Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Helmut O. B. Schellong

Moderne C-Programmierung

Kompendium und Referenz

Mit 380 farbigen Kodeabschnitten auf CD-ROM

Springer

Helmut Schellong
Tiefer Grund 12
32108 Bad Salzuflen
autor@schellong.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN-10 3-540-23785-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-23785-3 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: durch den Autor
Herstellung: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142/YL - 5 4 3 2 1 0

Vorwort

Motivation für die Entwicklung der Sprache C war die Existenz des UNIX® Betriebssystems, das ab 1970[1] bekannt wurde. Es sollte erreicht werden, daß UNIX auf andere Prozessoren portiert werden konnte, ohne jedesmal das Betriebssystem vollständig in Assembler neu schreiben zu müssen, sondern eher nur zu 5 %. Die Kommandos dieses Betriebssystems konnten dann gänzlich ohne Neuprogrammierung erzeugt werden – einfach durch Kompilierung.

Deshalb ist bis heute traditionell ein C-Entwicklungssystem für jedes UNIX-System vorhanden. Oft ist es ohne C gar nicht erhältlich.

Lehrgrundlage für dieses Buch waren die beiden Bücher *Programmieren in C* [5] der Autoren Kernighan/Ritchie, wobei Dennis M. Ritchie eher der Erfinder der Sprache C und Brian W. Kernighan eher der Buchautor ist.

Diese Bücher gelten als *C-Bibel* und werden in der C-Szene mit K&R1 und K&R2 bezeichnet. Die Sprache C im Sinne dieser Bücher wird K&R-C bzw. ANSI-C genannt. Ein K&R3 aufgrund des neuen C-Standards von 1999 ist leider nicht in Sicht. Vielleicht ist *dieses* Buch ein gewisser Ersatz dafür.

Die Sprache C war 1974 fertig entwickelt. Das englische Original des Buches K&R1 erschien dann 1978. Ab 1983 gab es Standardisierungsbemühungen, die zum ANSI-Standard C89 und dann auch zum internationalen Standard ISO/IEC C90 führten. Am 1.Dez.1999 wurde der nun aktuelle C-Standard C99 veröffentlicht, der bis dahin mit C9X bezeichnet wurde.

Der aktuelle Standard [4] wurde als Informationsquelle bei der Erstellung dieses Buches benutzt. Beispielkode wurde einem früheren Entwurf des Standards [3] entnommen.

[1] Ein Zeitstempelwert von 0 s unter UNIX ergibt: 01.01.1970 00:00:00

Der C-Standard ist ein englischsprachiges Dokument mit etwa 560 Seiten, das die Sprache C in einer Art *juristischen Vertragssprache* beschreibt, sich eher an Entwickler von Compilern wendet und somit als Lehrbuch oder Tutorial/Kompendium ungeeignet ist. Es wird daher auch nicht daraus zitiert.

Bei Befragungen von Dennis Ritchie[2] sagte dieser beispielsweise auf die Frage, wie ihm der neue C-Standard C99 gefällt, daß ihm der alte Standard C89 gefiel. Er hätte sich gewünscht, daß das Komitee mehr Standhaftigkeit gezeigt hätte gegenüber den vielen Vorschlägen neuer C99 Merkmale. Er kritisiert den beträchtlich gesteigerten Umfang des neuen Standards, und beispielsweise den neuen Typ `_Complex`, der in Folge die Libraries sehr aufbläht.

Es existiert Ende 2004, fünf Jahre nach Veröffentlichung von C99, auch kein einziger C-Compiler, der C99 zu 100 % unterstützt. Bei dieser Bewertung muß bedacht werden, daß ja alle Bibliotheken und alle Header-Dateien gemäß C99 dabei sein müssen. Aber zum Glück unterstützen einige Compiler C99 annähernd zu 100 %. Darunter auch der sehr bekannte `gcc`. C99 ist zwar kein Flop, aber die Stimmung darüber ist bedeckt.

Dieses Buch wendet sich an Leser, die bereits programmieren können und C oder eine andere Programmiersprache ausreichend (oder besser) beherrschen. Unter dieser Voraussetzung ist es sicher möglich, C anhand dieses Buches zu erlernen oder seine Kenntnisse (stark) zu verbessern.

Es liegt die Erfahrung vor, daß wenn Programmierer mit einiger Vorerfahrung eine weitere, für sie neue Programmiersprache erlernen wollen, sie diese neue Sprache möglichst schnell *erfassen* wollen, mit Hilfe von kompakten aber vollständigen Listen, Tabellen und ähnlichen Darstellungsformen nebst Erklärungen in Kurzform, begleitet von Kodebeispielen.

Dieses Buch stellt darauf ab und bietet im Teil I auf etwa 40 Seiten diese Möglichkeit, wobei gleichzeitig eine Referenz gegeben ist.

