

# Programmieren in C

Skript zur Vorlesung

WS 2016/17

9. Auflage Oktober 2016 (V9.0)

© IT.SERVICES

Ruhr-Universität Bochum | D-44780 Bochum

Nur zum persönlichen Gebrauch durch die Veranstaltungsteilnehmer bestimmt.

# Inhalt

---

1. Grundlagen	1
2. Standard-Datentypen	9
3. Konvertierungen für Standard-Datentypen	15
4. Ausdrücke	17
5. Aufzählungen (enum)	27
6. Anweisungen, Kontrollstrukturen	29
7. Funktionen	37
8. Parameter und Argumente	45
9. Eigenschaften von Vereinbarungen	47
10. Zeiger	55
11. Arrays	61
12. Zeiger und Arrays	67
13. Strings und Arrays aus Zeichen	77
14. Strukturen (struct, union)	81
15. Dynamische Datenstrukturen	89
16. Typdefinitionen	97
17. Rechnen mit Bits und Bytes	99
18. Sicherheitsaspekte	105
A. Ein- und Ausgabe	107
B. Zufallszahlen	113
C. Standard-Bibliothek im Überblick	115
D. Präprozessor-Direktiven	119
E. Unbehandelte Themen	123
F. Was gibt's Neues? (Sprachnormen)	125
G. ASCII-Codetabelle	127
Index	129

# Vorwort

---

Dieses Skript soll als Hilfsmittel zur Vorlesung *Programmieren in C* im Wintersemester 2016/17 dienen. Es ist nicht als Lehrbuch zum Selbststudium ausgelegt, erhebt keinen Anspruch auf thematische Vollständigkeit und ist auch nicht als Sprachreferenz gedacht.

Die in der Vorlesung vorgestellten Beispiele sind zum großen Teil nicht in diesem Skript enthalten. Stattdessen werden diese für die Teilnehmer zum Herunterladen und Ausprobieren im LMS Moodle (<https://moodle.ruhr-uni-bochum.de>) bereitgestellt.

Andererseits umfasst das Skript auch Ergänzungen. So werden einige Randthemen vervollständigt, welche in der Vorlesung nur nebenläufig angesprochen werden. Die Praxis vergangener Vorlesungen lässt vermuten, dass einige der Themen aus Zeitgründen in der Vorlesung leider nicht behandelt werden können.

*Reinhard Mares*



# 1. Grundlagen

---

## Charakterisierung der Programmiersprache C

### **C ist eine höhere Programmiersprache.**

C erlaubt das Erstellen von Programmen (Software) in einer abstrakten Sprache, die für den Programmierer verständlich, nicht jedoch auf einem System (Computer, Prozessor, ...) direkt ausführbar ist. C-Programme müssen daher vor der Ausführung übersetzt werden.

### **C ist eine imperative Programmiersprache.**

Ein C-Programm beinhaltet eine Folge von Anweisungen, die den Zustand des Programmspeichers Schritt für Schritt verändern.

### **C ist eine prozedurale Programmiersprache.**

Hierunter versteht man den Ansatz, Programme aus kleineren Lösungen für Teilprobleme aufzubauen. Derartige Prozeduren werden in C als Funktionen bezeichnet.

### **C ist eine strukturierte Programmiersprache.**

Strukturierte Programmierung bezeichnet einen Konsens über fundamentale Leitsätze (Lehrmeinung, Paradigma), welcher die Programmerstellung beschränkt auf wenige Kontrollstrukturen und das Paradigma der prozeduralen Programmierung. – Dies wird von C unterstützt. (Leider kann man aber auch unstrukturierte C-Programme erstellen.)

### **C ist genormt (und auch wieder nicht).**

C ist entstanden ca. 1972 und trägt die Handschrift von Dennis Ritchie (Bell Labs). Der ursprüngliche Sprachumfang darf mittlerweile als historisch angesehen werden. Davon abgeleitet haben sich folgende Sprachstandards:

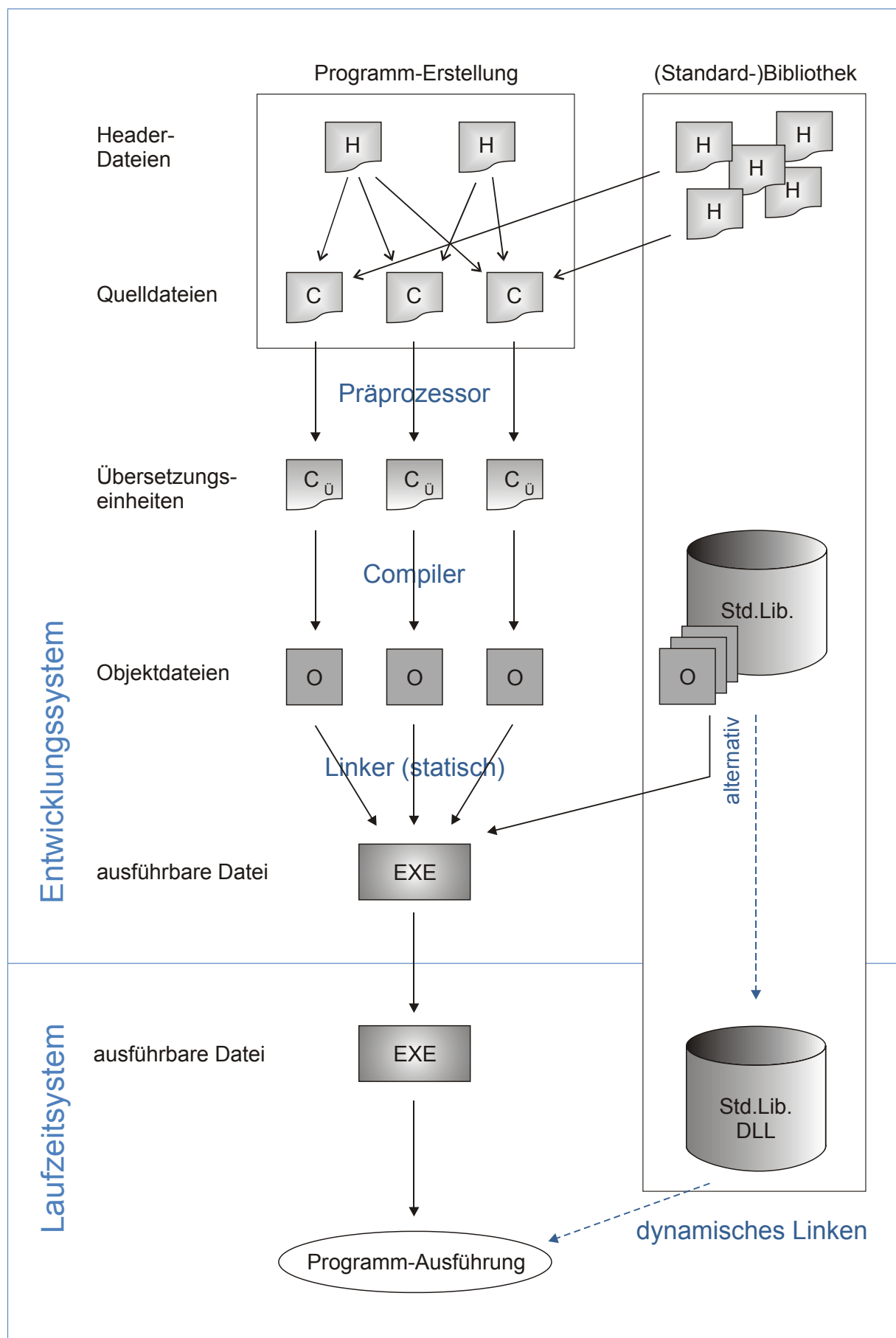
- ISO/IEC 9899:1990 (**C90**), praktisch identisch zu ANSI X3.159-1989 (**C89**)
- ISO/IEC 9899/AMD1:1995 (**C95**), Ergänzung *Amendment 1* zu C90
- ISO/IEC 9899:1999 (**C99**)
- ISO/IEC 9899:2011 (**C11**)

In der Praxis stellt sich leider heraus, dass die neueren Normen häufig nur unvollständig realisiert worden sind. Erfreulicherweise stimmen jedoch die meisten der in diesem Skript vorgestellten Sprachmittel quer über alle (Nicht-)Standard-Systeme immer noch überein.

Auf einige Restriktionen in C89/C90 werden wir dabei eingehen; einige interessante Erweiterungen in den Standards C99 und C11 führen wir zusätzlich an. Programmierung in einem älteren Stil, der von C99 nicht mehr geduldet wird, stellen wir dagegen generell nicht vor. Den neuesten Standard C11 können wir im Wesentlichen mit C99 gleichsetzen, da die für einen C-Einsteiger nicht-relevanten Teile ohnehin unbehandelt bleiben.

Der Anhang F enthält eine Liste der wichtigsten Neuerungen von C95, C99 und C11.

## Vom C-Programm zur Ausführung



## C-Quelldateien

Ein C-Programm besteht aus einer oder mehreren **Programmdateien (Quelldateien)**. Diese enthalten die C-Bestandteile als Text und sind damit für den Programmierer per Editor bearbeitbar. Die klassische Endung der Dateinamen lautet: `.c`

Eine Programmdatei beinhaltet insbesondere eine Folge von Vereinbarungen, wobei mindestens eine dieser Vereinbarungen die Definition einer **Funktion** sein wird. Funktionen beinhalten Anweisungen.

- Vorläufig erstellen wir in unseren Beispielen nur eine einzige Programmdatei, darin nur eine einzige Funktion mit Namen `main`.

Es gibt noch eine besondere Art von Programmdateien: die **Header-Dateien**. Diese beinhalten Vereinbarungen, welche von mehreren Programmdateien benötigt werden und dort eingeblendet werden. Vor der eigentlichen Übersetzung fügt ein **Präprozessor** (Software) eine angefragte Header-Datei in die entsprechende Programmdatei textuell ein. Hierdurch entstehen dann erst die eigentlichen **Übersetzungseinheiten**, welche vom Compiler verarbeitet werden.

Header-Dateien ermöglichen somit eine gewisse Modularität auf der C-Programmirebene, jedoch auf äußerst primitive Weise. Die Namen der Header-Dateien enden nach Konvention auf: `.h`

- Vorläufig erstellen wir noch keine eigenen Header-Dateien; wir müssen jedoch Header der Standard-Bibliothek in unsere Beispiele importieren.

## Standard-Bibliothek

Neben den selbst erstellten Programmdateien benötigt man zur Programmerstellung auch noch die **Standard-Bibliothek**. Diese Sammlung beinhaltet insbesondere vorgefertigte Funktionen für spezielle Aufgaben. Bereits in unseren ersten Beispielen sind wir darauf angewiesen, den Teil zur Ein- und Ausgabe zu nutzen.

## Übersetzen, Binden und Ausführen

Die Übersetzungseinheiten müssen jetzt durch einen **Compiler** übersetzt werden. Durch diesen Vorgang entstehen „bedeutungsgleiche“ Programmstücke im Objektcode. Nun muss man nur noch die Einzelstücke aus selbsterstellten Funktionen und Bibliotheksfunktionen zusammenpuzzeln.

Eine Möglichkeit besteht darin, alle erforderlichen Teile zu einer einzigen ausführbaren Datei zusammenzubinden. Dieser Vorgang erfolgt typischerweise während der Programmentwicklung als letzter Schritt (**statisches Binden/Linken**).

Alternativ besteht auch die Möglichkeit, den Objektcode der Bibliothek insgesamt auf dem ausführenden System abzulegen (Laufzeitbibliothek). Bei Ausführung des Programms steht damit benötigter Code ebenfalls bereit. Der Bezug zu diesen Programmteilen wird aber erst beim Ladevorgang bzw. zur Laufzeit aufgelöst (**dynamisches Binden/Linken**).

Mischformen der beiden Link-Arten sind praktikabel. Im vorausgehenden Diagramm wird statisch gebunden, alternativ aber auch die Laufzeitbibliothek dynamisch gelinkt (gestrichelte Pfeile rechts). Die Möglichkeit für den Programmierer, eine eigene Bibliothek zu erstellen, ist nicht eingezeichnet.

Damit stehen wir kurz vor der Ausführung. Das erstellte ausführbare Programm (jetzt im Maschinencode) muss nur noch in den Arbeitsspeicher des ausführenden Systems geladen und gestartet werden. In modernen Betriebssystemen gehören *Loader* und *Process Scheduler* zu den elementaren Bestandteilen.

### Zusammenfassung

Für die Erstellung und Ausführung eines C-Programms benötigt man also diverse Softwareprodukte: Editor (möglichst syntaxorientiert), Präprozessor, Compiler, Linker und die Standard-Bibliothek. Gerne auch noch einen *Debugger* zur Fehlerverfolgung und eine Projekt-Verwaltung.

Moderne PC-Entwicklungssysteme integrieren diese Software-Werkzeuge und erlauben die Ausführung eines C-Programms auf Knopfdruck, ohne dass man sich mit den einzelnen Teilen auseinandersetzen muss. Die konkreten Vorgänge hängen ab vom jeweils gewählten Entwicklungssystem und vom Zielsystem, also dem System, auf dem der Maschinencode letztendlich ausgeführt werden soll. Eine Liste von Entwicklungssystemen findet sich am Ende dieses Kapitels.

### Erstes Beispiel

<code>#include &lt;stdio.h&gt;</code>	← Hier wird eine Header-Datei importiert; diese trifft Vereinbarungen zur Ein-/Ausgabe.
<code>int main ()</code>	← Definition der Funktion <code>main</code> : Die Anweisungen dazu folgen innerhalb der geschweiften Klammern.
<code>{</code>	
<code>printf ("hello, world\n");</code>	← 1. Anweisung: Es wird eine vorhandene Funktion aus der E/A-Bibliothek zur Ausgabe aufgerufen.
<code>return 0;</code>	← 2. Anweisung: Die Funktion <code>main</code> wird beendet.
<code>}</code>	

Eine Funktion (hier: `printf`) wird aufgerufen über ihren Namen mit einer Liste von Argumenten in runden Klammern. In diesem Beispiel ist es allerdings nur ein einziges Argument. Das nachfolgende Semikolon schließt den Ausdruck mit dem Funktionsaufruf als Anweisung ab.

Das Argument ist hier eine Zeichenkette (String) eingeschlossen in Doppelapostrophe. Dieser String wird ausgegeben, im Regelfall auf dem Monitor. Die Zeichenfolge `\n` bewirkt einen Zeilenwechsel (Sprung auf den Anfang der nächsten Zeile).

Die im Beispiel vorkommenden Zeilenwechsel, Leerzeile und Einrückungen dienen der besseren Lesbarkeit des Programms. Syntaktisch korrekt ließe sich das Beispiel auch mit nur 2 Zeilen formulieren. Wir werden mit jedem neuen Sprachmittel implizit einen pragmatischen Vorschlag unterbreiten, wie der Programmcode gut lesbar einzusetzen ist. Die syntaktischen Grundlagen hierfür folgen noch in diesem Kapitel.

Das Beispiel wird in einer Programmdatei beispielsweise mit dem Namen `main.c` abgelegt. Die Datei wird danach übersetzt (genauer: Präprozessor → Compiler → Linker). Danach kann das Programm geladen und gestartet werden.

Die Ausgabe bei Ausführung des Programms:

`hello, world`

Mit einem Zeilenwechsel nach dem letzten Zeichen `'d'`.



## Syntaktische Grundlagen

### Zeichensatz

Eine Übersetzungseinheit besteht aus:

- Buchstaben: A–Z a–z \_ (keine Umlaute, ohne ß)
- Ziffern: 0–9
- Sonderzeichen, z.B. " ; )
- Leerraum (*white spaces*) (Leerzeichen, Tabulatoren, Zeilen- und Seitenende)

Groß-/Kleinschreibung ist in C von Bedeutung!

### Syntaktische Einheiten (Eingabesymbole, *tokens*)

Aus den verfügbaren Zeichen werden nun die folgenden syntaktischen Einheiten gebildet.

- **reservierte Wörter** (Schlüsselwörter)

Diese haben eine unveränderbare syntaktische und semantische Bedeutung.

<code>alignof</code> <sup>C11</sup>	<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>	<code>const</code>
<code>continue</code>	<code>default</code>	<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>
<code>extern</code>	<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>	<code>inline</code> <sup>C99</sup>
<code>int</code>	<code>long</code>	<code>register</code>	<code>restrict</code> <sup>C99</sup>	<code>return</code>	<code>short</code>
<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>struct</code>	<code>switch</code>	<code>typedef</code>
<code>union</code>	<code>unsigned</code>	<code>void</code>	<code>volatile</code>	<code>while</code>	
<code>_Alignas</code> <sup>C11</sup>	<code>_Atomic</code> <sup>C11</sup>	<code>_Bool</code> <sup>C99</sup>	<code>_Complex</code> <sup>C99</sup>	<code>_Generic</code> <sup>C11</sup>	
<code>_Imaginary</code> <sup>C99</sup>	<code>_Noreturn</code> <sup>C11</sup>	<code>_Static_assert</code> <sup>C11</sup>	<code>_Thread_local</code> <sup>C11</sup>		

- **Namen** (Bezeichner)

Namen müssen ungleich der reservierten Wörter sein. Sie bestehen aus Buchstaben (inkl. Unterstrich) und Ziffern, beginnen mit einem Buchstaben. Mindestens die ersten 31 Zeichen sind zur Unterscheidung verwendeter Namen von Bedeutung. Systemabhängig dürfen Namen, welche gelinkt werden müssen, allerdings noch weiter eingeschränkt werden.

Es gibt ferner eine Reihe von Empfehlungen und Übereinkommen – nicht syntaktischen Regeln – bezüglich einer geeigneten Namenswahl. In diesem Skript werden für Namen überwiegend Kleinbuchstaben genutzt, von einigen Fällen der Großschreibung abgesehen.

- Normalfall: Kleinschreibung, bei Bedarf zusätzliche Ziffern nach einem Unterstrich  
Bspl.: `x anton berta main printf hal_9000`
- Namen aus zusammengesetzten Begriffen mit großen Wortanfängen im Inneren  
Bspl.: `naechstesJahr registerEintrag xAchse`
- große Anfangsbuchstaben nur für Etiketten (Kapitel 5, 14) und `typedef` (Kapitel 16)  
Bspl.: `Tag Farbe Verweis Punkt Strecke Bruch`
- durchgängig groß nur in Aufzählungen (Kapitel 5) und für PP-Makros (Anhang D), bei Bedarf mit zusätzlichen Unterstrichen zur Trennung von Begriffen  
Bspl.: `ROT GELB BLAU ANZAHL PI MINIMUM INT_MAX`

- **Literale/Konstanten**

Diese bezeichnen (konstante) Werte.

Bspl.: `4711` (ganze Zahl)    `"hello, world\n"` (eine Zeichenkette<sup>1</sup>)

- **Operatoren**

Hierdurch werden Rechenoperationen angestoßen, beispielsweise `+` `-` `*` `/` für die vier Grundrechenarten.

- **Separatoren**

Diese Zeichen dienen zur syntaktischen Trennung von Sprachbestandteilen. In unserem ersten Beispiel fassen die Klammern `{ }` die Anweisungen der Funktion zu einem Block zusammen, das Semikolon schließt die zwei vorhandenen Anweisungen jeweils ab.

- **Kommentare**

Diese dienen der Programmdokumentation und haben keine semantische Bedeutung. Sie werden durch die Sonderzeichen `/*` eingeleitet und durch `*/` beendet<sup>1</sup>.

Bspl.: `/* Dieser Kommentar geht über mehr  
als eine Zeile. Er endet hier: */`

Die Einleitung einer Kommentarzeile durch `//` ist erst ab dem Standard C99 möglich.

Bspl.: `// Dieser Kommentar umfasst nur diese eine Zeile.  
return 0; // Der Kommentar folgt direkt einer Anweisung.`

## Formatierung der Programmdateien

Folgen reservierte Wörter, Bezeichner oder Literale unmittelbar aufeinander, so **müssen** sie durch Leerraum oder einen Kommentar getrennt werden. Zwischen den syntaktischen Einheiten **dürfen** beliebig viel Leerraum oder Kommentare stehen.

Bspl.: `int main ( ) { printf ( "Hello, world!\n" ) ; return 0 ; }`

Die Kästchen markieren die syntaktischen Einheiten in unserer ersten Funktion `main`. Dazwischen und am Anfang/Ende darf beliebiger Leerraum eingefügt werden.

Zwischen dem reservierten Wort `int` und dem Bezeichner `main` muss Leerraum eingefügt werden, ebenso zwischen `return` und `0`.

(Das Leerzeichen hinter dem Komma ist dagegen Bestandteil der konstanten Zeichenkette.)

Diese Flexibilität erlaubt unterschiedliche Formatierungsstile.

<sup>1</sup> Zeichenketten und Kommentare dürfen alle darstellbaren Zeichen (wie z.B. auch Umlaute und ß) enthalten. Aufgrund der Problematik unterschiedlicher Code-Tabellen empfehlen wir jedoch eine Beschränkung auf den ASCII-Zeichenvorrat (siehe Tabelle im Anhang G).

Wir bevorzugen einen etwas modifizierten **Allman-Stil**, bei dem auch die öffnende geschweifte Klammer eine eigene Zeile erhält. Im Normalfall rücken wir hier im Skript um 2 Zeichen ein.

```
Bspl.: #include <stdio.h>

                                     // Leerzeile (optional)

int main ()
{
    printf ("hello, world\n");
    return 0;
}
```

Zum Vergleich unser Eingangsbeispiel im **1TBS-Stil** in nur 6 Zeilen mit 4 Zeichen Einrückung.

```
Bspl.: #include <stdio.h>

                                     // Leerzeile (optional)

int main() {
    printf("hello, world\n");
    return 0;
}
```

**Präprozessor-Direktiven** unterliegen eigenen syntaktischen Regeln (siehe Anhang D). Daher sollten die erforderlichen `#include`-Direktiven stets exakt so eingefügt werden, wie wir dies vorgeben.

## Ergänzende Hinweise zu Entwicklungssystemen

### C und C++

Die meisten Entwicklungssysteme für C sind heutzutage tatsächlich C++-Entwicklungssysteme. Im Idealfall gibt es zu Beginn der Programmerstellung eine Option, welche die Sprachauswahl regelt. Entscheidend hierfür ist aber letztendlich die Dateierweiterung. Daher ist besonders darauf zu achten, dass nicht unbeabsichtigt Programmdateien mit der Dateierweiterung `.cpp` erstellt werden.

### Konsolfenster

Wird ein C-Programm in einem Betriebssystem mit grafischer Benutzeroberfläche ausgeführt, so neigen die entstehenden Konsolfenster dazu, sich bei Programmende automatisch wieder zu schließen. So „blitzt“ unser erstes Beispiel nur kurz auf dem Bildschirm auf und verschwindet sofort wieder.

Im vorgesehenen Referenzsystem tritt dieser Vorfall in der Entwicklungsumgebung nicht auf. Anders ist jedoch das Verhalten beim Start eines Programms direkt aus dem Betriebssystem heraus.

Die Abhilfe kann gegebenenfalls darin bestehen, am Ende des Programms eine kurze Eingabe anzufordern. Unser Programm wartet dann bis zur Betätigung der Return-Taste. Eine mögliche Anweisung dazu wäre:

```
    getchar ();
```

am Ende der Funktion `main`

Alternativ ist, wenn das System es versteht, auch dies eine gute Lösung:

```
#include <stdlib.h>
```

am Anfang der Quelldatei

```
system ("PAUSE");
```

am Ende der Funktion `main`

## Vorlagen

Entwicklungssysteme geben häufig bei der Programmerstellung eine Schablone für den äußeren Rahmen eines Programms vor. Diese kann bereits die Vereinbarung der Funktion `main` und eine Lösung gegen das vorzeitige Schließen des Konsolfensters umfassen. Der Vorschlag weicht häufig von der Form unserer Beispiele ab. Dies sollte nicht beunruhigen; es sei freigestellt, ob man den gegebenen Vorschlag umschreibt oder ihn so belässt.

## Referenzsystem

Als Referenzsystem in der Vorlesung wird **Code::Blocks** (für Windows) mit integriertem **GNU C/C++ Compiler** genutzt.

Dieses Skript enthält keine weitere Information zu diesem Entwicklungssystem. Teilnehmer können stattdessen eine Einführung aus dem LMS Moodle herunterladen. Diese enthält neben anderen Dingen auch Hinweise, welche Einstellungen zum Sprachumfang alternativ nach C90-, C99- oder C11-Norm vorzunehmen sind.

## Beispiele für Entwicklungssysteme

- ✓ **Code::Blocks (Linux, MacOS, Windows)**  
<http://www.codeblocks.org>
- Microsoft Visual Studio (Windows)  
<http://studierendensoftware.ruhr-uni-bochum.de>
  - Programm „MS DreamSpark Premium“ (vormals MSDN AA) für Studierende berechtigter Fachrichtungen
  - Programm „MS DreamSpark“ für alle Studierenden
- Unix/Linux: Ein C-Compiler (cc oder gcc) ist im Regelfall bereits installiert.

Eine weitere Übersicht:

<http://www.thefreecountry.com/compilers/cpp.shtml>

## 2. Standard-Datentypen

### Datentypen

Ein C-Programm beinhaltet eine Folge von Anweisungen, die den Zustand der Datenobjekte im Programmspeicher Schritt für Schritt verändern. Diese Objekte sind typgebunden; d.h. es wird in C vereinbart, zu welchem Typ ein konkretes Objekt gehört.

Der Datentyp bestimmt dann:

- die Werthaltigkeit des Objekts  
So ist festgelegt, welche Werte ein Objekt dieses Typs annehmen kann. Dadurch bestimmt wird auch die Größe des Objekts im Arbeitsspeicher.
- die anwendbaren Operationen  
Ebenso ist geregelt, mit welchen Mitteln das Objekt in C verarbeitet werden kann.

### Ganzzahlige Typen

Diese Typen dienen zur Realisierung von ganzen Zahlen. Die Werthaltigkeit ist beschränkt auf einen zusammenhängenden Zahlbereich, d.h. es gibt insbesondere einen kleinsten und einen größten Wert für den jeweiligen Typ. Der kleinste Wert kann negativ sein oder aber Null.

Die Basisdatentypen sind `char` und `int`. Objekte vom Typ `char` sind meistens 8 bit groß; es ist nicht festgelegt, ob deren Wert vorzeichenbehaftet ist. Für `int` sollte es die „natürliche“ Objektgröße für ganze Zahlen im Zielsystem sein, z.B. 16 bit, 32 bit, 64 bit.

Zusätzlich gibt es Varianten dieser elementaren Typen. `short` bzw. `long` spezifizieren einen kleineren bzw. größeren Wertebereich, sofern möglich. `unsigned` bedeutet, dass es keine Vorzeichen gibt. `signed` erlaubt explizit positive und negative Werte.

Hieraus ergibt sich folgende Liste der Datentypbezeichnungen. Der Wertebereich/Speicherbedarf nimmt in dieser Tabelle von oben nach unten nicht ab.

mit Vorzeichen	ohne Vorzeichen	Vorzeichen unbestimmt
<code>signed char</code>	<code>unsigned char</code>	<code>char</code>
<del><code>signed short</code></del> <del><code>int</code></del>	<code>unsigned short</code> <del><code>int</code></del>	
<del><code>signed</code></del> <code>int</code>	<code>unsigned</code> <del><code>int</code></del>	
<del><code>signed long</code></del> <del><code>int</code></del>	<code>unsigned long</code> <del><code>int</code></del>	

Die reservierten Wörter `signed` und `int` dürfen in einigen Fällen auch entfallen; wo dies möglich ist, haben wir diese durchgestrichen.

Üblicherweise benutzen wir in unseren Beispielen den Typ `int` für ganze Zahlen. Wenn es uns wichtig erscheint, dass kein Vorzeichen vorliegt (natürliche Zahl), verwenden wir hier und da auch `unsigned int`. Zum Gebrauch der `char`-Datentypen folgt gleich noch ein spezieller Abschnitt.

Die Header-Datei `<limits.h>` enthält die konkreten, implementationsabhängigen Grenzwerte der ganzzahligen Typen. Diese Datei kann importiert werden und die enthaltenen Werte dann per Programm ausgewertet werden. Hier exemplarisch und auszugsweise die Daten aus einem ungenannten Entwicklungssystem. Weitere Details im Anhang C.

```
#define SHRT_MIN    (-32768)           /* minimum (signed) short value */
#define SHRT_MAX     32767            /* maximum (signed) short value */
#define INT_MIN      (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX       2147483647       /* maximum (signed) int value */
#define LONG_MIN      (-2147483647L - 1) /* minimum (signed) long value */
#define LONG_MAX       2147483647L     /* maximum (signed) long value */
```

Man erkennt, dass in diesem Fall der Typ `int` vom Bereich her mit `long` übereinstimmt.

### Literale (Konstanten) der int-Typen

Ganzzahlige Literale lassen sich in verschiedenen Formen zur Basis 10, 8 oder 16 darstellen.

- Die dezimale Darstellung (Basis 10) besteht aus einer Folge von Ziffern 0 – 9, welche nicht mit einer Null beginnen darf.

Bspl.: 4711 567409

- Die oktale Darstellung (Basis 8) hingegen beginnt immer mit einer Null. Mögliche Ziffern sind 0 bis 7.

Bspl.: 0362 06357

- Die hexadezimale Form (Basis 16) beginnt mit `0X` oder `0x`. Für die 16 benötigten Ziffern werden die Ziffern 0 – 9 und die Zeichen `A – F`, `a – f` herangezogen.

Bspl.: 0X5A 0x5A 0x5a

Diese Formen bestimmen zwar den **Wert** des ganzzahligen Literals, sagen zunächst aber noch nichts über dessen Typ aus. Dieser wird ebenfalls noch beeinflusst durch optionale Suffixe. (Suffixe in beliebiger Reihenfolge, Groß/Kleinschreibung ohne Bedeutung.) Genauer regelt diese Auflistung. Der **Typ** ist abhängig von Form, Suffix und Wert des Literals. Es ist nachfolgend jeweils der erste Typ, welcher den Wert aufnehmen kann.

dezimal, kein Suffix	int long unsigned long
oktal oder hexadezimal, kein Suffix	int unsigned int long unsigned long
Suffix <code>u</code>	unsigned int unsigned long
Suffix <code>l</code>	long unsigned long
Suffixe <code>u</code> und <code>l</code>	unsigned long

Zur Beachtung: Es gibt keine `short`-Literele und keine negativen Literale. Derartige Werte können aber durch Berechnung eines Ausdrucks entstehen. Im zweitfolgenden Beispiel geschieht dies durch eine sogenannte Typkonvertierung bzw. durch Anwendung der unären Minus-Operation.

```
Bspl.: 47      47u      47L      47ul      0x2Fu1
        |        |        |        |        |
        int     unsigned int long    unsigned long unsigned long
```

```
Bspl.: /* Vereinbarung von Variablen */
short s;
int i;
long l;

/* Anweisungen mit Wertzuweisung */
s = 47;           Typkonvertierung, weil links: short und rechts: int
i = 47;           Typen identisch
l = 47L;          Typen identisch
i = -47;          rechte Seite: int-Ausdruck
```

(Mehr zu Variablen, Zuweisungen, Konvertierungen und Ausdrücken folgt.)

## char-Datentypen

Die **ganzzahligen (!)** Datentypen `char`, `signed char` und `unsigned char` dienen zur Bearbeitung von Zeichen. Sie haben einen Wertebereich, der die Abspeicherung der Ordinalwerte gemäß Codetabelle (im Regelfall ein 256-Zeichen-Code) erlaubt.

<code>signed char</code>	Wertebereich -128 bis 127
<code>unsigned char</code>	Wertebereich 0 bis 255
<code>char</code>	implementationsabhängig entweder wie <code>signed char</code> oder wie <code>unsigned char</code>

Eine Codetabelle nimmt die eindeutige Zuordnung eines Zeichens zu seinem Ordinalwert vor. Wird dieser ganzzahlige Wert binär dargestellt, so erhält man die Speicherdarstellung (Bits) des zugeordneten Zeichens. Von Codetabellen wird im Allgemeinen vorausgesetzt, dass die Ziffern, die Großbuchstaben und die Kleinbuchstaben jeweils aufeinanderfolgend dicht abgelegt sind. Ein Beispiel einer Codetabelle befindet sich im Anhang G.

Da `char`-Werte in C ganze Zahlen repräsentieren, dürfen sie benutzt werden, wo immer ein ganzzahliger Wert erwartet wird. Umgekehrt werden wir in Kapitel 13 und im Anhang A erleben, dass auch der Datentyp `int` zur Verarbeitung von Zeichen benutzt wird.

Die gebräuchliche Aussage, dass `char`-Werte Zeichen sind, ist somit nicht wirklich korrekt, wird aber dennoch geduldet, da dies den Zweck dieser Datentypen beschreibt. Der Umstand, dass es sich aus Sicht von C eigentlich um Zahlen handelt, wird erst in Ausdrücken sichtbar. So kann es den Anfänger sicher etwas irritieren, wenn auf Objekten, die man als Zeichen ansieht, plötzlich arithmetische Rechenoperationen ausgeführt werden.

## Literale (Konstanten) des char-Typs

Für die Darstellung von `char`-Werten gibt es spezielle Literale.

- Ein Zeichen eingefügt in (einfache) Apostrophe:  
'A' ist der **ganzzahlige** Ordinalwert des Zeichens A.
- Ferner gibt es (innerhalb der Apostrophe) folgende Ersatzdarstellungen:

<i>new line</i> :	<code>\n</code>	<i>single quote</i> :	<code>\'</code>	<i>alert</i> :	<code>\a</code>
<i>horizontal tab</i> :	<code>\t</code>	<i>double quote</i> :	<code>\"</code>	<i>null char</i> :	<code>\0</code>

und weitere ...

```
Bspl.: /* Vereinbarung von Variablen */
char cz;
int i;
int diff;

/* Wertzuweisungen */
cz = 'z';           Typen identisch
i = 'Z';           Typkonvertierung, weil links: int und rechts: char
diff = cz - 'a';    rechte Seite: Ausdruck mit ganzen Zahlen
```

Da das lateinische Alphabet (26 Zeichen) in den ASCII-Zeichencode eingearbeitet ist, ergibt sich hier als Differenz zwischen 'z' und 'a' der Wert 25 für die Variable `diff`.

(Mehr zu Variablen, Zuweisungen, Konvertierungen und Ausdrücken folgt.)

## Wahrheitswerte

Als strukturierte Programmiersprache ermöglicht es C, Anweisungen in Abhängigkeit von Bedingungen auszuführen. **Bedingungen** sind Rechenausdrücke, die einen Wahrheitswert „falsch“ oder „wahr“ liefern.

Zur Behandlung von **Wahrheitswerten** gibt es in C dennoch keinen originären Standard-Datentyp. Als Wahrheitswerte dienen stattdessen Zahlen (bevorzugt: Datentyp `int`). Der Wert Null bedeutet „falsch“; jeder Wert ungleich Null bedeutet „wahr“.

Mit C99 wurde dieser Umstand etwas ausgebessert: Über die Header-Datei `<stdbool.h>` werden die Präprozessor-Makros (vgl. Anhang D) `bool`, `false` und `true` vereinbart. `bool` steht für den Datentyp `_Bool`, `false` für den Wert 0 und `true` für 1.

<b>Bspl.: C klassisch</b>	<b>C99: #include &lt;stdbool.h&gt;</b>
<pre>/* Vereinbarung */ int fertig;</pre>	<pre>// Vereinbarung bool fertig;</pre>
<pre>/* Wertzuweisung */ fertig = 0;</pre>	<pre>// Wertzuweisung fertig = false;</pre>

Wir werden in unseren Beispielen auf dem klassischen Weg bleiben, auch wenn es etwas verwirren mag, wenn Objekte, die man als Zahlen ansieht, als Bedingungen genutzt werden können.



## Gleitpunkt-Typen

Für die Darstellung von Zahlen mit Dezimalteil und einem 10er-Exponenten benutzt man die Datentypen `float`, `double` und `long double`.

Bezüglich des Speicherbedarfs, des Wertebereichs und der Genauigkeit gilt:

$$\text{float} \leq \text{double} \leq \text{long double}$$

Die Header-Datei `<float.h>` enthält die konkreten, implementationsabhängigen Grenzwerte der Gleitpunkt-Typen. Hier wieder exemplarisch und auszugsweise die Daten aus einem ungenannten Entwicklungssystem. Weitere Details findet man im Anhang C.

```
#define DBL_DIG      15          /* # of decimal digits of precision */
#define DBL_MAX      1.7976931348623158e+308 /* max value */
#define DBL_MAX_10_EXP 308      /* max decimal exponent */
#define DBL_MIN      2.2250738585072014e-308 /* min positive value */
#define DBL_MIN_10_EXP (-307)    /* min decimal exponent */

#define FLT_DIG      6          /* # of decimal digits of precision */
#define FLT_MAX      3.402823466e+38F    /* max value */
#define FLT_MAX_10_EXP 38      /* max decimal exponent */
#define FLT_MIN      1.175494351e-38F    /* min positive value */
#define FLT_MIN_10_EXP (-37)    /* min decimal exponent */
```

Man beachte die geringe Genauigkeit des Typs `float` (hier ein 32bit-System); daher ist der Gebrauch des Typs `double` für Gleitpunkt-Arithmetik meistens erforderlich.

### Literale (Konstanten) der Gleitpunkt-Typen

Gleitpunkt-Zahlen beinhalten einen Dezimalpunkt oder einen Exponentialteil oder beides.

Bspl.: `47.11` `47.0` `0.11` `47.` `.11` mit Dezimalpunkt  
`47.11E3` `47.11E-5` `47.11e-5` mit Dezimalpunkt und Exponentialteil<sup>1</sup>  
`4E2` `4.E3` `.4e3` richtig, aber unschön

All diese Literale sind vom Typ `double`, da sie keinen Suffix haben.

Literale mit Suffix `F` oder `f` sind vom Typ `float`, mit Suffix `L` oder `l` vom Typ `long double`.

```
Bspl.: /* Vereinbarung von Variablen */
double d;
float f;

/* Wertzuweisungen */
d = 25;           Typkonvertierung, weil links: double und rechts: int
d = 25.0;        Typen identisch
f = 25.0F;       Typen identisch
```

(Mehr zu Variablen, Zuweisungen und Konvertierungen folgt.)

<sup>1</sup> Der Wert z.B. von `47.11E3` ist  $47.11 \times 10^3 = 47110$ .

## Lokale Objekte

Objekte können wir zu Beginn innerhalb von Funktionen vereinbaren. Wir unterscheiden dabei zwischen veränderlichen Objekten und nichtveränderlichen Objekten. Erste bezeichnen wir als Variablen. Letzte werden auch gerne Konstanten genannt; dies sollte aber nicht zu Verwechslungen mit den konstanten Literalen führen.

**Variablen** werden vereinbart mit einem Datentyp und einer Liste von Namen. Enthält die Liste mehr als einen Namen, werden diese untereinander durch Kommas getrennt. Der Abschluss einer solchen Definition erfolgt durch ein Semikolon.

```
Bspl.: int i, j, k, alpha, beta;
       unsigned int gamma;
       double x, xNeu;
```

Derartig vereinbarte Variablen erhalten damit noch keinen wohldefinierten Wert; die Werte ergeben sich zufällig aus alten Speicherinhalten. Man kann diese Variablen jedoch gleich bei der Vereinbarung explizit mit Startwerten initialisieren. Die zugewiesenen Werte werden dabei häufig konstant<sup>2</sup> sein.

```
Bspl.: int i = 1, j = 2, k = 3, alpha = 0, beta = 0;
       unsigned int gamma = 0u;
       double x = 0.0, xNeu = 1.0;
```

Alle Werte sind typidentisch zu den jeweiligen Variablen. Es wird keine Konvertierung fällig.

Mit dem zusätzlichen Attribut `const` vereinbart man Objekte, deren Werte sich zur Laufzeit des Programms nicht mehr verändern lassen, also (symbolische) **Konstanten**. Diese müssen explizit initialisiert werden.

```
Bspl.: const int diesesJahr    = 2016,
       naechstesJahr = diesesJahr + 1;
```

### *Sind symbolische Konstanten wirklich konstant?*

Unter zeitlichen Aspekten gibt es zwei Arten von Konstanten. Auf der einen Seite haben wir syntaktische Einheiten, die über alle Programmläufe hinweg einen identischen Wert repräsentieren. Zu dieser Gruppe zählen wir die Literale, also explizite Wertdarstellungen. Dazu gehören die Präprozessor-Makros (Anhang D), wenn sie so vereinbart werden wie hier in den Header-Dateien `<limits.h>` und `<float.h>`. Weitere „echte“ Konstanten tauchen noch im Kapitel 5 auf. Nur solche Konstanten können dann auch konstante Ausdrücke bilden (Details dazu im Kapitel 4).

