C - Programmierrichtlinie

Sommersemester 2014

Prof. Dr. Steffen Reith
Steffen.Reith@hs-rm.de

Hochschule RheinMain
Fachbereich Design Informatik Medien

Erstellt von:	Steffen Reith
Zuletzt Überarbeitet von:	Steffen Reith
Email:	Steffen.Reith@hs-rm.de
Erste Version vollendet:	Februar 2007
Version:	–revision–
Date:	2014-04-24

Inhaltsverzeichnis

1 Einleitung

1.1 Ziele

Dieses Dokument beschreibt einen Satz von Regeln für die Programmierung in der Programmiersprache C. Die Richtlinien haben folgende Zielsetzung: Steigerung der SW-Qualität durch Vermeidung gefährlicher Sprachkonstrukte Verkürzung der Einarbeitungszeit in fremde Software durch Einheitlichkeit im Aufbau, in der Formatierung und in der Namensvergabe erhöhte Wiederverwendbarkeit der Software durch einheitliche und sinnvolle Kommentierung Erleichterte Portierung auf andere Compiler, Betriebssysteme oder Rechnerarchitekturen.

1.2 Richtlinien und Empfehlungen

Die meisten der Regeln in diesem Dokument sind als verbindliche Richtlinien zu verstehen, die eingehalten werden müssen und deren Einhaltung auch geprüft werden kann. Einige der Regeln sind mit dem Hinweis „Empfehlung“ gekennzeichnet. Diese Regeln beruhen auf positiven Erfahrungen in zahlreichen Projekten, sind aber nicht unumstritten. Teilweise sind auch Argumente für und wider dieser Regeln aufgeführt. Jeder Entwickler kann für sich selbst entscheiden, ob er die mit Empfehlung gekennzeichneten Regeln anwenden will oder nicht. Die Einhaltung dieser Regeln wird nicht geprüft.

1.3 Vorgehensweise für den Projektleiter

Der Projektleiter muss vor dem Beginn der Implementierungsphase entscheiden, wie er mit den Programmierrichtlinien umgehen will. Am einfachsten ist es, für ein Projekt diese Anweisung unverändert zu übernehmen. Dies sollte in den meisten Fällen möglich sein, da Regeln, zu denen es bekanntermaßen unterschiedliche Auffassungen gibt, als Empfehlungen gekennzeichnet sind. Falls es schwerwiegende Gründe gibt, diese C Programmierrichtlinien Beschreibung Aufbau von Quelldateien Richtlinien für ein Projekt abzuwandeln, so muss dies dokumentiert werden, was durch die Nummerierung der einzelnen Regeln erleichtert wird. Beispiel für eine QM-konforme Abwandlung der Standard-Regeln: Für das Projekt XY gelten die C-Programmierrichtlinien mit Ausnahme von Abschnitt 2.9.2. Diese wird ersetzt durch die Regel „Eingerückt wird immer um 4 Stellen“. Der Projektleiter kann einzelne Regeln außer Kraft setzen oder abändern oder verbindliche Regeln in Empfehlungen umwandeln. Umgekehrt kann er auch für seine Projekte zusätzliche Regeln einführen, oder Regeln die im Standard nur Empfehlungen sind zu verbindlichen Vorschrift erklären.

2 Programmierrichtlinien

2.1 Aufbau von Quelldateien

2.1.1 C-Dateien

Ein compilierfähiges C-Modul muss folgenden Aufbau besitzen:

- Datei-Kommentar mit mindestens folgenden Bestandteilen:
  - Dateiname
  - Autor (bevorzugt aus Änderungsliste ersichtlich)
  - Projekt
  - Copyright-Vermerk

  - Beschreibung der Funktion
  - Änderungsliste (manuell oder durch Versionsverwaltung generiert)

- `#include`-Anweisungen
- Makro-Definitionen
- Typ-Definitionen
- Prototypen für lokale Funktionen
- Definitionen für globale Konstanten
- Definitionen für globale Variablen
- Definitionen für modulweit sichtbare lokale Konstanten
- Definitionen für modulweit sichtbare lokale Variablen
- Globale und lokale Funktionen in beliebiger Reihenfolge

Einzelne Bestandteile können fehlen, die Reihenfolge darf aber nicht verändert werden (siehe Template). Ausnahme: benötigt eine Header Datei eine `#define-Anweisung`, so steht diese unmittelbar vor der(n) relevanten `#include`-Anweisung(en).

