

Programmierrichtlinien für die Programmiersprache



(Stand: März 2007)

Die Programmierrichtlinien geben Hinweise zur Gestaltung und Dokumentation der Software und bewirken, dass das Programm einfacher zu lesen, zu testen, zu warten, zu portieren und zu bedienen ist. Solche Regeln sind vielleicht *nicht lebensnotwendig* für die kleinen Programme, die im Rahmen der Programmierausbildung an der Hochschule erstellt werden, aber zwingend, wenn größere Programme in einer Firma entwickelt werden. Jede Firma hat im Allgemeinen *ihre* Programmierrichtlinien, damit die Programme auf der einen Seite ein einheitliches Erscheinungsbild aufweisen (unabhängig vom jeweiligen Programmierer) und auf der anderen Seite einfach (zumindest teilweise) in anderen Projekten wieder verwendet werden können (Kostensparnis). Da jede Informatikerin und jeder Informatiker früher oder später mit solchen Regeln konfrontiert wird, basiert auch die Programmierung in meinen Lehrveranstaltungen auf Regeln, die die Freiheiten der Programmiersprache sinnvoll einschränken oder nicht unbedingt erforderliche Sprachmittel erzwingen.

Die hier aufgeführten Regeln und Empfehlungen sind bei der Entwicklung der Software einzuhalten bzw. zu berücksichtigen. **Falls ein Programm in erheblichem Umfang von diesen Regeln abweicht, wird es mit *nicht ausreichend* (5.0) bewertet.**

- 1) Das Programm soll nach Möglichkeit nur die im **ISO-Standard** definierten Sprachmittel verwenden.
- 2) Jede Datei beginnt mit einem *Kopf*, der folgenden Aufbau hat:

```

/* Beschreibung des Dateiinhalts und wichtige Bemerkungen.
 *
 * Datei: <Dateiname>           Autor:   <Name>
 * Datum: <...>                 Version: <Versionsnummer>
 *
 * Historie: <WER hat WANN WAS geaendert / ergaenzt>
 *           Neue Eintraege kommen immer vor die alten Eintraege.
 */

```

Die Felder *Version* und *Historie* dürfen bei kleinen Programmen fehlen.

- 3) Nach dem Dateikopf wird folgende Programmstruktur verwendet:
 - a) Anweisungen an den Präprozessor:
 - **Alle** „*.h“ - Dateien öffnen. Pfadnamen sind im Allgemeinen nicht erlaubt. Notwendige Suchpfade werden über die entsprechende Compileroption definiert.
 - Symbolische Konstanten und Makros definieren. Damit das Programm einfach an neue Anforderungen angepasst werden kann, sollten im Programmtext keine festen Werte, sondern symbolische Konstanten verwendet werden (z. B. für die Größe eines Feldes).
 - b) neue Typen definieren
 - c) Funktionsprototypen deklarieren
 - d) globale Variablen deklarieren
 - e) Funktionen definieren (die erste Funktion **muss** die Funktion **main** sein!)
- 4) Jede Funktion (Prozedur) beginnt ebenfalls mit einem *Kopf*, der mindestens eine Funktionsbeschreibung, sowie eine Beschreibung der Ein- und Ausgabeparameter, des Rückgabewertes und der Fehlerausgänge enthält.
- 5) Das Programm wird kommentiert. Es werden nur **sinnvolle Kommentare** verwendet, die nicht unmittelbar aus dem Programmtext erkennbar sind. Kommentare sollen das Programm leichter lesbar machen. Beispiel:

```

falsch:   iobase[0x100] = 0x08;      /* setze iobase[0x100] auf 8 */
             printf ("\n");           /* Leerzeile ausgeben      */
richtig: iobase[0x100] = 0x08;      /* enable FIFO and interrupt */

```
- 6) Alle Kommentare werden in der gleichen Sprache geschrieben. Englische Kommentare sind im Allgemeinen kürzer.
- 7) Kommentare hinter Anweisungen beginnen in der 41. Spalte und enden vor der 72. Spalte (**Tabulator verwenden!**). Ab der 73. Spalte steht ggf. das Kommentarendenzeichen. Alle Kommentarzeichen stehen untereinander. Längere Kommentare werden vor die Anweisung geschrieben (ab Blocktiefe bis zur 72. Spalte). Der Tabulator muss grundsätzlich auf **acht** Zeichen eingestellt werden.