Symbolische Konstanten sind **finale** Objekte: Damit wird der Compiler angewiesen, nach der Anfangswertbelegung keine Veränderung am Objekt mehr zuzulassen. Dies schließt nicht ein, dass bereits der Startwert irgendwie konstant sein muss. Ein so vereinbartes lokales Objekt kann daher in verschiedenen Programmläufen, aber auch bei der Wiederholung von Blöcken (Kapitel 6, 9) jeweils einen unterschiedlichen Wert annehmen. Konsequenterweise dürfen symbolische Konstanten nicht in konstanten Ausdrücken vorkommen, deren Berechnung einmalig zur Übersetzungszeit erfolgt.

---

<sup>2</sup> Wir verwenden dazu vorläufig und überwiegend Literale; allgemein sind jedoch auch Ausdrücke möglich.

## 3. Konvertierungen für Standard-Datentypen

---

Unter **Konvertierung** verstehen wir die Umwandlung des Wertes eines Objekts in einen anderen Datentyp. Viele Situationen bewirken derartige Konvertierungen, so können z.B. in Ausdrücken (siehe Kapitel 4) ganze und Gleitpunkt-Zahlen gemischt werden.

Dieses Kapitel fasst die Standard-Umwandlungen, die auf den bisher bekannten Datentypen wirken, zusammen und benennt die zu erwartenden Ergebnisse. Diese Aussagen sind im Laufe der Zeit zu ergänzen. Dabei sind zwei Fragen zu unterscheiden: **Wie** wird gewandelt und **wann** wird gewandelt?

### Wie wird gewandelt?

#### *Integer Promotion (C90: Integral Promotion)*

Hierunter versteht man die Umwandlung weniger präziser ganzzahliger Datentypen, z.B. von `char` oder `short`. Wenn ein `int` alle Werte des ursprünglichen Typs repräsentieren kann, wird der Wert in einen `int` konvertiert; andernfalls wird er nach `unsigned int` konvertiert. Die Anpassung ist somit stets exakt, d.h. werterhaltend.

#### Ganzzahlige Konvertierungen

Eine Konvertierung zu einem „umfassenden“ Datentyp erfolgt immer ohne Wertveränderung. Ist diese Voraussetzung nicht gegeben, hängt der Vorgang vom konkret vorliegenden Wert ab.

Wenn eine ganze Zahl in einen vorzeichenbehafteten Typ (`signed char`, `short`, `int`, `long`) konvertiert wird, bleibt der Wert unverändert, wenn er im neuen Typ dargestellt werden kann; andernfalls ist das Ergebnis systemabhängig.

Wenn eine ganze Zahl in einen `unsigned`-Typ konvertiert wird, ist die Situation nur unproblematisch, wenn der anzupassende (positive) Wert im neuen Typ dargestellt werden kann. Der andere Fall ist für uns nicht von Interesse.<sup>1</sup>

#### Gleitpunkt-Konvertierungen

Gleitpunkt-Werte können in einen beliebigen Gleitpunkt-Typ konvertiert werden. Dabei erfolgen die Konvertierungen

`float → double → long double`

immer exakt. Wenn in umgekehrter Richtung der Wert in den Bereich des weniger präzisen Typs fällt, so kann das Resultat der Umwandlung sowohl der nächsthöhere als auch der nächstniedrigere Wert sein. Wenn das Resultat aus dem Wertebereich herausfällt, ist das Verhalten nicht definiert.

---

<sup>1</sup> Ansonsten müssten wir uns jetzt über Modulo-2<sup>n</sup>-Arithmetik und Zweierkomplement-Repräsentation unterhalten.

## Konvertierungen zwischen ganzzahligen und Gleitpunkt-Typen

Bei der Umwandlung eines Gleitpunkt-Wertes in einen ganzzahligen Typ werden die Nachpunktstellen abgeschnitten<sup>2</sup>. Das Ergebnis der Umwandlung ist undefiniert, wenn der Wert nicht im ganzzahligen Typen darstellbar ist.

Umwandlungen ganzzahliger Werte in Gleitpunkt-Werte sind im arithmetischen Sinne in dem Umfang korrekt, wie die Darstellung dies erlaubt. Genauigkeitsverluste treten nur dann auf, wenn der ganzzahlige Wert nicht exakt als Gleitpunkt-Wert dargestellt werden kann.

## Wann wird gewandelt?

### Automatische Konvertierungen

Viele Operatoren (s. Kapitel 4) bewirken implizit eine Typkonvertierung. Der Effekt besteht darin, dass die Operanden einer Operation in einen gemeinsamen Typ umgewandelt werden. Der Regelfall der Konvertierung einer arithmetischen Operation funktioniert nach folgendem Prinzip<sup>3</sup>:

- So „niedrig“ wie der „höchste“ Typ vorgibt, aber nicht unterhalb von `int`.

Die Hierarchie „niedriger/höher“ der Typen bezieht sich dabei auf den Umfang des Wertebereichs; ein vorzeichenloser Typ wird höher bewertet als der entsprechende vorzeichenbehaftete Typ; die Gleitpunkt-Typen liegen höher als die ganzzahligen Typen. Die *Integer Promotion* bewirkt ferner, dass im Regelfall nicht mit den „kleinen“ Datentypen unterhalb von `int` gerechnet wird und derartige Objekte bei Operationen stets vorher hochkonvertiert werden.

**Warnung!** Beim Zusammentreffen von vorzeichenlosen und vorzeichenbehafteten Typen ganzer Zahlen zeigen die Konvertierungsregeln auch ungewöhnliche Ergebnisse. So wird bei der nachfolgenden Multiplikation `*` der negative `int`-Wert `-1` doch tatsächlich in den höherwertigen, aber vorzeichenlosen Typ `unsigned int` konvertiert. Diese Anpassung liefert daher „irgendwas Positives“ und die Multiplikation berechnet somit einen positiven Wert, welcher in diesem Beispiel letztendlich nach `double` gewandelt wird.

Bspl.: <code>int i = -1;</code> <code>unsigned int u = 1;</code> <code>double d = i * u;</code>	keine Konvertierung erforderlich implizite Anpassung <code>1 → 1u</code> Unerwartet? Die Variable <code>d</code> wird hierdurch nicht mit dem Wert <code>-1.0</code> initialisiert.
---	--

### Explizite Typkonvertierungen

Eine explizite („ausdrücklich gewollte“) Konvertierung kann mit der *cast*-Notation erfolgen.

**Form:** `( Typname ) Ausdruck`

Das Ergebnis des Ausdrucks wird dann in den angegebenen Typ konvertiert. Jede Standard-Konvertierung kann auf diese Weise auch explizit angestoßen werden. Die Sinnhaftigkeit solchen Tuns liegt voll in der Hand des Programmierers.

<sup>2</sup> Prozessorabhängig: Verschiedene Systeme können negative Zahlen in verschiedene Richtung kürzen.

<sup>3</sup> Mit einer Sonderregelung für die Verknüpfung von `long int` mit `unsigned int`.

## 4. Ausdrücke

### Übersicht

C erlaubt das Aufstellen von Ausdrücken analog zu den bekannten arithmetischen Regeln mit mehrfacher, verschachtelter Klammerung (runde Klammern). Vorläufig verwenden wir in Ausdrücken nur Objekte (Literele, Variablen, Konstanten) der Standard-Datentypen.

Im Ausdruck werden die Operationen unter Berücksichtigung der Klammerung gemäß der Priorität (Stufe) und der Assoziativität der Operatoren ausgeführt. Im Laufe der Berechnung wird jeweils eine Typkonvertierung für betroffene Operanden ausgeführt, sofern dies erforderlich ist. Einen ersten Überblick über die vorhandenen Operationen gibt diese Liste.

Stufe	Operator	Operation
15L	. ->	Auswahloperationen
15L	[ ]	Indizierung
15L	( )	Funktionsaufruf
14R	sizeof	Größe in Bytes
14RX	++ --	Inkrement, Dekrement
14R	~	bitweises Komplement
14R	!	logische Negation
14R	- +	unäres Minus und Plus
14R	& *	Adressoperation, Dereferenzierung
14R	( )	Typkonvertierung ( <i>cast</i> )
13L	* / %	Multiplikation, Division und Rest
12L	+ -	Addition und Subtraktion
11L	<< >>	Bitverschiebung
10L	< <= > >=	Vergleichsoperationen
9L	== !=	Gleichheit, Ungleichheit
8L	&	bitweises AND
7L	^	bitweises XOR
6L		bitweises OR
5LX	&&	logisches UND
4LX		logisches ODER
3RX	? :	bedingter Ausdruck
2R	= *= /= %= += -= <<= >>= &= ^=  =	Zuweisungsoperationen
1L	,	Sequenz

Bei verketteten Operationen gleicher Stufe:

L = Assoziativität von links nach rechts

R = Assoziativität von rechts nach links

X = Es gelten besondere Regelungen, die in den folgenden Abschnitten erläutert werden.

## Syntaktische Besonderheiten

Objekte „alleine“ – ohne jegliche Verwendung eines Operators – gelten bereits als Ausdrücke und dürfen folglich dort verwendet werden, wo ein Ausdruck erwartet wird. Dies ist lediglich eine Übereinkunft, um die Sprachregeln einfacher zu halten.

Ein Ausdruck mit abschließendem Semikolon wird zu einer (Ausdrucks-)Anweisung.

Bspl.: `double c, d = 7.7;` Vereinbarung der Variable `d` mit **Initialisierung** =  
Das Literal `7.7` alleine bildet den initialisierenden **Ausdruck**.

`c = d + 1.1;` Ausdruck mit enthaltener **Zuweisung** =  
Das Semikolon macht daraus eine **Anweisung**.

Anweisung

## Verlassen des Wertebereichs

Wird bei Ausführung einer Operation der Wertebereich des Ergebnistyps verlassen, so ist das Ergebnis systemabhängig. Mit einem vernünftigen Ergebnis aus algorithmischer Sicht ist dann nicht mehr zu rechnen. Die Einhaltung der Wertebereiche liegt in der Verantwortung des Programmierers.

Bspl.: Seien  $a$  und  $b$  positive ganze Zahlen. Sei  $g$  der größte gemeinsame Teiler von  $a$  und  $b$ . Dann berechnet sich das kleinste gemeinsame Vielfache  $v$  wie folgt:

$v = a * b / g;$  Vorsicht: Das Produkt kann entsprechend groß werden.  
 $v = a / g * b;$  besserer Ansatz

Bei Gleitpunkt-Werten können Ergebnisse nur im Rahmen der bestehenden Genauigkeit exakt sein.

## Ganzzahlige Arithmetik

Bei der arithmetischen Verknüpfung von ganzen Zahlen ist das Ergebnis wiederum eine ganze Zahl. Wir erwähnen dies insbesondere für den Divisionsoperator.

Bspl.:  $13/5$  liefert **kein** Gleitpunkt-Ergebnis!

Wir rechnen wie in der Grundschule:

$$\begin{array}{r} 13 : 5 = 2 \text{ Rest: } 3 \\ \underline{10} \\ 3 \end{array}$$

Dementsprechend liefert  $13/5$  mit  $2$  das ganzzahlige Divisionsergebnis und  $13\%5$  mit  $3$  den Divisionsrest<sup>1</sup>.

Die Operanden von  $\%$  müssen ganzzahlig sein. Für negative Zahlen sind die Details systemabhängig.

Bspl.: Zum Vergleich: Die Ausdrücke  $13.0/5.0$  und  $13/5.0$  und  $13.0/5$  liefern jeweils  $2.6$  als Gleitpunkt-Ergebnis, da die ganzen Zahlen  $13$  und  $5$  automatisch zum Typ `double` konvertiert werden.

<sup>1</sup> Der Operator  $\%$  wird häufig auch als Modulo-Operator bezeichnet. Dies ist nicht so ganz glücklich, da auch negative Reste systemabhängig entstehen können.

## Wertzuweisung

Die Wertzuweisung ist (im Gegensatz zu einigen anderen Programmiersprachen) eine Operation innerhalb eines Ausdrucks und keine explizite Anweisung. Allerdings wird ja jeder Ausdruck, also auch eine Wertzuweisung, durch ein angehängtes Semikolon zu einer Anweisung.

**Form:** *Variable* = *Ausdruck*                    ist ein Ausdruck  
           *Variable* = *Ausdruck* ;                ist eine Anweisung

Der Operator = hat diese beiden Wirkungen.

- Die *Variable* (vorläufig) auf der linken Seite erhält den Wert des rechten *Ausdrucks*. Dabei wird gegebenenfalls die rechte Seite entsprechend konvertiert.
- In dem Ausdruck, der die Zuweisung umgeben kann, entspricht diese typ- und wertmäßig dem linken Operanden (*Variable*), nachdem dieser den neuen Wert erhalten hat.

Bspl.: `int i, j, k;`

`i = 27;`

rechte Seite: Literal (Konstante) als Ausdruck

`j = -27;`

rechte Seite: Ausdruck mit unärem Minus

`k = i + 16;`

rechte Seite: Ausdruck mit zweistelligem Operator

`i = (j = 3) * (k = 5);`

entspricht: `j = 3; k = 5; i = j * k;`

`i = j = k = 3;`

entspricht: `k = 3; j = k; i = j;`

## Weitere Zuweisungsoperationen

Durch Kombination einer Rechenoperation mit nachfolgender Wertzuweisung entsteht eine Reihe weiterer Operationen (Operatoren siehe obige Tabelle, Stufe 2). Das nachfolgende Beispiel zeigt die Systematik, wenn man den Operator + durch einen der anderen zulässigen Operatoren ersetzt.

Bspl.: `i += 3` entspricht: `i = i + 3`

## Inkrement und Dekrement

Diese Operationen können nur auf Variablen angewandt werden. ++ und -- gibt es in Präfix- und Postfix-Notation, werden dem Operanden (hier: i) vorangestellt oder folgen diesem:

`++i`

`i++`

`--i`

`i--`

Der Operator hat zwei Wirkungen:

- In beiden Notationen wird die Variable um 1 erhöht/erniedrigt.
- In der Präfix-Notation ist das Operations-Ergebnis für den umgebenden Ausdruck der neuberechnete Wert. In der Postfix-Notation ist das Operations-Ergebnis der „alte“ Wert.

Tipp: Liest man den Ausdruck von links nach rechts, kann man auf die Wirkung schließen.

`++i` heißt: Erst `i` erhöhen, danach einsetzen.

`i++` heißt: Erst `i` einsetzen, danach erhöhen.

Bspl.: `int i = 6, j;`

`j = ++i - 5;`

entspricht: `i += 1; j = i - 5;`

`j = i++ - 5;`

entspricht: `j = i - 5; i += 1;`

Bspl.: `i = i + 1;`

„Erhöhe `i` um 1.“ – Diese 4 Anweisungen leisten das Gleiche.

`i += 1;`

`++i;`

`i++;`

**Warnung!** Nicht für jeden Ausdruck ist die Operationsreihenfolge eindeutig bestimmt. So ist beispielsweise für `a*b+c*d` nicht festgelegt, ob die linke oder die rechte Multiplikation zuerst ausgeführt wird. Aus arithmetischer Sicht ist dies auch unerheblich.

Dies kann jedoch zu Mehrdeutigkeiten führen, wenn an einer Stelle im Ausdruck der Wert einer Variablen verändert wird (Wertzuweisung, Inkrement/Dekrement) und an anderer Stelle auf die Variable zugegriffen wird.

Solche Konstrukte sollten daher prinzipiell vermieden oder zumindest gründlich durchdacht werden.

Bspl.: `int i = 1, x, y, z;`

`x = (i+1) * (i-1);`

o.k.

`y = ++i * --i;`

mehrdeutig je nach Operationsreihenfolge

`z = ++i * (i-1);`

Die Klammer erzeugt keinen Vorrang vor dem Inkrement!

## Vergleiche

Die Vergleichsoperatoren sind:

`<` kleiner

`<=` kleiner/gleich

`>` größer

`>=` größer/gleich

`==` gleich

`!=` ungleich

} Diese Vergleiche liegen eine Prioritätsstufe niedriger als die ersten 4 Operatoren.

Die Vergleichsoperationen sind durchführbar auf allen Standard-Datentypen und liefern einen Wahrheitswert im Sinne von Kapitel 2. Wenn die Relation „falsch“ ist, lautet das Ergebnis 0; ist die Relation „wahr“, ist das Ergebnis 1.

Bspl.: `int t, jahr = 2016;`

`t = (jahr%4 == 0);`

berechnet, ob das Jahr durch 4 teilbar ist<sup>2</sup>

<sup>2</sup> Die Klammern sind optional und dienen nur der besseren Lesbarkeit.



## Logische Operationen

Die wichtigsten Operatoren für Wahrheitswerte sind:

!	Negation/Verneinung
&&	Konjunktion „und“
	Disjunktion „oder“

Eine logisch „falsche“ Verknüpfung wird wiederum durch das Ergebnis 0 repräsentiert; ist die Verknüpfung „wahr“, ist das Ergebnis 1. Weitere Operatoren sind anwendbar, da Wahrheitswerte Zahlen sind; zu erwähnen sind folgende Operationen.

==	!=	Vergleiche „gleich“, „ungleich“
=		Wertzuweisung

Bspl.: Es soll in einem Ausdruck festgestellt werden, ob eine Zahl  $x$  zwischen 1 und 10 liegt.

<code>1 &lt;= x &lt;= 10</code>	Falsche Formel! Assoziativ gleichwertig zu: <code>(1 &lt;= x) &lt;= 10</code> Der linke Operand des rechten Vergleichs ist nicht $x$ , sondern das Ergebnis des linken Vergleichs!
<code>1 &lt;= x &amp;&amp; x &lt;= 10</code>	o.k.
<code>(1 &lt;= x) &amp;&amp; (x &lt;= 10)</code>	o.k.

Bspl.: `int schaltjahr, jahr = 2016;`  
`schaltjahr = (jahr%4==0 && jahr%100!=0) || jahr%400==0;` <sup>2</sup>

### Shortcut

Bei den beiden Operatoren `&&` und `||` gibt es eine Besonderheit. Zunächst wird der linke Operand ausgewertet. Lässt dieser bereits die Bestimmung des Ergebnisses zu, dann wird der rechte Operand **nicht mehr** ausgewertet.

Diese Eigenschaft ist zwar effektiv, aber auch nicht ganz unkritisch, denn hierdurch entstehen Ausdrücke, die nicht mehr vollständig bewertet werden müssen. Man darf sich also nicht darauf verlassen, dass eine wertverändernde Operation oder ein Funktionsaufruf innerhalb des rechten Teilausdrucks unabhängig von den konkret vorliegenden Werten in jedem Fall ausgeführt wird.

## Bedingter Ausdruck (?:-Operator)

Ein ungewöhnlicher Operator, denn er besitzt drei Operanden.

**Form:** *Bedingung* ? *Ausdruck<sub>1</sub>* : *Ausdruck<sub>2</sub>*

Die *Bedingung* ist wie immer ein Ausdruck, der einen Wahrheitswert liefert. *Ausdruck<sub>2</sub>* darf unmittelbar keine der Zuweisungsoperationen (s. Tabelle, Stufe 2) sein<sup>3</sup>.

---

<sup>3</sup> Dann muss man halt zusätzlich klammern.

Ist die *Bedingung* „wahr“, so wird *Ausdruck<sub>1</sub>* berechnet, andernfalls *Ausdruck<sub>2</sub>*. Es ist ausdrücklich auch hier ein *Shortcut* festgelegt, demzufolge nur der zutreffende innere Ausdruck berechnet wird! Dessen Wert ist dann das Ergebnis des bedingten Ausdrucks. Der Typ des Ergebnisses wird unabhängig von der *Bedingung* bestimmt durch Anwendung der Konvertierungsregeln auf die beiden inneren Ausdrücke.

Wir untersuchen den fiktiven Ausdruck:  $a ? b : c ? d : e$

Der  $?:$ -Operator ist rechts-assoziativ, indem dieser Ausdruck gleichwertig ist zu:  $a ? b : (c ? d : e)$ . Trotz Klammerung erfolgt die Abarbeitung aber von links, da zuerst die Bedingung  $a$  berechnet werden muss. Dies folgt aus der Zusage, dass nur **ein** innerer Ausdruck ausgewertet wird. Eventuell wird die Klammer also überhaupt nicht bestimmt.

Bspl.: 

```
int i, vz;
i = ... ;
vz = i>0 ? 1 : (i<0 ? -1 : 0);
```

Gilt  $i>0$ , so erhält  $vz$  den Wert 1 und die Klammer wird nicht bewertet. Andernfalls wird der geklammerte Ausdruck berechnet: Ist dabei  $i<0$ , so ist das Ergebnis -1, sonst 0. (Die Klammerung ist optional.)

## Sequenz (Komma-Operator)

Der Komma-Operator erlaubt es, mehrere Ausdrücke syntaktisch zu einem Ausdruck zusammenzufügen. Hierdurch wird es möglich, in einigen Situationen auch mehrere Ausdrücke „unterzubringen“, obwohl von der Syntax her eigentlich nur ein Ausdruck erlaubt ist.

**Form:**  $Ausdruck_1, Ausdruck_2, \dots, Ausdruck_n$

Die Bearbeitung erfolgt von links nach rechts; Wert und Typ des gesamten Ausdrucks entsprechen dem des rechten Ausdrucks. Der Komma-Operator hat von allen Operatoren die niedrigste Priorität.

Bspl.: 

```
for (i=1, j=-1; i<100; i++, j--) ...
```

Hier darf syntaktisch jeweils nur ein einziger Ausdruck stehen.  
(for-Anweisung im Kapitel 6)

Das Kommazeichen wird auch für andere Zwecke benutzt. Wir weisen ausdrücklich darauf hin, dass z.B. bei Funktionsaufrufen (siehe gleich) in den Klammern **kein** Komma-Operator angegeben ist.

## Weitere Operationen

Die **Auswahloperationen** gehören zum Thema Strukturen (Kapitel 14); die **Indizierung** gehört zu Arrays (Kapitel 11). Im Zusammenhang mit Zeigern (Kapitel 10) verwendet man die **Adressoperation** und die **Dereferenzierung**. Die **Größe in Bytes** bestimmen wir mittels `sizeof` für dynamische Datenstrukturen (Kapitel 15). Die **bitweisen Operationen** stehen im Mittelpunkt von Kapitel 17.

**Funktionsaufrufe** betrachten wir zum Ende dieses Kapitels mit den Schwerpunkten „mathematische Funktionen“ und „Ein-/Ausgabe“.

## Konstante Ausdrücke

Derartige Ausdrücke können bereits vom Compiler zur Übersetzungszeit einmalig berechnet und durch ihren Wert ersetzt werden. In der C-Syntax finden konstante Ausdrücke eine besondere Verwendung, zum Beispiel bei den Aufzähltypen (Kapitel 5), für `case`-Marken (Kapitel 6), bei der Initialisierung statischer Objekte (Kapitel 9) oder als Längenangaben für Arrays (Kapitel 11).

Von einigen Sonderregelungen abgesehen, ist zu beachten:

- Alle Operanden in einem konstanten Ausdruck müssen Konstanten<sup>4</sup> (z.B. Literale) sein.
- Folglich sind insbesondere Wertzuweisungen, Inkrement und Dekrement innerhalb eines konstanten Ausdrucks nicht anwendbar. Auch Sequenz und Funktionsaufruf sind verboten.

## Funktionsaufruf

Eine Funktion wird aufgerufen innerhalb eines Ausdrucks durch Ihren Namen gefolgt von einer Liste von Argumenten (Ausdrücken). Die Liste kann je nach Funktion eventuell auch leer sein. Das Klammerpaar muss vorhanden sein, auch wenn es keine Argumente gibt.

**Form:** *Funktionsname* ( *Argument<sub>1</sub>* , *Argument<sub>2</sub>* , ... , *Argument<sub>n</sub>* )  
bzw.  
*Funktionsname* ( )

Die Argumente werden berechnet und die Funktion damit ausgeführt. Liefert sie ein Ergebnis zurück, so wird dieses in den Ausdruck eingesetzt, von dem aus die Funktion aufgerufen wurde.

Bspl.: `double d;`

`d = 2 * acos (-1);`  
  
 Aufruf

mathematische Funktion `acos`  
(Es erfolgen automatische Konvertierungen.)

`printf ("Wert: %f\n", d);`  
  
 Aufruf

`printf`-Funktion zur Ausgabe, mehr dazu gleich

In diesem Beispiel wird das `double`-Ergebnis der Funktion `acos` in einem Ausdruck weiter verwertet (mit `2.0` multipliziert und zugewiesen), das Ergebnis der Funktion `printf` hingegen nicht.

Eigene Funktionen können wir noch nicht erstellen. Allerdings stellt uns C mit der Standard-Bibliothek eine Grundausstattung vorgefertigter Funktionen bereit. Auf den nächsten Seiten befinden sich eine Beschreibung der mathematischen Funktionen und ein vorläufiger Blick auf den Bereich der E/A.

<sup>4</sup> Symbolische Konstanten sind nicht zulässig. Dies erläutert die abschließende Bemerkung aus dem Kapitel 2.

## Mathematische Funktionen

Zur Nutzung dieser Funktionen ist die Header-Datei `<math.h>` zu importieren.

Die Tabelle enthält die „klassischen“ mathematischen Funktionen. Dieser Umfang wurde mit dem Sprachstandard C99 wesentlich erweitert. Die neuen Funktionen listen wir hier nicht auf und verweisen stattdessen auf die im Anhang C befindlichen Links.

<code>double</code>	<b><code>acos</code></b>	<code>(double);</code>	Arcus-Cosinus, $x \in [-1,1]$
<code>double</code>	<b><code>asin</code></b>	<code>(double);</code>	Arcus-Sinus, $x \in [-1,1]$
<code>double</code>	<b><code>atan</code></b>	<code>(double);</code>	Arcus-Tangens
<code>double</code>	<b><code>atan2</code></b>	<code>(double, double);</code>	Arcus-Tangens von $x/y$
<code>double</code>	<b><code>ceil</code></b>	<code>(double);</code>	Aufrunden $\lceil x \rceil$
<code>double</code>	<b><code>cos</code></b>	<code>(double);</code>	Cosinus
<code>double</code>	<b><code>cosh</code></b>	<code>(double);</code>	Cosinus-Hyperbolicus
<code>double</code>	<b><code>exp</code></b>	<code>(double);</code>	Exponentialfunktion $e^x$
<code>double</code>	<b><code>fabs</code></b>	<code>(double);</code>	Absolutbetrag $ x $
<code>double</code>	<b><code>floor</code></b>	<code>(double);</code>	Abrunden $\lfloor x \rfloor$
<code>double</code>	<b><code>fmod</code></b>	<code>(double, double);</code>	Gleitpunkt-Rest von $x/y$
<code>double</code>	<b><code>frexp</code></b>	<code>(double, int*);</code>	Zerlegung in Normalform
<code>double</code>	<b><code>ldexp</code></b>	<code>(double, int);</code>	$x \cdot 2^n$
<code>double</code>	<b><code>log</code></b>	<code>(double);</code>	natürlicher Logarithmus $\ln(x)$ , $x > 0$
<code>double</code>	<b><code>log10</code></b>	<code>(double);</code>	Logarithmus zur Basis 10 $\log_{10}(x)$ , $x > 0$
<code>double</code>	<b><code>modf</code></b>	<code>(double, double*);</code>	Zerlegung in ganzzahligen Teil und Rest
<code>double</code>	<b><code>pow</code></b>	<code>(double, double);</code>	Potenzierung <sup>5</sup> $x^y$
<code>double</code>	<b><code>sin</code></b>	<code>(double);</code>	Sinus
<code>double</code>	<b><code>sinh</code></b>	<code>(double);</code>	Sinus-Hyperbolicus
<code>double</code>	<b><code>sqrt</code></b>	<code>(double);</code>	Quadratwurzel, $x \geq 0$
<code>double</code>	<b><code>tan</code></b>	<code>(double);</code>	Tangens
<code>double</code>	<b><code>tanh</code></b>	<code>(double);</code>	Tangens-Hyperbolicus

Das Ergebnis der Funktionsaufrufe ist stets vom Typ `double`.

In den runden Klammern stehen die Typen für die erforderlichen Argumente. Typkonvertierung findet gegebenenfalls statt. (`int*` bzw. `double*` liefern Ergebnisse zurück. Dazu erfährt man später im Kapitel 10 mehr.)

Winkel werden bei den trigonometrischen Funktionen im Bogenmaß (*radian*) gemessen.

<sup>5</sup> Fehler, falls  $x = 0$  und  $y \leq 0$ , oder bei  $x < 0$  und  $y$  nicht ganzzahlig.

## Vorläufige Hinweise zur Ein- und Ausgabe

Zur Nutzung der folgenden Verfahren ist die Header-Datei `<stdio.h>` zu importieren. Bei der Ausgabe von Werten wird die `printf`-Funktion, zur Eingabe die `scanf`-Funktion herangezogen. Die Ausgabe bezieht sich damit vorläufig auf den Monitor, die Eingabe auf die Tastatur.

Eine weitergehende Darstellung der formatierten E/A befindet sich im Anhang A.

### Ausgabe

Die **printf-Funktion** erlaubt eine variable Anzahl von Argumenten; dabei steht immer zuerst die Zeichenkette (String), welche ausgegeben werden soll.

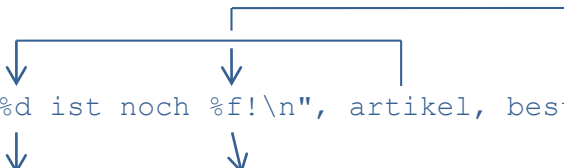
Innerhalb dieses Strings markieren dann Platzhalter die Positionen für die weiteren Argumente, sofern vorhanden. Die Platzhalter sind **Formatelemente**, von denen die wichtigsten in der nachfolgenden Tabelle aufgelistet sind.

Bei der Ausgabe werden dann die Formatelemente durch die jeweiligen Werte der Argumente ersetzt.

Bspl.: 

```
int    artikel = 231;
double bestand = 45.3;

printf ("Der Bestand von %d ist noch %f!\n", artikel, bestand);
```



Ausgabe: Der Bestand von 231 ist noch 45.300000!

### Eingabe

Eine Eingabe kann analog durch die **scanf-Funktion** gelesen werden. Das erste Argument ist wiederum ein Format-String, der jetzt die Eingabe steuert.

- Formatelemente stehen wiederum als Platzhalter für die weiteren Argumente.
- Andere Zeichen müssen dem nächsten Zeichen in der Eingabe entsprechen.
- Durch ein Leerzeichen kann auch Leerraum (siehe Kapitel 1) spezifiziert werden. Leerraum wird damit in der Eingabe explizit erlaubt, aber nicht verwertet.

Eine besondere Form haben vorläufig die weiteren Argumente: Da diese mit den eingelesenen Werten belegt werden sollen, müssen dort Variablen stehen und diesen ist jeweils ein `&`-Zeichen voranzustellen<sup>6</sup>.

Bspl.: 

```
scanf ("%d%lf", &artikel, &bestand);
```

Ein ganzzahliger Wert für den `artikel` und ein Gleitpunkt-Wert für den `bestand` sollen eingelesen werden. Die Werte sind in diesem Fall (keine Angabe zur Feldbreite) bei der Eingabe durch Leerraum voneinander zu trennen.

Gleichwertig wäre auch der Format-String `"%d %lf"` mit expliziter Angabe von Leerraum.

<sup>6</sup> Hintergrund: Die Parameter der Funktion sind sog. Zeiger, die beim Funktionsaufruf mit den Adressen der Argument-Variablen belegt werden müssen. Im Kapitel 10 dazu mehr.

Bspl.: `scanf ("%d,%lf", &artikel, &bestand);`

Jetzt sind die Eingabe-Werte durch ein Komma zu trennen. Da jedem Zahlwert bei der Eingabe Leerraum vorangestellt werden darf, kann man in diesem Beispiel zusätzlichen Leerraum bei der Eingabe nach dem Komma eingeben.

Optional Leerraum auch vor dem Komma wäre mittels `"%d ,%lf"` möglich.

Bspl.: `unsigned int tag, monat, jahr;`  
`scanf ("%2u.%2u.%4u", &tag, &monat, &jahr);`

Einlesen eines Datums: Die Formatelemente enthalten hier zusätzlich eine Angabe zur maximalen Breite (2 bzw. 4 Stellen). Es muss ferner je ein Punkt direkt hinter dem Tag und dem Monat mit eingegeben werden. Eine korrekte Eingabe wäre z.B.: 8.11.2016

Bspl.: `char zeichen;`  
`scanf ("%c", &zeichen);`                      `scanf (" %c", &zeichen);`  
 Direkt das nächste Zeichen wird eingelesen.      Hier wird jedoch Leerraum zuvor ignoriert.

#### Formatelemente (auszugsweise)

FE	Datentyp	Darstellung/Anmerkung
<code>%d</code>	<code>int</code>	dezimal ggf. mit Vorzeichen
<code>%u</code>	<code>unsigned int</code>	dezimal ohne Vorzeichen
<code>%hd</code>	<code>short</code>	dezimal ggf. mit Vorzeichen
<code>%hu</code>	<code>unsigned short</code>	dezimal ohne Vorzeichen
<code>%ld</code>	<code>long</code>	dezimal ggf. mit Vorzeichen
<code>%lu</code>	<code>unsigned long</code>	dezimal ohne Vorzeichen
<code>%f</code>	Ausgabe: <code>double</code> Eingabe: <code>float</code>	dezimal in sog. Fixpunkt-Notation
<code>%lf</code>	Eingabe: <code>double</code>	dezimal in sog. Fixpunkt-Notation
<code>%c</code>	Ausgabe: <code>unsigned char</code> Eingabe: <code>char</code>	ein Zeichen (auch Steuerzeichen und Leerraum)
<code>%%</code>		Ersatzdarstellung für das Zeichen <code>%</code>

Die Verwendung ungeeigneter Formatelemente kann zu Systemfehlern, Nicht-Ausführung, falschen Ergebnissen oder Genauigkeitsverlust führen.

Sonderfälle: Aufgrund der *Integer Promotion* (siehe Kapitel 3) können die `short`-Datentypen auch mit den Formatelementen `%d` bzw. `%u` ausgegeben werden. Gleiches gilt für die `char`-Typen; es erfolgt dann die Ausgabe des Ordinalwertes. – Nicht näher definiert ist in den Normen die Situation bei der Ausgabe von `float`. Die Praxis, wonach `%f` funktioniert, kann daher nicht verallgemeinert werden. – Nicht aufgelistet wurde ferner `%Lf` für `long double`, da hierbei Probleme bei der Eingabe im verwendeten Referenzsystem festgestellt wurden.

## 5. Aufzählungen (enum)

### Vereinbarung von Aufzähltypen

Ein neuer ganzzahliger Datentyp lässt sich dadurch vereinbaren, dass man seine Werte auflistet.

**Form:** `enum Etikett { Liste_von_Namen } ;`

Das *Etikett* (engl. *tag*) muss syntaktisch ein Name sein. Die Kombination aus dem reservierten Wort `enum` und dem *Etikett* wirkt ab jetzt wie ein neuer Typname.

Die Namen in der Liste repräsentieren die ganzzahligen `int`-Werte des Typs (wie Literale), sind dabei insbesondere konstant. Diese Namen müssen eindeutig sein. Die zugeordneten Werte beginnen mit Null und erhöhen sich mit der Schrittweite 1.

Ein Name in der Liste kann auch in der Form `Name = konstanter_ganzzahliger_Ausdruck` vereinbart werden und bezeichnet dann den Wert des Ausdrucks; nachfolgende Bezeichner setzen die Progression von diesem Wert aus fort.

**Bspl.:** `enum Tag {MO, DI, MI, DO, FR, SA, SO};`  
`enum Farbe {KLAR=-1, WEISS, ROT, GELB, BLAU, SCHWARZ=10, GRAU};`

-1	0	1	2	3	10	11

Die entstehenden Aufzähltypen heißen: `enum Tag` und `enum Farbe`

Diese Vereinbarungen nehmen wir vorläufig nur zu Beginn einer Funktion vor; sie gelten dann jeweils von der Stelle der Vereinbarung bis zum Ende der Funktion. (Abweichende Regeln werden später noch besprochen.)

### Variablen und Ausdrücke

Von einem Aufzähltyp können nun in der bekannten Form lokale **Variablen** vereinbart werden.

**Bspl.:** `enum Tag heute = DO;` mit Initialisierung, Wert: 3  
`enum Tag morgen;` ohne Initialisierung, Wert: undefiniert

Die Idee, mittels `const` zusätzliche symbolische Konstanten zu vereinbaren, macht hingegen keinerlei Sinn, legt doch bereits der Aufzähltyp Namen für konstante Werte fest.

Wie mit allen ganzzahligen Objekten auch, können **Ausdrücke** gebildet werden. Bei der Wertzuweisung an eine Variable von einem Aufzähltyp wird die Zulässigkeit der Zuweisung bezogen auf den Wertebereich nicht zwingend von System überprüft.

**Bspl.:** `morgen = heute + 1;`  
`heute = 37;` nicht zulässig; Prüfung möglich, aber optional

## Kombinierte Form

Die Vereinbarung eines Aufzähltyps und seiner Variablen lässt sich auch kombinieren. Dabei steht das *Etikett* weiterhin vor der Klammerung, die Namen der Variablen hingegen am Ende der Vereinbarung. Diese Form ermöglicht es ferner, auf die Vergabe des *Etiketts* auch gänzlich zu verzichten und dennoch Variablen vom Aufzähltyp zu vereinbaren. Bei Nutzung aller Möglichkeiten kann die kombinierte Form recht unleserlich wirken.

**Form<sup>1</sup>:** `enum Etikettopt { erste_Liste_von_Namen } zweite_Liste_von_Namen ;`

Die *erste\_Liste\_von\_Namen* vereinbart die konstanten Werte des Aufzähltyps wie bisher; die *zweite\_Liste\_von\_Namen* bezeichnet die vereinbarten Variablen. Auch hier sind Initialisierungen statthaft.

Fehlt das optionale *Etikett*, so kann man aus formalen Gründen keine weiteren typgleichen Variablen nachträglich vereinbaren.

```
Bspl.: /* Vereinbarung ohne Etikett */
        enum {OBEN, RECHTS, UNTEN, LINKS} blick;

        /* Anweisung */
        blick = RECHTS;
```

Die Variable `blick` kann mit einem der vier Werte des namenlosen Aufzähltyps belegt werden.

## Ein- und Ausgabe

Bei der Ausgabe eines Objekts mittels der `printf`-Funktion gibt es kein Formatelement, um den Namen aus dem Aufzähltyp anzuzeigen. `%d` gibt den zugehörigen ganzzahligen Wert aus. Die Situation bei der Eingabe und der `scanf`-Funktion ist analog.

```
Bspl.: printf ("Blickrichtung ist %d.\n", blick);
```

```
Ausgabe:  Blickrichtung ist 1.
```

Leider wird nicht der Name `RECHTS` ausgegeben.

---

<sup>1</sup> Der Index *opt* bringt zum Ausdruck, dass dieser Bestandteil auch wegfallen darf.



## 6. Anweisungen, Kontrollstrukturen

Wir haben bisher überhaupt nur zwei Anweisungen genutzt: die Ausdrucksanweisung und die `return`-Anweisung. Die nachfolgende Liste gibt eine vollständige Übersicht über alle Anweisungen in C. Unter den Kontrollstrukturen finden wir dabei die für eine strukturierte Programmierung erforderlichen Auswahlanweisungen und die Wiederholungen.

- Ausdrucksanweisung
- Blöcke (Blockanweisung)
- Kontrollstrukturen
  - Auswahlanweisungen
    - `if`-Anweisung (Fallunterscheidung)
    - `switch`-Anweisung (Fallauswahl)
  - Wiederholungen (Schleifen)
    - `while`-Anweisung (abweisende Schleife)
    - `do`-Anweisung (nichtabweisende Schleife)
    - `for`-Anweisung
  - Sprunganweisungen
    - `break`-Anweisung
    - `continue`-Anweisung
    - `return`-Anweisung
    - `goto`-Anweisung

### Blöcke

Ein Block fasst Vereinbarungen und Anweisungen zusammen.

