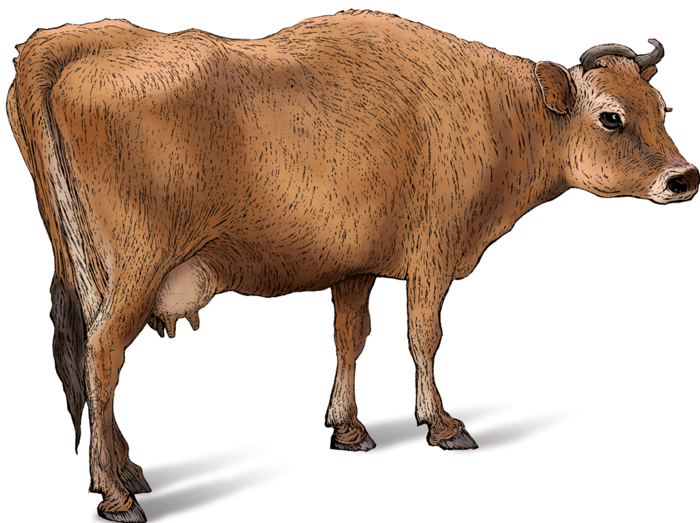
O'REILLY®

2. Auflage
Aktuell zu C18

C kurz & gut

O'Reillys Taschenbibliothek



Ulla Kirch
Peter Prinz

C
kurz & gut

Ulla Kirch und Peter Prinz

O'REILLY®

Ulla Kirch und Peter Prinz

Lektorat: Alexandra Follenius
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Michael Oréal, *www.oreal.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-107-3
PDF 978-3-96010-297-7
ePub 978-3-96010-298-4
mobi 978-3-96010-299-1

2. Auflage
Copyright © 2019 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhalt

Vorwort 5

C-Sprachbeschreibung 7

Grundlagen 7
Elementare Datentypen 15
Konstanten 21
Ausdrücke und Operatoren 25
Typumwandlungen 37
Anweisungen 40
Deklarationen 48
Variablen 50
Abgeleitete Typen 53
Funktionen 71
Bindung von Bezeichnern 80
Präprozessordirektiven 82

Standardbibliothek 97

Standard-Header-Dateien 97
Ein-/Ausgabe 98
Grenzwerte und Klassifizierung von Zahlen 111
Mathematische Funktionen 115
Zeichenklassifizierung/Umwandlung 124
String-Verarbeitung 126

Sortieren und Suchen 131
Speichermanipulationen 133
Dynamische Speicherverwaltung 134
Zeit und Datum 135
Prozesskontrolle 137
Multithreading 142
Internationalisierung 161
Funktionen mit Bereichsüberprüfung 164

Index 169

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren O'Reilly-Büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei oreilly.plus+:

www.oreilly.plus

Vorwort

Die Programmiersprache C wurde in den 70er-Jahren von Dennis Ritchie in den Bell Labs (Murray Hill, USA) entwickelt, um das Betriebssystem UNIX auf einer DEC PDP-11 zu implementieren. Die Ursprünge der Sprache C gehen auf die typenlosen Programmiersprachen BCPL (*Basic Combined Programming Language*, entwickelt von M. Richards) und B (entwickelt von K. Thomson) zurück. 1978 wurde von Brian Kernighan und Dennis Ritchie die erste allgemein zugängliche Sprachbeschreibung veröffentlicht, der sogenannte K&R-Standard.

C ist eine in hohem Maße portable Sprache, die sich an der Architektur der heutigen Rechner orientiert. Der eigentliche Sprachkern ist relativ klein und enthält nur wenige hardwareabhängige Bestandteile. So gehören zum Sprachumfang weder Anweisungen für die Ein-/Ausgabe noch Speicherverwaltungstechniken. Für diese Aufgaben stehen Funktionen in der umfangreichen C-Standardbibliothek zur Verfügung.

