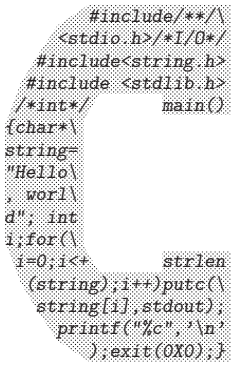
Programmierung in



Günter Egerer
Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich

1 Literatur

[1] ZAM-Benutzerhandbuch:
Einführung in die Programmiersprache C
FZJ-ZAM-BHB-0095

[2] Benutzerhandbücher für C-Compiler Ihres Systems

[3] Brian W. Kernighan, Dennis M. Ritchie
Programmieren in C - 2. Ausgabe ANSI C
Mit dem C-Reference Manual in deutscher Sprache
Hanser, 1990

[4] Clovis L. Tondo, Scott E. Gimpel
Das C-Lösungsbuch (zu [3]) - 2. Ausgabe
Hanser, 1990

[5] Samuel P. Harbison, Guy L. Steele Jr.
C: A Reference Manual - Fifth Edition
Prentice-Hall, 2002

[6] André Willms / C-Programmierung lernen - Anfangen, anwenden, verstehen
Addison-Wesley, Juni 1998

[7] Helmut Erlenkötter / C, Programmieren von Anfang an
Rowohlt Taschenbuch, Dezember 1999

[8] Karlheinz Zeiner / Programmieren lernen mit C
4. aktualisierte und überarbeitete Auflage
Hanser, Oktober 2000

[9] Jürgen Wolf
C von A bis Z
Galileo Press, September 2003

2 Einleitung

2.1 Historie von C

ALGOL60

↓

CPL

(Combined Programming Language)

↓ (-)

1967 BCPL

(Based Combined Programming Language)

Martin Richards (MIT [1])

Systemprogrammierung

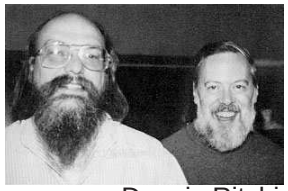
↓ (-)

1970 B

Ken Thompson (AT&T Bell Labs)

Implementierungssprache für UNIX

↓ (+ Datentypen)



Ken Thompson u. Dennis Ritchie

1972 **C**

Dennis M. Ritchie (Bell Labs)

Implementierungssprache für UNIX

[1] MIT = Massachusetts Institute of Technologie

1978 C-Sprachdefinition im Buch
The C Programming Language
von Brian W. Kernighan und Dennis M. Ritchie
(**K&R-C**)

1983 Bildung des ANSI[2] -Komitees X3J11 zur Standardisierung von C

1989, Dezember **ANSI C**

1990 Übernahme des ANSI Standards als internationaler Standard ANSI/ISO[3]/IEC[4] 9899:1990

1999, Dezember
überarbeitete Version des C-Standards:
C99 (ISO/IEC 9899:1999)

2011, Dezember
Neuauflage des C-Standards:
C11 (ISO/IEC 9899:2011)

[2] ANSI = **A**merican **N**ational **S**tandards **I**nstitute
[3] International Organization for Standardization
[4] **I**nternational **E**lectrotechnical **C**ommission

2.2 Eigenschaften von C

- einsetzbar für allgemeine Anwendungen in unterschiedlichen Bereichen
- effiziente Möglichkeiten zur Formulierung von Algorithmen
- verfügt über hinreichend viele Kontrollstrukturen
- viele Datentypen
- mächtige Menge von Operatoren
- Operatoren nicht auf zusammengesetzte Objekte (wie z.B. Zeichenketten) als Ganzes anwendbar (Ausnahme: Zuweisung von Strukturen)
- keine Anweisungen für Ein-/Ausgabe (statt dessen Bibliotheksfunktionen)
- sehr hohe Portabilität
- ermöglicht modulares Programmieren
- C-Compiler erzeugen i. Allg. effizienten Code

2.3 Beispielprogramm: Ausgabe von Zahlen (Version 1)

```
#include <stdio.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  return 0;   /* oder:  exit(EXIT_SUCCESS);   */
}
```

#define-Präprozessor-Direktive

definiert ein Makro.

Makro:

- Bezeichner, der im Programmtext, der seiner Definition folgt, durch den in der Definition festgelegten Text ersetzt wird.
- Innerhalb der in Anführungszeichen eingeschlossenen Strings findet jedoch keine Ersetzung statt.
- Makronamen werden konventionsgemäß mit Großbuchstaben geschrieben.
- Makros dienen der Abkürzung häufig benutzter Zeichenfolgen sowie der Benennung von Konstanten.

```
#include <stdio.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  return 0;   /* oder:  exit(EXIT_SUCCESS);   */
}
```

Die Zeilen

```
printf("LIMIT = %d\n", LIMIT);
while ( i <= LIMIT )
```

werden vom Präprozessor ersetzt durch:

```
printf("LIMIT = %d\n", 5);
while ( i <= 5 )
```



```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  exit(EXIT_SUCCESS);   /* oder:  return 0;   */
}
```

Makro **EXIT_SUCCESS**:

- definiert in der vom Compiler bereitgestellten Datei **stdlib.h**. Diese muss mittels **#include** in den Quelltext eingefügt werden.
- wird ersetzt durch den Wert 0 (als Rückgabewert des Programms steht 0 für erfolgreiche Programmausführung).

```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  exit(EXIT_SUCCESS);   /* oder:  return 0;   */
}
```

#include-Präprozessor-Direktive
wird vom Präprozessor durch den Inhalt der angegebenen Datei (**Header-Datei** – auch **Definitionsdatei** genannt) ersetzt.

stdio.h und **stdlib.h** sind Header-Dateien, die zur C-Implementierung gehören und Vereinbarungen für Funktionen der Standardbibliothek enthalten:

stdio.h	Bibliotheksfunktionen für Ein-/Ausgabe (z.B. **printf**)
stdlib.h	allgemeine Hilfsfunktionen (z.B. **exit**), Definition der Makros **EXIT_SUCCESS** und **EXIT_FAILURE**, u.a.


```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  exit(EXIT_SUCCESS);   /* oder:  return 0;[5]  */
}
```

exit(EXIT_SUCCESS);
Aufruf der Bibliotheksfunktion **exit**: bewirkt die Beendigung der Programmausführung (Die Kontrolle geht wieder an das Betriebssystem.).
Das Argument (**EXIT_SUCCESS**; allgemein ein Ausdruck, der einen Integer-Wert liefert) wird an das Betriebssystem zurückgegeben.
Die fehlerfreie Ausführung des Programms wird durch den Wert 0 (bzw. **EXIT_SUCCESS**), eine Fehlerbedingung durch einen Wert $\neq 0$ (bzw. **EXIT_FAILURE**) angezeigt. Innerhalb von **main** gilt: `exit(status);` $\Leftrightarrow$ `return status;`

[5] **C99**, C++: „`return 0;`“ darf am Ende von **main** weggelassen werden.

```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  exit(EXIT_SUCCESS);   /* oder:  return 0;   */
}
```

int main(void)

- **main** Name der C-Funktion, bei der die Programmausführung beginnt. (Jedes C-Programm muss eine Funktion namens **main** enthalten.)
- **int** zeigt hier an, dass **main** einen Integer-Wert an das Betriebssystem zurückgibt.
- **void** zeigt an, dass der Funktion **main** keine Argumente übergeben werden.

{} fassen mehrere Anweisungen zu einer syntaktischen Einheit (Block) zusammen (entsprechen **begin-end** in Pascal).


```
#include <stdio.h>
#include <stdlib.h>
#define LIMIT 5

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis 5 aus       */
  int i;

  printf("LIMIT = %d\n", LIMIT);
  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  exit(EXIT_SUCCESS);   /* oder:  return 0;   */
}
```

- **printf** dient der formatierten Ausgabe auf die Standardausgabe (**stdout**).
- **"%d\n"** String (Zeichenkette), der Text und Konvertierungsspezifikationen enthalten kann (Kontroll-String). Für jedes nachfolgende Argument sollte eine Konvertierungsspezifikation vorhanden sein.
- **%d** Konvertierungsspezifikation - gibt an, dass ein ganzzahliger Dezimalwert ausgegeben werden soll.
- **\n** Fluchtsymbol-Darstellung (*escape sequence*) für das Zeichen Zeilenende (*newline*).

2.4 Ausgabe von Zahlen (Version 2)

```
#include <stdio.h>

int main( void )   /* gibt die ganzen Zahlen  */
{                  /*   von 1 bis "limit" aus */
  int i, limit;

                   /* Wert fuer "limit" ein-  */
                   /*   lesen:                */
  scanf("%d", &limit);
  printf(
    "Die ganzen Zahlen von 1 bis %d:\n", limit);

  i = 1;
  while ( i <= limit )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  return 0;
}
```

---

scanf dient dem formatierten Lesen von der Standardeingabe (**stdin**).

& Adressoperator - liefert die Adresse seines Operanden (`&limit` ist die Speicheradresse der Variablen „`limit`“).

2.5 Beispielprogramm 2: Kopieren der Eingabe zur Ausgabe (Version 1)

```
#include <stdio.h>

int main( void )   /* kopiert Standardeingabe */
{                  /*   zur Standardausgabe   */
  int c;

  c = getchar();
  while ( c != EOF )
  {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

---

getchar() liest ein Zeichen von **stdin**. Ist kein Zeichen mehr vorhanden, wird **EOF** zurückgegeben.

EOF zeigt an, dass das Dateiende erreicht wurde. **EOF** ist ein Makro, das zu einem negativen Integer-Wert (häufig -1) expandiert.

putchar(c) gibt das Zeichen „c“ auf **stdout** aus.

Dateiende von der Tastatur (betriebssystemabhängig):

- unter UNIX-Systemen: <Ctrl>-d

2.6 Kopieren der Eingabe zur Ausgabe (Version 2)

```
#include <stdio.h>

int main( void )   /* kopiert Standardeingabe */
{                  /*   zur Standardausgabe   */
  int c;

  while ( (c = getchar()) != EOF )
    putchar(c);
  return 0;
}
```

Während in der Anweisung