2.1.2 H-Dateien

Für den Aufbau von Includefiles (.H-Dateien) gibt es zwei Möglichkeiten:

1. Analog zu den C-Files:

   - Datei-Kommentar mit mindestens folgenden Bestandteilen:
     - Dateiname
     - Autor (bevorzugt aus Änderungsliste ersichtlich)
     - Projekt
     - Copyright-Vermerk
     - Beschreibung der Funktion
     - Änderungsliste (manuell oder durch Versionsverwaltung generiert)
   - `#include`-Anweisungen
   - Makro-Definitionen
   - Typ-Definitionen
   - Deklarationen für globale Konstanten
   - Deklarationen für globale Variablen
   - Deklarationen für globale Funktionen

   Einzelne Bestandteile können fehlen, die Reihenfolge darf aber nicht verändert werden (siehe Template).

2. Alternativ gibt es auch die Möglichkeit, Makros, Typen, Variablen und Funktionen in logisch zusammengehörige Gruppen zusammenzufassen. z.B. Typ- Definitionen zu den Funktionen, bei denen diese Typen als Parameter benutzt werden.

2.2 Kommentare zu Variablendefinitionen

Kommentare zu Variablendefinitionen werden in der gleichen Zeile nachgestellt:
Beispiel:
`int i; /* schleifenzaehler */`

2.3 Funktionskommentare

Funktionskommentare stehen **vor** dem Funktionskopf

2.4 Bestandteile eines Funktionskommentars

Ein Funktionskommentar muss mindestens folgende Bestandteile haben:

- Funktionsname,
- Kurzbeschreibung der Funktion

und soweit relevant:

- Beschreibung der Parameter,
- Beschreibung des Returncodes,
- Seiteneffekte,
- Vorbedingungen,
- verwendete globale Ressourcen.

2.5 Zeilenlänge (Empfehlung)

Es darf nicht über Spalte 78 hinausgeschrieben werden

2.6 Definitionsort und Sichtbarkeit

2.6.1 Extern-Deklaration

`Extern`-Deklarationen von globalen Variablen stehen grundsätzlich nur in Includefiles, niemals in C-Files.

2.6.2 include-Anweisungen

Ein Includefile, in dem globale Variablen oder Funktionen deklariert sind, muss auch in den C-Files eingezogen werden, in denen diese Variablen bzw. Funktionen definiert sind. Nur so kann der Compiler die Übereinstimmung prüfen.

2.6.3 Definitionen von Variablen und Funktionen

Definitionen von Variablen und Funktionen stehen in C-Files und nicht in Includefiles.

2.6.4 Lokale Variablen und Typen

Variablen und Typen, die nur innerhalb einer Funktion benutzt werden, müssen auch dort definiert sein, und dürfen nicht unnötigerweise modulweit oder gar global sichtbar sein.

2.6.5 Static-Deklarationen

Variablen und Funktionen, die nur innerhalb eines Moduls benutzt werden, müssen auch dort als „`static`“ definiert sein, und dürfen nicht unnötigerweise global sichtbar sein.

2.6.6 Lokale Typen

Typen, die nur in einem Modul benutzt werden, müssen dort definiert sein.

