PEARSON Education

ADDISON-WESLEY

Helmut Herold

C-Kompaktreferenz

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

3 Headerdateien

3.1 Überblick über die Headerdateien

Headerdatei	ANSI-C	Kurzbeschreibung
<assert.h>	x	Testmöglichkeiten in einem Programm
<ctype.h>	x	Umwandlung/Klassifikation von Zeichen
<dirent.h>		Directory-Einträge
<errno.h>	x	Fehlerkonstanten
<float.h>	x	Limits/Eigenheiten für Gleitpunkttypen
<limits.h>	x	Implementierungskonstanten
<math.h>	x	Mathematische Konstanten/Funktionen
<setjmp.h>	x	Nichtlokale Sprünge
<signal.h>	x	Signale
<stdarg.h>	x	Variabel lange Argumentlisten
<stddef.h>	x	Standarddefinitionen
<stdio.h>	x	Standard-E/A-Bibliothek
<stdlib.h>	x	Allgemein nützliche Funktionen
<string.h>	x	Stringbearbeitung
<time.h>	x	Datum und Zeit

Tabelle 3-1: Die Headerdateien im Überblick

3.2 Primitive Systemdatentypen

Datentyp	Kurzbeschreibung
caddr_t	Speicheradresse
clock_t	Uhrticks
dev_t	Gerätenummern

Tabelle 3-2: Überblick über die primitiven Systemdatentypen

Datentyp	Kurzbeschreibung
fpos_t	Schreib-/Lesezeiger-Position in Datei
mode_t	Öffnungsmodus für Dateien
off_t	Dateigrößen und Offsets
ptrdiff_t	Ergebnis bei Zeigersubtraktion
size_t	Größe von Objekten
ssize_t	Rückgabetyp von Funktionen, die eine Byteanzahl liefern
time_t	Zähler für die Kalenderzeitsekunden
wchar_t	Vielbyte-Zeichen

Tabelle 3-2: Überblick über die primitiven Systemdatentypen (Fortsetzung)

3.3 Limits, Konstanten

In Tabelle 3-3 werden folgende Abkürzungen verwendet:

l = <limits.h>, s = <stdio.h>,
* = optional. (Ist kein * angegeben, so muß Konstante in der entsprechenden Headerdatei definiert sein.)

Konstante	Header	Minimalwert	Beschreibung
CHAR_BIT	l	8	max. Bitanzahl für 1 Byte
CHAR_MAX	l	SCHAR_MAX bzw. UCHAR_MAX	max. Wert für **char**
CHAR_MIN	l	0 oder SCHAR_MIN	min. Wert für **char**
FOPEN_MAX	s	8	max. Anzahl offener Dateien pro Prozeß
INT_MAX	l	32767	max. Wert für **int**
INT_MIN	l	-32768	min. Wert für **int**
LONG_MAX	l	2147483647	max. Wert für **long**
LONG_MIN	l	-2147483648	min. Wert für **long**
MB_LEN_MAX	l	1	max. Bytes für Vielbyte-Zeichen
NAME_MAX	l*	14	max. Byteanzahl für Dateinamen
OPEN_MAX	l*	16	max. Anzahl offener Dateien je Prozeß
PATH_MAX	l*	255	max. Byteanzahl in relativen Pfadnamen
SCHAR_MAX	l	127	max. Wert für **signed char**
SCHAR_MIN	l	-127	min. Wert für **signed char**
SHRT_MAX	l	32767	max.Wert für **signed short**
SHRT_MIN	l	-32767	min. Wert für **signed short**

Tabelle 3-3: Limits und Konstanten im Überblick

Konstante	Header	Minimalwert	Beschreibung
SSIZE_MAX	l	32767	max. Wert für Typ **ssize_t**
TMP_MAX	s	10000	max. Anzahl unterschiedlicher temporärer Dateinamen
UCHAR_MAX	l	255	max. **unsigned char**-Wert
UINT_MAX	l	65535	max. **unsigned int**-Wert
ULONG_MAX	l	4294967295	max. **unsigned long**-Wert
USHRT_MAX	l	65535	max. **unsigned short**-Wert

Tabelle 3-3: Limits und Konstanten im Überblick (Fortsetzung)

3.4 Von ANSI C vordefinierte Makros

Vordefiniertes Makro	Bedeutung
__LINE__	Zeilennummer in der momentanen Quelldatei (ganzzahlige Konstante)
__FILE__	Name der momentanen Quelldatei (Stringkonstante)
__DATE__	Übersetzungsdatum der aktuellen Quelldatei (`"mmm tt jjjj"`; z.B. `"Jun 14 1989"` oder `"Jun  4 1989"`)
__TIME__	Übersetzungszeit der aktuellen Quelldatei (`"hh:mm:ss"`; z.B.: `"14:32:53"`).
__STDC__	Erkennungsmerkmal für ANSI-C-Compiler: Hat diese Konstante den Wert 1, so liegt ein ANSI-C-Compiler vor-

Tabelle 3-4: Von ANSI C vordefinierte Makros

3.5 <stdarg.h> – Abarbeiten variabler Argumentlisten

Hier sehen Sie als Beispiel eine zentrale Fehlerroutine, die wie *printf* funktioniert. Hierfür gibt es zwei Möglichkeiten: mit der Funktion *vsprintf* (in `fehl_meld1`) oder durch wiederholten Aufruf von *va_arg* (in `fehl_meld2`).

```
void  fehl_meld1(const char *fmt, ...) /*------- fehl_meld1 ----------------------*/
{
   va_list  az;
   char     puffer[MAX_ZEICHEN];

   va_start(az, fmt);
   vsprintf(puffer, fmt, az);
   fprintf(stderr, "%s\n", puffer);
   va_end(az);
}
```

```
void  fehl_meld2(const char *fmt, ...) /*------- fehl_meld2 ----------------------*/
{
   va_list  az;
   char  puffer[MAX_ZEICHEN];

   va_start(az, fmt);
   while (*fmt) {
      if (*fmt != '%') {
         putc (*fmt, stderr);
      } else {
         switch(*++fmt) {
             case 'c' : fprintf(stderr, "%c", va_arg(az, int));     break;
             case 'd' : fprintf(stderr, "%d", va_arg(az, int));     break;
             case 'f' : fprintf(stderr, "%f", va_arg(az, double));  break;
             case 's' : fprintf(stderr, "%s", va_arg(az, char *));  break;
               .........
         }
      }
      fmt++;
   }
   fprintf(stderr, "\n");
   va_end(az);
}
```

