FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Ausbildung von
Mathematisch-Technischen Assistenten/innen

Programmierung in C
Vorlesungsskript

Günter Egerer

FZJ-ZAM-BHB-0140

1. Auflage

(letzte Änderung: 05.11.96)

Copyright-Notiz

© Copyright 1999 by Forschungszentrum Jülich GmbH,
Zentralinstitut für Angewandte Mathematik (ZAM). Alle Rechte vorbehalten.
Kein Teil dieses Werkes darf in irgendeiner Form ohne schriftliche Genehmigung des ZAM reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wie, wo und wann findet man Beratung und Dispatch?

Beratung und Dispatch finden Sie im Erdgeschoß des Zentralinstituts für Angewandte Mathematik.

Öffnungszeiten:	Montag bis Freitag	8.15 - 17.00 Uhr
Telefonische Beratung:	Montag bis Freitag	8.15 - 18.45 Uhr

Publikationen des ZAM stehen im PostScript-Format auf dem WWW-Server des Forschungszentrums unter der URL: <http://www.fz-juelich.de/zam/docs/printable/> zur Verfügung. Eine Übersicht über alle Publikationen des ZAM erhalten Sie unter der URL: `<http://www.fz-juelich.de/zam/docs>` .

Wichtige Telefonnummern und Electronic-Mail-Adressen im ZAM

Die Electronic-Mail-Adressen der verschiedenen Beratungsdienste lauten:
***thema*.zam@fz-juelich.de**

Fachgebiet	**Berater**	**Telefon**	***thema***
Beratung und Betrieb	O. Büchner u.a.	**6400**	***beratung***
Dispatch	D. Aderhold, Ch. Dohmen, St. Meier	5642	*dispatch*
Bestellung von Dokumentation		5642	*literatur*
KFAnet/Internet	R. Niederberger	4772	*kfanet*
PCs im KFAnet	R. Grallert	6421	*kfanet-pc*
WiN und Mail	M. Sczimarowsky	6411	*win*
ISDN- und Modem-Zugang	Dr. L. Radermacher	6587	*isdn*
WWW	Dr. S. Höfler-Thierfeldt, M. Baetzen	6765	*www*
X-Terminals	O. Mextorf	2519	*x-terminals*
Backup, Archivierung	U. Schmidt	6577	*backup*
Supercomputer	Dr. N. Attig	4416	*sc*
Fortran	G. Groten	6589	*fortran*
C und C++	G. Egerer	2339	*c*
Mathematik	Dr. B. Steffen	6431	*mathematik*
Statistik	Dr. W. Meyer	6414	*statistik*
Mathematische Software	I. Gutheil R. Zimmermann	3135 4136	*mathsoft*
Graphik	Ma. Busch, M. Boltes	4100	*graphik*
Textverarbeitung, Druckerunterstützung	St. Graf, Her. Schumacher	6578	*text, drucker*
Zentrale Datenbank	W. Elmenhorst, B. v. Studnitz	6762	*oracle*

	Telefon
Telefax-Nr. des Dispatch und der Beratung	Fax 2810
Rufbereitschaft zentrale Rechnersysteme	6400
Rufbereitschaft Netzwerke	6440
Geräteservice: Workstations, PCs, X-Terminals,	6555
dezentrale Drucker und Plotter	2435
Telefax-Nr. für PC-Wartung	Fax 6424
Druckausgabe bei BD-SG	2167

Die jeweils aktuelle Version dieser Tabellen finden Sie unter der URL:
`<http://www.fz-juelich.de/zam/zam-general/contact.shtml>`

Inhalt

1 Literatur . . . 1
2 Einleitung . . . 2
2.1 Historie von C . . . 2
2.2 Eigenschaften von C . . . 3
2.3 Beispielprogramm: Ausgabe von Zahlen (Version 1) . . . 4
2.4 Ausgabe von Zahlen (Version 2) . . . 11
2.5 Beispielprogramm 2: Kopieren der Eingabe zur Ausgabe (Version 1) . . . 12
2.6 Kopieren der Eingabe zur Ausgabe (Version 2) . . . 13
2.7 Übersetzen, Binden und Ausführen . . . 14
2.8 Aufgaben . . . 17
3 Grundelemente der Sprache . . . 19
3.1 Zeichensatz . . . 19
3.2 Alternativdarstellungen (Trigraph-Sequenzen) . . . 21
3.3 Kommentar . . . 22
3.4 Grundsymbole (Token) . . . 24
3.4.1 Schlüsselwörter . . . 25
3.4.2 Bezeichner . . . 26
3.4.3 Operatoren . . . 28
3.4.4 Punktsymbole . . . 29
3.4.5 Konstanten . . . 30
3.4.5.1 Integer-Konstanten . . . 31
3.4.5.2 Gleitkommakonstanten . . . 35
3.4.5.3 Character-Konstanten (Zeichenkonstanten) . . . 37
3.4.5.4 Aufzählungskonstanten (enumeration constants) . . . 39
3.4.6 Strings (string literals) . . . 40
3.4.7 Character- und Stringkonstanten für große Zeichensätze . . . 41
3.5 Regeln für das Einfügen von Zwischenraumzeichen . . . 42
3.6 Fortsetzungszeilen . . . 43
3.7 Aufgaben . . . 44

4 Basistypen, Operatoren und Ausdrücke . . . 46
4.1 Basistypen (Übersicht) . . . 46
4.2 sizeof-Operator . . . 47
4.3 Integer-Typen . . . 49
4.3.1 Funktionen in <ctype.h> . . . 54
4.3.2 Logische Werte . . . 56
4.3.3 Aufzählungstyp (enumerated type) . . . 57
4.3.4 Integer-Erweiterung (integral promotion) . . . 59
4.3.5 Formatierte Ausgabe von Integer-Typen . . . 61
4.3.6 Formatierte Eingabe von Integer-Typen . . . 68
4.3.7 Aufgaben . . . 74
4.4 Gleitkommatypen . . . 76
4.4.1 Formatierte Ausgabe von Gleitkommatypen . . . 79
4.4.2 Formatierte Eingabe von Gleitkommatypen . . . 82
4.4.3 Aufgabe . . . 83
4.5 Zuweisungsoperator . . . 84
4.6 cast-Operator (explizite Typumwandlung) . . . 85
4.7 Typumwandlungen . . . 86
4.8 Arithmetische Umwandlungen . . . 88
4.9 Arithmetische Operatoren . . . 91
4.10 Inkrement- und Dekrement-Operatoren . . . 94
4.11 Operatoren für Bitmanipulationen . . . 96
4.12 Vergleichsoperatoren . . . 101
4.13 Logische Operatoren . . . 102
4.14 Weitere Zuweisungsoperatoren . . . 104
4.15 Bedingungsoperator (conditional operator) . . . 106
4.16 Komma-Operator . . . 108
4.17 Analyse von Ausdrücken . . . 109
4.18 Bewertung von Ausdrücken . . . 112
4.19 Konstante Ausdrücke mit integralem Typ . . . 115
4.20 Aufgaben . . . 116
4.21 Mathematische Funktionen . . . 118
4.22 Behandlung von Fehlern durch Bibliotheksfunktionen . . . 123

5 Anweisungen (statements) . . . 126
5.1 Übersicht . . . 126
5.2 leere Anweisung . . . 127
5.3 Ausdrucksanweisung . . . 127
5.4 gelabelte Anweisungen . . . 128
5.5 Verbundanweisung . . . 128
5.6 bedingte Anweisungen . . . 129
5.7 Wiederholungsanweisungen . . . 133
5.8 Sprunganweisungen . . . 136
5.9 Aufgabe . . . 140
6 Funktionen und Programmstruktur . . . 141
6.1 Vereinbarungen . . . 141
6.2 Funktionsdeklaration . . . 142
6.3 Funktionsdefinition . . . 144
6.4 Funktionsaufruf . . . 147
6.5 top-level-Vereinbarungen . . . 149
6.6 Gültigkeitsbereich innerhalb der Quelldatei (lexical scope) . . . 150
6.7 Namensräume (name spaces) . . . 154
6.8 Speicherklassen . . . 155
6.9 Definition und Deklaration von top-level-Variablen . . . 158
6.10 Bindung . . . 160
6.11 Header-Dateien . . . 164
6.12 Aufgabe . . . 167
7 Präprozessor . . . 170
7.1 Präprozessor-Direktiven . . . 170
7.2 Einfügen von Dateien (#include-Direktive) . . . 171
7.3 Makrodefinition und Expansion . . . 173
7.4 Probleme bei der Verwendung von Makros . . . 182
7.5 Löschen von Makrodefinitionen . . . 192
7.6 Bedingte Übersetzung . . . 193
7.7 Aufgaben . . . 196
7.8 Bedingtes Einfügen von Header-Dateien . . . 198
7.9 Sonstige Direktiven . . . 199
7.10 Vordefinierte Makronamen . . . 201

8 Zeiger und Vektoren . . . 203
8.1 Adreßoperator . . . 203
8.2 Verweis-, Inhalts-, Dereferenzierungsoperator . . . 205
8.3 Generische Zeiger . . . 208
8.4 Nullzeiger . . . 209
8.5 Formatierte Ausgabe von Zeigerwerten . . . 210
8.6 Typzusätze (Attribute für Typen) . . . 211
8.7 Dynamische Speicherverwaltung . . . 214
8.8 Adreß-Arithmetik (Arithmetik mit Zeigern) . . . 217
8.9 Beziehung zwischen Zeigern und Vektoren . . . 220
8.10 Initialisierung . . . 225
8.11 Vektoren von Zeigern, Zeiger auf Zeiger . . . 229
8.11.1 Argumente aus der Kommandozeile . . . 230
8.12 Ein-/Ausgabe von Zeichen und Zeichenketten . . . 234
8.12.1 Funktionen zur Ausgabe einzelner Zeichen . . . 234
8.12.2 Funktionen zur Eingabe einzelner Zeichen . . . 236
8.12.3 Funktionen zur Ausgabe von Zeichenketten . . . 238
8.12.4 Funktionen zur Eingabe von Zeichenketten . . . 239
8.12.5 Formatierte Ausgabe von Zeichenketten . . . 240
8.12.6 Formatierte Eingabe von Zeichenketten . . . 241
8.12.7 Funktionen zur formatierten Ein-/Ausgabe . . . 244
8.13 Umwandlungsfunktionen . . . 245
8.14 Funktionen in <string.h> . . . 248
8.14.1 Kopierfunktionen . . . 250
8.14.2 Verkettungsfunktionen . . . 254
8.14.3 Vergleichsfunktionen . . . 256
8.14.4 Suchfunktionen . . . 259
8.14.5 Sonstige Stringfunktionen . . . 268
8.14.6 Beispielprogramm: Verketten von 2 Strings . . . 269
8.14.7 Beispielprogramm: MS-DOS-Dateiname . . . 271
8.14.8 Aufgaben . . . 273
8.15 Mehrdimensionale Vektoren (Felder, Arrays) . . . 274
8.15.1 Initialisierung mehrdimensionaler Vektoren . . . 279
8.15.2 Aufgaben . . . 281
8.16 Zeiger auf Funktionen . . . 282
8.16.1 Aufgaben . . . 288

9 Strukturen 289
9.1 Strukturvereinbarungen 289
9.2 Zulässige Operationen 292
9.3 Unvollständiger Strukturtyp 293
9.4 Speicherabbild einer Struktur 295
9.5 Zeiger auf Strukturen 298
9.6 Vektoren von Strukturen 299
9.7 Rekursive Strukturen 299
10 typedef-Vereinbarung 301
10.1 Aufgabe 303
11 Union (Vereinigung) 304
11.1 Aufgabe 306
12 Bitfelder 307
13 Variable Argumentlisten 310
13.1 Makros in <stdarg.h> 310
13.2 Spezielle Ausgabefunktionen: vprintf, vfprintf und vsprintf 314
14 Funktionen zur Erzeugung von Pseudo-Zufallszahlen 319
15 Funktionen zum Sortieren und Suchen 320
16 Fehlersuche 322
17 Funktionen zur Beendigung eines Programmes 323
18 Schnittstelle zur Betriebssystemumgebung 325
19 Ein-/Ausgabefunktionen in <stdio.h> 328
19.1 Eröffnen einer Datei 328
19.2 Abschließen einer Datei 331
19.3 Abfragen und Setzen von Fehlerbedingungen 332
19.4 Abfragen und Setzen der Dateiposition 333
19.5 unformatierte Ein-/Ausgabe 339
19.6 sonstige Dateioperationen 346
20 Signalbehandlung 352
20.1 Signale 352
20.2 Funktionen in <signal.h> 355
21 Funktionen in <setjmp.h> 359
21.1 Nicht-lokale Sprünge 359
22 Zeitfunktionen in <time.h> 365
22.1 Datentypen zur Darstellung von Zeiten 365
22.2 Beschreibung der Funktionen 366

A C-Programmierwerkzeuge 369
A.1 make 375
A.1.1 Dependency lines 377
A.1.2 Makros 378
A.1.3 Suffix rules 380
Index 381

Literatur

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

[2] Benutzerhandbücher für C-Compiler Ihres Systems

[3] Kernighan, B.W./ D.M. Ritchie
Programmieren in C - 2. Ausg. mit dem neuen ANSI Standard
Hanser, 1990

[4] Tondo, C./ E. Gimpel
Das C-Lösungsbuch
(zu [4])
Hanser, 1990

[5] Samuel P. Harbison/ Guy L. Steele Jr.
C: A Reference Manual - Third Edition
Prentice-Hall, 1991

[6] Samuel P. Harbison/ Guy L. Steele Jr.
C Ein Referenzhandbuch
Wolfram's, 1992

Einleitung

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)

1972 **C**
Dennis M. Ritchie (Bell Labs)
Implementierungssprache für UNIX

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**

[1] MIT = Massachusetts Institute of Technologie

[2] ANSI = American National Standards Institute

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

Beispielprogramm: Ausgabe von Zahlen (Version 1)

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

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

  i = 1;
  while ( i <= LIMIT )
  {
    printf("%d\n", i);
    i = i + 1;
  }
  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.

Die Zeile

```
while ( i <= LIMIT )
```

wird vom Präprozessor ersetzt durch:

```
while ( i <= 5 )
```

Entsprechend kann **EXIT_SUCCESS** (Wert, der erfolgreiche Programmausführung anzeigen soll) in der Zeile

```
exit(EXIT_SUCCESS);
```

vom Präprozessor ersetzt werden durch:

```
exit(0);
```

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.

#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.

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).

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*).

Ausgabe von Zahlen (Version 2)

```
#include <stdio.h>
#include <stdlib.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;
  }
  exit(EXIT_SUCCESS);
}
```

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

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

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

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

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

  c = getchar();
  while ( c != EOF )
  {
    putchar(c);
    c = getchar();
  }
  exit(EXIT_SUCCESS);
}
```

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

Kopieren der Eingabe zur Ausgabe (Version 2)

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

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

  while ( (c = getchar()) != EOF )
    putchar(c);
  exit(EXIT_SUCCESS);
}
```


Übersetzen, Binden und Ausführen

Namenskonventionen

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

Übersetzen und Binden

unter UNIX: cc [-o *exename*] [*optionen*] *fn*.c [fn_2.c ...]

- übersetzt die angegebenen Programme
- startet automatisch den Bindevorgang
- erzeugt eine ausführbare Datei mit dem Namen *exename*. (Default für *exename*: a.out)

wichtige Optionen:

-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 **libm.a** (*IEEE Math Library*) in den Bindevorgang einzubeziehen.

-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. (Default für *definition* ist 1.) Die Option kann mehrfach angegeben werden.

-P, -E geben an, daß nur der Präprozessor (nicht der Compiler) aufgerufen werden soll. Für die Ausgabe des Präprozessors wird eine **.i**-Datei erzeugt (-P) oder **stdout** benutzt (-E).

-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 C for AIX Compilers (die genannten Optionen sind standardmäßig alle ausgeschaltet):

-qsrcmsg bewirkt, daß Fehlermeldungen zusammen mit der fehlerhaften Quelltextzeile ausgegeben werden.

-qinfo veranlaßt den Compiler, zusätzliche Meldungen auszugeben, die auf fehlende Funktionsprototypen hinweisen.

-qinitauto=*hh*
weist den Compiler an, alle **auto**-Objekte (Variablen, Felder, . . . mit der Speicherklasse **auto**) byteweise mit dem hexadezimal angegebenen Wert *hh* (z.B. FA) zu initialisieren.

-qro plaziert Zeichenkettenkonstanten im schreibgeschützten (*read-only*) Speicher.

-qroconst plaziert auf top-level definierte **const**-Objekte im schreibgeschützten (*read-only*) Speicher.

-qattr erzeugt eine Attribut-Liste (enthält Informationen zu allen in einer Quelldatei benutzen Namen).

-qxref erzeugt eine Cross-Reference-Liste.

Online-Hilfe (für weitere Optionen):

man cc

Ausführen

exename [*argumente*] [<*fid*$_1$] [{ >*fid*$_2$ | >>*fid*$_2$ }]

Ein-/Ausgabeumlenkung (*input/output redirection*):

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

>*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).

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. (Läßt sich diese Datei 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, die zur C-Implementierung gehören benutzt. 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 (irgendeine) Datei zu kopieren. (Die Zieldatei sollte vorher nicht existieren.)
   - Hängen Sie mit Hilfe des Programms eine Kommentarzeile an die Kopie an.

Grundelemente der Sprache

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).

Alternativdarstellungen (*Trigraph*-Sequenzen)

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

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|
```


Kommentar

- beginnt mit **/***
- endet mit ***/**
- Compiler behandelt einen Kommentar wie ein Leerzeichen.

Beispiele:

```
/* Kommentare koennen sich ueber
   mehrere Zeilen erstrecken.
*/

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

/**************************************/
/*   Wenn Sie moechten, koennen Sie   */
/*   Kommentare auch umrahmen.        */
/**************************************/

/* Kommentare
   koennen /* 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
   . . .
#endif
```

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

Grundsymbole (Token)

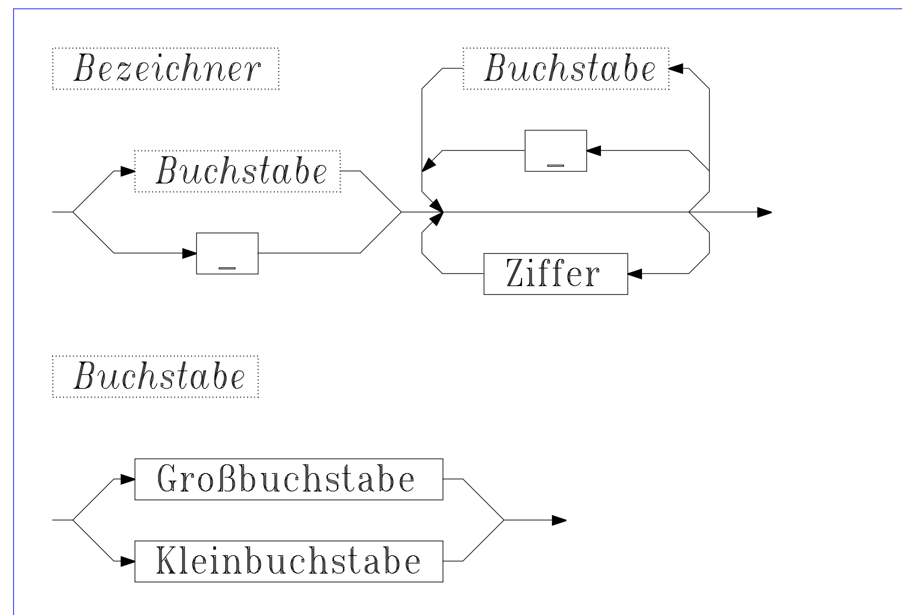
1. Schlüsselwörter
2. Bezeichner
3. Operatoren
4. Punktsymbole
5. Konstanten
6. Strings

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
```

- reservierte Wörter
  (können nicht umdefiniert werden)
- haben eine spezielle Bedeutung
- Kleinschreibung signifikant
- Schlüsselwörter **asm** und **fortran** oft als Erweiterung (z.B. bei CRAY; jedoch nicht bei C for AIX)

Bezeichner



- sind Namen für Variablen, Typen, Funktionen, Makros, ...
- Schlüsselwörter nicht als Bezeichner erlaubt
- beliebig lange Zeichenfolgen, jedoch betrachtet jede Implementierung nur eine bestimmte Anzahl von Zeichen als signifikant.

- „interne“ Bezeichner:
  - ☐ mindestens die ersten 31 Zeichen signifikant
  - ☐ Groß-/Kleinschreibung wird unterschieden
  - ☐ dürfen nicht mit „_*Großbuchstabe*“ oder „__“ beginnen (reserviert für die C-Implementierung)
- „externe“ Bezeichner:
  - ☐ mindestens 6 Zeichen signifikant (C for AIX: 250)
  - ☐ u.U. keine Unterscheidung von Groß- und Kleinbuchstaben (z.B. Compiler unter älteren Betriebssystemen; jedoch Unterscheidung z.B. auf UNIX-Systemen)
  - ☐ dürfen nicht mit „_“ beginnen

Konventionen:

- Makronamen werden i. allg. vollständig mit Großbuchstaben geschrieben
- anderen Namen werden vollständig klein geschrieben (oder in Groß-/Kleinschreibung)

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. beim Ansprechen einer Komponente in einer Struktur)
- verursacht Seiteneffekte

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.

Konstanten

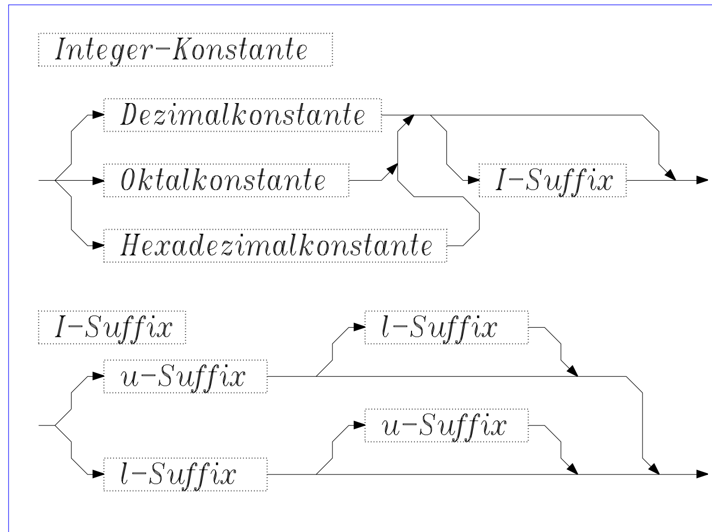
Eine Konstante

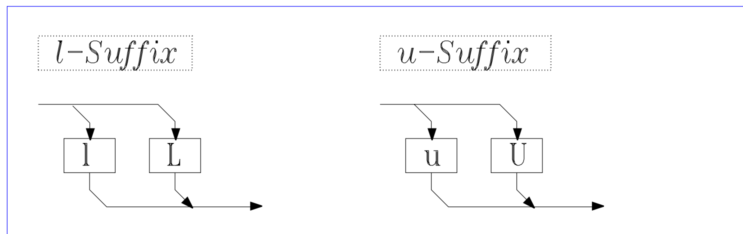
- repräsentiert einen numerischen Wert.
- besitzt einen bestimmten Datentyp.

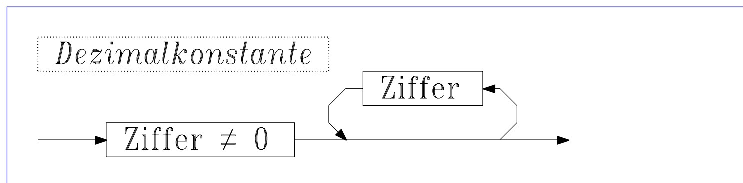
Arten von Konstanten:

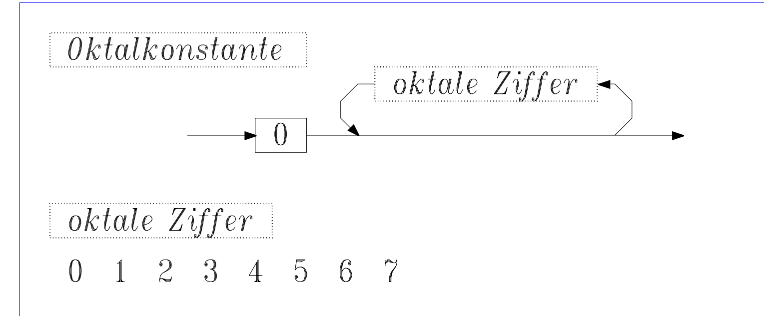
1. Integer-Konstanten
2. Gleitkommakonstanten (*floating constants*)
3. Character-Konstanten
4. Aufzählungskonstanten (*enumeration constants*)

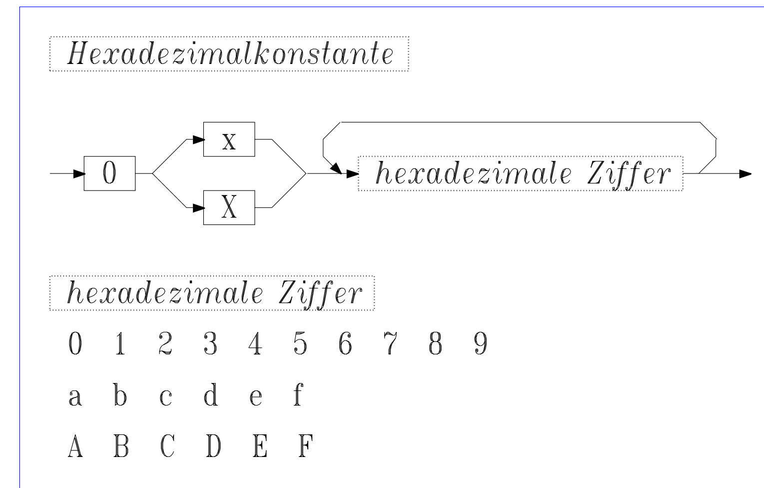
Integer-Konstanten











Bestimmung des Typs von Integer-Konstanten

Konstante	Typ
Dezimalkonstante ohne Suffix	**int** **long int** **unsigned long int**
Oktal- oder Hexadezimalkonstante ohne Suffix	**int** **unsigned int** **long int** **unsigned long int**
Konstante mit u-Suffix	**unsigned int** **unsigned long int**
Konstante mit l-Suffix	**long int** **unsigned long int**
Konstante mit l- und u-Suffix	**unsigned long int**
Anmerkung: Von den angegebenen Typen wird jeweils der *erste* ausgewählt, der die Darstellung des Wertes erlaubt.	

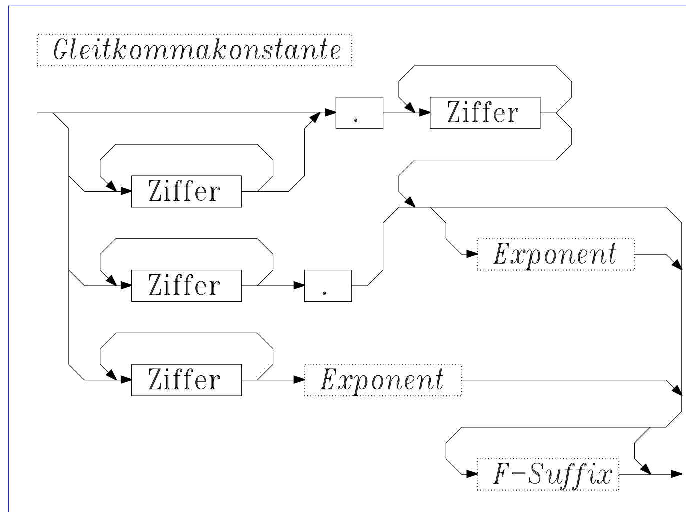
Beispiele:

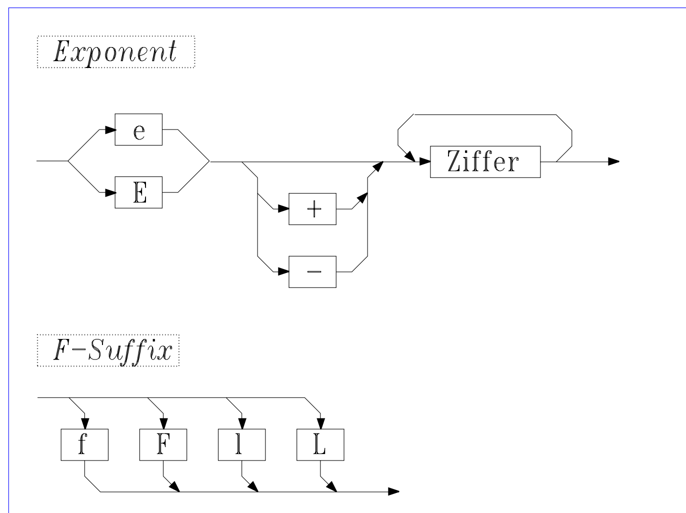
Annahmen:

int	16 Bit (Wertebereich: $-32768 \leq x \leq 32767$)
⇒**unsigned int**	16 Bit (Wertebereich: $0 \leq x \leq 65535$)
long int	32 Bit

485	0	123	0173	0x7b	**int**
32u	0100000	0xFFFF	0xFFFFU		**unsigned int**
0L	42976				**long int**

Gleitkommakonstanten





Gleitkomma-Typen

Konstante	Typ
ohne Suffix	**double**
mit Suffix f oder F	**float**
mit Suffix l oder L	**long double**

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:

```
'\''
'\"'
'"'
'\\'    ('\' wird nicht durch ' abgeschlossen)
'\0'    (Zeichen mit dem Wert 0)
'\15'
'\015'
'\x000D'    (die letzten 3 Konstanten haben den
             gleichen Wert)
```


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_1 o_2$ \$o_1 o_2 o_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**

Beispielsweise ist in der Aufzählung

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

blau eine Aufzählungskonstante.

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

- die Länge eines Strings ist die Anzahl der Zeichen (ohne \0)
- ein String der Länge n belegt einen Speicherbereich von n+1 Bytes
- aufeinanderfolgende Strings werden automatisch zu einem String verkettet (z.B.: "ab" "c"). Das Null-Zeichen wird erst **nach** einer Verkettung angehängt.

Beispiele:

```
"\""
"\'"
"'"
""   (Null-String)
"Was ist das\?\?!\n"
"\x0DEin Zeichen mit Wert 0xDE geht voraus."
"\x0D"  "Ein Zeichen mit Wert 0xD geht voraus."
```

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-character literal*)

Regeln für das Einfügen von Zwischenraumzeichen

Zwischenraumzeichen:

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

Für die Verwendung von Zwischenraumzeichen (außerhalb von Zeichenkonstanten und Strings) gilt:

1. Zwischen 2 aufeinanderfolgenden Token (Grundsymbolen), die Bezeichner, Schlüsselwort oder Konstante sind, muß mindestens ein Zwischenraumzeichen stehen.

2. Zwischen 2 aufeinanderfolgenden Token, die Operator, Punktsymbol oder Strings sind, dürfen Null, ein oder mehrere Zwischenraumzeichen auftreten.

3. Operatoren und Punktsymbole, die sich aus mehreren Zeichen zusammensetzen (z.B. „++“ oder „...“) müssen ohne Zwischenraumzeichen geschrieben werden.

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"
```


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

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

```
a.  pipesendintake           c.  O77U
b.  Const                    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"	

Basistypen, Operatoren und Ausdrücke

Basistypen (Übersicht)

Hierzu zählen alle arithmetischen Datentypen mit Ausnahme des Aufzählungstyps.

arithmetische Datentypen:

- integrale Typen (repräsentieren ganzzahlige Werte)
  - ☐ **char**
  - ☐ Integer-Typen mit Vorzeichen (*signed integer types*)
    - **signed char**
    - [**signed**] **int**
    - [**signed**] **long** [**int**]
    - [**signed**] **short** [**int**]
  - ☐ Integer-Typen ohne Vorzeichen (*unsigned integer types*)
    - **unsigned char**
    - **unsigned** [**int**]
    - **unsigned long** [**int**]
    - **unsigned short** [**int**]
  - ☐ Aufzählungstypen (*enumerated types*)
- Gleitkommatypen
  - ☐ **float**
  - ☐ **double**
  - ☐ **long double**

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.
     ⇒ Er kann keine Seiteneffekte erzeugen.

Typ des Resultatwertes:

- abhängig von C-Implementierung
- integraler Typ ohne Vorzeichen: **size_t**
  - definiert in **stddef.h**
  - 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 veraendert! */
```

Integer-Typen

- **mit Vorzeichen**

  Der betragsmäßig größte darstellbare Wert ist i. allg. halb so groß wie beim entsprechenden Typ **ohne** Vorzeichen.

  – [**signed**] **short** [**int**]

  Größe: **sizeof**(**char**) ≤ **sizeof**(**short int**)

  Wertebereich:

  - angegeben durch Makros: **SHRT_MIN**, **SHRT_MAX**
  - definiert in **limits.h**

  $$W_{shrt} = \{ i \mid \mathbf{SHRT_MIN} \leq i \leq \mathbf{SHRT_MAX} \wedge \mathbf{SHRT_MIN} \leq -32767 \wedge \mathbf{SHRT_MAX} \geq +32767 \}$$

– {**int** | **signed** [**int**]}

Größe:
sizeof(**short int**)≤**sizeof**(**int**)≤**sizeof**(**long int**)
entspricht i. allg. der Wortlänge (Registerlänge) des Rechners.

Wertebereich:

$$W_{int} = \{ i \mid \mathbf{INT_MIN} \leq i \leq \mathbf{INT_MAX} \wedge \mathbf{INT_MIN} \leq -32767 \wedge \mathbf{INT_MAX} \geq +32767 \}$$

– [**signed**] **long** [**int**]

Größe:
sizeof(**int**) ≤ **sizeof**(**long int**)

Wertebereich:

$$W_{long} = \{ i \mid \mathbf{LONG_MIN} \leq i \leq \mathbf{LONG_MAX} \wedge \mathbf{LONG_MIN} \leq -(2^{31} - 1) \wedge \mathbf{LONG_MAX} \geq +2^{31} - 1 \}$$ [3]

[3] $2^{31} - 1 = 2\,147\,483\,647$

- **ohne Vorzeichen**

Größen:

unsigned–Typen belegen jeweils genauso viel Speicherplatz wie der korrespondierende **signed**–Typ (außerdem gelten die gleichen Alignment-Anforderungen):

sizeof(**unsigned** *type*) == **sizeof**(**signed** *type*)

Wertebereiche:

- **unsigned short** [**int**]

  W_{ushrt}={i|0 ≤ i ≤ **USHRT_MAX** ∧ **USHRT_MAX** ≥ 65535}

- **unsigned** [**int**]

  W_{uint}={i|0 ≤ i ≤ **UINT_MAX** ∧ **UINT_MAX** ≥ 65535}

- **unsigned long** [**int**]

  W_{ulong}={i|0 ≤ i ≤ **ULONG_MAX** ∧ **ULONG_MAX** ≥ $2^{32} - 1$}[4]

[4] $2^{32} - 1$ = 4 294 967 295

- **Integer-Typen zur Darstellung von Zeichen**
  - **signed char**
  - **unsigned char**
  - **char**

    entspricht **signed char** oder **unsigned char** (abhängig von C-Implementierung)

  Ein **char**–Objekt belegt per Definition **1 Byte**[5] (**sizeof**(**char**) == 1).

  für Source- und Ausführungszeichensatz gilt:

  Der Wert eines Zeichens, das eine Dezimalziffer ist, muß sich aus dem Wert des Zeichens für die nächstkleinere Dezimalziffer durch Addition von 1 ergeben (z.B. ’5’ ==’4’+1)

  ⇒ durch Subtraktion des Wertes ’0’ erhält man die entsprechende Dezimalziffer:
  ’5’-’0’ → 5

[5] **CHAR_BIT** (in **limits.h** definiertes Makro) gibt die Anzahl Bits pro Byte an: **CHAR_BIT** ≥8

Wertebereiche:

– **signed char**

W_{schar}={i|**SCHAR_MIN** ≤ i ≤**SCHAR_MAX** ∧ **SCHAR_MIN** ≤ –127 ∧ **SCHAR_MAX** ≥ +127}

– **unsigned char**

W_{uchar}={i|0 ≤ i ≤ **UCHAR_MAX** ∧ **UCHAR_MAX** ≥ 255}

– **char**

W_{char}={i|**CHAR_MIN**≤i≤**CHAR_MAX**}

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

Funktion	**Zeichenklasse**
isspace(*c*)	Zwischenraumzeichen (' ', '\n', '\r', '\t', '\v', '\f')
ispunct(*c*)	Interpunktionszeichen (*punctuation marks*): druckbare Zeichen außer Leerzeichen, Buchstaben und Ziffern
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*

Logische Werte

werden durch die integralen Typen dargestellt:

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
```

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*

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:

```
/* Typdeklarationen */

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

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

/* 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 ) ...
```

Integer-Erweiterung
(*integral promotion*)

Ein Objekt des Typs **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.

falls für einen Integer-Typ *T* gilt:	**wird *T* umgewandelt nach**
sizeof(*T*) < **sizeof**(**int**)	**int**
T ist **signed** und **sizeof**(*T*) == **sizeof**(**int**)	**int**
T ist **unsigned** und **sizeof**(*T*) == **sizeof**(**int**)	**unsigned int**

Beispiel:

```
unsigned short s=1;
  ...
printf("s = %d\n", s);
```

Implementierung A:

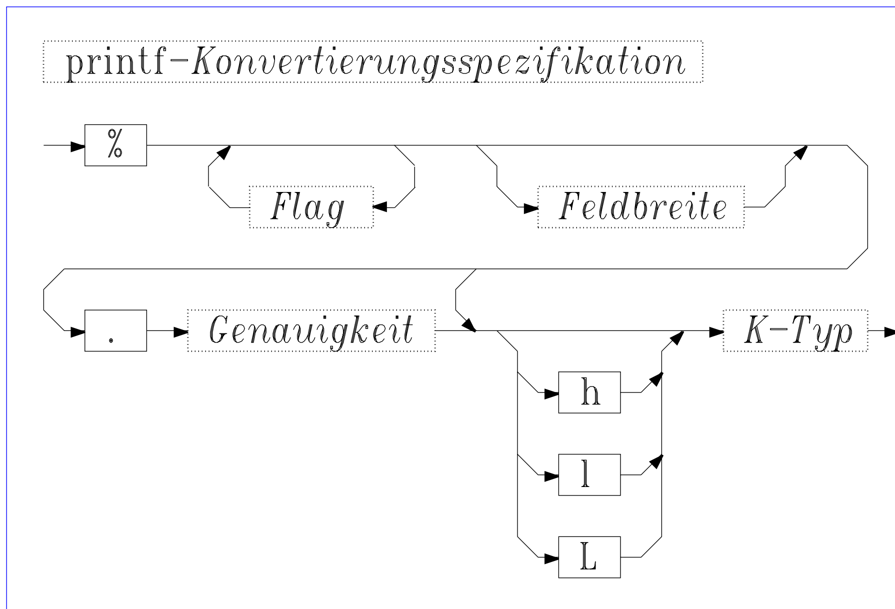
Hier gelte: **sizeof**(**short**) < **sizeof**(**int**)
Umwandlung: s → **int**

Implementierung B:

Hier gelte: **sizeof**(**short**) == **sizeof**(**int**)
Umwandlung: s → **unsigned int**

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 der Parameterliste von **printf** bestimmt wird.)