Dieses Buch basiert auf einem Dokument [1], das seit Januar 2000 als Webseite im Internet existiert und dort außerordentlich beliebt ist. Es erreichten den Verfasser eMail-Zuschriften, die genau die zuvor beschriebene Kompaktheit, die Listen, die Tabellen, den daraus resultierenden schnellen Zugriff und die gleichzeitige relative Vollständigkeit lobten. Ebenso wurde die Kommandoliste zum Editor `vi` kommentiert (Anhang) [2].

Dieses papierne Buch *setzt da noch einen drauf*, indem es besser gegliedert, umfangreicher, noch sorgfältiger geschrieben und fehlerfreier ist.

[2] Dennis Ritchie prägte auch sehr wesentlich die Entwicklung von UNIX.

Der C-Standard verwendet ein Vokabular, das in einem Lernbuch nicht dienlich wäre, wie beispielsweise *Übersetzungseinheit* für *C-Datei*, *Übersetzer* für *Compiler*, *Basisausführungszeichensatz* für *Zeichensatz*, und so weiter. In diesem Buch hingegen werden allgemein übliche Begriffe verwendet, wie sie in der täglichen Praxis anzutreffen sind (▶ S. 203).

Im Zusammenhang mit Programmierung gibt es einige Abhängigkeiten von der verwendeten Plattform (Prozessor, Betriebssytem). In diesem Buch wird nötigenfalls von der gängigsten Plattform ausgegangen, nämlich INTEL iX86. Das ist u. a. bei konkreten Zahlenbeispielen und Bitrepräsentationen oft unvermeidbar.

Die Sprache C ist allerdings mit großem Abstand die portabelste Programmiersprache überhaupt, so daß in der Programmierpraxis mit C kaum unterschiedlich programmiert werden muß, um verschiedene Plattformen zu bedienen.

Desweiteren ist C generell wohl die am weitesten verbreitete Programmiersprache. Entwicklungssysteme für Mikrokontroller haben nahezu ausnahmslos C-Compiler. Eine andere Tendenz ist nicht in Sicht. Das wäre auch Unfug, denn für die Programmierung von Betriebssystemen, deren Kommandos und die Millionen von Programmen für Mikrokontroller ist eine eher wenig abstrahierende Sprache wie C auch am geeignetsten.

C ist wegen ihrer Feinkörnigkeit, Rekursivität, Flexibilität und Ressourcenschonung für die zuvor aufgeführten Anwendungsgebiete einfach ideal. Es können aber auch alle anderen Gebiete gut bis ausreichend mit C bedient werden, besonders wenn es auf Ressourcenschonung und maximale Effizienz ankommt. Wenn es darauf nicht ankommt, sind manche anderen Programmiersprachen komfortabler und *sicherer*.

Das Manuskript zu diesem Buch wurde hergestellt mit: dem Editor `gvim`, dem vom Verfasser selbst entwickelten Skript-Interpreter `bsh` und – LaTeX.

UNIX-Betriebssystem war FREEBSD V4.11 mit Programmpaket `teTeX-3.0`.

Die Buch-CD enthält eine `.html`-Datei mit den Kodeabschnitten des Buches in Farbe. Ebenso eine `.txt`-Datei.

Bad Salzuflen, November 2004 *Helmut O. B. Schellong*

Inhaltsverzeichnis

Teil I

Erfassung der Sprache C / Referenz

1

C-Schlüsselwörter

Schlüsselwörter sind in allen Programmiersprachen reservierte Wörter. In C sind auch alle Namen `_[A-Z]...` und `__...` reserviert, auch `_...` bereichsweise. Weiterhin durch die Standard-<Header> eingeführte Bezeichner.

1.1 Liste der Schlüsselwörter

Typen von Datenobjekten

`char`	Integer
`short`	Integer
`int`	Integer (Voreinstellung)
`long`	Integer
`float`	Gleitkomma
`double`	Gleitkomma
`void`	leer/unbestimmt
`unsigned`	Integer ohne Vorzeichen
`signed`	Integer (explizit) mit Vorzeichen
`struct`	Beliebig zusammengesetzter Typ (Struktur)
`union`	Auswahltyp/Vereinigungstyp
`enum`	Aufzählungstyp (Enumeration; Integer)
`typedef`	Individuelle Typvereinbarung

Byte-Anzahl von Daten-Objekten und -Typen

`sizeof`	Beispiele: `sizeof(int), sizeof i`

Speicherklassen von (Daten-)Objekten

`auto`	Voreinstellung in Funktionen
`register`	(Möglichst) ein Prozessor-Register verwenden
`static`	Name/Symbol wird nicht exportiert; statisch
`extern`	Objekt (public), von außen her bekanntmachen

Typqualifizierer von Daten-Objekten

`const`	Nach Initialisierung read-only
`volatile`	Keine Optimierung/Stets Direktzugriff

