6 Lexikalische Konventionen

„Lexikalisch“ bedeutet „ein Wort (eine Zeichengruppe) betreffend“, ohne den Textzusammenhang (Kontext), in dem dieses Wort steht, zu berücksichtigen. Eine lexikalische Einheit ist eine zusammengehörige Zeichengruppe beziehungsweise ein Wort eines Programmtextes.

Der **Quellcode** eines C-Programms besteht aus lexikalischen Einheiten und Trennern wie zum Beispiel Leerzeichen. Jede lexikalische Einheit darf nur Zeichen aus dem Zeichenvorrat (Zeichensatz) der Sprache umfassen.

Im Folgenden werden die lexikalischen Konventionen besprochen, also gewissermaßen die Rechtschreibregeln von C.

6.1 Zeichenvorrat von C

Um ein Programm in der Programmiersprache C zu verfassen, müssen die Anweisungen an den Computer in Quelldateien geschrieben werden. Diese Dateien müssen einen sogenannten „**Basiszeichensatz**“ unterstützen, sprich, ein minimales Set an Zeichen, welches von einem Compiler verarbeitet werden kann.

Der Basiszeichensatz für Quelldateien in C enthält die folgenden Zeichen: Die 26 Groß- und Kleinbuchstaben des lateinischen Alphabets, die 10 arabischen Dezimalziffern, sowie 29 Interpunktionszeichen:

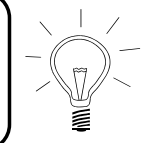
```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~
```

Des Weiteren definiert der Basiszeichensatz das Leerzeichen, den Tabulator sowie ein paar wenige Steuerzeichen für Zeilenumbruch und Ende der Quelldatei. Zudem wird das sogenannte „**Nullzeichen**“ `'\0'` definiert, bei dem alle Bits null sind. Dieses wird beispielsweise verwendet, um ein Stringende zu markieren.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_6.

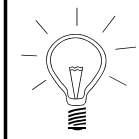
J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_6

> Die Zeichen des Basiszeichensatzes werden durch ein einziges Byte dargestellt und reichen für die Programmierung in C vollkommen aus.



Während der Ausführung des Programmes werden jedoch noch mehr Zeichen benötigt. Insbesondere spezifiziert C zusätzliche Steuerzeichen wie Alarm, Backspace oder Zeilenvorschub. Diese haben in einer Quelldatei keine Bedeutung, jedoch während der Ausführung. Es wird somit zwischen „**Quellzeichensatz**“ und „**Ausführungszeichensatz**“ unterschieden.

> C definiert sowohl einen Quell- als auch einen Ausführungszeichensatz. Beide definieren einen jeweiligen Basiszeichensatz, sprich ein minimales Set an Zeichen, welche entweder während der Compilierung oder während der Laufzeit verfügbar sein müssen.

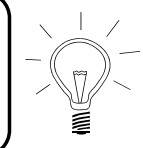


Ursprünglich stellte der Zeichensatz **ISO 646** den Basiszeichensatz von C dar. Durch die fortlaufende Standardisierung hat sich daraus der **ASCII-Zeichensatz** gebildet, welcher heute als Basis für die meisten Programmiersprachen gilt. Mehr dazu kann im Anhang → D nachgelesen werden.

Im Lauf der Jahre wurden die Ansprüche an Zeichensätze immer größer. Während für das Schreiben von Quellcode bis heute der Basiszeichensatz ausreichend ist, genügte selbiger während der Ausführung schon bald nicht den Anforderungen. Beispielsweise werden für Ausgaben auf dem Bildschirm zusätzliche Zeichen wie äöü benötigt oder gar chinesische Schriftzeichen.

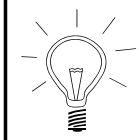
Ein Zeichensatz, welcher solche erweiterten Zeichen enthält, wird als „erweiterter Zeichensatz“ bezeichnet.

> Ein erweiterter Zeichensatz umfasst nebst dem Basiszeichensatz noch zusätzliche implementations- oder länderspezifische Zeichen, beispielsweise äöü.



Erst durch die Einführung dieser erweiterten Zeichensätze wurde es überhaupt möglich, Programme für den internationalen Markt zu erstellen. Um zu ermöglichen, dass solche internationale Zeichen überhaupt in den Programmcode geschrieben werden können, erlaubt C somit auch erweiterte Zeichensätze für den Quellcode.

> Sowohl Quell- als auch Ausführungszeichensatz enthalten jeweils den zugehörigen Basiszeichensatz, können aber auch stets ein erweiterter Zeichensatz sein. Quell- und Ausführungszeichensatz müssen dabei nicht gleich sein.