In der Praxis ergeben sich daraus wesentliche Vorteile:

- Portabilität auf Quellcodeebene.
- Erzeugung von effizientem Maschinencode.
- C-Compiler sind auf allen gängigen Systemen verfügbar.

Diese Referenz beschreibt im ersten Teil die Programmiersprache C und im zweiten Teil die C-Standardbibliothek. Die Sprachbeschreibung basiert auf der ISO/IEC-Norm 9899, die ursprünglich 1990 von der *International Standards Organization* verabschiedet, 1999 und 2011 wesentlich erweitert und zuletzt 2018 geringfügig geändert wurde.

Die ISO/IEC-Norm 9899 ist im Internet erhältlich unter

http://www.iso.org/standards.html

Der Standard von 1999 (ISO/IEC 9899:1999, kurz: C99) wird heute von allen gängigen C-Compilern unterstützt. Die Erweiterungen im Standard von 2011 (ISO/IEC 9899:2011, kurz: C11) und 2018 (ISO/IEC 9899:2018, kurz: C18) sind noch nicht bei allen C-Compilern implementiert und werden deshalb im Buch mit C11 bzw. C18 gekennzeichnet. Zu den Erweiterungen von C11 gehören z.B. atomare und generische Typen sowie Multithreading. In C18 wurde lediglich das Makro `__STDC_VERSION__` neu definiert, und die Fehlerbehandlung wurde verbessert.

C-Sprachbeschreibung

Grundlagen

Ein C-Programm besteht aus einzelnen »Bausteinen«, den *Funktionen*, die sich gegenseitig aufrufen.

```
/*  Head.c: Das Programm gibt den Anfang einer      *
 *  Textdatei auf die Standardausgabe aus.          *          Kommentar
 *  Aufruf: Head  Dateiname                         */
#include <stdio.h>                                 Präprozessordirektiven
#define  LINES  22
void showPage( FILE *);        // Prototyp                   Deklarationen
int main( int argc, char **argv)                          Die Funktion main()
{
    FILE *fp;      int exit_code = 0;
    if( argc != 2 )
    {
       fprintf(stderr, "Aufruf: Head Dateiname\n");
       exit_code = 1;
    }
    else if( (fp = fopen(argv[1], "r")) == NULL )
    {
       fprintf(stderr, "Fehler beim Öffnen der Datei!\n");
       exit_code = 2;
    }
    else
    {
       showPage(fp);
       fclose(fp);
    }
    return exit_code;
}
void showPage( FILE *fp)       // Eine Bildschirmseite
{                              // ausgeben.            Weitere Funktionen
    int  count = 0;
    char line[81];
    while(count < LINES && fgets(line, 81, fp) != NULL)
    {
       fputs(line, stdout);
       ++count;
    }
}
```

Jede Funktion löst eine bestimmte Aufgabe. Sie ist entweder selbst erstellt oder eine fertige Routine aus der Standardbibliothek. Die Funktion `main()` hat eine besondere Rolle: Sie bildet das steuernde Hauptprogramm. Jede andere Funktion entspricht einem Unterprogramm.

Struktur eines C-Programms

Das obige Beispiel zeigt, wie ein C-Programm strukturiert ist. Das Programm besteht aus den Funktionen `main()` und `showPage()`. Es gibt den Anfang einer Textdatei aus, deren Name beim Starten des Programms in der Kommandozeile angegeben werden muss.

Die *Anweisungen*, aus denen die Funktionen bestehen, bilden zusammen mit den notwendigen Deklarationen und Präprozessordirektiven den *Quellcode* eines C-Programms. Dieser wird bei kleineren Programmen in eine *Quelldatei* geschrieben.

Größere C-Programme bestehen aus mehreren Quelldateien, die getrennt bearbeitet und übersetzt werden können. Dabei werden in einer Quelldatei die Funktionen zusammengefasst, die auch logisch eine Einheit bilden, wie etwa die Funktionen für die Bildschirmausgabe. Informationen, die in mehreren Quelldateien erforderlich sind, wie z.B. Deklarationen, werden in Header-Dateien gestellt. Diese können mit der `#include`-Direktive in eine Quelldatei kopiert werden.

