Ohne nachträgliche erzwungene Formatierung wäre *diff* in solchen Fällen nutzlos.

Auch Kleinigkeiten können nützlich sein

Aber auch mit einzelnen kleinen Regeln kann man nützliche Wirkungen erzielen. Hält man sich daran, Funktionen in *C* stets so zu schreiben:

```
int
foo(int bar)
{
        ...
```

dann kann man auf der Kommandozeile einfach mit

```
$ grep '^foo(' *.c
```

nach der Datei suchen, in der die jeweilige Definition steht.

Indenting im engeren Sinne

Die wichtigsten Fragen bei der Formatierung sind, welche Codeblöcke man einrückt und wieviel Leerraum man dafür verwendet. Das nennt man ‹Indenting›. In (der Schreibweise nach) *C*-ähnlichen Sprachen wie *Java* haben sich hier ein paar Standardmethoden ausgeprägt.

Indenting in *C*-ähnlichen Sprachen – Tafel

```
if (condition) {
        statement;
}
```

Kernighan-und-Ritchie-Stil

```
if (condition)
        {
        statement;
        }
```

Whitesmiths-Stil

```
if (condition)
{
        statement;
}
```

Allman-Stil

```
if (condition)
  {
    statement;
  }
```

GNU-Stil

Indenting in *C*-ähnlichen Sprachen – Beschreibung

Dennis Ritchie ist der Erfinder von *C,* und er hat zusammen mit Brian Kernighan das erste Buch über diese Sprache geschrieben. Die Beispiele in diesem Buch verwenden diesen Indenting-Stil, ebenso wie der meiste ursprüngliche Unix-Code und auch der Linux-Kernel. Da sich die Leute, die für Sun die *Java*-Bücher setzen, daran orientiert haben, hat sich dieser Stil auch bei *Java*-Programmierern verbreitet.

Eric Allman hat *sendmail* programmiert, einst das wichtigste Mailserverprogramm im sich entwickelnden Internet.

‹Whitesmiths› war eine Firma, die um 1980 einen der ersten *C*-Compiler für Nicht-Unix-Systeme hergestellt hat.

Den Stil, den das GNU-Projekt favorisiert, mögen die meisten Leute außerhalb dieses Projektes nicht besonders. Man darf also keineswegs annehmen, dass dieser Stil generell bei Open-Source-Programmieren beliebt sei.

Die ersten drei Stile kann man genausogut mit acht wie mit vier Spalten Einrückung umsetzen. Manche Leute favorisieren sogar nur zwei Spalten.

Sekundäres Indenting

Wenn man innerhalb eines einzelnen Statements oder innerhalb einer Bedingung umbricht, setzt man sekundäres Indenting ein, um das deutlich zu machen. Dafür gibt es i. W. folgende Möglichkeiten:

```
if (thisConditionIsTooLongSoWeHave
                == toSplitIt)
        thenDoThis;
```

```
if (thisConditionIsTooLongSoWeHave
    == toSplitIt)
        thenDoThis;
```

Wie man sieht, hat die obere Methode den Vorteil, dass man die Zugehörigkeit besser erkennt. Bei der unteren hat man hingegen mehr Platz für die zweite Zeile, sodass man seltener gezwungen ist, auch sie noch umzubrechen.

Tabulatoren und Leerzeichen

Für das primäre Indenting sind Tabulatoren am besten geeignet. Gute Editoren erlauben es, die Weite von Tabulatoren einzustellen. Damit kann dann jeder Programmierer den Leerraum ganz nach seinem Geschmack konfigurieren, ohne dass irgendeine Änderung im Quelltext erforderlich wird.

