# C-Programmierung:
# Hinweise zum Vermeiden von Programmierfehlern



Stand 8. Februar 2004

http://home.ntz.de/phleps/programming/c_fehler_vermeiden.pdf

Ein kluger Mann macht nicht alle Fehler selbst. Er gibt auch anderen eine Chance.

*Sir Winston Churchill*

## Zusammenfassung

Beim Programmieren mit C gibt es – wie auch bei anderen Programmiersprachen – einige Fallgruben. In manche bin ich im Lauf der Zeit selbst gefallen, auf andere wurde ich durch verschiedene Bücher aufmerksam. Viele der Fallgruben lassen sich leicht vermeiden – wenn man sie kennt.
Mit den folgenden Hinweisen möchte Ihnen ich die Chance bieten, zumindest einige der Fehler nicht nochmals selbst machen zu müssen. Die Liste ist jedoch nur als Einstieg gedacht und ist keineswegs vollständig. All denen, die professionell in C programmieren, sei deshalb das Studium weiterer Literatur zum Thema "*Fehlerfrei Programmieren*" dringend empfohlen.

# Inhaltsverzeichnis

# 1 Compiler-Optionen

- Verwenden Sie beim Übersetzen der Programme den höchsten Warning-Level des Compilers. Beseitigen Sie nicht nur die Ursachen der Fehlermeldungen, sondern überprüfen Sie auch die Ursachen für Warnungen.
  Je nach den Eigenschaften des verwendeten Compilers kann es auch sinnvoll sein, den Quellcode zusätzlich durch das Programm *lint*[1], oder durch einen weiteren Compiler prüfen zu lassen.

  **Grund:** Die Warnungen liefern oft Hinweise auf Stellen im Quellcode, die nicht nicht das tun, was beabsichtigt war.

# 2 Funktions-Prototypen

- Deklarieren Sie für jede Funktion einen Funktions-Prototypen **nach ANSI**!
  Geben Sie bei den Prototypen immer sowohl die Parameter-Typen, als auch die Parameter-Namen an.

  **Grund:** Funktionsprototypen verhindern Funktionsaufrufe mit falschen, zu vielen oder fehlenden Argumenten. Außerdem sorgen sie bei Bedarf für korrekte Typ-Konvertierungen.

- Geben Sie bei Funktionsprototypen und Funktionsdefinitionen den Funktionstyp auch bei Funktionen vom Typ `int` immer explizit mit an.

  **Grund:** Wenn der Funktionstyp nicht angegeben ist, verwendet der C-Compiler automatisch den Typ `int`. Es ist aber nicht mehr erkennbar, ob die Funktion wirklich vom Typ `int` sein soll, oder ob die Typ-Angabe vergessen wurde.

- Der Prototyp muß immer vor dem ersten Funktionsaufruf und vor der Funktionsdefinition stehen.

  **Grund:** Damit erkennt der Compiler (zumindest bei entsprechend hoher Warnstufe) fehlende Prototypen.

# 3 Funktionsergebnis

- Übergeben Sie an aufrufende Funktionen niemals Zeiger auf lokale auto-Variablen:
  **Falsch:** ☹

```
int localVar;
...
return & localVar;    /* VERBOTEN !!! */
```

  **Grund:** Lokale Variablen werden auf dem Stack abgelegt. Nach dem Ende jeder Funktion wird deren Stack-Bereich freigegeben und kann damit von anderen Funktionen neu belegt werden.

- Wenn das Ergebnis von `getchar()` einer Variablen zugewiesen und auf EOF geprüft wird, müssen Sie dazu eine Variable vom Typ `int` verwenden.

  **Grund:** Bei Variablen vom Typ `char` ist nicht sichergestellt, daß der Wert `EOF` immer von gültigen Zeichen unterschieden werden kann.

[1] *lint* ist ein Programm zur Prüfung der Syntax und der Semantik des Quellcodes, das mehr Warnungen liefert als viele C-Compiler.

# 4 Funktions-Makros

- Setzen Sie bei Makro-Definitionen ("Funktions-Makros") immer sowohl den kompletten Ersatztext, als auch jeden einzelnen Makro-Parameter in Klammern.

  Beispiel: `# define QUADRAT(x) ( (x) * (x) )` /* ☺ */

  **Falsch:** `# define QUADRAT(x)` ~~`x * x`~~ /* ☹ */

  **Grund**: Ohne die Klammern würde z. B. der Ausdruck `4 * QUADRAT` $\underbrace{(1+2)}_{x}$ durch

  $4 * \underbrace{1+2}_{x} * \underbrace{1+2}_{x}$ ersetzt und damit nicht das gewünschte Ergebnis 36 liefern.