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

h 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

K-Typ	**Argument-typ**	**Argument wird dargestellt als**
d, i	**int**	Dezimalzahl
o	**int**	Oktalzahl (ohne Vorzeichen)
x	**int**	Hexadezimalzahl (ohneVorzeichen; Ziffern>9: 'a'-'f')
X	**int**	Hexadezimalzahl (ohneVorzeichen; Ziffern>9: 'A'-'F')
u	**int**	Dezimalzahl (ohne Vorzeichen)
c	**int**	einzelnes Zeichen (Argument → **unsigned char**; **h**, **l** nicht zulässig)
%	-	bewirkt die Ausgabe eines %-Zeichens

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

Beispiele:

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

int main( void )
{
  char     c = '3';
  int      i = -1;
  long int lo = 456;

  printf("%c %d %i %u\n", c, c, c, c);
  printf("%o %x %X\n", c, c, c);

  printf("%d %u %hu %ld\n", i, i, i, lo);
  printf("%%\n");
  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
3 51 51 51
63 33 33
-1 4294967295 65535 456
%
```

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

int main( void )
{
  int spos, sneg;
  int fwidth;

  spos = 123;
  sneg = -spos;
  printf(
    "....+....1....+.\n%d%d\n\n",
    spos, sneg);
  printf(              /* Feldbreite 6 */
    "....+....1....+.\n%6d%6d\n\n",
    spos, sneg);
  printf(              /* Flag - */
    "....+....1....+.\n%-6d%-6d\n\n",
    spos, sneg);
  printf(              /* Flag + */
    "....+....1....+.\n%+6d%+6d\n\n",
    spos, sneg);
  printf(              /* Flag <Leerzeichen> */
    "....+....1....+.\n% d% d\n\n",
    spos, sneg);
  printf(              /* Flag 0 */
    "....+....1....+.\n%06d%06d\n\n",
    spos, sneg);
  printf(              /* mindestens 4 Ziffern */
    "....+....1....+.\n%6.4d%6.4d\n\n",
    spos, sneg);
  printf(
    "....+....1....+.\n%1d%1d\n\n",
    spos, sneg);
```



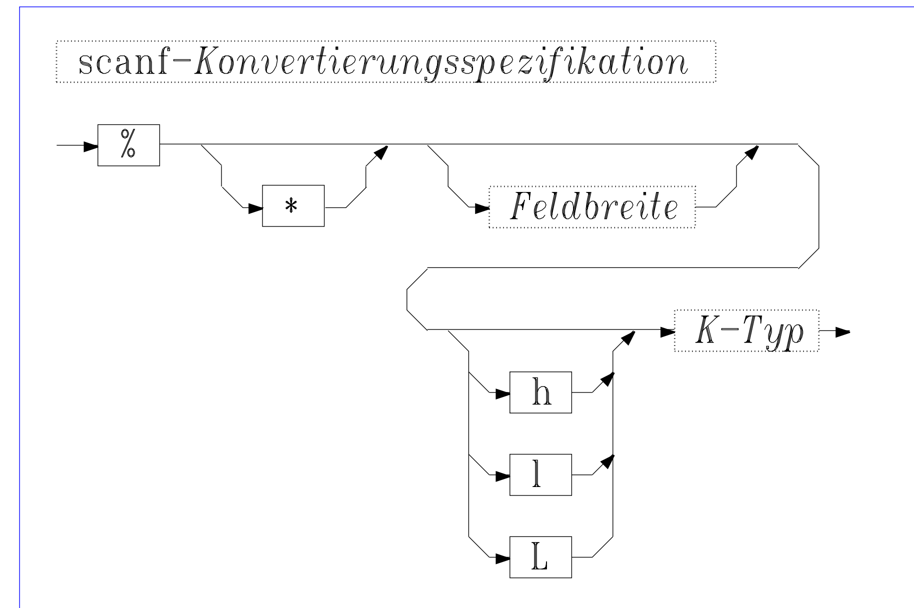
```
  fwidth = 8;
  printf(              /* Feldbreite variabel */
    "....+....1....+.\n%*d%0*d\n\n",
    fwidth, spos, fwidth-2, sneg);

  printf("%o, %o\n\n", spos, sneg);
  printf("%#o, %#o\n\n", spos, sneg);/* Flag # */

  printf("%x, %X\n\n", spos, sneg);
  printf("%#x, %#X\n\n", spos, sneg);/* Flag # */

  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
....+....1....+.
123-123

....+....1....+.
   123  -123

....+....1....+.
123    -123

....+....1....+.
  +123  -123

....+....1....+.
 123-123

....+....1....+.
000123-00123

....+....1....+.
  0123 -0123

....+....1....+.
123-123

....+....1....+.
     123-00123

173, 37777777605

0173, 037777777605

7b, FFFFFF85

0x7b, 0XFFFFFF85
```


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 in der Parameterliste von **scanf**.
Feldbreite	maximale Breite des Eingabefeldes (Dezimalzahl). Standardmäßig beginnt das Eingabefeld mit dem nächsten Zeichen in der Eingabe, das kein Zwischenraumzeichen ist (vorausgehende Zwischenraumzeichen werden überlesen[6]), und geht bis zum ersten Zeichen, das der Formatspezifikation nicht mehr entspricht.
h	in Verbindung mit den Formatbuchstaben: d, i, n zeigt an, daß das zugehörige Argument ein Zeiger auf ein **short**-Objekt ist. o, u, x 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 *** o, u, x zugehöriges Argument: **unsigned long ***
L	→ formatierte Eingabe von Gleitkommatypen

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

Basistypen, Operatoren und Ausdrücke
Formatierte Eingabe von Integer-Typen

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.

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; keine Interpretation als Oktalkonstante)
o	**unsigned ***	[+|-]*Oktalkonstante* (führende **0** nicht erforderlich)
x	**unsigned ***	[+|-]*Hexadezimalkonstante* (**0x**- bzw. **0X**-Präfix nicht erforderlich)
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 gelesenen Datenelemente oder **EOF**, falls vor Zuweisung des ersten Datenelementes das Dateiende erreicht wird

Beispiele:

```
#include <stdio.h>
#include <stdlib.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);

  exit(EXIT_SUCCESS);
}
```

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
```

Ausgabe:

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

Fehlerstelle:

```
4  km      21 km     33 km
             |
```

Beispiel für %n in Verbindung mit **printf**:

```
int count, n;

printf("Beispiel %d:%n\n", count, &n);
while ( n > 0 )
{
  putchar('-');
  n = n - 1;
}
putchar('\n');
```


Aufgaben

1. Testen Sie, wie sich der von Ihnen benutzte Compiler verhält, wenn in einem Programm eine Aufzählungskonstante an eine Variable eines anderen Aufzählungstyps zugewiesen wird.
2. Geben Sie die Resultatwerte der Ausdrücke

   USHRT_MAX + 2
   UINT_MAX + 2

   aus und erklären Sie die Ausgabe.

   Macht es einen Unterschied, ob die Ausdrücke direkt als Argumente für **printf** angegeben oder zunächst an eine entsprechende **unsigned**-Variable zugewiesen werden?
3. Führen Sie das folgende Programm aus, und geben Sie dabei für `zahln` jeweils den Wert *n* ein.

   Erklären Sie die Ausgabe und begründen Sie, warum das Programm andere Ergebnisse liefert, wenn es mit der Option -O (Code-Optimierung) übersetzt wird.

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

int main( void )
{
  short int zahl2, zahl1 = 0;
  int       zahl3;

  zahl2 = 123;
  scanf("%d%d", &zahl1, &zahl3);
  printf(
    "zahl1 = %d\nzahl2 = %d\nzahl3 = %d\n",
    zahl1, zahl2, zahl3);
```

```
  printf(
    "zahl1 = %hd\nzahl2 = %hd\nzahl3 = %hd\n",
    zahl1, zahl2, zahl3);

  scanf("%d%d", &zahl1, &zahl2);
  printf(
    "zahl1 = %d\nzahl2 = %d\nzahl3 = %d\n",
    zahl1, zahl2, zahl3);
  printf(
    "zahl1 = %hd\nzahl2 = %hd\nzahl3 = %hd\n",
    zahl1, zahl2, zahl3);
  exit(EXIT_SUCCESS);
}
```

4. Modifizieren Sie das 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 (⇒ der zweite Aufruf von **scanf** soll immer aus einer neuen Zeile lesen.).
   b. Ändern Sie den Typ der Variablen:

      s1, **s3** → **long**

      i, **vH** → **short**

Gleitkommatypen

Hierzu zählen die Typen **float**, **double** und **long double**.

float

sizeof(**float**) ≤ **sizeof**(**double**)

Eine Reihe von Merkmalen der Gleitkommatypen sind in der Header-Datei **float.h** definiert:

- größte darstellbare Zahl:
  FLT_MAX ≥ 1E+37
- betragsmäßig kleinste darstellbare Zahl:
  FLT_MIN ≤ 1E-37
- Genauigkeit (Anzahl Dezimalziffern):
  FLT_DIG ≥ 6
- kleinste Zahl x für die gilt: $1.0\mathrm{F} + x \neq 1.0\mathrm{F}$
  FLT_EPSILON ≤ 1E-5F

double

sizeof(**float**) ≤ **sizeof**(**double**) ≤ **sizeof**(**long double**)

- größte darstellbare Zahl:
  DBL_MAX ≥ 1E+37
- betragsmäßig kleinste darstellbare Zahl:
  DBL_MIN ≤ 1E-37
- Genauigkeit (Anzahl Dezimalziffern):
  DBL_DIG ≥ 10
- kleinste Zahl x für die gilt: $1.0 + x \neq 1.0$
  DBL_EPSILON ≤ 1E-9

long double

sizeof(**double**) ≤ **sizeof**(**long double**)

- größte darstellbare Zahl:
  LDBL_MAX ≥ 1E+37
- betragsmäßig kleinste darstellbare Zahl:
  LDBL_MIN ≤ 1E-37
- Genauigkeit (Anzahl Dezimalziffern):
  LDBL_DIG ≥ 10
- kleinste Zahl *x* für die gilt: $1.0L + x \neq 1.0L$
  LDBL_EPSILON ≤ 1E-9L

weitere Makros in **float.h**:

FLT_ROUNDS
gibt die Rundungsart für die Gleitkomma-Addition an:

-1 unbestimmt
0 in Richtung 0, d.h. Abschneiden (*truncation*)
1 zum nächsten darstellbaren Wert
2 in Richtung +∞, d.h. es wird immer aufgerundet
3 in Richtung -∞, d.h. es wird immer abgerundet

$$x = s\, b^e \sum_{k=1}^{p} f_k\, b^{-k}$$

s Vorzeichen (±1)

b Basis für die Darstellung des Exponenten (für alle Gleitkommatypen gleich): **FLT_RADIX**

e Exponent (Integer-Zahl zwischen **..._MIN_EXP** u. **..._MAX_EXP**); die Makros **..._MIN_10_EXP** und **..._MAX_10_EXP** geben den kleinsten bzw. größten zulässigen Exponenten bezogen auf die Basis 10 an

p Genauigkeit (Anzahl der Ziffern bzgl. Basis *b* in der Mantisse): **FLT_MANT_DIG**, **DBL_MANT_DIG**, **LDBL_MANT_DIG**

f Ziffer der Mantisse

Formatierte Ausgabe von Gleitkommatypen

printf-Konvertierungsspezifikation

darf enthalten:

Flags -, +, Leerzeichen, 0

\# gibt an, daß in jedem Fall ein Dezimalpunkt ausgegeben werden soll.
In Verbindung mit dem Formatbuchstaben g oder G werden zusätzlich auch Nullen am Ende des Nachkommateils ausgegeben.

Feldbreite

Genauigkeit in Verbindung mit den Formatbuchstaben:

f, e, E Anzahl der Ziffern nach dem Dezimalpunkt (Standardvorgabe: 6)

g, G Anzahl signifikanter Ziffern

L zeigt an, daß das zugehörige Argument den Typ **long double** hat.

K-Typ

Standarderweiterung von **float**-Argumenten:

Ein Argument von **printf** mit dem Typ **float** wird automatisch in den Typ **double** umgewandelt.

K-Typ	**Argumenttyp**	**Argument wird dargestellt als**
f	**double**	[-]*int_part.dddddd*
e, E	**double**	[-]*d.dddddd*e±*dd* bzw. [-]*d.dddddd*E±*dd*
g, G	**double**	entspricht %e bzw. %E, falls Exponent≥Genauigkeit oder Exponent<-4, andernfalls %f

Beispiele:

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

int main( void )
{
  float f;
  double d;
  long double ld;

  f = 3.14159f;
  d = 123456.789012345;
  ld = 0.12345678901234567890123456789012L;
  printf(
    "%f  %.3f  %.0f  %#.0f  %6.3f  %06.3f\n",
    d, d, f, f, f, f);
  printf("%Le  %.10LE  %.0Le  %#.0LE\n",
    ld, ld, ld, ld);
  printf("%e  %g  %G\n",
    0.0001, 0.0001, 0.00001);
  printf("%.3g  %.6G\n", d, d);
  printf("%.4g  %#.4G\n", 100.0, 100.0);
  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
123456.789012  123456.789  3  3.   3.142  03.142
1.234568e-01  1.2345678901E-01  1e-01  1.E-01
1.000000e-04  0.0001  1E-05
1.23e+05  123457
100  100.0
```


Formatierte Eingabe von Gleitkommatypen

K-Typ	**Argumenttyp**	**Eingabe**
f, e, g	**float ***	[+|-]*Gleitkommakonstante* (ohne *F-Suffix*; außerdem darf der Dezimalpunkt weggelassen werden)

Die Konvertierungsspezifikation darf unmittelbar vor *K-Typ* einen der folgenden Buchstaben enthalten:

l zeigt an, daß das zugehörige Argument ein Zeiger auf ein **double**-Objekt ist.

L zeigt an, daß das zugehörige Argument ein Zeiger auf ein **long double**-Objekt ist.

Aufgabe

Erklären Sie (möglichst exakt) wie sich das Weglassen des Buchstaben **l** in der Konvertierungsspezifikation beim Einlesen einer **double**-Variablen auf dem von Ihnen verwendeten Rechner auswirkt.

Zuweisungsoperator

Syntax:

lvalue = *expr*

lvalue

Ausdruck, der ein modifizierbares Objekt im Speicher bezeichnet (z.B. Variablenname)

expr

Ausdruck, dessen Wert an *lvalue* zugewiesen wird; falls der Typ von *expr* und *lvalue* ein arithmetischer Typ ist, wird der Wert von *expr* zuvor in den Typ von *lvalue* umgewandelt

Resultatwert der Zuweisung:

der in den Typ von *lvalue* umgewandelte Wert von *expr*

cast-Operator (explizite Typumwandlung)

Syntax:

(*typename*) *expr*

typename

Datentyp, in den der Wert von *expr* umgewandelt werden soll

Resultatwert:

- der in den Typ *typename* umgewandelte Wert von *expr*
- ist kein *lvalue*

Beispiele:

```
enum Wochentag { Montag, Dienstag, Mittwoch,
  Donnerstag, Freitag, Samstag, Sonntag };
  ...
enum Wochentag Tag;
int i;
double d;
    ...
  i = (int) d;
  Tag = (enum Wochentag) (i % 7); /* i modulo 7 */
```


Typumwandlungen

- Integer-Typ → Integer-Typ
  Der ursprüngliche Wert bleibt unverändert, falls er im neuen Typ darstellbar ist, andernfalls gilt:
  a. bei Umwandlung in einen[7]
     i.) gleich großen **unsigned**–Typ:
         Bitmuster bleibt unverändert
     ii.) größeren **unsigned**–Typ:
         höherwertige Bits werden mit Einsen (Kopien des Vorzeichenbits) aufgefüllt
     iii.) kleineren **unsigned**–Typ:
         höherwertige Bits werden abgeschnitten
  b. bei Umwandlung in einen **signed**–Typ:
     Resultat abhängig von C-Implementierung
- Gleitkommatyp → Integer-Typ
  Nachkommateil wird abgeschnitten
  (Der Resultatwert ist undefiniert, falls der ganzzahlige Teil im neuen Typ nicht darstellbar ist.)

[7] zur Vereinfachung wird angenommen, daß Integer-Werte im Zweierkomplement dargestellt werden

- Gleitkommatyp → Gleitkommatyp
  bei Umwandlung in einen
  a. gleich großen oder größeren Typ:
     Wert bleibt unverändert
  b. kleineren Typ:
     Wert wird nach oben oder unten gerundet
     (Der Resultatwert ist undefiniert, falls der Wert im neuen Typ nicht darstellbar ist.)
- Integer-Typ → Gleitkommatyp
  Wert wird ggf. nach oben oder unten gerundet

© Forschungszentrum Jülich

Arithmetische Umwandlungen

- werden auf die Operanden der meisten binären Operatoren angewandt
- bewirken, daß die Operanden in einen gemeinsamen Typ (= Resultattyp) umgewandelt werden

© Forschungszentrum Jülich

Umwandlungsregeln:

Die erste anwendbare Regel bestimmt die durchzuführende Umwandlung.

	ist ein Operand vom Typ	wird der andere Operand umgewandelt nach
1.	**long double**	**long double**
2.	**double**	**double**
3.	**float**	**float**

Die folgenden Regeln werden erst überprüft, nachdem für beide Operanden die Integer-Erweiterung durchgeführt ist.

4.	**unsigned long int**	**unsigned long int**
5.	**long int**	**long int** oder **unsigned long int**[8]
6.	**unsigned int**	**unsigned int**

Falls keine der Regeln zutrifft, haben beide Operanden den Typ **int**.

[8] hat der andere Operand den Typ **unsigned int** und schließt der Wertebereich von **long int** den Wertebereich von **unsigned int** nicht vollständig ein, dann werden beide Operanden nach **unsigned long int** umgewandelt.

Beispiel:

Kopieren der Eingabe zur Ausgabe (falsche Version)

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

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

  while ( (c = getchar()) != EOF )
    putchar(c);
  exit(EXIT_SUCCESS);
}
```

Annahmen:

- **char** entspricht **unsigned char**
- **EOF** == –1

Arithmetische Operatoren

Operator	Operation	O-Typ[9]	IE[10]	aU[11]
+ *a*	Identität	a	•	
- *a*	Vorzeichenumkehr	a	•	
a + *b*	Addition	a, ...		•
a - *b*	Subtraktion	a, ...		•
a * *b*	Multiplikation	a		•
a / *b*	Division (Falls *a* und *b* einen Integer-Typ haben und mindestens einer der beiden Werte negativ ist, dann hängt der Resultatwert von der C-Implementierung ab. [→ Beispiel])	a		•
a % *b*	Modulo (Der Ausdruck (*a*/*b*)**b* + *a*%*b* muß *a* liefern.)	i		•

[9] Typ der (des) Operanden: a = arithmetisch, i = Integer-Typ

[10] IE = Integer-Erweiterung

[11] aU = arithmetische Umwandlungen

Beispiele:

folgende Vereinbarung wird vorausgesetzt:

```
int i = 7;
```

Ausdruck	Resultattyp	Resultatwert
i / 2	**int**	3
i % 2	**int**	1
((float) i) / 2	**float**	3.5f
i / -2.0	**double**	-3.5
i / -2	**int**	-3 oder -4
i % -2	**int**	1 oder -1

Bibliotheksfunktionen **div** und **ldiv**:

```
#include <stdlib.h>
div_t div(int a, int b);
ldiv_t ldiv(long int a, long int b);
```

- Resultatwert:
  - Struktur mit dem Typ **div_t** bzw. **ldiv_t**
  - 2 **int**– bzw. **long**–Komponenten: **quot**, **rem**
- Resultat auch für negative Argumente eindeutig definiert (gleiche Resultate wie in FORTRAN)

Beispiel:

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

int main( void )
{
  div_t erg;

  printf("   a |   b | a/b | a%%b\n");
  printf("-----+-----+-----+-----\n");

  erg = div(7, -2);
  printf("   7 |  -2 | %3d | %3d \n",
    erg.quot, erg.rem);
   /*  -3        1 */
  exit(EXIT_SUCCESS);
}
```


Inkrement- und Dekrement-Operatoren

Operator	Operation
`++`i	Präfix-Inkrementierung ⇔ $i = i + 1$ i wird zuerst inkrementiert; Resultatwert der Operation ist der neue Wert von i
`--`i	Präfix-Dekrementierung ⇔ $i = i - 1$ i wird zuerst dekrementiert; Resultatwert ist der neue Wert von i
i`++`	Postfix-Inkrementierung der Wert von i ist der Resultatwert der Operation; anschließend wird i inkrementiert
i`--`	Postfix-Dekrementierung der Wert von i ist der Resultatwert; anschließend wird i dekrementiert

- i muß ein l-Wert sein
- Resultat ist kein l-Wert

⇒ `(i++)++` ist unzulässig

Beispiel:

soll die Wirkung von `++` und `--` verdeutlichen

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

int main( void )
{
  int a=0, b=0, c=0;

  a = ++b + ++c;
  printf("%d %d %d\n", a, b, c);   /* 2  1  1 */

  a = b++ + c++;
  printf("%d %d %d\n", a, b, c);   /* 2  2  2 */

  a = ++b + c++;
  printf("%d %d %d\n", a, b, c);   /* 5  3  3 */

  a = b-- + --c;
  printf("%d %d %d\n", a, b, c);   /* 5  2  2 */

  a = ++c + c;          /* maschinenabhaengig */
  printf("%d %d %d\n", a, b, c);

  exit(EXIT_SUCCESS);
}
```


Operatoren für Bitmanipulationen

Operator	Operation	IE[12]	aU[13]
a << n	Links-Shift um n Stellen (Bits)	•	
a >> n	Rechts-Shift um n Stellen (Bits)	•	
a **&** b	bitweises Und		•
a | b	bitweises Oder		•
a ^ b	bitweises exklusives Oder		•
~ a	bitweises Komplement	•	

- alle Operanden (a, b, n) müssen einen Integer-Typ haben
- Resultat der Shift-Operationen ist nur definiert, falls gilt: $0 \leq n <$ Anzahl der Bits von a nach IE
- Shift-Operationen liefern Resultat mit dem Typ von a nach IE
- für Links-Shift gilt:
  - von rechts werden Nullbits nachgeschoben
  - $\Leftrightarrow a * 2^n$

[12] IE = Integer-Erweiterung

[13] aU = arithmetische Umwandlungen

- für Rechts-Shift gilt:
  - falls *a* **unsigned**–Typ hat, werden von links Nullbits nachgeschoben (*logical shift*)
  - andernfalls ist das Resultat von der C-Implementierung abhängig: Es kann auch das Vorzeichenbit nachgeschoben werden (*arithmetic shift*).
  - $\Leftrightarrow a/2^n$, für $a \geq 0$

Beispiel 1:

soll die Wirkung der Operatoren für Bitmanipulationen verdeutlichen

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

int main( void )
{
  unsigned i = 0x39;      /*          .. .011 1001 */
  unsigned j = 0x77;      /*         ... 0111 0111 */
  signed   k = -1;        /* 111. ..11 1111 1111 */

  printf("   i    ==  %#x\n", i);
  printf("   j    ==  %#x\n", j);
  printf("   k    ==  %#x\n", k);
  printf("i << 4  ->  %#x\n",
    i << 4);              /*    ...011 1001 0000 */
  printf("i >> 4  ->  %#x\n",
    i >> 4);              /* 0000 .... .... .011 */
  printf("k >> 4  ->  %#x\n",
    k >> 4);              /* 0000 111. ..11 1111 */
                    /* oder  1111 111. ..11 1111 */
  printf(" i & j  ->  %#x\n",
    i & j);               /* 000. ..00 0011 0001 */
  printf(" i | j  ->  %#x\n",
    i | j);               /* 000. ..00 0111 1111 */
  printf(" i ^ j  ->  %#x\n",
    i ^ j);               /* 000. ..00 0100 1110 */
  printf("  ~i    ->  %#x\n",
    ~i);                  /* 111. ..11 1100 0110 */
  exit(EXIT_SUCCESS);
}
```

Beispiel 2:

Häufig werden Bit-Operatoren dazu verwendet, Flags, die in einer ganzzahligen Variablen codiert sind, zu setzen, zu löschen und abzufragen.

Flag:

- entspricht einem einzelnen Bit
- wird als Potenz von 2 definiert

```
/*================================================*/
/*                                                */
/* Implementierung der Zeichenklassen-Tests in    */
/* ctype.h benutzt ein Feld von Flag-Woertern:    */
/*                                                */
/*================================================*/
#define DIGIT   01 /* Ziffer                      */
#define LOWER   02 /* Kleinbuchstabe              */
#define UPPER   04 /* Grossbuchstabe              */
#define XDIGIT 010 /* hexadezimale Ziffer         */
  ...
char ctype[128];
    ...
  ctype['a'] = LOWER|XDIGIT; /*     ... 1010    */
    ...
                /* Resultatwert von isalpha: */
  return ( ctype['a']        /*     ... 1010    */
           & (LOWER|UPPER)   /*   & ... 0110    */
         );                  /* -------------   */
                             /*  -> ... 0010    */

    ...
```


Beispiel 3:

zeigt eine weitere Anwendung für die Bit-Operatoren

```
/*================================================*/
/* Der folgende Ausdruck liefert den Wert eines */
/* Bit-Feldes, das innerhalb einer unsigned int-*/
/* Variablen a liegt.                           */
/* Die genaue Lage des Bit-Feldes ist durch die */
/* Variablen p und n gegeben:                   */
/*   p = Position, an der das Bit-Feld beginnt  */
/*       (Es ist davon auszugehen, dass die     */
/*       Bits von a entsprechend ihrer Wertig-  */
/*       keit durchnumeriert sind;              */
/*       0 <= p < Anzahl der Bits von a)        */
/*   n = Laenge des Bit-Feldes                  */
/*================================================*/
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
  unsigned int a = 0x77;  /*      ... 0111 0111 */
  int p = 5, n = 3;       /*            11 0    */

  printf("%d\n",
    (a>>(p-n+1))      /* Bit-Feld an niederwer-  */
                      /*   tiges Ende schieben   */
      & (~((~0)<<n))  /* Bitfolge mit n Einsen   */
                      /*   am niederwertigen     */
    );                /*   Ende                  */
  exit(EXIT_SUCCESS);
}
```

Vergleichsoperatoren

Operator	Vergleich
$a < b$	kleiner
$a > b$	größer
$a <= b$	kleiner gleich
$a >= b$	größer gleich
$a == b$	gleich
a **!=** b	ungleich

- Resultatwert: 0 oder 1
- Resultattyp: **int**
- falls *a* und *b* einen arithmetischen Typ haben, werden die arithmetischen Umwandlungen durchgeführt

Logische Operatoren

Operator	Operation
a **&&** b	logische UND-Verknüpfung; ist *a* falsch (*a* == 0), wird *b* *nicht* bewertet ($\Rightarrow$ in *b* enthaltene Seiteneffekte treten dann nicht auf)
$a \mid\mid b$	logische ODER-Verknüpfung; ist *a* wahr (*a* != 0), wird *b* *nicht* bewertet
! a	logische Umkehrung (Negation; $\Leftrightarrow$ *a* == 0)

- Resultatwert: 0 oder 1
- Resultattyp: **int**
- für die Operanden *a* und *b* gilt:
  a. jeder Operand muß einen arithmetischen Typ haben oder ein Zeiger sein
  b. der Typ der Operanden braucht nicht übereinzustimmen

Beispiel:

```
( i<MAX ) && ( (buf[i]=getchar()) != EOF )
       /* Falls i>=MAX, wird kein Zeichen     */
       /*   von der Standard-Eingabe gelesen! */
```


Weitere Zuweisungsoperatoren

Syntax:

lvalue *operator*= *expr*

operator

steht für einen der folgenden Operatoren:
+ - * / % << >> & | ^

operator=

ist *ein* Token (, d.h. zwischen *operator* und = darf kein Zwischenraumzeichen stehen)

lvalue *operator*= *expr*

- entspricht der Zuweisung:
  lvalue = *lvalue* *operator* (*expr*)
- einziger Unterschied: *lvalue* wird nur einmal bewertet

Beispiele:

```
j *= 3+k;      ⇔    j = j*(3+k);

a[++i] += 3;  ⇎    a[++i] = a[++i]+3;
/* Unterschied:                                */
/*    links wird i um 1, rechts um 2 erhoeht */
```


Bedingungsoperator
(*conditional operator*)

ternärer[14] Operator (d.h. Operator benötigt 3 Operanden)

Syntax:

$$logical_expr \; \mathbf{?} \; expr_1 \; \mathbf{:} \; expr_2$$

- *logical_expr* entscheidet über den Resultatwert:
  - *logical_expr* **!=** 0 (wahr):

    Wert von $expr_1$ ist Resultatwert ($expr_2$ wird *nicht* bewertet)
  - *logical_expr* == 0 (falsch):

    Wert von $expr_2$ ist Resultatwert ($expr_1$ wird *nicht* bewertet)
- Resultattyp: der gemeinsame Typ von $expr_1$ und $expr_2$ nach den arithmetischen Umwandlungen (sofern diese beiden Operanden einen arithmetischen Typ haben)
- Resultat ist kein l-Wert

[14] ternär = dreiwertig

Beispiele:

```
int a, b, max, n;


max = (a>b) ? a : b;


/* folgendes Programmstueck laesst sich mit  */
/* Hilfe des Bedingungoperators kompakter    */
/* formulieren:                              */

printf("Element gefunden: ");
if ( n == 1 )
  printf("%d Suchschritt\n", n);
else
  printf("%d Suchschritte\n", n);


printf("Element gefunden: %d Suchschritt%s\n",
  n, (n == 1) ? "" : "e");
```


Komma-Operator

Syntax:

$$expr_1 \; , \; expr_2$$

- $expr_1$ wird zuerst bewertet ($\Rightarrow$ in $expr_1$ enthaltene Seiteneffekte sind aufgetreten, wenn $expr_2$ bewertet wird)
- Resultatwert: Wert von $expr_2$ (Der Resultatwert von $expr_1$ wird nicht weiterverwendet.)
- Resultattyp: Typ von $expr_2$

Beispiel:[15]

```
int i = 1, j = 2, k;

k = (i++, 2*j);
/* k wird der Wert 4 zugewiesen */
```

[15] sinnvolle Verwendung → **for**–Anweisung

Analyse von Ausdrücken

Vorrang und Zuordnung der Operatoren

Ebene	Kategorie	Operatoren	Zuordnung
1	primär	`() [] -> .`	links→rechts
2	unär	`! ~ ++ -- + -` `* & (`*type*`) sizeof`	rechts→links
3	multiplikativ	`* / %`	links→rechts
4	additiv	`+ -`	links→rechts
5	verschiebend	`<< >>`	links→rechts
6	relational	`< <= > >=`	links→rechts
7	gleich	`== !=`	links→rechts
8	bitweise	`&`	links→rechts
9	bitweise	`^`	links→rechts
10	bitweise	`|`	links→rechts
11	logisch	`&&`	links→rechts
12	logisch	`||`	links→rechts
13	konditional	`?:`	rechts→links
14	zuweisend	`= += -= *= /= %=` `&= ^= |= <<= >>=`	rechts→links
15	Komma	`,`	links→rechts

Basistypen, Operatoren und Ausdrücke
Analyse von Ausdrücken

Ziel der Analyse eines Ausdrucks ist es, die Operanden für jeden Operator zu identifizieren.

Beispiele:

```
            a & b << 2
/* Vorrang:   8    5      */
            a &(b << 2)

            k % 3 == 0 ? i : x + 1
/* Vorrang:   3    7 13  13    4     */
           (k % 3)== 0 ? i : x + 1
           (k % 3)== 0 ? i :(x + 1)
          ((k % 3)== 0)? i :(x + 1)
```

Operatoren mit gleichem Vorrang werden wie in der Tabelle angegeben zugeordnet:

Zuordnung von *links→rechts*

bedeutet, daß der im Ausdruck am weitesten links stehende Operator als erster seine Operanden erhält.

Beispiele:

```
            -i++
/* Vorrang: 2  2              */
/* Zuordnung: rechts -> links */
            -(i++)
```

```
            a = b = c = 0;
/* Vorrang:  14  14  14       */
/* Zuordnung: rechts -> links */
            a = b =(c = 0);
            a =(b =(c = 0));
```

Falls eine Zeichenfolge auf unterschiedliche Weise in Operatoren zerlegt werden kann, wird zunächst immer der Operator gewählt, der aus den meisten Zeichen besteht.

Beispiele:

`a+++b` ⇔ `a++ + b`

`a+++++b` ⇔ `a++ ++ + b /* unzulaessig */`

Bewertung von Ausdrücken

Reihenfolge der Bewertung ist, mit wenigen Ausnahmen, undefiniert.

⇒ Reihenfolge, in der Seiteneffekte auftreten, ist undefiniert. (I. allg. ist erst *nach* vollständiger Bewertung eines Ausdrucks garantiert, daß die Seiteneffekte des Ausdrucks aufgetreten sind.)

Beispiele:

```
int a[10], i = 7;
  ...
a[i] = i++;      /* Reihenfolge, in der Ope-   */
                 /*   randen bewertet werden,  */
                 /*   undefiniert              */

int c;
  ...
(c << 8) +            /* undefiniert (Absicht  */
  (c = getchar())     /*   war, 2 Zeichen in   */
                      /*   16 Bit zu packen)   */

int n = 3;
  ...
printf("%d %d\n", ++n, power(2,n));
                 /* Reihenfolge, in der Argu-  */
                 /*   mente bewertet werden,   */
                 /*   undefiniert              */
```

```
int i = 7;
  ...
i++ * i++        /* kann statt 56 auch 49 er-  */
                 /*   geben (undefiniert)      */

int a, b, c;
  ...
a + (b + c)      /* darf vom Compiler bewertet */
                 /*   werden wie (a + b) + c   */

double a, b, c;
  ...
a + (b + c)      /* darf vom Compiler nicht    */
                 /*   bewertet werden wie      */
                 /*   (a + b) + c, da die      */
                 /*   beiden Ausdruecke z.B.   */
                 /*   fuer die Werte           */
                 /*     a = DBL_EPSILON / 2.0; */
                 /*     b = 1.0;               */
                 /*     c = -1.0;              */
                 /*   unterschiedliche         */
                 /*   Resultatwerte liefern    */
```


Operatoren mit definierter Bewertungsreihenfolge:

&& || ?: ,

Bei diesen Operatoren wird jeweils der linke Operand als erster bewertet.

Beispiele:

```
while ( ((c=getchar()) != EOF) && (c != '\n') )
  ...

n = 3;
k = (++n, power(2,n));
                /* beim Aufruf der Funktion   */
                /*   power hat n den Wert 4   */
```

Unterscheidung zwischen Komma-Operator und Komma als Punktsymbol:

> Ein Komma außerhalb einer Vereinbarungs- oder Argumentliste dient als Operator.

Ausnahme:

```
f(a,(t=3,t+2),c);    /* an f werden 3 Argu-   */
                     /*   mente uebergeben:   */
                     /*      a, 5, c          */
```

Konstante Ausdrücke mit integralem Typ

Verwendung:

- Dimensionslänge(n) bei der Vereinbarung eines Feldes
- als Wert einer Aufzählungskonstanten
- als Wert einer **case**-Marke
- Länge eines Bit-Feldes

Einschränkungen[16]:

Ein konstanter Ausdruck darf keine der Operatoren

```
=   ++   --   ,
```

und keine Funktionsaufrufe enthalten. **cast**-Operatoren müssen einen integralen Resultattyp spezifizieren.

zulässige Operanden sind:

- Integer-Konstanten
- Aufzählungskonstanten
- Character-Konstanten (Zeichenkonstanten)
- **sizeof**-Ausdrücke
- Gleitkommakonstanten als Operanden einer **cast**-Operation

[16] Die Einschränkungen gelten nicht innerhalb eines Operanden von **sizeof**.

Aufgaben

1. Geben Sie einen Ausdruck an, der das Bit mit dem Wert 2^3 in einer Variablen vom Typ **int** löscht.
2. Ersetzen Sie den Ausdruck

   ```
   (i << 3) + (i << 2)
   ```

   durch einen gleichwertigen (möglichst kurzen) Ausdruck, in dem ausschließlich arithmetische Operatoren verwendet werden.
3. Was passiert, wenn beim Kodieren einer Abfrage auf Ungleichheit, die Zeichen des Operators **!=** versehentlich vertauscht werden, also statt `a != b` der Ausdruck `a =! b` kodiert wird?
4. Warum läßt sich folgendes Programm u.U. nicht übersetzen?

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #define MINUS_EINS -1

   int main( void )
   {
     int i = 7;

     printf("%d\n", i-MINUS_EINS);
     exit(EXIT_SUCCESS);
   }
   ```

   Korrigieren Sie das Programm.

5. Bestimmen Sie die Resultatwerte der folgenden Ausdrücke. Geben Sie außerdem immer den Typ des Resultats und die Zwischenschritte bei der Bewertung an.

