
Programmiersprache C

J.-P. Kuska

21. Dezember 1998

Inhaltsverzeichnis

1 Grundkonzept	4
1.1 Kommentare	4
1.2 Bezeichner	4
1.3 Reservierte Wörter	4
1.4 Daten Vereinbarungen	4
1.5 Konstanten	5
1.6 Ausdrücke und Operatoren	6
1.6.1 Wertzuweisung	6
1.6.2 Typumwandlungsoperator	7
1.6.3 Zugriff auf Vektorelemente	7
1.6.4 Adreß und Inhaltsoperator	7
1.6.5 Arithmetische Operatoren	8
1.6.6 Vergleichsoperatoren	8
1.6.7 Logische Operatoren	8
1.6.8 Bitoperatoren	8
1.6.9 Inkrement- und Dekrement- Operatoren	8
1.6.10 Komma als Operator	9
1.6.11 Entscheidungsoperator	9
1.6.12 Der sizeof-Operator	10
1.7 Steuerung des Programmablaufs	10
1.8 Block-Anweisung	10
1.8.1 if-Anweisung	10
1.8.2 switch-Anweisung	12
1.8.3 while-Anweisung	12

1.8.4	do-while-Anweisung	12
1.8.5	for-Anweisung	12
1.8.6	Unbedingte Sprungbefehle	13
1.9	Definition von Funktionen	14
2	Felder, Strukturierte Datentypen und Zeiger	17
2.1	Felder	17
2.2	Mehrdimensionale Felder	19
2.3	Aufzählungen	19
2.4	Strukturen	20
2.5	Unionen	23
2.6	typedef Vereinbarung	23
2.7	Lineare Listen – ein Beispiel	25
3	Der Preprozessor	31
3.1	Makrodefinitionen	31
3.2	Bedingte Compilierung	32
3.3	Weitere Preprozessorbefehle	33
4	Standard-Bibliothek	33
4.1	Ein- und Ausgabe: <stdio.h>	33
4.1.1	Dateioperationen	34
4.1.2	Formatierte Ausgabe	35
4.1.3	Formatierte Eingabe	37
4.1.4	Ein- und Ausgabe von Zeichen	38
4.1.5	Direkte Ein- und Ausgabe	39
4.1.6	Positionieren in Dateien	39
4.1.7	Fehlerbehandlung	39
4.2	Tests für Zeichenklassen: <ctype.h>	41
4.3	Funktionen für Zeichenketten: <string.h>	41
4.4	Variable Argumentliste: <stdarg.h>	42
4.5	Mathematische Funktionen: <math.h>	44
4.6	Hilfsfunktionen <stdlib.h>	44
4.6.1	Speicherverwaltung	45
4.6.2	Kommunikation mit dem Betriebssystem	45
4.6.3	Weiter Hilfsfunktionen	46

5	Variablen und Funktionen in einem Projekt	47
6	Doppelt verkettete Liste	50
7	Sortieren	57

1 Grundkonzept

C ist eine imperative Programmiersprache funktionalen Elementen. Viele Anweisung haben einen Wert, der weiter verwendet werden kann. Die Beliebtheit von C beruht auf der Standardisierung der Syntax *und* der Bibliotheken. C ermöglicht ein modularen Programmaufbau. Das Hauptprogramm ist immer die Funktion `main`. `main` gibt stets einen ganzzahligen Wert an das aufrufende Betriebssystem zurück. Das beliebte „Hallo Welt!“ Programm hat die folgende Gestalt, und liefert bei erfolgreicher Ausführung 0 an das Betriebssystem zurück.

```
#include <stdio.h>

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

1.1 Kommentare

Ein Kommentar beginnt in C mit `/*` und endet mit `*/`.

1.2 Bezeichner

Ein Bezeichner für Funktionen und Variablen ist eine Folge aus den Buchstaben A–Z, a–z, 1–0 und `_`, die nicht mit einer Ziffer beginnt. Bei einem Bezeichner sind normalerweise die ersten 31 Zeichen signifikant. Bei Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden, `dummy` und `Dummy` sind also verschiedene Namen.

1.3 Reservierte Wörter

Die folgenden Wörter sind reserviert und dürfen nur in ihrer vordefinierten Bedeutung benutzt werden.

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

1.4 Daten Vereinbarungen

Eine Datenvereinbarung hat die Form

```
<Speicherklasse> <Typ>
    <Bezeichner>$_1$, $... $<Bezeichner>$_n$
```

Alle Daten müssen vereinbart werden. Die Vereinbarung von Daten folgt dem Prinzip, die Deklaration möglichst ähnlich dem Aufruf zu gestalten. Dadurch sind Daten- und Datentypvereinbarungen relativ schwer zu lesen. Die Vereinbarungen der Grundtypen kann zum Beispiel so aussehen.

```
char          c;
short int     short_number;
```

```

int          a_longer_number;
long int     a_very_long_number;
float        x_low_precision;
double       y_double_precision, z_an_other_number;
long double  xl_high_precision;

```

Bei den ganzzahligen Datentypen gibt es `char`, welches eine Speichereinheit (normalerweise ein Byte) unfaßt, `short int`, `int` und `long int`. C behandelt Variablen vom Typ `char` als Zahlen. Für den Bereich der Zahlen gilt `char ≤ short int ≤ int ≤ long int`. Der Wertebereich und die Länge hängen von der jeweiligen Maschinenarchitektur ab. Jedem dieser Typen kann ein `unsigned` vorangestellt werden.

Als *Speicherklasse* sind `auto`, `register`, `static` und `extern` möglich. Die Angaben `auto` und `register` legen die automatische Speicherklasse fest und können nur in Funktionen verwendet werden. `register` ist äquivalent zu einer `auto` Definition, deutet aber an, das diese Variable besonders häufig benutzt wird und daher in der Registern des Prozessors gehalten werden soll. *Die Speicheradresse einer Registervariable kann nicht berechnet werden.* Beide Angaben vereinbaren die Variable.

Die Speicherklasse `static` legt innerhalb einer Funktion fest, daß dauerhaft für die Variable Speicherplatz reserviert wird. Normalerweise ist der Wert einer Variablen innerhalb einer Funktion bei jedem Aufruf anfänglich undefiniert. Bei einer statischen Variable ist das nicht der Fall. Zum Beispiel würde die Funktion

```

void every_ten(void)
{ static int count=0;

  if ((count % 10)==0)
    printf("You are a poor user.");
  count++;
}

```

nur bei jedem zehnten Aufruf eine Ausgabe erzeugen. In einem Modul wird mit `static` eine nur in diesem Modul globale Variable erzeugt.

Mit `extern` wird die Variable nur deklariert aber *nicht* vereinbart. Damit wird eine globale Variable für das ganze Projekt definiert. Alle Module referenzieren diese Variable als `extern` im Hauptprogramm wird diese Variable dann vereinbart und ist für alle anderen Module erreichbar.

1.5 Konstanten

Eine ganzzahlige Konstante besteht aus einer Ziffernfolge. Sie wird normalerweise dezimal interpretiert. Falls die Folge mit 0 beginnt, wird sie oktall interpretiert. Oktale Konstanten dürfen die Ziffern 8 und 9 nicht enthalten. Beginnt die Folge mit 0x oder 0X wird sie hexadezimal interpretiert. Zu den hexadezimalen Ziffern gehören a (oder A)–f (oder F). Der Buchstabe u oder U kann angehängt werden um anzuzeigen, daß der Wert vorzeichenlos ist. Der Buchstabe l oder L kann angehängt werden, der angibt, daß die Konstante vom Typ `long` ist.

Zeichenkonstanten sind eine Folge von ein oder mehreren Zeichen, eingeschlossen in einfache „quotes“, also zum Beispiel `'A'`. Zeichenkonstanten dürfen den einfachen „quote“ nicht enthalten. Spezielle Zeichenkonstanten existieren für

new line	<code>\n</code>	backslash	<code>\\</code>
tab	<code>\t</code>	question	<code>\?</code>
vertical tab	<code>\v</code>	quote	<code>\'</code>
backspace	<code>\b</code>	double quote	<code>\"</code>
carrige return	<code>\r</code>	oktale Zahl	<code>\ooo</code>
paper feed	<code>\f</code>	hexadezimale Zahl	<code>\xhh</code>
bell	<code>\a</code>		

Zeichenkettenkonstanten sind in „double quotes“ eingeschlossen. Somit sind

```
char s[]="Hallo Welt!";
char a_string[]="A";
char a_char='A';
```

die ersten beiden Vereinbarungen Zeichenkettenkonstanten, a_char ist aber eine Zeichenkonstante.

Gleitpunktkonstanten werden mit einem Dezimalpunkt angegeben. Optional kann ein Exponent folgen.

```
double one=1.;      double two=2.0;
double huge=1.0e10; double tiny=1.E-10;
long double lpi=3.14159265358979323846L;
```

Um float und long double Konstanten zu vereinbaren, muß ein f oder F bzw. l oder L angehängt werden. Ohne diese Postfixe sind Gleitpunktkonstanten vom Typ double und werden erst zur Laufzeit konvertiert.

1.6 Ausdrücke und Operatoren

1.6.1 Wertzuweisung

Beispiele für Wertzuweisungen sind schon angegeben worden. Die Zuweisung erfolgt mit

$\langle l-Value \rangle = \langle Ausdruck \rangle$

ein $\langle l-Value \rangle$ ist dabei ein Ausdruck, der sich auf einen modifizierbaren Hauptspeicherbereich bezieht. Mehrfachzuweisungen sind möglich, da das funktionale Konzept von C impliziert, daß auch die Wertzuweisung selbst eine Wert hat. Die Klammerung bei Mehrfachzuweisungen ist folgendem Beispiel zu entnehmen.

```
float a,b,c;

a=b=c=1.0f;
a=(b=(c=1.0f));
```

Für die häufig benutzen Ausdrücke der Form

$x =x \langle Operator \rangle \langle Ausdruck \rangle$

Gibt es die Operatoren +=, -=, *=, /= und %=.

+=	a+=b	$a = a + b$
-=	a-=b	$a = a - b$
=	a=b	$a = ab$
/=	a/=b	$a = a/b$
%=	a%=b	$a = a \text{ mod } b$

1.6.2 Typumwandlungsoperator

Mit $\langle\langle Typname \rangle\rangle \langle Variable \rangle$ kann eine Typkonvertierung explizit erzwungen werden. So wird zum Beispiel mit

```
float f;
int i;

f=(float) i;
```

i in eine float-Zahl konvertiert.

1.6.3 Zugriff auf Vektorelemente

Auf die Elemente eines Vektors wird mit $[\langle Ausdruck \rangle]$ zugegriffen. Dabei ist $[\]$ ein Postfixoperator. Um die ersten vier Elemente eines Vektors mit 1 zu initialisieren wäre zum Beispiel die Konstruktion

```
float vector[10];

vector[0]=vector[1]=vector[2]=vector[3]=1.0f;
```

denkbar.

1.6.4 Adreß und Inhaltsoperator

Um die Adresse einer Variablen zu bestimmen, wird der Operator $\&$ verwendet. Die Syntax dafür lautet:

```
 $\langle Zeiger \rangle = \&\langle Variable \rangle$ 
```

Um auf den Inhalt des $\langle Zeigers \rangle$ wieder zuzugreifen, wird der Inhaltsoperator benutzt:

```
 $\langle Variable \rangle = *\langle Zeiger \rangle$ 
```

Ein Beispiel soll das verdeutlichen

```
double variable;
double *adresse;      /* Zeiger auf double vereinbaren. */
double x,times4;

x=1.0;
variable=2.0;
adresse= &variable;  /* Zuweisen der Adresse. */
(*adresse)=4.0;      /* Veraendern des Wertes
                       von variable. */
printf(" %g",variable); /* Das gibt 4.0 aus. */

times4= (*adresse)*x;
```

Die Vereinbarung eines Zeigers auf $\langle Typ \rangle$ mit

```
 $\langle Typ \rangle * \langle Bezeichner \rangle$ 
```

soll andeuten, daß beim Aufruf die Konstruktion $*\langle Bezeichner \rangle$ den Typ $\langle Typ \rangle$ hat.

1.6.5 Arithmetische Operatoren

Als elementare arithmetische Operatoren sind +, -, *, / und % vorhanden. Dabei haben + und - als unäre Operatoren die Bedeutung eines Vorzeichens. Als binäre Operatoren gelten:

+	a+b	$a + b$	Addition
-	a-b	$a - b$	Subtraktion
*	a*b	ab	Multiplikation
/	a/b	a/b	Division, ganzzahliger Anteil bei ganzen Zahlen
%	a % b	$a \bmod b$	Rest der Division bei ganzen Zahlen

1.6.6 Vergleichsoperatoren

>	a>b	$a > b$	größer
<	a<b	$a < b$	kleiner
>=	a>=b	$a \geq b$	größer gleich
<=	a<=b	$a \leq b$	kleiner gleich
==	a==b	$a = b$	gleich
!=	a!=b	$a \neq b$	ungleich

1.6.7 Logische Operatoren

In C gibt es keinen Datentyp für logische Werte. Es gilt die Konvention, daß ein Ausdruck falsch ist, wenn er den Wert 0 hat, anderenfalls wahr.

!	! a	$\neg a$	Negation
&&	a && b	$a \wedge b$	und
	a b	$a \vee b$	oder

1.6.8 Bitoperatoren

Um die Bits einzelner (ganzer) Zahlen zu modifizieren, stellt C die Bitoperationen

~	~a	$\neg a$	bit negation
&	a & b	$a \wedge b$	und
	a b	$a \vee b$	oder
^	a ^ b	$(a \wedge \neg b) \vee (\neg a \wedge b)$	exklusives oder
<<	a<<b	$a \times 2^b$	Bits nach links schieben
>>	a>>b	$a/2^b$	Bits nach rechts schieben

Zusätzlich gibt es für Bitoperatoren die Zuweisungen

&=	a&= b	$a = a \wedge b$	und
=	a = b	$a = a \vee b$	oder
^=	a^=b	$a = (a \wedge \neg b) \vee (\neg a \wedge b)$	exklusives oder
<<=	a<<=b	$a = a \times 2^b$	Bits nach links schieben
>>=	a>>=b	$a = a/2^b$	Bits rechts schieben

1.6.9 Inkrement- und Dekrement- Operatoren

Mit den Operatoren

`++<1-Value>;`