Bemerkung: Da der Quellcode ein Bestandteil der Dokumentation ist, muss er im DIN A4 Format ausgedruckt werden. Bei einer Zeichendichte von 12 cpi (characters per inch; entspricht einer Zeichengröße von 10 pt (point)) können auf einer Zeile bei einem 3 cm großen linken und 2 cm großen rechten Rand max. 75 Zeichen gedruckt werden.

- 8) Mehrzeilige Kommentare werden in der folgenden Form geschrieben:

```
/* ...
 * ...
 */
```

- 9) Programm und Kommentar werden **gleichzeitig** erstellt. Es ist **verboten**, zuerst das Programm zu erstellen und den Kommentar erst dann hinzuzufügen, wenn das Programm „läuft“.
- 10) Es wird entsprechend der Blocktiefe eingerückt. Einrücktiefe: genau **zwei** Zeichen je Block. Zusammengehörnde Schlüsselwörter bzw. Klammern stehen untereinander.

Beispiel:

<pre>if (...) { Anweisung(en); } else { Anweisung(en); }</pre>	<pre>if (...) { Anweisung(en); } else { if (...) { Anweisung(en); } }</pre>	<pre>switch (...) { case ...: Anweisung(en); break; default: Anweisung(en); break; }</pre>
--	---	--

Die geschweiften Klammern müssen auch dann verwendet werden, wenn nur eine Anweisung vorhanden ist.

- 11) Zwischen einem Funktionsnamen bzw. Schlüsselwort und der öffnenden Klammer sowie den einzelnen Ausdrücken in einer (Parameter-) Liste steht ein Leerzeichen.

Beispiel: `printf ("...", par1, par2, ...);`

- 12) Binäre und ternäre Operatoren und Operanden werden durch ein Leerzeichen getrennt

Beispiele: `x = a + b;` `z = ((x < y) ? x : y);`

- 13) Die Größe von Anweisungen und Ausdrücken muss überschaubar sein.
- 14) Ausdrücke werden ggf. durch Klammern strukturiert (besser lesbar). Die Anweisung

```
while ((table[i][0] != c) && (i < tab_len))
```

ist verständlicher als die *hinreichende C* Anweisung

```
while (table[i][0] != c && i < tab_len)
```

oder gar die *notwendige* Anweisung

```
while(table[i][0]!=c&& i<tab_len).
```

- 15) Die Größe der Funktionen und Moduln muss überschaubar sein.
- 16) Alle Anweisungen enden vor der 72. Spalte.
- 17) Je Zeile ist nur eine Anweisung erlaubt.
- 18) Anweisungen mit Seiteneffekten müssen vermieden werden.
- 19) Deklarationen erfolgen so lokal wie möglich (Modularisierung, *Information Hiding*).
- 20) Namen müssen sorgfältig gewählt und als Kommentar erklärt werden. Aus dem Namen soll nach Möglichkeit die Bedeutung der Größe abgeleitet werden können. Zur Strukturierung der Namen wird das Zeichen „_“ verwendet oder eine Mischung aus Groß- und Kleinbuch-

staben. Namen sollten nicht zu lang gewählt werden, da arithmetische Ausdrücke sonst leicht unübersichtlich werden. Deutsche Sonderzeichen sind in Namen verboten!

Beispiel: `piv_key` (oder `PivKey`) `/* Schluessel des Pivots */`
 `anf` `/* Index auf Anfang der Liste */`

- 21) Namen dürfen im Allgemeinen nicht mit dem Zeichen „_“ beginnen oder enden, da solche Namen für interne Namen des Compilers bzw. der Laufzeitbibliothek verwendet werden (Gefahr von Namens-Kollisionen).
- 22) Namen in `typedef` und `#define` Anweisungen werden im Allgemeinen in Großbuchstaben geschrieben.
- 23) Interne Namen oder Makronamen müssen sich in den ersten 31 Zeichen unterscheiden und externe (globale) Namen in den ersten 6 Zeichen.
- 24) Bei der Deklaration sind Anfangsinitialisierungen zu vermeiden, da solche Programme im Allgemeinen nicht im (E)PROM oder ähnlichen Speichern lauffähig sind.
- 25) Gleiche Dinge müssen gleich behandelt werden, z. B. entweder nur deutsche oder nur englische Namen wählen. Mischungen sollen möglichst vermieden werden.
- 26) Nicht unnötig Code kopieren, sondern Unterprogramme und Makros verwenden.
Aber: Keine Zerstückelung des Programms in beliebig viele Funktionen.
- 27) Speziallösungen sind zu vermeiden (möglichst Standardfunktionen verwenden).
- 28) Betriebssystem- bzw. Compiler-abhängige Anweisungen in einem eigenen Modul realisieren (in einer eigenen Datei), da das Programm dann einfacher auf einen anderen Rechner oder Compiler portiert werden kann.

