

1 Ein- und Ausgabefunktionen

Die Sprache C selbst enthält keine Sprachelemente zur Programm-Ein- und Ausgabe. Vielmehr müssen jegliche Programm-Ein- und Ausgabe mittels Funktionen realisiert werden. Daher sind in der zu jedem C-System gehörenden Standardbibliothek eine Reihe entsprechender Funktionen enthalten.

Durch die Normung der Standardbibliothek (ANSI-C-Bibliothek) ist bei Anwendung dieser Funktionen eine weitgehende Portabilität gewährleistet. Allerdings existieren in vielen C-Bibliotheken neben den in der ANSI-Norm festgelegten Standard-Funktionen weitere, nicht genormte Funktionen, deren Anwendung allerdings die Portabilität herabsetzt.

Jegliche Ein- und Ausgabe in C geschieht über die Datei-Schnittstelle. Geräte, wie z. B. der Drucker oder eben die Konsole werden dabei auch als Dateien behandelt. In der Standardbibliothek sind einigen Geräten zugeordnete Standard-Dateien (sog. Filestreams) definiert:

- `stdin` (Standard-Eingabe, Tastatur)
- `stdout` (Standard-Ausgabe, Bildschirm)
- `stderr` (Standard-Fehlerausgabe, Bildschirm)

Unter Bezugnahme auf die Standard-Dateien `stdin` und `stdout` kann somit eine Ein-/Ausgabe über die Konsole mit Hilfe der allgemeinen Dateibearbeitungsfunktionen realisiert werden. Es existieren darüber hinaus aber für die Ein- und Ausgabe über die Konsole spezielle relativ einfach anwendbare Funktionen:

- Funktionen zur zeichenweisen Ein- und Ausgabe (einschließlich Funktionen zur Ein- und Ausgabe von Zeichenketten)
- Funktionen zur formatierten Ein- und Ausgabe

Anmerkung: Mit Hilfe der in einigen Betriebssystemen (z. B. Windows, Linux) realisierten Umleitung der Standard-Eingabe und Standard-Ausgabe auf der Kommandozeilenebene kann mit diesen Funktionen auch eine Bearbeitung anderer Dateien realisiert werden.

Mit den Ein-/Ausgabefunktionen der Standard-Bibliothek eng verknüpft ist die Header-Datei

`stdio.h`

In ihr sind die für die Anwendung der Funktionen benötigten Funktionsdeklarationen (Function Prototypes) sowie einige Typen und Konstante (Makros), die mit der Realisierung und Anwendung der Funktionen in Zusammenhang stehen, definiert. U.a. wird die Konstante **EOF** definiert, die zur C-internen Kennzeichnung des Dateiendes dient. Der Wert dieser `int`-Konstanten ist nicht mit dem Wert eines eventuellen im Betriebssystem verwendeten Dateiende-Zeichens (z.B. `<Strg>-D` unter Linux/UNIX, `<Strg>-Z` unter Windows/DOS) identisch, sondern beträgt i. a. **-1**.

Zur problemlosen und einfachen Anwendung der Standardbibliotheks-Funktionen ist es daher zweckmäßig die Header-Datei `stdio.h` mittels

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

in das C-Programm-Modul einzubinden.

Hinweis: Damit wird keine Bibliotheksfunktion inkludiert, sondern nur das Aussehen der Funktion für das C-Modul bekannt gemacht!

Die Ausgabefunktion printf

```
int printf(controlstring, arg1, arg2, ... )
```

Das folgende Programm gleicht dem ersten C-Beispiel:

```
#include <stdio.h>

int main(void)
{
    printf("Hallo Welt\n");
    return 0;
}
```

Die Ausgabe dieses Programms ist

Hallo Welt

Dahinter kommt ein Zeilenvorschub (`\n`). Das erste Argument von `printf` ist ein Formatstring - ein String der das Ausgabeformat beschreibt. Entsprechend den C-Konventionen muss der String mit einem NUL-Zeichen (`\0`) abgeschlossen sein. Wenn der String als Konstante geschrieben wird, ist automatisch garantiert, dass er richtig abgeschlossen ist.

Die `printf`-Funktion kopiert die Zeichen aus dem Format auf die Standardausgabe, bis entweder das Ende des Strings oder ein `%`-Zeichen erreicht wird. Anstatt das im Format gefundene `%`-Zeichen auszugeben, sucht `printf` nach weiteren Zeichen hinter dem `%`-Zeichen, um herauszufinden, wie das nächste Argument umgewandelt werden soll. Das umgewandelte Argument wird anstelle des `%`-Zeichens und der nächsten paar Zeichen ausgegeben. Da das Format im obigen Beispiel kein `%`-Zeichen enthält, entspricht die Ausgabe genau den im Format angegebenen Zeichen. Das Format legt zusammen mit den entsprechenden Argumenten jedes einzelne Zeichen in der Ausgabe fest. Dazu gehört auch der Zeilenvorschub, mit dem eine Zeile abgeschlossen wird.

Der Rückgabewert von `printf` ist die Anzahl der ausgegebenen Zeichen.

Die `printf`-Funktion hat zwei verwandte Funktionen: `fprintf` und `sprintf`. Während `printf` auf die Standardausgabe schreibt, kann `fprintf` nur auf eine Ausgabedatei schreiben. Die entsprechende Datei muss in der `fprintf`-Funktion als erstes Argument angegeben werden. Daher bedeuten `printf(Ausgabe);` und `fprintf(stdout, Ausgabe);` exakt dasselbe.

Die Funktion `sprintf` wird eingesetzt, wenn die Ausgabe nicht in eine Datei erfolgen soll. Das erste Argument von `sprintf` ist die Adresse eines Zeichenarrays, in dem `sprintf` seine Ausgabe ablegt. Dass das Array groß genug ist, um die von `sprintf` erzeugte Ausgabe aufzunehmen, liegt in der Verantwortlichkeit des Programmiers. Die weiteren Argumente sind identisch mit `printf`. Die Ausgabe von `sprintf` wird immer mit einem NUL-Zeichen abgeschlossen. Die einzige Möglichkeit, um ein NUL-Zeichen auf andere Art und Weise auszugeben, ist die Verwendung des Formats `%c`.

Alle drei Funktionen liefern als Ergebnis die Anzahl der übertragenen Zeichen zurück. Im Fall von `sprintf` wird das NUL-Zeichen am Ende der Ausgabe nicht mitgezählt. Wenn `printf` oder `fprintf` während des Schreibens auf einen Ein-/Ausgabefehler treffen, geben Sie einen negativen Wert zurück. In diesem Fall kann man nicht mehr feststellen, wieviele Zeichen geschrieben wurden. Da `sprintf` keine Ein/Ausgabe durchführt, sollte niemals ein negativer Wert zurückgegeben werden.

Da der Formatstring die Datentypen der weiteren Argumente festlegt und dieser Formatstring während der Ausführung erstellt werden kann, ist es für eine C-Implementierung sehr schwer festzustellen, ob die Argumente von

`printf` die richtigen Datentypen enthalten. Wenn man also `printf("%d\n", 0.1);` schreibt oder `printf("%g\n", 'w');` dann erhält man nur Unsinn. Es ist aber äußerst unwahrscheinlich, dass dies vor dem Programmstart entdeckt werden kann. Einigen C-Compilern entgeht auch das Problem einer Anweisung wie `fprintf("error\n");` Der Programmierer hat hier `fprintf` verwendet, weil er eine Meldung auf `stderr` ausgeben wollte, hat aber vergessen, `stderr` anzugeben. Wahrscheinlich wird das Programm abstürzen, da `fprintf` den Formatstring als Dateistruktur interpretiert.

Einfache Formatangaben

Jedes Formatelement wird mit einem `%`-Zeichen eingeleitet, hinter dem wenn auch manchmal nicht sofort - ein Zeichen folgt, das als Formatcode bezeichnet wird, mit dem die Art und Weise der Umwandlung bestimmt wird. Andere Zeichen können wahlweise zwischen dem `%`-Zeichen und dem Formatcode angegeben werden. Diese Zeichen dienen zur näheren Spezifikation des Ausgabeformats und werden später noch ausführlich erläutert. Das häufigste Format ist sicherlich `%d`, das einen Integerwert in Dezimalschreibweise ausgibt. Zum Beispiel ergibt

```
printf("2 + 2 = %d\n", 2 + 2);
```

die Ausgabe `2 + 2 = 4` hinter der ein Zeilenvorschub folgt.

Das Format `%d` ist eine Anforderung, dass ein Integer ausgegeben werden soll. Es muss daher ein entsprechendes `int`-Argument vorliegen. Der Dezimalwert des Integer ersetzt das Format `%d` ohne vorangestellte oder nachfolgende Nullen während der Kopie auf die Ausgabe. Wenn der Integer negativ ist, wird als erstes Zeichen ein Minuszeichen ausgegeben.

Das Format `%u` verarbeitet einen Integer so, als wäre er `unsigned`. Deshalb ergibt `printf("%u\n", -37);` auf einer Maschine mit 32-Bit-Integern die Ausgabe `4292967259`.

Beachten Sie, dass `char`- und `short`-Argumente vor der Ausgabe mit `printf` automatisch zu einem `int` erweitert werden. Das kann auf Maschinen, bei denen `char`-Werte als `signed` verarbeitet werden, zu einigen Überraschungen führen. Um diese Probleme zu vermeiden, sollten Sie das Formatelement `%u` für `unsigned`-Werte reservieren. Auch `float`-Werte werden vor der Ausgabe mit `printf` automatisch in einen `double`-Wert umgewandelt. Das ist auch der Grund, warum es im ANSI-C89/90-Standard kein `%lf` für die Ausgabe von `double`-Werten gibt, sondern nur `%f` für die Ausgabe von `double` oder `float`-Werten. Nicht zu verwechseln mit der Eingabe von Fließkommazahlen mithilfe von `scanf`. Hier wird zwischen `double` (`%lf`) und `float` (`%f`) unterschieden!

Die Formatelemente `%o`, `%x` und `%X` geben Integerwerte mit der Basis 8 oder 16 aus. Das Element `%o` liefert oktale Ausgaben, während die Elemente `%x` und `%X` hexadezimale Ausgaben erzeugen. Der einzige Unterschied zwischen `%x` und `%X` ist:

- `%x` liefert die Buchstaben a, b, c, d, e und f für die Werte der Ziffern 10 bis 15.
- `%X` verwendet die Buchstaben A, B, C, D, E und F

Oktale und hexadezimale Werte sind immer vorzeichenlos. Zum Beispiel gibt

```
int n = 108;
printf("%d dezimal => %o oktal => %x hexadezimal\n", n, n, n);
```

die Ausgabe

```
108 dezimal => 154 oktal => 6c hexadezimal
```

Das Formatelement `%s` dient zur Ausgabe von Strings: Das entsprechende Argument muss die Startadresse eines Strings sein. Die Zeichen werden ab der Stelle, die vom Argument adressiert wird, bis zum ersten erkannten NUL-Zeichen (`'\0'`) ausgegeben. Ein String mit einem `%s`-Formatelement **muss** mit einem `'\0'`-Zeichen abgeschlossen sein. Dies ist die einzige Möglichkeit, damit `printf` das Ende des Strings erkennen kann. Wenn ein String, der an ein `%s`-Element übergeben wird, nicht richtig abgeschlossen ist, dann wird `printf` die Ausgabe solange fortsetzen, bis es irgendwo im Speicher ein `'\0'`-Zeichen findet. Die Ausgabe kann dann wirklich sehr lang und selbstverständlich auch fehlerhaft werden.

Da das Formatelement `%s` jedes Zeichen im entsprechenden Argument ausgibt, bedeuten `printf(s);` und `printf("%s", s);` nicht dasselbe. Das erste Beispiel behandelt jedes `%`-Zeichen in `s` als den Anfang eines Formatcodes. Das kann zu Problemen führen, wenn ein anderer Formatcode als `%%` vorkommt, da dann das entsprechende Argument fehlt. Das zweite Beispiel gibt jeden mit NUL abgeschlossenen String aus.

Merke: `printf("%s", s);` ist vorzuziehen!

Das Formatelement `%c` gibt ein einzelnes Zeichen aus: `printf("%c", c);` entspricht `putchar(c);`, hat aber den Vorteil, dass man den Wert von `c` auch in einem größeren Zusammenhang ausgeben kann. Das Argument, das bei einem `%c` Formatelement angegeben werden muss, ist ein `int`, der bei der Ausgabe in einen `char` umgewandelt wird. Zum Beispiel ergibt

```
printf("Der Dezimalwert von '%c' ist %d\n", '*', '*');
```

die Ausgabe `Der Dezimalwert von '*' ist 42`

Drei Formatelemente stehen für die Ausgabe von Fließkommazahlen zur Verfügung: `%g`, `%f` und `%e`. Das Formatelement `%g` ist am nützlichsten, wenn man Fließkommazahlen darstellen will, die nicht in Spalten ausgegeben werden müssen. Damit wird der entsprechende Wert (der unbedingt ein `float` oder `double` sein muss) ohne nachfolgende Nullen mit bis zu sechs signifikanten Ziffern ausgegeben. Zusammen mit `math.h` ergibt

```
printf("Pi = %g\n", 4 * atan(1.0));
```

die Ausgabe `Pi = 3.14159`. Führende Nullen werden in der Genauigkeit nicht berücksichtigt. Die Werte werden nicht abgeschnitten, sondern gerundet: `printf("%g\n", 2 / 3.0);` liefert die Ausgabe `0.666667`. Wenn die Zahl größer als 999999 ist, dann würde der Wert entweder mit mehr als sechs signifikanten Stellen oder falsch Wert ausgegeben. Das Formatelement `%g` löst dieses Problem, indem solche Werte in der wissenschaftlichen Schreibweise ausgegeben werden:

```
printf("%g\n", 123456789.0);
```

liefert die Ausgabe `1.23456e+08`. Der Wert wird wiederum auf sechs signifikante Stellen gerundet. Wenn die Größenordnung des Werts zu klein ist, wird die erforderliche Anzahl an Zeichen zur Darstellung der Werte ebenfalls sehr groß. Es ist zum Beispiel sehr unschön, wenn man $\pi \cdot 10^{-10}$ als `0.00000000031459` schreibt. Sowohl kompakter als auch leichter zu lesen ist `3.14159e-10`. Diese beiden Darstellungsformen weisen genau dieselbe Länge auf, wenn der Exponent -4 ist (zum Beispiel ist `0.000314159` genauso lang wie `3.14159e-04`). Das `%g`-Formatelement fängt daher erst bei einem Exponenten von -5 mit der wissenschaftlichen Darstellung an.

Das Formatelement `%e` verwendet zur Ausgabe von Fließkommazahlen in jedem Fall einen expliziten Exponenten: `n` wird im `%e`-Format als `3.141593e+00` ausgegeben. Das Formatelement `%e` gibt immer sechs Ziffern hinter dem Dezimalpunkt aus, und nicht bloß sechs signifikante Ziffern.

Mit dem Formatelement `%f` werden Fließkommazahlen immer ohne einen Exponenten ausgegeben, so dass `n` als `3.14159` ausgegeben wird. Auch das `%f`-Format gibt sechs Ziffern hinter dem Dezimalpunkt aus. Ein sehr kleiner Wert kann demnach als Null erscheinen, auch wenn das gar nicht der Fall ist, und eine sehr große Zahl wird mit vielen Ziffern ausgegeben:

```
printf("%f\n", 1e38);
```

wird als `1000000000000000000000000000000000000.000000` ausgegeben. Da die Anzahl der hier ausgegebenen Ziffern die Genauigkeit der meisten Computer übersteigt, kann das Ergebnis auf verschiedenen Maschinen unterschiedlich sein.

Die Formatelemente `%E` und `%G` verhalten sich genauso wie die entsprechenden Formatelemente mit Kleinbuchstaben, außer dass der Exponent mit einem großen `E` und nicht mit einem kleinen `e` dargestellt wird.

Das Formatelement `%%` gibt ein `%`-Zeichen aus. Es ist insofern einzigartig, als in diesem Fall kein entsprechendes Argument angegeben werden muss. Die Anweisung

```
printf("%%d gibt einen Dezimalwert aus\n");
```

ergibt also die Ausgabe `%d gibt einen Dezimalwert aus`.

Modifizierer

Die Funktion `printf` verarbeitet auch noch weitere Zeichen, mit denen ein Formatelement genauer spezifiziert werden kann. Diese Zeichen werden zwischen dem `%`-Zeichen und dem folgenden Formatcode angegeben.

Ganzzahlen gibt es in drei verschiedenen Längen: `char`, `short`, `long` und `int`. Wenn eine kurze Ganzzahl als Funktionsargument angegeben wird, wird er automatisch in einen `int` umgewandelt. Das gilt auch für die Funktion `printf`, aber für die `long`-Ganzzahl benötigen wir noch eine Möglichkeit, um sie zweifelsfrei anzugeben. Dies erreicht man durch ein `l` direkt vor dem Formatcode, so dass sich dann `%ld`, `%lo`, `%lx`, `%lX` und `%lu` als neue Formatcodes ergeben. Diese modifizierten Codes verhalten sich dann für `long` wie ihre nicht modifizierten Entsprechungen.

Der Modifizierer für die **Ausgabebreite** vereinfacht die Ausgabe von Werten in Feldern fester Länge. Er wird zwischen dem `%`-Zeichen und dem nachfolgenden Formatcode angegeben und legt die Mindestanzahl an Zeichen fest, die mit dem entsprechenden Formatelement ausgegeben werden sollen. Wenn der auszugebende Wert das Feld nicht voll ausfüllt, werden Leerzeichen auf der linken Seite eingefügt, damit das Feld breit genug ist. Falls der Wert zu groß für das Feld ist, wird das Feld entsprechend vergrößert. Der Modifizierer für die Ausgabebreite schneidet ein Feld niemals ab. Wenn man mit diesem Modifizierer Zahlenspalten ausgeben will, dann verschiebt ein zu großer Wert die nachfolgenden Werte in der Zeile nach rechts. Der Modifizierer für die Ausgabebreite kann bei allen Formatcodes angegeben werden, sogar bei `%%`. Beispielsweise gibt die Anweisung `printf("%8s\n");` ein rechts ausgerichtetes `%`-Zeichen in einem acht Zeichen langen Feld aus.

Der Modifizierer für die **Genauigkeit** legt die Anzahl der Ziffern in der Darstellung von Zahlen oder Strings fest. Der Modifizierer wird mit einem Dezimalpunkt angegeben, hinter dem mehrere Ziffern folgen. Er steht hinter dem `%`-Zeichen und dem Modifizierer für die Ausgabebreite, aber immer noch vor dem Formatcode:

- Bei den Ganzzahl-Formatelementen `%d`, `%o`, `%x` und `%u` gibt dieser Modifizierer die Mindestanzahl der Zeichen an, die ausgegeben werden sollen. Wenn der Wert nicht so viele Ziffern umfasst, werden Nullen vorangestellt. Die Anweisung `printf("%.2d/%.2d/%.4d\n", 7, 14, 1789);` ergibt also die

Ausgabe 07/14/1789.

Allgemein gültigere Anweisung hierfür ist aber:

```
printf("%02d/%02d/%04d\n", 7, 14, 1789);
```

Sie erzeugt die gleiche Ausgabe.

- Bei den Formatelementen %e, %E und %f gibt die Genauigkeit die Anzahl der Ziffern hinter dem Dezimalpunkt an. Wenn die Flags (die wir gleich danach besprechen) nichts anderes angeben, erscheint nur dann ein Dezimalpunkt, wenn die Genauigkeit größer als Null ist. Zum Beispiel:

```
double pi;  
pi = 4 * atan(1.0);  
printf("%.0f %.1f %.2f %.3f %.6f %.10f\n",  
        pi, pi, pi, pi, pi, pi);  
printf("%.0e %.1e %.2e %.3e %.6e %.10e\n",  
        pi, pi, pi, pi, pi, pi);
```

die folgende Ausgabe:

```
3 3.1 3.14 3.142 3.141593 3.1415926536 3e+00 3.1e+00 3.14e+00  
3.1415926536e+00
```

- Bei den Formatelementen %g und %G gibt die Genauigkeit die Anzahl der signifikanten Ziffern an, die ausgegeben werden sollen. Wenn in den Flags nichts anderes angegeben wurde, werden alle bedeutungslosen Nullen entfernt und der Dezimalpunkt gelöscht, falls dahinter keine Ziffern mehr folgen.
- Im Falle der %s-Formatelemente gibt die Genauigkeit an, wieviele Zeichen des entsprechenden Strings ausgegeben werden sollen. Wenn im String nicht genug Zeichen vorhanden sind, um die geforderte Genauigkeit zu erfüllen, wird die Ausgabe gekürzt. Mit Hilfe des Modifizierers für die Ausgabebreite kann die Ausgabe wieder verlängert werden. Mit Hilfe des Formatelements %15.15s wäre garantiert, dass genau 15 Zeichen ausgegeben werden.
- In den Formatelementen c und % wird die Genauigkeit nicht berücksichtigt.

Flags

Zwischen dem %-Zeichen und der Feldbreite können noch weitere Zeichen angegeben werden, mit denen man die Wirkung eines Formatelements weiter beeinflussen kann. Diese Zeichen werden als Flags bezeichnet. Die Flagzeichen haben folgende Bedeutungen:

- Das Flag - wirkt sich nur dann aus, falls eine Breite angegeben wurde (da man nur dann etwas zum Ausfüllen hat, wenn die Breite größer ist, als es für den Wert notwendig ist). In diesem Fall erscheinen alle Leerzeichen zum Auffüllen auf der rechten, und nicht auf der linken Seite.
Beim Ausdruck von Strings in Spalten fester Länge sieht es normalerweise besser aus, wenn man sie links ausrichtet. Daher ist ein Format wie %14s meistens ein Fehler und sollte wohl %-14s heißen.
- Das Flag + gibt an, dass jeder numerische Wert mit einem Vorzeichen als erstes Zeichen ausgegeben werden soll. Es werden also auch alle nicht negativen Werte mit einem Pluszeichen ausgegeben. Das Flag hat keinerlei Bezug zum Flag -.
- Wenn ein Leerzeichen als Flag verwendet wird, dann bedeutet das, dass ein einzelnes Leerzeichen vor jedem numerischen Wert erscheinen soll, wenn kein Vorzeichen vorhanden ist. Dies wird meist für links ausgerichtete Spalten von Zahlen verwendet, in denen kein Pluszeichen vorkommen soll. Wenn die Flags + und Leerzeichen im selben Formatelement angegeben werden, hat das Flag + den Vorrang. Die Formatelemente % e und %+e sind für die Ausgabe von Zahlenspalten in wissenschaftlicher Schreibweise besser geeignet als das normale Formatelement %e: Durch das Vorzeichen (oder das Leerzeichen) wird sichergestellt, dass alle Dezimalpunkte untereinanderstehen.

- Das Flag # verändert das Format der numerischen Werte geringfügig abhängig von den einzelnen Formatelementen. Beim %o-Format wird die Genauigkeit erhöht, und zwar so weit, dass die erste ausgegebene Ziffer eine 0 ist. Damit können Oktalwerte so ausgegeben werden, wie sie von C-Programmierern am leichtesten erkannt werden. In gleicher Weise stehen bei den Formatelementen %#x und %#X die Zeichenfolgen 0x bzw. 0X vor der Ausgabe.
Das Flag # erzwingt die Ausgabe eines Dezimalpunkts, auch wenn dahinter keine weiteren Ziffern mehr kommen, und es verhindert das Weglassen nachfolgender Nullen in dem Formaten %g und %G.

Die Flags sind außer dem Leer- und dem Pluszeichen alle voneinander unabhängig.

Variable Feldbreite und Genauigkeit

Viele C-Programme definieren sorgfältig die Länge eines Strings als feste Konstante, damit sie leichter zu ändern ist geben aber die Ausgabebreite in den Ausgabeanweisungen als Integerkonstante an. Es wäre also nicht allzu klug, wenn wir eines unserer früheren Beispiele wie folgt umschreiben würden:

```
#define NAMESIZE 14
char name[NAMESIZE+1];
...
printf(" ...%.14s ... ", ... , name, ...);
...
```

Jemand der später einmal NAMESIZE verändern will, wird wahrscheinlich übersehen, dass er alle printf-Aufrufe ebenfalls verändern muss. Es ist jedoch nicht möglich, NAMESIZE direkt in der printf-Anweisung anzugeben:

```
printf("... %.NAMESIZES, ...", name, ...);
```

funktioniert nicht, da der Präprozessor innerhalb von Strings keine Ersetzungen vornimmt.

printf erlaubt daher, dass die Feldbreite oder Genauigkeit indirekt angegeben werden kann. Dazu schreibt man anstelle der Feldbreite oder der Genauigkeit das Zeichen *. In diesem Fall holt sich printf die tatsächlichen Werte aus der Argumentliste, bevor der Wert ausgegeben wird. Im obigen Beispiel sollte es also

```
printf("... %.*s, ...", ..., NAMESIZE, name, ...);
```

heißen. Wenn die Konvention mit dem * sowohl für die Feldbreite als auch die Genauigkeit verwendet wird, erscheint das Argument mit der Feldbreite zuerst, dahinter kommt das Argument für die Genauigkeit und anschließend der Wert, der ausgegeben werden soll.

```
printf("%*.s\n", 12, 5, str);
```

hat also dieselbe Wirkung wie

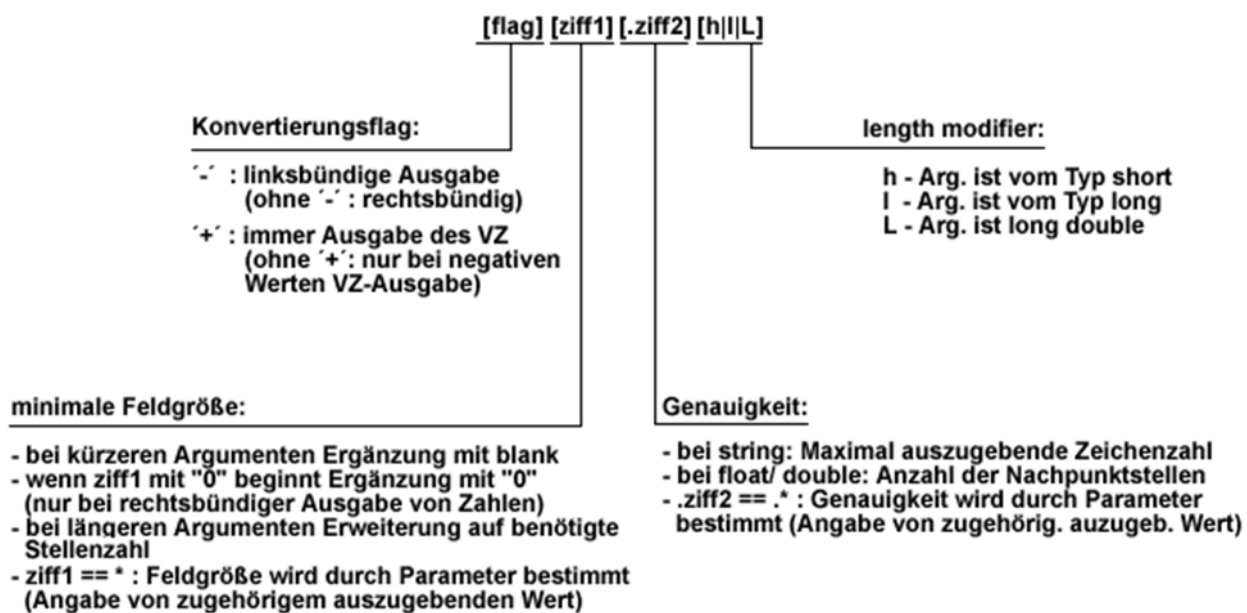
```
printf("%12.5s\n", str);
```

Wenn das Zeichen * für die Feldbreite verwendet wird und der entsprechende Wert negativ ist, hat das denselben Effekt, als ob das Flag - ebenfalls angegeben worden wäre.

Zusammenfassung: Formatierte Ausgabe in C mit "printf"

Allgemeine Form:	<code>printf(Controlstring, Arg1, Arg2, ...)</code>
Controlstring:	Ausgabe von Text und Steuerung der Ausgabeformate. Er kann enthalten: <ul style="list-style-type: none">• darstellbare Zeichen• Zeichenersatzdarstellungen ('\n', '\t' usw.)• Umwandlungsspezifikationen

Arg1, Arg2,:	Die auszugebenden Werte (Argumente). Anzahl und Typ sind durch den "controlstring" festgelegt
Umwandlungsspezifikation:	<p>Sie haben die Form: % [Formatangabe] Konvertierungszeichen</p> <p><i>Konvertierungszeichen:</i></p> <p>c Einzelzeichen d, i Integerzahl (dezimal, konegativ) u Integerzahl (dezimal, nur positiv) o Integerzahl (oktal, nur positiv, ohne führende "0") x, X Integerzahl (sedezimal, nur positiv, ohne führende "0x") e, E Gleitpunktzahl (float oder double) in Exponentendarstellung f Gleitpunktzahl (float oder double) in Dezimalbruchdarstellung g, G kürzeste Darstellung von e oder f s String p Pointer (implementierungsabhängige Darstellung)</p> <p><i>Formatangabe:</i></p>



Die Eingabefunktion scanf

```
scanf(controlstring, arg1, arg2, ... )
```

"scanf" liest die naechsten Zeichen von stdin, interpretiert sie entsprechend den im Steuerstring "controlstring" vorliegenden Typ- und Formatangaben und weist die dem gemäß konvertierten Werte den durch ihre Adresse referierten Variablen arg1, arg2, ... zu. Anzahl und Typ der Variablen sind durch "controlstring" festgelegt.

Die der Zeichen-Werte-Konvertierung zugrundeliegenden Umwandlungsspezifikationen werden durch White-Space-Character (Blank, Tab, Newline) bzw. durch Längenangaben im Steuerstring getrennt.

Funktionswert: Die Anzahl der erfolgreich zugewiesenen Werte bzw. EOF (= -1), wenn beim Lesen des ersten Wertes versucht wurde, über die Eingabe des Dateiendezeichens hinaus zu lesen.