   Die folgenden Vereinbarungen werden vorausgesetzt:

```
short int  s=10;
int        i, j=25, n=1, a=5, b=2, c=4;
long  int  k=3L;
float      x, f=0.5f;
double     d=3.2, y=2.3, z=0.0;
```

   a. `x = d * (i = ((int) 2.9 + 1.1) / d)`

   b. `1 + 3 * (n += s << 1) / 5`

   c. `++ a / b ++ * -- c`

   d. `z != y && j + 1 == ! k + 26`

   e. `(double) j / s * f`

   f. `k % 3 == 0 ? j : x + 1`

   g. `~~++j^s`

   h. `k += - - div(7,3).rem << 2`

Mathematische Funktionen

Trigonometrische Funktionen

#include <math.h>	
double sin(double *x*);	$\sin(x)$, x in Bogenmaß
double cos(double *x*);	$\cos(x)$, x in Bogenmaß
double tan(double *x*);	$\tan(x)$, x in Bogenmaß
double asin(double *x*);	$\arcsin(x)$ im Bereich $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, $x \in [-1, 1]$
double acos(double *x*);	$\arccos(x)$ im Bereich $[0, \pi]$, $x \in [-1, 1]$
double atan(double *x*);	$\arctan(x)$ im Bereich $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$
double atan2(double *y*, double *x*);	$\arctan\left(\frac{y}{x}\right)$ im Bereich $[-\pi, \pi]$

Anmerkung: **atan2** erlaubt die Berechnung des Arkustangens auch für sehr große Quotienten *y/x*, die als **double**-Wert nicht mehr darstellbar sind. Das Argument *x* darf 0 sein, falls $y \neq 0$ gilt.

Hyperbolische Funktionen

#include <math.h>

double sinh(double *x***);**	Sinus Hyperbolicus
double cosh(double *x***);**	Cosinus Hyperbolicus
double tanh(double *x***);**	Tangens Hyperbolicus

Exponential– und Logarithmusfunktionen

#include <math.h>

double exp(double *x***);**	Exponentialfunktion e^x
double log(double *x***);**	natürlicher Logarithmus $\ln(x)$, $x > 0$
double log10(double *x***);**	Logarithmus zur Basis 10 $\log_{10}(x)$, $x > 0$

#include <math.h>

double frexp(double *value***, int** **exp***);**

$$value = m \cdot 2^{exponent}$$

Spaltet das Argument *value* in eine normalisierte Mantisse *m* (Resultatwert) und einen Exponenten auf. Für *m* gilt: $0.5 \leq m < 1$. Der Exponent ist eine ganzzahlige Potenz von 2 und wird in der Variablen, auf die der Zeiger *exp* verweist, abgespeichert.

Basistypen, Operatoren und Ausdrücke
Mathematische Funktionen

Beispiel:

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

int main( void )
{
  double mantisse;
  int    iexp;

  mantisse = frexp(0.25, &iexp);
  printf(
    "value = 0.25, mantisse = %f, iexp = %d\n",
    mantisse, iexp);
  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
value = 0.25, mantisse = 0.500000, iexp = -1
```

```
#include <math.h>
double ldexp(double x, int exp);        x · 2^exp
```

$x \cdot 2^{exp}$

```
#include <math.h>
double modf(double value, double *iptr);
```

Spaltet *value* in einen ganzzahligen Teil, der in der Variablen, auf die *iptr* zeigt, abgelegt wird, und einen Nachkommateil (Resultatwert) auf. (Beide Teile haben das gleiche Vorzeichen.)

Potenzfunktionen

```
#include <math.h>
double pow(double x, double y);
double sqrt(double x);
```

Funktion	Ergebnis
double pow(double *x*, **double** *y*);	x^y
double sqrt(double *x*);	$+\sqrt{x}, x \geq 0$

Sonstige mathematische Funktionen

#include <math.h>

Funktion	Ergebnis
double ceil(double *x*);	nächstgrößerer ganzzahliger Wert oder *x*, falls *x* ganzzahlig
double floor(double *x*);	nächstkleinerer ganzzahliger Wert oder *x*, falls *x* ganzzahlig
double fabs(double *x*);	$\|x\|$
double fmod(double *x*, **double** *y*);	Gleitkomma-Divisionsrest von $\frac{x}{y}$

Anmerkung: Der Resultatwert von **fmod** hat das gleiche Vorzeichen wie das Argument *x*.

#include <stdlib.h>

Funktion	Ergebnis
int abs(int *j*);	$\|j\|$
long int labs(long int *j*);	$\|j\|$
div_t div(int *j*, **int** *k*);	Quotient und Divisionsrest von $\frac{j}{k}$
ldiv_t ldiv(long int *j*, **long int** *k*);	Quotient und Divisionsrest von $\frac{j}{k}$

Behandlung von Fehlern durch Bibliotheksfunktionen

Bestimmte Bibliotheksfunktionen signalisieren der aufrufenden Funktion einen Fehler, indem sie einen Fehlercode an die globale Variable **errno** (definiert in **<errno.h>**) zuweisen. Die Bedeutung der einzelnen Fehlercodes ist durch die Implementierung festgelegt.

- **errno** hat zu Beginn der Programmausführung den Wert 0
- das Zurücksetzen nach einem Fehler ist Aufgabe des Programmierers.

Für zwei Fehlercodes sind in **<math.h>** Makros definiert:

ERANGE zeigt an, daß der Resultatwert einer mathematischen Funktion nicht als **double**-Wert dargestellt werden kann (*range error*).

Im Falle eines *Overflows* wird anstelle des korrekten Resultatwertes der Wert des Makros **HUGE_VAL**[17] (mit dem Vorzeichen des korrekten Resultatwertes) zurückgegeben.

Ob auch ein *Underflow* als *range error* angezeigt wird, hängt von der Implementierung ab.

Beispiele: `exp(x)` und `x` zu groß
`log(0.0)` und $-\infty$ nicht darstellbar

EDOM zeigt an, daß ein Argumentwert für eine mathematische Funktion außerhalb des zulässigen Wertebereichs liegt (*domain error*). Resultatwert: implementierungsabhängig

Beispiel: `log(-2.0)`

Die Implementierung kann für eigene Fehlercodes zusätzliche Makros definieren.

[17] **HUGE_VAL** ist in **<math.h>** als positiver **double**-Ausdruck – häufig ist es der größte darstellbare **double**-Wert – definiert.

Umsetzen eines Fehlercodes in eine Fehlermeldung:

```
#include <stdio.h>
void perror(const char *string);
```

perror schreibt die Fehlermeldung, die dem Wert von **errno** entspricht, auf die Standardfehlerausgabe (**stderr**). Der Meldung geht die Zeichenkette *string* (gefolgt von einem Doppelpunkt) voraus.

Beispiele:

```
perror("Error");  /* Ausgabe: "Error: ... "  */
perror(NULL);     /* Meldung ohne Praefix    */
```

```
#include <string.h>
char *strerror(int error);
```

strerror liefert einen Zeiger auf die Fehlermeldung, die dem Fehlercode *error* entspricht.

Beispiel:

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
```

```
int main( void )
{
  int save;

  /* garantiert falsch */
  sqrt(-1.0);
  save = errno;
  printf("What does sqrt say to -1?");
  if (save)
    printf(": %s", strerror(save));
  printf("\n");
  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
What does sqrt say to -1?: Domain error
```

Empfehlung für das Kodieren von **errno**-Abfragen:

Bevor in einem Programm der Wert von **errno** überprüft wird, sollte man sich die folgenden Fragen stellen:

1. Können die Argumentwerte nicht bereits ***vor*** dem Aufruf einer Bibliotheksfunktion getestet werden?
2. Kann eine Fehlersituation nicht ebenso einfach durch Überprüfen des Resultatwertes festgestellt werden?

Anweisungen
(*statements*)

Übersicht

- leere Anweisung (*null statement*)
- Ausdrucksanweisung (*expression statement*)
- gelabelte Anweisungen (*labeled statements*)
  - **case**
  - **default**
- Verbundanweisung (*compound statement*)
- bedingte Anweisungen (*selection statements*)
  - **if**
  - **else**
  - **switch**
- Wiederholungsanweisungen (*iteration statements*)
  - **while**
  - **do**
  - **for**
- Sprunganweisungen (*jump statements*)
  - **break**
  - **continue**
  - **return**
  - **goto**

leere Anweisung

Anweisung, die nur aus einem Semikolon (**;**) besteht.

Beispiel:

```
/* Suche das naechste Zeichen, das nicht  */
/* Zwischenraumzeichen ist:               */
 while ( isspace(c = getchar()) )
   ;
```

Ausdrucksanweisung

Syntax:

*expr***;**

- *expr* wird berechnet.
- Wert von *expr* wird ignoriert.
- Ausdrucksanweisung wird benutzt, um die Seiteneffekte von *expr* zu erzeugen.

Beispiele:

```
x = 0;
i++;
printf(...); /* Der Funktionswert (Anzahl    */
             /*   der geschriebenen Zeichen) */
             /*   wird ignoriert.            */
```


gelabelte Anweisungen

Syntax:

*identifier***:** *statement*

Jeder C-Anweisung kann eine Marke (Label) vorangestellt werden, damit die Anweisung als Sprungziel für eine **goto**-Anweisung erreichbar wird. Die speziellen Marken

- **case** *const_expr*
- **default**

können nur im Zusammenhang mit der **switch**-Anweisung benutzt werden.

Verbundanweisung

Syntax:

$\{declaration_list_{opt}\ \ statement_list_{opt}\}$

Die Verbundanweisung (oft auch als Block bezeichnet) dient dazu mehrere Anweisungen syntaktisch zu einer einzigen Anweisung zusammenzufassen.

bedingte Anweisungen

if-Anweisung

Syntax:

1. **if** (*condition*)
   statement

2. **if** (*condition*)
   $statement_1$
   else
   $statement_2$

Anmerkungen:

- *condition* bezeichnet einen Ausdruck mit arithmetischem Typ oder einem Zeigertyp. Der Resultatwert des Ausdrucks wird als logischer Wert interpretiert. Daher kann eine Anweisung der Form

  ```
  if (expr != 0) ...
  ```

  einfach verkürzt werden zu

  ```
  if (expr) ...
  ```

- **else** wird immer dem letzten **if** innerhalb desselben Blocks zugeordnet.

Beispiele:

```
if ( j >= 0 )
  if ( j == 0 )
    k = 0;
  else
    k = 1;  /* j > 0 */

if ( j >= 0 )
{
  if ( j == 0 )
    k = 0;
}
else
  k = -1;  /* j < 0 */
```

Mehrfachauswahl durch Schachtelung:

if ($condition_1$)
   $statement_1$
else if ($condition_2$)
   $statement_2$
else if ($condition_3$)
   $statement_3$
 . . .
else
   $statement_n$

switch-Anweisung

Syntax:

switch (*expr*) *statement*

- *expr* muß einen Integer-Typ haben.
- für *expr* wird IE durchgeführt.

statement hat spezielle Form:

{
  case $const_expr_1$**:**
    $statement_{1.1}$**;**
    $statement_{1.2}$**;**
    . . .
    $statement_{1.i}$**;**
    break;
  case $const_expr_2$**:**
    $statement_{2.1}$**;**
    . . .
    $statement_{2.j}$**;**
    break;
  . . .
  default: $statement_{k.1}$**;**
    . . .
    $statement_{k.l}$**;**
    break;
}

- $const_expr_i$ ist ein ganzzahliger konstanter Ausdruck, der in den Typ von *expr* nach IE umgewandelt wird.
- innerhalb einer **switch**-Anweisung darf nur eine einzige **case**-Marke den Wert $const_expr_i$ haben.

Nach Auswertung von *expr* können 3 Fälle auftreten:

1. Der Wert von *expr* stimmt mit dem Wert einer **case**-Marke überein. In diesem Fall wird die Anweisung mit dieser Marke als nächstes ausgeführt.
2. Hat keine **case**-Marke den Wert von *expr*, wird die Anweisung mit der **default**-Marke als nächstes ausgeführt.
3. Trifft Fall 2 zu und es ist keine **default**-Marke vorhanden, dann wird die Anweisung als nächstes ausgeführt, die der von **switch** abhängigen Anweisung folgt.

Beispiel:

siehe Programm „Zählen von Zeichen“

Wiederholungsanweisungen

while-Schleife

Syntax:

> **while (** *condition* **)** *statement*

condition: siehe **if**-Anweisung

do-Schleife

Syntax:

> **do**
> *statement*
> **while (** *condition* **);**

condition: siehe **if**-Anweisung

Beispiel:

```
do
{
  printf("Weiter? (j/n)\n");
  c = getchar();
  while ( getchar() != ’\n’ )
    ;
} while( ((c=tolower(c)) != ’j’) && (c != ’n’) );
```


for-Schleife

Syntax:

> **for (** $expr1_{opt}$ **;** $condition_{opt}$ **;** $expr2_{opt}$ **)** *statement*

condition:

- Standardvorgabe: 1
- siehe **if**-Anweisung

Falls *statement* keine **continue**-Anweisung enthält, entspricht dies:

> $expr_1$**;** /* Initialisierung der Schleife */
> **while (** *condition* **)**
> **{**
> *statement*
> $expr_2$**;** /* Reinitialisierung */
> **}**

Beispiele:

```
for ( summe = 0, i = 1 ; i <= MAX ; i++ )
  summe += i;

for ( c = 0, zeile = 1  ; c < 256 ; c++,
      (zeile < PAGELEN) ? zeile++ : (zeile = 1) )
  ...
```

Beispielprogramm: Zählen von Zeichen

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

int main( void ) /* zaehlt Ziffern , Zwischen- */
{                /*   raumzeichen und sonstige */
  int c, i, nwhite, nother, ndigit[10];

  nwhite = nother = 0;
  for (i = 0; i < 10; i++)
    ndigit[i] = 0;
  while ((c = getchar()) != EOF) {
    switch (c) {
      case '0': case '1': case '2': case '3':
      case '4': case '5': case '6': case '7':
      case '8': case '9':
                 ndigit[c-'0']++;
                 break;
      case ' ':
      case '\n':
      case '\t': nwhite++;
                 break;
      default:   nother++;
                 break;
    }
  }
  printf("digits = ");
  for (i = 0; i < 10; i++)
    printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n",
    nwhite, nother);
  exit(EXIT_SUCCESS);
}
```


© Forschungszentrum Jülich

Sprunganweisungen

return-Anweisung

Syntax:

> **return** *expr*$_{opt}$**;**

- gibt die Kontrolle an die rufende Funktion zurück.
- der Wert von *expr* wird als Resultatwert zurückgegeben (zuvor wird er in den Resultattyp der Funktion umgewandelt).
- das Erreichen der abschließenden geschweiften Klammer in einem Funktionsblock hat die gleiche Wirkung wie die Anweisung `return;`

break-Anweisung

Syntax:

> **break;**

- ist nur innerhalb einer Wiederholungsanweisung (**while**-, **do**- oder **for**-Schleife) oder innerhalb einer **switch**-Anweisung zulässig.
- die Ausführung wird mit der Anweisung fortgesetzt, die der Wiederholungsanweisung bzw. **switch**-Anweisung folgt, welche die **break**-Anweisung unmittelbar enthält.

© Forschungszentrum Jülich

continue-Anweisung

Syntax:

continue;

- ist nur innerhalb einer Wiederholungsanweisung (**while**-, **do**- oder **for**-Schleife) zulässig.
- bricht die aktuelle Iteration ab, indem zum Ende des Schleifenkörpers verzweigt wird.

Beispiel:

```
while ( ... )
{
   ...
  continue;   /* goto loop_end; */
   ...
/* loop_end:; */}
```

unbedingte Sprunganweisung

Syntax:

goto *label***;**

Beispielprogramm: Fallgeschwindigkeit

```
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double fTime(double meters)
{
  double time;

  errno = 0;
  time = sqrt(meters * 2 / 9.8);
  /*
   * es waere einfacher zuerst auf (meters < 0)
   * abzufragen, aber so wird demonstriert, wie
   * sqrt() errno setzt
   */
  if (errno)
  {
    printf("Sorry, but you can't fall up\n");
    time = HUGE_VAL;
  }
  return(time);
}
```

```
int main( void )
{
  double height;
  int    count;

  for(;;)
  {
    printf("Enter height in meters\n");
    if ((count = scanf("%lf", &height)) == EOF)
      break;

    if (count != 1)
    {
      printf("invalid floating-point number\n");
      while(getchar() != '\n')
        ;
      continue;
    }

    printf("It takes %3.2f sec. to fall %3.2f "
      "meters\n", fTime(height), height);
  }
  exit(EXIT_SUCCESS);
}
```


Aufgabe

Welchen Wert hat die Variable `sum` nach Ausführung des folgenden Programmabschnitts?

```
int i, sum = 0;

for ( i = 0 ; i < 10 ; i++ )
{
  switch(i)
  {
    case 0:
    case 1:
    case 3:
    case 5:
      sum++;
    default:
      continue;
    case 4:
      break;
  }
  break;
}
```

Funktionen und Programmstruktur

Vereinbarungen

In C werden zwei Arten von Vereinbarungen unterschieden:

Definition — **erzeugt** das vereinbarte Objekt (, d.h. eine Vereinbarung, die Definition ist, veranlaßt den Compiler, **Speicherplatz** für dieses Objekt anzulegen.)

Deklaration — **beschreibt** die Eigenschaften eines an anderer Stelle erzeugten Objekts und hat allein den Zweck, die Verbindung zwischen Bezeichner und Objekt herzustellen.

Beispiele:

```
double sqr(double x)    /* Funktionsdefinition */
{
  return(x*x);
}

int main( void )
{
  double val = 3.14;
  double sqr(double x);/* Funktionsdeklaration */

  printf("%f\n", sqr(val));
  exit(EXIT_SUCCESS);
}
```


Funktionsdeklaration

hat meist die Form:

result_type declarator(*parameter_type_list*)**;**

result_type
vereinbart den Typ des Resultatwertes der Funktion. Zulässige Typen sind:

- arithmetische Typen
- Strukturtypen
- Vereinigungstypen
- Zeigertypen
- **void** (für Funktionen, die **keinen** Resultatwert liefern)

declarator
bezeichnet das Objekt, das als „Funktion mit den Argumenten *parameter_type_list* und dem Resultattyp *result_type*“ vereinbart wird.

parameter_type_list

- Liste von (durch Komma getrennten) Vereinbarungen
- vereinbart die Typen der Parameter
- Parameternamen dürfen weggelassen werden(, d.h. Parameternamen sind effektiv Kommentare)
- Falls *declarator* als „Funktion **ohne** Argumente“ vereinbart werden soll, muß anstelle der Liste das Schlüsselwort **void** kodiert werden.

Eine Funktionsdeklaration dieser Art heißt **Funktionsprototyp**.

Funktionsdeklaration im alten (K&R-C) Stil:

result_type declarator **();**

vereinbart *declarator* als

„Funktion mit **unbekannten** Argumenten und Resultattyp *result_type*".

implizite Funktionsdeklaration:

Folgt einem Bezeichner, der noch nicht vereinbart wurde, das Zeichen `(`, dann wird dieser Bezeichner **implizit vereinbart** als

„Funktion mit unbekannten Argumenten und Resultattyp **int**".

Funktionsdefinition

Funktionsdefinition im neuen (ANSI C) Stil

Syntax:

result_type$_{opt}$ *declarator* **(** *parameter_type_list* **)**
block

result_type
: Standardvorgabe: **int**

declarator
: Funktionsname (Dieser darf in Klammern eingeschlossen sein.)

parameter_type_list
- Liste von (durch Komma getrennten) Parametervereinbarungen
- Parameternamen dürfen **nicht** weggelassen werden

Beispiel:

```
int max(int a, int b, int c)
{
  int m;

  m = (a>b) ? a : b;
  return( (m>c) ? m : c );
}
```

Funktionsdefinition im alten (K&R-C) Stil

Syntax:

> $\textit{result_type}_{opt}$ $\textit{declarator}$**(**$\textit{identifier_list}_{opt}$**)**
> $\textit{declaration_list}_{opt}$
> *block*

result_type
: Standardvorgabe: **int**

declarator
: Funktionsname (Dieser darf in Klammern eingeschlossen sein.)

identifier_list

- Liste von (durch Komma getrennten) Bezeichnern
- benennt die Parameter
- Wird *identifier_list* weggelassen, dann ist *declarator* eine „Funktion **ohne** Argumente“.

declaration_list

- vereinbart die Datentypen der (in der *identifier_list* angegebenen) Parameter
- Falls die *declaration_list* für einen Parameter keine Vereinbarung enthält, wird für diesen Parameter der Typ **int** angenommen.

Beispiele:

```
int max(a, b, c)
int a, b, c;    /* Deklarationsliste für die    */
                /*   Parameter                  */
{
  /* ... */
}

max(a, b, c)
{
  /* ... */
}
```

Funktionsaufruf

Syntax:

function_expr(*argument_list*$_{opt}$)

function_expr
: Ausdruck vom Typ „Funktion mit Resultattyp T“ (z.B. Funktionsname) oder vom Typ „Zeiger auf Funktion, die T liefert“

argument_list
: Liste von (durch Komma getrennten) Ausdrücken (Funktionsargumente)

Übergabeart: „*call by value*“ (Wertübergabe)

Von jedem Argument wird eine Kopie (Parameterobjekt) erzeugt.

Man kann jedoch einen Zeiger in der Absicht übergeben, daß die Funktion das Objekt ändert, auf das der Zeiger verweist.

implizite Umwandlung von Argumenten

- Für einen Funktionsaufruf im Gültigkeitsbereich eines Funktionsprototypen (Funktionsdeklaration im neuen Stil) gilt:

  Argumente werden wie bei einer Zuweisung in die Typen der zugehörigen Parameter umgewandelt.

- Für einen Funktionsaufruf im Gültigkeitsbereich einer Funktionsdeklaration im alten Stil gilt:

  Standarderweiterung für jedes Argument:

  - für Integer-Argumente: Integer-Erweiterung
  - **float→double**

  Der erweiterte Typ des Argumentes muß dem Typ des zugehörigen Parameters entsprechen:

  - Funktions**definition** im alten Stil:

    erweiterter Typ des Argumentes == erweiterter Typ des Parameters

  - Funktions**definition** im neuen Stil:

    erweiterter Typ des Argumentes == Typ des Parameters(, der in diesem Fall nicht erweitert wird)

top-level-Vereinbarungen

- Variablen und Typen können außerhalb von allen Funktionen vereinbart werden.
- Funktionen müssen immer auf top-level **definiert** werden.

Informationsaustausch zwischen Funktionen:

1. mit Hilfe von Argumenten und Resultatwerten
2. über globale Variablen (top-level-Variablen)

Gültigkeitsbereich innerhalb der Quelldatei (*lexical scope*)

gibt an, wo innerhalb der Quelldatei die Vereinbarung eines Bezeichners bekannt ist.

Der Gültigkeitsbereich

1. für einen **auf top-level** vereinbarten Bezeichner reicht vom Punkt der Vereinbarung bis zum Ende der Quelldatei.
2. für einen **am Anfang eines Blockes** vereinbarten Bezeichner reicht vom Punkt der Vereinbarung bis zum Ende des Blocks.
3. für einen formalen **Parameter** reicht
   a. in einer Funktions**definition** vom Punkt der Vereinbarung bis zum Ende des Funktionsblocks.
   b. in einer Funktions**deklaration** vom Punkt der Vereinbarung bis zum Ende des Funktionsprototyps.
4. einer **Anweisungsmarke** ist die gesamte Funktion, welche die Definition der Marke enthält.
5. für einen **Makronamen** beginnt mit der auf die **#define**-Direktive folgenden Zeile und reicht bis zum Ende der Quelldatei oder bis zur ersten **#undef**-Direktive, die die Definition dieses Makros aufhebt.

```
#define EOS '\0'

int global_i;

double sqr(double x);

int example(int a, int b)
{
  int m = 10, n = m;

    /* ... */
  if (m == 0)
  {
    int n = m, m;

      /* ... */
   n:
      /* ... */
  }
  else
  {
    int global_i;

      /* ... */
  }
 label:
    /* ... */
}

enum Farben {rot, gruen, blau, pink=rot};

double sqr(double x)
{
    /* ... */
}
```



```
#define EOS '\0'

int global_i;

double sqr(double x);

int example(int a, int b)
{
  int m = 10, n = m;

    /* ... */
  if (m == 0)
  {
    int n = m, m;

      /* ... */
   n:
      /* ... */
  }
  else
  {
    int global_i;

      /* ... */
  }
 label:
    /* ... */
}

enum Farben {rot, gruen, blau, pink=rot};

double sqr(double x)
{
    /* ... */
}
```

```
#define EOS '\0'

int global_i;

double sqr(double x);

int example(int a, int b)
{
  int m = 10, n = m;

    /* ... */
  if (m == 0)
  {
    int n = m, m;

      /* ... */
   n:
      /* ... */
  }
  else
  {
    int global_i;

      /* ... */
  }
 label:
    /* ... */
}

enum Farben {rot, gruen, blau, pink=rot};

double sqr(double x)
{
    /* ... */
}
```

label

n (Marke)

rot

x

Namensräume (*name spaces*)

1. Präprozessor-Makronamen
2. Anweisungsmarken
3. Namen (*tags*) von Strukturen, Vereinigungen und Aufzählungen
4. Komponenten von **jeder einzelnen** Struktur und Vereinigung
5. alle anderen Bezeichner (Variablen, Funktionen, **typedef**-Namen, Aufzählungskonstanten)

Speicherklassen

Konvention:

Speicherklassenangabe immer an erster Stelle in einer Vereinbarung

auto

- nur in Definitionen von Variablen am Anfang eines Blocks
- Standardvorgabe (wird meist weggelassen)
- **auto**-Objekt wird bei jedem Blockeintritt erzeugt (und initialisiert)
- bei Verlassen des Blocks wird der Speicherplatz für **auto**-Objekte freigegeben

⇒ rekursive Funktionen sind möglich

register

- Hinweis für Compiler, daß auf das Objekt häufig zugegriffen wird
- **register** ⇔ **auto**
  (jedoch Adressoperator **&** nicht erlaubt)
- einzige Speicherklasse, die für Parameter angegeben werden kann

static

- in Vereinbarungen innerhalb eines Blocks und auf top-level zulässig (auch für Funktionen)
- statische Speicherallokation
- **static**-Variablen werden nur einmal vor Beginn der Programmausführung initialisiert

Beispiel: Ein Pseudo-Zufallszahlen-Generator

```
#define FAKTOR 25173u
#define MODULO 65536u
#define INKREMENT 13849u
#define INIT_SAAT 17u

unsigned short int random( void )
{
  static unsigned short int saat = INIT_SAAT;

  saat = (FAKTOR*saat + INKREMENT) % MODULO;
  return( saat );
}
```

extern

- in top-level-Vereinbarungen und in **Deklarationen** am Anfang eines Blockes
- statische Speicherallokation

top-level-Variablen haben immer statische Speicherallokation, auch wenn keine Speicherklasse angegeben ist.

top-level- und **static**-Variablen, die nicht explizit initialisiert werden, haben bei Beginn der Programmausführung den Wert 0.

(**auto**- und **register**-Variablen sind undefiniert.)

Definition und Deklaration von top-level-Variablen

Speicher-klassen-angabe	**Vereinbarung auf top-level** mit Initialisierung	ohne Initialisierung	**Vereinbarung innerhalb eines Blocks**
—	**Definition**	vorläufige Definition (*tentative definition*)	Definition eines Objektes mit der Speicherklasse **auto**
extern	Definition	**Deklaration**	Deklaration, falls keine Initialisierung; andernfalls unzulässig
static	**Definition**	vorläufige Definition	Definition eines lokalen Objekts

Enthält eine Quelldatei für eine Variable nur vorläufige Definitionen, aber keine Definition, dann werden alle vorläufigen Definitionen zu einer einzigen Definition mit der Initialisierung 0.

Erscheint irgendwo in der aktuellen Quelldatei eine Definition, dann werden die vorläufigen Definitionen als Deklarationen behandelt.

Beispiel:

```
 ...
static int i;    /* vorläufige Definition      */
 ...
extern int i=25; /* Definition                 */
                 /* => vorläufige Definition   */
                 /*    wird  als Deklaration   */
                 /*    behandelt               */
```


Bindung

nur für auf top-level **definierte** Objekte wichtig (innerhalb von Blöcken **definierte** Objekte sind **ohne** Bindung)

interne Bindung
: Objekt ist nur innerhalb der Quelldatei bekannt, in der es definiert ist.

externe Bindung
: Objekt ist auch in anderen Quelldateien bekannt (Bezeichner ist dem Linker bekannt).

Bestimmen der Bindung

- bei Funktionsvereinbarungen (Definitionen und Deklarationen)
  - Angabe von **static**
    - interne Bindung
  - keine Speicherklassenangabe oder Angabe von **extern**
    - falls eine top-level-Vereinbarung für diese Funktion sichtbar:
      - gleiche Bindung wie auf top-level
    - andernfalls
      - externe Bindung

> ⇒ Enthält die erste top-level-Vereinbarung für eine Funktion (oder ein Objekt) die Angabe **static**, hat der Name interne, andernfalls hat er externe Bindung.

- bei der Vereinbarung von anderen Objekten
  - Angabe von **static**

    interne Bindung
  - Angabe von **extern** (in Deklarationen auch innerhalb von Blöcken möglich)

    falls eine top-level-Vereinbarung für diesen Bezeichner sichtbar:

      gleiche Bindung wie auf top-level

    andernfalls

      externe Bindung

    Beispiel:

```
 ...
static int i;    /* interne Bindung     */
 ...
extern int i=25; /* interne Bindung     */
```

  - keine Speicherklassenangabe

    externe Bindung

    Beispiel:

```
 ...
static int i; /* interne Bindung        */
 ...
int i=25;     /* externe Bindung        */
              /*   (unzulässig)         */
```

Für jedes Objekt muß **genau eine** Definition vorhanden sein. Bei Objekten mit externer Bindung gilt dies für das gesamte Programm.

Beispiel:

```
/*  D A T E I  1  */

#include <stdio.h>
#include <stdlib.h>

extern void show_i(void); /* Deklaration,      */
                          /*   externe Bindung */
extern void set_i(int newval); /* Deklaration, */
                               /*   externe    */
                               /*   Bindung    */

static int i=10;          /* Definition,       */
                          /*   interne Bindung */

int main( void )          /* Definition,       */
{                         /*   externe Bindung */
  show_i();               /* 999 */
  set_i(12);
  printf("%d\n", i);      /*  10 */
  show_i();               /*  12 */
  exit(EXIT_SUCCESS);
}
```

```
/*  D A T E I  2  */

#include <stdio.h>

extern int i;       /* Deklaration,               */
                    /*   externe Bindung          */

void show_i(void) /* Definition von 'show_i',  */
                    /*   externe Bindung          */
{
  printf("%d\n", i);
  return;
}


/*  D A T E I  3  */

#include <stdio.h>

int i;              /* vorläufige Definition,    */
                    /*   externe Bindung          */

void set_i(int newval) /* Definition,             */
                         /*   externe Bindung    */
{
  i = newval;
  return;
}

extern int i=999; /* Definition (=> vorläufige */
                    /*   Definition wird als     */
                    /*   Deklaration behandelt), */
                    /* gleiche Bindung wie oben  */
```


Header-Dateien

Zu jeder Quelldatei, die irgendwelche Objekte (Funktionen oder Variablen) zur Verwendung in anderen Quelldateien bereitstellt, sollte eine Header-Datei erstellt werden.

In die Header-Datei gehören:

- **Deklarationen** der Objekte mit externer Bindung und ggf. Makrodefinitionen.
- **keine Definitionen** von Objekten, die Speicherplatz belegen!

Betrachtet man die Quelldatei als *Modul*, dann spezifiziert die Header-Datei die einzige von außen, d.h. von anderen Modulen, sichtbare Schnittstelle (*Export-Schnittstelle*).

Empfehlung: Die Header-Datei sollte auch in die Quelldatei (das Modul) selbst eingefügt werden.

Hinweis: Nach dem Verändern einer Header-Datei, sollte man nicht vergessen, alle Quelldateien, die diese Header-Datei einfügen, neu zu übersetzen.

Beispiel: Modul zur Verwaltung eines Stacks

```
/*  Dateiname: stack.c */

#include "stack.h"

#define MAX_STACK 25

static int stack[MAX_STACK];
static int top = 0;


void create( void )     /* Stack "erzeugen"    */
{
  /* ... */
}

void push( int element )/* Element auf Stack   */
{                       /*   ablegen           */
  /* ... */
}

int pop( void )         /* Element vom Stack   */
{                       /*   holen             */
  /* ... */
}

/*    .
      .
      .   */
```


zugehörige Header-Datei:

```
/*  Dateiname: stack.h */

/* Schnittstellenfunktionen: */

                        /* Stack "erzeugen":   */
extern void create( void );
                        /* Element auf Stack   */
                        /*   ablegen:          */
extern void push( int element );
                        /* Element vom Stack   */
                        /*   holen:            */
extern int pop( void );

/*    .
      .
      .   */
```

Aufgabe

Gegeben seien die auf den folgenden Seiten aufgelisteten 3 Dateien.

a. Geben Sie alle Stellen an, wo ein Objekt mit dem Namen *i* definiert wird. Geben Sie für jedes Objekt *i* außerdem alle Stellen an, wo es referiert wird.

b. Was gibt das Programm aus?


```
 1  /*  D A T E I  1  */
 2
 3  extern int next( void );
 4  extern int last( void );
 5  extern int new( int );
 6  extern int reset( void );
 7
 8  int i;
 9
10  int main( void )
11  {
12    int i=1, j;
13
14    i += reset();
15    for ( j=1 ; j<=2 ; j++ )
16    {
17      printf("%d, %d\n", i, j);
18
19      printf("%d\n", next());
20
21      printf("%d\n", last());
22
23      printf("%d\n", new(i+j));
24    }
25  }
```

```
 1  /*  D A T E I  2  */
 2
 3  static int i=10;
 4
 5  int next( void )
 6  {
 7    return(i += 1);
 8  }
 9
10  int last( void )
11  {
12    return(i -= 1);
13  }
14
15  int new( int i )
16  {
17    static int j = 5;
18    return(i = j += i);
19  }
```

```
 1  /*  D A T E I  3  */
 2
 3  extern int i;
 4
 5  int reset( void )
 6  {
 7    return(i);
 8  }
```


Präprozessor

Präprozessor-Direktiven

- beginnen mit # (vor und hinter # sind Zwischenraumzeichen erlaubt)
- haben eine vom Rest der Sprache unabhängige Syntax
- können an beliebigen Stellen vorkommen (häufig am Anfang einer Quelldatei)
- müssen innerhalb einer (logischen) Zeile kodiert werden

Einfügen von Dateien (#include-Direktive)

#include-Direktive hat eine der folgenden Formen:

1. **#include** <*filename*>

- für Header-Dateien des Systems
- nach der Datei *filename* wird an einer Reihe von implementierungsdefinierten Stellen gesucht. In UNIX-Systemen üblicherweise in den Verzeichnissen (*Directories*):
  1.) `/usr/include/`
  2.) `/usr/local/include/`

Beispiel:

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

2. **#include** "*filename*"

- für Header-Dateien des Benutzers
- nach *filename* wird zuerst an „lokalen" Stellen – unter UNIX z.B. im aktuellen Verzeichnis – gesucht, danach an denselben Stellen, wie bei der ersten Form.

Beispiel:

```
#include "stack.h"
```


Präprozessor
Einfügen von Dateien (#include-Direktive)

3. **#include** *token_sequence*

- *token_sequence* muß zu einer der ersten beiden Formen expandieren
- wird selten verwendet

Beispiel:

```
#ifdef OLD_VERSION
  #define INCFILE "old_header.h"
#else
  #define INCFILE "header.h"
#endif
/* ... */
#include INCFILE
```

Jede **#include**-Direktive wird durch den Inhalt der angegebenen Header-Datei ersetzt.

Die eingefügte Datei kann selbst auch wieder **#include**-Direktiven enthalten.

Makrodefinition und Expansion

Einfache Makros

Syntax:

#define *identifier token_sequence*$_{opt}$

veranlaßt den Präprozessor anschließend jedes Vorkommen von *identifier* durch die angegebene Folge von Symbolen (*token_sequence*) zu ersetzen (außer in Strings und Zeichenkonstanten).

Die eingesetzten Symbole werden wiederholt auf Makroaufrufe hin untersucht. (Die Definition des erzeugenden Makros wird während dieses Prozesses unterdrückt.)

Beispiele:

```
#define LINE_LEN 80
#define MAX_CONT_LINES 24
#define MAX_LINES (1+MAX_CONT_LINES)
#define TEXT_DIM \
          (MAX_LINES*LINE_LEN + 1 /* für '\0' */)
```

Die Zeile

```
char txtbuf[TEXT_DIM];
```

expandiert zu:

```
char txtbuf[(MAX_LINES*LINE_LEN + 1 )];
char txtbuf[((1+MAX_CONT_LINES)*LINE_LEN + 1 )];
char txtbuf[((1+24)*LINE_LEN + 1 )];
char txtbuf[((1+24)*80 + 1 )];
```


Beispiel für leere *token_sequence*:

```
#define PUBLIC
  /* keine Angabe für globale Objekte */
#define PRIVATE static
  /* 'static'-Objekte sind privat */
```

Danach sind z.B. Funktionsvereinbarungen der Art

```
PUBLIC void f(/*...*/)
PRIVATE int g(/*...*/)
```

erlaubt. Diese werden expandiert zu:

```
 void f( )
static int g( )
```

Beispiel für ein Makro, das in seiner eigenen Definition vorkommt:

```
#define int long int
```

Hierdurch wird das Schlüsselwort **int** überall im folgenden Quelltext durch **long int** ersetzt. (Das Expandieren von **int** wird nicht rekursiv fortgesetzt.)

Makros mit Parametern

Makrodefinition

Syntax:

#define *identifier*(*parameter_list*$_{opt}$) *token_sequence*$_{opt}$

parameter_list
: Liste von (durch Komma getrennten) Bezeichnern (Parameternamen)

Vorsicht: Zwischen Makroname (*identifier*) und der öffnenden Klammer dürfen keine Zwischenraumzeichen kodiert werden.

Beispiel:

```
#define sqr(x) x*x
```


Makroaufruf

Syntax:

identifier(*argument_list*$_{opt}$)

argument_list

- Liste von (durch Komma getrennten) Argument-Symbolfolgen
- Kommas innerhalb von verschachtelten runden Klammern, Stringkonstanten und Zeichenkonstanten gelten nicht als Argumenttrenner
- runde Klammern müssen paarweise auftreten (Klammern innerhalb von Stringkonstanten und Zeichenkonstanten zählen hierbei nicht.)

Zwischen *identifier* und der öffnenden Klammer sind Zwischenraumzeichen erlaubt.

Beispiele:

```
sqr(3)               /* expandiert zu 3*3         */
sqr(var)             /* expandiert zu var*var     */
sqr(div(7,3).quot) /* expandiert zu             */
                   /* div(7,3).quot*div(7,3).quot */
```

Makroexpansion:

1. Die Argument-Symbolfolgen werden auf Makroaufrufe untersucht, und bei Bedarf wird expandiert.

   (Dabei werden die Symbolfolgen genauso bearbeitet, als würden sie im Quelltext außerhalb eines Makroaufrufs vorkommen.)

2. In einer Kopie der Symbolfolge (*token_sequence*) aus der Makrodefinition werden die Parameternamen durch die entsprechenden Argument-Symbolfolgen ersetzt.

   (Ausgenommen sind Parameternamen innerhalb von Stringkonstanten und Zeichenkonstanten.)

3. Die Symbolfolge, die sich durch die Ersetzung in Schritt 2 ergibt, wird wiederholt auf Makroaufrufe hin untersucht.

   (Die Definitionen erzeugender Makros werden während dieses Prozesses unterdrückt.)

4. Das Ergebnis aus Schritt 3 ersetzt den gesamten Makroaufruf.

   Anmerkung:

   Auch wenn sich als endgültige Expansion (Ergebnis aus Schritt 4) eine syntaktisch korrekte Präprozessor-Direktive ergibt, wird diese vom Präprozessor nicht interpretiert, sondern als (vermeintlicher) C-Code an den Compiler durchgereicht.

Beispiel:

```
sqr(sqr(3))
```

Die Argument-Symbolfolge `sqr(3)` expandiert zu `3*3`, der gesamte Makroaufruf zu `3*3*3*3`.

#-Operator

kann nur innerhalb von Makros mit Parametern benutzt werden:

> # *parametername*

wird (in Schritt 2) ersetzt durch

> "*Argument-Symbolfolge*"

Im einzelnen gilt:

- In der Argument-Symbolfolge evtl. vorkommende Makroaufrufe werden **nicht** expandiert. (# *parametername* wird also durch eine Stringkonstante ersetzt, die eine vor Schritt 1 erzeugte Kopie der Argument-Symbolfolge enthält.)
- Zwischenraumzeichen, die am Anfang oder am Ende der Argument-Symbolfolge kodiert sind, werden entfernt.
- Jede in der Argument-Symbolfolge eingebettete Sequenz von Zwischenraumzeichen wird (außerhalb von Stringkonstanten und Zeichenkonstanten) durch jeweils ein einzelnes Leerzeichen ersetzt.
- Damit der Wert der erzeugten Stringkonstante mit der ursprünglichen Argument-Symbolfolge übereinstimmt, wird das Zeichen \
  a. vor jedem Anführungszeichen (“)
  b. vor jedem Zeichen \
  eingefügt.

Beispiel:

```
#define display(f,expr) \
          printf(#expr " = %" #f "\n", expr)
```


Der Makroaufruf `display(d,x/y)`

expandiert zu `printf("x/y" " = %" "d" "\n", x/y)`

somit ist der Effekt: `printf("x/y = %d\n", x/y)`

Das Makro `display` kann also anstelle von **printf** aufgerufen werden:

```
#include <stdio.h>
#include <stdlib.h>
#include "mydef.h"
                  /* 'mydef.h' enthalte die    */
                  /*   Definition von 'display'*/

int main( void )
{
  double x = 2.0, y = 3.0;

  display(f,x/y);
  display( f, x / y );
  display( f, x  /  y );
  display(d,2/3);
  display(s,    "Was ist das?"    "-  \?  -"   );
  exit(EXIT_SUCCESS);
}
```

Ausgabe:

```
x/y = 0.666667
x / y = 0.666667
x / y = 0.666667
2/3 = 0
"Was ist das?" "-  \?  -" = Was ist das?-  ?  -
```

##-Operator

ist innerhalb der Symbolfolge (*token_sequence*) bei beiden Formen der Makrodefinition erlaubt:

$$token_1 \ \#\# \ token_2$$

verkettet die beiden Token zu einem neuen Symbol.

Im einzelnen gilt:

- Falls $token_i$ ein Makroname ist, wird dieser **nicht** expandiert.
- Falls $token_i$ ein Parameter ist, werden in der entsprechenden Argument-Symbolfolge evtl. vorkommende Makroaufrufe **nicht** expandiert. ($token_i$ wird also durch eine vor Schritt 1 erzeugte Kopie der Argument-Symbolfolge ersetzt.)
- Unmittelbar bevor die Expansion eines Makros wiederholt auf Makroaufrufe untersucht wird, wird aus den beiden Token, die Operanden von ## sind, ein neues Symbol durch Verkettung gebildet.

Beispiel:

```
#define paste(front,back) front ## back


 printf("%d\n", paste(name,1));
```

expandiert zu:

```
 printf("%d\n", name1);
```

Ohne den Operator ## lautete die Expansion:

```
 printf("%d\n", name 1);
```


Probleme bei der Verwendung von Makros

- Makros und ihre Argumente können mit ihrem Kontext interagieren.
- Seiteneffekte innerhalb eines Argumentes können (bei unsicheren Makros) mehrfach auftreten.

Beispiel:

```
#define abs(x) x<0 ? -x : x
#define sqr(x) x * x

int main( void )
{

   ...