Programmierung des Ablaufes

`if`	Bedingte Verzweigung (falls ja)
`else`	Bedingte Verzweigung (falls nein)
`while`	Schleife
`for`	Schleife
`do`	Schleife (do-while)
`switch`	Fallunterscheidung (case)
`case`	Fallunterscheidung (Fallzweig)
`default`	Fallunterscheidung (default-case)
`break`	Hinaussprung: Schleife/switch
`continue`	Fortsetzungssprung in Schleifen
`goto`	Allgemeiner unbedingter Sprung
`return`	Beenden einer Funktion (Rücksprung)

Neuer C-Standard C99

`inline`	Funktionen Inline
`restrict`	Privater Speicherbereich für Zeiger
`_Bool`	Wertebereich `{0,1}`
`_Complex`	z = real + imaginär
`_Imaginary`	imaginär

Die Schreibweise `_Xyyy` bei den neuen Schlüsselwörtern wurde wahrscheinlich so gewählt, weil beispielsweise `bool` als Preprocessor-Definition zuvor bereits weit verbreitet war und/oder wegen des Zusammenhangs von C und C++.

1.2 Erklärung einiger besonderer Schlüsselwörter

const

```
const int ci= 16;
const int *cip= adr;
```

Hiernach sind `ci= x;` und `*cip= x;` Fehler!

```
int * const ipc= &ci;
```

Hiernach wäre `ipc++;` ein Fehler! Aber, diese Definition ist problematisch, denn `(*ipc)++;` wäre ja erlaubt, was aber `ci` ändern würde, `ci` ist jedoch konstant!

```
int const * const ccip= &ci;
```

Hiernach sind `*ccip= x;` und `ccip= a;` Fehler!

```
extern int atoi(const char *);
```

Optimierung möglich, da der Compiler davon ausgehen darf, daß die Funktion `atoi` auf das per Adresse übergebene Objekt keine Schreibzugriffe vornimmt.

volatile

```
static void seKulmination(void)
{
   int volatile rf=0;
   if (setjmp(J)==1 && rf>0)  return;
   // ...
   if (seKul>9)  rf=2, ++a;
   if (a)  longjmp(J, 1);
   ErrV= rf;  // cc
   return;
}
```

Ohne `volatile` hält der Compiler die Variable `rf` wahrscheinlich zeitweilig in einem Register und ändert sie *nicht* an der Stelle `rf=2`, sondern erst an der Stelle `cc`, weil sie bis dahin gar nicht gebraucht wird. Sie wird aber doch gebraucht, falls `longjmp()` aufgerufen wird, denn `longjmp()` kehrt nicht zurück, sondern springt nach `setjmp()`, wo `rf>0` steht. Und genau *das* kann der Compiler nicht wissen!

Mit `volatile` wird auf die Variable `rf` bei *jedem* Vorkommnis *direkt* zugegriffen. Eine weitere Fehlermöglichkeit existiert im Zusammenhang mit globalen Variablen, die laufend in Interrupt-Routinen verändert werden. (▶ S. 209)

typedef

Dieses Schlüsselwort gestattet die Definition eigener Typen. Komplexe Typen sind besonders einfach zu definieren durch mehrstufige Anwendung.

```
typedef  unsigned char  BYTE;
typedef  BYTE  *BYTEP;

BYTE a, b=2, c, *bp=0;
```

```
typedef struct _FILE_
{
   int            __cnt;     // 4
   unsigned char *__ptr,     // 8
                 *__base,    //12
                  __flag,    //13
                  __file,    //14
                  __buf[2];  //16(Dummy)
} FILE, *FILEP;

extern FILE __iob[];

#define stdin   (&__iob[0])
#define stdout  (&__iob[1])
#define stderr  (&__iob[2])

FILE *fp;
fp= fopen("COM2", "rb");
if (!fp)  PrintErr(E_OPENF, "COM2"), exit(2);
```

Hier wurden zuletzt die Typen `FILE` und `FILEP` definiert. Anschließend ein Array `__iob[]` bekannt gemacht, aus Strukturen vom `FILE`-Typ als Elemente, das außerhalb angelegt ist, weshalb es als *unvollständiger Typ* angegeben wurde (`[`ohne Angabe`]`), da die Elementanzahl in der aktuellen C-Quelle unbekannt ist. Man sieht auch, daß die vordefinierten `FILE`-Pointer `stdin`, `stdout`, `stderr`, die Adressen der ersten drei Array-Elemente sind, vom Typ `FILE*` beziehungsweise `FILEP`.
(▶ 37, 132, 210, 115)

```
typedef  int A[10][2];
A a, b;
int a[10][2], b[10][2];
```

Es wurde ein Typ `A` definiert. Die beiden letzten Zeilen haben gleiche Wirkung.