Die Umwandlungsspezifikation haben ein einheitliches Format:

```
%[*][ziff]Konvertierungszeichen
|      |
|      | maximale Eingabefeldgröße (Kann weggelassen werden.
|      | Bei Konvertierungszeichen c: Anzahl der einzulesenden
|      | Zeichen.)
|
| Zeichen zum Überlesen des nächsten Eingabefeldes
| (assignment suppression; kann weggeleassen werden.)
```

Konvertierungszeichen:

c Zeichen(-folge) (auch blanks, tabs und newlines werden gelesen)
Default (keine Feldgrößenangabe): 1 Zeichen

d Integer (dezimal)

i Integer (dezimal) oder oktal (mit führender "0") oder sedezimal (mit führendem "0x" bzw "0X")

o Integer (oktal, mit oder ohne führende "0")

x Integer (sedezimal, mit oder ohne führendem "0x" bzw "0X")

u Integer (dezimal, nur positiv)

e, f, g Gleitpunktzahl (beliebige Darstellung)

s String (Ergänzung mit abschließendem '\0')

p Pointer

h vor d,i,o,u,x Kurze Integerzahl (short)

l vor d,i,o,u,x Lange Integerzahl (long)

l vor e,f,g double

L vor e,f,g long double

Beispiel:

```
int    i, jwert;
float  zahl;
...
scanf ("%2d %f %*d %d", &i, &zahl, &jwert)
...
```

Nach Eingabe von: 56789 0123 457 haben die Variablen folgende Werte: i = 56, zahl = 789.0, jwert = 457. Die Zahl 0123 wird überlesen!

Die Eingabefunktion gets

```
char *gets(char *s);
```

"gets" liest die nächsten Zeichen von stdin bis zum nächsten Newline-Character ('\\n') bzw bis das Dateiende erreicht ist. Die gelesenen Zeichen werden in dem durch "s" referierten String ohne eventuell gelesenes '\\n' abgelegt. An s wird automatisch das Stringende-Zeichen '\\0' angehängt. Funktionswert ist die Anfangsadresse des Strings "s" oder ein NULL-Pointer bei Erreichen des Dateiendes ohne vorheriges Lesen von Zeichen. Im Fehlerfall wird ebenfalls ein NULL-Pointer zurückgegeben. Es wird nicht überprüft, ob genügend Speicherplatz für den String zur Verfügung steht.

Die Ausgabefunktion puts

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

"puts" gibt den durch "s" referierten String (ohne '\\0'-Character!) nach stdout aus, wobei ein abschließendes '\\n' angefügt wird. Funktionswert ist ein nicht-negativer Wert bei Fehlerfreiheit und EOF (-1) im Fehlerfall.

Beispiel:

```
/* String einlesen und wieder ausgeben */
#include <stdio.h>

int main(void)
{
    char str[100];
    if (gets(str) != NULL)
        puts(str);
    else
        printf("Leereingabe\\n");
    return 0;
}
```

Die Eingabefunktion getchar

```
int getchar(void);
```

"getchar" liest das nächste Zeichen von stdin und gibt das gelesene Zeichen (als int-Wert zurück. Bei Eingabe des Fileendezeichens für Textdateien oder im Fehlerfall wird EOF (-1) zurückgegeben.

Beispiel:

```
/* Zeichen kopieren von stdin nach stdout */
#include <stdio.h>

int main(void)
{
    int ch;
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

```
    return 0;
}
```

Die Ausgabefunktion putchar

```
int putchar(int c);
```

"putchar" gibt das Zeichen "c" (nach Umwandlung in unsigned char) nach stdout aus. Funktionswert: das ausgegebene Zeichen (als int-Wert) oder EOF (-1) im Fehlerfall.

Das folgende Programm demonstriert eine kreative Form der Ausgabe.

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

void xmastree(int order);

int main(int argc, char *argv[])
{
    if (argc > 1)
        xmastree(atoi(argv[1]));
    else
        puts("usage: xmas order");
    return 0;
}

/*
 * Diese Funktion druckt den Weihnachtsbaum
 */
void xmastree(int order)
{
    int top, height, width, ornament,
        leading_spaces, middle_spaces;

    top = order + 1; /* 'order + 1' wird öfter gebraucht */
    /* Die führenden Leerzeichen zentrieren den Baum in der Mitte:
     * order * (order + 1) / 2 */
    leading_spaces = order * top / 2 + 1;
    middle_spaces = 0;



    /* Baum ausgeben, Schleife für die Hoehe */
    for(height = 0; height < top; ++height)
    {
        /* Die 'Aeste' */
        for(width = 0; width <= height; ++width)
        {
            /* Verzierung an jedem Ast bis auf den ersten */
            ornament = height && !width;
            if(width) middle_spaces += 2;
        }
    }
}
```

```
/* Trick: Statt Schleife lassen wir printf die Leerzeichen
   drucken
*/
printf("%*s", leading_spaces, "");
if (ornament)
    putchar('*');
else
    --leading_spaces;
/* Trick wie oben '/' etliche leerzeichen '\ ' */
printf("/%*s\\", middle_spaces, "");
if (ornament)
    putchar('*');
putchar('\n');
}
}
}
```

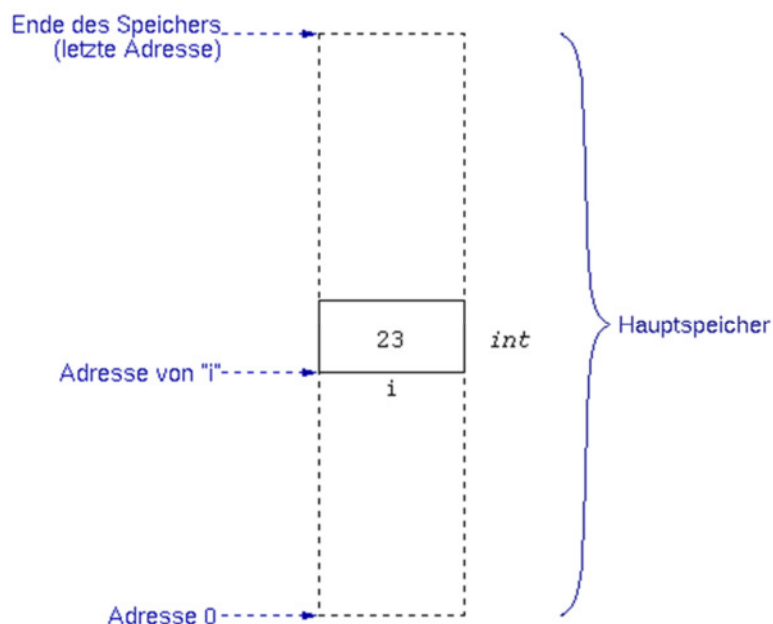
2.1 Grundlagen Zeiger

- Zeigerkonstanten
- Zeigervariablen

Zeigervariablen (oftmals auch nur kurz: Zeiger) benötigen wie alle anderen Variablen Speicherplätze. Eine Zeigervariable ist eine Variable, die die Startadresse eines (beliebigen) anderen Objektes enthält. Man kann auf dieses Objekt indirekt über einen Zeiger zugreifen. Diese Speicheradressen sind entweder Startadressen von **anderen Variablen** oder **Adressen von Funktionen** (d. h. die Adresse der ersten Maschinenanweisung der Funktion).

```
zeiger_auf_x:  → x: 
```

- bei der Parameterübergabe
- bei der Bearbeitung von Arrays
- für Textverarbeitung
- Zugriff auf spezielle Speicherzellen



Seite 13

Deklaration einer Zeigervariablen, die auf eine Variable des angegebenen Typs zeigt:

```
int *pc;
```

Eigentlich sollte man ja `int* pc` schreiben, denn gemeint ist ein *Pointer auf int*, der den Name *pc* hat. Aber seit Anbeginn der Sprache C wird das Sternchen direkt vor den Variablennamen gesetzt (dem Compiler ist es übrigens egal).

Der Grund ist dann zu erkennen, wenn mehrere Variablen in einer Definition angegeben werden:

```
int* pc1, pc2;
```

Hier suggeriert die Angabe, dass zwei Zeigervariablen definiert wurden, jedoch handelt es sich bei der Variablen `pc2` um eine `int`-Variable und nicht um eine Zeigervariable.

Um zwei Zeigervariablen auf `int`-Variablen zu definieren, muss man folgendes schreiben:

```
int *pc1, *pc2;
```

Zeiger zeigen immer auf Objekte eines bestimmten Typs. Sie müssen daher deklariert werden. Zur Zeigerdefinition wird der Asterisk (*) verwendet.

```
int *px;           /* px ist Zeiger auf int */
char *zs;          /* zs ist Zeiger auf char */
int x=3, y;
px = &x;           /* px = (Speicher-)Adresse der Variablen x */
y = *px;           /* y = Wert der Variablen x */
```

Zeiger sind also an bestimmte Objekttypen gebunden. Ein Zeiger auf `int` kann also beispielsweise nur Adressen von `int`-Variablen aufnehmen. Eine Ausnahme bildet der "Generic Pointer", der auf ein beliebiges Objekt zeigen kann.

```
void *pc;
```

Zusätzlich bieten Zeiger die Möglichkeit Objekte als **nur lesbar** zu betrachten.

```
const int *pc;
```

Solche Deklarationen (bzw. Definitionen) sind am besten von rechts nach links zu lesen:

Die Variable `pc` ist ein Zeiger (anzeigt durch `*`) auf ein `int`-Objekt, das konstant (`const`) ist.

```
int const *pc;
```

Auch hier kann man es wie folgt lesen:

Die Variable `pc` ist ein Zeiger (anzeigt durch `*`) auf ein konstantes (`const`) `int`-Objekt.

Beide Schreibweisen bedeuten also das gleiche!

```
const int *px;      /* px ist Zeiger auf int, der konstant (nur
                    lesbar) ist */

int x=5, y;
px = &x;             /* px = (Speicher-)Adresse der Variablen x */
y = *px;            /* y = Wert der Variablen x */
```



```
x = 3;
/*      *px = 3;      => Es ist nicht mehr erlaubt x ueber diesen
                        Weg zu setzen */
```

Dagegen bedeutet

```
int * const pc;
```

pc ist ein konstanter (const) Zeiger, der auf ein int-Objekt zeigt. In diesem Fall ist der Wert des int-Objekts veränderbar. Der Wert von pc darf aber nur initialisiert werden, nicht aber danach geändert werden.

```
int x=5, y;
int *const px=&x;      /* px ist ein Zeiger auf die int-Variable x
                        Der Wert von px kann nicht mehr geändert
                        werden. */

*px = 3;               /* Die Variable x erhält den Wert 3 */

/*      px = &y;      => Es ist nicht mehr erlaubt px auf y
                        zeigen zu lassen. */
```

In Verbindung mit Zeigern werden hauptsächlich zwei zueinander inverse Operatoren benutzt:

1. Der Adressoperator &, der angewendet auf ein Objekt, die Adresse dieses Objekts liefert.

```
pc = &c;
```

Der Adressoperator liefert immer eine Zeigerkonstante.

2. & kann auf Variablen und Arrayelemente angewendet werden, nicht aber auf Arraynamen selbst (Warum? Ein Arrayname hat keine Adresse, er ist eine Zeigerkonstante!). Ebenso haben natürlich Variablen in der Speicherklasse register keine Adressen. Wird der Adressoperator auf ein Register angewendet, so wird die Speicherklasse register ignoriert.
3. Beispiele für die Anwendung des Adressoperators und das Speichern der Adresse in einem Zeiger:

```
px = &x;      /* px erhält als Wert die Adresse von x */
pf = &f[5];    /* pf erhält als Wert die Adresse des
                6. Elementes von f */
```

4. Der Inhaltsoperator *, der angewendet auf einen Zeiger das Objekt liefert, das unter dieser Adresse abgelegt ist.

```
c = *pc;
*pc = 5;
```

Achtung: Hier wird der Asterisk (*) nicht in einer Deklaration verwendet, d. h. der Asterisk stellt jetzt den Inhaltsoperator dar!

Beispiel:

```
y = *px;          /* y erhält den Wert des Objektes,
                  dessen Adresse in px steht */
px = &x;          /* px "zeigt" nun auf x */
y = *px;          /* y = x; */
```

Die folgenden Programmbeispiele zeigen den Gebrauch dieser beiden Operatoren:

```
#include <stdio.h>

int main(void)
{
    int x = 1, y = 2, z[10];
    int *ip;          /* ip ist ein Zeiger auf int */

    ip = &x;          /* ip zeigt nun auf x */
    printf("ip: %d\n", ip);
    y = *ip;          /* y ist nun gleich 1 */
    printf(" y: %d\n", y);
    *ip = 0;          /* x ist nun gleich 0 */
    printf(" x: %d\n", x);
    ip = &z[0];        /* ip zeigt nun auf z[0] */
    printf("ip: %d\n", ip);

    return 0;
}
```

```
#include <stdio.h>

int main(void)
{
    int zahl;
    int *zeiger;

    /* Lass die Zeigervariable auf die Variable zeigen */
    zeiger=&zahl;

    /* Setze den Wert der Variablen mit Hilfe der Zeiger-
       variablen */
    *zeiger = 5;

    /* Gib zur Kontrolle den Wert aus */
    printf("Der folgende Wert sollte 5 sein: %d\n", zahl);

    return 0;
}
```

Die Deklaration des Zeigers `ip` in `int *ip;` besagt, dass der Ausdruck `*ip` vom Datentyp `int` ist, bzw. `ip` auf den Datentyp `int` zeigt. In der Zuweisung `ip = &x;` wird der Zeigervariablen `ip` die Adresse von `x` zugewiesen;

man sagt auch "ip zeigt auf x". Damit hat *ip denselben Wert wie x, nämlich 1, der in der Zuweisung `y = *ip;` der Variablen `y` zugewiesen wird und somit den ursprünglichen Wert 2 überschreibt. Durch die Zuweisung `*ip = 0;` erhält auch `x` den Wert 0, wie man mit `printf` bestätigen kann. Durch die Zuweisung `ip = &z[0];` zeigt der Zeiger `ip` auf das Anfangselement des Feldes `z`.

Einige Grundregeln:

Die Kombination ***Zeiger** kann in Ausdrücken überall dort auftreten, wo auch das Objekt, auf das der Zeiger zeigt, selbst stehen könnte:

```
y = *px + 10;
y = *px + *px;
printf("%d\n", *px);
*px = 0;
py = px;      /* falls py auch Zeiger auf int */
```

Bei der Verwendung des Operators `*` muss man die Operatorrangfolge und -assoziativität genau beachten. Dies erscheint zunächst etwas schwierig, da dieser Operator ungewohnt ist. Hier einige Beispiele mit dem `*` Operator und anderen Operatoren:

```
y = *px + 1;      /* Inhalt von px plus 1 */
y = *(px+1);      /* Inhalt der Adresse px+1 */
*px += 1;          /* Inhalt von px = Inhalt von px plus 1 */
(*px)++;          /* Inhalt von px inkrementieren */
*px++;            /* wie *(px++); (Assoziativität)
                  Inhalt der Adresse px; px = px plus 1 */
*++px;            /* Inhalt der Adresse px+1; px = px plus 1 */
```

Besonders wichtig:

1. `*` und `&` haben höhere Priorität als arithmetische Operatoren.
2. Werden `*` und `++` direkt hintereinander verwendet, wird der Ausdruck von rechts nach links abgearbeitet.

Ein weiteres Beispiel, das mit Pointern spielt:

```
#include <stdio.h>

int main()
{
    int *pt1, *pt2;
    int var1 = 10;
    int var2 = 20;

    pt1 = &var1;      /* pt1 zeigt nun auf var1 */
    pt2 = pt1;         /* pt2 zeigt nun auch auf var1 */
    /*                */
    *pt1 = *pt1 + 1;    /* --> var1 = var1 + 1; */
    /*                */
    pt1 = &var2;        /* pt1 zeigt nun auf var2 */
    (*pt1)++;          /* --> var2 = var2 + 1; */
    *pt2 = 15;         /* --> var1 = 15; */
    /*                */
    pt1 = &var1;        /* pt1 zeigt nun auf var1 */
    /*                */
}
```

ursprünglich [Programmieren in C](#)
von Prof. Jürgen Plate

```
pt2 = &var2;      /* pt2 zeigt nun auf var2      */
*pt2 = *pt1;      /* --> var2 = var1;      */
*pt2 = 30;        /* --> var2 = 30;      */
/*              */
pt2 = pt1;        /* pt2 zeigt nun auch auf var1 */
*pt2 = 30;        /* --> var1 = 30;      */

return 0;
}
```

Zeiger haben nur dann sinnvolle Werte, wenn sie die Adresse eines Objektes oder NULL enthalten. NULL ist eine globale symbolische Konstante, die in der Standardbibliothek definiert ist und überall als NULL-Zeiger benutzt werden kann. Für den Zeigerwert NULL ist garantiert, dass er auf eine ungültige Adresse hinzeigt. NULL ist definiert als die Adresse der Speicherstelle 0; Diese Speicherstelle darf weder gelesen noch beschrieben werden. Sie sollten auch immer NULL (anstelle des Wertes 0) verwenden, denn es macht das Programm dadurch lesbarer. Die symbolische Konstante ist wie folgt definiert:

```
#define NULL ((void *) 0)
```

2.2 Zeiger und Funktionsargumente

Argumente vom Zeigertyp eröffnen die Möglichkeit, aus einer Funktion heraus Objekte in der rufenden Funktion anzusprechen und zu verändern. Die Funktion `swap()` hat dies gezeigt.

Beispiel:

```
/* Funktion zum Vertauschen ihrer Argumente */
/* 1. Versuch, FALSCH, da Werte nur innerhalb */
/* von swap getauscht werden */
void swap(int x,int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main(void)
{
    int a = 1, b = 2;
    swap(a,b);
    printf("a=%d, b=%d\n",a,b); /* Ueberraschung! */
                                /* Ausgabe: a=1, b=2 */
    return 0;
}

/* 2. Versuch, RICHTIG, da Objekte, auf die die */
/* Adressen zeigen, ausgetauscht werden */
void swap(int *px, int *py)
{
    int temp;
```

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

int main(void)
{
    int a = 1, b = 2;
    swap(&a, &b);          /* Zeigerkonstanten uebergeben */
    printf("a=%d, b=%d\n", a, b); /* Diesmal klappt es: */
                                /* Ausgabe: a=2, b=1 */
    return 0;
}
```

Im anschließenden Beispiel zur Ermittlung des betragsgrößten Elementes einer $n \times n$ -Matrix mit Hilfe der Funktion `maxval` kann das Ergebnis auch ohne `return` Anweisung übergeben werden. Im Beispiel geschieht das über die Zeigervariable `pamax`. Beim Aufruf der Funktion `maxval` wird `&amax`, die Adresse von `amax`, als Argument übergeben. Die Zeigervariable `pamax` von `maxval` zeigt dann auf `amax`, d. h. der Parameter `*pamax` hat denselben Wert wie `amax`.

Die Berechnung in der Funktion `maxval` erfolgt mit `*pamax`. Der von der Funktion `maxval` ermittelte Wert, auf den `*pamax` zeigt, kann in der aufrufenden Funktion `main` als Variable `amax` verwendet werden.

```
/* Betragsgroesstes Element einer n*n Matrix */
/* Rueckgabe mittels einer Zeigervariablen */
#include <stdio.h>
#include <math.h>

#define NMAX 5

int maxval(double a[NMAX][NMAX], int n, double *pamax);

int main(void)
{
    double a[NMAX][NMAX] = { { 1.3, 2.6, -8.7, 23.4, 12.0 },
                              { 777.0, 5.0, 3.1, -45.0, 0.1 },
                              { 0.01, 33.7, 11.4, -1.0, 99.9 },
                              { 1.0, -2.0, 3.0, -4.0, 1.1 },
                              { 567.9, -222.0, 2.2, 3.14, -12.4 } };

    double amax;
    int i, j, n;

    n = NMAX;
    printf("Eingabedaten:\n");
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            printf("%f ", a[i][j]);
        printf("\n");
    }
    maxval(a, n, &amax);
    printf("betragsgroesstes Element= %f\n", amax);
}
```

```
    return 0;
}

void maxval(double a[NMAX][NMAX], int n, double *pamax)
{
    int i, j;
    *pamax = 0.0;
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            if(fabs(a[i][j]) > *pamax)
                *pamax = fabs(a[i][j]);
        }
    }
}
```

2.3 Zeigerarithmetik

Es können mit Zeigern bestimmte arithmetische Operationen und Vergleiche durchgeführt werden. Es sind natürlich nur solche Operationen erlaubt, die zu sinnvollen Ergebnissen führen. Zu Zeigern dürfen ganzzahlige Werte addiert und es dürfen ganzzahlige Werte subtrahiert werden. Zeiger dürfen in- und dekrementiert werden und sie dürfen voneinander subtrahiert werden

Dies ist i. a. nur sinnvoll, wenn beide Zeiger auf Elemente des gleichen Objektes (z. B. ein Array) zeigen.

Man kann Zeiger mittels der Operatoren `>`, `>=`, `<`, `<=`, `!=` und `==` miteinander vergleichen. Wie bei der Zeigersubtraktion ist das aber i. a. nur dann sinnvoll, wenn beide Zeiger auf Elemente des gleichen Arrays zeigen. Eine Ausnahme bildet hier der Zeigerwert `NULL`, denn viele Bibliotheksfunktionen liefern im Fehlerfall einen `NULL`-Zeiger zurück.

Alle anderen denkbaren arithmetischen und logischen Operationen (Addition von Zeigern, Multiplikation, Division, Shifts oder Verwendung von logischen Operatoren, sowie Addition und Subtraktion von **float** oder **double** Werten) sind mit Zeigern **nicht** erlaubt.

Es ist sinnvoll, Zeigervariablen mit `NULL` zu initialisieren, d. h. Zeigern, die keinen definierten Wert enthalten, wird der Wert `NULL` zugewiesen.

Wie funktioniert nun aber die Zeigerarithmetik? Sei `ptr` ein Zeiger und `N` eine ganze Zahl, dann bezeichnet `ptr + N` das `n`-te Objekt im Anschluss an das Objekt, auf das `ptr` gerade zeigt. Es wird also nicht der Wert `N` zu `ptr` direkt addiert, sondern `N` wird vorher mit der Typlänge des Typs, auf den `ptr` zeigt, multipliziert. Dieser Typ wird aus der Deklaration von `ptr` bestimmt.

Beispiel-Unterprogramm: Länge der Zeichenkette `s`

```
int strlen(const char *s)
{
    const char *p = s; /* Pointer zeigt auf das erste Element */
    while (*p != '\0') /* Solange der String nicht zuende ist */
        p++;          /* Pointer incrementieren */
    return p-s;        /* Laenge = aktueller Wert - Anfangswert */
}
```



```
}
```

Beispiel: 2 Versionen von strcmp: Vergleich zweier Strings s und t. Die Funktion liefert als Ergebnis: Ergebnis < 0 wenn s kleiner t ist,

Ergebnis = 0 wenn beide Strings identisch sind, Ergebnis > 0 wenn s > t ist

```
/* Version mit Arrays */
int strcmp(const char s[], const char t[])
{
    int i, ok=1;
    i = 0;
    while (ok && s[i] == t[i])
        if (s[i] == '\0')
            ok=0;
        else
            i++;
    return s[i] - t[i];
}
```

```
/* Version mit Pointern */
int strcmp(const char *s, const char *t)
{
    int diff=0;
    for ( ; *s == *t; s++,t++)
        if (*s == '\0')
            diff=0;
        else
            diff=*s - *t;
    return diff;
}
```

Folgende Operationen sind möglich, sollten jedoch mit Vorsicht verwendet werden.

pc = (int*)10 Zuweisen der Adresse 10 an die Zeigervariable pc
x = *(int*)10 Zuweisen des Inhalts der Adresse 10 an die Variable x
Mit diesen Operationen kann man auf beliebige Speicherplätze zugreifen. Nützlich sind solche Befehle bei systemnaher Programmierung, z. B. bei der systemnahen Programmierung von Mikrocontollern.

2.4 Zeiger und Array

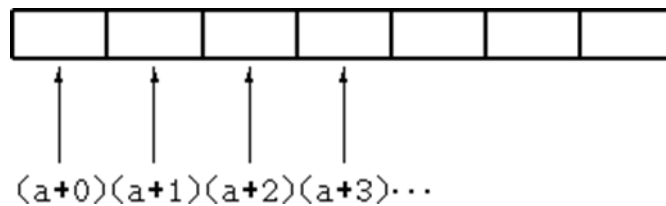
In C besteht zwischen Zeigern und Feldern eine ausgeprägte Korrespondenz. Jede Operation, die durch die Indizierung von Feldelementen formuliert werden kann, kann auch mit Zeigern ausgedrückt werden. Die Zeigerversion wird im Allgemeinen schneller sein; sie ist aber am Anfang schwerer zu verstehen.

a ist ein Integer-Array und pa ein Zeiger auf den Datentyp int, deklariert durch

```
int a[10];          /* Array mit 10 Elementen */
int *pa;            /* Zeiger auf int */
```

Zur Erinnerung:

a hat 10 Elemente, wobei a[0] das erste und a[9] das letzte Element ist. a[i], das i-te Element ist genau i Positionen vom Anfang a[0] entfernt.



Die Zuweisung

```
pa = &a[0];
```

bewirkt, dass pa auf das 0-te Element von a zeigt, d. h. dass pa die Adresse von a[0] enthält. Statt der obigen Zuweisung kann auch die folgende

```
pa = a;
```

verwendet werden, wo auf der rechten Seite der Feldname a steht. Die einzelnen Elemente des Feldes a können auf drei verschiedene Weisen angesprochen werden:

```
a[i]      i-tes Feldelement
*(pa+i)   Pointer pa + i * (Laenge eines Elementes von a)
*(a+i)    Arrayanfang + i * (Laenge eines Elementes von a)
```

a ist also in C äquivalent zu &a[0]. So kann man statt pa = &a[0] auch schreiben pa = a. Die Verwandtschaft zwischen Arrays und Pointern kann beim Programmieren recht praktisch und effizient sein, führt aber auch zu unsauberem Stil und unverständlichen Programmen.

Das folgende Beispiel soll die Äquivalenz von Arrays und Pointern etwas verdeutlichen:

```
#include <stdio.h>

int main()
{
    float messwerte[12]; /* Array mit 12 float-Komponenten */
    float *zeiger;        /* Zeiger auf ein float */

    /* 2 äquivalente Zuweisungen an die erste Arraykomponente */
    messwerte[0] = 1.23;
    *messwerte   = 1.23;

    /* 2 äquivalente Zuweisungen an die dritte Arraykomponente */
    messwerte[2]   = 4.56;
    *(messwerte + 2) = 4.56;

    /* 2 äquivalente Zuweisungen: Der Zeiger zeigt auf die
       erste Arraykomponente */
    zeiger = messwerte;
    zeiger = &messwerte[0];

    /* 2 äquivalente Zuweisungen: Der Zeiger zeigt auf die
       dritte Arraykomponente */
    zeiger = &messwerte[2];
    zeiger = messwerte + 2;

    messwerte = zeiger; /* Unzulaessig, weil 'messwerte'
                        eine Zeigerkonstante ist */
}
```

```
    return 0;
}
```

Noch ein Beispiel für die Äquivalenz von Arrays und Pointern:

```
void arraytest(char x[])
{    /* Ersetzt im Array x alle 'e' durch ".",
      Array wird als Parameter uebergeben */
    int z;

    for (z = 0; x[z]!='\0'; z++)
        if (x[z] == 'e') x[z] = '.';
}
```

```
void pointertest(char *px)
{    /* Ersetzt im Array x alle 'e' durch ".",
      ein Pointer wird als Parameter uebergeben */
    int z;

    for (z = 0; *(px+z]!='\0'; z++)
        if (*(px+z) == 'e') *(px+z) = '.';
}
```

Beide Unterprogramme können mit einem Array als Parameter aufgerufen werden und liefern das gleiche Ergebnis.

Im folgenden Programmbeispiel werden die drei Möglichkeiten in der Anweisung `printf` einander gegenübergestellt:

```
/* Zeiger und Felder */
#include <stdio.h>
int main(void)
{
    int i, a[10] = {12, 23, 34, 35, 36, 37, 44, 46, 48, 50} ;
    int *pa;

    pa = a; /* oder: pa = &a[0]; */
    for (i=0; i<10; i++)
        printf("a[%d]= %d\t *(pa+%d)= %d\t *(a+%d)= %d\n",
               i, a[i], i, *(pa+i), i, *(a+i));
    return 0;
}
```

Die Korrespondenz von Indizieren und Zeigerarithmetik ist daher sehr eng: Die Schreibweise `pa + i` bedeutet die Adresse des *i*-ten Objekts hinter demjenigen, auf welches `pa` zeigt. Es gilt:

Jeder Array-Index-Ausdruck kann auch als Pointer-Offset-Ausdruck formuliert werden und umgekehrt.

Der Ausdruck `pa + i` bedeutet **nicht**, dass zu `pa` der Wert *i* addiert wird, sondern *i* legt fest, wie oft die Länge des Objekt-Typs von `pa` addiert werden muss.

Es besteht jedoch ein gravierender Unterschied zwischen Array-Namen (Zeigerkonstanten) und Zeigervariablen:

- Ein Zeiger ist eine Variable, deren Wert jederzeit geändert werden darf, z. B.:

```
int x, *px;  
px = &x;      /* Veränderung ist jederzeit zulässig */
```

- Ein Array-Name ist eine Zeigerkonstante, ihr Wert ist nicht veränderbar! Z. B.:

```
int z, y[5];  
y = &z;      /* U n z u l ä s s i g */
```

Beispiel zur Verwandtschaft zwischen Zeigern und Arrays: Äquivalente Formulierungen der Funktion `strcpy()`

```
void strcpy(char str1[], const char str2[])  
{  
    int i=0;  
    while((str1[i]=str2[i]) != '\0')  
        i++;  
}
```

```
void strcpy(char *str1, const char *str2)  
{  
    int i=0;  
    while((str1[i]=str2[i]) != '\0')  
        i++;  
}
```

```
void strcpy(char *str1, const char *str2)  
{  
    int i=0;  
    while((*str1+i)= *(str2+i)) != '\0')  
        i++;  
}
```

```
void strcpy(char *str1, const char *str2)  
{  
    while((*str1 = *str2) != '\0')  
    {  
        str1++;  
        str2++;  
    }  
}
```

```
void strcpy(char *str1, const char *str2)  
{  
    while ((*str1++ = *str2++));  
}
```

Beispiel: Zeichenketten und Pointer:

```
int main(void)
{
    int    i = 0;
    char   zeichenKette1[] = "Hier stehen ganz viele Zeichen",
           zeichenKette2[32] = {'D','i','e',
                                ' ','z','w','e','i','t','e',
                                ' ','Z','e','i','c','h','e','n','k','e',
                                't','t','e','\0'},
           *zeichenKette3 = "Heiter geht's weiter!",
           *zeichenPtr    = NULL;

    printf("String 1: %s\n", zeichenKette1);
    printf("String 2: %s\n", zeichenKette2);
    printf("String 3: %s\n\n", zeichenKette3);

    printf("Zeichen 5 aus String 1: %c\n", zeichenKette1[5]);
    printf("String 1 ab Zeichen 5: %s\n\n", &zeichenKette1[5]);

    zeichenPtr = zeichenKette2;

    printf("ZeichenPtr      zeigt auf: %s\n", zeichenPtr);
    printf("ZeichenPtr + 4 zeigt auf: %s\n", zeichenPtr+4);

    for(i = 0; i < 24; i++)
    {
        printf("%c", *(zeichenPtr+i));
    }
    printf("\n\n");
    return 0;
}
```

Ausgabe:

String 1: Hier stehen ganz viele Zeichen
String 2: Die zweite Zeichenkette
String 3: Heiter geht's weiter!

Zeichen 5 aus String 1: s
String 1 ab Zeichen 5: stehen ganz viele Zeichen

ZeichenPtr zeigt auf: Die zweite Zeichenkette
ZeichenPtr + 4 zeigt auf: zweite Zeichenkette
Die zweite Zeichenkette

Im nächsten Programmbeispiel werden drei Versionen einer Funktion vom Datentyp `int` einander gegenübergestellt, die zur Ermittlung der Länge einer Zeichenkette dienen.

