Programmierrichtlinien

für die Programmiersprache

C

(Juli 2015)

Die Programmierrichtlinien geben Hinweise zur Gestaltung und Dokumentation der Software und bewirken, dass das Programm einfacher zu lesen, zu testen, zu warten und zu portieren 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 (Kostenersparnis). 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**.

Hochschule Fulda, Fachbereich AI, Prof. Dr. S. Groß

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 Compileroption „-I“ definiert.

      – Symbolische Konstanten und andere Makros definieren. Damit der Code einfach an neue Anforderungen angepasst werden kann, sollten immer symbolische Konstanten statt fester Werte benutzt werden (z. B. für die Größe eines Feldes).

   b) neue Typen definieren
   c) Funktionsprototypen deklarieren
   d) globale Variablen deklarieren
   e) Funktionen definieren (**„main ()“ muss die erste Funktion sein!**)

4) Jede Funktion beginnt ebenfalls mit einem *Kopf*, der mindestens eine Funktionsbeschreibung, sowie eine Beschreibung der Ein- und Ausgabeparameter, des Rückgabewertes, der Fehlerausgänge sowie der Seiteneffekte 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:   val = val + 1;           /* inkrementiere den Wert von val*/
          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 Kommentarendezeichen. Alle Kommentarzeichen stehen untereinander. Längere Kommentare (mehr als etwa 30 Zeichen) 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) Alle Anweisungen enden ebenfalls vor der 72. Spalte. Ggf. kann eine Zeile mit dem Zeilenfortsetzungszeichen „\“ umgebrochen werden (ist normalerweise aber nicht erforderlich).

9) Mehrzeilige Kommentare werden in der folgenden Form geschrieben:

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

10) Programm und Kommentar werden **gleichzeitig** erstellt. Es ist **verboten**, zuerst das Programm ohne Kommentar zu erstellen und den Kommentar erst dann hinzuzufügen, wenn das Programm „läuft“, da in der Regel für eine nachträgliche Kommentierung wegen neuer Projekte keine Zeit zur Verfügung steht.

11) Jeder neue Block wird um **genau zwei Zeichen** eingerückt. Zusammengehörende Schlüsselwörter und Klammern stehen untereinander. Die öffnende geschweifte Klammer darf in einer eigenen Zeile unter dem ersten Buchstaben der darüberliegenden Zeile oder am Ende einer Zeile stehen. Sie dürfen sich für eine Variante entscheiden und müssen sie dann konsistent in allen Programmen benutzen. Blöcke mit nur einer Anweisung müssen ebenfalls in Klammern stehen.

Beispiel:

```
if (...)
{
  statement(s);
}
else
{
  statement(s);
}
```

```
if (...)
{
  statement(s);
}
else
{
  for (...)
  {
    statement(s);
  }
}
```

```
switch (...)
{
  case ...:
    statement(s);
    break;
  ...
  ...
  default:
    statement(s);
    break;
}
```

oder (diese Version entspricht besser den Einrückungsregeln von *Xemacs*)

```
if (...) {
  statement(s);
} else {
  statement(s);
}
```

```
if (...) {
  statement(s);
} else {
  for (...) {
    statement(s);
  }
}
```

```
switch (...) {
case ...:
  statement(s);
  break;
...
...
default:
  statement(s);
  break;
}
```

12) Zwischen einem Funktionsnamen bzw. Schlüsselwort und der öffnenden Klammer sowie den Elementen in einer durch Komma oder Semikolon getrennten Liste steht ein Leerzeichen.

Beispiel:
```
printf ("...", par1, par2, ...);
for (int i = 0; i < ...; ++i)
```

13) Binäre und ternäre Operatoren und Operanden werden durch ein Leerzeichen getrennt

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

14) Die Größe von Anweisungen und Ausdrücken muss überschaubar sein.

15) 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).
```

16) Je Zeile ist nur eine Anweisung erlaubt.

17) Anweisungen mit Seiteneffekten müssen vermieden werden.

18) Deklarationen erfolgen so lokal wie möglich (Modularisierung, *Information Hiding*).

19) 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 „_“ oder eine Mischung aus Groß- und Kleinbuchstaben (*CamelCase*) verwendet. Namen sollten nicht zu lang gewählt werden, da arithmetische Ausdrücke sonst leicht unübersichtlich werden. Sonderzeichen einer Sprache, z. B. deutsche Umlaute (ä, ö, ü, ß), sind verboten (auch in Kommentaren)!