2

Elementare Datentypen

2.1 Liste der Datentypen

Bitanzahl und Wertbereich sind nachfolgend für typische Intel-Plattformen gezeigt (<limits.h> <float.h>):

Typ	Breite	Wertbereich/Signifikanz
char	8	-128...+127
signed char	8	-128...+127
unsigned char	8	0...255
short	16	-32768...+32767
unsigned short	16	0...65535
int	32	-2147483648...+2147483647
unsigned	32	0...4294967295
long	32	siehe int
unsigned long	32	siehe unsigned
long long	64	-9223372036854775808... +9223372036854775807
unsigned long long	64	0...18446744073709551615
int	16	DOS: siehe short
unsigned	16	DOS: siehe unsigned short
float	32	7 Stellen
double	64	15 Stellen
long double	80	19 Stellen (Intel iX86)
long double	128	33 Stellen (z. B. HP-UX)
sizeof(long double)	10	DOS
sizeof(long double)	12	32 Bit (10+2 FüllByte)
sizeof(long double)	16	HP-UX

Die Integer-Typen wurden in Kurzschreibweise angegeben; Hinter short, unsigned und long kann jeweils noch int stehen: z. B.: unsigned long int

2.2 Erklärungen zu den Datentypen

`signed` kann auch mit anderen Integer-Typen als `char` verwendet werden, was aber redundant wäre. Eine Vorzeichenbehaftung des Typs `char` ist implementationsabhängig! Genau betrachtet sind `char`, `signed char` und `unsigned char` drei verschiedene Typen.

Die meisten Compiler haben eine sehr sinnvolle Option, die global umfassend `char` als `unsigned char` bewertet. Diese Option erzeugt oft etwas kleinere und schnellere Programme und beseitigt eine Reihe von potentiellen Problemen. Auch Zeichenkonstanten `'c'` sind dabei `(int)(unsigned char)`. Wurde diese Option gegeben, hat `signed char` einen Sinn. Allerdings will man höchst selten im Zahlenbereich eines `char` arithmetische Berechnungen mit Vorzeichen betreiben. Andererseits können beispielsweise Temperaturwerte platzsparend gespeichert werden, die dann bei Verwendung automatisch (implizit) vorzeichenerhaltend auf `int` erweitert werden (▶ S. 63).

1er-Komplement-Prozessoren können in 8 Bit nur `-127..+127` darstellen, nicht `-128..+127`. Dafür haben sie eine negative und eine positive `0`. Der Ansi-Standard fordert `-127..+127` als Mindestwertebereich für einen vorzeichenbehafteten `char`.

Die Wertebereiche in der obenstehenden Tabelle entsprechen den Ansi-Mindestforderungen, mit der Ausnahme, daß laut Ansi die negativen Werte um 1 positiver sind und daß für `int` der Wertbereich eines `short` ausreicht, was auch für deren `unsigned`-Varianten gilt.

`long long` ist neu seit dem Standard C99. Der Compiler `gcc` kannte schon zuvor `long long`, Borland Builder kennt `__int64` und `-2345i64`, `2345ui64`. Der neue Ansi-Standard enthält: `long long`, `<stdint.h>`: `int64_t`, etc.

Achtung, es gibt Plattformen, auf denen ein Byte 32 Bit hat! Und die Typen `char`, `int`, `long`, ... haben dort alle 32 Bit!, und `sizeof(typ)` für all diese Typen ist `1`, was korrekt ist.

void

```
void funktion(void)  { /*...*/  return; }
```

Die Funktion `funktion` hat *keinen* Rückgabewert und erhält *keine* Argumente beim Aufruf: `funktion();`

```
(void)printf("...");
```

Der Rückgabewert der Funktion `printf` (die einen solchen hat: `int`) soll explizit *ignoriert* werden. Dies wird aber meist implizit belassen, indem ein Rückgabewert einfach gänzlich unbenutzt bleibt.

void

```
# define uchar  unsigned char

void *memset_F(void *d0, int i, register unsigned n)
{
   register uchar *d= (uchar*)d0;
   register uchar  c= (uchar )i;
   while (n > 0)  *d++ = c, --n;
   return (d0);
}

# undef uchar
```

Der Funktion `memset_F` kann die Adresse von Zielobjekten *beliebigen Typs* übergeben werden, weil hier mittels `void*` ein unbestimmter Adressentyp vereinbart wurde. Innerhalb der Funktion wird in eine `uchar`-Adresse umgewandelt. Bei Variablen, die eine `void`-Adresse enthalten (oben: `d0`), können weder `*d0` noch `d0++` durchgeführt werden, weil der Compiler nicht weiß, mit welcher Breite er auf das Objekt zugreifen soll und um wieviel er die Adresse erhöhen soll. Der Compiler kennt nur `sizeof(d0)`, nicht jedoch `sizeof(*d0)`! Es können damit nur Zuweisungen und Vergleiche auf Gleichheit oder Ungleichheit vorgenommen werden.

```
int iarr[64];
struct dirent dea[100];

memset_F(iarr, 0, sizeof(iarr));
memset_F(dea , 0, sizeof(dea));
memset_F(&dea[8], -1, sizeof(*dea));
memset_F(&dea[8], -1, sizeof(struct dirent));

// sizeof(dea)/sizeof(*dea) ist gleich 100
```

`memset()` schreibt ggf. 0-Bits und löscht somit nur bedingt typgerecht (▶ 126).