```
c = getchar();
```

der Resultatwert der Zuweisung ignoriert wurde, wird er hier im Ausdruck

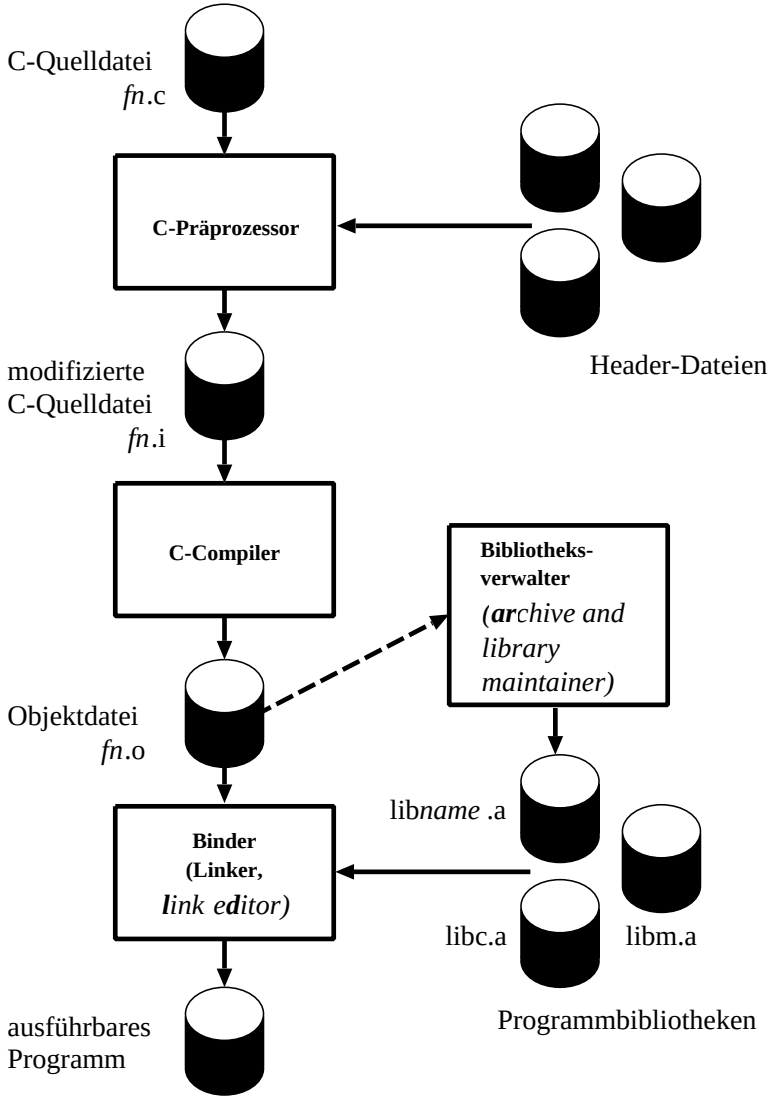
```
(c = getchar()) != EOF
```

für den Vergleich mit dem Wert **EOF** verwendet.

Ein Ausdruck, der nicht nur einen Ergebniswert liefert, sondern darüber hinaus eine weitere Aktion bewirkt (z.B. hier die Veränderung der Variablen c), wird als Ausdruck mit Seiteneffekt bezeichnet.

2.7 Übersetzen, Binden und Ausführen

Vom Quelltext zum ausführbaren Programm



Namenskonventionen

C-Quelldatei	*fn*.c
Header-Datei	*fn*.h

Übersetzen und Binden

unter UNIX: cc *optionen*$_{opt}$ *fn*.c *fn*$_2$.*c*$_{opt}$ ...

- übersetzt die angegebenen Quelldateien
- startet automatisch den Bindevorgang
- erzeugt eine ausführbare Datei

wichtige Optionen:

-o *exename*
: gibt der ausführbaren Datei den Namen *exename* anstatt a.out.

-c
: verhindert den Aufruf des Binders. Anstelle einer ausführbaren Datei wird für jede Eingabedatei eine **.o**-Datei (Objektdatei) erzeugt.

-lm
: weist den Binder an, die Objektbibliothek **lib<u>m</u>.a** (*IEEE Math Library*) in den Bindevorgang einzubeziehen.

-l*name*
: Objektbibliothek **lib*name*.a** hinzubinden.

-L*dir*
: spezifiziert ein Verzeichnis (*dir*), das vom Binder bei der Suche nach Objektbibliotheken durchsucht werden soll. Die Option kann mehrfach angegeben werden.

-I*dir*
: spezifiziert ein Verzeichnis (*dir*), das vom Präprozessor bei der Suche nach Header-Dateien durchsucht werden soll. Die Option kann mehrfach angegeben werden.

-D*makroname=definition*
: definiert ein Präprozessor-Makro. Die Option kann mehrfach angegeben werden.
Beispiel:
```
cc -o beispiel -DLIMIT=5 beispiel.c
```

-D*makroname*
: entspricht -D*makroname*=1

-E
: gibt an, dass nur der Präprozessor (nicht der Compiler) aufgerufen werden soll. Die Ausgabe des Präprozessors erfolgt auf **stdout**. Um die Ausgabe als Datei zu erhalten, muss zusätzlich die Option -o *dateiname* angegeben werden: z.B. -E -o *fn*.i

-g
: speichert Informationen in der Objektdatei (und schließlich im ausführbaren Programm), die es dem Benutzer eines symbolischen Debuggers ermöglichen, die Objekte (Variablen, Funktionen, ...) des Programmes über ihre Namen anzusprechen.

-O
: erzeugt optimierten Code.

wichtige Optionen des GNU C Compilers:

`-Wall` weist den Compiler an, durch die Ausgabe von zusätzlichen Warnungen auf möglichst viele (potenzielle) Problemstellen im Code hinzuweisen.

`-ansi -pedantic`
veranlasst den Compiler, durch zusätzliche Warnungen auf die Verwendung von Sprachkonstrukten hinzuweisen, die nicht dem ANSI-Standard entsprechen. (Soll der C99-Standard zugrunde gelegt werden, ist anstatt `-ansi` die Option `-std=c99` anzugeben.)

`-ansi -pedantic-errors`
Sprachkonstrukte, die nicht dem ANSI-Standard entsprechen, sollen nicht akzeptiert und als Fehler gemeldet werden.

Online-Hilfe (für weitere Informationen):

man cc [6]

[6] unter Linux ggf.: man gcc

Ausführen

exename *argumente*$_{opt}$ *io_umlenkung*$_{opt}$

Eingabeumlenkung (*input redirection*):

$<$*fid* Eingabe von der Standardeingabe (**stdin**) wird aus der Datei *fid* und nicht von der Tastatur gelesen.

Ausgabeumlenkung (*output redirection*):

$>$*fid* Umleiten der Standardausgabe (**stdout**) in die Datei *fid* (falls *fid* existiert, geht der alte Inhalt verloren).

$>>$*fid* Ausgabe auf **stdout** wird an das Ende von *fid* angehängt (falls die Datei *fid* nicht existiert, wird sie angelegt).

Beispiel:

```
cc -o kopiere kopiere1.c
./kopiere <kopiere1.c >kopiere2.c
```

2.8 Aufgaben

1. **Beispielprogramm „Ausgabe von Zahlen (Version 1)“**

   - Compilieren Sie das Programm und führen Sie es aus.
   - Löschen Sie die **#define**-Präprozessor-Direktive und definieren Sie das Makro LIMIT bei der Übersetzung. (Was passiert, wenn die Direktive im Programm nicht gelöscht wird?)
   - Rufen Sie nur den Präprozessor auf und sehen Sie sich dessen Ausgabe an. (Speichern Sie die Ausgabe in einer .i-Datei und versuchen Sie diese zu übersetzen.)
   - Lagern Sie die **#define**-Präprozessor-Direktive in eine Header-Datei mit dem Namen „mydef.h“ aus und fügen Sie diese Datei in das Programm ein.
     Hinweis:
     Die Form
     `#include <`*`Dateiname`*`>`
     wird für Header-Dateien benutzt, die zur C-Implementierung gehören. Für benutzerdefinierte Header-Dateien ist die folgende Form anzuwenden:
     `#include "`*`Dateiname`*`"`
   - Versuchen Sie durch Auslassen von Zeichen (Programmteilen) oder Hinzufügen von Text Fehlermeldungen zu erzeugen und untersuchen Sie diese.

2. **Beispielprogramm „Kopieren der Eingabe zur Ausgabe“**

   - Compilieren Sie das Programm und führen Sie es aus. (Überlegen Sie sich vorher, was für „Dateiende“ eingegeben werden muss!)
   - Listen Sie mit Hilfe des Programms eine Datei auf dem Bildschirm auf.
   - Benutzen Sie das Programm, um eine Datei zu kopieren. (Die Zieldatei sollte vorher nicht existieren.)
   - Hängen Sie mit Hilfe des Programms eine Kommentarzeile an die Kopie an.

3 Grundelemente der Sprache

3.1 Zeichensatz

- Großbuchstaben:

  ```
  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
  ```

- Kleinbuchstaben:

  ```
  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
  ```

- Ziffern: `0 1 2 3 4 5 6 7 8 9`

- Sonderzeichen:

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

- Leerzeichen und Zeilenendezeichen

- Steuerzeichen:
  horizontaler Tabulator, vertikaler Tabulator und Seitenvorschub

Für ein C-Programm sind zwei Zeichensätze erforderlich:

- Zeichensatz für die Übersetzung (Source-Zeichensatz)
- Zeichensatz für die Ausführung (Ausführungszeichensatz; Zeichen hieraus finden sich z.B. in Strings.)

Anforderungen:

- Beide Zeichensätze müssen die aufgezählten Zeichen enthalten.
- Der Ausführungszeichensatz muss zusätzlich die folgenden Steuerzeichen beinhalten:

  Alarmzeichen, Backspace und Zeilenrücklauf (*carriage return*).

Source- und Ausführungszeichensatz sind normalerweise gleich, sie können sich jedoch unterscheiden (Cross-Compiler).

3.2 Alternativdarstellungen (*Trigraph*-Sequenzen)

***Trigraph*-Sequenz**	**repräsentiertes Zeichen**	**C99**, C++: ***Digraph*-Sequenz**
??=	#	%:
??(	[	<:
??/	\	
??)	]	:>
??'	^	
??<	{	<%
??!	|	
??>	}	%>
??-	~	

Trigraph-Sequenzen werden auch in Zeichenkonstanten und Strings interpretiert.

Beispiele:

```
printf("Was ist das??/n");
```

wird interpretiert als `printf("Was ist das?\n");`

```
printf("Was ist das??!\n");
```

erzeugt die Ausgabe: `Was ist das|`

3.3 Kommentar

- beginnt mit `/*`
- endet mit `*/`
- Compiler behandelt einen Kommentar wie ein Leerzeichen.
- **C99**, C++: zusätzlich `//`-Kommentar: beginnt mit `//` und geht bis zum Ende der (logischen) Zeile.

Beispiele:

```
/* Kommentare können sich über
   mehrere Zeilen erstrecken.
*/

/***
 ***   Ein sehr langer Kommentar kann
 ***   auf diese Weise geschrieben
 ***   werden, um ihn von dem
 ***   umgebenden Programm abzuheben.
 ***/

/*************************************/
/*   Wenn Sie möchten, können Sie    */
/*   Kommentare auch umrahmen.       */
/*************************************/

/* /*-Kommentare
   können /* nicht */ geschachtelt werden.*/
```

Um einen Programmteil (einschließlich der /*-Kommentare) vorübergehend von der Übersetzung auszuschließen, können die Präprozessor-Direktiven **#ifdef** und **#endif** benutzt werden.

Beispiel:

```
#ifdef DEBUG
                    /* Wert des Makros LIMIT   */
                    /*   ausgeben:             */
  printf("LIMIT = %d\n", LIMIT);
#endif
```

Der Code zwischen den beiden Präprozessor-Direktiven wird nur dann an den Compiler weitergegeben, wenn ein Makro DEBUG definiert ist.

3.4 Grundsymbole (Token)

1. **Schlüsselwörter**
2. **Bezeichner**
3. **Operatoren**
4. **Punktsymbole**
5. **Konstanten**
6. **Strings**

3.4.1 Schlüsselwörter

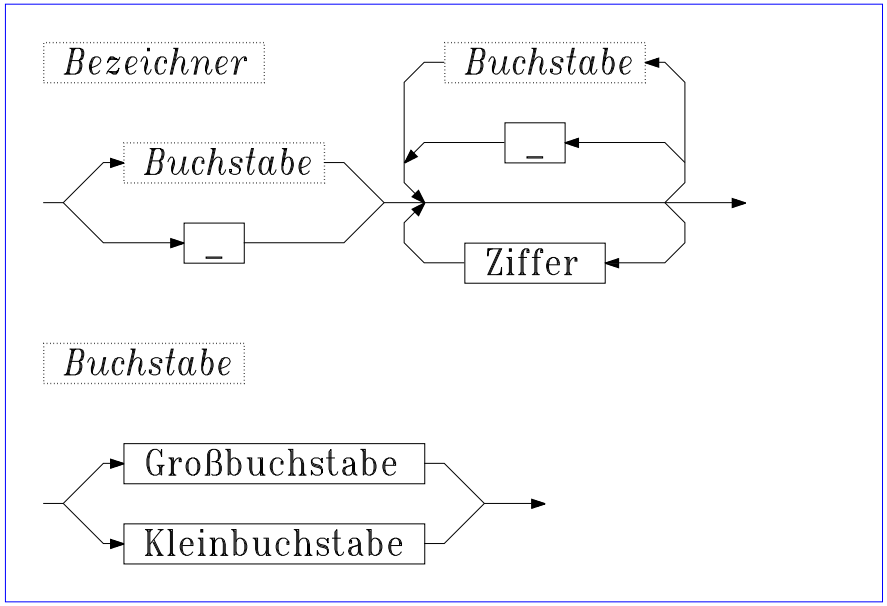
```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

zusätzliche Schlüsselwörter von **C99**:

```
inline      _Bool       _Imaginary
restrict    _Complex
```

- reservierte Wörter
  (können nicht umdefiniert werden)
- haben eine spezielle Bedeutung
- Kleinschreibung signifikant
- weitere Schlüsselwörter als Erweiterung möglich
  (z.B. bei CRAY: **asm**, **fortran**)

3.4.2 Bezeichner



- sind Namen für Variablen, Typen, Funktionen, Makros, ...
- Groß-/Kleinschreibung wird unterschieden
  (in **C99** auch für „externe" Namen verbindlich)
- Schlüsselwörter nicht als Bezeichner erlaubt
- sollten nicht mit „_" beginnen
  (Mit „_*Großbuchstabe*" oder „__" beginnende Bezeichner, in bestimmten Fällen auch andere mit „_" beginnende Bezeichner, sind für die C-Implementierung reserviert.)

- sind beliebig lange Zeichenfolgen, jedoch darf eine Implementierung nur eine bestimmte Anzahl von Zeichen als signifikant betrachten, wobei die Anzahl signifikanter Zeichen die folgenden Mindestwerte nicht unterschreiten darf:
  - ◇ 31 (C99: 63) Zeichen für „interne” Bezeichner
  - ◇ 6 (C99: 31) Zeichen für „externe” Bezeichner

Konventionen:

- Makronamen werden i. Allg. vollständig mit Großbuchstaben geschrieben
- andere Namen werden vollständig kleingeschrieben (oder in Groß-/Kleinschreibung)

3.4.3 Operatoren

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

Ein Operator spezifiziert eine Operation, die auf einem oder mehreren Operanden durchgeführt werden soll und eine (oder mehrere) der folgenden Auswirkungen hat:

- liefert einen Resultatwert
- ergibt die Bezeichnung für ein Objekt
  (z.B. `[]` (Index-Operator): `v[i]` repräsentiert wie ein Variablenname ein Datenobjekt und nicht nur dessen Inhalt. Ein Ausdruck mit dieser Eigenschaft (l-Wert) kann auch auf der linken Seite der Zuweisung verwendet werden.)
- verursacht Seiteneffekte
  (z.B. = (Zuweisungsoperator))

3.4.4 Punktsymbole

```
[  ]  (  )  {  }  *  ,  :  =  ;  ...  #
```

- Ein Punktsymbol ist ein Symbol mit syntaktischer (und evtl. semantischer) Bedeutung, das jedoch keine Operation repräsentiert, die einen Wert liefert.
- sind Grundsymbole, die sich in keine andere Kategorie einordnen lassen.
- Einige Symbole können je nach Kontext als Punktsymbol oder als Operator dienen.

Beispiel:

```
/*     Initialisierung (keine Operation)
       ↓                                          */
int i = 0, v[5];   /* [] sind Punktsymbol        */
/*           ↑ ↑
      vereinbaren v als Array mit 5 Elementen */

/* Vereinbarungen
  ------------------------------------------
   ausführbare Anweisungen:                   */

v[i] = 7;   /* [] sind Index-Operator          */
```


3.4.5 Konstanten

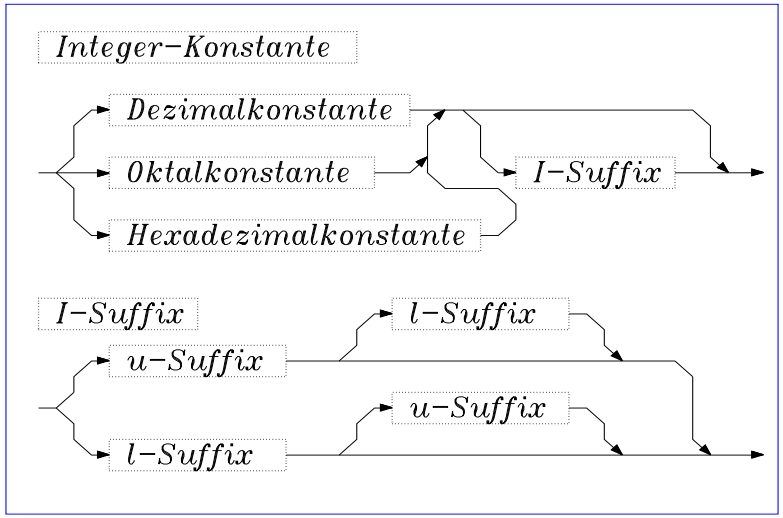
Eine Konstante

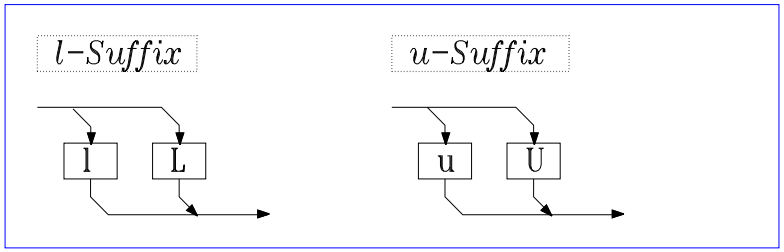
- repräsentiert einen numerischen Wert.
- besitzt einen bestimmten Datentyp.

Arten von Konstanten:

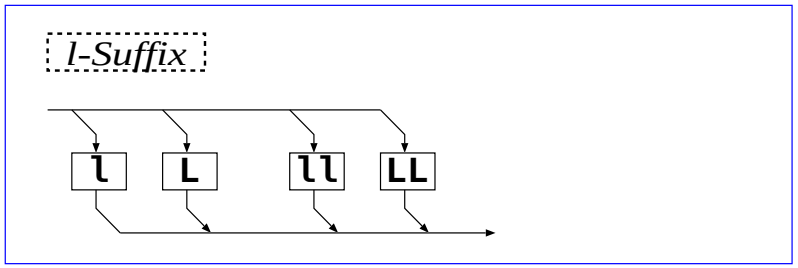
1. Integer-Konstanten
2. Gleitkommakonstanten (*floating constants*)
3. Character-Konstanten
4. Aufzählungskonstanten (*enumeration constants*)

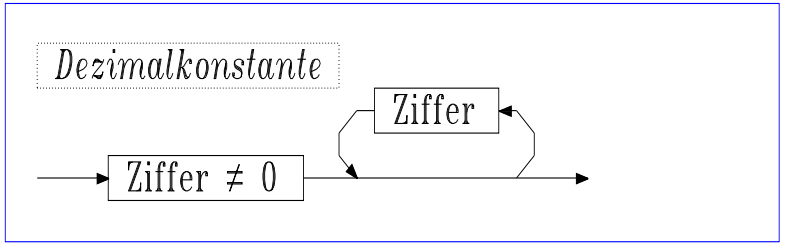
Integer-Konstanten

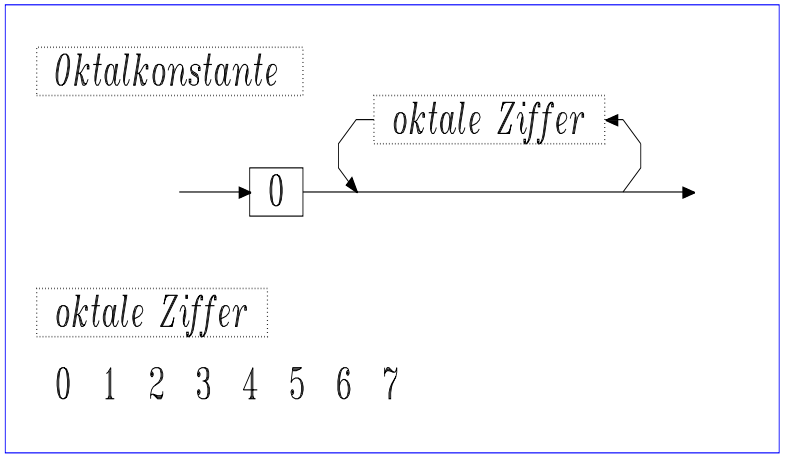


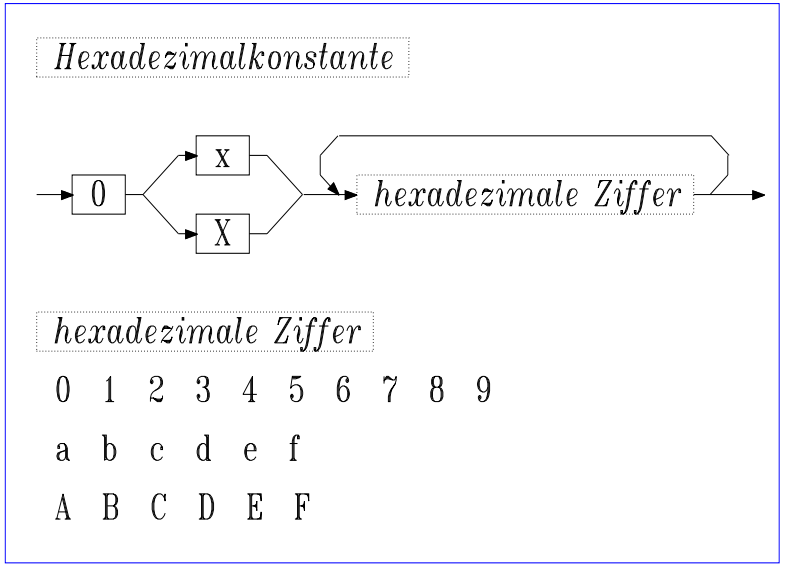


C99:









Bestimmung des Typs von Integer-Konstanten

mögliche Typen:

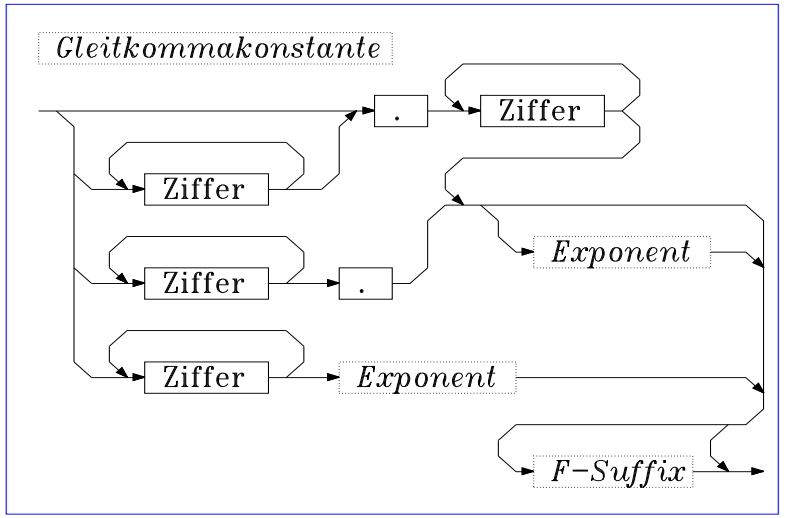
① **int** (z.B. 32 Bit, INT_MAX: 2147483647)
② **unsigned** int (z.B. UINT_MAX: 4294967295)
③ **long** int (z.B. 32 Bit)
④ **unsigned long** int
C99: ❺ **long long** int (mindestens 64 Bit)
C99: ❻ **unsigned long long** int

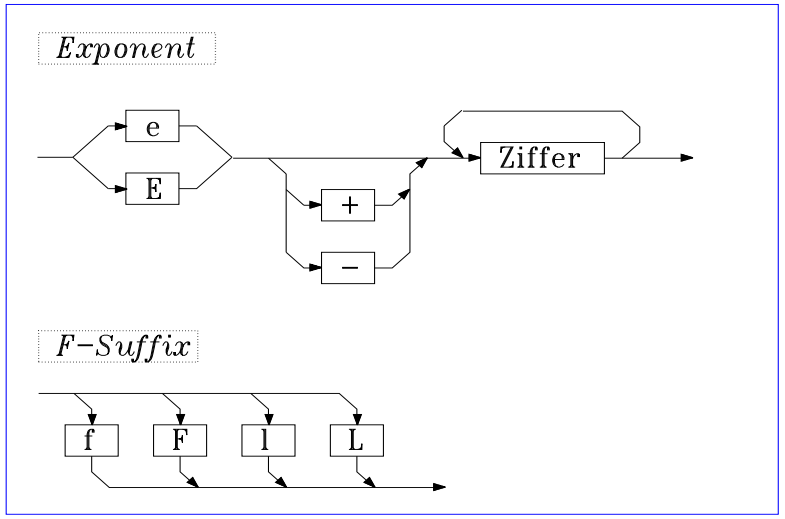
Suffix	Dezimalkonstante	Oktal- oder Hexadezimalkonstante
-	①, ③, ④ (C99: ①, ③, ❺)	①, ②, ③, ④, ❺, ❻
u(U)	②, ④, ❻	
l(L)	③, ④ (C99: ③, ❺)	③, ④, ❺, ❻
u(U) und l(L)	④, ❻	
ll(LL)	❺	❺, ❻
u(U) und ll(LL)	❻	

Anmerkung: Von den angegebenen Typen wird jeweils der *erste* ausgewählt, der die Darstellung des Wertes erlaubt.

Beispiele: `9` ①, `011L` ③, `0xffffffff` ②, `0x7fffffff` ① `3221225471` ④ (C99: ❺), `0xfull` ❻

Gleitkommakonstanten





Gleitkomma-Typen

Suffix	Typ der Konstante
`f` oder `F`	**float**
-	**double**
`l` oder `L`	**long double**

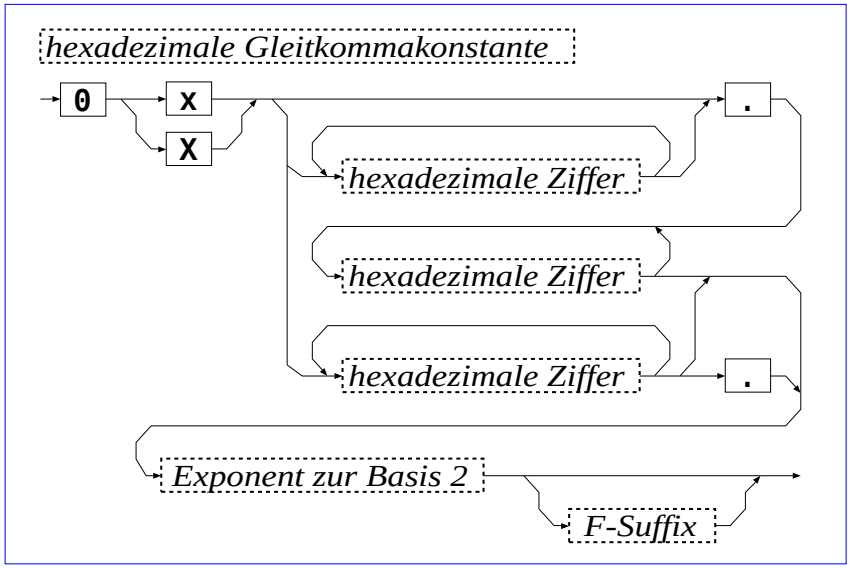
Beispiele:

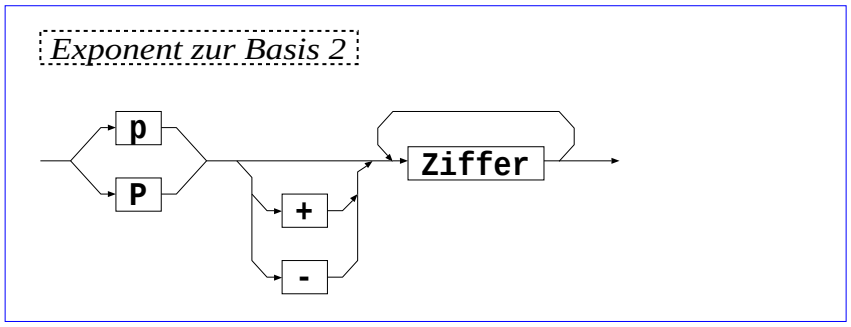
Konstante	Wert	Typ
`.1`	0.10	**double**
`0.1`	0.10	**double**
`0.1f`	0.10	**float**
`0.1L`	0.10	**long double**
`10.`	10.00	**double**
`1e2`	$1 \cdot 10^{2} = 100.00$	**double**
`1E-2`	$1 \cdot 10^{-2} = 0.01$	**double**
`1.5e+3`	$1.5 \cdot 10^{3} = 1500.00$	**double**

Vorsicht: Die Angabe vieler Dezimalstellen hat **keinen** Einfluss auf den Typ (die Genauigkeit) der Konstante.

```
long double pi1 = 3.14159265358979323846264338327;
long double pi2 = 3.14159265358979323846L;

printf("pi1 = %.20Lf\n", pi1);
     /* Ausgabe: pi1 = 3.14159265358979311600 */
printf("pi2 = %.20Lf\n", pi2);
     /* Ausgabe: pi2 = 3.14159265358979323851 */
```


C99: Hexadezimale Gleitkommakonstanten





Beispiele:

Konstante	Wert
`0x0.8p0`	$8 \cdot 16^{-1} = 0.5$
`0X1P-1`	$1 \cdot 16^{0} \cdot 2^{-1} = 0.5$
`0x.1P3`	$1 \cdot 16^{-1} \cdot 2^{3} = 0.5$
`0x1p+10`	$1 \cdot 2^{10} = 1024.0$
`0x1.000002p0f`	$1 + 2 \cdot 16^{-6} = 1.000000119209290$f (bei 24 Bit Mantisse der nächstgrößere auf 1.0f folgende **float**-Wert)

Vorsicht: Exponent **nicht** mit `e(E)` beginnen lassen: `0x1e+10` ist **keine** hexadezimale Gleitkommakonstante, sondern ein (zulässiger) Ausdruck, der den Integer-Wert `40` liefert.

Anmerkung:

Diese Art Konstanten spezifizieren kein Bitmuster, das der internen Repräsentation entspricht. Deshalb ist es auch nicht möglich, spezielle Werte wie z.B. *NaN* (*Not-a-Number*) in dieser Form anzugeben.

Character-Konstanten (Zeichenkonstanten)

- repräsentieren ein einzelnes Zeichen des Ausführungszeichensatzes
- das darzustellende Zeichen wird in (einfache) Hochkommata eingeschlossen (z.B.: `'3'`)
- die Zeichen `'` und Zeilenendezeichen können nur durch die entsprechende Fluchtsymbol-Darstellung angegeben werden
- Character-Konstanten haben den Datentyp **int**
- der Wert ist der numerische Wert des Zeichens, der durch die Ordnung der Zeichen im Zeichensatz gegeben ist (Ordnungszahl, maschinenabhängig)

Beispiele:

Konstante	Zeichen	Wert
`'\''`	'	z.B. 39
`'\"'` oder `'"'`	"	z.B. 34
`'\\'`	\	z.B. 92
`'\'`	Fehler: kein abschließendes '	
`'0'`	0	z.B. 48
`'\60'`	z.B. 0	48
`'\060'`	z.B. 0	48
`'\x30'`	z.B. 0	48
`'\0'`	NUL (Null-Zeichen)	0

Fluchtsymbol-Darstellungen (*escape sequences*)

escape sequence	repräsentiertes Zeichen
`\'`	'
`\"`	"
`\?`	?
`\\`	\
`\a`	Alarmzeichen
`\b`	Backspace
`\f`	Seitenvorschub (*form feed*)
`\n`	Zeilenendezeichen
`\r`	Zeilenrücklauf (*carriage return*)
`\t`	horizontaler Tabulator
`\v`	vertikaler Tabulator
`\`o_1 `\`o_1o_2 `\`$o_1o_2o_3$	Zeichen mit dem oktal angegebenen Wert
`\x`$h_1 \ldots h_n$	Zeichen mit dem hexadezimal angegebenen Wert
Anmerkung: Zur Darstellung des Zeichens \ ist innerhalb von Character-Konstanten und Strings **immer** die Fluchtsymbol-Darstellung erforderlich.	

Aufzählungskonstanten (*enumeration constants*)

- sind Bezeichner, die in einer Aufzählung zur Benennung von Integer-Konstanten verwendet werden
- haben den Typ **int**

Beispiel:

In der Aufzählung

```
enum Beispiel { blau, gruen, gelb };
```

sind `blau`, `gruen` und `gelb` Aufzählungskonstanten. Sie werden durch die Werte 0 (`blau`), 1 (`gruen`) und 2 (`gelb`) dargestellt.

3.4.6 Strings (*string literals*)

- bestehen aus Null, ein oder mehr Zeichen, die in Anführungszeichen (") eingeschlossen sind (z.B.: `"abc"`)
- innerhalb eines Strings sind Fluchtsymbol-Darstellungen erlaubt (notwendig für " und Zeilenendezeichen)
- repräsentiert ein Feld (*array*), dessen Elemente die angegebenen Zeichen sind. Als letztes Element wird ein Null-Zeichen (\0) angefügt, um das String-Ende zu markieren.
  Beispiel: | a | b | c | \0 | ⇔ | 97 | 98 | 99 | 0 |
- die Länge eines Strings ist die Anzahl der Zeichen (ohne \0)
- ein String der Länge *n* belegt einen Speicherbereich von *n*+1 Bytes
- aufeinander folgende Strings werden automatisch zu einem String verkettet (z.B.: `"ab" "c"`). Das Null-Zeichen wird erst **nach** einer Verkettung angehängt.

