

Programmierleitfaden für C

Auszug aus

Schaefer, M.; Gnedina, A.; Bömer, T.; Büllsbach, K.-H.; Grigulewitsch, W.; Reuß, G.; Reinert, D.: **Programmierregeln für die Erstellung von Software für Steuerungen mit Sicherheitsaufgaben**. Schriftenreihe der Bundesanstalt für Arbeitsschutz und Arbeitsmedizin, Dortmund. Wirtschaftsverlag NW, Bremerhaven 1998
(zurzeit vergriffen)

Ansprechpartner:

Berufsgenossenschaftliches Institut
für Arbeitsschutz – BGIA
Zentralbereich
Prof. Dr. Dietmar Reinert
53754 Sankt Augustin
Tel.: 02241 231-2750
Fax: 02241 231-2234
Dietmar.Reinert@HVBG.de



9	PROGRAMMIERLEITFADEN FÜR C	3
9.1	Die Geschichte der Entwicklung	3
9.2	Konzepte und Konstrukte von C	4
9.2.1	Erstellung von C-Programmen	4
9.2.2	Module und Funktionen	5
9.2.3	Datenorganisation	6
9.2.4	Operatoren	8
9.2.5	Kontrollstrukturen.....	10
9.3	Stärken und Schwächen der Sprache	10
9.3.1	Generelle Vorbemerkungen	10
9.3.2	Benutzung dynamischer Objekte (nach [80]).....	12
9.3.3	Rekursionen (nach [80])	12
9.3.4	Mehrfache Ein/Ausgänge (nach [80])	12
9.3.5	Überladen (nach [80]).....	13
9.3.6	Einfluß der Grammatik (nach [80])	13
9.3.7	Probleme mit Pointern (nach [80]).....	13
9.4	Regeln und Einschränkungen im Sprachumfang	14
9.4.1	Generelles.....	14
9.4.2	Dokumentation des Quelltextes	15
9.4.3	Modulare Programmierung	17
9.4.4	Datenorganisation	21
9.4.5	Arithmetische Berechnungen, Operatoren	28
9.4.6	Kontrollfluß	32

9 Programmierleitfaden für C

9.1 Die Geschichte der Entwicklung

Nach der Explosion im Bereich der Programmiersprachen der 60er Jahre kehrten die Sprachentwickler mit einer neuen Wertschätzung der Einfachheit und der Konsistenz einer Sprache zurück. C erzielt die Einfachheit durch die Beschränkung von Ausdrücken, durch Reduzierung der Komplexität des Typsystems und der Laufzeitumgebung und dadurch, dass der Zugriff auf die zugrundeliegende Maschine erleichtert wird. Aus diesem Grund wird C manchmal als Programmiersprache der „mittleren Ebene“, im Gegensatz zu einer Hochprogrammiersprache, bezeichnet. Teilweise ist der Erfolg von C der Popularität des Betriebssystems UNIX zuzuschreiben, das überwiegend in C implementiert wurde. Als Sprache zur Erstellung eines Betriebssystems mußte C eine zu Assembler vergleichbare Flexibilität haben.

C selbst hat wenige neue Konzepte zum Entwurf von Programmiersprachen beigetragen. Ihr Erfolg lag hauptsächlich in ihrer Einfachheit und Gesamtkonsistenz des Entwurfs, vielleicht als Ergebnis der Entwicklung durch eine sehr kleine Gruppe von Mitarbeitern.

Die Sprache C wurde 1972 in den Bell Laboratories in Murray Hill, New Jersey, von D. Ritchie für die Programmierung des Betriebssystems UNIX entwickelt. Vorläufer von C sind die Sprachen BCPL (Basic Combined Programming Language) und B von K. Thompson. Als Nachfolger der Sprache B erhielt die von Ritchie entworfene Sprache den Namen C.

Der erste Versuch der Standardisierung der Programmiersprache C wurde von ihrem Entwickler Dennis M. Ritchie in Form des C Reference Manual vorgestellt. Das Handbuch ist als Anhang von [76] veröffentlicht worden. Für viele Jahre war dieses Buch die einzige Beschreibung der Sprache C. Natürlich haben sich im Laufe der Zeit Änderungen im Sprachumfang ergeben, die nicht von allen Compiler-Herstellern übernommen wurden. Dadurch hat die Portabilität zwischen unterschiedlichen Rechnern, aber auch zwischen unterschiedlichen Compiler-Implementierungen für den gleichen Rechner gelitten.

Das "User-Group"-Komitee stellte 1984 den UNIX 1984 Standard für eine Standardisierung unter dem Betriebssystem UNIX vor. Ein wichtiger Aspekt war dabei die Festlegung einer Grundmenge von Bibliotheksroutinen. Viele der Empfehlungen sind in die System V Interface Definition von AT&T eingegangen, die eine Basis für Zugriffe auf das Betriebssystem und andere Bibliotheksroutinen darstellt.

Ein anderer Versuch C zu standardisieren ist der ANSI C-Standard, der 1983 eingeführt wurde. ANSI C macht die Programmiersprache C leistungsfähiger. Dabei wurden zahlreiche Eigenschaften, die in früheren Versionen nicht möglich waren, hinzugefügt, so z.B. die Möglichkeit, Informationen in den Funktionsheader zu übernehmen, was sowohl die Lesbarkeit als auch die Zuverlässigkeit des Programms verbessert.

Der ANSI-Standard für die Sprache C ist der Versuch, Portabilität, Wartungsmöglichkeit und Effizienz der Sprache C auf den unterschiedlichsten Computersystemen sicherzustellen. Als Grundstein verwendet der Standard das C Reference Manual und den UNIX 1984 Standard. Innerhalb von nur 6 Jahren wurde 1989 der ANSI-Standard [77] von C und bereits ein Jahr später ein wortgleicher ISO/IEC Standard [78] veröffentlicht.

9.2 Konzepte und Konstrukte von C

C bietet die Möglichkeit, fundamentale Objekte der Hardware (einzelne Bits, Worte, Adressen) mittels einfacher Befehle effizient zu manipulieren, wobei die Sprachkonstrukte mit denen anderer Programmiersprachen, wie z.B. Pascal, durchaus vergleichbar sind. Die Intension bei der Entwicklung der Programmiersprache C war, Assemblersprachen, welche zur Programmierung von Betriebssystemroutinen benötigt wurden, weitgehend zu ersetzen, um so die Fehlerhäufigkeit bei der Programmierung zu reduzieren. Ein gutes Beispiel liefert das Betriebssystem UNIX, dessen Code aus über 90% C-Code besteht.

9.2.1 Erstellung von C-Programmen

C-Programme werden in drei Schritten aufgebaut.

Zuerst werden sie mit einem Texteditor im Standard ASCII-Zeichensatz in den Computer eingegeben. Die einzelnen Worte, auch Token¹ genannt, sind mit Leerzeichen zu trennen oder durch Operanden in Klammern einzuschließen. Ausnahmen stellen die primären Operatoren „->“, „.“ und „[]“, Klammern und unitäre Operatoren wie !p, ~b, ++i dar, die grundsätzlich kein Leerzeichen zwischen sich und dem Operanden haben. Einige Token in C, wie z.B. /, * und =, sind nur ein Zeichen lang. Andere C-Token, wie /*. == und die Variablenbezeichnungen, sind mehrere Zeichen lang. Wenn ein C-Compiler auf / trifft, hinter dem ein * steht, dann muß er entscheiden können, ob die zwei Zeichen als zwei getrennte Token oder als ein einziges Token interpretiert werden sollen. C beantwortet diese Frage mit einer einfachen Regel: nimm immer das größtmögliche Stück. Das bedeutet, daß man ein C-Programm dadurch in Token zerlegt, daß man von links nach rechts geht und jedesmal das größtmögliche Token verwendet. Diese Strategie wird hin und wieder auch als habgierig (greedy) bezeichnet. Token (außer Strings und Zeichenkonstanten) enthalten niemals einen Leerraum (Leerzeichen, Tabulatoren oder Zeilenvorschübe). == ist z.B. ein Token, während = = zwei Token sind und der Ausdruck a---b bedeutet dasselbe wie a-- - b und nicht a - --b. Ähnlich wird bei einem /., hinter dem direkt ein * steht, ein Kommentar eingeleitet. Dabei spielt es keine Rolle, in welchem Zusammenhang das Token vorkommt.

Die entstandene Quelldatei erhält einen Dateinamen mit der Erweiterung .c. Diese Quelldatei wird danach vom Compiler zusammen mit den in der Quelldatei durch

¹ Der Begriff Token bezieht sich auf einen Teil eines Programms, der genau dieselbe Rolle wie ein Wort in einem Satz spielt. Die gleiche Zeichenfolge kann in einem bestimmten Zusammenhang zu einem Token gehören und in einem anderen Zusammenhang zu einem anderen Token.

#include aufgerufenen Headerdateien (.h) (die Headerdateien sollen die Deklarationen für ein großes Programm für alle Teildateien einheitlich halten) gelesen, wodurch eine Objektdatei entsteht, eine Übersetzung in Instruktionen, die der verwendete Computer versteht. Diese Objektdatei (und unter Umständen noch weitere Objektdateien) werden vom Binder zusammen mit der Bibliothek der Standardfunktionen von C eingelesen, daraus entsteht dann ein ausführbares Programm.

Ein C-Programm kann aus mehreren Quelldateien zusammengesetzt werden. Jede wird separat kompiliert, die entstandenen Objektdateien werden dann vom Binder zu einem ausführbaren Programm zusammengesetzt.

Zu jedem C-Compiler gehört eine Bibliothek mit den Standardfunktionen. Die C-Programme können diese Funktionen verwenden, der Binder baut sie nach Bedarf in das ausführbare Programm ein. Die Standardbibliotheken setzen sich aus Routinen zusammen, deren Verwendung sich im Laufe der Zeit als sinnvoll herausgestellt hat. Einige der Routinen (z. B. `read()` und `write()`) sind Funktionen der Systemebene und stellen die Verbindung zu dem unter der Sprache liegenden Betriebssystem her. Andere Routinen (z.B. Zeichenkettenfunktionen und Makros) gehören zur externen Unterprogramm-bibliothek.