2.6.7 Abhängige Include-Files

Wenn in einem Include-File „A“ Definitionen aus einem anderen Include-File „B“ benutzt werden (z.B. Typen oder Makros), so muss „Bïn „A“ mit `#include` eingebunden werden. D.h. der Benutzer eines Include-Files ist nicht dafür verantwortlich, welche Voraussetzungen dieses Include-File benötigt, vielmehr muss sich das Include-File diese Voraussetzungen selbst schaffen. Für Include-Files von projektweiter Bedeutung kann von dieser Regel abgewichen werden, um Übersetzungszeiten zu reduzieren. Diese Situation muss aber in einem Kommentar vermerkt werden.

2.6.8 Explizite Einbindung von Include-Files

Wenn in einem C-File explizit Informationen aus einem Include-File benutzt werden, das bereits indirekt eingebunden wird, so muss dieses aus Gründen der Nachvollziehbarkeit auch direkt eingebunden werden.

2.6.9 Schutz vor Mehrfacheinbindung

Jede Include-Datei muss sich selbst durch ein Konstrukt der folgenden Art gegen mehrfaches Einbinden schützen:

```
#ifndef FILENAME_H
#define FILENAME_H
.....
#endif /* FILENAME_H */
```

Außerhalb der ifndef - endif -Klammer dürfen nur Kommentare stehen.

2.7 Funktionen und Prototypen

2.7.1 Prototypen für globale Funktionen

Zu jeder globalen Funktion muss ein Prototyp in einem Includefile mit vorangestelltem „`extern`“ existieren.

2.7.2 Prototypen für lokale Funktionen

Prototypen für lokale Funktionen stehen im .C-File mit vorangestelltem „`static`“.

2.7.3 Prototypen und Parameter

Bei Prototypen keine Parameternamen angegeben.

2.7.4 Funktionen ohne Rückgabewert

Funktionen ohne Returncode bei Prototyp und Definition explizit vom Typ „`void`“.

2.7.5 Funktionen ohne Parameter

Bei Funktionen ohne Parameter bei Prototyp und Definition explizit „`void`“ in der Parameterliste.

2.8 Verwendung von #define

Hinweis: Die Schreibweise von Makronamen ist in Abschnitt 2.10.2 beschrieben.

2.8.1 Klammerung von Makrodefinitionen

Makrodefinitionen müssen vollständig geklammert werden, d.h. um jeden Parameter und um das gesamte Makro sind Klammern zu setzen. Wenn das Makro einen Ausdruck darstellt, sind die äußeren Klammern rund, wenn es eine Anweisung darstellt, sind sie geschweift. Bei einfachen Konstantendefinitionen - auch solchen mit einem cast-Operator kann die Klammerung entfallen. Beispiele:

```
#define MACRO(a,b) ((a)+(b))
#define SWAP(x,y,h) { (h)=(x); (x)=(y); (y)=(h); }
#define MAX_UNIT 2
#define MAX_DWORD (tDWORD)0xFFFFFFFFUL
```

2.8.2 Schlüsselwörter und Makros

Schlüsselwörter von C und vordefinierte Bezeichner dürfen nicht umdefiniert werden. Das gilt auch z.B. für Klammern („{}“ nicht durch „begin“ „end“ ersetzen).

2.8.3 Nicht-standardisierte Schlüsselwörter

Nicht-Standardisierte Schlüsselwörter wie z.B. „_near“, „_far“, „_pascal“ etc sollten dagegen - wenn sie benutzt werden - immer in folgender Weise umdefiniert werden:

```
#define FAR _far
```

Das erleichtert die Portierung auf Compiler, die diese Schlüsselwörter nicht unterstützen.

2.9 Formatierungsregeln

Hinweis: Die folgenden Regeln zur Formatierung von C Source Code sind überwiegend als Empfehlung gekennzeichnet. Es gibt auch andere weitverbreitete Stile und welchen man verwendet ist letztlich Geschmackssache. Wichtig ist aber, dass innerhalb eines Projekts ein einheitlicher Stil gepflegt wird.