# 5 Variablen

- Definieren Sie innerhalb des Gültigkeitsbereichs einer Variablen nicht nochmals eine Variable gleichen Namens in einem inneren Block `{ ... }`.

  **Grund:** Wenn es z. B. eine globale Variable `zaehler` gibt, und Sie innerhalb einer Funktion, die im Gültigkeitsbereich dieser globalen Variablen liegt, nochmals eine lokale Variable oder einen Parameter mit dem selben Namen `zaehler` definieren, wird dies bei einer Fehlersuche oder bei späteren Programmänderungen zur Verwirrung (und damit möglicherweise zu einem Programmfehler) führen.

# 6 Stringkonstanten

- Versuchen Sie nie, den Inhalt von Stringkonstanten (Zeichenketten zwischen `"..."`) zu verändern. Wenn der Inhalt eines Strings verändert werden soll, muß dafür ein Feld reserviert werden:

  **Richtig:** ☺

  ```
  char stringFeld [] = "Feldinhalt darf veraendert werden";

  stringFeld [3] = 'X';
  ```

  **Falsch:** ☹

  ```
  char * stringKonst = "aendern des Inhalts verboten";
  ~~stringKonst [3] = 'X';~~    /* VERBOTEN !!! */
  ```

  **Grund:**

  1. Der Compiler kann mehrfach vorkommende Stringkonstanten wie z. B.

     ```
     char * stringKonst1 = "aendern des Inhalts verboten";
     char * stringKonst2 = "verboten";
     ```

     zusammenfassen und evtl. nur einmal im Speicher ablegen. Änderungen am Inhalt können sich dann unbeabsichtigt auch auf andere Variablen auswirken.
  2. Bei manchen Rechnerplattformen werden Stringkonstanten in Speicherbereichen abgelegt, bei denen das System sicherstellt, daß das Programm nur Lese-Zugriffe durchführen kann.

  Programme, die den Inhalt von Stringkonstanten verändern, funktionieren deshalb möglicherweise nicht mehr, wenn sie – nach erneutem Compilieren – auf einer anderen Rechnerplattform verwendet werden sollen.

# 7 Numerische Werte

- Halten Sie die in der folgenden Tabelle[2] angegebenen Wertebereiche ein, die für ANSI-C als Mindestbereiche[3] festgelegt sind.
  Kontrollieren sie beim Programmieren für Embedded-Systeme, ob die dort verwendeten Wertebereiche evtl. vom ANSI-Standard abweichen.

| Datentyp | Zulässiger Wertebereich | | | | |
|---|---|---|---|---|---|
| char | 0 | <= | x | <= | 127 |
| signed char | -127 [4] | <= | x | <= | 127 |
| unsigned char | 0 | <= | x | <= | 255 |
| | Größe unbekannt, mindestens 8 Bits | | | | |
| int | -32767 | <= | x | <= | 32767 |
| signed int | -32767 | <= | x | <= | 32767 |
| unsigned int | 0 | <= | x | <= | 65535 |
| | Größe unbekannt, mindestens 16 Bits | | | | |
| short | wie int | | | | |
| long | -2147483647 | <= | x | <= | 2147483647 |
| signed long | -2147483647 | <= | x | <= | 2147483647 |
| unsigned long | 0 | <= | x | <= | 4294967295 |
| | Größe unbekannt, mindestens 32 Bits | | | | |
| int i:n | 0 | <= | x | <= | $2^{n-1}-1$ |
| signed int i:n | $-(2^{n-1}-1)$ | <= | x | <= | $2^{n-1}-1$ |
| unsigned int i:n | 0 | <= | x | <= | $2^{n-1}$ |
| | Größe unbekannt, mindestens n Bits | | | | |

  **Grund:** Dies vermeidet Fehler, die beim Portieren zwischen Plattformen mit unterschiedlichen Architekturen auftreten können.

- Verwenden Sie hinter Konstanten vom Typ `long` immer den Großbuchstaben 'L', nie den Kleinbuchstaben 'l'.