In den letzten Jahren wurden einige Erweiterungen an den Bibliotheken vorgenommen. Dabei handelt es sich um Pufferverwaltungsroutinen, neue Stringbearbeitungsfunktionen, erweiterte Ein-/Ausgabefehlerbehandlungsroutinen und Routinen für die Arbeit mit Debuggern. Der ANSI-Standard spezifiziert eine Grundmenge von System- und externen Routinen.

Im allgemeinen stellt ein C-System noch eine umfassende Bibliothek von Funktionen zur Verfügung, welche die Ein-/Ausgabe, die Manipulation von Zeichenreihen, die Speicherverwaltung realisieren, Funktionen zum Suchen und Sortieren von Feldern, mathematische Funktionen und vieles mehr umfassen. Die Bibliotheksfunktionen für die Eingabe von der Tastatur bzw. Ausgabe auf dem Bildschirm (in C Konsole genannt) sind Spezialfälle der Funktionen für die Datei-Ein/Ausgabe. Durch Umleitung der Ein/Ausgabe können auch die Funktionen für die Konsol-Ein/Ausgabe für den Zugriff auf Dateien verwendet werden.

Ein lauffähiges C-Programm setzt sich aus verschiedenen Funktionen zusammen, die von einer Hauptfunktion, genannt "`main()`", aufgerufen werden. Wie bei jeder anderen Funktion wird von `main()` erwartet, daß ein Integer-Wert zurückgegeben wird, wenn nicht ausdrücklich der Datentyp des Ergebnisses deklariert wurde. Viele C-Implementierungen verwenden dieses Ergebnis von `main()`, um dem Betriebssystem mitzuteilen, ob das Programm erfolgreich (Rückgabewert 0) abgeschlossen wurde oder nicht.

9.2.2 Module und Funktionen

C-Programme setzen sich aus Funktionen zusammen. In einer Funktionsdefinition wird spezifiziert, welche Variablen erzeugt und verwendet werden und welche Anweisungen ausgeführt werden sollen. Eine einzelne Quelldatei kann mehr als eine dieser Funktionen enthalten. Um Funktionen auszuführen, muß man sie aufrufen. Beim Ausführen eines C-Programms wird die Funktion mit dem Namen `main()` aufgerufen.

Funktionen können erwarten, daß ihnen beim Aufruf Werte übergeben werden. Diese Parameter oder Argumente werden von der aufgerufenen Funktion weitgehend so behandelt wie die in der Funktion selbst definierten Variablen.

Beim Funktionsaufruf darf man die Klammern () nicht vergessen. Im Gegensatz zu manchen anderen Programmiersprachen benötigt C bei einem Funktionsaufruf eine Argumentenliste, auch wenn keine Argumente angegeben wurden. Wenn f also eine Funktion ist, dann ist f(); eine Anweisung, mit der diese Funktion aufgerufen wird, aber f; wird nur die Adresse der Funktion ermitteln, es erfolgt aber kein Aufruf.

Jede Funktion enthält einen Rückgabewert an die aufrufende Instanz, der individuell vereinbart werden kann, standardmäßig aber ein Integer ist. Zum Beispiel bedeutet:

```
float ff();
```

daß der Ausdruck ff() ein float ist, und daraus folgt wiederum, daß ff eine Funktion ist, die einen float liefert. Analog bedeutet

```
float *pf;
```

daß der Ausdruck *pf einen float ergibt, so daß pf ein Zeiger auf einen float darstellt.

Diese Formen werden in Deklarationen genauso miteinander kombiniert, wie in einem Ausdruck.

```
float *g(), (*h)();
```

besagt also, daß *g() und (*h)() float-Ausdrücke sind. Da () gegenüber * den Vorrang hat, bedeutet *g() dasselbe wie *(g()): g ist also eine Funktion, die einen Zeiger auf einen float liefert, und h ist ein Zeiger auf eine Funktion, die einen float zurückgibt.

Die Funktionen sind die Module der Sprache C, mit denen große Problemstellungen in kleinere Einheiten zerlegt werden. Für die Gestaltung einer Funktion in C gelten die allgemeinen Regeln des „Geheimnisprinzips“ nach Kap. 4.2.5.

9.2.3 Datenorganisation

In C gibt es mehrere Datentypen: Zeichen, ganze Zahlen, Gleitkommazahlen. Eine vollständige Auflistung nach [84] gibt Tabelle 1:

Datentyp	andere Bezeichnungen	minimaler Wertebereich	
		von	bis
char	signed char	-128	127
int	signed signed int	-32.768	32.767

Datentyp	andere Bezeichnungen	minimaler Wertebereich	
		von	bis
short	signed short short int signed short int	-32.768	32.767
long	long int signed long signed long int	-2.147.483.648	2.147.483.647
unsigned char	-	0	255
unsigned	unsigned int	0	65.535
unsigned short	unsigned short int	0	65.535
unsigned long	unsigned long int	0	4.294.967.295
enum	-	0	65.535
float ²	-	3.4 ^{+/-38} (7 Stellen)	
double ²	-	1.7 ^{+/-308} (15 Stellen)	
long double ²	-	1.7 ^{+/-308} (15 Stellen)	

Tabelle 1: Datentypen und deren Wertebereiche in C (nach [84], Seite 35)

Variablen müssen explizit vereinbart werden, um dem Compiler ihren Typ mitzuteilen. Beim Vereinbaren einer Variablen wird Speicherplatz für den Wert der Variablen reserviert und ihr Name und Typ dem Compiler mitgeteilt.

Jede Variablendeklaration in C besteht aus zwei Teilen: einem Typ und einer Liste von Angaben, den Deklaratoren, die einem Ausdruck ähnlich sind. Ein Deklarator sieht aus wie ein Ausdruck, der einen bestimmten Typ ergibt. Der einfachste Deklarator ist eine Variable:

```
float f, g;
```

gibt an, daß die Ausdrücke f und g bei der Auswertung den Datentyp float ergeben.

² „Die Genauigkeit von Fließkommawerten ist immer nur angenähert, da die Anzahl der Zeichen zur Beschreibung eines Dezimalwertes nicht mit der Anzahl der Zeichen übereinstimmt, die zur Speicherung der binären Entsprechung benötigt werden. Werte einfacher Genauigkeit können 6 oder 7 Nachkommastellen aufweisen; Werte doppelter Genauigkeit hingegen verfügen über 14 bzw. 15 Stellen.“ (Zitat [84], Seite 35)

Wenn nötig, kann man sich auch die Position einer Variablen im Speicher besorgen. Diese Adresse (&f, &g) oder dieser Zeiger (*f, *g) kann einer Funktion übergeben werden. Es ist also möglich, Variablen zu vereinbaren, die solche Adressen als Werte enthalten.

Beispiel C 1

```
int x, y;
int *px;
int zuweis(x, y)      /* diese Funktion weist y */
{                    /* den Wert von x zu */
    px = &x;         /* die Adresse von x wird px zugewiesen */
    y = *px;         /* der Inhalt der Adresse wird y zugewiesen
*/
}
```

Neben einfachen Variablen kann man auch Felder von Variablen vereinbaren. C kennt nur eindimensionale Arrays und die Größe eines Arrays muß mit einer Konstante zur Compilationszeit festgelegt werden. Das Element eines Arrays kann jedoch ein Objekt mit einem beliebigen Typ sein, darunter auch ein anderes Array. Damit ist es relativ einfach möglich mehrdimensionale Arrays aufzubauen. Der Zugriff auf ein Element eines Feldes geschieht über einen Index. Eine Besonderheit von C ist die Zählung der Elemente in einem Array. Wenn ein Array 10 Elemente enthält, liegen die Indizes zwischen 0 und 9. Ein Array mit 10 Elementen hat ein nulltes Element aber kein zehntes. Nahezu alle Array-Operationen werden in C nur mit Zeigern durchgeführt, auch wenn die Operationen so geschrieben werden, daß sie wie ein Index ausschauen. Das bedeutet, daß jede Index-Operation einer Zeigeroperation entspricht, so daß man das Verhalten von Indizes vollständig mit Hilfe von Zeigern beschreiben kann.

Strukturen erlauben die Kombination von unterschiedlichen Datentypen unter einem einzigen Namen. „Strukturen sind nützlich, um komplizierte Daten zu organisieren, insbesondere in großen Programmen, denn in vielen Situationen erlauben sie, daß man eine Gruppe von zusammengehörigen Variablen als Einheit und nicht separat behandeln kann.“ (Zitat [76] Seite 129) Einer Struktur kann man beliebige Namen (über das "tag") zuordnen. Sobald eine Struktur einen konkreten Namen hat wird für diesen Namen Speicherplatz für alle in der Struktur vereinbarten Datentypen reserviert. Die Varianten oder „unions“ sind eine Abart der Strukturen. Bei einer Union reserviert der Compiler aber nur die Speicherkapazität, die zur Aufnahme des größten deklarierten Datentyps benötigt wird. Dies bedeutet, daß die Mitglieder einer union sich überlappen. Bit-Felder fassen mehrere Objekte in einem einzelnen Maschinenwort zusammen. Es ist durch Bit-Felder möglich einzelne Flags zu definieren.

9.2.4 Operatoren

Die Sprache C enthält folgende (die Tabellen sind [79] entnommen) mathematische, logische, Bit- und Vergleichsoperatoren sowie Zuweisungen:

Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
++	Inkrementieren
--	Dekrementieren
%	Modulus

Tabelle 2: Mathematische Operatoren

Operator	Funktion
<	kleiner als
<=	kleiner gleich
=	gleich
>	größer als
>=	größer gleich
!=	ungleich

Tabelle 3: Vergleichsoperatoren

Nur bei den vier Operatoren `&&`, `||`, `?:` und `,` ist die Reihenfolge der Auswertung festgelegt. `&&` und `||` werten zuerst den linken Operanden aus und dann den rechten, falls dies noch erforderlich ist. Der Operator `?:` nimmt drei Operanden:

`a?b:c` wertet zuerst `a` aus und dann entweder `b` oder `c`, je nachdem, welchen Wert `a` hatte. Der Komma-Operator `,` wertet seinen linken Operanden aus, verwirft das Ergebnis und wertet dann den rechten Operanden aus³.

Alle anderen C-Operatoren werten ihre Operanden in einer nicht festgelegten Reihenfolge aus.