```
#include <stdio.h>

int strlen0(const char s[]);
int strlen1(const char *s);
int strlen2(const char *s);

int main(void) /* Laenge einer Zeichenkette ermitteln */
{
    char name[50+1];

    printf("Zeichenkette mit max. 50 Zeichen eingeben:\n");
    scanf("%s", name);
    printf(" name: %s\n", name);
    printf("Laenge von name mit strlen0: %d\n", strlen0(name));
    printf("Laenge von name mit strlen1: %d\n", strlen1(name));
    printf("Laenge von name mit strlen2: %d\n", strlen2(name));
    return 0;
}

int strlen0(const char s[]) /* liefert die Laenge von s */
{
    int i;
    for (i = 0; s[i] != '\0'; i++)
        ;
    return i;
}

int strlen1(const char *s) /* 1. Zeigerversion */
{
    int i;
    for (i = 0; *s != '\0'; s++, i++)
        ;
    return i;
}

int strlen2(const char *s) /* 2. Zeigerversion */
{
    const char *p;
    for (p = s; *p != '\0'; p++)
        ;
    return p - s;
}
```

Die Funktion `strlen0` verwendet semantisch keine Zeigervariable und hat als Parameter ein Feld vom Datentyp `char` unbestimmter Länge. Tatsächlich wird aber auch hier ein Pointer deklariert, nur darf dieser innerhalb der Funktion nicht geändert werden.

In der `for` Anweisung der Funktion `strlen0` wird die Laufvariable `i` ausgehend vom Anfangswert Null solange inkrementiert, wie das `i`-te Zeichen der beim Aufruf übergebenen Zeichenkette vom Nullzeichen `'\0'` am Ende der Zeichenkette verschieden ist. Der Wert von `i`, für den `s[i]` gleich `'\0'` gilt, wird nicht mehr inkrementiert und über die `return` Anweisung als Länge der Zeichenkette an die rufende Funktion `main` zurückgegeben.

Die Funktionen `strlen1` und `strlen2` haben jeweils einen Zeiger auf den Datentyp `const char` als Parameter. Wenn beim Aufruf von `strlen1` eine Zeichenkette als Argument übergeben wird, wird die Adresse des ersten

Elements übergeben, so dass der Zeiger `s` auf das erste Element der übergebenen Zeichenkette zeigt. In der `for` Anweisung von `strlen1` wird geprüft, ob das Objekt, auf welches `s` zeigt, vom Nullzeichen am Ende der Zeichenkette verschieden ist. Ist dies nicht der Fall, werden der Zeiger `s` und die Laufvariable `i` inkrementiert. Durch die Inkrementierung von `s` weist diese Zeigervariable auf das nächste Element der beim Aufruf übergebenen Zeichenkette. Wenn aber der Zeiger `s` auf das Nullzeichen am Ende der Zeichenkette zeigt, findet keine Inkrementierung von `s` und `i` mehr statt. Der Wert von `i` wird durch die `return` Anweisung an `main` zurückgegeben.

In der Funktion `strlen2` wird `p` als Zeiger auf den Datentyp `const char` vereinbart. Im ersten Ausdruck in der `for` Anweisung wird `p` so initialisiert, dass `p` auf das erste Zeichen der als Argument übergebenen Zeichenkette zeigt. In der Schleifenbedingung wird geprüft, ob das Nullzeichen am Ende der Zeichenkette erreicht ist. Ist dies noch nicht der Fall, wird der Zeiger `p` inkrementiert, so dass er auf das nächste Element der Zeichenkette zeigt. Ist das Nullzeichen aber erreicht, findet eine Inkrementierung von `p` nicht mehr statt. Die Zeigerdifferenz `p-s`, die als Ergebnis zurückgegeben wird, gibt die Anzahl der Inkrementierungen von `p` an, also die Länge der Zeichenkette.

Das folgende Beispiel zeigt, wie man ein Array von Zeigern initialisieren kann:

```
char *month_name(int n)
{
    static char *name[] = { /* Array von Zeigern */
        "falscher Monat", /* Ein String ist ein char-Array */
        "Januar",         /* und daher durch seine */
        "Februar",        /* Anfangsadresse charakterisiert */
        "Maerz",
        "April",
        "Mai",
        "Juni",
        "Juli",
        "August",
        "September",
        "Oktober",
        "November",
        "Dezember"
    };

    return ((n < 1 || n > 12) ? name[0] : name[n]);
}
```

Beispiel: Sortieren von `n` eingegebenen `int`-Werten

An die Sortierfunktion `ssort` werden nur zwei Zeiger übergeben, die auf das erste und letzte Element des Feldes zeigen. Innerhalb von `ssort` wird auch nur mit Zeigern gearbeitet.

```
#include <stdio.h>
#define MAX 100

void ausgabe(char *text, int *feld, int m);
void ssort(int *first, int *last);
```

```
int main(void)
{
    int feld[MAX];
    int m, anz;

    printf("Eingabe von n int-Werten (unsortiert) - bis EOF\n");
    for (m = 0; m < MAX; m++)
    { /* Einlesen bis EOF erreicht wurde
        oder bis MAX Werte eingelesen wurden */
        anz = scanf("%d", &feld[m]);
        if (anz == EOF) break;
    }
    printf("\n");

    ausgabe("Unsortiert:", feld, m);

    ssort(&feld[0], &feld[m]);
        /* oder kuerzer: ssort(feld, &feld[m]); */

    ausgabe("Sortiert:", feld, m);

    return 0;
}

void ausgabe(char* text, int *feld, int m)
{ /* Array ausgeben */
    int j;

    printf("%s\n", text);
    for(j = 0; j < m; j++)
    {
        printf(" %6d\n", feld[j]);
    }
    puts(""); /* Ausgabe eines Zeilenvorschubs */
}

void ssort(int *first, int *last)
    /* Sortieren durch direktes Einfuegen */
{
    int *i, *j, temp;

    for(i = first; i < last; i++)
    {
        for(j = i-1; j >= first && *j > *(j+1); j--)
        {
            temp = *(j+1);
            *(j+1) = *j;
            *j = temp;
        }
    }
}
```

Zeiger auf mehrdimensionale Arrays

Zur Erinnerung: bei statischer Speicherbelegung sind mehrdimensionale Arrays intern eindimensional angelegt, die Arrayzeilen liegen alle hintereinander. Definiert man beispielsweise

```
float matrix[N][M];
```

Dann berechnet sich die die Adresse von `matrix[1][2]` mit der Zeigerarithmetik zu:

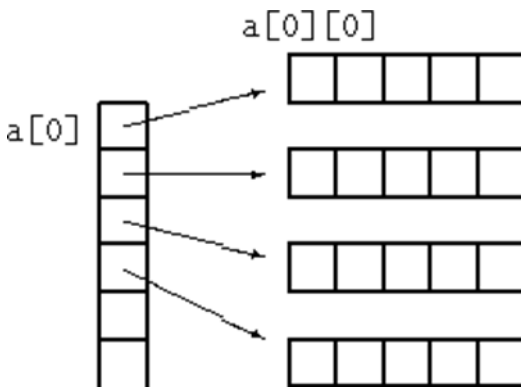
$$\begin{aligned}\text{matrix}[1][2] &\equiv *(\text{matrix}[1] + 2) \\ &\equiv *(*(\text{matrix} + 1) + 2) \\ &\equiv *(*\text{matrix} + 1 * M + 2)\end{aligned}$$

Der letzte Ausdruck ist von der Form, wie er intern tatsächlich berechnet wird. Die Zeilenlänge M muss dem Compiler also explizit bekannt sein!

Alternative: Adressen zu den Zeilenanfängen können explizit abgespeichert werden.
Beispiel:

```
int aa[N][M], *a[N], i;  
for(i=0; i<N; i++)  
    a[i] = aa[i];
```

ab dann nur noch Verwendung von `a[k][l]`.



"echt 2-dimensional"; Adressen werden nicht berechnet, sondern nur gelesen; Zeilenlänge muss dazu nicht bekannt sein. Nachteil: es ist (geringfügig) mehr Speicherplatz nötig.

Zeiger auf Strukturen

Strukturpointer finden so häufig Verwendung, dass es dafür eine eigene Syntax gibt; bei gegebener Adresse auf eine Strukturvariable werden deren Elemente mit '`->`' angesprochen, in der Form:

```
Zeiger auf Strukturvariable -> Elementname
```

Beispiel: mit der Strukturdefinition

```
struct point
{
    double  px, py;
    int     farbe;
} punkt, sprueh[100];
```

sind folgende Aufrufe äquivalent:

```
punkt.px           entspricht (&punkt)->px
sprueh[20].px      entspricht (sprueh+20)->px
```

Nun wird noch ein Zeiger auf punkt definiert:

```
struct point *zeiger;
```

Mit der Anweisung `zeiger = &punkt;` kann man zeiger auf punkt zeigen lassen. Der Zugriff auf die Komponente `px` von `punkt` würde in reiner Pointerschreibweise lauten (erster Versuch):

```
irgendwas = *zeiger.px;
```

Diese Notation ist aber zweideutig. Bedeutet das jetzt "*Das worauf zeiger zeigt, Komponente px*" oder "*Das worauf zeiger.px zeigt*"? Diese Zweideutigkeit wird man los durch (zweiter Versuch):

```
irgendwas = (*zeiger).px;
```

Nun ist es eindeutig "*Das worauf zeiger zeigt, Komponente px*". Der Operator `->` vereinfacht das nun nur noch und wir schreiben (Endfassung):

```
irgendwas = zeiger->px;
```

Das folgende Beispiel beginnt mit der Deklaration eines namenlosen struct-Typs für Variablen, die Personaldaten von Angestellten (Name, Personalnummer, Gehalt) aufnehmen. Ausserdem werden drei struct-Variablen `ang1`, `ang2`, `ang3` und eine Zeigervariable auf solche structs definiert:

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

int main(void)
{
    struct personal
    {
        char  name[20];
        int   pers_nr;
        float gehalt;
    };
    struct personal ang_1 = { "Gundel Gakeley", 3344, 5000.00 };
    struct personal ang_2, ang_3, *ang_zeiger;

    /* Uebertragung aller Werte von ang1 nach ang2 */
    ang_2 = ang_1;

    /* Setzen der Werte von ang3 durch Einzelzuweisungen */
    ang_3.pers_nr = 4733;
    ang_3.gehalt  = ang_1.gehalt + ang_2.gehalt;
    strcpy(ang_3.name, "Donald Duck");
```

```
/* Zuweisung der Adresse von ang1 an ang_zeiger */
ang_zeiger = &ang_1;

/* Aenderung der Namenskomponente von ang_1 ueber Pointer */
strcpy(ang_zeiger->name, "Dagobert Duck");
return 0;
}
```

2.5 Zeiger-Array

Zeiger-Arrays sind Arrays deren Komponenten Zeiger sind. Genauso wie man Vektoren aus den Grunddatentypen (char, int, float und double) bilden kann, kann man dies auch mit Zeigern tun. Ein Vektor von Zeigern wird so definiert:

```
Grunddatentyp *Vektorname []
gelesen von rechts nach links (wegen des Vorranges von []): Vektor von Zeigern. Dagegen ist
(*Vektorname) []
ein Zeiger auf einen Vektor!
```

Die Komponenten des Vektors können natürlich auch Adressen von Arrays sein. Damit ergibt sich eine Ähnlichkeit zu mehrdimensionalen Arrays.

Beispiel:

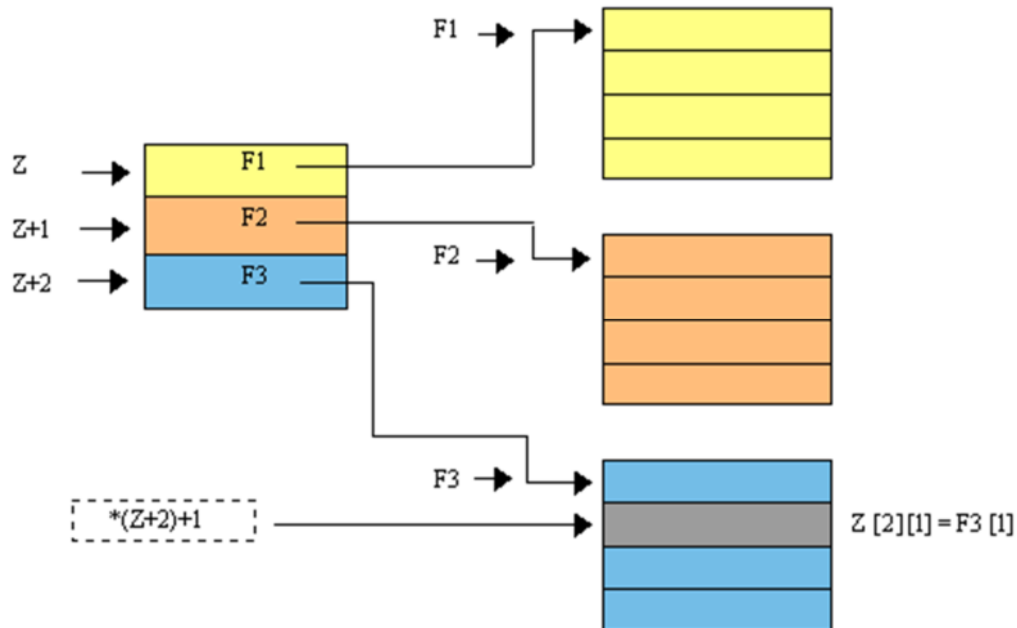
```
char *Z[3];    /* Definition für ein Array mit 3 Elementen,
                die jeweils Zeiger auf char-Typen sind.
                Z ist also ein Zeiger auf ein Array mit
                3 char-Zeigern */

char F1 [4],   /* Definition für 3 */
     F2 [4],   /* char-Arrays mit jeweils */
     F3 [4];   /* der Länge 4 */

Z[0] = F1;     /* Wertzuweisung an die char-Zeiger */
Z[1] = F2;
Z[2] = F3;
```

Die Situation stellt sich bildlich folgendermaßen dar:

ursprünglich [Programmieren in C](#)
von Prof. Jürgen Plate



Man sieht folgende Analogie zwischen Zeiger-Arrays und mehrdimensionalen Arrays:

$Z+i$	ist ein Zeiger auf das $(i+1)$ -te Element des Zeiger-Arrays Z
$*(Z+i) = Z[i]$	ist das Array, auf das das $(i+1)$ -te Element des Zeiger-Arrays Z zeigt, also ein Zeiger auf das erste Element dieses Arrays.
$*(Z+i) + j = Z[i] + j$	ist ein Zeiger auf das $(j+1)$ -te Element des Arrays, auf das das $(i+1)$ -te Element des Zeiger-Arrays Z zeigt.
$*(*(Z+i) + j) = *(Z[i] + j) = Z[i][j] = (*(pa+i))[j]$	ist das $(j+1)$ -te Element des Arrays, auf das das $(i+1)$ -te Element des Zeiger-Arrays Z zeigt.

Das obige Beispiel geht davon aus, dass die Elemente des Zeiger-Arrays alle auf Arrays gleicher Größe verweisen. Dies bedingt die große Ähnlichkeit zu mehrdimensionalen Arrays. Bei häufigen Anwendungen von Zeiger-Arrays besitzen die referenzierten Arrays unterschiedliche Längen, z.B. verschieden lange Strings.

```
/* Programm: ZEIGER-Array */
/* Aehnlichkeit und Unterschied zu mehrdim. Arrays */

#include <stdio.h>

char tagFeld[][11] =
{
    "Montag",      /* Dies ist ein 2-dimensionales Array */
    "Dienstag",    /* Die zweite Dimension ist fest 11, */
    "Mittwoch",    /* die erste wird durch die Initialisierung */
    "Donnerstag",  /* festgelegt mit 7 - für jeden String */
    "Freitag",     /* werden genau 11 Zeichen reserviert */
}
```



```
"Samstag",
"Sonntag"
};

char *tagZeiger[] =
{
    "Montag",      /* Dies ist ein Zeigerarray mit genau */
    "Dienstag",    /* 7 Zeigern als Elemente. Die Strings */
    "Mittwoch",    /* besitzen unterschiedliche Laenge */
    "Donnerstag",
    "Freitag",
    "Samstag",
    "Sonntag"
};

int indexLesen(int *,int *);

int main(void)
{
    int i, j;
    char Ca, Cb;

    while (indexLesen(&i, &j) >= 0)
    {
        /* Zugriff auf das 2-dimensionale Array */
        ca = tagFeld[i][j];
        /* Zugriff über das Zeiger-Array */
        cb = tagZeiger[i][j];

        printf("TagFeld [%2d][%2d]    = %c (= %2x hex)\n", i, j, ca, ca);
        printf("TagZeiger [%2d][%2d] = %c (= %2x hex)\n",i, j, cb, cb);
    }

    return 0;
}

int indexLesen(int *ip, int *jp)
{
    int retval=1;
    printf("\nZeile:  "); scanf("%d", ip);
    if(*ip < 0)
        retval=-1;
    else
    {
        printf("\nSpalte: "); scanf("%d", jp);
        if(*jp < 0)
            retval=-1;
    }
    return retval;
}
```

Das Ergebnis eines Probelaufs sieht so aus:

```
Zeile: 0
Spalte: 4
TagFeld [ 0][ 4] = a (= 61 hex)
TagZeiger [ 0][ 4] = a (= 61 hex)

Zeile: 2
Spalte: 9
TagFeld [ 2][ 9] = (= 0 hex) /* Wie laesst sich dieser Unterschied
*/
TagZeiger [ 2][ 9] = D (= 44 hex) /* erkl hoeren ? */

Zeile: -1
```

Das folgende Beispiel zeigt, wie man einen Vektor von Zeigern initialisieren kann:

```
char *month_name(int n) /* liefert Name des n. Monats */
{
    static char *name[] = { /* Vektor von Zeigern */
        "falscher Monat", /* String ist ein char- */
        "Januar", /* Vektor und daher durch */
        "Februar", /* seine Anfangsadresse */
        "Maerz", /* charakterisiert */
        "April",
        "Mai",
        "Juni",
        "Juli",
        "August",
        "September",
        "Oktober",
        "November",
        "Dezember"
    };
    return ((n < 1 || n > 12) ? name[0] : name[n]);
}
```

2.6 Zeiger auf Funktionen

In C kann man nicht nur Zeiger auf Datentypen, sondern auch auf Funktionen definieren, die man dann wie Variablen verwenden kann. Damit wird es möglich, Funktionen an Funktionen zu übergeben. Der Zeiger enthält dann die Startadresse der Funktion im Arbeitsspeicher. Ein Zeiger auf eine Funktion wird folgendermaßen vereinbart: Rückgabety p (*Funktionsname) (Funktionsargumente)

Beachten Sie die Klammer um den Stern und den Funktionsnamen (*Funktionsname). Die runde Klammer ist zwingend erforderlich - vgl. Prioritäten!

Im Gegensatz dazu definiert *Funktionsname () eine Funktion, die einen Zeigerwert liefert.

Ein erstes Beispiel:

```
double (*zFunc) (int i);
```

zFunc ist ein Zeiger auf eine Funktion, die einen int-Parameter erwartet und einen double-Wert zurückliefert.

Was kann man mit Funktionszeigern anstellen? Es ist möglich,

- Zeigervariable, die auf Funktionen zeigen zu vereinbaren
- Funktionszeigervariablen Werte zuzuweisen
- Arrays von Funktionszeigern zu definieren
- Zeiger auf Funktionen als Parameter zu übergeben
- Zeiger auf Funktionen als Rückgabewerte zu erhalten

Die Wertzuweisung an eine Funktionszeigervariable erfolgt wie bei den Arrays, so ist auch bei Funktionen der Name allein als Zeiger auf die Funktion festgelegt. Es wird kein &-Operator angewendet, der Funktionsname ist eine Adreßkonstante! Beispiel:

```
int funkt(int);          /* Deklaration der Funktion funkt() */
int (*zFunkt) (int);     /* Definition der Funktionszeigervariablen */

zFunkt = funkt;          /* Zuweisung der Adresse der Funktion */
                        /* als Wert an den Zeiger */
```

Der Aufruf einer Funktion über einen Funktionszeiger erfolgt durch Dereferenzieren (mit *) des Zeigers und Angabe der aktuellen Parameterliste. Zum Beispiel:

```
ergebnis = (*zFunkt) (2*i);
```

Es stellt sich die Frage, wozu sowas nötig sein sollte. Man kann auf diese Weise allgemein verwendbare Funktionen programmieren. Angenommen, sie wollen eine Funktion schreiben, die den Graphen einer beliebigen Funktion zeichnet. Ohne Funktionszeiger müßte die Funktion jeweils im Quellcode eingefügt und dann das Programm neu kompiliert und ausgeführt werden. Natürlich kann man sowas auch automatisieren. Ein Programm öffnet eine Quelldatei, modifiziert sie, startet die Compilierung und führt schließlich das erzeugte Programm aus. Das Ganze ist aber recht kompliziert. Einfacher und eleganter ist es da sicher, an die Grafikfunktion einfach einen Zeiger auf eine beliebige Funktion zu übergeben.

Ein erstes Beispiel:

```
#include <stdio.h>

int i;
int funzel();          /* Deklaration der Funktion */
int (*zFunkt) ();      /* Definition des Funktionszeigers */

int (*druck) (char *format, ...); /* ja, das geht :- ) */

int main(void)
{
    zFunkt = funzel;    /* Zuweisung der Funktionsadresse */
    i = (*zFunkt) ();   /* Funktion ausführen */
}
```

```
/* Stimmen die Adressen? */
printf("Adresse funzel(): %p, Adresse zFunkt: %p\n",
      zFunkt, funzel);

druck = printf;          /* Zuweisung der Funktionsadresse */
/* druck ausfuehren: */
(*druck) ("Sie mal einer da! I war uebrigens: %d\n", i);

return 0;
}

int funzel()
{
    printf("Funzel was here!\n");
    return 42;
}
```

Die Ausgabe sieht etwa so aus:

funzel was here!

Adresse funzel(): 0x401120, Adresse zFunkt: 0x401120

Sie mal einer da! I war uebrigens: 42

Natürlich sind auch Arrays von Funktionspointern möglich, Beispiel:

```
double (*trig[3])(double), x;

trig[0] = exp;
trig[1] = sin;
trig[2] = cos;
```

Die Aufrufe wie z. B. `sin(x)` und `(*trig[1])(x)` sind dann äquivalent. Funktionen können so mit einem Index versehen werden.

Funktionspointer und Funktionvariable erlauben es, generische Funktionen zu schreiben. Als Beispiel diene hier die Bibliotheksfunktion `qsort()` aus `<stdlib.h>`:

```
void qsort(void *base, size_t nel, size_t width,
          int (*compar) (const void *, const void *));
```

`qsort()` sortiert ein Array `base[0]` bis `base[nel-1]` von Objekten der Größe `width` in aufsteigender Reihenfolge. Die Vergleichsfunktion `compar()` gibt einen negativen Wert zurück, wenn ihr erstes Argument kleiner ist als das zweite, Null wenn die Argumente gleich sind und einen positiven Wert, wenn das zweite größer als das erste ist.

In `qsort()` werden Vergleiche immer mit der Funktion `compar()` vorgenommen:

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar) (const void *, const void *))
{
    ...

    if ((*compar)(base[i], base[j]) < 0)
        swap(base, i, j);

    ...

}
```

Mit der Wahl einer geeigneten Vergleichsfunktion können also Arrays beliebigen Typs sortiert werden. Beispiel:

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

int agecompare(const void *i, const void *j); /* Alter vergleichen ... */
int namecompare(const void *i, const void *j); /* Namen vergleichen ... */

typedef struct
{
    char *name;
    int age;
} Person;

int main(void)
{
    Person a[ARRAYSIZE];

    ... /* Initialisierungen ... */

    /* a nach Alter sortieren */
    qsort(a, ARRAYSIZE, sizeof(Person), agecompare);
    ...
    /* a nach Namen sortieren */
    qsort(a, ARRAYSIZE, sizeof(Person), namecompare);
    ...

    return 0;
}

int agecompare(const void *i, const void *j)
{
    int ii, jj, retval=0;

    ii = ((const Person*)i)->age;
    jj = ((const Person*)j)->age;

    if (ii > jj) retval=1;
    if (jj > ii) retval=-1;
    return retval;
}
```

```
int namecompare(const void *i, const void *j)
{
    const char *ii, *jj;

    ii = ((const Person*)i)->name;
    jj = ((const Person*)j)->name;

    return strcmp(ii, jj);
}
```

2.7 Argumente der Kommandozeile

Im Allgemeinen hat diese Funktion `main` zwei Parameter: `argc` und `argv`, die Angaben über die Kommandoparameter beim Programmaufruf enthalten.

`argc` ist ein `int`-Parameter und enthält die Anzahl der Parameter der Kommandozeile einschließlich des Programmaufrufes selbst (hat also immer mindestens den Wert 1).

`argv` ist ein Vektor mit Zeigern auf Zeichenketten. Diese Zeichenketten enthalten die Aufrufparameter der Kommandozeile, wobei der 1. Parameter der Programmaufruf selbst ist. Letztes Argument ist die Vektorkomponente `argv[argc - 1]`.

Beispiel: Programm `echo` gibt alle Argumente auf die Standardausgabe aus. Der Programmaufruf auf Betriebssystemebene sieht so aus:

```
echo Hier kommt das Echo!
Die Werte von argc und argv im aufgerufenen Programm lauten:
argc = 5
argv[0]: "echo"
argv[1]: "Hier"
argv[2]: "kommt"
argv[3]: "das"
argv[4]: "Echo!"
argv[5]: NULL
```

Beispielprogramm: 3 Versionen von `echo`

```
/* 1. Version */
int main(int argc, char *argv[])
{
    int i;
    for (i=1; i < argc; i++)
        printf("%s", argv[i]);
    putchar('\n');
    return 0;
}

/* 2. Version */
int main(int argc, char *argv[])
{
    while (--argc > 0)
```

```
    printf("%s", *++argv);  
    putchar('\n');  
    return 0;  
}  
/* 3.Version */  
int main(int argc, char **argv)  
{  
    while (--argc > 0)  
        printf("%s ", *++argv);  
    putchar('\n');  
    return 0;  
}
```

3 Dateien

Dateien sind ein allgemeines Konzept für die permanente Speicherung bzw. Ein-/Ausgabe. Dateien bestehen aus beliebig vielen Komponenten eines bestimmten Datentyps. Je nach Dateart können die einzelnen Komponenten sequentiell oder wahlfrei gelesen werden. Am Ende einer Datei können weitere Komponenten hinzugefügt werden. Die Abbildung der abstrakten Dateistruktur auf reale Speichergeräte (Platte, Band, Drucker, etc.) erfolgt durch das Betriebssystem.

Eine Datei besitzt also lauter Komponenten gleichen Typs. Zusammen mit der Dateivariablen wird implizit auch ein Pufferbereich im Arbeitsspeicher definiert, der mindestens eine Dateikomponente (→ aktueller Wert) aufnehmen kann. Auf diesen Datenwert wird dann über die Dateivariablen zugegriffen. Zusammen mit dem Typ "Datei" müssen einige Standardoperationen (sog. "Primitiven") definiert sein, die den Zugriff auf die Datei erlauben:

- **Open**
Eröffnen einer Datei zur Inspektion des Inhalts (Lesen) oder zum Anfügen neuer Komponenten am Ende (Schreiben). Mit dieser Prozedur wird nicht nur der Zugriffsmodus festgelegt, sondern gegebenenfalls auch eine leere Datei generiert (beim Modus "Schreiben" bei einer noch nicht vorhandenen Datei).
- **Close**
Beenden des Zugriffs auf eine Datei. Das Betriebssystem wird angewiesen, die Datei zu schließen. Damit verbunden sind Verwaltungsvorgänge des Betriebssystems, z. B. Wegschreiben des Inhalts des Pufferbereichs.
- **Read, Write**
Lesen und Schreiben von Komponenten einer Datei. Die Arbeitsweise dieser Operationen hängt von der Art der Datei ab. Auf diese Arbeitsweise wird in einem folgenden Kapitel noch näher eingegangen.

3.1 Zugriffsart

Nach Art des Zugriffs auf die Komponenten wird unterschieden in sequentielle Dateien und Dateien mit wahlfreiem Zugriff.

