Richtlinie zur Software-Entwicklung in der Programmiersprache C

"Ein gut geschriebenes Programm hat ein sauberes Layout, verwendet sinnvolle Namen, ist ausführlich kommentiert und verwendet Konstrukte der Sprache derart, dass maximale Sicherheit und Lesbarkeit des Programmes erreicht werden. Die Erstellung eines solchen Programms erfordert vom Programmierer Sorgfalt, Disziplin und ein gutes Stück handwerklichen Stolz."

Ian Sommerville, "Software Engineering", Addison-Wesley Verlag 1987

Diese Richtlinie beschreibt Konventionen für die Programmierung in der Programmiersprache C entsprechend dem ANSI C-Standard[1]. Konventionen sind Regeln, an die sich der Programmierer freiwillig hält, deren Einhaltung aber nicht vom Compiler erzwungen wird. Die Anwendung der Regeln soll dazu beitragen, die Qualität neu erstellter Software hinsichtlich folgender Kriterien zu erhöhen:
Lesbarkeit, Überprüfbarkeit, Zuverlässigkeit, Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit

Inhaltverzeichnis

Die Kapitel 1 bis 9 enthalten Regeln, die schon in einfachen Programmen in der elementaren Programmier-Ausbildung beachtet werden. Das Kapitel 10 enthält entsprechende ergänzende bzw. zusätzliche Regeln, die vor allem bei der Erstellung größerer Programme, die aus mehreren Modulen bestehen, von Bedeutung sind. Nach einer Regel folgt in den meisten Fällen noch eine kurze Erläuterung bzw. Begründung.

Einige Entscheidungen zum Layout wurden willkürlich getroffen. Für die Einrücktiefe bei Verschachtelungen hätte z.B. auch ein anderer Wert festgelegt werden können. Die Einheitlichkeit steht in solchen Fällen im Vordergrund, da sie die Lesbarkeit verbessert. Deshalb wird vorgeschlagen, bei der Modifizierung eines schon vorhandenen Programms dessen (u.U. als schlechter empfundenen) Stil zu übernehmen, als verschiedene Stile zu vermischen.

Ein Standard ist nur dann nützlich, wenn er konsequent eingehalten wird. Deshalb die folgende Bitte: Wenn Sie mit einer Regel nicht einverstanden sind, ignorieren Sie sie nicht, sondern sprechen Sie einen der Autoren an und lassen sich vom ihm überzeugen (oder umgekehrt!).

Version: 2.0, erstellt am: 26.04.2007
von: Prof. Dr. Peter Bathelt (peter.bathelt@fh-nuernberg.de)
Prof. Dr. Helmut Herold (helmut.herold@fh-nuernberg.de)
Prof. Dr. Peter Jesorsky (peter.jesorsky@fh-nuernberg.de)
Prof. Dr. Bruno Lurz (bruno.lurz@fh-nuernberg.de)
für: Studenten und Mitarbeiter des Fachbereichs
Elektro-, Feinwerk- und Informationstechnik (FB efi) der
Georg-Simon-Ohm-Fachhochschule Nürnberg (GSO-FH-Nürnberg)

Nürnberg, den 26.04.2007

[1] Die Zusammenstellung der Regeln basiert auf der langjährigen Erfahrung der Autoren mit der Programmiersprache C in Lehre und Praxis. Sie orientiert sich auch an Empfehlungen verschiedener Autoren in öffentlich zugänglichen Quellen.

1 Aufbau von C-Quellcode-Dateien

R11 **Eine übersetzbare .c-Datei besitzt einen Dateikopf mit mindestens den folgenden Angaben:**

```
/*********************************************************************\
* Kurzbeschreibung:
*
* Datum:     Autor:
* <<DATUM>>  <<AUTOR>>
*
\*********************************************************************/
```

R12 **Eine .h-Datei (Headerdatei oder "include"-Datei) enthält keine Variablen- und Funktions-Definitionen. Sie besitzt einen Dateikopf mit mindestens den folgenden Angaben:**

```
/*********************************************************************\
* Kurzbeschreibung:
*
* Datum:     Autor:
* <<DATUM>>  <<AUTOR>>
*
\*********************************************************************/
```

R13 **In `#include`-Anweisungen werden .h-Dateien nicht mit absoluten Pfadnamen bezeichnet.**

Damit wird verhindert, dass jede Verschiebung der .h-Dateien in ein anderes Verzeichnis automatisch eine Anpassung der absoluten Pfadnamen in den entsprechenden #include-Anweisungen der .c-Dateien nach sich ziehen muss.

2 Kommentare

R21 **Kommentare enthalten zusätzliche Informationen, die nicht direkt aus dem Programmcode hervorgehen. Kein Kommentar ist besser als ein sinnloser oder gar falscher Kommentar.**

R22 **Ein kurzer Kommentar zur Erläuterung der Bedeutung einer Variablen oder einer Anweisung wird der Variablen-Definition bzw. der Anweisung in derselben Zeile nachgestellt.**

Die Bedeutung von Variablen oder einzelnen Anweisungen wird auf diese Weise übersichtlich und eindeutig dokumentiert.

R23 **Ein längerer Kommentar wird einer Anweisung oder einer längeren Folge von logisch zusammengehörenden Anweisungen, die kommentiert werden sollen, vorangestellt. Er hat die gleiche Einrücktiefe wie die nächste Anweisung und wird durch eine Leerzeile vom vorherigen Code abgesetzt.**

Auf diese Weise werden Folgen von logisch zusammengehörenden Anweisungen als logisch zusammengehörend gekennzeichnet und ihre Bedeutung als Ganzes dokumentiert.

Beispiele zu den Regeln **R21** bis **R23** sind:

```
int main( void )
{
   int temper;    /* Jeweils aktueller Temperaturwert */

   Anweisung      /* Einlesen des Sollwerts */

   /* Berechnung und Ausgabe des Korrekturwerts */
   Anweisung(en)

   return 0;
}
```

R24 **Kommentare werden nicht geschachtelt.**

Geschachtelte Kommentare werden nicht von allen C-Compilern unterstützt und sind auch schwer lesbar.