3.6 <ctype.h> – Klassifizieren oder Umwandeln von Zeichen

Funktion	liefert TRUE, wenn ....., und sonst FALSE
int *isalnum*(int *zeich*)	*zeich* ein alphanumerisches Zeichen (A...Z,a...z,0...9) ist
int *isalpha*(int *zeich*)	*zeich* ein Buchstabe aus dem Alphabet (A...Z,a...z) ist
int *iscntrl*(int *zeich*)	*zeich* ein Steuerzeichen (Hexa-Code: 0x00 ... 0x1f und 0x7f) ist
int *isdigit*(int *zeich*)	*zeich* eine Ziffer (0...9) ist
int *isgraph*(int *zeich*)	*zeich* ein druckbares Zeichen (Leerzeichen ausgenommen) ist
int *islower*(int *zeich*)	*zeich* ein Kleinbuchstabe (a...z) ist
int *isprint*(int *zeich*)	*zeich* ein druckbares Zeichen (Hexa-Code: 0x20..0x7E) ist
int *ispunct*(int *zeich*)	*zeich* ein druckbares, aber kein Leer- oder alphanumumerisches Zeichen ist
int *isspace*(int *zeich*)	*zeich* ein Zwischenraumzeichen (Leerzeichen, **\f**, **\n**, **\r**, **\t**, **\v**) ist
int *isupper*(int *zeich*)	*zeich* ein Großbuchstabe (A...Z) ist
int *isxdigit*(int *zeich*)	*zeich* eine hexadezimale Ziffer (0...9,a...f,A...F) ist

Tabelle 3-5: Die von ANSI C vorgeschriebenen Funktionen in <ctype.h>

Zusätzlich müssen noch die beiden folgenden Funktionen in <ctype.h> definiert sein:

Funktion	Bedeutung
`int tolower(int zeich)`	Ist *zeich* ein Großbuchstabe, dann liefert *tolower* den entsprechenden Kleinbuchstaben; ansonsten wird *zeich* unverändert zurückgegeben.
`int toupper(int zeich)`	Ist *zeich* ein Kleinbuchstabe; dann liefert *toupper* den entsprechenden Großbuchstaben, ansonsten wird *zeich* unverändert zurückgegeben.

Tabelle 3-6: Zwei weitere von ANSI C vorgeschriebene Funktionen in <ctype.h>

Daneben dürfen weitere Funktionen definiert sein. Ihr Name muß mit **is*k*** oder **to*k*** (***k***=Kleinbuchstabe) beginnen, wie z.B.:

`int` **`isascii`**`(int zeich)`
liefert TRUE, wenn *zeich* ein ASCII-Zeichen ist, sonst FALSE.

3.7 <assert.h> – Testmöglichkeit mit der assert-Funktion

`NDEBUG`
Wenn definiert, werden *assert*-Aufrufe vom Compiler ignoriert.

`void` **`assert`**`(int ausdruck);`
Wenn *ausdruck* == 0 ist, wird das Programm mit einer Fehlermeldung beendet.

3.8 <errno.h> – Anzeigen von Fehlern in Bibliotheksfunktionen

`EDOM`
Ganzzahlige Konstante für Domainfehler; wird bei ungültigem Argument gesetzt (z.B. **`sqrt(-2.3)`**).

`ERANGE`
Ganzzahlige Konstante für Bereichsfehler; wird bei nicht darstellbarem Ergebnis (z.B. wenn dieses zu groß ist) gesetzt.

`errno`
Viele Bibliotheksfunktionen setzen diese globale **int**-Variable auf einen von 0 verschiedenen Wert, wenn bei ihrer Ausführung ein Fehler auftritt. ANSI C garantiert nur, daß `errno` beim Programmstart auf 0 gesetzt wird; auch wird `errno` niemals von den Bibliotheksfunktionen zurückgesetzt.

3.9 <limits.h> – Limits für ganzzahlige Datentypen

Konstantenname	Garantierter Mindestwert	Beschreibung
CHAR_BIT	8	maximale Bitanzahl für ein Byte
SCHAR_MIN	-127	Minimalwert für **signed char**
SCHAR_MAX	+127	Maximalwert für **signed char**
UCHAR_MAX	255	Maximalwert für **unsigned char**
CHAR_MIN	SCHAR_MIN oder 0	Minimalwert für **char**
CHAR_MAX	SCHAR_MAX oder UCHAR_MAX	Maximalwert für **char**
MB_LEN_MAX	1	maximale Bytes für Vielbyte-Zeichen
SHRT_MIN	-32767	Minimalwert für **short int**
SHRT_MAX	+32767	Maximalwert für **short int**
USHRT_MAX	65535	Maximalwert für **unsigned short**
INT_MIN	-32767	Minimalwert für **int**
INT_MAX	+32767	Maximalwert für **int**
UINT_MAX	65535	Maximalwert für **unsigned int**
LONG_MIN	-2147483647	Minimalwert für **long int**
LONG_MAX	+2147483647	Maximalwert für **long int**
ULONG_MAX	4294967295	Maximalwert für **unsigned long int**