Quelldateien haben die Endung *.c*, Header-Dateien die Endung *.h*. Eine Quelldatei zusammen mit den darin inkludierten Header-Dateien wird *Übersetzungseinheit* genannt.

Die Reihenfolge, in der Funktionen definiert werden, ist nicht vorgeschrieben. Zum Beispiel könnte die Funktion `showPage()` auch vor der Funktion `main()` stehen. Eine Funktion darf jedoch nicht innerhalb einer anderen Funktion definiert werden.

Zur Gestaltung der Quelldatei: Generell gilt, dass der Compiler jede Quelldatei sequenziell bearbeitet und den Inhalt in »Tokens« (kleinste Bestandteile) zerlegt, wie zum Beispiel Funktionsnamen und Ope-

ratoren. Tokens können durch beliebig viele Zwischenraumzeichen getrennt werden, also durch Leer-, Tabulator- oder Newline-Zeichen. Es kommt nur auf die Reihenfolge an und nicht auf ein bestimmtes Layout, wie etwa die Aufteilung in Zeilen und Spalten. Eine Ausnahme bilden die *Präprozessordirektiven*, also die Befehle, die der Präprozessor vor der Kompilierung ausführen soll. Sie beginnen mit dem Doppelkreuz `#` und nehmen stets eine eigene Zeile ein.

Kommentare sind Zeichenfolgen, die entweder durch `/* */` eingeschlossen sind oder mit `//` beginnen und bis zum Zeilenende reichen. In der ersten Übersetzungsphase, in der noch kein Objektcode erzeugt wird, erfolgt die Ersetzung aller Kommentare durch *ein* Leerzeichen.

Außerdem werden die Präprozessordirektiven ausgeführt.

Zeichensätze

Der C-Standard unterscheidet zwei Zeichensätze: Der erste ist der *Quellzeichensatz*, der aus den Zeichen besteht, die in einer Quelldatei verwendet werden dürfen. Der zweite ist der *Ausführungszeichensatz*. Dieser besteht aus den Zeichen, die bei der Ausführung eines Programms interpretiert werden, wie z.B. die Zeichen eines Strings.

Beide Zeichensätze enthalten den *Basiszeichensatz*. Dieser umfasst

- die 2 × 26 Buchstaben des englischen Alphabets:
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
- die zehn Dezimalziffern: `0 1 2 3 4 5 6 7 8 9`
  (wobei der Zeichencode einer von 0 verschiedenen Dezimalziffer um eins größer ist als der Zeichencode des Vorgängers)
- die folgenden 29 Grafikzeichen:
  `! " # % & ' ( ) * + , - . / : ;`
  `< = > ? [ \ ] ^ _ { | } ~`
- die Zwischenraumzeichen:
  Leerzeichen, horizontaler und vertikaler Tab, neue Zeile, neue Seite

Der Ausführungszeichensatz enthält darüber hinaus noch folgende Zeichen:

- das Null-Zeichen `\0`, um das Ende von Strings zu markieren
- die Steuerzeichen, die durch einfache *Escape-Sequenzen* repräsentiert werden, um Ausgabegeräte wie Bildschirme oder Drucker zu steuern

Tabelle 1: Die standardisierten Escape-Sequenzen

Escape-Sequenz	Wirkung bei der Ausgabe	Escape-Sequenz	Wirkung bei der Ausgabe
`\a`	Alert (Ton)	`\'`	das Zeichen `'`
`\b`	Backspace	`\"`	das Zeichen `"`
`\f`	Form Feed	`\?`	das Zeichen `?`
`\n`	Line Feed	`\\`	das Zeichen `\`
`\r`	Carriage Return	`\o \oo \ooo` (o = Oktalziffer)	Zeichen mit dem entsprechenden oktalen Code
`\t` `\v`	Horizontal Tab Vertical Tab	`\xh..` (`h..` = Folge von Hex-Ziffern)	Zeichen mit dem entsprechenden hexadezimalen Code
		`\uhhhh` `\Uhhhhhhhh`	Zeichen mit diesem universellen Zeichennamen