3 Formatierung

R31 **Eine Zeile enthält maximal 72 Zeichen und höchstens eine Anweisung. Eine umgebrochene Anweisung wird in der Fortsetzungszeile zusätzlich eingerückt.**

Eine Zeile mit 72 Zeichen ist bei der üblichen Zeilenbreite von 80 Zeichen je Zeile vollständig am Bildschirm darstellbar und damit auf einen Blick erfassbar. Zusätzlich wird sichergestellt, dass Zeilen beim Ausdrucken auch jeweils in einer Zeile gedruckt werden können und nicht umgebrochen oder abgeschnitten werden. Durch die Beschränkung auf eine Anweisung pro Zeile und das zusätzliche Einrücken wird die Lesbarkeit verbessert und die Möglichkeit geschaffen, im Quelltext-Debugger jede Anweisung einzeln auszuführen.

R32 **Anweisungsblöcke werden durch Einrückungsebenen gruppiert. Zusammengehörige blockbegrenzende geschweifte Klammern stehen untereinander in der gleichen Spalte.**

Das Fehlen einer Klammer stellt einen häufigen Programmierfehler dar, der durch das vorgeschlagene Layout besonders schnell gefunden und beseitigt werden kann.

R33 **Eine öffnende geschweifte Klammer hat die gleiche Einrückungsebene wie die Zeile davor. Nach einer solchen Klammer wird eingerückt.**

So *wird die logische Zugehörigkeit einer Block-Anweisung zur vorangehenden Zeile betont und das Erfassen des Umfangs eines Blockes auf einen Blick erleichtert.*

R34 **Einzelanweisungen nach if, else, for, while, do stehen eingerückt in einer separaten Zeile. Wenn sich die Einzelanweisung über mehrere Zeilen erstreckt, werden { } eingesetzt.**

R35 **Eingerückt wird immer um 3 Leerzeichen. Tabulatoren sind verboten. (Optionen von Editoren, die mehrere Leerzeichen in Tabulatoren umwandeln, werden ausgeschaltet).**

Das Verbot von Tabulatoren stellt eine einheitliche Darstellung von C-Quellcode in jedem Editor sicher.

R36 **Nach einem Komma oder Semikolon folgt stets ein Leerzeichen oder eine neue Zeile.**

R37 **Alle binären Operatoren, außer "." und "->", werden von ihren Operanden durch ein Leerzeichen getrennt.**

Damit soll eine gute Lesbarkeit sichergestellt werden.

R38 **Vereinbarungs- und Anweisungsteil eines Funktionsblocks werden durch eine Leerzeile getrennt.**

In C dürfen Vereinbarungen und Anweisungen (anders als in C++) nicht vermischt werden. Das Ende des Vereinbarungsteils und der Beginn des Anweisungsteils sollten deshalb gut zu erkennen sein.

R39 **In Zeigerdefinitionen werden Grundtyp und Zeigeroperator durch Leerzeichen getrennt:**

`char *p1, *p2; /* richtig */`	`char* p1, p2; /* falsch */`

Beispiele zu den Regeln **R31** bis **R37** sind:

```
if (result > 0)
{
   Anweisungen
}
else
{
   Anweisungen
}
```

```
if (result > 0)
   Anweisung
else
{
   Anweisung-
     ueber-mehrere-Zeilen
}
```

```
for (i = 0; i < MAX; i++)
{
   Anweisungen
}
```

```
for (i = 0; i < MAX; i++)
   Anweisung
```

```
while (i < MAX)
{
   Anweisungen
}
```

```
while (i < MAX)
   Anweisung
```

```
do
{
   Anweisungen
} while (i < MAX);
```

```
do
   Anweisung
while (i < MAX);
```

```
switch (command)
{
   case 1:
      Anweisungen
      break;

   case 2: case 3:
      Anweisungen
      break;

   default:
      Anweisungen
      break;
}
```

```
typedef struct
{
   char day;
   char month;
   int year;
} Date;

void delay(long duration)
{
   int i, k;

   Anweisungen
}
```


4 Schreibweise von Namen (Bezeichnern)

R41 **Namen (Bezeichner) beginnen immer mit einem Buchstaben.**
Variablen- und Funktionsnamen beginnen immer mit einem Kleinbuchstaben und enthalten keine Unterstriche. Sie können z.B. mit Groß- und Kleinschreibung gegliedert werden, wie folgendes Beispiel zeigt.

```
newElement = getNextElement();
```

Das erste Zeichen eines Namens muss ein Buchstabe oder Unterstrich sein. Ein führender Unterstrich zeigt an, dass der Name einer Bibliotheksfunktion (z.B. `_itoa()`) zwar vom Compilerhersteller zur Verfügung gestellt wird, aber nicht Teil von ANSI C ist. Deshalb sollten eigene Namen immer mit einem Buchstaben (und nicht mit einem Unterstrich) beginnen.

- *Bezüglich der Groß-/Kleinschreibung hat sich kein einheitlicher Standard durchgesetzt, im K&R-Standard (Kerninghan & Ritchie: "The C Programming Language") werden jedoch Variablen- und Funktionsnamen konsequent klein geschrieben, während `#define`-Konstanten und Makros groß geschrieben werden.*
- *Die Lesbarkeit von Namen, die sich aus mehreren Teilbegriffen zusammensetzen, wird durch zusätzliche Strukturierung verbessert. Dazu wird meist gemischte Groß-/ Kleinschreibung verwendet oder es werden zusätzliche Unterstriche eingesetzt, z.B.:*
  `nextHandle` *bzw.* `next_handle` *statt* `nexthandle`*.*
  Die Autoren sind der Meinung, dass die erste Variante (`nextHandle`*) lesbarer ist, und haben deshalb diese Form in die Richtlinien aufgenommen.*

*Ein Funktionsname ist durch seine Parameterliste leicht erkennbar. Variablennamen können dagegen leichter mit Typnamen verwechselt werden. Aus diesem Grund beginnen Variablennamen stets mit einem Kleinbuchstaben und Typnamen mit einem Großbuchstaben (siehe auch **R45**).*

R42 **Generell werden nichtssagende Faulheitsnamen wie a, b, c, ... vermieden, d.h. es werden sprechende Namen verwendet. Ausnahmen sind Namen wie i, j, k, m, n als lokale Schleifenzähler. Der Name l wird wegen Verwechslungsgefahr mit Ziffer 1 weggelassen!**