Tabelle 3-7: Von ANSI C vorgeschriebene Limits für ganze Zahlen in <limits.h>

3.10 <float.h> – Limits/Eigenschaften für Gleitpunktdatentypen

Konstante	Beschreibung
FLT_RADIX	Basis für die Exponentendarstellung; meist 2 (>=2)
FLT_MANT_DIG	Anzahl der Mantissenstellen in **float**
DBL_MANT_DIG	Anzahl der Mantissenstellen in **double**
LDBL_MANT_DIG	Anzahl der Mantissenstellen in **long double**
FLT_DIG	Anzahl der signifikanten dezimalen Ziffern in **float** (>=6)
DBL_DIG	Anzahl der signifikanten deimalen Ziffern in **double** (>=10)
LDBL_DIG	Anzahl der signifikanten deimalen Ziffern in **long double** (>=10)
FLT_MIN_EXP	Kleinster negativer FLT_RADIX-Exponent für **float**-Werte
DBL_MIN_EXP	Kleinster negativer FLT_RADIX-Exponent für **double**-Werte
LDBL_MIN_EXP	Kleinster negativer FLT_RADIX-Exponent für **long double**
FLT_MIN_10_EXP	Kleinster negativer Zehnerexponent für **float**-Werte (<=-37)

Tabelle 3-8: Von ANSI C vorgeschriebene Limits/Eigenschaften für Gleitpunktzahlen in <float.h>

Konstante	Beschreibung
DBL_MIN_10_EXP	Kleinster negativer Zehnerexponent für **double**-Werte (<=-37)
LDBL_MIN_10_EXP	Kleinster negativer Zehnerexponent für **long double** (<=-37)
FLT_MAX_EXP	Größter FLT_RADIX-Exponent für **float**-Werte
DBL_MAX_EXP	Größter FLT_RADIX-Exponent für **double**-Werte
LDBL_MAX_EXP	Größter FLT_RADIX-Exponent für **long double**-Werte
FLT_MAX_10_EXP	Größter Zehnerexponent für **float**-Werte (>=+37)
DBL_MAX_10_EXP	Größter Zehnerexponent für **double**-Werte (>=+37)
LDBL_MAX_10_EXP	Größter Zehnerexponent für **long double**-Werte (>=+37)
FLT_MAX	Größter darstellbarer endlicher **float**-Wert (>=1E+37)
DBL_MAX	Größter darstellbarer endlicher **double**-Wert (>=1E+37)
LDBL_MAX	Größter darstellbarer endlicher **long double**-Wert (>=1E+37)
FLT_EPSILON	Kleinster positiver **float**-Wert x, für den noch gilt: 1.0+x!=x (<=1E-5)
DBL_EPSILON	Kleinster positiver **double**-Wert x, für den noch gilt: 1.0+x!=x (<=1E-9)
LDBL_EPSILON	Kleinster positiver **long double**-Wert x, für den noch gilt: 1.0+x!=x (<=1E-9)
FLT_MIN	Kleinster normalisierter positiver **float**-Wert (<=1E-37)
DBL_MIN	Kleinster normalisierter positiver **double**-Wert (<= 1E-37)
LDBL_MIN	Kleinster normalisierter positiver **long double**-Wert (<=1E-37)

Tabelle 3-8: Von ANSI C vorgeschriebene Limits/Eigenschaften für Gleitpunktzahlen in <float.h> (Fortsetzung)

In <float.h> ist noch eine Konstante definiert, die den Rundungsmodus für Gleitpunktwerte festlegt:

FLT_ROUNDS
–1 = nicht festgelegt
0 = zu 0 hin
1 = zum nähesten darstellbaren Wert hin
2 = auf +∞ zu
3 = auf -∞ zu

3.11 <math.h> – Mathematische Funktionen

Funktion	Liefert als Ergebnis
double **acos**(double *x*)	Arcuscosinus von *x*
double **asin**(double *x*)	Arcussinus von *x*
double **atan**(double *x*)	Arcustangens von *x*
double **atan2**(double *y*, double *x*)	Arcustangens von *y*/*x*

Tabelle 3-9: Von ANSI C vorgeschriebene mathematische Funktionen in <math.h>

Funktion	Liefert als Ergebnis
double **ceil**(double *x*)	kleinste ganze Zahl nicht kleiner als *x*; wird zum Aufrunden verwendet
double **cos**(double *x*)	Cosinus von *x*
double **cosh**(double *x*)	Cosinus hyperbolicus von *x*
double **exp**(double *x*)	e^x (e steht 2.718281..)
double **fabs**(double *x*)	Absolutwert von *x*
double **floor**(double *x*)	größte ganze Zahl nicht größer als *x*; wird zum Abrunden verwendet
double **fmod**(double *x*, double *y*)	Gleitpunktrest von *x*/*y* (*x*-i**y*, wobei i eine solche Ganzzahl ist, daß das Ergebnis das gleiche Vorzeichen wie *x* und kleineren Betrag als y hat
double **frexp**(double *wert*, int **exp*)	wandelt *wert* in eine normalisierte double-Form [0.5,1) * 2^{*exp} um, wobei der Rückgabewert aus dem Intervall [0.5,1) ist.
double **ldexp**(double *x*, int *exp*)	*x* * 2^{exp}
double **log**(double *x*)	natürlichen Logarithmus von *x*
double **log10**(double *x*)	Zehnerlogarithmus von *x*
double **modf**(double *wert*, double **iptr*)	Nachkommateil von *wert*, Vorkommateil wird in **iptr* gespeichert.
double **pow**(double *x*, double *y*)	x^y
double **sin**(double *x*)	Sinus von *x*
double **sinh**(double *x*)	Sinus hyperbolicus von *x*
double **sqrt**(double *x*)	Quadratwurzel von *x*
double **tan**(double *x*)	Tangens von *x*
double **tanh**(double *x*)	Tangens hyperbolicus von *x*

Tabelle 3-9: Von ANSI C vorgeschriebene mathematische Funktionen in <math.h> (Fortsetzung)

Zusätzlich ist eine Konstante definiert, die von Funktionen zurückgegeben wird, falls der wirkliche Wert nicht darstellbar ist:

HUGE_VAL
: sehr großer **double**-Wert

3.12 <stddef.h> – Standarddefinitionen

Datentyp	Bedeutung
ptrdiff_t	vorzeichenbehafteter Ganzzahltyp für das Subtraktionsergebnis zweier Zeiger
size_t	vorzeichenloser Ganzzahltyp für das Ergebnis des **sizeof**-Operators
wchar_t	ganzzahliger Datentyp, der den ganzen Wertebereich aller vorgegebenen Zeichen (wie z.B. auch ganz spezieller Grafikzeichen) abdecken kann.
NULL	Nullzeigerkonstante (oft als 0, 0L oder (void*)0 definiert)

Tabelle 3-10: Von ANSI C vorgeschriebene Datentypen in <stddef.h>

Zusätzlich muß noch folgendes Makro in <stddef.h> definiert sein:

offsetof(*struktur_typ*, *struktur_komponente*)
liefert das Offset von struktur_komponente in struktur_typ (in Bytes, wobei size_t der Rückgabetyp ist). Falls es sich bei der angegebenen struktur_komponente um ein Bitfeld handelt, ist das Verhalten undefiniert.

3.13 <stdlib.h> – Allgemein nützliche Funktionen

Hier sind unter anderem auch die beiden in <stddef.h> vorhandenen Datentypen size_t und wchar_t und die NULL-Konstante definiert. Daneben sind noch die folgenden beiden Datentypen

div_t Strukturtyp für den Rückgabewert der Funktion *div*

ldiv_t Strukturtyp für den Rückgabewert der Funktion *ldiv*

und die folgenden vier Konstanten definiert:

Konstante	Bedeutung
EXIT_SUCCESS	Exit-Status für erfolgreiche Beendigung. Diese Konstante wird als Argument für exit verwendet.
EXIT_FAILURE	Exit-Status für nicht erfolgreiche Beendigung. Diese Konstante wird als Argument für exit verwendet.
RAND_MAX	Maximaler Rückgabewert für Funktion *rand*
MB_CUR_MAX	Maximale Byteanzahl für Vielbyte-Zeichen (niemals > MB_LEN_MAX)

3.13.1 Allokieren und Freigeben von Speicherplatz

void ***malloc**(size_t *groesse*);
allokiert einen Speicherbereich von groesse Bytes.

`void *calloc(size_t anzahl, size_t groesse);`
allokiert einen Speicherbereich, der groß genug ist, um `anzahl` Objekte von `groesse` Bytes aufzunehmen. Alle Bytes in diesem Speicherbereich werden mit dem Wert 0 initialisiert.

`void *realloc(void *zeiger, size_t groesse);`
verändert die Größe des Speicherbereichs, auf den `zeiger` zeigt, nach `groesse`. Der Inhalt dieses neuen Objekts bleibt unverändert bis zur kleineren der alten oder neuen Größe.
`realloc(NULL, groesse)` ist identisch zu **`malloc(groesse)`**.

`void free(void *zeiger);`
bewirkt die Freigabe des Speicherbereichs, auf den `zeiger` zeigt.

3.13.2 Environment-Variablen

`char *getenv(const char *name);`
durchsucht die "Environment-Tabelle" des entsprechenden Betriebssystems nach einer Environment-Variable `name` und liefert den Inhalt dieser Environment-Variable als Rückgabewert.

3.13.3 Programmbeendigung

`int atexit(void (*func) (void));`
trägt die Funktion, auf die `func` zeigt, in die Liste von Funktionen ein, die vor einer normalen Beendigung des Programms noch aufzurufen sind.

`void exit(int status);`
bewirkt eine "normale Programmbeendigung".

`void abort(void);`
bewirkt einen anormalen Programmabbruch.

`int system(const char *string);`
übergibt das Kommando `string` an das entsprechende Betriebssystem, damit dieses vom zugehörigen Kommandoprozessor interpretiert und ausgeführt wird.

3.13.4 Zufallszahlen

`int rand (void);`
liefert eine Pseudo-Zufallszahl aus dem Bereich von 0 bis `RAND_MAX` (>= 32767).

`void srand(unsigned int startwert);`
legt den Startpunkt für eine Folge von Pseudo-Zufallszahlen fest.

3.13.5 Absolutwerte

```
long int labs(long int j);
```

```
int abs(int j);
```

liefern den Absolutwert zum ganzzahligen Argument `j`.

3.13.6 Binäre Suche und Quicksort

```
void *bsearch(const void *such_zeig,
              const void *start_addr,
              size_t anzahl,
              size_t groesse,
              int (*vergleichs_routine)(const void*,const void*));
```

realisiert die binäre Suche.

```
void qsort(void *array,
           size_t anzahl,
           size_t groesse,
           int (*vergl_funktion)(const void *, const void *));
```

realisiert den Quicksort von Hoare.

3.13.7 Quotient und Rest einer Division

```
div_t div(int zaehler, int nenner);
```

```
ldiv_t ldiv(long int zaehler, long int nenner);
```

Diese Funktionen berechnen den Quotienten und den Rest der Division `zaehler/nenner`. Ist die Division ungenau, ergibt sich als Quotient der Betrag der Ganzzahl, der kleiner als der Betrag des mathematischen Quotienten ist. Der Rückgabetyp `div_t` bzw. `ldiv_t` ist eine Struktur, die folgende Komponente enthält:

```
int quot; /* Quotient (long int bei ldiv_t) */
int rem;  /* Rest     (long int bei ldiv_t) */
```

Kann das Ergebnis nicht dargestellt werden, liegt ein undefiniertes Verhalten vor, ansonsten muß folgendes gelten:

```
quot * nenner + rest = zaehler
             20 div  7 =  2 Rest  6
            -20 div  7 = -2 Rest -6
             20 div -7 = -2 Rest  6
            -20 div -7 =  2 Rest -6
```

3.13.8 Konvertierung von Strings in numerische Werte

```
double atof(const char *string);
```

wandelt eine Zahl, die als `string` gespeichert ist, in einen **double**-Wert um, den sie als Funktionswert liefert. Außer dem Verhalten im Fehlerfall ist *atof* äquivalent zu **`strtod(string, (char **)NULL)`**.