Operator	Funktion
=	Wert zuweisen
+=	<code>a += 5</code> entspricht <code>a = a+5</code>
-=	<code>a -= 5</code> entspricht <code>a = a-5</code>
*=	<code>a *= 5</code> entspricht <code>a = a*5</code>
/=	<code>a /= 5</code> entspricht <code>a = a/5</code>
++	<code>a++</code> entspricht <code>a = a+1</code>
--	<code>a--</code> entspricht <code>a = a-1</code>

Tabelle 4: Zuweisungen

³Ein Komma, das zwei Funktionsargumente trennt, ist kein Kommaoperator. Z.B. werden `x` und `y` in einer nicht definierten Reihenfolge in `f(x,y)` geholt, während dies in `g((x,y))` nicht der Fall ist. In letzterem Beispiel erhält `g` nur ein Argument. Der Wert des Arguments ergibt sich durch die Auswertung von `x`, dessen Ergebnis wieder verworfen wird, und der anschließenden Auswertung von `y`.

Operator	Funktion
&&	logisches UND
	logisches ODER
!	logisches NICHT

Tabelle 5: Logische Operatoren

Operator	Funktion
>>	Addition
<<	Subtraktion
~	Multiplikation
&	Division
	Inkrementieren
^	Dekrementieren

Tabelle 6: Bit-Operatoren

9.2.5 Kontrollstrukturen

In C beendet ein Semikolon Anweisungen. Die geschweiften Klammern { und } dienen dazu, Vereinbarungen und Anweisungen in einem Block zusammenzufassen, der dadurch syntaktisch äquivalent zu einer einzelnen Anweisung wird.

Entscheidungen werden in C durch die if-else-Anweisung, else if Ketten und die switch-Anweisung als besondere Art von Auswahl unter mehreren Alternativen gesteuert.

Als Schleifen stehen als bekannteste Variante die universell einsetzbare for-Anweisung, die while-Schleife und die do-while Schleife zu Verfügung. Bei den ersten beiden Varianten wird die Abbruchbedingung zu Beginn der Schleife überprüft, während die letztere das Abbruch Kriterium erst am Ende der Schleife nach mindestens einem Durchgang abfragt.

Grundsätzlich können Schleifen strukturiert über die Anweisungen „break“ und „continue“ verlassen werden.

Der „goto“- Befehl ermöglicht jederzeit eine nicht strukturierte Beeinflussung des Kontrollflusses. Während mit „break“ immer nur eine Schleife verlassen werden kann, ist es möglich mit der goto-Anweisung auf beliebige Marken zu springen und damit beliebig tief verschachtelte Schleifen zu verlassen.

9.3 Stärken und Schwächen der Sprache

9.3.1 Generelle Vorbemerkungen

Während der letzten Jahre wurde C zur Universalsprache für alle Gebiete, von Betriebssystemen bis zu Anwendungsprogrammen. C stellt sowohl die low-level-Leistungsfähigkeit zur Verfügung, die notwendig ist, um einen effizienten und schnellen Datenaustausch mit der Hardware durchzuführen als auch die modernen Strukturen, Datentypen und Portabilität, wie sie von einer höheren Programmiersprache erwartet werden.

Die Programmiersprache C ist insbesondere auf Rechnern mit dem Betriebssystem UNIX weit verbreitet. Auf Grund dieser Verbreitung wird C oft für Projekte eingesetzt, für die diese Sprache weder gedacht noch besonders geeignet ist. Daher hat

es schon häufig harte und zum Teil ungerechte Kritik an der Sprache C gegeben. Dabei heißt es meist, C sei zu kryptisch und ermögliche keine strukturierte Programmierung. Wie schon das vorherige Kapitel gezeigt hat, kann in C recht sauber modular programmiert werden. Bei großen Projekten können sich jedoch sicherlich Probleme ergeben, da eine modulare und strukturierte Programmierung besondere Anforderungen an die Disziplin der Programmierer stellt. N. Wirth hat in der Zeitschrift *Computerwoche* im November 1989 diese Anforderungen, die C an den Programmierer stellt, wie folgt zusammengefaßt: „Der Vorteil einer echten höheren Programmiersprache, und dazu zähle ich C nicht, ist der, daß die Sprachregeln vom Compiler überprüft werden können... C läßt strukturierte Programmierung zu, genau wie Assemblercode, aber es unterstützt sie nicht. Mehr Disziplin als selbst gute Programmierer aufbringen, ist nötig, um Fehler zu vermeiden.“

Der Sprachumfang von C ist vergleichsweise gering (z.B. im Vergleich zu ADA), was im Hinblick auf den ursprünglichen Einsatzbereich der Sprache günstig ist. C definiert keine Speicherverwaltung, keine Anweisungen zur Ein-/Ausgabe und keine Datei-Zugriffstechniken. Auch Konzepte zur Programmierung paralleler Abläufe sind im Sprachumfang nicht enthalten. So können C-Übersetzer zwar mit relativ geringem Aufwand implementiert werden, der Programmierer erhält aber für diese Konzepte wenig Unterstützung. Auf der anderen Seite führt der geringe Aufwand für die Implementierung von Compilern zu einer weiten Verbreitung von C, so daß sie in den letzten 10 Jahren auf nahezu jeder Ausbildungseinrichtung für Informatikingenieure und Elektroingenieure gelehrt wurde. Der gute Bekanntheitsgrad ist für die Sicherheitstechnik ein Vorteil, da man in einer Sprache, in der man Erfahrung hat, weniger Fehler macht, als in einer neu erlernten Sprache.

Funktionen können rekursiv aufgerufen, aber nicht geschachtelt definiert werden. C ist nicht streng typengebunden und erlaubt relativ viele implizite und explizite Datenumwandlungen. In C ist die getrennte Übersetzung von Programmteilen möglich, wobei die Übersetzungseinheiten Dateien sind.

Grundsätzlich läßt sich sagen, daß der Entwurf von C mehr Wert auf schnelle Übersetzung als auf Fehlerentdeckung legt. Bei größeren Systemen kann es durchaus teuer werden, wenn die Programmierer an Stelle des Übersetzers Syntaxfehler suchen müssen. Dies trifft insbesondere auf Änderungen zu, die nach Abschluß der Programmentwicklung während der Wartung vorgenommen werden sollen.

Die Freiheit von C zwingt zur Aufstellung eigener Regeln und Konventionen, wie zum Beispiel Disjunktheit der Namensräume für verschiedene Module oder konsequente Verwendung von `static` zur Einschränkung der Sichtbarkeit von Namen. Nur bei Einhaltung solcher Regeln (siehe Kap. 9.4) ist die Definition und Implementierung abstrakter Datentypen in C möglich, ohne allerdings die Sicherheit gegen unberechtigte Benutzung des abstrakten Typs zu erreichen, die z.B. ADA bietet. Bei mehreren Programmierern dürfte die Einhaltung dieser Regeln nicht mehr gewährleistet sein, da sie nicht automatisch durch den Übersetzer geprüft werden können.

Die Popularität von C ist vermutlich zum Teil darauf zurückzuführen, daß man kleine Programme „quick und dirty“ in C programmieren kann. Hierbei hilft auch die im allgemeinen umfangreiche Bibliothek. Ganz so schnell lassen sich ADA-Programme

vielleicht nicht schreiben, sie sind aber auch lesbarer, robuster und enthalten erfahrungsgemäß weniger Fehler als vergleichbare C-Programme.

9.3.2 Benutzung dynamischer Objekte (nach [80])

Die Funktionen „malloc“ und „free“ stellen Speicherplatz auf dem Stapelspeicher zur Verfügung. Dabei sind deren Standardfunktionen stark optimiert, so daß bei Fehlern geringe oder überhaupt keine Informationen außer einem klassischen Speicherabzug ("dump") gegeben werden. Probleme mit dynamischen Objekten findet man häufig in graphischen Benutzerschnittstellen. Sie führen in der Regel zu einem völligen Absturz des Rechnersystems. Eine der Ursachen ist die mangelnde Freigabe von nicht mehr benötigtem Speicherplatz oder die Freigabe von nicht zugewiesenen Speicher. Derartige Fehler können durch Werkzeuge wie dbmalloc (erhältlich im FTP Archive der University of Kent at Canterbury, UK) aufgedeckt (debugged) werden. In jedem Fall sollte eine Plausibilitätskontrolle bei der Zuweisung von Speicherplatz eingefügt werden (z.B. `if ((mp = malloc(sizeof(object))) == NULL) { Zuweisung verweigert mit entsprechender Fehlerreaktion}`).

9.3.3 Rekursionen (nach [80])

C-Funktionen können sich selbst aufrufen. Dazu muß die Funktion ihre lokalen Variablen dynamisch speichern. Falls dabei kein weiterer dynamischer Speicher vorhanden ist, führt dies zu einem katastrophalen Fehler, da C keine Fehlerbehandlung für Rekursionsfehler hat. Trotzdem können Rekursionen oft kompakter, wesentlich leichter zu schreiben und zu verstehen sein als eine nicht rekursive Implementierung. Insbesondere bei rekursiv definierten Datenstrukturen wie Bäumen empfiehlt sich der Einsatz rekursiver Algorithmen [76]. Zur Vermeidung der o.g. Probleme sollten in sicherheitsrelevanten Projekten die dynamischen Speicheranforderungen der Rekursion bestimmt und der erforderliche Speicher garantiert werden.

9.3.4 Mehrfache Ein/Ausgänge (nach [80])

C erlaubt es Funktionen an beliebiger Stelle zu verlassen. Dies kann dann vorteilhaft sein, wenn es sich um tief verschachtelten Code handelt. Für sicherheitstechnischen C-Code sollte die zyklomatische Komplexität pro Funktion nicht größer als 10 sein⁴, so daß die Forderung eines Einganges und eines Ausganges mit max. einem zusätzlichen Fehlerausgang eingehalten werden kann.

⁴ Hatton [80] schlägt auf Seite 203 vor: „ Functions with a cyclomatic complexity greater than 10 should have mandatory manual inspection to determine the reason for so many decisions, unless the decisions correspond largely to the presence of case statements. Components with a cyclomatic complexity greater than around 30 should only be allowed with explicit sign-off by the development manager responsible, as such components are strongly associated with unreliability in the life cycle. Again, the presence of case statements should be taken into account. ... Functions with a static path count of more than 1000 should be subject to manual inspection, and again, only allowed with explicit sign-off.“

Der Inhalt von i hängt in diesem Beispiel entscheidend von der Maschine ab und kann u.U. überhaupt nicht verarbeitet werden [80].