Ein Programm ist verständlicher, wenn man die Bedeutung einer Variablen sofort an ihrem Namen erkennen kann. Die Namenswahl für lokale Schleifenzähler kann man mit dem Umkehrschluss begründen: Eine Variable, die keine Bedeutung (über einen längeren Programmabschnitt) hat, soll auch keinen besonderen Namen tragen und jederzeit "wiederverwendbar" sein. Die Wahl der Buchstaben `i, j, ..., n` *für lokale Schleifenzähler hat historische Gründe. In der Ur-Sprache FORTRAN haben alle Variablen, die mit diesen Buchstaben beginnen, implizit den Typ* `int` *und als Schleifenzähler werden ja bevorzugt* `int`*-Variablen eingesetzt.*

R43 **Selbstdefinierte Makronamen und `#define`-Konstanten werden vollständig groß geschrieben. Ziffern und Unterstriche sind erlaubt, aber nicht als führende Zeichen.**

Beispiel: `#define NO_OF_ELEMENTS 10`

Einen Namen, der über `#define` *definiert wird, bekommt der Compiler gar nicht mehr zu sehen. Er wird vorher vom Präprozessor verarbeitet und durch die in der Konstanten-/Makrodefinition angegebene Zeichenfolge ersetzt. Mit diesem Textersatz sind spezifische Gefahren verbunden (siehe auch Begründung zu **R53**). Außerdem spielen Effizienzfragen eine Rolle. Bei jedem Makroaufruf wird der dazugehörige Programmcode neu eingefügt (größerer Speicherbedarf), dafür entfällt die Parameterübergabe und der eigentliche Funktionsaufruf (höhere Geschwindigkeit). Zur besseren Erkennung werden Konstanten und Makros deshalb durch konsequente Großschreibung hervorgehoben*.

R44 **Die Namen der Werte in einer enum-Typvereinbarung werden wie `#define`-Konstantennamen vollständig groß geschrieben.**

Die Werte einer enum-Typvereinbarung können in vielen Fällen mehrere `#define`*-Konstanten ersetzen:*

```
typedef enum {EZERO, ENOPATH = 3, ENMFILE = 18} DosError;
```

ersetzt z.B. die folgenden Konstanten:

```
#define EZERO    0
#define ENOPATH  3
#define ENMFILE 18
```

Da liegt es nahe, die Konstanten-Schreibweise zu übernehmen.

R45 **Selbstdefinierte typedef-Namen beginnen immer mit einem Großbuchstaben. Ziffern und Unterstriche sind erlaubt, aber nicht als führende Zeichen.**

Beispiel: `typedef unsigned short int Word;`

Um einen Typnamen leicht von einem Variablennamen zu unterscheiden, sollten Variablennamen stets mit einem Kleinbuchstaben und Typnamen mit einem Großbuchstaben beginnen.

R46 **Zeigervariablen werden mit dem Präfix p gekennzeichnet, Zeiger auf Zeiger entsprechend mit dem Präfix pp.**

Wird jede Indirektionsebene durch ein Präfix p im Namen gekennzeichnet, wird die Lesbarkeit und die Verständlichkeit von Programmen deutlich verbessert.

R47 **Namen werden in einer Sprache formuliert, z.B. einheitlich in deutsch oder in englisch.**

Der konsequente Einsatz nur einer Sprache in der Namenswahl verbessert das ästhetische Erscheinungsbild des Programms und damit auch die Lesbarkeit. Abgesehen davon wird die Softwareentwicklung zunehmend internationalisiert. Sie müssen also davon ausgehen, dass Sie selbst einmal Programme lesen müssen, die in einer anderen Sprache geschrieben sind. Umgekehrt kann es sein, dass ein von Ihnen geschriebenes Programm später von einem Programmierer einer anderen Muttersprache weiterentwickelt wird. Da die Sprache C (wie fast alle Programmiersprachen) angelsächsische Wurzeln hat und ihre Schlüsselwörter ohnehin in englisch formuliert sind, bietet es sich in den meisten Fällen an, diese Sprache auch für eigene Namen einzusetzen.

5 Definitionen von Konstanten und Makros

R51 **Eine Konstante, die mehr als einmal benutzt wird, erhält über eine Konstantendefinition einen aussagekräftigen symbolischen Namen, der an ihrer Stelle ausschließlich verwendet wird (Ausnahmen: 0 und 1).**

```
#define ANZAHL_TUEREN 5   oder:   const int anzahlTueren = 5;
```

Die Erfahrung zeigt, dass Konstanten (z.B. die Größe eines Puffers) über verschiedene Versionsstände des Programms doch verändert werden! Wird für eine Konstante ein Name definiert, und wird dieser Name dann auch konsequent eingesetzt, dann braucht bei einer Änderung der Konstanten nur eine einzige Programmzeile geändert zu werden.
*Regel **R51** dient also vor allem der Wartbarkeit und Lesbarkeit eines Programms.*

R52 **In einer Makrodefinition wird jeder Parameter und zusätzlich der gesamte Makro geklammert. Stellt der Makro einen Ausdruck dar, sind die äußeren Klammern rund, stellt er eine Anweisung dar, sind sie geschweift. Auf die äußeren Klammern kann nur verzichtet werden, wenn in der Makrodefinition keine Operatoren eingesetzt werden.**

```
#define LOWER  10
#define UPPER  (LOWER + 100)
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define SWAP(x, y, z) { (z) = (x); (x) = (y); (y) = (z); }
#define SQUARE(x) ((x)*(x))
```

Das Problem ist: viele Makroaufrufe funktionieren auch ohne diese Klammerung, sie werden niemals einem "Härtetest" unterzogen. Werden solche Makros aber dann später in einem anderen Programm in anderem Zusammenhang aufgerufen, kommt es zu bösen Überraschungen. Die Problematik lässt sich durch ein einfaches Beispiel verdeutlichen: Sie definieren einen Makro, der die Summe seiner beiden Parameter berechnet, und ignorieren die geforderte Klammerung:

```
#define SUM(n1, n2) n1 + n2
```

Wenn Sie den Makro dann später wie folgt aufrufen:

```
n = SUM(1, 2);
```

wird die Variable n *auch korrekt den Wert* 3 *erhalten. Wenn Sie ihn jedoch in der Form*

```
n = 2 * SUM(1, 2);
```

aufrufen, erwarten Sie natürlich den Wert 6 *für die Variable* n*. Der Präprozessor nimmt jedoch den (im Folgenden immer fett markierten) Textersatz*

```
n = 2 * 1 + 2;
```

vor und der Variablen n *wird der Wert* 4 *zugewiesen. Um diesen Fehler zu eliminieren, müssen wir den Makro insgesamt klammern:*

```
#define SUM(n1, n2) (n1 + n2)
```


Der Präprozessor nimmt dann folgenden Textersatz vor:

```
n = 2 * (1 + 2);
```

und n *erhält den Wert* 6*. Leider ist auch diese Version des Makros noch nicht "sicher", wie der folgende Makroaufruf zeigt:*

```
n = SUM(1, 2 << 1);
```

Wir erwarten für die Variable n *den Wert* 5*, erhalten aber den Wert* 6*, weil der Präprozessor folgenden Textersatz vornimmt:*

```
n = (1 + 2 << 1);
```

und der Operator '+' eine höhere Priorität hat als der Operator '<<'.