- **Sequentielle Dateien (sequential files)**
In den Anfängen der Computertechnik gab es nur sequentielle Dateien, da das Lesen/Schreiben bei Magnetbändern und Lochstreifen nur streng sequentiell möglich war. Auch der Drucker wird über sequentielle Dateivariablen angesprochen. Beim Öffnen einer Datei muss zwischen Lesen und Schreiben unterschieden werden:
 - **Lesen**
Der Dateizeiger wird auf die erste Komponente positioniert. Nach dem Zugriff auf eine Komponente wird auf die nächstfolgende Komponente positioniert.
Das Ende einer Datei wird durch eine spezielle Standardfunktion (`fEOF()`) oder einen speziellen Wert (`EOF` = End Of File) zurückgemeldet.
 - **Schreiben**
Der Dateizeiger wird hinter der letzten Komponente positioniert; durch das Schreiben wird die Datei um eine Komponente erweitert. Danach wird wieder hinter die Datei positioniert.
- **Dateien mit wahlfreiem Zugriff (random files)**
Bei Plattenspeicher kann wahlfrei auf jeden Datenblock der Platte zugegriffen werden. Mit der Verbreitung solcher Speicher lag es nahe, den Datentyp "Datei" um diese Möglichkeit zu erweitern. Bei sequentiellen Dateien wird die Positionierung auf eine Komponente implizit vorgenommen und unterliegt nicht dem Einfluß des Programms. Bei Dateien mit wahlfreiem Zugriff werden die Lese- und Schreiboperationen um die Angabe eines Komponentenindex ergänzt. Der Komponentenindex wird dabei in der Regel von 1 ab aufwärts gezählt. Damit besitzt die Datei mit wahlfreiem Zugriff eine starke Ähnlichkeit mit dem Datentyp "Array". Der

Versuch, eine Komponente zu lesen, die "hinter" dem Dateiende liegt, führt zu einem Fehler. Beim Versuch, eine Komponente zu schreiben, deren Index I größer als die Anzahl N der aktuell vorhandenen Komponenten ist, können zwei Fälle unterschieden werden:

- $I = N + 1$: Die Datei wird um eine Komponente erweitert (wie bei der sequentiellen Datei).
- $I > N + 1$: Bei einigen Systemen führt der Versuch zu einem Fehler. Bei anderen Implementierungen wird der "Zwischenraum" mit leeren Komponenten aufgefüllt (gefährlich!).

3.2 Grundlegende Abläufe auf Dateien

Der Zugriff auf Laufwerke bzw. allg. auf die Peripherie erfolgt in C ebenfalls über Zeiger (sog. *Streams* oder *Filepointer*). Diesbezüglich gilt alles als "Datei"; prinzipiell muss eine Datei vor dem Zugriff auf sie geöffnet, und hinterher wieder geschlossen werden. Verwendet werden Funktionen, Datentypen, etc. aus der Standardbibliothek `<stdio.h>`.

Wenn in einem Programm nichts weiter vereinbart wird, so wirken die Schreib- und Lesebefehle auf sogenannte Standarddateien. Diese sind durch das Betriebssystem vordefiniert (z. B. Tastatur als Eingabe und Bildschirm als Ausgabe). Will man das ändern, so kann man die Standard-E/A z. B. durch UNIX-Kommandos umlenken.

Programmname < Dateiname

Dieses Kommando bewirkt, dass das Programm von der angegebenen Datei liest.

Programmname > Dateiname

Dieses Kommando bewirkt, dass das Programm auf die angegebene Datei ausgibt.

Die sogenannten *Standard-Filepointer* sind immer initialisiert, die zugehörigen Dateien immer geöffnet:

<code>stdin</code>	Standardeingabe	(Tastatur)
<code>stdout</code>	Standardausgabe	(Bildschirm)
<code>stderr</code>	Fehlerausgabe	(Bildschirm!)

Die Unterscheidung zwischen `stdout` und `stderr` ist beispielsweise dann relevant, wenn man die Bildschirmausgabe (`stdout`) in irgendwelche Files umleitet (z. B. mit '>' unter Eingabeaufforderungen in Windows/DOS und UNIX), aber verhindert werden soll, dass auch etwaige Fehlermeldungen dorthin "verschwinden".

Ein einfaches Beispiel für Zugriff auf die Standarddateien `stdin` und `stdout`:

```
/* Programm 'cat' */  
  
#include <stdio.h>  
  
int main(void)  
{  
    int ch;  
    while ((ch = getchar()) != EOF)  
        putchar(ch);  
    return 0;  
}
```

Solange Eingabedaten von der Standardeingabe kommen werden sie auf der Standardausgabe wieder ausgegeben. Das Programm "echo" also die Standardeingabe. Mit der Ausgabeumleitung kann man damit Tastatureingaben in eine Datei schreiben, z. B. durch den Programmaufruf: `cat > mein.text`. Mittels Eingabeumleitung kann man Dateiinhalte auf den Bildschirm bringen, z. B.: `cat < mein.text`.

Eine Unschönheit von C kann man an diesem Programm gut erkennen. Die Variable `ch` ist nicht - wie man es erwarten sollte - als `char` definiert, sondern als `int`. Das liegt einzig und alleine daran, dass die vordefinierte Konstante `EOF` den Wert `-1` besitzt und daher bei der Verwendung einer `char`-Variablen das Datei- bzw. Eingabeende nicht erkannt würde.

Das Eingabeende unter Linux/UNIX wird durch das Zeichen `Ctrl-D` (bzw. auf der deutschen Tastatur `Strg-D`) repräsentiert. Unter DOS/Windows ist es das Zeichen `Ctrl-Z` (bzw. auf der deutschen Tastatur `Strg-Z`).

Beispiel: Kopieren der Standardeingabe zur Standardausgabe; Zählen der Byteanzahl.

```
#include <stdio.h>

unsigned copy(void);

void main()
{
    unsigned bytes;
    bytes = copy();
    fprintf (stderr, "%u Bytes kopiert.\n", bytes);
}

unsigned copy(void)
{
    int c;
    unsigned bytes;
    bytes = 0;
    while ((c = getchar()) != EOF)
    {
        putchar(c);
        bytes = bytes + 1;
    }
    return bytes;
}
```

Beispiel: Kopieren der Standardeingabe zur Standardausgabe; Numerieren aller Zeilen.

```
#include <stdio.h>

#define YES 1
#define NO 0

int main(void)
{
    short nl_anz;
    int c, zeile_nr;
    zeile_nr = 1;
```

```
nl_anz = YES;
while ((c = getchar()) != EOF)
{
    if(nl_anz)
    {
        printf("%6d ", zeile_nr);
        zeile_nr = zeile_nr + 1;
    }
    if(c != '\n')
        nl_anz = NO;
    else
        nl_anz = YES;
    putchar(c);
}
return 0;
}
```

Beispiel: Worte, Zeichen und Zeilen zählen.

```
#include <stdio.h>

#define IMWORT      1
#define AUSSEN      0

main()
{
    int c, wo;
    int nc, nw, nl;                /* Zeichen, Worte, Zeilen */

    wo = AUSSEN;                   /* Anfangswerte */
    nc = nw = nl = 0;

    while((c = getchar()) != EOF)
    {
        ++nc;
        if(c == '\n') ++nl;
        if( (c == '\n') || (c == '\t') || (c == ' ') )
            wo = AUSSEN;
        else
            if(wo == AUSSEN)
            {
                wo=IMWORT;
                ++nw;
            }
    }

    printf("\nDas waren %d Zeichen, %d Worte und %d Zeilen.\n", nc,
           nw, nl);
    return 0;
}
```

Beispiel: Wortlängen im Eingabestrom ermitteln

Es wird die Länge eines jeden Wortes bestimmt und die Längen zwischen 1 und 25 gezählt. Alle Wörter, die länger als 25 Zeichen sind, werden in einem 26. Zähler registriert. Als Ausgabe erhält man die Anzahlen der der Worte:

```
#include <stdio.h>

#define MAXWORDLEN 26                /* maximale Wort-Laenge + 1*/

int main(void)
{
    int c;                          /* gelesenes Zeichen */
    int wl, zl;                     /* Wortlaenge, Zeilenlaenge */
    int i;                          /* Schleifenzaehler */
    int words, lines;               /* Wortzaehler, Zeilenzaehler */
    int maxline, maxword;           /* Maximallaengen */
    int wordlen[MAXWORDLEN];        /* speichert die Wortlaengen */

    zl = words = maxline = maxword = lines = 0;
    for (i=0; i < MAXWORDLEN; i++)
        wordlen[i] = 0;
    c = getchar();                  /* 1. Zeichen lesen */
    while (c != EOF)
    {
        while (!((('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z'))
            && (c != '\n') && (c != EOF)))
        {
            c = getchar();          /* Wortanfang suchen */
            /* Zeichen lesen */
            zl++;                   /* Zeilenlaenge erhoehen */
        }
        wl = 0;                   /* Wortlaenge auf 0 setzen */
        while ((c != EOF)
            && (((('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z'))
            || ('0' <= c && c <= '9') || (c == '_'))))
        {
            c = getchar();          /* naechstes Wort lesen */
            /* Zeichen lesen */
            zl++;                   /* Zeilenlaenge erhoehen */
            wl++;                   /* Wortlaenge erhoehen */
        }
        if (wl > 0) words++;        /* Wortzaehler erhoehen */
        if (wl > maxword) maxword = wl; /* max. Wortlaenge */
        if (wl >= MAXWORDLEN)      /* groesser als MAXWORDLEN? */
            wordlen[MAXWORDLEN-1]++; /* ja - ins letzte Element */
        else
            wordlen[wl-1]++;        /* passenden Zaehler erhoehen */

        if ((c == '\n') || (c == EOF)) /* neue Zeile? */
        {
            lines++;               /* Zeilenzaehler erhoehen */
            if (zl > maxline)
                maxline = zl;      /* max. Zeilenlaenge */
            zl = 0;
            c = getchar();
        }
    }
}
```

```
    }  
}  
  
for ( i=0; i < MAXWORDLEN; i++ )    /* Maximalzahl der Worte */  
    printf("%10d | %10d\n", i, wordlen[i]);  
  
return 0;  
}
```

Sequentielle Dateien

Will man zusätzlich zu den Standard-Dateien noch weitere Dateien auf der Platte benutzen, so muss man diese Dateien an das Programm anbinden. Alle Befehle zur Dateibearbeitung sind nicht Teil von C. Die Standardbibliotheken enthalten aber viele Funktionen zum Arbeiten mit Dateien.

Im folgenden Beispiel wird eine Datei sequentiell beschrieben und wieder gelesen. An diesem Beispiel werden einige grundlegende Befehle erklärt, weshalb der Quelltext mit Zeilennummern in eckigen Klammern versehen wurde. Die Zeilennummern finden Sie in den Erklärungen unten wieder:

```
[ 1] #include <stdio.h>  
[ 2]  
[ 3] int main(void)  
[ 4] {  
[ 5]     FILE *fp;  
[ 6]     int i,xi;  
[ 7]     static char dateiname[]="daten.datei";  
[ 8]     char text[80];  
[ 9]  
[10]     /* Beschreiben der Datei */  
[11]     fp = fopen(dateiname,"w");  
[12]     if (fp == NULL)  
[13]     {  
[14]         fprintf(stderr,"Datei %s kann nicht zum Schreiben\  
[15]                     geoeffnet werden\n",dateiname);  
[16]         exit(1);  
[17]     }  
[18]     for (i=0; i<10; i=i+2)  
[19]         fprintf(fp,"%d\n",i);  
[20]     fclose(fp);  
[21]  
[22]     /* Lesen der Datei */  
[23]     fp = fopen("meine.dat","r");  
[24]     if (fp == NULL)  
[25]     {  
[26]         fprintf(stderr,"Datei %s kann nicht zum Lesen\  
[27]                     geoeffnet werden\n",dateiname);  
[28]         exit(2);  
[29]     }  
[30]     while(fscanf(fp,"%d",&xi) == 1)  
[31]     {  
[32]         printf("%d",xi);  
[33]     }
```

```
[34]    fclose(fp);  
[35]    exit(0);  
[36] }
```

Im Programm finden wir folgende Dateianweisungen:

1. **[5] Definieren eines Zeigers auf eine Dateiverwaltungsstruktur:**

```
FILE *<Dateizeiger>
```

FILE ist eine spezielle Stream-Struktur, der Datentyp für Dateien. Er ist in der Standardbibliothek `<stdio.h>` als Struktur festgelegt, die Informationen über die Datei enthält (z. B. Pufferadresse, Zugriffsrechte usw.).

Mit obiger Anweisung wird ein Zeiger auf den Datentyp FILE definiert.

2. **[11], [23] Öffnen einer Datei:**

```
<Dateizeiger> = fopen(<Dateiname>, <Zugriffsmodus>);
```

Die Funktion fopen verbindet den externen Namen der Datei mit dem Programm und liefert als Ergebnis den Zeiger auf die Beschreibung der Datei. Im Fehlerfall wird der NULL-Zeiger zurückgeliefert. Die Funktion ist definiert als

```
FILE *fopen(const char *filename, const char *modus)
```

als Zugriffsmodus steht zur Verfügung eine Kombination von "a", "r", "w" und "+":

- 'r' (Lesen (*read*))
- 'w' (Schreiben (*write*))
- 'a' (Anhängen (*append*))
- 'r+' (Lesen und Schreiben)
- 'w+' (Schreiben und Lesen)
- 'a+' (Lesen an bel. Position, Schreiben am Dateiende)

Durch anhängen eines Zusatzes kann festgelegt werden, ob es sich bei der zu bearbeitenden Datei um eine Binär- oder Textdatei handelt:

- 't' (für *text*)
- 'b' (für *binary*)

Die Funktion fopen reagiert folgendermaßen:

- Beim Öffnen einer existierenden Datei
 - zum Lesen: keine Probleme
 - zum Anhängen: keine Probleme
 - zum Schreiben: Inhalt der Datei geht verloren
- Beim Öffnen einer nicht existierenden Datei
 - zum Lesen: Fehler, Ergebnis ist NULL-Zeiger
 - zum Anhängen: neue Datei wird angelegt
 - zum Schreiben: neue Datei wird angelegt

Maximal `FOPEN_MAX` Dateien können gleichzeitig geöffnet werden, maximale

Dateinamenlänge: `FILENAME_MAX`.

Zwischen Lesen und Schreiben ist ein Aufruf von `fflush()` oder ein Positionierungsvorgang nötig.

3. [14], [19], [26] formatierte Ausgabe auf Datei:

```
fprintf(<Dateizeiger>, "<Format>", <Werte>);
```

Entspricht der Funktion `printf` und schreibt die angegebenen Werte im angegebenen Format auf die Datei.

Dateizeiger verweist auf die Datei, auf die geschrieben wird.

Es darf nicht `NULL` als Dateizeiger übergeben werden!

`fprintf` ist definiert als

```
int fprintf(FILE*, const char *format, ...);
```

4. [30] formatierte Eingabe von Datei:

```
fscanf(<Dateizeiger>, "<Format>", <Werte>);
```

Entspricht der Funktion `scanf` und liest die angegebenen Werte im vereinbarten Format der Datei. Es darf nicht `NULL` als Dateizeiger übergeben werden!

`fscanf` ist definiert als

```
int fscanf(FILE*, const char *format, ...);
```

Sie liefert als Rückgabewert die Anzahl der erfolgreich eingelesenen Werte oder -1 (EOF), falls das Dateiende erreicht wurde.

Alternativ kann das Erreichen des Dateiendes mithilfe der folgenden Funktion überprüft werden:

```
feof(<Dateizeiger>);
```

Die Funktion `feof` liefert den Integerwert 1, wenn das Dateiende gelesen wurde, sonst 0. Es darf nicht `NULL` als Dateizeiger übergeben werden!

```
int feof(FILE*);
```

Damit die Funktion einen korrekten Wert liefert, muss vorher mit einer Dateifunktion am Ende gelesen worden sein!

5. [20], [34] Schließen einer Datei:

```
fclose(<Dateizeiger>);
```

Die Datei wird geschlossen, vom Programm abgehängt und der Platz für den Filebeschreibungsbereich wieder freigegeben. Beim Schreiben auf Dateien sollte die Datei geschlossen werden, sobald alle Schreiboperationen abgeschlossen sind, da erst beim Schließen die Dateipuffer auf die Platte geschrieben und die Informationen über die Datei in der Dateiverwaltung aktualisiert werden. `fclose` ist definiert als `int fclose(FILE*)`.

Auch hier darf nicht `NULL` als Dateizeiger übergeben werden!

Im 2. Programmbeispiel werden von der Datei "daten.dat" Datenzeilen gelesen. Jede Zeile enthält einen Integerwert, einen double-Wert und ein Wort. Diese Daten werden dann mit einer vorangestellten Zeilennummer in die Datei "tabelle.txt" geschrieben.

Das Programm geht davon aus, dass eine Zeile nicht länger als 99 Zeichen ist.

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

```
int main(void)
{
    char msg[100];
```

```
int  n, i, ok=1, anzahl;
double d;
FILE *fpin, *fpout;
/* Dateien oeffnen */
fpin  = fopen("daten.dat", "rt");
fpout = fopen("tabelle.txt", "wt");
if((fpin == NULL) || (fpout == NULL))
{ /* ... hat nicht funktioniert */
    fprintf(stderr, "Fehler: I/O");
    ok=0;
}
/* jetzt neue Datei mit anlegen, mit den Wertesaetzen zeilenweise
   und mit Zeilennummern versehen */
i = 1;
while(ok) /* falls bisher kein Fehler auftrat */
{
    anzahl=fscanf(fpin, "%d%lf%s", &n, &d, msg);
    if (anzahl>=0 && anzahl<3)
    {
        fprintf(stderr, "Datei entspricht nicht dem Format.\n");
        ok=0;
    }
    else
    {
        if(feof(fpin)) /* alternativ: if (anzahl==EOF) */
            ok=0; /* Dateiende gelesen, sofort aufhoeren! */
        else
            fprintf(fpout, "%d>  %s  %d  %f\n", i++, msg, n, d);
    }
}
/* Dateien nur schliessen, falls erfolgreich geoeffnet */
if (fpout)
    fclose(fpout);
if (fpin)
    fclose(fpin);

return 0;
}
```

Ein weiteres Beispiel: Programm zum Kopieren einer Datei. Das Programm hat zwei Kommandozeilen-Parameter (Programmparameter), die Namen von Quell- und Zieldatei. In diesem Beispiel wird besonders auf die Fehlerbehandlung eingegangen. Wenn etwas schief geht, interessiert uns die Ursache, die bei vielen Funktionen als Nummer in `errno` und als englischer Text in `strerror(errno)` hinterlegt sind. Deshalb wird die Headerdatei `errno.h` eingebunden.

Sämtliche Funktionen, deren Ergebniswerte nicht mehr interessieren, wurden mit dem Kommentar `/* <- */` versehen:

- Wenn `fprintf(stderr, ...)` nicht mehr funktioniert, hilft sowieso nichts mehr.
- Wenn es schon einen anderen Fehler gegeben hat, interessiert es nicht mehr, ob die Dateien noch geschlossen werden können.

Sie sehen, in C sollte man sehr, sehr sorgfältig alle Ergebniswerte korrigieren, dies ist häufig viel aufwendiger als der Teil, der die gewünschten Aktionen durchführt.

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

int main(int argc, char *argv[])
{
    FILE *infile,*outfile;
    int c,i;

    if(argc != 3)
    {
        fprintf(stderr,
            "Aufruf: %s quelldatei zieldatei\n", argv[0]); /* <- */
        return EXIT_FAILURE;
    }
    infile = fopen(argv[1],"rb");
    if(infile == NULL)
    {
        fprintf(stderr,
            "Fehler beim Öffnen der Datei %s: %s\n",
            argv[1], strerror(errno)); /* <- */
        return EXIT_FAILURE;
    }
    outfile = fopen(argv[2],"wb");
    if(outfile == NULL)
    {
        fprintf(stderr,
            "Fehler beim Erzeugen der Datei %s: %s\n",
            argv[2], strerror(errno)); /* <- */
        fclose(infile); /* <- */
        return EXIT_FAILURE;
    }
    while((c = getc(infile)) != EOF)
        if(putc(c,outfile) == EOF)
        {
            fprintf(stderr,
                "Fehler beim Schreiben der Datei %s: %s\n",
                argv[2], strerror(errno)); /* <- */
            fclose(infile); /* <- */
            fclose(outfile); /* <- */
            return EXIT_FAILURE;
        }
    if(ferror(infile))
    {
        fprintf(stderr,
            "Fehler beim Lesen der Datei %s: %s\n",
            argv[1], strerror(errno)); /* <- */
        fclose(infile); /* <- */
        fclose(outfile); /* <- */
    }
}
```

```
    return EXIT_FAILURE;
}
if(fclose(infile) == EOF)
{
    fprintf(stderr,
        "Fehler beim Schließen der Datei %s\n", argv[1]); /* <- */
    fclose(outfile); /* <- */
    return EXIT_FAILURE;
}
if(fclose(outfile) == EOF)
{
    fprintf(stderr,
        "Fehler beim Schließen der Datei %s\n", argv[2]); /* <- */
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
```

Auch hier kann Code und Wiederholungen (häufige `fclose`-Aufrufe) eingespart werden, wenn man das Programm nicht vorzeitig mit `return` beendet, sondern mit geeigneten Kontrollstrukturen (wie z. B. `if-else`) dafür Sorge trägt, dass der Programmablauf in jedem Fall bis zur letzten Zeile der `main`-Funktion läuft.

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

int main(int argc, char *argv[])
{
    FILE *infile=NULL, *outfile=NULL;
    int c, i, result=EXIT_FAILURE;

    if(argc != 3)
    {
        fprintf(stderr,
            "Aufruf: %s Quelldatei Zieldatei\n", argv[0]); /* <- */
    }
    else
    {
        infile = fopen(argv[1], "rb");
        if(infile == NULL)
        {
            fprintf(stderr,
                "Fehler beim Öffnen der Datei %s: %s\n",
                argv[1], strerror(errno)); /* <- */
        }
        else
        {
            /* Nur wenn das Öffnen der Quelldatei erfolgreich war, dann
               versuchen die Zieldatei zu öffnen */
            outfile = fopen(argv[2], "wb");
            if(outfile == NULL)
            {
```

```
fprintf(stderr,
    "Fehler beim Erzeugen der Datei %s: %s\n",
    argv[2], strerror(errno));          /* <- */
}
}
if (infile!=NULL && outfile!=NULL)    /* beide erfolgreich
                                     geoeffnet */
{
    result=EXIT_SUCCESS;              /* ab hier gilt der Ablauf als
                                     erfolgreich */
    while(result==EXIT_SUCCESS && (c = getc(infile)) != EOF)
        if(putc(c,outfile) == EOF)
        {
            fprintf(stderr,
                "Fehler beim Schreiben der Datei %s: %s\n",
                argv[2], strerror(errno)); /* <- */
            result=EXIT_FAILURE;
        }
    if(ferror(infile))
    {
        fprintf(stderr,
            "Fehler beim Lesen der Datei %s: %s\n",
            argv[1], strerror(errno));    /* <- */
        result=EXIT_FAILURE;
    }
}
if(infile!=NULL)
    if(fclose(infile) == EOF)
    {
        fprintf(stderr,
            "Fehler beim Schließen der Datei %s\n", argv[1]); /* <- */
        result=EXIT_FAILURE;
    }
if(outfile!=NULL)
    if(fclose(outfile) == EOF)
    {
        fprintf(stderr,
            "Fehler beim Schließen der Datei %s\n", argv[2]); /* <- */
        result=EXIT_FAILURE;
    }
}
/* Das Programm erreicht auch im Fehlerfall diese Zeile */
return result;
}
```

3.3 Textdateien

Dies sind Dateien, deren Komponenten Schriftzeichen sind (Typ char). Sie nehmen eine Schlüsselrolle ein, da die Eingabe- und Ausgabedaten der meisten Computerprogramme Textfiles sind (darunter fällt beispielsweise auch die Druckausgabe). Ein Programm kann vielfach allgemein als eine Datentransformation von einer Textdatei in eine andere aufgefasst werden.

Das Zeilenende-Problem

Nun sind Texte i. a. in Zeilen unterteilt, und es stellt sich die Frage, wie diese Zeilenstruktur auszudrücken ist. In der Regel enthält der in einem Datenverarbeitungssystem (DVS) verwendete Zeichencode spezielle Steuerzeichen, von denen eines als Zeilenende-Zeichen verwendet werden kann. Bedauerlicherweise verhindert die Realisierung von Textdateien auf Betriebssystemebene eine einfache Realisierung der Textdatei.

- Bei einigen Betriebssystemen wird ein einziges Steuerzeichen als Zeilenende interpretiert (z. B. bei Linux/UNIX: Linefeed, PowerPC-Mac: Carriage Return). Das verwendete Zeichen ist jedoch nicht einheitlich festgelegt.
- Bei anderen Systemen besteht das Zeilenende aus zwei Zeichen (in der Regel Carriage Return und Linefeed, z. B. bei Windows/MS-DOS).
- Es sind auch Systeme bekannt, die keine Steuerzeichen verwenden.

Diese Unterschiede machen es oft notwendig, Textdateien bei der Übertragung zwischen verschiedenen Rechner- bzw. Betriebssystemen zu konvertieren.

Gepufferte E/A vs. ungepufferte E/A

Wie bei den meisten Betriebssystemen wird auch bei Windows/DOS oder Linux/UNIX die Ausgabe gepuffert. Das heißt, es wird erst etwas ausgegeben, wenn ein Zeilenende an das Ausgabegerät gesendet wird, oder gar erst, wenn das Programm zuende ist. Falls Sie das nicht möchten, müssen also dafür sorgen, dass nach jedem Zeichen wirklich auch eine Bildschirmausgabe erfolgt. Sie erreichen dies durch den Funktionsaufruf `fflush(stdout)`; nach jeder Ausgabe.

Die Funktion `fflush(FILE *datei)` sorgt für sofortiges Wegschreiben des internen Dateipuffers.

Auch die Eingabe wird wie die Ausgabe gepuffert. Das heißt, es wird beim Aufruf von `getchar()` gewartet, bis nach dem Zeichen die Enter-Taste gedrückt wird. Um das zu vermeiden (wenn z. B. nur ein einziger Tastendruck erfolgen soll), muss das System in den ungepufferten Betrieb geschaltet werden. Sie erreichen dies unter Linux/UNIX durch Umschalten des Terminals in den ungepufferten (raw-)Modus. Das kann durch den Aufruf des Systemkommandos `stty` erreicht werden:

```
system ("stty raw");
```

Will man zusätzlich verhindern, dass das eingegebene Zeichen vom Betriebssystem geecho't wird, kann man den Aufruf erweitern:

```
system ("stty raw -echo");
```

Danach wird das eingegebene Zeichen auch nicht mehr automatisch auf dem Bildschirm geecho't, sondern erst durch die Ausgabe in Ihrem Programm. Auf diese Weise lassen sich beispielsweise auch Passworteingaben realisieren oder es kann verhindert werden, dass die Eingabe eine Bildschirmmaske verunstaltet. Der Funktionsaufruf `fflush(stdout)`; nach jeder Ausgabe kann dann auch entfallen. Bevor das Programm beendet wird, müssen Sie jedoch wieder in den "Normalmodus" zurückschalten:

```
system ("stty -raw");  
bzw.  
system ("stty -raw echo");
```

Weitere Dateifunktionen für Textdateien

(neben der bereits besprochenen Funktionen `fscanf` und `fprintf`)

- `int putc(int c, FILE *f)` und `int fputc(int c, FILE *f)`
Schreiben ein Zeichen in die Datei. Sie arbeiten wie `putchar()`. `putc()` kann ein Makro sein.
- `int getc(FILE *f)` und `int fgetc(FILE *f)`
Lesen ein Zeichen aus der Datei. Sie arbeiten wie `getchar()`. (`getc()` kann Makro sein)
- `int puts(const char *s)` und `int fputs(const char *s, FILE *f)`
Schreiben eine Zeichenkette in die Datei. `s` ist ein Zeiger (Verweis) auf die Zeichenkette. Die Funktionen liefern `ok (0)` oder `EOF`. `puts()` ergänzt die Zeichenkette um ein Zeilenende-Zeichen (`newline`).
- `char *gets(char *s)`
Liest eine Zeichenkette aus der Datei. Die Funktion liefert entweder die Zeichenkette `s` oder `NULL`, falls das Dateiende erreicht wurde. Bei einem Lesefehler wird ebenfalls `NULL` zurückgegeben. `gets()` legt statt des Newline-Zeichens (`\n`) ein Nullbyte am Stringende ab.
- `char *fgets(char *s, int n, FILE *f)`
Liest eine Zeichenkette aus der Datei. Die Funktion liefert entweder die Zeichenkette `s` oder `NULL`, falls das Dateiende erreicht wurde. Es werden jedoch maximal `n-1` Zeichen gelesen. `fgets()` legt ein Nullbyte am Stringende ab.
- `int ungetc(int c, FILE *f)`
Diese Funktion stellt das Zeichen `c` in den Eingabepuffer zurück. Es kann dann mit dem nächsten `getchar()`, `getc()` oder `fgetc()` wieder gelesen werden. `ungetc()` liefert entweder das Zeichen `c` oder `EOF` als Status zurück.

Beispiel: Das folgende Programm macht in Textdateien solche Zeichen sichtbar, die normalerweise nicht sichtbar sind, nämlich Steuerzeichen mit den ASCII-Codes 0 bis 31. Diese Zeichen werden hexadezimal ausgegeben und als Markierung ein `'\'` davor, so würde das Formfeed-Zeichen als `'\0xC'` ausgegeben. Die Zeichen "Tabulator", "Newline" und das Leerzeichen bleiben unverändert, die Zeilenstruktur der Datei bleibt also erhalten. Auf der Kommandozeile können beliebig viele Dateinamen angegeben werden. Bleibt die Kommandozeile leer, wird automatisch von der Standardeingabe gelesen.

```
#include <stdio.h>
#include <stdlib.h>    /* Fuer exit() */

void vis(FILE *fp);

int main(int argc, char *argv[])
{
    int i;          /* Zaehler fuer Parameter */
    FILE *fp;       /* Eingabedatei */

    if (argc == 1) /* keine Datei angegeben */
        vis(stdin);
    else           /* Dateiliste abarbeiten */
    {
        for (i = 1; i < argc; i++)
            if ((fp=fopen(argv[i], "r")) == NULL)
            {
                fprintf(stderr, "Can't open %s\n", argv[1]);
                exit(1);
            }
        else
        {
            vis(fp);
            fclose(fp);
        }
    }
}
```

```
    }  
  }  
  return 0;  
}  
  
void vis(FILE *fp)  
{  
    int c;  
    while ((c = getc(fp)) != EOF)  
        if (isascii(c) &&  
            (isprint(c) || c=='\n' || c=='\t' || c==' '))  
            putchar(c);  
        else  
            printf("\\%02x", c);  
}
```

Als nächstes schreiben wir ein Programm, das den Mittelwert aus den eingelesenen Zahlen bildet:

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char *argv[])  
{  
    double x; /* Eingabe-Werte */  
    int r; /* Rueckgabewert von fscanf */  
    int n; /* Anzahl der Werte */  
    double sx; /* in sx werden die eingelesenen Werte aufsummiert */  
    double mx; /* Mittelwert der x-Werte */  
    int i; /* Zaehler fuer Parameter */  
    FILE *fp; /* Eingabedatei */  
  
    if (argc == 1) /* keine Datei angegeben */  
    {  
        fprintf(stderr, "Keine Datei angegeben!\n");  
    }  
    else /* Dateiliste abarbeiten */  
    {  
        for (i = 1; i < argc; i++)  
            if ((fp=fopen(argv[i], "r")) == NULL)  
            {  
                fprintf(stderr, "Can't open %s\n", argv[i]);  
            }  
            else /* Datei bearbeiten */  
            {  
                n = 0; /* Zaehler und Summe auf Null setzen */  
                sx = 0;  
                while((r = fscanf(fp, "%lf", &x))==1) /* Wert einlesen */  
                {  
                    n = n + 1;  
                    sx = sx + x;  
                }  
                mx = sx/n;  
            }  
    }  
}
```

```
    printf("Datei:  %s\n", argv[i]);  
    printf("    Anzahl der verarbeiteten Werte: %d\n", n);  
    printf("    Mittelwert: %f\n", mx);  
    fclose(fp);  
}  
}  
return 0;  
}
```

Aufgabe: Erweitern Sie das Mittelwert-Programm so, dass gleichzeitig das Minimum und das Maximum der eingelesenen Zahlen ermittelt und ausgegeben wird.

Oft braucht man ein Programm, dass verschiedene Dateien nach einer Zeichenfolge durchsucht. In Linux/UNIX gibt es dazu das recht mächtige Tool `grep` und seine "Verwandten" `egrep` und `fgrep`. Hier soll eine einfache Variante vorgestellt werden, die `mgrep` heie. Das Programm kann diverse Optionen verarbeiten:

- `i` Gro-/Kleinschreibung nicht beachten
- `l` nur Dateinamen ausgeben
- `c` Zeilen zhlen
- `h` kein Dateinamen/Zeilennummer ausgeben
- `n` Zeilennummer ausgeben
- `v` alle Zeilen ausgeben, die nicht bereinstimmen

Die Optionen werden durch ein `'-'`-Zeichen eingeleitet, um sie im Programm vom Suchbegriff und Dateinamen unterscheiden zu knnen, also z. B. `"-i"` oder `"-i -c"`.

Die Kommandozeile hat folgenden Aufbau:

```
mgrep <Optionen> <Suchbegriff> <Dateiname(n)>
```

wobei die Optionen auch fehlen drfen, es werden dann Voreinstellungen verwendet. Der Teil des Programms, der die Kommandozeile auswertet, ist etwa genau so lang, wie der eigentliche Suchteil.