```
--<l-Value>;

<l-Value>++;

<l-Value>--;
```

wird der *ganzzahlige* *<l-Value>* um jeweils 1 vergrößert oder verkleinert. Die Verwendung der Operatoren als Präfix entspricht *erst verändern* und dann den veränderten Wert zurück geben. Die Postfixnotation bedeutet verwendet den alten Wert als Ergebnis. Als Funktionen¹ `praefix_pp(&i)`, `postfix_pp(&i)` geschrieben ergibt sich für `++i` und `i++`

```
int praefix_pp(int *i) /* fuer ++i */
{
    (*i)+=1;
    return *i;
}

int postfix_pp(int *i) /* fuer i++ */
{ int ii;

    ii=(*i);
    (*i)+=1;
    return ii;
}
```

1.6.10 Komma als Operator

Der Kommaoperator wird besonders häufig in `for` Anweisungen verwendet. Seine Syntax lautet

```
<Ausdruck> , <Ausdruck>
```

Ausdrücke, die durch Komma getrennt sind, werden von links nach rechts bewertet. Der Wert der Kommaoperation ist der des *letzten* Ausdrucks. So wird zum Beispiel mit

```
double f(double, double, double);
double a, c, t, x;

x=f(a, (t=3.0, t+2.0), c); /* Aufruf der Funktion f
                           mit f(a, 5.0, c) */
```

die Funktion `f` aufgerufen. Die Variable `t` hat danach den Wert 3.0.

1.6.11 Entscheidungsoperator

Der Entscheidungsoperator ist die eine Verknüpfung von 3 Ausdrücken.

```
<logischer Test> ? <Test wahr> : <Test falsch>
```

¹ Da der Wert des Funktionsarguments verändert werden soll, `C` aber stets die Argumente in den „stack“ kopiert, muß ein Zeiger auf das Argument übergeben werden.

Liefert *⟨logischer Test⟩* einen Wert $\neq 0$, so ist der Wert des Entscheidungsoperators *⟨Test wahr⟩* sonst *⟨Test falsch⟩*. Hier ein Beispiel, das auch die Verwendung des Kommaoperators demonstriert.

```
double a,b,max,min;

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

/* nun beides zusammen */
max=(a>b ? min=b, a : min=a, b);
```

1.6.12 Der sizeof-Operator

Dieser Operator liefert die Anzahl der Bytes, die benötigt werden, um ein Objekt mit dem Typ seines Operanden zu speichern. Der Operand ist entweder ein Ausdruck, der nicht bewertet wird oder ein Typname in Klammern.

1.7 Steuerung des Programmablaufs

1.8 Block-Anweisung

Ein Block wird durch { eingeleitet und durch } beendet. Am Anfang *jedes* Blockes können Variablen vereinbart werden, auch wenn dies eine nur selten genutzte Eigenschaft ist. An jeder Stelle im Programm, an der eine einzelne Anweisung erlaubt ist, kann auch ein Block stehen. Eine einfache Anweisung muß mit einem Semikolon ; abgeschlossen werden, ein Block nicht. In C steht das Semikolon am Ende jeder Anweisung und nicht zwischen den Anweisungen wie zum Beispiel in Pascal.

1.8.1 if-Anweisung

Die if-Anweisung hat die Form

```
if(⟨Test⟩)
  ⟨Anweisung⟩1
```

optional kann ein else Zweig folgen

```
if(⟨Test⟩)
  ⟨Anweisung⟩1
else
  ⟨Anweisung⟩2
```

if-Anweisungen können geschachtelt werden. Der else-Zweig wird dabei dem letzten losen if-Zweig zugeordnet. Die verschachtelte if-Anweisung

```
if (x==0.0)
  if (y==0.0)
    ...
  else
    z=x/y;
```

wird also interpretiert als wäre sie

Operator		Assoziativität
()	Funktionsaufruf	von links her
[]	Feldelement	
.	struct oder union Element	
->	Zeiger auf struct Element	
!	logische Negation	von rechts her
~	Bit not	
-	unäres Minus	
+	unäres Plus	
++	Inkrement	
--	Dekrement	
&	Adresse von	
*	Inhalt von	
(type)	Typumwandlung	
sizeof	Größe in Bytes	
*	Multiplikation	von links her
/	Division	
%	Rest	
+	Addition	von links her
-	Subtraktion	
<<	bitweise links Verschiebung	von links her
>>	bitweise rechts Verschiebung	
<	arithmetisches Kleiner	von links her
>	arithmetisches Größer	
<=	arithmetisches Kleiner-Gleich	
>=	arithmetisches Größer-Gleich	
==	arithmetisches Gleich	von links her
!=	arithmetisches Ungleich	
&	bitweises Und	von links her
^	bitweises exklusives Oder	von links her
	bitweises Oder	von links her
&&	logisches Und	von links her
	logisches Oder	von links her
? :	bedingter Ausdruck	von rechts her
=	Zuweisung	von rechts her
auch += -= *= /= %= <<= >>= &= ^= =		
,	Kommaoperation	von links her

Tabelle 1: Vorrang und Assoziativität der Operatoren. Weiter oben stehende Operatoren haben eine höhere Priorität.

```

if (x==0.0) {
    if (y==0.0)
        ...
    else
        z=x/y;
}

```

geklammert.

1.8.2 switch-Anweisung

Für die Auswahl aus mehreren Alternativen stellt C die `switch`-Anweisung zur Verfügung.

```

switch (<Ausdruck>) {
    case <Konstante>1 :
        <Anweisungen>1
        break;
    case <Konstante>2 :
        <Anweisungen>2
        break;
    :
    default : <Anweisungen>d
}

```

Gilt $\langle \text{Ausdruck} \rangle = \langle \text{Konstante} \rangle_n$, so werden die $\langle \text{Anweisungen} \rangle_n$ ausgeführt. Die `break`-Anweisung sollte am Ende nicht fehlen, sonst werden auch die folgende `case` Zweige getestet und die Anweisungen eines eventuell vorhandenen `default` Falles ausgeführt.

1.8.3 while-Anweisung

Mit `while` wird eine abweisende Schleife programmiert.

```

while (<Test>)
    <Anweisung>

```

Die $\langle \text{Anweisung} \rangle$ wird solange ausgeführt, bis der $\langle \text{Test} \rangle$ den Wert 0 ergibt.

1.8.4 do-while-Anweisung

Mit `do-while` wird eine annehmende Schleife programmiert.

```

do
    <Anweisung>
while (<Test>);

```

Die $\langle \text{Anweisung} \rangle$ wird ausgeführt, der $\langle \text{Test} \rangle$ ausgewertet und die $\langle \text{Anweisung} \rangle$ solange wiederholt, bis der $\langle \text{Test} \rangle$ den Wert 0 liefert.

1.8.5 for-Anweisung

Eine einfache Zählschleife wird mit der `for`-Anweisung vereinbart.

```

for(<Initialisation>; <Test>; <Inkrement>)
    <Anweisung>

```

Die for-Anweisung entspricht der Konstruktion

```
⟨Initialisation⟩
while(⟨Test⟩) {
    ⟨Anweisung⟩
    ⟨Inkrement⟩;
}
```

mit Ausnahme des Verhaltens von `continue`. Die `for`-Anweisung ist sehr leistungsfähig. Zum Beispiel läßt sich die Binär-Suche in einem Feld von Daten ebenfalls durch eine `for`-Anweisung beschreiben, dabei sei v ein geordneter Vektor von Zahlen und es wird zur Zahl x der Index i gesucht, für den gilt $v_i \leq x < v_{i+1}$.

```
int binsearch(double x, double v[], int n)
{ int low,high,med;

  for(low=0,high=n-1,med=high/2; low<=high; med=(low+high)/2)
    if (x<v[med])
      high=med-1;
    else if (x>v[med])
      low=med+1;
    else
      return med;
  return (med>=0 ? med : -1); /* not found */
}
```

1.8.6 Unbedingte Sprungbefehle

C stellt die Befehle `break`, `continue` und `goto` für diesen Zweck zur Verfügung.

Die `break`-Anweisung wird verwendet, um die Abarbeitung einer unmittelbar übergeordneten `switch`-, `while`-, `do-while`- oder `for`-Anweisung abzubrechen. Auch eine unendliche Schleife kann damit verlassen werden.

Die `continue` Anweisung kehrt zum Anfang umgebenden Schleife zurück. Bei einer `while`- oder `do-while`-Anweisung wird zuerst der Schleifentest ausgeführt. Bei einer `for`-Anweisung werden zunächst die *Inkrement*-Anweisungen ausgeführt. Soll zum Beispiel aus den positiven Elementen eines Vektors v die Wurzel gezogen werden, so wäre folgende Konstruktion möglich:

```
for (i=0; i<n; i++) {
    if (v[i]<0.0)
        continue;
    v[i]=sqrt(v[i]);
}
```

Die `goto` Anweisung springt mit

```
    goto ⟨Label⟩;
    ⋮
    ⟨Label⟩:⟨Anweisung⟩;
```

zum Label $\langle Label \rangle$ und führt das Programm von dort aus weiter.

1.9 Definition von Funktionen

Funktionen sind die grundlegende Struktur in einem C-Programm. Die Definition einer Funktion hat die Gestalt

```
<Speicherklasse> <Typ> <Funktionsname>(<Parameterliste>)  
  {<lokale Variablen>  
  
    <Anweisungen>  
  }
```

Fehlt die *<Speicherklasse>*, so wird `auto` angenommen. Fehlt die *<Typ>* Angabe, so wird `int` angenommen. Die *<Parameterliste>* hat die Form

```
<Typ>1 <Variablenname>1, <Typ>2 <Variablenname>2, ...
```

Funktionen dürfen rekursiv sein, sich also selbst aufrufen. Die Parameter werden stets auf den Stack kopiert. Eine Funktion kann also den Inhalt einer Variablen *nicht* ändern. Die Rekursivität sei am Beispiel der Fakultät, $n! = \prod_{i=1}^n i$, $0! = 1$, verdeutlicht:

```
long int factorial(long int n)  
{  
    if (n==0)  
        return 1;  
    if (n==1)  
        return 1;  
    return n*factorial(n-1);  
}
```

Das Ergebnis einer Funktionsberechnung wird mit `return <Resultat>` zurückgegeben. Fehlt ein `return` Befehl oder wird er während der Abarbeitung nicht erreicht, so ist der Wert der Funktion undefiniert. Die Abarbeitung wird dann mit dem letzten Befehl vor der schließenden Klammer beendet.

Das Konzept von Prozeduren oder Unterprogrammen ist eigentlich nicht in C vorgesehen. Die Verwendung des Ergebnisses einer Funktion ist jedoch nicht notwendig. Um bei einer Funktion anzuzeigen, daß sie keinen sinnvollen Rückgabewert liefert, ist der Typ `void` eingeführt worden. Ebenso ist bei einer Funktion, die keine Parameter hat, `void` als Parameterliste anzugeben. Hier das „Hallo Welt!“ Programm mit einem Unterprogramm.

```
#include <stdio.h>  
  
void SayHello(void)  
{  
    printf("Hallo Welt!\n");  
}  
  
int main(void)  
{  
    SayHello();  
    return 0;  
}
```

Sollen Funktionen wie Prozeduren gebraucht werden, also Argumente verändern, so ist nicht der Wert der Variablen (die verändert werden soll) sondern ihre Adresse zu übergeben. Die Adresse wird dann zwar auf den Stack kopiert und kann nicht verändert werden, aber der Inhalt ist veränderbar. Die Prozedur Swap, die die Werte Ihrer Argumente vertauscht, sieht dann also so aus:

```
void Swap(double *x, double *y)
{ double tmp;

  tmp=( *x);
  ( *x)=( *y);
  ( *y)=tmp;
}
```

/* Der Aufruf erfolgt mit

```
double x,y;

if (x<y)
  Swap(&x,&y);
```

Als ein weiteres Beispiel für eine Funktion sei die Nullstellensuche $f(x) = 0$ mit der Newton-Regel $x_{n+1} = x_n - f(x_n)/f'(x_n)$ demonstriert werden. Die Funktion soll, ausgehend von einem Startwert x_0 die Nullstelle $f(x^*) = 0$ approximieren, bis eine bestimmte Genauigkeit $|x_n - x_{n+1}| < \varepsilon$ erreicht ist oder $|f(x_n)| < \varepsilon$ gilt. Das Newton-Verfahren findet nur einfache Nullstellen.

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

double newton_iter(double x,
                  double (*f)(double),
                  double (*dfdx)(double),
                  double eps,
                  int maxit)
{ int iter;
  double dx,fx,dfx;

  iter=0;
  do {
    fx=f(x);
    dfx=dfdx(x);
    dx=-fx/dfx;
    x+=dx;
  }
  while ((fabs(fx)>eps) && (fabs(dx)>eps) && ((iter++)<maxit));
  return x;
}

int main(void) {
```

```

    printf("Pi = %10.20g\n", newton_iter(3.0,sin,cos,1.0E-
16,20));
    return 0;
}

```

Die Funktion `newton_iter` erhält als Startwert den Parameter $x_0 = x$ und die Funktionen $f(x)$ und $f'(x) = df/dx$. Die Genauigkeit ε wird als Parameter `eps` übergeben. Um eine Oszillation und somit eine unendliche Schleife zu verhindern, wird noch eine maximale Anzahl von Iterationen `maxit` als Parameter übergeben. Die Approximation für die Nullstelle wird dann als Resultat zurückgeliefert.