**Form:** { *Folge\_von\_Vereinbarungen*<sub>opt</sub> *Folge\_von\_Anweisungen*<sub>opt</sub> }

In der optionalen *Folge\_von\_Vereinbarungen* befinden sich die Vereinbarungen lokaler Objekte. Eine derartige Vereinbarung gilt von der Stelle ihres Vorkommens bis zum Blockende. Sie wird mit Verlassen eines Blocks ungültig. Im Regelfall gibt es die vereinbarten Objekte dann auch nicht mehr.

Die optionale *Folge\_von\_Anweisungen* umfasst Anweisungen, welche der Reihe nach ausgeführt werden.

- Auch das Klammerpaar einer Funktionsdefinition (Funktion `main`) begrenzt einen Block. Wenn wir bisher verlangt haben, dass eine Vereinbarung dort „zu Beginn“ zu erfolgen habe, war damit die *Folge\_von\_Vereinbarungen* gemeint. Eine oder mehrere Ausdrucksanweisungen zusammen mit einer `return`-Anweisung bildeten bis jetzt die *Folge\_von\_Anweisungen*.

Der Block selbst ist wiederum eine Anweisung; d.h. ein Block darf syntaktisch überall dort stehen, wo eine Anweisung erwartet wird. Blöcke können somit auch ineinander eingesetzt werden und bilden dann eine geschachtelte Struktur. Liegt ein Block innerhalb eines anderen Blocks, so spricht man von einem inneren bzw. einem äußeren Block. Welche Auswirkung dies auf die enthaltenen Vereinbarungen hat, wollen wir in diesem Kapitel noch nicht untersuchen.

## Blöcke ab dem Standard C99

Einige aus C++ bekannte Erweiterungen sind zurück in die Sprache C geflossen. – Vereinbarungen müssen jetzt nicht mehr zwingend zu Beginn eines Blockes stehen, d.h. Vereinbarungen und Anweisungen können auch eine vermischte Folge bilden. Damit wird auch ein Programmierstil unterstützt, bei dem die Vereinbarungen möglichst „ortsnah“ zur Nutzung getroffen werden.

## Auswahanweisungen

### if-Anweisung

Die `if`-Anweisung dient zur Formulierung von Alternativen (Fallunterscheidungen).

**Form:** `if ( Bedingung ) Anweisung1`  
 oder  
`if ( Bedingung ) Anweisung1 else Anweisung2`

Die *Bedingung* muss einen Wahrheitswert liefern; die *Anweisungen* können beliebige einzelne Anweisungen sein. Soll mehr als eine Anweisung „untergebracht“ werden, so nutzt man Blöcke.

### Wirkung

Ist die *Bedingung* „wahr“ (ungleich Null), dann wird die (darauffolgende) *Anweisung<sub>1</sub>* ausgeführt.

Ist die *Bedingung* nicht erfüllt, so wird, falls ein `else`-Teil vorhanden ist, dessen *Anweisung<sub>2</sub>* ausgeführt; ansonsten wird keine Anweisung ausgeführt. Ein `else`-Teil bezieht sich immer auf das unmittelbar vorausgehende `if`.

Bspl.: `if (a < 0)`  
       `a *= -1;`

eine Anweisung direkt

Bspl.: `if (b < 0)`  
       `s = -1;`  
       `else`  
       `s = 1;`

Bspl.: `if (c < 0)`  
       `{`  
           `printf ("c wird positiv!\n");`  
           `c *= -1;`  
       `}`

mehrere Anweisungen im Block:

1. Anweisung
2. Anweisung

### Geschachtelte if-Anweisungen

Häufig besteht eine Alternative aus mehr als zwei Fällen. Eine geschachtelte Folge von `if`-Anweisungen ist der allgemeinste Weg, derartige Konstruktionen zu formulieren. Dabei werden die Bedingungen der einzelnen Fälle der Reihe nach überprüft, wobei die dazu erforderlichen Anweisungen systematisch ineinander eingesetzt werden.

Wir demonstrieren dies am Beispiel des Signums (Vorzeichen); das Signum ist 1 für Werte größer als Null, -1 für Werte kleiner als Null und 0 für den Wert Null selbst.

```

Bspl.: if (d > 0)
        s = 1;
    else
        if (d < 0)
            s = -1;
        else
            s = 0;

```

Die graue `if`-Anweisung wurde als *Anweisung<sub>2</sub>* in eine äußere `if`-Anweisung eingesetzt.

Das Beispiel zeigt dabei insbesondere auch, wie sich durch den sinnvollen Einsatz von Zeilenwechseln und Einrückungen die Lesbarkeit des Programmtextes erhöht. Allerdings ergibt sich auf diese Weise bei größerer Fallzahl eine Kaskade, bei der die inneren `if`-Anweisungen immer weiter nach rechts eingerückt werden. Daher hat sich ein anderer Formatierungsstil eingebürgert: `else` und `if` werden in eine Zeile geschrieben und die Einrückung bleibt in allen Fällen gleich.

```

Bspl.: if (d > 0)
        s = 1;
    else if (d < 0)
        s = -1;
    else
        s = 0;

```

Dieses Beispiel ist syntaktisch völlig identisch zum Vorgänger.

Lediglich der Leerraum (vgl. Kapitel 1) wurde anders eingesetzt.

Jetzt kann man die drei Fälle des Beispiels deutlicher erkennen. Durch das Einfügen weiterer `else-if`-Konstrukte lässt sich die Fallzahl natürlich beliebig erhöhen. Damit ergibt sich das folgende Schema, wenn auch noch die einfachen Anweisungen durch Blöcke ersetzt werden.

```

if ( Bedingung1 )
{
    Vereinbarungen/Anweisungen1
}
else if ( Bedingung2 )
{
    Vereinbarungen/Anweisungen2
}
else if ( Bedingung3 )
{
    Vereinbarungen/Anweisungen3
}
...
else
{
    Vereinbarungen/Anweisungenelse
}

```

Zur Effektivität sei noch angemerkt, dass die Fälle nicht gleichgewichtet sind. Je tiefer ein Fall sinkt, je mehr Bedingungen müssen vor seinem Auffinden überprüft werden.

### switch-Anweisung

Sind alle Fälle einer Fallauswahl durch das Aufzählen von Werten eines ganzzahligen Typs charakterisierbar, lässt sich die Fallauswahl häufig realisieren durch die `switch`-Anweisung.

**Form:** `switch ( Ausdruck ) Anweisung`

Die *Anweisung* ist sinnvollerweise stets ein Block, in welchem bestimmte Anweisungen durch Marken eingeleitet werden. Vereinbarungen können nicht markiert werden.

**Form:** `case Wert :      oder      default :`

Damit ergibt sich folgendes Schema:

```
switch ( Ausdruck )
{
    case Wert1 :
        Anweisungen1
        break;
    case Wert2a :
    case Wert2b :
        Anweisungen2
        break;
    ...
    default :
        Anweisungendefault
}
```

Dabei gilt:

- Der *Ausdruck* muss vom ganzzahligen Typ sein. Speziell sind auch Aufzähltypen anwendbar.
- Die *Werte* sind konstante Ausdrücke; im Regelfall werden dies aber einfach nur Literale sein. Sie müssen ebenfalls ganzzahlig sein. *Werte* dürfen nicht doppelt auftreten; die `default`-Marke darf höchstens einmal auftreten.
- Sollen ferner lokale Objekte in Fällen des `switch`-Blockes vereinbart werden, so müssen damit die jeweiligen Folgen von *Anweisungen* gegebenenfalls noch zu inneren Blöcken erweitert werden.

### Wirkung

Der *Ausdruck* wird berechnet. Tritt sein Wert unter den Marken auf, so werden die der Marke folgenden *Anweisungen* ausgeführt. Das Auftreten einer weiteren, nachfolgenden Marke beendet die vorausgehende Anweisungsfolge **nicht**; dies bedeutet, dass so alle weiteren Anweisungen bis zum Ende des Blocks ausgeführt werden.

Abhilfe: Die im obigen Schema eingesetzten `break`-Anweisungen beenden dagegen die `switch`-Anweisung nach der Behandlung jedes Falls.

Ist der Wert des *Ausdrucks* von allen *Werten* verschieden, so werden die *Anweisungen* hinter der `default`-Marke ausgeführt, falls diese vorkommt; ansonsten werden keine Anweisungen ausgeführt.

## Wiederholungen (Schleifen)

Zur Wiederholung von Anweisungen stehen in C verschiedene Sprachmittel zur Verfügung.

Schleifen, die von einer expliziten Bedingung abhängig sind, können durch die `while`-, die `do`- bzw. die `for`-Anweisung realisiert werden. Welche davon jeweils benutzt wird, hängt von den Bestandteilen ab, welche für die zu realisierende Schleife zu formulieren sind.

### (a) Initialisierung

Festlegung von Werten oder Eigenschaften (sog. Invarianten) vor Ausführung der ersten Wiederholung

### (b) Bedingung

Kontrolle der Wiederholung: In C sind dies primär Wiederholungs-, keine Abbruchbedingungen, d.h. wenn die Bedingung „wahr“ ist, wird wiederholt.

### (c) Inkrementierung

Neubelegung von Objekten aus (b)

Fehlt diese, so bleibt die Bedingung stets gleich und man erhält Schleifen, welche entweder nie oder aber „endlos“ ausgeführt werden.

### (d) Anweisungen (eine oder mehrere)

Anweisungen, welche wiederholt werden sollen

Die Bestandteile (c) und (d) sind meist ineinander verwoben.

## while-Anweisung

Die `while`-Anweisung dient zur Realisierung einer abweisenden Schleife.

**Form:** `while ( Bedingung ) Anweisung`

Die *Bedingung* muss einen Wahrheitswert liefern. Die *Anweisung* kann jede beliebige Anweisung sein; in vielen Fällen wird dies ein Block mit mehreren Anweisungen sein.

## Wirkung

Die *Anweisung* wird solange wiederholt, wie die *Bedingung* „wahr“ (ungleich Null) ist. Der Test erfolgt **vor** jedem Wiederholungsdurchlauf (abweisende Schleife<sup>1</sup>).

Bspl.: Größter gemeinsamer Teiler

```
unsigned int i = ... , j = ... ;  
while (i != j)  
    if (i < j)  
        j -= i;  
    else  
        i -= j;  
printf ("Der ggT ist: %u\n", i);
```

Beide Werte müssen positiv sein.

---

<sup>1</sup> Ist die Bedingung von Anfang an „falsch“, wird die enthaltene Anweisung nie durchgeführt.

### do-Anweisung

Die `do`-Anweisung dient zur Realisierung einer nichtabweisenden Schleife.

**Form:** `do Anweisung while ( Bedingung ) ;`

Die *Bedingung* muss einen Wahrheitswert liefern. Die *Anweisung* kann jede beliebige Anweisung sein, in vielen Fällen ein Block.

### Wirkung

Die *Anweisung* wird solange wiederholt, wie der *Ausdruck* „wahr“ (ungleich Null) ist. Der Test erfolgt **nach** jedem Wiederholungsdurchlauf (nichtabweisende Schleife<sup>2</sup>).

### for-Anweisung

Die `for`-Anweisung dient zur Konstruktion allgemeiner Schleifen. Wir werden sie meist für Zählschleifen nutzen.

**Form:** `for ( Initialisierung ; Bedingung ; Inkrementierung ) Anweisung`

Dabei gilt:

- Die *Initialisierung*, die *Bedingung* und die *Inkrementierung* dürfen entfallen; fehlt die *Bedingung*, so wirkt dies wie „wahr“. Ansonsten müssen diese drei Bestandteile Ausdrücke sein. Ist es erforderlich, mehrere Objekte zu initialisieren oder zu inkrementieren, so wird gerne der Komma-Operator eingesetzt.
- Die *Anweisung* kann jede beliebige Anweisung sein, in vielen Fällen ein Block.

### Wirkung

Eine `for`-Schleife ist weitgehend<sup>3</sup> äquivalent zu folgender `while`-Schleife.

```
Initialisierung ;
while ( Bedingung )
{
    Anweisung
    Inkrementierung ;
}
```

Bspl.: Zählschleife „für *i* von 1 bis 10“

```
{
    int i;
    for (i = 1; i <= 10; i++)
        printf ("%d\n", i);
}
```

<sup>2</sup> Die enthaltene Anweisung wird somit immer mindestens einmal durchgeführt.

<sup>3</sup> Sofern keine `continue`-Anweisung vorkommt. Nur bei der `for`-Schleife selbst wird die Inkrementierung für den nächsten Durchlauf noch ausgeführt.

### for-Anweisung ab dem Standard C99

Neben der klassischen Möglichkeit darf die *Initialisierung* der `for`-Schleife auch die Vereinbarung initialisierter Variablen sein. Derartige Schleifenvariablen sind dann nur innerhalb der `for`-Anweisung gültig vereinbart.

Bspl.: Zählschleife „für `i` von 1 bis 10“

```
for (int i = 1; i <= 10; i++)  
    printf ("%d\n", i);
```

## Sprunganweisungen

### break-Anweisung

Die `break`-Anweisung dient auch zur Formulierung allgemeiner Formen von Wiederholungen in Verbindung mit den vorangehend beschriebenen „Basisschleifen“.

**Form:** `break ;`

Zum Beispiel nach folgendem Schema:

```
while (1)                                Die Bedingung ist immer „wahr“. Hier erfolgt kein Abbruch.  
{  
    ...  
    if ( Bedingung1 ) break;  
    ...  
    if ( Bedingungn ) break;  
    ...  
}
```

### Wirkung

Die `break`-Anweisung beendet die Ausführung der aktuellen Anweisung, und zwar der **innersten** `switch`-Anweisung bzw. der **innersten** Wiederholungsanweisung, welche die `break`-Anweisung umgibt. Äußere Anweisungen sind davon nicht betroffen. Die Bedingungen im obigen Schema sind jetzt Abbruchbedingungen.

### continue-Anweisung

Zur Steuerung des Ablaufs von Wiederholungen gibt es auch noch die `continue`-Anweisung.

**Form:** `continue ;`

### Wirkung

Diese Anweisung darf nur innerhalb von Wiederholungen vorkommen. Sie beendet den aktuellen Wiederholungsdurchlauf (der Anweisungen der innersten Schleife) und fährt mit der Bedingungskontrolle der nächsten Wiederholung fort, bei `for`-Schleifen mit der Inkrementierung und danach der Bedingungskontrolle.

### Marken und goto-Anweisung

Marken wollen wir nur innerhalb der `switch`-Anweisung nutzen. Die `goto`-Anweisung, welche eine markierte Anweisung „anspringt“ und die Programmbearbeitung an dieser Stelle fortsetzt, widerspricht dem Paradigma der strukturierten Programmierung. Wir werden sie daher nicht verwenden.



# 7. Funktionen

## Grundlagen

C ist eine prozedurale Programmiersprache, d.h. sie unterstützt den algorithmischen Ansatz, Programme aus kleineren Lösungen für Teilprobleme aufzubauen. Derartige Prozeduren werden in C als Funktionen bezeichnet. Funktionen sind ferner die Voraussetzung, um eine Wiederverwendbarkeit von Software zu organisieren.

Eine C-Funktion weist durchaus Gemeinsamkeiten mit dem Funktionsbegriff aus der Mathematik auf. So stellt auch eine C-Funktion eine Abbildung von Argumenten auf ein Ergebnis dar; Definitions- und Wertebereich werden eindeutig festgelegt; die Funktionsvorschrift wird durch einen Funktionsblock dargestellt.

Es gibt aber auch Unterschiede. Eine C-Funktion kann beliebige Anweisungen enthalten und damit über das Funktionsergebnis hinaus weitere „Seiteneffekte“ auslösen. Beispielsweise können auch globale Variablen (Kapitel 9) oder über Zeiger referenzierte Objekte (Kapitel 10) im Wert verändert, Objekte im dynamischen Speicher (Kapitel 15) erzeugt oder Ein-/Ausgabe betrieben werden. Damit endet die Ähnlichkeit mit den Funktionen der Mathematik.

Ferner sind Funktionsergebnis und Parameter/Argumente<sup>1</sup> bei der Programmierung optional. Das funktionale Ergebnis darf somit bei der Vereinbarung einer C-Funktion auch entfallen; in diesem Fall ist dann nur noch der „Seiteneffekt“ von Bedeutung. Ebenso sind unter Umständen auch keine Argumente erforderlich.

Bspl.: Für eine ganze Zahl  $x$  sei folgende Funktion (Mathematik) gegeben.

$$\text{expand}(x) := \begin{cases} x + 1 & \text{falls } x > 0 \\ x - 1 & \text{falls } x < 0 \\ 0 & \text{sonst} \end{cases}$$

Im Vorgriff auf die nachfolgenden Abschnitte setzen wir dies bereits jetzt schon einmal in eine entsprechende C-Funktion um.

```
int expand (int x)
{
    if (x > 0)
        return x + 1;
    if (x < 0)
        return x - 1;
    return 0;
}
```

Für die ganzen Zahlen (Ergebnis und Parameter) benutzen wir dabei den Datentyp `int`. Das Funktionsergebnis wird fallbezogen durch die enthaltenen `return`-Anweisungen festgelegt.

<sup>1</sup> Die genaue Bedeutung von Parametern und Argumenten wird im Kapitel 8 festgelegt.

## Vereinbarungen

### Funktionsdeklaration

Mit einer Deklaration (auch **Prototyp** genannt) werden der Name einer Funktion, der Ergebnistyp der Funktion und die Parameter vereinbart. Parameter behandeln wir erst im nächsten Kapitel.

**Form:** *Speicherklassen-Attribut*<sub>opt</sub> *Typ*<sup>2</sup> *Name* ( *Liste\_von\_Parametern*<sub>opt</sub> ) ;

Dabei gilt:

- Als (optionales) *Speicherklassen-Attribut* benutzen wir vorläufig nur `extern`<sup>3</sup>.
- Der Ergebnistyp `void` sagt aus, dass es kein Funktionsergebnis gibt.
- `void` als Parameterliste legt fest, dass die Funktion keine Parameter hat.
- Eine leere Parameterliste `()` trifft keine Aussage zu den Parametern<sup>4</sup>.

```
Bspl.: extern int    anton  (void);
        int    berta  (void);
        double caesar  ();
        int    main   ();
        void   sub     ();
        |      |      |
        Typ2   Name    Parameter
```

### Funktionsdefinition

Eine Definition hat einen ähnlichen Aufbau, beinhaltet aber zusätzlich noch einen Funktionsrumpf (*function body*). Der Funktionsrumpf ist ein Block.

**Form:** *Speicherklassen-Attribut*<sub>opt</sub> *Typ*<sup>2</sup> *Name* ( *Liste\_von\_Parametern*<sub>opt</sub> ) *Block*

Es gelten die gleichen Bedingungen wie oben, mit folgenden Ergänzungen.

- Wir verwenden hier kein *Speicherklassen-Attribut*<sup>3</sup>.
- Eine leere Parameterliste `()` bedeutet, dass es keine Parameter gibt.
- Eine Funktionsdefinition darf nicht innerhalb einer anderen Funktion erfolgen.

```
Bspl.: int anton (void)
{
    int i;

    i = ... ;           irgendwelche Berechnungen, die i belegen

    return i;
}
```

<sup>2</sup> Obwohl der Typ auch optional sein kann, wollen wir diesen immer explizit angeben. Siehe auch Kapitel 18.

<sup>3</sup> Details zu diesem Attribut gibt es später im Kapitel 9. Wir führen es hier nur bei Prototypen an, wenn sich die zugehörige Definition der Funktion in einer anderen Programmdatei befindet.

<sup>4</sup> Da hiermit dem Compiler die syntaktische Kontrolle über die Parameter zum Teil entzogen wird und danach inkonsistente Funktionsaufrufe erfolgen können, verwenden wir dies nur für Funktionen ohne Parameter.

## return-Anweisung

Eine Funktion beendet sich spätestens mit der letzten Anweisung des Blocks, liefert dann aber in vielen Fällen kein sinnvolles Ergebnis zurück<sup>5</sup>. Abhilfe schafft die `return`-Anweisung.

**Form:** `return Ausdruckopt ;`

Der Wert des *Ausdrucks* liefert das Ergebnis der Funktion. Der *Ausdruck* muss vom Ergebnistyp der Funktion sein oder sich gegebenenfalls konvertieren lassen. Ferner beendet die `return`-Anweisung die Funktion, d.h. die Bearbeitung der Anweisungen im Funktionsrumpf wird abgebrochen.

Fehlt der *Ausdruck*, wird kein sinnhafter Wert zurückgeliefert. Ein fehlender *Ausdruck* ergibt daher nur Sinn für eine Funktion vom Ergebnistyp `void` und ist in diesem Fall auch so vorgeschrieben.

```
Bspl.: int berta ()
{
    int res = anton ();      Aufruf obiger Funktion anton liefert eine ganze Zahl.
    if (res > 0)
        return 1;
    if (res < 0)
        return -1;
    return 0;
    printf ("Hallo!\n");    wird niemals ausgeführt (dead code6)
}
```

## Aufruf von Funktionen

Funktionsaufrufe haben wir bereits im Kapitel 4 behandelt, wobei wir bisher nur Funktionen aus der Standard-Bibliothek genutzt haben. Nun sind es halt auch selbstdefinierte Funktionen.

Funktionen können beliebig oft aufgerufen werden. Jeder Aufruf einer Funktion bewirkt die Ausführung des in der Funktionsdefinition enthaltenen Blocks.

```
Bspl.: int emil = anton () * berta ();
```

Die Funktionen `anton` und `berta` werden innerhalb eines Ausdrucks aufgerufen. Die Funktionen werden ausgeführt, ihre Ergebnisse in den Ausdruck eingesetzt und danach der Ausdruck weiter berechnet.

```
Bspl.: anton ();
```

Dieser Funktionsaufruf, formal auch ein Ausdruck, wird dank eines Semikolons zur Ausdrucksanweisung. Die Funktion wird ausgeführt, das Funktionsergebnis aber nicht weiter verwertet.

Für eine Funktion mit Ergebnistyp `void` repräsentiert das Vorgehen im zweiten Beispiel den einzig zulässigen Aufruf.

<sup>5</sup> In einigen Situationen werden Fehlerschlüssel zurückgeliefert.

<sup>6</sup> Wird unter Umständen vom Compiler beanstandet.

## Programmstrukturen

Nachfolgend betrachten wir exemplarisch die Struktur „Funktion ruft Funktion auf“. Damit eröffnet sich erstmalig die Möglichkeit, ein Programm in Form mehrerer Programmdateien zu realisieren.

Die Beispiele beschreiben modellhaft die Verwendung der C-Sprachmittel. In der Praxis wird man diese Modelle mischen. Wichtig ist es, eine gute Lesbarkeit und Organisation im Programmierprojekt zu erreichen. Die Struktur sollte einem logischen Prinzip folgen.

Wir bevorzugen die Modelle 2 und 4.

### Modell 1

Unsere zwei Funktionsdefinitionen werden in einer Programmdatei hintereinander abgelegt. Damit die aufgerufene Funktion `sub` in der aufrufenden Funktion `main` auch entsprechend bekannt ist, erfolgt die Definition von `sub` vor der von `main`.

Programmdatei

```
#include <stdio.h>

void sub ()
{
    printf ("Funktion sub\n");
}

int main ()
{
    printf ("Funktion main\n");
    sub ();
    return 0;
}
```

← Aufruf der Funktion `sub`

Es handelt sich hierbei um eine korrekte und häufig für kleine Projekte verwendete Struktur.

Unschön ist, dass der Anfang (sprich: die Startfunktion `main`) des Programms an Ende der Programmdatei steht. Die Reihenfolge eventuell weiterer Funktionsdefinitionen kann bei dieser Struktur nicht beliebig gewählt werden.

Vertauscht man die Reihenfolge der beiden Funktionen, gibt es Probleme. Leider ist es so, dass dann eine implizite Deklaration zum Funktionsnamen `sub` vorgenommen wird. (In C99/C11-Systemen sollte das „implizite `int`“ unterbunden sein; meist gibt es aber nur eine Warnung.) In der Folge führt dies im Beispiel dazu, dass diese implizite Deklaration nicht mit unserer tatsächlichen Definition übereinstimmt. Der Text der Warnung bzw. Fehlermeldung des Compilers kann daher etwas verwirrend sein und muss entsprechend interpretiert werden.

## Modell 2

Die beiden Funktionsdefinitionen werden in einer Programmdatei abgelegt. Die aufgerufene Funktion wird vorab deklariert, aber erst nach `main` definiert.

### Programmdatei

```
#include <stdio.h>

/* Prototyp */
void sub ();

int main ()
{
    printf ("Funktion main\n");
    sub ();
    return 0;
}

void sub ()
{
    printf ("Funktion sub\n");
}
```

Es handelt sich um eine gängige Struktur.

Die Reihenfolge der Funktionsdefinitionen kann durch vorherige Deklaration in der Programmdatei nach den Vorstellungen des Programmierers frei gestaltet werden.

## Modell 3

Das Programm wird in zwei Programmdateien zerlegt. Damit die aufgerufene Funktion in der aufrufenden Funktion bekannt ist, verbleibt in dieser Programmdatei die entsprechende Deklaration.

### Programmdatei<sub>1</sub>

```
#include <stdio.h>

/* Prototyp */
extern void sub ();

int main ()
{
    printf ("Funktion main\n");
    sub ();
    return 0;
}
```

### Programmdatei<sub>2</sub>

```
#include <stdio.h>

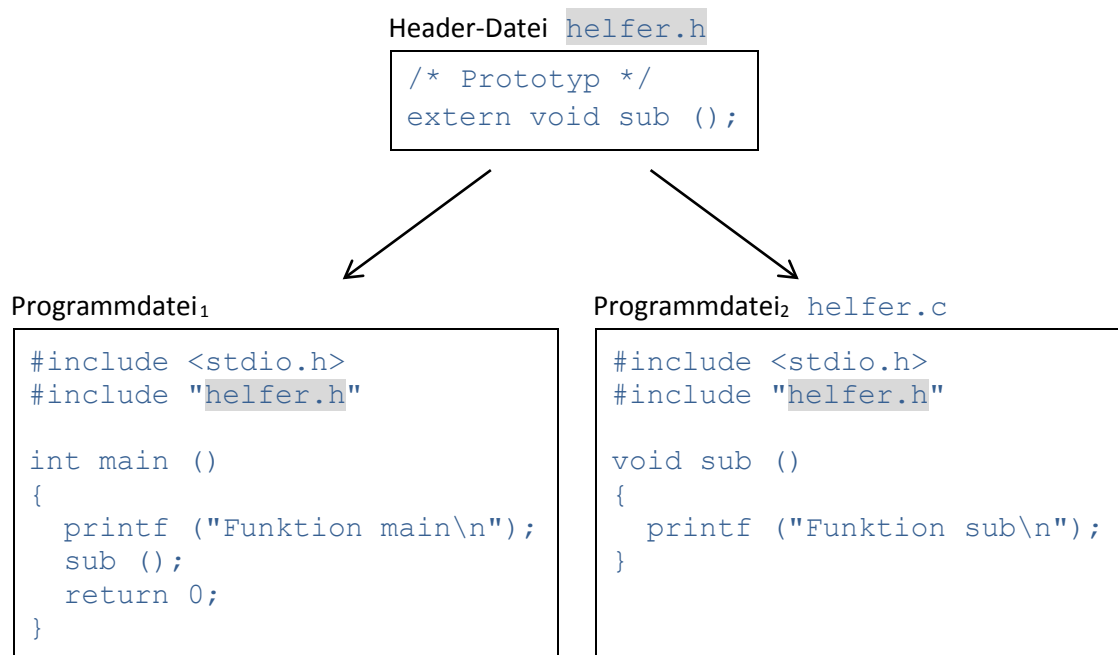
void sub ()
{
    printf ("Funktion sub\n");
}
```

Es handelt sich um eine korrekte, aber eher ungewöhnliche Struktur.

Liegen weitere Programmdateien vor, kann diese Struktur nachteilig sein. Die Prototypen müssten gegebenenfalls mehrfach explizit deklariert werden (Schreibaufwand und potentielle Fehlerquelle).

## Modell 4

Das Programm bleibt in zwei Programmdateien zerlegt. Die notwendige Funktionsdeklaration befindet sich in einer separaten Header-Datei (hier mit Namen `helfer.h`), welche mittels einer Präprozessor-Direktive eingefügt wird.



Es handelt sich hierbei um eine zugängliche Struktur.

Die Header-Datei mit dem Prototyp wurde auch in die zweite Programmdatei importiert, was in diesem Beispiel nicht unbedingt erforderlich ist, aber die Programmsicherheit weiter unterstützt.

In größeren Programmen wird man entsprechend den Anforderungen mehrere Funktionen in einer Programmdatei ablegen, mehrere Programmdateien haben und auch mehrere Header-Dateien anlegen. – Wenn eine einzige Programmdatei die ergänzenden Definitionen zu einer Header-Datei vornimmt („Schnittstelle“ und „Implementierung“, wie im Beispiel), schlagen wir vor, diese Dateien auch gleichartig zu benennen.

Zur Syntax: Man beachte, dass die Dateinamen bei der Standard-Bibliothek in spitzen Klammern eingeschlossen sind. Für unsere eigene Header-Datei sind es hingegen Doppelapostrophe.

## Kompilation und Binden

Die einzelnen Programmdateien müssen übersetzt, die entstandenen übersetzten Programmstücke zusammengebunden werden. Darauf sind wir bereits im Kapitel 1 eingegangen.

Moderne Entwicklungssysteme unterstützen die Programmentwicklung, indem Sie zusammengehörige Programmdateien organisatorisch in Form eines **Projektes** zusammenfassen. Soll das neue ausführbare Programm erstellt werden, weiß das Entwicklungssystem, welche Programmdateien dazugehören und veranlasst die Kompilation und das Binden/Linken weitgehend automatisch.

Zu erwähnen ist speziell für Unix und Derivate noch das `make`-Tool, mit dem die Abhängigkeiten und die daraus resultierenden Arbeitsschritte festgelegt und automatisiert ausgeführt werden können.

## Anmerkungen zur Funktion `main`

Jedes C-Programm startet sich durch den Aufruf einer Funktion mit Namen `main`. Ansonsten ist die Funktion `main` eine „ganz normale“ Funktion ohne Besonderheiten. Das Ergebnis der Funktion (`return`-Anweisung) kann dabei von der Systemumgebung ausgewertet werden. 0 gilt als erfolgreiche Programmausführung.

Ferner kann die Methode `main` auch Argumente des Programmstarts abfragen. Dazu ist allerdings eine besondere Vereinbarung der Parameterliste erforderlich, die wir hier nicht behandeln.

Die Vorstellung, wie die Funktion `main` aufzubauen ist, hat sich im Laufe der Zeit mit den verschiedenen Sprachstandards durchaus geändert. So war es üblich, den expliziten Ergebnistyp `int` auch wegzulassen und auf die Voreinstellung zu bauen. Heute sieht man diese Angabe als verpflichtend an. Ebenfalls wechselnd stellt sich der Umgang mit der `return`-Anweisung dar, auf die man für ein normales Programmende auch verzichten kann.

Hinzu kommt noch die Schwierigkeit, dass Entwicklungssysteme eigene Vorstellungen entwickeln.

Wir haben uns daher entschieden, Ergebnistyp und `return`-Anweisung auch bei der `main`-Funktion unabhängig vom Standard immer mit anzugeben.

## Mehr zu Funktionen (Ausblick)

Ruft eine Funktion A eine Funktion B auf, so stellt sich die Frage, wie diese beiden Funktionen Werte (Daten, Objekte, Informationen) untereinander austauschen können. Dazu gibt es verschiedene Techniken; hier eine kurze Übersicht mit einer symbolischen Darstellung der Datenwege und Verweisen auf kommende Kapitel.

- Funktionsergebnis / `return`-Anweisung (in diesem Kapitel)  
Datenfluss:  $A \Leftarrow B$
- Parameterübergabe (*call by value*) (Kapitel 8)  
Datenfluss:  $A \Rightarrow B$
- Parameterübergabe mit Zeigern (*call by reference*) (Kapitel 10)  
Datenfluss:  $A \Leftrightarrow B$
- Globale Objekte (Kapitel 9)  
Datenfluss: 

Global Data
-------------

$\Updownarrow$   
A

$\Updownarrow$   
B





## 8. Parameter und Argumente

### Parameter- und Argumentliste

Parameter dienen zum Datenaustausch zwischen aufrufender und aufgerufener Funktion. Dazu wird die aufzurufende Funktion mit einer Liste von Parametern versehen. Beim Aufruf werden dann in einer weiteren Liste Argumente (Ausdrücke) für die Parameter angegeben. – Wir benutzen hier die Terminologie „Parameter/Argument“. Alternativ ist ebenfalls die Sprechweise „formaler Parameter“ und „aktueller Parameter“ gebräuchlich.

Die Bestandteile der Listen werden durch Kommas voneinander getrennt. Die Zuordnung zwischen Parametern und Argumenten geschieht anhand der Position in der jeweiligen Liste. Die Listen müssen gleich lang sein; Funktionen mit variablen Listen behandeln wir an dieser Stelle nicht.

### Vereinbarung von Parametern

Die Vereinbarung von Parametern erfolgt durch Angabe des Typs und eines Namens (Bezeichner) für jeden Parameter einzeln. Es ist syntaktisch nicht möglich, bei gleichem Typ die nachfolgende Typangabe in der Parameterliste wegfallen zu lassen. Ein Parameter wirkt im Funktionsrumpf unter seinem Namen wie eine lokale Variable des entsprechenden Typs.

Ein Parametername ist optional in einer Funktionsdeklaration. Es verbleibt dann nur die Typangabe.

Bspl.: Die Tabelle der mathematischen Funktionen im Kapitel 3 enthält die entsprechenden Deklarationen mit Parametern aus der Header-Datei `<math.h>`.

Bspl.: Ausgabe eines Musters (Dreieck aus Sternchen)

```
#include <stdio.h>

/* Prototyp */
void show (char, int);

int main ()
{
    int i;
    for (i = 10; i >= 1; i--)
    {
        show ('*', i);
        printf ("\n");
    }
    return 0;
}

void show (char zeichen, int anzahl)
{
    while (anzahl-- > 0)
        printf ("%c", zeichen);
}
```

Deklaration: 2 Parameter

Aufruf: 2 Argumente

Definition: 2 Parameter

Hier werden die Parameter wie Variablen eingesetzt.

## Parameterübergabe

Beim Funktionsaufruf (siehe Kapitel 4 und 7) werden nun die Argumente (Ausdrücke) berechnet und die Parameter mit den berechneten Werten der Reihe nach initialisiert. Gegebenenfalls werden dabei die Werte typkonvertiert. Danach wird der Funktionsrumpf (Block) ausgeführt. – Diese Technik der Übergabe wird als *call by value* bezeichnet.

C kennt nur diese eine Übergabetechnik. Insbesondere gibt es originär kein *call by reference*, d.h. wir können (noch) keine Ergebnisse aus der aufgerufenen Funktion an die aufrufende Funktion über Parameter zurückliefern. Allerdings werden wir durch die Verwendung von Zeigern diese zweite Übergabetechnik später einmal nachstellen. Dann greifen wir das folgende Beispiel noch einmal auf.

Bspl.: Dieses Beispiel basiert auf der irrigen Annahme, dass durch die Wertzuweisung an den Parameter `x` auch das Argument `z` einen neuen Wert erhalten würde. Die Parameterübergabe bewirkt aber insgesamt nur einmalig die Initialisierung von `x` mit dem Wert von `z`. Eine weitere Wirkung gibt es nicht.

```
#include <stdio.h>

/* Prototyp */
void belege (double);

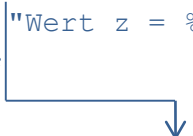
int main ()
{
    double z = 0.0;
    belege (z);
    printf ("Wert z = %f\n", z);
    return 0;
}

void belege (double x)
{
    x = 55.5;
    printf ("Wert x = %f\n", x);
}
```

z wird hierdurch **nicht** neu belegt!  
Ausgabe: Wert z = 0.000000

Übergabe: x erhält den Wert von z.

Nur Parameter x wird neu belegt.



## Konstante Parameter

Parameter dürfen auch mit dem zusätzlichen Attribut `const` vereinbart werden. Es gelten die oben getroffenen Aussagen mit einer Ausnahme: Der Parameter wirkt jetzt (nur) im Funktionsrumpf wie ein Objekt, dessen Wert sich nicht mehr verändern lässt. (Ein konstanter Parameter verlangt nicht, dass auch das Argument konstant sein muss.)

Bspl.: Modifikation der vorausgehenden Funktion `show`

```
void show (const char zeichen, const int anzahl)
{
    int j;
    for (j = 1; j <= anzahl; j++)
        printf ("%c", zeichen);
}
```

Hier werden die Parameter wie symbolische Konstanten eingesetzt.

## 9. Eigenschaften von Vereinbarungen

---

Dieses Kapitel fasst Aussagen der vorangegangenen Kapitel noch einmal zusammen, klärt unbehandelte Situationen und stellt das Thema in einem größeren Zusammenhang dar.

### Deklaration – Definition – Vereinbarung

Eine **Deklaration** führt einen Namen in ein Programm ein. Für Datenobjekte wird dabei insbesondere der Typ festgelegt, für Funktionen der Ergebnistyp und eventuell die Parametertypen. Eine Deklaration „realisiert“ das Vereinbarte aber noch nicht.

Eine **Definition** leistet das Gleiche wie eine Deklaration, liefert aber gleichzeitig noch dazu die Realisierung. – Bei Datenobjekten wird an dieser Stelle der zugehörige Speicher reserviert und gegebenenfalls erfolgt auch eine Initialisierung. – Eine Funktionsdefinition besitzt einen Funktionsrumpf (Block) mit den lokalen Vereinbarungen und Anweisungen. Eine vorhandene Parametervereinbarung zählt dann ebenfalls mit zu den Definitionen. – Auch vollständige Typvereinbarungen (bisher: Aufzähltypen) sehen wir als Definitionen an.

Für eine Entität (Datenobjekt, Funktion, ...) kann es in einem Programm immer nur eine Definition, möglicherweise aber beliebig viele stimmige Deklarationen geben.

Eine **Vereinbarung** ist entweder eine Deklaration oder eine Definition. Wir benutzen diesen Begriff als gemeinsamen Oberbegriff.

### Gültigkeitsbereiche

Eine Vereinbarung ist immer nur innerhalb eines gewissen Textbereichs (*scope*) gültig. Diese Eigenschaft ist syntaktischer Art und bezieht sich auf einen Bereich des C-Programmtexts.

- Eine Vereinbarung in einem Block  
ist gültig von der Vereinbarung bis zum Blockende (lokale Vereinbarung).
- Eine Vereinbarung außerhalb aller Blöcke  
ist gültig von der Vereinbarung bis zum Dateiende (globale Vereinbarung).

Diese Auflistung ist zwar etwas vereinfacht, beschreibt aber die für uns interessanten Fälle. Einzelregelungen werden mit einigen Themen aufgeführt, z.B. für `for`-Schleifen gemäß Standard C99/C11, bei Parametern von Funktionsvereinbarungen, im Zusammenhang mit Strukturen.

Eine Mehrdeutigkeit ein und desselben Namens durch überlappende Gültigkeitsbereiche der jeweiligen Vereinbarungen darf grundsätzlich nicht auftreten. Allerdings eröffnen einige Sprachmittel neue „Namensräume“ derart, dass sie nicht mit anderen Namensräumen kollidieren. So kann beispielsweise ein Etikett oder der Name einer Strukturkomponente (Kapitel 14) unabhängig vom gemeinsamen Namensraum der Objekte und Funktionen gewählt werden.