2.9.1 Geschweifte Klammern (Empfehlung)

Zusammengehörige geschweifte Klammern stehen untereinander in der gleichen Spalte.

2.9.2 Einrückung (Empfehlung)

Nach einer öffnenden geschweiften Klammer wird eingerückt.

2.9.3 Einrückung von geschweiften Klammern (Empfehlung)

Eine öffnende geschweifte Klammer steht auf dem gleichen Niveau wie die Zeile davor.

2.9.4 Einzelanweisungen

Einzelanweisungen nach „`if`“, „`else`“, „`for`“, „`while`“ und „`do`“ stehen eingerückt in einer separaten Zeile.

2.9.5 Einrückungstiefen (Empfehlung)

Eingerückt wird immer um 2 Stellen.

2.9.6 Blanks und Tabs

Zum Einrücken werden Blanks verwendet. Tabs sind verboten.

2.9.7 Runde Klammern (Empfehlung)

Zusammengehörige runde Klammern stehen entweder in einer Zeile oder in einer Spalte. Gute Beispiele finden sich in Abbildung 1.

2.10 Schreibweise von Bezeichnern

Hinweis: Die folgenden Regeln zur Schreibweise von Bezeichnern haben sich in zahlreichen Projekten bewährt. Es gibt allerdings auch Gegenargumente, insbesondere zur Verwendung von Typ-Präfixen. Der ANSI/ISO-Standard für C garantiert bei Bezeichnern mit external Linkage nur die Unterscheidung anhand der ersten sechs Buchstaben ohne Berücksichtigung von Groß- und Kleinschreibung. Bei älteren Entwicklungsumgebungen, die lediglich den Standard implementiert haben, kann deshalb die Verwendung langer Typ-Präfixe zu Namenskonflikten führen.

2.10.1 Variablen und Konstanten (Empfehlung)

Variablen und Konstanten werden bei zusammengesetzten Namen mit Groß- und Kleinschreibung gegliedert. Erster Buchstabe klein. Keine Unterstriche verwenden.
Beispiel:

```
int myLowValue;
```

Falls es notwendig ist, auf den Typ der Variable hinzuweisen, kann die folgende Variante verwendet werden:
Variablen und Konstanten werden in Anlehnung an die „Ungarische Notation“ benannt: <Typ-Präfix> + <Name>; Präfix grundsätzlich klein, erster Buchstabe des Namens groß, bei zusammengesetzten Namen mit Groß-/Kleinschreibung gliedern, keine Unterstriche verwenden. Die Präfixe für Integerwerte werden im Abschnitt „Typen fester Größe“ definiert.
Standard-Typ-Präfixe:

```
char * sz zero terminated string
bool b
float f
double d
pointer p
function pointer pfn
```

```
if ( wA > wB )
{
  Anweisungen;
}
else
{
  Anweisungen;
}

for ( s=1; s<=4; s++ )
{
  Anweisungen;
}

while ( sA > sB )
{
  Anweisungen;
}

do
{
  Anweisungen;
} while ( sA > sB );

switch ( s )
{
  case 1: Anweisung; break;
  case 2: Anweisung; break;
  default: Anweisung; break;
}

void func
(
  char chY,
  long lZ
)
{
  Anweisungen;
}
```

```
if ( bX == 0 )
  Anweisung;
else
  Anweisung;

for ( s=1; s<=4; s++ )
  Anweisung;

while ( sA > sB )
  Anweisung;

do
  Anweisung;
  while ( sA > 0 );

typedef struct
{
  short s;
  long l;
} tNAME;

void func( long l )
{
  Anweisungen;
}
```

Abbildung 1: Beispiel für die Benutzung von runden Klammern

```
global/static g
Falls notwendig:
near pointer np
far pointer lp
huge pointer hp
```

Beispiele:

```
bool bOk, *pb;
int8_t i8Index, *pi8Next, **ppi8;
static char *szString[] = "Das ist ein String";
*pi8Next = i8Index;
pi8Next = &i8Index;
pi8Next = *ppi8;
*pi8Next = **ppi8;
ppi8 = &pi8Next;
```

Auch für selbstdefinierte Typen sollten Präfixe definiert und verwendet werden. Für den Umgang mit existierenden Quellen soll noch darauf hingewiesen werden, dass auch abweichende Konventionen für die Präfixe existieren: f - flag/bool, d - float, df - double.