Werden Vektoren als Funktionsparameter übergeben, so wird nur die Anfangsadresse des Vektors in den Stack kopiert. Eine Funktion kann also jederzeit den Inhalt bestimmter Feldelemente ändern. Die Deklarationen

```

int binsearch(double x, double v[], int n);
int binsearch(double x, double *v, int n);

```

sind also äquivalent. Bei der Übergabe eines Vektors darf *kein* Adressoperator verwendet werden. Auf diese Äquivalenz von Zeigern und Vektoren wird später noch eingegangen.

2 Felder, Strukturierte Datentypen und Zeiger

2.1 Felder

Felder können in C mit

```
<Speicherklasse> <Typ>  
    <Bezeichner> [ <Konstanten Ausdruck> ]
```

vereinbart werden. Die Angabe der Dimension der Felder erfolgt dabei in eckigen Klammern nach dem Bezeichner. Gültige Vereinbarungen für Felder wären

```
float  fvector[30];  
double dvector[30+1];  
char   string[256];
```

Felder beginnen grundsätzlich mit dem Index 0. Bei der Vereinbarung muß aber die Anzahl der Elemente angegeben werden. Bei dem im Beispiel vereinbarten Datum `dvector` sind also die Aufrufe `dvector[0]` bis `dvector[30]` zulässig. Es gibt keine Möglichkeit, die Verletzung der Feldgrenzen zu überprüfen. Felder können *nicht* als Ergebnis von Funktionen zurück geliefert werden. Als Argumente von Funktionen wird grundsätzlich nur die Anfangsadresse des Feldes übergeben. Funktionen können daher jederzeit den Inhalt von Feldelementen verändern. Die Dimension des Feldes braucht nicht angegeben zu werden. Mit

```
void clear_vector(int n, double v[])  
{ int i;  
  for (i=0; i<n; i++)  
    v[i]=0.0;  
}
```

können also beliebige Felder von `double`-Zahlen auf Null gesetzt werden. Da nur die Anfangsadresse in den Stack kopiert wird, werden die Werte in `v[i]` tatsächlich verändert. C verhält sich dabei so wie FORTRAN bei der Feldübergabe oder wie Pascal wenn die Felder als Wechselwertparameter übergeben werden.

In C besteht ein grundlegender Zusammenhang zwischen Zeigern und Feldern. Die Deklaration eines Feldes

```
double vector[30];
```

gibt dem Compiler an, daß er Speicher für 30 `double`-Zahlen bereitstellen soll, daß dieser Speicher bei `vector[0]` beginnt und nach jeweils `sizeof(double)` Bytes ein neues Element folgt. Gespeichert wird nur die Anfangsadresse des Feldes, also ein Zeiger auf `double`. Dieser Zeiger ist durch diese Vereinbarung eine Konstante. Ein Beispiel soll das verdeutlichen.

```
double vector[30];  
double *p_vector;  
  
p_vector=vector;  
for (i=0; i<30; i++)  
  p_vector[i]=(double) i;
```

`p_vector` ist ein Zeiger auf `double` dem die Anfangsadresse von `vector` zugewiesen wird. Nach dieser Zuweisung greifen die Aufrufe von `p_vector[i]` und `vector[i]` auf den selben Speicherbereich zu.

Wird ein Zeiger vom Typ $\langle Typ \rangle$ inkrementiert, so wird sein Wert um $sizeof(\langle Type \rangle)$ Bytes erhöht. Mit der Anweisung $p_vector+=1$ oder $p_vector++$ würde also $p_vector[0]$ auf das Element $vector[1]$ zeigen. Der C Compiler wandelt stets sämtliche Aufrufe von $vector[i]$ in $*(vector+i)$ um. Die Version $*(vector+i)$ bedeutet dabei: erhöhe den Zeiger $vector$ um $i*sizeof(double)$ Bytes und greife mit dem Inhaltsoperator $*$ auf diese Adresse zu. Das letzte Beispiel könnte also auch so aussehen.

```
double vector[30];
double *p_vector;

p_vector=vector;
for (i=0; i<30; i++,p_vector++)
    (*p_vector)=(double) i;

p_vector=vector; /* Zeiger wieder auf den alten Wert setzen. */
```

Die letzte Version ist etwas schneller, da die Multiplikationen ($i*sizeof(double)$) bei der Adressberechnung eingespart werden. Natürlich kann man damit erstklassige Fehler produzieren, sehr beliebt ist zum Beispiel:

```
int ivector[30];
int *p_ivector;

p_ivector=ivector;
    /* eigentlich soll hier p_ivector[0]+=1 */
p_ivector+=1;
    /* rueckt aber nur den Zeiger vor,
    richtig w"are es mit */
(*p_ivector)+=1;
```

Prinzipell können auch Zeiger dekrementiert werden, vorausgesetzt die Speicheradresse existiert.

```
double vector[30];
double *p_vector;

p_vector=(vector+2);
p_vector[-1]=0.0; /* setzt vector[1] auf 0 */
p_vector[-2]=0.0; /* setzt vector[0] auf 0 */
```

Allgemein ist für einen Zeiger p nicht garantiert, daß $(p-n)+n = p$ gilt. Bei Systemen mit einer Speichersegmentierung (zum Beispiel MS-DOS) kann es dabei zu einem unterschreiten der Segmentgrenze kommen, wenn p genau am Beginn eines Segments liegt. Die im Beispiel demonstrierte Methode ist allerdings sicher, da durch die Vereinbarung von $vector$ die Adresse $(p_vector-2)+2$ existiert.

Bei Argumenten von Funktionen gilt die Äquivalenz von Zeigern und Feldern auch. Die Deklarationen

```
void clear_vector1(int n, double v[])
{
    int i;
    for (i=0; i<n; i++)
        v[i]=0.0;
```

```

    }
void clear_vector2(int n, double *v)
{ int i;
  for (i=0; i<n; i++)
    v[i]=0.0;
}

```

sind völlig gleichwertig. Wenn man eine Feld übergeben will, sollte man die Variante `v[]` verwenden; wenn die Funktion in einem Nebeneffekt den Wert einer Variablen ändern soll, so ist die zweite Version der Deklaration günstiger.

2.2 Mehrdimensionale Felder

Die Vereinbarung mehrdimensionaler Felder erfolgt mit:

```

<Speicherklasse> <Typ>
<Bezeichner> [ <Konstanten Ausdruck>1 ]
                [ <Konstanten Ausdruck>2 ] ...

```

Die Feldelemente werden *zeilenweise* abgespeichert wie in Pascal. Die Vereinbarung

```
double mat[10][10];
```

reserviert also einen Speicherbereich von 100 `double`-Zahlen, die in der Folge `mat[0][0]`, `mat[0][1]`, `mat[0][2]`, ..., `mat[0][9]`, `mat[1][0]`, ... hintereinander im Speicher liegen. Werden mehrdimensionale Felder an Funktionen übergeben, kann der vordere Index fehlen. Zum Beispiel für die Multiplikation $C = AB$ der quadratischen Matrizen A und B mit der Dimension $(n, 10)$ ergibt sich eine Funktion der Gestalt:

```

void matrix_times(double a[][10], double b[][10], int n,
                 double c[][10])
{ int i,j,k;

  for (i=0; i<n; i++)
    for(j=0; j<n; j++)
      for(c[i][j]=0.0, k=0; k<n; c[i][j]+=a[i][k]*b[k][j], k++)
        ;
}

```

Bei zweidimensionalen Feldern ist `mat` ein Zeiger auf `double *`.

2.3 Aufzählungen

In C können Aufzählungstypen vereinbart werden mit

```
enum <Bezeichner> { <Aufzählungsliste> }
```

die *<Aufzählungsliste>* enthält entweder eine durch Kommas getrennte Liste von Bezeichnern oder Bezeichner gefolgt von einem Gleichheitszeichen und einer Konstanten. Mögliche Vereinbarungen sind

```

enum boolean { False, True };
enum months { Jan=1, Feb, Mar, Apr, May, Jun, Jul,
             Aug, Sep, Oct, Nov, Dec };

```

Die Namen in der *⟨Aufzählungsliste⟩* werden als Konstanten vom Typ `int` vereinbart, und sie können überall dort auftreten, wo Konstanten verlangt sind. Gibt es keine Aufzählungskonstante mit `=`, so beginnen die Werte der entsprechenden Konstante bei 0 und werden in Schritten von 1 in der Reihenfolge der Vereinbarung von links nach rechts weitergezählt. Eine Aufzählungskonstante mit `=` definiert für den angegebenen Namen den Wert; die nachfolgenden Namen werden von dort aus weitergezählt.

2.4 Strukturen

Eine Struktur ist eine Menge von Variablen unterschiedlichen Typs, die unter einem einzigen Namen zusammengefaßt werden. Strukturen sind das C Analogon zu den `record`-Variablen in Pascal. Strukturen werden mit

```
struct ⟨Bezeichner⟩ {
    ⟨Typ⟩1  ⟨Bezeichner⟩1;
    ⟨Typ⟩2  ⟨Bezeichner⟩2;
    ⋮
    ⟨Typ⟩N  ⟨Bezeichner⟩N;
};
```

vereinbart. Will man zum Beispiel die Koordinaten im dreidimensionalen Raum zu einer Struktur zusammenfassen, so wäre die Deklaration

```
struct point3d { float x;
                float y;
                float z; };
```

möglich. Sollen dann Variablen von diesem Typ vereinbart werden so kann das mit

```
struct point3d { float x;
                float y;
                float z; } p1,p2;
.
.
.
struct point3d  zerop,p4;
```

```
zerop.x=zerop.y=zerop.z=0.0f;
```

geschehen.

Mit dem Punkt-Operator wird (wie in Pascal) auf die einzelnen Komponenten zugegriffen (ein Analogon zur `with`-Anweisung in Pascal gibt es nicht).

Funktionen können auch Strukturen als Ergebnis haben. Eine einfache Arithmetik für komplexe Zahlen $z = x + iy$ ($i = \sqrt{-1}$) soll das verdeutlichen. Mit

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

struct Complex { double x,
                  y;
};
```

```

struct Complex cmplx(double x, double y)
{ struct Complex z;

    z.x=x;
    z.y=y;
    return z;
}

```

wird der komplexe Datentyp definiert. Die Funktion `cmplx` dient dazu, sich das einzelne Zuweisen der Komponenten zu ersparen. Addition $z = z_1 + z_2$ und Subtraktion $z = z_1 - z_2$ sind einfach zu implementieren.

```

struct Complex add(struct Complex z1, struct Complex z2)
{ struct Complex z;

    z.x=z1.x+z2.x;
    z.y=z1.y+z2.y;
    return z;
}

```

```

struct Complex minus(struct Complex z1, struct Complex z2)
{ struct Complex z;

    z.x=z1.x-z2.x;
    z.y=z1.y-z2.y;
    return z;
}

```

Für die komplexe Multiplikation von $z_1 = x_1 + iy_1$ und $z_2 = x_2 + iy_2$ gilt für $z = z_1 z_2 = (x_1 x_2 - y_1 y_2) + i(y_1 x_2 + x_1 y_2)$

```

struct Complex times(struct Complex z1, struct Complex z2)
{ struct Complex z;

    z.x=z1.x*z2.x-z1.y*z2.y;
    z.y=z1.y*z2.x+z1.x*z2.y;
    return z;
}

```

Bei der Berechnung des Betrages $|x + iy| = \sqrt{x^2 + y^2}$ muß man darauf achten keine Überläufe zu erzeugen. Daher wird jeweils der größere Wert x oder y ausgeklammert. Der günstigste Weg für die Berechnung ist durch

$$|x + iy| = \begin{cases} |x| \sqrt{1 + (y/x)^2} & |x| \geq |y| \\ |y| \sqrt{(x/y)^2 + 1} & |x| < |y| \end{cases}$$

gegeben.