Eine globale oder lokale Vereinbarung wird verdeckt bei Vereinbarung desselben Namens in einem im Gültigkeitsbereich eingeschlossenen Block, d.h. in einem (inneren) Block kann ein Name erneut für eine andere lokale Entität vergeben werden. Das Gleiche gilt sinngemäß für Parameter.

```

Bspl.: /* global oder lokal */
double zulu;
int zulu;

```

Fehler: Name ist mehrdeutig.

```

Bspl.: /* global */
double xara;
void xara ()
{
}

```

Fehler: Name ist mehrdeutig.

```

Bspl.: /* global oder lokal*/
double Farbe;
enum Farbe {ROT, GELB, GRUEN};

```

zulässig: unterschiedliche Namensräume

```

Bspl.: /* global */
double yana;
void test ()
{
    /* lokal */
    int yana;
    ...
}

```

zulässig: Globale Vereinbarung wird verdeckt.

```

Bspl.: /* global */
double mara;
void sub_1 (int mara)
{
}
void sub_2 (int mara)
{
}

```

zulässig: als Parametername

ebenso

```

Bspl.: void yoko ()
{
    double yoko;
    {
        int yoko;
        ...
    }
    {
        char yoko;
        ...
    }
    ...
}

```

zulässig: Funktionsname wird verdeckt.

zulässig: im inneren Block

ebenso

Nachtrag: Die globalen Variablen-Vereinbarungen dieser Seite sind vorläufige Definitionen im Sinne der nachfolgenden Ausführungen. Alle anderen Vereinbarungen sind Definitionen.

## Bindung

Ein Name, der in mehr als einer Übersetzungseinheit verwendet werden kann, heißt extern gebunden (*external linkage*), falls er dieselbe Entität bezeichnet; andernfalls heißt er intern gebunden (*internal linkage*). Bei interner Bindung können also unterschiedliche Übersetzungseinheiten durchaus unterschiedliche Entitäten gleichen Namens besitzen.

## Lebensdauer und Speicherklassen

Die Lebensdauer von Objekten ist nicht zu identifizieren mit der Gültigkeit ihrer Vereinbarung. Während die Gültigkeit eine textbezogene Eigenschaft einer Vereinbarung ist, indem sie einen Bereich des C-Quelltextes markiert, bezeichnet die Lebensdauer die temporäre Existenz eines Objekt bezogen auf einen Programmablauf.

### Speicherklassen

Geregelt wird die Lebensdauer über die Zugehörigkeit zu einer der folgenden drei Speicherklassen.

- automatisch vom Zeitpunkt der Vereinbarung bis zum Ausführungsende des umgebenden Blocks
- statisch gesamte Laufzeit des Programms
- dynamisch demnächst bei Zeigern auf namenlose Objekte (Kapitel 15)

Parameter gehören immer zur automatischen Speicherklasse. Globale Objekte sind nicht in einem Block vereinbart worden und können folglich nur zur statischen Speicherklasse gehören.

## Speicherklassen-Attribute

Das Speicherklassen-Attribut ist eine optionale Spezifizierung zu Beginn einer Vereinbarung. Dieses Attribut bestimmt die Speicherklasse, aber auch die Bindung und die Art der Vereinbarung.

Bspl.: `static double d;`  
`extern int len;`  
|  
Speicherklassen-Attribut

Näheres regelt diese Tabelle.

Attribut	Speicherklasse	Bindung	Vereinbarung	Sonstiges
auto	automatisch	intern	Definition	nur lokal
register	automatisch	intern	Definition	nur lokal
static	statisch	intern	Definition	
extern	statisch	extern	Deklaration	<sup>1</sup>

<sup>1</sup> Einige Systeme lassen auf globaler Ebene Initialisierungen zu. Die Vereinbarung wird dann zu einer Definition. Keinesfalls ist dies aber mit einer lokalen, externen Vereinbarung möglich.

## Anmerkungen

- Die vollständige Tabelle gilt für Objekte (Variablen, Konstanten), jedoch nicht für Parameter. Für Funktionen und Typvereinbarungen gelten besondere Regeln, siehe unten.
- Das Attribut `register` weist den Compiler auf häufige Nutzung des Objekts hin, ist aber ansonsten ohne weitere Bedeutung. Nicht jeder Datentyp eignet sich dazu (systemabhängig).
- Eine explizite Initialisierung statischer Objekte darf nur mit einem konstanten Ausdruck erfolgen. Objekte der statischen Speicherklasse ohne Initialisierung werden ansonsten implizit mit Null initialisiert (im jeweiligen Typ).
- Funktionen gehören stets zu einer statischen Speicherklasse (`static` oder `extern`).
- Die Attribute sind auf Typvereinbarungen (bisher: Aufzähltypen) selbst nicht anwendbar<sup>2</sup>, aber durchaus auf Objekte dieser Typen.

## Voreinstellungen

Speicherklassen-Attribute können in Vereinbarungen auch fehlen; wir haben sie bisher auch nicht benötigt. In solchen Fällen treten Voreinstellungen in Kraft.

- Lokale Objekte      Voreinstellung: `auto`
- Globale Objekte      Diese gehören zur statischen Speicherklasse und haben externe Bindung. Mit einer Initialisierung handelt es sich um die Definition einer Variablen oder auch einer (symbolischen) Konstanten.  
Ohne eine Initialisierung handelt es sich aber um eine vorläufige Definition (*tentative definition*) zu einem Namen. Gibt es in keiner Übersetzungseinheit dazu eine explizite Definition, so bilden alle vorläufigen Definitionen zusammen eine Definition mit impliziter Initialisierung Null; andernfalls wirken sie wie Deklarationen.
- Funktionen      Voreinstellung: `extern`
- Typvereinbarungen      Die Vereinbarung zum Typ selbst wirkt statisch mit interner Bindung. Für Objekte eines selbstvereinbarten Typs gelten hingegen die üblichen Regeln.

## Beispiele

Mit den nachfolgenden beiden Beispielen vergleichen wir globale und lokale Vereinbarungen.

Bspl.: außerhalb aller Blöcke (global):

	Speicherklasse	Bindung	Vereinbarung	Initialisierung
<code>int a;</code>	statisch	extern	vorläufige Def.	vorläufig 0
<code>int b = 36;</code>	statisch	extern	Definition	36
<code>extern int c;</code>	statisch	extern	Deklaration	nicht hier
<code>static int d;</code>	statisch	intern	Definition	0

<sup>2</sup> Einige Systeme erzeugen Warnungen, andere ignorieren die Angabe stillschweigend.

Bspl.: innerhalb eines Blocks (lokal):

	Speicherklasse	Bindung	Vereinbarung	Initialisierung
<code>int a;</code>	automatisch	intern	Definition	nein
<code>int b = 36;</code>	automatisch	intern	Definition	36
<code>extern int c;</code>	statisch	extern	Deklaration	nicht hier
<code>static int d;</code>	statisch	intern	Definition	0

Nachfolgendes Beispiel demonstriert den Unterschied zwischen lokalen Objekten der automatischen und der statischen Speicherklasse.

Bspl.: `#include <stdio.h>`

```
/* Prototyp */
void sub (int);

int main ()
{
    int i;
    for (i = 1; i <= 4; i++)
        sub (i);
    return 0;
}

void sub (int p)
{
    int a = 0;
    static int s;
    a++; s++;
    printf ("p=%d a=%d s=%d\n", p, a, s);
}
```

entsteht als Parameter mit jedem sub-Aufruf neu,  
wird über das Argument unterschiedlich initialisiert

automatische Speicherklasse,  
in jedem Aufruf mit 0 initialisiert,  
danach inkrementiert

statische Speicherklasse,  
wird **einmal** implizit mit 0 initialisiert,  
der inkrementierte Wert „überlebt“  
den jeweiligen Funktionsaufruf

Ausgabe: p=1 a=1 s=1  
p=2 a=1 s=2  
p=3 a=1 s=3  
p=4 a=1 s=4

Das folgende Beispiel zeigt, wie eine externe Bindung etabliert werden kann.

Bspl.: Programmdatei<sub>x</sub>

```
/* Deklaration */
extern double mabo;
```

Programmdatei

```
/* globale Definition */
double mabo = 0.0;
```

Der Variablenname `mabo` wird hier extern gebunden. Rechts liegt dabei eine Definition vor, entsprechend der Voreinstellung für globale Variablen mit Initialisierung. Weitere Deklarationen wie links können nach Bedarf global oder lokal erfolgen. – Auf die sinnhafte Verwendung von Header-Dateien gehen wir mit diesem Beispiel anschließend noch ein.

## Header-Dateien

(Zum Umgang mit Header-Dateien vergleiche Kapitel 1 und Kapitel 7.)

Aus den vorangegangenen Abschnitten lassen sich einige Empfehlungen zum Inhalt von Header-Dateien herleiten.

Folgende Dinge kann eine Header-Datei sinnvoll enthalten.

Kommentare	<code>/* Hallo! */</code>
Präprozessor-Direktiven	<code>#include "irgendwas.h"</code>
Variablen-Deklarationen	<code>extern double mabo;</code>
Konstanten-Definitionen	<code>static const float pi = 3.141593;</code>
Typvereinbarungen	<code>enum Farbe {ROT, GELB, GRUEN};</code>
Prototypen	<code>extern void sub ();</code>

Im Gegenzug sollte eine Header-Datei dies nie enthalten.

Variablen-Definitionen	<code>double mabo = 0.0;</code>
Funktionsdefinitionen	<code>void sub () { ... }</code>

Diese Empfehlungen sind nicht Bestandteil der Sprachdefinition, sondern Vorschläge für einen vernünftigen Einsatz der Header-Dateien.

Wir vermeiden in unseren Beispielen mit mehreren Programmdateien vorläufige Definitionen, weil wir den Punkt der Definition gerne genau lokalisieren möchten. Entsprechend wollen wir auch keine vorläufigen Definitionen in Header-Dateien einfügen. Das darf man aber durchaus anders sehen und nachfolgend gibt es auch ein Beispiel hierzu. – Das Beispiel zeigt, wie man zweckmäßig externe Bindungen bei Variablen unter Verwendung einer Header-Datei aufbaut.

Ein gesonderter Blick auf die Konstanten: Diese können unbedenklich in mehrfacher Kopie in einem Programm vorhanden sein. Insofern ist in diesem Fall eine `static`-Definition in der Header-Datei opportun. Die Bindung der einzelnen Kopien ist dann jeweils intern. Das Gleiche gilt sinngemäß für selbstvereinbarte Datentypen. (Siehe obige Vereinbarungen zu `pi` und `Farbe`.)

Prototypen wollen wir grundsätzlich mit dem nicht unbedingt erforderlichen Attribut `extern` einleiten, wenn sich die zugehörige Definition der Funktion in einer anderen Programmdatei befindet. Das gilt so auch für eine derartige Header-Dateien. Dies hat allerdings nur eine kommentierende Wirkung. Die zugehörige Funktionsdefinition erfolgt bei uns ohne Speicherlassen-Attribut. Dieses Vorgehen entspricht dem Kapitel 7. (Siehe auch obige Vereinbarungen zu `sub`.)

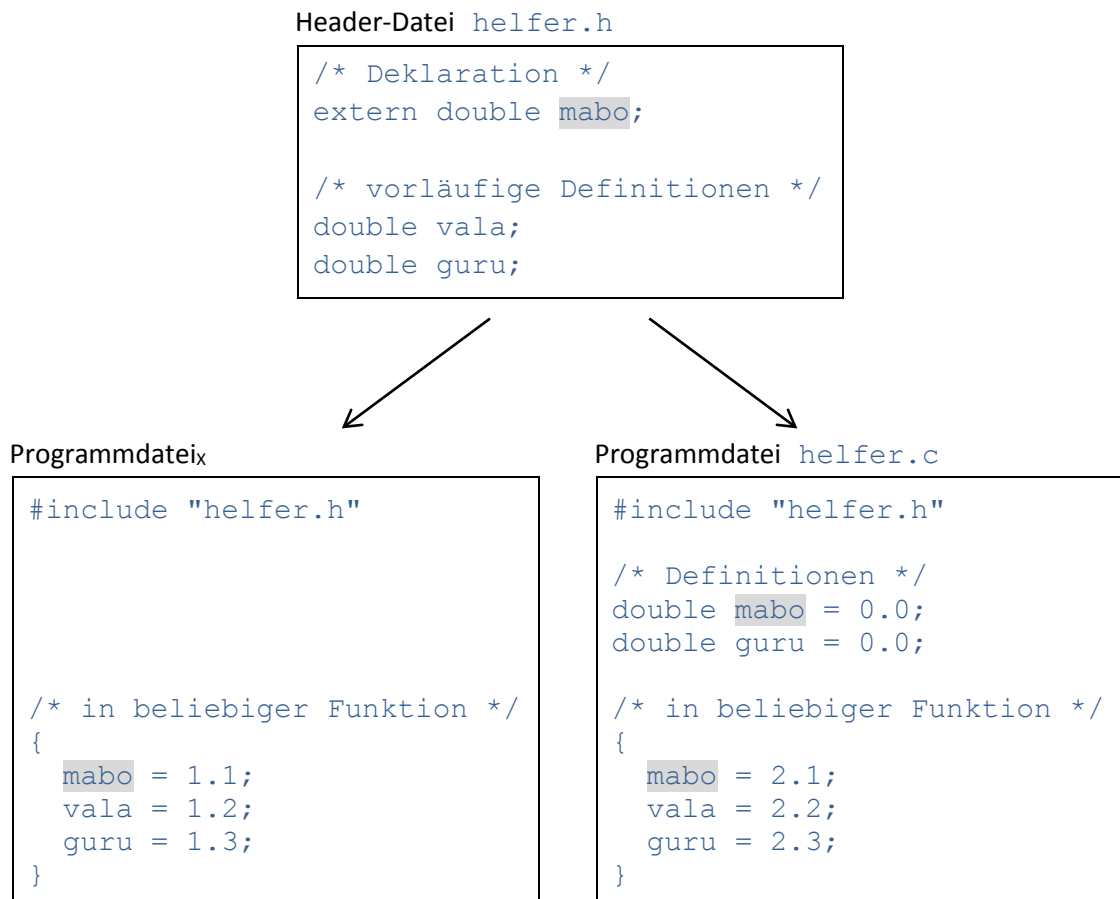
Auf Funktionen, welche nicht extern gebunden werden, gehen wir in diesem Zusammenhang nicht näher ein. Für eine statische Funktion, also mit interner Bindung, ergibt ein Prototyp in einer Header-Datei selten einen Sinn.



### Beispiel (Modell 4)

Wir nehmen eine Programmstruktur aus dem Kapitel 7 als Grundlage und vereinbaren wie oben eine globale Variable `mabo` mit externer Bindung.

Dazu gibt es genau eine Definition in einer Programmdatei und eine Deklaration in einer Header-Datei, die jedoch in beliebig viele weitere Programmdateien mittels einer `#include`-Direktive importiert werden kann. So wird die Variable in all diesen Programmdateien bekannt (eine Entität).



Alternativ der Umgang mit vorläufigen Definitionen:

- Da es keine explizite Definition gibt, bilden alle importierten, vorläufigen Definitionen von `vala` zusammen wie eine Definition mit impliziter Initialisierung Null.
- Im Gegensatz dazu gibt es zu `guru` eine explizite Definition und die vorläufige Definition in der Header-Datei wirkt somit auch nach dem Import in eine beliebige Programmdatei jeweils nur wie eine Deklaration.

Wir wiederholen hier noch einmal unsere Empfehlung, auf vorläufige Definitionen in einem solchen Zusammenhang zu verzichten. Wirkliche Vorteile ergeben sich durch eine Nutzung nicht. Im Beispiel `mabo` hingegen wird deutlicher ersichtlich, was Deklaration und was Definition ist.

## Rekursionen

Unter einer Aufrufhierarchie verstehen wir eine geordnete Liste von Funktionsnamen: An oberster Stelle steht die Startfunktion `main`; mit jedem Funktionsaufruf wird die aufgerufene Funktion am Ende hinzugefügt; bei deren Beendigung wird sie wieder gestrichen. Die Liste ist somit dynamisch. Sie zeigt uns jeweils zu einem spezifischen Zeitpunkt, auf welchem Weg durch die Funktionen der aktuelle Punkt der Ausführung erreicht wurde.

C-Funktionen können sich selbst direkt oder indirekt aufrufen. Dies bewirkt, dass in der Aufrufhierarchie der Funktionsname mehr als einmal vorkommt. Rekursionen machen nur Sinn, wenn aus algorithmischer Sicht sichergestellt ist, dass die Aufrufhierarchie endlich ist.

Im rekursiven Fall werden bei der Ausführung die vereinbarten lokalen Objekte der automatischen Speicherklasse inkl. der Parameter mehrfach bereitgestellt. Für jede Ebene der Aufrufhierarchie gibt es dann einen entsprechend angelegten Bereich im Speicher (*stack*). Lokale Objekte der statischen Speicherklasse existieren hingegen nur genau einmal und werden auch nur einmal initialisiert.

Bspl.: Als Beispiel für eine Rekursion ziehen wir die beliebte Definition der mathematischen Fakultät heran:

$$n! := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{falls } n > 0, n \in \mathbb{N} \end{cases}$$

Bei der Umsetzung in eine C-Funktion dienen hier vorzeichenlose Datentypen. Man beachte, dass an diversen Stellen implizite Typkonvertierungen stattfinden. Bei einem System mit 32 bit für die verwendeten ganzzahligen Typen liefert `fak(13)` noch den korrekten Wert, bei `fak(14)` wird bereits der Wertebereich überschritten.

```
unsigned int fak (unsigned int n)
{
    if (n == 0)
        return 1;
    else
        return n * fak(n-1);           Hier erfolgt der direkte rekursive Aufruf.
}
```

Der Parameter `n` muss beim Programmablauf mehrfach realisiert werden, z.B. erzeugt der Aufruf von `fak(3)` die erste Entität `n` gleich 3. Rekursiv wird dann `fak(2)` aufgerufen, erzeugt wird eine weiterer Parameter `n` gleich 2, etc. Mit Berechnung von `fak(0)` können die rekursiven Aufruf der Reihe nach aufgelöst werden, die jeweiligen Zwischenergebnisse werden dabei ausgewertet und der Speicher wird Ebene um Ebene wieder freigegeben.

# 10. Zeiger

## L- und R-Wert

Wir verwenden den Namen einer Variablen in unterschiedlicher Bedeutung: Einmal verstehen wir diesen als Bezeichnung für einen Speicherplatz des entsprechenden Typs. An anderer Stelle meinen wir damit aber den Wert, der an diesem Platz gespeichert ist. Wir verdeutlichen dies an der einfachen Anweisung:

$$c = c + 1;$$

An dieser Stelle soll **nicht** der Wert eingesetzt werden.

Hier meinen wir den Speicherplatz (die Speicheradresse).

Dieser Umstand motiviert in C die Einführung folgender Begriffe.

- Unter einem **L-Wert** (*lvalue*) versteht man die **Adresse** eines Objekts, d.h. die „Nummer“ des zugeordneten Speicherplatzes<sup>1</sup> im Arbeitsspeicher.

Die Buchstabe **L** bezieht sich auf *location* und auf die Seite links/*left* der Zuweisung.

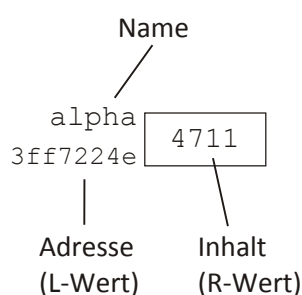
- Mit **R-Wert** (*rvalue*) bezeichnet man hingegen den Inhalt eines Objekts, also den Inhalt des zugehörigen Speicherplatzes interpretiert anhand des verwendeten Datentyps.

**R** steht hier für *read* und für die Seite rechts/*right* der Zuweisung.

Die Verwendung jeweils des L- oder R-Werts ist kontextabhängig. Im obigen Beispiel der Wertzuweisung ist festgelegt, dass links der L-Wert und rechts der R-Wert gemeint sind. An einer Stelle, an welcher der R-Wert verwendet würde, kann durch Benutzung des **Adressoperators** & der L-Wert eingesetzt werden.

Wir halten häufig die Begriffe „Adresse“ und „Inhalt/Wert“ für einprägsamer und werden weiterhin auch diese Bezeichnungen benutzen.

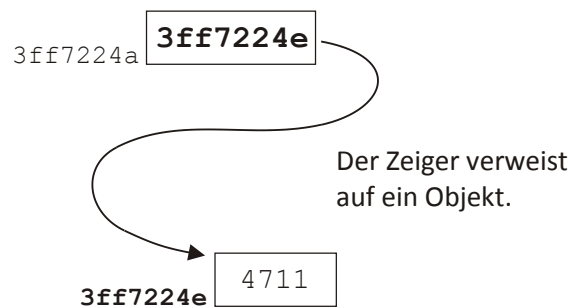
Ferner vereinbaren wir folgende graphische Darstellung für die Speicherinhalte:



<sup>1</sup> Die im Speicher realisierten Objekte sind je nach Datentyp unterschiedlich groß. Daher müsste man eigentlich immer von der **Anfangs**adresse reden, welche diejenige Speicherzelle benennt, an der die Ablage des Objektes beginnt.

## Grundlagen

Ein **Zeiger** (*pointer*) ist ein Datenobjekt, welches auf ein anderes Objekt im Arbeitsspeicher verweisen kann. Die nachfolgende Graphik veranschaulicht diese Beziehung.



Der Inhalt (R-Wert) des Zeigers ist eine Adresse, also der L-Wert des Objekts, auf das der Zeiger verweist. Dieses Objekt bezeichnen wir als **angezeigtes** oder **referenziertes** Objekt. Folgen wir einem Zeiger, so **dereferenzieren** wir den Zeiger.

Ein Zeiger kann auch „leer“ sein und damit auf kein Objekt verweisen. In C entspricht dies dem Wert (Inhalt) 0. Gebräuchlich ist hierfür auch die Bezeichnung NULL-Zeiger.

Zeiger sind im Regelfall typbehaftet, d.h. ein Zeiger kann nur auf Objekte eines festgelegten **Basistyps** verweisen<sup>2</sup>. Auch wenn wir in den nachfolgenden Beispielen zunächst nur Zeiger auf die Standard-Datentypen betrachten, so können doch Zeiger zu jedem beliebigen Datentyp vereinbart werden.

## Vereinbarung

Zur Vereinbarung eines variablen Zeigers wird ein Stern \* in die bekannte Form der Variablen-Vereinbarung zwischen der Typangabe und dem Objektname eingefügt. Leerraum ist dabei ohne Bedeutung.

Bspl.: <code>int* a;</code>	<code>a</code> kann auf <code>int</code> -Variablen zeigen.
<code>int *b;</code>	<code>b</code> ebenso
<code>double * c;</code>	<code>c</code> kann auf <code>double</code> -Variablen zeigen.
	Der Datentyp von <code>a</code> und <code>b</code> heißt <code>int*</code> (Zeiger auf <code>int</code> ); der von <code>c</code> heißt <code>double*</code> (Zeiger auf <code>double</code> ).

### Warnung!

<code>int* a, b;</code>	hat die Bedeutung: <code>int *a; int b;</code>
	und nicht: <code>int *a; int *b;</code>
	<code>b</code> ist somit <b>kein</b> Zeiger.
<code>int *c, *d;</code>	zwei Zeiger

<sup>2</sup> Es gibt auch Zeiger vom Typ `void*`. Diese „wissen“, wohin sie zeigen, aber nicht, worauf sie zeigen. Derartige Zeiger werden wir nicht eingehender behandeln.

## Wertzuweisung

Einem Zeiger kann die Adresse (L-Wert) eines Objekts des jeweiligen Datentyps zugewiesen werden.

```
Bspl.: int v = 4711;
       int *p, *q;
       double *r;

       p = &v;                p zeigt auf die Variable v.
           ↑
       Adressoperator für die Adresse (L-Wert) der Variablen
```

Ebenso kann der Inhalt (R-Wert) eines anderen Zeigers gleichen Typs zugewiesen werden.

```
Bspl.: q = p;                q erhält den Inhalt (R-Wert) von p.
                               q und p zeigen danach auf das gleiche Objekt, hier v.
```

Die Konvertierungsregeln des Kapitels 3 sind auf Zeiger **nicht** übertragbar. Unstimmige Zuweisungen erzeugen allerdings meist nur Warnungen bei der Übersetzung.

```
Bspl.: r = &v;                nicht zulässig
```

## Initialisierung

Zeiger können bei ihrer Entstehung auch initialisiert werden, d.h. sie zeigen sofort auf ein Objekt. Es gelten weiterhin die bisher bekannten Regeln zur Initialisierung.

```
Bspl.: int v = 4711;
       int *p = &v;
       int *q = p;
```

## Dereferenzierung

Ist `p` ein (nichtleerer) Zeiger, so bezeichnet `*p` in Ausdrücken das angezeigte Objekt (L- und R-Wert).

```
Bspl.: int u = 1234, v = 4711, w = 1909;
       int *p = &u;           p zeigt auf u.                (Definition mit Initialisierung)
       p = &v;                p zeigt auf v.                (Wertzuweisung an Zeiger)
       *p = w;                entspricht: v = w;             Dereferenzierung
       ++*p;                  v wird um 1 erhöht.
       (*p)++;                ebenso                          (Klammerung erforderlich)
       printf ("%d", *p);      Ausgabe: 1911
       printf ("%p", p);       Ausgabe3 einer Adresse        (Formatelement %p)
       *p = 0;                 v ist jetzt 0.
       p = 0;                  p ist jetzt ein leerer Zeiger.
```

<sup>3</sup> Der Standard C90 fordert zusätzlich eine Konvertierung: `printf ("%p", (void*)p);`

## Adressarithmetik

Addition und Subtraktion von ganzzahligen Werten auf Zeigern bewirken Berechnungen in Einheiten der Speichergröße des zugrundeliegenden Basistyps.

```
Bspl.: int i = 1;
       int *p = &i;
```

```
*p += 2;
```

Dereferenzierung und „normale“ Arithmetik:

Der Wert des angezeigten Objekts wird um 2 erhöht;  
i erhält also den Wert 3.

```
p += 2;
```

Adressarithmetik:

Der Inhalt (R-Wert) von p wird um die Größeneinheit von zwei int-Speicherplätzen erhöht;  
p zeigt nicht mehr auf i, sondern 2 int-Plätze „dahinter“.

Die Adressarithmetik werden wir eingehender im übernächsten Kapitel mit Bezug zu Arrays nutzen.

## Zeiger auf Konstanten und konstante Zeiger

... sind nicht dasselbe!

„**Zeiger auf Konstante**“ bedeutet, dass das angezeigte Objekt als Konstante angesehen wird, d.h. es lässt sich über diesen Zugriff nicht verändern.

Der Zeiger selbst ist variabel, kann also im Laufe der Zeit auch auf verschiedene Objekte zeigen.

```
Bspl.: const int i = 2404;
```

```
const int *p = &i;    p ist ein Zeiger auf const int und zeigt auf i.
```

```
p = ... ;            zulässig
```

```
*p = ... ;           nicht zulässig
```

„**Konstanter Zeiger**“ heißt, dass ein Zeiger selbst nicht veränderbar ist, also immer auf das gleiche Objekt zeigt. – Konstante Zeiger müssen initialisiert werden.

Das angezeigte Objekt wird hierdurch nicht als konstant angesehen.

```
Bspl.: int i = 2404;
```

```
int* const p = &i;    p ist ein konstanter Zeiger auf int und zeigt auf i.
```

```
p = ... ;            nicht zulässig
```

```
*p = ... ;           zulässig
```

Ja, es gibt auch konstante Zeiger auf Konstanten.

```
Bspl.: const int k = 2404;
```

```
const int* const z = &k;
```

## Mehr zu Funktionen und Parametern

### Zeiger als Parameter

Wir haben die Übergabetechnik *call by value* bisher nur auf den Standard-Datentypen angewandt. Sie wirkt aber in gleicher Weise auch bei Zeigern: Beim Aufruf einer Funktion werden die Parameter mit den Werten der Argumente initialisiert.

Allerdings gibt es nun eine Besonderheit. Bei korrekter Anwendung beinhaltet ein Zeiger-Parameter nach der Parameterübergabe eine Adresse aus der Umgebung des aufrufenden Ausdrucks. Durch Dereferenzierung dieses Parameters kann man auf das angezeigte Objekt zugreifen. War dies zum Beispiel eine Variable, so kann man damit auch diese Variable ändern.

Wir verdeutlichen dieses Vorgehen am Beispiel der Parameterübergabe aus dem Kapitel 8 und verändern zunächst die Funktion `belege` aus dem dortigen Beispiel.

```
Bspl.: void belege (double *x)    x ist diesmal ein Zeiger.
      {
          *x = 55.5;              x wird dereferenziert.
      }
```

Die Wertzuweisung ändert jetzt nicht mehr den Wert des Parameters selbst, sondern den Wert des vom Zeiger-Parameter angezeigten Objekts.

Wird diese Funktion nun stimmig aufgerufen, so ändert sie tatsächlich eine Variable aus der aufrufenden Funktion. Dazu muss als Argument die Adresse der Variablen übergeben werden.

```
Bspl.: double z = 0.0;
      double *pz;

      pz = &z;

      belege (pz);                Aufruf mit Zeiger auf z als Argument
      belege (&z);                Aufruf mit Adressoperator
```

Beide Funktionsaufrufe sind gleichwertig, denn übergeben wird jeweils die Adresse von `z` als Wert für den Zeiger-Parameter `x` von `belege`. Folglich zeigt dieser jetzt auch auf `z`. Die Wertzuweisung in `belege` ändert `z`.

Resultat: Der Wert der Variablen `z` wird durch die Funktion `belege` auf den Wert 55.5 geändert.

So haben wir nun ein Verfahren, mit dem wir bei einem Funktionsaufruf eine Variable aus der Aufrufumgebung per Parameterübergabe ändern können:

- Der Parameter wird als Zeiger des entsprechenden Typs vereinbart.
- Der Parameter wird in der Funktion zur Nutzung dereferenziert, wenn auf ein angezeigtes Objekt aus der Aufrufumgebung zugegriffen werden soll. Insbesondere dann natürlich, wenn dieses geändert werden soll.
- Das Argument muss eine Adresse liefern, typischerweise durch Anwendung des Adressoperators auf eine Variable.

Unter der Übergabetechnik **call by reference** versteht man eine Parameterübergabe, bei der nicht der Wert eines Objekts, sondern nur ein Verweis (Referenz) auf das Objekt übergeben wird. Über diese Referenz lässt sich dann ein variables Argument auslesen und auch ändern.

Viele Programmiersprachen, z.B. C++ und Java, erlauben diese Technik als Alternative zum *call by value*. Dabei braucht nur der Parameter formal entsprechend spezifiziert werden. Die weiteren Vorgänge (Dereferenzierung und Bewertung der Argumente) laufen automatisiert ab. In diesem engeren Sinne verfügt C über keine vergleichbare Übergabetechnik.

Allerdings können wir diese Technik mit Zeigern nachstellen. Sofern wir das entsprechende Vorgehen konsequent beachten (Zeiger - Dereferenzierung - Adressoperator), steht uns ein gleichartiges Verfahren zur Verfügung. Diese Vorgehensweise wollen wir für unsere Zwecke ebenso unter dem Begriff *call by reference* zusammenfassen.

Bspl.: Hier noch einmal das Beispiel aus Kapitel 8, jetzt komplett auf **call by reference** angepasst.

```
#include <stdio.h>

/* Prototyp */
void belege (double*);

int main ()
{
    double z = 0.0;
    belege (&z);
    printf ("Wert z = %f\n", z);
    return 0;
}

void belege (double *x)
{
    *x = 55.5;
    printf ("Wert *x = %f\n", *x);
}
```

z wird neu belegt!  
Ausgabe: Wert z = 55.500000  
Übergabe: x erhält die Adresse von z.

### Funktionen mit Adressen als Ergebnis

Es ist möglich Funktionen zu vereinbaren, die als Ergebnis einen Zeigertyp haben. Natürlich muss dann über eine `return`-Anweisung im Funktionsrumpf auch ein entsprechendes Ergebnis vom Zeigertyp bestimmt werden. Ansonsten gelten die bisherigen Regeln.

```
Bspl.: int *minimum (int *a , int *b)
{
    if (*a < *b)
        return a;
    else
        return b;
}
```

Die Funktion soll die **Adresse** eines Minimums der angezeigten `int`-Objekte berechnen.  
Daher wird in den `return`-Anweisungen auch **nicht** dereferenziert.

Es liegt kein Verfahren *call by reference* zur Parameterübergabe vor; hier soll vielmehr von der Aufgabe her tatsächlich mit Zeigern (Parameter und Funktionsergebnis) operiert werden.



# 11. Arrays

## Allgemeines zu Datenstrukturen

Ein Datenobjekt, das aus mehreren **Komponenten** (oder **Elementen**) besteht, nennt man zusammengesetzt, im Gegensatz zu elementaren Daten wie zum Beispiel bei den Standard-Datentypen. Müssen alle Komponenten dabei vom selben **Basistyp** sein, so heißt die Datenstruktur des Objekts **homogen**, ansonsten **heterogen**.

Im Allgemeinen besitzen die Komponenten eines zusammengesetzten Objekts eine Beziehung zueinander, eine Struktur. Diese ist entweder **statisch** vorgegeben oder sie wird im Programm erst erstellt und kann auch verändert werden, d.h. sie ist **dynamisch**.

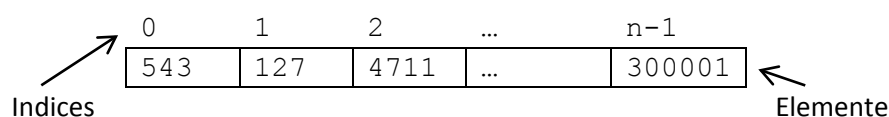
Zu jeder Datenstruktur muss es geeignete **Zugriffsmechanismen** geben, die den Zugriff auf einzelne Komponenten erlauben. Ist der Zugriff auf jede Komponente jederzeit möglich, so bezeichnen wir dies als **wahlfreien Zugriff** (*random access*).

## Eindimensionale Arrays

Ein Array besitzt eine homogene, statische Datenstruktur. Der Basistyp der Elemente kann ein Standard-Datentyp, ein Zeigertyp, aber auch einer derjenigen Typen sein, die wir in den kommenden Kapiteln noch behandeln. Der Zugriff auf die Elemente erfolgt durch **Indizierung**. Besitzt ein Array  $n$  Elemente, so erfolgt der Zugriff über die Indices von 0 bis  $n-1$ .

Wir bevorzugen den englischen Begriff *array*, weil die deutschen Übersetzungen (z.B. Feld, Reihe, Vektor, ...) unterschwellig Dimensionen nahelegen (Feld = zweidimensional) oder sich mit Fachbegriffen (Reihe in der Analysis, Vektor als Element eines Vektorraums) nur unzureichend treffen.

Die Elemente eines Arrays liegen im Speicher dicht zusammen. So kann man sich ein Array der Länge  $n$  vom Basistyp `int` vorstellen:



## Vereinbarung (variable Elemente)

**Form:** *Speicherklassen-Attribut*<sub>opt</sub> *Basistyp* *Name*<sub>1</sub> [ *Länge*<sub>1</sub> ] , *Name*<sub>2</sub> [ *Länge*<sub>2</sub> ] ... ;

Dabei gilt:

- Das *Speicherklassen-Attribut* `register` ist nicht statthaft; ansonsten gelten die Aussagen des Kapitels 9. Nur Arrays der statischen Speicherklasse werden implizit mit Null initialisiert.
- In älteren Standards muss eine *Länge* stets ein konstanter Ausdruck sein, von ganzzahligem Typ mit positivem Wert. Der Standard C99 erlaubt jedoch bei Arrays der automatischen Speicherklasse für die *Längen* auch Ausdrücke, die variabel sein dürfen. Für C11 gilt dies gegebenenfalls auch; man beachte jedoch die Einschränkung aus dem Anhang F.

Ein derartig vereinbartes Array ist variabel in dem Sinne, dass die einzelnen Elemente des Arrays veränderliche Objekte sind. Die Länge eines Arrays ist hingegen nach der Vereinbarung während der Lebensdauer des Arrays nicht mehr zu ändern; auch in C99/C11 bleibt die Datenstruktur statisch. Größendynamische Datenstrukturen, die diese Restriktion umgehen, lernen wir später noch kennen.

Bspl.: `double alpha[30], bravo[1000];`      zwei Arrays unterschiedlicher Länge

Bspl.: Im Standard C99 darf die Länge von Arrays der automatischen Speicherklasse insbesondere von Variablen, symbolischen Konstanten oder auch Parametern abhängen.

```
void func (int n)
{
    double charlie[n+1];
    ...
}
```

Array mit variabel vereinbarter Länge

## Zugriff

Der Zugriff auf ein Element eines Arrays erfolgt nun durch Indizierung (L- und R-Wert).

**Form:** *Arrayname* [ *Index* ]

Der *Index* kann ein beliebiger ganzzahliger Ausdruck sein, insbesondere also eine `int`-Variable oder eine Konstante (Literal), mit Werten zwischen 0 und *Länge*−1 des Arrays.

```
Bspl.: double delta[10], echo[10];
int i;

for (i = 0; i < 10; i += 2)
    delta[i] = 1.0;
for (i = 1; i < 10; i += 2)
    delta[i] = 0.1;

for (i = 0; i < 10; i++)
    printf ("%d: %f\n", i, delta[i]);

for (i = 0; i < 10; i++)
    echo[i] = delta[i];
```

Die 10 Elemente des Arrays `delta` werden der Reihe nach belegt, an gerader Position mit 1.0, an ungerader Position mit 0.1.

Index und Wert der Elemente werden der Reihe nach ausgegeben.

Die Elemente werden der Reihe nach in das Array `echo` kopiert.

Eine direkte Wertzuweisung zwischen Arrays ist nicht statthaft. Eine Kopie muss wie im vorgehenden Beispiel immer elementweise erstellt werden.

## Wichtig!

In C wird die Einhaltung der Indexgrenzen nicht überwacht. Dies bedeutet, dass über den vereinbarten Indexbereich hinweg auf den Speicher zugegriffen werden kann. Welche Wirkung dies hat, kann nicht vorhergesagt werden. Eventuell greift man damit auf den Speicher anderer Datenobjekte zu; gegebenenfalls wird auch das Programm vom Betriebssystem abgebrochen. Die Verantwortung liegt beim Programmierer.

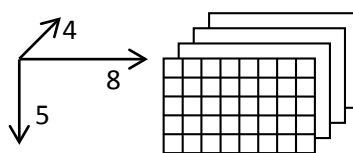
So ist mit obigem Beispiel die Wirkung eines Zugriffs auf ein vermeintliches Element `delta[10]` nicht kalkulierbar.

## Mehrdimensionale Arrays

C kennt auch Arrays in mehreren Dimensionen, also mit mehrfacher Indizierung. Diese Aussage ist nicht so ganz wahr. So ist z.B. ein zweidimensionales Array in Wirklichkeit ein eindimensionales Array, bei dem die Elemente wiederum Arrays sind. Diese Konstruktion setzt sich bei höheren Dimensionen entsprechend fort.

Bspl.: `int quader[4][5][8];` erzeugt ein rechteckiges Gebilde für 160 `int`-Elemente

Gut trifft hier die Vorstellung von 4 Matrizen zu, die in dieser Skizze hintereinander angeordnet werden. Jede Matrix wiederum besteht aus 5 Zeilen der Länge 8.



`quader[2]`

die dritte 5x8-Matrix in diesem Gebilde

`quader[2][3]`

die vierte Zeile der Länge 8 in der dritten Matrix

`quader[2][3][0]`

der erste `int`-Wert gerade dieser Zeile

Die in einigen Programmiersprachen übliche Notation, bei der mehrere Indices in nur einem Klammerpaar angegeben sind, ist in C nicht möglich: `quader[4, 5, 8]` ist syntaktisch falsch.

## Initialisierung durch Aggregate

Die Elemente eines Arrays können bei der Vereinbarung durch Angabe einer Werteliste in geschweiften Klammern<sup>1</sup> `{ }` explizit initialisiert werden. Eine Ausnahme bilden jedoch Arrays mit variabler Längenangabe gemäß Standard C99/C11; diese können nicht initialisiert werden.

In den Wertelisten (Aggregaten) müssen konstante Ausdrücke stehen, konvertierbar zum Basistyp. Im Normalfall sind es meist Konstanten (Literele). Es dürfen nicht mehr Werte in der Liste sein, als die Länge des Arrays vorgibt. Ist die Liste zu kurz, so werden überzählige Elemente mit Null initialisiert.

Die Werteliste kann bei mehrdimensionalen Arrays in linearer Form<sup>2</sup> oder entsprechend der Dimensionierung in geschachtelter Form angegeben werden. Folgendes Beispiel zeigt die Details.