`int` **`atoi`**`(const char *string);`
wandelt eine Zahl, die als `string` gespeichert ist, in einen **int**-Wert um, den sie als Funktionswert liefert. Außer dem Verhalten im Fehlerfall ist *atoi* äquivalent zu **`(int)strtol(string, (char **)NULL, 10).`**

`long int` **`atol`**`(const char *string);`
wandelt eine Zahl, die als `string` gespeichert ist, in einen **long int**-Wert um, den sie als Funktionswert liefert. Außer dem Verhalten im Fehlerfall ist *atol* äquivalent zu **`strtol(string, (char**)NULL, 10).`**

`double` **`strtod`**`(const char *string, char **end_zeig);`
(***string to double***) wandelt eine Zahl, die als `string` gespeichert ist, in einen **double**-Wert um und liefert diesen als Funktionswert. Falls für `end_zeig` kein `NULL`-Zeiger übergeben wurde, wird nach einer erfolgreichen Umwandlung die Adresse eines nicht konvertierbaren Rests im Zeiger abgelegt, auf den `end_zeig` zeigt.
Bei einer erfolgreichen Umwandlung liefert *strtod* die durch Umwandlung erhaltene Gleitpunktzahl, andernfalls den Wert 0.

`long` **`strtol`**`(const char *strizng, char **end_zeig, int basis);`
`unsigned long` **`strtoul`**`(const char *string, char **end_zeig, int basis);`
strtol (***str**ing **to l**ong*) und *strtoul* (***str**ing **to u**nsigned **l**ong*) wandeln eine Zahl, die als `string` gespeichert ist, in einen **long**- bzw. **unsigned-long**-Wert um und liefern diesen als Funktionswert. `basis` legt dabei die Basis des Zahlensystems fest, in das diese Zahl umzuwandeln ist. Falls für `end_zeig` kein Nullzeiger übergeben wurde, wird nach einer erfolgreichen Umwandlung die Adresse eines nicht konvertierbaren Rests im Zeiger abgelegt, auf den `end_zeig` zeigt. Bei einer erfolgreichen Umwandlung liefern *strtol* bzw. *strtoul* die durch die Umwandlung erhaltene ganze Zahl, andernfalls den Wert 0.

Folgender Codeausschnitt demonstriert an *strtod* die Verwendung der drei Funktionen *strtod*, *strtol*, *strtoul*.

```
double   zahl;
char     string[100], *rest, zeichk[100];

    rest = zeichk;
    zahl = strtod (string, &rest);
    if (string == rest)
       printf ("%s ist keine erlaubte Gleitpunktzahl\n", string);
    printf ("%lg (Gleitpunktzahl) / %s (Rest)\n", zahl, rest);
```

3.13.9 Vielbyte-Zeichen

`int` **`mblen`**`(const char *vb_zeig, size_t n);`
liefert die Bytezahl, aus denen sich das Vielbyte-Zeichen zusammensetzt, auf das `vb_zeig` zeigt; äquivalent zu **`mbtowc( (wchar_t *)0, vb_zeig, n) );`**

`int` **`mbtowc`**`(wchar_t *pwc, const char *vb_zeig, size_t n);`
konvertiert ein Vielbyte-Zeichen nach `wchar_t`.

`int` **`wctomb`**`(char *vb_zeig, wchar_t wchar);`
konvertiert `wchar_t`-Zeichen in Vielbyte-Zeichen.

`size_t` **`mbstowcs`**`(wchar_t *pwcs, const char *vb_zeig, size_t n);`
konvertiert mit *mbtowc* eine Folge von Vielbyte-Zeichen aus dem Speicherplatz `vb_zeig` in Datentyp `wchar_t` und speichert die entsprechenden Codes (nicht mehr als `n`) an Adresse `pwcs`.

`size_t` **`wcstombs`**`(char *vb_zeig, const wchar_t *pwcs, size_t n);`
konvertiert mit *wctomb* eine Folge von Codes aus dem Speicherplatz `pwcs` in eine Folge von entsprechenden Vielbyte-Zeichen (nicht mehr als `n`) und schreibt diese an die Adresse `vb_zeig`.

3.14 <string.h> – Umgang mit Zeichenketten

Diese Headerdatei definiert ein weiteres Mal den bereits in `<stddef.h>` definierten Datentyp `size_t` und die ebenfalls dort definierte `NULL`-Zeigerkonstante.

`void *`**`memchr`**`(const void *adress, int such_zeich, size_t n);`
sucht das erste Vorkommen von *such_zeich* in den ersten *n* Zeichen des Speicherbereichs, auf den *adress* zeigt, und gibt die Adresse des gefundenen Zeichens oder `NULL`-Zeiger zurück, falls *such_zeich* nicht gefunden werden konnte.

`int` **`memcmp`**`(const void *adress1, const void *adress2, size_t n);`
vergleicht die ersten *n* Zeichen des Speicherbereichs, auf den *adress1* zeigt, mit den ersten *n* Zeichen des Speicherbereichs, auf den *adress2* zeigt. Diese Funktion liefert als Funktionswert eine negative Zahl (wenn Bytekette von *adress1* < Bytekette von *adress2*), 0 (wenn Bytekette von *adress1* == Bytekette von *adress2*) oder eine positive Zahl (wenn Bytekette von *adress1* > Bytekette von *adress2).*
Der Rückgabewert ist die Differenz der beiden ersten verschiedenen Zeichen in den Speicherbereichen *adress1* und *adress2.*

`void *`**`memcpy`**`(void *ziel, const void *quelle, size_t n);`
kopiert *n* Zeichen von dem Speicherplatz, auf den *quelle* zeigt, in den Speicherbereich, auf den *ziel* zeigt. Falls die beiden *n*-byte langen Speicherbereiche sich überlappen, ist das Verhalten undefiniert (siehe auch *memmove*). *memcpy* liefert die Adresse *ziel* als Funktionswert.