```

double cabs( struct Complex z)
{ double t;

    if (z.x==0.0)
        return fabs(z.y);
    if (z.y==0.0)

```

```

    return fabs(z.x);
if (fabs(z.x)>fabs(z.y)) {
    t=z.y/z.x;
    t*=t;
    return fabs(z.x)*sqrt(1.0+t);
}
else {
    t=z.x/z.y;
    t*=t;
    return fabs(z.y)*sqrt(1.0+t);
}
}

```

Ein ähnliches Vorgehen ist bei der komplexen Division notwendig, da bei

$$\frac{x_1 + iy_1}{x_2 + iy_2} = \frac{(x_1x_2 + y_1y_2) + i(y_1x_2 - x_1y_2)}{x_2^2 + y_2^2}$$

erneut ein Fließkommaüberlauf möglich ist. Günstiger ist es deshalb mit

$$\frac{x_1 + iy_1}{x_2 + iy_2} = \begin{cases} \frac{(x_1 + y_1(y_2/x_2)) + i(y_1 - x_1(y_2/x_2))}{x_2 + y_2(y_2/x_2)} & |x_2| \geq |y_2| \\ \frac{(x_1(x_2/y_2) + y_1) + i(y_1(x_2/y_2) - x_1)}{x_2(x_2/y_2) + y_2} & |x_2| < |y_2| \end{cases}$$

zu arbeiten.

```

struct Complex cdiv(struct Complex z1, struct Complex z2)
{ struct Complex z;
  double den,t;

  if ((z2.x==0.0) && (z2.y==0.0)) {
    printf("Error division by zero in cdiv()");
    return z1;
  }
  if (z2.y==0.0) {
    t=1.0/z2.x;
    z.x=z1.x*t;
    z.y=z1.y*t;
    return z;
  }
  if (z2.x==0.0) {
    t=1.0/z2.y;
    z.x=z1.y*t;
    z.y=-z1.x*t;
    return z;
  }
  if (fabs(z2.x)>=fabs(z1.y)) {
    t=z2.y/z2.x;
    den=1.0/(z2.x+z2.y*t);
    z.x=(z1.x+z1.y*t)*den;
    z.y=(z1.y-z1.x*t)*den;
    return z;
  }
}

```

```

else {
    t=z2.x/z2.y;
    den=1.0/(z2.x*t+z2.y);
    z.x=(z1.x*t+z1.y)*den;
    z.y=(z1.y*t-z1.x)*den;
    return z;
}
}

```

2.5 Unionen

Eine Union ist ein Variable, die Objekte verschiedenen Types verbindet, Im Gegensatz zu Strukturen geschieht das nicht in separaten Speicherzellen wie bei Strukturen sondern in sich überlappenden Speicherzellen. Eine Union ist dem Variant-record von Pascal ähnlich – es fehlt aber der Speicherplatz für den case Selektor. Die Vereinbarung erfolgt mit

```

union <Bezeichner> {
    <Typ>1 <Bezeichner>1;
    <Typ>2 <Bezeichner>2;
    ⋮
    <Typ>N <Bezeichner>N;
};

```

Wie bei Strukturen werden die Alternative mit dem Punktoperator ausgewählt. Ein sinnvolles Beispiele wäre der Fall unseres dreidimensionalen Punktes. Um Rotationen im Dreidimensionalen durchzuführen, ist die gleichzeitige Indizierung der Koordinaten als Feld sinnvoll.

```

union point3d { struct p {float x,y,z};
                float v[3]; };

```

```

union point3d p1,p2;

```

Sollen Vektortransformationen mit den Punkten p1 und p2 durch geführt werden sind die räumlichen Koordinaten unter p1.v[i] anzusprechen. Will man direkt auf die Koordinaten Bezug nehmen, so mit p1.p.x. Es gilt, daß sich jeweils p1.v[0] und p1.p.x, p1.v[1] und p1.p.y und p1.v[2] und p1.p.z auf denselben Speicherbereich beziehen.

2.6 typedef Vereinbarung

Neue Datentypen können mit typedef vereinbart werden. Der Syntax dafür lautet

```

typedef <Typvereinbarung> <Typbezeichner>;

```

Hier einige Beispiele

```

typedef char * String;

```

```

typedef double vector[20];
typedef vector matrix[20];

```

mit String ist damit eine neuer Datentyp vereinbart, der äquivalent zu einem Zeiger auf char ist, vector vereinbart ein Feld von 20 double Zahlen. In

der darauf folgenden Zeile wird dann ein Datentyp `matrix` vereinbart, der eine 20×20 Matrix beschreibt.

Besonders nützlich erweist sich `typedef` bei Strukturen. Im Beispiel der komplexen Arithmetik war es etwas lästig, ständig `struct Complex` in den Funktionsargumenten und Deklarationen schreiben zu müssen. Mit `typedef` kann man vereinfachen

```
typedef struct complex { double x,y; } Complex;
```

Damit können (und müssen) sämtliche `struct` wegfallen, da der neue Datentyp `complex` zum Synonym für `struct complex` geworden ist. Der *<Typbezeichner>* muß sich nicht von dem Strukturnamen unterscheiden. Möglich und sinnvoll ist auch

```
typedef struct Complex { double x,y; } Complex;
```

Rekursive Datenstrukturen wie lineare Listen oder Binärbäume können ebenfalls mit `typedef` vereinbart werden. Zwei Möglichkeiten, diese beiden rekursiven Datenstrukturen zu vereinbaren seien jeweils demonstriert. Die lineare Liste läßt sich mit

```
/* Verion 1 - lineare Liste */
typedef struct lin_lst1 * LstPtr;
typedef struct lin_lst1 {
    char *what;
    LstPtr next;
} lin_lst1;

/* Version 2 - lineare Liste */
typedef struct lin_lst2 {
    char *what;
    struct lin_lst2 *next;
} lin_lst2;
```

deklarieren. Der Binärbaum kann mit

```
/* Verion 1 - Binaerbaum */
typedef struct bin_tree1 * TreePtr;
typedef struct bin_tree1 {
    char * what;
    TreePtr left;
    TreePtr right;
} bin_tree1;

/* Verion 2 - Binaerbaum */
typedef struct bin_tree2 {
    char *what;
    struct bin_tree2 *left;
    struct bin_tree2 *right;
} bintree2;

bintree2 *bin_tree_ptr;
```

vereinbart werden. Die jeweils erste Variante ist Pascal ähnlicher, die zweite kürzer. Um auf den Inhalt der Elemente eines Zeigers auf eine Struk-

zur zuzugreifen ist eine Konstruktion `(*bin_tree_ptr).what` erforderlich, da der Inhaltsoperator eine geringere Priorität hat als der „.“ Operator. Als Abkürzung für diese Konstruktion existiert der `->` Operator. Die Verwendung von `(*bin_tree_ptr).what` entspricht `bin_tree_ptr->what` im Beispiel.

2.7 Lineare Listen – ein Beispiel

Um rekursive Datenstrukturen in C etwas genauer zu betrachten seien einige Funktionen für eine lineare Liste von Zeichenketten implementiert. Die Datenstruktur besteht aus einer Struktur, die eine Zeichenkette `char *str` und einen Zeiger auf den Nachfolger der Liste enthält. Die Definition dieser Struktur sieht so aus

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

typedef struct s_lst {  char *str;
                       struct s_lst *next; } s_lst;
```

Um die Liste aufzubauen, benötigen wir Routinen aus der Standardbibliothek `<stdlib.h>`. Diese Routinen sollen dynamisch Speicher anlegen und freigeben. Mit der Bibliotheksroutine `void * malloc(size_t size)` wird ein Speicherbereich der Größe `size` reserviert. `malloc()` gibt einen Zeiger auf den Beginn dieses Speicherbereiches zurück. Mit der Bibliotheksfunktion `void free(void * ptr)` kann der mit `malloc()` reservierte Speicher wieder freigegeben werden. Dem `NIL` von LISP und dem `nil` von Pascal entspricht in C `NULL`.

Zum Aufbau einer neuen Liste dient die Funktion `Create`, sie reserviert zuerst Speicher für das Listenelement und dann für den String `str`. Der Inhalt des Strings `s` muß dann nach `str` kopiert werden. Um eine vollständige Liste mit einem Element zu erhalten, ist es dann notwendig, den `next` Zeiger auf `NULL` zu setzen. `Create` liefert dann einen Zeiger auf die erzeugte Liste zurück.

```
void error(char *error_msg)
{
    printf("Error in string list ...\n");
    printf("%s\n",error_msg);
}

s_lst * Create(char *s)
{
    s_lst *lst;

    lst=(s_lst *) malloc((size_t) sizeof(s_lst));
    if (!lst) {
        error("failed to allocate memory 1 in Create.");
        return lst;
    }
    lst->str=
        (char *) malloc((size_t) (strlen(s)+1)*sizeof(char) );
    if (!lst) {
        error("failed to allocate memory 2 in Create.");
        free(lst);
        return NULL;
    }
}
```

```

    }

    strcpy(lst->str,s);

    lst->next=NULL;
    return lst;
}

```

Es gibt zwei Möglichkeiten, weitere Listenelemente anzufügen. Prepend erzeugt ein neues Listenelement mit `first_str` und fügt es an den Anfang der bereits existierenden Liste `tail` an. Die neue Liste wird als Ergebnis zurück geliefert. Die zweite Variante, die Liste zu vergrößern, besteht im Anhängen eines Listenelements. Append leistet genau dies.

```

s_lst * Prepend(s_lst *tail, char *first_str)
{ s_lst *head;

    head=Create(first_str);
    if (!head) {
        error("failed to get memory in Prepend.");
        return tail;
    }
/* Verbinden des neuen Kopfes mit
dem Rest der Liste.          */
    head->next=tail;
    return head;
}

s_lst * Append(s_lst *head, char *last_str)
{ s_lst *lst, *tail;

    tail=Create(last_str);
    if (!tail) {
        error("failed to get memory in Append.");
        return head;
    }
/* Zum Ende der Liste wandern ... */
    for (lst=head; lst->next; lst=lst->next)
        ;
/* und die Verbindung herstellen. */
    lst->next=tail;
    return head;
}

```

Früher oder später gilt es, sich der erzeugten Listen wieder zu entledigen. Mit Destroy wird der Speicher der gesamten Liste freigegeben. Der Leser beachte die Reihenfolge der Speicherfreigaben mit `free`. Aus rein ästhetischen Gründen gibt Destroy eine NULL zurück.

```

s_lst * Destroy(s_lst *lst)
{ s_lst *tail;

    while (lst) {

```

```

        tail=lst->next; /* Sichern des Listenendes. */
        free((void *) lst->str); /* Speicherfreigeben fuer
                                str und dann fuer den
                                Listenkopf. */

        free((void *) lst);
        lst=tail;
    }
    return NULL;
}

```

Eine etwas kompliziertere Operation ist das Umkehren der Liste, ohne eine Kopie zu erzeugen. Reverse realisiert eine solche „in place“ Umkehrung. Man beachte die Verwendung der Rekursion in diesem Beispiel.

```

s_lst * Reverse(s_lst *lst)
{
    s_lst *head,*tail;

    if (!lst)
        return NULL; /* Leere Liste. */
    if (!lst->next) /* Nur ein Element. */
        return lst;
    /* Den Rest der Liste umkehren. */
    head=tail=Reverse(lst->next);
    /* An das Ende der umgekehrten Liste gehen ... */
    while (tail->next)
        tail=tail->next;
    /* ... den alten Kopf anhängen, ... */
    tail->next=lst;
    /* und ordentlich abschliessen. */
    lst->next=NULL;
    return head;
}

```

Recht trivial ist die Bestimmung der Länge einer Liste mit Length.

```

int Length(s_lst *lst)
{
    int len;

    if (!lst)
        return 0;
    for (len=0; lst; lst=lst->next, len++)
        ;
    return len;
}

```

Die Funktion Pop ist eine Funktion mit einem Nebeneffekt. Sie liefert den Listenkopf als Liste mit einem Element zurück und schneidet gleichzeitig den Kopf von der Liste ab. Hier sei die Aufmerksamkeit auf die Verwendung eines Zeigers auf den Zeiger des Listenanfanges gelenkt, um die Modifikation der Liste selbst zu realisieren.

```

s_lst * Pop(s_lst **lst)
{
    s_lst *head;

```

```

    head>(*lst);          /* Listenkopf sichern.          */
    (*lst)=(**lst).next; /* Liste modifizieren.          */
    head->next=NULL;     /* Listenkopf zu einer Liste
                           mit einem Element machen. */
    return head;
}

```

Eine Funktion, die eine Kopie einer bereits vorhandenen Liste erstellt, ist Copy.

```

s_lst * Copy(s_lst *lst)
{ s_lst *clst;

    if (!lst)
        return NULL;

    for (clst=Create(lst->str),lst=lst->next; lst; lst=lst-
>next)
        clst=Append(clst,lst->str);
    return clst;
}

```

Die Funktion Join liefert eine *neue* Liste, die aus den aneinandergelagerten Kopien von lst1 und lst2 besteht.

```

s_lst * Join(s_lst *lst1, s_lst *lst2)
{ s_lst *tail, *clst1, *clst2;

    clst1=Copy(lst1);
    clst2=Copy(lst2);
    tail=clst1;
    /* Zum Ende von lst1 gehen ... */
    while (tail->next)
        tail=tail->next;
    /* ... und das Ende mit dem Anfang verkn"upfen. */
    tail->next=clst2;
    return clst1;
}