Heutzutage hat sich sowohl für Quell- als auch für den Ausführungszeichensatz **Unicode** und insbesondere die **UTF-8-Codierung** durchgesetzt. Unicode beinhaltet den für C erforderlichen Basiszeichensatz für Quelldateien und Laufzeit, erweitert ihn jedoch um rund 4 Millionen Zeichen.

Wie diese Vielzahl an Zeichen während des Programmierens angesprochen werden kann, wird in Kapitel → 6.5.6 erläutert.

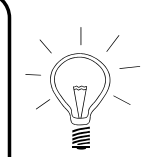
6.2 Lexikalische Einheiten

In der Sprache C haben viele Zeichen des Quellzeichensatzes je nach Situation eine andere Bedeutung. Insbesondere Interpunktionszeichen wie Komma, Minuszeichen oder Klammern kommen an den unterschiedlichsten Stellen vor. Die Aufgabe des Compilers ist es, diese Zeichen gemäß genau festgelegten syntaktischen Regeln zu lesen und ihnen eine Bedeutung (Semantik) zuzuordnen. Siehe auch Kapitel → 5.1.

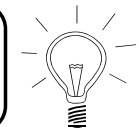
Ein Programm besteht für einen Compiler zunächst nur aus einer Folge von Bytes. Die erste Phase des Compilierlaufs besteht aus einer lexikalischen Analyse (siehe Kapitel → 5.1.1), die der sogenannte Scanner durchführt.

Der Scanner hat unter anderem die Aufgabe, Zeichengruppen, welche eine lexikalische Einheit bilden, zu finden. Eine solche Einheit wird auch „token“ genannt.

Eine lexikalische Einheit wird gefunden, indem man die Trenner, die sie begrenzen, findet. Trenner sind Zwischenraum (Whitespace-Zeichen), Operatoren und Interpunktionszeichen.



Zu den **Whitespace-Zeichen** gehören Leerzeichen, Tabulator, Zeilentrenner sowie Kommentare.



Kommentare zählen auch zu den Whitespace-Zeichen. Das ist zunächst etwas verwirrend, denn in einem Kommentar steht ja tatsächlich etwas. Nach dem Präprozessorlauf sind die Kommentare jedoch entfernt und an ihre Stellen wurden Leerzeichen eingesetzt. Wie Kommentare geschrieben werden, kann in Kapitel → 2.2.1 nachgelesen werden.

Zwischen zwei aufeinanderfolgenden lexikalischen Einheiten kann eine beliebige Anzahl an Whitespace-Zeichen eingefügt werden. Damit hat man die Möglichkeit, ein Programm optisch so zu gestalten, dass die Lesbarkeit verbessert wird.

Auch Operatoren und Interpunktionszeichen sind Trenner. Für den Compiler ist `A&&B` lesbar (&& ist der logische UND-Operator zwischen `A` und `B`). Um die Lesbarkeit von Code zu steigern, empfiehlt es sich, nicht alleine auf die Trenner-Eigenschaft der Operatoren zu bauen, sondern nach jeder lexikalischen Einheit Leerzeichen einzugeben. Im genannten Beispiel also besser `A && B` schreiben!

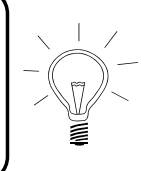
Die lexikalischen Einheiten werden sodann vom Parser gemäß der Syntax der Sprache geprüft (siehe Kapitel → 5.1.2) und grob in folgende Kategorien unterteilt:

- Reservierte Wörter (Schlüsselwörter)
- Bezeichner
- Literale Konstanten
- Operatoren und Interpunktionszeichen

Auf diese Kategorien wird in den folgenden Kapiteln im Einzelnen eingegangen.

6.3 Reservierte Wörter (Schlüsselwörter)

Die Namen von **Schlüsselwörtern** (auf Englisch „**keywords**“) sind reserviert. Die Bedeutung dieser Schlüsselwörter ist von der Programmiersprache festgelegt und kann nicht verändert werden.



Die Namen von Schlüsselwörtern dürfen nicht als Bezeichner von Objekten des Programms, also beispielsweise von Variablen und Funktionen oder selbst definierten Datentypen, verwendet werden.