Will man Text an einer ganz bestimmten Stelle ausrichten, verwendet man ab dem Indenting-Level besser Leerzeichen. Das kann man z. B. sinnvoll einsetzen, um Strings umzubrechen.

```
int foo(int bar) {
|   TAB   |if (condition) {
|   TAB   ||   TAB   |System.out.println("This is a " +
|   TAB   ||   TAB   |                       "long message");
```

bleibt auch mit vier Zeichen pro Tabulator korrekt formatiert:

```
int foo(int bar) {
| TAB |if (condition) {
| TAB || TAB |System.out.println("This is a " +
| TAB || TAB |                       "long message");
```

Präzedenz vs. Klammerung

Bei komplizierteren Ausdrücken kann man darüber streiten, wieviele Klammern am sinnvollsten sind:

```
if ((((a + b) == (c + d)) && ((e - f) == (g - h))) ||
    (((i * j) == (k * l)) && ((m / n) == (o / p)))) {

if (((a + b == c + d) && (e - f == g - h)) ||
    ((i * j == k * l) && (m / n == o / p))) {

if ((a + b == c + d && e - f == g - h) ||
    (i * j == k * l && m / n == o / p)) {

if (a + b == c + d && e - f == g - h ||
    i * j == k * l && m / n == o / p) {
```

Alle Ausdrücke sind semantisch gleichwertig. Die Frage ist im wesentlichen, welche Präzedenzregeln man für so klar hält, dass man durch das Einsparen von Klammern weniger nach passenden Paaren suchen muss. Das hängt sehr von der persönlichen Erfahrung ab, was eine strikte Position fragwürdig macht.

Leerzeichen um Klammern

Manche Programmierer finden es übersichtlicher, wenn unmittelbar aufeinanderfolgende Klammern mit Leerzeichen getrennt werden:

```
if ( ( (a + b) % c) / d) {
```

Das widerspricht allerdings den allgemeinen Regeln für das Setzen von Klammern in Texten, wie sie etwa im Duden zu finden sind, und wirkt daher auf andere Programmierer ungewohnt oder störend.

```
if (((a + b) % c) / d) {
```

Leerzeichen in arithmetischen Ausdrücken

Arithmetische Ausdrücke kann man gedrängt oder geweitet schreiben:

```
i = j + k * n;

i = j + k*n;

i = j+k*n;

i=j+k*n;
```

Das zweite Beispiel hat immerhin den Vorteil, dass die Präzedenz visualisiert wird. Das kann aber manchmal auch zur Falle werden, wenn die Anordnung nämlich falsch ist.

```
i = j+k * n;
```

Man sollte sich hier auch nicht zuviel Mühe machen. Komplizierte Statements spaltet man i. a. besser auf, ggf. sogar durch Einführung weiterer Variablen.

Minus oder negativ

Man sollte zwischen einer negativen Konstante und der Substraktion einer positiven Konstante mittels Leerzeichen unterscheiden:

```
a = b - 1;
```

```
a = b + -1;
```

nicht

```
a = b -1;
```

Das Semikolon

Insbesondere in *for*-Schleifen ist der Leerraum um das Semikolon interessant:

```
for (i=0;i<n;i++) {

for (i = 0; i < n; i++) {

for (i = 0 ; i < n ; i++) {
```

Hier muss man abwägen, ob einem Platzverbrauch (ganz oben), Orientierung an typographischen Konventionen (Mitte) oder Deutlichkeit am wichtigsten sind.

Funktionen und Methoden vs. Schlüsselwörter

Häufig wird favorisiert, Funktions- bzw. Methodenaufrufe und Schlüsselwörter so deutlich wie möglich durch Leerraum und Klammersetzung zu unterscheiden:

```
callFunction(argument);

return value;
```

Und das sind ein paar andere Möglichkeiten:

```
callFunction (argument);
callFunction ( argument ) ;

return(value);
return (value);
return ( value ) ;
```

Leerzeichen im allgemeinen

Auf Leerzeichen ist zu achten. Man kann zwar prinzipiell beides schreiben,

```
int foo(int bar) {
```

```
int foo(int bar){
```

sollte sich aber für eines entscheiden und mindestens innerhalb derselben Datei konsequent verwenden.

Leerzeilen zwischen Statements

Ob und wie oft er Code innerhalb derselben Funktion oder Methode durch Leerzeilen trennt, ist meist die eigene Entscheidung des Programmierers.

Das heißt aber natürlich nicht, dass man einfach wahllos Leerzeilen einfügen sollte.