```

Zur Ausgabe der Liste dient die Funktion PrintList. NULL soll dabei (LISP-artig) als „()“gedruckt werden, sonst sollen die Listenelemente von Klammern umschlossen sein und entweder durch Komma (print_komma=1) oder Leerzeichen getrennt werden. Hier sei die Aufmerksamkeit auf die Verwendung der static Variablen recursion gelenkt. recursion behandelt die Sonderfälle am Anfang und am Ende der Liste.

```

void PrintList(s_lst *lst, int print_komma)
{ static int recursion=0;

    if ((!lst) && (!recursion)) { /* Leere Liste. */
        printf("()\n");
        return;
    }
    if (!lst) { /* Ende einer Liste beim rekursiven Auf-
ruf. */

```

```

    printf("\n");
    return;
}
if (!recursion) { /* Neue Liste */
    printf(" ( ");
    recursion=1; /* recursion Flag setzen. */
    printf("%s",lst->str);
    if ((print_komma) && (lst->next))
        printf(", ");
    else
        printf(" ");
    /* Beginn der Rekursion. */
    PrintList(lst->next,print_komma);
    recursion=0; /* recursion Flag zurueck setzen.*/
    return;
}
printf("%s",lst->str); /* normale Rekursion */
if ((print_komma) && (lst->next))
    printf(", ");
else
    printf(" ");
PrintList(lst->next,print_komma);
}

```

Als Abschluß noch ein Testprogramm, das die Verwendung der Funktionen demonstriert.

```

int main() {
    s_lst *l,*l1,*l2;

    l=Create("!");
    l=Prepend(l,"World");
    l=Prepend(l,"Hello");
    PrintList(l,0); /* #1 */
    l=Destroy(l);

    l=Append(Append(Append(
        Append(Append(Create(
            "Dog"), "Cat"), "Mouse"), "Lion"), "Horse"), "Gnu");
    l1=Append(Append(
        Append(Append(Create(
            "Apfel"), "Cherry"), "Pear"), "Plum"), "Lemon");
    PrintList(l,1); /* #2 */
    l=Reverse(l);
    PrintList(l,1); /* #3 */
    printf("%d\n",Length(l)); /* #4 */
    PrintList(l2=Pop(&l),1); /* #5 */
    PrintList(l,1); /* #6 */
    l2=Destroy(l2);
    l2=Join(l,l1);
    PrintList(l1,1); /* #7 */
    l=Destroy(l);
}

```

```
l1=Destroy(l1);
PrintList(l2,1);          /* #8 */
l2=Destroy(l2);
return 0;
}
```

Die Ausgabe des Testprogramms sieht so aus:

```
1 ( Hello World ! )
2 ( Dog, Cat, Mouse, Lion, Horse, Gnu )
3 ( Gnu, Horse, Lion, Mouse, Cat, Dog )
4 6
5 ( Gnu )
6 ( Horse, Lion, Mouse, Cat, Dog )
7 ( Appel, Cherry, Pear, Plum, Lemon )
8 ( Horse, Lion, Mouse, Cat, Dog, Appel, Cherry, Pear, Plum, Lemon )
```

3 Der Preprozessor

Die Compilierung eines C-Programmes erfolgt in mehreren Schritten. Bevor ein C-Programm dem Syntax-Parser übergeben wird, expandiert der Preprozessor die mit # eingeleiteten Makrodefinitionen, include-Anweisungen und Anweisungen zur bedingten Compilierung.

Die Verwendung des Preprozessorbefehles #include ermöglicht es, in das C-Programm eine andere Datei einzusetzen. Die häufigste Verwendung dieses Befehls besteht im Einfügen der Definitionen der Standard-Bibliotheken. Die Zeile

```
#include <Dateiname>
```

wird durch den Preprozessor mit Inhalt der Datei *Dateiname* ersetzt. Nach dieser Datei wird an einer implementationsabhängigen Folge von Stellen gesucht (meist im Verzeichnis /include). Mit

```
#include "Dateiname"
```

wird die Datei zuerst im Verzeichnis der Quelldatei gesucht. Ist diese Suche erfolglos, so wird entsprechend der ersten Form weiter gesucht.

Die Zeichen ", ', \ und /*, dürfen nicht im Namen *Dateiname* vorkommen.

3.1 Makrodefinitionen

Eine Preprozessoranweisung der Form

```
#define <Bezeichner> <Token-Sequence>
```

veranlaßt den Preprozessor, alle auf diese Definition folgenden Vorkommen von <Bezeichner> im Quelltext durch <Token-Sequence> zu ersetzen. Mit

```
#define <Bezeichner>(<Bezeichner-Liste>) <Token-Sequence>
```

wird ein Makro mit Parametern definiert, die durch <Bezeichner-Liste> festgelegt werden. Die öffnende Klammer muß dem <Bezeichner> unmittelbar (ohne Leerzeichen) folgen. Um eine definiertes Makro wieder zu entfernen, wird die Sequenz

```
#undef <Bezeichner>
```

verwendet. Es ist kein Fehler, wenn #undef auf einen unbekannt Namen angewendet wird. Als Beispiel für parameterlose Makros seien

```
#define MAX_N    20
#define PI      3.1415926
#define FALSE   0
#define TRUE    1
```

```
typedef double vector[MAX_N];
typedef vector matrix[MAX_N];
```

angeführt. Die Verwendung von Makros mit Parametern hat den Vorteil, daß man sie unabhängig vom Variablentyp verwenden kann und man sich den Aufwand für einen Funktionsaufruf spart. Hier einige Beispiele

```
#define ABS(x)    ((x)>0 ? (x) : -(x))
#define MAX(x,y) ((x)>(y) ? (x) : (y))
#define SQR(x)    ((x)*(x))
```

```
#define NEWSTRING(n) (char *)malloc((size_t) (n)*sizeof(char))
```

Sowohl ABS als auch MAX arbeiten unabhängig vom Typ der aktuellen Parameter. Der Makro NEWSTRING spart in erster Linie Schreibarbeit, wenn ein neuer String dynamisch erzeugt werden soll.

Die Verwendung von Makros hat allerdings auch Nachteile. Der Preprozessor expandiert die Argumente und wertet sie dabei eventuell mehrfach aus. Handelt es sich bei den Argumenten um Funktionen mit einem Nebeneffekt, so führt das zu schönen, schwer zu findenden Fehlern. Um wenigstens einfache Fehler zu vermeiden, sollten beim Aufruf die Argumente immer geklammert werden. Die Probleme, die Nebeneffekte mit sich bringen, sind damit nicht beseitigt. Der Aufruf

```
y=MAX(x+=2.0, z);
```

würde zu

```
y=((x+=2.0)>(z) ? (x+=2.0) : (z));
```

expandieren und im Falle $x > z - 2$, x um 4.0 erhöhen, statt der erwarteten Vergößierung um 2.0.

3.2 Bedingte Compilierung

Bedingte Compilierung wird vor allem eingesetzt, um eine gewisse Unabhängigkeit des Quelltextes von der Hardware oder dem verwendeten Compiler zu erzielen. Mit

```
#if <Konstanter-Ausdruck>
#ifdef <Konstanter-Ausdruck>
#ifndef <Konstanter-Ausdruck>
#ifdef <Konstanter-Ausdruck>
...
#else
...
#endif
```

können definierte Konstanten abgefragt werden, und entsprechende Modifikationen des Programmcodes eingesetzt werden. Zu jedem `#if` muß ein `#endif` vorhanden sein. Darüber hinaus ist es möglich, statt `#else` auch `#elif` zu verwenden und mit

```
defined <Bezeichner>
defined (<Bezeichner>)
```

das Vorhandensein bestimmter Bezeichner abzufragen. Ist ein Bezeichner definiert, so liefert `defined` 1L, sonst 0L. In den `<Konstanter-Ausdruck>` der `#if`-Konstruktionen findet vor der Auswertung eine normale Makroexpansion statt. Um zum Beispiel einen Pfad unter MS-DOS oder UNIX zu vereinbaren, sind die Definitionen

```
#undef MSDOS
#define UNIX

#ifdef UNIX
#define PATH "~/world/font"
#elif defined MSDOS
```

```
#define PATH "C:\\world\\font"  
#endif
```

sinnvoll.

3.3 Weitere Preprozessorbefehle

Mit

```
#line <Konstante> "<Dateiname>"
```

```
#line <Konstante> "<Dateiname>"
```

kann dafür gesorgt werden, daß der Compiler die (modifizierte) Zeilennummer in einer Fehlermeldung angibt. Dies ist besonders nützlich, wenn man mit anderen Programmen C-Programme erzeugt. Die Anweisung veranlaßt die Ausgabe der Fehlermeldung (*Fehlermeldung*).

Eine Kontrollzeile der Form

```
#pragma <Compileranweisung>
```

veranlaßt den Preprozessor, eine implementationsabhängige Aktion auszuführen. Meist sind Compilerschalter auch als `pragma` zu setzen. Unbekannte *<Compilerschalter>* werden ignoriert.

Der Preprozessor kennt die vordefinierten Bezeichner

```
__LINE__ /* aktuelle Zeilennummer */  
__FILE__ /* Name der Datei, die gerade compiliert wird */  
__DATE__ /* Datum */  
__TIME__ /* Uhrzeit */  
__STDC__ /* 1 wenn der Compiler dem Standard entspricht,  
          0 sonst */
```

4 Standard-Bibliothek

Die Funktionen der Standard-Bibliothek müssen bei jedem ANSI-C Compiler vorhanden sein. Soll das Programm portabel sein, dürfen nur diese Bibliotheksfunktionen verwendet werden. Zur Standard-Bibliothek gehören die Header

```
<assert.h> <float.h> <math.h> <stdarg.h> <stdlib.h>  
<ctype.h> <limits.h> <setjmp.h> <stddef.h> <string.h>  
<errno.h> <locale.h> <signal.h> <stdio.h> <time.h>
```

Auf diese Header wird mit

```
#include <header>
```

zugegriffen.

4.1 Ein- und Ausgabe: `<stdio.h>`

Ein Datenstrom ist eine Quelle oder ein Ziel von Daten, die mit einer Datei oder einem Ausgabegerät verbunden ist. Es gibt zwei Arten von Datenströmen, binäre Information und Text. Ein Text besteht aus einer Folge Zeilen, die aus Zeichen

bestehen und mit `\n` abgeschlossen sind. Ein Binärstrom ist eine Folge un bearbeiteter Bytes zur Aufzeichnung interner Daten. Binärdateien sind normalerweise hardwareabhängig.

Wird ein Datenstrom geöffnet und mit einem Gerät verbunden, so erhält man einen Zeiger vom Typ `FILE`, in dem die Kontrollinformationen des Datenstroms hinterlegt sind. Wenn ein Programm ausgeführt wird, so sind die drei Datenströme `stdin`, `stdout` und `stderr` bereits geöffnet. Wird die Ein- und Ausgabe nicht umgeleitet, so ist `stdin` mit der Tastatur verknüpft und `stdout` mit dem Bildschirm. Sowohl `stdin` als auch `stdout` sind gepuffert.

4.1.1 Dateioperationen

Mit

```
FILE *fopen(const char *⟨Dateiname⟩, const char *⟨mode⟩)
```

wird die Datei `⟨Dateiname⟩` geöffnet und liefert einen Zeiger auf einen Datenstrom oder `NULL` bei Mißerfolg. Für `⟨mode⟩` sind die erlaubten Werte:

"r"	Textdatei zum Lesen öffnen
"w"	Textdatei zum Schreiben öffnen oder erzeugen
"a"	anfügen, Textdatei zum Schreiben am Dateiende, öffnen oder erzeugen
"r+"	Textdatei zum Ändern öffnen (Lesen und Schreiben)
"w+"	Textdatei zum Ändern erzeugen (vorhandene Dateien überschreiben)
"a+"	anfügen, Textdatei zum Ändern am Dateiende, öffnen oder erzeugen

Enthält `⟨mode⟩` nach diesen Zeichen noch das Zeichen `b` (zum Beispiel `"rb"` oder `"w+b"`), so wird auf eine binäre Datei zugegriffen.

```
FILE *freopen(const char *⟨Dateiname⟩,  
             const char *⟨mode⟩, FILE *⟨stream⟩)
```

öffnet eine Datei mit dem angegebenen Zugriff `⟨mode⟩` und verknüpft `⟨stream⟩` damit. Mit `freopen` werden normalerweise die mit `stderr`, `stdin` oder `stdout` verknüpften Dateien geändert.

```
int fflush(FILE *⟨stream⟩)
```

erzwingt ein Schreiben gepufferter, aber noch nicht geschriebener Daten eines Ausgabestroms `⟨stream⟩`. Die Funktion liefert `EOF` bei einem Fehler, sonst `Null`.

```
int fclose(FILE ⟨stream⟩)
```

schreibt noch nicht geschriebene Daten, beseitigt noch nicht gelesene Eingabedaten, gibt den Puffer frei und schließt den Datenstrom. Bei einem Fehler wird `EOF` als Ergebnis geliefert, sonst `Null`.

```
int remove(const char *⟨Dateiname⟩)
```

löscht die Datei `⟨Dateiname⟩`. Bei einem Fehler wird `EOF` als Ergebnis geliefert, sonst `Null`.

```
int rename(const char *⟨Alter Name⟩,  
          const char *⟨Neuer Name⟩)
```

Zeichen	Argument, Umwandlung
d, i	int, dezimal mit Vorzeichen
o	int, oktal ohne Vorzeichen (ohne führende Null)
x, X	int, hexadezimal ohne Vorzeichen (ohne führende 0x oder 0X)
u	int, dezimal ohne Vorzeichen
c	int, einzelnes Zeichen nach Umwandlung in unsigned char
s	char *, aus einer Zeichenkette werden Zeichen ausgegeben bis vor \0
f	double, dezimal als ±mmm.ddd, die Genauigkeit legt die Anzahl der d fest.
e, E	double, dezimal als ±mmm.dddde±xx, die Genauigkeit legt die Anzahl der d fest.
g, G	double, %e oder %E wird verwendet wenn der Exponent kleiner als -4 oder nicht kleiner als die Genauigkeit ist; sonst wird %f benutzt.
p	void *; als Zeiger
n	int *; die Anzahl der bisher von printf ausgegebenen Zeichen wird im Argument abgelegt.
%	kein Argument wird umgewandelt; ein % wird ausgegeben.