Beispiele:

String	Länge	Character-Feld
`"\""`	1	`"` `\0`
`"\'"` oder `"'"`	1	`'` `\0`
`""` (Null-String)	0	`\0`
`"Was ist das\?\?!\n"`		`...` `d` `a` `s` `?` `?` `!` `\n` `\0`
`"\x0dEnde"`	4	`\xde` `n` `d` `e` `\0`
`"\x0d" "Ende"`	5	`\xd` `E` `n` `d` `e` `\0`
`"\015Ende"`	5	`\xd` `E` `n` `d` `e` `\0`

3.4.7 Character- und Stringkonstanten für große Zeichensätze

Für die Codierung sehr großer Zeichensätze (z.B. einen japanischen Zeichensatz), die nicht innerhalb eines Bytes codiert werden können, beschreibt der C-Standard zwei Möglichkeiten:

1. Multibyte Characters

   z.B.: `'abc'`
   (*multi-character constant*,
   Wert abhängig von C-Implementierung)

2. Wide Characters

   z.B.: `L'm'` (*wide character constant*)
   `L"abc"` (*wide string literal*)

3.5 Regeln für das Einfügen von Zwischenraumzeichen

Zwischenraumzeichen:

- Leerzeichen (und Kommentar)
- Zeilenendezeichen
- horizontaler Tabulator
- vertikaler Tabulator
- Seitenvorschub

Zwischenraumzeichen (außerhalb von Zeichenkonstanten und Strings) dienen dazu, Token (Grundsymbole) voneinander zu trennen und sorgen - richtig eingesetzt - für die gute Lesbarkeit eines Quelltextes. Für ihre Verwendung gilt:

1. Zwischen 2 aufeinander folgenden Token, die Bezeichner, Schlüsselwort, Integer- oder Gleitkommakonstante sind, muss mindestens ein Zwischenraumzeichen stehen.
2. Zwischen 2 aufeinander folgenden Token, von denen mindestens eines Operator, Punktsymbol, Character-Konstante oder ein String ist, sind beliebig viele (auch null) Zwischenraumzeichen erlaubt.
3. Operatoren und Punktsymbole, die sich aus mehreren Zeichen zusammensetzen (z.B. „++” oder „`...`”) müssen ohne Zwischenraumzeichen geschrieben werden.

3.6 Fortsetzungszeilen

Die Zeichenkombination „\Zeilenendezeichen” bewirkt, dass die nächste Zeile als logische Fortsetzung der aktuellen Zeile betrachtet wird. („\Zeilenende” wird nicht Bestandteil der logischen Zeile.)

Beispiele:

```
                printf("Kleinbuchstaben: abcd\
efghijklmnopqrstuvwxyz\n");

/* besser: */     printf("Kleinbuchstaben: abcd"
                    "efghijklmnopqrstuvwxyz\n");

#define KLEINBUCHSTABEN "abcdefghijklmnopqrst"\
                         "uvwxyz"

#define KLEINBUCHSTABEN \
          "abcdefghijklmnopqrstuvwxyz"
```

C99, C++: auch `//`-Kommentare können auf diese Weise fortgesetzt werden:

```
//#define KLEINBUCHSTABEN \
          "abcdefghijklmnopqrstuvwxyz"
```

3.7 Aufgaben

1. Nehmen Sie an, die folgenden Zeichensequenzen würden von einem ANSI C Compiler verarbeitet.

   Welche Sequenzen würden als eine Folge von Grundsymbolen (Token) erkannt werden? Wie viele Grundsymbole würden in jedem Fall gefunden werden?
   (Lassen Sie sich nicht davon irritieren, dass einige Grundsymbol-Folgen in einem korrekten C-Programm nicht vorkommen können.)

   a. `X++Y`
   b. `-12uL`
   c. `1.37E+6L`
   d. `"String ""FOO"""`
   e. `"String+\"FOO\""`
   f. `x**2`
   g. `"X??/"`
   h. `B$C`
   i. `A*=B`

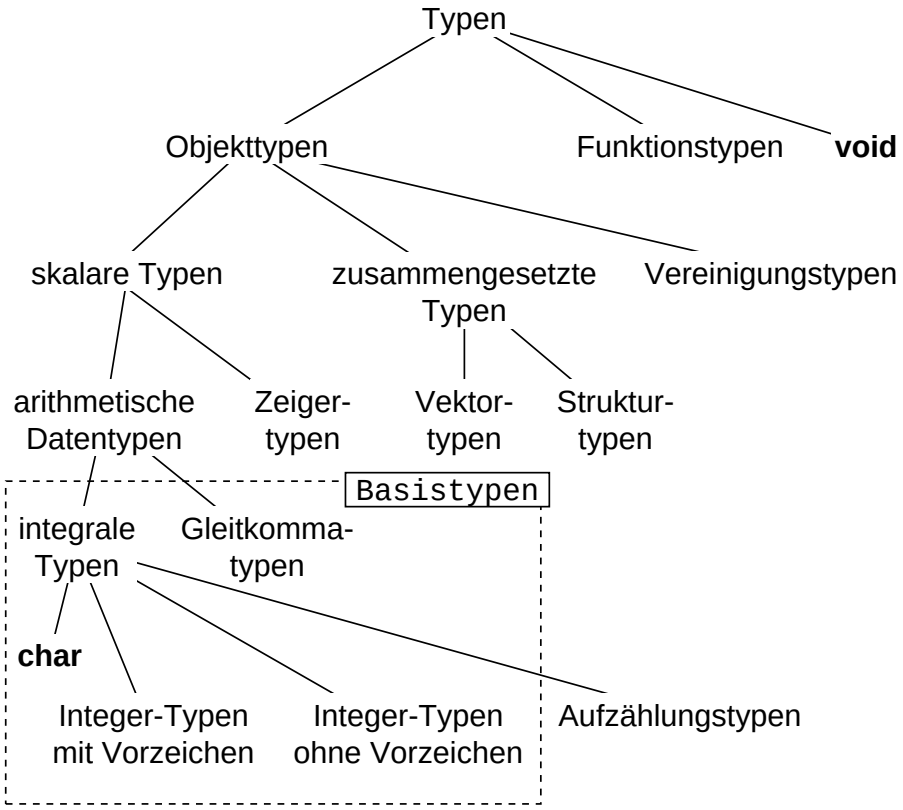
2. Die folgenden Bezeichner sind ungünstig gewählt. Was ist an ihnen zu beanstanden?

   a. `pipesendintake`
   b. `Const`
   c. `O77U`
   d. `SYS$input`

3. Bestimmen Sie die Längen der folgenden Strings und geben Sie an, welche Strings übereinstimmen.

	String	Länge
1	`"ab"`	
2	`"5\678"`	
3	`"5\0678"`	
4	`"5\00678"`	
5	`"5\000678"`	
6	`"a\"b"`	
7	`"a""b"`	
8	`"\xFFL"`	
9	`"\x000FFL"`	
10	`"\x0F" "FL"`	
11	`"12\'34"`	
12	`"12'\34"`	
13	`"\??/n"`	
14	`"\??\n"`	
15	`"%d%%\n"`	

4 Datentypen (Übersicht)



arithmetische Datentypen - Aufzählungstypen = Basistypen

4.1 Basistypen (Übersicht)

- **char**
- Integer-Typen mit Vorzeichen (*signed integer types*)
  - **signed char**
  - signed$_{opt}$ **short** int$_{opt}$
  - signed$_{opt}$ int$_{opt}$
  - signed$_{opt}$ **long** int$_{opt}$

  C99 signed$_{opt}$ **long long** int$_{opt}$
- Integer-Typen ohne Vorzeichen (*unsigned integer types*)
  - **unsigned char**
  - **unsigned short** int$_{opt}$
  - **unsigned** int$_{opt}$
  - **unsigned long** int$_{opt}$

  C99 **unsigned long long** int$_{opt}$

  C99 **_Bool** (oder **bool**[7])
- Gleitkommatypen [8]
  - **float**
  - **double**
  - **long double**

  C99 **float _Complex** (oder **float complex**[9])

  C99 **double _Complex** (oder **double complex**[9])

  C99 **long double _Complex** (oder **long double complex**[9])

[7] nach Einfügen von **<stdbool.h>**

[8] Hierzu zählen auch imaginäre Typen (**float _Imaginary**, . . .), die optionaler Bestandteil von C99 sind.

[9] nach Einfügen von **<complex.h>**

4.2 Vektortypen (*array types*)

erlauben die Vereinbarung von Feldern (*arrays*). Ein Feld besteht aus einer festen Anzahl gleichartiger Elemente. Der Typ eines Feldes (Vektortyp) beschreibt Anzahl und Datentyp der Feldelemente.

4.2.1 Eindimensionale Felder: Vektoren

Syntax der Vereinbarung:

type identifier **[** *const_expr*$_{opt}$ **]**

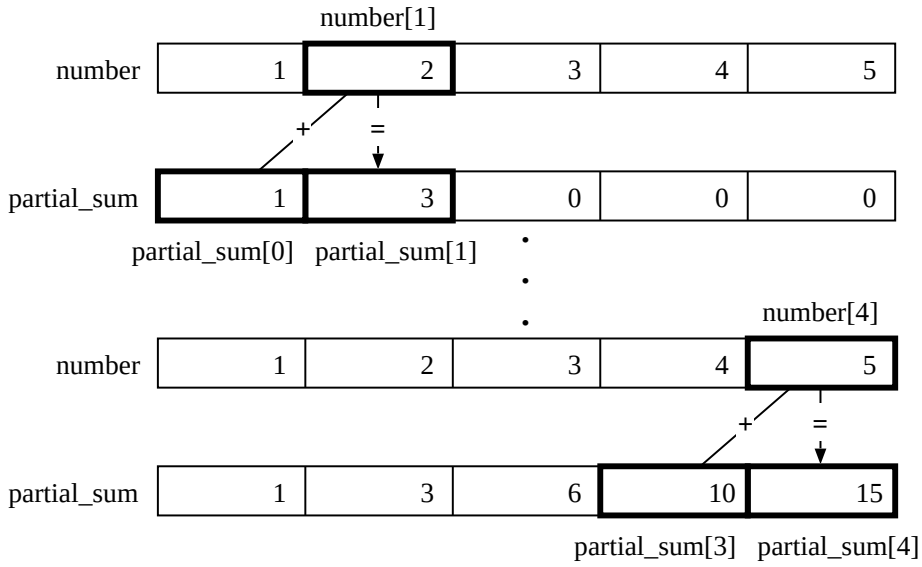
type	Datentyp der Vektorelemente
identifier	Name des Vektors
const_expr	• legt die Anzahl der Vektorelemente fest (Vektorlänge) • darf z.B. weggelassen werden, wenn die Größe des Vektors durch die Anzahl der Initialisierungswerte festgelegt wird.