Der numerische Wert eines Zeichens, der sogenannte *Zeichencode*, ist abhängig von der Implementierung. Im C-Standard ist lediglich festgelegt:

- Jedes Zeichen des Basiszeichensatzes wird in einem Byte gespeichert.
- Das Null-Zeichen wird in einem Byte dargestellt, in dem alle Bits den Wert `0` haben.
- Der Zeichencode jeder Dezimalziffer ist um 1 größer als der Code ihres Vorgängers.

In Kommentaren, Strings und Zeichenkonstanten können – abhängig vom jeweiligen Compiler – beliebige andere Zeichen verwendet werden, zum Beispiel das Dollarzeichen oder deutsche Umlaute. Die Nutzung dieser Zeichen kann jedoch Auswirkungen auf die

Portabilität haben. Alle verwendbaren Zeichen bilden den *erweiterten Zeichensatz* (engl. *Extended Character Set*), der stets den Basiszeichensatz umfasst.

Für die Darstellung von Zeichen des erweiterten Zeichensatzes, die zur Speicherung mehr als ein Byte erfordern, gibt es zwei Möglichkeiten:

- Der Typ `wchar_t` (engl. *Wide Character Type*) ist geeignet, alle Zeichen des erweiterten Zeichensatzes in Codes fester Breite darzustellen. `wchar_t` ist ein ganzzahliger Typ, der in den Header-Dateien *stddef.h*, *stdlib.h* und *wchar.h* definiert ist. Viele Implementierungen verwenden die Unicode-Formate UTF-16 oder UTF-32, die den standardisierten ASCII-Code umfassen und Zeichen in zwei bzw. vier Bytes darstellen.
- Multibyte-Zeichen werden in einem oder mehreren Bytes dargestellt. Dabei werden die Zeichen des Basiszeichensatzes in einem Byte gespeichert, und es ist sichergestellt, dass in keinem Multibyte-Zeichen außer dem Null-Zeichen alle Bits auf `0` gesetzt sind. Der UTF-8-Zeichensatz wurde entwickelt, um alle Unicode-Zeichen darzustellen. Er verwendet ein bis vier Bytes zur Darstellung eines Multibyte-Zeichens.

Zeichen, die nicht im Basiszeichensatz enthalten sind, können durch ihren universellen Zeichennamen (engl. *Universal Character Name*) dargestellt werden. Das ist der Unicode-Wert des Zeichens, der mit dem Präfix `\u` oder `\U` beginnt und vier bzw. acht Hexadezimalziffern besitzt.

Beispiel:

```
wchar_t pfund = L'\u00A3'    // alternativ:
                             // L'\U000000A3'
```

Universelle Zeichennamen können in Bezeichnern, Zeichenkonstanten und String-Literalen verwendet werden, um Zeichen darzustellen, die nicht im Basiszeichensatz enthalten sind. Sie werden jedoch nicht von allen Compilern unterstützt.

Der C-Standard unterstützt auch die *Trigraph-Sequenzen*. Diese erlauben es, wichtige grafische Zeichen auch auf Tastaturen einzugeben, die diese Zeichen nicht zur Verfügung stellen. Zum Beispiel kann das Zeichen | durch die Sequenz `??!` dargestellt werden.