Tabelle 2: printf-Umwandlungen

ändert den Namen einer Datei. Bei einem Fehler wird ein Wert verschieden von Null als Ergebnis geliefert.

```
FILE *tmpfile(void)
```

erzeugt eine temporäre Datei mit dem Zugriff "w+b", die automatisch gelöscht wird, wenn der Zugriff abgeschlossen ist oder das Programm normal beendet wird. tmpfile liefert NULL, wenn der Datenstrom nicht erzeugt werden konnte.

```
char *tmpnam(char s[L_tmpnam])
```

erzeugt eine Zeichenkette, die nicht der Name einer existierenden Datei ist. tmpnam(s) speichert die Zeichenkette in s und liefert als Resultat auch s.

```
int setvbuf(FILE *⟨stream⟩, char *⟨Puffer⟩,
            int ⟨mode⟩, size_t ⟨Size⟩)
```

setvbuf kontrolliert die Pufferung bei einem Datenstrom; die Funktion muß vor dem ersten Lese- und Scheibzugriffen aufgerufen werden. Hat ⟨mode⟩ der Wert _IOBF, so wird vollständig gepuffert, _IOBF sorgt für eine zeilenweise Pufferung bei Textdateien und _IONBF verhindert das Puffern. Ist ⟨Puffer⟩ nicht NULL, so wird ⟨Puffer⟩ verwendet. ⟨Size⟩ legt die Puffergröße fest. Bei einem Fehler wird ein Wert verschieden von Null als Ergebnis geliefert.

```
void setbuf(FILE *⟨stream⟩, char *⟨Puffer⟩)
```

Wenn ⟨Puffer⟩ den Wert NULL hat, wird nicht gepuffert, sonst ist setbuf äquivalent zu (void) setvbuf(stream, buf, _IOBF, BUFSIZE).

4.1.2 Formatierte Ausgabe

Eine formatierte Ausgabe für Text-Dateien erfolgt mit

```
int fprintf(FILE *⟨stream⟩, const char *⟨Format⟩, ...)
```

`fprintf` wandelt die Parameter in Zeichen um und schreibt sie in $\langle stream \rangle$. Der Resultatwert ist die Anzahl der geschriebenen Zeichen, der Wert ist negativ, wenn es zu einem Fehler kam.

Das $\langle Format \rangle$ ist eine Zeichenkette, die aus Formatierungsangaben und gewöhnlichen Zeichen besteht. Jede Formatangabe beginnt mit einem `%`-Zeichen und endet mit einem Formatzeichen. Zwischen dem `%` und dem Formatzeichen kann der Reihe nach folgendes angegeben sein.

- Steuerzeichen

- : linksbündig ausgeben

- + : immer mit Vorzeichen ausgeben

Leerzeichen : wenn das erste Zeichen kein Vorzeichen ist, wird ein Leerzeichen vorangestellt

- 0 : bei numerischen Umwandlungen wird die Feldbreite mit Nullen aufgefüllt

- eine Zahl, die die minimale Feldbreite festlegt

- ein Punkt, der die Feldbreite von der Genauigkeit trennt

- eine Zahl, die die maximale Anzahl von Zeichen festlegt, die von einer Zeichenkette ausgegeben werden können, oder die Zahl der Ziffern, die nach dem Dezimalpunkt ausgegeben werden

- ein Buchstabe als Längenangabe: `h,l` oder `L`, `h` entspricht der Ausgabe als `short` oder `unsigned short`, `l` entspricht der Ausgabe als `long` oder `unsigned long` und `L` entspricht der Ausgabe als `long double`.

Die Feldbreite und/oder Genauigkeit kann jeweils als `*` vereinbart werden; dann wird die Feldbreite und/oder Genauigkeit durch die nächsten zwei/ein Argumente festgelegt, die den Wert `int` haben.

```
int printf(const char * $\langle Format \rangle$ , ...)
```

`printf` ist äquivalent zu `fprintf(stdout, ...)`.

```
int sprintf(char * $\langle String \rangle$ , const char * $\langle Format \rangle$ , ...)
```

`sprintf` arbeitet wie `fprintf`, nur wird die Ausgabe in die Zeichenkette $\langle String \rangle$ geschrieben. Mit `sprintf` können Umwandlungen einer Zahl in eine Zeichenkette realisiert werden.

```
vprintf(const char * $\langle Format \rangle$ , va_list arg);  
vfprintf(FILE * $\langle stream \rangle$ , const char * $\langle Format \rangle$ ,  
va_list arg);  
vsprintf(char * $\langle String \rangle$ , const char * $\langle Format \rangle$ ,  
va_list arg);
```

Diese Funktionen sind äquivalent zu `printf`, `fprintf` und `sprintf`, jedoch wird die variable Argumentliste durch `arg` ersetzt.

Zeichen	Eingabedaten, Argumenttyp
d	dezimal, ganzzahlig, int *
i	ganzzahlig; int *, der Wert kann oktal (mit 0 am Anfang) oder hexadezimal (mit 0x oder 0X) angegeben werden
o	oktal ganzzahlig, int *
x	hexadezimal ganzzahlig, int *
u	dezimal ohne Vorzeichen, int *
c	ein oder mehrere Zeichen, char *, die Eingabezeichen werden in einem Vektor abgelegt bis die Feldbreite erreicht ist (Voreinstellung 1), eine \0 wird nicht angefügt.
s	Folge von Nicht-Zwischenraumzeichen ohne ", char *; eine abschließende \0 wird angefügt.
e, f, g	Gleitpunkt, float *
p	Zeiger wie ihn printf ausgibt, void *
[...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen in der angegebenen Klasse; char *
[^...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen <i>nicht</i> in der angegebenen Klasse; char *
%	erkennt ein %; keine Zuweisung

Tabelle 3: scanf-Umwandlungen

4.1.3 Formatierte Eingabe

Diese Funktionen behandeln die Eingabe von Daten unter Formatkontrolle.

```
int fscanf(FILE *⟨stream⟩, const char *⟨Format⟩, ...)
```

`fscanf` liest vom Datenstrom `⟨stream⟩` unter der Kontrolle von `⟨format⟩` und legt die umgewandelten Werte mit Hilfe der Argumente ab, *die alle Zeiger sein müssen*. `fscanf` liefert EOF, wenn vor der ersten Umwandlung das Dateiende erreicht wird oder ein Fehler auftritt, sonst wird die Anzahl der umgewandelten Argumente geliefert.

Die Formatzeichenkette enthält die Angaben zur Interpretation der Eingabe. Sind in der Formatzeichenkette noch andere Zeichen enthalten so gelten die Konventionen

- Leerzeichen und Tabulatoren werden ignoriert
- gewöhnliche Zeichen (außer %), diese müssen dem nächsten Zeichen nach einem Zwischenraum entsprechen.

Die Umwandlungsangaben bestimmen die Umwandlungen des Eingabefeldes. Das Resultat wird im Inhalt der Zeiger der Argumentliste abgelegt. Mit * wird die Zuweisung verhindert und dieser Eintrag übergangen. So würde

```
fscanf(f, "%*s %g", str, &x);
```

eine am Anfang der Zeile stehende Zeicherkette überlesen und dann eine Zahl gelesen und in `x` abgelegt. Den Formatzeichen `d`, `i`, `n`, `o`, `u` und `x` kann ein `h` vorausgehen, wenn das Argument ein Zeiger auf `short` ist, oder der Buchstabe `l`, wenn es sich um einen Zeiger auf `long` handelt. Vor `e`, `f` und `g` kann ein `l` stehen, wenn das Argument ein Zeiger auf `double` ist und ein `L` wenn es sich um einen Zeiger auf `long double` handelt.

```
int sscanf(FILE *stream, const char *Format, ...)
```

ist äquivalent zu `scanf`; die Eingabezeichen jedoch aus einer Zeichenkette stammen.

4.1.4 Ein- und Ausgabe von Zeichen

```
int fgetc(FILE *stream)
```

```
int getc(FILE *stream)
```

`fgetc` liefert das nächste Zeichen in *stream* als `unsigned char` oder EOF bei einem Fehler. `getc` ist äquivalent zu `fgetc`.

```
char * fgets(char *string, int n, FILE *stream)
```

liest eine Zeile aus *stream*, es werden nicht mehr als $n - 1$ Zeichen gelesen. Der Zeilentrenner wird im Vektor mit abgelegt. `fgets` liefert NULL bei einem Fehler oder *string*.

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

```
int putc(int c, FILE *stream)
```

schreibt das Zeichen in *stream*, das Ergebnis ist `c` oder EOF bei einem Fehler.

```
char * fputs(char *string, FILE *stream)
```

schreibt eine Zeile (die `\n` nicht zu enthalten braucht) in *stream*. `fputss` liefert EOF bei einem Fehler.

```
int getchar(void)
```

entspricht `getc(stdin)`.

```
char * gets(char *string)
```

liest die nächste Zeile von `stdin` in den Vektor *string* und ersetzt den Zeilentrenner durch `\0`.

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

entspricht `fputc(c, stdout)`.

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

`puts` schreibt die Zeichenkette *string* und einen Zeilentrenner in `stdout`. Die Funktion liefert im Falle eines Fehlers EOF, sonst einen nichtnegative Wert.

```
int ungetc(int char, FILE *stream)
```

schreibt *char* in den *stream* zurück. Nur ein Zeichen kann zurückgestellt werden. EOF darf nicht zurück geschrieben werden.

4.1.5 Direkte Ein- und Ausgabe

```
size_t fread(void *⟨Ziel⟩, size_t ⟨Size⟩,  
             size_t ⟨Anzahl⟩,  
             FILE *⟨stream⟩)
```

`fread` liest aus `⟨stream⟩` in den Vektor `⟨Ziel⟩` höchstens `⟨Anzahl⟩` Objekte der Größe `⟨Size⟩`. `fread` liefert die Anzahl der gelesenen Objekte. Der Zustand des Datenstromes muß mit `feof` oder `ferror` untersucht werden.

```
size_t fwrite(void *⟨Quelle⟩, size_t ⟨Size⟩,  
             size_t ⟨Anzahl⟩,  
             FILE *⟨stream⟩)
```

`fwrite` schreibt in `⟨stream⟩` vom Vektor `⟨Quelle⟩` `⟨Anzahl⟩` Objekte der Größe `⟨Size⟩`. Bei einem Fehler sind das weniger als `⟨Anzahl⟩`.

4.1.6 Positionieren in Dateien

```
int fseek(FILE *⟨stream⟩, long ⟨offset⟩, int ⟨origin⟩)
```

`fseek` setze die Dateiposition für `⟨stream⟩`; beim weiteren Lesen oder Schreiben wird auf Daten an der neuen Position zugegriffen. Für eine binäre Datei wird die Position auf `⟨offset⟩` Zeichen relativ zu `⟨origin⟩` eingestellt. Für `⟨origin⟩` sind die Werte `SEEK_SET` (Dateianfang), `SEEK_CUR` (aktuelle Position) oder `SEEK_END` (Dateiende) möglich. Bei fehlerfreier Ausführung liefert `fseek` den Wert Null.

```
long ftell(FILE *⟨stream⟩)
```

liefert die aktuelle Dateiposition oder `-1L` bei einem Fehler.

```
void rewind(FILE *⟨stream⟩)
```

ist äquivalent zu `fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

```
int fgetpos(FILE *⟨stream⟩, fpos_t *⟨Zeiger⟩)
```

`fgetpos` sichert die aktuelle Position für `⟨stream⟩` bei `*⟨Zeiger⟩`. Dieser Wert kann später bei `fsetpos` verwendet werden. Bei einem Fehler liefert `fgetpos` einen Wert ungleich Null.

```
int fsetpos(FILE *⟨stream⟩, const fpos_t *⟨Zeiger⟩)
```

Diese Funktion positioniert `⟨stream⟩` auf die Position `*⟨Zeiger⟩`. Bei einem Fehler liefert `fsetpos` einen Wert ungleich Null.

4.1.7 Fehlerbehandlung

```
void clearerr(FILE *⟨stream⟩)
```

löscht eine Dateiende- oder Fehlernotiz für `⟨stream⟩`.

```
int feof(FILE *⟨stream⟩)
```

liefert einen von Null verschiedenen Wert, wenn für `⟨stream⟩` das Dateiende erreicht ist.

```
int ferror(FILE*<stream>)
```

liefert einen von Null verschiedenen Wert, wenn für *<stream>* ein Fehler notiert ist.

```
int perror(const char *<string>)
```

gibt *<String>* und eine Fehlermeldung aus. Die Ausgabe erfolgt mit dem Format

```
fprintf(stderr, "%s: %s", s, "Fehlermeldung");
```

4.2 Tests für Zeichenklassen: <ctype.h>

Jede der Testfunktionen hat ein `int`-Argument, dessen Wert entweder `EOF` oder ein `unsigned char` ist. Die Funktionen liefern Null, wenn der Test fehlschlägt sonst einen Wert ungleich Null.