Grundsätzlich ist es in C möglich Zeiger auf Zeiger usw. zu bilden. Derartige Konstrukte sind sehr schwer zu verstehen und mehr als zwei Ebenen von Zeigern sollten deshalb nicht benutzt werden. Zusätzlich sollten Plausibilitätsabfragen (assertions) und der Einsatz dynamischer Analyse-Werkzeuge⁵ Probleme zur Laufzeit abfangen.

9.4 Regeln und Einschränkungen im Sprachumfang

Neben den folgenden Regeln und Einschränkungen sind alle Regeln des allgemeinen Teils (siehe Kapitel 4) zu beachten.

Die meisten der Regeln und Einschränkungen sind auf Grund von [80], [82], [75] und [83] entstanden.

9.4.1 Generelles

C 1: Einsatz eines Syntax-Checkers

Der Quellcode sollte mit einem Syntaxchecker, vergleichbar C-lint⁶, mit dem höchsten Level, das noch Warnungen erzeugt, überprüft werden.

Derartige Syntaxchecker überprüfen die meisten der unten aufgeführten Regeln. Nach [80] erlauben sie auch Fehler durch undefiniertes Verhalten nach der Portierung von C-Programmen (siehe Anhang G.2 aus [78]) weitgehend zu vermeiden. Meldungen des Syntaxcheckers sollten zu Änderungen im Programm oder schriftlichen Begründungen im Programmtext führen.

C 2: Regeln zur Erstellung von include-Dateien

Es sollten möglichst wenige eigene include-Dateien erstellt werden. Dabei sollten nicht gebrauchte Deklarationen möglichst nicht inkludiert werden. Es darf keine Ketten von Inclusionen geben. Die Schnittstellen- und Implementations-Deklarationen sind zu trennen.

⁵ Das Werkzeug Logicope der Firma Verilog ermöglicht dynamische Analysen für nicht Echtzeit-Software in den unterschiedlichsten Sprachen. Ein C-spezifisches Werkzeug ist QA C Dynamic des Productes Safe C [81].

⁶ Die Verfasser verwenden je nach Controller C-lint der Firma Keil, für PC-Software PC-lint der Firma Gimpel Software, Collegeville, PA 19426, USA, jeweils mit den Warning-Levels W3.

☞ C 3: Gemeinsame include-Datei pro Projekt

Alles was von generellem Interesse im Projekt ist, sollte durch `#include` und `#define` in einer gemeinsamen include-Datei vereinbart werden. Alle Programme in diesem Projekt sollten sich auf diese gemeinsame include-Datei beziehen. Alle projektspezifischen include-Dateien sind über `#include „datei.h“` einzubinden, alle anderen Dateien über `#include <datei.h>`.

Diese Regel erhöht die Wartungsfreundlichkeit und die Konsistenz im Projekt.

☞ C 4: include-Dateien am Beginn einer Datei

Jede Datei sollte zu Beginn alle benötigten `#include`-Dateien einbinden. Danach stehen die für diese Datei lokalen `#defines`, erst dann folgen Deklarationen und Funktionsdefinitionen.

Die include-Dateien stellen den Kontext des Programmcodes dar.

☞ C 5: Keine Doppeldefinitionen durch `#ifndef`

Jede include-Datei muß mit einem `ifndef` beginnen, das testet ob einige `#define` Symbole bereits definiert sind, und enden mit einem `#endif`.

Beispiel C 3

```
local.h:
    #ifndef LOCAL_H
    #define LOCAL_H

    .... Definitionen

    #endif
```

☞ C 6: Keine Initialisierung in include-Dateien

Include-Dateien dürfen keine Initialisierungen enthalten.

Es bleibt bei Mißachtung der Regel unklar, welcher Funktion die initialisierten Daten "gehören". Die definierende Instanz ist ebenfalls nicht lokalisierbar.

9.4.2 Dokumentation des Quelltextes

☞ C 7: Verbot verschachtelter Kommentare

Verschachtelte Kommentare sind verboten.

Nicht alle C-Compiler erlauben verschachtelte Kommentare.

☞ C 8: Regeln zur Variablendeklaration

Grundsätzlich sind Variablennamen explizit zu deklarieren und in kleinen Buchstaben zu schreiben. Die Reihenfolge sollte sein: externe Variablen, andere Variablen (jeweils pro Typ alphabetisch geordnet). Initialisierungen sind über das Gleichheitszeichen mit einer Zuordnung pro Zeile vorzunehmen.

☞ C 9: Definition von Konstanten

Konstanten, die sich im Lebenszyklus der Software ändern können, sind über `#define` und in Großbuchstaben zu deklarieren. Konstanten, die nur in einer Datei benötigt werden, sollen direkt in dieser Datei definiert werden, alle anderen Konstanten in einer include-Datei.

Typische Konstanten, z.B. `BUFSIZ`, `NULL`, `EOF` aus `<stdio.h>`, sollten bevorzugt benutzt werden. Alle nicht typischen Konstanten gehören in den nutzerspezifischen Standard-header.

☞ C 10: Dokumentation fehlender break-Anweisungen bei switch

Das Fehlen der `break`-Anweisung bei `switch`-Anweisungen muß kommentiert sein.

(Siehe Kap. 9.4.6 und die Beispiele daraus).

☞ C 11: Bedingung bei der if-Anweisung klar beschreiben

Bei der `if`-Anweisung ist die Überprüfung, ob der Ausdruck in Klammern Null ergibt oder nicht, zu verdeutlichen. Dabei muß man die Rangfolge der Operatoren beachten (am besten die Ausdrücke in Klammern einschließen (siehe auch ☞ C 45)).

Angenommen, die Konstante `FLAG` ist als Integer definiert, bei der genau ein Bit in der Binärdarstellung gesetzt ist, und man will prüfen, ob dieses Bit in der Integervariable `flags` gesetzt ist, dann schreibt man dies häufig so:

```
if (flags & FLAG)
```

Aus Gründen besserer Lesbarkeit und Transparenz soll man diese Überprüfung folgendermaßen verdeutlichen:

```
if ((flags & FLAG) != 0) ...
```

Dabei muß man die Rangfolge der Operatoren beachten, die Anweisung ohne innere Klammern

```
if (flags & FLAG !=) ...
```

bedeutet

```
if (flags & (FLAG != 0)) ...
```

was nicht gemeint war.

9.4.3 Modulare Programmierung

C 12: Zyklomatische Zahl für Funktionen möglichst kleiner 10

Funktionen mit der zyklomatischen Zahl größer 10 müssen manuell überprüft werden (Code-Inspektionsverfahren), wenn die große Zahl der Verzweigungen sich nicht durch case-Anweisungen erklärt [80]. Vermieden werden sollten (außer bei Verwendung der switch-Anweisung) Funktionen mit einer zyklomatischen Zahl größer 30. Die Mehrzahl der Funktionen sollte aus weniger als 50 Zeilen Quelltext bestehen⁷.

Allein das Prinzip der Modularität erfordert eine Beschränkung der Komplexität einer Funktion. Die 50 Zeilen Forderung entstammt nach [83] empirischen Beobachtungen aus "well-crafted C code". Eine künstliche Verkleinerung von Funktionen nur um das Kriterium zu erfüllen trägt nicht unbedingt zur Übersichtlichkeit und damit zur Sicherheit des Programmcodes bei.

C 13: Verwendung von Prototypen zur Funktionsdeklaration

Eine Funktion muß man so deklarieren, wie sie verwendet wird. Jede Funktion muß über Prototypen deklariert werden. Eine implizite Funktionsdeklaration ist nicht erlaubt. Der Rückgabotyp einer Funktion muß stets deklariert werden. Bei der Deklaration von Funktionszeigern empfiehlt sich der Einsatz von typedef.

Nur durch die Verwendung von Funktionsprototypen lassen sich zahlreiche Interface-Inkonsistenzen vermeiden. [80] gibt alternativ mit Tabelle 4.5 eine Liste von 46 Punkten, die explizit verboten werden müssen, wenn auf das Prototyping verzichtet wird.

Beispiel C 4

```
/* Prototyp für die Sicherheitsfunktion NOT-AUS */
int SiNotaus ( int nstop,           /* Stopflag */
               long ltime,         /* Zeitfenster */
               int maschine1,      /* Maschine1 */
               int maschine2,      /* Maschine2 */
               int maschine3);     /* Maschine3 */
```

C 14: Benutzungsverbot der Funktionen: setjmp, longjmp, offsetof und signal

Da die Standard Bibliotheksfunktionen setjmp, longjmp, offsetof und signal nur sehr wage definiert sind, sollten sie, aus Gründen der Portierbarkeit von C-Programmen, nicht oder nur in begründeten Einzelfällen verwendet werden [80].

⁷ [80] schlägt für Funktionen mit einer Anzahl von statischen Pfaden größer 1000 eine manuelle Inspektion vor.

☞ C 15: Benutzungsverbot der Funktionen `system`, `#pragma`, `volatile` und `register`

Die Funktionen `system`, `#pragma`, `volatile` und `register` können zu Fehlern bei der Implementierung führen und sollten deshalb nicht benutzt werden.

Speziell `system` sollte aus Gründen der Portierbarkeit nicht benutzt werden.

☞ C 16: Neudefinition problematischer Standardfunktionen

Folgende Standard-Funktionen sind nicht klar definiert und sollten vermieden oder konsistent neu definiert werden: `ungetc`, `fopen`, `fgetpos`, `ftell`, `remove`, `rename`, `bsearch`, `qsort`, `time`, `date`, `clock`, `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, `isupper`, `perror`, `strerror`, `calloc`, `malloc`, `realloc`, `exit`, `fmod` und alle mathematischen Gleitkomma-Funktionen.

Als Beispiele für eine konsistente Neudefinition schlägt [80] z.B. vor: Rückgabe eines vereinbarten Wertes für jeden erfolgreichen Aufruf von `fgetpos` bzw. Überprüfung des Nullzeigers oder eine Anforderung 0 Bytes zuzuweisen mit einer definierten Antwort bei der Benutzung von `malloc`. Die Beispiele zeigen, daß für jede der Funktionen entsprechende Plausibilitätskontrollen (`assertions`) umgesetzt werden sollten.