  /* Die beiden folgenden Ausdruecke liefern   */
  /*   nicht das beabsichtigte Ergebnis:       */
  printf("%d\n", -3 * abs(i+j));
  printf("%d\n", sqr(i+1));

   ...

  /* 'j' wird hier nicht nur um 1 erhoeht:     */
  sqr(j++);

   ...

}
```

Ausgabe des Präprozessors:

```
int main( void )
{

   ...



  printf("%d\n", -3 * i+j<0 ? -i+j : i+j);
  printf("%d\n", i+1 * i+1);

   ...


  j++ * j++;

   ...

}
```

Die Ausdrücke werden interpretiert als:

```
((((-3) * i)+j)<0) ? ((-i)+j) : (i+j)
(i+(1 * i))+1
```

Abhilfe:

- Ausdrücke und Parameter in Klammern einschließen:
  ```
  #define abs(x) ( (x)<0 ? -(x) : (x) )
  #define sqr(x) ( (x) * (x) )
  ```
- unsichere Makros vermeiden

- Ein Makro kann mit einer umliegenden Kontrollstruktur interagieren.

Beispiel 1:

```
#define debug(f,expr) \
          if ( debug_mode ) \
            printf( #expr " = %" #f "\n", (expr))

int main( void )
{

   ...

  if ( a >= 0 )
    debug(d,a);
  else
    printf("a ist negativ\n");

   ...

}
```

Ausgabe des Präprozessors:

```
int main( void )
{

   ...

  if ( a >= 0 )
    if ( debug_mode ) printf( "a" " = %" "d" "\n"
, (a));
  else
    printf("a ist negativ\n");

   ...

}
```

Bessere Implementierungen von **debug** sind:

```
#define debug(f,expr) \
  (  ( debug_mode ) ? \
    printf( #expr " = %" #f "\n", (expr)) : 0  )

#define debug(f,expr) \
  (  ( debug_mode ) && \
    printf( #expr " = %" #f "\n", (expr))  )
```


Beispiel 2: Vertauschen von zwei Variablen

```
#define swap(a,b,temp) temp = a; a = b; b = temp

int main( void )
{
  double x, y, dhelp;
  int    i, j, itmp;

   ...

  swap(y, x, dhelp);
  /* vertauscht y mit x */

  if ( i > j )
    swap(i, j, itmp);
    /* fuehrt zu Problem */

   ...

}
```

Ausgabe des Präprozessors:

```
int main( void )
{
  double x, y, dhelp;
  int    i, j, itmp;

   ...

  dhelp = y; y = x; x = dhelp;


  if ( i > j )
    itmp = i; i = j; j = itmp;


   ...

}
```


Lösungsversuch:

Anweisungen werden zu einer Verbundanweisung zusammengefaßt.

```
#define swap(a,b,temp) \
          {temp = a; a = b; b = temp;}

int main( void )
{
  double x, y, dhelp;
  int    i, j, itmp;

   ...

  swap(y, x, dhelp);
  /* vertauscht y mit x */

  if ( i > j )
    swap(i, j, itmp);
    /* kein Problem mehr*/

   ...

  if ( i > j )
    swap(i, j, dhelp);
  else   /* fuehrt zu Problem */
  {
     ...
  }

   ...

}
```

Ausgabe des Präprozessors:

```
int main( void )
{
  double x, y, dhelp;
  int    i, j, itmp;

   ...

  {dhelp = y; y = x; x = dhelp;};


  if ( i > j )
    {itmp = i; i = j; j = itmp;};


   ...

  if ( i > j )
    {dhelp = i; i = j; j = dhelp;};
  else
  {
     ...
  }

   ...

}
```

Der Versuch diese Ausgabe zu übersetzen, mißlingt:

```
       20 |   else
               ..a.......
a - 1506-046: (S) Syntax error.
```


Lösung:

Mehrere Ausdrucksanweisungen werden mit Hilfe des Komma-Operators als syntaktisch eine Ausdrucksanweisung kodiert.

```
#define swap(a,b,temp) \
          ((void)((temp)=(a),(a)=(b),(b)=(temp)))

int main( void )
{
  double x, y, dhelp;
  int    i, j, itmp;

  /* ... */
  if ( i > j )
    swap(i, j, itmp);
    /* vertauscht i mit j */

  /* ... */

  if ( i > j )
    swap(i, j, dhelp);
  else   /* kein Problem mehr*/
  {
    /* ... */
  }

  /* ... */

}
```

Vorteile des Makros gegenüber der Implementierung als Funktion:

- effizienter Inline-Code
- keine Festlegung auf spezielle Datentypen
- einfacher Aufruf des Makros, da „*call by reference*“ für die Argumentübergabe nicht erforderlich

Löschen von Makrodefinitionen

#undef *identifier*

- löscht die Definition des Namens *identifier* für den Präprozessor.
- Es ist kein Fehler, wenn *identifier* nicht definiert war.

Hinweis:

Die Implementierung einer Bibliotheksfunktion kann durch eine Makrodefinition in der entsprechenden Header-Datei erfolgen. (Beispielsweise sind die Funktionen in <ctype.h> häufig als Makros implementiert.) Es darf davon ausgegangen werden, daß diese Makrodefinitionen sicher sind, die Argumente jedes Makros also nur ein einziges Mal bewertet werden.

In der Regel muß eine so definierte Bibliotheksfunktion zusätzlich jedoch auch als Funktion implementiert sein. Wird die Funktionsform benötigt, kann die Makrodefinition auf eine der beiden folgenden Arten unterdrückt werden:

global durch

#undef *function_name*

lokal (im Funktionsaufruf)
durch Einschließen des Funktionsnamens in runde Klammern

Beispiel: `(abs)(x)`

Da dem Funktionsnamen keine öffnende Klammer folgt, liegt kein Makroaufruf mehr vor.

Bedingte Übersetzung

Die Präprozessor-Direktiven **#if**, **#ifdef**, **#ifndef**, **#elif**, **#else** und **#endif** ermöglichen es, die Übersetzung bestimmter Programmabschnitte von Bedingungen abhängig zu machen:

#if $const_logical_expr_1$
    $text_1$
#elif $const_logical_expr_2$
    $text_2$
#elif $const_logical_expr_3$
    $text_3$
 . . .
#else
    $text_n$
#endif

Anmerkungen:

- $const_logical_expr_i$ bezeichnet einen ganzzahligen konstanten Ausdruck, der weder einen **sizeof**- noch einen **cast**-Operator und auch keine Aufzählungskonstanten enthalten darf. Dafür ist der **defined**-Operator zulässig. Der Resultatwert des Ausdrucks wird als logischer Wert interpretiert.
- Makros innerhalb von $const_logical_expr_i$ werden normal expandiert, sofern sie nicht Operand des **defined**-Operators sind.

- nach der Makroexpansion noch vorhandene Namen werden durch 0L ersetzt.
- **#elif**-Direktiven sowie die davon abhängigen Texte dürfen fehlen. Das gleiche gilt für **#else**.

defined-Operator

Syntax:

1. **defined** *macro_name*
2. **defined** **(** *macro_name* **)**

- ergibt 1L (wahr), falls *macro_name* mittels **#define** definiert wurde, andernfalls 0L

Direktiven **#ifdef** und **#ifndef**

#ifdef *macro_name*

⇔ `#if defined(macro_name)`

#ifndef *macro_name*

⇔ `#if !defined(macro_name)`

Beispiel:

```
#if defined(AIX) && (__STDC__ == 1)
  #define R_OK 04       /* Lesen erlaubt        */
  #define W_OK 02       /* Schreiben erlaubt    */
  #define X_OK 01       /* Ausfuehrung erlaubt */
  #define F_OK 00       /* Datei existiert      */
  extern int access( const char *fid, int mode );
#elif defined(AIX) || \
      (defined(SOLARIS2X) && defined(GCC))
  #include <unistd.h>
#elif defined(UNICOS) || \
        (defined(SUNOS) && defined(GCC))
  #include <sys/unistd.h>
#elif defined(CMS) && defined(IBMC)
  #define R_OK 04       /* Lesen erlaubt        */
  #define W_OK 02       /* Schreiben erlaubt    */
  #define X_OK 01       /* Ausfuehrung erlaubt */
  #define F_OK 00       /* Datei existiert      */
#else
  #error "function 'access' not declared: \
check system/compiler definition"
#endif

/* ... */

#if defined(CMS) && defined(IBMC)
static int access(
    const char *fid,
    int        mode
  )
{
  /* ... */
}
#endif /* CMS && IBMC */
```


Aufgaben

1. Welches Problem kann bei folgendem Programm auftreten?

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

int main( void )
{
  int putchar(int);
  int i;

  printf("hello, world\n");
  for (i = 0; i < 12; i++)
    putchar('-');
  putchar('\n');
  exit(EXIT_SUCCESS);
}
```

2. Gegeben seien zwei Header-Dateien, sowie eine Quelldatei, die im folgenden aufgelistet sind. Geben Sie die Ausgabe an, die der Präprozessor für die angegebene Quelldatei produziert.

```
/*  Header-Datei: blue.h  */

enum BlueColors
  { LightBlue, SkyBlue, Blue, MidnightBlue };
#include "red.h"
```

```
/*  Header-Datei: red.h  */

#ifndef RED_H_INCLUDED
#define RED_H_INCLUDED
#include "blue.h"
enum RedColors
  { Orange, Tomato, Red, VioletRed };
#endif /* RED_H_INCLUDED */


/*  Quelldatei: prg.c  */

#include "blue.h"
#include "red.h"
```

3. Zwei Makros seien definiert durch

```
#define M(x)    M ## x
#define MM(M,y) M = # y
```

Geben Sie die Expansion des Aufrufs **M(M) (A,B)** an.

Bedingtes Einfügen von Header-Dateien

Jede Header-Datei kann selbst Header-Dateien einfügen, die ihrerseits von weiteren Dateien abhängen können. Normalerweise wird der Benutzer einer Header-Datei diese Abhängigkeiten nicht mehr durchschauen, so daß ein und dieselbe Header-Datei leicht mehrfach eingefügt werden kann.

Damit dies nicht zu Fehlern führt, sollte jede Header-Datei eine Abfrage enthalten, die ein mehrfaches Einfügen der Datei verhindert.

Beispiel:

```
/*  Header-Datei: stack.h */

#ifndef STACK_H_INCLUDED
#define STACK_H_INCLUDED

/* Schnittstellenfunktionen: */

                        /* Stack "erzeugen":   */
extern void create( void );
                        /* Element auf Stack   */
                        /*   ablegen:          */
extern void push( int element );
                        /* Element vom Stack   */
                        /*   holen:            */
extern int pop( void );

/*   .
     .
     .   */

#endif /* STACK_H_INCLUDED */
```

Sonstige Direktiven

#error *message*$_{opt}$

- gibt beim Übersetzen eine Fehlermeldung aus, die den Text *message* beinhaltet.
- Makros innerhalb von *message* werden zuvor expandiert.
- mit dem Ausführen dieser Direktive soll die Übersetzung unmittelbar beendet werden. (Der C-Standard legt dies jedoch nicht verbindlich fest.)

#pragma *token_sequence*$_{opt}$

- Effekt ist implementierungsabhängig
- unbekannte **#pragma**-Direktive wird ignoriert.
- Makros innerhalb von *token_sequence* werden normal expandiert.

Beispiel: Hinweis zur Vektorisierung geben (CRAY)

```
#pragma _CRI ivdep /* ignore vector dependency */
```


#line *number*
#line *number* "*filename*"
#line *token_sequence*

- die dezimale Integer-Konstante *number* legt die Zeilennummer der nächsten Quelltextzeile neu fest.
- *filename* spezifiziert einen Namen für die Quelldatei, der anstelle des aktuellen Namens (z.B. in Fehlermeldungen) verwendet werden soll.
- *token_sequence* muß zu einer der ersten beiden Formen expandieren.
- diese Direktive ermöglicht Programmier-Werkzeugen, die C-Quelldateien modifizieren, den Bezug zur ursprünglichen Quelldatei aufrechtzuerhalten.

#

- leere Direktive
- hat keine Wirkung

Vordefinierte Makronamen

__LINE__ Zeilennummer der aktuellen Quelltextzeile (dezimale Integer-Konstante)

__FILE__ Name der aktuellen Quelldatei (Stringkonstante)

__DATE__ Datum der Übersetzung (Stringkonstante der Form "*Mmm dd yyyy*", z.B. `"Aug  3 1995"`)

__TIME__ Zeitpunkt der Übersetzung (Stringkonstante der Form "*hh*:*mm*:*ss*")

__STDC__ ist mit Wert 1 nur dann definiert, wenn die Implementierung dem ANSI-Standard genügt.

Beispiel:

```
       /* ... */

 98    if ( (errcode = f( /* ... */ )) != 0 )
 99      error(errcode, __FILE__, __LINE__);

       /* ... */

119    if ( (errcode = f( /* ... */ )) != 0 )
120      error(errcode, __FILE__, __LINE__);

       /* ... */
```



```
void error(
    int        errcode, /* Fehlercode           */
    const char *file,   /* Name der Quelldatei */
    int        line     /* Zeilennummer         */
  )
{
  switch (errcode) {

    /* ... */

    default:
      printf("Undefined error number %d, file "
        "'%s', line %d\n", errcode, file, line);
      break;
  }
  return;
}
```

Ausgabe: z.B.

```
Undefined error number 35, file 'bsp.c', line 120
```

Zeiger und Vektoren

Adreßoperator

Syntax:

1. **&***lvalue*
2. **&***vector*
3. **&***function*

Resultatwert:

Adresse des Objektes *lvalue*, des Vektors *vector* bzw. der Funktion *function*.
(Die Adresse einer Konstanten oder eines Ausdrucks kann nicht bestimmt werden, wenn der Typ weder Vektor noch Funktion ist.)

Resultattyp:

1. Zeiger auf ein Objekt mit dem Typ von *lvalue*.
2. Zeiger auf einen Vektor. Anzahl und Typ der Vektorelemente sind durch den Typ von *vector* festgelegt.
3. Zeiger auf eine Funktion mit dem Typ von *function*.

Beispiel:

```
char *cp;           /* Zeiger auf ein Objekt    */
                    /*   mit dem Typ 'char'     */
char zeichen;

cp = &zeichen;      /* 'cp' zeigt auf 'zeichen' */
```

Verweis-, Inhalts-, Dereferenzierungsoperator
(*indirection, dereferencing operator*)

Syntax:

**pointer*

Resultatwert:

- Objekt, auf das *pointer* zeigt
- Ausdruck ist ein l-Wert

Die Syntax der Vereinbarung eines Zeigers imitiert die Verwendung des Dereferenzierungsoperators:

```
int *ip;      /* Die Definition besagt, dass   */
              /*   'ip' dereferenziert werden  */
              /*   darf und '*ip' ein 'int'-   */
              /*   Wert ist.                   */
```

Beispiel:

```
int i = 1, j, k[5];
int *ip = &i, *jp = &j;

j = *ip;           /* <=> j = i;               */
*ip = 0;           /* <=> i = 0;               */
ip = &k[3];        /* <=> ip = &(k[3]);        */
*ip = 0;           /* <=> k[3] = 0;            */
```


Zeiger als Funktionsargumente („*call by reference*“)

Beispiel: Vertauschen von zwei Variablen

```
void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
  return;
}
```

Aufruf:

```
  int a, b;

  /* ... */

  swap(&a, &b);
```

Verwendung von Zeigern in Ausdrücken

Beispiele:

```
   *ip = *ip + 10;
⇔  *ip += 10;

   ++*ip
⇔  ++(*ip)         /* Objekt, auf das 'ip'     */
                   /*   zeigt, wird erhöht     */

   *ip++
⇔  *(ip++)         /* Zeiger wird erhöht       */

   (*ip)++         /* Objekt, auf das 'ip'     */
                   /*   zeigt, wird erhöht     */
```

Vorsicht: syntaktisches Problem

```
int i = 3, j = 1, *ip = &i;

i = i/*ip;                    /* <=> i/i   */
-i+j ? printf("...", ...) : 0;
```


Generische Zeiger

spezieller Zeigertyp:

void *	/* unspezifischer, generischer Zeiger */

Im Gegensatz zu allen anderen Zeigertypen beinhaltet dieser Typ nicht die Art des Objektes, das an der Adresse erwartet wird.

- ein generischer Zeiger darf nicht dereferenziert werden.
- jeder Zeiger kann in einen generischen Zeiger und zurück umgewandelt werden (ausgenommen Zeiger auf Funktionen).

Nullzeiger

spezieller Zeigerwert:

```
#include <stddef.h>
/* oder   #include <stdio.h> */
/* oder   #include <stdlib.h> */
/*          . . .             */

NULL   /* Makro */
```

- soll anzeigen, daß auf kein Objekt verwiesen wird.
- mögliche Definition:
  `#define NULL ((void *)0)`
- als logischer Wert interpretiert, hat **NULL** den Wert „false“.

Formatierte Ausgabe von Zeigerwerten

printf-Konvertierungsspezifikation

darf enthalten:

Flag - linksbündige Ausgabe

Feldbreite minimale Breite des Ausgabefeldes

K-Typ

K-Typ	**Argument-typ**	**Ausgabe**
p	**void ***	Zeigerwert (Darstellung implementierungsabhängig)

Typzusätze (Attribute für Typen)

const

- Wert dieser Objekte wird nicht verändert.
- dürfen initialisiert werden.
- können vom Compiler im schreibgeschützten (*read-only*) Speicher angelegt werden. (nur Variablen mit statischer Speicherallokation)
- können Optimierungsmöglichkeiten verbessern.
- der Versuch ein **const**-Objekt zu verändern führt zu einem implementierungsabhängigen Resultat (z.B. Fehlermeldung; Compiler darf **const** jedoch auch ignorieren).

Beispiele:

```
const int ic = 37;
const int *pc;      /* Zeiger auf "Konstante"   */
int * const cp;     /* konstanter Zeiger        */
int i, *p;
void f(const int *ip);

ic = 5;             /* unzulässig */

pc = &i;            /* ok */
f(&i);              /* ok */

p = &ic;            /* unzulässig */

p = (int *) pc;     /* ok */
*p = 5;             /* ok */

p = (int *) &ic;    /* ok */
*p = 5;             /* formal ok, aber Laufzeit-*/
                    /*   fehler möglich         */

pc = &ic;           /* ok */
*pc = 5;            /* unzulässig */
```

volatile

Der Typzusatz **volatile** zeigt dem Compiler an, daß das zugehörige Objekt auf eine für den Compiler nicht sichtbare Art und Weise durch andere Teile des Programms (z.B. Signalbehandlungsroutinen), durch die Hardware oder andere Programme verändert werden kann. Der Compiler muß deshalb den Wert des Objektes immer genau an den Stellen lesen bzw. abspeichern, an denen diese Aktionen im Programm angegeben sind.

volatile dient somit dazu, bestimmte Optimierungen zu verhindern.

Dynamische Speicherverwaltung

```
#include <stdlib.h>
void *calloc(size_t count, size_t size);
```

legt Speicherbereich für *count* Objekte der Größe *size* Bytes an. Der Bereich wird mit Null-Bytes initialisiert.

Resultatwert: Zeiger auf Speicherbereich oder **NULL**, falls die Anforderung nicht erfüllt werden kann. (implementierungsabhängig, falls *count*==0 oder *size*==0)

```
#include <stdlib.h>
void *malloc(size_t size);
```

legt Speicherbereich für ein Objekt der Größe *size* Bytes an. Der Bereich wird nicht initialisiert.

Resultatwert: analog zu **calloc**

```
#include <stdlib.h>

void *realloc(void *p, size_t size);
```

ändert die Größe des Objektes, auf das *p* zeigt, in *size* Bytes ab. Bis zur kleineren der alten und neuen Größe bleibt der Inhalt unverändert. Wird der Bereich für das Objekt größer, so ist der zusätzliche Bereich uninitialisiert.

- falls *p*==**NULL**: äquivalent zu **malloc**
- *p*!=**NULL** und *size*==0: Bereich wird freigegeben

Resultatwert: Zeiger auf **neuen** Bereich oder **NULL** (in letzterem Fall ist **p* unverändert)

Anmerkung:

Von **calloc**, **malloc** oder **realloc** angelegte Speicherbereiche sind immer so ausgerichtet, daß sie der größtmöglichen Alignment-Anforderung genügen. Daher können Objekte jeden Typs darin abgelegt werden.

```
#include <stdlib.h>

void free(void *p);
```

gibt Bereich, auf den *p* zeigt, frei (*p*==**NULL** bewirkt nichts).

Bereich muß mit einer der Funktionen **calloc**, **malloc** oder **realloc** angelegt worden sein.

Beispiel:

```
#include <stdlib.h>

int *p;

  ...

 p = (int *) malloc( sizeof(int) );

  ...

 free(p);
```

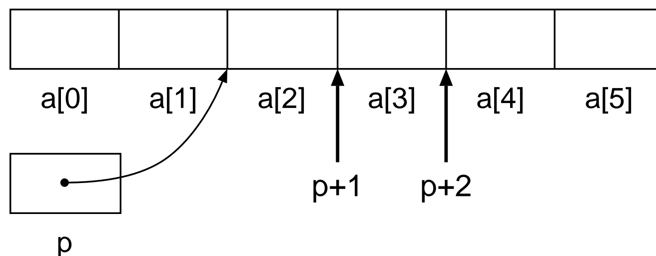
Adreß-Arithmetik (Arithmetik mit Zeigern)

zulässige Operationen:

- Zuweisung von Zeigern
  - ein Operand ist ein Zeiger, der andere ein generischer Zeiger
  - beide Operanden sind Zeiger gleichen Typs (beim rechten Operanden darf **const** oder **volatile** fehlen)
  - linker Operand Zeiger, rechter Operand Ausdruck mit Wert 0 → **NULL**
- Addition einer ganzen Zahl zu einem Zeiger

  Beispiel:

```
int a[6];
int *p = &a[2];
```



Zeiger und Vektoren
Adreß-Arithmetik (Arithmetik mit Zeigern)

- Subtraktion einer ganzen Zahl von einem Zeiger

  z.B.: `p-2`

- Subtraktion zweier Zeiger (auf Elemente des gleichen Vektors)

  Resultat:

  Integer-Wert mit Vorzeichen (implementierungsabhängig, aber als **ptrdiff_t** in **stddef.h** definiert), der den Abstand zwischen den Objekten repräsentiert, auf die die Zeiger verweisen.

  Beispiel:

```
int *p = &a[1], *q = &a[3];
q-p ⟶ 2
q -= 3;
q-p ⟶ -1
```

- Vergleich zweier Zeiger gleichen Typs

  `<   >   <=   >=   ==   !=`

  Für die Operationen `<`, `>`, `<=` und `>=` ist der Resultatwert nur dann definiert, wenn die Zeiger auf Elemente des gleichen Vektors oder auf Komponenten der gleichen Struktur verweisen.

  Die Operatoren `==` und `!=` erlauben zusätzlich den Vergleich eines Zeigers mit einem generischen Zeiger (einschließlich **NULL**).

  Beispiel:

```
int *p;
                /* Durchlaufen des Vektors 'a' */
for (p = &a[0]; p <= &a[5]; p++)
    ... *p ... ;
```

- explizite Umwandlung eines Zeigers in einen anderen Zeigertyp
  - kann Adressierungsfehler verursachen, wenn der ursprüngliche Zeiger nicht auf ein Objekt verweist, das geeignet im Speicher ausgerichtet ist.
  - In der Regel erfordert die Umwandlung von Zeigertypen keine Änderung der internen Repräsentation der Zeigerwerte. In einigen Fällen können Zeiger unterschiedlichen Typs jedoch auch unterschiedlich groß oder verschieden aufgebaut sein.[18]
- explizite Umwandlung eines Zeigers in einen (ausreichend großen) Integer-Typ
- explizite Umwandlung eines Integer-Wertes in einen Zeiger

  ermöglicht z.B. bei einem Rechner mit *memory-mapped input/output* die Definition eines Zeigers auf einen Ausgabepuffer des Systems.

[18] Beispielsweise könnten **char**-Zeiger (sowie generische Zeiger) auf „wortadressierten“ Rechnern (wie z.B. Cray) größer als andere Zeigertypen sein oder in ihrer internen Repräsentation Bits verwenden, die für andere Zeigertypen ohne Bedeutung sind. (Letzteres gilt für Cray.)

Beziehung zwischen Zeigern und Vektoren

Ist der Typ eines Ausdrucks „**Vektor von T**“, dann gilt:

- Wert des Ausdrucks:
  Zeiger auf das erste Element des Vektors
- Typ des Ausdrucks:
  wird abgeändert in „**Zeiger auf T**“
- Ausdruck ist kein l-Wert
  (Ausnahme: Parameter mit Typ „Vektor von T“)

Ausnahmen:

- Ausdruck ist Operand des **sizeof**-Operators
- Ausdruck ist Operand des Adreßoperators
- Ausdruck ist ein Zeichenketten-Initialisierer für einen **char**-Vektor

Beispiele:

```
int a[6];    /* 'a' verhält sich wie ein      */
             /*   konstanter Zeiger auf das   */
             /*   erste Element des Vektors   */

"abcd"+1     /* verhält sich wie der String   */
             /*   "bcd"                       */
```

Jede Operation mit Vektorindizes kann auch (i. allg. effizienter) mit Zeigern formuliert werden:

`p = &a[0]`	⇔	`p = a`
`&a[i]`	⇔	`a+i`
`a[i]`	⇔	`*(a+i)`

`a[i]` wird vom Compiler in `*(a+i)` umgewandelt.

⇒ Umgekehrt können auch Zeiger mit Vektorindizes verwendet werden:

`*p`	⇔	`p[0]`
`*(p+i)`	⇔	`p[i]`

Übergabe eines Vektors an eine Funktion:

Ein Vektorname als Parameter ist immer eine Zeiger**variable**, unabhängig, ob der formale Parameter als Vektor (z.B. `char s[]`) oder als Zeiger (`char *s`) vereinbart wurde.

Falls *vector* Formalparameter ist, liefert **sizeof**(*vector*) also die Größe eines Zeigers.
⇒ Eine Funktion kann die Größe eines übergebenen Vektors nicht ermitteln.

Beispiel:

```
/* Laenge eines Strings bestimmen: */
  length = strlen("abc");


/* Version 1 (mit Vektorindizes): */
int strlen(const char s[])
{
  int i;

  i = 0;
  while ( s[i] != '\0' )
    i++;
  return(i);
}


/* Version 2 (mit Zeigerarithmetik): */
int strlen(const char *s)
{
  int n;

  for ( n = 0; *s != '\0'; s++ )
    n++;
  return(n);
}
```

```
/* Version 3: */
int strlen(const char *s)
{
  int n = 0;

  while ( *s++ )
    n++;
  return(n);
}


/* Version 4: */
int strlen(const char *p = s;
```


Deklaration eines (an anderer Stelle definierten) Vektors:

- Dimensionslänge darf fehlen
- Vereinbarung als Zeiger nicht zulässig

Beispiel:

```
/*  D A T E I  1  */

#define VEKLEN 10

int v1[VEKLEN];         /* Definition            */
int v2[VEKLEN];         /* Definition            */

/* ... */


/*  D A T E I  2  */

#define VEKLEN 10

extern int v1[];        /* Deklaration           */
extern int v2[VEKLEN];  /* Deklaration           */

/* ... */

int main( void )
{
  /* ... */
  len = sizeof(v1);     /* unzulässig            */
  len = sizeof(v2);     /* ok                    */
  /* ... */
}
```

Initialisierung

skalare Objekte

- statische Objekte

  werden durch konstante Ausdrücke initialisiert.

  Beispiel:

```
static int i = 7 + 12;
static int j = { 7 + 12 };
/* beide Anweisungen sind äquivalent */
```

  Wenn keine explizite Initialisierung erfolgt, haben statische Objekte den Wert 0, 0.0 oder NULL.[19]

- **auto**- oder **register**-Objekte

  werden durch beliebige Ausdrücke initialisiert.

  Beispiel:

```
auto int *p = (int *) malloc(9*sizeof(int));
```

  Wenn keine explizite Initialisierung erfolgt, haben diese Objekte einen undefinierten Wert.

Vektoren (einschließlich Zeichenvektoren)

werden durch eine Liste **konstanter** Ausdrücke, die in geschweifte Klammern eingeschlossen ist, initialisiert.

Die Liste darf mit einem Komma enden.

[19] Die meisten älteren Compiler initialisieren einfach mit Null-Bytes. (Hierdurch ergeben sich jedoch nicht bei jedem Rechner die oben angegebenen Werte.)

Beispiele:

```
int a[3] = {1, 2, 3};

double b[2] = {0.0, 1.5, 2.3};/* unzulässig */

static int i;
auto int j;
int *c[2] = {&i};    /* ok, 2 Elemente:     */
                     /*   &i, NULL          */
int *d[2] = {&j};    /* unzulässig, da kein */
                     /*   konstanter Aus-   */
                     /*   druck vorliegt    */
```

Die Größe des Vektors kann durch die Anzahl der Initialisierungen festgelegt werden:

```
float f[] = {-1.0f, 0.0f};
```

Die Anzahl der Elemente kann im Programm durch folgenden Ausdruck ermittelt werden:

```
#define VEKLEN ( sizeof(f) / sizeof(float) )
```

oder allgemeiner durch

```
#define vectorsize(v) \
          ( sizeof(v) / sizeof(v[0]) )
```

Sonderfall: Initialisierung von Zeichenvektoren

```
char s[] = {'T', 'e', 'x', 't', '\0'};
```

kann kürzer geschrieben werden:

```
char s[] = "Text";    /* 5 Elemente           */

char s[4] = "Text";   /* ok, aber '\0' wird   */
                      /*   nicht abge-        */
                      /*   speichert          */

char s[15] = "Text";  /* ok, Rest wird mit    */
                      /*   Null-Bytes aufge-  */
                      /*   füllt              */

char s[2] = "Text";   /* unzulässig           */
```

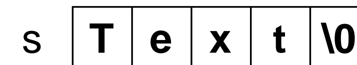

Gegenüberstellung von Vektor und Zeiger:

```
char s[] = "Text";   (1)
```

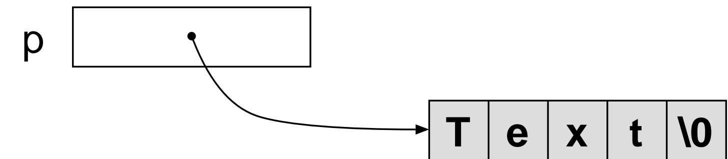
sollte nicht verwechselt werden mit

```
char *p  = "Text";   (2)
```

(1) das definierte Objekt ist ein Vektor



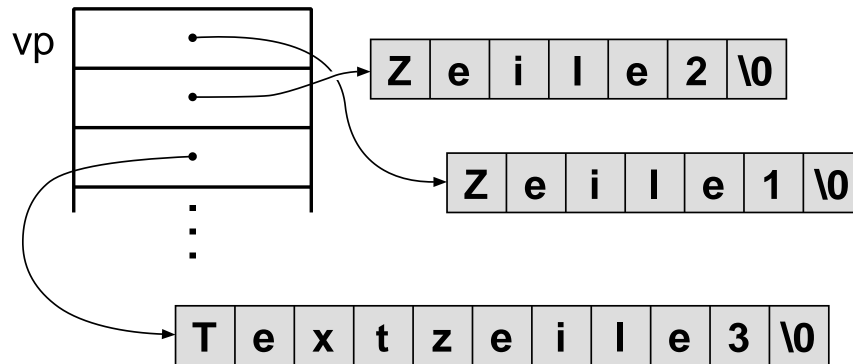
(2) das definierte Objekt ist ein Zeiger




```
s[2] = 's';       /* ok                    */
p[2] = 's';       /* unzulässig (Resul-    */
                  /*   tat undefiniert)    */

s++;              /* unzulässig            */
p++;              /* ok                    */

s = "abc";        /* unzulässig;           */
                  /*   korrekt wäre:       */
                  /*     strcpy(s,"abc");*/
p = "abc";        /* ok                    */

p = s;            /* ok                    */
p[2] = 's';       /* jetzt ok              */
```

Vektoren von Zeigern, Zeiger auf Zeiger

Beispiel:




```
char *vp[] = {                ⇔ char *(vp[])
   "Zeile1",
   "Zeile2",
   "Textzeile3",
   /* ... */
};
```

- Da `vp` ein Vektorname ist, kann er selbst auch als Zeiger behandelt werden.
- In Formalparametervereinbarungen ist die Vereinbarung

  ```
  char *p[]
  ```

  äquivalent zu

  ```
  char **p
  ```


Argumente aus der Kommandozeile

Dieses Kapitel dient gleichzeitig als Anwendungsbeispiel für Vektoren von Zeigern.

Beim Aufruf eines C-Programmes können an die Funktion `main` Argumente aus der Kommandozeile übergeben werden.

Statt

```
int main( void ) {  ...
```

ist die Definition

```
int main( int argc, char *argv[] ) {  ...
```

bzw.

```
int main( int argc, char **argv ) {  ...
```

zu verwenden.

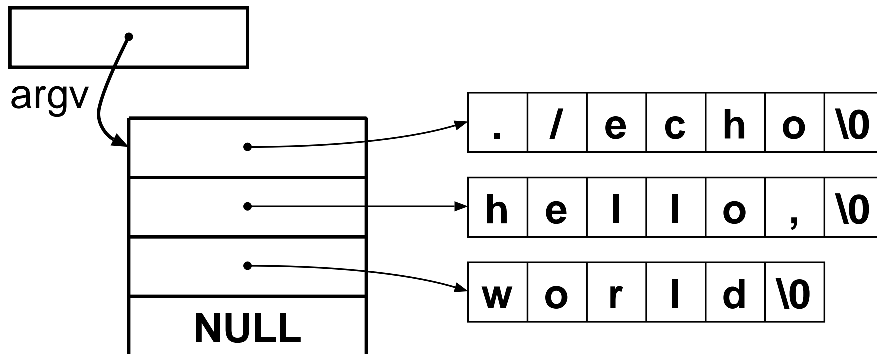
argc Anzahl der Argumente in der Kommandozeile (***arg**ument **c**ount*). Da der Kommandoname immer als Argument übergeben wird, gilt: **argc**≥1

argv Zeiger auf einen Vektor, der Zeiger auf die Argumente (Zeichenketten) aus der Kommandozeile enthält (***arg**ument **v**ector*). **argv**[0] zeigt immer auf den Kommandonamen. **argv**[**argc**] ist immer ein **NULL**-Zeiger.