`void *`**`memmove`**`(void *ziel, const void *quelle, size_t n);`
kopiert `n` Zeichen von dem Speicherplatz, auf den *quelle* zeigt, in den Speicherbereich, auf den *ziel* zeigt. Im Gegensatz zu *memcpy* garantiert diese Funktion bei einer Überlappung der beiden Speicherbereiche einen korrekten Kopiervorgang. *memmove* liefert die Adresse *ziel* als Funktionswert.

`void *memset(void *adress, int zeich, size_t n);`
schreibt den Wert von *zeich* in jedes der ersten *n* Zeichen des Speicherbereichs mit der Adresse *adress*. *memset* liefert die Adresse *adress* als Funktionswert. Beispielaufrufe sind:

```
memset(striche, '-', 100);
memset(zeich_array, ' ', 2000);
memset(int_array, 20, 100*sizeof(int));
```

`char *strcat(char *kett1, const char *kett2);`
kopiert String *kett2* (einschließlich `\0`) ans Ende des Strings *kett1*, wobei das erste Zeichen von *kett2* das abschließende `\0` von *kett1* überschreibt. Falls die beiden Strings *kett1* und *kett2* sich überlappen, ist das Verhalten undefiniert. *strcat* liefert den Zeiger *kett1* auf den Anfang des gesamten Strings.

`char *strchr(const char *kett, int such_zeich);`
sucht das erste Vorkommen von *such_zeich* im String *kett*. Das abschließende `\0` wird als Teil des Strings angesehen. *strchr* liefert die Adresse des gefundenen Zeichens oder `NULL`, falls *such_zeich* nicht im String *kett* vorkommt.

`int strcmp(const char *kett1, const char *kett2);`
vergleicht die beiden Strings *kett1* und *kett2* byteweise und liefert einen positiven Wert (wenn *kett1* > *kett2*), einen negativen Wert (wenn *kett1* < *kett2*) oder 0 (wenn *kett1* und *kett2* völlig gleich sind).
Der Rückgabewert ist die Differenz der beiden ersten nicht übereinstimmenden Zeichen in *kett1* und *kett2*.

`int strcoll(const char *kett1, const char *kett2);`
verhält sich wie *strcmp*, außer daß lokal-spezifische Vergleichsregeln (durch die *categorie* `LC_COLLATE` in der *setlocale*-Funktion festgelegt) angewendet werden.

`int strcpy(char *ziel, const char *quelle);`
kopiert den String *quelle* (einschließlich `\0`) in den Speicherbereich, auf den *ziel* zeigt. Falls dieser Kopiervorgang auf Objekte angewendet wird, die sich überlappen, ist das Verhalten undefiniert. *strcpy* liefert *ziel* als Funktionswert.

`int strcspn(const char *kett1, const char *kett2);`
liefert die Länge des Teilstrings in *kett1* (von Anfang an), der keine Zeichen aus `kett2` enthält.

`char *strerror(int fehler_nr);`
liefert die Adresse der zu *fehler_nr* gehörigen Fehlermeldung.

`size_t strlen(const char *zeichk);`
liefert die Länge des Strings *zeichk* (ohne abschließendes `\0`).

`char *strncat(char *kett1, const char *kett2, size_t n);`
kopiert von *kett2* nicht mehr als *n* Zeichen ans Ende des Strings *kett1*. Das abschließende `\0` wird immer an das Ende des zusammengehängten Strings geschrieben. Somit ist die Länge für den neu entstandenen String:

```
if (strlen(kett2) > n)
   strlen(kett1)+n+1      /* + 1 für abschließendes \0 */
else
   strlen(kett1)+strlen(kett2)+1   /* + 1 für abschließendes \0 */
```

strncat liefert den Zeiger *kett1* auf den Anfang des gesamten zusammengehängten Strings.
Falls sich die beiden Strings *kett1* und *kett2* überlappen, dann liegt ein undefiniertes Verhalten vor.

`int strncmp(const char *kett1, const char *kett2, size_t n);`
vergleicht bis zu `n` Zeichen der Strings *kett1* und *kett2* byteweise und liefert als Rückgabe einen positiven Wert (wenn *kett1>kett2)*, einen negativen Wert (wenn *kett1<kett2*) oder 0 (wenn *kett1* und *kett2* gleich sind).
Beachten Sie, daß nur bis zu *n* Zeichen in den beiden Strings verglichen werden. Der Funktionswert ergibt sich aus der Differenz der beiden ersten nicht übereinstimmenden Zeichen in *kett1* und *kett2*.

`char *strncpy(char *kett1, const char *kett2, size_t n);`
kopiert nicht mehr als *n* Zeichen aus *kett2* in den String *kett1*. Falls sich überlappende Strings kopiert werden, ist das Verhalten undefiniert. Wenn die Länge von *kett2* kleiner als *n* Zeichen ist, dann wird im String *kett1* für die fehlenden Zeichen `\0` angehängt. *strcpy* liefert den Zeiger *kett1* als Rückgabewert.

VORSICHT: Wenn String *kett2* länger als *n* Zeichen ist, wird kein `\0` angehängt.

`char *strpbrk(const char *kett1, const char *kett2);`
sucht in *kett1* das erste Vorkommen eines Zeichens aus *kett2* und liefert dann entweder die Adresse des gefundenen Zeichens oder `NULL`, falls kein Zeichen aus *kett2* in *kett1* vorkommt.

`char *strrchr(const char *zeichk, int zeich);`
sucht in *zeichk* das letzte Vorkommen von *zeich*. Das abschließende `\0` wird als Bestandteil von *zeichk* betrachtet. *strrchr* liefert die Adresse des gefundenen Zeichens oder einen `NULL`-Zeiger, falls *zeich* nicht gefunden werden kann.

`size_t strspn(const char *kett1, const char *kett2);`
liefert die Länge des Teilstrings in *kett1* (von Anfang an), der nur aus Zeichen von *kett2* besteht.