Eine vollständige Erklärung dieser Schlüsselwörter kann erst in den späteren Kapiteln erfolgen. Im Folgenden werden alle Schlüsselwörter von C kurz aufgelistet:

`asm`	Einfügen von Assemblercode
`auto`	Speicherklassen-Bezeichner
`break`	Zum Herausspringen aus Schleifen oder der `switch`-Anweisung
`case`	Auswahl-Fall in `switch`-Anweisung
`char`	Typ-Bezeichner
`const`	Qualifikator für Typangabe
`continue`	Fortsetzungsanweisung
`default`	Standardeinsprungmarke in `switch`-Anweisung
`do`	Teil einer Schleifen-Anweisung
`double`	Typ-Bezeichner
`else`	Einleitung einer Alternative
`enum`	Aufzählungs-Typ-Bezeichner
`extern`	Speicherklassen-Bezeichner
`float`	Typ-Bezeichner
`for`	Schleifenanweisung
`goto`	Sprunganweisung
`if`	Beginn einer bedingten Anweisung
`inline`	Funktions-Spezifikator zur Spezifikation von Inline-Funktionen
`int`	Typ-Bezeichner
`long`	Typ-Modifikator beziehungsweise Typ-Bezeichner
`register`	Speicherklassen-Bezeichner

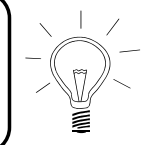
`restrict`	Qualifikator für Typangabe zur Einschränkung eines Zugangs zu einem Objekt über einen bestimmten Pointer
`return`	Rücksprung-Anweisung
`short`	Typ-Modifikator beziehungsweise Typ-Bezeichner
`signed`	Typ-Modifikator beziehungsweise Typ-Bezeichner
`sizeof`	Operator zur Bestimmung der Größe von Variablen, Typen und Konstanten
`static`	Speicherklassen-Bezeichner
`struct`	Strukturvereinbarung
`switch`	Auswahlanweisung
`typedef`	Typnamenvereinbarung
`union`	Datenstruktur mit Alternativen
`unsigned`	Typ-Modifikator beziehungsweise Typ-Bezeichner
`void`	Typ-Bezeichner
`volatile`	Qualifikator für Typangabe
`wchar_t`	Typ-Bezeichner
`while`	Schleifenanweisung
`_Alignas`	Alignment-Spezifikator
`_Alignof`	Operator zur Bestimmung des Alignments von Typen
`_Atomic`	Zur Deklaration von atomaren Variablen
`_Bool`	Typ-Bezeichner
`_Complex`	Verwendet zur Spezifikation komplexer Datentypen
`_Generic`	Zur Simulation von Overloading
`_Imaginary`	Imaginäre Einheit
`_Noreturn`	Funktions-Spezifikator für eine Funktion, die nicht zum Aufrufer zurückkehrt
`_Static_assert`	Für statische Prüf-Ausdrücke
`_Thread_local`	Speicherklassen-Bezeichner für eine Variable pro Thread

Einige dieser Schlüsselwörter wurden erst in neueren Standards eingeführt. Des Weiteren fehlen in dieser Liste die Präprozessor-Direktiven sowie implementationsabhängige Schlüsselwörter, welche je nach Compiler zusätzlich definiert sein können.

6.4 Bezeichner

Ein **Bezeichner** (auf Englisch „**identifier**“) ist nichts anderes als ein Name, welcher im Quellcode einer bestimmten Einheit gegeben wird. Häufig wird auch der Begriff „**Symbol**“ verwendet.

> Ein Bezeichner, besteht aus einer Zeichenfolge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt. In C zählt auch der Unterstrich _ zu den Buchstaben.



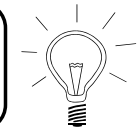
Bezeichner werden in C an verschiedenen Stellen verwendet:

- Variablennamen
- Funktionsnamen
- Etiketten (auf Englisch „tags“) von Strukturen, Unionen, Bitfeldern und Aufzählungs-Typen
- Komponenten von Strukturen
- Alternativen von Unionen
- Aufzählungs-Konstanten
- Typnamen (`typedef`-Namen)
- Marken
- Makronamen (symbolische Konstanten)
- Makroparameter

Hierbei ist zu beachten, dass ein Parser streng zwischen Groß- und Kleinbuchstaben unterscheidet. Der englische Ausdruck hierfür ist „**case sensitive**“. Bezeichner, die sich durch Groß- und Kleinschreibung unterscheiden, stellen verschiedene Namen dar. So ist beispielsweise der Name `alpha` ein anderer Name als `Alpha`.

Bezeichner können heutzutage beliebig lange sein. Zudem sind in modernen Compilern ab dem C99-Standard auch die Zeichen des erweiterten Zeichensatzes (beispielsweise ä, ö, ü, ß) erlaubt. In manchen Programmen finden sich gar Emojis als Bezeichner.

Ein Bezeichner muss zwingend mit einem Klein- oder Großbuchstaben beginnen!



Namen, die mit einem Unterstrich _ oder zwei Unterstrichen beginnen, sind zwar erlaubt, sollten gemäß Standard jedoch nicht verwendet werden, da sie für systemspezifische Bibliotheksfunktionen reserviert sind, was zu unerwarteten Konflikten führen kann.