Beispiel: Löschen einer Datei

```

Datei 1 (nur ISO C):      ret_code = del_file ("x012345.tmp");
Datei 2 (systemspezifisch): int del_file (char *file_name)
                          {
                              int ret_code;

                              #ifdef UNIX_C
                                  ret_code = unlink (file_name);
                              #endif
                              #ifdef TURBO_C
                                  ret_code = remove (file_name);
                              #endif
                              return ret_code;
                          }
    
```

- 29) Defensiv programmieren (auch *unmögliche* Fälle abfangen). Eine *switch*-Anweisung erhält z. B. auch dann einen *default*-Zweig, wenn die augenblickliche Programmversion alle möglichen Fälle abdeckt. Auf diese Weise werden bei späteren Programmiererweiterungen Fehler u. U. schneller gefunden, wenn die Erweiterung auch in der *switch*-Anweisung berücksichtigt werden muss (aber nicht berücksichtigt worden ist).
- 30) Nicht zu früh optimieren!
- 31) Falls das Programm eine Eingabe benötigt, wird zuerst eine aussagefähige Eingabeaufforderung ausgegeben (z. B. *Umsatz (in Euro):* oder *linke Intervallgrenze:*).

- 32) Bei fehlerhaften Eingaben wird ein Hilfetext ausgegeben, der den Fehler und die gewünschte Eingabe erläutert, bevor die Eingabe erneut angefordert wird.
- 33) Falls das Programm auf mehrere Dateien verteilt ist, sollte das Generierungsprogramm *make* bzw. *GNU make* verwendet werden.
- 34) Das Programm wird kommentiert **und** dokumentiert. Die Dokumentation muss mit einem *normalen* Editor oder einem an der Hochschule Fulda *verfügbaren* Textverarbeitungssystem erstellt werden.
- 35) Die Dokumentation besteht aus einer Benutzerbeschreibung und einer Detaildokumentation.
- 36) Die Benutzerbeschreibung muss für sich allein, d. h. ohne den Rest der Arbeit, verständlich sein. Sie muss das Format und die Wirkung aller Anweisungen (einschließlich Fehlerausgänge) vollständig beschreiben. Es wird empfohlen, die Benutzerbeschreibung ggf. zum besseren Verständnis mit Beispielen zu versehen.
- 37) Die Detaildokumentation des Programms muss so beschaffen sein, dass alle Wartungsarbeiten (Fehlersuche, Programmiererweiterungen, Umstellen auf ein neues Betriebssystem usw.) hinreichend schnell durchgeführt werden können. Bei der Detaildokumentation können sowohl die Benutzerbeschreibung als auch das Quellprogramm als vorhanden vorausgesetzt werden, so dass darauf Bezug genommen werden kann. Für die Detaildokumentation sind im Allgemeinen besonders wichtig:
 - a) Eine vollständige Beschreibung sämtlicher Daten und Algorithmen (Variablen, Konstanten, Tabellen, Arbeitsbereiche, Dateien usw.).
 - b) Ein Grobschema des Systems, z. B. als Beschreibung durch Schichten abstrakter Maschinen mit Modulen als Funktionseinheiten und einem groben Ablaufschema.
Einzelabläufe können im Allgemeinen unmittelbar dem Quellprogramm entnommen werden. Bei Abläufen mit sehr vielen Kontrollstrukturen sind grafische Hilfsmittel (z. B. UML-Diagramme, Nassi-Shneiderman-Diagramme oder Flussdiagramme) nützlich.
 - c) Hinweise zur Arbeitsumgebung: Rechnertyp, Betriebssystem- und Compilerversion, Anmerkungen zum Binden und Laden. Auf alle Besonderheiten ist deutlich hinzuweisen.
- 38) Englische Ausdrücke sollen im laufenden Text nach Möglichkeit nicht verwendet werden (z. B. File, Directory, Buffer usw.; besser: Datei, Dateiverzeichnis, Puffer, Zwischenspeicher, usw.). Falls kein bekanntes deutsches Wort zur Verfügung steht, wird der englische Begriff in Anführungszeichen gesetzt oder kursiv geschrieben.
- 39) Es ist **verboten**, englische Wörter mit deutschen Vorsilben zu versehen oder zu steigern (z. B. gescannt, gelinkt, geschiftet, Verclusterung).
- 40) Die Funktionsfähigkeit des Programms **muss** präsentiert werden (beim Übersetzen des Programms darf der Compiler auch dann keine Warnungen ausgeben, wenn **alle** möglichen Warnungen in der strengsten Stufe aktiviert werden). Zur Präsentation müssen ggf. geeignete Beispiele vorbereitet werden.