`char *strstr(const char *kett1, const char *kett2);`

sucht in *kett1* das erste Vorkommen des Strings *kett2* (ohne abschließendes \0). *strstr* liefert entweder einen Zeiger auf den gefundenen String oder einen NULL-Zeiger, falls *kett2* kein Teilstring von *kett1* ist. Wenn *kett2* ein String der Länge 0 ist, so liefert diese Funktion *kett1* zurück.

`char *strtok(char *kett1, const char *kett2);`

Eine Folge von *strtok*-Aufrufen bricht den String *kett1* in eine Folge von Teilstrings (Token), wobei die "Bruchstellen" durch *kett2* festgelegt werden. Der erste Aufruf von *strtok* bewirkt, daß in *kett1* das erste Zeichen gesucht wird, das nicht als Trennzeichen in *kett2* vorkommt. Wird kein solches Zeichen gefunden, gibt *strtok* NULL zurück. Wird so ein Zeichen gefunden, ist dies der Anfang des ersten Teilstrings. Von nun an sucht *strtok* nach einem Trennzeichen: Kann keines gefunden werden, erstreckt sich der Teilstring bis zum Ende von *kett1*, und nachfolgende Aufrufe von *strtok* werden fehlschlagen. Wird ein solches Trennzeichen gefunden, wird es mit \0 überschrieben; so wird das Ende des Teilstrings festgelegt. *strtok* merkt sich den Zeiger auf das nächste Zeichen, von wo aus bei dem Aufruf *strtok(NULL,...);* die nächste Suche nach einem Teilstring beginnt. *strtok* gibt einen Zeiger auf das erste Vorkommen eines Teilstrings zurück oder einen NULL-Zeiger, falls keiner gefunden werden kann. Die Trennzeichen, die mit *kett2* angegeben werden, können bei jedem Aufruf verschieden sein.

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

char trennzeich[]=",;:";

int main (void)
{
    char  zeile[100], *einzel_name;
    int   i=0;

    printf("Gib die Liste der Namen (mit , oder ; oder : getrennt ein\n");
    gets(zeile);
    einzel_name = strtok(zeile, trennzeich);
    while (einzel_name != NULL) {
       printf("Name %d : %s\n", ++i, einzel_name);
       einzel_name = strtok(NULL, trennzeich);
    }
    return(0);
}
```

```
Gib die Liste der Namen (mit , oder ; oder : getrennt ein
Meier    Franz;;;,;;;Wasser-Fritz:Feuer Emil;Danne Doris-Annette:::::↵
Name 1 : Meier    Franz
Name 2 : Wasser-Fritz
Name 3 : Feuer Emil
Name 4 : Danne Doris-Annette
```

`size_t` **`strxfrm`**`(char *`*`nach`*`, const char *`*`von`*`, size_t` *`max_groesse`*`);`

wandelt einen lokal-spezifischen String *von* in eine "C-normale" Form (englisch-amerikanisch) um und speichert den umgewandelten String an Adresse *nach*. *strcmp* – angewandt auf zwei so umgewandelte Strings – liefert das gleiche Ergebnis wie *strcoll* auf die zwei Originalstrings. Nach *nach* werden nie mehr als *max_groesse* Zeichen (`\0` mitgerechnet) geschrieben. Überlappen sich beide Strings, ist das Verhalten undefiniert. Ist für *max_groesse* der Wert `0` angegeben, darf *nach* ein `NULL`-Zeiger sein. *strxfrm* liefert die Länge des umgewandelten Strings (ohne `\0`). Falls sie Wert >= *max_groesse* liefert, ist der Speicherinhalt von *nach* unbestimmt.

Neben obigen Funktionen dürfen noch weitere in `<string.h>` definiert sein, wenn ihr Name mit

str*k* oder **mem***k* (*k* steht für Kleinbuchstabe) oder **wcs***k* (*k* steht für Kleinbuchstabe) beginnt.

3.15 <stdio.h> – Standard-E/A-Funktionen

3.15.1 Öffnen einer Datei

FILE ***fopen**(const char *pfadname*, const char *modus*);
gibt zurück: `FILE`-Zeiger (bei Erfolg); `NULL` bei Fehler

Mit dem Argument `modus` wird die Zugriffsart für die Datei `pfadname` festgelegt. Tabelle 3-11 zeigt mögliche Angaben für `modus` bei *fopen* und bei *freopen*.

modus-Argument	Bedeutung
"r" oder "rb"	(read) zum Lesen öffnen
"w" oder "wb"	(write) zum Schreiben öffnen (neu anlegen oder Inhalt einer existierenden Datei löschen)
"a" oder "ab"	(append) zum Schreiben am Dateiende öffnen; nicht existierende Datei wird angelegt
"r+", "r+b" oder "rb+"	zum Lesen und Schreiben öffnen
"w+", "w+b" oder "wb+"	zum Lesen und Schreiben öffnen; Inhalt einer existierenden Datei wird gelöscht
"a+", "a+b" oder "ab+"	zum Lesen und Schreiben ab Dateiende öffnen

Tabelle 3-11: Mögliche modus-Angaben bei fopen und freopen

Tabelle 3-12 zeigt die Einschränkungen/Auswirkungen für die einzelnen Öffnungsmodi.

Einschränkung bzw. Auswirkung	r	w	a	r+	w+	a+
Datei muß zuvor existieren.	x			x		
Alter Dateiinhalt geht verloren.		x			x	
Aus Datei kann gelesen werden.	x			x	x	x
In Datei kann geschrieben werden.		x	x	x	x	x
Nur am Dateiende kann geschrieben werden.			x			x

Tabelle 3-12: Einschränkungen bzw. Auswirkungen für die einzelnen Öffnungsmodi

- Ein sofortiges Lesen nach Schreiben ohne *fflush*, *fseek*, *fsetpos* oder *rewind* ist nicht möglich.
- Ein sofortiges Schreiben nach Lesen ohne *fseek*, *fsetpos* oder *rewind* ist nicht möglich, außer bei Dateiende.
- Die Fehler- und `EOF`-Flags werden beim Öffnen einer Datei zurückgesetzt.

3.15.2 Öffnen einer Datei mit bereits existierenden Stream

FILE ***freopen**(const char **pfadname*, const char **modus*, FILE **fz*);
gibt zurück: `FILE`-Zeiger (bei Erfolg); `NULL` bei Fehler

freopen versucht, zuerst die entsprechende Datei, die mit `fz` verbunden ist, zu schließen. Mögliche Fehler beim Schließversuch werden ignoriert. Danach ordnet diese Funktion den `FILE`-Zeiger `fz` der Datei *pfadname* zu.