```
#include <stdio.h>  
#include <ctype.h>  
#include <string.h>  
  
#define MAXLINE 1024                /* max. Zeilenlnge */  
#define TRUE  1  
#define FALSE 0  
  
char upper    = FALSE;              /* Option i */  
char count    = FALSE;              /* Option c */  
char fnonly    = FALSE;             /* Option l */  
char linen    = FALSE;              /* Option n */  
char header    = TRUE;              /* Option h */  
char nomatch  = FALSE;              /* Option v */
```

```
void usage(void)
{
    printf("Aufruf: mgrep [<Optionen>] <Suchmuster> "
           "<Datei(en)> \n\n");
    printf("Optionen: i Groß-/Kleinschreibung nicht beachten\n");
    printf("                l nur Dateinamen ausgeben\n");
    printf("                c Zeilen zählen\n");
    printf("                h kein Dateinamen/Zeilennummer ausgeben\n");
    printf("                n Zeilennummer ausgeben\n");
    printf("                v alle Zeilen ausgeben, die nicht "
           "übereinstimmen\n\n");
    exit(1);
}

void delete(char *str, int pos, int n)
/* n Zeichen aus str ab pos löschen */
{
    int i = pos + n;
    while ((str[pos++] = str[i++]) != '\0');
}

void del_spaces(char *line)
/* Führende Leerzeichen (Space und Tab) entfernen */
{
    while (line[0]==' ' || line[0]==0x09) delete(line, 0, 1);
}

FILE *open_file(const char *name)      /* Datei öffnen */
{
    FILE *fp;
    if ((fp=fopen(name,"r")) == 0)
    {
        fprintf(stderr,"Datei %s kann nicht geöffnet werden", name);
        exit(1);
    }
    return fp;
}

char strstrc(const char *str1, const char *str2)
/* Feststellen, ob str2 in str1 enthalten ist          */
/* In Abhängigkeit von upper Groß-/Kleinschreibung    */
/* beachten bzw. nicht beachten                        */
{
    if (upper)
    {
        int i, n, len;
        n = strlen(str2);
        len = strlen(str1) - n;
        for (i = 0; i <= len; i++)
            if (strnicmp(&str1[i], str2, n) == NULL) return TRUE;
    }
    return strstr(str1, str2) != NULL;
}
```



```
int main(int argc, char *argv[])
{
    char line[MAXLINE+1], search[MAXLINE+1], name[300], match;
    int i, lineno, cnt = 0;
    FILE *fp;

    if (argc < 3) usage();          /* falsche Parameteranzahl */

    /* zuerst die Optionen bearbeiten */
    while (argc > 2 && argv[1][0] == '-')
    {
        printf("Argument: %s\n", argv[1]);
        switch (argv[1][1])
        {
            case 'i':
            case 'I': upper = TRUE; break;
            case 'n':
            case 'N': linen = TRUE; break;
            case 'c':
            case 'C': count = TRUE; break;
            case 'l':
            case 'L': fnonly = TRUE; break;
            case 'h':
            case 'H': header = FALSE; break;
            case 'v':
            case 'V': nomatch = TRUE; break;
            default: usage(); exit(1);
        }
        argc--;
        argv++;
    }

    /* jetzt sollte noch Suchbegriff und Dateiname da sein */
    if (argc < 2)
    {
        usage();
        exit(1);
    }

    /* Suchbegriff uebernehmen */
    strcpy(search, argv[1]);
    argc--;
    argv++;

    /* Text in allen Dateien suchen */
    for (i = 1; i < argc; i++)
    {
        lineno=0;
        strcpy(name, argv[i]);
        printf("Durchsuchen: %s\n", name);
        fp = open_file(name);
        while (fgets(line, MAXLINE, fp) != NULL)
```

```
{
    if (((match = strstr(line, search)) == TRUE
        && !nomatch) || (!match && nomatch))
    {
        cnt++;
        if (!count)
        {
            if (header)
            {
                printf("%s ", name);
                if (fnonly)
                {
                    putchar('\n');
                    break;
                }
                if (linen) printf("%04d", linen);
                putchar(':');
            }
            del_spaces(line);
            printf("%s", line);
        }
        linen++;
    }
    fclose(fp);
}
if (count) printf("%d Zeilen gefunden", cnt);
return 0;
}
```

Wenn ASCII-Zeichen auf eine Zeichenkette eingelesen werden, muss gegebenenfalls eine Umwandlung in eine Zahl erfolgen. Die folgende Funktion atof() leistet das Gewünschte:

```
double atof(char s[])
/* Zeichenkette s nach double wandeln */
{
    double val, power;
    int i, sign;
    for (i = 0; s[i] == ' ' || s[i] == '\n' || s[i] == '\t'; i++)
        ; /* Zwischenraum uebergehen */
    sign = 1;
    if (s[i] == '+' || s[i] == '-') /* Vorzeichen */
        if (s[i++] == '-') sign = -1;
    for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + (s[i] - (int)'0');
    if s[i] == '.') i++;
    for (power = 1; s[i] >= '0' && s[i] <= '9'; i++)
    {
        val = 10*val + (s[i] - (int)'0');
        power = power*10;
    }
    return (sign*val/power);
}
```

3.4 Binärdateien

Bei binärer Ein- und Ausgabe auf Dateien werden die Daten nicht in "lesbarer" Form abgelegt, sondern die Interndarstellung der Speicherinhalte wird direkt (byteweise) in die Datei übertragen. Binäres Schreiben einer long-Variablen benötigt also stets vier Byte Speicherplatz, wogegen der erforderliche Speicherplatz bei formatierter Ausgabe von der Größe der Zahl bzw. vom Format abhängt.

Die Funktion `fwrite` schreibt eine angegebene Anzahl von Datenelementen gleicher Größe in eine Datei. übergeben werden muss:

- Die Adresse des ersten Datenelements. Da nicht von vorne herein klar ist, welche Daten geschrieben werden sollen, wird hier ein Zeiger auf `void` übergeben. Der Zeiger auf die aktuell zu schreibenden Daten kann problemlos in einen solchen gewandelt werden.
- Die Größe eines einzelnen Datenelements. Hierzu gibt es den vordefinierten Typ `size_t`, der normalerweise ein vorzeichenloser Ganzzahltyp ist. Die aktuelle Größe wird normalerweise mit dem `sizeof`-Operator ermittelt.
- Die Anzahl der zu schreibenden Datenelemente
- Die Ausgabedatei

`fwrite` wird also definiert als:

```
size_t fwrite(const void *pt, size_t size, size_t n, FILE *f)
```

Die auszugebenden Daten müssen zusammenhängend im Speicher stehen. Dies ist bei Vektoren stets der Fall, ebenso wie bei Speicherplatz, der durch einen einzelnen `malloc()`-Aufruf (siehe Zeiger) zur Verfügung gestellt wurde.

Beispielprogramm für binäres Schreiben:

```
#include <stdio.h>

int main(void)
{
    int feld[] = {3, 91, 2134, 6, 33, 267, 9123, -5, 22, 0};
    FILE *pf;
    int retval=0;
    char dateiname[] = "daten.bin"; /* Dateiname */
    pf = fopen(dateiname, "wb");
    if (pf == NULL)
    {
        printf("Fehler beim Oeffnen der Datei\n");
        retval=1;
    }
    else
    {
        fwrite(feld, sizeof(int), 10, pf);
        /* Das Feld wird komplett auf einmal geschrieben */
        fclose(pf);
    }
    return retval;
}
```

Die Größe der Datei `daten.bin` ist also `10 * sizeof(int)`.

Die Funktion `fread` ist die zu `fwrite` gehörige analoge Einlesefunktion. Ihr Rückgabewert ist die Anzahl der tatsächlich gelesenen Bytes. Diese Zahl kann kleiner sein, als die Zahl der zu lesenden Bytes, wenn das Dateiende vorzeitig erreicht wurde. Sie ist definiert als:

```
int fread(void *ptr, size_t size, size_t n, FILE *f)
```

Das folgende Programm kopiert eine Datei unter Verwendung der beiden Funktionen `fread()` und `fwrite()`. Dabei wird ein `char`-Array als Puffer verwendet. Diese Kopiermethode ist wesentlich schneller als zeichenweises Kopieren. Das Dateiende wird dadurch erkannt, dass `fread()` den Wert 0 zurückliefert. `fwrite()` schreibt genau so viele Bytes, wie von `fread()` gelesen wurden.

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

#define MAXBUF 32768 /* Puffergroesse */

int main(void)
{
    char buffer[MAXBUF];          /* Kopierpuffer */
    FILE *rf, *wf;                /* Lesedatei, Schreibdatei */
    int numread;                  /* Anzahl gelesener Zeichen */
    char readfrom[] = "daten.bin"; /* Dateinamen */
    char writeto[] = "daten.bak";

    rf = fopen(readfrom, "rb");
    if (rf == NULL)
    {
        printf("Fehler beim Oeffnen der Datei\n");
        exit(1);
    }
    wf = fopen(writeto, "wb");
    if (wf == NULL)
    {
        printf("Fehler beim Oeffnen der Datei\n");
        fclose(rf);
        exit(1);
    }
    while((numread = fread(buffer, sizeof(char), MAXBUF, rf))>0)
    {
        fwrite(buffer, sizeof(char), numread, wf);
    }
    fclose(wf);
    fclose(rf);
    return 0;
}
```

Beispiel: Zum Speichern von Daten für das Telefonverzeichnis des Fachbereichs wird die folgende Struktur definiert:

```
struct adresse_t
```

```
{
    char name[30];
    char vorname[30];
}
```

```
int raumnummer;  
int telefon;  
};
```

Gegeben ist eine **Binärdatei** namens "teldat.dat", die Datensätze vom Struktur-Typ "Adresse" enthält. Gesucht wird eine Funktion `tabelle`, welche die Binärdatei zum Lesen öffnet, und alle Datensätze nacheinander als formatierte Tabelle ausdruckt:

```
int tabelle(char dateiname[])  
{  
    FILE * fp;  
    struct adresse_t data;  
  
    fp = fopen(dateiname, "rb");  
    if (fp == NULL)  
        return 1; /* Fehler */  
  
    printf("+-----+-----+-----+\n");  
    printf("| Name                | Raum | Tel. | \n");  
    printf("+-----+-----+-----+\n");  
    while (fread(&data, sizeof(data), 1, fp) == 1)  
    {  
        printf("| %-10s %-24s | %4d | %4d | \n",  
               data.vorname, data.name, data.raumnummer, data.telefon);  
    }  
    printf("+-----+-----+-----+\n");  
    fclose(fp);  
    return 0;  
}
```

Beachten Sie, dass im Gegensatz zum Feld oben, nun der Adressoperator '&' bei `fread` verwendet werden muss.

Die Daten der Datei "teldat.dat" sollen in eine zweite Binärdatei umkopiert werden. Dafür wird die Funktion `kopiere` verwendet, welche die Binärdatei mit den Adressen (also die Quelldatei) zum Lesen und gleichzeitig eine weitere Binärdatei, die Zieldatei, zum Schreiben öffnet. Die zweite Binärdatei soll Datensätze enthalten, die folgender Struktur genügen:

```
struct neueAdresse_t  
{  
    char name[30];  
    int telefon;  
};
```

Aufgabe der Funktion ist es, jeweils einen Datensatz aus der Quelldatei zu lesen, die Inhalte in einen Datensatz der Zieldatei zu übertragen und diesen dann in die Zieldatei zu schreiben:

```
int kopiere(char quelldateiname[], char zieldateiname[])  
{  
    FILE *qfp;  
    FILE *zfp;  
    struct adresse_t data;
```

```
struct neueAdresse_t nData;

qfp = fopen(quelldateiname, "rb");
if (qfp == NULL) return 1;
zfp = fopen(zieldateiname, "wb");
if (zfp == NULL)
{
    fclose(qfp);
    return 1;
}
while (fread(&data, sizeof(data), 1, qfp) == 1)
{
    strcpy(nData.name, data.name);
    nData.telefon = data.telefon;
    fwrite(&nData, sizeof(nData), 1, zfp);
}
fclose(qfp);
fclose(zfp);
return 0;
}
```

3.5 Dateien mit wahlfreiem Zugriff

Auf die einzelnen Datensätze einer Datei kann direkt zugegriffen werden. Dazu stehen folgende Befehle zur Verfügung:

```
int fseek(FILE *f, long offset, int origin)
```

Mit dieser Funktion kann man einen Dateipositionszeiger auf eine bestimmte Position der durch `f` referierten Datei setzen.

Die Funktion positioniert um einen `offset`, der in Bytes gezählt wird. Der Wert `origin` legt fest, worauf sich `offset` bezieht:

- `SEEK_SET` oder 0:
Die neue Position ergibt sich aus Dateianfang + Offset.
- `SEEK_END` oder 1:
Die neue Position ergibt sich aus der aktuellen Position + Offset.
- `SEEK_CUR` oder 2:
Die neue Position ergibt sich aus Dateiende - Offset.

Der Rückgabewert ist 0, wenn die Positionierung erfolgreich war, sonst -1.

Die Funktion `long ftell(FILE *f)` gibt die aktuelle Position in der Datei an, auf die der Dateizeiger weist. Im Fehlerfall liefert `ftell` den Wert -1.

Die Funktion `void rewind(FILE *f)` positioniert auf den Dateianfang und löscht den Fehlerstatus.

Beispiel: Programm mit wahlfreiem Zugriff auf eine Datei

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

```
int main(void)
{
    long pos;
    int count;
    FILE *fp;
    int mode = 0;
    char c;

    fp = fopen("daten.bin", "w+");
    if (fp == NULL)
    {
        printf("Fehler beim Oeffnen der Datei\n");
        exit(1);
    }
    /* Datei beschreiben */
    fputs("abcdefghijklmnopqrstuvwxyz", fp);
    puts("abcdefghijklmnopqrstuvwxyz");
    printf("\n");

    /* Wahlfreier Zugriff auf Datei */
    printf("Eingabe der Position im File (0 bis 25):\n");
    scanf("%ld", &pos);
    fseek(fp, pos, mode);
    pos = ftell(fp);
    printf("Dateiposition ist %ld\n", pos);
    fread(&c, 1, 1, fp);
    printf("\nBuchstabe an dieser Position: %c\n\n", c);
    fclose(fp);
    return 0;
}
```

Beispiel: Hexdump

Der Inhalt einer Datei soll hexadezimal und in ASCII ausgegeben werden. Für die Bildschirmausgabe soll das Programm die Datei blockweise anzeigen, jeweils 16 Zeilen. Der folgende Bildschirmabzug zeigt die Ausgabe:

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000 :	2F	2A	20	48	65	78	64	75	6D	70	20	65	69	6E	65	72	/* Hexdump einer
00000010 :	20	44	61	74	65	69	20	2A	2F	0D	0A	0D	0A	23	69	6E	Datei */....#in
00000020 :	63	6C	75	64	65	20	3C	73	74	64	69	6F	2E	68	3E	0D	clude <stdio.h>.
00000030 :	0A	23	69	6E	63	6C	75	64	65	20	3C	73	74	64	6C	69	.#include <stdli
00000040 :	62	2E	68	3E	0D	0A	0D	0A	23	64	65	66	69	6E	65	20	b.h>....#define
00000050 :	52	45	54	20	31	33	20	20	20	20	20	20	20	20	20	20	RET 13
00000060 :	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	2F	/
00000070 :	2A	20	52	65	74	75	72	6E	2D	54	61	73	74	65	20	2A	* Return-Taste *
00000080 :	2F	0D	0A	23	64	65	66	69	6E	65	20	42	45	45	50	20	/...#define BEEP
00000090 :	70	75	74	63	68	61	72	28	27	5C	30	37	27	29	20	20	putchar('\07')
000000A0 :	20	20	20	20	20	20	20	20	20	20	2F	2A	20	53	69	67	/* Sig
000000B0 :	6E	61	6C	74	6F	6E	20	61	75	73	67	65	62	65	6E	20	nalton ausgeben
000000C0 :	2A	2F	0D	0A	0D	0A	0D	0A	2F	2A	20	67	6C	6F	62	61	*/...../* globa
000000D0 :	6C	65	20	56	61	72	69	61	62	6C	65	20	2A	2F	0D	0A	le Variable */..
000000E0 :	46	49	4C	45	20	2A	65	69	6E	3B	20	20	20	20	20	20	FILE *ein;
000000F0 :	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	

Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
---------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	------------------

Anmerkungen zum Programm:

- Da immer Blöcke von 16 Zeilen zu je 16 Zeichen dargestellt werden und in jeder Zeile zweimal auf die Daten zugegriffen wird, ist es sinnvoll, zumindest eine Zeile komplett in einen Puffer (Feld) einzulesen. Weil seitenweise geblättert werden soll, ist es sogar ratsam, eine komplette Bildschirmseite ($16 \times 16 = 256$ Bytes) auf einmal von der Datei in den Puffer zu lesen und dann darzustellen.
- Da es unwahrscheinlich ist, dass eine Datei genau ein Vielfaches von 256 Bytes lang ist, muss am Dateiende der restliche Puffer mit einem Wert gefüllt werden, der anzeigt, dass die Daten nicht mehr zur Datei gehören. Vorschlag: Wert der Datenbytes = 0 bis 255, Füller = -1). Die Datei muss natürlich Byte für Byte gelesen werden.
- Da vorwärts und rückwärts geblättert werden soll, muss wahlfrei auf die Datei zugegriffen werden. Dazu verwendet man eine Long-Variable, die die aktuelle Leseposition (erstes zu lesendes Byte) speichert und die dann um 256 incrementiert oder decrementiert wird. Beim vor- und zurückblättern muss man natürlich auch auf Dateianfang oder -ende testen (`fEOF()`). Das Positionieren innerhalb der Datei erfolgt mit der Funktion `fseek()`.
- Hexzahlen sollen mit führenden Nullen ausgegeben werden. Bei der ASCII-Darstellung sollen Zeichen mit den Codes 0 bis 31 als Punkt dargestellt werden.

Das Programm sieht dann so aus (aus Gründen der Übersichtlichkeit wurde auf ungepuffertes Einlesen verzichtet).

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

#define RET 13                                /* Return-Taste */
#define BEEP putchar('\07')                  /* Signalton ausgeben */

/* globale Variablen */
FILE *ein;                                  /* Eingabedatei */
char name[4096];                             /* Dateiname */
int ende = 0;                               /* Dateiende erreicht? */
long position;                              /* aktuelle Position in 'ein' */
int puffer[16][16];                         /* Puffer fuer eine Seite */

/* Funktions-Deklarationen */
void seitelesen(long position);              /* Puffer aus Datei lesen */
void ausgabe(long position);                /* Puffer darstellen */

/* Hauptprogramm */
int main(int argc, char *argv[])
{
    int command, quit = 0;                  /* Bediener-Kommando */

    if (argc < 2)                          /* Datei ueber Kommandozeile? */
    {
        printf("Dateiname: ");             /* Dann ueber Tastatur */
        scanf("%s", &name);
    }
    else
        strcpy(name, argv[1]);
    ein = fopen(name, "rb");                /* Eingabe eroeffnen */
```



```
if (ein == NULL)
{
    printf("Eingabedatei kann nicht eroeffnet werden\n");
    exit(1);
}

position = 0; /* Dateianfang */
for(;quit==0;) /* Hauptschleife */
{
    seitelesen(position);
    ausgabe(position);
    command = getchar(); /* Befehlstaste lesen */
    switch (command)
    {
        case '+' : if (feof(ein) == 0) /* naechster Block */
                    position = position + 256;
                    else BEEP;
                    break;

        case 'b' : /* vorhergehender Block */
        case 'B' :
        case '-' : if (position >= 256)
                    position = position - 256;
                    else BEEP;
                    break;

        case 'Q' : /* Beenden */
        case 'q' : quit=1;
                    break;

        default : BEEP; /* alle anderen Tasten */
    } /* end switch */
} /* end for */
fclose(ein);
return 0;
}

/* Funktionsdefinitionen */
void seitelesen(long position) /* Puffer aus Datei lesen */
{
    int i, j; /* Feldindexe */
    int zeichen; /* gelesendes Zeichen */

    fseek(ein, position, SEEK_SET); /* Pos. ab Dateianfang */
    i = j = 0;
    do
    {
        /* zeichenweise einlesen */
        zeichen = getc(ein);
        puffer[i][j] = zeichen;
        j++;
        if (j == 16) (i++, j = 0);
    } /* bis Puffer voll oder EOF */

    while (i*16+j < 256 && zeichen != EOF);
    while (i*16+j < 256) /* Rest-Puffer auffuellen */
}
```

```

{
    puffer[i][j] = -1;
    j++;
    if (j == 16) (i++, j = 0);
}
}

void ausgabe(long position)                /* Puffer darstellen */
{
    int i, j;                             /* Feldindexe */

    printf("Dump von   *** %s ***   \n",name);
    printf("\n");
    printf("Adresse      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F");
    printf("      0123456789ABCDEF\n");
    printf("\n");
    for (i = 0; i < 16; i++)                /* 16 Zeilen */
    {
        printf("%08lX : ",position);
        for (j = 0; j < 16; j++)            /* Inhalt hexadezimal */
            if (puffer[i][j] < 0)
                printf(" ");
            else
                printf(" %02X",puffer[i][j]);
        printf(" ");
        for (j = 0; j < 16; j++)            /* Inhalt als ASCII-Zeichen */
            if (puffer[i][j] > 31 && puffer[i][j] != 127)
                printf("%c",puffer[i][j]);
            else
                if (puffer[i][j] < 0)        /* Datei zuende - Blanks */
                    printf(" ");
                else
                    printf(".");            /* nicht druckbares Zeichen */
        printf("\n");
        position = position + 16;
    }
    printf("\n");
    printf("Adresse      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F");
    printf("      0123456789ABCDEF\n");
    printf("\n");
    printf("Weiter mit <+>          Zurueck mit <B>/<->");
    printf("          Abbruch mit <Q>\n");
}

```

3.6 Weitere Dateifunktionen (tabellarisch)

- FILE *freopen(const char *filename, const char *mode, FILE *stream)
liefert stream oder NULL. Sie wird oft verwendet zum Umleiten von stdin, stdout oder stderr.
- Für die Fehlerüberprüfung während der Dateibearbeitung stellt ANSI-C folgende Funktionen zur Verfügung:
 - int ferror(FILE *f)
liefert bei einem Fehler den Fehlercode oder 0, wenn kein Fehler auftrat.

- o `void perror(const char *s)`
gibt eine Fehlermeldung aus. Der Aufruf hat die gleiche Wirkung wie:
`printf("%s: %s\n", s, strerror(ferror()))`.
- o `void clearerr(FILE *f)`
löscht den Fehlerstatus.

3.7 Dateiverwaltungsfunktionen

Dateien werden mithilfe eines Dateinamens auf einem persistenten Speichermedium abgelegt. Abgesehen von den bereits besprochenen Funktionen wie `fopen`, `fclose`, `freopen`, ... stehen u. a. auch folgende Dateiverwaltungsoperationen möglich:

- `int remove(const char *filename)`
Löscht eine Datei. Die Funktion liefert 0 oder einen Fehlercode.
- `int rename(const char *oldname, const char *newname)`
Benennt eine Datei um. Die Funktion liefert 0 oder einen Fehlercode.

Diese Funktionen dürfen **nicht** auf geöffnete Dateien angewendet werden.

Die meisten Betriebssysteme unterstützen auch die hierarchische Ablage von Dateien in Verzeichnissen. ANSI-C kann aber nicht davon ausgehen, dass Dateien über Verzeichnisse organisiert werden. Aus diesem Grund sind die nachfolgenden Funktionen nicht in der Norm spezifiziert und können deshalb auch betriebssystemabhängig sein.

Nachfolgend wird die Verzeichnisverwaltung am Beispiel von Linux/UNIX dargestellt.

Um ein Verzeichnis auszulesen, wird es zunächst geöffnet (`opendir`), danach werden die Einträge gelesen (`readdir`) und schließlich wird es wieder geschlossen (`closedir`). Analog zum Dateihandle gibt es ein Verzeichnishandle, das einen Zeiger auf den Datentyp `DIR` ist. Informationen über den Eintrag liefert die von `readdir` gelieferte Struktur `dirent`.