Bspl.: `int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};`<sup>2</sup>

```
int b[3][3] = {{1, 2, 3},
               {4, 5, 6},
               {7, 8, 9}};
```

`int c[3][3] = {1, 2, 3};`<sup>2</sup>

`int d[3][3] = {{1}, {4}, {7}};`

Ergebnis:

1	2	3
4	5	6
7	8	9

explizit nur für die erste Zeile

explizit nur für die erste Spalte

(Restliche Elemente jeweils mit 0.)

<sup>1</sup> Es handelt sich hierbei nicht um einen Block.

<sup>2</sup> Einige Systeme erzeugen dazu allerdings Warnungen.

Es gibt noch einen Sonderfall. Fehlt die Angabe zur ersten Array-Länge, so wird die Länge dieser Dimension durch die Werteliste dem Bedarf entsprechend automatisch bestimmt.

Bspl.: `int e[][3] = {{1},{4},{7}};`

Das Aggregat legt die hier fehlende Länge 3 für die erste Dimension fest.

## Konstante Arrays

Arrays können auch konstant sein, d.h. die Elemente des Arrays lassen sich nach der Initialisierung nicht mehr verändern. Ein solches Array muss natürlich initialisiert werden.

Bspl.: `const int f[] = {1, 2, 3};`

## Mehr zu Funktionen und Parametern

Arrays können als Argumente übergeben werden. Allerdings wird nicht wirklich das Array mit seinen Elementen selbst, sondern „nur“ ein Verweis auf das Array übergeben (*call by reference*). Die genauen Zusammenhänge untersuchen wir dazu im nächsten Kapitel. Hier betrachten wir mehr die praktische Handhabung.

### Array-Parameter

Ein Array-Parameter wird natürlich in der Parameterliste einer Funktion vereinbart.

**Form:** *Basistyp Name* [ *Länge<sub>1</sub>* ] [ *Länge<sub>2</sub>* ] ...

Dabei gilt:

- Einige Angaben sind optional und können entfallen:
  - im Prototypen der *Name* des Parameters, jedoch nicht in jedem Fall;
  - im Prototypen und in der Funktionsdefinition die *Länge<sub>1</sub>* der ersten (links stehenden) Dimension; die Klammern müssen aber stehen bleiben.
- Der Basistyp kann jeder Basistyp für Reihen sein. Insbesondere sind auch Zeigertypen und konstante Typen zulässig (Attribut `const`).
- Die angegebenen *Längen* müssen Ausdrücke von ganzzahligem Typ und positivem Wert sein.
  - In älteren Standards muss jede *Länge* ein konstanter Ausdruck sein.
  - Variable Formen sind erst ab dem Standard C99 möglich. So kann eine *Länge* ein Ausdruck mit zuvor vereinbarten Parametern oder aber mit globalen Variablen sein.

Im Funktionsrumpf wirkt der Parameter dann wie ein entsprechend lokal vereinbartes Array. Eine konkrete Länge für die erste Dimension ist dabei – als besondere Ausnahme – nicht vorgegeben.

### Arrays als Argumente

Bei einem Funktionsaufruf darf der Name eines Arrays als Argument angegeben werden. Dieses Argument ist verträglich zu einem Array-Parameter, wenn beider Basistyp und alle Details der Dimensionierung – mit Ausnahme der „offenen“ Länge in der ersten Dimension – übereinstimmen.

Die Typkonvertierungen des Kapitels 3 sind nicht anwendbar.

Ist der Parameter als *variables Array* vereinbart, so wirkt sich jede Veränderung eines Elements des Parameters direkt auf das Argument aus, d.h. die Elemente des Arguments können durch die Funktion im Wert verändert werden. Argumente dürfen daher auch nur *variable Arrays* sein.

Ist der Parameter als *konstantes Array* vereinbart, so kann die Funktion per Parameter auf die Elemente des Arguments nur lesend zugreifen. Argumente können konstante Arrays sein, aber in spezifischen Fällen, die jetzt nicht vorgestellt werden, auch *variable Arrays*. Änderungen der Werte sind über den Parameter keinesfalls möglich.

### **Wichtig!**

C-Systeme überprüfen nicht zwingend die Dimensionen und Längen von Parametern und Argumenten auf Übereinstimmung. Da ohnehin immer die Länge der ersten Dimension eines Array-Parameters „offen“ ist, besteht stets die Gefahr, dass das Argument weniger Elemente enthält als angenommen. So fällt man unter Umständen über eine ungeeignete Indizierung des Parameters aus dem zulässigen Speicherbereich heraus. Es gibt auch keine Möglichkeit, in einer Funktion die tatsächliche Länge eines Array-Parameters zu bestimmen. Eine programmiertechnische Abhilfe kann darin bestehen, die Länge der ersten Dimension als zusätzlichen Parameter zu übergeben. Allerdings ist weiterhin der Programmierer dafür verantwortlich, diese Angabe dann auch konsequent zu beachten.

**Bspl.:** Ausgabe eines zweidimensionalen Arrays

```
#include <stdio.h>

/* Prototypen */
void show (int p[3][3]);           Funktionsdeklaration,
void show (int p[][3]);           jetzt ohne erste Längenangabe,
void show (int[][3]);             jetzt ohne Parametername

int main ()
{
    int b[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

    show (b);                      Argument ist das Array b.
    return 0;
}

void show (int p[][3])             Funktionsdefinition,
{                                  optional ohne erste Längenangabe
    int i, j;
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            printf ("%d,%d] -> %d  ", i, j, p[i][j]);
        printf ("\n");
    }
}
```

Im Kontext dieses Beispiels bezeichnen dann *b* und *p* das gleiche Array.

Bspl.: So lässt sich obige Funktion ab dem Standard C99 auf eine variable Array-Länge, speziell in der zweiten Dimension, umschreiben. Die Länge der ersten Dimension ist ohnehin „offen“.

```
void show (int lng, int p[][lng])
{
    for (int i = 0; i < lng; i++)
    {
        for (int j = 0; j < lng; j++)
            printf ("%d,%d -> %d ", i, j, p[i][j]);
        printf ("\n");
    }
}
```

Im Prototypen der umgeschriebenen Funktion darf der für die Länge vergebene Name nicht entfallen:

```
void show (int lng, int[][lng]);
```

Beim Aufruf dieser Funktion ist dann ein Argument mehr anzugeben:

```
show (3, b);
```

Soll man im Prototypen Parameternamen mit angeben oder nicht? Wir sind der Meinung, dass hier nur aussagekräftige Namen im Sinne einer Dokumentation etwas zu suchen haben. Die Namen selbst sind ohne weitere Bedeutung und können sogar von der Funktionsdefinition abweichen. – Die Situation ändert sich allerdings, wenn man wie im vorausgehenden Beispiel mit variablen Längen bei Array-Parametern arbeiten möchte.

### Funktionen mit Arrays als Ergebnis

Eine Funktion kann syntaktisch kein Array als Ergebnis zurückliefern. Diese Einschränkung ist jedoch von geringer Bedeutung. Einmal, so haben wir gerade gesehen, können Arrays auch als Ergebnisparameter über *call by reference* ausgeliefert werden. Ferner werden wir im nächsten Kapitel sehen, wie man mit Zeigern zu Arrays umgeht. Zeigertypen sind als Funktionsergebnis zulässig.

## 12. Zeiger und Arrays

In diesem Kapitel stellen wir einen Bezug zwischen Zeigern und Arrays her, der auf einem grundlegenden Prinzip von C beruht. Hierdurch ergeben sich neue Möglichkeiten; bekannte Dinge aus den Kapiteln 10 und 11 erscheinen in einem neuen Licht.

### Array- und Zeiger-Notation

Ausgangssituation (eindimensional)

**Ein Array steht in Ausdrücken für die Adresse des ersten Elements des Arrays und wirkt damit wie ein konstanter Zeiger auf das erste Element.**

Für jedes beliebige Array `a` gilt:

$$a \equiv \&a[0]$$

Mit dem Symbol  $\equiv$  wollen wir in diesem Kapitel illustrieren, dass beide Seiten gleichwertig und syntaktisch ersetzbar sind. In einem C-Ausdruck kann eine derartige Ersetzung jederzeit durchgeführt werden, sofern dies im Einklang zu den übrigen Regeln steht. Zu beachten sind dabei die Bestimmungen zur Verwendung von L- bzw. R-Werten sowie die Prioritätsregeln. Bei einer konkreten Umsetzung kann möglicherweise eine zusätzliche Klammerung erforderlich werden.

Ist das Array konstant, so bleibt diese Eigenschaft natürlich erhalten, d.h. die Wirkung entspricht einem konstanten Zeiger auf ein diesmal konstantes Element.

Bspl.: `int a[5];`

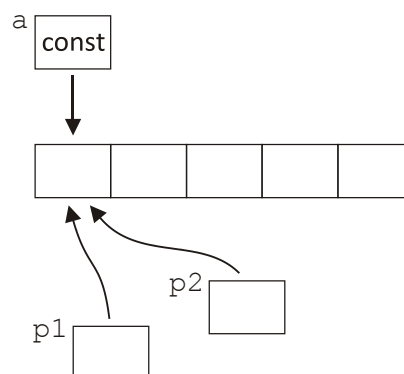
```
int *p1 = &a[0];
```

```
int *p2 = a;
```

Der Zeiger `p1` zeigt auf erstes Element des Arrays.

`p2` ebenso!

Die Situation dieses ersten Beispiels wollen wir grafisch wiedergeben. Dabei soll modellhaft unter dem Array-Namen `a` ein konstanter Zeiger auf den Anfang des Arrays eingezeichnet werden, um die Wirkung des Array-Namens im obigen Sinne zu veranschaulichen. Auf Indizierung und Inhalte der 5 Elemente verzichten wir.



Selbstverständlich können wir jetzt durch Dereferenzierung von `a`, `p1` oder `p2` auf das erste Array-Element zugreifen.

### Adressarithmetik auf Arrays

Da laut Kapitel 10 die Addition und Subtraktion für Zeiger in Einheiten der Speichergröße des zugrundeliegenden Basistyps erfolgt, ergibt sich nun zwangsläufig, dass  $a+1$  rechnerisch die Adresse des Elements  $a[1]$  sein muss, denn dieses `int`-Element folgt  $a[0]$  unmittelbar. Setzt man die Überlegung fort, erkennt man, dass allgemein jedes Element  $a[i]$  an der Adresse  $a+i$  liegt.

Für jedes beliebige Array  $a$  und einen Index  $i$  gilt somit:

$$a+i \equiv \&a[i]$$

### Zugriff auf Array-Elemente

Liefert uns  $a$  die Adresse des ersten Array-Elements, so liefert  $*a$  durch eine Dereferenzierung das erste Array-Element  $a[0]$  selbst:

$$a[0] \equiv *a$$

Wie greifen wir aber nun beispielsweise auf das Array-Element mit dem Index 2 zu?

- Laut Kapitel 11 natürlich wie bisher über eine Indizierung:  $a[2]$
- Mittels der Adressarithmetik können wir aber auch zu  $a$  als Zeiger den Wert 2 hinzuaddieren und damit die Adresse des Elements zwei Plätze weiter im Array berechnen. Dann führen wir noch eine Dereferenzierung aus:  $*(a+2)$

Allgemein gilt also für ein beliebiges Array  $a$  und einen Index  $i$  folgende Ersetzungsformel:

$$a[i] \equiv *(a+i)$$

Dies sagt in knapper Form aus, dass ein Ausdruck aus Array und Index äquivalent zu einem aus Zeiger und Abstand ist. Die Klammerung  $()$  ist aufgrund der Priorität von  $*$  vor  $+$  zwingend erforderlich<sup>1</sup>.

### Mehrdimensionale Arrays

Mehrdimensionale Arrays sind in C eindimensionale Arrays aus Elementen, die selbst wieder Arrays mit einer um 1 verminderten Dimensionalität sind. Entsprechend können die Ersetzungsregeln auch auf jede einzelne Dimension angewandt werden.

Die beiden folgenden Beispiele erläutern, wie sich die Ersetzungsregel  $a \equiv \&a[0]$  jeweils auf die beiden Dimensionen eines zweidimensionalen Arrays (hier:  $b$ ) auswirkt.

Bspl.: `int b[3][5];`

Diese Matrix besteht aus 3 Zeilen zu je 5 Elementen.

Damit lauten jetzt die Ersetzungsregeln:

$$b \equiv \&b[0]$$

$b$  wirkt wie ein konstanter Zeiger auf die erste Zeile der Matrix.

$$b[i] \equiv \&b[i][0]$$

$b[i]$  wirkt wie ein konstanter Zeiger auf das erste Element der Zeile mit dem Index  $i$ .

<sup>1</sup>  $*a+i$  würde die Summe von  $a[0]$  und  $i$  berechnen.



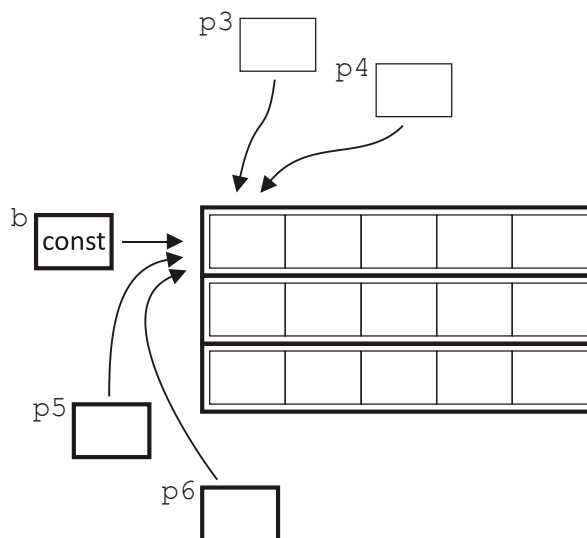
Wir haben grau hinterlegt, welche Operationen aufgrund der ursprünglichen Ersetzungsregel zusammen eliminiert werden.

In der direkt vorausgegangenen Ersetzung sind die Prioritätsregeln zu beachten; hier ist die zweite Indizierung und die nachfolgend auszuführende Adressoperation betroffen. Der davon eingeschlossene Ausdruck `b[i]` bestimmt das (eindimensionale) Array, auf welchem die Ersetzungsregel angewandt wird.

Bspl.: `int *p3 = &b[0][0];`    `p3 und p4 sind Zeiger auf int-Elemente.`  
`int *p4 = b[0];`  
`int (*p5)[5] = &b[0];`    `p5 und p6 sind Zeiger auf Zeilen vom Typ int[5].`  
`int (*p6)[5] = b;`

Neue Syntax: Die Form der Vereinbarungen zu `p5` und `p6` werden wir gleich in einem weiterführenden Beispiel (beta) noch einmal ausführlicher beleuchten.

Auch hier eine grafische Wiedergabe. Der Name `b` taucht mit einem konstanten Zeiger auf, um auch hier die Wirkung des Array-Namens zu verdeutlichen. Das zweidimensionale Array haben wir dazu optisch mit einer Zeilenstruktur versehen. Fett umrahmte Zeiger verweisen auf Zeilen der Matrix, andere Zeiger auf `int`-Elemente.



Dereferenzierung von `p3` und `p4` liefert das erste `int`-Element der Matrix, Dereferenzierung von `b`, `p5` und `p6` hingegen die erste Zeile der Matrix.

Auch die weiteren Ersetzungsregeln lassen sich bedeutungsgleich auf mehrdimensionale Arrays übertragen. Die nachfolgenden Beispiele demonstrieren im zweidimensionalen Fall die Anwendung der Regel `a[i] ≡ *(a+i)` in beiden Dimensionen.

Man beachte, dass die Adressarithmetik nicht allgemein in Einheiten bezogen auf den Basistyp des Arrays durchgeführt wird, sondern sich in höheren Dimensionen auf den konkreten Typ des angezeigten (Teil-)Arrays bezieht. Andernfalls wäre eine Ersetzung in dieser formalen Art nicht durchgängig möglich.

Bspl.: So kann man auf die **Elemente** der zuvor vereinbarten Matrix  $b$  zugreifen.

$$b[i][j] \equiv (* (b+i)) [j] \equiv * (b[i]+j) \equiv * (* (b+i) + j)$$

Adressarithmetik bezogen auf Zeilen
Adressarithmetik bezogen auf Elemente (Basistyp)

Bspl.:  $b[i] \equiv * (b+i)$

So erfolgt der Zugriff auf die **Zeile** mit Index  $i$ .

### Indizierung von Zeigern

Wir haben bisher gesehen, wie man auf Array-Elemente mittels Zeigern zugreifen kann. Die Kehrseite dieser Medaille ist, dass man auch jeden Zeiger jederzeit mit einem Index versehen kann.

Für einen beliebigen Zeiger  $p$  und einen Index  $i$  gilt:

$$p[i] \equiv * (p+i)$$

Ob das algorithmisch sinnvoll ist, muss der Programmierer entscheiden. – Zeigt  $p$  beispielsweise auf den Anfang eines Arrays  $a$ , so gilt für jeden Index  $i$ , dass  $p[i]$  identisch zu  $a[i]$  ist.

Aus den obigen Regeln lassen sich nun in Umkehrung alle weiteren Ersetzungsregeln herleiten, zum Beispiel dass die Adresse von  $p[i]$  gleich  $p+i$  sein muss.

Wir fassen alle Ersetzungsregeln daher noch einmal tabellarisch zusammen.<sup>2</sup>

Array-Notation $\equiv$ Zeiger-Notation
$\&x[0] \equiv x$
$\&x[i] \equiv x+i$
$x[0] \equiv *x$
$x[i] \equiv *(x+i)$

- Die Tabelle gilt für beliebige Arrays  $x$  wie auch für beliebige Zeiger  $x$  und einen Index  $i$ .
- Die syntaktischen Regeln bestimmen, dass die Ausdrücke  $\&x[0]$ ,  $\&x[i]$  und  $x+i$  nur als R-Werte verwendet werden dürfen.<sup>3</sup> Die freie Nutzung der Ersetzungsregeln ist daher in diesen Fällen dementsprechend eingeschränkt.
- Die Prioritätsregeln für Ausdrücke gelten weiterhin; bei Anwendung der Ersetzungsregeln kann eine zusätzliche Klammerung erforderlich werden.
- Bei mehrdimensionalen Arrays lassen sich die Ersetzungsregeln in den jeweiligen Dimensionen einzeln anwenden. Die Regeln gelten analog für Zeiger auf Zeiger.

Ein nachfolgendes Beispiel (`delta`) demonstriert im zweidimensionalen Fall die Nutzung von Zeigern auf Zeiger zusammen mit einer doppelten Indizierung.

<sup>2</sup> Mit dem Wert Null für  $i$  ergeben sich die zwei Regeln 1 und 3 aus den Regeln 2 und 4.

<sup>3</sup> Nur Variablen – sowie entsprechende Array-Elemente (Indizierung), angezeigte Objekte (Dereferenzierung) und Struktur-Komponenten (Punkt- und Pfeil-Auswahloperationen aus Kapitel 13) – können L-Werte liefern.

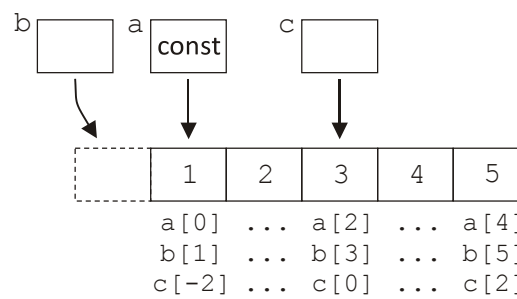
Bspl.: Konstruktion eines Index-Shifts

```
int a[] = {1, 2, 3, 4, 5};
int *b = a - 1;
int *c = a + 2;
```

`b` zeigt rechnerisch auf den Speicherplatz direkt vor dem Array `a` und somit verweist `b+1` auf das erste Array-Element selbst. Daher greift `b[1] ≡ *(b+1)` auf das erste Element `a[0]` zu und so weiter. Wir können folglich die fünf Array-Elemente auch als `b[1]` bis `b[5]` ansprechen und haben derart eine Indizierung eingeführt, die bei 1 und nicht Null beginnt.

`c` zeigt auf das Array-Element `a[2]`. Damit können wir auf dieses Element auch mittels `c[0] ≡ *c` zugreifen. Für das erste Array-Element müssen wir den Index um 2 verringern, d.h. das Element `a[0]` entspricht `c[-2] ≡ *(c-2)`. Mithin kann man auf die fünf Array-Elemente ebenso über `c[-2]` bis `c[2]` zugreifen.

Die nachfolgende Skizze zeigt die zulässigen Indexbereiche für die jeweiligen Zeiger.



## Beispiel: Zweidimensionale Datenstrukturen

In diesem Beispiel wollen wir – ausgehend von einem zweidimensionalen Array – ein oder zwei Array-Dimensionen durch Zeiger ersetzen.

Bspl.: <code>double alpha[3][5];</code>	ein zweidimensionales Array
<code>double (*beta)[5];</code>	ein Zeiger auf ein 5-elementiges Array
<code>double *gamma[3];</code>	ein 3-elementiges Array von Zeigern
<code>double **delta;</code>	ein Zeiger auf Zeiger

In allen vier Fällen sind z.B. die Notationen `alpha[1][3]`, `beta[1][3]`, `gamma[1][3]` und `delta[1][3]` syntaktisch zulässig, sofern die vorhandenen Zeiger tatsächlich entsprechend belegt worden sind.

### alpha und beta

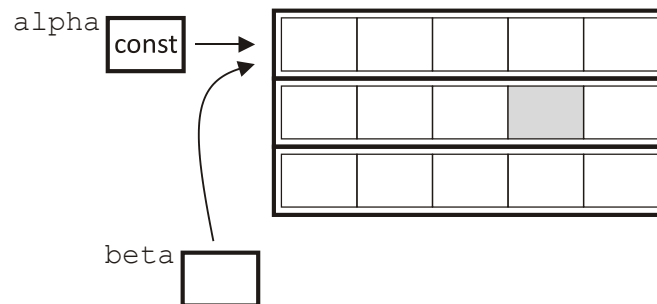
`alpha` ist ein zweidimensionales Array, für das 3x5 Speicherplätze bereitgestellt werden. Ferner wirkt der Name `alpha` wie ein Zeiger auf die erste Zeile aus 5 `double`-Elementen. – `beta` ist aber in dieser Form gerade auch als ein Zeiger auf `double[5]` vereinbart.

Folglich sind `beta` und `alpha` als Zeiger typgleich, mit der Einschränkung, dass `alpha` als Array einen konstanten Zeiger darstellt.

Die Wertzuweisung

```
beta = alpha;
```

ist somit nur in dieser Richtung zulässig.



Die zweite Zeile in der Matrix heißt nun `alpha[1]` oder `beta[1]` oder `*(alpha+1)` oder `*(beta+1)`. Die Adresse der zweiten Zeile wird dabei **berechnet**, letztendlich als Summe der Anfangsadresse des Arrays plus der Zeilenlänge (im Sinne der Adressarithmetik).

Das grau markierte Element kann in Array-Notation bestimmt werden als `alpha[1][3]` oder `beta[1][3]`.

### gamma und delta

Für das Zeiger-Array `gamma` werden drei Zeiger vom Typ `double*` angelegt. Ferner wirkt der Name `gamma` wie ein Zeiger auf den ersten dieser drei Zeiger. – `delta` ist jedoch auch ein Zeiger auf Zeiger; der Basistyp `double` ist ebenfalls gleich.

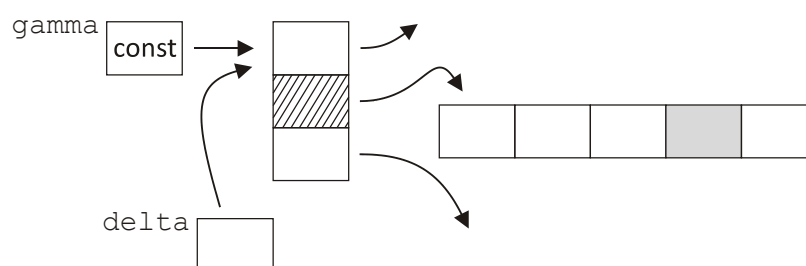
Folglich sind `delta` und `gamma` als Zeiger typgleich, mit der Einschränkung, dass `gamma` als Array einen konstanten Zeiger darstellt.

Die Wertzuweisung

```
delta = gamma;
```

ist somit nur in dieser Richtung zulässig.

In der nachfolgenden Grafik haben wir noch zusätzlich den zweiten Zeiger aus dem Zeiger-Array<sup>4</sup> exemplarisch auf ein `double`-Array der Länge 5 zeigen lassen. Dieser Vorgang ist nicht Bestandteil der bisherigen Vereinbarungen und Anweisungen.



<sup>4</sup> Die Elemente haben wir diesmal im Gegensatz zu unseren bisherigen Skizzen vertikal angeordnet.

Das schraffierte Element im Zeiger-Array heißt nun `gamma[1]` oder `delta[1]` oder `*(gamma+1)` oder `*(delta+1)`. Dort ist die Anfangsadresse des `double`-Arrays **abgespeichert**, welches wir als zweite Zeile ansehen.

Das grau markierte Element kann in Array-Notation bestimmt werden als `gamma[1][3]` oder `delta[1][3]`.

### Zusammenfassung

Die obigen Vereinbarungen ermöglichen eine zweidimensionale Datenstruktur mit entsprechender Indizierung. Bei allen Gemeinsamkeiten dieser Beispiele gibt es aber doch bedeutende Unterschiede.

Auf eine konstruktive Abweichung haben wir schon hingewiesen: Bei `alpha/beta` werden die Adressen der Zeilen (statisch) berechnet, bei `gamma/delta` sind diese Adressen in den variablen Elementen des Zeiger-Arrays abgespeichert.

Nur bei einem Array werden tatsächlich auch Elemente angelegt. Alle Zeiger hingegen müssen noch belegt werden, d.h. auf den Anfang eines Arrays richtigen Typs und ausreichender Länge verweisen. Solche Arrays müssen gegebenenfalls zu den Beispielen auch noch vereinbart werden.

Diese Umstände zusammen eröffnen neue Möglichkeiten: Da die „echten“ Zeiger (hier: `beta`, `delta` und die drei Elemente im Zeiger-Array `gamma`) variabel sind, können diese während des Programmlaufs auch tatsächlich verändert werden. Es ist von besonderem Interesse, dies im Zusammenspiel mit dem dynamischen Speicher (Kapitel 15) auszunutzen, da es diese Zeiger dann ermöglichen, unterschiedliche und flexible Dimensionslängen einzuführen.

Diese Übersicht zeigt die Möglichkeiten.

Bspl.: <code>double alpha[3][5];</code>	statische Datenstruktur
<code>double (*beta)[5];</code>	größendynamisch in 1. Dimension
<code>double *gamma[3];</code>	größendynamisch in 2. Dimension
<code>double **delta;</code>	größendynamisch

Wir setzen die Überlegungen fort im Kapitel 15. Dort werden wir zum Zeiger `delta` eine komplette dynamische Datenstruktur erzeugen, die eine Dreiecksmatrix bildet.

Die Datenstrukturen unterscheiden sich noch in einem weiteren Punkt. Arrays, insbesondere auch mehrdimensionale, nehmen immer einen zusammenhängenden Speicherbereich in Anspruch. So könnte man einen Zeiger vom Basistyp auf das erste Element verweisen lassen und diesen durch fortlaufende Inkrementierung durch das gesamte (auch mehrdimensionale) Array laufen lassen. Im Beispiel `gamma/delta` wird dieses Kriterium im Regelfall nicht erfüllt sein.

## Parameter und Argumente

Aus der Ausgangssituation dieses Kapitels resultiert eine weitere Konsequenz für Array-Argumente. Der Aufruf einer Funktion `func (a)` mit einem Array-Argument `a` hat die gleiche Wirkung wie der Aufruf `func (&a[0])`. Übergeben wird also eine Adresse.

**Tatsächlich ist ein Array-Parameter, wie wir ihn im Kapitel 11 eingeführt haben, ein Zeiger.**

Nun gibt es einen interessanten Effekt: Durch Umsetzung der Array-Notation in eine Zeiger-Notation gleicht die Vorgehensweise im Kapitel 11 dem Verfahren, welches wir auch in Kapitel 10 vorgestellt haben (*call by reference*). Die nachfolgende Tabelle demonstriert dies im eindimensionalen Fall.

<i>call by reference</i>	Array-Notation	≡	Zeiger-Notation
Funktionskopf/Parameter	<code>func (int x[])</code>	≡	<code>func (int *x)</code>
Zugriff im Funktionsblock	<code>x[i]</code>	≡	<code>*(x+i)</code>
Aufruf/Array-Argument	<code>func (a)</code>	≡	<code>func (&amp;a[0])</code>
Vergleiche mit ...	Kapitel 11		Kapitel 10

Dem Zeiger-Parameter rechts in der Tabelle kann man nicht unmittelbar ansehen, ob er für ein einzelnes Objekt oder den Beginn eines Arrays solcher Objekte vorgesehen ist. Ferner bleibt dabei die Länge eines Arrays unspezifiziert.

Genau dies deckt sich mit der „offenen“ Länge in der ersten Dimension des Array-Parameters links und der fehlenden Funktionalität zur Bestimmung dieser Größe in der Funktion.

Ob es sinnvoll ist, die Parameter innerhalb der Funktion zu indizieren und, falls ja, in welchem Indexbereich, hängt von einem geeigneten Argument ab. Überwacht durch C wird dieser Vorgang in keinem Fall.

**Bspl.:** Hier wird eine eindimensionale Reihe zweimal ausgegeben. Die Funktion `show_1` setzt dabei auf die Array-Notation, `show_2` nutzt Zeiger-Eigenschaften.

```
#include <stdio.h>

/* Prototypen */
void show_1 (int[], int);           alternativ: int*
void show_2 (int*, int);           alternativ: int[]

int main ()
{
    int a[] = {11, 22, 33, 44, 55};

    show_1 (a, 5);
    show_2 (a, 5);
    return 0;
}
```

```

void show_1 (int a[], int size)           alternativ: int *a
{
    int i;
    for (i = 0; i < size; i++)
        printf ("%d ", a[i]);
    printf ("\n");
}

void show_2 (int *p, int repeat)          alternativ: int p[]
{
    while (repeat--)
        printf ("%d ", *p++);
    printf ("\n");
}

```

Geeignet wären auch die Parameter `const int a[]` bzw. `const int *p` und entsprechend geänderte Prototypen. Dazu braucht das Array `a` in der Funktion `main` nicht konstant sein.

Für mehrdimensionale Array gilt entsprechend: In der ersten Dimension wirken Array-Notation und Zeiger-Schreibweise für Parameter völlig identisch.

In den weiteren Dimensionen ist es allerdings erheblich, ob diese eine vorgegebene Länge haben oder über Zeiger abgewickelt werden. Hier spiegelt sich der konstruktive Unterschied<sup>5</sup>, den wir in den vorausgegangenen Beispielen erläutert haben, auf gleiche Weise wieder.

Bspl.: `(double p[][5])`  $\equiv$  `(double (*p)[5])`

Mit diesen Parametern wird ein Zeiger auf ein 5-elementiges `double`-Array (bzw. auf den Anfang einer Matrix aus 5-elementigen Zeilen) vereinbart.

Diese Parameter sind geeignet für die obigen Variablen `alpha` und `beta` als Argument.

Bspl.: `(double *p[])`  $\equiv$  `(double **p)`

Als Parameter wird ein Zeiger vereinbart, der auf einen Zeiger (bzw. den Anfang eines Zeiger-Arrays) verweisen kann, welcher weiter auf ein `double`-Objekt (bzw. den Anfang eines `double`-Arrays) zeigen kann.

Geeignete Argumente wären `gamma` und `delta`.

Wir verzichten auf den Einsatz konstanter Zeiger als Parameter. Sowohl in Zeiger- als auch in Array-Notation sind daher alle Parameter hier variabel, so auch im nächsten Beispiel. Dort werden allerdings die angezeigten Elemente als konstant vereinbart.

---

<sup>5</sup> Wird die jeweilige Adresse einer Zeile über die Länge **berechnet** oder liegt die Adresse **abgespeichert** vor?

Bspl.: `(const char cs[]) ≡ (const char *cs)`

Auch diese Parameter-Vereinbarungen sind gleichwertig.

Derartige Parameter (konstantes Array bzw. Zeiger auf Konstante) kommen mehrfach im nachfolgenden Kapitel und im Anhang A vor und dürfen mit konstanten oder veränderlichen Zeichenketten (Strings, `char`-Arrays) initialisiert werden.

Es mag etwas überraschen, dass die vorausgehende Funktion auch mit einem nicht-konstanten Array als Argument aufgerufen werden kann. Wir erinnern dazu an die Aussage, dass Parameter für konstante Arrays in spezifischen Fällen auch variable Arrays als Argumente akzeptieren (Kapitel 11). Dies ist eine solche Situation. Im mehrdimensionalen Fall funktioniert es hingegen nicht.



# 13. Strings und Arrays aus Zeichen

## Grundlagen

Eine in Doppelapostrophen eingeschlossene Folge von Zeichen<sup>1</sup> bildet einen Zeichen-Array, also ein Array vom Basistyp `char`. Die Doppelapostrophe sind syntaktischer Natur und gehören nicht zur Zeichenkette selbst. Als Zugeständnis an Funktionen, das Ende einer Zeichenkette selbst finden zu können, wird diese intern durch ein Nullzeichen abgeschlossen. Derartig abgeschlossene Zeichenketten bezeichnen wir auch als **Strings**. Die interne Darstellung erlaubt auch „leere“ Strings, Zeichenketten ohne Inhalt, indem bereits das erste abgespeicherte Zeichen das Nullzeichen ist.

Mit **Nullzeichen** (*NUL*) ist der Ordinalwert<sup>2</sup> 0 und nicht der Wert für die Ziffer 0 nach Codetabelle gemeint. Der Wert wird mittels des Zeichens in Ersatzdarstellung `'\0'` dargestellt, keinesfalls aber mit `'0'`. Ebenfalls kann man kurzerhand das `int`-Literal 0 benutzen, nimmt dann aber eine erforderliche Typkonvertierung in Kauf.

Bspl.: `"Bond 007"` besitzt 9 `char`-Zeichen. Das Leerzeichen<sup>3</sup> ist in Strings ein konkretes Zeichen.

'B'	'o'	'n'	'd'	' '	'0'	'0'	'7'	NUL
-----	-----	-----	-----	-----	-----	-----	-----	-----

Datenobjekte für Strings können als Arrays von Basistyp `char` oder aber auch als `char`-Zeiger vereinbart werden. Allerdings gibt es dabei zu beachtende Unterschiede.

```
Bspl.: char b1[] = {'B','o','n','d',' ','0','0','7','\0'};
char b2[] = "Bond 007";
char *b3 = "Bond 007";
```

In allen drei Fällen entsteht der gewünschte String (mit abschließendem Nullzeichen). `b1[i]`, `b2[i]` und `b3[i]` liefern für einen zulässigen Index `i` den gleichen `char`-Wert.

`b1` und `b2` bezeichnen als Arrays unveränderlich jeweils einen Speicherbereich, der mit den angegebenen Zeichen initialisiert wurde. Die einzelnen Elemente können nachfolgend verändert werden.

`b3` ist hingegen als Zeiger derart initialisiert worden, dass er auf eine konstante Zeichenkette verweist. Der Versuch, Zeichen zu ändern, führt zu undefinierten Ergebnissen. Jedoch kann der Zeiger selbst verändert werden, so dass er auf etwas anderes zeigt.

```
Bspl.: b3 = "Dr. No";
```

Man beachte ferner, dass bei Wertzuweisungen völlig im Sinne der bisherigen Kapitel nur Adressen und nicht die Zeichen-Arrays selbst kopiert werden. So bewirkt mit diesem Beispiel ein Ausdruck `b3 = b2` lediglich, dass nun auch der Zeiger `b3` auf die Zeichenkette des Arrays `b2` zeigt. Für eine echte Kopie des Zeicheninhalts muss eine Funktion der Standard-Bibliothek herangezogen werden.

<sup>1</sup> Es gelten die gleichen Ersatzdarstellungen wie im elementaren Datentyp `char`.

<sup>2</sup> Alle Bits sind auf 0 gesetzt.

<sup>3</sup> Im Gegensatz zum Begriff des Leerraums im C-Quellcode.

## Standard-Funktionen für Strings

Für die Deklaration der nachfolgenden Funktionen ist die Header-Datei `<string.h>` zu importieren.

Diese Liste ist nicht vollständig.

```
char * strcpy (char *s, const char *ct);
```

Zeichenkette `ct` in `s` kopieren, inklusive `'\0'`; liefert `s`.

```
char * strncpy (char *s, const char *ct, size_t n);
```

höchstens `n` Zeichen aus `ct` in `s` kopieren; liefert `s`.  
Mit `'\0'` auffüllen, wenn `ct` weniger als `n` Zeichen hat.

```
char * strcat (char *s, const char *ct);
```

Zeichenkette `ct` hinten an die Zeichenkette `s` anfügen; liefert `s`.

```
char * strncat (char *s, const char *ct, size_t n);
```

höchstens `n` Zeichen von `ct` hinten an die Zeichenkette `s` anfügen  
und `s` mit `'\0'` abschließen; liefert `s`.

```
int strcmp (const char *cs, const char *ct);
```

Zeichenketten `cs` und `ct` vergleichen;  
liefert `<0`, wenn `cs < ct`, `0`, wenn `cs == ct`, oder `>0`, wenn `cs > ct`.

```
int strncmp (const char *cs, const char *ct, size_t n);
```

höchstens `n` Zeichen von `cs` mit der Zeichenkette `ct` vergleichen;  
liefert `<0`, wenn `cs < ct`, `0`, wenn `cs == ct`, oder `>0`, wenn `cs > ct`.

```
char *  strchr (const char *cs, int c);
```

liefert Zeiger auf das erste `c` in `cs` oder NULL-Zeiger, falls nicht vorhanden.

```
char * strrchr (const char *cs, int c);
```

liefert Zeiger auf das letzte `c` in `cs` oder NULL-Zeiger, falls nicht vorhanden.

```
size_t strspn (const char *cs, const char *ct);
```

liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich in `ct` vorkommen.

```
size_t strcspn (const char *cs, const char *ct);
```

liefert Anzahl der Zeichen am Anfang von `cs`, die sämtlich nicht in `ct` vorkommen.

```
char * strpbrk (const char *cs, const char *ct);
```

liefert Zeiger auf die Position in `cs`, an der irgendein Zeichen aus `ct` erstmals vorkommt,  
oder NULL-Zeiger, falls keines vorkommt.

```
char * strstr (const char *cs, const char *ct);
```

liefert Zeiger auf erste Kopie von `ct` in `cs` oder NULL-Zeiger, falls nicht vorhanden.

```
size_t strlen (const char *cs);
```

liefert Länge von `cs` (ohne `'\0'`).

Diese Kurzbeschreibungen der Funktionen wurden entnommen aus dem Anhang des Lehrbuchs *Kernighan/Ritchie – Programmieren in C* und geringfügig angepasst.

Dazu noch folgende Ergänzungen:

- Die String-Parameter sind immer dann als `const` vereinbart worden, wenn das Argument nicht in der Funktion verändert wird. Dies heißt umgekehrt, dass nicht-konstante Parameter im Regelfall in der Funktion tatsächlich verändert werden. Nur im ersten Fall sind konstante Strings, insbesondere String-Literale, als Argumente erlaubt.
- Einige Funktionen liefern einen `char*`-Wert, also eine Adresse, als Ergebnis zurück. Diese verweist dann auf ein Zeichen des ersten Arguments.
- `c` steht für einen `int`-Wert, der intern in `char` verwandelt wird.
- `n` hat den Typ `size_t`, einen vorzeichenlosen, ganzzahligen Typen. Der konkrete Typ ist implementationsabhängig.
- Die Beschreibungen der Vergleichsfunktionen `strcmp` und `strncmp` sind insofern etwas irreführend, da natürlich nicht die R-Werte der Zeiger-Parameter `cs` und `ct` verglichen werden! Es finden vielmehr lexikographische Vergleiche der angezeigten Strings auf Basis der zugrundeliegenden Codetabelle statt. `strcmp("AMEISE", "Adler")` liefert einen negativen Wert, da `'M' < 'd'` gilt (siehe ASCII-Codetabelle). Man erkennt ferner an diesem Beispiel, dass die Groß-/Kleinschreibung beim Vergleich nicht ignoriert wird.

## Ein- und Ausgabe

Bereits in unserem ersten Beispiel haben wir einen String mittels der Funktion `printf` ausgegeben. Später haben wir gesehen, dass im ersten String-Argument dieser Funktion Formatelemente als Platzhalter für die weiteren Argumente ausgewertet werden.