Die Vektorelemente werden in einem zusammenhängenden Speicherbereich abgelegt. Der **Index-Operator** [ ] ermöglicht den Zugriff auf einzelne Vektorelemente:

identifier[0] . . . *identifier*[*const_expr*-1]

Beispiel:

```
#include <stdio.h>
#define MAXLEN 5

int main( void ) /* Partialsummen zur Folge   */
{                /*   der Zahlen von 1 bis 5  */
  int number[] = {1, 2, 3, 4, 5}, i,
      partial_sum[MAXLEN] = {1/* 0, 0, 0, 0*/};

  for (i = 1; i < MAXLEN; i++)
  {
    partial_sum[i] = partial_sum[i-1]+number[i];
    printf("1 + ... %d = %d\n", number[i],
      partial_sum[i]);
  }
  return 0;
}
```



Bereichsüberschreitung:

- liegt vor, wenn beim Zugriff auf *identifier*[*i*] der Index *i* außerhalb der zulässigen Grenzen liegt, also
  $i < 0$ oder $i \geq$ *const_expr*
  gilt.
- häufige Ursache für Laufzeitfehler (z.B. *Segmentation fault*)

Zeichenkettenkonstante (String):

- **konstanter** Vektor mit Elementen des Typs **char**
- darf indiziert, jedoch nicht verändert werden:

```
"abc"[1]  →  'b'
"abc"[1] = 'B';  /* unzulässig */
```


4.2.2 Zweidimensionale Felder: Matrizen

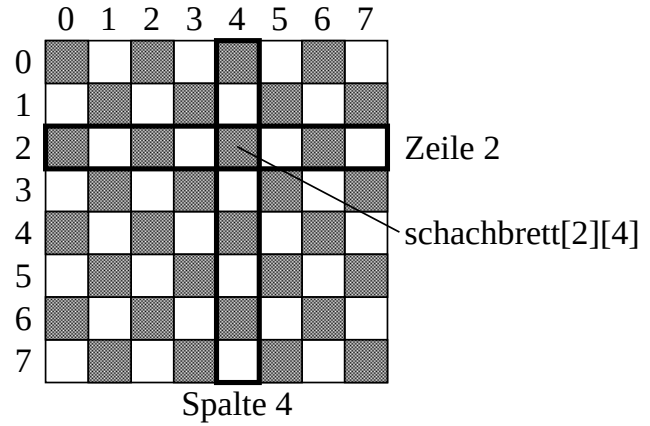
Syntax der Vereinbarung:

type identifier **[** *const_expr*$_1$ $_{opt}$ **][** *const_expr*$_2$ **]**

type	Datentyp der Matrixelemente
identifier	Name der Matrix
const_expr$_1$	Anzahl der **Zeilen**
const_expr$_2$	Anzahl der **Spalten**

Die Elemente einer Matrix werden zeilenweise in einem zusammenhängenden Speicherbereich angeordnet. Die Vorstellung eines zweidimensionalen Feldes entspricht dagegen einem rechteckigen Bereich mit Zeilen und Spalten.

Ein Beispiel für eine Matrix ist ein Schachbrett:



Zugriff auf ein Matrixelement

erfolgt durch zweifache Indizierung: *identifier* [*zeile*][*spalte*]

Für die Indizes *zeile* und *spalte* muss gelten:

$0 \leq$ *zeile* $<$ *const_expr*$_1$

$0 \leq$ *spalte* $<$ *const_expr*$_2$

Beispiel:

```
char schachbrett[8][8];
int z, s;

for (z = 0; z < 8; z++)
  for (s = 0; s < 8; s++)
    if ( (z+s)%2 == 0 )
      schachbrett[z][s] = 'X';
    else
      schachbrett[z][s] = ' ';
```

Beispiel für Initialisierung:

```
int m[][3] = { {1, 2, 3},
               {4, 5, 6} };
```

1	2	3
4	5	6

5 Arithmetische Typen, Operatoren und Ausdrücke

5.1 sizeof-Operator

unärer Operator (d.h. Verwendung wie ein Vorzeichen)

Syntax:

1. **sizeof(***typename***)**
2. **sizeof** *expression*

Resultatwert:

1. die Größe, in Bytes, eines Objekts mit dem (als Argument spezifizierten) Datentyp
2. die Größe, in Bytes, eines Objekts mit dem Typ des Ausdrucks
   - Der Typ des Ausdrucks (= Typ des Resultatwertes des Ausdrucks) wird zur Übersetzungszeit ermittelt.
   - Der Ausdruck selbst wird **nicht** berechnet.
     $\Rightarrow$ Er kann keine Seiteneffekte erzeugen.

Typ des Resultatwertes:

- abhängig von C-Implementierung
- integraler Typ ohne Vorzeichen: **size_t**
  - definiert in **<stddef.h>**
    (Definition vergleichbar mit Makrodefinition: z.B.
    `#define size_t unsigned long )`
  - Die Verwendung dieses Typs ist z.B. **sinnvoll**, **wenn** der Argumenttyp ein sehr großes Feld ist, dessen Größe möglicherweise nicht mehr als **int**-Wert darstellbar ist. **Andernfalls** kann das Ergebnis von **sizeof** problemlos nach **int** konvertiert werden.

Beispiele:

```
char c;  int  i;

sizeof c     ==  sizeof(char)
sizeof(i)    ==  sizeof(int)
sizeof(i=5)  ==  sizeof(int)
   /* Der Wert von i wird nicht verändert! */

double a[N][N][N];
int elem_size = sizeof(double);
size_t array_size = sizeof(a);

printf("%lu\n", (unsigned long) array_size);
printf("%zu\n", array_size);  // C99
```


5.2 Integer-Typen

dienen zur Darstellung ganzzahliger Werte. Der Bereich der darstellbaren Zahlen ist für jeden Typ durch Makros angegeben, die in **<limits.h>** definiert sind:

Typ	Größe/ Bytes	Wertebereich (kleinster, größter darstellbarer Wert)	
signed char	1	**SCHAR_MIN** $\leq$ -127, **SCHAR_MAX** $\geq$ +127	
unsigned char	1	0, **UCHAR_MAX** $\geq$ 255	
char	1	**CHAR_MIN**, **CHAR_MAX**	
short	≥ 2	**SHRT_MIN** $\leq$ -32767, **SHRT_MAX** $\geq$ +32767	
unsigned short	≥ 2	0, **USHRT_MAX** $\geq$ 65535	
int	$\geq 2^{10}$	**INT_MIN** $\leq$ -32767, **INT_MAX** $\geq$ +32767	
unsigned int	≥ 2	0, **UINT_MAX** $\geq$ 65535	
long	≥ 4	**LONG_MIN** $\leq$ -2147483647, **LONG_MAX** $\geq$ +2147483647	$2^{31}-1$
unsigned long	≥ 4	0, **ULONG_MAX** $\geq$ 4294967295	$2^{32}-1$
long long (C99)	≥ 8	**LLONG_MIN** $\leq$ -9223372036854775807, **LLONG_MAX** $\geq$ +9223372036854775807	$2^{63}-1$
unsigned long long (C99)	≥ 8	0, **ULLONG_MAX** $\geq$ 18446744073709551615	$2^{64}-1$

[10] Die Größe eines **int** entspricht i. Allg. der Wortlänge (Registerlänge) des Rechners.

unsigned-Typen haben immer die gleiche Größe wie der jeweils korrespondierende **signed**-Typ (außerdem gelten die gleichen Alignment-Anforderungen). Deshalb ist der größte darstellbare Wert eines **unsigned**-Typs i. Allg. doppelt so groß wie beim entsprechenden **signed**-Typ.

Für die Größen der **signed**-Typen gilt:

sizeof(short) $\leq$ **sizeof(int)** $\leq$ **sizeof(long)** $\leq$ **sizeof(long long)**

5.2.1 Darstellung von Zeichen

Hierzu dient der Typ **char**. Dieser entspricht **signed char** oder **unsigned char** (abhängig von C-Implementierung).

Ein **char**-Objekt belegt per Definition **1 Byte** [11] (**sizeof(char)** == 1).

für Source- und Ausführungszeichensatz gilt lt. C-Standard:

Alle Zeichen, die dezimale Ziffern repräsentieren, also die Zeichen `'0'-'9'`, folgen im Zeichensatz lückenlos und in aufsteigender Reihenfolge aufeinander:

Zeichen	ASCII-Wert
`'0'`	48
`'1'`	49
`'2'`	50
...	
`'9'`	57

$\Rightarrow$ durch Subtraktion der Zeichenkonstante `'0'` von einem Ziffernzeichen erhält man den der Dezimalziffer entsprechenden Wert: `'5'-'0'` $\rightarrow 5$ (ASCII: 53 - 48 $\rightarrow$ 5)

[11] **CHAR_BIT** (definiert in **<limits.h>**) gibt die Anzahl Bits pro Byte an: **CHAR_BIT** ≥ 8

5.2.2 Funktionen in <ctype.h>

- jede der folgenden Funktionen erwartet ein Argument vom Typ **int**
- Wert des Argumentes: ein als **unsigned char** darstellbarer Wert oder **EOF**
- jede Funktion liefert einen Resultatwert vom Typ **int**

Zeichenklassen-Tests:

Ist das Argument *c* ein Zeichen aus der jeweiligen Zeichenklasse, wird ein Wert $\neq 0$, andernfalls 0 zurückgegeben.

Funktion	Zeichenklasse
isdigit(*c***)**	dezimale Ziffern (`'0'-'9'`)
isxdigit(*c***)**	hexadezimale Ziffern (`'0'-'9'`, `'a'-'f'`, `'A'-'F'`)
islower(*c***)**	Kleinbuchstaben (`'a'-'z'`)
isupper(*c***)**	Großbuchstaben (`'A'-'Z'`)
isalpha(*c***)**	Buchstaben
isalnum(*c***)**	Buchstaben und Ziffern
isspace(*c***)**	Zwischenraumzeichen (`' '`, `'\n'`, `'\r'`, `'\t'`, `'\v'`, `'\f'`)
C99 **isblank(***c***)**	Leerzeichen (`' '`, `'\t'`)
ispunct(*c***)**	Interpunktionszeichen (*punctuation marks*): druckbare Zeichen außer Leerzeichen, Buchstaben und Ziffern

Funktion	Zeichenklasse
isgraph(*c***)**	druckbare Zeichen, kein Leerzeichen
isprint(*c***)**	druckbare Zeichen (einschließlich Leerzeichen): jedes Zeichen, das eine Abdruckstelle auf einem Ausgabegerät beansprucht
iscntrl(*c***)**	Kontrollzeichen: nicht druckbare Zeichen

Umwandlungsfunktionen:

Funktion	Beschreibung
tolower(*c***)**	ist *c* ein Großbuchstabe, dann wird der entsprechende Kleinbuchstabe zurückgegeben, andernfalls ist der Resultatwert *c*
toupper(*c***)**	ist *c* ein Kleinbuchstabe, dann wird der entsprechende Großbuchstabe zurückgegeben, andernfalls ist der Resultatwert *c*

Beispiel:

```
#include <stdio.h>
#include <ctype.h>
...
 char c;
 ...
 do
 {
   printf("Weiter? (j/n) ");
   scanf(" %c", &c);
 } while ( ((c = tolower(c)) != 'j') && (c != 'n') );
```

5.2.3 Logische Werte

werden durch die integralen Typen dargestellt[12]:

ganzzahliger Wert	entspricht
0	false
jeder Wert $\neq 0$	true

Beispiel:

```
#include <stdio.h>
#include <ctype.h>

int main( void )
{
  int c;

    ...
            /* ignoriere Zwischenraumzeichen: */
  while ( isspace(c = getchar()) )
    ;
 /* c ist ein Zeichen ungleich Zwischenraum   */
 /*   oder EOF                                */
    ...
```

Vereinbarung eigener Konstanten z.B. durch:

```
#define FALSE 0
#define TRUE  1
```

[12] **C99**: eigener Typ **_Bool**, Makros in **<stdbool.h>**: **bool**, **true**, **false**
C++: **bool**, **true** und **false** sind Schlüsselwörter.