```
#include <sys/types.h>    /* manchmal benötigt */
#include <dirent.h>       /* Headerfile fuer die
                           Verzeichnis-Funktionen */
```

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

`opendir()`

Die Funktion `opendir()` erhält als Parameter den Namen des Verzeichnisses als Zeichenkette. Der Rückgabewert ist ein Zeiger auf das Verzeichnishandle. Ein Fehler wird dadurch angezeigt, dass dieser Zeiger den Wert `NULL` hat.

`readdir()`

Die Funktion `readdir()` liest den nächsten Eintrag im Verzeichnis und erhält als Rückgabewert einen Zeiger auf eine Struktur `dirent`. Dieser Zeiger verweist auf eine statische Variable, die nur bis zum nächsten `readdir()` gültig ist (wird jedesmal überschrieben). Unterschiedliche `DIR`-Handles haben aber unterschiedliche Variablen. Die Variable hat unter UNIX folgende Struktur (bei anderen Betriebssystemen kann auch die Struktur anders sein):

```
struct dirent
{
    long            d_ino;           /* Inode Nummer */
    off_t           d_off;          /* Offset zum naechsten dirent */
    unsigned short  d_reclen;       /* Laenge dieses Eintrags */
    char            d_name[NAME_MAX+1]; /* Dateiname */
};
```

Für das Anwenderprogramm ist meist nur der Name des Eintrags interessant. Will man mehr über diesen Eintrag erfahren, beispielsweise, ob es wieder ein Verzeichnis ist, verwendet man andere Systemaufrufe, z. B. `stat()`.

closedir()

Zuletzt wird das Verzeichnis mit `closedir()` wieder geschlossen. Ein Beispielprogramm für das Auslesen eines Verzeichnisses sieht so aus:

```
#include <sys/types.h>
#include <dirent.h>

int main(void)
{
    DIR *dirHandle;
    struct dirent * dirEntry;

    dirHandle = opendir("."); /* oeffne aktuelles Verzeichnis */
    if (dirHandle != NULL)
    {
        while ((dirEntry = readdir(dirHandle)) != NULL)
        {
            puts(dirEntry->d_name);
        }
        closedir(dirHandle);
    }
    return 0;
}
```

Wenn man mit den Dateinamen noch weiter arbeiten will, kann der jeweilige Name auch in eine eigene Variable übertragen werden. Die Länge des Arrays passen Sie am besten an die Größe an, die das Betriebssystem vorgibt. Sie ist als Konstante in `types.h` vorgegeben. Eine Funktion zum Lesen eines Verzeichnisses könnte dann folgendermaßen aussehen:

```
void read_dir(const char *dirname)
{
    DIR *dir;
    struct dirent *dirzeiger;
    char dateiname[MAXNAMELEN+1]; /* MAXNAMELEN definiert in
                                   types.h */

    printf("%s\n", dirname);

    dir = opendir(dirname);
    if(dir != NULL)
    {
        while((dirzeiger = readdir(dir)) != NULL)
        {
            strcpy(dateiname, (*dirzeiger).d_name);
        }
    }
}
```

```
    if (dateiname[0] != '.') printf("%s\n", dateiname);  
}  
closedir(dir);  
}  
}
```

rewinddir()

Diese Funktion setzt den Lesezeiger wieder auf den Anfang des Verzeichnisses. Die Syntax des Befehls lautet:
`void rewinddir(DIR *dir);`

getcwd()

Die Funktion `getcwd()` ermittelt das aktuelle Arbeitsverzeichnis. Dazu hat das aufrufende Programm einen Puffer für den Namen zur Verfügung zu stellen, der groß genug ist. Die Größe wird als weiterer Parameter übergeben. Reicht dieser Speicher nicht aus, gibt `getcwd()` `NULL` zurück.

`#include <unistd.h>`

```
char *getcwd(char *namebuffer, size_t size);
```

In einigen Systemen ist es zulässig, `NULL` als Parameter für `namebuffer` zu übergeben. Dann alloziert `getcwd()` selbst den benötigten Speicher und gibt den Zeiger darauf zurück. Die Anwendung muss dann durch einen Aufruf von `free()` den Speicher wieder freigeben.

chdir()

Mit der Funktion `chdir()` wird das aktuelle Arbeitsverzeichnis gewechselt.

`#include <unistd.h>`

```
int chdir(const char *Pfad);
```

Bei Erfolg gibt die Funktion 0, sonst -1 zurück. Die Fehlernummer findet sich in der globalen Variablen `errno`.

Anlegen und Löschen von Verzeichnissen: mkdir(), rmdir()

Die Funktionen zum Anlegen und Löschen der Verzeichnisse heißen wie die entsprechenden UNIX-Befehle. Beim Anlegen werden Zugriffsrechte übergeben. Wie das UNIX-Kommando `rmdir` kann auch die Funktion nur leere Verzeichnisse löschen.

`#include <fcntl.h>`

`#include <unistd.h>`

```
int mkdir(const char *Pfadname, mode_t mode);
```

```
int rmdir(const char *Pfadname);
```

Bei Erfolg geben die Funktionen 0, ansonsten -1 zurück. Die Fehlernummer findet sich in der Variablen `errno`.

3.8 Pipes und Named Pipes

Für ein Betriebssystem ist es wichtig, dass mehrere Prozesse miteinander kommunizieren können. Im Laufe der Zeit haben sich zahlreiche Verfahren dafür entwickelt, wobei die Pipe sich sehr einfach anwenden lässt. Gerade bei Unix oder Linux spielen Pipes eine wichtige Rolle.

Auch hier handelt es sich um einen Mechanismus, der betriebssystemspezifisch ist und deshalb nicht in der ANSI-C Norm festgelegt wurde. Somit stehen die nachfolgend beschriebenen Funktionen nicht alle ANSI-C-Compilern zur

Verfügung oder haben einen anderen Namen (bzw. andere Parameter und Rückgabewerte).
Die Mechanismen von Pipes und Named Pipes werden hier anhand von Linux bzw. Unix dargestellt.

Nehmen wir an, Sie wollen ein Programm zur Anzeige der gerade laufenden Prozesse schreiben oder die Ausgabe eines beliebigen anderen Kommandos in Ihrem C-Programm weiterverarbeiten. Dazu wird einfach eine Pipe zwischen dem externen und dem eigenen Programm eingerichtet, wobei das externe Programm in die Pipe schreiben und das eigene Programm die Daten lesen soll.

Dazu wird zuerst eine Pipe und ein Kindprozess erzeugt. Der Kindprozess schließt die nicht verwendete Seite der Pipe (in diesem Fall die Leseseite). Nun überlagert sich der Kindprozess mittels `exec()` mit einer Shell, die ihrerseits das externe Programm startet. Das alles muss man aber nicht selbst erledigen, sondern die Arbeit wird von der Bibliotheksfunktion `popen()` übernommen.

popen()

Öffnen einer Pipe

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

```
FILE* popen(const char* commandline, const char* mode);
```

Der erste Parameter `commandline` ist eine Zeichenkette mit dem gewünschten Kommandoaufruf. Der zweite Parameter `mode` ist entweder "r" für das Lesen aus der Pipe oder "w" für das Schreiben in die Pipe. Bei erfolgreicher Ausführung erhält man einen Dateizeiger zurück, bei einem Fehler wird `NULL` geliefert. Nach Beendigung sollte man die Pipe wieder schließen:

pclose()

Schliessen einer Pipe.

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

```
int pclose(FILE* filep);
```

Das folgende Beispiel zeigt das Abrufen der Uhrzeit:

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

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

```
int main(void)
```

```
{
    FILE* pipe;          /* Filehandle fuer die Pipe */
    char datum[10];      /* Eingabepuffer */

    /* Pipe zum date-Programm lesend oeffnen */
    pipe = popen("date +%H-%M-%S", "r");
    if (pipe!=NULL)
    {
        /* genau 10 Zeichen (hh-mm-ss\n\0) lesen,
           Stringterminator '\0' wird von fgets angehaengt */
        fgets(datum, 10, pipe);

        /* Pipe wieder schliessen */
        pclose(pipe);
        /* irgendwas machen ... */
        printf ("%s\n", datum);
    }
}
```

```
    return 0;
}
```

Solche Pipes erlauben das "Anzapfen" beliebiger Programme um entweder deren Ausgabe im eigenen Programm weiterzuverarbeiten oder um den Input für ein externes Programm zu erzeugen. So kann man beispielsweise mit relativ wenig Aufwand aus den eigenen Daten eine Steuerdatei für das Tool GnuPlot erzeugen und so aus dem eigenen Programm heraus beliebige Zwei- oder Dreidimensionale Grafiken erzeugen.

Die bisher besprochenen Pipes fordern eine "Verwandtschaft" zwischen den jeweiligen Prozessen. Will man eine Verbindung zwischen beliebigen Prozessen herstellen, kann man sich auf "Named Pipes" (FIFOs) stützen → Interprozesskommunikation (IPC). Mit dieser Methode läßt sich auch einfach eine Client/Server-Anwendung realisieren. Named Pipes kann man wie Dateien mit dem Kommando `mkfifo` (oder mit dem allgemeineren Funktionsaufruf `mknod`) in einem beliebigen Verzeichnis anlegen. Über die FIFO läuft dann die Kommunikation zwischen den Prozessen (FIFO bezeichnet übrigens das Arbeitsprinzip: First In, First Out).

Im Gegensatz zu `popen` und `pclose`, die eine `FILE`-Struktur zum Referieren der Pipe verwenden, gehören die Funktionen `mkfifo` (und `mknod`) zu den sog. Low-Level-IO-Funktionen. Aus diesem Grund sollte analog dazu zur Dateibearbeitung ebenfalls Low-Level-IO-Funktionen verwendet werden (d. h. `open` anstelle von `fopen`, `close` anstelle von `fclose`, usw.). Diese Funktionen werden im nachfolgenden Kapitel genauer beschrieben.

Natürlich kann eine FIFO auch über ein C-Programm erzeugt werden. Dazu existiert eine Bibliotheksfunktion:

mkfifo()

Einrichten einer Named Pipe

```
#include <sys/types.h>
#include <sys/stat.h>
```

```
int mkfifo(const char* pathname, mode_t mode);
```

Der erste Parameter gibt Pfad und Namen an, also die Position im Dateisystem, wo die Named Pipe angelegt werden soll. Der zweite Parameter gibt den Erzeugungsmodus an. Hier gibt es sehr viele Möglichkeiten, die im der Manualpage erklärt werden. Das folgende Beispiel legt eine FIFO an, um danach in diese Pipe etwas zu schreiben. Danach werden die Daten wieder aus der Pipe gelesen.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
```

```
int main(void)
{
    int pipe;                /* File-Handle der Pipe */
    char puffer[500];        /* Lese-Puffer */
    char test[] = "Dies ist ein Test\n"; /* Ausgabestring */

    /* Pipe erzeugen Mode = Schreiben/Lesen
       Besitzer darf schreiben und lesen,
```

```
    Gruppenmitglieder und andere nur lesen
*/
mkfifo("fifo01", O_RDWR | S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

/* Pipe als Datei oeffnen */
pipe = open("fifo01", O_RDWR);
if (pipe != -1)          /* konnte die Pipe geoeffnet werden? */
{
    /* irgendetwas (mit NUL-Char) in die Pipe schreiben */
    write (pipe, test, strlen(test)+1);

    /* nun versuchen wir, das wieder zu lesen */
    if (read(pipe, &puffer, sizeof(puffer)) < 0)
        printf("Pipe leer !\n");
    else
        printf("%s\n",puffer);

    /* Pipe schliessen */
    close(pipe);
    /* Die Pipe ist weiter vorhanden! */
}
else
    fprintf(stderr, "Fehler beim Oeffnen der Pipe: %s\n",
            strerror(errno));

return 0;
}
```

Schreibt ein Prozess in eine Pipe, die noch nicht von einem anderen Prozess zum Lesen geöffnet wurde, wird das Signal SIGPIPE generiert.

Das folgende Beispiel zeigt, wie eine Client-Server-Anwendung aussehen könnte. Client und Server werden einfach aus dem obigen Beispiel "herausgezogen". Der Server legt eine Pipe mit Lese/Schreibzugriff an. Danach wartet er auf Daten, die in diese Pipe geschrieben werden, um sie auf der Standardausgabe auszugeben. Der Client darf erst nach dem Server gestartet werden, da die Pipe dann schon existieren muss. Er schreibt in die Pipe und beendet sich dann.

```
/** CLIENT **/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main(void)
{
    int pipe;                /* File-Handle der Pipe */
    char test[] = "Dies ist ein Test\n"; /* Ausgabestring */

    /* Pipe als Datei oeffnen */
    if ((pipe = open("fifo01", O_WRONLY))== -1)
```



```
fprintf(stderr, "Fehler beim Oeffnen der Pipe: %s\n",
        strerror(errno));
else
{
    /* irgendetwas in die Pipe schreiben */
    write (pipe, test, strlen(test)+1);

    /* Pipe schliessen */
    close(pipe);
}
return 0;
}

/**** SERVER ****/
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main(void)
{
    int pipe;                                /* File-Handle der Pipe */
    char puffer[500];                        /* Lese-Puffer */

    /* Pipe erzeugen Mode = Schreiben/Lesen */
    mkfifo("fifo01", O_RDWR | S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

    /* Pipe als Datei oeffnen */
    if ((pipe = open("fifo01", O_RDONLY))==-1)
        perror("Fehler beim Oeffnen der Pipe");
    else
    {
        /* aus der Pipe lesen */
        read(pipe, &puffer, sizeof(puffer));
        printf("Text vom Client: %s\n", puffer);
        /* Pipe schliessen */
        close(pipe);
    }
    return 0;
}
```

3.9 Low-Level-I/O

Die folgenden Funktionen sind nicht Teil des Posix-Standards. Sie sind trotzdem fast überall implementiert, da sie für den Low-Level-Zugriff auf Dateien und Geräte (z. B. die serielle Schnittstelle) notwendig sind. Hier begeben Sie sich ins Reich der Systemprogrammierung. Die fünf elementaren Low-Level-Funktionen sind:

- `open()` - Datei- oder Gerätehandle öffnen

- `close()` - Handle schließen
- `read()` - Datenblock lesen
- `write()` - Datenblock schreiben
- `lseek()` - Positionieren

Das Dateihandle (Dateideskriptor) ist hier nicht wie weiter oben der Pointer auf eine Struktur, sondern eine schlichte ganze Zahl. Die Standardhandles sind 0 (Standardeingabe), 1 (Standardausgabe) und 2 (Standardfehlerausgabe). Als Headerdateien werden benötigt:

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Open und Close

Öffnen einer Datei/eines Gerätes

```
#include <unistd.h>
#include <fcntl.h>
```

```
int open(const char *pfad, int modus);
int open(const char *pfadname, int flags, mode_t zugriffsrechte);
```

```
int close(int handle);
```

Der Funktion werden der Name und der gewünschte Zugriffsmodus übergeben, der optionale dritte Parameter erlaubt das Setzen von Zugriffsrechten. Die Funktion liefert einen Dateideskriptor zurück, der in den weiteren Funktionsaufrufen als Parameter die Datei identifiziert. Die möglichen Zugriffsmodi sind:

<code>O_RDONLY</code>	- nur lesen
<code>O_WRONLY</code>	- nur schreiben
<code>O_RDWR</code>	- lesen und schreiben
<code>O_APPEND</code>	- anhaengen
<code>O_TRUNC</code>	- beim schreibenden Öffnen wird die Dateilänge zu Beginn auf Null gesetzt
<code>O_CREAT</code>	- anlegen, falls noch nicht vorhanden
<code>O_EXCL</code>	- in Kombination mit <code>O_CREAT</code> verhindern des Überschreibens
<code>O_NOCTTY</code>	- kein Kontrollterminal des Prozesses
<code>O_NONBLOCK</code>	- bei Pipe/Gerät sind die I/O-Operationen nicht blockierend (Socket)
<code>O_SYNC</code>	- Schreibvorgang wird direkt ausgeführt
<code>O_BINARY</code>	- Binaerdatei (DOS/Windows)
<code>O_TEXT</code>	- Textdatei (DOS/Windows)

Gegebenenfalls kann man die Werte "verodern" (z. B. `O_WRONLY` | `O_CREAT`).

Beim Erstellen einer Datei müssen ggf. auch Zugriffsrechte spezifiziert werden.

Diese sind bei `open` in Verbindung mit `O_CREAT` oder bei erstellenden Funktionen wie `mkfifo` oder `mknod` anzugeben.

Die Zugriffsrechte beim Erstellen der Datei lauten:

<code>S_IRUSR</code>	0400	Besitzer erhält Leseberechtigung
<code>S_IWUSR</code>	0200	Besitzer erhält Schreibberechtigung
<code>S_IXUSR</code>	0100	Besitzer erhält Ausführungsberechtigung

```
(nur bei ausführbaren Dateien oder
Verzeichnissen interessant)
S_IRWXU  0700  Besitzer erhält alle Berechtigungen
           (entspricht: S_IRUSR | S_IWUSR | S_IXUSR)

S_IRGRP  0040  Gruppenmitglieder erhalten Leseberechtigung
S_IWGRP  0020  Gruppenmitglieder erhalten Schreibberechtigung
S_IXGRP  0010  Gruppenmitglieder erhalten Ausführungsberechtigung
S_IRWXG  0070  Gruppenmitglieder erhalten alle Berechtigungen

S_IROTH  0004  Andere Benutzer erhalten Leseberechtigung
S_IWOTH  0002  Andere Benutzer erhalten Schreibberechtigung
S_IXOTH  0001  Andere Benutzer erhalten Ausführungsberechtigung
S_IRWXO  0007  Andere Benutzer erhalten alle Berechtigung
           (ACHTUNG: ggf. werden damit Sicherheitslücken erzeugt)
```

Sofern Sie mit der oktalen Schreibweise der Zugriffsrechte unter UNIX/Linux vertraut sind, können Sie die Werte der zweiten Spalte (s. oben) verwenden.

Die führende Null nicht vergessen, sonst ist es keine Oktalzahl.

Die Funktion liefert die Handlenummer zurück oder bei Fehler -1.

Die Datei wird mit `close (handle)` wieder geschlossen. Die Funktion liefert 0 zurück oder bei Fehler -1.

Beispiel:

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

int main(void)
{
    int fd;

    fd = open("Testdatei", O_RDONLY); /* Datei muss existieren */
    if (fd == -1)
        perror("Datei existiert nicht.");
    else
        close(fd);
    return 0;
}
```

Der Aufruf von `open` erzeugt keine Datei, falls sie nicht existiert. Das muss der Zugriffsparameter `O_CREAT` regeln, z. B.:

```
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    int fd;

    fd = open("Testdatei", O_WRONLY | O_EXCL | O_CREAT, 0644);
```

```
if (fd!=-1)
    close(fd);
return 0;
}
```

Read und Write

Lesen und Schreiben, sowohl auf geblockte als auch ungeblockte Dateien/Geräte.

```
#include <unistd.h>
#include <fcntl.h>
```

```
int write(int fh, const void *puffer, size_t bytezahl);
int read(int fh, const void *puffer, size_t bytezahl);
```

Transferränge sind immer Bytes. Wichtig ist, dass die Transferränge (bytezahl) nie mehr als die PuffergröÙe betragen darf. Das folgende Beispielprogramm kopiert eine Datei:

```
#include <unistd.h>
#include <fcntl.h>
```

```
int main(void)
{
    int in, out;           /* Filehandles */
    char buffer[1024];     /* Datei-Puffer */
    int count;

    in = open("Quelle",O_RDONLY); /* muss vorhanden und lesbar sein */
    out = open("Ziel", O_WRONLY | O_TRUNC | O_CREAT, 0644);

    if (in!=-1 && out!=-1)
        while((count = read(in, buffer,1024)) > 0)
            write(out, buffer, count);

    if (out!=-1)
        close(out);
    if (in!=-1)
        close(in);
    return 0;
}
```

read() versucht, die angegebene Anzahl Bytes in den Puffer einzulesen, wobei die tatsächliche Anzahl vom Aufruf zurückgeliefert wird. Deshalb sollten stets nur count Bytes weggeschrieben werden. Der letzte Block kann auch weniger als 1024 Bytes enthalten, deshalb kann man nicht einfach immer 1024 wegschreiben.

Lseek

lseek() leistet das Gleiche wie fseek() und dient zum Verschieben des File-Deskriptors innerhalb der geöffneten Datei.

```
#include <unistd.h>
#include <fcntl.h>
```

```
long lseek(int fh, long offset, int mode);
```

Die Datei wird wieder durch den File-Deskriptor `fh` identifiziert, der natürlich zuvor geöffnet bzw. erzeugt wurde. Um wieviele Bytes der File-Deskriptor von der Position wie verschoben werden soll, wird mit `offset` angegeben.

Die Angaben von `mode` sind dieselben wie bei `fseek()`:

`SEEK_SET` oder 0 – vom Dateianfang aus um `offset` Bytes verschieben

`SEEK_CUR` oder 1 – von der aktuellen Position um `offset` Bytes verschieben

`SEEK_END` oder 2 – vom Dateiende um `offset` Bytes verschieben

Die Funktion gibt den Wert der aktuellen Position des File-Deskriptors zurück. Bei einem Fehler gibt diese Funktion –1 zurück. Es sollte keinesfalls auf "`< 0`" geprüft werden, sondern auf `-1`, da es möglicherweise Gerätedateien gibt, die einen negativen Wert zurückliefern. Beispiele:

```
long akt_pos;
```

```
akt_pos = lseek(fh, 0L, SEEK_CUR);
```

Deskriptor auf den Dateianfang setzen:

```
lseek(fh, 0L, SEEK_SET);
```

Deskriptor um 222 Bytes nach vorn versetzen:

```
lseek(fh, 222L, SEEK_CUR);
```

Deskriptor um 100 Bytes zurücksetzen:

```
lseek(fh, -100L, SEEK_CUR);
```

Oft wird von einem Gerät mittels `read()` gelesen (serielle Schnittstelle, USB-Schnittstelle, I/O-Port bei Embedded Systemen etc.), aber man will die Daten zeilenweise weiterverarbeiten. Der Puffer enthält natürlich irgendwo Newline-Zeichen, man muss sie nur Suchen. Aus diesem Grund soll hier das Splitten des Puffers in einzelne Zeilen etwas näher betrachtet werden. Normalerweise wäre es ein Zufall, wenn ein Zeilenende genau mit dem Pufferende zusammenfällt, und man den String per Standardfunktion splitten kann. Die Regel ist, dass ein Puffer den Anfang und der nächste Puffer das Ende einer Zeile enthält. Darum soll jetzt die Arbeit mit mehreren aufeinanderfolgenden Datenblöcken näher betrachtet werden:

Die erste Möglichkeit, die sich anbietet, ist das zeichenweise Lesen von der Netzwerk-Schnittstelle (das Betriebssystem puffert ja die Pakete für uns). Sobald dann ein Newline (`'\n'`) auftritt, wird die Funktion verlassen und gibt den String zurück. Es versteht sich von selbst, dass vom aufrufenden Programm ein Zeichen-Array zur Verfügung gestellt werden muss und dass man dessen Länge nicht überschreitet. Die folgende Funktion `get_line()` liest von einer vorhergeöffneten Datei genau eine Zeile ein (der Unterstrich in `get_line()` ist notwendig, weil `getline()` eine Bibliotheksfunktion ist). Als Parameter werden neben dem Datei-Descriptor das Array und dessen maximale Länge übergeben:

```
int get_line(int fd, char *buffer, unsigned int len)
{
    /* read a '\n' terminated line from fd into buffer
     * of size len. The line in the buffer is terminated
     * with '\0'. It returns -1 in case of error and -2 if the
     * capacity of the buffer is exceeded.
     * It returns 0 if EOF is encountered before reading '\n'.
     */
    int numbytes = 0;
    int ret;
    char buf;

    buf = '\0';
    while ((numbytes < len) && (buf != '\n'))
    {
        ret = read(fd, &buf, 1);    /* read a single byte */
        if (ret == 0) break;        /* nothing more to read */
        if (ret < 0) return -1;    /* error or disconnect */
        buffer[numbytes] = buf;    /* store byte */
        numbytes++;
    }
}
```

```
    numbytes++;
}
if (buf != '\n')
    return -2; /* numbytes > len */
buffer[numbytes-1] = '\0'; /* overwrite '\n' */
return numbytes;
}
```

Nachteil dieser Lösung ist die Geschwindigkeit bzw. deren Fehlen. Durch die vielen `read()`-Aufrufe ist die Funktion ziemlich langsam. Besser wäre eine Lösung, bei der ein Datenpaket komplett eingelesen und dann Zeile für Zeile ans aufrufende Programm weitergereicht wird. Genau das macht die folgende Funktion, bei der die Parameter die gleiche Aufgabe haben wie oben. Diese Funktion hat einen internen Puffer, der mittels `read()` gefüllt wird und dessen Inhalt Stück für Stück bei jedem Aufruf weitergegeben wird. Dazu verwendet die Funktion die statischen Variablen `bufptr`, `count` und `mybuf`, deren Werte erhalten bleiben und bei jedem Aufruf wieder zur Verfügung stehen. Werden mit `read()` mehrere Zeilen gelesen, bleibt der jeweilige Rest in `mybuf` erhalten und wird beim nächsten Aufruf der Funktion verarbeitet:

```
int readline(int fd, char *buffer, unsigned int len)
{
    /* read a '\n' terminated line from fd into buffer
    * bufptr of size len. The line in the buffer is terminated
    * with '\0'. It returns -1 in case of error or -2 if the
    * capacity of the buffer is exceeded.
    * It returns 0 if EOF is encountered before reading '\n'.
    * Notice also that this routine reads up to '\n' and overwrites
    * it with '\0'. Thus if the line is really terminated with
    * "\r\n", the '\r' will remain unchanged.
    */
    static char *bufptr;
    static int count = 0;
    static char mybuf[1500];
    char *bufx = buffer;
    char c;

    while (--len > 0) /* repeat until end of line */
    { /* or end of external buffer */
        count--;
        if (count <= 0) /* internal buffer empty -->
                        read data */
        {
            count = read(fd, mybuf, sizeof(mybuf));
            if (count < 0) return -1; /* error or disconnect */
            if (count == 0) return 0; /* nothing to read - so reset */
            bufptr = mybuf; /* internal buffer pointer */
        }
        c = *bufptr++; /* get c from internal buffer */
        if (c == '\n')
        {
            *buffer = '\0'; /* terminate string and exit */
            return buffer - bufx;
        }
        else
        {
            *buffer++ = c; /* put c into external buffer */
        }
    }
}
```

ursprünglich [Programmieren in C](#)
von Prof. Jürgen Plate

```
    }  
    return -2;                                /* external buffer to short */  
}
```

Beim Senden von Daten sollte man eigentlich nicht zwischenpuffern, sondern jede Zeile sofort auf die Reise schicken
- schließlich wartet der Empfänger darauf.

4 Strukturen

Strukturen

Während Vektoren eine Zusammenfassung von Objekten gleichen Typs sind, handelt es sich bei einer Struktur um eine Zusammenfassung von Objekten möglicherweise verschiedenen Typs zu einer Einheit. Die Verwendung von Strukturen bietet Vorteile bei der Organisation komplexer Daten. Beispiele sind Personaldaten (Name, Adresse, Gehalt, Steuerklasse, usw.) oder Studentendaten (Name, Adresse, Studienfach, Note).

Strukturen werden mit Hilfe des Schlüsselwortes `struct` vereinbart

```
struct Strukturname { Komponente(n) } Strukturvariable(n) Init. ;
```

Strukturname ist optional und kann nach seiner Definition für die Form (den Datentyp) dieser speziellen Struktur verwendet werden, d. h. als Abkürzung für die Angaben in den geschweiften Klammern. Strukturkomponenten werden wie normale Variable vereinbart. Struktur- und Komponentennamen können mit anderen Variablennamen identisch sein ohne dass Konflikte auftreten, da sie vom Compiler in einer separaten Tabelle geführt werden.

Der Aufruf der einzelnen Elemente erfolgt dann nicht über Indizes, sondern über deren Namen. Beispiel (für Definition/Deklaration):

<pre>struct datum { int tag; int monat; int jahr; char mon_name[4]; };</pre>	Legt nur die Form der Struktur datum fest
<pre>struct datum { int tag; int monat; int jahr; char mon_name[4]; } geb_dat, heute;</pre>	Erzeugt zusätzlich die Strukturvariablen geb_dat und heute
<pre>struct point { double spx, spy; int farbe; char label; } spot1;</pre>	point ist der Strukturname, spx, spy, etc. sind Elementnamen und spot1 ist die deklarierte Variable
<pre>struct point punkt1, punkt2;</pre>	ebenfalls so definierte Variablen

Durch die Angabe einer (oder mehrerer) Strukturvariablen wird diese Struktur erzeugt (d. h. Speicherplatz dafür bereitgestellt). Strukturvereinbarungen ohne Angabe einer Strukturvariablen legen nur die Form (den Prototyp) der Struktur fest.

Die geschlossene Initialisierung erfolgt (analog zu den Arrays) bei der Deklaration. Zum Beispiel:

```
struct datum heute = {26, 9, 1987, "jun"};
struct point spot2 = {2.8, -33.7, 15, 'A'};
```


Für den Elementzugriff gibt es zwei eigene Operatoren. Der direkte Zugriff wird dabei mit dem Punktoperator . nach folgendem Schema durchgeführt (Der Operator -> wird bei den Pointern besprochen):

Strukturvariable . Komponente

Beispiel:

```
punkt1.farbe = 11;
punkt2.spy = spot2.spy;
heute.tag = 22;
heute.monat = 1;
heute.jahr = 2000;
```

Strukturvariable können an Funktionen übergeben werden, und Funktionen können Strukturen als Rückgabetyt haben. Beispiel (mit obiger Definition):

```
/* createpoint : bepackt Struktur 'point' */
struct point createpoint(double x, double y, int farbe, char label)
{
    struct point dummy;
    dummy.spx = x;
    dummy.spy = y;
    dummy.farbe = farbe;    /* gleiche Bezeichnungen */
    dummy.label = label;    /* interferieren NICHT */
    return dummy;
}
```

Strukturen können als Elemente ebenfalls wieder Strukturen enthalten (allerdings nicht sich selbst) und Strukturen können zu Vektoren zusammengefaßt werden:

```
struct kunde
{
    char name[NAMLAEL];
    char adresse[ADRLAEL];
    int kund_nr;
    struct datum liefer_dat;
    struct datum rech_dat;
    struct datum bez_dat;
};

struct kunde kunden1, kunden2, ... ;
struct kunde kunden[KUNANZ];
```

Programmbeispiel: Komplexe Arithmetik (typedef siehe unten)

```
#include <stdio.h>

struct complex_t
{
    double r;
    double i;
};

typedef struct complex_t Complex;
```

```
Complex makecomplex(double, double);
Complex sum(Complex, Complex);
Complex product(Complex, Complex);
Complex power(Complex, unsigned);
void compprint(Complex);

int main(void)
/* Berechnet Potenzen komplexer Zahlen */
{
    unsigned k;
    Complex basis, result;
    basis = makecomplex(1,1);
    for (k=0; k < 10; ++k)
    {
        result = power(basis, k);
        printf("%2d    ", k);
        compprint(result);
        putchar('\n');
    }
    return 0;
}

Complex makecomplex(double r, double i)
/* Komplexe Zahl erzeugen */
{
    Complex tmp;
    tmp.r = r; tmp.i = i;
    return tmp;
}

Complex sum(Complex a, Complex b)
/* Summe zweier komplexer Zahlen */
{
    a.r += b.r;
    a.i += b.i;
    return a;
}

Complex product(Complex x, Complex y)
/* Produkt zweier komplexer Zahlen */
{
    Complex u;
    u.r = x.r * y.r - x.i * y.i;
    u.i = x.r * y.i + x.i * y.r;
    return u;
}

Complex power(Complex basis, unsigned expo)
/* Potenz einer komplexen Zahl */
{
    Complex u = {1, 0};
    while (expo > 0)
    {
```

```

if (expo % 2)
{
    expo--;
    u = product(basis, u);
}
else
{
    expo = expo/2;
    basis = product(basis, basis);
}
}
return u;
}

void compprint(Complex z)
/* Druckt eine komplexe Zahl */
{
    if ((z.r != 0) && (z.i != 0))
        printf("%5.2f + %5.2f * i", z.r, z.i);
    else if ((z.r == 0) && (z.i != 0))
        printf("%5.2f * i", z.i);
    else if ((z.r != 0) && (z.i == 0))
        printf("%5.2f", z.r);
    else printf("0");
}

```

Varianter Record: union

Während eine Struktur mehrere Variablen (verschiedenen Typs) enthält, ist eine Variante eine Variable, die (aber natürlich nicht gleichzeitig) Objekte verschiedenen Typs speichern kann. Verschiedene Arten von Datenobjekten können so in einem einzigen Speicherbereich maschinenunabhängig manipuliert werden. Syntaktisch sind `union` und `struct` analog (bis auf Initialisierung). Der wesentliche Unterschied ist, dass eine Variable vom Typ `union` zu einer Zeit immer nur *eines* deren angegebener Elemente enthalten kann. Beispiel:

```

union utype
{
    int n;
    double d;
} irgendwas;

double zahl;

irgendwas.n = 3;
irgendwas.d = 11.7;
zahl = irgendwas.n;      /* in diesem Fall: falscher Wert!
                           Die Bytefolge des double-Werte wird
                           als int-Wert interpretiert.
                           */

```

Allerdings ist Buchführung notwendig, um zu wissen, welcher Datentyp in welcher `union`-Variablen zuletzt abgespeichert wurde (deshalb der obige Fehler). Beispiel:

```

if (utype == INT)
    printf("%d\n", uval.ival);
else if (utype == FLOAT)
    printf("%f\n", uval.fval);

```

```
else if (utype == STRING)
    printf("%s\n", uval.pval);
else
    printf("bad type %d in utype\n", utype);
```

Häufiger ist dagegen die Verwendung zur Neuinterpretation einer maschinenspezifischen Bytesequenz. Dafür werden folgende Datenstrukturen in einer 32-bit Architektur genauer untersucht:

```
struct {
    unsigned int n;    /* 4 Bytes Speicherbedarf */
    char *cp;         /* 4 Bytes Speicherbedarf */
    double wert;      /* 8 Bytes Speicherbedarf */
} svar;
```

Die Variable svar belegt somit 16 Bytes und jede Komponente erhält ihren eigenen Speicher.