Einige Beispiele für zulässige und unzulässige Namen:

`querSumme`	Zulässig
`quer_summe`	Zulässig, Unterstrich ist ein Buchstabe
`quer-Summe`	Unzulässig, Bindestrich ist kein Buchstabe
`8Summe`	Unzulässig, beginnt mit einer Ziffer
`_summe`	Zulässig aber nicht empfohlen, reserviert für Sprache und Compiler.
`quärSumme`	Zulässig, aber erweitertes Zeichen. Zudem falsch geschrieben.
`Quer☺`	Zulässig, aber erweitertes Zeichen.

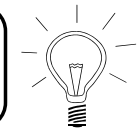
6.5 Literale Konstanten

Es gibt verschiedene Arten von **literalen Konstanten**, wobei jede literale Konstante einen definierten Datentyp besitzt:

- Integer-Konstanten
- Gleitpunkt-Konstanten
- Zeichen-Konstanten
- Konstante Strings

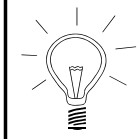
6.5.1 Integer-Konstanten

Integer-Konstanten sind Zahlen wie `1234`. Sie sind vom Typ `int`.



Integer-Konstanten lassen sich in verschiedenen Zahlensystemen darstellen.

Außer der gewöhnlichen Abbildung im Dezimalsystem (Basis 10) gibt es in C noch die Darstellungsform im Hexadezimalsystem (Basis 16) und im Oktalsystem (Basis 8). Ab dem Standard C23 ist zudem die Darstellung im Binärsystem (Basis 2) erlaubt.



Die genannten Zahlensysteme und die Vorgehensweise bei der Umrechnung von einer Darstellung in eine andere werden im Anhang → C ausführlich erklärt. Hier wird nur die Schreibweise und Interpretation behandelt.

Eine **Dezimalzahl** besteht aus den Dezimalziffern `0`, `1`, ... `9`, also beispielsweise `72656`. Die erste Ziffer darf dabei nicht `0` sein. Wenn die erste Ziffer `0` ist, wird die Konstante als eine Oktalzahl interpretiert, welche nur die Ziffern `0` bis `7` beinhalten darf. **Oktalzahlen** werden heutzutage nur noch selten benutzt.

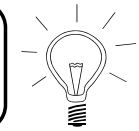
In der Programmierung besonders wichtig sind **Hexadezimalzahlen**. Für die zusätzlichen Ziffernwerte `10` bis `15` verwendet man in C die Buchstaben `a` bis `f` oder `A` bis `F`. Dabei ist der Wert von `a` oder `A` gleich `10`, der Wert von `b` oder `B` gleich `11`, ... Um eine Zahl als Hexadezimalzahl zu kennzeichnen, muss sie mittels `0x` oder `0X` eingeleitet werden, beispielsweise `0x6aff33b1`.

Binärzahlen sind erst ab dem Standard C23 erlaubt. Sie verwenden nur die beiden Ziffern `0` und `1` und müssen ähnlich wie Hexadezimalzahlen mittels `0b` oder `0B` eingeleitet werden. Beispielsweise `0b01100010`.

Integer-Konstanten sind in C streng genommen stets positiv. Benötigt man einen negativen Wert, so schreibt man in C einfach den einstelligen Minus-Operator davor, wie beispielsweise `-85`.

Der Datentyp einer Integer-Konstante ist grundsätzlich `int`. Dieser Typ ist implementationsabhängig und kann somit je nach Umgebung unterschiedlich groß werden. Entsprechend müssen manchmal Konstanten mittels einem speziellen **Suffix** markiert werden, wenn sie sehr große Zahlen darstellen.

> Sehr große Integer müssen manchmal mit dem Suffix `ll` oder `LL` markiert werden. Beispielsweise `100000000000ll`



Das Suffix `ll` steht für „`long long`“. Dies bezeichnet eine Typerweiterung des `int`-Typs, welche heutzutage nicht mehr so verwendet werden sollte. Im Anhang → E wird genauer darauf eingegangen.

Ein weiteres Suffix, welches manchmal für Integer-Konstanten verwendet wird ist `u` oder `U`, was für `unsigned` steht. Damit wird eine Konstante explizit als vorzeichenlos markiert.

Ab dem Standard C23 können Zahlen auch mit Trennzeichen versehen werden. Mittels des Apostrophs `'` können so große Zahlen lesbarer gemacht werden. Es gibt keine Regeln, wie groß die Zifferngruppen sein sollten, die mittels der Trennzeichen entstehen – der Compiler überliest sie einfach. Für Dezimalzahlen empfiehlt es sich somit beispielsweise, jeweils nach drei Ziffern zu trennen, für Hexadezimalzahlen ist eine Trennung nach zwei, vier oder acht Ziffern möglicherweise sinnvoller.