Beispiel

Im folgenden wird eine *iterative Lösung* des Sortierverfahrens *Quicksort* vorgestellt.

Detalldokumentation (Kurzfassung):

Das Sortierverfahren *Quicksort* arbeitet folgendermaßen: . . .

Die Geschwindigkeit des Sortierens hängt wesentlich von der Wahl des *Pivots* ab. Während n Datensätze im Mittel in $O(n \log n)$ Zeiteinheiten sortiert werden können, benötigt das Verfahren $O(n^2)$ Zeiteinheiten, wenn als *Pivot* immer der größte bzw. kleinste Schlüssel der Folge gewählt wird. Der *Pivot* kann auf folgende Arten bestimmt werden: . . .

Nachdem die Folge in zwei Teillisten aufgeteilt worden ist, muss entschieden werden, welche Teilliste im Keller abgelegt und welche sofort weiterbearbeitet wird. Falls immer die rechte (linke) Teilliste im Keller gespeichert und die linke (rechte) Teilliste weiterbearbeitet wird, müssen im Keller im ungünstigsten Fall $O(n)$ Teillisten der Länge *eins* zwischengespeichert werden. Wird dagegen immer die längere Teilliste im Keller zwischengespeichert und die kürzere Teilliste weiterbearbeitet, dann muss der Keller höchstens $O(\log n)$ Teillisten aufnehmen. Eine Teilliste muss nur dann gespeichert werden, wenn sie mindestens zwei Elemente enthält, da Teillisten mit einem Element bereits sortiert sind.

Beschreibung der Implementierung: . . .

In diesem Teil werden die verwendeten Datenstrukturen und der Aufbau der Funktion beschrieben. Für die lokalen Variablen *anf*, *anf_tmp*, *end* und *end_tmp* könnte diese Beschreibung z. B. folgendermaßen aussehen.

Die Indexgrenzen der aktuell zu bearbeitenden Teilliste werden vom Keller entfernt und in *anf_tmp* bzw. *end_tmp* gespeichert. Die Größen *anf* bzw. *end* werden als Schleifenvariablen verwendet, um die Teilliste in zwei neue Teillisten aufzuteilen, von denen die Linke nur Schlüsselwerte enthalten darf, die kleiner als der Schlüssel des *Pivots* sind und die Rechte nur solche Schlüssel, die größer oder gleich dem Schlüssel des *Pivots* sind.

Benutzerbeschreibung (Kurzfassung):

Die Funktion *qsort* sortiert die Elemente eines Feldes in aufsteigender Reihenfolge. Jedes Feldelement besteht aus dem Sortierschlüssel und einem Verweis auf den Datensatz. Die Sortierschlüssel müssen nicht eindeutig sein.

Definition: `int qsort (ELEM feld[], long links, long rechts);`

qsort ist eine iterative Implementierung des *Quicksort*-Algorithmus zum Sortieren der Elemente eines Feldes. Das Feld darf bis zu 1.048.576 Einträge enthalten. Jeder Eintrag des Feldes muss der Struktur *ELEM* entsprechen, die folgendermaßen definiert ist:

```
typedef struct { long wert;           /* Sortierschlüssel           */
                char *info;         /* Zeiger auf Datensatz      */
                } ELEM;
```

Die Funktion erwartet folgende Parameter:

- 1) einen Zeiger auf das erste Element des zu sortierenden Feldes
- 2) die linke Grenze (den ersten Index) der zu sortierenden Teilfolge
- 3) die rechte Grenze (den letzten Index) der zu sortierenden Teilfolge