```
#include <stdio.h>

int main(void)
{
    unsigned i;
    union {
        unsigned n;          /* 4 Bytes */
        char *cp;            /* 4 Bytes */
        unsigned char bytes[sizeof(char *)]; /* ebenfalls 4 Bytes als
                                                unsigned char-Array */
    } uvar;

    uvar.cp = "Hallo"; /* Startadresse des Strings -> cp */
    printf("\n%s" ab Adresse: %u (dez.)\n", uvar.cp, uvar.n);

    puts("Adresse als Bytesequenz:");
    for (i=0; i<sizeof(uvar.bytes); i++)
        printf("%02x ", uvar.bytes[i]);
    putchar('\n');

    printf("Startadresse in Pointer-Darstellung: %p\n", uvar.cp);
    return 0;
}
```

Der Speicherbedarf von uvar beträgt dagegen 4 Bytes, da sich die Komponenten einen gemeinsamen Speicher teilen. Ändert man die Komponente uvar.cp so ändert sich auch der Wert in uvar.n und die Werte im Array uvar.bytes.

Die einzelnen Komponenten einer Union können durchaus unterschiedlich lang sein. Der von der Union belegte Speicherplatz richtet sich dann nach der Komponente mit dem größten Speicherbedarf.

Ein Beispiel für Struktur- und Variantenvereinbarung ist die Definition der Strukturen WORDREGS und BYTEREGS sowie der Varianten REGS für MS-DOS Funktionsaufrufe:

```
struct WORDREGS {
    unsigned int ax;
```

```
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
    unsigned int si;
    unsigned int di;
    unsigned int cflag;
};

struct BYTEREGS {
    unsigned char al, ah;
    unsigned char bl, bh;
    unsigned char cl, ch;
    unsigned char dl, dh;
};

union REGS {
    struct WORDREGS x;
    struct BYTEREGS h;
};

union REGS inregs, outregs;
inregs.x.bx = 0x12;    /* BX Register auf Hex 12 stellen */
inregs.h.ah = 0x10;    /* AH Register auf Hex 10 stellen */
c = outregs.x.cx;      /* CX Register nach c kopieren */
```

typedef

Mit typedef kann man neue Datentypnamen definieren (Nicht aber neue Datentypen!). typedef ist #define ähnlich, aber weitergehender, da erst vom Compiler verarbeitet und nicht nur einfacher Textersatz. Die Anwendungen von typedef liegen darin, ein Programm gegen Protabilitätsprobleme abzuschirmen und für eine bessere interne Dokumentation zu sorgen. Die Syntax lautet "typedef <bekannter Typ> <neuer Typ>", z.B.:

```
typedef int t_count;
typedef unsigned long int bigint;
```

Auch wenn der neue Typ t_count dem int entspricht, kann er Programme "fehlerbewusster" machen. Definiert man eine Funktion mit einem Parameter von Typ t_count wird - bei entsprechend strikter Compilereinstellung bereits bei der Übersetzung ein Fehler gemeldet, wenn ein int-Parameter übergeben wird.

Durch typedef wird aber auch der Aufbau von Structures verschleiert, so dass einige Programmierer keinen Gebrauch davon machen. Zur Syntax: Der neue Typname steht an genau der Stelle, an der *ohne* typedef der Variablenname stünde. Beispiele:

```
typedef struct verbund
{
    int i;
    float f;
    double df;
} Collect;
```

Collect ist hier also *keine* Strukturvariable, sondern der so neu definierte Typname!

5 Dynamische Datenstrukturen

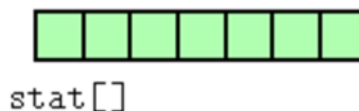
5.1 Dynamische Speicherverwaltung

Bisher waren Datenobjekte (Variable), die in einem Programm definiert werden, statische Objekte. Das heißt, dass der C-Compiler für die Bereitstellung von Speicher für diese Objekte sorgt und dass ihre Anzahl und Größe (also der Speicherbedarf) zum Zeitpunkt der Übersetzung festgelegt sein muss. Bei manchen Algorithmen ergibt sich das Problem, dass die Größe eines Objektes (z. B. Arrays) bzw. die Anzahl der Objekte erst zur Programmlaufzeit angegeben werden kann.

Mit den bisher bekannten Datenstrukturen der Sprache C bleibt nur die Notlösung: Definition statisch angelegter Objekte mit Maximalgröße. Zum einen wird hier oft Speicher verschwendet und zum anderen reicht mitunter der verfügbare Speicher nicht für die Maximalwerte aus und das Programm wird nicht gestartet, obwohl in der aktuellen Situation wesentlich weniger Speicher gebraucht würde. Das Ganze ist also wenig flexibel.

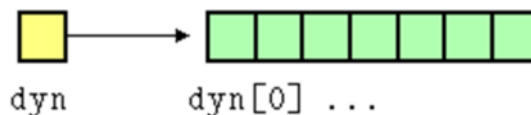
Die Lösung liegt in einer dynamischen Speicherallokation bei Auftreten des Platzbedarfs, d. h. Definition der Objekte zur Laufzeit (→ dynamisch). Diese Lösung ist am Bedarf ausgerichtet und der belegte Platz kann sogar wieder dynamisch freigegeben werden. Dynamisch allokierte Objekte können nicht wie statische Objekte definiert werden. Sie werden auch nicht über einen Namen, sondern nur über ihre Adresse angesprochen. Diese Adresse entsteht bei der Speicherbelegung ("Allokation") und wird normalerweise einer Zeiger-Variablen zugewiesen.

Eine statische Variable, etwa ein Array, hat einfach die Form:



Bei dynamischer Allokation geht man folgendermaßen vor:

- Es wird also zuerst eine Zeigervariable entsprechenden Typs definiert, die aber noch keinen sinnvollen Wert besitzt.
- Dann wird für das Objekt, auf das die Zeigervariable verweisen soll, ausreichend Speicher allokiert und nun der Zeigervariablen die Adresse dieses Speichers zugewiesen.



Es ist wichtig, sich den Unterschied zum äquivalenten statischen Array klarzumachen.

Während *dyn* eine *Variable* ist, deren Speicherplatz sich aus `&dyn` ergibt, ist *stat* eine *konstante Adresse* (→ Zeigerkonstante). Interessant ist die Frage, was denn dann `&stat` ist. Da eine Zeigerkonstante keine Adresse besitzt, stellt diese Darstellung einen Fehler dar. Ältere Compiler ignorieren diesen und setzen `stat` und `&stat` gleich. Andere Compiler (die in älteren Versionen diesen Fehler ignorierten) geben womöglich nur eine Warnung aus.

Die dynamische Speicherbelegung erfolgt mittels spezieller, in der Standardbibliothek enthaltenen, Speicherallokationsfunktionen. `size_t` ist dabei ein maschinenabhängig definierter Datentyp `unsigned int`, definiert in `<stdlib.h>`, `<stdio.h>`, `<stddef.h>`, etc.).

- `void *malloc(size_t size)`
belegt einen `size` Bytes großen zusammenhängenden Bereich und liefert die Anfangsadresse davon zurück.
- `void *calloc(size_t nobj, size_t size)`
liefert die Anfangsadresse zu `nobj*size` Bytes großem Bereich zurück; der Inhalt dieses Bereiches ist mit dem Wert 0 initialisiert.
- `void *realloc(void *ptr, size_t size)`
Veränderung der Größe eines allokierten Speicherblocks.
- `void free(void *)`
wird schließlich verwendet, um nicht mehr benötigten dynamisch belegten Speicherplatz wieder freizugeben. Der NULL-Pointer darf nicht übergeben werden!
- `size_t sizeof(something)`
ist ein Operator, wobei `something` nicht nur eine einfache oder strukturierte Variable sein kann, sondern auch ein Datentyp. Der Operator liefert die Länge des vom Argument belegten Arguments). Es ist ratsam, statt einer konstanten Größe eine Datenelemente anzugeben, die Größe mit `sizeof()` zu bestimmen. Erstens kann man sich nicht vertun und zweitens sind solche Programme portabler. Also z. B. statt `calloc(4, 100)` besser `calloc(sizeof(int), 100)` verwenden.

Die Speicheralkations-Funktionen liefern die Anfangsadresse des allokierten Blocks als void-Pointer (`void *`) es ist daher kein Type-Cast bei Zuweisung an Pointer-Variable erforderlich. Um dem Programm mehr Klarheit zu geben, schadet es aber auch nicht, Type-Cast zu verwenden.

Die Funktionen `malloc` und `calloc` liefern bei Fehler (z. B. wenn der angeforderte Speicherplatz nicht zur Verfügung steht) den Nullpointer `NULL` zurück. Nach jedem Aufruf sollte deshalb deren Rückgabewert getestet werden!

Dies kann in gewissen Anwendungsfällen auch mit `void assert(int);` geschehen.

`assert()` ist ein Makro und benötigt das Headerfile `<assert.h>` und u. U. `<stdio.h>`. Ist das Argument `NULL`, wird das Programm abgebrochen, der Modulname und die Programmzeilennummer des `assert`-Aufrufs ausgegeben.

Da ein Abbruch des Programms manchmal andere Ressourcen (wie z. B. Dateien, Netzverbindungen, o. ä.) nicht korrekt schließt, wird diese Methode oft nur während des Entwicklungsstadiums eines Programms angewendet.

Für `malloc`, `calloc` und `free` wird das Headerfile `<stdlib.h>` oder `<alloc.h>` benötigt.

Der für die dynamische Speicherverwaltung zur Verfügung stehende Speicherbereich wird als **Heap** bezeichnet. Die Lebensdauer dynamisch allokierten Speichers ist nicht an die Ausführungszeit eines Blocks gebunden. Nicht mehr benötigter dynamisch allokiert Speicher ist explizit freizugeben (`free()`). Ein erstes Beispiel:

```
int *ip;
ip = (int *) malloc(n*sizeof(int));
ip zeigt hier auf n Speicherplätze vom Datentyp int. Mit
free(ip);
kann der bereitgestellte Speicherplatz wieder freigegeben werden.
```

Im folgenden Beispielprogramm wird dynamisch Speicherplatz für Objekte bereitgestellt, die einfachen Variablen der Datentypen `int`, `float` und `double` entsprechen. Nach dem Ablegen von eingelesenen Zahlenwerten auf diesen Objekten und dem anschließenden Ausdrucken wird der zugeteilte Speicherplatz wieder freigegeben.

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

```
int main(void)
{
    int *ip; float *fp; double *dp;

    /* Speicher anfordern */
    ip = (int *) malloc (sizeof(int));
    fp = (float *) malloc (sizeof(float));
    dp = (double *) malloc (sizeof(double));

    if (ip==NULL || fp==NULL || dp==NULL)
        fprintf(stderr, "Mind. eine Allokation klappte nicht!");
    else
    {
        printf("Drei Zahlen eingeben:\n");
        scanf("%d %f %lf", ip, fp, dp);
        printf("%d, %f, %f\n", *ip, *fp, *dp);
    }

    /* Speicher freigeben (nur falls nicht NULL im Pointer steht) */
    if (ip!=NULL)
        free(ip);
    if (fp!=NULL)
        free(fp);
    if (dp!=NULL)
        free(dp);
    return 0;
}
```

Das folgende Demonstrationsprogramm zur Verwendung der Bibliotheksfunktionen zur dynamischen Speicherverwaltung liest eine beliebige Anzahl von double-Werten vom Terminal und gibt sie dann wieder aus. Beachten Sie die Verwendung von `sizeof()` und die Fehlerprüfung.

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

int main(void)
{
    int i, anz;
    double *dfeld;

    printf("\nAnzahl der double-Werte ? ");
    scanf("%d", &anz);

    if (anz <= 0)
    {
        printf("\nAnzahl muss >0 sein !\n");
        exit(1);
    }

    if ((dfeld = malloc(anz*sizeof(double))) == NULL)
    {
        printf("\nSpeicherplatz reicht nicht aus !\n");
        exit(1);
    }
}
```



```
}

/* Einlesen der double-Werte */
for (i = 0; i < anz; i++)
{
    printf ("%d. Doublewert bitte:", i+1);
    scanf ("%f", dfeld+i);
}
printf ("\n");

/* Ausgabe der double-Werte */
for (i = 0; i < anz; i++)
    printf (" Doublewert Index: %d  = %f:\n", i, dfeld[i]);

free(dfeld); /* Hier kann dfeld nur ungleich NULL sein */
return 0;
}
```

Ein weiteres Beispiel definiert einen struct-Typ für Angestellte. Die Komponente "chef" ermöglicht es, einen Pointer aufzunehmen, der auf eine entsprechende zweite Struktur verweist, welche die Informationen über den Vorgesetzten des Angestellten enthält. Da der Chef wiederum einen Vorgesetzten haben kann usw., lassen sich so Firmenhierarchien aufbauen.

Es wird im Programm Speicherplatz für die Daten des Angestellten und der beiden übergeordneten Vorgesetzten reserviert. Dabei erfolgt gleichzeitig die Verkettung der Strukturen über ihre chef-Pointer, so dass die gewünschte Hierarchie entsteht.

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

struct mitarbeiter
{
    char  name[20+1];
    int   pers_nr;
    float gehalt;
    struct mitarbeiter *chef;
};

void mem_error(const char *mess);
void out_chefs(struct mitarbeiter *mitarb);

int main(void)
{
    struct mitarbeiter *ang_p, *oberboss_p;
                                /* Zeiger auf die Daten zweier
                                Mitarbeiter */

    ang_p =                      /* Mitarbeiter anlegen */
        (struct mitarbeiter *) malloc(sizeof(struct mitarbeiter));

    if (!ang_p) /* entspricht ang_p==NULL */
```

```
    mem_error("beim Mitarbeiter");
else
{
    (*ang_p).pers_nr = 1234;          /* Entweder so ... */
    ang_p->pers_nr    = 1234;          /* oder so!         */
    strcpy(ang_p->name, "P. Schmidt");
    ang_p->gehalt=2500;

    ang_p->chef =                      /* der direkte Vorgesetzte */
        (struct mitarbeiter *) malloc(sizeof(struct mitarbeiter));

    if (!(ang_p->chef))
        mem_error("beim direkten Vorgesetzten");
    else
    {
        ang_p->chef->pers_nr = 123;
        strcpy(ang_p->chef->name, "T. Huber");
        ((*ang_p).chef)->gehalt=3700;

        oberboss_p=
            (struct mitarbeiter *) malloc(sizeof(struct mitarbeiter));

        ((*ang_p).chef).chef = oberboss_p;
                                /* und dessen Vorgesetzter */

        if (!(ang_p->chef->chef))
            mem_error("beim Oberboss");
        else
        {
            ang_p->chef->chef->pers_nr = 1;
            strcpy(ang_p->chef->chef->name, "A. Obermeier");
            ang_p->chef->chef->gehalt=5700;

            ((*(*ang_p).chef).chef).chef = NULL;
                                /* Oberboss hat keinen
                                   Chef mehr */
        }

        /* lesbarer waere hier: --> */
        if (!oberboss_p)
            mem_error("beim Oberboss");
        else
        {
            oberboss_p->pers_nr = 1;
            strcpy(oberboss_p->name, "A. Obermeier");
            oberboss_p->gehalt=5700;

            oberboss_p->chef = NULL; /* Oberboss hat keinen
                                       Chef mehr */
        }
        /* <-- Ende der Alternativversion */
    }
}
```

```
}

out_chefs(ang_p); /* Chefs von P. Schmidt ausgeben*/
out_chefs(oberboss_p); /* Chefs von A. Obermeier ausg. */
/* hier waere noch die Freigabe des Speichers sinnvoll:
   free_all(ang_p);
   */
return 0;
}

/* Ausgabe eines Speicherfehlers */
void mem_error(const char *mess)
{
    fprintf(stderr, "Speicherfehler %s!\n", mess);
}

/* Ausgabe der Chefs */
void out_chefs(struct mitarbeiter *mitarb)
{
    if (!mitarb) return; /* wird NULL uebergeben, dann raus! */

    printf("Chefs von %s:", mitarb->name);
    if (mitarb->chef==NULL)
        printf(" keiner");
    else
    {
        mitarb=mitarb->chef; /* setze Mitarbeiter-Zeiger auf
                               den Chef */
        while (mitarb != NULL) /* wiederhole solange noch ein Chef
                                vorhanden ist */
        {
            printf(" -> %s", mitarb->name);
            mitarb=mitarb->chef; /* setze Mitarbeiter auf dessen Chef */
        }
    }
    putchar('\n');
}
```

Eine typische Freigabefunktion für diese verkettete Liste könnte wie folgt aussehen:

```
void free_all(struct mitarbeiter *first)
{
    while (first)
    {
        /* Rette zuerst den Zeiger auf den Chef */
        struct mitarbeiter *next=first->chef;
        free(first); /* ab hier werden alle Komponenten ungueltig */
        first=next; /* zeige nun mit first auf den Chef */
    }
}
```

Funktion zum Aneinanderhängen zweier Strings in einem neu bereitzustellenden Speicherbereich. Im Gegensatz zur Bibliotheksfunktion `strcat` ist hier sichergestellt, dass genügend Speicherplatz bereitsteht.

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

char *addstring(const char *s1, const char *s2)
{
    char *as;
    if ((as=malloc(strlen(s1)+strlen(s2)+1)) != NULL)
    {
        strcpy(as, s1);
        strcat(as, s2);
    }
    return as;
}
```

Unterprogramm zum Umwandeln deutscher Umlaute in ihre Ersatzdarstellung. Der Aufrufer ist für die Freigabe des zurückgegebenen Strings verantwortlich:

```
char* umlaut(char* eingabe)
{
    int i = 0, j = 0;
    char *tmp1 = NULL;
    char *tmp2 = NULL;

    /* Wir sparen uns das Ermitteln der Laenge des      */
    /* Eingabestrings und nehmen mal den worst case an */
    tmp1= malloc(2*strlen(eingabe) + 1);
    if(tmp1 != NULL)
    {
        /* Kopieren und Ersetzen */
        while(eingabe[i] != '\0')
        {
            switch (eingabe[i])
            {
                case 'ä': tmp1[j++] = 'a'; tmp1[j++]='e'; break;
                case 'ö': tmp1[j++] = 'o'; tmp1[j++]='e'; break;
                case 'ü': tmp1[j++] = 'u'; tmp1[j++]='e'; break;
                case 'Ä': tmp1[j++] = 'A'; tmp1[j++]='e'; break;
                case 'Ö': tmp1[j++] = 'O'; tmp1[j++]='e'; break;
                case 'Ü': tmp1[j++] = 'U'; tmp1[j++]='e'; break;
                case 'ß': tmp1[j++] = 's'; tmp1[j++]='s'; break;
                default: tmp1[j++] = eingabe[i];
            }
            i++;
        }
        tmp1[j] = '\0';
        /* Nun den String genau passend zurueckgeben */
        tmp2 = malloc(strlen(tmp1) + 1);
        if (tmp2 != NULL)
            strcpy(tmp2, tmp1);
        free(tmp1);
    }
    return tmp2;
}
```

Einlesen einer unbekannten Anzahl von Zeilen (max. Anzahl 100). Nach Abschluss der Eingabe erfolgt Ausgabe der Zeilen in umgekehrter Reihenfolge (letzte Zeile zuerst).

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

#define MAXANZ 100
#define MAXLEN 136

unsigned zeilenEin(char *zeilenFeld[], unsigned max)
{
    char *next, temp[MAXLEN+1];
    unsigned laenge, i=0;
    int ok=1;

    while (ok && fgets(temp, MAXLEN+1, stdin)!=NULL)
    {
        laenge = strlen(temp);
        if (i < max && (next=malloc(laenge+1))!= NULL)
        {
            strcpy(next, temp);
            zeilenFeld[i++] = next;
        }
        else
            ok=0;
    }
    return i;
}

void zeilenAusUndFreigabe(char *zeilenFeld[], unsigned anz)
{
    unsigned i;
    for (i=anz; i>0; i--)
    {
        printf("%s\n", zeilenFeld[i-1]);
        free(zeilenFeld[i-1]);
    }
}

int main(void)
{
    char *zeilen[MAXANZ];
    unsigned anz;

    Anz=zeilenEin(zeilen, MAXANZ);
    printf("\n%u Zeilen gelesen\n\n", anz);
    zeilenAusUndFreigabe(zeilen, Anz);
    return 0;
}
```

Im letzten Beispiel wird dynamisch Speicherplatz für Objekte bereitgestellt, die den Datentyp `double` besitzen und die Form von ein- und zweidimensionalen Feldern haben. Die Größe der Felder, d. h. Zeilenzahl und Spaltenzahl der

Matrix und Anzahl der Komponenten des Arrays werden erst zur Laufzeit des Programms eingelesen. Anschließend werden die eingelesenen Zahlenwerte als Matrixelemente, bzw. Arraykomponenten abgelegt und wieder ausgedruckt. Am Ende wird der bereitgestellte Speicherplatz wieder freigegeben.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    double **a,      /* Eingabematrix */
           *u;       /* Eingabevektor */
    int i,j,n,m;

    printf("Zeilenzahl m und Spaltenzahl n eingeben: ");
    scanf("%d %d", &m, &n);

    /* Speicherplatz bereitstellen */
    a = (double **) malloc (m*sizeof(double *));
    for (i=0; i<m; i++)
        a[i] = (double *) malloc (n*sizeof(double));
    u = (double *) malloc (n*sizeof(double));

    printf("%dx%d Matrix A eingeben\n", m, n);
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            scanf("%lf", &a[i][j]);

    printf("%d Vektor u eingeben\n", n);
    for(j=0; j<n; j++)
        scanf("%lf", &u[j]);

    printf("Eingabedaten: Matrix A\n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
            printf("%f ", a[i][j]);
        printf("\n");
    }

    printf("Eingabedaten: Vektor u\n");
    for(j=0; j<n; j++)
        printf("%f ", u[j]);
    printf("\n");

    /* Speicherplatz freigeben */
    for (i=0; i<m; i++)
        free (a[i]);
    free (a);
    free (u);
    return 0;
}
```

Hier zeigt a auf m Speicherplätze vom Datentyp (double *). Jeder dieser Speicherplätze a[i] ist selbst Zeiger auf n Speicherplätze vom Datentyp double. Ebenso zeigt u auf n Speicherplätze vom Datentyp double.

Zeiger und Strukturen

Im folgenden Beispiel wird eine Struktur für eine Adresse verwendet:

```
struct kunde
{
    int kundennr;
    char vorname[15];
    char nachname[20];
    char strasse [25];
    char plz[5];
    char ort[25];
};
```

Im folgenden Programmbeispiel geht es darum, eine beliebige Zahl von Adressen einzulesen. Zur Speicherung wird kein Array verwendet, sondern ein zusammenhängender Speicherbereich, der dynamisch allokiert wird und über einen Pointer angesprochen werden kann. Dazu wird zuerst gefragt, wieviele Adressen eingegeben werden sollen und dementsprechende Speicherplatz mit `malloc()` belegt.

Bei der Verwendung von Zeigern im Zusammenhang mit Strukturen wird häufig folgender Ausdruck benötigt, um auf eine einzelne Komponente zuzugreifen:

```
(*zeiger).strukturkomponente
```

Dafür gibt es folgende abgekürzte Schreibweise:

```
zeiger -> strukturkomponente
```

Die Zeichen - und > bilden einen Pfeil und bringen so sehr anschaulich das Zeigerkonzept zum Ausdruck.

Da wir nicht nur eine Strukturvariable haben, sondern (implizit) ein ganzes Feld, benötigen wir Schleifen zum Einlesen und Ausdrucken. Der Schleifenindex `i` beginnt bei 0 und wird so lange erhöht, bis die Anzahl der Kunden erreicht ist. Dieser Index muss zum Wert des Zeigers addiert werden, so dass Ausdrücke der Form

```
(zeiger + i) -> strukturkomponente
```

entstehen. Hat `i` beispielsweise den Wert 3, dann wird der Wert `3 * sizeof(struct kunde)` zum Zeiger addiert. Der Zeiger zeigt dadurch auf das drittnächste Element.

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

```
/* Eigene sichere String-Eingabefunktion */
int getstr(char *str, unsigned max)
{
    unsigned i=0;
    int ch, ok=1;
    while ((ch=getchar())!=EOF && ch!='\n')
    {
        if (i<max)
            str[i++]=ch;    /* Nur in String ablegen, falls i<max */
        else
            ok=0;
    }
    str[i]='\0';    /* String-Endekennung sichern */
    return ok;
}
```

```
int main(void)
{
    struct kunde
    {
        int kundennr;
        char vorname[15+1];
        char nachname[20+1];
        char strasse [25+1];
        char plz[5+1];
        char ort[25+1];
    };

    char antwort;
    int i, anzahl;
    struct kunde *person;

    printf("Wie viele Kunden wollen Sie eintragen? ");
    scanf("%d", &anzahl);
    getchar(); /* Liest RETURN-Zeichen aus dem Puffer */

    /* Platz fuer anzahl kunden reservieren */
    person = (struct kunde *) malloc(anzahl * (sizeof(struct kunde)));
    if (person)
    {
        printf("Geben Sie die Kundendaten ein:\n");
        for (i = 0; i < anzahl; i++)
        {
            printf("\nKundennummer: "); scanf("%d",
                                                &(person+i)->kundennr);
            getchar(); /* RETURN entfernen */
            printf("\nVorname.....: "); getstr((person+i)->vorname, 15);
            printf("\nNachname.....: "); getstr((person+i)->nachname, 20);
            printf("\nStrasse: "); getstr((person+i)->strasse, 25);
            printf("\nPostleitzahl: "); getstr((person+i)->plz, 5);
            printf("\nWohnort.....: "); getstr((person+i)->ort, 25);
        }

        printf("\n\nSie haben folgende Daten eingegeben:\n");
        for (i = 0; i < anzahl; i++)
        {
            printf("\nKundennummer: %d\n", (person+i)->kundennr);
            printf("%s %s\n", (person+i)->vorname, (person+i)->nachname);
            printf("%s\n", (person+i)->strasse);
            printf("%s %s\n", (person+i)->plz, (person+i)->ort);
        }
        free(person); /* Speicher freigeben */
    }
    else
        fprintf(stderr, "Kein Speicher verfuegbar.\n");

    return 0;
}
```


Beispiel: Anwendung von Bibliotheksfunktionen. In `time.h` sind Funktionen und Strukturen zur Ermittlung von Datum und Uhrzeit enthalten. Für die Ausgabe braucht man einen Zeiger `tp`, der auf eine in `time.h` definierte Struktur `tm` zeigt.

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

int main(void)
{
    time_t zeitpunkt; /* time_t definiert in time.h */
    struct tm *tp;    /* tm definiert in time.h */

    time(&zeitpunkt); /* Uhrzeit und Datum im internen
                       Format: */
                       /* Sekunden seit 1.1.1970 GMT */
    printf("%ld\n", zeitpunkt); /* so viele sind es */

    tp=localtime(&zeitpunkt); /* umwandeln in etwas Lesbareres */

    printf("Uhrzeit: %02d:%02d:%02d\n",
           tp->tm_hour, tp->tm_min, tp->tm_sec);

    printf("Tag: %02d\n", tp->tm_mday);
    printf("Monat: %02d\n", tp->tm_mon+1);
    printf("Jahr: %04d\n", tp->tm_year+1900);

    return 0;
}
```

Gefahren bei der Rückgabe von Zeigern

Bei Zeigern, die als Rückgabewert einer Funktion dienen, muss darauf geachtet werden, dass sie nicht auf lokale Objekte verweisen, die nach dem Funktionsaufruf verschwunden sind.

Negativ-Beispiel 1: Die lokale Variable existiert nur während des Funktionsaufrufs. Die Ausgabe ist unbestimmt, sie könnte zufällig auch 1234 lauten.

```
int *murks(void)
{
    int x = 1234;

    return &x;
}

int main(void)
{
    int *ip;

    ip = murks();
    printf("%d\n", *ip);
    return 0;
}
```

Negativ-Beispiel 2: Die Funktion soll ein Duplikat des als Parameter übergebenen Strings erzeugen und einen Zeiger darauf zurückgeben. Der Speicherplatz für das lokale Feld neu wird jedoch beim Verlassen der Funktion wieder freigegeben und die Kopie zerstört.