Beispiel:

Quelldatei: `echo.c`

```
cc -o echo echo.c

./echo hello, world
```




```
/* echo.c (Version mit Vektorindizes): */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i;

  for (i = 1; i < argc; i++)
    printf("%s%s", argv[i],
      (i < argc-1) ? " " : "\n");
  exit(EXIT_SUCCESS);
}
```

```
/* echo.c (Version mit Zeigerarithmetik): */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  while (--argc > 0)
    printf("%s%s", *++argv,
      (argc > 1) ? " " : "\n");
  exit(EXIT_SUCCESS);
}
```

```
/* echo.c (Version ohne 'argc'): */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  while ( *++argv != NULL )
    printf("%s%s", *argv,
      (*(argv+1) != NULL) ? " " : "\n");
  exit(EXIT_SUCCESS);
}
```

Beispiel: Erkennen von UNIX-Kommandos

```
  while ( (--argc > 0) && ((*++argv)[0] == '-') )
    while (c = *++argv[0])
      switch(c)
      {
        /* ... */
      }
```


Ein-/Ausgabe von Zeichen und Zeichenketten

Funktionen zur Ausgabe einzelner Zeichen

#include <stdio.h>

int fputc(int *c*, FILE ****stream*);**

int putc(int *c*, FILE ****stream*);** /* Makro-Version */

fputc und **putc** sind äquivalente Funktionen. Sie geben das Zeichen *c* auf die Datei aus, die mit dem Datei-Zeiger *stream* verknüpft ist.

putc kann jedoch als Makro implementiert sein. In diesem Fall kann das Argument für *stream* mehrfach bewertet werden, so daß ein Ausdruck mit Seiteneffekten vermieden werden sollte.

Resultatwert: *c* oder (bei Fehler) **EOF**

Beispiel:

```
FILE *output, *outfp[3];
int c, i = 0, j = 0;
  ...
if ( (output = fopen("myfil", "w")) != NULL )
{
  putc(c, output);
    ...
  outfp[0] = stdout;
  outfp[1] = output;
  putc(c, outfp[i++]);   /* unzulässig     */
    ...
  fputc(c, outfp[j++]);  /* ok             */
```

#include <stdio.h>

int putchar(int *c*);

putchar kann anstelle von **putc** aufgerufen werden, wenn die Ausgabe auf **stdout** erfolgen soll.

Implementierung als Makro:

```
#define putchar(c) putc(c, stdout)
```


Funktionen zur Eingabe einzelner Zeichen

#include <stdio.h>

int fgetc(FILE ****stream*);**

int getc(FILE ****stream*);** /* Makro-Version */

fgetc und **getc** sind äquivalente Funktionen. Sie lesen das nächste Zeichen von der mit *stream* verknüpften Datei ein.

getc kann jedoch als Makro implementiert sein. In diesem Fall kann das Argument für *stream* mehrfach bewertet werden, so daß ein Ausdruck mit Seiteneffekten vermieden werden sollte.

Resultatwert: das gelesene Zeichen oder **EOF**, falls das Dateiende erreicht ist oder ein Fehler auftritt.

#include <stdio.h>

int getchar(void);

getchar kann anstelle von **getc** aufgerufen werden, wenn die Eingabe von **stdin** erfolgen soll.

Implementierung als Makro:

```
#define getchar() getc(stdin)
```

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

ungetc schreibt das Zeichen *c* in den internen Puffer der mit *stream* verknüpften Eingabedatei zurück, so daß die nächste Leseoperation dieses Zeichen als das nächste von *stream* einzulesende Zeichen vorfindet.

Der Standard garantiert lediglich, daß (pro Datei) **ein** einzelnes Zeichen zurückgeschrieben werden kann. Eine Implementierung darf jedoch auch das Zurückschreiben mehrerer Zeichen ermöglichen. In diesem Fall werden die durch mehrere **ungetc**-Aufrufe zurückgestellten Zeichen in der umgekehrten Reihenfolge (LIFO) wieder eingelesen.

Ein Aufruf von **ungetc** ist auch dann zulässig, wenn noch kein Zeichen von *stream* gelesen wurde. **EOF** darf nicht zurückgeschrieben werden.

Da die Eingabedatei selbst nicht verändert wird, gehen die zurückgestellten Zeichen verloren, sobald eine der Dateipositionierungsfunktionen **fseek**, **fsetpos** oder **rewind** für *stream* aufgerufen wird.

Resultatwert: *c* oder (bei Fehler) **EOF**

Funktionen zur Ausgabe von Zeichenketten

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

fputs gibt die Zeichenkette *string* (ohne das abschließende '\0'-Zeichen) auf die mit *stream* verknüpfte Datei aus.

Resultatwert: nicht-negativer Wert (z.B. 0) oder (bei Fehler) **EOF**

```
#include <stdio.h>

int puts(const char *string);
```

puts gibt die Zeichenkette *string* (ohne das abschließende '\0'-Zeichen) auf **stdout** aus. Im Gegensatz zu

```
fputs(string, stdout);
```

wird anschließend noch ein Zeilenendezeichen (\n) geschrieben.

Resultatwert: analog zu **fputs**

Funktionen zur Eingabe von Zeichenketten

```
#include <stdio.h>
char *fgets(char *string, int n, FILE *stream);
```

fgets liest eine Zeile (einschließlich des Zeilenendezeichens) von der mit *stream* verknüpften Datei ein und speichert sie (einschließlich des Zeilenendezeichens) in dem Zeichenvektor *string* ab.

Maximal werden *n*-1 Zeichen gelesen. (Enthält eine Zeile mehr als *n*-1 Zeichen, wird also nur der Anfang dieser Zeile übertragen. Der Rest verbleibt ungelesen in der Eingabe.)

An die eingelesene Zeichenfolge wird in jedem Fall ein '\0'-Zeichen angehängt.

Resultatwert: *string* oder **NULL**, falls ein Fehler auftritt oder vor dem Lesen des ersten Zeichens bereits das Dateiende erreicht ist.

```
#include <stdio.h>
char *gets(char *string);
```

gets liest eine Zeile (einschließlich des Zeilenendezeichens) von **stdin** ein und speichert sie (im Gegensatz zu **fgets** **ohne** das Zeilenendezeichen) in dem Zeichenvektor *string* ab. '\0' wird angehängt.

Vorsicht: Stringüberschreitung möglich, da keine Maximallänge angegeben werden kann.

Resultatwert: analog zu **fgets**

Formatierte Ausgabe von Zeichenketten

printf-Konvertierungsspezifikation

darf enthalten:

Flag - linksbündige Ausgabe

Feldbreite minimale Breite des Ausgabefeldes

Genauigkeit maximale Anzahl der Zeichen, die von einer Zeichenkette ausgegeben werden.

K-Typ

K-Typ	**Argument-typ**	**Ausgabe**
s	**char ***	die Zeichen der Zeichenkette ohne Anführungszeichen (")

Beispiel:

```
char s[4] = "Text";   /* '\0' wird nicht     */
                      /*  abgespeichert      */

printf("%.4s", s);
```

Formatierte Eingabe von Zeichenketten

scanf-Konvertierungsspezifikation

darf enthalten:

*	Überspringen des Eingabefeldes
Feldbreite	maximale Breite des Eingabefeldes (Die Angabe ist sinnvoll, um Stringüberschreitungen zu vermeiden.)
K-Typ	

K-Typ	**Argument-typ**	**Wirkung**
c	**char ***	Es werden genau so viele Zeichen in den Zeichenvektor, dessen Adresse übergeben wurde, übertragen, wie durch die *Feldbreite* (Standardvorgabe: 1) angegeben ist. Zwischenraumzeichen werden wie jedes andere Zeichen übertragen. '\0' wird **nicht** hinzugefügt.
s	**char ***	Zunächst werden Zwischenraumzeichen in der Eingabe überlesen. Das erste Zeichen, das kein Zwischenraumzeichen ist, sowie alle folgenden Zeichen bis zum nächsten Zwischenraum, werden in den als Argument übergebenen Zeichenvektor übertragen. '\0' wird hinzugefügt.
[$c_1 \ldots c_n$]	**char ***	Überträgt die längste **nicht-leere** Zeichenfolge, die nur aus den angegebenen Zeichen $c_1 ... c_n$ besteht. '\0' wird hinzugefügt. Zwischenraumzeichen werden wie jedes andere Zeichen behandelt. []$c_2 ... c_n$] enthält auch].
[ˆ$c_1 \ldots c_n$]	**char ***	Überträgt die längste **nicht-leere** Zeichenfolge, die keines der Zeichen $c_1 ... c_n$ enthält. '\0' wird hinzugefügt. Zwischenraumzeichen werden wie jedes andere Zeichen behandelt. [ˆ]$c_2 ... c_n$] schliesst auch] aus.

Beispiel:

```
char s[4];
int count;

while ((count = scanf("%3[01234]", s)) == 1)
  printf("count = %d, s = %s\n", count, s);
printf("count = %d\n", count);
```

Eingabe:

```
123456
```

Ausgabe:

```
count = 1, s = 123
count = 1, s = 4
count = 0
```

Beispiel: Zeilen einlesen

```
char line[80+1];
int c;

*line = '\0';    /* erforderlich, da Leer-     */
                 /*   zeilen nicht übertragen */
                 /*   werden                  */
while (scanf("%80[^\n]", line) != EOF)
{
                 /* Rest der Zeile einschl.   */
                 /*   Zeilenendezeichen lesen:*/
  while (((c = getchar()) != '\n') && c != EOF)
    ;
  /* ... */
  *line = '\0';
}
```


Funktionen zur formatierten Ein-/Ausgabe

#include <stdio.h>

int fprintf(FILE ****stream*, const char ****format***, . . .);**

int sprintf(char ****s*, const char ****format***, . . .);**

- **fprintf** schreibt auf die mit *stream* verknüpfte Datei.
- **sprintf** schreibt auf den Zeichenvektor, auf den der Zeiger *s* verweist. '\0' wird hinzugefügt.

ansonsten analog zu **printf**

Beispiel:

```
fprintf(stderr, "Error: ...");
```

#include <stdio.h>

int fscanf(FILE ****stream*, const char ****format***, . . .);**

int sscanf(const char ****s*, const char ****format***, . . .);**

- **fscanf** liest von der mit *stream* verknüpften Datei.
- **sscanf** liest von dem Zeichenvektor *s*.

ansonsten analog zu **scanf**

Umwandlungsfunktionen

```
#include <stdlib.h>
long strtol(const char *str, char **endptr, int base);
```

strtol wandelt die Zeichenkette *str* in einen **long**-Wert um, indem die Zeichen interpretiert werden, die am Anfang der Zeichenkette eine in der Basis *base* dargestellte Zahl spezifizieren. Eine Dezimalzahl (*base*==10 oder *base*==0) muß der Form

[+|-]*Dezimalkonstante*

entsprechen. Vorausgehende Zwischenraumzeichen werden ignoriert.

Der **char**-Zeiger, dessen Adresse als *endptr* übergeben wurde, verweist anschließend auf das erste Zeichen in *str*, das nicht umgewandelt werden konnte. Wird dieser Zeiger auf den Rest der Zeichenkette nicht benötigt, kann beim Aufruf **NULL** für *endptr* angegeben werden.

base spezifiziert die Basis eines Zahlensystems:

2≤*base*≤36

Die Ziffern 10-35 einer umzuwandelnden Zahl werden durch a-z (A-Z) dargestellt. *base*==0 ermöglicht die Umwandlung von Dezimal-, Oktal- (führende Null) und Hexadezimalzahlen (Präfix 0x bzw. 0X (auch bei *base*==16 erlaubt)).

Resultatwert: umgewandelte Zahl oder 0L, falls keine Umwandlung möglich ist. (Bei zu großen Zahlen wird **LONG_MAX** bzw. **LONG_MIN** zurückgegeben und **errno** auf **ERANGE** gesetzt.)


```
#include <stdlib.h>
unsigned long strtoul(const char *str,
                      char **endptr, int base);
```

analog zu **strtol**, jedoch wird *str* in einen **unsigned long**-Wert umgewandelt. (Bei zu großen Zahlen wird **ULONG_MAX** zurückgegeben und **errno** auf **ERANGE** gesetzt.)

```
#include <stdlib.h>

double strtod(const char *str, char **endptr);
```

strtod wandelt die Zeichenkette *str* in einen **double**-Wert um, indem die Zeichen interpretiert werden, die am Anfang der Zeichenkette eine Zahl der Form

[+|-]*Gleitkommakonstante*
(ohne *F-Suffix*; außerdem darf der Dezimalpunkt weggelassen werden)

spezifizieren. Vorausgehende Zwischenraumzeichen werden ignoriert.

endptr hat dieselbe Bedeutung wie bei **strtol**.

Resultatwert: umgewandelte Zahl oder 0.0, falls keine Umwandlung möglich ist. (Bei *Overflow* wird **HUGE_VAL** (mit korrektem Vorzeichen), bei *Underflow* 0.0 zurückgegeben. In beiden Fällen wird **errno** auf **ERANGE** gesetzt.)

```
#include <stdlib.h>
int atoi(const char *str);
```

atoi ist äquivalent zu

```
(int) strtol(str, (char **) NULL, 10);
```

```
#include <stdlib.h>

long atol(const char *str);
```

atol ist äquivalent zu

```
strtol(str, (char **) NULL, 10);
```

```
#include <stdlib.h>

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

atof ist äquivalent zu

```
strtod(str, (char **) NULL);
```


Funktionen in <string.h>

- Kopierfunktionen

  strcpy kopiert String in einen anderen

  strncpy kopiert *n* Zeichen eines Strings in einen anderen

  memcpy kopiert Speicherbereich in einen anderen

  memmove kopiert Speicherbereich in einen anderen, mit dem er überlappen darf[20]

- Verkettungsfunktionen

  strcat verkettet 2 Strings

  strncat verkettet String mit *n* Zeichen eines anderen

- Vergleichsfunktionen

  strcmp vergleicht 2 Strings

  strncmp vergleicht String mit den ersten *n* Zeichen eines anderen

  memcmp vergleicht 2 Speicherbereiche

[20] Bei allen anderen Kopierfunktionen ist das Ergebnis für überlappende Strings (bzw. Speicherbereiche) nicht definiert.

- Suchfunktionen

strchr	findet erstes Vorkommen eines Zeichens in einem String
strcspn	liefert Anzahl der Zeichen am Anfang eines Strings, die sämtlich **nicht** in einem anderen String vorkommen
strpbrk	findet erstes Vorkommen in einem String von irgendeinem Zeichen aus einem anderen String
strrchr	findet **letztes** Vorkommen eines Zeichens in einem String
strspn	liefert Anzahl der Zeichen am Anfang eines Strings, die sämtlich in einem anderen String vorkommen
strstr	findet einen String innerhalb eines anderen Strings
strtok	zerlegt einen String in „Grundsymbole“ (**Tok**en)
memchr	findet erstes Vorkommen eines Zeichens in einem Speicherbereich

- Sonstige Stringfunktionen

strlen	liefert die Länge eines Strings
strerror	liefert die Fehlermeldung zu einer Fehlernummer
memset	füllt Speicherbereich mit Kopien eines Zeichens auf

Kopierfunktionen

char *strcpy(char ****string1*, const char ****string2***);**

- kopiert Zeichenkette auf die *string2* zeigt (einschließlich des Zeichens '\0') in den Zeichenvektor, auf den *string1* verweist
- gibt *string1* zurück

Beispiel:

siehe Programm „Verketten von 2 Strings“

char *strncpy(char ***_string1_, const char ***_string2_**,**
size_t _n_**);**

- kopiert die ersten _n_ Zeichen der Zeichenkette _string2_ in den Zeichenvektor _string1_
- falls **strlen(**_string2_**)<**_n_ gilt, wird _string1_ mit Null-Bytes ('\0') bis zur Länge _n_ aufgefüllt
- gibt _string1_ zurück

Vorsicht: Dem kopierten String folgt **kein** '\0'-Zeichen, wenn **strlen(**_string2_**)**≥_n_ gilt.

void *memcpy(void ***_region1_, const void ***_region2_**,**
size_t _n_**);**

- kopiert _n_ Bytes von _region2_ nach _region1_. Im Gegensatz zu **strncpy** wird der Kopiervorgang nicht durch ein '\0'-Zeichen in _region2_ beendet.
- gibt _region1_ zurück

Beispiel:

```
int a[3], b[3] = { 1, 0, 3 };

memcpy(a, b, 3*sizeof(a[0]));
                   /* entspricht:
                        for ( i=0; i<3; i++ )
                          a[i] = b[i];
                   */
```

void *memmove(void ***_region1_,**
const void ***_region2_, size_t** _n_**);**

- gleiche Funktion wie **memcpy**, jedoch dürfen die Speicherbereiche _region1_ und _region2_ überlappen

Beispiel:

```
/* Zeichenkette um 1 Zeichen     */
/*   nach links rotieren         */

char nums[] = "0123456789", sav;
int len;

len = strlen(nums);
sav = *nums;
memmove(nums, nums+1, --len);
nums[len] = sav;
```


Verkettungsfunktionen

char *strcat(char ***_string1_, const char ***_string2_**);**

- kopiert alle Zeichen von _string2_ (einschließlich des '\0'-Zeichens) an das Ende von _string1_. (Das '\0'-Zeichen am Ende von _string1_ wird duch das erste Zeichen von _string2_ überschrieben.)
- gibt _string1_ zurück

Beispiel:

siehe Programm „Verketten von 2 Strings“

char *strncat(char **string1*, **const char** **string2*,
size_t *n***);**

- kopiert **höchstens** die ersten *n* Zeichen der Zeichenkette *string2* an das Ende von *string1*
- Kopiervorgang wird beendet, wenn *n* Zeichen kopiert wurden oder das Ende ('\0'-Zeichen) von *string2* erreicht wird.
- an den Resultatstring wird in jedem Fall ein '\0'-Zeichen angehängt (Der Zeichenvektor auf den *string1* verweist muß daher mindestens **strlen(***string1***)+***n***+1** Bytes groß sein.)
- gibt *string1* zurück

Beispiel:

siehe Programm „MS-DOS-Dateiname“

Vergleichsfunktionen

int strcmp(const char **string1*, **const char** **string2***);**

- vergleicht die Zeichenketten *string1* und *string2* Zeichen für Zeichen (einschließlich der '\0'-Zeichen)
- bei Ungleichheit entscheidet das erste Paar ungleicher Zeichen (c_1, c_2) über den Resultatwert
- Resultatwert:

< 0	wenn $c_1 < c_2$ (gleichbedeutend mit *string1* < *string2*)
0	wenn beide Strings identisch sind
> 0	wenn $c_1 > c_2$ (gleichbedeutend mit *string1* > *string2*)

Beispiel:

```
#define BUFLEN 80
    ...
  char buf[BUFLEN+1];
    ...
  while( (gets(buf) != NULL) &&
           strcmp(buf,"quit") ) ...
```

int strncmp(const char ***_string1_,**
const char ***_string2_, size_t** _n_**);**

- vergleicht _string1_ zeichenweise mit **höchstens** den ersten _n_ Zeichen von _string2_. (Zeichen **nach** einem '\0'-Zeichen werden nicht verglichen.)
- ansonsten analog zu **strcmp**

Beispiel:

```
strncmp(buf, "help ", 5)
```


int memcmp(const void ***_region1_,**
const void ***_region2_, size_t** _n_**);**

- vergleicht byteweise die Speicherbereiche _region1_ und _region2_
- _n_ spezifiziert die Größe der Speicherbereiche in Bytes
- Resultatwert analog zu **strcmp**

Beispiel:

```
char carray[10];
int  iarray[10];
char *s = "ab";
  ...
strcmp(carray, s)    /* OK */
memcmp(carray, s, 3) /* OK */
strcmp(iarray, s)    /* Illegal, Typ des 1.  */
                     /*   Arguments nicht    */
                     /*   char *             */
memcmp(iarray, s, 3) /* OK, Arg. werden nach */
                     /*   void *             */
                     /*   umgewandelt        */
```

Suchfunktionen

char *strchr(const char *_string_, int _character_);

- sucht nach dem Zeichen *character* innerhalb der Zeichenkette *string* (Das ’\0’-Zeichen am Ende von *string* wird in die Suche mit einbezogen.)
- liefert Zeiger auf das erste Zeichen *character* innerhalb des Strings oder **NULL**, falls das Zeichen nicht gefunden wird.

Beispiel:

```
char *line = "help help";
  ...
strchr(line, ’ ’);   /* liefert line+4 */
```


size_t strcspn(const char *_string1_,
const char *_string2_);

- liefert die Länge des Teilstrings am Anfang von *string1*, der nur aus Zeichen besteht, die **nicht** in *string2* enthalten sind. (Diese Länge stimmt mit dem Index des ersten Zeichens in *string1* überein, das auch in *string2* vorkommt.)

Beispiel:

```
char *line = "Temperatur = 123";
  ...
strcspn(line, "0123456789")  /* ergibt 13 */
```

**char *strpbrk(const char *_string1_,
const char *_string2_);**

- liefert Zeiger auf das erste Zeichen in *string1*, das auch in *string2* (*break string*) vorkommt
- Falls kein Zeichen der Zeichenkette *string2* in *string1* vorkommt, wird **NULL** zurückgegeben

Beispiel:

```
char *line = "a, b, c oder d";
  ...
strpbrk(line, ",;")  /* liefert line+1 */
```


char *strrchr(const char *_string_, int _character_);

- sucht nach dem **letzten** (am weitesten rechts vorkommenden) Zeichen *character* innerhalb der Zeichenkette *string*
- liefert Zeiger auf das letzte Zeichen *character* innerhalb des Strings oder **NULL**, falls das Zeichen nicht gefunden wird

Beispiel:

```
char *line = "a, b, c oder d";
  ...
strrchr(line, ',')  /* liefert line+4 */
```

size_t strspn(const char **string1*,
const char **string2*);

- liefert die Länge des Teilstrings am Anfang von *string1*, der nur aus Zeichen besteht, die in *string2* enthalten sind. (Diese Länge stimmt mit dem Index des ersten Zeichens in *string1* überein, das **nicht** in *string2* vorkommt.)

Beispiel:

```
char *line = "00021 Zeile 21";
  ...
strspn(line, "0123456789")  /* ergibt 5 */
```


Zeiger und Vektoren
Suchfunktionen

char *strstr(const char **string1*,
const char **string2*);

- sucht nach *string2* innerhalb der Zeichenkette *string1* (Das ’\0’-Zeichen am Ende von *string2* wird dabei nicht berücksichtigt.)
- liefert Zeiger auf das erste Vorkommen von *string2* oder **NULL**, falls *string2* nicht gefunden wird. (Falls *string2* ein Nullstring ist, wird *string1* zurückgegeben.)

Beispiel:

```
char *str1 = "needle in a haystack";
  ...
strstr(str1, "hay")  /* liefert Zeiger auf   */
                     /*   "haystack"         */
```

char *strtok(char *_string1_, const char *_string2_);

- zerlegt die Zeichenkette *string1* in „Grundsymbole“ (**Tok**en)
- *string2* enthält sämtliche Zeichen, die benutzt werden können, um Token zu begrenzen.
- die Zerlegung erfolgt durch eine Folge von Aufrufen der Funktion **strtok**:
  - ☐ beim ersten Aufruf wird *string1* übergeben. **strtok** sucht nach dem ersten Zeichen, das nicht in *string2* enthalten ist. Wird ein solches Zeichen gefunden, dann ist es das erste Zeichen des ersten Tokens. Andernfalls kann *string1* nicht in Token zerlegt werden. In diesem Fall gibt **strtok NULL** zurück.

    Falls der Anfang des ersten Tokens gefunden wurde, wird nach einem Zeichen gesucht, das in *string2* enthalten ist. Dieses wird durch ein '\0'-Zeichen ersetzt, um das Ende des ersten Tokens zu markieren. **strtok** gibt einen Zeiger auf dieses Token zurück, sichert jedoch zuvor unter Zuhilfenahme einer statischen Zeigervariablen den Rest der Zeichenkette *string1*.

  - ☐ Folgeaufrufe werden dadurch gekennzeichnet, daß **NULL** an den Parameter *string1* übergeben wird. **strtok** sucht dann in dem Teilstring, der im vorausgehenden Aufruf gesichert wurde, nach dem nächsten Token und gibt einen Zeiger auf dieses Token zurück oder **NULL**, falls kein Token gefunden wird.
- Die Begrenzungszeichen für die Token in *string2* können bei jedem Aufruf verschieden sein.

Beispiel:

```
char *token;
char string[] = "a string, of, ,tokens";
  ...
token = strtok(string, ",");
do
{
  printf("token: <%s>\n", token);
} while ( token = strtok(NULL, ",") );
```

Ausgabe:

```
token: <a string>
token: < of>
token: < >
token: <tokens>
```

void *memchr(const void **region*, int *character*, size_t *n*);

- sucht nach dem Zeichen *character* innerhalb des Speicherbereichs *region*
- *n* spezifiziert die Größe des Speicherbereiches in Bytes
- Resultatwert analog zu **strchr**

Sonstige Stringfunktionen

void *memset(void **buffer*, int *character*, size_t *n*);

- füllt den Speicherbereich *buffer* mit Kopien des Zeichens *character*
- *n* spezifiziert die Größe des Speicherbereiches in Bytes
- gibt *buffer* zurück

Beispiel:

```
#define BUFLEN 80
  char buf[BUFLEN+1];
    ...
  memset(buf, '-', BUFLEN);
  buf[BUFLEN] = '\0';
```

Beispielprogramm: Verketten von 2 Strings

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

char *
combine(char **a, char **b)
{
  if ( *a == NULL )
  {
    *a = *b;
    *b = NULL;
    return ( *a );
  }
  else
    if ( *b == NULL )
      return ( *a );
  if ( (*a = realloc(*a,strlen(*a)+strlen(*b)+1))
       != NULL
     )
    return ( strcat(*a, *b) );
  else
    return ( NULL );
}
```



```
char *
copy(char *s)    /* Copy a string into  */
                 /*   a malloc'ed hole. */
{
  size_t len;
  char *ret;

  if ( !(len = strlen(s)) )
    return ( NULL );
  if ( (ret = (char *) malloc(len+1)) == NULL )
    return ( NULL );
  return ( strcpy(ret, s) );
}

int main( void )
{
  char *a, *b;

  a = copy("A fine string. ");
  b = copy("Another fine string. ");

  puts( combine(&a, &b) );
  exit ( EXIT_SUCCESS );
}
```

Beispielprogramm: MS-DOS-Dateiname

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

char *
dosfname
  (
    char       *dosname, /* Dateiname      */
    const char *name,    /* Name           */
    const char *ext      /* Suffix (file   */
                         /*   extension)   */
  )
{
  *dosname = '\0';
  /* strncpy() garantiert kein '\0' */
  strncat(dosname, name, 8);
  if ( (ext != NULL) && *ext )
  {
    strcat(dosname, ".");
    strncat(dosname, ext, 3);
  }
  return ( dosname );
}
```



```
int main( void )
{
  char fileid[13];
  FILE *outfil;

  if ( (outfil =
         fopen(dosfname(fileid, "A_LONG_FILENAME"
           , "EXTENSION"), "w")
       ) != NULL
     )
  {
    fputs(fileid, outfil);
   /* Aus historischen Gruenden gibt fputs den */
   /* String unveraendert aus, waehrend puts   */
   /* das Zeichen '\n' automatisch anhaengt.   */
    putc('\n', outfil);
  }
  else
    puts(fileid);
  exit ( EXIT_SUCCESS );
}
```

Aufgaben

1. Das folgende Programm ist inkorrekt. Wenn man es mit dem C for AIX Compiler ausführt, wird folgendes ausgegeben:

   ```
   abcSonne onne abcSonne
   ```

   Diese Ausgabe ist durchaus verständlich und sagt etwas über den verwendeten Compiler aus. Erklären Sie was passiert und korrigieren Sie das Programm.

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

   int main( void )
   {
     char *p1 = "abc", *p2 = "Sonne";

     printf("%s %s %s\n", p1, p2, strcat(p1,p2));
     exit(EXIT_SUCCESS);
   }
   ```

2. Schreiben Sie eine Funktion **replace**, die in einer Zeichenkette *string* das Zeichen *oldch* durch das Zeichen *newch* ersetzt. *oldch* soll an **allen** Positionen, wo es in *string* vorkommt, geändert werden. Die Funktion soll einen Zeiger auf *string* zurückgeben.

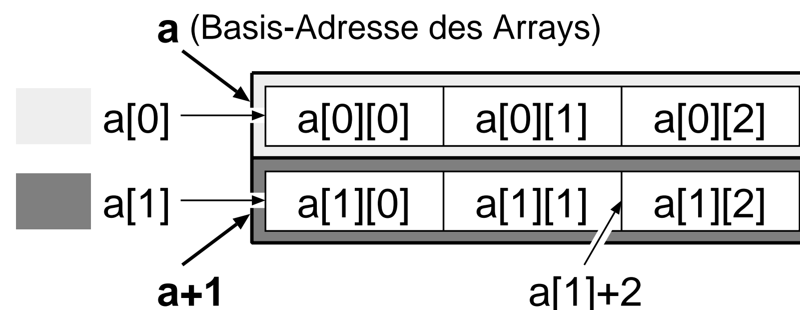
Mehrdimensionale Vektoren (Felder, *Arrays*)

Die Elemente eines Vektors können wiederum Vektoren sein.

Beispiele:

```
int a[2][3];       /* zweidimensionales Feld   */
float b[2][2][2]; /* dreidimensionales Feld   */
```

Die Elemente eines Arrays werden zeilenweise (allg.: der letzte Index variiert am schnellsten) in einem zusammenhängenden Speicherbereich angeordnet.



Durch die Vereinbarung

```
int a[2][3];   ⇔   int (a[2])[3];
                          ↕
                   int  name[3];
```

werden die Elemente `a[i]` des Vektors `a` als **int**-Vektoren der Länge 3 vereinbart:

`a[i]`: Vektor mit 3 Elementen des Typs **int** → Zeiger auf **int**

`a`   : Vektor, dessen Elemente **int**-Vektoren der Länge 3 sind → Zeiger auf einen **int**-Vektor der Länge 3

Zugriff auf ein Element:

```
      a[i][j]    ⇎   a[i,j] ≡ a[j]
        ⇕
    *(a[i]+j)
        ⇕
   *(*(a+i)+j)
        ⇕
    (*(a+i))[j]
```

Adreß-Umrechnungsfunktion:

```
*(&a[0][0] + 3*i + j)   ⇔   (&a[0][0])[3*i + j]
             ↑
        Spaltenanzahl
```

Beachte: `&a[0][0]` hat den Typ **int ***.

In der Adreß-Umrechnungsfunktion kommen die Längen aller Dimensionen (außer der ersten) vor.

⇒ In allen Fällen, in denen die Größe des Feldes bei einem eindimensionalen Vektor weggelassen werden darf (Funktionsparameter, externe Deklarationen), kann bei einem mehrdimensionalen Feld die **erste** Dimensionsangabe weggelassen werden.
Die Längen aller anderen Dimensionen **müssen** angegeben werden. (Als Dimensionslängen sind nur **konstante** Ausdrücke erlaubt!)

Beispiel:

```
void printMatrix(int m[][3], int rowc)
{
  ...
    m[i][j]
  ...
}
```

Bei der Vereinbarung eines Formalparameters gilt:

```
int m[][3]    ⇔    int (*m)[3]
```

`int (*m)[3]` vereinbart einen Zeiger auf einen **int**-Vektor der Länge 3.

```
 float f[][2][2]    ⇔    float (*f)[2][2]
        ⇕
float f[2][2][2]
```

Die Länge der ersten Dimension wird vom Compiler ignoriert.

Problem:

Wie kann eine Funktion implementiert werden, die die Bearbeitung unterschiedlich dimensionierter Matrizen erlaubt?

erster Lösungsvorschlag:

Die als Argument übergebene Matrix wird als eindimensionaler Vektor angenommen. Zum Zugriff auf ein Matrixelement muß dann die Position dieses Elementes innerhalb des Vektors berechnet werden. Hierzu wird die Adreß-Umrechnungsfunktion verwendet. Da die Spaltenanzahl in der Adreß-Umrechnungsfunktion variabel angegeben werden kann, erlaubt diese Methode die Bearbeitung beliebig dimensionierter Matrizen.

Beispiel:

```
void printMatrix(void *m_, int rowc, int colc)
{
  int *m = (int *) m_;
  ...
    m[colc*i+j]    /* Zugriff auf ein Element */
 /* m[i][j]        /* unzulässig, da ’m’      */
                   /*   nicht als Matrix      */
                   /*   vereinbart ist        */
  ...
}
```


Aufruf:

```
  ...
int a1[10][10], a2[3][2];
  ...
printMatrix(a1, 10, 10);
  ...
printMatrix(a2, 3, 2);
```

Allgemein gilt für den Zugriff auf ein Element eines n-dimensionalen Feldes $\mathbf{a}[dim_1] \ldots [dim_n]$ die folgende Adreß-Umrechnungsfunktion:

$$\begin{aligned}\mathbf{(m = (int\ *)\ a)[}\ & dim_n * \ldots * dim_3 * dim_2 * i_1 + \\ & dim_n * \ldots * dim_3 * i_2 + \\ & \ldots \\ & dim_n * i_{n-1} + \\ & i_n\ ] \quad \rightarrow \quad \mathbf{a}[i_1] \ldots [i_n]\end{aligned}$$

zweiter Lösungsvorschlag: folgt später

Initialisierung mehrdimensionaler Vektoren

Wenn eine Komponente eines zusammengesetzten Typs selbst ein zusammengesetzter Typ ist, gelten die Initialisierungsregeln rekursiv für jede Komponente.

Das heißt für ein zweidimensionales Feld, daß jede Zeile wie ein eindimensionaler Vektor initialisiert wird.

Beispiele:

```
int a[][2] = {
   {1, 2},
   {3, 4}
};

float b[4][3] = {        /* initialisiert       */
   {1}, {2}, {3}, {4}    /*   1. Spalte von 'b' */
};                       /* (Die restlichen     */
                         /*   Elemente erhalten */
                         /*   den Wert 0.0f.)   */
```

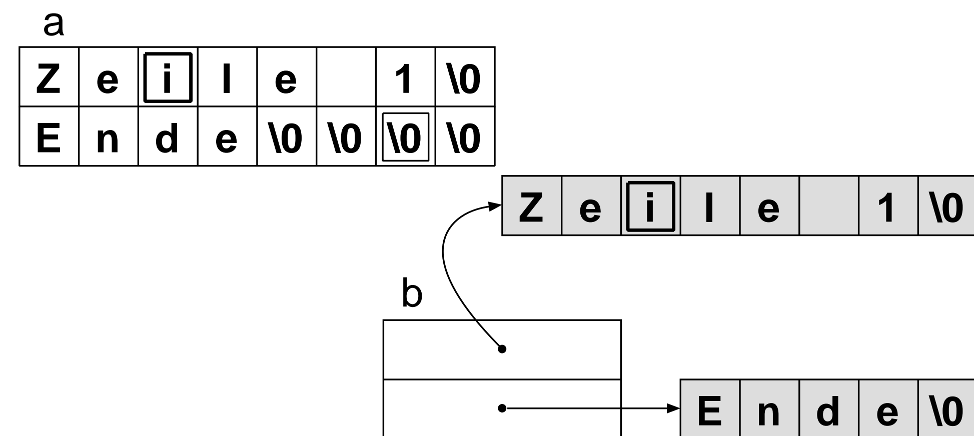
Wenn die Initialisierung einer Komponente mit zusammengesetztem Typ nicht mit { beginnt, werden die Subkomponenten initialisiert, indem aus der Initialisierer-Liste so viele Initialisierer wie nötig genommen werden. Etwa verbleibende Initialisierer werden für die Initialisierung der nächsten Komponente verwendet.

Beispiel:

```
float b[4][3] = {        /* initialisiert       */
   1, 2, 3, 4            /*   1. Zeile + 1. Ele-*/
};                       /*   ment der 2. Zeile */
```


Gegenüberstellung von mehrdimensionalen Vektoren und Vektoren von Zeigern:

```
char a[][8] = {"Zeile 1", "Ende"};
char *b[] = {"Zeile 1", "Ende"};
```




```
printf("%s", a[0]);          printf("%s", b[0]);
   /* -> Zeile 1 */             /* -> Zeile 1 */

a[0][2]                      b[0][2]
/* zum Zugriff auf    */    /* benutzt keine      */
/* dieses Element     */    /* Adress-Umrechnungs-*/
/* wird Adress-Um-    */    /* funktion;          */
/* rechnungsfunktion  */    /* Verarbeitung von   */
/* verwendet          */    /* 'b' schneller ?    */

a[1][6] /* ok */             b[1][6] /* unzulässig */
```

(Footnote marker [21] attached to b[0][2].)

[21] ⇒ zweiter Lösungsvorschlag für das Problem von Seite 277: Anstelle der Matrix wird ein Hilfsvektor mit Zeigern übergeben, die auf die Zeilen der Matrix verweisen, indem sie jeweils auf das erste Element einer Zeile zeigen.