```
Beispiel: num_threads (or numThreads) /* number of parallel threads    */
          PI_25                       /* 25 digits of pi              */
```

20) 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).

21) Namen in `typedef` und `#define` Anweisungen werden im Allgemeinen in Großbuchstaben geschrieben.

22) Bei der Deklaration sind Anfangsinitialisierungen zu vermeiden, da solche Programme im Allgemeinen nur im RAM lauffähig sind (der Compiler initialisiert die Speicherzelle in der Regel direkt mit dem Wert und erzeugt keine Anweisung, um den Wert in die Speicherzelle zu laden).

23) Gleiche Dinge müssen gleich behandelt werden, z. B. entweder nur deutsche oder nur englische Kommentare und Namen wählen.

24) Nicht unnötig Code kopieren, sondern Unterprogramme und Makros verwenden.

Aber: Keine *Zerstückelung* des Programms in beliebig viele Funktionen.

25) Speziallösungen sind zu vermeiden (möglichst Standardfunktionen verwenden).

26) Betriebssystem- bzw. Compiler-abhängige Anweisungen in eigenen Dateien realisieren, da das Programm dann einfacher auf einen anderen Rechner oder Compiler portiert werden kann.

Beispiel: Erzeugen eines Dateiverzeichnisses

Datei 1 (nur ISO C):
```
ret_code = create_directory (...);
```

Datei 2 (systemspezifisch):
```
int create_directory (char *dir_name, ...)
{
  int ret_code;

  ret_code = EXIT_FAILURE;

  #ifdef UNIX
    ret_code = mkdir (dir_name, ...);
  #endif

  #ifdef MS_WINDOWS
    ret_code = EXIT_SUCCESS;
    if (!CreateDirectory (dir_name, ...))
    {
      ret_code = (int) GetLastError();
    }
  #endif
  return ret_code;
}
```

27) 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 Programmerweiterungen Fehler u. U. schneller gefunden, wenn die Erweiterung auch in der *switch*-Anweisung berücksichtigt werden muss (aber nicht berücksichtigt worden ist).

28) Nicht zu früh optimieren!

29) Falls das Programm eine Eingabe benötigt, wird zuerst eine aussagefähige Eingabeaufforderung ausgegeben (z. B. *Umsatz (in Euro):* oder *linke Intervallgrenze:*).

30) Bei fehlerhaften Eingaben wird ein Hilfetext ausgegeben, der den Fehler und die gewünschte Eingabe erläutert, bevor die Eingabe erneut angefordert wird.

31) Falls das Programm auf mehrere Dateien verteilt ist, sollte das Generierungsprogramm *make* bzw. *GNU make* verwendet werden.

32) Das Programm wird kommentiert **und** dokumentiert. Die Dokumentation muss mit einem *normalen* Editor oder einem an der Hochschule Fulda *verfügbaren* Textverarbeitungssystem erstellt werden.

33) Die Dokumentation besteht aus einer Benutzerbeschreibung und einer Detaildokumentation.

34) Die Benutzerbeschreibung muss für sich allein, d. h. ohne den Rest der Arbeit, verständlich sein. Sie muss die Bedienung des Programms (einschließlich Fehlerausgänge) vollständig beschreiben. Es wird empfohlen, die Benutzerbeschreibung ggf. zum besseren Verständnis mit Beispielen zu versehen.

35) Die Detaildokumentation des Programms muss so beschaffen sein, dass alle Wartungsarbeiten (Fehlersuche, Programmerweiterungen, 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, sodass 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 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.

36) 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. Es ist **verboten**, englische Wörter mit deutschen Vorsilben zu versehen oder zu steigern (z. B. gelinkt, geshiftet, Verclusterung, …).

37) 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.

Detaildokumentation (Kurzfassung):

Quicksort ist ein Teile-und-Herrsche Algorithmus, der die Elemente eines Feldes sortiert und folgendermaßen arbeitet: ...

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 *left, left_tmp, right* und *right_tmp* könnte diese Beschreibung z. B. folgendermaßen aussehen.

Die Indexgrenzen der aktuell zu bearbeitenden Teilliste werden vom Keller entfernt und in *left_tmp* bzw. *right_tmp* gespeichert. Die Größen *left* bzw. *right* 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_iterative* 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_iterative (ELEM array[], long left_index, long right_index);`

qsort_iterative ist eine iterative Implementierung des *Quicksort*-Algorithmus zum Sortieren der Elemente eines Feldes. Das Feld darf bis zu $2^{20} = 1.048.576$ Elemente enthalten. Jedes Element des Feldes muss der Struktur *ELEM* entsprechen, die folgendermaßen definiert ist:

```
typedef struct { long key;             /* sort key                    */
                 char *msg;            /* message                     */
               } 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 EXIT_SUCCESS zurück, wenn der Sortierlauf fehlerfrei durchgeführt werden konnte und EXIT_FAILURE, falls das Sortierfeld zu viele Elemente enthält, sodass der Kellerbereich zu klein ist.