5.2.4 Aufzählungstyp (*enumerated type*)

vereinbart eine Menge von konstanten Integer-Werten, die durch Namen bezeichnet werden.

Syntax:

enum *type_tag*$_{opt}$ {*e_list*}

enum *type_tag*$_{opt}$ {*e_list* **,** } **C99**

enum *type_tag*

type_tag
benennt den Typ.

e_list

- Liste von (durch Komma getrennten) Bezeichnern
- diesen Bezeichnern kann explizit ein bestimmter Wert (konstanter Ausdruck mit integralem Typ) zugeordnet werden.
- für Bezeichner ohne explizite Wertzuweisung gilt:
  - der erste Bezeichner der Liste erhält den Wert 0
  - ansonsten ergibt sich der Wert aus dem Wert des vorangehenden Bezeichners durch Addition von 1
- Aufzählungskonstanten müssen lediglich den Typ **int** haben; weitere Überprüfungen *können* erfolgen (z.B. Zuweisung einer Aufzählungskonstante an eine Variable eines anderen Aufzählungstyps)

Beispiele:

```
/* Typdefinition: */

enum Farben {
  schwarz = 1,
  rot,          /* rot == 2            */
  blau,         /* blau == 3           */
  gruen,        /* gruen == 4          */
  pink = 2,
  tuerkis,      /* tuerkis == 3        */
  gelb          /* gelb == 4           */
};

/* Variablendefinitionen: */

enum Farben f1, f2 = schwarz;
enum { Montag, Dienstag, Mittwoch, Donnerstag,
       Freitag, Samstag, Sonnabend = Samstag,
       Sonntag } Tag, Termin;

/* Verwendung der Variablen z.B. in Zuweisung
   oder Abfrage:
*/
    ...
  f1 = rot;
    ...
  if ( f1 == schwarz ) ...
```



```
enum { BELL='\a', BACKSPACE='\b', TAB='\t',
       NEWLINE='\n', VTAB='\v', RETURN='\r' };

/* Alternative zu Makrodefinitionen:          */
/*   #define BELL '\a'                       */
/*     ...                                   */
/*   #define RETURN '\r'                     */
```

Selbstgestrickter **bool**-Typ:

```
/* Dieses Beispiel kann mit ANSI C-, C99-     */
/* oder C++-Compiler übersetzt werden.        */
#include <ctype.h>

#ifndef __cplusplus

#  if __STDC_VERSION__ >= 199901L
#    include <stdbool.h>
#  else
     enum bool {false, true};
#    define bool enum bool
#  endif

#endif

int main( void ) {
  bool name_found = false;  int c;
    ...
  name_found = (isalpha(c) != 0);
  /* ANSI C: Vergleich mit 0 notwendig, da im */
  /*   Gegensatz zu C99 und C++ keine auto-   */
  /*   matische Konvertierung sicherstellt,   */
  /*   dass der zugewiesene Wert 0 oder 1 ist */
```

5.2.5 Integer-Erweiterung (*integral promotion*)

Ein Objekt des Typs **_Bool** (C99), **char** oder **short int** (**signed** oder **unsigned**), sowie ein Objekt mit einem Aufzählungstyp darf innerhalb eines Ausdrucks immer anstelle eines **int** bzw. **unsigned int** Objektes benutzt werden.

Der kleinere Typ wird dann automatisch in den Typ **int** bzw. **unsigned int** umgewandelt:

Integer-Erweiterung

- stellt sicher, dass der ursprüngliche Wert erhalten bleibt.
- wird automatisch auch auf die Argumente von **printf** angewandt.

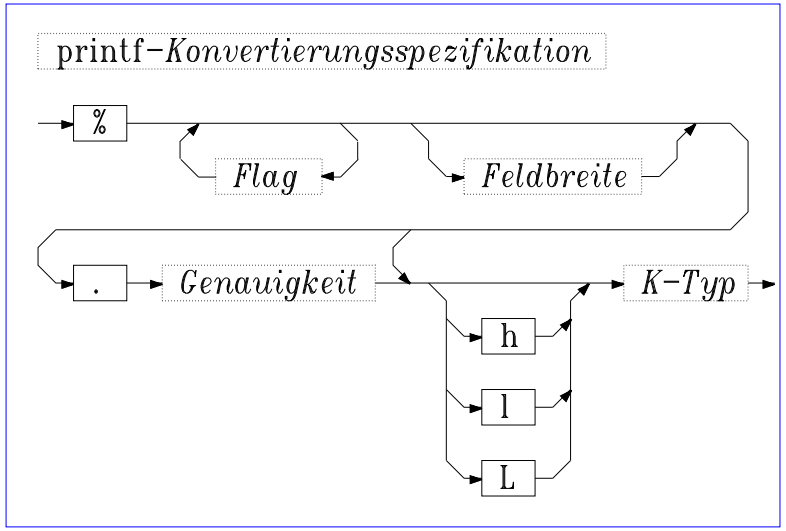
Beispiel:

```
char c;
  ...
printf("%d\n", c);
        ↓     ↓
   erwarteter  char
  Argumenttyp:  ↓ Integer-Erweiterung
        int     int  ✓
```

für ursprünglichen Typ *T* gilt:	erweiterter Typ
sizeof(*T*) < **sizeof**(**int**)	**int**
T ist **signed** und **sizeof**(*T*) == **sizeof**(**int**)	**int**
T ist **unsigned** und **sizeof**(*T*) == **sizeof**(**int**)	**unsigned int**

5.2.6 Formatierte Ausgabe von Integer-Typen

printf-Konvertierungsspezifikationen



Flag

- `-` linksbündige Ausgabe
- `+` eine Zahl wird in jedem Fall mit Vorzeichen ausgegeben.
- *Leerzeichen* einer Zahl, die ohne Vorzeichen ausgegeben wird, wird ein Leerzeichen vorangestellt.
- `0` eine Zahl wird mit führenden Nullen ausgegeben.
- `#` eine Oktalzahl wird mit einer führenden Null ausgegeben; einer Hexadezimalzahl $\neq 0$ wird der Präfix `0x` bzw. `0X` vorangestellt.

Feldbreite minimale Breite des Ausgabefeldes (Zahl oder `*`; `*` bewirkt, dass die Feldbreite durch das nächste Argument im Aufruf von **printf** bestimmt wird.)

Genauigkeit minimale Anzahl auszugebender Ziffern (ggf. werden führende Nullen erzeugt; zulässige Angaben: Zahl oder `*`)

h *(halfword)* bewirkt Umwandlung des Argumentes vor der Ausgabe nach **short** bzw. **unsigned short**.

l zeigt an, dass das zugehörige Argument den Typ **long** bzw. **unsigned long** hat.

L → formatierte Ausgabe von Gleitkommatypen

K-Typ Formatbuchstabe, der die Art der Umwandlung bestimmt

In C99 sind unmittelbar vor *K-Typ* außer den Buchstaben h, l und L auch folgende Angaben zulässig:

Buchstabe(n)	Bedeutung
hh	vor Ausgabe Umwandlung nach **char**
ll	Argument vom Typ **long long** bzw. **unsigned long long**
j	Argument vom Typ **intmax_t**[13] bzw. **uintmax_t**[13]
z	Argument vom Typ **size_t**[14]
t	Argument vom Typ **ptrdiff_t**[14]

[13] definiert in **<stdint.h>** (C99)
[14] definiert in **<stddef.h>**

K-Typ	Argument-typ	Argument wird dargestellt als
d, **i**	**int**	Dezimalzahl
o	**unsigned**	Oktalzahl
x	**unsigned**	Hexadezimalzahl (Ziffern > 9: `'a'-'f'`)
X	**unsigned**	Hexadezimalzahl (Ziffern > 9: `'A'-'F'`)
u	**unsigned**	Dezimalzahl
c	**int**	einzelnes Zeichen (Argument → **unsigned char**; h, hh, ll, j, z, t nicht zulässig; l hat spezielle Bedeutung bei Verwendung großer Zeichensätze)
%	**-**	bewirkt die Ausgabe eines %-Zeichens

Resultatwert von **printf**: Anzahl der ausgegebenen Zeichen (Ein negativer Wert weist auf einen Fehler hin.)

Beispiele:

```
int tag = 5, monat = 7;

printf("%.2d.%02d.", tag, monat);
                /* Ausgabe: 05.07.          */
```

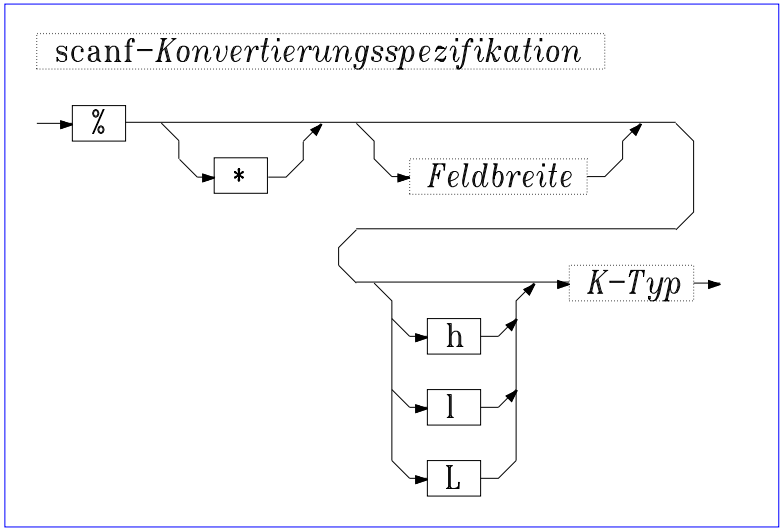
```
#include <stdio.h>

int main( void ) {
  int spos = 123, sneg = -spos;
  int width;

  printf("<%d><%d>\n", spos, sneg);
                 /* <123><-123>                */

  printf("<%6d><%6d>\n",       /* Feldbreite 6 */
    spos, sneg);  /* <   123><  -123>           */

  printf("<%-6d><%-6d>\n",     /* Flag -       */
    spos, sneg);  /* <123   ><-123  >           */

  printf("<%+6d><%+6d>\n",     /* Flag +       */
    spos, sneg);  /* <  +123><  -123>           */

  printf("<% d><% d>\n",  /* Flag <Leerzeichen>*/
    spos, sneg);  /* < 123><-123>               */

  printf("<%06d><%06d>\n",     /* Flag 0       */
    spos, sneg);  /* <000123><-00123>           */

  printf(              /* mindestens 4 Ziffern */
    "<%6.4d><%6.4d>\n", spos, sneg);
                 /* <  0123><-0123>            */
```



```
  printf("<%1d><%1d>\n", spos, sneg);
                 /* <123><-123>                */

  printf("<%.0d><%.0d><%.0d>\n", spos, sneg, 0);
                 /* <123><-123><>              */

  width = 8;
  printf(              /* Feldbreite variabel  */
    "<%*d><%0*d>\n", width, spos, width-2, sneg);
                 /* <     123><-00123>         */

  printf("%o, %#o\n",          /* Flag #       */
    8, 8);       /* 10, 010                    */

  printf("%x, %X, %#x\n",      /* Flag #       */
    10, 10, 10);  /* a, A, 0xa                 */

  printf("%hd, %hu, %hx\n", -1, -1, -1);
                 /* -1, 65535, ffff            */

  printf("%d, %u, %c, %%\n", '0', '0', '0');
                 /* 48, 48, 0, %               */

  return 0;
}
```

5.2.7 Formatierte Eingabe von Integer-Typen

scanf-Konvertierungsspezifikationen



* unterdrückt die Zuweisung des gelesenen Wertes (entspricht dem Überspringen des nächsten Eingabefeldes). Daher benötigt diese Konvertierungsspezifikation auch *kein* Argument aus der Argumentliste des **scanf**-Aufrufs.