Aufgaben

1. Was gibt das folgende Programm aus?

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

void prt( int (*a)[3] )
{
  printf("%d %d %d %d\n",
    a[1][0], **(a+1), *(*a+1), **a+1);
  return;
}

int main( void )
{
  int t[3][3] = { { 2,  4,  6},
                  {12, 14, 16},
                  {22, 24, 26} };

  prt(t);
  exit(EXIT_SUCCESS);
}
```

2. Zeigen Sie anhand eines Beispiels, wie eine Matrix M mit 12 Spalten und einer erst zur Laufzeit berechenbaren Anzahl von Zeilen mit Hilfe der dynamischen Speicherallokation in der benötigten Größe erzeugt werden kann.
   Es soll möglich sein die Matrix auf die übliche Art und Weise zu indizieren ($M[i][j]$).

Zeiger auf Funktionen

Auch hier imitiert die Syntax der Vereinbarung die Syntax der (für das vereinbarte Objekt typischen) Verwendung in einem Ausdruck.

Vereinbarung:

erfolgt analog zur Vereinbarung einer Funktion, mit dem Unterschied, daß der *Funktionsname* durch
(* *type_qualifier_list* $_{opt}$ *Zeigername*)
ersetzt wird.

Beispiel:

```
double f1(double x);  /* f1: Funktion mit    */
                      /*   Resultat 'double' */

double (*fp)(double x);
                      /* fp: Zeiger auf Funk-*/
                      /*   tion mit Resultat */
                      /*   'double'          */
```

Bei dieser Vereinbarung können die Parameternamen (wie in einer Funktions**deklaration**) weggelassen werden:

```
double (*fp)(double);
```

Analog zur impliziten Typumwandlung von Ausdrücken des Typs “Vektor” in einen Zeigertyp, gilt für Funktionen:

> Ein Ausdruck vom Typ “Funktion mit Resultattyp T” wird in einen Wert des Typs “Zeiger auf Funktion, die T liefert” umgewandelt.
> (Ausnahme: Ausdruck wird als Operand des Adreß-operators (&) verwendet.[22])

⇒ ein Funktionsname ist ein **konstanter Zeiger** auf die Funktion

umgekehrt gilt:

ein Wert vom Typ “Zeiger auf Funktion” kann wie ein Funktionsname verwendet werden (d.h. Zeiger auf Funktionen brauchen nicht dereferenziert werden.)

Ein Objekt vom Typ “Funktion” darf nicht als Operand des **sizeof**–Operators verwendet werden.

[22] Wegen dieser Ausnahme ist es möglich, diese implizite Umwandlung auch explizit auszudrücken.

Zeiger und Vektoren
Zeiger auf Funktionen

Verwendung von Zeigern auf Funktionen:

1. zum Aufruf einer Funktion
2. in Zuweisung (auf beiden Seiten)
3. als Argument in einem Funktionsaufruf
4. als Resultatwert einer Funktion

Beispiele:

zu 1.+2.:

```
#include <math.h>
double erg;
double (*fp)(double x) = fabs;
                           /* = &fabs; */
fp(-1.0); ⇔ (*fp)(-1.0); ⇔ fabs(-1.0);
erg = (fp = sqrt)(2.0); ⇔ erg = sqrt(2.0);
```

zu 3.:

Vereinbarung einer Funktion, die als Argument einen Zeiger auf eine Funktion erwartet:

```
void u1(char (*fp)(long lval), float fval);
```

```
/* fp: Parameter vom Typ "Zeiger auf Funktion */
/*     mit einem Argument vom Typ 'long' und  */
/*     Resultattyp 'char'"                    */
```

in Deklaration dürfen Parameternamen wegfallen:

```
void u1(char (*)(long), float);
```

Wird ein **Parameter** mit dem Typ “Funktion mit Resultattyp T” vereinbart, so wird die Vereinbarung in “Zeiger auf Funktion, die T liefert” abgeändert(, d.h. der Formalparameter für eine Funktion ist immer eine Zeiger**variable**):

```
void u1(char f(long lval), float fval);
void u1(char (long), float);
```

zu 4.:

Funktion mit Resultattyp “Zeiger auf Funktion”:

```
int (*u2(void))(double);
```

```
/* u2: Funktion ohne Parameter, die einen     */
/*     Zeiger auf eine Funktion mit einem     */
/*     Argument vom Typ 'double' und Resul-   */
/*     tatwert 'int' liefert                  */
```

zu 3.+4.:

```
int (*u( char (*) (long),
         float
       )
    ) (double);
```

```
/* u: Funktion mit zwei Argumenten. Das erste */
/*    Argument ist ein Zeiger auf eine Funk-  */
/*    tion, die einen 'long'-Wert als Argu-   */
/*    ment erwartet und einen 'char'-Wert     */
/*    liefert, das zweite ein 'float'-Wert.   */
/*    Die Funktion 'u' gibt einen Zeiger auf  */
/*    eine Funktion zurück. Diese Funktion    */
/*    erwartet als einziges Argument einen    */
/*    'double'-Wert und gibt einen 'int'-Wert */
/*    zurück.                                 */
```


Ein Zeiger auf eine Funktion kann **ausschließlich**[23] in einen Zeiger auf einen anderen Typ von Funktion umgewandelt werden. Die Rückumwandlung in den ursprünglichen Typ ergibt den ursprünglichen Zeiger.

Beispiel:

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

#define MAXLEN   1000
#define MAXLINES 2000


void sortiere
  (
    void *v[], int vlen,
    int (*cmp)(const void *, const void *)
  )
{
  ...
      if ( (*cmp)(v[j], v[min]) < 0 )
  ...
}
```

[23] Da es auf Computern mit segmentiertem Speicher häufig Speichermodelle gibt, bei denen Zeiger auf Datenobjekte eine andere Größe haben als Zeiger auf Funktionen, sind andere Umwandlungen (auch die Umwandlung in einen generischen Zeiger) nicht zulässig.

```
int intcmp              /* Funktion, die zwei   */
  (                     /*   'int'-Werte ver-   */
    const void *val1,   /*   gleicht; Resultat- */
    const void *val2    /*   wert analog zu     */
  )                     /*   'strcmp'           */
{
  return(*((int *)val1) - *((int *)val2));
}


int main( void )
{
  int *intv[MAXLEN];
  char *line[MAXLINES];
  ...

  sortiere((void **) intv, MAXLEN, intcmp);
  ...

  sortiere((void **) line, MAXLINES,
    (int (*)(const void *,const void *)) strcmp);
  ...
}
```


Aufgaben

1. Beschreiben Sie (in Worten) die Objekte, die durch die folgenden Vereinbarungen gegeben sind:

```
a.  double (*p[3])(void)
b.  double *(*q)[3]
c.  enum e *a(int (*b)(int))
d.  void *funct(
      const void *key,
      const void *base,
      size_t n,
      size_t size,
      int (*comp)(const void *, const void *)
               );
e.  void ( *fp(int sig, void (*func)(int)) )
      ( int );
```

2. **Vektor mit Zeigern auf Funktionen**

   Gegeben seien die folgenden fünf Funktionen, die als Argument einen generischen Zeiger erwarten und keinen Funktionswert zurückgeben:

```
process_stmt_begin
process_stmt_help
process_stmt_set
process_stmt_print
process_stmt_end
```

   Vereinbaren Sie einen Vektor, dessen Elemente konstante Zeiger auf diese Funktionen sind.

Strukturen

Eine Struktur ist ein aus einer oder mehreren benannten Komponenten (*members*) bestehendes Objekt. Die Komponenten können verschiedene Datentypen haben (auch zusammengesetzte Typen wie Vektoren und auch Strukturen sind erlaubt).

Strukturvereinbarungen

Syntax:

> **struct** *type_tag*$_{opt}$ **{***member_decl_list***}**
>
> **struct** *type_tag*

type_tag (*structure tag*, Etikett)
: benennt den Strukturtyp

member_decl_list
: Folge von Deklarationen für die Komponenten der Struktur. Die Deklarationen unterscheiden sich von anderen Vereinbarungen nur dadurch, daß keine Initialisierer erlaubt sind.

Initialisierung:

erfolgt durch eine in { } eingeschlossene Liste von konstanten Ausdrücken oder durch einen Ausdruck mit dem gleichen Strukturtyp (letzteres nur bei Strukturen mit der Speicherklasse **auto**).

Beispiele:

```
struct point {      /* vereinbart den Struktur-*/
  double x;         /*   typ 'struct point'    */
  double y;         /*   (Typdefinition)       */
}/* hier ist Strukturtyp
      vollständig definiert */;

struct point lu = {5.0, 8.0},
             ro = {25.0, 20.0};
    /* vereinbart und initialisiert die       */
    /*   Strukturen 'lu' und 'ro' (Definition)*/

/* Alternative Definition (ohne Initialisie-  */
/* rung):                                     */
struct { double x, y; } lu, ro;
    /* vereinbart Strukturtyp und ist gleich- */
    /*   zeitig Variablendefinition           */

/* Struktur, die eine Struktur als Komponente */
/* enthält:                                   */
struct fenster {
  struct point lu;
  struct point ro;
} window = { {5.0, 8.0}, {25.0, 20.0} };
```

Enthält die Definition eines Strukturtyps kein Etikett (*type_tag*), dann sind Bezugnahmen auf diesen Typ durch nachfolgende Vereinbarungen nicht möglich, da durch eine Definition dieser Art immer ein neuer (von allen anderen Strukturtypen verschiedener) Strukturtyp vereinbart wird.

Beispiel:

```
int main( void )
{
  struct {int a; int b;} s1 = {10, 99};
  struct {int a; int b;} s2 = s1;
      /* unzulässig, denn 's2' und 's1' haben */
      /*   nicht den gleichen Typ              */

  /* ... */

  struct {int a; int b;} s3 = {10, 99},
                         s4 = s3;
      /* ok (Auf top-level wäre 's3' als      */
      /*   Initialisierer jedoch unzulässig.) */

  /* ... */
```


Zulässige Operationen

- Zuweisung

  Strukturen können als Ganzes einander zugewiesen werden (Eine Struktur ist ein l-Wert, wenn keine Komponente das **const**-Attribut hat und alle Komponenten mit einem Struktur- oder Vereinigungstyp l-Werte darstellen.[24] Beide Operanden müssen den gleichen Strukturtyp haben.)

  – bei der Übergabe einer Struktur an eine Funktion wird die gesamte Struktur an den Formalparameter zugewiesen (*call by value*)
  ⇒ i. allg. ist es effizienter einen Zeiger auf die Struktur zu übergeben (*call by reference*)

  – eine Struktur kann als Resultatwert von einer Funktion zurückgegeben werden

  Beispiele:

```
lu = ro;
void clipline(struct point p1,
                struct point p2);
struct point transform(struct point);
```

- Adresse einer Struktur bestimmen

  Beispiel:

```
&window         /* Resultattyp:         */
                /*   struct fenster *   */
```

[24] Man beachte, daß auch Strukturen, die einen Vektor als Komponente enthalten, l-Werte sein können.

- Zugriff auf Komponenten

  *Struktur***.***Komponentenname*

  Struktur kann auch ein Ausdruck (z.B. ein Funktionsaufruf) sein.

  Beispiele:

```
lu.x             /* -> 5.0; Ausdruck ist    */
                 /*   ein l-Wert            */
window.lu        /* Struktur vom Typ        */
                 /*   struct point (l-Wert) */
window.lu.x      /* -> 5.0                  */
div(7,2).quot    /* kein l-Wert             */
```

Beachte: Vergleiche von Strukturen sind nicht zulässig.

Unvollständiger Strukturtyp

Eine Typangabe der Form

struct *type_tag*

darf bereits **vor** der Definition dieses Typs verwendet werden, wenn dabei die **Größe** der Struktur **nicht benötigt** wird.

Strukturen
Unvollständiger Strukturtyp

Beispiel:

```
struct point init( void );
     /* 'struct point' ist unvollständiger Typ */

struct point *ptr_to_struct;

struct point {
  double x;
  double y;
}/* hier ist der Typ vollständig definiert */;

struct point p;     /* hier ist kein unvoll-    */
                    /*   ständiger Typ erlaubt  */

int fkt( void ) {
  struct point {    /* Neuer Typ im Gültigkeits-*/
    int x;          /*   bereich der Funktion   */
    int y;
    int z;
  } d3_koordinate;

  /* ... */
}

/* struct point {double x; double y;};          */
     /* unzulässige Neudefinition des Typs     */

int main( void ) {
  struct point p = init();
     /* hier wird der auf top-level definierte */
     /*   Typ benutzt                          */

  /* ... */
}
```

Speicherabbild einer Struktur

Die Komponenten einer Struktur werden in der Reihenfolge ihrer Vereinbarung im Speicher abgelegt. Wegen der Ausrichtung der Komponenten auf Adressen, die die jeweilige Alignment-Anforderung der Komponente erfüllen, kann eine Struktur „unbenannte Lücken“ (*padding*) enthalten.

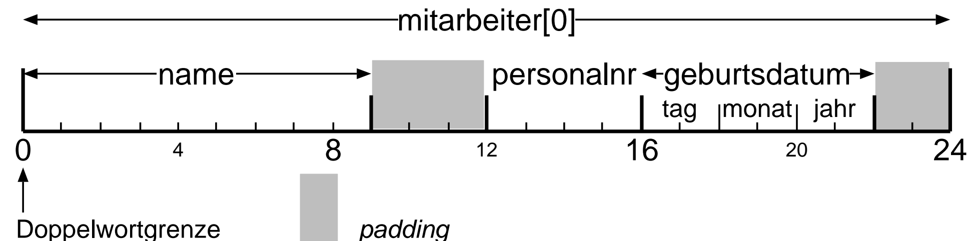
$$\Rightarrow \text{Größe einer Struktur} \neq \sum \text{Größe der Komponenten}$$

Eine „Lücke“ am Ende der Struktur stellt die Einhaltung der Alignment-Anforderungen für den Fall sicher, daß ein Vektor von Strukturen gebildet wird.

Beispiel:

```
#define MAX_NAME 8

struct person {
  char                 name[ MAX_NAME + 1 ];
  unsigned int         personalnr;
  struct {
    short int tag;
    short int monat;
    short int jahr;
  }                    geburtsdatum;
};

struct person mitarbeiter[10];
```

Speicherabbild (bei Verwendung des C for AIX Compilers):



offsetof-Makro:

```
#include <stddef.h>
size_t offsetof(structure_type, membername);
```

offsetof erlaubt es, die Lage der Komponente *membername* innerhalb einer Struktur des Typs *structure_type* zu bestimmen.

Resultatwert: Abstand (in Bytes) der Komponente *membername* vom Anfang der Struktur.

gebräuchliche Implementierung:

```
#define offsetof(struct_type, member) \
          (size_t) &(((struct_type *)0)->member)
```

Beispiel:

`offsetof(struct person, geburtsdatum)` → 16

Zeiger auf Strukturen

Beispiel:

```
struct {
  int  len;
  char *str;
} *p;  /* Zeiger auf die vereinbarte  */
       /*   Struktur                  */
```

Zugriff auf die Komponenten:

`(*p).len` ⇔ **`p->len`**

Beispiele für die Verwendung von `->` in Ausdrücken:

Ausdruck	wird interpretiert als	Wirkung
`++p->len`	`++(p->len)`	inkrementiert 'len'
`(++p)->len`		inkrementiert 'p'
`p++->len`	`(p++)->len`	inkrementiert 'p'
`*p->str`	`*(p->str)`	bezeichnet das Objekt, auf das 'str' zeigt (l-Wert)
`*p->str++`	`*((p->str)++)`	inkrementiert die Komponente 'str' nach dem Zugriff
`(*p->str)++`	`(*(p->str))++`	inkrementiert das Objekt, auf das 'str' zeigt
`*p++->str`	`*((p++)->str)`	inkrementiert 'p' nach dem Zugriff auf die Komponente 'str'

Vektoren von Strukturen

Beispiel:

```
struct point {    /* oder falls Typ vorher */
  int x;          /*   definiert wurde:    */
  int y;          /* struct point          */
} polyline[] = {  /*   polyline[] = { ...  */
  {  0, 21 },
  { 28,  1 },
  { 18,  3 },
};
```

polyline[1].x $\longrightarrow$ 28

Rekursive Strukturen

Beispiel: Element einer einfach verketteten Liste

```
struct list_elem {
  char *element;
  struct list_elem *next;
};
```

Standard-Formulierung zum Durchlaufen einer verketteten Liste:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
  ...
```


Falls zwei Strukturtypen derart zu vereinbaren sind, daß jede Struktur eine Bezugnahme auf den jeweils anderen Strukturtyp beinhaltet, dann kann durch eine Deklaration der Form

struct *type_tag***;**

sichergestellt werden, daß sich die eine Bezugnahme immer auf den später vereinbarten Strukturtyp bezieht, und nicht auf einen Strukturtyp, der in einem umschließenden Gültigkeitsbereich vereinbart wurde.

Beispiel:

```
struct s { /* ... */ };

void f( void )
{
  struct s;   /* macht die Typdefinition im   */
              /*   umschliessenden Gültig-    */
              /*   keitsbereich unwirksam     */
  struct t {
    /* ... */
    struct s *ptr_to_s;
  };

  struct s {
    /* ... */
    struct t *ptr_to_t;
  };

  /* ... */
```

typedef-Vereinbarung

Beispiele:

```
typedef int Counter;    /* #define Counter int */
Counter i, *vp[10];

typedef char *String;
String p;
...
p=(String) malloc(100);

typedef struct list_elem *listptr;
typedef struct list_elem {
  char    *element;
  listptr next;
} listelem;

listptr allocelem( void )
{
  return( (listptr) malloc(sizeof(listelem)) );
}
```

Der mit **typedef** vereinbarte Typname tritt in der Position eines Variablennamens auf und nicht am Ende der Vereinbarung:

```
typedef char String80[80+1];

String80 Seite[60];  ⇔  char (Seite[60])[80+1];
```


Vorteile:

- bessere Lesbarkeit des Programms
- Programm gegen Portabilitätsprobleme parametrisierbar (**size_t**, **ptrdiff_t**)

Aufgabe

Gegeben seien die folgenden Vereinbarungen:

```
typedef char alpha3_t[3];

static char v1[] = "a";

static struct {
  alpha3_t *str1;
  char     *cp;
  alpha3_t str2;
  char     ch;
} s1 = {&s1.str2, v1, "b", 'c'};
```

Zeichnen Sie eine Skizze, in der die Objekte, die durch diese Vereinbarungen definiert werden, mit ihrem Inhalt aufgeführt sind.

Union (Vereinigung)

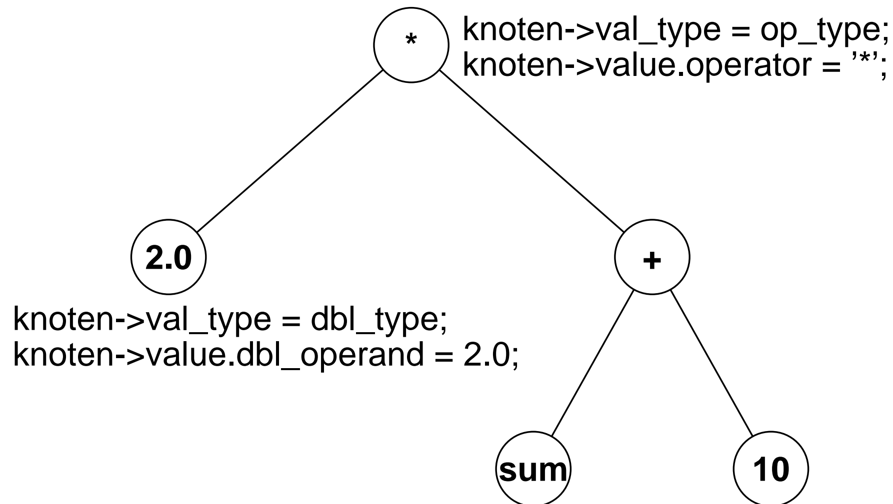
- die Komponenten einer Union (Alternativen) überlagern sich im Speicher (d.h. sie beginnen alle an der gleichen Adresse)
- Initialisierung mit Wert, der zum Typ der ersten Alternative passt
- gleiche Operationen wie bei Strukturen zulässig
- Unionen können innerhalb von Strukturen auftreten und umgekehrt

Beispiel:

Bäume zur Speicherung arithmetischer Ausdrücke

```
enum value_type { op_type, int_type,
                  dbl_type, var_type };

union node_value {
  char    operator;
  int     int_operand;
  double  dbl_operand;
  char    var_operand[31];
};

struct node {
  enum value_type   val_type;
  union node_value  value;
  struct node       *left;
  struct node       *right;
};

struct node *root, *knoten;
```

Für den Ausdruck `2.0*(sum+10)` ergibt sich folgender Baum:



Aufgabe

Schreiben Sie zunächst eine Funktion

```
void printbit(char ch);
```

die die Binärdarstellung (interne Darstellung) des Argumentes *ch* auf die Standardausgabe (**stdout**) schreibt. (Beispielsweise soll der Aufruf **printbit('\x32')** die Ausgabe **0011 0010** erzeugen.)

Verwenden Sie diese Funktion, um eine weitere Funktion bereitzustellen. Diese soll die Binärdarstellung eines (als Argument übergebenen) **double**-Wertes ausgeben.

Bitfelder

Ein Bitfeld ist eine Struktur- oder Unionkomponente, die aus einer (bei der Vereinbarung) explizit festgelegten Anzahl von Bits besteht.

Bitfelder erlauben die Definition „kleiner ganzer Zahlen“ innerhalb einer Speichereinheit (Wort). (Damit ermöglichen sie alternativ zu den Operatoren für Bitmanipulationen den Zugriff auf Teile eines Wortes.)

Syntax:

integer_type $membername_{opt}$ **:** *const_expr*

integer_type
: legt den Typ des Bitfeldes fest. Zulässige Typen sind:

int (entspricht **signed** oder **unsigned**)
signed [int]
unsigned [int]

membername
: Name des Bitfeldes (gleichzeitig Name einer Struktur- bzw. Unionkomponente)
Wird der Name weggelassen, dann kann das Bitfeld nicht angesprochen werden. Es wird auch bei der Initialisierung nicht berücksichtigt. Die einzige Funktion eines solchen Bitfeldes ist es, eine Lücke zu füllen (*padding*).

const_expr
: ganzzahliger konstanter Ausdruck, der die Länge (= Anzahl der Bits) des Bitfeldes festlegt. Die Länge eines Wortes darf nicht überschritten werden:

$0 < const_expr \leq \mathbf{sizeof}(\mathbf{int})*\mathbf{CHAR_BIT}$

Bitfelder

Ein unbenanntes Bitfeld darf außerdem mit der Länge 0 vereinbart werden, um dem Compiler anzuzeigen, daß ein unmittelbar folgendes Bitfeld auf Wortgrenze ausgerichtet werden soll. (Der Rest eines bereits teilweise belegten Wortes bleibt dann frei.)

Beispiele:

```
union {
  unsigned int i1 : 3;  /* Wertebereich:      */
                        /*    0 ... 7         */
  signed int   i2 : 3;  /* Wertebereich:      */
                        /*   -4 ... 3         */
} u = {7};

struct bit_code {
  unsigned left  : 1, right : 1,
           above : 1, below : 1;
} flags;      /* Struktur mit 4 Bitfeldern    */
```

Bitfelder unterliegen der Integer-Erweiterung. Daher können sie wie andere ganzzahlige Werte in arithmetischen Ausdrücken verwendet werden:

```
flags.left = (x < xmin);

printf("u.i2 = %d\n" , u.i2);     /* -> -1    */
```

Die meisten Eigenschaften von Bitfeldern sind implementierungsabhängig:

- **int** wird als **signed** oder **unsigned** interpretiert.
- mehrere Bitfelder in einem Wort können von links nach rechts oder umgekehrt angeordnet sein.
- maximale Länge hängt von der Wortlänge des Rechners ab.
- Paßt ein Bitfeld nicht mehr in ein Wort, das bereits durch vorausgehende Bitfelder teilweise gefüllt ist, kann es auf dieses und das nachfolgende Wort verteilt werden. Ebensogut kann es vollständig im nächsten Wort abgelegt werden. (Der Rest des angefangenen Wortes bleibt dann frei (*padding*).)

Anwendungen für Bitfelder:

- Verringerung des Speicherbedarfs für eine Struktur
  (Nachteil: Vergrößerung des ausführbaren Programmes und der Ausführungszeit)
- Abbildung extern definierter Objekte (wie z.B. Schnittstellen von Peripheriegeräten)
  (Nachteil: keine Portabilität)

Einschränkungen:

- Adreßoperator darf nicht auf Bitfelder angewendet werden.
- Vektoren von Bitfeldern können nicht vereinbart werden. (Vektoren von Strukturen, die Bitfelder enthalten, sind jedoch zulässig.)

Variable Argumentlisten

Beispiel für eine Funktion mit variabler Argumentliste:

```
int printf(const char *, ...);
```

Deklaration „ . . . “ bedeutet, daß Anzahl und Typen der entsprechenden Argumente variieren können. („ . . . “ ist nur am Ende einer Argumentliste zulässig).

Auf die Argumente, die „ . . . “ entsprechen, wird Argument-Erweiterung (Integer-Erweiterung, **float**→**double**) angewendet.

Makros in <stdarg.h>

Zur Bearbeitung variabler Argumentlisten benötigt man eine Variable vom Typ **va_list** (Zeiger auf die einzelnen Argumente, *handle*):

```
va_list ap;    /* argument pointer */
```

ap muß mit dem Makro **va_start** initialisiert werden:

```
void va_start(va_list ap, lastarg);
```

lastarg ist der letzte benannte Parameter vor „ . . . “.

Anschließend liefert jeder Aufruf des Makros **va_arg** ein Argument:

> *type* **va_arg(va_list** *ap*, *type*);

Vor Verlassen der Funktion muß die Bearbeitung abgeschlossen werden:

> **void va_end(va_list** *ap*);

Beispiel: Verketten mehrerer Strings

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *
multcat(int numargs, ... )
{
     va_list argptr;
     char *result;
     int i, siz;

     /* get size required */
     va_start(argptr, numargs);
     for(siz = i = 0; i < numargs; i++)
          siz += strlen(va_arg(argptr, char *));

     if ((result = calloc(siz + 1, 1)) == NULL) {
          fprintf(stderr, "Out of space\n");
          exit(EXIT_FAILURE);
     }
     va_end(argptr);

     va_start(argptr, numargs);
     for(i = 0; i < numargs; i++)
          strcat(result, va_arg(argptr, char *));
     va_end(argptr);
     return(result);
}
```

```
int
main(void)
{    printf(multcat(5, "One ", "two ", "three ",
          "testing", ".\n"));
     exit(EXIT_SUCCESS);
}
```


Spezielle Ausgabefunktionen: vprintf, vfprintf und vsprintf

formatierte Ausgabefunktionen, an die variable Argumentlisten weitergegeben werden können:

```
#include <stdio.h>

int vprintf(const char *format, va_list arg);

int vfprintf(FILE *stream, const char *format,
             va_list arg);

int vsprintf(char *s, const char *format,
             va_list arg);
```

Diese Funktionen setzen wie **va_arg** die Initialisierung von *arg* mittels **va_start** voraus. Ebenso muß vor Verlassen der Funktion **va_end** mit dem Argument *arg* aufgerufen werden.

Beispiel: error-Funktion

```
  /* ... */
#include "error.h"

  /* ... */

  FILE *outfp;
  char outfid[256];

  /* ... */

  if ( (outfp = fopen(outfid, "w")) == NULL )
    error(err, "Unable to open output file %s",
      outfid);

  /* ... */
```

Ausgabe:

```
Permission denied
Error: Unable to open output file ./output

No such file or directory
Error: Unable to open output file test/output
```



```
#ifndef ERROR_H_INCLUDED
#define ERROR_H_INCLUDED

/* Deklarationen und Typdefinitionen */

typedef enum {      /* Fehlerklassen:           */
  warn,             /* Warnung                  */
  nonterm_err,      /* Fehler, der nicht zum    */
                    /*   sofortigen Programm-   */
                    /*   abbruch fuehrt         */
  err,              /* Fehler                   */
  intern_err        /* interner Fehler          */
} errclass_type;

/* Schnittstellenfunktionen: */

extern void
error   /* Fehlerbehandlungsroutine (gibt Feh- */
        /*   lermeldung aus und beendet ggf.   */
        /*   die Ausfuehrung des Programms)    */
  (
    errclass_type errclass,  /* Fehlerklasse   */
    const char    *format,   /* Fehlermeldung  */
        /*   (Diese darf dieselben Formatan-   */
        /*   gaben enthalten wie der Kontroll- */
        /*   String der Funktion 'printf'.)    */
    ... /* Argumente, die in die Fehlermeldung */
        /*   einzufuegen sind (Die Anzahl ist  */
        /*   durch die Zahl der Formatangaben  */
        /*   in der Fehlermeldung festgelegt.) */
  );

#endif /* ERROR_H_INCLUDED */
```

```
#include <stdarg.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "error.h"

void
error   /* Fehlerbehandlungsroutine (gibt Feh- */
        /*   lermeldung aus und beendet ggf.   */
        /*   die Ausfuehrung des Programms)    */
  (
    errclass_type errclass, /* Fehlerklasse    */
    const char    *format,  /* Fehlermeldung   */
    ...                     /* Argumente, die  */
                            /*   in die Feh-   */
                            /*   lermeldung    */
                            /*   einzufuegen   */
                            /*   sind          */
  )
{
  va_list argptr;


  if ( errno != 0 ) {
    fprintf(stderr, "%s\n", strerror(errno));
    errno = 0;
  }
```



```
  if ( format != NULL ) {
    switch ( errclass ) {
      case warn:
             fprintf(stderr, "Warning: ");
             break;
      case nonterm_err:
      case err:
             fprintf(stderr, "Error: ");
             break;
      case intern_err:
             fprintf(stderr, "Internal Error: ");
             break;
      default:
             fprintf(stderr,
               "Internal Error: "
                 "Undefined error class %d\n",
               (int) errclass);
             fprintf(stderr, "???: ");
             errclass = err;
             break;
    }
    va_start(argptr, format);
    vfprintf(stderr, format, argptr);
    va_end(argptr);
    fprintf(stderr, "\n");
  }
  if ( errclass > nonterm_err ) {
                        /* Programm abbrechen     */
    fclose(stderr);
    exit(EXIT_FAILURE);
  }
  else
    return;
}
```

Funktionen zur Erzeugung von Pseudo-Zufallszahlen

```
#include <stdlib.h>
int rand (void);
void srand(unsigned int seed);
```

rand liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis **RAND_MAX**; dieser Wert ist mindestens 32767.

srand dient dazu, vor dem ersten Aufruf von **rand**, einen Startwert (*seed*) für die Folge von Pseudo-Zufallszahlen zu setzen. **rand** liefert für einen bestimmten Ausgangswert *seed* immer die gleiche Folge von Zufallszahlen (Deshalb werden diese als **Pseudo**-Zufallszahlen bezeichnet.). Wird **srand** nicht aufgerufen, dann ist der Ausgangswert 1.

Sollen nicht jedesmal die gleichen Zufallszahlen erzeugt werden, empfiehlt es sich, den Startwert z.B. in Abhängigkeit von der Uhrzeit zu wählen. Beispielsweise könnte der Resultatwert der Funktion **time** als Startwert benutzt werden.

Funktionen zum Sortieren und Suchen

```
#include <stdlib.h>
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

qsort sortiert einen Vektor *base*[0] . . . *base*[*n*-1] von Objekten der Größe *size* in aufsteigender Reihenfolge. *cmp* ist ein Zeiger auf eine Funktion, mit der zwei Elemente des Vektors verglichen werden können. Diese Vergleichsfunktion muß einen negativen Wert liefern, wenn ihr erstes Argument kleiner als ihr zweites Argument ist, Null, wenn sie mit gleichen Argumenten aufgerufen wird, und in jedem anderen Fall einen positiven Wert.

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *, const void *));
```

bsearch sucht im Vektor *base*[0] . . . *base*[*n*-1] von Objekten der Größe *size* ein Element, das mit ***key** übereinstimmt. Der übergebene Vektor muß aufsteigend sortiert sein. Das Sortierkriterium ist durch die Funktion *cmp* festgelegt. (Für die Funktion *cmp* gilt das gleiche wie bei **qsort**.)

Resultatwert: Zeiger auf gefundenes Element oder **NULL**, falls kein Element mit dem Wert ***key** existiert. (Falls ein Element mehrfach im Vektor vorkommt, ist undefiniert, welches zurückgegeben wird.)

Fehlersuche

```
#include <assert.h>
void assert(int expression); /* Makro */
```

Falls *expression* bei der Ausführung den Wert 0 (FALSE) ergibt, wird eine Meldung der Art

Assertion failed: expression, file **__FILE__**, line **__LINE__**

auf der Standardfehlerausgabe (**stderr**) ausgegeben. Die Programmausführung wird durch Aufruf von **abort** abgebrochen. Wenn beim Einfügen von **<assert.h>** ein Makro **NDEBUG** definiert ist, wird **assert** ignoriert.

Funktionen zur Beendigung eines Programmes

```
#include <stdlib.h>
void abort( void );
```

anormale Beendigung des Programms (kann durch Signalbehandlungsfunktion für das Signal **SIGABRT** abgefangen werden)

```
#include <stdlib.h>

void exit( int status );
```

für normale Beendigung des Programms. Aktionen:

- Ausführen der mit **atexit** festgelegten Funktionen (in umgekehrter Reihenfolge ihrer Hinterlegung)
- Ausgabe von Dateipuffern
- Abschließen noch offener Dateien
- Kontrolle wird an Betriebssystem zurückgegeben

status 0 gilt als erfolgreiches Ende (ebenso **EXIT_SUCCESS**)


```
#include <stdlib.h>

int atexit( void (*fcn)(void));
```

hinterlegt die Funktion *fcn*, damit sie bei normaler Beendigung des Programms aufgerufen wird.

Resultatwert $\neq 0$: *fcn* konnte nicht hinterlegt werden.

Schnittstelle zur Betriebssystemumgebung

```
#include <stdlib.h>
char *getenv(const char *name);
```

Viele Betriebssysteme erlauben die Definition von Variablen, die dazu dienen, Programmen Informationen aus der Betriebssystemumgebung zugänglich zu machen (*environment variables*).

Beispiel:

UNIX (Shell-Variablen), Korn-Shell:

export *var_name*=*wert*

getenv liefert den (Zeichenketten-)Wert der Variablen *name* aus der Betriebssystemumgebung. **getenv** kann einen statischen Speicherbereich benutzen, um den Wert der Variablen abzuspeichern, so daß dieser in einem folgenden Aufruf von **getenv** überschrieben wird.

Wenn keine Variable mit dem angegebenen Namen gefunden wird, ist der Resultatwert **NULL**.