Die verschiedenen `int`-Datentypen werden in Kapitel → 7.2 genauer behandelt.

Beispiele für Integer-Konstanten sind:

`14`	`int`-Konstante in Dezimaldarstellung mit dem Wert `14` dezimal
`014`	`int`-Konstante in Oktaldarstellung mit dem Wert `12` dezimal
`0x14`	`int`-Konstante in Hexadezimaldarstellung mit dem Wert `20` dezimal
`14ll`	`long long`-Konstante in Dezimaldarstellung mit dem Wert `14`
`14u`	`unsigned`-Konstante in Dezimaldarstellung mit dem Wert `14`
`0x14ull`	`unsigned long long`-Konstante in Hexadezimaldarstellung
`14'633'165`	Große Zahl mit Tausendertrennzeichen (erst ab C23)

6.5.2 Gleitpunkt-Konstanten

Gleitpunkt-Konstanten (Fließkomma-Konstanten) haben einen `.` (**Dezimalpunkt**) oder ein `E` beziehungsweise `e` oder beides. Entweder der ganzzahlige Anteil vor dem Punkt oder der Dezimalbruch-Anteil nach dem Punkt darf fehlen, aber nicht beide zugleich.

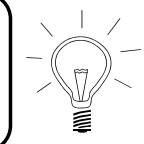
Beispiele für Gleitpunkt-Konstanten sind:

```
300.0      300.      .3      3e2      3.e2      .3e3      .3E3
```

Der Teil einer Gleitpunkt-Zahl vor dem `E` beziehungsweise `e` ist die sogenannte „**Signifikante**“ (früher „**Mantisse**“), der Teil dahinter der sogenannte „**Exponent**“. Wird ein Exponent angegeben, so ist die Signifikante mit 10^{Exponent} zu multiplizieren.

`3e2 = 3 * 10^2 = 300`

> Eine Gleitpunkt-Konstante hat den Typ `double`. Durch die Angabe eines optionalen Typ-Suffixes `f` oder `F` wird der Typ der Konstanten zu `float`.



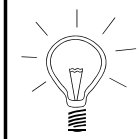
So ist `10.3` vom Typ `double`, `10.3f` vom Typ `float`.

Weitere Informationen über die Suffixe können im Anhang → E nachgelesen werden.

6.5.3 Zeichen-Konstanten

Eine **Zeichen-Konstante** ist ein Zeichen eingeschlossen in einfachen Hochkommas, beispielsweise `'A'`. Der Wert der Zeichen-Konstanten ist gegeben durch den numerischen Wert des Zeichens im aktuellen Ausführungszeichensatz.

> Obwohl eine Zeichen-Konstante vom Compiler im Arbeitsspeicher als `char`-Typ, das heißt als ein Byte, abgelegt wird, ist der Typ einer Zeichen-Konstanten, auf die in einem Programm zugegriffen wird, der Typ `int`.

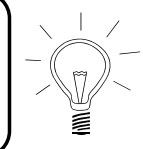


Mit Zeichen-Konstanten kann man rechnen wie mit Integer. So hat beispielsweise das Zeichen `'0'` im ASCII-Zeichensatz den Wert `48`. Zeichen-Konstanten werden aber meist gebraucht, um Zeichen zu vergleichen. Schreibt man die Zeichen als Zeichen-Konstanten und nicht als Integer, so ist man bei den Vergleichen unabhängig vom verwendeten Zeichensatz des Rechners.

Es gibt auch Zeichen-Konstanten mit mehreren Zeichen innerhalb der einfachen Hochkommas, beispielsweise `'A2B2'`. Der Standard spezifiziert, dass der Wert eines solchen Literals implementationsabhängig sei. Häufig werden ebensolche Konstanten mit zwei Zeichen als 16-Bit-Integer-Zahl und solche mit 4 Zeichen als 32-Bit-Integer-Zahl interpretiert. Dass dieses Verhalten jedoch plattformübergreifend funktioniert, ist nicht garantiert.

6.5.4 Konstante Strings

> **Konstante Strings** sind Folgen von Zeichen, die in Anführungszeichen eingeschlossen sind. Die Anführungszeichen sind nicht Teil der Strings, sondern begrenzen sie nur.