Feldbreite maximale Breite des Eingabefeldes (Dezimalzahl). Das Standard-Eingabefeld beginnt in der Regel mit dem nächsten Zeichen in der Eingabe, das kein Zwischenraumzeichen ist (vorausgehende Zwischenraumzeichen werden überlesen[15]), und endet vor dem ersten Zeichen, das der Formatspezifikation nicht mehr entspricht.

[15] ⇒ **scanf** liest über Zeilenende hinweg

h in Verbindung mit den Formatbuchstaben:
- **d**, **i**, **n** zeigt an, dass das zugehörige Argument ein Zeiger auf ein **short**-Objekt ist.
- **u**, **o**, **x**[16] zeigt an, dass das zugehörige Argument ein Zeiger auf ein **unsigned short**-Objekt ist.

l in Verbindung mit den Formatbuchstaben:
- **d**, **i**, **n** zugehöriges Argument: **long ***
- **u**, **o**, **x**[16] zugehöriges Argument: **unsigned long ***

L → formatierte Eingabe von Gleitkommatypen

In C99 sind unmittelbar vor einem der Formatbuchstaben **d**, **i**, **n**, **u**, **o** oder **x**[16] außer den Buchstaben h und l auch folgende Angaben zulässig:

Buchstabe(n)	Typ des zugehörigen Arguments
hh	**signed char *** bzw. **unsigned char ***
ll	**long long *** bzw. **unsigned long long ***
j	**intmax_t**[17] ***** bzw. **uintmax_t**[17] *****
z	**size_t**[18] *****
t	**ptrdiff_t**[18] *****

[16] darf auch als Großbuchstabe angegeben werden.
[17] definiert in **<stdint.h>** (C99)
[18] definiert in **<stddef.h>**

K-Typ	Argumenttyp	Eingabe
d	**int ***	[+|-]*Dezimalkonstante* (führende Nullen zulässig; keine Interpretation als Oktalkonstante)
i	**int ***	[+|-]*Integer-Konstante* (ohne *I-Suffix*)
u	**unsigned ***	[+|-]*Dezimalkonstante* (führende Nullen zulässig; keine Interpretation als Oktalkonstante)
o	**unsigned ***	[+|-]*Oktalkonstante* (führende `0` optional)
x[16]	**unsigned ***	[+|-]*Hexadezimalkonstante* (`0x`- bzw. `0X`-Präfix optional)
c	**char ***	ein einzelnes Zeichen (Zwischenraumzeichen werden wie jedes andere Zeichen übertragen; *Feldbreite* > 1 nur zulässig, falls Argument auf **char**-Feld zeigt)
n	**int ***	benötigt keine Eingabe; die Anzahl der bisher durch diesen **scanf**-Aufruf gelesenen Zeichen wird der Variablen, deren Adresse übergeben wurde, zugewiesen. (auch in Verbindung mit **printf** möglich, wobei die Anzahl der bisher ausgegebenen Zeichen zugewiesen wird.)
%	**-**	% (keine Zuweisung)

Resultatwert von **scanf**: Anzahl der *zugewiesenen* Datenelemente oder **EOF**, falls vor Zuweisung des ersten Datenelementes das Dateiende erreicht wird.

Beispiele:

```
int i;

scanf("%d", &i);  /* Eingabe: 0109         */
                  /* neuer Wert von i: 109 */
scanf("%i", &i);  /* Eingabe: 0109         */
                  /* neuer Wert von i: 8   */
```

Vorsicht:

Der Typ der Variable, an die der gelesene Wert zugewiesen werden soll, muss *exakt* zur Konvertierungsspezifikation passen.

```
int i = 0;
  ...
scanf("%c", &i);  /* Eingabe: 0109         */
```

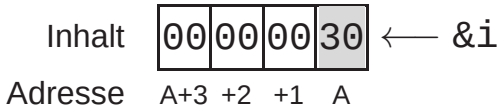
erwarteter Argumenttyp: ↓ `%c` — ↓ `&i`

char * $\neq$ **int *** $\Rightarrow$ Fehler

Fehler bleibt auf Rechner mit „*little endian*“-Architektur (z.B. PC mit Intel-Prozessor) u.U. unbemerkt. (Andererseits gibt z.B. der GNU C Compiler beim Übersetzen mit der Option `-Wall` eine Warnung aus, die auf das Problem hinweist.)

„*little endian*“-Systeme[19]

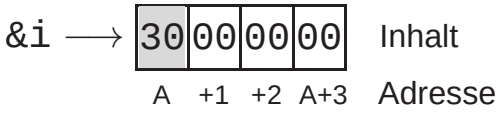
Byteanordnung (im Speicher) von rechts nach links



neuer Wert von `i`: `0x30 = 48 ≙ '0'`

„*big endian*“-Systeme[20]

Byteanordnung von links nach rechts



neuer Wert von `i`: `0x30000000 = 805306368`

[19] z.B. PCs mit Intel-Prozessoren

[20] z.B. IBM PowerPC-Systeme wie JUQUEEN (Jülich Blue Gene/Q)

Das erste Argument von **scanf** (Kontroll-String) darf neben Konvertierungsspezifikationen auch noch andere Zeichen enthalten:

Zwischenraumzeichen
: veranlasst, dass alle Zwischenraumzeichen bis zum nächsten Zeichen, das kein Zwischenraumzeichen ist, überlesen werden.
(I. Allg. bewirkt jedoch bereits die Konvertierungsspezifikation, dass dem Eingabefeld vorausgehende Zwischenraumzeichen überlesen werden.)

jedes andere Zeichen
: verlangt, dass das nächste Zeichen in der Eingabe das angegebene Zeichen ist.

Beispiele:

```
int i; char ch;

scanf("%d", &i);     /* Eingabe:      123      */
                     /* neuer Wert von i: 123  */
scanf(" %i", &i);    /* Eingabe:      123      */
                     /* neuer Wert von i: 123  */
scanf("%c", &ch);    /* Eingabe:  123          */
                     /* neuer Wert von ch: ' ' */
scanf(" %c", &ch);   /* Eingabe:      123      */
                     /* neuer Wert von ch: '1' */
scanf(" Anzahl :%d", &i);
                     /* Eingabe: Anzahl: 123   */
```

Vorsicht:

```
int t, m;

printf("Bitte Tag eingeben: ");
scanf("%d\n", &t); /* Problem: Monat muss be- */
                   /*   reits hier eingegeben */
                   /*   werden.               */
                   /* besser: scanf("%d", ... */
printf("Bitte Monat eingeben: ");
```

Beispiele für %n:

```
int count, n, i;

printf("Beispiel %d:%n\n", count, &n);
for (i = 0; i < n; i++)
  putchar('-');
putchar('\n');
```

```
int n1, n2;

scanf(" ");     /* Zwischenraum überlesen     */
scanf("Anzahl%n", &n1);
scanf(" ");     /* Zwischenraum überlesen     */
scanf(":%n", &n2);
if ( (n1 != 6) || (n2 != 1) )
  printf("Fehler in Eingabe!\n");
```


Beispiel:

```
#include <stdio.h>

int main( void )
{
  int Zaehler, s1, s3, i, vH;

  Zaehler = scanf("%d km %*dkm%d km", &s1, &s3);
  printf("Zaehler = %d\n", Zaehler);
  printf("s1 = %d; s3 = %d\n", s1, s3);

  Zaehler = scanf("%d %% %2d", &vH, &i);
  printf("Zaehler = %d\n", Zaehler);
  printf("vH = %d %%; i = %d\n", vH, i);

  return 0;
}
```

Eingabe:

```
4 km   21km    33 km
   5%  -37
```

Ausgabe:

```
Zaehler = 2
s1 = 4; s3 = 33
Zaehler = 2
vH = 5 %; i = -3
```

Eingabe:

```
4 km   21 km    33 km
          ↑
```

Fehlerstelle

Ausgabe:

```
Zaehler = 1
s1 = 4; s3 = -1
Zaehler = 0
vH = 0 %; i = -2140743990
```

5.2.8 Aufgaben

1. Untersuchen Sie, wie sich der von Ihnen benutzte Compiler verhält, wenn in einem Programm eine Aufzählungskonstante an eine Variable eines *anderen* Aufzählungstyps zugewiesen wird.
   Gibt es eine Option, die den Compiler veranlasst, auf das Problem hinzuweisen?

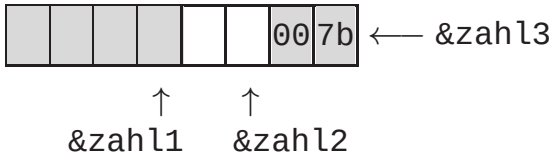
2. Das folgende Programm ist inkorrekt. Wenn es mit dem GNU C Compiler (ohne Optimierung) auf einem PC-Linux-System ausgeführt wird, und dabei jeweils die vorgeschlagenen Werte eingegeben werden, erhält man die im Anschluss an das Programm aufgelistete Ausgabe.

```
#include <stdio.h>

int main(void) {
  int       zahl1;
  short int zahl2, zahl3 = 123;

  printf("Bitte Werte für ‘Zahl1’ und "
    "‘Zahl2’ (z.B. 1 2) eingeben: ");
  scanf("%d%d", &zahl1, &zahl2);
  printf(
    "zahl1 = %d\nzahl2 = %d\nzahl3 = %d\n",
    zahl1, zahl2, zahl3);

  printf("Bitte Werte für ‘Zahl2’ und "
    "‘Zahl3’ (z.B. -2 3) eingeben: ");
  scanf("%d%d", &zahl2, &zahl3);
  printf(
    "zahl1 = %d\nzahl2 = %d\nzahl3 = %d\n",
    zahl1, zahl2, zahl3);
  return 0;
}
```


Ausgabe:

```
Bitte Werte ... (z.B. 1 2) eingeben: 1 2
zahl1 = 0
zahl2 = 2
zahl3 = 123
Bitte Werte ... (z.B. -2 3) eingeben: -2 3
zahl1 = 65535
zahl2 = 0
zahl3 = 3
```

Man kann herausfinden, dass bei Verwendung des GNU C Compilers die drei Variablen des Programms hintereinander in der folgenden Reihenfolge im Speicher des „*little endian*“-Systems ablegt werden:



a) Geben Sie genau an, welche Werte die einzelnen Bytes im Speicher annehmen, wenn **scanf** die eingegebenen Zahlen an die entsprechenden Variablen zuweist, und erklären Sie so die Ausgabe des Programms.
b) Korrigieren Sie das Programm, ohne jedoch die Typen der Variablen abzuändern.

3. Modifizieren Sie das auf Seite 81 aufgelistete Beispielprogramm zur formatierten Eingabe von Integer-Typen:
   a) Testen Sie jeweils den Resultatwert von **scanf**. Im Fehlerfall ist der Rest der aktuellen Eingabezeile auf **stdout** auszugeben. ($\Rightarrow$ Der zweite Aufruf von **scanf** soll immer aus einer neuen Zeile lesen.)
   b) Ändern Sie den Typ der Variablen:
      `s1`, `s3` $\rightarrow$ **long**
      `i`, `vH` $\rightarrow$ **short**