Erst bei voller Berücksichtigung der Regel ***R52****:*

```
#define SUM(n1, n2)   ((n1) + (n2))
```

wird der korrekte Wert ermittelt, wie der folgende (endgültige) Textersatz zeigt:

```
n = ((1) + (2 << 1));
```

Der Ersatztext des letzten Beispiel-Makros SUM() *wurde in runde Klammern eingeschlossen, da er einen Ausdruck (und keine Anweisung) darstellt. Fast alle Makros werden als Ausdrücke definiert und sind dadurch universell einsetzbar, weil ein Ausdruck an viel mehr Stellen eines C-Programms auftreten kann als eine Anweisung. Benötigt der Makro dagegen die Definition zusätzlicher Variablen oder besteht er aus mehreren Anweisungen, dann wird der Makro in geschweifte Klammern eingeschlossen und stellt damit einen Block (das ist auch eine Art von Anweisung) dar, in dem zu Beginn zusätzliche Variablen definiert werden können, deren Gültigkeitsbereich auf diesen Block beschränkt ist. Durch die geschweiften Klammern wird sichergestellt, dass der Makro nicht versehentlich als Ausdruck eingesetzt wird.*

R53 Die Argumente von Makro-Aufrufen enthalten keine (direkten oder indirekten) Seiteneffekte wie Inkrement-Operatoren, Zuweisungen und Funktionsaufrufe.

korrekt!	**verboten**!
`x++;` `z = MAX(x, y);`	`z = MAX(x++, y);`
`c = getchar();` `z = MAX(c, y);`	`z = MAX(getchar(), y);`

Enthält ein Makro direkte oder indirekte Seiteneffekte, dann kommt es zu den im Folgenden beschriebenen Programmfehlern.

Wird der schon in Regel ***R52*** *eingesetzte Makro*

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

in der verbotenen Form

```
z = MAX(x++, y);
```

*mit einem Seiteneffekt (*x++*) aufgerufen, dann erzeugt der Präprozessor den (im Folgenden fett dargestellten) Ersatztext:*

```
z = ((x++) > (y) ? (x++) : (b));
```

Wenn die Bedingung x++ > y *erfüllt ist, wird* x *zweimal inkrementiert!*

Ein (nicht ganz so offensichtlicher) Seiteneffekt liegt auch beim Aufruf von getchar() vor, weil dadurch der Inhalt des Tastaturpuffers verändert wird. Der verbotene Aufruf

```
z = MAX(getchar(), y);
```

wird vom Präprozessor zu folgendem Ersatztext expandiert:

```
z = ((getchar()) > (y) ? (getchar()) : (y));
```

Wenn das erste mit getchar() *eingelesene Zeichen größer als* y *ist, dann wird ein zweites Zeichen mit* getchar() *eingelesen und an* z *zugewiesen. Das erste eingelesene Zeichen wird dabei einfach "verschluckt"!*

6 Definitionen von Datentypen

R61 **Ein neuer `struct`-, `union`- und `enum`-Datentyp wird immer mit dem Sprachelement `typedef` definiert, selbst wenn nur eine einzige Variable dieses Typs erzeugt wird.**

Ein Strukturdatentyp kann mit einem **`tag`***- und/oder* `typedef`*-Namen vereinbart werden. Der* **`tag`***-Name muss aber (anders als in C++) immer zusammen mit dem Schlüsselwort* **`struct`** *eingesetzt werden, während der* **`typedef`***-Name allein den Datentyp vollständig beschreibt.*

R62 **Bei der Definition eines `struct`-Datentyps kann zusätzlich zum `typedef`-Namen ein `tag`-Name angegeben werden. Notwendig ist dies jedoch nur bei der Definition eines rekursiven Datentyps. Der `tag`-Name wird aus dem `typedef`-Namen durch Voranstellen von `tag` gebildet.**

nur `typedef`-Name: **korrekt**!	`typedef`- und tag-Name: **korrekt**, aber `tag`-Name wird nie eingesetzt (siehe **S63**)!	nur `tag`-Name: **verboten**!

`typedef struct`
`{`
`  char day;`
`  char month;`
`  int year;`
`} Date;`
	`typedef struct tagDate`
	`{`
	`  char day;`
	`  char month;`
	`  int year;`
	`} Date;`
		~~`typedef struct tagDate`~~
		~~`{`~~
		~~`  char day;`~~
		~~`  char month;`~~
		~~`  int year;`~~
		~~`};`~~

`typedef`- und `tag`-Name: **korrekt**, `tag`-Name notwendig!

```
typedef struct tagListElem
{
   char *pInfo;
   struct tagListElem *pNext;
} ListElem;
```

Bei einem "rekursiven" Datentyp muss man (nur einmalig bei Typvereinbarung) den tag-Namen einsetzen und gibt ihm bewusst einen "gespreizten" Namen, damit er im Folgenden nicht eingesetzt wird. Im letzten Beispiel muss für die Definition der Strukturkomponente `pNext` *der tag-Name verwendet werden, da der* `typedef`*-Name noch nicht bekannt ist. Wird später eine Zeigervariable dieses Typs benötigt, ist man aber nicht mehr auf den tag-Namen angewiesen:*

```
ListElem *pHead;
```

R63 **Nach abgeschlossener Typdefinition wird bei der Definition von Strukturvariablen und Zeigern ausschließlich der `typedef`-Name und nicht `struct tag`- verwendet.**

`typedef`-Name: **korrekt**!	`tag`-Name: **verboten**!
`Date birthday;` `ListElem *pFirst;`	~~`struct tagDate birthday;`~~ ~~`struct tagListElem *pFirst;`~~

Hinweis: Die Regeln für den `struct`-Datentyp gelten entsprechend für `union`- und `enum`-Datentypen.