Beispiele für konstante Strings sind etwa `"Kernighan"` oder `"Ritchie"`. Ein konstanter String wird intern dargestellt als ein Array von Zeichen. Arrays werden in Kapitel → 8.3 eingeführt. Am Schluss des Arrays wird vom Compiler ein zusätzliches **Nullzeichen**, das Zeichen `'\0'`, angehängt, um das Stringende zu charakterisieren. Das folgende Bild gibt ein Beispiel:

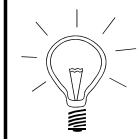
'R'	'i'	't'	'c'	'h'	'i'	'e'	'\0'

Stringverarbeitungsfunktionen benötigen das Zeichen `'\0'`, damit sie das Stringende erkennen. Deshalb muss bei der Speicherung von Strings stets ein Speicherplatz für das Nullzeichen vorgesehen werden. So stehen in dem String `"Hallo Welt"` zwischen den Anführungszeichen 10 Zeichen (inklusive Leerzeichen). Für die Speicherung dieses Strings werden 11 Zeichen benötigt (10 Zeichen + Nullzeichen).

Eine Zeichen-Konstante `'a'` und ein String `"a"` mit einem einzelnen Zeichen sind zwei ganz verschiedene Dinge. `"a"` ist ein Array aus den Zeichen `'a'` und einem Nullzeichen `'\0'`.

Befindet sich das Zeichen `'\0'` innerhalb eines Strings, so wird von einer Stringverarbeitungsfunktion an dieser Stelle das Stringende erkannt und der Rest des Strings wird nicht gelesen.

Stehen in einem Quellprogramm mehrere Strings hintereinander, wie beispielsweise `"Hallo " "Welt"`, so erzeugt der Präprozessor daraus durch Verkettung einen einzigen String. Dabei ist dann nur am Ende das Zeichen `'\0'` angehängt.

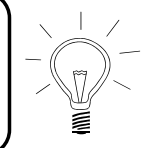


6.5.5 Ersatzdarstellungen

Gewisse Zeichen können nicht so einfach in Zeichen-Konstanten oder Strings eingegeben werden. Beispielsweise ist das Zeichen `'` für Zeichen-Konstanten nicht verwendbar, da es das Ende der Zeichen-Konstante markiert. Genauso kann das Zeichen `"` für Strings nicht benutzt werden. Auch Kontrollzeichen wie das Nullzeichen oder ein Zeilenende sind nicht so einfach in den Quellcode zu schreiben.

Hierfür existieren sogenannte **Ersatzdarstellungen**, auf Englisch auch „**escape sequences**“ genannt. Ersatzdarstellungen werden stets mit Hilfe eines **Backslash** \ (Gegenschrägstrich) konstruiert.

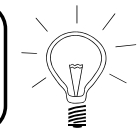
Mit Ersatzdarstellungen kann man Steuerzeichen oder Zeichen, die auf dem Eingabegerät nicht vorhanden oder nur umständlich zu erhalten sind, angeben.



Die Ersatzdarstellung für einen Zeilentrenner beispielsweise ist `\n`. Das n ist von Newline abgeleitet. `\n` ist ein Steuerzeichen, welches zur Laufzeit des Programmes den Text-Cursor an den Beginn der nächsten Zeile setzt.

Ersatzdarstellungen werden immer als mehrere Zeichen in den Quellcode geschrieben, werden aber vom Compiler in ein einziges Zeichen umgewandelt.

Das erste Zeichen ist immer ein Backslash. Die weiteren Zeichen legen die Bedeutung fest.



Die folgende Tabelle zeigt die Ersatzdarstellungen in C:

Ersatz	Bedeutung	Zeichen
`\0`	Nullzeichen	`0x0`
`\a`	Klingelzeichen	`0x7`
`\b`	Backspace	`0x8`
`\t`	Tabulatorzeichen	`0x9`
`\n`	Zeilenende-Zeichen	`0xa`
`\v`	Vertikal-Tabulator	`0xb`
`\f`	Seitenvorschub (Form Feed)	`0xc`
`\r`	Wagenrücklauf	`0xd`
`\\`	Gegenschrägstrich (Backslash)	`\`
`\?`	Fragezeichen	?
`\'`	Einfaches Hochkomma	'
`\”`	Anführungszeichen (doppeltes Hochkomma)	”
`\ooo`	oktaler Wert	
`\xhh`	hexadezimaler Wert	
`\uhhhh`	Unicode Zeichen mit 16 Bits	
`\Uhhhhhhhh`	Unicode Zeichen mit 32 Bits	

Die Oktal- und Hexadezimalddarstellungen werden heutzutage kaum mehr genutzt und werden hier nicht weiter behandelt. Sie wurden größtenteils verdrängt durch die Ersatzdarstellungen `\uhhhh` und `\Uhhhhhhhh`. Diese erlauben es, Zeichen explizit im Unicode-Format angeben zu können. Auf Unicode und diese Ersatzdarstellungen wird im nächsten Kapitel genauer eingegangen.