2.10.2 Makros

Selbstdefinierte Makronamen werden groß geschrieben, Ziffern und Unterstriche sind erlaubt. Unterstriche am Anfang sollen nicht verwendet werden, dies ist Systemquellen vorbehalten.
Beispiel:

```
#define NR_OF_ELEMENTS 10
```

2.10.3 Typen (Empfehlung)

Selbstdefinierte Typnamen werden kleingeschrieben und durch ein angehängtes „_t“ gekennzeichnet.
Beispiel:

```
typedef unsigned short word_t;
```

2.10.4 Funktionen (Empfehlung)

Funktionsnamen werden mit Groß/Kleinschreibung gegliedert, der erste Buchstabe klein. Sie sollen keinen Unterstrich enthalten.
Beispiel:

```
getNextHandle()
```

Vorschlag: Die Namen globaler Funktionen sollten durch einen Präfix gekennzeichnet werden, der den Funktionsbaustein kennzeichnet, zu dem sie gehören. Mit Funktionsbaustein ist hier eine größere, logisch in sich abgeschlossene Softwareeinheit mit längerfristig stabilen Schnittstellen nach außen gemeint. Es ist i.A. nicht sinnvoll, den Namen einer Funktion mit einer Kennung der Datei zu versehen, in der sie definiert ist, da sich durch Aufspaltung oder Umgruppierung innerhalb eines Funktionsbausteins die Zuordnung Funktion zu Datei leicht ändern kann.

Der Präfix wird komplett großgeschrieben und dem Funktionsnamen getrennt durch „_“ vorangestellt.
Beispiel, FCU für File-Compare-Unit:

```
FCU_getNextHandle()
```

2.10.5 Typen fester Größe

Mit ANSI C99 wurden in der Header-Datei stdint.h ganzzahlige Datentypen mit einer vorgegebenen Breite eingeführt. Die Breite N eines ganzzahligen Typs ist die Anzahl der Bits, die zur Darstellung von Werten des Typs einschließlich dem Vorzeichenbit verwendet wird (i.a. N=8, 16, 32, 64). Sollte der verwendete Compiler noch nicht ANSI C 99-konform sein, ist die entsprechende Header-Datei (stdint.h) selbst zu definieren. Folgende Datentypen sollten mindestens definiert sein:

Datentyp	Präfix	Länge	vorzeichenbehaftet
`int8_t`	i8	Breite exakt 8 Bit	ja
`int16_t`	i16	Breite exakt 16 Bit	ja
`int32_t`	i32	Breite exakt 32 Bit	ja
`uint8_t`	u8	Breite exakt 8 Bit	nein
`uint16_t`	u16	Breite exakt 16 Bit	nein
`uint32_t`	u32	Breite exakt 32 Bit	nein
`int_fast8_t`	ifast8	Schnellster Typ mit Mindestbreite von 8 Bit	ja
`int_fast16_t`	ifast16	Schnellster Typ mit Mindestbreite von 16 Bit	ja
`int_fast32_t`	ifast32	Schnellster Typ mit Mindestbreite von 32 Bit	ja
`uint_fast8_t`	ufast8	Schnellster Typ mit Mindestbreite von 8 Bit	nein
`uint_fast16_t`	ufast16	Schnellster Typ mit Mindestbreite von 16 Bit	nein
`uint_fast32_t`	ufast32	Schnellster Typ mit Mindestbreite von 32 Bit	nein
`intmax_t`	imax	Breite maximal (größter ganzzahliger Wert)	ja
`uintmax_t`	umax	Breite maximal (größter ganzzahliger Wert)	nein