Das **Formatelement** für einen String lautet `%s`.

```
Bspl.: printf ("hello, world\n");
       printf ("%s", "hello, world\n");
       printf ("%s\n", "hello, world");
       printf ("hello, %s\n", "world");
```

Der erste String wird ausgegeben; jedoch wird vorher das Formatelement durch den 2. String ersetzt.

```
Bspl.: char *Vorname = "Charly";
       char Name[] = "Brown";

       printf ("Mein Name ist %s %s.\n", Vorname, Name);

Ausgabe: Mein Name ist Charly Brown.
```

Die auszugebende Zeichenkette muss korrekt durch ein Nullzeichen abgeschlossen sein.

```
Bspl.: char e[] = {'L', 'i', 'n', 'u', 's'};

       printf ("%s", e);
```

Fehler!

Aufgrund des fehlenden Nullzeichens zum Abschluss der Zeichenkette kann die `printf`-Funktion das Ende nicht erkennen und wird versuchen, weitere Inhalte aus dem Speicher auszugeben, die sich zufällig hinter dieser Zeichenkette befinden. Nach einer unsinnigen Ausgabe trifft sie dabei irgendwann noch auf ein Nullzeichen, oder das Programm wird fehlerhaft beendet aufgrund eines unzulässigen Speicherzugriffs.

Das Formatelement `%s` wird auch für die Eingabe mittels `scanf` genutzt. Aus Sicherheitsgründen raten wir dringend dazu, dies stets mit einer sinnvollen Feldbreite zu versehen (*width*, syntaktische Form in Anlage A). So stellt beispielsweise das Formatelement `%20s` sicher, dass maximal zwanzig Zeichen dem Textstrom entnommen werden.

- Leerraum in der Eingabe wird zunächst übergangen. Dann wird eine Folge von Zeichen gelesen<sup>4</sup>. Falls eine Feldbreite angegeben ist, bestimmt diese die Anzahl der einzulesenden Zeichen; ansonsten ist diese nicht begrenzt. Der Lesevorgang endet vorzeitig beim nächsten Leerraum. Hierzu zählen laut Kapitel 1 insbesondere das Leerzeichen und das Zeilenende.
- Das zugehörige Argument muss vom Typ `char*` sein und auf ein Array zeigen, das tatsächlich die einzulesenden Zeichen und das abschließende Nullzeichen aufnehmen kann. Ansonsten erfolgen undefinierte Speicherzugriffe.

```
Bspl.: char prename[21];
char name[21];
int year;

printf ("Vor- und Zuname eingeben: ");
scanf ("%20s%20s", prename, name);

printf ("Geburtsjahr: ");
scanf ("%d", &year);

printf ("Vorname: %s Zuname: %s geboren: %d\n",
                                             prename, name, year);
```

Dieses Programmstück funktioniert so nur dann, wenn sich der **Anwender** an gewisse Spielregeln hält.

Zunächst einmal dürfen für einen Namen nicht mehr als 20 Zeichen eingegeben werden. Ansonsten verbleiben Zeichen, die nachfolgend noch anderweitig in der Eingabe verwertet werden.

Auf die Anfrage nach den Namen sind genau zwei Zeichenketten, die selbst keinen Leerraum enthalten, gegebenenfalls aber voneinander durch Leerraum getrennt sind, einzugeben. Gibt der Anwender fälschlich mehr ein (z.B. `Otto von Bismarck`), so wird die dritte Zeichenkette (`Bismarck`) als Eingabe für die ganze Zahl mittels `%d` bewertet. Gibt es hingegen in der Eingabe gar nur eine verwertbare Zeichenkette, so wird die nachfolgende Jahreseingabe zum Namen.

Das Beispiel deutet bereits an, welcher Aufwand für die Erstellung einer sicheren Anwender-Schnittstelle zu betreiben ist. Dabei muss beispielsweise entschieden werden, ob wie hier ein Leerzeichen einzelne Strings trennen darf oder ob vielleicht das Zeilenende (ein String pro Zeile) einzig sinnvoll erscheint. Ferner fehlt in diesem Beispiel noch eine Verarbeitung von überzähligen Zeichen bei zu langen Eingaben.

Sicher ist auch nicht in jedem Fall `scanf` die geeignete Wahl. Alternativen stellen wir im Anhang A vor. Zu erwähnen ist dabei insbesondere die Funktion `fgets`, die Ähnliches leistet, sich aber in Details unterscheidet (Umgang mit Leerzeichen und Zeilenende).

---

<sup>4</sup> Die einzulesende Zeichenfolge ist nicht in Doppelapostrophe einzuschließen.

# 14. Strukturen (struct, union)

In diesem Kapitel stellen wir eine neue Datenstruktur vor, die selbst als Struktur bezeichnet wird bzw. deren Objekte als Strukturen bezeichnet werden. Dies führt leider zu einer gewissen Inflation des Begriffes „Struktur“: Programmstrukturen (strukturierte Programmierung), Kontrollstrukturen, Datenstrukturen, ...

Alternative Bezeichnungen für diese neuen Strukturen sind auch Verbund oder *record*.

## Strukturen in C

### Datenstruktur

Strukturen sind zusammengesetzte Datenobjekte mit einer heterogenen, statischen Datenstruktur. Der Zugriff auf die einzelnen Komponenten erfolgt über Namen (Bezeichner). Wir vereinbaren hier den Begriff „Komponente“, während wir bei Arrays weiterhin von „Elementen“ sprechen.

Die Vereinbarung einer Struktur ist aufwändig im Vergleich zu einem Array, denn wir müssen für jede Komponente den Typ und den Bezeichner festlegen. Daher raten wir insbesondere bei umfangreichen Strukturen dazu, zunächst einen Strukturtyp zu vereinbaren und erst in der Folge die benötigten Objekte dieses Typs.

### Vereinbarung von Strukturtypen

**Form:** `struct Etikett { Folge_von_Vereinbarungen } ;`

Das *Etikett* (engl. *tag*) muss syntaktisch ein Name sein. Die Kombination aus dem reservierten Wort `struct` und dem *Etikett* wirkt ab jetzt wie ein neuer Typname.

Die *Folge\_von\_Vereinbarungen* deklariert die Komponenten wie Variablen oder Konstanten. Eine Komponente darf den gleichen Namen wie ein Objekt aus der Umgebung besitzen, ohne dass ein Konflikt auftritt. Als Datentypen kommen alle bekannten Typen in Frage, also insbesondere auch Arrays, Zeiger und Strukturen selbst. Speicherlassen-Attribute und Initialisierungen sind an dieser Stelle allerdings nicht statthaft.

Bspl.: `struct KfzKenn`

```
{  
    char *ort;  
    char bubu[3];  
    unsigned int zahl;  
};
```

← Semikolon nicht vergessen!

Aufbau eines KFZ-Kennzeichens

- Zeiger auf Ortsangabe
- max. zwei weitere Zeichen (String)
- natürliche Zahl

Der entstehende Strukturtyp heißt: `struct KfzKenn`

Diese Vereinbarung darf lokal oder global erfolgen. Es gelten die bekannten Regeln für Gültigkeitsbereiche. Besteht ein Programm aus mehreren Programmdateien, die auf eine derartige Vereinbarung zugreifen sollen, ist die Vereinbarung in einer Header-Datei äußerst sinnvoll.

## Vereinbarung von Strukturen und deren Initialisierung

Mit der Vereinbarung eines Strukturtyps wie im obigen Beispiel wird noch kein Speicherplatz angelegt. Dies geschieht erst mit der Vereinbarung von Objekten (Variablen, Konstanten). Unter Verwendung der Typbezeichnung nutzt man dazu die gleiche Form wie bei den Standard-Datentypen. Auch gelten weiterhin die im Kapitel 9 aufgestellten Regeln; dies trifft insbesondere auf die Speicherklassen-Attribute und das Attribut `const` zu.

Bspl.: `struct KfzKenn meinAuto;`

|  
 Typbezeichnung    Name

Erfolgt diese Vereinbarung lokal, so gehört die neue Strukturvariable mit Namen `meinAuto` zur automatischen Speicherklasse und wird nicht initialisiert.

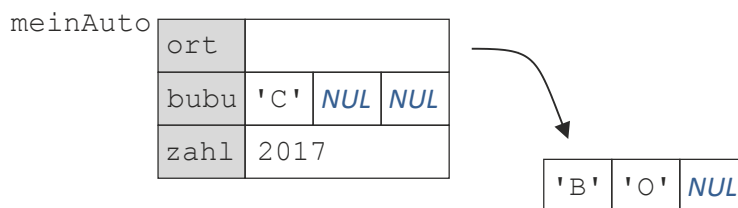
Jetzt können wir die Komponenten von Strukturen aber auch explizit initialisieren. Die Regeln entsprechen der Initialisierung von Arrays. Auf diese Weise werden auch konstante Komponenten einer Struktur bzw. konstante Strukturen initialisiert.

Bspl.: `struct KfzKenn meinAuto = {"BO", "C", 2017};`

Wieder entsteht eine Strukturvariable `meinAuto` des genannten Strukturtyps. Die Zeiger-Komponente `ort` verweist auf den konstanten String "BO"; das Zeichen-Array `bubu` wird mit dem String "C" initialisiert; `zahl` erhält den Wert 2017.

(Dass die Ortsangabe ein Zeiger ist, hingegen für die Buchstaben-Kombination `bubu` ein Array vereinbart wurde, ist rein demonstrativ. Man vergleiche dies mit dem Kapitel 13.)

Und so stellen wir uns das entstandene Objekt grafisch vor, wobei die Komponenten alternativ auch horizontal angeordnet werden können. Die grau hinterlegten Bezeichner werden in der Realität nicht im Objekt mitgespeichert. Man kann an dieser Darstellung auch kritisieren, dass alle Komponenten gleich breit dargestellt werden, da dies nicht unbedingt den Speicher-Tatsachen entspricht.



## Kombinierte Form

Die Vereinbarung eines Strukturtyps und seiner Variablen lässt sich auch kompakt realisieren.

**Form:** *Attribute*<sub>opt</sub> `struct` *Etikett*<sub>opt</sub> { *Folge\_von\_Vereinbarungen* } *Liste\_von\_Namen* ;

Die *Folge\_von\_Vereinbarungen* beschreibt die Komponenten der Struktur wie bisher auch; die *Liste\_von\_Namen* bezeichnet die gleichzeitig vereinbarten Variablen. Dabei sind Initialisierungen möglich. – *Attribut* kann ein Speicherklassen-Attribut für die Variablen sein (Kapitel 9). Tritt `const` als *Attribut* auf, werden statt Variablen natürlich Konstanten vereinbart. – Fehlt das optionale *Etikett*, so kann man aus formalen Gründen keine weiteren typgleichen Objekte nachträglich vereinbaren.

```

Bspl.: struct KfzKenn
{
    char *ort;
    char bubu[3];
    unsigned int zahl;
}
↑ meinAuto = {"BO", "C", 2017};
Kein Semikolon!

```

Hier werden die vorausgehenden Beispiele kombiniert in eine einzige Vereinbarung umgesetzt.

### Zugriff

Der Zugriff auf eine Komponente einer Struktur erfolgt durch den Punkt-Operator (L- und R-Wert).

**Form:** *Ausdruck* . *Komponentenname*

Der *Ausdruck* wird berechnet und muss eine Struktur liefern, auf deren Komponente dann zugegriffen wird. Der *Ausdruck* kann im einfachsten Fall natürlich ein Strukturname selbst sein.

Bspl.: Hier und nachfolgend setzen wir das Eingangsbeispiel (Vereinbarung oben) sinngemäß fort.

<code>meinAuto.ort = "EN";</code>	Die Zeiger-Komponente <code>ort</code> von <code>meinAuto</code> zeigt nun auf den konstanten String "EN".
<code>strcpy (meinAuto.bubu, "TE");</code>	Die Array-Komponente <code>bubu</code> der Strukturvariablen <code>meinAuto</code> erhält durch eine String-Funktion eine neue Belegung.
<code>meinAuto.zahl = 99;</code>	Die Komponente <code>zahl</code> von <code>meinAuto</code> erhält per Zuweisung den Wert 99.
<code>show (meinAuto);</code>	Diese Funktion definieren wir später noch.
<b>Ausgabe:</b> Fahrzeug: EN TE 99	

### Wertzuweisung

Eine Wertzuweisung zwischen Strukturen ist möglich. Dabei wird der gesamte Inhalt der Struktur, also die Werte aller Komponenten, kopiert. (Strukturnamen wirken nicht wie Adressen oder Zeiger.)

<code>struct KfzKenn meinKrad;</code>	Vereinbarung an geeigneter Stelle
<code>meinKrad = meinAuto;</code>	vollständige Kopie
<code>meinKrad.ort = "HER";</code>	Änderung der Zeiger-Komponente <code>ort</code>
<code>meinKrad.bubu[0] = 'N';</code>	Änderung erstes Zeichen von <code>bubu</code>
<code>show (meinAuto);</code>	
<code>show (meinKrad);</code>	
<b>Ausgabe:</b> Fahrzeug: EN TE 99	
Fahrzeug: HER NE 99	

## Konstanten

Das folgende Beispiel zeigt, wie sich konstante Komponenten und konstante Strukturen auswirken.

Bspl.: <code>struct Beta {int z; const int n};</code>	Die Komponente <code>n</code> ist konstant.
<code>struct Beta x = {1, 2};</code>	<code>x</code> ist eine Variable.
<code>const struct Beta y = {3, 4};</code>	<code>y</code> ist ein (symbolische) Konstante.
<code>x.n = 1;</code>	jeweils <b>nicht</b> erlaubt
<code>y.z = 2;</code>	
<code>y.n = 3;</code>	
<code>y = x;</code>	

## Mehr zu Funktionen und Parametern

### Strukturen als Parameter

Die Parameterübergabe erfolgt nach der bekannten Technik *call by value*. Dies bedeutet, dass der Parameter mit dem Wert des Arguments initialisiert wird.

Bspl.: Wir setzen das Eingangsbeispiel fort und liefern die noch fehlende Funktion `show` nach.

```
void show (const struct KfzKenn k)
{
    printf ("Fahrzeug: %s %s %d\n", k.ort, k.bubu, k.zahl);
}
```

Bspl.: <code>show (meinAuto);</code>	So haben wir vorweg bereits die Struktur <code>meinAuto</code> ausgegeben.
--------------------------------------	--

Bei der Übergabetechnik *call by value* tritt auch bei Strukturen der bereits bekannte Umstand ein, dass wir auf diesem Weg eine Variable als Argument nicht verändern können. Wird die Änderung mindestens einer Komponente einer Strukturvariablen angestrebt, so übergeben wir stattdessen einen Zeiger, siehe Übergabetechnik *call by reference* im Kapitel 10.

Bspl.: <code>void folgeZahl (struct KfzKenn *p)</code>	Der Parameter <code>p</code> ist ein Zeiger.
<code>{</code>	{ erst Dereferenzierung des Zeigers, danach Komponentenzugriff, zuletzt Inkrementierung
<code>    (*p).zahl++;</code>	
<code>}</code>	
Bspl.: <code>folgeZahl (&amp;meinAuto);</code>	{ Aufruf: Adressoperator erforderlich (oder entsprechender Zeiger)
<code>show (meinAuto);</code>	
Ausgabe:      Fahrzeug: EN TE 100	

Bei einer normalen Parameterübergabe ohne Zeiger wird eine Struktur-Variable als Argument tatsächlich kopiert. Im obigen Beispiel `show` wird also `meinAuto` auf `k` umkopiert. Bei wirklich großen Strukturen, z.B. wenn diese als Komponenten lange Arrays beinhalten, kann dies ineffektiv werden. Diesen Zeitaufwand kann man sich ersparen, indem man nicht die Struktur übergibt, sondern auch hier mit einem Zeiger als Parameter arbeitet.



## Funktionen mit Strukturen als Ergebnis

Eine Funktion darf eine Struktur als Ergebnis zurückliefern. Hier sollte man aus Effektivitätsgründen ebenfalls darüber nachdenken, ob nicht ein entsprechender Zeiger ausreichend wäre. Im folgenden Beispiel ist dies nicht erwünscht, weil das Ergebnis einer Strukturvariablen zugewiesen werden soll.

```

Bspl.: struct KfzKenn scanKfzKennzeichen ()
{
    struct KfzKenn k;

    printf ("Keine Leerzeichen eingeben!\n");
    printf ("Laengen beachten!\n");

    k.ort = (char*) calloc (4, sizeof(char));           s. nächstes Kapitel
    printf ("Ort (max 3): ");
    scanf ("%3s", k.ort);

    printf ("Buchstaben (max 2): ");
    scanf ("%2s", k.bubu);

    printf ("Zahl: ");
    scanf ("%u", &k.zahl);

    printf ("\nGelesen wurde\n");
    show (k);

    return k;
}

```

```

Bspl.: meinKrad = scanKfzKennzeichen ();

```

Auf diese Weise können jetzt neue Daten z.B. für `meinKrad` eingelesen werden.

## Zeiger, Arrays und Strukturen

Es dürfen **Zeiger** als Komponenten in Strukturen vorkommen; das haben wir mit dem Zeiger `ort` bereits so vollzogen. – Ebenso dürfen Zeiger auf Strukturen verweisen. Dies haben wir mit dem Zeiger-Parameter in unserem Beispiel `folgeZahl` auch schon realisiert.

Der Zugriff über einen Zeiger auf die Komponenten einer Struktur tritt recht häufig auf. Daher gibt es zusätzlich den „Pfeil“-Operator `->` für genau diesen Zweck, der die Schreibweise etwas vereinfacht. Er ersetzt somit gleichzeitig die Dereferenzierung und den Punkt-Operator.

Für einen Zeiger `p` und eine Komponente `k` gilt:

$$p \rightarrow k \quad \equiv \quad (*p) . k$$

Bspl.: In der bereits vorgestellten Funktion `folgeZahl` können wir somit noch diese syntaktische Ersetzung vornehmen.

```

void folgeZahl (struct KfzKenn *p)
{
    p->zahl++;
}

```

Eine Struktur darf ein **Array** enthalten; in unserem Beispiel ist `bubu` ein Zeichen-Array. Umgekehrt dürfen auch Arrays über einem Strukturtyp gebildet werden.

```
Bspl.: struct KfzKenn fuhrpark[500];           erzeugt Array mit 500 Strukturen
        fuhrpark[222]                = meinAuto;      belegt ein Array-Element gesamt
        fuhrpark[222].bubu[0] = 'G';                 aus "TE" wird nun "GE"
```

## Geschachtelte Strukturen

Eine Komponente einer Struktur darf eine Struktur eines anderen Typs sein<sup>1</sup>. Dazu betrachten wir ein weiteres Beispiel: Eine Strecke soll über ihren Start- und Endpunkt in einem Koordinatensystem definiert werden.

```
Bspl.: struct Punkt
{
    double x, y;
};

struct Strecke
{
    struct Punkt start, ziel;
};

struct Strecke s = {{1.0, 2.0}, {3.0, 4.0}};

printf ("s.start.x = %f\n", s.start.x);
printf ("s.start.y = %f\n", s.start.y);
printf ("s.ziel.x   = %f\n", s.ziel.x);
printf ("s.ziel.y   = %f\n", s.ziel.y);
```

Hier haben wir zunächst separate Datentypen für Punkte und Strecken eingeführt. Dies hat den großen Vorteil, dass wir nachfolgend bei Bedarf weitere Variablen und auch Funktionsparameter dieser Typen vereinbaren können. Mit `s` wurde eine konkrete Strukturvariable erzeugt und initialisiert.

```
Bspl.: struct Strecke
{
    struct {double x, y;} start, ziel;
};

struct Strecke s = {{1.0, 2.0}, {3.0, 4.0}};
```

Wir streichen jetzt die separate Vereinbarung zu `Punkt`. Stattdessen treffen wir unter Verzicht auch auf das Etikett `Punkt` die Vereinbarungen zu Punkten in der kombinierten Form innerhalb der Vereinbarung zu `Strecke`.

Ansonsten ändert sich nichts; die Zugriffe funktionieren genauso wie im obigen Beispiel. Die Typangabe `struct Punkt` kann man jetzt natürlich nicht mehr für weitere Variablen oder Parameter verwenden.

---

<sup>1</sup> Erlaubt sind ebenfalls Zeiger auf die Struktur selbst. Dies zeigen wir im Kapitel 15 / Beispiel 2.

```

Bspl.: struct
{
    struct {double x, y;} start, ziel;
}
    s = {{1.0, 2.0}, {3.0, 4.0}};

```

Da es nun kein Etikett mehr gibt, müssen wir die Variable `s` unmittelbar mit der Strukturbeschreibung vereinbaren (kombinierte Form).

Auch hier funktionieren noch alle Zugriffe genauso wie im ersten Beispiel. Mit dieser Vereinbarung haben wir uns jetzt jedoch alle Möglichkeiten verbaut, weitere typgleiche Objekte zu bezeichnen.

In der nachfolgenden Variante unseres Beispiels wollen wir gleichzeitig noch zwei weitere Dinge demonstrieren:

- Die Komponenten von `Strecke` sollen einmal zu Zeigern werden.
- Die Definition von `Punkt` soll syntaktisch nach der von `Strecke` erfolgen. Gemäß den Gültigkeitsregeln deklarieren wir daher vorher den Strukturtyp.

```

Bspl.: struct Punkt;                                Deklaration (unvollständiger Typ)
struct Strecke
{
    struct Punkt *start, *ziel;
};

struct Punkt
{
    double x, y;
};

struct Punkt p1 = {1.0, 2.0}, p2 = {3.0, 4.0};

struct Strecke s;
s.start = &p1;
s.ziel  = &p2;

printf ("s.start->x = %f\n", s.start->x);
printf ("s.start->y = %f\n", s.start->y);
printf ("s.ziel->x  = %f\n", s.ziel->x);
printf ("s.ziel->y  = %f\n", s.ziel->y);

```

Wie man sieht, wird die Initialisierung nun etwas komplexer. So kann man jetzt auf die beiden Punkte nicht nur über `*s.start` und `*s.ziel`, sondern auch noch direkt über deren Namen `p1`, `p2` zugreifen.

Der Pfeil-Operator erlaubt diese Notation für die Komponenten der Punkte:

```
s.start->x    statt    (*s.start).x
```

## Union

Eine Spielart der Struktur stellt in C die Union dar. Hier werden die Komponenten nicht hintereinander im Speicher abgelegt, sondern überlappend, wobei jede Komponente die gleiche Anfangsadresse hat. Man kann dann logisch immer nur eine der Komponenten nutzen. Dies macht algorithmisch nur Sinn, wenn die Komponenten nie zusammen auftreten können. Eine Union erlaubt es aber auch, den gleichen Speicherinhalt unterschiedlich zu interpretieren. Mehr dazu später.

Bei den Vereinbarungen wird das reservierte Wort `struct` gegen `union` ausgetauscht.

Die Reihenfolge der Komponentenvereinbarungen in der Union spielt grundsätzlich keine Rolle; wird jedoch die Union initialisiert, so beziehen sich die Angaben auf die erstvereinbarte Komponente.

Die weiteren Regeln sind – von obigen Besonderheiten abgesehen – identisch.

Ein Beispiel befindet sich am Ende von Kapitel 17.

# 15. Dynamische Datenstrukturen

## Dynamischer Speicher

Bisher wurde die Speicherverwaltung in Variablen- oder Konstanten-Definitionen versteckt. Wir haben damit Objekte der automatischen oder statischen Speicherklasse erzeugen können, was unter anderem die Lebensdauer dieser Objekte bestimmt hat. Nun nehmen wir die Sache selbst in die Hand und nutzen die Möglichkeiten einer dynamischen Speicherverwaltung.

Ein C-Programm nutzt allgemein vier Speicherbereiche.

Code	Maschinencode des Programms
Data	statische Speicherklasse
Stack <sup>1</sup>	automatische Speicherklasse
Heap <sup>1</sup>	dynamischer Speicher

Der dynamische Speicher wird also logisch getrennt von dem Speicher, den wir bis jetzt z.B. für globale und lokale Variablen, Funktionen etc. genutzt haben. *Stack* und *Heap* funktionieren ähnlich: Bei einer Speicheranforderung wird der benötigte Speicher zur Verfügung gestellt, sofern noch verfügbar. Dies geschieht im *Stack* automatisch entsprechend der Blockstruktur. Im *Heap* veranlasst jedoch der Programmierer die Anforderung.

## Reservierung und Freigabe

### sizeof-Operator

Diesen Operator benötigen wir, um die Größe von Objekten in Bytes zu bestimmen. Der Operand ist entweder ein Ausdruck, welcher allerdings selbst **nicht** berechnet wird, oder ein Typname in Klammern. Bei einem Array wird die Größe des gesamten Arrays bestimmt!

```
Bspl.: int i;
       int *p;
       int v = {1, 2, 3, 4, 5}

       sizeof i    oder   sizeof (int)    für einen int-Wert,
       sizeof p    oder   sizeof (int*)   für einen int-Zeiger,
       sizeof v    oder   sizeof (int[5]) für fünf int-Werte.
```

Achtung: Das Ergebnis des Operators ist vom Typ `size_t`. Dies ist ein vorzeichenloser, ganzzahliger Datentyp, der nicht pauschal mit einem der vorgestellten Standard-Datentypen übereinstimmen muss. – Die Abfrage für Arrays funktioniert nur eingeschränkt bei Parametern. Man erhält dann die Größe des Zeiger-Parameters, nicht die Länge des Arrays. – Bei Strukturen gibt `sizeof` den tatsächlichen Speicherbedarf an. Dieser kann größer sein als die Summe der Komponenten, da abhängig vom Adressraum des Laufzeitsystems Lücken und Auffüllungen erforderlich sein können.

---

<sup>1</sup> Die Zugriffsmethodik für *Stacks* und *Heaps* aus informationstechnischer Sicht interessiert uns nicht weiter.

Die nachfolgenden Funktionen sind in der Header-Datei `<stdlib.h>` deklariert, vgl. mit Anhang C.

### Funktion malloc

Das Argument dieser Funktion bestimmt die Größe des zu reservierenden Speichers, gemessen in Bytes. Bei Erfolg liefert die Funktion die Anfangsadresse des zugewiesenen Speicherbereichs zurück, bei Nichterfolg ist das Ergebnis 0 (NULL-Zeiger). Es erfolgt keine Initialisierung.

Das Ergebnis der Funktion ist vom Typ `void*`, eine nicht an einen Typ gebundene Adresse. Diese kann automatisch auf einen anderen Zeigertyp gewandelt werden. Wir bevorzugen in unseren Beispielen dennoch eine explizite Typkonvertierung (*cast*).

```
Bspl.: double *p = (double*) malloc (10 * sizeof(double));
```

Es wird Speicher für zehn `double`-Werte reserviert, der nicht initialisiert wird. Das Ergebnis von `malloc` wird explizit auf den Typ `double*` gewandelt und dient zur Initialisierung des Zeigers `p`. Dieser verweist damit auf zehn `double`-Objekte im dynamischen Speicher. Die Objekte sind damit auch über die Array-Notation variabel zugreifbar: `p[0]` bis `p[9]`

Die hier entstandene Datenstruktur bezeichnen wir gemeinhin als „dynamisches Array“, da sie den gleichen Aufbau mit Indizierung bietet wie ein klassisches, statisches Arrays (siehe Kapitel 11).

```
Bspl.: double *s = (double*) malloc (10 * sizeof(double)) - 1;
```

Die explizite Typumwandlung wird zwingend erforderlich, wenn die Adresse noch verschoben werden soll (Index-Shift, siehe Kapitel 12). Der Zugriff erfolgt jetzt über: `s[1]` bis `s[10]`

(Eine erfolglose Speicherzuordnung wird in diesen Beispielen nicht beachtet.)

### Funktion calloc

Diese Funktion kennt zwei Argumente: die Anzahl und die Größe der anzulegenden Objekte. Im Unterschied zu `malloc` initialisiert `calloc` den angelegten Speicher aber auch noch mit Null.

```
Bspl.: double *p = (double*) calloc (10, sizeof(double));
```

Wirkung wie im oberen Beispiel; jetzt werden jedoch die zehn `double`-Plätze initialisiert.

### Funktion realloc

Diese Funktion ändert die Größe des Bereichs, auf den das 1. Argument verweist, auf den Wert des 2. Arguments ab. Der „vordere“ Teil behält dabei seine Werte. Neuer Speicher wird nicht initialisiert.

```
Bspl.: p = (double*) realloc (p, 20 * sizeof(double));
```

Das „dynamische Array“ wird auf zwanzig `double`-Elemente vergrößert.

(Ein vergleichbarer Schritt wäre bei einem klassischen, statischen Array nicht möglich.)

Die obigen Vereinbarungen mit Initialisierung müssen lokal erfolgen, da nicht-konstante Ausdrücke aufgrund von Funktionsaufrufen vorliegen. Die Größen-Argumente der Funktionen dürften ferner auch variabel sein. – Die im Speicher angelegten Objekte besitzen keinen fest verknüpften C-Namen. Werden hier die Zeiger neu belegt, gehen die Zugriffswege auf die Objekte verloren.

## Funktion free

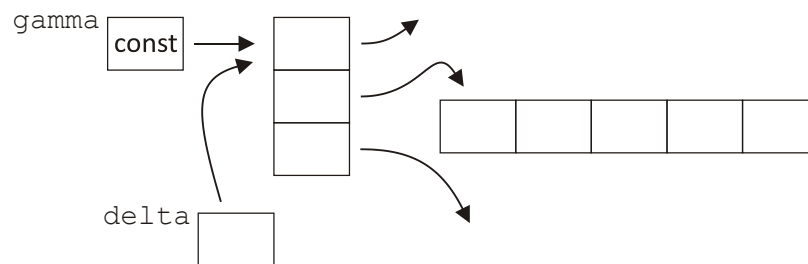
`free` gibt den Bereich frei, auf den das Argument zeigt. Dieser Bereich muss vorher mittels `calloc`, `malloc` oder `realloc` angelegt worden sein.

Bspl.: `free (p);`  
`p = 0;`

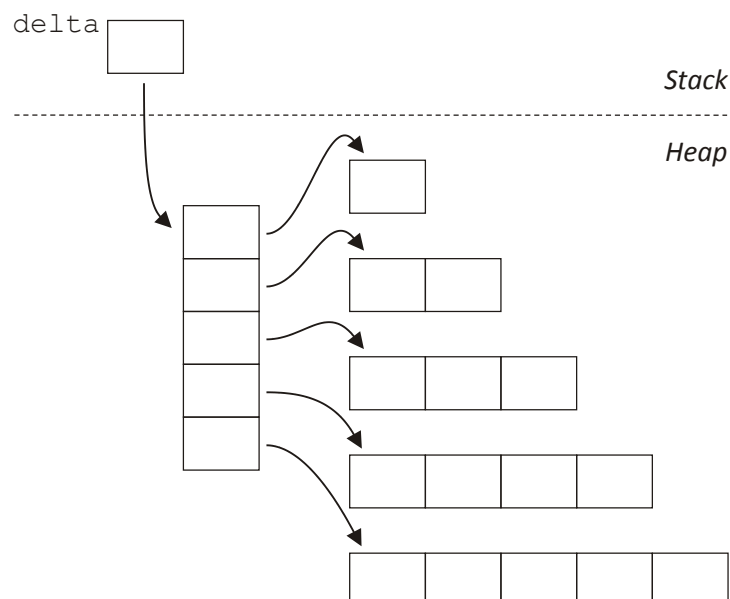
Der bisher von `p` angezeigte Speicher kann nun wieder für andere Zwecke benutzt werden. Die zweite Anweisung stellt sicher, dass nicht doch noch ein ungewollter Zugriff erfolgt.

## Beispiel 1: Dreiecksmatrix

Wir greifen zurück auf ein Beispiel aus Kapitel 12. Dort haben wir hergeleitet, wie man über eine Zeigerverkettung zweidimensionale Datenstrukturen bilden kann, die auch Zeilen unterschiedlicher Länge akzeptieren. Dabei hat uns besonders die Dualität zwischen Zeiger- und Array-Notation interessiert. Die nachfolgende Skizze aus dem Kapitel 12 zeigt noch einmal exemplarisch die dortige Situation.



Wir untersuchen nun, wie man über einen Zeiger auf Zeiger eine Struktur im *Heap* aufbaut, die einer Dreiecksmatrix entspricht. Die nachfolgende Skizze zeigt unser Ziel. Ausgehend von einem Zeiger `delta` sollen über ein Array von Zeigern die einzelnen Zeilen der Größe nach referenziert werden. Im Gegensatz zur Ausgangssituation oben werden alle Arrays nun im dynamischen Speicher realisiert. `gamma` „verliert“ dadurch seinen Namen (und umfasst nun 5 Zeiger).



```

Bspl.: double **create (unsigned int lng)
{
    double **pp = (double**) malloc (lng * sizeof(double*));
    unsigned int i;
    for (i = 0; i < lng; i++)
        pp[i] = (double*) calloc (i+1, sizeof(double));
    return pp;
}

```

Mit dem Zeiger `pp` wird zunächst ein „dynamisches Array“ aus `double`-Zeigern angelegt. Nachfolgend wird jeder dieser Zeiger `pp[i]` der Reihe nach ebenfalls belegt, damit er jeweils auf einen Bereich zunehmender Größe (1, 2, 3, ...) von `double`-Objekten verweist. Somit entsteht die Struktur, die wir skizziert haben. `pp` liefert das Funktionsergebnis, also die Anfangsadresse des Arrays von Zeigern. Auf eine Fehlerbehandlung wurde hier verzichtet<sup>2</sup>.

Die Zeilen unserer Dreiecksmatrix liegen nicht unbedingt im Speicher dicht und in gegebener Reihenfolge hintereinander. Der Speicher für jede einzelne Zeile wird individuell „irgendwo“ an einer freien Stelle im *Heap* besorgt.

```

Bspl.: double **delta = create (5);

```

Damit kann nun das gewünschte Objekt erzeugt werden. Alle `double`-Elemente werden dabei mit 0.0 initialisiert. Auch wenn in diesem Beispiel nur die Konstante 5 steht, das Argument von `create` darf ein variabler Ausdruck sein!

Auf die Elemente der Dreiecksmatrix lässt sich nun per Indizierung zugreifen (Kapitel 12).

## Beispiel 2: Warteschlange

Viele Datenbestände hält man traditionell in Form von **Listen**. Eine derartige Liste als Datenobjekt beinhaltet eine Folge gleichartiger Einträge; es liegt somit eine homogene Datenstruktur vor. Listen sind größendynamisch. Sie können insbesondere auch leer sein, was in der Regel auch die Ausgangssituation ist. Neue Elemente kommen dann hinzu, vorhandene Elemente können wieder gestrichen werden.

Eine Liste erlaubt keinen wahlfreien Zugriff auf alle Elemente, d.h. man kann nicht zu jedem Zeitpunkt auf jedes Element direkt zugreifen. Stattdessen hat man zunächst nur einen Zugriff auf wenige ausgesuchte Elemente der Liste, z.B. auf das erste oder letzte Element. Von dieser Position aus muss man sich dann gegebenenfalls sequentiell Element für Element durch die Liste bewegen.

Es gibt verschiedene Arten von Listen bezüglich der konkreten Zugriffsoperationen. Wir untersuchen hier im Beispiel **Warteschlangen** mit folgenden Operationen:

- Anfügen eines Elements am **Ende** der Liste (Sonderfall: Einfügen des ersten Elements)
- Entnahme eines Elements vom **Anfang** der Liste, sofern vorhanden
- Ausgabe (Anzeigen) der kompletten Liste vom Anfang bis zum Ende

---

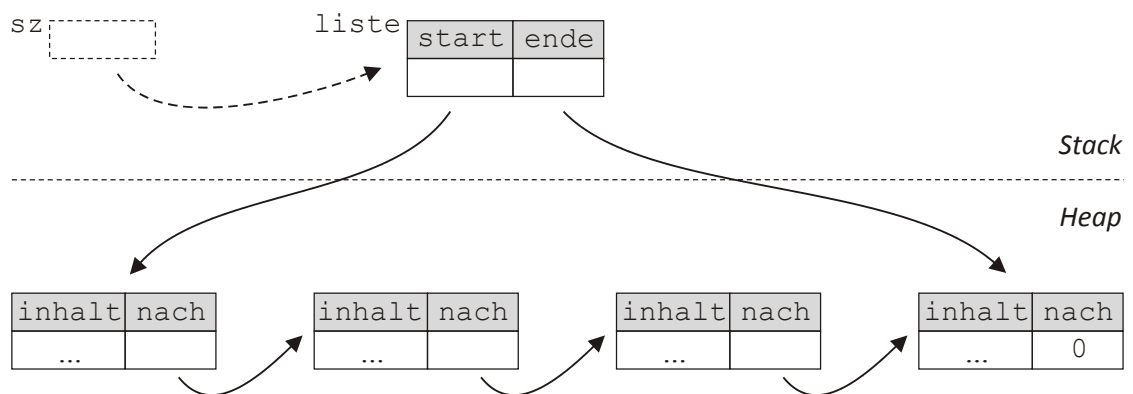
<sup>2</sup> Das Beispiel mit Fehlerbehandlung wird in der Lehrveranstaltung zum Download angeboten.



## Listen in C

Grundsätzlich könnte man eine Liste auch derart implementieren, indem man ein Array hilfsweise als Container heranzieht. Allerdings erweist sich dieses Vorgehen als etwas hölzern, da Arrays die natürliche Dynamik zur „Entnahme am Anfang“ fehlt. Wir verfolgen diese Idee daher auch nicht weiter.

Stattdessen nutzen wir für unsere Warteschlange einen anderen Ansatz, bei dem jedes Element zusätzlich einen Verweis auf seinen Nachfolger in der Liste enthält. Folgende Skizze zeigt den Aufbau; alle Zeiger verweisen vom Typ her jeweils auf die komplette, angezeigte Struktur (`struct`). Es gibt zwei Zeiger (`liste.start` und `liste.ende`), an denen die Warteschlange aufgehängt wird. Die Liste enthält im Moment 4 Elemente. (Der Zeiger `sz` spielt zunächst noch keine Rolle.)



Mit diesem Entwurf ist noch eine weitere Entscheidung zu treffen: Was ist `inhalt`? Es besteht die Möglichkeit, den gewünschten Listeneintrag selbst in das entsprechende Element aufzunehmen. Alternativ kann man hier auch „nur“ einen Zeiger auf den Eintrag zur Komponente machen. In unserem Beispiel haben wir die erste Option getroffen; dies bedeutet, dass der Listeneintrag dann auch tatsächlich in ein neues Listenelement als `inhalt` hineinkopiert werden muss.

Wie vereinbaren wir nun unsere Warteschlange und wie sehen die Zugriffsoperationen aus? In C gibt es für diesen Zweck keine vorgefertigten Datenstrukturen. Wir müssen diese daher selbst aufbauen und beginnen mit der Vereinbarung der benötigten Datentypen.

```
Bspl.: /* Header-Datei "schlange.h" */

struct Element
{
    char inhalt[31];
    struct Element *nach;
};

struct Schlange
{
    struct Element *start, *ende;
    int zaehler;
};

/* Prototypen */
extern void anhaengen (struct Schlange*, const char*);
extern int  entnehmen (struct Schlange*, char*);
extern void show      (const struct Schlange);
```

An der Vereinbarung zu `struct Element` ist besonders zu erwähnen, dass es erlaubt ist, Zeiger auf diesen Typ innerhalb der Vereinbarung als Komponenten einzusetzen. – Wie man sieht, wird hier als `inhalt` ein Zeichen-Array vereinbart. – Die Vereinbarung zu `struct Schlange` enthält neben dem `start`- und `ende`-Zeiger eine zusätzliche Komponente `zaehler` zum Mitzählen der Elemente in der Warteschlange.