6.5.6 Der Unicode in C

In diesem Kapitel werden verschiedene Zeichencodierungen angesprochen. Da das Thema äußerst umfangreich ist, kann hier nur das wichtigste beschrieben werden. Es wird bereits jetzt auf Anhang → D verwiesen, wo eine Vertiefung zum **Unicode** und zu den Codierungen **UTF-8**, **UTF-16** und **UTF-32** gegeben ist.

Üblicherweise werden Zeichen-Konstanten mittels einfacher und Strings mittels doppelter Anführungszeichen geschrieben. Dies veranlasst den Compiler, die innerhalb der Anführungszeichen enthaltenen Zeichen in den Ausführungszeichensatz umzuwandeln, welcher durch den Compiler vorgegeben ist.

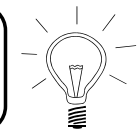
Je nach System definiert ein Compiler sogar zwei Ausführungszeichensätze: Einen, bei welchem die Zeichen den Typ **char** haben und einen, bei welchem die Zeichen den Typ **`wchar_t`** haben. Der Typ `wchar_t` steht für „**wide character**“ und hat die Aufgabe, erweiterte Zeichen zu speichern. Um anzugeben, dass ein Zeichen oder ein String den `wchar_t`-Typ speichern soll, wurde das **Präfix** `L` verwendet:

`"hello"`	`char` Zeichen gemäß Compilereinstellungen
`L"hello"`	`wchar_t` Zeichen gemäß Compilereinstellungen

In den gängigen Systemen wurde für den `char`-Typ die systeminterne Codierung verwendet. Auf Macintosh-Systemen war dies lange Zeit „MacRoman“ und später UTF-8, auf Windows war dies „Windows-1252“, auch umgangssprachlich bekannt als „ANSI“. Für den `wchar_t`-Typ wird oftmals die UTF-16 oder UTF-32 Codierung des Unicodes verwendet, doch standardisiert ist dies nicht. Jeder Compiler kann dies selbst festlegen.

Zu früheren Zeiten war die Steuerung des Compilers somit die einzige Möglichkeit, einen bestimmten Ausführungszeichensatz zu erzwingen. Wie genau dem Compiler mitgeteilt wird, welchen Ausführungszeichensatz er verwenden soll, ist jedoch implementationsspezifisch und somit nicht portabel. Um dem Abhilfe zu schaffen, wurde mit C11 die Initialisierung von Strings mit Unicode standardisiert.

> Um einen String explizit mit Unicode zu codieren, verwendet man das Präfix `u8`, `u` oder `U`.

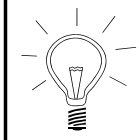


Bei Verwendung eines der Präfixe `u8`, `u` oder `U` kann beim Schreiben des Codes direkt angegeben werden, in welcher Unicode-Codierung der String schlussendlich zur Laufzeit verfügbar sein soll:

```
u8"hello"      UTF-8
u"hello"       UTF-16
U"hello"       UTF-32
```

Die Zeichen werden jeweils vom Compiler im Quellzeichensatz eingelesen. Alle normalen Zeichen werden in den gewünschten Laufzeitzeichensatz umkonvertiert. Falls Zeichen mittels Ersatzdarstellung enthalten sind, werden diese direkt in den gewünschten Laufzeitzeichensatz übertragen. So können mittels der Ersatzdarstellungen `\uhhhh` und `\Uhhhhhhhh` direkt Unicode-Zeichen mittels ihres eindeutigen Hexadezimalcodes eingegeben werden.

> Im Gegensatz zur Ersatzdarstellung für Oktal- und Hexadezimalwerte muss man sich bei der Ersatzdarstellung für Unicode-Zeichen nicht darum kümmern, in welcher Codierung der String intern tatsächlich gespeichert wird. Der Compiler erledigt dies automatisch.



Der konvertierte String wird vom Compiler schlussendlich direkt in das Maschinenprogramm hineinkodiert.

Für die Codierungen UTF-16 und UTF-32 existieren die beiden Typen `char16_t` und `char32_t`. Sie dürfen explizit nur für diese Codierungen verwendet werden. Für UTF-8-Zeichen gibt es vorerst keinen Datentyp `char8_t`, denn der Wert kann in einer Variablen vom Typ `char` gespeichert werden. Erst ab dem Standard C23 wird dieser Typ definiert sein. Er darf explizit nur für die UTF-8-Codierung verwendet werden.