Tabelle 2: Die Trigraph-Sequenzen

Trigraph-Sequenz	Bedeutung	Trigraph-Sequenz	Bedeutung
??=	#	??<	{
??(	[	??!	|
??/	\	??>	}
??)	]	??-	~
??'	^		

Bezeichner

Bezeichner (engl. *Identifier*) sind *Namen* von Variablen, Funktionen, Makros, Datentypen usw. Für die Bildung von Bezeichnern gelten die folgenden Regeln:

- Ein Bezeichner besteht aus einer Folge von Buchstaben (`A` bis `Z`, `a` bis `z`), Ziffern (`0` bis `9`), universellen Zeichennamen und Unterstrichen (_).
- Die universellen Zeichennamen müssen Buchstaben und Ziffern einer Sprache repräsentieren.
- Das erste Zeichen darf keine Ziffer sein, auch kein universeller Zeichenname, der eine Ziffer repräsentiert.
- Groß- und Kleinbuchstaben werden unterschieden.
- Ein Bezeichner kann beliebig lang sein. Signifikant sind in der Regel nur die ersten 31 Zeichen.

Schlüsselwörter sind reserviert und dürfen nicht als Bezeichner verwendet werden. Die Schlüsselwörter in alphabetischer Reihenfolge:

auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

Daneben gibt es *reservierte Bezeichner*, wie z.B. Namen von Standardfunktionen oder Standardmakros, die in einem Programm nicht als Bezeichner für Variablen, Funktionen, Datentypen usw. verwendet werden dürfen. Auch Namen, die mit zwei Unterstrichen oder einem Unterstrich und einem Großbuchstaben beginnen, sind reservierte Bezeichner.

Für *externe Namen* (Bezeichner von Funktionen und Variablen mit externer Bindung) sind weitere Einschränkungen möglich, die vom jeweiligen Linker abhängen: In portablen C-Programmen sollten externe Namen so gewählt werden, dass nur die ersten acht Zeichen signifikant sind, auch wenn der Linker Groß- und Kleinbuchstaben *nicht* unterscheidet.

Beispiele für Bezeichner:

> gültig: `a, DM, dm, FLOAT, _var1, topOfWindow`
> ungültig: `do, 586_cpu, zähler, nl-flag, US_$`

Namensklassen und Geltungsbereiche

Jeder Bezeichner gehört zu genau einer der vier Namensklassen, nämlich:

- Namen von *Marken* (engl. *Labels*).
- Namen von Strukturen, Unions und Aufzählungen (*Tags*). Das sind Namen, die einem der Schlüsselwörter *struct*, *union* oder *enum* folgen (siehe den Abschnitt *Abgeleitete Typen* auf Seite 53).
- Namen von *Struktur-* oder *Union-Komponenten*. Jeder Struktur- oder Union-Typ hat eine separate Namensklasse für seine Komponenten.
- Alle anderen Bezeichner. Diese heißen auch *gewöhnliche Bezeichner*.

Bezeichner verschiedener Namensklassen dürfen identisch sein. So kann z.B. ein Label-Name auch gleichzeitig als Funktionsname verwendet werden. Am häufigsten werden jedoch Namensklassen im Zusammenhang mit Strukturen eingesetzt: Der gleiche Bezeichner kann als Struktur-, Komponenten- und Variablenname benutzt werden.

Beispiel: `struct person { char *person; /*...*/} person;`

Auch die Komponenten verschiedener Strukturen können gleiche Namen haben.

Jeder Bezeichner hat innerhalb der Quelldatei seinen *Geltungsbereich*. Das ist der Teil, in dem der Bezeichner verwendet werden kann. Die vier möglichen Geltungsbereiche sind:

Funktionsprototyp
: Bezeichner, die in der Parameterliste eines Prototyps stehen, haben den Geltungsbereich *Funktionsprototyp*. Dieser endet mit dem Prototyp. Solche Bezeichner haben damit nur den Charakter eines Kommentars.

Funktion
: Nur Namen von Marken (*Labels*) haben den Geltungsbereich *Funktion*. Der Geltungsbereich besteht aus dem Funktionsblock, in dem das Label definiert ist. Label-Namen müssen in einer Funktion eindeutig sein. Die `goto`-Anweisung führt einen Sprung zu einem Label in derselben Funktion aus.