Beispiel:
```
#define STACK_SIZE 20         /* up to 2^20 data sets         */

/* create the array with the data sets on the heap and not
 * "automatic" on the stack to save stack space and to avoid
 * segmentation faults due to a stack overflow.
 */
static ELEM array[NUM_ELEM];  /* array to sort                */
long i;                       /* loop variable                */
int  ret;                     /* return value of a function   */

srand ((unsigned int) time (NULL)); /* init random # generator*/
for (i = 0; i < NUM_ELEM; ++i)      /* initialize data sets   */
{
  array[i].key = rand() % NUM_ELEM;
  if ((array[i].key & 1) == 0)
  {
    array[i].msg = "even key";
  }
  else
  {
    array[i].msg = "odd  key";
  }
}
ret = qsort_iterative (array, 0, NUM_ELEM - 1);
if (ret != EXIT_SUCCESS)
{
  fprintf (stderr, "ERROR: Function \"qsort_iterative\" returns "
           "error code %d\n", ret);
  return ret;
}
...
```

Quellprogramm der Funktion *qsort_iterative*:

```
/* Iterative implementation of the sorting algorithm "Quicksort".
 * The indices of the bigger sub-array are stored on the stack, while
 * the smaller sub-array will be processed immediately, so that at
 * most O(log n) sub-arrays must be stored on the stack, if the array
 * contains n data sets. The stack is implemented with an array.
 *
 * input parameters:    array        (sub-)array to sort
 *                      left_index   left index of (sub-)array to sort
 *                      right_index  right index of (sub-)array to sort
 * output parameters:   array        sorted array
 * return value:        EXIT_SUCCESS sorting succeeded
 *                      EXIT_FAILURE error "stack overflow" due to too
 *                                     many data sets in "array"
 * side effects:        elements of "array" will possibly be rearranged
 */

int qsort_iterative (ELEM array[], long left_index, long right_index)
{
  int   stack_idx;                    /* index in "stack array"       */
  long  left, left_tmp,               /* index of leftmost element    */
        right, right_tmp,             /* index of rightmost element   */
```

```
      piv_key;                          /* key of pivot element         */
ELEM  tmp;                              /* temporary value for "swap"   */
struct { long l,                        /* left index                   */
              r;                        /* right index                  */
       } indices_stack[STACK_SIZE];     /* stack for sub-array indices  */

/* initialize stack with the very first array indices                   */
stack_idx = 0;
indices_stack[stack_idx].l = left_index;
indices_stack[stack_idx].r = right_index;

do                                      /* process top array of stack   */
{
  left_tmp  = indices_stack[stack_idx].l;
  right_tmp = indices_stack[stack_idx].r;
  --stack_idx;
  do                                    /* partition this array         */
  {
    left    = left_tmp;
    right   = right_tmp;
    piv_key = array[(left + right) / 2].key;
    do                                  /* swap elements if necessary   */
    {
      /* look for the first smaller key on the right side               */
      while (piv_key < array[right].key)
      {
        --right;
      }
      /* look for the first larger key on the left side                 */
      while (array[left].key < piv_key)
      {
        ++left;
      }
      if (left <= right)                /* perform swap operation       */
      {
        tmp          = array[left];
        array[left]  = array[right];
        array[right] = tmp;
        --right;                        /* adjust indices               */
        ++left;
      }
    } while (left <= right);

    /* determine bigger sub-array and store its indices on stack       */
    if ((right - left_tmp) < (right_tmp - left))
    {
      if (left < right_tmp)             /* store right sub-array        */
      {
        ++stack_idx;
        if (stack_idx < STACK_SIZE)
        {
          indices_stack[stack_idx].l = left;
          indices_stack[stack_idx].r = right_tmp;
        }
        else
        {
          fprintf (stderr, "Error in function \"qsort_iterative\"!\n"
                   "  Too many data sets (stack is too small).\n");
          return EXIT_FAILURE;
        }
      }
      right_tmp = right;                /* proceed with left sub-array  */
    }
```

```
      else
      {
        if (left_tmp < right)           /* store left sub-array         */
        {
          ++stack_idx;
          if (stack_idx < STACK_SIZE)
          {
            indices_stack[stack_idx].l = left_tmp;
            indices_stack[stack_idx].r = right;
          }
          else
          {
            fprintf (stderr, "Error in function \"qsort_iterative\"!\n"
                     "  Too many data sets (stack is too small).\n");
            return EXIT_FAILURE;
          }
        }
        left_tmp = left;                /* proceed with left sub-array  */
      }
    } while (left_tmp < right_tmp);     /* end: partition this array    */
  } while (stack_idx >= 0);             /* end: process top array ...   */
  return EXIT_SUCCESS;
}
```