7 Definitionen von Funktionen und Prototypen

R71 **Der Ergebnistyp einer Funktion wird immer explizit angegeben, auch der Datentyp `int`. Dies gilt ebenfalls für das Hauptprogramm `main()`, das nach dem ANSI C-Standard den Ergebnistyp `int` hat.**

ANSI C lässt zu, dass man den Ergebnistyp einer Funktion weglässt, und nimmt in diesem Fall den Typ `int` *an. Diese Schreibweise erweckt aber leicht den falschen Eindruck, dass nichts (also der Typ* `void`*) zurückgegeben wird, und wird deshalb nicht verwendet.*

R72 **Bei Funktionen, die kein Ergebnis liefern, wird als Ergebnistyp explizit `void` angegeben.**

R73 **Bei einer Funktion ohne Parameter wird in die Parameterliste explizit `void` eingetragen. Funktions-Definitionen mit Parametern erfolgen stets gemäß dem ANSI C-Standard, d.h. mit vollständiger Datentypangabe innerhalb der Parameterliste.**

Wird im Prototyp einer Funktion der Parametertyp `void` *verwendet, stellt der Compiler sicher, dass beim Funktionsaufruf tatsächlich kein Aufrufargument vorhanden ist. Bei einem (unvollständigen) Prototyp bedeutet eine leere Parameterliste nicht etwa, dass die Funktion keine Parameter hat, sondern nur, dass man über ihre Parameter nichts weiß! Der Compiler nimmt dann beim Funktionsaufruf keine Typprüfung und nur eine eingeschränkte Typanpassung vor. Unvollständige Prototypen werden deshalb nicht eingesetzt! Da der Prototyp einer Funktion mit der Funktionsdefinition übereinstimmen sollte, verwenden wir* `void` *auch bei der Definition.*

8 Definitionsort und Geltungsbereich

R81 **Definitionen von Variablen und Funktionen stehen immer in .c-Dateien und nicht in .h-Dateien.**

.h-Dateien können in mehrere verschiedene C-Dateien inkludiert werden. Wären Variablen und Funktionen in solchen .h-Dateien definiert, so würden sie in jeder der .c-Dateien neu und damit insgesamt mehrfach definiert werden. Dies kann zu verschiedenen Problemen führen.

R82 **Variablen, die nur innerhalb einer Funktion Verwendung finden, werden in dieser Funktion definiert.**

Variablen sollten nur soweit sichtbar sein, wie sie benötigt werden. Dies dient zur Vermeidung von fremden, nicht erwünschten Zugriffen und von Konflikten bei der Namensgebung (information hiding). Außerdem gehören lokale Variablen defaultmäßig der Speicherklasse `auto` *an und belegen Speicherplatz nur für die Dauer des Funktionsaufrufs.*

9 Verwendung von Sprachelementen

R91 **Jeder Block besitzt nur einen Ausgang.**
- Innerhalb einer Schleife werden `return`- und `break`-Anweisungen vermieden.
- Eine Funktion besitzt maximal eine `return`-Anweisung am Ende.

Begründete Ausnahmen sind zulässig.

Ein einziger Ausgang erleichert die Lesbarkeit und Wartbarkeit von und auch die Fehlersuche in Funktionen. Ein einziger Haltepunkt (Breakpoint) reicht aus, um das Programm unmittelbar vor dem Abschluss der Funktion zu unterbrechen. Ausnahmen sind für eine effiziente Fehlerbehandlung zugelassen, wobei dann aber nur ein zusätzlicher Fehlerausgang hinzugefügt werden sollte. Diese Regel hilft auch, Fehler zu vermeiden, die eine Folge nicht durchgeführter Aufräumaktionen sein können, wie im folgenden Beispiel demonstriert ist:

korrekt!

```
{
   ...
   ptr = (int *)malloc(...);
   if (test == 0)
   {
      ...
      status = TRUE;
   }
   else
                  status = FALSE;
   free(ptr);
   return status;
}
```

verboten!

```
{
   ...
   ptr = (int *)malloc(...);
   if (test == 0)
   {
      ...


   }
   else
      return FALSE;
   free(ptr);
   return TRUE;
}
```

Im `else` *Zweig der verbotenen Code-Sequenz wird durch den frühzeitigen Rücksprung die Freigabe des allokierten Speichers "vergessen".*

R92 **`goto` wird nur bei Fehlerausgängen oder in anderen speziellen Situationen verwendet und durch einen Kommentar begründet.**

Sprünge in Programmen erschweren die Lesbarkeit erheblich.

R93 **`continue` wird nur in begründeten Ausnahmesituationen verwendet.**

`continue` *wirkt ähnlich wie ein Sprungbefehl mit goto.*

R94 **In Ausdrücken darf die Auswertungsreihenfolge der Operanden keinen Einfluss auf das Ergebnis haben. So darf eine Variable, die inkrementiert oder dekrementiert wird, im selben Ausdruck und im selben Funktionsaufruf nicht noch einmal verwendet werden.**

korrekt!	**verboten**! (compilerabhängig!)
`result = f(x);` `result = result + g(&x);`	`result = f(x) + g(&x); (Anweisung-1)`
`++x;` `result = f(x, g(x));`	`result = f(++x, g(x)); (Anweisung-2)`
`arr[i] = i;` `i++;`	`arr[i] = i++; (Anweisung-3)`

In ANSI C ist die Auswertungsreihenfolge von Operanden in Ausdrücken nicht festgelegt. In Anweisung-1 (von "verboten!") hängt es vom Compiler ab, ob zuerst `f(x)` *und danach* `g(&x)` *oder zuerst* `g(&x)` *und dann* `f(x)` *aufgerufen wird. Wird* `x` *aber in* `g()` *verändert (wegen Übergabe von* `&x` *möglich), so wird* `f(x)` *in den zwei Fällen mit jeweils unterschiedlichem Wert für* `x` *aufgerufen, was i. Allg. zu unterschiedlichen Ergebnissen führen wird. Sei in Anweisung-2 (bei "verboten!")* `x = 5`*, so kann der Aufruf von* `f()` *entweder mit* `f(6,g(6))` *oder auch mit* `f(6,g(5))` *erfolgen. Ebenso ist in Anweisung-3 (bei "verboten!") sowohl* `arr[6] = 5` *als auch* `arr[5] = 5` *möglich.*