Block
: Bezeichner, die in einem Block deklariert und keine Marken sind, haben den Geltungsbereich *Block*. Auch die Parameter in einer Funktionsdefinition haben den Geltungsbereich Block. Er beginnt mit der Deklaration und endet mit der Klammer }, die den Block schließt.

Datei
: Bezeichner, die außerhalb aller Blöcke deklariert sind und zu keinem Prototyp gehören, haben den Geltungsbereich *Datei*. Dieser beginnt mit der Deklaration des Bezeichners und reicht bis zum Ende der Quelldatei.

Ein Bezeichner, der kein Label-Name ist, muss nicht notwendigerweise in seinem ganzen Geltungsbereich *sichtbar* sein: Wird nämlich der Bezeichner mit derselben Namensklasse in einem inneren Block erneut deklariert, ist die äußere Deklaration vorübergehend verdeckt. Die äußere Deklaration wird wieder sichtbar, wenn der Geltungsbereich für die innere Deklaration endet.

Elementare Datentypen

Der *Typ* einer Variablen bestimmt, wie viel Speicher diese Variable belegt und wie das dort gespeicherte Bitmuster interpretiert wird. Entsprechend bestimmt der Typ einer Funktion, wie der Return-Wert interpretiert werden muss.

Datentypen sind entweder vordefiniert oder abgeleitet. Die vordefinierten Datentypen in C sind die *elementaren Typen* (engl. *Basic Types*) und der Typ `void`. Die elementaren Typen bestehen aus den *ganzzahligen Typen* (engl. *Integer Types*) und den *Gleitpunkttypen* (engl. *Floating Types*).

Ganzzahlige Typen

Es gibt fünf ganzzahlige Typen mit Vorzeichen, nämlich `signed char`, `short int` (kurz: `short`), `int` , `long int` (kurz: `long`) und `long long int` (kurz: `long long`). Jedem dieser Typen entspricht ein ganzzahliger Typ ohne Vorzeichen, der den gleichen Speicherplatz belegt. Den vorzeichenlosen Typ erhält man durch Voranstellen des Typspezifizierers `unsigned`, also z.B. `unsigned int`.

Die Typen `char`, `signed char` und `unsigned char` sind formal verschieden. Je nach Einstellung des Compilers ist aber `char` gleichbedeutend mit `signed char` oder mit `unsigned char`. Dagegen hat der Typspezifizierer `signed` für die Datentypen `short`, `int`, `long` und `long long` keine Bedeutung, da sie stets mit Vorzeichen interpretiert werden. So bezeichnen z.B. `short` und `signed short` denselben Datentyp.

Die Größe der ganzzahligen Typen ist nicht festgelegt. Es gilt aber folgende Reihenfolge: `char <= short <= int <= long <= long long`. Der Typ `short` ist hierbei mindestens 2 Byte, der Typ `long` mindestens 4 Byte und der Typ `long long` mindestens 8 Byte groß. Die aktuellen Wertebereiche können der Header-Datei *limits.h* entnommen werden.

Außerdem gibt es den Typ `_Bool`, der mit C99 zur Darstellung boolescher Werte eingeführt wurde. Der boolesche Wert `true` (»wahr«) wird durch `1` und `false` (»falsch«) durch `0` repräsentiert. Sofern die

Header-Datei *stdbool.h* inkludiert ist, können auch `bool` statt `_Bool` und die Bezeichner `true` und `false` verwendet werden.