☞ C 17: Keine variable Argumentenanzahl bei Funktionen

Funktionen dürfen keine variable Anzahl von Argumenten besitzen. Die Anzahl der Argumente sollte möglichst gering sein (Anhaltspunkt: kleiner sechs).

Die Übergabe von Argumenten an eine Funktion sollte übersichtlich und klar definiert sein. Die variable Anzahl von Argumenten schafft Verständnis- und Portabilitätsprobleme.

☞ C 18: Kein Adressoperator für die Argumente von Funktionen

Der Adressoperator darf nicht auf Funktionsparameter angewendet werden.

`char` und `short` werden z.B. häufig vor der Übergabe an eine Funktion auf `int` aufgeweitet. Sie liegen bei der Übergabe also physikalisch anders vor als sie in der Funktionsdeklaration vereinbart wurden.

☞ C 19: Nur ein `return` pro Funktion

Pro Funktion darf nur eine `return`-Anweisung eingesetzt werden. Die `return`-Anweisung sollte am Ende der Funktion stehen.

Diese Regel dient der Übersichtlichkeit und Eindeutigkeit beim Verlassen einer Funktion.

☞ C 20: Rückgabewerte von Funktionen immer überprüfen

Rückgabewerte von Funktionen sollten immer abgefragt und entsprechende Reaktionen programmiert werden.

Es darf nicht sein, daß eine Funktion im Rückgabewert ein Problem zu erkennen gibt und das Programm darauf nicht reagiert.

☞ C 21: Klammern aller Parameter und der Definition bei Makros

Sowohl jeder Parameter in einer Makrodefinition als auch die Makrodefinition selbst müssen eingeklammert sein. Dabei muß man sicherstellen, daß die Argumente im Makro keine Seiteneffekte enthalten.

Man sieht oft Formulierungen wie:

```
#define abs(x) ((x)>=0)?(x):- (x)
```

oder

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

Die Klammern in diesen Makros verhindern mögliche Probleme in bezug auf die Rangstellung. Nehmen wir z.B. an, daß abs folgendermaßen definiert ist:

```
#define abs(x) x>0?x:-x
```

Der Ausdruck `abs(a-b)` ergibt in diesem Fall `a-b>0?a-b:-a-b` was natürlich ein falsches Ergebnis ist: der Teilausdruck `-a-b` entspricht `(-a)-b` und nicht dem beabsichtigten `-(a-b)`.

Auch wenn Makrodefinitionen voll geklammert sind, wird trotzdem jeder Operand, der zweimal verwendet wird, auch zweimal ausgewertet. Im Ausdruck `max(a,b)` wird `a`, wenn `a` größer als `b` ist, zweimal ausgewertet: einmal im Vergleich und dann noch einmal, um den Wert zu berechnen, der sich in `max` ergibt.

Dies kann nicht nur ineffektiv, sondern es kann auch falsch sein, wie folgendes Beispiel zeigt:

Beispiel C 5

```
groesster = x[10];
i = 1;
while (i<n)
  groesster = max(groesster, x[i++]);
```

Diese Befehlsfolge würde ausgezeichnet funktionieren, wenn `max` eine echte Funktion wäre, aber sie ist fehlerhaft, wenn `max` ein Makro ist. Um dies zu verstehen, wollen wir einige Elemente von `x` initialisieren:

```
x[0] = 2;
x[1] = 3;
x[2] = 1;
```

Im ersten Durchlauf der Schleife wird die Zuweisung zu

```
groesster = ((groesster)>(x[i++])
             ?(groesster):(x[i++]));
```

erweitert. Zuerst wird groesster mit x[i++] verglichen. Da i den Wert 1 hat und x[1] den Wert 3 hat, ist dieser Ausdruck falsch. Als Seiteneffekt ergibt sich, daß i nun 2 wird. Da die Relation falsch ist, erhält groesster nun den Wert von x[i++]. i ist jedoch 2, so daß groesster den Wert von x[2] erhält, in dem 1 steht und i ist jetzt 3. Um sich dieser Sorgen zu entledigen, muß man sicherstellen, daß die Argumente im Makro max keine Seiteneffekte enthalten:

```
groesster = x[0];
for (i = 1; i < n; i++)
    groesster = max(groesster, x[i]);
```

☞ C 22: Ausdrücke und nicht Anweisungen

Bei Makrodefinitionen muß man Ausdrücke Anweisungen vorziehen.

Das Argument im Makro

```
assert (x>y);
```

ist ein Ausdruck. Wenn dieser Ausdruck Null ergibt, wird die Ausführung des Programms mit einer entsprechenden Fehlermeldung abgebrochen. Indem man assert als Makro realisiert, kann der Dateiname und die Zeilennummer in der Fehlermeldung aufgenommen werden. Mit anderen Worten,

```
assert (x>y);
```

sollte überhaupt nichts machen, wenn x größer als y ist. Ansonsten aber soll das Programm abgebrochen werden.

Die richtige Methode ist, das Makro von assert wie einen Ausdruck und nicht wie eine Anweisung zu schreiben:

```
#define assert (e) \
    ((void) ((e) || _assert_error(_FILE_, _LINE_)))
```

Diese Definition beruht auf der sequentiellen Auswertung des Operators ||. Wenn e zutrifft, dann kann der Wert von

```
(void) ((e) || _assert_error(_FILE_, _LINE_))
```

bestimmt werden, ohne daß man

```
_assert_error(_FILE_, _LINE_)
```

auswerten muß. Wenn e aber nicht zutrifft, muß

```
_assert_error(_FILE_, _LINE_)
```

ausgewertet werden. Der Aufruf von assert_error gibt die Meldung aus, daß die Behauptung nicht zutrifft.

9.4.4 Datenorganisation

C 23: Eindeutige Definition globaler Objekte

Jedes globale (externe) Objekt darf nur einmal definiert sein. Es ist darauf zu achten, daß globale Objekte nicht durch lokale verdeckt werden.

Jedes globale Objekt muß irgendwo einmal definiert werden.

Die Deklaration

```
extern int a;
```

ist jedoch keine Definition von `a`. Dadurch, daß das Schlüsselwort `extern` angegeben wurde, wird zwar gesagt, daß `a` eine externe Integervariable ist. Vom Standpunkt des Linkers aus, ist eine solche Deklaration aber eine Referenz auf ein externes

Objekt `a`, die aber an dieser Stelle nicht definiert wird. Aus diesen Gründen muß in einem Programm, in dem

```
extern int a;
```

vorkommt, auch irgendwo

```
int a;
```


stehen. Das kann in derselben Programmdatei oder in einer anderen Datei sein.

Wenn `int a;` in zwei oder mehreren verschiedenen Quelldateien vorkommt, oder wenn z.B. `int a = 7;` in der einen Datei steht und `int a = 9;` in der anderen, lassen die meisten Compiler so ein Programm nicht zu.

Wenn eine externe Variable in mehreren Dateien ohne einen Anfangswert definiert ist, dann akzeptieren einige Systeme das Programm, während dies bei anderen nicht der Fall ist.

C 24: Gemeinsame externe Objekte mehrerer Funktionen als static in eine Datei

Wenn es mehrere Funktionen gibt, die sich dieselben externen Objekte miteinander teilen, dann sollte man alle Funktionen in eine einzige Datei stecken und die benötigten Objekte in derselben Datei als `static` deklarieren.

Dadurch kann man ungewollte Überschneidungen von vornherein ausschließen (siehe auch  C 23.). Die Deklaration `static int a;` bedeutet dasselbe wie `int a;` innerhalb derselben Quelldatei, mit dem Unterschied, daß `a` vor anderen Dateien versteckt ist.

C 25: Definition aller externen Objekte in einer header-Datei

Die Stelle, wo man ein externes Objekt deklariert, sollte eine Deklarationsdatei (header-Datei) sein, die von jedem Modul eingelesen wird, in dem dieses externe

Objekt verwendet wird. Insbesondere sollte diese Datei auch von dem Modul eingelesen werden, in dem das Objekt definiert wird.

Wenn z.B. `filename` ein Teil eines Programms ist, das aus mehreren Modulen besteht, in denen jeweils der Name einer bestimmten Datei bekannt sein muß und der Dateiname bei einer Änderung in jedem Modul verändert wird, die Änderung aber nur an einer einzigen Stelle erfolgen soll, erreicht man dies, indem man eine Datei `file.h` erstellen, die folgende Deklaration enthält: `extern char filename[]`; Jede C-Quelldatei, die auf dieses externe Objekt zugreifen will, sollte folgende Anweisung enthalten: `#include "file.h"`

In einer C-Quelldatei kann `filename` ein Anfangswert zugewiesen werden. Dies sei in diesem Beispiel die Datei `file.c`:

```
#include "file.h"
char filename[] = "/etc/passwd";
```

Man muß beachten, daß `file.c` eigentlich zwei Deklarationen für `filename` enthält, nachdem die `include`-Anweisung ausgeführt wurde:

```
extern char filename[];
char filename[] = "/etc/passwd";
```

Dies ist dann zulässig, wenn alle Deklarationen übereinstimmen und zumindest eine der Deklarationen eine Definition ist.

C 26: Keine Neudefinition von existenten Datentypen

C Standard Datentypen dürfen nicht durch eigene Definitionen neu implementiert werden. Eine Definition unter neuem Namen dagegen ist möglich.

C 27: Definition eines neuen Datentyps für besonders große Integer

Besonders große Integer muß man als neuen Datentyp definieren.

Die meisten modernen Maschinen verwenden 16-Bit-Zeichen. Es gibt jedoch eine wachsende Anzahl an Implementierungen mit 32-Bit-Zeichen, damit größere Zeichensätze unterstützt werden können.

Der ANSI-Standard fordert, daß ein `long`-Integer mindestens 32 Bit groß ist und `short` oder normale Integer mindestens 16 Bit lang sind. Da die meisten Maschinen 16-Bit-Zeichen verwenden und die einfachsten Integergrößen für solche Maschinen 16 und 32 Bit sind, werden diese Grenzwerte auch von fast allen der älteren C-Compiler erfüllt.

In der Praxis bedeutet das, daß man sich nicht auf eine bestimmte Genauigkeit verlassen kann. Die portabelste Lösung ist, daß man eine große Variable als einen neuen Datentyp definiert:

```
typedef long zehnmil;
```

Nun kann man diesen Typ für die Deklaration aller Variablen dieser Größe verwenden und weiß ganz sicher, daß man nur eine einzige Typdefinition ändern muß, um alle Variablen anzupassen, falls dies erforderlich ist.