Die Sprache C liefert jedoch nur ein Grundgerüst. Man muss sich bei Verwendung von Unicode darum kümmern, dass zum richtigen Zeitpunkt die richtige Codierung verfügbar ist. Um zwischen den Codierungen zu wechseln, werden Standardbibliotheksfunktionen bereitgestellt. Die neuen Typen sowie die Umwandlungsfunktionen werden in der Bibliothek <`uchar.h`> definiert.

6.6 Operatoren und Interpunktionszeichen 

Mit **Interpunktionszeichen** wie Punkt, Komma oder Klammern wird die Sprache gegliedert. Interpunktionszeichen treten sowohl als einzelne Sprachelemente auf als auch in Kombination mit anderen Zeichen.

Einige Interpunktionszeichen werden sowohl als eigenständige Sprachelemente als auch für Operatoren verwendet. Der Strichpunkt beispielsweise tritt als Ende einer Anweisung oder in `for`-Schleifen als Trenner von Ausdrücken auf. Das Komma dient als Trenner von Listenelementen beispielsweise in der Parameterliste von Funktionen, ist jedoch auch als eigener Operator zulässig. Der Doppelpunkt wird bei der Definition von Bitfeldern benötigt wie auch als Teil des Bedingungs-Operators. Das Symbol * wird sowohl für die Definition von Pointern als auch als Multiplikations-Operator benötigt.

6.6.1 Operatoren

Operatoren sind die ausführenden Elemente der Sprache. Die meisten Operatoren bestehen aus einem oder mehreren zusammengesetzten Interpunktionszeichen. In C werden die folgenden Symbole für Operatoren verwendet:

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

Einige wenige Operatoren von C verwenden anstelle von Interpunktionszeichen ein Schlüsselwort und werden deswegen hier nicht aufgeführt. Sie werden jedoch alle in Kapitel → 9 und → 21 behandelt.

6.6.2 Interpunktionszeichen

Folgende Tabelle ist eine unvollständige Auflistung von Zeichen, wenn sie nicht als Operatoren verwendet werden:

`" "`	Kennzeichnung von Strings
`' '`	Kennzeichnung von einzelnen Zeichen
`*`	Pointer, Funktionspointer (also Pointer auf Funktionen), geklammerte Kommentare `/* */`
`/`	Kommentare `//` und `/* */`
`%`	`printf()`- und `scanf()`-Formatierung (als Teil des Strings)
`()`	Klammerung, treten stets paarweise auf
`[]`	Arrays, treten stets paarweise auf
`{}`	Blöcke, treten stets paarweise auf
`;`	Begrenzer
`:`	Sprungmarken, Bitfelder
`,`	Variablendefinitionen, Argumente- und Parameterlisten, Initialisierungen, Enumerationen
`.`	Teil von Gleitpunkt-Werten (Literal), Teil von Ellipse `...`
`\`	Escape-Character
`#`	Präprozessor-Direktiven
`_`	Gilt als Buchstabe

Nicht in dieser Liste aufgeführt sind gewisse Ziffern und Buchstaben, welche manchmal als Präfix oder Suffix verwendet werden.

6.7 Übungsaufgaben

Aufgabe 1: Schreibweise für literale Konstanten

a) Welche der folgenden Konstanten sind syntaktisch richtig, was ist falsch?

```
#define ALPHA   -1e-0
#define BETA    -e12
#define GAMMA    .517
#define DELTA    3+
```

b) Welche dieser Konstanten sind vom Typ `double` und welche vom Typ `int`?

```
#define ZAHL1   55.5e5
#define ZAHL2   55.
#define ZAHL3   55e5f
#define ZAHL4   55
#define ZAHL5   55.5
```

c) Geben Sie obige Konstanten mit `printf()` aus. Wo wird `%d` benötigt, wo `%f`? Was passiert, wenn es verwechselt wird?

Aufgabe 2: Ausgabe von ASCII-Zeichen

Schreiben Sie ein Programm, das folgenden String ausgibt:

```
"Achtung,\ndas geheime Passwort lautet\t\x1b 1234\t\nDanke!"
```

Geben Sie den String Zeichen für Zeichen aus. Aber ersetzen Sie dabei:

- Jedes Zeilenende mit dem Text `"- STOP -"`
- Jeden Tabulator mit dem Text `"\n********\n"`
- Jedes Zeichen `\x1b` (Escape-Taste) mit dem Text `"- VERBINDUNG UNTERBROCHEN -"`, woraufhin die Abarbeitung stoppt.

Benutzen Sie die Funktion `strlen()` der `<string.h>`-Bibliothek, um die Länge des Strings zu ermitteln. Um ein einzelnes Zeichen von Typ `char` auszugeben, kann das Formatelement `%c` für `printf()` verwendet werden.