Der Ausdruck `(void)v;` hat *keinen* Wert, sondern es werden eventuell nur Seiteneffekte bewirkt. Dies bei diesem Ausdruck, falls `v` mit `volatile` qualifiziert ist. In diesem Fall erfolgte ein Blind-Lesevorgang von `v`. Ohne `volatile` würde der Compiler diesen Ausdruck weg-optimieren (▶ 111).

Hingegen der Ausdruck `(void*)&v;` hat einen Wert, nämlich einen Adressenwert, der vom Originaltyp her unverändert sein muß. Der Typ `void*` ist ein zentraler, temporärer Träger für *alle* Adressentypen.

Jede Adresse `void*` kann ohne explizite Typumwandlung mit beliebigen Adressentypen verknüpft werden, im Rahmen der erlaubten Operationen. Der obenstehende Typ-Cast `(uchar*)d0` ist redundant.

```
typedef unsigned       uint;
typedef unsigned char  byte;

void *memcpy_F(void *d0, const void *s0, register uint n)
{
   register byte *d=d0;
   register const byte *s=s0;
   if (n)  do  *d = *s, ++d,++s;  while (--n);
   return d0;
}

int memcmp_F(const void *d0, const void *s0, uint n)
{
   register const byte *d=d0, *s=s0;
   if (n && d!=s)  {
     do  if (*d!=*s)  return *d - *s;
     while (--n&&(++d,++s,1));
   }
   return 0;
}

uint strlen_F(const byte *s0)
{
   register const byte *s=s0;
   while (*s)  ++s;
   return (s-s0);
}

int strcmp_F(const byte *d0, const byte *s0)
{
   register const byte *d=d0, *s=s0;
   if (d!=s)  {
     for (;  1;  ++d,++s)  {
        if (*d!=*s)  return *d - *s;
        if (!*s)  break;
     }
   }
   return 0;
}
```

Es ist zu beachten, daß `*d - *s` wegen der `int`-Promotion (▶ 63) zu falschen Ergebnissen führte, sollten vorzeichenbehaftete Byte-Werte verglichen werden. Bei Gleichheit `d!=s` würde ein und dieselbe Byte-Folge verglichen.

3

Punktuatoren und Operatoren

3.1 Punktuatoren

```
[ ]  ( )  { }  .  ->
++  --  &  *  +  -  ~  !
/  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
?  :  ;  ...
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #  ##
<:  :>  <%  %>  %:  %:%:
```

Punktuatoren sind Symbole mit unabhängiger syntaktischer und semantischer Signifikanz. Abhängig vom Kontext können sie Operatoren sein. Alle Klammern sind nur als Klammer*paar* gültig.

```
<:  :>  <%  %>  %:  %:%:

 [  ]    {  }    #   ##
```

Die Punktuatoren in der hier oberen Zeile sind Ersatzsymbole. Die untere Zeile zeigt deren Bedeutung. Diese Ersatzsymbole berücksichtigen beispielsweise Tastaturen, die die Ursprungssymbole nicht erzeugen können.

{ }

Die geschweiften Klammern (Blockklammern) dienen hauptsächlich der Zusammenfassung von Anweisungen, um diese gemeinsam beispielsweise von einer Bedingungsprüfung abhängig zu machen oder sie einem Funktionskopf zuzuordnen, etc.

:

Der Doppelpunkt als Punktuator steht hinter Sprungzielmarken und zeigt innerhalb von Strukturen ein Bitfeld an.

;

Das Semikolon ist ebenfalls *kein* Operator, sondern das Abschlußzeichen einer jeden Anweisung. Ein Semikolon ohne Anweisung davor ist eine Leer-Anweisung, die beliebig redundant gesetzt werden kann:

```
{ /*...*/ a=4;;;;;;;;; };;;;;
```

Diese erlaubte Redundanz ist bei Makros nützlich. Im Kopf einer `for`-Schleife müssen sich grundsätzlich zwei Semikoli befinden, als Trenner/Abschluß. Das Semikolon ist ein Sequenzpunkt.

...