Außer den Strukturtypen haben wir in der vorausgehenden Header-Datei auch Prototypen für die gewünschten Zugriffsoperationen vereinbart. In einer Programmdatei müssen wir diese nun durch entsprechende Funktionsdefinitionen realisieren.

```
Bspl.: /* Programmdatei "schlange.c" */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "schlange.h"

void anhaengen (struct Schlange *sz, const char *plus)
{
    if (sz->start == 0 || sz->ende == 0 || sz->zaehler == 0)
    {
        /* Erstes Element in leerer Liste */
        sz->start = sz->ende
            = (struct Element*) calloc (1, sizeof(struct Element));
        sz->zaehler = 1;
    }
    else
    {
        /* Hinten anfügen */
        sz->ende->nach
            = (struct Element*) calloc (1, sizeof(struct Element));
        sz->ende = sz->ende->nach;
        sz->zaehler++;
    }

    /* Echte Kopie */
    strcpy (sz->ende->inhalt, plus);
}
```

Der Parameter `sz` wird als Zeiger vereinbart (*call by reference*), da eine vorhandene Warteschlange durch die Funktion verändert werden soll. Wir haben den Parameter `sz` in der vorausgehenden Skizze bereits mit eingezeichnet. Ausgehend von diesem Zeiger wird mittels des Pfeil-Operators `->` (Kombination aus Dereferenzierung und Komponentenzugriff) auf die jeweilige Komponente zugegriffen. – Im Sonderfall der leeren Liste sollen `start`- und `ende`-Zeiger auf das mittels `calloc` neu erzeugte Element zeigen. – Andernfalls wird das neue Listenelement am Ende der Liste dynamisch erzeugt und danach der `ende`-Zeiger versetzt. – In jedem Fall wird abschließend der String-Parameter `plus` in das neue Element kopiert.

Und jetzt entnehmen wir ein Element vorne aus unserer Warteschlange.

```
Bspl.: int entnehmen (struct Schlange *sz, char *minus)
{
    if (sz->start == 0 || sz->ende == 0 || sz->zaehler == 0)
        return 0;
    else
    {
        struct Element *loesch = sz->start;
        strcpy (minus, loesch->inhalt);
        sz->start = loesch->nach;
        if (sz->start == 0)
            sz->ende = 0;
        free (loesch);
        sz->zaehler--;
        return 1;
    }
}
```

Die Funktion liefert das Ergebnis 0 („falsch“), falls die Liste leer ist; andernfalls 1 („wahr“). – Der Inhalt des ersten Elements wird auf den String-Parameter `minus` kopiert; für den erforderlichen Speicherplatz muss der Anwender der Funktion sorgen. – Danach rückt der `start`-Zeiger um ein Element nach hinten (oder wird 0, wenn es nur ein Element gibt). – Falls die Liste nun leer geworden ist, setzen wir auch den `ende`-Zeiger neu. – Der `loesch`-Zeiger verweist noch auf das zu entfernende Element, womit die `free`-Funktion den Speicher letztendlich freigeben kann.

Wir schließen unsere Implementierung ab mit einer Funktion, welche die ganze Warteschlange vom Anfang bis zum Ende durchläuft.

```
Bspl.: void show (const struct Schlange ws)
{
    struct Element *p = ws.start;
    printf ("Anzahl = %d\n[start] -> ", ws.zaehler);
    while (p)
    {
        printf ("%s -> ", p->inhalt);
        p = p->nach;
    }
    printf (" [NULL]\n\n");
}
```

Der konstante Parameter `ws` spezifiziert die auszugebende Warteschlange, ist dabei kein Zeiger im Vergleich zu `sz` bei den Beispielen zuvor. – Der Zeiger `p` zeigt auf das erste Element der Warteschlange und rückt mit jeder Wiederholung ein Element vor. Das Verfahren bricht ab, nachdem `p` beim letzten Element den Wert 0 erhalten hat. – Die Funktion funktioniert auch bei einer leeren Liste; natürlich werden dann aber keine Inhalte ausgegeben.

Unsere Vereinbarungen können wir nun in einer anderen Programmdatei nutzen, indem wir die erstellte Header-Datei importieren.

```
Bspl.: #include "schlange.h"

...

struct Schlange liste = {0};           erzeugt eine leere Warteschlange
char antwort[31];

...

anhaengen (&liste, "Anton");           fügt einen String hinten an
show (liste);

if (entnehmen (&liste, antwort))       entnimmt einen String vorne
    printf ("Entnommen: %s\n", antwort);
show (liste);
```

Wichtig: Im Sinne der vorausgegangenen Programmierung gilt eine Warteschlange als leer, wenn mindestens eine der vorhandenen drei Komponenten, welche z.B. in der `if`-Abfrage von `anhaengen` abgefragt werden, mit 0 belegt ist. Es ist daher unbedingt erforderlich, dass eine neue Liste entsprechend initialisiert wird.

## Konstruktion dynamischer Datenstrukturen

In den vorausgegangenen Beispielen haben wir das Potential des dynamischen Speichers auf verschiedene Weisen genutzt. Die Konstruktion der Dreiecksmatrix basiert auf der Möglichkeit, „dynamische Arrays“ – auch von Zeigern – im *Heap* anzulegen; der Zugriff erfolgte entsprechend durch Indizierung. Im Falle der Warteschlange haben wir hingegen einzelne Struktur-Objekte erzeugt, die durch Zeiger „verlinkt“ wurden; die Zugriffsverfahren wurden explizit programmiert.

Die von uns programmierten Datenstrukturen sind nun in dem Sinne dynamisch, dass sich Ihre Größe verändern lässt. Im Falle der Warteschlange ist dies mit den beiden Funktionen `anhaengen` und `entnehmen` offensichtlich. Für eine Größenänderung der Dreiecksmatrix haben wir hier zwar keine Funktion implementiert, es sollte jedoch nachvollziehbar sein, dass eine Verlängerung des vorhandenen „dynamischen Arrays“ von Zeigern in der ersten Dimension (mittels der Standard-Funktion `realloc`) das Anhängen neu erzeugter Zeilen für die zweite Dimension ermöglicht<sup>3</sup>.

Insofern sind die hier vorgestellten Beispiele repräsentativ. Mit den gleichen Mitteln kann man auch andere dynamische Datenstrukturen erzeugen. Die Thematik könnte sich über array-ähnliche Konstrukte sowie weitere Formen von Listen, der Konstruktion von (binären) Bäumen bis zu allgemeinen Graphen/Netzen fortentwickeln. Mit zunehmender Komplexität würden natürlich dann auch die zu implementierenden Verfahren anspruchsvoller. Im Rahmen dieser C-Einführung ist jedoch eine Fortsetzung des Themas nicht vorgesehen.

---

<sup>3</sup> Das entsprechende Beispiel wird in der Lehrveranstaltung zum Download angeboten.

## 16. Typdefinitionen

Neue Typnamen können in C mittels `typedef` vereinbart werden. Hierdurch werden allerdings keine wirklich neuen Dinge eingeführt; es werden lediglich synonyme Namen für Typen festgelegt, die auch anders angegeben werden können. Es gibt auch keine neuen semantischen Regeln: Objekte, die über `typedef`-Typen vereinbart werden, haben exakt die gleichen Eigenschaften wie Objekte, deren Vereinbarungen explizit formuliert wurden.

In einer `typedef`-Vereinbarung wird der neue Typname syntaktisch an der Position einer Variablen in einer Vereinbarung angegeben; bei Aufzähltypen und Strukturtypen bezieht sich dieser Vergleich auch auf die kombinierte Form aus Typ- und Variablen-Vereinbarung. Um die Typbezeichnungen etwas hervorzuheben, haben wir sie in den nachfolgenden Beispielen grau hinterlegt.

Wir stellen unsere bisherige Vorgehensweise der neuen Art exemplarisch gegenüber.

### Aufzähltypen

Bspl.: <i>bisher</i>	<code>enum Farbe {ROT, GELB, BLAU};</code>	← Typvereinbarung
	<code>enum Farbe func (enum Farbe);</code>	← Prototyp
	<code>enum Farbe f = BLAU;</code>	← Variablenvereinbarung
<i>jetzt</i>	<code>typedef enum {ROT, GELB, BLAU} Farbe;</code>	
	<code>Farbe func (Farbe);</code>	
	<code>Farbe f = BLAU;</code>	

### Strukturen

Bspl.: <i>bisher</i>	<code>struct Bruch {int z, n};</code>
	<code>struct Bruch func (struct Bruch);</code>
	<code>struct Bruch b = {3, 7};</code>
<i>jetzt</i>	<code>typedef struct {int z, n;} Bruch;</code>
	<code>Bruch func (Bruch);</code>
	<code>Bruch b = {3, 7};</code>

### Arrays

Bspl.: <i>bisher</i>	Typen wurden nicht namentlich festgelegt. Es wurden immer sofort die Objekte vereinbart.	
	<code>void func (int[10]);</code>	oder <code>(int[])</code> oder <code>(int*)</code>
	<code>int v[10] = {1, 2, 3, 4, 5};</code>	
<i>jetzt</i>	<code>typedef int Zehn[10];</code>	
	<code>void func (Zehn);</code>	
	<code>Zehn v = {1, 2, 3, 4, 5};</code>	

Im Zusammenspiel mit Strukturen können allerdings aufgrund unterschiedlicher Gültigkeitsbereiche Probleme entstehen. Wir greifen dazu auf ein Beispiel aus Kapitel 15 zurück und versuchen daran einmal die typedef-Umsetzung vorzunehmen.

Bspl.: *bisher*

```
struct Element
{
    char inhalt[31];
    struct Element *nach;
};
```

Hier dürfen bereits Zeiger-Komponenten vom Strukturtyp vereinbart werden.

*jetzt*

```
typedef struct
{
    char inhalt[31];
    Element *nach;
}
Element;
```

Fehler: In diesem Fall ist jedoch der Name des Strukturtyps innerhalb der geschweiften Klammern noch nicht bekannt.

Um diese Problemsituation zu umgehen, kann man beide Formen auch dual vereinbaren. Wir definieren dazu zunächst einen Strukturtyp nach bisheriger Form und erst nachfolgend den gewünschten typedef-Typ für die weitere Nutzung.

Bspl.: *dual*

```
struct Element
{
    char inhalt[31];
    struct Element *nach;
};

typedef struct Element Element;
```

Da Etiketten und typedef-Namen in verschiedenen Namensräumen (s. Kapitel 9) liegen, gibt es keinen Konflikt zwischen den Typbezeichnungen (hier: `struct Element` und `Element`). Die gewählten Namen können unterschiedlich sein, müssen es aber nicht. Die beiden resultierenden Typangaben im Beispiel sind jetzt zueinander konform.

Bspl.: *dual* Dies lässt sich analog auch in kombinierter Form vereinbaren.

```
typedef struct Element
{
    char inhalt[31];
    struct Element *nach;
}
Element;
```

In einem Beispiel des Anhangs D zeigen wir, wie man in dieser Situation das Gewünschte auf andere Weise auch durch ein Präprozessor-Makro erreichen kann.

Die vorausgehenden Beispiele beziehen sich auf nicht-elementare Datentypen. `typedef` lässt sich aber durchaus auch auf elementare Typen anwenden, insbesondere auch auf Zeigertypen.

# 17. Rechnen mit Bits und Bytes

Die nachfolgenden Sprachmittel sind stark rechnerabhängig; insbesondere die Speicherlängen in den ganzzahligen Datentypen spielen dabei eine wichtige Rolle. Programme, die diese Mittel verwenden, besitzen daher gegebenenfalls eine eingeschränkte Portabilität zu verschiedenen Prozessoren hin.

In einigen der folgenden Beispiele setzen wir stillschweigend für `int` eine Länge von 32 bit voraus, nur um die Darstellung überschaubar zu halten. Dies stimmt natürlich nicht immer mit der Realität konkreter Systeme überein.

## Grundlagen

Es gibt keinen eigenen Datentyp für Bits in C; trotzdem verfügt C über leistungsfähige Bit-Operatoren. Solche Operationen werden auf ganzzahligen Operanden angewandt, also auf `int`-Objekten, aber auch auf `char` und die `short`-, `long`-, `signed`- und `unsigned`-Varianten.

Bit-Operatoren wirken auf die einzelnen Bits der Speicherplätze. Die Zählung der Bits ist maschinenabhängig:

$b_{n-1} b_{n-2} \dots b_2 b_1 b_0$  oder umgekehrt

Bspl.: `int i=230, j=31;`

`i: 0000 0000 0000 0000 0000 0000 1110 0110`

`j: 0000 0000 0000 0000 0000 0000 0001 1111`

Das Bit mit der niedrigsten Wertigkeit (ganz rechts) hat hier die Position 0, das mit der höchsten Wertigkeit (ganz links) die Position 31.

In dieser Situation wird auch gerne alternativ auf die hexadezimale oder oktale Notation (siehe Kapitel 2) zurückgegriffen, da diese sich leichter in eine binäre Darstellung umrechnen lassen.

Bspl.: `int i=230, j=31;`

oder:

`int i=0xe6, j=0x1f;`

`1110 0110`

oder:

`int i=0346, j=037;`

`011 100 110`

Zur **Ausgabe** in hexadezimaler oder oktaler Form nutzen wir die Formatelemente `%X`, `%x` und `%o`. Die `printf`-Funktion interpretiert die zugehörigen Argumente als `unsigned int`. Bei der Anwendung auf negative Werte wird daher die interne Realisierung (Zweierkomplement) dargestellt und nicht ein vorzeichenbehafteter Wert ausgegeben! Bei `%X` werden die hexadezimalen Ziffern A bis F großgeschrieben; bei `%x` hingegen klein a bis f. Die Präfixe `0X...` bzw. `0x...` (hexadezimal) und `0...` (oktal) werden nicht mit ausgegeben.

Bei Nutzung der Formatelemente `%X`, `%x` und `%o` kann die **Eingabe** mit oder ohne Präfix erfolgen. Die Ziffern werden immer anhand des Formatelements interpretiert; ein Vorzeichen ist statthaft. Das entsprechende Argument für die `scanf`-Funktion muss vom Typ `int*` sein.

Eine dezimale Ausgabe von `int`-Werten erfolgt mittels der Formatelemente `%i` oder `%d`. Bei der Eingabe erlaubt `%i` jedoch die dezimale, hexadezimale und oktale Form, wobei die Ziffern wie in C üblich nach dem Präfix bewertet werden. Das bisher benutzte `%d` liest dagegen immer dezimal ein.

## Bit-Operatoren

Nachfolgend beschreiben wir die einzelnen Bit-Operatoren, wie sie bereits in der Tabelle des Kapitels 4 aufgelistet sind. Dort entnehme man auch die Priorität und die Assoziativität der Operatoren.

Bit-Komplement: `~ op`

Ein Ergebnisbit erhält genau dann den Wert 1, wenn das entsprechende Bit des Werts von `op` den Wert 0 hat, und 0 sonst (Einerkomplement). *Integer Promotion* findet ggf. statt.

```
Bspl.: int i=230, j;
       j = ~i;

       i:    0000 0000 0000 0000 0000 0000 1110 0110
       j:    1111 1111 1111 1111 1111 1111 0001 1001
```

Ausgabe von j liefert: -231

Bitverschiebung nach links: `op << n`

Vershoben werden die Bits des Werts von `op` um `n` Positionen nach links. Die höchstwertigen Bits gehen verloren, rechts wird mit 0 aufgefüllt. *Integer Promotion* findet ggf. statt.

```
Bspl.: int i=230, j, k;
       j = i << 1;
       k = i << 4;

       i:    0000 0000 0000 0000 0000 0000 1110 0110
       j:    0000 0000 0000 0000 0000 0001 1100 1100
       k:    0000 0000 0000 0000 0000 1110 0110 0000
```

Ausgabe von j liefert: 460  
Ausgabe von k liefert: 3680

Bitverschiebung nach rechts: `op >> n`

Vershoben werden die Bits des Werts von `op` um `n` Positionen nach rechts. Die niedrigstwertigen Bits gehen verloren, links wird bei positiven Werten mit 0 aufgefüllt, bei negativen wird systemabhängig aufgefüllt. *Integer Promotion* findet ggf. statt.

```
Bspl.: int i=230, j, k;
       j = i >> 1;
       k = i >> 4;

       i:    0000 0000 0000 0000 0000 0000 1110 0110
       j:    0000 0000 0000 0000 0000 0000 0111 0011
       k:    0000 0000 0000 0000 0000 0000 0000 1110
```

Ausgabe von j liefert: 115  
Ausgabe von k liefert: 14



Bei den vorausgegangenen Operatoren findet gegebenenfalls eine *Integer Promotion* statt, d.h. das Ergebnis der Operation ist mindestens vom Typ `int` (siehe Kapitel 3). Bei den nachfolgenden Operatoren ist dies nicht der Fall. Es können somit auch Ergebnisse der „kleineren“ ganzzahligen Datentypen entstehen.

Wir setzen in den folgenden Beispielen voraus, dass `short int` eine Länge von 16 bit hat.

Bitweises AND:  $op_1 \& op_2$

Ein Ergebnisbit erhält genau dann den Wert 1, wenn die entsprechenden Bits von  $op_1$  und  $op_2$  beide gleich 1 sind, und 0 sonst.

```
Bspl.: short i=230, j=31, k;
       k = i & j;

       i:  0000 0000 1110 0110
       j:  0000 0000 0001 1111
       k:  0000 0000 0000 0110      Ausgabe von k liefert: 6
```

Bitweises XOR:  $op_1 \wedge op_2$

Ein Ergebnisbit erhält genau dann den Wert 1, wenn von den entsprechenden Bits von  $op_1$  und  $op_2$  genau eines gleich 1 ist, und 0 sonst.

```
Bspl.: short i=230, j=31, k;
       k = i ^ j;

       i:  0000 0000 1110 0110
       j:  0000 0000 0001 1111
       k:  0000 0000 1111 1001      Ausgabe von k liefert: 249
```

Bitweises OR:  $op_1 | op_2$

Ein Ergebnisbit erhält genau dann den Wert 1, wenn mindestens eines der entsprechenden Bits von  $op_1$  und  $op_2$  gleich 1 ist, und 0 sonst.

```
Bspl.: short i=230, j=31, k;
       k = i | j;

       i:  0000 0000 1110 0110
       j:  0000 0000 0001 1111
       k:  0000 0000 1111 1111      Ausgabe von k liefert: 255
```

## Weitere Bit-Operatoren

Bei den bisherigen Bit-Operationen wurden die Werte der Operanden nicht verändert; es wurde ein Wert innerhalb eines Ausdrucks berechnet. Wie bei den arithmetischen Operatoren auch, gibt es jedoch auch Kombinationen eines (dualen) Bit-Operators mit nachfolgender Wertzuweisung:

`<<=`    `>>=`    `&=`    `^=`    `|=`

Bspl.: `int i=230;`

`i:`    `0000 0000 0000 0000 0000 0000 1110 0110`

`i <<= 1;`

wirkt wie: `i = i << 1;`

`i:`    `0000 0000 0000 0000 0000 0001 1100 1100`

Ausgabe von `i` liefert: 460

## Beispiel: Bitweise Ausgabe eines ganzzahligen Wertes

Mit folgenden Anweisungen können wir uns die Bits eines `int`-Wertes anschauen. Zunächst geben wir zur Protokollierung auch den Wert mittels der Formatelemente `%d`, `%x` und `%o` aus. Man beachte dabei die bereits erwähnten Auswirkungen bei negativen Werten.

Bspl.: `int i = 2802;`

oder auch ein anderer ganzzahliger Typ<sup>1</sup>

`int bitpos = 8 * sizeof i - 1;`

`printf ("dez.: %d\n", i);`

Ausgabe dezimal<sup>1</sup>

`printf ("hex.: %x\n", i);`

Ausgabe hexadezimal<sup>1</sup>

`printf ("okt.: %o\n", i);`

Ausgabe oktal<sup>1</sup>

`printf ("bin.: ");`

Ausgabe bitweise

`for (; bitpos >= 0; bitpos--)`

`{`

`printf ("%d", i >> bitpos & 1);`

`if (bitpos%4 == 0)`

`printf (" ");`

`}`

`printf ("\n");`

Für die bitweise Darstellung erstellen wir eine Schleife über die Bitpositionen von hoch nach niedrig; darin verschieben wir jeweils das entsprechende Bit mittels `>>` an die Position 0. Hierauf wenden wir dann die AND-Operation `&` mit dem zweiten Operanden 1 (= binär nur eine 1 an Position 0) an; dies bewirkt, dass nur das entscheidende Bit erhalten bleibt, während alle anderen Bits auf jeden Fall den Wert 0 erhalten. Der daraus resultierende ganzzahlige Wert (0 oder 1) wird dann ausgegeben.

Die `if`-Abfrage dient nur dazu, nach jeweils 4 Stellen ein Leerzeichen auszugeben.

Ausgabe:

`dez.: 2802`

`hex.: af2`

`okt.: 5362`

(32bit-System)

`bin.: 0000 0000 0000 0000 0000 1010 1111 0010`

<sup>1</sup> Details sind ggf. anzupassen.

## Bitfelder

Ein Bitfeld ist eine Menge von nebeneinanderliegenden Bits innerhalb einer Speichereinheit, die man gemeinhin als „Speicherwort“ bezeichnet. **Fast alles bei Bitfeldern ist implementationsabhängig.** Die Bits können von „links nach rechts“ oder umgekehrt angeordnet werden; auch das Überschreiten der Wortgrenzen hängt von der Implementierung ab.

Bitfelder können nur innerhalb von Strukturen als Komponenten vereinbart werden, wo sie eine Modifikation des Datentyps `int` darstellen. Es ist angeraten, Bitfelder stets auf Basis von `unsigned int` zu erstellen, da ansonsten noch ein Vorzeichenbit zu berücksichtigen ist.

**Form:** `unsigned int Name : Breite ;`

*Breite* ist dabei ein konstanter Ausdruck mit nicht-negativem ganzzahligem Wert. Mit der besonderen *Breite* 0 kann man eine Ausrichtung an der nächsten Wortgrenze verlangen. Die maximale *Breite* ist implementationsabhängig.

Die Komponente *Name* wirkt dann genau wie eine ganzzahlige Komponente mit der entsprechenden Genauigkeit. Es gibt jedoch Einschränkungen: So gibt es keine Arrays von und keine Zeiger auf Bitfelder. `sizeof` ist für Bitfelder nicht definiert.

```
Bspl.: #include <stdio.h>

struct Bitfelder
{
    unsigned int bit0 :1;
    unsigned int bit1 :1;
    unsigned int bit2 :1;
    unsigned int bit3 :1;
    unsigned int rest :28;
};

union IntBits
{
    int integer;
    struct Bitfelder bits;
};

int main ()
{
    union IntBits bsp = {13};

    printf ("als int-Wert (dez.): %d\n", bsp.integer);
    printf ("vordere Bits (dez.): %d\n", bsp.bits.rest);
    printf ("hintere Bits (bin.): ");
    printf ("%d", bsp.bits.bit3);
    printf ("%d", bsp.bits.bit2);
    printf ("%d", bsp.bits.bit1);
    printf ("%d", bsp.bits.bit0);
    printf ("\n");

    return 0;
}
```

In diesem Beispiel wird eine Union erstellt, in der sich eine `int`-Komponente und ein Bitfeld den gleichen Speicherplatz teilen. Daher kann der identische Speicherplatz auf unterschiedliche Weise interpretiert werden. Die 4 Komponenten `bit3` bis `bit0` erlauben derart den direkten Zugriff auf die letzten 4 Bits der ganzen Zahl.

Die Initialisierung `{13}` bezieht sich auf die erste `union`-Komponente (`int`).

Ausgabe:      als `int`-Wert (dez.): 13  
             vordere Bits (dez.): 0  
             hintere Bits (bin.): 1101

# 18. Sicherheitsaspekte

---

Hat man ein Programm übersetzt und erfolgreich getestet und ist auch von der algorithmischen Korrektheit des programmierten Verfahrens überzeugt, so gibt es bei der Ausführung eines C-Programms immer noch zwei große Unsicherheitsfaktoren, die einen erfolgreichen Programmlauf unterbinden können.

- Das System verhält sich doch nicht in jeder Situation so, wie erwartet.  
Eine ganze Reihe von Punkten in den vorangegangenen Kapiteln haben wir als „systemabhängig“ oder „nicht definiert“ bezeichnet. Egal, welche vermeintlichen Erfahrungen da vorliegen, der nächste Lauf – spätestens auf einem neuen System – wird es anders machen!
- Der Anwender (Benutzer) verhält sich nicht so, wie man es dachte.  
Es wird immer einen Benutzer geben, der nicht das eingibt, was man vorgesehen hatte, oder der versucht, ein Zeichen mehr als erlaubt einzugeben!

Daher geben wir nachfolgend einige Verhaltensregeln mit, die vor solch misslichen Situationen schützen sollen.

## Initialisierung von Variablen

Variablen der automatischen Speicherklasse werden nicht implizit mit Anfangswerten belegt. Die Werte ergeben sich zufällig aus dem vorhandenen Speicherinhalt, womit irgendwelche Prämissen, was deren Belegung angeht, fatal falsch sein können. (Der Forderung, wirklich jede Variable unabhängig vom algorithmischen Verfahren explizit zu initialisieren, haben wir uns in den Beispielen allerdings nicht immer angeschlossen.)

## Beschränkungen in der Arithmetik

Ist wirklich der richtige Standard-Datentyp gewählt worden? Reicht der Wertebereich? Werden vorzeichenbehaftete und vorzeichenlose ganzzahlige Typen verknüpft? Leisten die Standard-Konvertierungen das, was man benötigt? Reicht die Genauigkeit der Gleitpunkt-Zahlen? – Es gibt keine unmittelbare Reaktion des Systems, wenn bei der Programmausführung in der Arithmetik etwas „aus dem Ruder läuft“.

## Zeiger, Adressarithmetik und Arrays

Zeiger sind ohne Zweifel ein wichtiges Instrument in C. Die Gefahr, über Zeiger unkoordiniert auf Speicherplätze zuzugreifen, ist jedoch immanent, da fast keine Kontrollmechanismen<sup>1</sup> vorliegen. Das gleiche gilt sinngemäß für die Arrays (Einhaltung der Indexgrenzen). Die Situation verschärft sich noch bei Parametern, da nur Adressen und keine wirkliche Objektinformation übergeben werden. Unglücklich, wenn aufrufende und aufgerufene Funktion darauf bauen, die jeweils andere würde den Speicher bereitstellen.

Der Standard C11 bietet einige Maßnahmen zur Prüfung von Indexgrenzen an; siehe hierzu Anhang F.

---

<sup>1</sup> Die Ergebnisse der Funktionen des dynamischen Speichers sollten kontrolliert werden (ungleich 0).

## Strings

Dies ist eine Konsequenz aus dem vorangegangenen Punkt, da Strings in Arrays aus Zeichen abgelegt werden. Hier kommt es häufig zu sogenannten Pufferüberläufen, weil der Programmierer einfach nicht mit überlangen Strings gerechnet hat. Die Abhilfe kann darin bestehen, auf andere Funktionen auszuweichen, z.B. `strncpy` statt `strcpy`. – Auch hier gilt der Verweis auf C11 im Anhang F.

## Eingabe

Für eine ernsthafte Anwendung muss die Eingabe des Benutzers überwacht werden! Also erst die Zeichenkette einlesen, dann auf Gültigkeit überprüfen und letztendlich erst verarbeiten bzw. umwandeln.

Wir warnen insbesondere noch einmal vor unbedachter Anwendung der Funktion `scanf`. So kann der Aufruf `scanf ("%s", str)` – ohne Angabe einer Feldbreite – die Größe des Speichers hinter dem Argument `str` nicht bewerten, d.h. es kann hier blitzschnell zu dem schon erwähnten Pufferüberlauf bei einer überlangen Eingabe kommen. Besser arbeitet man mit einer sinnvollen Feldbreite. Gleiches gilt natürlich auch für `fscanf` und in Analogie sollte man die Funktion `fgets` gegenüber der Funktion `gets` bevorzugen. `gets` wurde obendrein im Standard C 11 entfernt.

## Ergebnistyp und Parameter von Funktionen

Immer in den Vereinbarungen den Ergebnistyp einer Funktion mit angeben und Funktionen mit einem Ergebnistyp ungleich `void` mit einer `return`-Anweisung beenden (natürlich mit Ausdruck). Im Prototypen keine leere Parameterliste angeben, wenn die Funktionsdefinition Parameter besitzt.

## Externe Bindungen

Bei einer externen Bindung eines Objektes wird immer nur der Name gebunden; es finden keine Konsistenzprüfungen statt: Das Objekt könnte z.B. vom unterschiedlichen Typ in verschiedenen Programmdateien vereinbart sein. Es kann vom gleichnamigen Typ sein, aber der Typ ist unterschiedlich definiert. Auch bei Funktionen geht es nur um den Namen; Ergebnistyp und Parameter spielen bei der Bindung keine Rolle. Daher sollten die entsprechenden Vereinbarungen stets nur in einer einzigen Header-Datei festgelegt werden. Die konsequente Nutzung von Header-Dateien schützt vor Inkonsistenzen bei externer Bindung.

## Warnungen (Compiler)

Die Warnungen des Compilers sind zu beachten, auch wenn sie im Moment nicht stören. Sogleich der Warnung nachgehen und die Ursache beseitigen. Was jetzt noch harmlos ist, kann in einer „Katastrophe“ enden.

## CERT<sup>2</sup>-Empfehlungen

Ein umfangreicher Satz von Regeln und Empfehlungen für die C-Programmerstellung liegt vor:

<https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>

---

<sup>2</sup> Computer Emergency Response Team (CERT) der Carnegie Mellon University

# A. Ein- und Ausgabe

Ein- und Ausgabe sind keine Teile der Programmiersprache C selbst, sondern werden mit der Standard-Bibliothek über die Header-Datei `<stdio.h>` zur Verfügung gestellt. Daher haben wir das Thema auch bis zu dieser Stelle weitgehend ausgeklammert. Vorläufige Hinweise zur E/A sind am Ende von Kapitel 4 und Kapitel 13 eingestellt worden, zur hexadezimalen Darstellung im Kapitel 17.

## Grundlagen

Ein **Textstrom** ist eine Folge von Zeilen, welche wiederum aus Zeichen bestehen. Eine Zeile wird aus Sicht von C mittels `'\n'` abgeschlossen; dies entspricht so nicht unbedingt den Erfordernissen der Systemumgebung. Daher wird das Zeichen gegebenenfalls bei der E/A noch systemintern umgesetzt.

Neben den Textströmen gibt es auch noch **binäre Datenströme**, die einfach nur eine Folge von Bytes darstellen.

Jeder Strom muss mit einer Datei oder einem peripheren Gerät verknüpft sein; die Verbindung muss geöffnet werden und wird zu gegebener Zeit auch wieder geschlossen. Mit dem Start eines Programmes sind die drei Textströme `stdin`, `stdout` und `stderr` bereits geöffnet.

Eine Datei wird über einen sogenannten FILE-Zeiger verwaltet. Dieser zeigt auf eine Struktur, die Informationen über die Datei enthält. Wichtig: Wir vereinbaren immer nur solche Zeiger, niemals Objekte vom entsprechenden Typ selbst.

## Dateioperationen

In diesen Bereich fallen insgesamt 10 Funktionen, von denen uns hier aber nur die Funktionen zum Öffnen und Schließen einer Datei interessieren.

<code>FILE * <b>fopen</b> (const char *filename, const char *mode);</code>	zum Öffnen
<code>int <b>fclose</b> (FILE *stream);</code>	zum Schließen

Für den Parameter `mode` setzen wir als Argument ein:

`"r"` Textdatei zum **Lesen** öffnen.

`"w"` Textdatei zum **Schreiben** öffnen; der alte Inhalt wird ggf. weggeworfen.

`"a"` Textdatei zum **Anfügen** öffnen; der alte Inhalt bleibt ggf. erhalten.

Es gibt weitere Modi, insbesondere für Binärdateien den Zusatz `b`, also zum Beispiel `"rb"`.

```
Bspl.: FILE *eingabeDatei = fopen ("eingabe.txt", "r");
...
fclose (eingabeDatei);
```

Hier öffnen wir eine Datei mit Namen `eingabe.txt` zum Lesen aus dem gleichen Verzeichnis, in dem unser ausführbares Programm liegt. Ist die Datei nicht vorhanden, liefert die Funktion `fopen` einen NULL-Zeiger zurück. – Die untere Anweisung schließt die Datei wieder; danach ist kein Lesen mehr möglich.

## Formatierte E/A

### Ausgabe

Folgende Funktionen ermöglichen die Ausgabe unter Kontrolle eines Formatstrings (Parameter `ft`).

<code>int fprintf (FILE *stream, const char *ft, ...);</code>	Ausgabe in Datei
<code>int printf (const char *ft, ...);</code>	Ausgabe nach <code>stdout</code>
<code>int sprintf (char *s, const char *ft, ...);</code>	Ausgabe in String

Die Funktionen geben den Formatstring<sup>1</sup> `ft` aus, wobei enthaltene Formatelemente vorher ausgewertet werden. Wichtig sind die Formatelemente, die die Einsetzung des nächstfolgenden Arguments veranlassen. – Das Funktionsergebnis ist die Anzahl der geschriebenen Zeichen.

Der Aufruf der Funktion `printf (...)` hat stets die gleiche Wirkung wie: `fprintf (stdout, ...)`  
Der Standard-Strom `stdout` meint im Regelfall die Bildschirmausgabe.

`sprintf` ist im eigentlichen Sinne keine Ausgabefunktion, denn sie schreibt das Ergebnis in einen String und schließt diesen auch mit `'\0'` ab, verhält sich aber ansonsten wie die anderen Funktionen. Hiermit lässt sich z.B. auch ein Zahlwert in einen String umwandeln.

### Eingabe

Folgende Funktionen ermöglichen die Eingabe unter Kontrolle eines Formatstrings (Parameter `ft`).

<code>int fscanf (FILE *stream, const char *ft, ...);</code>	Eingabe aus Datei
<code>int scanf (const char *ft, ...);</code>	Eingabe aus <code>stdin</code>
<code>int sscanf (char *s, const char *ft, ...);</code>	Eingabe aus String

Diese Funktionen interpretieren die Eingabe anhand des Formatstrings<sup>1</sup> `ft` und legen die umgewandelten Werte hinter den nachfolgenden Argumenten ab, die alle typgeeignete L-Werte liefern müssen. Die Argumente können durch Anwendung des Adressoperators `&` auf die zu belegenden Variablen gebildet werden oder entsprechend vorhandene Zeiger sein. – Als Funktionsergebnis wird die Anzahl der erfolgreich gelesenen Werte zurückgeliefert. Wird sofort das Dateiende erreicht, ist das Ergebnis `EOF`<sup>2</sup>.

Der Aufruf der Funktion `scanf (...)` hat stets die gleiche Wirkung wie: `fscanf (stdin, ...)`  
Der Standard-Strom `stdin` meint im Regelfall die Gerätetastatur.

`sscanf` ist im engeren Sinne keine Eingabefunktion, denn sie liest die Eingabe aus einem String. Ansonsten verhält sie sich aber wie die beiden anderen Funktionen. Hiermit lässt sich z.B. auch ein String in einen Zahlwert umwandeln, sofern der String korrekt formatiert ist.

<sup>1</sup> Die Wirkung des Formatstrings wurde am Ende von Kapitel 4 behandelt.

<sup>2</sup> Mehr dazu im Abschnitt über zeichenweise E/A.



Bspl.: Aus einer Textdatei wird eine Folge von Gleitpunkt-Zahlen gelesen und in eine andere Textdatei die jeweils berechneten Sinus-Werte geschrieben.

```
#include <math.h>
#include <stdio.h>

int main ()
{
    FILE *in = fopen ("eingabe.txt", "r");
    if (in == 0)
        printf ("Problem: Die Eingabe-Datei existiert nicht!\n");
    else
    {
        FILE *out = fopen ("ausgabe.txt", "w");
        double d;
        int counter = 0;
        while (fscanf(in,"%lf",&d) == 1)
        {
            fprintf (out, "sin(%f) = %f\n", d, sin(d));
            counter++;
        }
        fclose (in);
        fclose (out);
        printf ("Erfolg: Berechnet wurden %d Werte.\n", counter);
    }
    return 0;
}
```

Die `fscanf`-Funktion liefert genau dann den Wert 1 zurück, wenn ein Wert für `d` erfolgreich gelesen wurde. (Es wird hier explizit auf die Anzahl 1 und nicht auf „wahr“ hin überprüft.)

## Formatelemente

Die Formatelemente sind ausführlich beschrieben unter der Quelle, die wir auch zu Beginn des Anhangs C aufführen, jeweils bei der entsprechenden Funktion. Hier direkte Links:

<http://www.cplusplus.com/reference/cstdio/printf>

<http://www.cplusplus.com/reference/cstdio/scanf>

Die von uns vorgestellten Formatelemente für die Ausgabe können teilweise noch mit Steuerzeichen (*flags*), einer Feldbreite (*width*), einer Genauigkeit (*precision*) und einer Länge (*length*) versehen werden. Welche Optionen möglich sind, hängt vom jeweiligen Typzeichen (*specifier*) ab.

**Form:** `%[flags][width][.precision][length]specifier`

Die eckigen Klammern kennzeichnen hier optionale Teile. Sie sind nicht Bestandteile des Formatelements. – Unter *flags* können mehrere Zeichen vorkommen, wozu auch das Leerzeichen gehört. Ansonsten dürfen hier keine Leerzeichen vorkommen.

Bspl.: Ausgabe von Werten mit verschiedenen Formatelementen  
(In der Ausgabe repräsentiert das Symbol `_` jeweils ein Leerzeichen.)

```
int i = 1234;      double d = 123.456;
```

Argumente <code>printf</code>	Ausgabe	Anmerkung
<code>("%d", i)</code>	1234	Standard-Ausgabe
<code>("%8d", i)</code>	____1234	Feldbreite 8, rechtsbündig
<code>("%-8d", i)</code>	1234____	Feldbreite 8, linksbündig
<code>("%+8d", i)</code>	____+1234	rechtsbündig, immer mit Vorzeichen
<code>("%-+8d", i)</code>	+1234____	linksbündig, immer mit Vorzeichen
<code>("%- 8d", i)</code>	_1234____	linksbündig, freier Platz für Vorzeichen
<code>("%- 8d", -i)</code>	-1234____	wie zuvor, hier aber negatives Argument
<code>("%08d", i)</code>	00001234	mit führenden Nullen, Feldbreite 8
<code>("%3d", i)</code>	1234	Standard-Ausgabe, weil Feldbreite zu klein
<code>("%f", d)</code>	123.456000	Standard-Ausgabe
<code>("%.4f", d)</code>	123.4560	4 Stellen Genauigkeit
<code>("%12.3f", d)</code>	____123.456	Feldbreite 12, 3 Stellen Genauigkeit
<code>("%-12.3f", d)</code>	123.456____	wie zuvor, aber linksbündig
<code>("%+12.3f", d)</code>	____+123.456	rechtsbündig, immer mit Vorzeichen
<code>("%-+12.3f", d)</code>	+123.456____	linksbündig, immer mit Vorzeichen
<code>("%- 12.3f", d)</code>	_123.456____	linksbündig, freier Platz für Vorzeichen
<code>("%- 12.3f", -d)</code>	-123.456____	wie zuvor, hier aber negatives Argument
<code>("%8.1f", d)</code>	____123.5	rechtsbündig, Rundung auf 1 Dezimalstelle
<code>("%3f", d)</code>	123.456000	Standard-Ausgabe, weil Feldbreite zu klein
<code>("%e", d)</code>	1.234560e+002	Ausgabe in Exponentialform
<code>("%E", d)</code>	1.234560E+002	statt <code>e</code> nun <code>E</code>
<code>("%12.3E", d)</code>	____1.235E+002	Feldbreite 12, 3 Stellen Genauigkeit
<code>("%-12.3E", d)</code>	1.235E+002__	wie zuvor, aber linksbündig
<code>("%+12.3E", d)</code>	____+1.235E+002	rechtsbündig, immer mit Vorzeichen
<code>("%-+12.3E", d)</code>	+1.235E+002__	linksbündig, immer mit Vorzeichen
<code>("%10E", d)</code>	1.234560E+002	Standard-Ausgabe, weil Feldbreite zu klein

Die Länge des Exponentialteils ist systemabhängig, beträgt aber mindestens zwei Ziffern.

Angaben zur Länge (*length*) sind auf den hier verwendeten Datentypen `int` und `double` nicht anwendbar. So ist `h` bei der Ausgabe vorgesehen für die Typen `short` oder `unsigned short`, `l` für `long` oder `unsigned long`, `L` für `long double`.