R95 **Jeder `case`-Zweig einer `switch`-Anweisung, der Anweisungen enthält, ist durch `break` abzuschließen. Der `default`-Zweig wird immer vorgesehen und ebenfalls durch `break` abgeschlossen. Wenn der `default`-Fall auftreten darf und keine Fehlersituation darstellt, bleibt der `default`-Zweig leer; andernfalls enthält er eine Fehlerbehandlung.**

Bei einem fehlenden ***`break`*** *entsteht die Gefahr, dass beim späteren Einfügen eines weiteren Falles die Ergänzung einer vorigen* ***`break`****-Anweisung übersehen wird und damit ein schwer auffindbarer Fehler entsteht. Ein* ***`default`****-Zweig mit einer Fehlerbehandlung hat im Fehlerfall gegenüber einem fehlenden* ***`default`****-Zweig den Vorteil, dass ein logischer Fehler (kein Treffer in den* `case`*-Zweigen) sofort erkannt wird.*

10 Ergänzende Regeln für größere Programme und Programme mit mehreren Modulen

Die Unterkapitel 10.1 bis 10.9 enthalten zusätzliche Regeln zu den korrespondierenden Kapiteln 1 bis 9. Da es zu den Kapiteln 3 und 9 keine zusätzlichen Regeln gibt, bleiben die entsprechenden Unterkapitel 10.3 und 10.9 leer.

10.1 Aufbau von C-Quellcode-Dateien

R111 **Eine übersetzbare .c-Datei besitzt den folgenden Aufbau:**

```
/*********************************************************************\
* Dateiname: <<dateiname.c>>
* Autor    : <<AUTOR>>
* Projekt  : <<PROJEKT>>
* Copyright (C) <<COPYRIGHT>>
*
* Kurzbeschreibung: Diese Datei zeigt den prinzipiellen Aufbau
* einer .c-Datei, der den C-Programmier-Regeln des Fachbereichs
* Elektro-, Feinwerk- und Informationstechnik (FB efi) der
* Georg-Simon-Ohm-Fachhochschule Nuernberg (GSO-FH-Nuernberg) genuegt.
*
* Datum:     Autor:       Grund der Aenderung:
* <<DATUM>>  <<AUTOR>>    Neuerstellung
* <<DATUM>>  <<AUTOR>>    <<AENDERUNGSGRUND>>
*
\*********************************************************************/

/*--- #includes der Form <...> ----------------------------------------*/

/*--- #includes der Form "..." ----------------------------------------*/

/*--- #define-Konstanten und Makros -----------------------------------*/

/*--- Datentypen (typedef) --------------------------------------------*/

/*--- Globale Konstanten ----------------------------------------------*/

/*--- Globale Variablen -----------------------------------------------*/

/*--- Modullokale Konstanten ------------------------------------------*/

/*--- Modullokale Variablen -------------------------------------------*/

/*--- Prototypen modullokaler Funktionen ------------------------------*/

/*--- Funktionsdefinitionen -------------------------------------------*/

int main(int argc, char *argv[])   /* loeschen, falls nicht benoetigt */
{
   return 0;
}
```

Werden bestimmte Programmelemente (z.B. globale Variablen) nicht verwendet, kann der betreffende Abschnitt entfallen. Die Reihenfolge der Angaben wird jedoch strikt eingehalten.

Die Prototypen globaler Funktionen fehlen, da sie in `.h`-Dateien hinterlegt werden und bei Bedarf über `.h`-Dateien einzufügen sind.

Hinweis 1: Die Begriffe **Deklaration** und **Definition** werden hier im üblichen Sinne verwendet. Bei der Definition einer Variablen oder einer Funktion belegt der Compiler Speicherplatz für diese Variable oder für den Code dieser Funktion; bei einer Deklaration wird lediglich auf die Existenz der Variablen oder der Funktion verwiesen.

Hinweis 2: Wird die Compilierung von .h-Dateien durch `#define`-Konstanten gesteuert, werden diese abweichend von **R111** vor den `#include`-Anweisungen der .h-Dateien definiert.

R112 **Eine .h-Datei (Headerdatei oder "include"-Datei) enthält keine Variablen- und Funktions-Definitionen. Sie besitzt den folgenden Aufbau:**

```
/*********************************************************************\
* Dateiname: <<dateiname.h>>
* Autor    : <<AUTOR>>
* Projekt  : <<PROJEKT>>
* Copyright (C) <<COPYRIGHT>>
*
* Kurzbeschreibung: Diese Datei zeigt den prinzipiellen Aufbau
* einer .h-Datei, der den C-Programmier-Regeln des Fachbereichs
* Elektro-, Feinwerk- und Informationstechnik (FB efi) der
* Georg-Simon-Ohm-Fachhochschule Nuernberg (GSO-FH-Nuernberg) genuegt.
*
* Datum:     Autor:       Grund der Aenderung:
* <<DATUM>>  <<AUTOR>>    Neuerstellung
* <<DATUM>>  <<AUTOR>>    <<AENDERUNGSGRUND>>
*
\*********************************************************************/

#ifndef <<DATEINAME_H>>
#define <<DATEINAME_H>>

/*--- #includes der Form <...> ----------------------------------------*/

/*--- #includes der Form "..." ----------------------------------------*/

/*--- #define-Konstanten und Makros -----------------------------------*/

/*--- Datentypen (typedef) --------------------------------------------*/

/*--- Globale Konstanten (extern const) -------------------------------*/

/*--- Globale Variablen (extern) --------------------------------------*/

/*--- Prototypen globaler Funktionen ----------------------------------*/

#endif /*<<DATEINAME_H>>*/
```

Bei `#include`*-Anweisungen werden System-*`.h`*-Dateien vor eigenen* `.h`*-Dateien eingefügt. Dies dient lediglich der Übersichtlichkeit, denn, falls Vereinbarungen aus System-.h-Dateien in folgenden* `.h`*-Dateien benötigt werden, sollten diese System-*`.h`*-Dateien über entsprechende* `#include`*-Anweisungen bereits in solche* `.h`*-Dateien eingefügt werden.*