  **Grund:** Der Kleinbuchstabe '`l`' kann leicht mit der Ziffer '`1`' verwechselt werden.

# 8 Inkrementieren und Dekrementieren

- Variablen, die inkrementiert oder dekrementiert werden, dürfen nur einmal innerhalb des Ausdrucks stehen.
  Das Ergebnis von

```
a[i] = i++ ;
```

  ist nicht definiert.

[2] Quelle: [3, S. 150]
[3] Quelle: [2, S. 257]
[4] nicht -128: Dieser Wert wäre nicht portierbar auf Systeme, die Zahlen als Wert + Vorzeichen darstellen.

# 9 Bit-Operationen

- Wenden Sie Operatoren zur Verknüpfung von Bits (`&`, `|`, `^` und `~`) nur auf *unsigned*-Datentypen an.

  **Grund:** Bit-Operationen auf *signed*-Datentypen könnten auf anderen Rechnerplattformen (z. B. mit anderen Wertebereichen) zu anderen Ergebnissen führen.

# 10 Vergleiche, Schleifenbedingungen

- Schreiben Sie bei Vergleichen mit einem konstanten Wert den konstanten Wert links vom Vergleichs-Operator:

  Beispiel: `if ( NULL == file )` /* ☺ */

  anstatt ~~`if ( file == NULL )`~~ /* ☹ */

  **Grund**: Bei Vergleichen wird gelegentlich statt des gewünschten Vergleichs-Operators `==` oder `!=` versehentlich nur ein `=` geschrieben. Damit erfolgt kein Vergleich, sondern eine Zuweisung.
  Wenn der konstante Wert auf der linken Seite des Operators steht, ist eine Zuweisung nicht möglich und der Compiler meldet einen Fehler.

- Prüfen Sie bei Schleifenbedingungen nicht auf Ungleichheit mit dem Grenzwert (`'!='`), sondern prüfen Sie, ob der Wert innerhalb des erlaubten Wertebereichs liegt.

  Beispiel: `while ( wert < GRENZWERT )` /* ☺ */

  anstatt ~~`while ( GRENZWERT != wert )`~~ /* ☹ */

  **Grund**: Falls der Wert den Grenzwert überspringt, bricht die Schleife bei Prüfung auf 'ungleich' nicht ab.

# 11 Schleifen- oder Verzweigungs-Blöcke

- Setzen Sie auch bei Blöcken, die nur aus einer Anweisung bestehen, immer geschweifte Klammern.

  **Grund:** Wenn bei einem Block wie z. B.

```
for ( index = 1 ; index < max ; index ++ )
  process ( array [ index ] );      /* schlechter Code !! ☹  */

...
```

  später noch eine weitere Anweisung (wie `printf()` im folgenden Beispiel) innerhalb der Schleife ausgeführt werden soll, werden die dann notwendig werdenden Klammern oft vergessen; einrücken reicht nicht aus, suggeriert aber, daß der Befehl Teil der Schleife sei.
  Wenn die geschweiften Klammern schon vorhanden sind, wird die neue Zeile intuitiv richtig eingefügt.

  **Richtig:** ☺

```
for ( index = 1 ; index < max ; index ++ )
{
  process ( array [ index ] );
  printf  ( ''%d\n'', index );
}
```

  **Falsch:** ☹

```
for ( index = 1 ; index < max ; index ++ )
  process ( array [ index ] );
  printf  ( ''%d\n'', index );      /* ist nicht mehr Teil der Schleife!! */
```

- Schreiben Sie leere Anweisungen immer in eine eigene Zeile.
  Beispiel:
  ```
  while ( bedingung )
  {
    ;                       /* ☺ */
  {
  ```
  anstatt ~~`while ( bedingung ) ;`~~ /* ☹ */

  **Grund**: Wenn das Semikolon der leeren Anweisung direkt hinter dem Vergleich bzw. der Schleifenbedingung steht, ist später nicht mehr erkennbar, ob das Semikolon absichtlich oder versehentlich dort steht.