Die Ellipsis wird verwendet, um bei Funktionen eine variable Anzahl von Argumenten anzuzeigen. Dieses beim Prototypen (Deklaration) und bei der Definition einer Funktion. Bei Makros ab C99 möglich. (▶ 202)

#

Das Doppelkreuz-Zeichen als erstes Nichtzwischenraumzeichen in einer Zeile macht diese Zeile zu einer Umgebung für den Preprocessor, fungiert hier also als Punktuator. Innerhalb einer solchen Zeile ist *keine weitere* Direktive (beginnend mit #) möglich, inbesondere nicht innerhalb einer Makrodefinition. Innerhalb der Definition eines Makros wirkt # als Stringizing-Operator auf Makro-Parameter.

##

In einer Preprocessor-Zeile innerhalb einer Makro-Definition wirkt diese Zeichenfolge als Tokenizer-Operator auf Makro-Parameter. (▶ 33)

3.2 Operatoren

Operatoren haben Operanden und veranlassen eine Operation/Aktion.

Fallender Rang von oben nach unten.
Zusammenfassung von links/rechts (L/R) her.

```
()    []    ->    .                                          L
*     +     -     !   ~   ++   --   &   (typ)   sizeof      R (unär)
*     /     %                                                L
+     -                                                      L
<<    >>                                                     L
<     <=    >     >=                                         L
==    !=                                                     L
&                                                            L
^                                                            L
|                                                            L
&&                                                           L
||                                                           L
?:                                                           R (ternär)
=     +=    -=    *=    /=    %=    &=    ^=    |=    <<=   >>=   R
,                                                            L
```

unär, monadisch:	Operator hat einen Operanden
binär, dyadisch:	Operator hat zwei Operanden
ternär, triadisch:	Operator hat drei Operanden

Die Operatoren * + - & werden **unär** als auch **binär** verwendet!
Das Komma wird als Operator **und** als Trenner verwendet!

Merksätze:

- Beim Programmieren immer an Rang und Zusammenfassungsrichtung aller beteiligten Operatoren denken!
- Bei jeglichem Verbinden, Verknüpfen, Zuweisen von Ausdrücken immer an deren Typ bzw. den Typ beider Seiten denken!

3.3 Operatoren, kurz erklärt

Zu jedem Operator wird nachstehend nur eine kurze Erklärung gegeben, und zwar, wie die Operatoren schwerpunktmäßig verwendet werden. Die Operatoren werden hier nicht erschöpfend erklärt, sondern andere Kapitel ergänzen dies, implizit/explizit, von jeweils einem anderen Hauptthema ausgehend.

( )

Ein Paar runde Klammern bildet den Funktionsaufruf-Operator. Im Zusammenhang mit anderen Verwendungen kann nicht mehr von Operator gesprochen werden, da es keine Operanden gibt: Mit runden Klammern kann eine gewünschte Verknüpfungsreihenfolge erzwungen werden, falls der vorgegebene Rang von Operatoren eine unerwünschte Verarbeitung zur Folge hätte. Beispiel: `a= (b + c) * d;`

Außerdem werden runde Klammern zur Abgrenzung, Einfassung, Zusammenfassung eingesetzt, beispielsweise bei Bedingungen. Desweiteren kann mit ihnen ein Kommalisten-Ausdruck gebildet werden, bei dem das letzte Abteil den Ausdruckwert repräsentiert.

[ ]

Eckige Klammern stehen hinter Array-Namen und geben das Array-Element an, auf das Zugriff erfolgen soll: `array_name[index_ausdruck]`

Das Resultat eines Index-Ausdrucks muß ein Integer-Wert sein. Auf das erste Array-Element wird mit Index 0 (`array_name[0]`) zugegriffen. Der Name eines Arrays *repräsentiert* die Adresse des ersten Array-Elements; ein Array-Name kann nicht Ziel einer Zuweisung sein, er kann *nur lesend* verwendet werden! Eckige Klammern dereferenzieren: `a[n]` entspricht `*(a+n)`. Eckige Klammern können deshalb auch hinter einer Adressen-Variablen stehen und dann auch einen negativen Index enthalten:

```
int ia[10];
int *iap, i;
iap= ia+5;
i= iap[-2];      // entspricht: i= ia[3];
i= (ia+5)[-2];   // dito
```

Bei Adressen-Ausdrücken haben `[0]` und `*` die gleiche dereferenzierende Wirkung. Bei `[char==signed char]` auf die ungewollte Entstehung negativer Indexwerte achten! `(unsigned char)` beseitigt solche Probleme.

->

Dies ist ein Zugriffsoperator, um von einer Struktur-*Adresse* ausgehend auf ein Mitglied der Struktur zuzugreifen:

```
long l;
struct test { int i; char c; long l; };
struct test  T;
struct test *TP= &T;
l= TP->l;
l= (*TP).l;          // *TP ist die Struktur T als Ganzes
l= TP[0].l;
l= T.l;
l= (&T)->l;
```

Die letzten fünf Zeilen haben die gleiche Wirkung! Man erkennt, daß es diesen Operator nur aus Komfortgründen gibt. Der Name einer Struktur repräsentiert *nicht* deren Adresse, wie bei Arrays, sondern ist die Struktur als Ganzes!