```
char *murks(const char * txt)
{
    char neu[100];
    char *n;

    n = neu;
    while ((*n++ = *txt++) != '\0'); /* kopieren */
    return n;
}

int main(void)
{
    char *strp;

    strp = murks("Autsch!");
    printf("%s\n", strp);
    return 0;
}
```

Besser wäre es, dynamisch Speicher zu allocieren und den Pointer zurückzugeben, dann entspricht die Funktion der Bibliotheksroutine strdup:

```
char *kein_murks(const char * txt)
{
    char *n;
    n = (char *) malloc(strlen(txt) + 1);
    if (n!=NULL)
        while ((*n++ = *txt++) != '\0'); /* kopieren */
    return n;
}

int main(void)
{
    char * strp;

    strp = kein_murks("So geht es!");
    if (strp) /* entspricht strp!=NULL */
    {
        printf("%s\n", strp);
        free(strp);
    }
    return 0;
}
```

5.2 Dynamische Datenstrukturen

Dynamische Datenstrukturen ändern ihre Struktur und den von ihnen belegten Speicherplatz während der Programmausführung. Sie sind aus einzelnen Elementen, den sogenannten 'Knoten', aufgebaut, zwischen denen

üblicherweise eine bestimmte Nachbarschafts-Beziehung (Verweise) besteht. Die Dynamik liegt im Einfügen neuer Knoten, Entfernen vorhandener Knoten und Änderung der Nachbarschaftsbeziehung. Der Speicherplatz für einen Knoten wird erst bei Bedarf zur Programmlaufzeit allokiert. Es handelt sich hierbei um Strukturen, die als 'listen' oder 'Bäume' bezeichnet werden.

Weiterhin ist der Speicherort der einzelnen Knoten im Voraus nicht bekannt (und interessiert auch nicht. "Benachbarte" Knoten, d. h. Knoten, die logisch nebeneinander angeordnet sind, liegen in der Regel weit auseinander. Die Beziehung der einzelnen Knoten untereinander wird sinnvollerweise über Zeiger hergestellt, die jeweils auf den Speicherort des "logischen Nachbarn" zeigen. Jeder Knoten wird daher neben den jeweils zu speichernden Nutzdaten mindestens einen Zeiger auf die jeweiligen "Nachbar"-Knoten enthalten. Man nenn solche Strukturen daher auch "verkettete Datenstrukturen" oder auch "selbstbezügliche (rekursive) Datenstrukturen". Die wichtigsten dieser Datenstrukturen sind:

- Lineare Listen
 - einfach verkettete Listen
 - doppelt verkettete Listen
 - Spezialformen (bezüglich der Anwendung)
- Neben der Art und Anzahl der Verkettung kann noch die Speicherungsverfahren/-strategie festgelegt werden:
- Queue (Pufferspeicher, FIFO)
 - Stack (Kellerspeicher, LIFO)
- Bäume
 - Binärbäume (zwei Nachfolger/Nachbarn)
 - Vielweg-Bäume (mehr als zwei Nachfolger/Nachbarn)
 - Allgemeine Graphen

Die wichtigsten Operationen mit dynamischen Datenstrukturen sind:

- Erzeugen eines neuen Elements
- Einfügen eines Elements
- Entfernen eines Elements
- Suchen eines Elements

In C lassen sich die einzelnen Elemente (Knoten) durch `struct` darstellen. Zur Realisierung verketteter Datenstrukturen müssen diese `structs` **Pointer** auf `struct` ihres eigenen Typs enthalten. Beispiel:

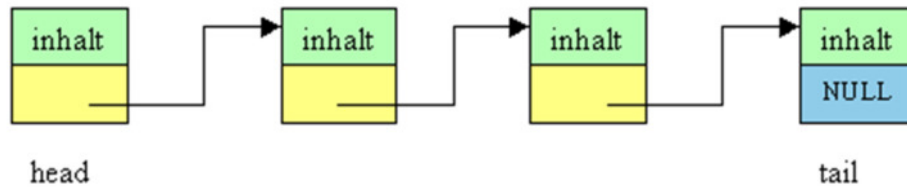
```
struct datum
{
    int tag;
    int monat;
    int jahr;
    int jahrestag;
    char mon_name[4];
    struct datum heute;          /* FALSCH */
};
```

```
struct datum
{
    int tag;
    int monat;
    int jahr;
```

```
int jahrestag;  
char mon_name[4];  
struct datum *heute;      /* RICHTIG */  
};
```

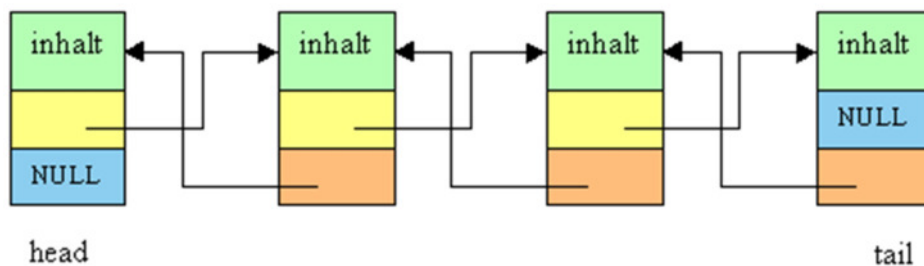
Aufbau verketteter Datenstrukturen mittels rekursiver Strukturen

Einfach verkettete Liste



```
struct listelement
{
    int inhalt;
    struct listelement *next;
};
```

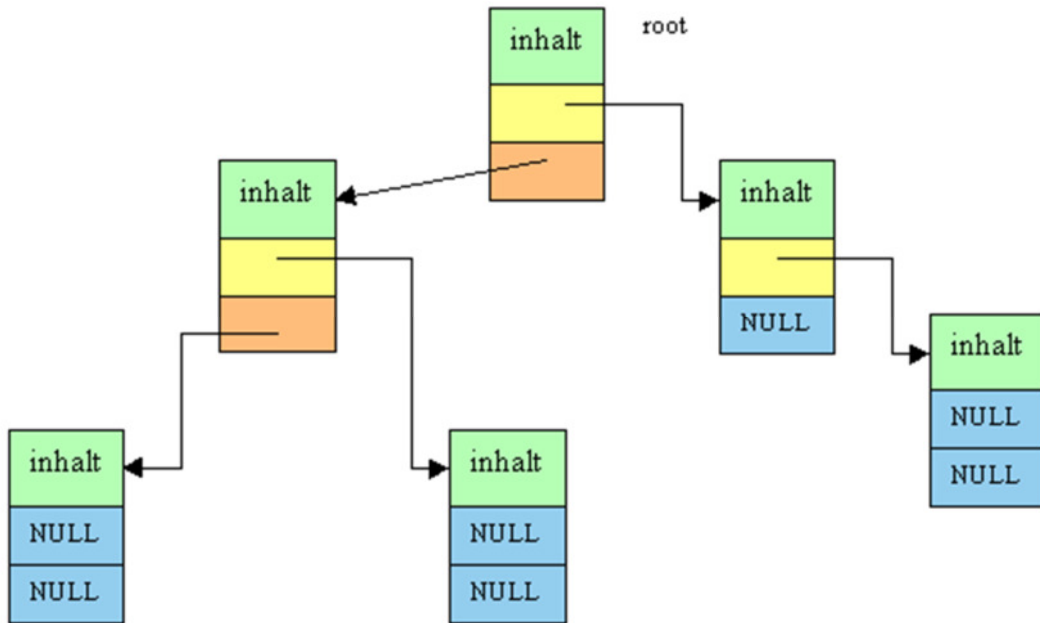
Doppelt verkettete Liste



```
struct listelement
{
    int inhalt;
    struct listelement *next;
    struct listelement *back;
};
```

Binärer Baum

ursprünglich [Programmieren in C](#)
von Prof. Jürgen Plate



```
struct baumelement
{
    int inhalt;
    struct baumelement *rechts;
    struct baumelement *links;
};
```

6 Überblick der Standard-Bibliotheken

Standard-Includefiles

<assert.h>
Kontrolle von Variablen
<ctype.h>
Einteilung der Zeichen in Gruppen
<errno.h>
Die Fehlervariable `errno`
<float.h>
Bereichsgrenzen von Gleitkommazahlen
<limits.h>
Bereichsgrenzen von Ganzzahlen
<locale.h>
Konstanten und Funktionen für lokale Gegebenheiten
<math.h>
Mathematische Funktionen
<setjmp.h>
Nichtlokale Sprünge im Programmablauf
<signal.h>
Behandlung externer Ereignisse
<stdarg.h>
Makros zur Verwaltung variabler Argumente (s. o.)
<stddef.h>
Häufig benötigte Konstanten
<stdio.h>
Ein- und Ausgabefunktionen und -makros
<stdlib.h>
Standardfunktionen
<string.h>
String- und Speicherbereichsfunktionen
<time.h>
Funktionen zum Umgang mit Datum und Zeit

Include-Datei `stddef.h`

`NULL`
nil-Zeiger, gleichbedeutend mit 0
`size_t`
Ergebnistyp von `sizeof`
`ptrdiff_t`
Ergebnistyp der Differenz zweier Zeiger
`wchar_t`
Wide character type

Include-Datei `errno.h`

`errno`

Globale Variable, wird von vielen Funktionen verwendet, enthält bei Fehler den Fehlercode

Include-Datei `stdio.h`

Bekannte Funktionen

```
int putchar(int)
    liefert Zeichen oder EOF, kann Makro sein
int getchar(void)
    liefert Zeichen oder EOF, kann Makro sein
int printf(const char *format, ...)
int scanf(const char *format, ...)
    - Adressen als Argumente
```

Dateiverwaltung/Dateibearbeitung

- Datenstruktur `FILE`
- `FILE *fopen(const char *filename, const char *modus)` liefert `NULL` bei Fehler

"r"

Zum Lesen, Datei muss existieren

"w"

Zum Schreiben, alter Inhalt wird ggf. gelöscht

"a"

Zum Anhängen, alter Inhalt bleibt ggf. erhalten

"r+"

Zum Lesen und Schreiben, alter Inhalt bleibt ggf. erhalten, kann überschrieben werden

"w+"

Zum Lesen und Schreiben, alter Inhalt wird gelöscht

"a+"

Zum Lesen und Anhängen, geschrieben wird immer am Ende

Zusatzzeichen "b" für Binärdateien

Zwischen Lesen und Schreiben `fflush()` oder Positionierungsfunktion nötig

- Maximal `FOPEN_MAX` Dateien gleichzeitig geöffnet, maximale Dateinamenlänge `FILENAME_MAX`
- `int fclose(FILE *f)` liefert 0 oder EOF
- Zu Beginn geöffnete Dateien: `FILE *const stdin;` und `FILE *const stdout;` und `FILE *const stderr;`
- `FILE *freopen(const char *filename, const char *mode, FILE *stream)` liefert `stream` oder `NULL`
- Verwendung zum Umleiten von `stdin`, `stdout`, `stderr`
- `FILE *tmpfile(void)` öffnet "w+b" und löscht nach Programmende
- `char *tmpnam(char s[L_tmpnam])` liefert einen von bis zu `TMP_MAX` verschiedenen temporären Dateinamen nach `s`, `s` darf `NULL` sein
- `int remove(const char *filename)` liefert 0 oder Fehler
- `int rename(const char *oldname, const char *newname)` liefert 0 oder Fehler
- `int fflush(FILE *stream)` liefert 0 oder EOF, Stream `NULL` = alle Dateien

- `int setvbuf(FILE *f, char *buf, int mode, size_t size)` liefert 0 oder Fehler; setzt Puffermechanismus auf vollständig (`mode = _IOFBF`), zeilenweise (`mode = _IOLBF`) oder ungepuffert (`mode = _IONBF`) und verwendet `buf` oder bei `buf = NULL` einen internen Puffer der Länge `size`
- `void setbuf(FILE *f, char *buf)` wie `(void) setvbuf(f, buf, _IOFBF, BUFSIZ)` oder bei `buf = NULL` ungepuffert

Ein- und Ausgabe in Dateien

- `int fprintf(FILE *f, const char *format, ...)` wie `printf()`
- `int fscanf(FILE *f, const char *format, ...)` wie `scanf()`
- `int sprintf(char *s, const char *format, ...)` wie `printf()`
- `int sscanf(char *s, const char *format, ...)` wie `scanf()`
- `int putc(int c, FILE *f)` und `int fputc(int c, FILE *f)` wie `putchar()` (`putc()` kann Makro sein)
- `int getc(FILE *f)` und `int fgetc(FILE *f)` wie `getchar()` (`getc()` kann Makro sein)
- `int puts(const char *s)` und `int fputs(const char *s, FILE *f)` liefert `ok` oder `EOF` (`puts()` ergänzt Zeilenende)
- `char *gets(char *s)` liefert `s` oder bei `EoF` oder Fehler `NULL`, liest beliebig viele Zeichen bis Zeilenende nach `s`, legt statt des `\n` ein Nullbyte ab
- `char *fgets(char *s, int n, FILE *f)` liefert `s` oder bei `EoF` oder Fehler `NULL`, liest maximal `n-1` Zeichen nach `s`, hört aber vorzeitig nach `\n` auf (das noch abgelegt wird), und hängt Nullbyte an
- `int ungetc(int c, FILE *f)` liefert `c` oder `EOF`, stellt ein Zeichen zurück
- `size_t fread(void *p, size_t size, size_t count, FILE *f)`
- `size_t fwrite(void *p, size_t size, size_t count, FILE *f)`

Positionierung in Dateien

- `int fseek(FILE *f, long offset, int origin)` positioniert um `offset` Zeichen relativ zum Dateianfang (`origin = SEEK_SET`), zum Dateiende (`origin = SEEK_END`) oder zur aktuellen Position (`origin = SEEK_CUR`)
- `long ftell(FILE *f)` liefert aktuelle Position oder `-1L`
- `void rewind(FILE *f)` positioniert am Dateianfang und löscht Fehlerstatus
- `int fgetpos(FILE *f, fpos_t *p)` liefert 0 oder Fehler, legt Position in `*p` ab
- `int fsetpos(FILE *f, const fpos_t *p)` liefert 0 oder Fehler, setzt Position aus `*p`

Fehlerbehandlung

- Wenn die letzte Standardein-/ausgabefunktion eine Fehlerbedingung geliefert hat, enthält `int errno` (definiert in `<errno.h>`) eine Fehlernummer
- `void clearerr(FILE *f)` löscht Fehlerstatus
- `int feof(FILE *f)` liefert Zahl bei `EoF` oder 0
- `int ferror(FILE *f)` liefert Zahl bei Fehlerstatus oder 0
- `void perror(const char *s)` gibt Fehlermeldung aus: `printf("%s: %s\n", s, strerror(errno))`

Include-Datei `ctype.h`

- `int toupper(int)` und `int tolower(int)`
- Bei `is..()` muss das einzige Argument EOF oder vom Typ `unsigned char` sein

Ergebnis ist ein Wert ungleich 0, wenn das Zeichen zur Gruppe gehört.

Umlaute sind keine Buchstaben

- `int isalnum(int)` Buchstabe oder Ziffer
- `int isalpha(int)` Buchstabe
- `int iscntrl(int)` Steuerzeichen
- `int isdigit(int)` Dezimale Ziffer
- `int isgraph(int)` Sichtbares Zeichen ohne Leerzeichen
- `int islower(int)` Kleinbuchstaben
- `int isprint(int)` Druckbares Zeichen mit Leerzeichen
- `int ispunct(int)` Sichtbares Zeichen ohne Leerzeichen, Buchstaben, Ziffern
- `int isspace(int)` Leerzeichen, `\f`, `\n`, `\r`, `\t`, `\v`
- `int isupper(int)` Großbuchstaben
- `int isxdigit(int)` Hexadezimale Ziffern

Include-Datei **string.h**

Stringfunktionen

- `char *strcpy(char *ziel, const char *quelle)` liefert `ziel`, kopiert ganzen String incl. Nullbyte
- `char *strncpy(char *ziel, const char *quelle, size_t anz)` liefert `ziel`, kopiert maximal `anz` Zeichen und füllt mit Nullbyte auf, Abschluß mit Nullbyte nicht gewährleistet
- `char *strcat(char *ziel, const char *quelle)` liefert `ziel`, hängt ganzen String an
- `char *strncat(char *ziel, const char *quelle, size_t anz)` liefert `ziel`, hängt bis zu `anz` Zeichen vom String plus Nullbyte an
- `int strcmp(const char *s1, const char *s2)` liefert kleiner/gleich/größer Null, wenn String `s1` lexikalisch vor/gleich/hinter `s2`
- `int strncmp(const char *s1, const char *s2, size_t anz)` wie `strcmp()`, wobei die Strings intern auf maximal `anz` Zeichen abgekürzt werden
- `char *strchr(const char *s, char c)` liefert Zeiger auf erstes Auftreten von `c` in `s` oder NULL
- `char *strrchr(const char *s, char c)` liefert Zeiger auf letztes Auftreten von `c` in `s` oder NULL
- `size_t strspn(const char *s, const char *set)` liefert Anzahl der Zeichen am Anfang von `s`, die alle in `set` vorkommen
- `size_t strcspn(const char *s, const char *set)` liefert Anzahl der Zeichen am Anfang von `s`, die alle nicht in `set` vorkommen
- `char *strpbrk(const char *s, const char *set)` liefert Zeiger auf erste Position in `s`, an der ein Zeichen aus `set` vorkommt
- `char *strstr(const char *vergl, const char *such)` liefert Zeiger auf erstes Auftreten des Strings `such` im String `vergl`
- `size_t strlen(const char *s)` liefert Länge von `s` ohne Nullbyte
- `char *strerror(int errno)` liefert Zeiger auf Fehlertext zur Fehlernummer
- `char *strtok(char *s, const char *delims)` liefert Zeiger auf das erste bzw. bei `s = NULL` das nächste Auftreten eines von Zeichen aus `delims` begrenzten Strings oder NULL, der

Delimiter in s hinter diesem Auftreten wird durch ein Nullbyte überschrieben, bei jedem Aufruf können andere delims angegeben werden

Speicherbereichsfunktionen

- `void *memcpy(void *ziel, const void *quelle, size_t anz)` liefert ziel, kopiert anz bytes
- `void *memmove(void *ziel, const void *quelle, size_t anz)` kopiert anz Bytes, klappt auch bei überlappenden Bereichen
- `int memcmp(const void *m1, const void *m2, size_t anz)` wie `strncmp()`, aber Nullbytes sind kein Ende
- `int memchr(const void *m, int c, size_t anz)` liefert Zeiger auf erstes Byte c in m oder NULL, falls nicht in den ersten anz Bytes
- `void *memset(void *m, int c, size_t anz)` liefert m, setzt anz Bytes ab m auf c

Include-Datei `limits.h`

- `CHAR_BIT` ≥ 8 - Anzahl Bits in char
- `CHAR_MAX` = `UCHAR_MAX` oder `SCHAR_MAX` - Maximaler Wert für char
- `CHAR_MIN` = 0 oder `SCHAR_MIN` - Minimaler Wert für char
- `UCHAR_MAX` ≥ 255 - Maximaler Wert für unsigned char
- `SCHAR_MAX` ≥ 127 - Maximaler Wert für signed char
- `SCHAR_MIN` < -127 - Minimaler Wert für signed char
- `USHRT_MAX` ≥ 65535 - Maximaler Wert für unsigned short
- `SHRT_MAX` ≥ 32767 - Maximaler Wert für short
- `SHRT_MIN` < -32767 - Minimaler Wert für short
- `UINT_MAX` ≥ 65535 - Maximaler Wert für unsigned int
- `INT_MAX` ≥ 32767 - Maximaler Wert für int
- `INT_MIN` < -32767 - Minimaler Wert für int
- `ULONG_MAX` ≥ 4294967295 - Maximaler Wert für unsigned long
- `LONG_MAX` ≥ 2147483647 - Maximaler Wert für long
- `LONG_MIN` < -2147483647 - Minimaler Wert für long

Für den RaspberryPi unter Rasbian/gcc werden folgende Werte geliefert:

```
CHAR_BIT   = 8
CHAR_MAX   = 255
CHAR_MIN   = 0
UCHAR_MAX  = 255
SCHAR_MAX  = 127
SCHAR_MIN  = -128
USHRT_MAX  = 65535
SHRT_MAX   = 32767
SHRT_MIN   = -32768
UINT_MAX   = 4294967295
INT_MAX    = 2147483647
INT_MIN    = -2147483648
ULONG_MAX  = 4294967295
LONG_MAX   = 2147483647
LONG_MIN   = -2147483648
```

Include-Datei `float.h`

- `FLT_RADIX` ≥ 2 - Basis der Exponentendarstellung, z. B. 2, 16
- `FLT_ROUNDS` - Art der Rundung bei Gleitkommaaddition
- `FLT_DIG` ≥ 6 - Genauigkeit von `float` in Dezimalziffern
- `FLT_EPS` ≤ 10 hoch -5 - Kleinster Wert mit `1.0 + FLT_EPS > 1.0`
- `FLT_MANT_DIG` - Länge der `float`-Mantisse in Basisziffern
- `FLT_MAX` ≥ 10 hoch 37 - Maximaler `float`-Wert
- `FLT_MAX_EPS` - Maximales n , für das `FLT_RADIX` hoch n minus 1 als `float` darstellbar
- `FLT_MIN` ≤ 10 hoch -37 - Minimaler `float`-Wert
- `FLT_MIN_EPS` - Minimales n , für das `10` hoch n als `float` normalisiert werden kann
- `DBL_DIG` ≥ 10 - Genauigkeit von `double` in Dezimalziffern
- `DBL_EPS` ≤ 10 hoch -9 - Kleinster Wert mit `1.0 + DBL_EPS > 1.0`
- `DBL_MANT_DIG` - Länge der `double`-Mantisse in Basisziffern
- `DBL_MAX` ≥ 10 hoch 37 - Maximaler `double`-Wert
- `DBL_MAX_EPS` - Maximales n , für das `DBL_RADIX` hoch n minus 1 als `double` darstellbar
- `DBL_MIN` ≤ 10 hoch -37 - Minimaler `double`-Wert
- `DBL_MIN_EPS` - Minimales n , für das `10` hoch n als `double` normalisiert werden kann

Include-Datei `stdlib.h`

Umwandlung Zeichenkette nach Zahl

- Führende Zwischenräume werden ignoriert
- Danach wird umgewandelt bis zu einem ungültigen Zeichen
- Bei über- oder Unterlauf wird `errno = ERANGE` und ein spezieller Wert zurückgegeben
- `double atof(const char *s)` wandelt Zeichenkette in `double` um (wie `strtod(s, NULL)`)
- `int atoi(const char *s)` wandelt Zeichenkette in `int` um (wie `(int) strtol(s, NULL, 10)`)
- `int atol(const char *s)` wandelt Zeichenkette in `long` um (wie `(int) strtol(s, NULL, 10)`)
- `double strtod(const char *s, char **dahinter)` wandelt Zeichenkette in `double` um und speichert Zeiger auf ungültiges Zeichen in `dahinter`; überlaufwert `HUGE_VAL` vorzeichenrichtig, Unterlaufwert 0
- `long strtol(const char *s, char **dahinter, int base)` wandelt Zeichenkette in `long` um und speichert Zeiger auf ungültiges Zeichen in `dahinter` ab; `base` darf 2 bis 36 (Ziffern 0 bis Z) oder 0 sein; bei 0 dient 10 oder 8 (bei führender 0) oder 16 (bei führendem 0x oder 0X) als Basis; überlaufwert vorzeichenrichtig `LONG_MAX` oder `LONG_MIN`
- `unsigned long strtoul(const char *s, char **dahinter, int base)` wie `strtol`, aber Umwandlung in `unsigned long` und überlaufwert `ULONG_MAX`

Zufallszahlen

- `int rand(void)` liefert Zufallszahl von 0 bis `RAND_MAX` ≥ 32767
- `void srand(unsigned int seed)` Setzt neuen Startwert (erster Wert 1)

Speicherverwaltung

- `void *calloc(size_t anz, size_t size)` liefert Zeiger auf gelöschten Speicher für `anz` Elemente der Größe `size` oder `NULL`
- `void *malloc(size_t size)` liefert Zeiger auf Speicherbereich der Größe `size` oder `NULL`

- `void *realloc(void *mem, size_t size)` liefert Zeiger auf neuen Speicherbereich mit geänderter Größe `size` mit dem alten Inhalt von `mem`, `mem` wird freigegeben, größere Länge uninitialisiert, kleine Länge abgeschnitten, oder `NULL`
- `void free(void *mem)` gibt Speicher wieder frei

Diverses

- `void abort(void)` bricht Programm brutal ab (wie `raise(SIGABRT)`)
- `void exit(int rc)` beendet Programm
wie `return(rc)` aus `main()`, `EXIT_SUCCESS=Ok`, `EXIT_FAILURE=Fehler`
- `int atexit(void (*f)(void))` liefert 0 oder Fehler, hinterlegt Funktion zur Ausführung am Ende
- `int system(const char *cmd)` bei `s=NULL` liefert 0, wenn kein Kommandoprozessor verfügbar, sonst implementierungsabhängig.
- `char *getenv(const char *name)` liefert Zeiger auf Umgebungsvariable oder `NULL`
- `int abs(int i)` liefert Absolutwert
- `long labs(long i)` liefert Absolutwert
- `div_t div(int n, int divisor)` liefert `struct div_t` aus `int quot` und `int rem`
- `ldiv_t ldiv(long n, long divisor)` liefert `struct ldiv_t` aus `long quot` und `long rem`
- `void qsort(void *mem, size_t anz, size_t size, int (*cmp)(const void *, const void *))` sortiert Vektor von `anz` Elementen der Größe `size` mit Hilfe der Vergleichsfunktion `cmp`, die `<0 / =0 / >0` liefert
- `void *bsearch(const void *key, const void *mem, size_t anz, size_t size, int (*cmp)(const void *, const void *))` liefert Zeiger auf Element `key` im aufsteigend sortierten Vektor aus `anz` Elementen der Größe `size` mit Hilfe der Vergleichsfunktion `cmp`, die `<0 / =0 / >0` liefert

Include-Datei `math.h`

Fehlercodes für `errno`

- `EDOM` = Illegales Argument (Ergebnis undefiniert)
- `ERANGE` = Immer bei *overflow* (Ergebnis $\pm \text{HUGE_VAL}$), kann bei *underflow* (Ergebnis 0)

Funktionen

- `double sin(double x)` liefert Sinus
- `double cos(double x)` liefert Cosinus
- `double tan(double x)` liefert Tangens
- `double asin(double x)` liefert Arcus Sinus $[-1,1]$ im Bereich $[-\pi/2, \pi/2]$
- `double acos(double x)` liefert Arcus Cosinus $[-1,1]$ im Bereich $[0, \pi]$
- `double atan(double x)` liefert Arcus Tangens im Bereich $]-\pi/2, \pi/2[$
- `double atan2(double y, double x)` liefert Arcus Tangens (y/x) im Bereich $[-\pi, \pi]$
- `double sinh(double x)` liefert Sinus hyperbolicus
- `double cosh(double x)` liefert Cosinus hyperbolicus
- `double tanh(double x)` liefert Tangens hyperbolicus
- `double exp(double x)` liefert e hoch x

- `double log(double x)` liefert $\ln(x)$, $x > 0$
- `double log10(double x)` liefert $\lg(x)$, $x > 0$
- `double pow(double x, double y)` liefert x hoch y , Fehler wenn $x=0$ und $y \leq 0$ oder $x < 0$ und y nicht ganzzahlig
- `double sqrt(double x)` liefert Wurzel aus x
- `double ceil(double x)` liefert kleinste ganze Zahl $\geq x$
- `double floor(double x)` liefert größte ganze Zahl $\leq x$
- `double fabs(double x)` liefert $|x|$
- `double ldexp(double x, int n)` liefert x mal 2 hoch n
- `double frexp(double x, int *n)` (Gegenteil zu `ldexp`) liefert Mantisse in $[1/2, 1[$, Exponent nach $*n$
- `double modf(double x, double *i)` liefert Ganzzahliger Teil, Rest nach $*i$, beides mit Vorzeichen von x
- `double fmod(double x, double y)` liefert Rest von x/y mit Vorzeichen von x , bei $y=0$ implementierungsabhängig

Include-Datei `assert.h`

- `void assert(int ausdruck)` (Makro) bricht, wenn `ausdruck=0`, Programm mit Meldung `Assertion failed: ausdruck, file filename, line nnn` ab
- `assert()` dient zur Kontrolle der Richtigkeit eines Programms und kann durch Definition von `NDEBUG` vor `#include <assert.h>` ausgeschaltet werden

Include-Datei `setjmp.h`

Nichtlokale Sprünge

- `int setjmp(jmp_buf buffer)` liefert `0` beim Aufruf und ungleich `0` beim Sprung, definiert Sprungmarke
- `int longjmp(jmp_buf buffer, int n)` springt zum mit `setjmp(buffer)` definierten Ort, n ist dann das von `0` verschiedene Ergebnis von `setjmp`
- Vorsicht: Die Funktion, in der `setjmp` aufgerufen wurde, muss bei `longjmp` noch aktiv sein, `auto`- und `register`-Variablen ohne `volatile` in dieser Funktion sind beim Ansprung über `longjmp` undefiniert
- `if(setjmp(buffer)) { ... von longjmp() ... } else { ... direkt ... } ...`

Include-Datei `signal.h`

Behandlung von Ausnahmebedingungen

- `void (*signal(int sig, void (*handler)(int)))(int)` setzt neuen `handler()` für Signaltyp `sig` und liefert alten `handler()`
- `int raise(int sig)` liefert `0` oder Fehler, sendet Signal `sig` ab

Signaltypen

- `SIGABRT` anomaler Programmabbruch (z. B. durch `abort()`)
- `SIGFPE` Arithmetikfehler wie Division durch Null oder Overflow
- `SIGILL` Illegaler Funktionstext oder Maschinenbefehl

- SIGINT Unterbrechung von Tastatur o. ä.
- SIGSEGV Illegaler Speicherzugriff
- SIGTERM Aufforderung zum Programmende
- u. a.

Handler

- SIG_DFT Der voreingestellte Handler
- SIG_IGN Ignorieren des Signals
- SIG_ERR (Rückgabewert von `signal()` bei Fehler)
- `void (*handler)(int)` Eigener Handler, Argument ist Signaltyp
- Beim Ende des Handlers an der Unterbrechungsstelle weitergemacht (wenn nicht `exit()`, `longjmp()` o. ä. verwendet wird)

Include-Datei `time.h`

Kalenderzeittyp `time_t`

Ortszeit-Struktur `struct tm`

- `int tm_sec` Sekunden nach voller Minute (0-61)
- `int tm_min` Minuten nach voller Stunde (0-59)
- `int tm_hour` Stunden seit Mitternacht (0-23)
- `int tm_mday` Tag im Monat (1-31)
- `int tm_mon` Monat *nach* Januar (0-11)
- `int tm_year` Jahr minus 1900
- `int tm_wday` Wochentag (0-6, 0=Sonntag)
- `int tm_yday` Tag im Jahr *nach* 1. Januar (0-365)
- `int tm_isdst` >0 Sommerzeit, =0 Winterzeit, <0 unbekannt

Funktionen, die die aktuelle Zeit liefern

- `clock_t clock(void)` liefert Systemticks seit Programmstart oder -1; Sekunden
= `clock() ÷ CLOCKS_PER_SEC`
- `time_t time(time_t *zeit)` liefert Kalenderzeit oder -1, wenn `zeit` ungleich NULL auch nach `zeit`

Funktionen, die mit Zeiten rechnen

- `struct tm *localtime(const time_t *zeit)` liefert Zeiger auf statische Struktur mit Ortszeit
- `struct tm *gmtime(const time_t *zeit)` liefert Zeiger auf statische Struktur mit UTC (früher GMT) oder NULL
- `double difftime(time_t zeit2, time_t zeit1)` liefert Zeitdifferenz in Sekunden
- `time_t mktime(struct tm *zeit)` wandelt `struct tm` in `time_t` um, -1 bei Fehler
- `char *asctime(const struct tm *zeit)` liefert Statische Zeichenkette in der Form "Mon Dec 31 23:59:59 1990\n"
- `char *ctime(const time_t *zeit)` liefert `asctime(localtime(*zeit))`

- `size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *werte)` legt Uhrzeit `werte` im Format `fmt` in Speicherbereich `s` der Länge `maxsize` ab, liefert Länge ohne `\0` oder 0, wenn es nicht paßt

Formatangaben für `strftime`

- `%a` Abgekürzter Name des Wochentags
- `%A` Voller Name des Wochentags
- `%b` Abgekürzter Name des Monats
- `%B` Voller Name des Monats
- `%c` Lokale Darstellung von Datum und Zeit
- `%d` Tag im Monat (01-31)
- `%H` Stunde (00-23)
- `%I` Stunde (00-11)
- `%j` Tag im Jahr (001-366)
- `%m` Monat (01-12)
- `%M` Minute (00-59)
- `%p` lokale Form von AM oder PM
- `%s` Sekunde (00-61)
- `%U` Woche im Jahr (Sonntag erster Wochentag, 00-53)
- `%w` Wochentag (0-6, Sonntag=0)
- `%W` Woche im Jahr (Montag erster Wochentag, 00-53)
- `%x` Lokale Form des Datums
- `%X` Lokale Form der Zeit
- `%y` Jahr zweistellig (00-99)
- `%Y` Jahr vierstellig (1900-)
- `%Z` Name der Zeitzone, falls gegeben
- `%%` steht für `%`