☞ C 28: Benutzung von short und long

Der Typ int sollte ausschließlich für Rückgabewerte von Funktionen oder für register-Variablen verwendet werden. In allen anderen Fällen sind die Typen short (16 bits oder mehr) und long (32 bits oder mehr) zu verwenden.

Die unreflektierte Benutzung des Typs int führt zu maschinenabhängigem Verhalten bei C Programmen. Bei der ausnahmsweise Verwendung von int sollte man nie mehr als 16 Bits Verarbeitungsbreite annehmen.

☞ C 29: Globale Datenstrukturen über structs darstellen

Komplexere globale Datenstrukturen sollen über struct-Konstrukte dargestellt werden, in denen die Daten die von der Aufgabenstellung zusammengehören auch zusammengefaßt sind.

☞ C 30: Variablen vom Typ union meiden

Variablen vom Typ union sollten möglichst nicht eingesetzt werden.

Wie in Kap. 9.2.3 erläutert, reserviert der Compiler für Varianten (unions) nur den Speicherplatz für die Aufnahme des größten deklarierten Teilnehmers einer Varianten. Das Programm muß wissen welcher Teilnehmer der Varianten derzeit aktiv ist und welche Größe der Teilnehmer hat, sonst kommt es u.U. zu Laufzeitfehlern. [84]

☞ C 31: Zeiger und Array-index Adressierung nicht mischen

Bei der Verwendung von Arrays dürfen die Adressierung einzelner Elemente über Zeiger und über Indizes nicht gemischt werden.

☞ C 32: Simulation mehrdimensionaler Arrays mit Zeigern

Mehrdimensionale Arrays sind mittels Zeigern zu simulieren.

Jede Index-Operation kann mit Hilfe von Zeigern beschrieben werden.

Mit der Array-Operation kann man nur die Größe bestimmen und einen Zeiger auf das Element 0 des Arrays ermitteln. Array-Operationen und deren entsprechende Zeigeroperationen sind austauschbar. Die Indizierung ist in den meisten anderen Sprachen vorhanden. In C ist sie über die Zeigerarithmetik definiert.

Beispiel C 6

```
int kalender [12] [31];
int *p;
int i;
```

Dies sagt aus, daß kalender ein Array von 12 Arrays mit jeweils 31 int-Elementen ist (und kein Array von 31 Arrays mit jeweils 12 int-Elementen), so daß sizeof(kalender) als Ergebnis 372 (31*12) mal sizeof(int) ergibt.

Wenn der Bezeichner kalender in einem anderen Zusammenhang als Operand von sizeof verwendet wird, wird er fast immer in einen Zeiger auf das Anfangselement von kalender umgewandelt.

Da kalender ein Array mit jeweils 31 int-Elementen ist, ist kalender [4] einfach das Element 4 dieses Arrays. Kalender [4] ist also eines der 12 Arrays mit 31 int-Elementen und verhält sich genauso. Zum Beispiel ergibt sizeof (kalender [4]) 31-mal die Größe eines int und nach einer Zuweisung wie

```
p = kalender [4];
```

zeigt p auf das Element 0 des Arrays kalender [4].

Da kalender [4] ein Array ist, kann es mit einem Index versehen werden:

```
i = kalender [4] [7];
```

Diese Anweisung entspricht wiederum genau dem Ausdruck:

```
i = *(kalender [4] + 7);
```

was seinerseits mit

```
i = (*(kalender + 4) + 7);
```

äquivalent ist.

In diesem Fall ist die Schreibweise mit eckigen Klammern deutlich einfacher.

Die Zuweisung `p = kalender;` ist nicht zulässig, da kalender ein Array von Arrays ist. Durch die Verwendung der Bezeichnung kalender wird es in einen Zeiger auf ein Array umgewandelt. Da p ein Zeiger auf ein int ist, wird mit der Zuweisung versucht, einen Zeiger mit einem bestimmten Datentyp an einen Zeiger mit einem anderen Datentyp zuzuweisen.

Dagegen zeigt monatp nach:

```
int kalender [12] [31];
int (*monatp) [31];
monatp = kalender;
```

auf das erste der 12 Arrays mit 31 Elementen, aus denen sich kalender zusammensetzt.

Da monatp auf ein Array mit 31 int-Werten zeigt, dann kann man mit monatp wie mit jedem anderen Zeigern durch kalender wandern:

Beispiel C 7

```
int (*monatp) [31];
for (monatp = kalender;
```

```

        monatp < &kalender [12]; monatp++)
/* Monat verarbeiten */

```

☞ C 33: Keine Abhängigkeit von der Datengröße

Ein Programm darf bei der Zuordnung von speziellen Bits in Variablen nicht von der Datengröße der Variablen abhängig sein. Generell sollte auf die direkte Manipulation von Einzelbits verzichtet werden bzw., wo unbedingt erforderlich, Bit-Felder in C eingesetzt werden. Eine separate Dokumentation ist in diesem Fall erforderlich.

Beispiel C 8 (siehe [83])

so:

```
b &= ~1;          /* schaltet die niederwertigen Bits von b aus */
```

aber nicht so:

```
b &= 0177776;    /* schaltet auf 32-bit Rechnern */
                /* auch höherwertige Bits aus */
```

☞ C 34: Unsigned Variablen bei rechts Shift

Wenn die freiwerdenden Bits in einer Verschiebung nach rechts gestört werden, muß die fragliche Variable als unsigned deklariert sein.

Wenn das Objekt, das verschoben wird, ein Vorzeichen hat, dann kann die Implementierung die freiwerdenden Positionen entweder mit Nullen oder mit Kopien des Vorzeichenbits ausfüllen. Wenn das Objekt unsigned ist, werden Nullen nachgeschoben. Aus diesen Gründen muß man das Objekt als unsigned deklarieren. Man darf dann davon ausgehen, daß die freiwerdenden Bits unabhängig von der Implementierung auf Null gesetzt werden.

☞ C 35: Benutzung der doppelten Anführungszeichen bei einfachen Zeichen

Die Benutzung von den einfachen ('a' - Zeichen) anstatt der doppelten („a“ - String) Anführungszeichen und umgekehrt muß in C überprüft und streng begründet sein. Die doppelten Anführungszeichen sind den einfachen vorzuziehen.

Ein Zeichen, das in einfachen Anführungszeichen steht, ist lediglich eine andere Methode, um den Integer-Wert anzugeben, der dem jeweiligen Zeichen in der Sortierreihenfolge der Implementierung entspricht. In einer ANSI-Implementierung bedeutet 'a' also genau dasselbe wie 0141 oder 97. Ein String, der in doppelten Anführungszeichen steht, ist andererseits nur eine abgekürzte Schreibweise für einen Zeiger auf das erste Zeichen eines namenlosen Arrays. Dieses wird mit den Zeichen zwischen den Anführungszeichen und einem zusätzlichen Zeichen, dessen Wert 0 beträgt, initialisiert. Die Anweisung

```
printf ("Hallo Welt\n");
```

entspricht also

```
char hallo[] = { 'H', 'a', 'l', 'l', 'o', ' ',
```

```
        'w', 'e', 'l', 't', '\n', 0 };
printf (hallo);
```

Da ein Zeichen in einfachen Anführungszeichen einen Integerwert darstellt und ein Zeichen in doppelten Anführungszeichen einen Zeiger darstellt, erwischt die Datenüberprüfung üblicherweise alle Stellen, an denen ein Datentyp anstelle eines anderen verwendet wird. So ergibt zum Beispiel die Anweisung

```
char *strich = '/';
```

eine Fehlermeldung, da '/' kein Zeiger auf ein Zeichen ist. Einige Implementierungen führen jedoch keine Überprüfung des Datentyps bei den Argumenten durch, insbesondere nicht bei den Argumenten von printf. Deshalb endet

```
printf ('\n')
```

anstelle von

```
printf ("\n")
```

zur Laufzeit möglicherweise mit einer unliebsamen Überraschung anstatt mit einer Compilermeldung.

Da ein Integer normalerweise groß genug ist, um mehrere Zeichen aufzunehmen, erlauben einige C-Compiler mehrere Zeichen in einer Zeichenkonstante oder in einer String-Konstante. Das bedeutet, daß man auch 'ja' anstelle von „ja“ schreiben kann, ohne daß dies entdeckt wird. Letzteres bedeutet „die Adresse der ersten von drei aufeinanderfolgenden Speicherstellen, die die Zeichen j, a und das NUL-Zeichen enthalten“. Die Bedeutung das 'ja' ist nicht exakt definiert, aber viele C-Implementierungen interpretieren es als „eine Integer-Zahl, die sich irgendwie aus den Werten der Zeichen j und a zusammensetzt“. Jede Ähnlichkeit zwischen den beiden Größen ist rein zufällig.

C 36: Eingeschränkte Verwendung von Null-Zeigern

Null-Zeiger darf man nur in einer Zuweisung oder innerhalb eines Vergleichs verwenden. Dabei soll man testen, ob die Implementierung das Lesen der Speicherstelle 0 zuläßt.

Ein Null-Zeiger zeigt auf kein Objekt. Beispielsweise ist der Wert von strcmp(p,q) nicht definiert, wenn p oder q Nullzeiger sind. Was in diesem speziellen Fall tatsächlich passiert, hängt von der C-Implementierung ab. Einige Implementierungen weisen einen hardwaremäßigen Zugriffsschutz für die Speicherstelle Null auf. Ein Programm, das einen Nullzeiger in einer solchen Implementierung mißbraucht, wird sofort abstürzen. Andere Implementierungen scheinen auf einen String zu zeigen, in dem normalerweise nur rein zufällige Werte stehen. Wieder andere erlauben sogar, daß die Speicherstelle Null sowohl beschrieben als auch gelesen werden darf. Beim Mißbrauch des Nullzeigers kann man auf so einer Implementierung das Betriebssystem überschreiben.