- Geben Sie für jeden `switch`-Block immer auch einen `default`-Zweig an.
  Wenn der `default`-Zweig nie durchlaufen werden darf, sichern Sie ihn mit `assert(0);` (siehe nächster Abschnitt) und `break;` ab:

```
switch ( bedingung )
{
  case WERT1:
         ...
         break;

  case ...
         ...
         break;

  default:
         assert ( 0 );  /* Dieser Zweig duerfte nie durchlaufen werden! */
         break;
}
```

# 12 assert()

- Verwenden Sie das Funktions-Makro `assert()`, um Ihre Annahmen über unzulässige Zustände oder unzulässige Parameter-Werte zu kontrollieren.
  Beispiel:
  ```
  /* Annahme: Die Variable 'datei' hat hier nie den Wert NULL */
  assert ( NULL != datei );  /* ☺ */
  ```

  Für `assert()` muß
  `# include <assert.h>`
  eingebunden werden.
  Wenn eine `assert`-Bedingung nicht erfüllt ist, bricht das Programm ab und gibt Namen und Zeilennummer der Quellcode-Datei sowie die fehlgeschlagene Bedingung als Fehlermeldung aus.
  Da diese Informationen nur für SW-Entwickler sinnvoll sind, Anwendern aber nichts sagen, dürfen Laufzeitfehler (z. B. fehlerhafte Eingaben, fehlgeschlagene Speicheranforderungen usw.) nicht mit `assert()` abgesichert werden!
  Außerdem darf das Argument von `assert()`-Anweisungen immer nur Vergleiche bzw. Abfragen enthalten, aber niemals Variablen- oder Speicher-Inhalte verändern.

  **Grund:** Wenn Sie `assert()` an den richtigen Stellen einsetzen, erkennen Sie sehr schnell und einfach Programmzustände, von denen Sie annahmen, daß sie nie auftreten würden.

# 13 Dynamischer Speicher

- Verwenden Sie für den Zugriff auf den Inhalt von dynamischem Speicher keine eigenen Zeiger, die mitten in den Speicherblock zeigen, sondern immer nur den Zeiger, der auf den Anfang des dynamischen Speicherblocks zeigt.

  **Richtig:** ☺

```
int * speicher;

speicher = malloc ( 10 * sizeof int );   /* dynam. Speicher anfordern  */
...
speicher [2] = 3;                        /* Wert eintragen             */
```

  **Falsch:** ☹

```
int * speicher;
int * inhalt;

speicher = malloc ( 10 * sizeof int );   /* dynam. Speicher anfordern  */
...
inhalt = speicher + 2;             /* Zeiger mitten in dynam. Speicher */
...
~~* inhalt = 3;~~                        /* VERBOTEN !!!               */
```

  **Grund:** Wenn der dynamisch reservierte Speicherblock mit `realloc()` vergrößert oder verkleinert wird, kann der Speicherblock dabei auf eine andere Adresse verschoben werden. Wenn die Änderung der Speichergröße erfolgreich war, liefert `realloc()` als Funktionsergebnis einen Zeiger auf die neue Adresse. Ein Zugriff über

```
speicher [2] = 3;
```

  erreicht damit weiterhin den gewünschten Inhalt.

  Zeiger wie `*inhalt` zeigen jedoch nach einer Verschiebung des dynamischen reservierten Speicherblocks durch `realloc()` nicht mehr auf die gewünschte Stelle innerhalb des Speicherblocks.

- Verhindern Sie, daß nach dem Freigeben von dynamischem Speicher nochmals auf den freigegebenen Speicherbereich zugegriffen wird.

  Eine einfache Methode ist, den Zeiger auf den Speicherbereich nach der Freigabe mit `NULL` zu besetzen.

  Beispiel:

```
free ( speicher );  /* dynam. Speicher freigeben */
speicher = NULL;    /* weitere Zugriffe verhindern ☺ */
```

  **Grund:** Bis zum Zugriff kann der freigegebene dynamische Speicherbereich wieder anderweitig belegt sein und völlig andere Daten enthalten.

# Anhang

## A Literaturverzeichnis

[1] Böhm, Oliver: Fehlerfrei programmieren mit C und C++
dpunkt.verlag, Heidelberg 2000, ISBN 3-932588-67-3

frühere Ausgabe: Tewi-Verlag, München 1998, ISBN 3-89362-578-x

[2] Kernighan, Brian W.; Ritchie, Dennis M.: Programmieren in C. Zweite Ausgabe ANSI C
Hanser Verlag 1990, ISBN 3–446–15497–3

[3] Maguire, Steve: Nie wieder Bugs! Die Kunst der fehlerfreien C-Programmierung
Microsoft Press 1995, ISBN 3-86063-345-7

[4] Thielen, David: No Bugs! Delivering Error-Free Code in C and C++
Addison-Wesley 1992, ISBN 0-201-60890-1

[5] van der Linden, Peter: Expert-C-Programmierung
Heise Verlag, Hannover 1995, ISBN 3-88229-047-1

# B Index