Beispiel:

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

int main( void )
{
  char buf[81], *is;

  while ( !feof(stdin) )
  {
    printf("Enter an environmental variable: ");
    fflush(stdout);

    if ( gets(buf) != NULL )
    {
      if ( (is = getenv(buf)) != NULL )
        printf("%s = %s\n", buf, is);
      else
        printf("Can't find %s\n", buf);
    }
  }
  exit(EXIT_SUCCESS);
}
```

Beispiel für Ausgabe unter UNIX:

```
Enter an environmental variable: HOME
HOME = /u/zdv/zdv064
```

```
#include <stdlib.h>
int system(const char *program);
```

übergibt den String *program* an den Kommandoprozessor des Systems zur Ausführung.

Resultatwert: implementierungsabhängig

NULL kann als Argument benutzt werden, um zu testen, ob es einen Kommandoprozessor gibt (Resultatwert≠0) oder nicht (Resultatwert=0). Durch den C-Standard ist jedoch nicht sichergestellt, daß **system** auch bei Schachtelung korrekt arbeitet (Läßt sich durch **system(NULL)** ermitteln.).

Kommunikationsmöglichkeiten zwischen rufenden und gerufenem Programm:

1. durch Übergabe von Argumenten in der Kommandozeile
2. durch Definition von Umgebungsvariablen
3. über Dateien (größte Portabilität)
   (Die Dateien sollten vor dem Aufruf von **system** mittels **fclose** abgeschlossen werden.)

Rückgabe von Informationen:

1. durch Resultatwert (implementierungsabhängig; bei vielen Systemen wird der **exit**-Code als Resultatwert zurückgegeben (z.B. C for AIX))
2. durch Dateien (größte Portabilität)

Ein-/Ausgabefunktionen in <stdio.h>

Eröffnen einer Datei

```
FILE *fopen(const char *filename,
            const char *mode);
```

fopen eröffnet die Datei mit dem Namen *filename* gemäß des spezifizierten Modus *mode*.

Resultatwert: Datei-Zeiger (vom Typ **FILE ***), falls die Datei eröffnet werden konnte, sonst **NULL**.

Zulässige Werte für *mode* sind:

- **"r"** Textdatei zum Lesen eröffnen
- **"rb"** Binärdatei zum Lesen eröffnen
- **"r+"** Textdatei sowohl für Lese- als auch für Schreibzugriff eröffnen (*update mode*).
- **"rb+"** oder **"r+b"** Binärdatei, *update mode*

Das Eröffnen in einen *read*-Modus setzt voraus, daß die Datei existiert.

Nach Eröffnen in einen *update*-Modus kann an jede Position der Datei geschrieben werden. (Lese- und Schreiboperatioen dürfen jedoch nicht aufeinanderfolgen. Dazwischen muß die Dateiposition durch eine der Funktionen **fseek**, **fsetpos** oder **rewind** neu gesetzt oder der Ausgabepuffer einer Datei muß mit **fflush** geschrieben werden.)

"w" Textdatei zum Schreiben eröffnen

"wb" Binärdatei zum Schreiben eröffnen

"w+" Textdatei, *update mode*

"wb+" oder **"w+b"**
Binärdatei, *update mode*

Beim Eröffnen in einen *write*-Modus wird die Datei angelegt, wenn sie noch nicht existiert.

Vorsicht: Der Inhalt einer bereits existierenden Datei ist nach dem Eröffnen gelöscht.

Für die *update*-Modi gilt das gleiche wie bei den *read*-Modi.

"a" Textdatei zum Erweitern eröffnen (Datei kann am Dateiende fortgeschrieben werden.)

"ab" Binärdatei zum Erweitern eröffnen

"a+" Textdatei, *update mode*

"ab+" oder "a+b"
Binärdatei, *update mode*

Nach dem Eröffnen in einen *append*-Modus ist die aktuelle Dateiposition das Ende der Datei. (Bei den anderen Modi ist es der Dateianfang.) Eine Datei wird angelegt, falls sie noch nicht existiert. Auch beim Eröffnen mit einem der *update*-Modi (**"a+"**, **"ab+"** oder **"a+b"**) ist Schreiben nur am Ende der Datei möglich.

Dateinamen sind auf **FILENAME_MAX** Zeichen begrenzt. Maximal **FOPEN_MAX** Dateien können gleichzeitig offen sein (**FOPEN_MAX** $\geq$ 8, einschließlich **stdin**, **stdout** und **stderr**).

Ein-/Ausgabefunktionen in <stdio.h>
Eröffnen einer Datei

FILE *freopen(const char ****filename*,**
const char ****mode*, FILE ****stream***);**

- analog zu **fopen**, jedoch wird der Datei-Zeiger mit dem die Datei verknüpft werden soll als Argument übergeben. Falls *stream* noch mit einer offenen Datei verknüpft ist, wird diese zuvor geschlossen (dabei evtl. auftretende Fehler werden ignoriert).
- ermöglicht das Umdefinieren der mit **stdin**, **stdout** oder **stderr** verknüpften Dateien (Diese Datei-Zeiger brauchen keine l-Werte zu sein, so daß der Resultatwert von **fopen** nicht zugewiesen werden kann.)
- eine weitere Anwendung ist das Ändern des Modus einer offenen Datei

Abschließen einer Datei

int fclose(FILE ****fp*);**

- schreibt Ausgabepuffer der mit dem Datei-Zeiger *fp* verknüpften Datei, bzw. löscht noch nicht gelesene, gepufferte Eingaben
- gibt Puffer frei
- schließt Datei
- Resultatwert: 0 oder (bei Fehlern) **EOF**
- *fp* kann anschließend mit einer (anderen) Datei verknüpft werden.

Bei normaler Beendigung eines Programmes wird **fclose** automatisch für jede offene Datei aufgerufen.

Abfragen und Setzen von Fehlerbedingungen

int feof(FILE ****stream*);**

feof liefert einen Wert ungleich 0, wenn die *end-of-file*-Bedingung für die mit *stream* verknüpfte Datei gesetzt ist.

Vorsicht: Die *end-of-file*-Bedingung ist erst dann gesetzt, **nachdem** eine Eingabefunktion versucht hat, über das Dateiende hinaus zu lesen.
⇒ Es ist keine Aussage möglich, ob das Dateiende bereits erreicht ist oder nicht, wenn **feof** den Wert 0 als Resultat liefert. (**feof** kann also nicht in der gleichen Weise verwendet werden, wie die entsprechende Funktion in Pascal.)

int ferror(FILE ****stream*);**

ferror liefert einen Wert ungleich 0, wenn die *error*-Bedingung für die mit *stream* verknüpfte Datei gesetzt ist.
(C for AIX setzt die *error*-Bedingung z.B. dann, wenn eine Schreiboperation nicht ausgeführt werden kann, weil kein Plattenplatz mehr zur Verfügung steht.)

void clearerr(FILE ****stream*);**

clearerr setzt die *end-of-file*- und *error*-Bedingung für *stream* zurück.

Abfragen und Setzen der Dateiposition

long int ftell(FILE ***stream*);**

ftell liefert die aktuelle Dateiposition der mit *stream* verknüpften Datei oder **-1L**, falls ein Fehler auftritt.

Dateiposition:

Binärdatei: Anzahl der Zeichen vom Anfang der Datei bis zur aktuellen Position

Textdatei: implementierungsdefinierter Wert (wegen implementierungsabhängiger Darstellung des Zeilenende-Zeichens)

int fseek(FILE ***stream*, long** *offset***, int** *origin***);**

fseek setzt die Dateiposition für *stream*.

Resultatwert: ungleich 0 bei Fehler

offset

Binärdatei: Wert (mit Vorzeichen), der die neue Dateiposition durch ihren Abstand (in Bytes) bezogen auf *origin* angibt.

Textdatei: entweder 0 oder ein Wert, den ein vorheriger Aufruf von **ftell** geliefert hat (*origin* muß im letzteren Fall den Wert **SEEK_SET** haben).

origin bestimmt, wie die neue Dateiposition angegeben wird:

SEEK_CUR relativ zur aktuellen Position

SEEK_END relativ zum Dateiende
(u.U. bei Binärdateien nicht unterstützt)

SEEK_SET relativ zum Anfang der Datei

Beispiel:

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

int main(int argc, char *argv[])
{
  char   *filename;
  FILE   *ifp;
  double randomAdj;
  int    c;

  switch ( argc )
  {
    case 1:
            filename = "fortunes.dat";
            break;
    case 2:
            filename = argv[1];
            break;
    default:
            printf("usage: fortune inputfile\n");
            exit(EXIT_FAILURE);
  }

  if ( (ifp = fopen(filename, "r")) == NULL )
  {
    printf("Cannot open %s\n", filename);
    exit(EXIT_FAILURE);
  }
```



```
  fseek(ifp, 0L, SEEK_END);
  randomAdj =
    ((double) ftell(ifp)) / ((double) RAND_MAX);
  srand( (unsigned int) time(NULL) );
  fseek(ifp, (long)(randomAdj * (double)rand()),
    SEEK_SET);

  while ( ((c = getc(ifp)) != '\n')
          && (c != EOF) )
    ;
  if ( (c == EOF) || ((c = getc(ifp)) == EOF) )
    fseek(ifp, 0L, SEEK_SET);
  else
    ungetc(c, ifp);

  while ( ((c = getc(ifp)) != '\n')
          && (c != EOF) )
  {
   /* display multi-line fortunes */
    putchar((c == '@') ? '\n' : c);
  }
  putchar('\n');
  exit(EXIT_SUCCESS);
}
```

Beispiel für Eingabedatei:

```
 . . .
 Every solution breeds new problems.
 . . .
 Living on Earth may be expensive, but it include
s an annual free trip@around the Sun.
 . . .
 Pascal Users:@  To show respect for the 313th an
niversary (tomorrow) of the@  death of Blaise Pas
cal, your programs will be run at half@  speed.
 . . .
```


Ein-/Ausgabefunktionen in <stdio.h>
Abfragen und Setzen der Dateiposition

void rewind(FILE ****stream*);**

rewind ist äquivalent zu

```
fseek(stream, 0L, SEEK_SET);
clearerr(stream);
```

int fgetpos(FILE ****stream*, fpos_t ****position***);**

fgetpos weist **position* die aktuelle Dateiposition zu.

Resultatwert: ungleich 0 bei Fehler

Diese Funktion kann im Gegensatz zu **ftell** auch bei extrem großen Dateien benutzt werden, deren Positionen nicht alle als **long**-Wert darstellbar sind.

Für kleinere Dateien sollte **ftell** benutzt werden.

int fsetpos(FILE ****stream*, const fpos_t ****position***);**

fsetpos setzt die Dateiposition auf einen Wert, den ein vorheriger Aufruf von **fgetpos** geliefert hat.

Resultatwert: ungleich 0 bei Fehler

Diese Funktion kann im Gegensatz zu **fseek** auch bei extrem großen Dateien benutzt werden, deren Positionen nicht alle als **long**-Wert darstellbar sind.

Für kleinere Dateien sollte **fseek** benutzt werden.

unformatierte Ein-/Ausgabe

size_t fwrite(const void ****buffer*, **size_t *size*, **size_t** *n*,
FILE ****stream* **);

fwrite gibt den Vektor *buffer*[0] . . . *buffer*[*n*-1] von Objekten der Größe *size* auf die mit *stream* verknüpfte Datei aus.

Resultatwert: Anzahl der ausgegebenen Elemente
(Ein Wert kleiner *n* weist auf einen Fehler hin.)

size_t fread(void ****buffer*, **size_t *size*, **size_t** *n*,
FILE ****stream*);**

fread liest einen Vektor *buffer*[0] . . . *buffer*[*n*-1] von Objekten der Größe *size* von der mit *stream* verknüpften Datei ein.

Resultatwert: Anzahl der gelesenen Elemente
(Ein Wert kleiner *n* weist auf einen Fehler hin. Hinweise auf die Fehlerursache erhält man mit **feof** und **ferror**.)

Beispielprogramm:
Erzeugen eines Telefonverzeichnisses mit Index

Die Daten für das Verzeichnis werden von einer Textdatei eingelesen und auf eine Binärdatei geschrieben. Die Eingabedatei muß bereits nach Namen sortiert sein.

Beispiel für Eingabedatei:

```
Action, G.                 DISTR  5162
Baker, R.                  DEVL   1152
Bramley, O. H.             ADMIN  6248
Cheeseman, D.              SUPP   8141
Cory, G.                   DEVL   8336
Elliott, D.                DEVL   9875
. . .
```

Auf der Ausgabedatei wird vor den eigentlichen Daten ein Index abgespeichert. Dieser enthält für jeden Buchstaben des Alphabets die Position des ersten zugehörigen Elementes in der Datei.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "error.h"

#define MAX_NAME 24
#define MAX_DST   6
#define OUTFIL   "telvz.dat"

typedef struct {
  char      name[ MAX_NAME + 1 ];
  char      dienststelle[ MAX_DST + 1 ];
  short int rufnummer;
} Mitarbeiter;
```

```
char *trim(char *str)
{
  register char *endp;

  if ( str == NULL )
    return ( str );

 /* start at end of string
                    while in string and spaces */
  for ( endp = strchr(str, '\0') ;
        (endp != str) && (*--endp == ' ') ;
      )
    *endp = '\0';   /* replace spaces with     */
                    /*   null characters       */
  return ( str );
}


int main( void )
{
  const char  *abc="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  FILE        *fp;
  long int    index[26];
  Mitarbeiter mitarb;
  char        *bi;          /* Buchstabenindex */
  int         c;

  if ( (fp = fopen(OUTFIL, "wb")) == NULL )
    error(err,
      "Unable to open output file \"%s\"",
      OUTFIL);
```


Ein-/Ausgabefunktionen in <stdio.h>
unformatierte Ein-/Ausgabe

```
  for ( c = 0 ; c < 26 ; c++ )
    index[c] = -1L;
  fwrite(index, sizeof(long), 26, fp);  c = '\0';

  while ( fgets(mitarb.name, MAX_NAME+1, stdin)
          != NULL
        )
  {
    trim(mitarb.name);
    getc(stdin);    /* Trennzeichen ueberlesen */
    fgets(mitarb.dienststelle, MAX_DST+1, stdin);
    fscanf(stdin, "%hd", &mitarb.rufnummer);
    while ( getc(stdin) != '\n' )
      ;
    if ( *mitarb.name != c )
    {
      c = *mitarb.name;
      if ( (bi = strchr(abc, c)) != NULL )
        index[ bi-abc ] = ftell(fp);
      else
        error(err, "Invalid name \"%s\"",
          mitarb.name);
    }
    fwrite(&mitarb, sizeof(mitarb), 1, fp);
  }

  fseek(fp, 0L, SEEK_SET);
  fwrite(index, sizeof(long), 26, fp);
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

Beispielprogramm: Suchen im Telefonverzeichnis

Das Programm liest Namen ein und gibt diese mit der zugehörigen Telefonnummer wieder aus.

Um die Suchzeit abzukürzen, wird der Index benutzt. Anstelle der gesamten Datei werden nur die Elemente durchsucht, die einen Namen mit gleichem Anfangsbuchstaben wie der gesuchte Name enthalten.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "error.h"

#define MAX_NAME  24
#define MAX_DST    6
#define LEN_TELNR  4
#define INFIL     "telvz.dat"

typedef struct {
  char      name[ MAX_NAME + 1 ];
  char      dienststelle[ MAX_DST + 1 ];
  short int rufnummer;
} Mitarbeiter;


int main( void )
{
  const char  *abc="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  FILE        *fp;
  long int    index[26];
  Mitarbeiter mitarb;
  char        name[ MAX_NAME + 1 ];
  char        *bi;          /* Buchstabenindex */
  int         cmp, rdcnt=1;
```



```
  if ( (fp = fopen(INFIL, "rb")) == NULL )
    error(err,
      "Unable to open input file \"%s\"", INFIL);

  fread(index, sizeof(long), 26, fp);

  while ( gets(name) != NULL )
  {
    if ( (bi = strchr(abc, *name)) != NULL )
    {
      if ( index[ bi-abc ] != -1L )
      {
        fseek(fp, index[ bi-abc ], SEEK_SET);
        while (
          (rdcnt++, fread(&mitarb,sizeof(mitarb),
                      1,fp)) &&
          ((cmp=strcmp(mitarb.name, name)) < 0) )
          ;
        if ( cmp == 0 )
          printf("%-24s %s %.*hd\n",
            mitarb.name, mitarb.dienststelle,
            LEN_TELNR, mitarb.rufnummer);
        else
          error(nonterm_err,
            "Name \"%s\" not found", name);
      }
      else
        error(nonterm_err,
          "Name \"%s\" not found", name);
    }
    else
      error(nonterm_err, "Invalid name \"%s\"",
        name);
  }
```

```
printf("%d \"fread\"-Aufruf%s\n", rdcnt,
    (rdcnt == 1) ? "" : "e");
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```


sonstige Dateioperationen

int fflush(FILE **stream*);

fflush gibt den Inhalt des Ausgabepuffers der mit *stream* verknüpften Datei aus. Wird **NULL** als Argument übergeben, wirkt **fflush** auf alle offenen Ausgabedateien. (Die Wirkung von **fflush** ist undefiniert, wenn *stream* zum Lesen eröffnet ist.)

Resultatwert: **EOF** bei Fehler, sonst 0

int setvbuf(FILE *_stream_, char *_buffer_, int _mode_, size_t _size_);

fopen (bzw. **freopen**) legt beim Eröffnen standardmäßig einen Puffer der Länge **BUFSIZ** (≥ 256) Bytes an.

setvbuf erlaubt es, die Größe eines Puffers und die Art der Pufferung zu verändern. Dazu muß **setvbuf** nach dem Eröffnen, jedoch vor allen anderen Dateioperationen aufgerufen werden.

buffer Adresse des neuen Puffers
oder **NULL** (dann wird ein Puffer angelegt)

size Puffergröße in Bytes

mode bestimmt die Art der Pufferung:

- **_IOFBF** vollständige Pufferung
- **_IONBF** keine Pufferung
- **_IOLBF** zeilenweise Pufferung (bei Textdateien; Ausgabe von ’\n’ bewirkt die Übertragung des Puffers.)

Resultatwert: ungleich 0 bei Fehler

void setbuf(FILE *_stream_, char *_buffer_);

Für *buffer* ungleich **NULL** ist **setbuf** äquivalent zu

```
(void) setvbuf(stream, buffer, _IOFBF, BUFSIZ);
```

Andernfalls erfolgt keine Pufferung:

```
(void) setbuf(stream, NULL);
          ⇔
(void) setvbuf(stream, NULL, _IONBF, 0);
```

FILE *tmpfile(void);

tmpfile erzeugt eine temporäre Datei und eröffnet sie in den Modus **wb+**. Beim Abschließen (z.B. durch **fclose**) bzw. bei Beendigung des Programms wird die Datei automatisch gelöscht.

Resultatwert: Datei-Zeiger, falls eine Datei erzeugt werden konnte, sonst **NULL**.

char *tmpnam(char *_name_);

tmpnam liefert einen noch nicht verwendeten Dateinamen (d.h. es existiert keine Datei mit diesem Namen). Jeder Aufruf liefert einen anderen Namen.

Die von **tmpnam** erzeugten Namen sind i. allg. mechanisch erzeugte Zeichenkombinationen, und werden deshalb hauptsächlich für temporäre Dateien benutzt, die dem Benutzer verborgen bleiben sollen.

Falls für *name* **NULL** übergeben wird, wird der Dateiname in einen statischen Speicherbereich geschrieben. Ein weiterer Aufruf mit **NULL** als Argument überschreibt den Speicherbereich mit einem anderen Namen.

Soll der erzeugte Name in einen bestimmten Speicherbereich geschrieben werden, kann die Adresse dieses Bereichs übergeben werden. (Der Bereich muß mindestens **L_tmpnam** Bytes groß sein.)

Maximal können **TMP_MAX** ($\geq$ 25) verschiedene Dateinamen generiert werden.

Resultatwert: Zeiger auf den erzeugten Namen

Beispiel:

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

void fatal(const char *string)
{
  fprintf(stderr, "%s\n", string);
  exit(EXIT_FAILURE);
}

int main( void )
{
  int i, files;
  FILE *fp;
  char buffer[L_tmpnam];

  if ( (fp = fopen(tmpnam(buffer),"w")) == NULL )
    fatal("Cannot open temporary file");
  printf("Temporary file name is %s\n", buffer);

  /* put realistic limit on number of names */
  files = (TMP_MAX > 100) ? 100 : (TMP_MAX - 1);
  for ( i = 0 ; i < files ; i++ )
    fprintf(fp, "%s\n", tmpnam(NULL));
  fclose(fp);
  exit(EXIT_SUCCESS);
}
```

```
int remove(const char *filename);
```

remove löscht die Datei *filename*.

Resultatwert: ungleich 0 bei Fehler

```
int rename(const char *oldname,
           const char *newname);
```

rename ändert den Namen der Datei *oldname* in *newname*.

Resultatwert: ungleich 0 bei Fehler

Signalbehandlung

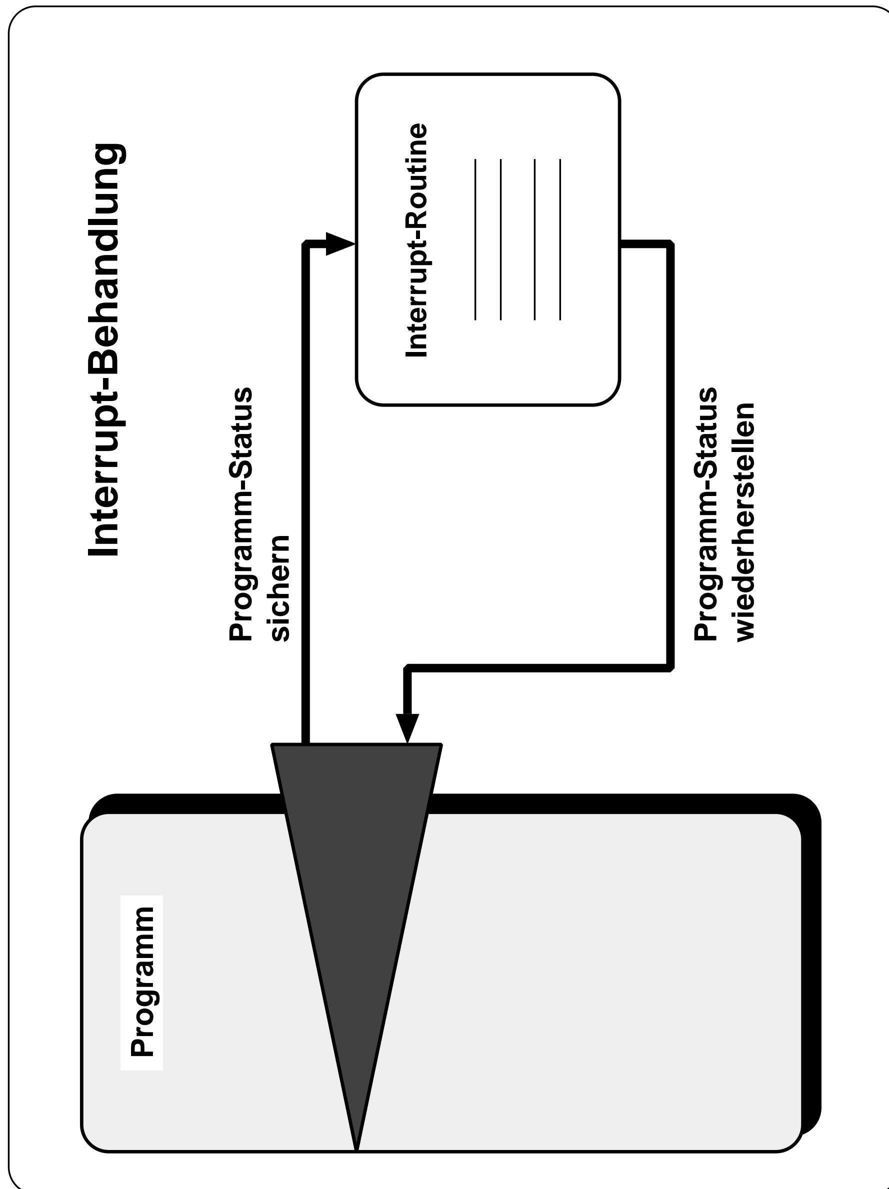
Signale

Ein Signal ist eine Ausnahmebedingung, die während der Programmausführung einen *Interrupt* (Programmunterbrechung) auslöst.

Man unterscheidet

- Hardware-Signale
  - vom Prozessor ausgelöst (z.B. *exponent overflow*)
  - von außen kommend (z.B. von Tastatur; asynchron)
- Software-Signale
  - vom Programm ausgelöst (häufig synchron)

Programm reagiert auf ein Signal durch Aufruf einer Interrupt-Routine (Signalbehandlungsroutine, *signal handler*). C erlaubt die Einrichtung eigener Routinen zur Behandlung bestimmter Signale.



Wenigstens die folgenden Signale werden unterstützt:

Signalname	***Bedeutung***
SIGABRT	anormale Beendigung des Programms (*abort*)
SIGFPE	Fehler bei arithmetischer Operation (z.B. Division durch 0.0, Overflow)
SIGILL	unzulässige Maschineninstruktion (*illegal operation*)
SIGINT	interaktiver Interrupt (*attention interrupt*), z.B. ausgelöst durch <Ctrl>-c (UNIX)
SIGSEGV	unzulässiger Speicherzugriff (*segment violation*)
SIGTERM	Signal zur Programmbeendigung (*termination*)

Die angegebenen Makronamen sind in **<signal.h>** definiert und expandieren zu konstanten positiven Integer-Ausdrücken.

Es bleibt der Implementierung überlassen, auf diese Signale nur zu reagieren, wenn sie durch einen expliziten Aufruf der Funktion **raise** erzeugt wurden.

Funktionen in <signal.h>

void (*signal(int *signame*, void (sig_handler*)(int)))**
(int);

signal installiert eine Signalbehandlungsroutine (*signal handler*) für das Signal *signame*.

Folgende Argumente für *sig_handler* sind möglich:

SIG_DFL	Standard-Signalbehandlungsroutine
SIG_IGN	um anzuzeigen, daß das Signal im folgenden ignoriert werden soll
sig_handler	benutzerdefinierte Funktion zur Signalbehandlung

Resultatwert: *sig_handler*, falls kein Fehler auftritt, sonst **SIG_ERR**

Nach dem Installieren einer benutzerdefinierten Funktion zur Signalbehandlung reagiert das Programm wie folgt, wenn es das entsprechende Signal empfängt:

- Default-Signalbehandlung (**SIG_DFL**) wird für das entsprechende Signal gesetzt (u.U. nicht für **SIGILL**)
- benutzerdefinierte Funktion wird aufgerufen (Die Funktion darf mit **return**, **abort**, **exit** oder **longjmp** enden.)

Anmerkung: Zu Programmbeginn wird für jedes Signal **SIG_DFL** oder **SIG_IGN** gesetzt.

int raise(int *signame*);

raise erzeugt das Signal *signame* und sendet es an das in der Ausführung befindliche Programm.

Resultatwert: ungleich 0 bei Fehler

Der Datentyp **sig_atomic_t**

- ist ein integraler Typ
- (Lesende und schreibende) Zugriffe auf Objekte dieses Datentyps sind *atomar*, d.h. sie können nicht unterbrochen werden.

Anmerkung: Objekte dieses Typs benötigen bei ihrer Vereinbarung häufig auch den Typzusatz **volatile**.

Beispiel:

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

static volatile sig_atomic_t gezogen =  0;
static volatile sig_atomic_t anzahl  = 49;
static short int kugeln[49];
```

```
void gewinnzahl(int sig_number)
{
  int i;

  if ( anzahl > 43 )
  {
    printf("%d. Gewinnzahl: %2hd\n",
      50 - ((int) anzahl), kugeln[gezogen]);

               /* gezogene Zahl rausnehmen:   */
    for ( i = gezogen+1 ;  i < anzahl ; i++ )
      kugeln[i-1] = kugeln[i];
    anzahl--;

    if ( signal(SIGINT, gewinnzahl) == SIG_ERR )
    {
      fprintf(stderr, "Could not set SIGINT\n");
      abort();
    }
  }
  else
  {
    printf("Zusatzzahl:    %2hd\n",
      kugeln[gezogen]);
    anzahl = -1;
  }
  return;
}
```



```
int main( void )
{
  int i;

  for ( i = 0 ; i < anzahl ; i++ )
    kugeln[i] = i+1;     /* Vektor mit Start-  */
                         /*   werten besetzen  */

  if ( signal(SIGINT, gewinnzahl) == SIG_ERR )
  {
    fprintf(stderr, "Could not set SIGINT\n");
    abort();
  }

  while ( gezogen < anzahl )
    ( gezogen+1 < anzahl ) ?
      (gezogen++) : (gezogen = 0);

  exit(EXIT_SUCCESS);
}
```

Funktionen in <setjmp.h>

Nicht-lokale Sprünge

int setjmp(jmp_buf *env***);**

- speichert Aufrufumgebung im Objekt *env*.
  (Der Typ von *env*, **jmp_buf**, ist ein Array-Typ.)
- Aufrufumgebung kann durch einen nachfolgenden Aufruf von **longjmp** wiederhergestellt werden.

Resultatwert: 0, falls **setjmp** direkt aufgerufen wurde;
ungleich 0 bei der Rückkehr von einem **longjmp**-Aufruf

setjmp ist nur in folgenden Kontexten zulässig:

- als Kontrollausdruck einer Auswahl (**if**, **switch**) oder einer Schleife (**setjmp** darf dabei mit einem konstanten Integer-Ausdruck verglichen werden oder muß Operand des **!**-Operators sein.)
- als Ausdruck einer Ausdrucksanweisung

void longjmp(jmp_buf *env***, int** *val***);**

- stellt die (beim letzten Aufruf von **setjmp**) in *env* gespeicherte Aufrufumgebung wieder her.
- die Funktion, die **setjmp** aufgerufen hat, darf noch nicht beendet sein.
- kann nicht von einer „geschachtelten“ Signalbehandlungsroutine gerufen werden.
- Programmausführung wird fortgesetzt, als ob der entsprechende **setjmp**-Aufruf gerade ausgeführt wurde und den Resultatwert *val* (bzw. 1, falls *val* gleich 0 ist) zurückgegeben hat.

Alle erreichbaren Objekte haben die Werte, die sie beim Aufruf von **longjmp** hatten (ausgenommen sind **auto**-Objekte ohne den Typzusatz **volatile**, die zwischen dem **setjmp**- und dem **longjmp**-Aufruf verändert wurden).

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>

jmp_buf cmd_prompt;


void fpe_handler(int sig_number)
{
  fprintf(stderr,
    "floating-point exception (division by zero,"
      " overflow, ...)\n");
  longjmp(cmd_prompt, 1);
}


double f(double x)
{
  return ( 1 / (1 - x*x) );
}
```



```
int main( void )
{
  double x;

  printf("f(x) = 1 / (1 - x*x)\n");
  if ( setjmp(cmd_prompt) != 0 )
    fprintf(stderr, "Returned from SIGFPE: Try "
      "another value for \"x\"\n");

                 /* Signalbehandlungsroutine     */
                 /*   fuer "floating-point       */
                 /*   exceptions" installieren:  */

  if ( signal(SIGFPE, fpe_handler) == SIG_ERR ) {
    fprintf(stderr, "Could not set SIGFPE\n");
    abort();
  }

  while ( scanf("%lf", &x) == 1 )
    printf("f(%g) = %g\n", x, f(x));

  if ( signal(SIGFPE, SIG_DFL) == SIG_ERR ) {
    fprintf(stderr, "Could not lower SIGFPE\n");
    abort();
  }

  exit(EXIT_SUCCESS);
}
```

Version für C for AIX Compiler:

```
/* Compile with:                            */
/*    cc -qflttrap=ov:zero:inv:enable \
                        -qfloat=nofold ...   */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <fptrap.h>

jmp_buf cmd_prompt;


fptrap_t get_flttrap_mask( void ) {
  fptrap_t mask = 0;

                  /* Invalid Operation Summary */
  if ( fp_is_enabled(TRP_INVALID) )
     mask |= TRP_INVALID;
                  /*            Divide by Zero */
  if ( fp_is_enabled(TRP_DIV_BY_ZERO) )
     mask |= TRP_DIV_BY_ZERO;
                  /*                  Overflow */
  if ( fp_is_enabled(TRP_OVERFLOW) )
     mask |= TRP_OVERFLOW;
                  /*                 Underflow */
  if ( fp_is_enabled(TRP_UNDERFLOW) )
     mask |= TRP_UNDERFLOW;
                  /*            Inexact Result */
  if ( fp_is_enabled(TRP_INEXACT) )
     mask |= TRP_INEXACT;
  return ( mask );
}
```



```
extern void fpe_handler(int sig_number);
extern double f(double x);


int main( void )
{
  fptrap_t mask;
  double   x;

  printf("f(x) = 1 / (1 - x*x)\n");
  mask = get_flttrap_mask();
  if ( setjmp(cmd_prompt) != 0 ) {
    fprintf(stderr, "Returned from SIGTRAP: Try "
      "another value for \"x\"\n");
    fp_enable(mask);
  }
             /* Signalbehandlungsroutine     */
             /*   fuer "Trace trap"-Signale  */
             /*   installieren:              */
  if ( signal(SIGTRAP, fpe_handler) == SIG_ERR )
  {
    fprintf(stderr, "Could not set SIGTRAP\n");
    abort();
  }

  while ( scanf("%lf", &x) == 1 )
    printf("f(%g) = %g\n", x, f(x));

  if ( signal(SIGTRAP, SIG_DFL) == SIG_ERR ) {
    fprintf(stderr, "Could not lower SIGTRAP\n");
    abort();
  }
  exit(EXIT_SUCCESS);
}
```

Zeitfunktionen in <time.h>

Datentypen zur Darstellung von Zeiten

clock_t arithmetischer Typ zur Darstellung von CPU-Zeiten

struct tm Strukturtyp mit (mindestens) den folgenden Komponenten:

int tm_sec;	Sekunden (0–61)[25]
int tm_min;	Minuten (0–59)
int tm_hour;	Stunden seit Mitternacht (0–23)
int tm_mday;	Tag des Monats (1–31)
int tm_mon;	Monate *seit* Januar (0–11)
int tm_year;	Jahre seit 1900
int tm_wday;	Wochentag seit Sonntag (0–6)
int tm_yday;	Tag seit 1. Januar (0–365)
int tm_isdst;	Sommerzeit-Flag (***D****aylight* ***S****aving* ***T****ime flag*): >0 Sommerzeit ==0 Normalzeit <0 Information nicht vorhanden

time_t arithmetischer Typ zur Darstellung von Datum und Uhrzeit

[25] Es sind bis zu 2 Schaltsekunden möglich.

Beschreibung der Funktionen

Funktionsprototyp	Beschreibung
clock_t clock(void);	Liefert die bis jetzt vom Programm verbrauchte Prozessorzeit. Der Resultatwert dividiert durch **CLOCKS_PER_SEC** liefert die Prozessorzeit in Sekunden. Falls die Prozessorzeit nicht verfügbar ist, wird **(clock_t) -1** zurückgegeben.
double difftime(**time_t** *time1*, **time_t** *time0*);	Liefert als Ergebnis die Differenz zwischen zwei Kalenderzeiten(*time1*-*time0*) in Sekunden.
time_t mktime(**struct tm** **timeptr*);	Wandelt die Zeitdarstellung in *timeptr* in eine Zeitdarstellung des Typs **time_t** um. Dabei erhalten die Komponenten von **timeptr* Werte in den angegebenen Bereichen.
time_t time(time_t **timer*);	Ermittelt die aktuelle Kalenderzeit. Ist *timer* ungleich **NULL**, so wird die Zeit auch nach ******timer* gespeichert. Das Funktionsergebnis ist **(time_t) -1**, falls keine Kalenderzeit verfügbar ist.

Die nächsten 4 Funktionen speichern ihr Resultat jeweils in einem statischen Speicherbereich und geben einen Zeiger auf den Bereich zurück. (Folgende Aufrufe überschreiben den Inhalt des jeweiligen Speicherbereiches.)

Funktionsprototyp	Beschreibung
char *asctime(const struct tm *_timeptr_);	Diese Funktion wandelt eine Zeit in einen String der Form _Thu Mar 14 08:22:52 1991\n\0_ um.
char *ctime(const time_t *_timer_);	⇔ **asctime(localtime(_timer_));**
struct tm *gmtime(const time_t *_timer_);	Die Funktion wandelt eine Kalenderzeit vom Typ **time_t** in eine Kalenderzeit vom Typ **struct tm** um, wobei zusätzlich diese Zeit in eine _Coordinated Universal Time (UTC)_ umgewandelt wird. Liefert **NULL**, falls _UTC_ nicht verfügbar.
struct tm *localtime(const time_t *_timer_);	Diese Funktion wandelt eine Kalenderzeit vom Typ **time_t** in eine Kalenderzeit vom Typ **struct tm** um. Gleichzeitig findet eine Anpassung an die lokale Zeit (Zeitzone) statt.
size_t strftime(char *_s_, size_t _maxsize_, const char *_format_, const struct tm *_timeptr_);	Flexible Funktion zur Erzeugung eines Datum-/Zeitstrings, wobei _format_ den Aufbau des Strings angibt.

Beispiel:

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

char *getdate(char *timestr)
{
  static char buf[9];
  time_t      now;

  if (timestr == NULL)
    timestr = buf;
  /* get date and time from system */
  time(&now);

  strftime(timestr, 9, "%m/%d/%y",
    localtime(&now));

  return(timestr);
}
```

Anhang A: C-Programmierwerkzeuge

Funktion/ Zweck	Kommando(s)	AIX	DEC	Sun	Cray
C Präprozessor	cpp	+	+	+	+
C Semantik Prüfprogramm	lint	+	+	+	+
automatische Generierung von ausführbaren Programmen	make	+	+	+	+
automatische Generierung von make-Files	bldmake[26]	+	+	-	-
Erstellung von Laufzeitstatistiken	prof	+	+	+	+
	gprof	+	+	+	-
C Beautifier/ Pretty Printer	cb	+	+	+	+
	indent	+	+	+	-
	tgrind	+	+	+	-
	c2latex[27]	+	+	+	-
Erzeugung von Cross-Reference-Listen	cflow	+	+	+	+
	cxref	+	+	+	+
symbolischer Debugger mit X-Window-Oberfläche	xldb	+	-	-	-
	debugger	-	-	+	-
	totalview	-	-	-	+
	xxgdb[28]	+	+	+	-
Run-Time-Debugging-Tool zumAuffinden von **malloc**-Problemen	purify[29]	-	-	(+)	-

[26] Eigenentwicklung der KFA (siehe TKI-0276 für nähere Informationen)

[27] Die Shell-Variable PATH muß ggf. um `/usr/local/texmf/bin` erweitert werden.

[28] X-Window-Schnittstelle zum GNU Debugger (`gdb`)

[29] kommerzielles Tool der Firma Pure Software

Beispiel für **tgrind**:

```
tgrind -lc \
  -h 'Beispiel: Verketten mehrerer Strings' \
  multcat.c
dvips -Pzam00pd tgrind.dvi
```

Ausgabe:

Beispiel: Verketten mehrerer Strings **main(multcat.c)**

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *
multcat(int numargs, ... )                                          multcat
{
        va_list argptr;                                                  10
        char *result;
        int i, siz;

        /* get size required */
        va_start(argptr, numargs);
        for(siz = i = 0; i < numargs; i++)
                siz += strlen(va_arg(argptr, char *));

        if ((result = calloc(siz + 1, 1)) == NULL) {
                fprintf(stderr, "Out of space\n");                       20
                exit(EXIT_FAILURE);
        }
        va_end(argptr);

        va_start(argptr, numargs);
        for(i = 0; i < numargs; i++)
                strcat(result, va_arg(argptr, char *));
        va_end(argptr);
        return(result);
}                                                                        30

int
main(void)                                                          main
{       printf(multcat(5, "One ", "two ", "three ",
                "testing", ".\n"));
        exit(EXIT_SUCCESS);
}
```

15:55 Oct 31 1995 *Page 1 of multcat.c*

Beispiel für **c2latex**:

```
/usr/local/texmf/bin/c2latex -a -c multcat.c
latex multcat.tex
dvips -Pzam00pd multcat.dvi
```

Ausgabe:

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

char *
multcat(int numargs, ... )
{
      va_list argptr;
      char *result;
      int i, siz;

      /* get size required */
      va_start(argptr, numargs);
      for(siz = i = 0; i < numargs; i++)
            siz += strlen(va_arg(argptr, char *));

      if ((result = calloc(siz + 1, 1)) == NULL) {
            fprintf(stderr, "Out of space\n");
            exit(EXIT_FAILURE);
      }
      va_end(argptr);

      va_start(argptr, numargs);
      for(i = 0; i < numargs; i++)
            strcat(result, va_arg(argptr, char *));
      va_end(argptr);
      return(result);
}

int
main(void)
{ printf(multcat(5, "One ", "two ", "three ",
            "testing", ".\n"));
      exit(EXIT_SUCCESS);
}
```

1

Beispiele für **cflow**:

```
cflow -DFIND2 findmain.c find.c
```

Ausgabe:

```
1 main: (void) int, <findmain.c 7>
2       time: () int, <>
3       srand: (unsigned) void, <>
4       rand: (void) int, <>
5       assign: (int, short) void, <find.h 55>
6       printf: (* const char, ellipsis) int, <>
7       prvek: (void) void, <find.h 61>
8               printf: 6
9       find2: (short) int, <find.h 32>
10 findelem: (short, int) int, <find.h 11>
```

invertierte Liste (*Caller*):

```
cflow -r findmain.c find.c
```

Ausgabe:

```
1 assign: (int, short) void, <find.h 55>
2         main : () int, <>
3 find2: (short) int, <find.h 32>
4 findelem: (short, int) int, <find.h 11>
5         main : 2
6 main: (void) int, <findmain.c 7>
7 printf: () int, <>
8         main : 2
9         prvek : () int, <>
  .
  .
  .
```

make

- ist ein für die Programmentwicklung außerordentlich hilfreiches Unix-Programm.
- generiert selbsttätig Kommandos zur Erzeugung bestimmter Dateien (z.B. Aufruf des Compilers, um Objektdatei zu erzeugen, oder Aufruf des Binders (Linkers) zur Erzeugung eines ausführbaren Programmes).
- führt die generierten Kommandos automatisch aus.
- ist in der Lage, nach einer Modifikation an einem größeren Programmsystem, selbständig herauszufinden, welche Quelldateien neu übersetzt werden müssen.