Die Funktion liefert 0 zurück, wenn der Sortierlauf fehlerfrei durchgeführt werden konnte und 1, falls das Sortierfeld zu viele Elemente enthält (der Keller zu klein ist).

```
Beispiel: #define ANZ_SATZ 20                /* Anzahl Datensätze          */
            ELEM feld[ANZ_SATZ];           /* zu sortierendes Feld      */
            int i,                          /* Schleifenvariable        */
                r_wert;                     /* Rückgabewert einer Funktion */

            for (i = 0; i < ANZ_SATZ; ++i)
            {
                feld[i].wert = (i * i) % ANZ_SATZ;
                if ((feld[i].wert & 1) == 0)
                {
                    feld[i].info = "gerader Schlüsselwert ";
                }
                else
                {
                    feld[i].info = "ungerader Schlüsselwert ";
                }
            }
            r_wert = qsort (feld, 0, ANZ_SATZ - 1);
            if (r_wert != 0)
            {
                fprintf (stderr, "Routine 'qsort' meldet Fehler!\n");
                return r_wert;
            }
            ...
```

Quellprogramm der Routine:

```
#define MAX_KELLER      20                /* bis zu 2^20 Datensätze    */

/* Iterative Lösung des Quicksort. Es wird immer die längere
 * Teilliste im Keller zwischengespeichert und die kürzere Teilliste
 * weiterbearbeitet.
 *
 * Parameter:      feld      Zeiger auf Anfang der zu sortierenden Folge
 *                links     linker Index der zu sortierenden Teilfolge
 *                rechts    rechter Index der zu sortierenden Teilfolge
 *
 * Funktionswert: 0        fehlerfreier Sortierlauf
 *                1        Keller zu klein (zu viele Datensätze)
 *
 * Seiteneffekt:  Die Reihenfolge der Elemente des Feldes, mit dem
 *                'qsort' aufgerufen wird, wird geändert.
 */

int qsort (ELEM feld[], long links, long rechts)
{
    int    index;                          /* Index in Keller          */
    long   anf, anf_tmp,                   /* Index auf Anfang der Liste */
           end, end_tmp,                   /* Index auf Ende der Liste   */
           pivot;                           /* Schlüssel des Pivots      */
    ELEM   tmp;                             /* Zwischenspeicher        */
    struct { long l,                          /* linker Index            */
            r;                                /* rechter Index           */
    } keller[MAX_KELLER];                  /* Keller für Teillisten    */

    index = 0;                              /* Keller initialisieren    */
    keller[index].l = links;
    keller[index].r = rechts;
```

```

do
    {
        anf_tmp = keller[index].l;
        end_tmp = keller[index].r;
        --index;
        do
            {
                anf = anf_tmp;
                end = end_tmp;
                pivot = feld[(anf + end) / 2].wert;
                do
                    {
                        /* vom Ende her kleineren Schluessel suchen
                        while (pivot < feld[end].wert)
                        {
                            --end;
                        }
                        /* vom Anfang her groesseren Schluessel suchen
                        while (feld[anf].wert < pivot)
                        {
                            ++anf;
                        }
                        if (anf <= end)
                        {
                            tmp = feld[anf];
                            feld[anf] = feld[end];
                            feld[end] = tmp;
                            --end;
                            ++anf;
                        }
                    } while (anf <= end);
                /* laengere Teilliste bestimmen und im Keller ablegen
                if ( (end - anf_tmp) < (end_tmp - anf) )
                {
                    if (anf < end_tmp)
                    {
                        ++index;
                        if (index < MAX_KELLER)
                        {
                            keller[index].l = anf;
                            keller[index].r = end_tmp;
                        }
                        else
                        {
                            fprintf (stderr, "Fehler in der Routine 'qsort'! "
                                "Zu viele Datensaeetze (Keller zu klein).\n");
                            return 1;
                        }
                    }
                    end_tmp = end;
                }
                else
                {
                    if (anf_tmp < end)
                    {
                        ++index;
                        if (index < MAX_KELLER)
                        {
                            keller[index].l = anf_tmp;
                            keller[index].r = end;
                        }
                        else
                    }
                }
            }
        }
    }

```

```

        fprintf (stderr, "Fehler in der Routine 'qsort'! "
                "Zu viele Datensätze (Keller zu klein).\n");
        return 1;
    }
    anf_tmp = anf;          /* rechten Teil weitersortieren */
} while (anf_tmp < end_tmp); /* Ende: Teillisten bilden */
} while (index >= 0);     /* Ende: oberste Liste bearbeiten */
return 0;
}

```