```
if (freopen("dateiname", "a", stdout) != stdout) /* stdout in dateiname umlenken */
   fehler_meld(....);
```

3.15.3 Schließen einer Datei

int **fclose**(FILE **fz*);
gibt zurück: 0 (bei Erfolg); `EOF` bei Fehler

Bevor *fclose* die Verbindung zwischen der Datei und dem `FILE`-Zeiger `fz` auflöst, überträgt diese Funktion die Inhalte aller noch nicht geleerten Ausgabepuffer in die entsprechende Datei. Die Inhalte von Eingabepuffern gehen verloren.

3.15.4 EOF- und Fehler-Flags

int **feof**(FILE **fz*);
gibt zurück: Wert verschieden von 0, wenn `EOF`-Flag für Datei *fz* gesetzt ist; 0 sonst

int **ferror**(FILE **fz*);
gibt zurück: Wert verschieden von 0, wenn Fehler-Flag für Datei *fz* gesetzt ist; 0 sonst

3.15.5 Löschen des Fehler- und EOF-Flags

void **clearerr**(FILE *fz);

3.15.6 Lesen eines Zeichens von stdin

int **getchar**(void);
gibt zurück: nächstes Zeichen aus `stdin` (bei Erfolg); `EOF` bei Dateiende oder Fehler

`getchar()` ist äquivalent zu dem Aufruf `getc(stdin)`.

3.15.7 Schreiben eines Zeichen auf stdout

int **putchar**(int *zeich*) ;
gibt zurück: `zeich` (bei Erfolg); `EOF` bei Fehler

`putchar(zeich)` ist äquivalent zu dem Aufruf `putc(zeich, stdout)`.

3.15.8 Lesen eines Zeichens aus einer Datei

int **getc**(FILE *fz);
int **fgetc**(FILE *fz);
beide geben zurück: nächstes Zeichen aus Datei *fz* (bei Erfolg); `EOF` bei Dateiende oder Fehler

3.15.9 Schreiben eines Zeichens in eine Datei

int **putc**(int *zeich*, FILE *fz);
int **fputc**(int *zeich*, FILE *fz);
beide geben zurück: `zeich` (bei Erfolg); `EOF` bei Fehler

3.15.10 Zurückschieben eines gelesenen Zeichens in den Eingabepuffer

int **ungetc**(int *zeich*, FILE *fz);
gibt zurück: `zeich` (bei Erfolg); `EOF` bei Fehler

```
while ( (zeich=fgetc(fz)) != EOF) /* Lesen einer Hexazahl aus einer Textdatei */
   if (isxdigit(zeich)) {
      ungetc(zeich, fz);
      fscanf(fz, "%lx", &hexzahl);
      printf("%lx=%lu\n", hexzahl, hexzahl);
   }
```

Ein solcher *Lookahead* ist eine typische Anwendung für *ungetc*.

3.15.11 Lesen einer ganzen Zeile von stdin

char *__gets__(char *_puffer_);
gibt zurück: Adresse `puffer` (bei Erfolg); `NULL` bei Dateiende oder Fehler

Da bei *gets* der Aufrufer anders als bei *fgets* keine Möglichkeit hat, die Größe des Puffers zu wählen, kann es zum Überlaufen des von *gets* gewählten Puffers kommen, wenn eine gelesene Zeile mehr Zeichen als die intern gewählte Pufferlänge hat. Wenn möglich, sollte also immer *fgets* anstelle von *gets* benutzt werden.

3.15.12 Lesen einer ganzen Zeile aus einer Datei

char *__fgets__(char *_puffer_, int _n_, FILE *_fz_);
gibt zurück: Adresse `puffer` (bei Erfolg); `NULL` bei Dateiende oder Fehler

fgets liest aus dem Stream `fz` entweder `n-1` Zeichen oder bis zum nächsten Neuezeilezeichen (`\n`) – je nachdem, was zuerst eintritt – und speichert gelesene Zeichen an der Adresse `puffer` ab, die mit `\0` abgeschlossen werden.

fgets unterscheidet sich von *gets* darin, daß es nicht nur von stdin lesen kann, sondern auch automatisch `\n` am Ende des Strings anhängt, wenn die Länge des gelesenen Strings $\leq n$ ist.

3.15.13 Schreiben einer ganzen Zeile auf stdout

int __puts__(const char *_puffer_);
gibt zurück: nichtnegativen Wert (bei Erfolg); `EOF` bei Fehler

puts gibt immer automatisch am Ende der ausgegebenen Zeichenkette noch ein `\n` aus, was *fputs* nicht tut.

3.15.14 Schreiben einer ganzen Zeile in eine Datei

int __fputs__(const char *_puffer_, FILE *_fz_);
gibt zurück: nichtnegativen Wert (bei Erfolg); `EOF` bei Fehler

3.15.15 Binäres Lesen und Schreiben ganzer Blöcke

size_t __fread__(void *_puffer_, size_t _blockgroesse_, size_t _blockzahl_, FILE *_fz_);
gibt zurück: Anzahl der gelesenen bzw. geschriebenen Blöcke

size_t __fwrite__(const void *_puffer_, size_t _blockgroesse_, size_t _blockzahl_, FILE *_fz_);
gibt zurück: Anzahl der gelesenen bzw. geschriebenen Blöcke

fread liest bis zu `blockzahl` Objekte, jedes mit `blockgroesse` Bytes, vom Stream `fz` an die Adresse `puffer`.