Die Konsequenzen aus dem Mißbrauch des Nullzeigers sind also in keinem C-Programm definiert. Die einfachste Methode, um diese Probleme zu entdecken, ist,

daß man die Programme auf einer Maschine testet, die das Lesen der Speicherstelle 0 nicht zuläßt. Das folgende Programm merkt sofort, wie eine Implementierung die Speicherstelle Null behandelt:

Beispiel C 9

```
#include <stdio.h>

main()
{
    char *p;

    p = NULL;
    printf("Speicherstelle 0 enthält %d\n", *p);
}
```

Dieses Programm stürzt auf einer Maschine, die das Lesen der Speicherstelle Null nicht zuläßt, sofort ab. Ansonsten gibt es in dezimaler Form aus, welche Zeichen in der Speicherstelle Null stehen.

☞ C 37: Zeiger vom Typ void nicht auf Funktionsadressen

Zeiger vom Typ void dürfen nicht auf Funktionsadressen zeigen, nur auf Datenadressen. [32]

☞ C 38: Keine Referenz auf den Null-Zeiger

Null-Zeiger sind keine leeren Strings. Wenn man 0 als NULL- Konstante definiert, und sie dann als Zeiger verwenden möchte, darf man diesen Zeiger niemals referenzieren.

Das Ergebnis einer Umwandlung eines Integer in einen Zeiger ist von der Implementierung abhängig, mit einer wichtigen Ausnahme. Diese Ausnahme bildet die Konstante 0, die garantiert in einen Zeiger umgewandelt wird, der sich von jedem zulässigen Zeiger unterscheidet. Aus Dokumentationsgründen wird dieser Wert oft symbolisch angegeben:

```
#define NULL 0
```

aber die Wirkung ist dieselbe. Wenn man also einer Zeigervariablen den Wert 0 zugewiesen hat, darf man nicht danach fragen, was im damit adressierten Speicher liegt.

☞ C 39: Eine implizite Typkonvertierung in C ist verboten

Die implizite Typkonvertierung birgt, insbesondere bei Pointer-Zuweisungen (z.B. `int = double;` `char = int`) die Gefahr undefinierter oder falscher Werte in sich und sollte deshalb grundsätzlich vermieden werden.

☞ C 40: Verbot der CAST-Konvertierung bei Elementen unterschiedlicher Speicherlänge

Die CAST-Konvertierung ist bei Elementen, die eine unterschiedliche Anzahl von Bytes für ihre Speicher-Darstellung benötigen (z.B. Integer 2 Bytes; Double 4 Bytes) verboten.

☞ C 41: Verbot der Pointer-Schachtelungen größer 2

Pointer-Schachtelung größer als 2 sind verboten. Bei der Schachtelung ist nur die „offene“ Schreibweise erlaubt.

Beispiel C 10

```
*POINTER[anzahl]      nicht aber:
**POINTER oder
***POINTER (=3-fach Schachtelung)
```

9.4.5 Arithmetische Berechnungen, Operatoren

☞ C 42: Testen auf Überlauf durch INT_MAX

Das Testen auf einen Überlauf ist mittels INT_MAX durchzuführen.

Nehmen wir an, das a und b zwei nicht negative int-Variablen sind und es ist zu testen, ob a + b einen Überlauf ergibt. Auf manchen Maschinen wird bei einer Addition ein internes Register in einen der vier folgenden Zustände gebracht: positiv, negativ, Null oder Überlauf. Auf einer solchen Maschine hätte der Compiler die Berechtigung das

Beispiel C 11

```
if (a + b < 0)
    fehlermeldung();
```

so zu übersetzen, daß a und b addiert werden und nachgeprüft wird, ob sich das interne Register im Zustand negativ befindet. Wenn diese Operation einen Überlauf erzeugt, dann befindet sich das Register im Überlaufzustand und der Test ist gescheitert.

Eine korrekte Methode wäre die Umwandlung von a und b in vorzeichenlose Integerwerte und die Benutzung von INT_MAX:

Beispiel C 12

```
if ((unsigned) a + (unsigned) b > INT_MAX)
    fehlermeldung();
```

In diesem Fall ist INT_MAX eine definierte Konstante, die den größten darstellbaren Integerwert enthält. ANSI-C definiert INT_MAX in limits.h. Wenn die Konstante noch nicht vorhanden ist, ist sie in eigenen Implementierung zu definieren.

☞ C 43: Verbot von Vergleichen von Zeigern

Vergleiche von Zeigern z.B. über `<`, `<=`, `>`, `>=`, sind verboten.

Bei Pointern, die in verschiedenen Speichersegmenten liegen, führt der Vergleich zu einem undefinierten Verhalten.

☞ C 44: Explizite Bezeichnung der Genauigkeit bei arithmetischen Umwandlungen

Bei arithmetischen Umwandlungen muß man dem Compiler explizit mitteilen, mit welcher Genauigkeit die Operation (z.B. `+`, `-` oder `~`) durchgeführt werden soll.

In einem Ausdruck können ein `char`-, ein `short-int`, ein `int`-Bit-Feld oder ihre vorzeichenbehafteten oder vorzeichenlosen Varianten verwendet werden, wann immer ein `int` oder `unsigned int` verwendet werden kann. Wenn ein Integer vom Typ `int` alle Werte des Originaltyps repräsentieren kann, wird der Wert in ein `int` umgewandelt. Im anderen Fall wird er in ein `unsigned int` umgewandelt. Diese Umwandlungen werden auf bestimmte Argumentausdrücke, auf Operanden von `+`, `-` oder `~` Operatoren oder auf beide Operanden bei Shift-Operatoren angewendet. Wenn man dem ANSI-Standard genau folgt, dann bedeutet das, daß der Code etwas größer werden kann, weil Ausdrücke, die vom Programmierer mit `char`-Genauigkeit deklariert wurden, mit `int`-Genauigkeit berechnet werden.

Beispiel C 13

```
#define START_OPERATION 0x5A

unsigned char c;
void test(void)
{
    if (c == ~START_OPERATION)
        /* irgendwas_ausfuehren */
}
```

Die Reaktion von ANSI-C ist, daß die `if`-Abfrage immer übersprungen wird. Die Gründe dafür sind, daß bei Operatoren wie `~` (Einerkomplement) der Wert auf zwei Bytes erweitert werden muß. Also ergibt `~START_OPERATION` den Wert `0xFFA5`. Die Variable `c` kann jedoch aufgrund ihres Typs `unsigned char` maximal den Wert `0x00FF` annehmen, das High-Byte bleibt immer 0. Der Vergleich lautet dann

```
if (0x00FF == ~ 005A).
```

Die richtige Programmvariante lautet:

```
#define START_OPERATION 0x5A

unsigned char c;
void test(void)
{
    if (c == (unsigned char)(~START_OPERATION))
        /* irgendwas_ausfuehren */
}
```

Man muß also dem Compiler explizit mitteilen, mit welcher Genauigkeit diese Operation durchgeführt werden soll.

C 45: Rangfolge von Operatoren durch Klammerung festlegen

Die Rangfolge von Operatoren muß durch Klammerung eindeutig festgelegt werden.

Ausschließlich die folgenden Ausdrücke garantieren, daß `expr1` vor `expr2` vor `expr3` ausgeführt wird:

```
expr1, expr2
```

```
expr1 ? expr2 : expr3
```

```
expr1 && expr2
```

```
expr1 || expr2
```

Insgesamt gibt es mehr als 15 Ebenen für die Rangfolge der Operatoren in C.

Die Operatoren, die am stärksten gebunden werden, sind diejenigen, die eigentlich gar keine Operationen darstellen: Indizierung, Funktionsaufruf und Strukturauswahl. Diese sind alle links-assoziativ: `a.b.c` bedeutet dasselbe wie `(a.b).c` und nicht `a.(b.c)`.

Als nächstes kommen die einstelligen Operatoren. Diese besitzen die höchste Rangfolge vor allen echten Operatoren. Da ein Funktionsaufruf vor den einstelligen Operatoren kommt, muß man `(*p)()` schreiben, um eine Funktion aufzurufen, auf die der Zeiger `p` zeigt. `*p()` bedeutet dasselbe wie `*(p())`. Typvorgaben sind einstelligen Operatoren und haben dieselbe Rangfolge wie alle anderen einstelligen Operatoren. Einstellige Operatoren sind rechts-assoziativ, so daß `*p++` als `*(p++)` (inkrementiere das Objekt, auf das `p` zeigt und inkrementiere anschließend `p`) und nicht als `(*p)++` (inkrementiere das Objekt, auf das `p` zeigt) interpretiert wird.

Anschließend kommen die binären Operatoren.

Die arithmetischen Operatoren haben die höchste Rangfolge, dann kommen die Shift-Operatoren, die relationalen Operatoren und die logischen Operatoren, der Zuweisungsoperator sowie schließlich der Bedingungsoperator. Die beiden wichtigsten Regeln, die man sich dazu merken sollte, lauten:

- Jeder logische Operator hat eine niedrigere Rangfolge als jeder der relationalen Operatoren.
- Die Shift-Operatoren binden stärker als die relationalen Operatoren, aber weniger fest als die arithmetischen Operatoren.

Eine Besonderheit ist, daß die sechs relationalen Operatoren nicht dieselbe Rangfolge besitzen: `==` und `!=` binden weniger stark als die anderen relationalen Operatoren.

Bei den logischen Operatoren haben keine zwei die gleiche Rangfolge. Die Bit-Operatoren binden stärker als die sequentiellen Operatoren, jeder Und-Operator

bindet stärker als der entsprechende Oder-Operator und die Bit-Version des exklusiven Oder-Operators \wedge kommt zwischen dem Und- sowie dem Oder-Operator.

Der Dreifachoperator für die Bedingung hat eine niedrigere Rangfolge als alle bisher erwähnten Operatoren. Damit kann ein Auswahl Ausdruck auch logische Kombinationen von relationalen Operatoren enthalten, wie zum Beispiel in:

```
steuer = einkommen > 40000 && wohnort < 5
      ? 3.5 : 2.0;
```

Dieses Beispiel zeigt, daß es sinnvoll ist, die Zuweisung erst nach dem Bedingungsoperator auszuwerten. Alle Zuweisungsoperatoren haben im übrigen dieselbe Rangfolge und werden von rechts nach links ausgewertet, so daß

```
platz_rekord = besucher_rekord = 0;
```

dasselbe bedeutet wie

```
besucher_rekord = 0;
platz_rekord = besucher_rekord;
```

Der niedrigste Operator von allen ist der Komma-Operator. Dies kann man sich leicht merken, da das Komma oft als Ersatz für den Strichpunkt verwendet wird, wenn ein Ausdruck anstelle einer Anweisung erforderlich ist.