## Zeichenorientierte E/A

### Dateiende

Bei der sequentiellen Verarbeitung einer Datei wird man irgendwann auf das Dateiende stoßen. Unterschiedliche Programmiersprachen gehen unterschiedlich mit dieser Situation um. So kann z.B. `EOF` (*end of file*) ein Zustand einer Datei sei, welcher abgefragt werden kann; alternativ vor dem Lesen oder auch nach einem Leseversuch. Ebenso ist es möglich, dass `EOF` ein spezielles Zeichen ist, welches beim Lesen angeliefert wird. Verfahren zur Dateibearbeitung lassen sich daher nicht immer 1:1 von einer Programmiersprache zu einer anderen übertragen.

In C ist `EOF` ein ganzzahliger Wert, der als Konstante in der Header-Datei `<stdio.h>` definiert ist. Da die (ganzzahligen) `char`-Datentypen für die Zeichen gemäß Codetabelle vorgesehen sind, ist hier kein Platz mehr für `EOF`; daher ist `EOF` vom Typ `int`. – **Diesem Vorgehen sind auch die Funktionen aus der Standard-Bibliothek angepasst.** – Deshalb arbeiten wir ebenfalls mit dem Zahltyp `int`, was irritieren mag, wo es sich doch dem Verständnis nach um Zeichenverarbeitung handelt. Erst nach Ausschluss des `EOF`-Vorkommens können wir, sofern dann noch gewünscht, eine Konvertierung z.B. zum Datentyp `unsigned char` hin vornehmen.

### Ausgabe und Eingabe

Folgende Funktionen ermöglichen die Ausgabe und Eingabe eines Zeichens (nicht einer Zahl!).

<code>int fputc</code>	<code>(int c, FILE *stream);</code>	Ausgabe in Datei
<code>int putchar</code>	<code>(int c);</code>	Ausgabe nach stdout
<code>int fgetc</code>	<code>(FILE *stream);</code>	Eingabe aus Datei
<code>int getchar</code>	<code>(void);</code>	Eingabe aus stdin

Die Eingabefunktionen liefern das gefundene Zeichen (auch Leerzeichen, Tabulatoren etc.) als Funktionsergebnis zurück; alle Funktionen liefern `EOF` bei Dateiende oder einem Fehler. – Intern interpretieren die Funktionen das Zeichen als `unsigned char`.

Einzelne Zeichen lassen sich alternativ auch mit der formatierten E/A über das Formatelement `%c` ausschreiben und einlesen.

### Bspl.: Erstellen einer Dateikopie

```
#include <stdio.h>

int main ()
{
    FILE *in = fopen ("quelle.txt", "r");
    if (in == 0)
        printf ("Problem: Die Eingabe-Datei existiert nicht!\n");
    else
    {
        FILE *out = fopen ("kopie.txt", "w");

        int c;
        int counter = 0;
    }
}
```

Wichtig: `int` statt Zeichentyp!

```

while ((c=fgetc(in)) != EOF)
{
    fputc (c, out);
    counter++;
}

fclose (in);
fclose (out);

printf ("Erfolg: Kopiert wurden %d Zeichen.\n", counter);
}

return 0;
}

```

Die `fgetc`-Funktion liefert im Erfolgsfall das gelesene Zeichen zurück; dieses wird auf `c` abgespeichert und ist ungleich `EOF`. `c` ist vom Typ `int` vereinbart worden und kann somit am Dateiende auch den Wert `EOF` aufnehmen. Die `while`-Schleife beendet sich dann.

## Zeilenorientierte E/A

Folgende Funktionen ermöglichen die Ausgabe und Eingabe von Strings unter Beachtung der Zeilenstruktur.

<code>int</code>	<b><code>fputs</code></b> ( <code>const char *s</code> , <code>FILE *stream</code> );	Ausgabe in Datei
<code>int</code>	<b><code>puts</code></b> ( <code>const char *s</code> );	Ausgabe nach <code>stdout</code>
<code>char *</code>	<b><code>fgets</code></b> ( <code>char *s</code> , <code>int n</code> , <code>FILE *stream</code> );	Eingabe aus Datei
<del><code>char *</code></del>	<del><b><code>gets</code></b> (<code>char *s</code>);</del> <small>C90-C99</small>	<del>Eingabe aus <code>stdin</code></del>

`puts` schließt jede Ausgabe automatisch mit einem Zeilenende `'\n'` ab, `fputs` hingegen nicht. Die beiden Ausgabefunktionen liefern `EOF` bei einem Fehler zurück.

`fgets` liest `n-1` Zeichen in das Zeichen-Array `s` ein, hört aber vorher auf, wenn ein Zeilenende gefunden wurde. Das Zeilenende wird gegebenenfalls mit abgelegt und der String mittels `'\0'` abgeschlossen. Der String steht im Erfolgsfall über den Parameter `s` zur Verfügung und als Funktionsergebnis. Im Fehlerfall oder bei einem Dateiende ist das Ergebnis der `NULL`-Zeiger.

~~Im Gegensatz hierzu besitzt `gets` keinen Längenparameter `n` und liest somit bis zum Zeilenende. Das Zeilenende wird im String `s` durch `'\0'` ersetzt. `gets` wurde mit dem Standard C11 gestrichen!~~

Wir erinnern an das abschließende Beispiel aus Kapitel 13, für das die Funktion `fgets` eine mögliche Alternative darstellt. Im Unterschied zu `scanf`/`fscanf` liest die Funktion `fgets` auch Leerraum aus der Eingabe mit ein. Ansonsten beachte man die Sicherheitshinweise aus Kapitel 18.

## Weitere Funktionen

Die Standard-Bibliothek stellt noch mehr zur E/A bereit. Hierzu gehören weitere Dateifunktionen (z.B. Löschen, Umbenennen von Dateien), der Bereich der direkten E/A, das Positionieren in Dateien und eine Fehlerbehandlung über die vorgestellten Möglichkeiten hinaus.

## B. Zufallszahlen

---

Als Pseudozufall wird bezeichnet, was scheinbar zufällig und nicht vorhersehbar erscheint, in Wirklichkeit jedoch berechenbar ist. In diesem Sinne erzeugen Pseudozufallszahlen-Generatoren pseudozufällige Zahlen.

So lassen sich in C pseudozufällige Zahlen über die Funktionen `rand` und `srand` generieren. Diese Funktionen sind deklariert in der Header-Datei `<stdlib.h>`, vgl. dazu Anhang C.

Durch wiederholten Aufruf von `rand` erhält man dabei als Funktionsergebnis jeweils das nächste Element einer Folge von Pseudozufallszahlen.

Ohne weitere Vorkehrungen wird dieser Aufruf jedoch bei mehrfachem Start eines Programms stets die identische Folge von Zahlen liefern. Die Funktion `srand` dient daher der Initialisierung des Generators mit einem *seed* (Saatkorn). Wird diese Initialisierung mit einem wirklich zufälligen Wert – beispielsweise anhand der aktuellen Uhrzeit – durchgeführt, so berechnen sich hieraus hinreichend unterschiedliche Zahlenfolgen.

Es sei ausdrücklich darauf hingewiesen, dass die Funktion `rand` natürliche Pseudozufallszahlen im Bereich zwischen 0 und `RAND_MAX`<sup>1</sup> berechnet. Je nach Aufgabenstellung müssen diese Werte nachfolgend noch in das gewünschte Intervall transformiert werden.

```
Bspl.: #include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main ()
{
    srand (time(0));

    for (int i = 1; i <= 5; i++)
    {
        int zufall = rand ();
        printf ("Generiert [0..%d]: %d\n", RAND_MAX, zufall);

        int zufall_1_6 = 1 + zufall % 6;
        printf ("Transformiert [1..6]: %d\n", zufall_1_6);
    }

    return 0;
}
```

In diesem Programm (C99/C11) wird zunächst die aktuelle Uhrzeit über die Funktion `time` als *seed* für den Generator herangezogen. Danach wird wiederholt eine Pseudozufallszahl berechnet und zusammen mit `RAND_MAX` ausgegeben.

Exemplarisch wird der jeweilige Wert zusätzlich noch in den Bereich 1 bis 6 transformiert und ebenfalls ausgegeben.

---

<sup>1</sup> `RAND_MAX` ist ein PP-Makro (siehe Anhang D), das zu einem konstanten, ganzzahligen Ausdruck mindestens vom Wert 32767 expandiert wird.

**Warnung zum Pseudozufallszahlen-Generator**

- Der C11-Standard selbst rät für gegebene Fälle zur Verwendung anderer Generatoren:  
*There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.*

Diese Empfehlung gilt inhaltlich auch für die vorausgegangenen Standards.

## C. Standard-Bibliothek im Überblick

Im Standard C90 wurden 15 Header-Dateien zur Standard-Bibliothek beschrieben, die zur Grundausrüstung eines C-Systems gehören. In Laufe der Zeit kamen weitere Header-Dateien hinzu und die vorhandenen Bereiche wurden zum Teil auch erweitert.

<code>assert.h</code>		Diagnose / Zusicherungen ( <i>assertions</i> )
<code>complex.h</code>	C99, C11 <sub>opt</sub>	komplexe Zahlen
<code>ctype.h</code>		Test und Umwandlung von Zeichen
<code>errno.h</code>		Fehlercode-Behandlung
<code>fenv.h</code>	C99	Einstellungen zu Gleitpunktzahlen
<code>float.h</code>		Konstanten zu Gleitpunkt-Typen
<code>inttypes.h</code>	C99	Format-Konvertierungen für ganzzahlige Typen
<code>iso646.h</code>	C95	PP-Makros für logische und bitweise Operatoren
<code>limits.h</code>		Konstanten zu ganzzahligen Typen
<code>locale.h</code>		Regionalisierung / Lokalisierung
<code>math.h</code>		mathematische Funktionen
<code>setjmp.h</code>		globale Sprungbehandlung
<code>signal.h</code>		externe (Betriebssystem-)Signalbehandlung
<code>stdalign.h</code>	C11	Speicher-Ausrichtung von Objekten
<code>stdatomic.h</code>	C11 <sub>opt</sub>	atomare Operationen zwischen Threads
<code>stdarg.h</code>		Behandlung variabler Parameterlisten
<code>stdbool.h</code>	C99	PP-Makros für Wahrheitswerte
<code>stddef.h</code>		Definition zusätzlicher Typen
<code>stdint.h</code>	C99	ganzzahlige Typen (systemunabhängig)
<code>stdio.h</code>		Ein- und Ausgabe
<code>stdlib.h</code>		Diverses
<code>stdnoreturn.h</code>	C11	Definition des PP-Makros <code>noreturn</code>
<code>string.h</code>		Zeichenkettenverarbeitung
<code>tgmath.h</code>	C99	mathematische Funktionen (typgenerisch)
<code>threads.h</code>	C11 <sub>opt</sub>	Unterstützung für Multithreading
<code>time.h</code>		Uhrzeit und Datum
<code>uchar.h</code>	C11	Typen und Funktionen für Unicode-Zeichen
<code>wchar.h</code>	C95	erweiterte Zeichensätze (> 1 Byte), Unicode
<code>wctype.h</code>	C95	wie <code>ctype.h</code> , aber für erweiterte Zeichensätze

- Benötigte Header-Dateien sind mittels der PP-Direktive

```
#include <Name>
```

in das jeweilige C-Programm einzufügen.

- Weitergehende Beschreibungen aller Header-Dateien sind an diesen Stellen erhältlich:

[http://en.wikipedia.org/wiki/C\\_standard\\_library](http://en.wikipedia.org/wiki/C_standard_library)

<http://www.cplusplus.com/reference/cstdlib>

Man erhält nicht nur eine Auflistung der Inhalte, sondern nach Klick auf einen Funktionsnamen auch eine ausführliche Beschreibung der meisten Funktionen.

Dies ist an dieser Stelle kaum zu übertreffen. Wir verzichten daher auch auf eine vollständige Wiedergabe und beschränken uns auf einige wenige Anmerkungen.

- Folgende Header-Dateien wurden – zumindest teilweise – an anderer Stelle beschrieben.

- |                         |                           |            |
|-------------------------|---------------------------|------------|
| ◦ <code>math.h</code>   | mathematische Funktionen  | Kapitel 4  |
| ◦ <code>stdio.h</code>  | Ein- und Ausgabe          | Anhang A   |
| ◦ <code>string.h</code> | Zeichenkettenverarbeitung | Kapitel 13 |

### ctype.h – Test und Umwandlung von Zeichen

Diese Header-Datei beschreibt Funktionen für die Bearbeitung einzelner Zeichen (`char`). Es mag etwas irritieren, dass die Parameter dem Typ `int` angehören. Dies ist dem Umstand geschuldet, dass auch noch der Wert `EOF` (*end of file*) – nicht mit einem realen Zeichen zu verwechseln – als Argument akzeptiert werden soll. (Man vergleiche Anhang A.)

<code>int isalnum (int c);</code>	<code>isalpha(c)</code> oder <code>isdigit(c)</code> ist wahr
<code>int isalpha (int c);</code>	<code>isupper(c)</code> oder <code>islower(c)</code> ist wahr
<code>int iscntrl (int c);</code>	Test auf Steuerzeichen
<code>int isdigit (int c);</code>	Test auf dezimale Ziffer
<code>int isgraph (int c);</code>	Test auf sichtbares (druckbares) Zeichen, kein Leerzeichen
<code>int islower (int c);</code>	Test auf Kleinbuchstabe (aber kein Umlaut oder ß)
<code>int isprint (int c);</code>	Test auf sichtbares (druckbares) Zeichen, auch Leerzeichen
<code>int ispunct (int c);</code>	Test auf sichtbares Zeichen, kein Leerzeichen, Buchstabe, Ziffer
<code>int ispace (int c);</code>	Test auf Leerraum
<code>int isupper (int c);</code>	Test auf Großbuchstabe (aber kein Umlaut)
<code>int isxdigit (int c);</code>	Test auf hexadezimale Ziffer
<code>int tolower (int c);</code>	Umwandlung in Kleinbuchstabe
<code>int toupper (int c);</code>	Umwandlung in Großbuchstabe



### float.h – Konstanten<sup>1</sup> zu Gleitpunkt-Typen

Die Konstanten beschreiben für die Datentypen `float`, `double` und `long double` die Wertebereiche und die Genauigkeiten von Mantisse und Exponent.

FLT_RADIX, FLT_ROUNDS			
FLT_DIG, FLT_EPSILON, FLT_MANT_DIG, FLT_MAX	für Typ <code>float</code>		
FLT_MAX_EXP, FLT_MIN, FLT_MIN_EXP			
DBL_DIG, DBL_EPSILON, DBL_MANT_DIG, DBL_MAX	für Typ <code>double</code>		
DBL_MAX_EXP, DBL_MIN, DBL_MIN_EXP			
LDBL_DIG, LDBL_EPSILON, LDBL_MANT_DIG, LDBL_MAX	für Typ <code>long double</code>		
LDBL_MAX_EXP, LDBL_MIN, LDBL_MIN_EXP			

### limits.h – Konstanten<sup>1</sup> zu ganzzahligen Typen

Für alle vorzeichenbehafteten Typen sind hier der kleinste und der größte Wert definiert, für die vorzeichenlosen Typen natürlich nur der größtmögliche Wert.

CHAR_BIT, CHAR_MAX, CHAR_MIN	für den Typ <code>char</code>
INT_MAX, INT_MIN	für den Typ <code>int</code>
LONG_MAX, LONG_MIN	für den Typ <code>long</code>
SCHAR_MAX, SCHAR_MIN	für den Typ <code>signed char</code>
SHRT_MAX, SHRT_MIN	für den Typ <code>short</code>
UCHAR_MAX	für den Typ <code>unsigned char</code>
UINT_MAX	für den Typ <code>unsigned int</code>
ULONG_MAX	für den Typ <code>unsigned long</code>
USHRT_MAX	für den Typ <code>unsigned short</code>

<sup>1</sup> Die Konstanten werden als Makros für den Präprozessor festgelegt.

Ein Beispiel dazu im Kapitel 2, weitere Details zum Präprozessor und zu Makros im Anhang D.

## stdlib.h – Diverses

Funktionen zu folgenden Bereichen: Prozesskontrolle, Speicherverwaltung, Typkonvertierung, Zufallszahlen, Sortieren, Suchen und noch etwas Mathematik

Die nachfolgende Tabelle ist nicht vollständig!

void	<b>exit</b>	(int status);	Programmende mit Status
int	<b>system</b>	(const char *command);	Ausführung eines externen Kommandos
void * <b>calloc</b> (size_t n, size_t size); void * <b>malloc</b> (size_t size); void * <b>realloc</b> (void *p, size_t size);			Speicherreservierung, siehe Kapitel 15
void	<b>free</b>	(void *p);	Freigabe des reservierten Speichers
double	<b>atof</b>	(const char *str);	Umwandlung String nach double
int	<b>atoi</b>	(const char *str);	Umwandlung String nach int
long	<b>atol</b>	(const char *str);	Umwandlung String nach long
int	<b>rand</b>	(void);	Zufallszahlen, siehe Anhang B
void	<b>srand</b>	(unsigned int seed);	
int	<b>abs</b>	(int n);	Absolutbetrag für ganzzahlige Typen
long	<b>labs</b>	(long n);	

# D. Präprozessor-Direktiven

---

## Grundlagen

Vor der eigentlichen Übersetzung bearbeitet ein Präprozessor (Software) die Programmdateien, wodurch erst die eigentlichen Übersetzungseinheiten entstehen, die vom Compiler verarbeitet werden (siehe Kapitel 1).

Die Bearbeitung erfolgt logisch in mehreren verschiedenen Durchläufen hintereinander. Einige Schritte sind dabei in der Praxis von geringerer Bedeutung, so werden beispielsweise irgendwann alle Kommentare entfernt. In vereinfachter Form stellt sich der Ablauf wie folgt dar.

- Folgt einem inversen Schrägstrich \ das Zeilenende, so werden beide entfernt. Hierdurch werden Zeilen verbunden, noch bevor die Eingabesymbole analysiert werden.
- Danach werden Direktiven befolgt und Makros expandiert.
- Benachbarte Zeichenketten (Literele) werden ganz am Schluss zu einem String zusammengefügt.

Bspl.: 

```
prin\  
tf ("hello,"  
    " world\n");
```

So sollte man die Anweisung aus unserem ersten Beispiel sicher nicht programmieren, aber der Präprozessor wird aufgrund des inversen Schrägstrichs \ die beiden ersten Zeilen und damit den Namen `printf` zusammensetzen. In einem nachfolgenden Schritt werden auch die beiden aufeinanderfolgenden Strings zu einem zusammengefügt.

Der Präprozessor wird ferner gesteuert durch Direktiven, welche in einer Zeile mit dem Zeichen # beginnen; vor # können auch noch Leerzeichen stehen. Umfasst eine Direktive mehrere Zeilen, so sind diese mittels \ zu verbinden. Die PP-Direktiven unterliegen ansonsten eigenen syntaktischen Regeln.

Die Wirkung einer PP-Direktive hält grundsätzlich bis zum Ende der Quelldatei bzw. Übersetzungseinheit an, unabhängig zu den Regeln für Gültigkeitsbereiche in C.

## Einfügen von Dateien

**Form:** `#include <Dateiname>`  
oder  
`#include "Dateiname"`

Eine Direktive dieser Form wird durch den Inhalt der angegebenen Datei ersetzt. Wir haben dies bisher zum Einsetzen von Header-Dateien genutzt.

In der ersten Form wird die Datei an einer Stelle gesucht, die implementierungsabhängig ist; im zweiten Fall wird vorher auch noch bei der ursprünglichen Programmdatei, also im Regelfall im gleichen Verzeichnis, gesucht.

## Makros und deren Expansion

**Form:** `#define Name Ersatztext`

Unter dem *Namen* wird ein sogenanntes **Makro** vereinbart. Dieses veranlasst, dass der Präprozessor nachfolgende Vorkommen von *Name* durch *Ersatztext* ersetzt; Leerraum vor und nach dem *Ersatztext* wird dazu entfernt. Der *Name* unterliegt den Regeln für Eingabesymbole (Kapitel 1) und der Ersatz erfolgt auch nur in diesem Kontext. Dies bedeutet speziell, dass im Programmtext innerhalb von Zeichenketten keine Ersetzung erfolgt.

Es war einmal weitgehend üblich, für den *Namen* keine Kleinbuchstaben zu verwenden. Allerdings wird bereits mit dem neueren Standard C99 dieses Verhalten unterlaufen (siehe z.B.: `bool`). Unsere Empfehlungen aus Kapitel 1 sehen die Großschreibung für Makros jedoch weiterhin vor.

Die Wirkung des Makros kann beendet werden durch:

**Form:** `#undef Name`

Eine der häufigsten Anwendungen für Makros besteht darin, allzeit konstante Werte, die im Quelltext mehrfach benötigt werden, an einer einzigen Stelle festzulegen.

Bspl.: `#define ANZAHL 1000`

Im folgenden Programmcode wird dann `ANZAHL` vom Präprozessor durch `1000` ersetzt.

```
double a[ANZAHL];
for (i = 0; i < ANZAHL; i++) ...
```

Makros stehen somit in einer gewissen Konkurrenz zu Vereinbarungen von symbolischen Konstanten.

Bspl: `#define PI 3.1415`

Das Makro bewirkt, dass überall, wo der Name `PI` auftaucht, dieser vor der eigentlichen Übersetzung durch die Konstante (Literal) `3.1415` ersetzt wird.

Eine Alternative dazu ist:

```
const double pi = 3.1415;
```

Diese `const`-Vereinbarung führt den Namen für eine (symbolische) Konstante `pi` ein und initialisiert diese mit dem angegebenen Wert.

Wir haben die `const`-Notation allgemein bevorzugt, weil diese in C auch in anderen Situationen Verwendung findet, z.B. bei konstanten Parametern, Arrays, Komponenten, ...

Symbolische Konstanten sind jedoch keine konstanten Ausdrücke (Kapitel 4); Makros erfüllen nach der Expansion dagegen diese Bedingung, sofern der Ersatztext einen konstanten Ausdruck darstellt.

Allerdings können symbolische Konstanten (und sogar Variablen) ab dem Standard C99 beispielsweise auch als Längenangabe für ein Array der automatischen Speicherklasse benutzt werden. Ein Vorteil beim Gebrauch eines Makros ist damit für derartige Zwecke nicht mehr erkennbar.

Gerne wird auch die etwas längliche Bezeichnung für einen Strukturtyp durch ein Makro ersetzt.

```
Bspl.: #define ELMNT struct Element
      ELMNT
      {
          char inhalt[31];
          ELMNT *nach;
      };
```

Die Expansion des Makros erzeugt hier eine Vereinbarung aus dem Kapitel 15. Nimmt man die Makro-Direktive in die dortige Header-Datei mit auf, kann man in dem Beispiel den Makronamen `ELMNT` wie einen Typnamen verwenden. Das Problem, welches wir im Kapitel 16 für `typedef`-Vereinbarungen beleuchtet haben, entsteht dabei nicht.

Die Vereinbarung des Strukturtyps alleine wirkt auf den ersten Blick syntaktisch falsch. Erst mit dem Wissen um die Ersetzung erkennt man die Korrektheit. Daher hat die Großschreibung von Makronamen durchaus einen praktischen Sinn.

Die Makro-Expansion wirkt textuell; man kann sie also auf beliebige Programmteile anwenden.

```
Bspl: #define FOREVER while (1)

      Nun kann man eine Endlosschleife wie folgt programmieren.

      FOREVER
      {
          ...
      }
```

Die `#define`-Direktive gibt es auch in parametrisierter Form. Diese birgt jedoch einige Fallstricke, die hier nicht behandelt werden sollen. Nur zur Demonstration das nachfolgende Beispiel.

```
Bspl.: #define MINIMUM(a, b) ((a)<(b) ? (a) : (b))
```

Auch wenn dies irgendwie nach einer Funktion aussieht, durch dieses Makro wird der Programmtext direkt expandiert, indem der angegebene bedingte Ausdruck eingefügt wird. Die Parameter des Makros werden textuell durch die Argumente ersetzt.

So wird: `MINIMUM(5*x, x+y)`

ersetzt durch: `((5*x)<(x+y) ? (5*x) : (x+y))`

Die auffällige Klammerung wurde in der Makro-Direktive pauschal eingesetzt, damit keine Prioritätsprobleme in und um den Ausdruck entstehen, speziell wenn einmal Ausdrücke mit niederwertigeren Operatoren als Argumente vorkommen.

Man beachte, dass dieses Makro dennoch seine Tücken hat.

So wird: `MINIMUM(++x, ++y)`

ersetzt durch: `((++x)<(++y) ? (++x) : (++y))`

Eine der beiden Variablen wird dabei zweimal inkrementiert und liefert letztendlich ein missverständliches Ergebnis.

## Weitere Bestandteile

Auf diese Möglichkeiten gehen wir nicht weiter ein.

- Ersatzdarstellungen für Zeichen (ein Relikt für Zeichen, die auf der Tastatur fehlten)
- bedingte Übersetzungen: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`
- Zeilenkontrolle: `#line`
- Fehlermeldungen: `#error`
- implementierungsabhängige Aktion: `#pragma`
- leere Direktive: `#`

Folgende Namen für PP-Makros sind vordefiniert.

<code>__LINE__</code>	Nummer der aktuellen Quellzeile, <code>int</code> -Literal
<code>__FILE__</code>	Name der Datei, String-Literal
<code>__DATE__</code>	Datum der Übersetzung, String-Literal
<code>__TIME__</code>	Uhrzeit der Übersetzung, String-Literal
<code>__STDC__</code>	standardkonforme Implementierung, 0 oder 1 (Relikt)
<code>__STDC_VERSION__</code>	Sprachversion, <code>long</code> -Literal, verfügbar seit dem Standard C95

Weitere Makros sind mit den Standards C99 und C11 hinzugekommen.

## E. Unbehandelte Themen

---

Um den Umfang überschaubar zu halten, sind Themen im reduzierten Umfang dargestellt worden. Diese Themen hingegen wurden nicht behandelt.

- Typ-Attribut `volatile`
- Erweiterte Zeichensätze (*wide characters*)
- Sprunganweisung `goto` und Marken außerhalb von `switch`
- Argumente aus der Kommandozeile (nicht im Skript, jedoch als Beispiel zu Kapitel 13)
- Variable Argumentlisten
- Zeiger auf Funktionen
- Schnittstelle zum Betriebssystem
- Erstellung eigener Bibliotheken
- Verwendung von `make`-Dateien

Ferner wurden aus dem Standard C99 (und entsprechend auch aus C11) nur die wesentlichen Neuerungen vorgestellt; der Anhang F verdeutlicht den Umfang.

Die Standard-Bibliothek wurde C90-basiert und auch nur in Teilen behandelt.





## F. Was gibt's Neues? (Sprachnormen)

---

Die folgenden Aufstellungen wurden Wikipedia entnommen:

[http://de.wikipedia.org/wiki/Varianten\\_der\\_Programmiersprache\\_C](http://de.wikipedia.org/wiki/Varianten_der_Programmiersprache_C)

### Wichtigste Neuerungen von C95

- Verbesserung der Unterstützung von Multibyte- und *wide character*-Zeichensätzen durch die Standardbibliothek.
- Hinzufügen von Diagraphen zur Sprache.
- Definition von Standard-Makros zur alternativen Schreibweise von Operatoren, zum Beispiel `and` für `&&`.
- ✓ Definition des Standard-Makros `__STDC_VERSION__`.

### Wichtigste Neuerungen von C99

- Unterstützung von komplexen Zahlen durch den neuen Datentyp `_Complex` und entsprechende Funktionen in der Standardbibliothek.
- Erweiterung der ganzzahligen Datentypen um einen mindestens 64 Bit breiten Typ `long long`, sowie um Typen mit vorgegebener Mindestbreite, zum Beispiel `int_least8_t` und `uint_least32_t`. Außerdem werden Integer mit exakter Breite spezifiziert, aber als optional bezeichnet – zum Beispiel `int32_t`.
- ✓ Lokale Felder (= Arrays) variabler Größe.
- ✓ Der boolesche Datentyp `_Bool`. Über einen eigenen Header `<stdbool.h>` wird für ihn ein Makro namens `bool` definiert.
- Weiter verbesserte Unterstützung für internationale Zeichensätze.
- Erweiterte Unterstützung von Gleitkommazahlen inklusive neuer mathematischer Funktionen in der C-Bibliothek.
- Alias-freie Zeiger (Schlüsselwort `restrict`).
- ✓ Frei platzierbare Deklaration von Bezeichnern (in C90 durften diese nur am Anfang eines Blocks stehen).
- Inline-Funktionen (Schlüsselwort `inline`).
- ✓ Verbot des „impliziten `int`“; Verbot impliziter Funktionsdeklarationen.
- Hexadezimale Gleitkommakonstanten. Ein- und Ausgabe in `scanf()` und `printf()` über `%a` und `%A`.
- Präprozessor-Makros mit variabler Parameteranzahl.
- ✓ Zulassen des aus C++ bekannten Zeilenkommentars `//`.

(Die angehakten Themen ✓ wurden behandelt.)

### Wichtigste Neuerungen von C11

- Unterstützung von Multithreading (`<threads.h>`, `<stdatomic.h>`)
- Angaben zur Speicherausrichtung von Objekten (`<stdalign.h>`)
- Neue Datentypen `char16_t` und `char32_t` zur verbesserten Unterstützung von Unicode, insbesondere UTF-16 und UTF-32 (`<uchar.h>`)
- Änderungen an der Standardbibliothek zur Prüfung von Feldgrenzen zur Laufzeit des Programms, um z.B. Pufferüberläufe wirksamer vermeiden zu können
- Unterstützung der internen dezimalen Darstellung von Gleitkommazahlen gemäß IEEE 754-2008
- Öffnen von Dateien mit exklusivem Lese-/Schreibrecht (Modus "x")
- Generische Ausdrücke (Schlüsselwort `_Generic`), generische mathematische Funktionen für Gleitkommazahlen und komplexe Zahlen (`<tgmath.h>`)
- ✓ Entfernung der Bibliotheksfunktion `gets`
- Einige in C99 geforderte Funktionalität ist Compiler-Herstellern bei C11 wieder freigestellt, wie z.B. lokale Felder (= Arrays) variabler Größe, komplexe Zahlen. Der Funktionsumfang kann mit Hilfe von Compiler-Defines (= Präprozessor-Makros) abgefragt werden: `__STDC_NO_COMPLEX__` (keine komplexe Zahlen), `__STDC_NO_VLA__` (keine lokalen Felder variabler Länge)

### Anmerkungen zum Standard C11

- Viele der mit dem Standard C11 eingeführten Neuerungen sind optional in dem Sinne, dass sie vorhanden sein können, aber nicht unbedingt müssen. Präprozessor-Makros dienen dabei zur Signalisierung des Ist-Zustandes der jeweiligen Implementierung.
- Wenn man es so sehen will, ist die Freistellung einiger C99-Funktionalitäten ein Rückschritt.

(Die angehakten Themen ✓ wurden behandelt.)

# G. ASCII-Codetabelle

Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII	Dez	Hex	Okt	ASCII
0	0x0	00	<i>NUL</i>	32	0x20	040	<i>SP</i>	64	0x40	0100	@	96	0x60	0140	`
1	0x1	01	<i>SOH</i>	33	0x21	041	!	65	0x41	0101	A	97	0x61	0141	a
2	0x2	02	<i>STX</i>	34	0x22	042	"	66	0x42	0102	B	98	0x62	0142	b
3	0x3	03	<i>ETX</i>	35	0x23	043	#	67	0x43	0103	C	99	0x63	0143	c
4	0x4	04	<i>EOT</i>	36	0x24	044	\$	68	0x44	0104	D	100	0x64	0144	d
5	0x5	05	<i>ENQ</i>	37	0x25	045	%	69	0x45	0105	E	101	0x65	0145	e
6	0x6	06	<i>ACK</i>	38	0x26	046	&	70	0x46	0106	F	102	0x66	0146	f
7	0x7	07	<i>BEL</i>	39	0x27	047	'	71	0x47	0107	G	103	0x67	0147	g
8	0x8	010	<i>BS</i>	40	0x28	050	(	72	0x48	0110	H	104	0x68	0150	h
9	0x9	011	<i>TAB</i>	41	0x29	051	)	73	0x49	0111	I	105	0x69	0151	i
10	0xA	012	<i>LF</i>	42	0x2A	052	*	74	0x4A	0112	J	106	0x6A	0152	j
11	0xB	013	<i>VT</i>	43	0x2B	053	+	75	0x4B	0113	K	107	0x6B	0153	k
12	0xC	014	<i>FF</i>	44	0x2C	054	,	76	0x4C	0114	L	108	0x6C	0154	l
13	0xD	015	<i>CR</i>	45	0x2D	055	-	77	0x4D	0115	M	109	0x6D	0155	m
14	0xE	016	<i>SO</i>	46	0x2E	056	.	78	0x4E	0116	N	110	0x6E	0156	n
15	0xF	017	<i>SI</i>	47	0x2F	057	/	79	0x4F	0117	O	111	0x6F	0157	o
16	0x10	020	<i>DLE</i>	48	0x30	060	0	80	0x50	0120	P	112	0x70	0160	p
17	0x11	021	<i>DC1</i>	49	0x31	061	1	81	0x51	0121	Q	113	0x71	0161	q
18	0x12	022	<i>DC2</i>	50	0x32	062	2	82	0x52	0122	R	114	0x72	0162	r
19	0x13	023	<i>DC3</i>	51	0x33	063	3	83	0x53	0123	S	115	0x73	0163	s
20	0x14	024	<i>DC4</i>	52	0x34	064	4	84	0x54	0124	T	116	0x74	0164	t
21	0x15	025	<i>NAK</i>	53	0x35	065	5	85	0x55	0125	U	117	0x75	0165	u
22	0x16	026	<i>SYN</i>	54	0x36	066	6	86	0x56	0126	V	118	0x76	0166	v
23	0x17	027	<i>ETB</i>	55	0x37	067	7	87	0x57	0127	W	119	0x77	0167	w
24	0x18	030	<i>CAN</i>	56	0x38	070	8	88	0x58	0130	X	120	0x78	0170	x
25	0x19	031	<i>EM</i>	57	0x39	071	9	89	0x59	0131	Y	121	0x79	0171	y
26	0x1A	032	<i>SUB</i>	58	0x3A	072	:	90	0x5A	0132	Z	122	0x7A	0172	z
27	0x1B	033	<i>ESC</i>	59	0x3B	073	;	91	0x5B	0133	[	123	0x7B	0173	{
28	0x1C	034	<i>FS</i>	60	0x3C	074	<	92	0x5C	0134	\	124	0x7C	0174	
29	0x1D	035	<i>GS</i>	61	0x3D	075	=	93	0x5D	0135	]	125	0x7D	0175	}
30	0x1E	036	<i>RS</i>	62	0x3E	076	>	94	0x5E	0136	^	126	0x7E	0176	~
31	0x1F	037	<i>US</i>	63	0x3F	077	?	95	0x5F	0137	_	127	0x7F	0177	<i>DEL</i>

Weitergehende Information:

[http://de.wikipedia.org/wiki/American\\_Standard\\_Code\\_for\\_Information\\_Interchange](http://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange)



# Index

---

## A

abweisende Schleife	33
Adressarithmetik	58, 68
Adresse	55
Adressoperator	55
Aggregat	63
alignof	5
Anweisung	29
Argument	23, 45, 59
Argument (Array)	64, 74
Argument (Struktur)	84
Array	61, 67, 86
Array aus Zeichen	77
Aufzählung	27
Ausdruck	17
Ausdrucksanweisung	18, 29
Ausgabe	25, 79, 107
Auswahanweisung	30
auto	49

## B

Basistyp	56, 61
bedingter Ausdruck	21
Bedingung	12, 21, 30, 33
Bezeichner	5
Binden	3, 49
Bitfeld	103
Bit-Operator	100
Block	29, 47
break	32, 35

## C

C11	1, 126
C90	1
C99	1, 125
<i>call by reference</i>	60, 64, 74
<i>call by value</i>	46
calloc (Funktion)	90
case	32
cast-Notation	16
char	11
Codetabelle (ASCII)	127
Compiler	3
const	14, 46, 58, 64
continue	35
<ctype.h>	116

## D

Dateioperation	107
Datenstruktur	61
Datentyp	9
default	32
#define	120
Definition	47
Deklaration	47
Dekrement	19
Dereferenzierung	57
dezimale Darstellung	10
do	34
double	13
Dreiecksmatrix	91
dynamische Datenstruktur	61, 89, 96
dynamischer Speicher	89
dynamisches Array	90

## E

Ein- und Ausgabe	25, 79, 107
Eingabesymbole	5
Element	61
else	30
Entwicklungssystem	7
enum	27
Ersetzungsregel	70
Etikett	27, 81
extern	49

**F**

Fallauswahl	32
Fallunterscheidung	30
fgetc (Funktion)	111
fgets (Funktion)	112
float	13
<float.h>	13, 117
for	34
Formatelement	25, 79, 99, 109
formatierte E/A	108
Formatierung	6
fprintf (Funktion)	108
fputc (Funktion)	111
fputs (Funktion)	112
free (Funktion)	91
fscanf (Funktion)	108
Funktion	3, 37
Funktionsaufruf	23, 39, 46

**G**

ganzzahliger Typ	9
getchar (Funktion)	111
Gleitpunkt-Typ	13
globale Vereinbarung	47
goto	36
Gültigkeit	47

**H**

Header-Datei	3, 42, 52, 119
Heap	89
hexadezimale Darstellung	10

**I**

if	30
#include	42, 119
Indizierung	61
Indizierung (Zeiger)	70
Initialisierung	14, 50, 63, 82
Inkrement	19
inline	5
int	9
Integer Promotion	15

**K**

Kommentar	6
Komponente	61, 81
Konsolfenster	7
Konstante (Literal)	6, 10, 12, 13
Konstante (symbolisch)	14
konstanter Ausdruck	23
konstanter Parameter	46
konstanter Zeiger	58
konstantes Array	64
Kontrollstruktur	29
Konvertierung	15

**L**

Lebensdauer	49
Leerraum	5
<limits.h>	10, 117
Linken	3
Liste	92
Literal	6, 10, 12, 13
logische Operation	21
lokale Vereinbarung	47
lokales Objekt	14
long	9, 13
L-Wert	55

**M**

main-Funktion	43
Makro	120
malloc (Funktion)	90
Marke	32, 36
<math.h>	24
mathematische Funktion	24

**N**

Name	5
nichtabweisende Schleife	34
Norm	1, 125
Nullzeichen	77
NULL-Zeiger	56

**O**

oktale Darstellung	10
Operation	17

**P**

Parameter	45, 59
Parameter (Array)	64, 74
Parameter (Struktur)	84
Pfeil-Operator	85
Präprozessor	3, 42, 119
printf (Funktion)	25, 108
Programmdatei	3, 40
Programmstruktur	40
Prototyp	38
Pseudozufall	113
Punkt-Operator	83
putchar (Funktion)	111
puts (Funktion)	112

**Q**

Quelldatei	3
------------	---

**R**

rand (Funktion)	113, 118
realloc (Funktion)	90
register	49
Rekursion	54
reserviertes Wort	5
restrict	5
return	39
R-Wert	55

**S**

scanf (Funktion)	25, 108
Schleife	33
Separator	6
Sequenz	22
short	9
Sicherheit	105
signed	9
size_t	79, 89
sizeof	89
Sonderzeichen	5
Speicherklasse	49
Speicherreservierung	89
Sprunganweisung	35
srand (Funktion)	113, 118
Stack	89
Standard	1, 125
Standard-Bibliothek	3, 115
Standard-Datentyp	9
static	49
statische Datenstruktur	61
<stdbool.h>	12
<stdio.h>	25, 107
<stdlib.h>	90, 113, 118
String	77
<string.h>	78
struct	81
Struktur	81
strukturierte Programmierung	1
switch	32
syntaktische Einheit	5

**T**

Typ	9
Typdefinition	27, 81, 97
typedef	97

**U**

Übersetzen	3
Übersetzungseinheit	3
union	88
unsigned	9

**V**

Variable	14
Vereinbarung	47
Vergleich	20
void	38, 56
volatile	123
vorläufige Definition	50

**W**

wahlfreier Zugriff	61
Wahrheitswert	12, 21
Warteschlange	92
Wertzuweisung	19, 83
while	33
Wiederholung	33

**Z**

Zählschleife	34
Zeichen	11
Zeichen-Array	77
Zeichenkette	77
zeichenorientierte E/A	111
Zeichensatz	5
Zeiger	55, 67, 85
Zeiger auf Konstanten	58
zeilenorientierte E/A	112
Zufallszahl	113
Zuweisung	19, 83