<code>isalnum(c)</code>	wahr für eine Zahl oder einen Buchstaben
<code>isalpha(c)</code>	wahr für einen Buchstaben
<code>iscntrl(c)</code>	Steuerzeichen
<code>isdigit(c)</code>	Ziffer
<code>isgraph(c)</code>	kein Steuer- oder Leerzeichen
<code>islower(c)</code>	kleiner Buchstabe
<code>isprint(c)</code>	druckbares Zeichen (auch Leerzeichen)
<code>ispunct(c)</code>	sichtbares Zeichen, kein Buchstabe, keine Ziffer, kein Leerzeichen
<code>isspace(c)</code>	Leerzeichen, Tabulator, Zeilen- und Seitenvorschub
<code>isupper(c)</code>	Großbuchstabe
<code>isxdigit(c)</code>	hexadezimale Ziffer

Zur Umwandlung von Groß- in Kleinbuchstaben und umgekehrt gibt es die Funktionen `tolower(c)` und `toupper(c)`.

4.3 Funktionen für Zeichenketten: <string.h>

In der <string.h> Bibliothek sind Funktionen zur Verarbeitung von Zeichenketten zusammengefaßt. Der Effekt dieser Funktionen ist undefiniert, wenn sich die Argumente überlappen (außer `memmove`).

<code>void *memcpy(void *s, const void *ct, size_t n)</code> <code>void *memmove(void *s, const void *ct, size_t n)</code>	kopiert <i>n</i> Zeichen von <i>ct</i> nach <i>s</i> , liefert <i>s</i> . wie <code>memcpy</code> , funktioniert aber auch wenn sich die Argumente überlappen.
<code>int memcmp(const void *cs, const void *ct, size_t n)</code>	Vergleicht die ersten <i>n</i> Zeichen von <i>cs</i> mit <i>ct</i> die Resultate entsprechen <code>strcmp</code> .
<code>void *memchr(void *s, char c, size_t n)</code>	liefert Zeiger auf das erste <i>c</i> in <i>cs</i> oder <code>NULL</code> .
<code>void *memset(void *s, char c, size_t n)</code>	setzt die ersten <i>n</i> Zeichen von <i>s</i> auf den Wert <i>c</i> , liefert <i>s</i> .

Diese Funktionen dienen der Manipulation von Objekten als Zeichevektoren. Natürlich können sie auch zur Manipulation andere Speicherinhalte verwendet werden.

<code>char * strcpy(char *dest, const char *source)</code> <code>char * strncpy(char *dest, const char *source, int n)</code>	kopieren eines Strings von <i>source</i> zu <i>dest</i> inklusive der <code>\0</code> . wie <code>strcpy</code> , es werden jedoch höchstens <i>n</i> Zeichen kopiert.
<code>char * strcat(char *head, const char *tail)</code> <code>char * strncat(char *head, const char *tail, int n)</code>	Aneinander hängen von <i>head</i> und <i>tail</i> . wie <code>strcat</code> , es werden jedoch höchstens <i>n</i> Zeichen von <i>tail</i> an <i>head</i> angefügt.

<code>int strcmp(const char *s1, const char *s2)</code>	Vergleich von <code>s1</code> mit <code>s2</code> . Die Funktion liefert einen Wert < 0 für $s1 < s2$, Wert 0 für $s1 = s2$, Wert > 0 für $s1 > s2$
<code>char *strncmp(char *s1, const char *s2, int n)</code>	wie <code>strcmp</code> , es werden jedoch höchstens n Zeichen verglichen
<code>char *strchr(const char *s, char c)</code> <code>char *strrchr(const char *s, char c)</code>	liefert eine Zeiger auf das erste Vorkommen von <code>c</code> in <code>s</code> , sonst <code>NULL</code> . liefert eine Zeiger auf das letzte Vorkommen von <code>c</code> in <code>s</code> , sonst <code>NULL</code> .
<code>size_t strspn(const char *s, const char *c)</code>	liefert die Anzahl der Zeichen am Anfang von <code>s</code> , die sämtlich in <code>c</code> vorkommen.
<code>size_t strspn(const char *s, const char *c)</code>	liefert die Anzahl der Zeichen am Anfang von <code>s</code> , die sämtlich <i>nicht</i> in <code>c</code> vorkommen.
<code>char *strpbrk(const char *cs, const char *ct)</code>	liefert einen Zeiger auf die Position in <code>cs</code> , an der ein Zeichen aus <code>ct</code> erstmals vorkommt, oder <code>NULL</code>
<code>char *strstr(const char *cs, const char *ct)</code>	liefert einen Zeiger auf die erste Kopie von <code>ct</code> in <code>cs</code> , oder <code>NULL</code> wenn es nicht vorhanden ist.
<code>char *strtok(char *s, const char *ct)</code>	durchsucht <code>s</code> nach Zeichenfolgen, die durch Zeichen aus <code>ct</code> begrenzt sind.
<code>size_t strlen(const char *cs)</code>	liefert die Länge von <code>cs</code> ohne die abschließende <code>\0</code> .

4.4 Variable Argumentliste: `<stdarg.h>`

In C ist es möglich, Funktionen mit variabler Argumentliste zu definieren. Weder die Anzahl noch der Typ der Argumente muß bei der Definition bekannt sein. Die in `<stdarg.h>` definierten Makros und Funktionen dienen dazu, die Funktionsargumente abzuarbeiten, deren Länge und Datentypen nicht bekannt sind. `lastarg` sei der letzte benannte Parameter der Funktion, in der Funktion mit einer variablen Argumentliste. Es muß in dieser Funktion ein Zeiger vom Typ `va_list` vereinbart werden, der der Reihe nach auf jedes Argument zeigt.

```
va_list ptr;
```

Mit dem Makro `va_start` muß dieser Zeiger dann initialisiert werden.

```
va_start(ptr, lastarg);
```

Jede folgende Ausführung des Makros `va_arg` liefert dann einen Wert, der den Datentyp und Wert des nächsten unbenannten Argumentes besitzt. Mit

```
type va_arg(ptr, type);
```

kann dann die Zuweisung erfolgen. Am Ende der Funktion mit variabler Argumentliste muß das Makro `va_end(ptr)` einmal aufgerufen werden. Zwei Beispiele sollen das verdeutlichen.

Oft ist es sinnvoll, die Ausgabe von Fehlermeldungen sowohl auf `stdout` oder `stderr` durchzuführen als auch in eine Datei, damit man die Liste nach dem Programmablauf studieren kann. Das doppelte Schreiben der Meldungen und ihrer

Formate wäre recht umständlich. Die Funktion `splice_fprintf` schreibt die Parameter in die beiden Datenströme `f1` und `f2`. Die Funktion `vfprintf` aus `<stdio.h>` erledigt das Abarbeiten der Argumentliste. `splice_fprintf` hat die volle Funktionalität der normalen `fprintf` Funktion.

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

int splice_fprintf(FILE *f1, FILE *f2, char *format, ...)
{
    va_list arg_ptr;
    int val;

    va_start(arg_ptr, format); /* Zeiger auf Argumente
                                initialisieren.*/
    vfprintf(f1, format, arg_ptr);
    /* Durchreichen der Argumente
       an vfprintf.*/
    val=vfprintf(f2, format, arg_ptr);
    va_end(arg_ptr);          /* Aufräumen auf dem Stack. */
    return val;
}
```

Ein weiteres Beispiel baut auf den String-Listen des Beispiels für lineare Listen auf. Die Notation

```
l=Append(Append(Create("Hello"), "World"), "!");
```

war etwas unübersichtlich bei längeren Listen. Die Funktion `List` erledigt die selbe Aufgabe mit dem Aufruf:

```
l=List("Hello", "World", "!", "");
```

Hier nun die Definition von `List`

```
s_lst * List( char *str, ...)
{
    va_list ptr;
    char *str1;
    s_lst *lst;

    va_start(ptr, str); /* ptr initialisieren. */
    lst=Create(str);    /* Liste mit einem Element
                        erzeugen. */
    while (! strcmp(str1=va_arg(ptr, char *), ""))
        /* Einen neuen String aus
           der Argumentliste holen */
        lst=Append(lst, str1); /* und anhängen. */
    va_end(ptr);         /* Aufräumen. */
    return lst;         /* Fertig. */
}
```

Die Funktion `List` erkennt das Ende der Argumentliste an der Übergabe eines Leerstrings als letzten Parameter. Dieser darf *keinesfalls* vergessen werden. Es gibt keine Möglichkeit, die Anzahl der Argumente zu bestimmen. Den `printf` Funktionen gelingt das mit Hilfe des Format-Strings, der `List`-Funktion mit dem

abschließenden Leerstring. Man könnte natürlich die Argumentanzahl auch als (festen) Parameter übergeben.

4.5 Mathematische Funktionen: <math.h>

C stellt die folgenden mathematischen Funktionen zur Verfügung. Der Rückgabewert ist stets double.

Funktion	Bedeutung	Bemerkungen
sin(double x)	$\sin x$	
cos(double x)	$\cos x$	
tan(double x)	$\tan x$	
asin(double x)	$\arcsin x$	$x \in [-1, 1]$, Ergebnisse im Bereich $[-\pi/2, \pi/2]$
acos(double x)	$\arccos x$	$x \in [-1, 1]$, Ergebnisse im Bereich $[0, \pi]$
atan(double x)	$\arctan x$	Ergebnisse im Bereich $[-\pi/2, \pi/2]$
atan2(double x, double y)	$\arctan(x/y)$	Ergebnisse im Bereich $[-\pi, \pi]$
sinh(double x)	$\sinh x$	
cosh(double x)	$\cosh x$	
tanh(double x)	$\tanh x$	
exp(double x)	$\exp x$	
log(double x)	$\ln x$	natürlicher Logarithmus, $x > 0$
log10(double x)	$\log_{10} x$	dekadischer Logarithmus, $x > 0$
pow(double x, double y)	x^y	Fehler für $x = 0, y \leq 0$ und $x < 0, y$ nicht ganzzahlig
sqrt(double x)	\sqrt{x}	$x > 0$
ceil(double x)	$\lceil x \rceil$	kleinster ganzzahliger Wert, der nicht kleiner als x ist
floor(double x)	$\lfloor x \rfloor$	größter ganzzahliger Wert, der nicht größer als x ist
fabs(double x)	$ x $	leicht zu verwechseln mit der <code>int abs(int n)</code> Funktion
Funktion	Bedeutung	Bemerkungen
ldexp(double x, int n)	$x \cdot 2^n$	
frexp(double x, int *exp)		zerlegt x in normalisierte Mantisse, die als Ergebnis geliefert wird und eine Potenz von 2, die in *exp abgelegt wird.
modf(double x, int *ip)		zerlegt x in einen ganzzahligen Anteil und einen Rest, der Rest wird als Resultat geliefert, der ganzzahlige Teil wird in *ip abgelegt.
fmod(double x, double y)		Rest der Division x/y

4.6 Hilfsfunktionen <stdlib.h>

In der <stdlib.h> sind die Speicherverwaltung und einige weitere Hilfsfunktionen enthalten.

4.6.1 Speicherverwaltung

```
void *calloc(size_t ObjectNo, size_t Size)
void *malloc(size_t Size)
void *realloc(void *Pointer, size_t Size)
void free(void *Pointer)
```

Mit der Funktion `malloc` wird dynamisch ein Speicherblock der Größe *Size* angefordert. Der Rückgabewert ist ein Zeiger auf den Speicherbereich oder `NULL`, wenn kein Speicher mehr vorhanden ist.

`calloc` reserviert Speicher für einen Vektor von aus *ObjectNo* Objekten der Größe *Size*. Rückgabewert ist der Zeiger auf den Anfang des reservierten Speichers oder `NULL`, wenn kein Speicher mehr vorhanden ist.

`realloc` verändert die Größe des Objektes, auf das *Pointer* zeigt, in *Size* ab. Bis zum Minimum der alten und der neue Größe bleibt der Inhalt unverändert. Das Resultat ist entweder ein Zeiger auf den neuen Speicherbereich, oder `NULL`, wenn die Größe nicht verändert werden kann. **Pointer* bleibt dann unverändert. Man beachte, daß `realloc` scheitern kann, das Ergebnis muß daher *immer* überprüft werden.

Mit `free` wird ein vorher vereinbarter Speicherbereich (auf den *Pointer* zeigt) wieder frei gegeben.

4.6.2 Kommunikation mit dem Betriebssystem