.

Der Punkt-Operator dient dem Zugriff auf Struktur-Mitglieder, von einer Struktur ausgehend. Siehe oben Operator ->

*

Dies ist der allgemeine Zugriffsoperator, um von einer Adresse ausgehend auf den Inhalt des adressierten Objektes zuzugreifen (Dereferenzierung):

```
int i, *ip;
ip= &i;
*ip= 5;
i= 5;
```

Die beiden letzten Zeilen haben gleiche Wirkung.

&

Dies ist der Adressen-Operator, der die Adresse eines Objektes liefert.

```
int *ip, **ipp;
ipp= &ip;
```

`register`-Variablen und Bitfelder haben keine &Adressen! Man kann auch Offsetadressen aus größeren Objekten erhalten, beispielsweise Adressen von Array-Elementen und Struktur-Mitgliedern. Letztlich hat jedes einzelne Byte von Objekten eine Adresse; ein Byte (`char`/`unsigned char`) ist gewöhnlich die kleinste adressierbare Einheit.

- +

Bei unärer Verwendung wird ein Wert negiert (NEG): `i= -i;` Hierdurch wechselt der Wert in `i` sein Vorzeichen. Der Operator `+` dient hier nur Symmetriezwecken.

~

Komplementierung aller Bits eines Integer-Wertes (NOT/Einer-Komplement):

```
~00000000 == 11111111
~11111111 == 00000000
~11001101 == 00110010
unsigned long ul= ~(0ul);
```

Achtung, die hier gezeigte Zahlendarstellung zur Basis 2 (Dualzahlen) ist in C *nicht* möglich!

!

Logisches NOT/NICHT.
Aus einem Wert `0` oder `0.0` resultiert `1` (TRUE).
Aus einem Wert ungleich `0` oder `0.0` resultiert `0` (FALSE).

```
if (!a)  a= b+2;
i= !!25;
i= !!(1-3);
```

`i` erhält in beiden Fällen den `int`-Wert `1`.
`(a)` entspricht `(a!=0)`, `(!a)` entspricht `(a==0)`.
Dieser Operator kann auch auf Adressen angewandt werden: Prüfung auf NULL.

++ - * ()

```
c= ++*a++;
c= ++*++a;
c= (*++a)--;        // Fehler!: *++a--
```

Wirkungsweisen (▶ 17):
Der Inhalt `*a` wird inkrementiert und zugewiesen, dann wird `a` selbst inkrementiert.
`a` wird inkrementiert, der Inhalt `*a` inkrementiert und zugewiesen.
`a` wird inkrementiert, der Inhalt `*a` zugewiesen, dann der Inhalt dekrementiert.

Es ist zu beachten, daß je nur *eine* Operation ++ oder `--` auf den Inhalt von `a` und den *anderen* Inhalt `*a` korrekt sind (▶ 240).

++ --

Inkrement / Dekrement: Ein Variableninhalt wird um `1` oder `1.0` erhöht bzw. reduziert (Addition / Subtraktion von `1` oder `1.0`). Es gibt einen Unterschied, je nach dem, ob diese Operatoren *vor* oder *nach* einem Variablennamen stehen: Präinkrement/-dekrement oder Postinkrement/-dekrement.

```
++i; i++; i+=1; i=i+1;
--i; i--; i-=1; i=i-1;
```

Diese je vier Anweisungen haben jeweils den gleichen Effekt.

```
a= b + ++i + c;
```

Der Wert des Ausdrucks `++i` ist der alte Wert des Inhalts von `i` plus `1`.

```
a= b + i++ + c;
```

Der Wert des Ausdrucks `i++` ist der alte Wert des Inhalts von `i`.
Achtung: der Inhalt von `i` wird (in beiden Fällen) möglicherweise erst beim Semikolon (`;`), dem nächsten Sequenzpunkt, inkrementiert! Man soll niemals den Inhalt ein und derselben Variable zwischen zwei Sequenzpunkten mehr als einmal ändern oder verwenden bei vorliegender Änderung!

```
a= b + i+1 + c, ++i;
a= b + i   + c, ++i;
```

Diese beiden Anweisungen zeigen das Sequenzpunkt-Verhalten der obenstehenden Varianten. Der Komma-Operator ist auch ein Sequenzpunkt. Im Zusammenhang mit `++ --` gibt es sehr erhebliche Unsicherheiten!

```
i-- = 2;  i++ = 2;
```