Damit **make** z.B. ein ausführbares Programm erzeugen kann, benötigt es Informationen über das Programmsystem. Diese werden **make** in einer Beschreibungsdatei (*Makefile*) mitgeteilt. Ein Makefile kann folgende Informationen beinhalten:

- Beschreibung der gegenseitigen Abhängigkeiten, der an der Erzeugung einer Datei (z.B. eines ausführbaren Programmes) beteiligten Dateien.
- Kommandos für die Erzeugung dieser Dateien.
- Makrodefinitionen, z.B. um Compiler-Optionen festzulegen.
- Regeln, die festlegen, wie aus einer Datei mit einer bestimmten Endung (z.B. *.c*) eine Datei mit einer anderen Endung (z.B. *.o*) erzeugt werden kann (*Suffix rule*).
- evtl. **include**-Anweisungen

I. allg. braucht die Beschreibung durch den Makefile nicht vollständig zu sein, da eine Reihe von Abhängigkeiten und Kommandos bereits **make**-intern definiert sind. Ebenso gibt es **make**-interne Standarddefinitionen für Makros.

Beachte: Ein Makefile ist **kein** Programm! Die Abarbeitung eines Makefiles erfolgt nicht-prozedural. (Daher ist auch die Reihenfolge, in der Abhängigkeiten und Makrodefinitionen aufgeführt sind, in der Regel ohne Bedeutung.)

Dependency lines

targets: $prerequisites_{opt}$

targets	Dateien, die erzeugt werden sollen.
prerequisites	Dateien, die benötigt werden, um die *target*-Dateien zu erzeugen.

Einer *dependency line* folgen häufig Kommandos, die ausgeführt werden sollen, um aus den *prerequisites* die *target*-Dateien zu erzeugen:

targets: $prerequisites_{opt}$
tab $command_1$
tab . . .
tab $command_n$

Makros

Makrodefinition

name=*string*

Makroaufruf

$(*name***)** oder **${***name***}**

Eine Makrodefinition darf Makroaufrufe enthalten. Die Definition eines Makros darf jedoch keine Bezugnahme auf sich selbst beinhalten. Die folgende Definition des Makros `CCOPTS` ist daher unzulässig.

```
CCOPTS = -DFIND2
DEBUGOPTS = -g
CCOPTS = $(CCOPTS) $(DEBUGOPTS)
```

Möglichkeiten zur Definition von Makros:

(Reihenfolge entspricht der Priorität:
1 = höchste Priorität, 4 = geringste Priorität)

1. Makrodefinition auf der Kommadozeile (**make**-Aufruf)
2. Makrodefinition innerhalb der Beschreibungsdatei (Makefile)
3. Shell-Variablen, z.B. $(HOME)
4. interne Standarddefinitionen von **make**

interne Makros

`$?` Liste der *prerequisites*, die neueren Datums als das aktuelle *target* sind. (In Kommandos einer *suffix rule* nicht zulässig.)

`$@` Name des aktuellen *targets*.

`$$@` Name des aktuellen *targets*. (Nur rechts vom Doppelpunkt innerhalb einer *dependency line* erlaubt.)

Beispiel:

```
prg: $$@.c
```

`$<` aktueller *prerequisite*-Name. Es handelt sich um den Namen einer Datei, die neueren Datums als das aktuelle *target* ist. (nur in *suffix rules*!)

`$*` aktueller *prerequisite*-Name ohne Suffix. (nur in *suffix rules*!)

Makro String Substitution

nicht bei allen Varianten von **make** verfügbar.

`${`*macroname***:***s1*=*s2***`}`**

Suffix rules

$\mathit{.suffix}_1\mathit{.suffix}_2$**:**

Eine *suffix rule* teilt **make** mit, daß eine Datei

*name***.*suffix***$_1$

prerequisite einer Datei *name***.*suffix***$_2$ sein kann (implizite Abhängigkeit).

Es sollte mindestens ein Kommando folgen, das auszuführen ist, um aus der Datei *name***.*suffix***$_1$ die Datei *name***.*suffix***$_2$ zu erzeugen.

Beispiel:

```
.c.o:
tab$(CC) $(CFLAGS) -c $<
```

Single-suffix rules

nicht bei allen Varianten von **make** verfügbar.

***.suffix*:**

Eine solche Regel spezifiziert, wie aus einer Datei *name***.*suffix*** eine Datei *name* erzeugt werden kann.

Beispiel:

```
.c:
tab$(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
```

Da es sich bei dieser Regel um eine interne Standardregel handelt, wird kein Makefile benötigt, um aus einer Quelldatei *name***.c** ein ausführbares Programm *name* zu erzeugen. Es genügt das Kommando

make *name*

Index

! (logische Umkehrung) 102
!= (ungleich) 101, 218
(Präprozessor-Direktive) 200
(Präprozessor-Operator) 179
(Präprozessor-Operator) 181
% (Modulo-Operator) 91
%% (Konvertierungsspezifikation) 63, 71
%= (Zuweisungsoperator) 104
%[. . .] (Konvertierungsspezifikation) 242
%[ˆ . . .] (Konvertierungsspezifikation) 242
& (Adreßoperator) 11, 203
& (bitweises Und) 96
&& (logisches Und) 102
&= (Zuweisungsoperator) 104
() (Funktionsaufruf) 147
. . . (variable Argumentliste) 310
* (Dereferenzierungsoperator) 205
* (Multiplikationsoperator) 91
*= (Zuweisungsoperator) 104
+ (Additionsoperator) 91
+ (Vorzeichenoperator) 91
++ (Inkrement-Operator) 94
+= (Zuweisungsoperator) 104
, (Komma-Operator) 108, 114
- (Subtraktionsoperator) 91
- (Vorzeichenoperator) 91
-- (Dekrement-Operator) 94
-= (Zuweisungsoperator) 104
-> (Komponentenverweis-Operator) 298
. (Komponentenauswahl-Operator) 293
/ (Divisionsoperator) 91
/= (Zuweisungsoperator) 104
< (kleiner) 101, 218
<< (Links-Shift) 96
<<= (Zuweisungsoperator) 104
<= (kleiner gleich) 101, 218
= (Zuweisungsoperator) 84
== (gleich) 101, 218
> (größer) 101, 218
>= (größer gleich) 101, 218
>> (Rechts-Shift) 96
>>= (Zuweisungsoperator) 104
?: (Bedingungsoperator) 106
[] (Index-Operator) 221
ˆ (bitweises exklusives Oder) 96
ˆ= (Zuweisungsoperator) 104
{} (Verbundanweisung) 9, 128
| (bitweises Oder) 96
|= (Zuweisungsoperator) 104
|| (logisches Oder) 102
˜ (bitweises Komplement) 96
\0 (Null-Zeichen) 40
\? 38, 40
\\ 37, 38
\' 37, 38
\" 38, 40

A

\a (Alarmzeichen) 38
ANSI C 2, 144, 201
abort() 323
abs() 122
acos() 118
Adreß-Arithmetik 217
Adreß-Umrechnungsfunktion 275, 278
Adreßoperator 11, 203
analysieren, Ausdruck 109
Anweisung
- Ausdrucks- 127
- bedingte Anweisungen 129
  - else 129
  - if 129
  - switch 131
- gelabelte Anweisungen 128
  - case 128, 131
  - default 128, 131
- leere 127
- Sprunganweisungen 136
  - break 136
  - continue 137
  - goto 137
  - return 136
- Verbund- 128
- Wiederholungsanweisungen 133
  - do 133
  - for 134
  - while 133

argc (*argument count*) 230
Argument-Erweiterung 80, 148, 310
Argumentliste, variable 310
argv (*argument vector*) 230
arithmetische Operatoren, *siehe* Operator
arithmetische Umwandlungen 88
arithmetischer Typ, *siehe* Datentyp
array, *siehe* Feld
asctime() 367
asin() 118
assert() 322
assert.h 322
atan() 118
atan2() 118
atexit() 324
atof() 247
atoi() 247
atol() 247
Aufzählungskonstante 39, 57
Aufzählungstyp 57
Ausdruck
- analysieren 109
- bewerten 112
- konstanter 115

Ausdrucksanweisung 127
ausführen, Programm 16
Ausgabe
- von Gleitkommatypen 79
- von Integer-Typen 61
- von Zeichenketten 238, 240
- von Zeigerwerten 210
- von einzelnen Zeichen 63, 234, 235

Ausgabeumlenkung 16
Ausrichtung
- eines Bitfeldes 308, 309
- von Strukturkomponenten 295
- von dynamisch angelegten Speicherbereichen 215

auto 155

B

\b (Backspace) 38
BUFSIZ 347
bedingte Anweisungen, *siehe* Anweisung
bedingte Übersetzung 23, 193
Bedingungsoperator 106
beenden, Programm 323
Beispielfunktionen/-programme/-makros, *siehe* Funktionen/Programm/Makros, Beispiel-
bewerten, Ausdruck 112
Bezeichner 26, 27
binden, Programm 14
Bindung 160
Bit-Operatoren, *siehe* Operator
Bitfeld 307
Block 9, 128
break-Anweisung 136
bsearch() 321

C

%c 63, 71, 242
CHAR_BIT 52
CHAR_MAX 53
CHAR_MIN 53
CLOCKS_PER_SEC 366
CPU-Zeit 365
call by reference 206, 292
call by value 147, 292
calloc() 214, 215
carriage return, *siehe* Zeilenrücklauf
case 128, 131
cast-Operator 85
ceil() 122
char 52, 53, 59
Character-Konstante, *siehe* Zeichenkonstante
clearerr() 332
clock() 366
clock_t 365
Compiler-Optionen 14
compound statement, *siehe* Verbundanweisung
conditional operator, *siehe* Bedingungsoperator
const 211
continue-Anweisung 137
cos() 118
cosh() 119
ctime() 367
ctype.h 54

D

%d 10, 63, 71
__DATE__ 201
DBL_DIG 76
DBL_EPSILON 76
DBL_MANT_DIG 78

DBL_MAX 76
DBL_MAX_10_EXP 78
DBL_MAX_EXP 78
DBL_MIN 76
DBL_MIN_10_EXP 78
DBL_MIN_EXP 78
Datei
  abschließen 331
  eröffnen 328
  erweitern 329
  -position 333
  temporäre 348, 349
  -Zeiger 328
Dateiende 12, 332
Datentyp
  arithmetischer Typ 46
    clock_t 365
    Gleitkommatyp 46
      double 76
      float 76
      long double 77
    Integer-Typ, *siehe* Datentyp, arithmetischer Typ, integraler Typ
    integraler Typ 46
      Aufzählungstyp 57
        enum 39, 57
      char 52, 53, 59
      int 9, 50
      long int 50
      ptrdiff_t 218
      short 49, 59
      sig_atomic_t 356
      signed char 52, 53, 59
      signed int 50
      signed long 50
      signed short 49, 59
      size_t 48
      unsigned char 52, 53, 59
      unsigned int 51
      unsigned long 51
      unsigned short 51, 59
    time_t 365
  Attribute für 211
  benennen, *siehe* typedef
  Funktionstyp, *siehe* Funktion
  Vereinigungstyp 304
  void 9, 142
  Zeigertyp
    Beispiel für, *siehe* Vereinbarung, eines Zeigers
    FILE * 328
    void * 208
  zusammengesetzte Typen
    Strukturtyp 289–291, 293
      div_t 92
      ldiv_t 92
      struct tm 365
    Vektortyp, *siehe* Vektor
      jmp_buf 359
Datum 201, 365
default 128, 131
#define 5, 173, 175
defined-Präprozessor-Operator 194
Definition 141
  Funktions- 141, 144, 145
  von top-level-Variablen 158
  vorläufige 158
Definitionsdatei, *siehe* Header-Datei
Deklaration 141
  Funktions- 141–143
    implizite 143
  von top-level-Variablen 158
    Vektoren 224
Dekrement-Operator 94
dereferencing operator, *siehe* Dereferenzierungsoperator
Dereferenzierungsoperator 205
Dezimalkonstante 32, 33
difftime() 366
div() 92, 122
div_t 92
do-Schleife 133
domain error, *siehe* EDOM
double 76
double-Konstante, *siehe* Gleitkommakonstante
dynamische Speicherverwaltung 214

E

%E 80
%e 80, 82
EDOM 123
EOF 12
ERANGE 123
EXIT_FAILURE 7, 8

EXIT_SUCCESS 6–8
einfach verkettete Liste 299
Eingabe
  von Gleitkommatypen 82
  von Integer-Typen 68
  von Zeichenketten 239, 241
  von einzelnen Zeichen 71, 236
Eingabeumlenkung 16
#elif 193
else 129
#else 193
end-of-file-Bedingung 332
#endif 23, 193
enum 39, 57
enumerated type, *siehe* Aufzählungstyp
enumeration constant, *siehe* Aufzählungskonstante
environment variables 325
errno 123, 125
errno.h 123
#error 199
escape sequence, *siehe* Fluchtsymbol-Darstellung
exit() 7, 323
exp() 119
explizite Typumwandlung, *siehe* cast-Operator
expression statement, *siehe* Ausdrucksanweisung
extern 157

F

F-Suffix 35, 36
%f 80, 82
\f (Seitenvorschub) 38
FILE * 328
__FILE__ 201
FILENAME_MAX 329
FLT_DIG 76
FLT_EPSILON 76
FLT_MANT_DIG 78
FLT_MAX 76
FLT_MAX_10_EXP 78
FLT_MAX_EXP 78
FLT_MIN 76
FLT_MIN_10_EXP 78
FLT_MIN_EXP 78
FLT_RADIX 78
FLT_ROUNDS 77
FOPEN_MAX 329
fabs() 122
false, logischer Wert 56, 209
fclose() 331
Feld 40, 274
  als Formalparameter 275–277
  Dimensionslängen eines 275
  eindimensionales, *siehe* Vektor
  indizieren 275
  Initialisierung von 279
  Speicherung eines 274
  Vereinbarung eines 274
  Vergleich mit Vektor von Zeigern 280
feof() 332
ferror() 332
fflush() 346
fgetc() 236
fgetpos() 338
fgets() 239
float 76
float-Konstante, *siehe* Gleitkommakonstante
float.h 76, 77
floor() 122
Fluchtsymbol-Darstellung 10, 37, 38, 40
fmod() 122
fopen() 328
for-Schleife 134
form feed, *siehe* Seitenvorschub
Formatbuchstabe, *siehe* Konvertierungsspezifikation
formatierte Ein-/Ausgabe, *siehe* Eingabe/Ausgabe
Fortsetzungszeile 43
fpos_t 338
fprintf() 244
fputc() 234
fputs() 238
fread() 339
free() 215
freopen() 330
frexp() 119
fscanf() 244
fseek() 334
fsetpos() 338
ftell() 333

Funktion
als Formalparameter 285
-argumente 147
-aufruf 147, 284
-definition 141, 144, 145
-deklaration 141–143
implizite 143
mit variabler Argumentliste 310
-name als Zeiger 283
-prototyp 142
Funktionen
Beispiel-
cmsfname() (MS-DOS-Dateiname) 271
combine() (Verketten von 2 Strings) 269
copy() (Kopieren eines Strings) 270
error() (Fehlerbehandlungsroutine) 315
fpe_handler() (Signalbehandlungsroutine) 361
getdate() (Datumstring erzeugen) 368
gewinnzahl() (Signalbehandlungsroutine) 357
multcat() (Verketten mehrerer Strings) 312
random() (Pseudo-Zufallszahlen-Generator) 157
strlen() (Länge eines Strings bestimmen) 222
swap() (Vertauschen von zwei Variablen) 206
trim() (Leerzeichen am Ende eines Strings entfernen) 341
in <ctype.h>
Umwandlungsfunktionen
tolower() 55
toupper() 55
Zeichenklassen-Tests
isalnum() 54
isalpha() 54
iscntrl() 55
isdigit() 54
isgraph() 55
islower() 54
isprint() 55
ispunct() 55
isspace() 55
isupper 54
isxdigit() 54
in <math.h>
Exponential– und Logarithmusfunktionen
exp() 119
frexp() 119
ldexp() 119
log() 119
log10() 119
modf() 119
hyperbolische Funktionen
cosh() 119
sinh() 119
tanh() 119
Potenzfunktionen
pow() 121
sqrt() 121
sonstige mathematische Funktionen
ceil() 122
fabs() 122
floor() 122
fmod() 122
trigonometrische Funktionen
acos() 118
asin() 118
atan() 118
atan2() 118
cos() 118
sin() 118
tan() 118
in <setjmp.h>
longjmp() 360
setjmp() 359
in <signal.h>
raise() 356
signal() 355
in <stdio.h>
Ausgabe
formatiert
fprintf() 244
printf() 10, 59, 61, 63, 79, 80
sprintf() 244
vfprintf() 314
vprintf() 314
vsprintf() 314
fputc() 234
fputs() 238
putc() 234
putchar() 12, 235
puts() 238
unformatiert
fwrite() 339
Dateipositionierung

fgetpos() 338
fseek() 334
fsetpos() 338
ftell() 333
rewind() 338
Dateizugriff
fclose() 331
fflush() 346
fopen() 328
freopen() 330
setbuf() 348
setvbuf() 347
tmpfile() 348
tmpnam() 349
Eingabe
fgetc() 236
fgets() 239
formatiert
fscanf() 244
scanf() 11, 68–71
sscanf() 244
getc() 236
getchar() 12, 236
gets() 239
unformatiert
fread() 339
ungetc() 237
Fehlerbehandlung
clearerr() 332
feof() 332
ferror() 332
perror() 124
Löschen/Umbenennen einer Datei
remove() 351
rename() 351
in <stdlib.h>
dynamische Speicherverwaltung
calloc() 214, 215
free() 215
malloc() 214, 215
realloc() 215
ganzzahlige Arithmetik
abs() 122
div() 92, 122
labs() 122
ldiv() 92, 122
Kommunikation mit Systemumgebung
abort() 323
atexit() 324
exit() 7, 323
getenv() 325
system() 327
Pseudo-Zufallszahlen
rand() 319
srand() 319
Suchen/Sortieren
bsearch() 321
qsort() 320
Umwandlungsfunktionen
atof() 247
atoi() 247
atol() 247
strtod() 246
strtol() 245
strtoul() 246
in <string.h> 248
Kopierfunktionen
memcpy() 252
memmove() 253
strcpy() 250
strncpy() 251
sonstige Stringfunktionen
memset() 268
strerror() 124
Suchfunktionen
memchr() 267
strchr() 259
strcspn() 260
strpbrk() 261
strrchr() 262
strspn() 263
strstr() 264
strtok() 265
Vergleichsfunktionen
memcmp() 258
strcmp() 256
strncmp() 257
Verkettungsfunktionen
strcat() 254
strncat() 255
in <time.h>
Zeithandhabung

clock() 366
difftime() 366
mktime() 366
time() 366
Zeitumwandlung
asctime() 367
ctime() 367
gmtime() 367
localtime() 367
strftime() 367
fwrite() 339

G

%G 80
%g 80, 82
gelabelte Anweisungen, *siehe* Anweisung
generischer Zeiger 208
getc() 236
getchar() 12, 236
getenv() 325
gets() 239
Gleitkommakonstante 35
Gleitkommatyp, *siehe* Datentyp
globale Variable 149
gmtime() 367
goto-Anweisung 137
Grundsymbol 24
Gültigkeitsbereich 150

H

%h<Formatbuchstabe> 62, 69
HUGE_VAL 123
Header-Datei 8, 14, 164
assert.h 322
ctype.h 54
errno.h 123
float.h 76, 77
limits.h 49, 52
math.h 118, 123
setjmp.h 359
signal.h 354, 355
stdarg.h 310
stddef.h 48, 209, 218, 297
stdio.h 8, 234, 314, 328
stdlib.h 8, 214, 245, 319, 323
string.h 248
time.h 365
Hexadezimalkonstante 32, 33

I

%i 63, 71
INT_MAX 50
INT_MIN 50
_IOFBF 347
_IOLBF 347
_IONBF 347
#if 193
if-Anweisung 129
#ifdef 23, 194
#ifndef 194
implementierungsabhängig 37, 52, 86, 91, 95, 97, 123, 199, 210, 211, 214, 218, 309, 327, 333, 354
implizite Funktionsdeklaration 143
#include 8, 171
Index-Operator 221
indirection operator, *siehe* Dereferenzierungsoperator
Inhaltsoperator, *siehe* Dereferenzierungsoperator
Initialisierung 15, 225
einer Union (Vereinigung) 304
von Strukturen 289
von Vektoren 225
mehrdimensionale Vektoren 279
Zeichenvektoren 227
von auto-, register-Variablen 155, 225
von top-level-, static-Variablen 156, 225
Inkrement-Operator 94
input redirection, *siehe* Eingabeumlenkung
int 9, 50
Integer-Erweiterung 59
Integer-Konstante 32, 33
Integer-Typ, *siehe* Datentyp
integral promotion, *siehe* Integer-Erweiterung
integraler Typ, *siehe* Datentyp
Interrupt 352
interaktiver 354
-Routine, *siehe* Signalbehandlungsroutine
isalnum() 54
isalpha() 54
iscntrl() 55
isdigit() 54
isgraph() 55
islower() 54
isprint() 55

ispunct() 55
isspace() 55
isupper() 54
isxdigit() 54
iteration statements, *siehe* Anweisung, Wiederholungsanweisungen

J

jmp_buf 359
jump statements, *siehe* Anweisung, Sprunganweisungen

K

K&R-C 2, 143, 145
Komma-Operator 108, 114
Kommandozeilen-Argumente 230
Kommentar 22
Konstante 30
Aufzählungs- 39, 57
benennen 5, 39
Gleitkomma- 35
Integer- 32, 33
Dezimalkonstante 32, 33
Hexadezimalkonstante 32, 33
Oktalkonstante 32, 33
String-, *siehe* String
Zeichen- 37
konstanter Ausdruck 115
Kontroll-String 10, 70
Konventionen 14, 27, 155
Konvertierungsspezifikation
printf- 10, 61, 79, 210, 240
scanf- 68, 82, 241
kopieren, String 248

L

%L<Formatbuchstabe> 79, 82
L_tmpnam 349
l-Suffix 32, 33, 36
l-Wert, *siehe lvalue*
%l<Formatbuchstabe> 62, 69, 82
LDBL_DIG 77
LDBL_EPSILON 77
LDBL_MANT_DIG 78
LDBL_MAX 77
LDBL_MAX_10_EXP 78
LDBL_MAX_EXP 78
LDBL_MIN 77
LDBL_MIN_10_EXP 78
LDBL_MIN_EXP 78
__LINE__ 201
LONG_MAX 50
LONG_MIN 50
labeled statements, *siehe* Anweisung, gelabelte Anweisungen
labs() 122
ldexp() 119
ldiv() 92, 122
ldiv_t 92
leere Anweisung 127
lexical scope, *siehe* Gültigkeitsbereich
limits.h 49, 52
#line 200
Liste, einfach verkettete 299
localtime() 367
log() 119
log10() 119
logische Operatoren, *siehe* Operator
logische Werte 56, 209
long double 77
long double-Konstante, *siehe* Gleitkommakonstante
long int 50
longjmp() 360
lvalue 84

M

main 9, 230
Makro 5
einfaches 173
definieren 5, 14, 173
-expansion 177
löschen 192
mit Parametern 175
aufrufen 176
definieren 175
Probleme mit 182
vordefiniertes 201
Vorteile von 191

Makros
Beispiel-
display() 179
swap() (Vertauschen von zwei Variablen) 190
vectorsize() (Größe eines Vektors ermitteln) 226
definiert in <assert.h>
assert() 322
definiert in <float.h>
DBL_DIG 76
DBL_EPSILON 76
DBL_MANT_DIG 78
DBL_MAX 76
DBL_MAX_10_EXP 78
DBL_MAX_EXP 78
DBL_MIN 76
DBL_MIN_10_EXP 78
DBL_MIN_EXP 78
FLT_DIG 76
FLT_EPSILON 76
FLT_MANT_DIG 78
FLT_MAX 76
FLT_MAX_10_EXP 78
FLT_MAX_EXP 78
FLT_MIN 76
FLT_MIN_10_EXP 78
FLT_MIN_EXP 78
FLT_RADIX 78
FLT_ROUNDS 77
LDBL_DIG 77
LDBL_EPSILON 77
LDBL_MANT_DIG 78
LDBL_MAX 77
LDBL_MAX_10_EXP 78
LDBL_MAX_EXP 78
LDBL_MIN 77
LDBL_MIN_10_EXP 78
LDBL_MIN_EXP 78
definiert in <limits.h>
CHAR_BIT 52
CHAR_MAX 53
CHAR_MIN 53
INT_MAX 50
INT_MIN 50
LONG_MAX 50
LONG_MIN 50
SCHAR_MAX 53
SCHAR_MIN 53
SHRT_MAX 49
SHRT_MIN 49
UCHAR_MAX 53
UINT_MAX 51
ULONG_MAX 51
USHRT_MAX 51
definiert in <math.h>
EDOM 123
ERANGE 123
HUGE_VAL 123
definiert in <signal.h>
SIG_DFL 355
SIG_ERR 355
SIG_IGN 355
SIGABRT 354
SIGFPE 354
SIGILL 354
SIGINT 354
SIGSEGV 354
SIGTERM 354
definiert in <stdarg.h>
va_arg() 311
va_end() 311
va_start() 310
definiert in <stddef.h>
NULL 209
offsetof() 297
definiert in <stdio.h>
BUFSIZ 347
EOF 12
FILENAME_MAX 329
FOPEN_MAX 329
_IOFBF 347
_IOLBF 347
_IONBF 347
L_tmpnam 349
SEEK_CUR 334
SEEK_END 334
SEEK_SET 334
TMP_MAX 349
definiert in <stdlib.h>
EXIT_FAILURE 7, 8
EXIT_SUCCESS 6–8
RAND_MAX 319
definiert in <time.h>
CLOCKS_PER_SEC 366
vordefinierte
__DATE__ 201
__FILE__ 201
__LINE__ 201
__STDC__ 201
__TIME__ 201
malloc() 214, 215
Marke 128
maschinenabhängig, *siehe* implementierungsabhängig
math.h 118, 123
mehrdimensionaler Vektor, *siehe* Feld
member, *siehe* Struktur, Komponente /union
memchr() 267
memcmp() 258
memcpy() 252
memmove() 253
memset() 268
mktime() 366
modf() 119
Modulo-Operator 91

N

%n 71, 73
\n (Zeilenendezeichen) 10, 38
NDEBUG 322
NULL 209
name space, *siehe* Namensraum
Namensraum 154
newline, *siehe* Zeilenendezeichen
Null-Byte, *siehe* Null-Zeichen
Null-String 40
Null-Zeichen 40
null statement, *siehe* leere Anweisung
Nullzeiger 209

O

%o 63, 71
offsetof() 297
Oktalkonstante 32, 33
Operator 28
() (Funktionsaufruf) 147
Adreß- (&) 11, 203
arithmetische Operatoren
% (Modulo) 91
* (Multiplikation) 91
+ (Addition) 91
+ (Identität) 91
- (Subtraktion) 91
- (Vorzeichenumkehr) 91
/ (Division) 91
Bedingungs- (?:) 106
Bit-Operatoren
& (bitweises Und) 96
<< (Links-Shift) 96
>> (Rechts-Shift) 96
ˆ (bitweises exklusives Oder) 96
| (bitweises Oder) 96
˜ (bitweises Komplement) 96
cast- 85
Dekrement- (- -) 94
Dereferenzierungs- (*) 205
Index- ([]) 221
Inkrement- (++) 94
Komma- (,) 108, 114
Komponentenauswahl- (.) 293
Komponentenverweis- (->) 298
logische Operatoren
! (logische Umkehrung) 102
&& (logisches Und) 102
|| (logisches Oder) 102
Operatoren mit definierter Bewertungsreihenfolge 114
Präprozessor-Operatoren, *siehe* Präprozessor
sizeof- 47
Vergleichsoperatoren
!= (ungleich) 101, 218
< (kleiner) 101, 218
<= (kleiner gleich) 101, 218
== (gleich) 101, 218
> (größer) 101, 218
>= (größer gleich) 101, 218
Vorrang eines 109
Zuordnung eines 109, 110
Zuweisungsoperatoren
%= 104
&= 104

- *= 104
- += 104
- -= 104
- /= 104
- <<= 104
- = 84
- >>= 104
- ^= 104
- |= 104

- *output redirection*, *siehe* Ausgabeumlenkung
- *Overflow* 123

P

- %p 210
- *padding* 295, 307, 309
- perror() 124
- Pointer, *siehe* Zeiger
- pow() 121
- #pragma 199
- Präprozessor
  - Ausgabe des 15
  - Direktive 170
    - # 200
    - #define 5, 173, 175
    - #elif 193
    - #else 193
    - #endif 23, 193
    - #error 199
    - #if 193
    - #ifdef 23, 194
    - #ifndef 194
    - #include 8, 171
    - #line 200
    - #pragma 199
    - #undef 192
  - Makro, *siehe* Makro(s)
  - Operatoren
    - # 179
    - ## 181
    - defined 194
- printf() 10, 59, 61, 63, 79, 80
- Programm
  - ausführen 16
  - beenden 323
  - Beispiel-
    - Ausgabe von Zahlen 4, 11
    - Echo der Kommandozeilen-Argumente 232
    - Erkennen von UNIX-Kommandos (unvollständig) 233
    - Fallgeschwindigkeit 138
    - fortune (UNIX-Kommando) 335
    - Kopieren der Eingabe zur Ausgabe 12, 13
    - Lottozahlen 356
    - MS-DOS-Dateiname 271
    - SIGFPE Signalbehandlung 361
      - Version für C for AIX Compiler 363
    - Telefonverzeichnis
      - durchsuchen 343
      - erstellen 340
    - Verketten mehrerer Strings 312
    - Verketten von 2 Strings 269
    - Zählen von Zeichen 135
    - Zeilen einlesen (unvollständig) 243
  - binden 14
  - übersetzen 14
- ptrdiff_t 218
- Punktsymbol 29, 114
- putc() 234
- putchar() 12, 235
- puts() 238

Q

- qsort() 320

R

- \r (Zeilenrücklauf) 38
- RAND_MAX 319
- raise() 356
- rand() 319
- *range error*, *siehe* ERANGE
- *read-only storage*, *siehe* schreibgeschützter Speicher
- realloc() 215
- register 156
- remove() 351
- rename() 351
- reservierte Wörter, *siehe* Schlüsselwörter
- return-Anweisung 136
- rewind() 338

S

- %s 240, 242
- SCHAR_MAX 53
- SCHAR_MIN 53
- SEEK_CUR 334
- SEEK_END 334
- SEEK_SET 334
- SHRT_MAX 49
- SHRT_MIN 49
- SIG_DFL 355
- SIG_ERR 355
- SIG_IGN 355
- SIGABRT 354
- SIGFPE 354
- SIGILL 354
- SIGINT 354
- SIGSEGV 354
- SIGTERM 354
- __STDC__ 201
- scanf() 11, 68–71
- Schlüsselwörter 25, 26
- schreibgeschützter Speicher 15, 211
- Seitenvorschub 38
- *selection statements*, *siehe* Anweisung, bedingte Anweisungen
- setbuf() 348
- setjmp() 359
- setjmp.h 359
- setvbuf() 347
- short 49, 59
- sig_atomic_t 356
- Signal 352
- *signal handler*, *siehe* Signalbehandlungsroutine
- signal() 355
- signal.h 354, 355
- Signalbehandlungsroutine 352, 355
- signed char 52, 53, 59
- signed int 50
- signed long 50
- signed short 49, 59
- sin() 118
- sinh() 119
- size_t 48
- sizeof-Operator 47
- sortieren, Vektor 320
- Speicherklasse 155
  - auto 155
  - extern 157
  - register 156
  - static 156
- Speicherverwaltung, dynamische 214
- sprintf() 244
- Sprunganweisungen, *siehe* Anweisung
- sqrt() 121
- srand() 319
- sscanf() 244
- Standardausgabe 10, 16
- Standardeingabe 11, 16
- Standarderweiterung von Argumenten, *siehe* Argument-Erweiterung
- *statement*, *siehe* Anweisung
- static 156
- stdarg.h 310
- stddef.h 48, 209, 218, 297
- stderr 124, 322, 330
- stdin 11, 16, 330
- stdio.h 8, 234, 314, 328
- stdlib.h 8, 214, 245, 319, 323
- stdout 10, 16, 330
- strcat() 254
- strchr() 259
- strcmp() 256
- strcpy() 250
- strcspn() 260
- strerror() 124
- strftime() 367
- String 10, 40
  - durchsuchen 249
  - kopieren 248
  - vergleichen 248
  - verketten 248
  - zerlegen 265
- string.h 248
- strlen() (*siehe auch* Funktionen, Beispiel-, strlen()) 249
- strncat() 255
- strncmp() 257
- strncpy() 251
- strpbrk() 261
- strrchr() 262
- strspn() 263
- strstr() 264

strtod() . . . 246
strtok() . . . 265
strtol() . . . 245
strtoul() . . . 246
struct . . . 289
struct tm . . . 365
structure tag, *siehe* Struktur, -typ, Etikett eines
Struktur . . . 289
  Initialisierung von . . . 289
  Komponente . . . 289
    Zugriff auf . . . 293, 298
  Lücke in einer . . . 295
  rekursive . . . 299
  Speicherabbild einer . . . 295
  -typ . . . 289–291
    Etikett eines . . . 289
    unvollständiger . . . 293
  Vektoren von . . . 299
  Zeiger auf . . . 298
  Zuweisung von . . . 292
Suffix . . . 32, 33, 35, 36
switch-Anweisung . . . 131
system() . . . 327

T

\t (horizontaler Tabulator) . . . 38
__TIME__ . . . 201
TMP_MAX . . . 349
Tabulator . . . 38
tan() . . . 118
tanh() . . . 119
temporäre Datei . . . 348, 349
tentative definition, *siehe* vorläufige Definition
time() . . . 366
time.h . . . 365
time_t . . . 365
tm, *siehe* struct tm
tmpfile() . . . 348
tmpnam() . . . 349
Token, *siehe* Grundsymbol
tolower() . . . 55
top-level-Variable . . . 149, 157
  definieren . . . 158
  deklarieren . . . 158
toupper() . . . 55
Trigraph-Sequenz . . . 21
true, logischer Wert . . . 56
typedef . . . 301
Typname . . . 301
Typumwandlung
  Argument-Erweiterung . . . 80, 148, 310
  arithmetische Umwandlungen . . . 88
  bei Funktionsaufruf . . . 148, 310
  bei Zuweisung . . . 84
  explizit, *siehe* cast-Operator
  Gleitkommatyp → Gleitkommatyp . . . 87
  Gleitkommatyp → Integer-Typ . . . 86
  implizite Umwandlung eines Funktionsnamens . . 283
  implizite Umwandlung eines Vektors . . . 220
  Integer-Erweiterung . . . 59
  Integer-Typ → Gleitkommatyp . . . 87
  Integer-Typ → Integer-Typ . . . 86
  Integer-Typ → Zeigertyp . . . 219
  Zeichenkette → Gleitkommatyp . . . 246, 247
  Zeichenkette → Integer-Typ . . . 245–247
  Zeigertyp → Integer-Typ . . . 219
  Zeigertyp → Zeigertyp . . . 208, 219, 286
Typzusatz
  const . . . 211
  volatile . . . 213

U

%u . . . 63, 71
u-Suffix . . . 32, 33
UCHAR_MAX . . . 53
UINT_MAX . . . 51
ULONG_MAX . . . 51
USHRT_MAX . . . 51
übersetzen, Programm . . . 14
  bedingt . . . 23, 193
Uhrzeit . . . 201, 365
#undef . . . 192
Underflow . . . 123
unformatierte Ein-/Ausgabe . . . 339
ungetc() . . . 237
union . . . 304
unsigned char . . . 52, 53, 59
unsigned int . . . 51
unsigned long . . . 51
unsigned short . . . 51, 59
unspezifischer Zeiger, *siehe* , generischer Zeiger

V

\v (vertikaler Tabulator) . . . 38
va_arg() . . . 311
va_end() . . . 311
va_list . . . 310
va_start() . . . 310
variable Argumentliste . . . 310
Vektor
  als Formalparameter . . . 221
  Argument- (argv) . . . 230
  Deklaration eines . . . 224
  Größe ermitteln . . . 221, 226
  indizieren, *siehe* Index-Operator
  Initialisierung von . . . 225–227
  mehrdimensionaler, *siehe* Feld
  sortieren . . . 320
  Umwandlung in Zeiger . . . 220
  Vergleich mit Zeiger . . . 228
  von Strukturen . . . 299
  von Zeigern . . . 229, 230, 280
Verbundanweisung . . . 128
Vereinbarung
  einer Struktur . . . 290
  eines Vektors . . . 224
  eines Zeigers . . . 205
  eines mehrdimensionalen Feldes . . . 274
  Funktions-, *siehe* Funktion, -definition/-deklaration
  top-level- . . . 149
Vereinigung . . . 304
vergleichen, String . . . 248
Vergleichsoperatoren, *siehe* Operator
verketten, String . . . 248
verkettete Liste . . . 299
Verweisoperator, *siehe* Dereferenzierungsoperator
vfprintf() . . . 314
void . . . 9, 142
void * . . . 208
volatile . . . 213
vordefinierte Makronamen . . . 201
vorläufige Definition . . . 158
Vorrang der Operatoren . . . 109
vprintf() . . . 314
vsprintf() . . . 314

W

Wertübergabe, *siehe call by value*
while-Schleife . . . 133
Wiederholungsanweisungen, *siehe* Anweisung

X

%X . . . 63
%x . . . 63, 71

Z

Zeichen
  \Fluchtsymbol . . . 10, 37, 38, 40
  -klassen . . . 54, 55
  -konstante . . . 37
  Null- . . . 40
  numerischer Wert eines . . . 37, 52
  Zeilenende- . . . 10, 38
  Zwischenraum- . . . 42, 55
Zeichenkette (*siehe auch* String) . . . 10
Zeichenkonstante . . . 37
Zeichensatz . . . 19, 20, 52
Zeiger
  als Funktionsargumente . . . 206
  Arithmetik mit . . . 217
  auf Funktion . . . 282
  auf Struktur . . . 298
  auf Vektor . . . 276
  auf Zeiger . . . 229, 230
  generischer . . . 208
  indizieren . . . 221
  Null- . . . 209
  -typ, *siehe* Vereinbarung, eines Zeigers
  Umwandlung von Funktionsnamen in . . . 283
  Umwandlung von Vektoren in . . . 220
  Vergleich mit Vektor . . . 228
  Zuweisung von . . . 217, 284
\Zeilenende . . . 43
Zeilenendezeichen . . . 10, 38
Zeilenrücklauf . . . 38
Zufallszahlen erzeugen . . . 319
Zuordnung der Operatoren . . . 109, 110
zusammengesetzte Typen, *siehe* Datentyp
Zuweisungsoperatoren, *siehe* Operator
Zwischenraumzeichen . . . 42, 55
  überlesen . . . 56, 70