Tabelle 3: Die ganzzahligen Typen mit Speicherplatz und Wertebereich

Typ	Speicherplatz	Wertebereich (dezimal)
`_Bool`	1 Byte	0 und 1
`char`	1 Byte	–128 bis 127 bzw. 0 bis 255
`unsigned char`	1 Byte	0 bis 255
`signed char`	1 Byte	–128bis 127
`int`	2 Byte oder 4 Byte	–32.768 bis 32.767 oder –2.147.483.648 bis 2.147.483.647
`unsigned int`	2 Byte oder 4 Byte	0 bis 65.535 oder 0 bis 4.294.967.295
`short`	2 Byte	–32.768 bis 32.767
`unsigned short`	2 Byte	0 bis 65.535
`long`	4 Byte	–2.147.483.648 bis 2.147.483.647
`unsigned long`	4 Byte	0 bis 4.294.967.295
`long long`	8 Byte	–9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
`unsigned long long`	8 Byte	0 bis 18.446.744.073.709.551.615

Mit C99 wurden in der Header-Datei *stdint.h* ganzzahlige Datentypen mit einer vorgegebenen Breite eingeführt. Die *Breite N* eines ganzzahligen Typs ist die Anzahl der Bits, die zur Darstellung von Werten des Typs einschließlich des Vorzeichenbits verwendet wird (i. Allg. $N = 8, 16, 32, 64$).

Tabelle 4: Ganzzahlige Typen vorgegebener Breite

Typ	Bedeutung
`int`*N*`_t`	Breite exakt *N* Bits.
`int_least`*N*`_t`	Breite mindestens *N* Bits.
`int_fast`*N*`_t`	Schnellster Typ mit Mindestbreite von *N* Bits.
`intmax_t`	Breite maximal (größter ganzzahliger Typ).
`intptr_t`	Breit genug, um den Wert eines Zeigers zu speichern.

Beispielsweise ist `int16_t` ein `int`-Typ, dessen Breite exakt 16 Bit ist, und `int_fast32_t` ist der schnellste `int`-Typ, der mindestens 32 Bit breit ist. Die Definition dieser Typen ist für die Breiten N = 8, 16, 32 und 64 gefordert. Andere Typen, wie z.B. `int24_t`, sind optional.

Beispiel:

```
int16_t val = -10;    // Integer-Variable
                      // Breite: exakt 16 Bit
```

Zu jedem der oben angegebenen `signed`-Typen gibt es auch einen `unsigned`-Typ mit dem Präfix `u`. So ist `uintmax_t` der `unsigned-int`-Typ maximaler Breite.

Seit C11 sind in der Header-Datei *uchar.h* die Typen `char16_t` und `char32_t` für Unicode-Zeichen definiert, die in 16 Bit oder in 32 Bit dargestellt werden können. Die Typen entsprechen den Typen `uint_least16_t` bzw. `uint_least32_t`. Zeichen des Typs `char16_t` sind in Implementierungen, die das Makro `__STDC_UTF_16__` definieren, im UTF-16-Format codiert. Ist das Makro `__STDC_UTF_32__` definiert, sind Zeichen des Typs `char32_t` im UTF-32-Format codiert.

Reelle und komplexe Gleitpunkttypen

Zur Darstellung von Zahlen mit gebrochenem Anteil (= reelle Zahlen) gibt es die drei Typen `float`, `double` und `long double`. Diese drei Datentypen werden *reelle Gleitpunkttypen* genannt.

Die Größe des Speicherplatzes und die interne Darstellung sind in C nicht festgelegt; sie können also von Compiler zu Compiler verschieden sein. Üblicherweise wird jedoch der IEEE-Standard IEC 60559 für binäre Gleitpunktarithmetik verwendet (IEEE = *Institute of Electrical and Electronics Engineers*). Von dieser Darstellung wird in der nachfolgenden Tabelle ausgegangen.

Tabelle 5: Reelle Gleitpunktzahlen

Typ	Speicherplatz	Wertebereich (dezimal, ohne Vorzeichen)	Genauigkeit (dezimal)
`float`	4 Byte	1.2E-38 bis 3.4E+38	6 Stellen
`double`	8 Byte	2.3E-308 bis 1.7E+308	15 Stellen
`long double`	10 Byte	3.4E-4932 bis 1.1E+4932	19 Stellen