C 46: Bevorzugung von && || und ? ,

Bei der Auswertung der Operanden muß man die Operatoren &&, ||, ?: und , allen anderen bevorzugen.

Einige C-Operatoren werten ihre Operanden immer in einer bekannten, genau angegebenen Reihenfolge aus. Bei manchen anderen ist das nicht der Fall. Betrachten wir z.B. den folgenden Ausdruck:

```
a < b && c < d
```

Die Sprachdefinition sagt aus, daß $a < b$ zuerst ausgewertet wird. Wenn a tatsächlich kleiner als b ist, dann muß $c < d$ ausgewertet werden, um den Wert des gesamten Ausdrucks zu bestimmen. Wenn a andererseits größer oder gleich b ist, dann wird $c < d$ überhaupt nicht ausgewertet.

Um $a < b$ auszuwerten, muß der Compiler entweder a oder b zuerst auswerten. Auf manchen Maschinen werden sie vielleicht sogar parallel ausgewertet.

Die Operatoren && und || sind wichtig, um garantieren zu können, daß die Prüfung in der richtigen Reihenfolge durchgeführt wird. Z.B. ist es in

```
if (y != 0 && x/y > toleranz)
    fehlermeldung ();
```

wichtig, daß x / y nur dann ausgewertet wird, wenn y ungleich Null ist.

☞ C 47: Keine Abhängigkeit in der Reihenfolge von Seiteneffekten

Programme dürfen nicht abhängig davon sein, in welcher Reihenfolge Seiteneffekte auftreten.

C garantiert nur, daß ein Seiteneffekt bei der nächsten Befehlsfolge abgeschlossen ist.

Beispiel C 14

```
a[i] = i++;      /* schlecht: */
                /* was wird zuerst ausgeführt: [] oder ++ ? */

++i + i        /* schlecht: was ist zuerst? */
```

9.4.6 Kontrollfluß

☞ C 48: Vorzug der while-Schleife vor der do-while Schleife

Die while-Schleife sollte anstelle der do-while-Schleife verwendet werden, es sei denn, die Logik des Problems erfordert explizit die Ausführung der Schleife unabhängig von den Schleifenbedingungen.

Das Abprüfen der Schleifenbedingungen vor dem ersten Ausführen der Schleife führt zu einem stets definierten Programmverhalten.

☞ C 49: „exit“ nur im Fehlerfall verwenden

Die exit-Anweisung ist dem Fehlerfall vorbehalten. Sie kann nach fehlender Plausibilität nach einer Plausibilitätsprüfung eingesetzt werden.

Die exit-Anweisung verletzt die Regeln der strukturierten Programmierung und sollte deshalb Ausnahmefällen vorbehalten bleiben.

☞ C 50: Einsatz geschweifeter Klammern in Verzweigungen

Um das Problem einer fehlenden else-Anweisung in if-Verzweigungen zu vermeiden muß man entweder geschweifte Klammer oder Makros benutzen.

Beispiel C 15

```
if (x == 0)
    if (y == 0) error ();
    else {
        z = x + y;
        f(&z);
    }
```

Die Absicht des Programmierers in diesem Ausschnitt war die Behandlung von zwei Hauptfällen: $x = 0$ und $x \neq 0$. Im ersten Fall soll in diesem Ausschnitt überhaupt nichts gemacht werden, außer wenn $y = 0$ zutrifft, wobei dann die Funktion error

aufgerufen werden soll. Im zweiten Fall soll das Programm z mit dem Wert von $x + y$ zu laden und dann f mit der Adresse von z als Argument aufrufen.

Dieses Programm macht aber in Wirklichkeit etwas völlig anderes. Der Grund liegt in der Regel, das ein else immer mit dem nächsten nicht abgeschlossenen if innerhalb desselben Blocks verbunden ist. Wenn wir die Einrückung des Fragments so anpassen, erhalten wie etwa folgendes:

```
if (x == 0)
    if (y == 0)
        error ();
else {
    z = x + y;
    f(&z);
}
```

Es passiert, mit anderen Worten, überhaupt nichts, wenn $x \neq 0$ ist. Um das zu erhalten, was durch der Einrückung des ursprünglichen Beispiels ausgedrückt wird, muß man schreiben:

Beispiel C 16

```
if (x == 0) {
    if (y == 0)
        error ();
} else {
    z = x + y;
    f(&z);
}
```

Das else ist hier mit dem ersten if verbunden, auch wenn das zweite if näher liegt, da das zweite if nun in geschweiften Klammern steht.

Die Forderung nach einem Abschlußsymbol vermeidet das Problem eines in der Luft hängenden else ein für alle mal, hat aber den Nachteil, daß die Programmtexte dadurch etwas länger werden. Einige C-Anwender haben diesen Effekt mit Hilfe von Makros zu erreichen versucht:

```
#define IF { if (
#define THEN ) then {
#define ELSE } else {
#define FI  } }
```

Damit kann man das obige C-Beispiel folgendermaßen schreiben:

```
IF x == 0
THEN IF y == 0
    THEN error ()
    FI
ELSE z = x + y;
    f(&z);
FI
```

C 51: if-else anstelle von continue

Wenn eine einzige if-else Kombination ein continue ersetzen kann, dann soll die if-else-Konstruktion bevorzugt werden.

Diese Regel erleichtert die Lesbarkeit von Programmen.

☞ C 52: Nutzung von else-if anstelle von switch

Die Anweisung „else-if“ ist der switch-Anweisung nur dann vorzuziehen, wenn:

- die Bedingungen sich nicht gegenseitig vollständig ausschließen
- die Reihenfolge der Auswertung der Bedingungen wichtig ist oder
- verschiedene Variablen bei den Bedingungen abgefragt werden.

Es ist nicht sinnvoll durch die Einführung zusätzlicher Variablen eine switch-Anweisung zu erzwingen, da die Lesbarkeit des Programmes darunter leidet.

☞ C 53: Zu jedem case ein break

Bei der switch-Anweisung sollte zu jeder Auswahl eine break-Anweisung gehören. Das absichtliche Weglassen der break-Anweisung ist nur dann erlaubt, wenn eine Kontrollstruktur ausgedrückt wird, die ansonsten sehr umständlich zu implementieren wäre (z.B. bei langen switch-Anweisungen, wenn die Verarbeitung bei einem der Fälle durch eine kurze Spezialbehandlung auf einen anderen Fall reduziert werden kann).

C weist die besondere Eigenschaft bei der switch-Anweisung auf, indem man von einer Fallunterscheidung direkt zur nächsten durchrutschen kann.

Beispiel C 17

so:

```
switch (farbe) {
case 1:  printf ("rot");
        break;
case 2:  printf ("gelb");
        break;
case 3:  printf ("blau");
        break;
}
```

aber nicht so:

```
switch (farbe) {
case 1:  printf ("rot");
case 2:  printf ("gelb");
case 3:  printf ("blau");
}
```

Die beiden Mehrfachverzweigungen liefern verschiedene Ergebnisse im Fall wenn die farbe den Wert 2 hat: gelb, bzw. gelbblau, weil die Programmsteuerung von der zweiten printf-Anweisung in zweitem Fall zur darauffolgenden Anweisung übergeht.

Die Möglichkeit die break-Anweisung wegzulassen, kann man auch als mächtiges Mittel effiziente Programme zu schreiben nutzen. Ein Programm z.B., das ein Interpreter für irgendein Computersystem sein soll, enthält wahrscheinlich eine switch-Anweisung, um jeden der verschiedenen Befehlscodes zu behandeln. Bei vielen Computern kommt es vor, daß eine Subtraktion mit einer Addition und einem

Vorzeichenwechsel des zweiten Operanden identisch ist. Deshalb ist es praktisch, wenn man einen Programmausschnitt folgendermaßen formulieren kann:

Beispiel C 18

```
case SUBTRAKTION:
    opnd2 = -opnd2;
    /* keine break-Anweisung*/
case: ADD:
    ...
```

Als weiteres Beispiel kann man sich den Teil eines Compilers vorstellen, der bei der Suche nach einem Token die Leerräume überspringt. In diesem Fall wird man Leerzeichen, Tabulatoren und Zeilenvorschübe identisch behandeln, mit dem einzigen Unterschied, daß bei einem Zeilenvorschub der Zeilenzähler inkrementiert werden muß:

Beispiel C 19

```
case '\n':
    zeilenzahl++;
    /* keine break-Anweisung*/
case: '\t':
case ' ':
```

C 54: Zu jedem switch eine default-Anweisung

Jede switch-Anweisung muß eine default-Anweisung enthalten, die das Verhalten beim Nichtzutreffen aller case-Alternativen eindeutig definiert.

Literaturverzeichnis (*Auszug*)

- [32] Eisenbahn-Bundesamt Mü 8004: Grundprinzipien für den Einsatz der problemorientierten Programmiersprache C in signaltechnisch sicheren Einrichtungen. München 1994
- [75] Weinert, Annette: Programmieren mit Ada und C: eine beispielorientierte Gegenüberstellung, 1992
- [76] Kerningham, B.; Ritchie, D.: Programmieren in C. München: Carl Hanser Verlag, 1983 - ISBN 3-446-13878-1
- [77] Norm ANSI X3.159-1989:
- [78] Norm ISO/IEC 9899:1990:
- [79] Bäuder, Irene; Bär, Jürgen: Quick C für Windows. Haar bei München: Markt&Technik-Verlag, 1992 – ISBN 3-87791-271-0
- [80] Hatton, Les: Safer C: Developing Software for High-integrity and Safety-critical Systems. London: McGraw-Hill Book Company, 1995 – ISBN 0-07-707640-0
- [81] Feuer, A. R.: Introduction to the Safe C Runtime Analyser. Boston: Catalytix Corp., 1985.
- [82] Koenig, A: Der C-Experte: Programmieren ohne Pannen. Bonn: Addison-Wesley Verlag (Deutschland) GmbH 1989 – ISBN 3-89319-233-6.
- [83] Plum, Thomas: C Programming Guidelines. Cardiff: Plum Hall Inc. 1984 – ISBN 0-911537-03-1.
- [84] Hansen, Augie: Programmieren lernen mit C. Braunschweig: Friedr. Vieweg & Sohn Verlagsgesellschaft mbH 1990 – ISBN 3-528-04694-5.