2.10.6 Datentyp bool

Boolsche Werte sollen durch den in ANSI C99 definierten Datentyp bool repräsentiert werden. Der boolsche Wert true („wahr“) wird durch 1 und false („falsch“) durch 0 repräsentiert. Sofern die Datei stdbool.h inkludiert wird, können die Bezeichner bool, true und false verwendet werden. Sollte der verwendete Compiler noch nicht ANSI C 99-konform sein, ist die entsprechende Header-Datei selbst zu definieren.

2.10.7 Sprache

Für Bezeichner und Kommentare ist eine einheitliche Sprache zu verwenden, dies ist Englisch sofern nichts anders definiert wurde.

2.11 Speziele C-Sprachkonstrukte

2.11.1 Goto

„`goto`“ darf nur bei Fehlerausgängen oder anderen speziellen Situationen verwendet werden.

2.11.2 continue (Empfehlung)

„`continue`“ darf nicht verwendet werden.
Hinweis: Die Erfahrung aus früheren Projekten hat gezeigt, dass die Wirkung von `continue` häufig nicht exakt verstanden wird und dadurch subtile Fehler entstehen können.

2.11.3 switch - default

Jede „`switch`“-Anweisung muss einen „`default`“-Zweig haben.

2.11.4 switch - break

Jeder `case`-Zweig muss ein break oder einen expliziten Hinweis auf das Fehlen von Break enthalten.

2.12 Sonstiges

2.12.1 Variableninitialisierung

Alle Variablen müssen vor dem ersten Lesezugriff explizit initialisiert werden. Ausnahmen: Der C-Standard garantiert, dass statische Daten mit jeweils für den Typ passenden Null-Werten (0, 0.0, Nullzeiger) initialisiert werden. Diese Eigenschaft kann ausgenutzt werden, z.B. um den Platz für zusätzlichen Initialisierungscode zu sparen. Es muss aber in einem Kommentar darauf hingewiesen werden.

2.12.2 Unbenutzte Funktionen

Die Software darf keine unbenutzten Funktionen enthalten. Einschränkung: Bei Libraries, die von mehreren Hauptprogrammen benutzt werden, kann es natürlich Funktionen geben, die eingebunden, aber nicht benutzt werden.

2.12.3 Klammern in Ausdrücken

In komplexeren Ausdrücken, müssen grundsätzlich Klammern gesetzt werden, auch wenn die C-Vorrangregeln für Operatoren dies nicht zwingend erfordern. Dies gilt auch für Ausdrücke, die vom Präprozessor ausgewertet werden.
Beispiele:

```
fOK = (w != 0);
if ( (wA > 0) || (wB < 0) )
...
#if (SEC_SECURITY == C) || (SEC_SECURITY_CLASS == CCC)
...
#endif /* (SEC_SECURITY == C) || (SEC_SECURITY_CLASS == CCC) */
```

2.12.4 Globale Pointer

Globale Pointer dürfen nicht auf Stack-Variable zeigen (Gefahr von „dangling references“)

2.12.5 Pfade in #include-Anweisungen

In `#include`-Anweisungen dürfen Includefiles nicht mit *absoluten* Pfaden bezeichnet werden.

2.12.6 Systemincludes

Systemincludes müssen *vor* anwenderdefinierten Dateien eingezogen werden.

3 Historie

Bearbeiter	Kürzel	Änderungsgrund	Version	Datum
Steffen Reith	streit	Neuerstellung	V1.0	31.08.2006
Steffen Reith	streit	kleine Fehler entfernt	V1.1	18.03.2012
Steffen Reith	streit	Umstellung auf XeLaTeX	V1.2	24.04.2014