R113 **Eine .h-Datei kann alternativ zu Regel R112 im Anschluss an den Datei-Kommentarkopf und die sich daran gegebenenfalls anschließenden #include-Anweisungen in einzelne logisch zusammenhängende Abschnitte unterteilt werden.**
In jedem dieser Abschnitte stehen dann die folgenden Angaben:

- #define-Vereinbarungen
- Definitionen von Datentypen
- Deklarationen globaler Konstanten (extern const ...) und globaler Variablen (extern ...)
- Deklarationen von Prototypen zu globalen Funktionen

R114 **Jede .h-Datei datei.h schützt sich durch folgenden Rahmen vor dem mehrfachen Einfügen in ein und dieselbe .c-Datei:**

```
#ifndef DATEI_H
#define DATEI_H
   /* eigentlicher Inhalt der .h-Datei datei.h */
#endif /* DATEI_H */
```

Außerhalb des `#ifndef ... #endif`- Bereichs steht nur Kommentar.

Durch den Rahmen wird ggfs. das mehrfache Einfügen einer `.h`*-Datei während eines Übersetzungsvorgangs verhindert und der Übersetzungsvorgang beschleunigt.*

10.2 Kommentare

R121 **Jeder Funktions-Definition (außer `main()`) wird ein Funktions-Kommentarkopf vorangestellt. Ein Funktions-Kommentarkopf enthält mindestens folgende Angaben:**

- Name der Funktion
- Kurzbeschreibung der Funktion einschließlich der aus besonderen Gründen verwendeten globalen und/oder modullokalen Variablen
- Erläuterung der Parameter, falls vorhanden
- Erläuterung des Funktionswerts, falls die Funktion ein Ergebnis zurückliefert

Muster für den Aufbau eines "Funktionskopfes"

```
/*****************************************************************\
* Funktionsname: <<FUNKTIONSNAME>>
*
* Kurzbeschreibung: Dieser Kommentar zeigt den prinzipiellen Aufbau
* eines Funktionskopfes, der den C-Programmier-Regeln des Fach-
* bereichs Elektro-, Feinwerk- und Informationstechnik (FB efi) der
* Georg-Simon-Ohm-Fachhochschule Nuernberg (GSO-FH-Nuernberg) genuegt.
*
* Parameter:
* <<PARAMETER1>> : <<BEDEUTUNG>>
* <<PARAMETER2>> : <<BEDEUTUNG>>
*
* Rueckgabewert: <<BEDEUTUNG>>
* <<WERT1>> : <<BEDEUTUNG EINES SPEZIELLEN WERTES, FALLS NOTWENDIG>>
* <<WERT2>> : <<BEDEUTUNG EINES SPEZIELLEN WERTES, FALLS NOTWENDIG>>
*
\*****************************************************************/
```

Auf diese Weise werden Funktionsweise und die Schnittstelle einer Funktion mit ihrem Eingangs-/Ausgangsverhalten gut erkennbar, systematisch und übersichtlich im C-Quellcode beschrieben.

10.3 Formatierung

Dieses Unterkapitel enthält keine zusätzlichen Regeln.

10.4 Schreibweise von Namen (Bezeichnern)

R141 **Variablen können in Anlehnung an die "Ungarische Notation" benannt werden. Bei dieser Schreibweise wird jeder Variablen ein Präfix vorangestellt, der den Datentyp der Variablen beschreibt:**

- Regeln für die Schreibweise <Typ-Präfix><Name>:
  - ⇒ Präfix grundsätzlich klein.
  - ⇒ Erster Buchstabe des Namens groß.
  - ⇒ Bei zusammengesetzten Namen mit Groß-/Kleinschreibung gliedern.
  - ⇒ Keine Unterstriche verwenden.
- Als Standard-Datentyp-Präfixe können z.B. verwendet werden:
  - ⇒ `char` `c`
  - ⇒ `short int` `s`
  - ⇒ `int` `i oder n`
  - ⇒ `long int` `l`
  - ⇒ `pointer` `p`
  - ⇒ `function pointer` `pfn`

Beispiele:

```
char cTest, *pcXY;
short int sIndex, *psNext, **ppsFirst;
```

Trägt eine Variable ihren Datentyp als Präfix im Namen, dann ist es leichter, kleine Programmausschnitte zu erfassen. Man muss nicht erst nach der Definition einer Variablen suchen, um ihren Einsatz zu verstehen. Werden (wie z.B. bei Windows-Programmen) Bibliotheken eingesetzt, die selbst die "Ungarische Notation" benutzen, dann liegt es nahe, diesen Stil einheitlich auch bei eigenen Namen zu verwenden.

Hinweis: In großen Projekten sollten die Namen globaler Funktionen und Variablen durch ein Präfix gekennzeichnet werden. Dieser benennt dann die .c-Datei oder einen aus mehreren .c-Dateien bestehenden Funktionsbaustein, in dem sie definiert sind.

10.5 Definitionen von Konstanten und Makros

R151 **Für nichtstandardisierte, compilerspezifische Schlüsselwörter wie `__near`, `__far`, `__pascal` werden - wenn sie benötigt werden - eigene Bezeichner definiert und verwendet.**

Beispiel: `#define FAR _ _far`

Mit der obigen Definition der Konstanten kann der gleiche Quellcode für verschiedene Zielsysteme verwendet werden.