Plenken

Dies ist ein allgemeines Problem der Formatierung von Textdateien und primär für Kommentare relevant. Die Regeln für das Schreibmaschinenschreiben sind seit jeher eindeutig:

```
Gehört vor Komma, Ausrufezeichen oder Fragezeichen
ein Leerzeichen; oder gar vor das Semikolon? Nein!
```

Das Verstoßen gegen diese Regel heißt im Internet-Jargon ‹plenken›.

Der Fehler rührt wohl daher, dass es im qualifizierten Textsatz durchaus üblich, ja sogar empfehlenswert ist, manche Satzzeichen mit einem *schmaleren* Zwischenraum abzusetzen. Wenn das aber nicht möglich ist, wie in gewöhnlichen Textdateien, oder auch nicht konsequent durchführbar ist, wie mit gewöhnlichen Office-Programmen, setzt man überhaupt kein Leerzeichen.

Das kann man bei Bedarf auch jederzeit im Duden nachlesen.

Glaubenskriege

Bei vielen der beschriebenen Fragen hängt die persönliche Präferenz sehr davon ab, wieviel Erfahrung man hat und in welchem Umfeld man programmieren gelernt hat.

Folgerichtig verstricken sich Programmierer, die das nicht sehen oder nicht sehen wollen, immer wieder in ebenso end- wie fruchtlose Auseinandersetzungen darüber, welches Vorgehen das einzig richtige sei.

Höflichkeit

Welche Position man zum Indenting auch immer bezieht, es gibt ein paar übergeordnete Regeln, an die man sich in jedem Fall halten sollte.

Wenn man eine Datei bearbeitet, schaut man zunächst sorgfältig nach, welchen Stil sie verwendet. Dann gibt es eigentlich nur zwei Möglichkeiten:

- Man hält sich genau daran. Dabei muss man natürlich aufpassen, dass der Texteditor nicht doch wieder automatisch umformatiert.

- Man formatiert die ganze Datei neu. Das geht natürlich nur, wenn der frühere Autor kein Interesse mehr am Code hat. In einem laufenden Projekt ist es einigermaßen unhöflich (und wieder: auf den Texteditor aufpassen!)

Wenn man Code zu einem Open-Source-Projekt beiträgt, sollte man natürlich nichts am Indenting ändern, sondern sich sorgfältig an den originalen Stil halten. Andernfalls macht man dem Maintainer unnötige Arbeit.

Sorgfalt und Konsistenz

Man sollte sich außerdem für einen Stil entscheiden und ihn innerhalb einer Datei konsistent verwenden.

Das heißt nicht unbedingt, dass man an allen Stellen strikt dasselbe machen muss. Es kann durchaus sinnvoll sein, etwa einen ganz langen Ausdruck anders zu formatieren als einen ganz kurzen.

Nur sollte man sich dann wieder bei allen langen und bei allen kurzen Ausdrücken an jeweils ein Prinzip halten.

In einem Projekt

Wenn viele Leute an einem Projekt beteiligt sind, werden auch viele Vorstellungen über Indenting aufeinander stoßen. Das kann man auf verschiedene Art lösen:

- Der Projektleiter bestimmt, was gemacht wird.
- Jede Gruppe einigt sich auf einen Stil. Bei einer geringeren Anzahl von Leuten ist eine Einigung schon eher möglich.
- Jeder macht, was er will, beachtet aber die Regeln zur Textformatierung, zur Höflichkeit und zur Konsistenz.

Zusammenfassung: Indenting

- Stil für Codeblöcke (Kernighan und Ritchie, Allman, Whitesmiths, GNU
- Spalten pro Einrückung
- Sekundäres Indenting
- Tabulatoren und Leerzeichen
- Präzedenz vs. Klammerung
- Leerzeichen bei Klammern, arithmetischen Ausdrücken, Minuszeichen, Semikolons, Schlüsselwörtern, und allgemein
- Leerzeilen zwischen Statements
- Plenken

Literatur

Die Darstellung des Indenting in *C*-ähnlichen Sprachen folgt dem *Jargon File 4.4.7,* <http://www.jargon.org>.