Das sind Fehler! Schon allein von den Sequenzpunkt-Regeln her! Die Ausdrücke `x++ x-- ++x --x` sind sogenannte *rvalues*, denen grundsätzlich keine weitere Änderung zugewiesen werden kann. Auch `&++x` ist nicht möglich, da `&` nur auf *lvalues* angewandt werden kann.

```
*a++ = *b++;
*a   = *b  ,  a++, ++b;
*++a = ++*b++;
```

Auch die erste Zeile oben ist korrekt, da nur die Adressen geändert werden, nicht aber der dereferenzierte Zielinhalt. Es wird ja an `a` nichts zugewiesen per `=` , sondern an `*a`. Die zweite Zeile hat die gleiche Wirkung.
*Wirkung*sweise dritte Zeile: `a` wird vor der Zuweisung an `*a` inkrementiert. Der Inhalt `*b` wird inkrementiert und dann an `*a` zugewiesen, `b` selbst wird erst danach inkrementiert.

(typ)

Typ-Cast: Explizite Umwandlung des Typs des Wertes eines Objektes/Ausdrucks in einen anderen Typ. Es können alle elementaren Typen untereinander umgewandelt werden. Ein Cast kann eine Wertveränderung bewirken!

```
unsigned u;
int  i=100;
char c;
int  *ipa, *ipb;

c= (char)i;
c=-2;              // c == 11111110
i= c;              // i == -2 == 111111...1111111111111110
u= c;              // u == 4294967294u

u= (unsigned)c;                    // u == 4294967294u
u= (unsigned)(unsigned char)c;     // u == 254 == 11111110
u= (unsigned char)c;               // u == 254

c= 127;
u=i=c;             // u == i == c == 127 == 01111111

ipb= ipa + 1;
i= (int)(ipb-ipa);                        // i == 1
i= ( (unsigned)ipb - (unsigned)ipa );     // i == 4
i= ( (char*)ipb - (char*)ipa );           // i == 4

struct kfl { char a; unsigned char b; } *sp;
  sp= (struct kfl*) &i;
++sp->b;
```

Die drei letzten Zeilen sind sicher *bemerkenswert* – sie sind aber korrekt, da `i` groß genug ist, um die Struktur vom Typ `kfl` aufnehmen zu können und weil kein Misalignment auftreten kann.

```
char Buf[256];
for (i=0;  i<sizeof(Buf)/sizeof(long);  ++i)
{
   ((long*)Buf)[i]= 0L;
}
```

Die Zuweisung in der `for`-Schleife funktioniert fein mit Intel-x86-Prozessoren, deren `AC`-Bit im Statusregister den Reset-zustand 0 hat, ist aber generell gewagt! Hier werden 64 longs auf 0 gesetzt, anstatt 256 mal 1 Byte. Mit anderen Prozessoren kann `long`-Misalignment auftreten und `256/sizeof(long)` kann einen Divisionsrest er-

geben und Probleme mit Padding-Bits sind möglich! Aber, mit **union** ist so etwas auch (ziemlich) portabel lösbar.

Adressen-Differenzen (s. o. `ipb-ipa`) haben den Typ `ptrdiff_t`, ein vorzeichenbehafteter Integer, der in aller Regel `int` entspricht. Klammerung und Typ-Cast können folglich entfallen. Compiler warnen davor, wenn bei einer Zuweisung ein Wert für das Zielobjekt zu breit ist und daher Bits abgeschnitten werden müssen.

sizeof

Dieser Operator ermittelt die Anzahl der Bytes von Typen und von Objekten, Objekt-Teilen, Objekt-Elementen. Bei Strukturen und Struktur-Typen können Füll-Bytes dabei sein – zwecks Alignment von Array-Elementen.

```
struct kfs { int i; char rel; } Kfs[2];
int Array[10];

sizeof(int)         // 4
sizeof(char)        // 1   (ist immer 1)
sizeof(char*)       // 4

sizeof(struct kfs)  // 8   also 3 Füll-Bytes
sizeof(*Kfs)        // 8
sizeof(Kfs)         // 16
sizeof(Kfs+0)       // 4   Pointer-Kontext erzwungen: +0
sizeof(Kfs[0].i)    // 4
sizeof(Kfs[1].rel)  // 1

sizeof( ( (struct kfs *)0 )->rel )  // 1
sizeof(&( (struct kfs *)8 )->rel )  // 4

sizeof(Array)       // 40
sizeof(*Array)      // 4
sizeof(Array[0])    // 4
sizeof(long[2][3])  // 24
sizeof("")          // 1
sizeof("abc")       // 4
sizeof('A')         // 4
sizeof((char)'A')   // 1
```

Der Typ des Resultats von `sizeof` ist `size_t`.
Fast immer `unsigned`: `typedef unsigned size_t;`
Die oben gezeigte pauschale Klammerung hinter `sizeof` ist nicht zwingend vorgeschrieben. Sie kann weggelassen werden, falls Mißinterpretation mit nachfolgender Syntax ausgeschlossen ist. (▶ 84)