Wird z.B. ein C-Programm für den PC im Real Mode des 80x86-Prozessors mit dem Speichermodell "Small" entwickelt und in diesem Programm eine 32-Bit-Zeigervariable benötigt, dann kann diese mit den Vereinbarungen

```
#define FAR __far

char FAR *ptr;     /* entspricht : char __far *ptr; */
```

erzeugt werden. Bei einem Compiler für einen anderen Prozessor, der das Schlüsselwort `__far` *nicht kennt, wird einfach eine andere* `#define`*-Anweisung eingesetzt und das unbekannte Schlüsselwort* `__far` *"weggezaubert":*

```
#define FAR

char FAR *ptr;     /* entspricht:  char *ptr; */
```

10.6 Definitionen von Datentypen

R161 **Bei der Entwicklung portabler Software können für die Grunddatentypen zusätzliche eigene Datentypnamen definiert werden. Solche Zusatzdefinitionen sind immer mit `typedef` durchzuführen und dann ausschließlich zu verwenden.**

Der ANSI C-Standard schreibt nicht vor, wie viele Bits z.B. der Datentyp int *hat. Am gebräuchlichsten sind z.B. im PC-Bereich Compiler, die für diesen Typ 16 oder 32 Bit vorsehen. Wird der Datentyp* int *jetzt für 16-Bit- und 32-Bit-Systeme in unterschiedlicher Weise umdefiniert:*

```
#if sizeof(int) == 2 /* hier nur Konstanten erlaubt! */
   typedef int I2;
   typedef long int I4;
#else
   typedef short int I2;
   typedef int I4;
#endif
```

und werden ausschließlich die neuen Typnamen verwendet,

```
I2 m;
I4 n;
for (m = 0; m < 10000; ++m)
for (n = 0; n < 100000; ++n)
```

dann kann ein Quellprogramm ohne Änderungen für verschiedene 16/32-Bit-Zielsysteme eingesetzt werden und es wird sichergestellt, dass einerseits keine Überläufe auftreten und andererseits keine "unnötig" großen Datentypen eingesetzt werden.

R162 **Der "nackte" Datentyp `char` wird nur dann verwendet, wenn es keine Rolle spielt, ob `char` als vorzeichenlos oder -behaftet interpretiert wird. Sonst wird immer `signed char` bzw. `unsigned char` eingesetzt oder nach R161 ein eigener, eindeutiger Typname definiert.**

Der "nackte" Datentyp `char` *ist nicht portabel, da er von manchen Compilern als* `signed char`*, von anderen als* `unsigned char` *interpretiert wird. Wird eine* `char`*-Variable* `c` *z.B. als Index eingesetzt -* `arr[c]` *- dann kann bei vorzeichenbehaftetem Datentyp* `char` *auf ein nicht vorhandenes Arrayelement zugegriffen werden.*

10.7 Definitionen von Funktionen und Prototypen

R171 **Ein an eine Funktion mit Hilfe eines Zeigers übergebenes Objekt (z.B. eine Struktur oder ein Array) wird gegen unbeabsichtigte Modifikation durch `const` geschützt.**

Beispiel: `void printDate(const Date *pDate);`

Bei versehentlicher Änderung des übergebenen Objekts erzeugt schon der Compiler eine Fehlermeldung, so dass der Fehler frühzeitig beseitigt werden kann. Ohne Verwendung von

`const` würde sich der Fehler erst zur Laufzeit des Programms bemerkbar machen und die Fehlerbeseitigung viel mehr Zeit in Anspruch nehmen.

R172 **Zu jeder Funktion existiert ein Prototyp gemäß dem ANSI C-Standard. Der Prototyp einer globalen Funktion steht in einer .h-Datei. Der Prototyp einer modullokalen Funktion wird in der definierenden .c-Datei als `static` deklariert.**

Über Prototypen kann der Compiler Typprüfungen und implizite Typanpassungen bei Funktionsaufrufen durchführen. Durch Prototypen in .h-Dateien wird die Konsistenz der Funktionsschnittstellen in allen Dateien und bei allen Aufrufen gewährleistet.

*Modullokale Funktionen werden wegen des Prinzips des "information hiding" (Erläuterung siehe Begründung zu Regel **R82**) durch `static` in ihrer Sichtbarkeit auf die Datei beschränkt.*

10.8 Definitionsort und Geltungsbereich

R181 **Variablen und Funktionen, die nur innerhalb einer .c-Datei Verwendung finden, werden in dieser .c-Datei als `static` definiert.**

*Das Schlüsselwort `static` bewirkt eine Verlängerung der Lebensdauer bei lokalen Variablen auf die gesamte Programmlaufzeit und die Beschränkung des Geltungsbereiches von globalen Variablen und Funktionen auf die definierende Datei. Die Forderung, Variablen und Funktionen, die nur innerhalb einer Datei verwendet werden, als `static` zu deklarieren, folgt aus dem allgemeinen Prinzip, Daten und Funktionen nur soweit für andere sichtbar zu machen wie unbedingt notwendig (information hiding, siehe auch Begründung zu Regel **S82**).*

R182 **Datentypen, die nur innerhalb einer .c-Datei Verwendung finden, werden in dieser definiert.**

R183 **`extern`-Bezüge auf globale Variablen und globale Funktionen (d.h. Prototypen) stehen grundsätzlich in Header-Dateien (.h-Dateien) und nicht in .c-Dateien.**

Die Wartbarkeit von `extern`-Bezügen wird gefördert, wenn sie nur an einer einzigen Stelle, nämlich in der .h-Datei, erscheinen.

R184 **Eine .h-Datei mit extern-Bezügen wird auch in die definierende .c-Datei über eine `#include`-Anweisung einbezogen.**

Inkonsistenzen in der Definition und der extern-Deklaration werden durch diese Maßnahme vom Compiler erkannt (z.B. abweichende Datentypen bei Funktionsparametern oder globalen Variablen).

R185 **Werden in einer .h-Datei `dateio.h` Deklarationen aus einer anderen .h-Datei `date.h` verwendet, so wird `date.h` in `dateio.h` mit `#include` (verschachtelt) eingefügt.**

Datei `date.h`

```
...
typedef struct
{
   char day;
   char month;
   int year;
} Date;
...
```

Datei `dateio.h`

```
...
#include "date.h"
void printDate(Date);
...
```

Datei `dateio.c`

```
...
#include "dateio.h"
...
{
   Date today;
...
   printDate(today);
...
}
```

*Eine einzufügende .h-Datei sollte alle ihre Informationen zum erfolgreichen Übersetzen mitbringen. Will man, alternativ zu dieser Regel, verschachtelte `#include`-Anweisungen in .h-Dateien vermeiden, so müssen alle benötigten .h-Dateien einzeln in die .c-Datei eingefügt werden. In diesem Fall muss die Reihenfolge der `#include`-Anweisungen so festgelegt werden, dass .h-Dateien mit bestimmten Deklarationen vor solchen .h-Dateien, die diese Deklarationen benutzen, eingefügt werden. Aus diesem Grund bevorzugen die Autoren die in **R185** festgelegte Vorgehensweise.*

10.9 Verwendung von Sprachelementen

Dieses Unterkapitel enthält keine zusätzlichen Regeln.