```
void abort(void)
void exit(int Status)
int atexit(void (*FuncName)(void))
int system(const char *Kommando)
char *getenv(const char *Name)
```

`abort` beendet das Programm normal. `exit` beendet das Programm normal, die `atexit` Funktionen werden in umgekehrter Reihenfolge ausgeführt und noch offene Dateien geschlossen. Der Rückkehrwert des Programms wird auf *Status* gesetzt.

Mit der `atexit` Funktion kann eine Funktion *FuncName* angegeben werden, die aufgerufen wird, wenn das Programm normal endet. `atexit` liefert einen Wert ungleich Null, wenn die Funktion nicht hinterlegt werden kann.

`system` sendet die Zeichenkette *Kommando* an das Betriebssystem zur Ausführung. Das Ergebnis ist implementationsabhängig.

`getenv` liefert die zu *Name* gehörende Zeichenkette aus der Umgebung oder `NULL` wenn keine solche Zeichenkette existiert.

4.6.3 Weiter Hilfsfunktionen

Funktion	Bemerkungen
<code>int abs(int n)</code>	$ n $ Absolutwert eines <code>int</code> Arguments
<code>long labs(long n)</code>	$ n $ Absolutwert eines <code>long int</code> Arguments
<code>double atof(const char *s)</code>	wandelt den String <code>s</code> in eine <code>double</code> Zahl um.

Funktion	Bemerkungen
<pre>int atoi(const char *s) long atol(const char *s) double strtod(const char s, char **endp) long strtol(const char *s, char **endp, int base) unsigned long strtoul(const char *s, char **endp, int base) int rand(void) int srand(unsigned int seed)</pre>	<p>wandelt den String <i>s</i> in eine <i>int</i> Zahl um. wandelt den String <i>s</i> in eine <i>long int</i> Zahl um.</p> <p>wandelt den String <i>s</i> in eine <i>double</i> Zahl um. Zwischenraum am Anfang wird ignoriert, der nicht umgewandelte Rest wird in <i>*endp</i> gespeichert</p> <p>wandelt den String <i>s</i> in eine <i>long int</i> Zahl um. Zwischenraum am Anfang wird ignoriert, der nicht umgewandelte Rest wird in <i>*endp</i> gespeichert falls <i>endp</i> nicht <i>NULL</i> ist. <i>Base</i> hat einen Wert zwischen 2 und 36 erfolgt die Umwandlung unter der Annahme, daß die Zahl in dieser Basis repräsentiert ist.</p> <p>wie <i>strtol</i>, nur daß die Umwandlung in <i>unsigned long</i> erfolgt</p> <p>liefert eine ganzzahlige Pseudozufallszahl im Bereich von 0 bis <i>RAND_MAX</i>.</p> <p>benutzt <i>seed</i> für eine neue Folge von Pseudozufallszahlen.</p>

5 Variablen und Funktionen in einem Projekt

Bei einem größeren Projekt ist es sinnvoll, die Funktionsdefinitionen auf mehrere Dateien zu verteilen. Dadurch gewinnt das Programm an Übersichtlichkeit und es müssen im Falle von Änderungen nur die geänderten Teile neu übersetzt werden. Ein separat kompiliertes Modul besteht aus einem Header-File (Endung **.h*), der die Deklaration der Funktionen (Funktionsprototypen) enthält, und der Definition der Funktionen (dem C-Quelltext). Wenn andere Module Funktionen aus diesem Modul benutzen sollen, muß der Header-File mit einer *#include*-Anweisung in diese Module eingelesen werden. Dadurch ist sichergestellt, daß der Compiler die Information über den Rückgabewert der Funktion und die Typen der formalen Parameter enthält. Die Verbindung der Funktionsaufrufe realisiert dann der Linker. Jeder Bezeichner darf nur in *einer* Bedeutung vorkommen. Soll in einem Modul eine Funktion definiert werden, die nur in diesem Modul verwendet werden soll, so muß diese Funktion mit der Speicherklasse *static* vereinbart werden. Der Compiler stellt dann sicher, daß der Name nicht nochmals vergeben wird. Innerhalb des definierenden Modules kann eine als *static* vereinbarte Funktion wie eine normale Funktion verwendet werden, ein anderes Modul kann diese Funktion aber nicht benutzen.

Sollen globale Variablen in einem (und nur in diesem) Modul definiert werden so sind diese ebenfalls mit *static* zu definieren. Globale Variablen für das ganze Projekt müssen in allen Modulen, die auf die Variablen zugreifen, als *extern* deklariert werden. In genau einem Module muß die globale Variable dann aber auch noch *definiert* werden.

Ein C-Programm kann die Deklaration von Funktionen und Variable beliebig oft enthalten, es darf aber stets nur eine *Definiton* geben. Im Falle von mehrfachen Deklarationen müssen diese übereinstimmen.

Ein Beispiel soll dies verdeutlichen. Das Projekt soll aus drei Dateien bestehen, dem Quelltext, der die `main()`-Funktion enthält, `test.c` und einem weiteren Modul `module1.c` in dem die Funktion `do_something` definiert wird die dann in der `main()`-Funktion benutzt wird. In der Datei `module1.c` wollen wir eine globale Variable `global_x` verändern. Damit der Compiler beim Übersetzen von `test.c` über die Parameter der Funktion `do_something` informiert ist, benötigen wir eine dritte Datei, den Header-File `module1.h`, der in diesem einfachen Beispiel nur den Prototyp von `do_something` enthält.

```
/* Header File module1.h */
/* Prototyp: Deklaration von do_something */
void do_something(int i);
```

Header-File für `module1.c`

Die Datei `module1.c` enthält die Definition von `do_something` und die Deklaration der Variablen `global_x`. Es wird auch noch eine im `module1.c` globale Funktion `blub` definiert und eine `int` Variable `local_x`, deren Gültigkeit nur auf `module1.c` beschränkt ist.

```
/* Definiton File module1.c */
#include "module1.h"

extern double global_x;
    /* Deklarieren der globalen Variablen */

static int local_x;
    /* local_x als global in module1.c */

static void blub(void)
{ ... }

void do_something(int i) /* die Definition */
{ ... }
```

Funktionsdefinitionen für `module1.h`

Das Hauptprogramm `test.c` nimmt den Prototyp von `do_something` via `#include` auf. Hier wird nun auch die globale Variable `global_x` definiert. Die Definition von `local_x` und `blub()` ist unabhängig von den Definitionen in `module1.c`.

```
/* Hauptprogramm test.c */
#include "module1.h"

double global_x; /* Definieren der globalen Variablen */

double local_x;

void blub(double x, double y)
{ ... }

int main(int argc, char *argv[]) {
    ...
    do_something(4);

    blub(2.0, 18.0);
}
```

Das Hauptprogramm greift beim Aufruf von `blub()` auf die eigene Funktion zu, nicht auf die in `module1.c` definierte. CSequence `local_x` hat *nichts* mit der in `module1.c` definierten Variablen gleichen Namens gemein. Sowohl `module1.c` als auch das Hauptprogramm können auf `global_x` zugreifen.

6 Doppelt verkettete Liste

Die doppelt verketteten Listen stellen eine Erweiterung der bereits besprochenen einfach verketteten Listen dar. Die doppelt verkettete Liste besteht aus einem Eintrag und jeweils einem Zeiger auf den Vorgänger und den Nachfolger. Der Vorgänger des ersten und der Nachfolger des letzten Listenelements ist NULL. Als Bibliotheken benötigt man:

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

Die Datenstruktur für die doppelt verkettete List hat die Form

```
typedef struct s_dlst { char *str;
                       struct s_dlst *prev;
                       struct s_dlst *next;
                       } s_dlst;
```

dabei ist prev der Zeiger auf den Vorgänger und next der Zeiger auf den Nachfolger. Der Vorteil der doppelten Verkettung besteht darin, daß man nicht mehr den Kopf der Liste aufbewahren muß. Von ein beliebiges Listenelement aus kann man zum Anfang oder zum Ende zurückkehren. Die folgenden Macros ermöglichen dies.

```
#define BeginList(lst)  if (lst) \
                        while(lst->prev) lst=lst->prev;
#define EndList(lst)   if (lst) \
                        while(lst->next) lst=lst->next;
```

Die Erzeugung eines einzelnen Listenelements geschieht völlig analog zum Beispiel der einfach verketteten Liste, nur daß nun auch der Vorgänger prev auf NULL gesetzt wird.

```
void error(char *error_msg)
{
    fprintf(stderr, "Error in string list ...\n");
    fprintf(stderr, "%s\n", error_msg);
}

s_dlst * DCreate(char *s)
{
    s_dlst *lst;

    lst=(s_dlst *) malloc((size_t) sizeof(s_dlst));
    if (!lst) {
        error("fail to allocate memory 1 in DCreate.");
        return lst;
    }
    lst->str=
        (char *) malloc((size_t) (strlen(s)+1)*sizeof(char));
    if (!(lst->str)) {
        error("fail to allocate memory 2 in DCreate.");
        free(lst);
        return NULL;
    }
}
```

```

    }
    strcpy(lst->str,s);
    lst->next=NULL;
    lst->prev=NULL;
    return lst;
}

```

Das Voranstellen und das Anhängen ist dem Beispiel der einfachen Verkettung ebenfalls sehr ähnlich.

```

s_dlst * DPrepend(s_dlst *tail, char *first_str)
{ s_dlst *head;

```

```

    head=DCreate(first_str);
    BeginList(tail);
    if (!head) {
        error("fail to get memory in DPrepend.");
        return tail;
    }
    head->next=tail; /* Verbinden von head mit tail. */
    tail->prev=head; /* Verbinden von tail mit head. */
    return head;
}

```

```

s_dlst * DAppend(s_dlst *head, char *last_str)
{ s_dlst *tail,*lst;

```

```

    tail=DCreate(last_str);
    lst=head;
    if (!tail) {
        error("fail to get memory in DAppend.");
        return head;
    }
    EndList(head);
    head->next=tail; /* Verbinden der Liste mit dem tail. */
    tail->prev=head; /* den Vorgaengerzeiger von tail auf
                    den vorletzten der neuen Liste
                    setzen. */

    return lst;
}

```

Die letzte zur Konstruktion der Listenstruktur dienende Funktion hat eine variable Argumentliste und entspricht dem Beispiel für lineare Listen; die Verkettung erfolgt jedoch mit den neuen Funktionen DCreate und DAppend.

```

s_dlst * DList( char *str, ...)

```

```

{ va_list ptr;
  char *str1;
  s_dlst *lst;

  va_start(ptr,str);
  lst=DCreate(str);
  while (! strcmp(str1=va_arg(ptr,char *),"")==0)

```

```

    lst=DAppend(lst, str1);
    va_end(ptr);
    return lst;
}

```

Mit `l=DList("First", "Second", "Third", "")` wird nun die doppelt verkettete Liste (First, Second, Third) konstruiert.

Die Beseitigung der Struktur aus dem Speicher übernimmt `DDestroy`.

```

s_dlst * DDestroy(s_dlst *lst)
{ s_dlst *tail;

  BeginList(lst);
  /* am Anfang der Liste beginnen */
  while (lst) {
    tail=lst->next;
    /* Schwanz der Liste retten */
    free((void *) lst->str);
    /* Inhalt freigeben */
    free((void *) lst);
    /* aktuelles Listenelement freigeben */
    lst=tail;
  }
  return NULL;
}

```

Während bei der einfach verketteten Liste das Umkehren der Liste einen recht aufwendigen rekursiven Algorithmus erforderte, genügt es bei der doppelt verketteten Liste, für jedes Element den Zeiger auf den Vorgänger mit dem Zeiger auf den Nachfolger zu vertauschen.

```

s_dlst *DReverse(s_dlst *lst)
{ s_dlst *tmp, *l;

  if (!lst) /* Leere Liste */
    return NULL;
  BeginList(lst);
  if (!lst->next) /* Liste mit einem Element */
    return lst;
  l=lst;
  while (l) {
    tmp=l->prev; /* Vertauschen der Zeiger */
    l->prev=l->next;
    l->next=tmp;
    l=l->prev; /* nach dem Vertauschen gehts
               rueckwaerts weiter. */

    if (!l)
      break;
  }
  BeginList(lst);
  return lst;
}

```

Bei den doppelt verketteten Listen erfordert das Einfügen und Löschen bestimmter Listenelemente besondere Sorgfalt. Beim Einfügen muß zuerst das neue Element ins mit der vorhanden Struktur verbunden werden, bevor die alten Verbindungen aufgelöst werden. Das neue Element soll an Stelle pos eingefügt werden.

```
s_dlst *DInsert(s_dlst *lst, char *what, int pos)
{ s_dlst *head, *ins;
  int i;

  if (!lst)
    return DCreate(what); /* lst ist leer */
  if (pos<=1) /* negative position */
    return DPrepend(lst,what);
  i=1;
  head=lst; /* bis zur pos-ten Stelle vorruecken */
  while ((i<pos) && (lst)) {
    lst=lst->next;
    i++;
  }
  if (!lst) /* Liste kuerzer als pos */
    return DAppend(head,what);
  ins=DCreate(what);
  (lst->prev)->next=ins; /* Verbindung mit dem Vor- */
  ins->prev=lst->prev; /* gaenger von lst. */
  lst->prev=ins; /* Verbindung mit lst. */
  ins->next=lst;
  return head;
}
```

Das Löschen eines Listenelements geschieht erst nach dem Herauslösen und neuen Verbinden der Datenstruktur.

```
s_dlst * DDelete(s_dlst *lst, int pos)
{ s_dlst *head, *tail;
  int i;

  if (pos<1)
    return lst;
  if (!lst)
    return lst;
  if (pos==1) { /* erstes Element loeschen */
    head=lst->next;
    free((void *) lst);
    head->prev=NULL;
    return head;
  }
  i=1; /* vorwaerts bis zur stelle pos */
  head=lst;
  while ((i<pos) && (lst)) {
    lst=lst->next;
    i++;
  }
}
```

```

        if (!lst) /* Position groesser als Laenge der Li-
ste */
        return head;
        (lst->prev)->next=lst->next;
        /* Umsetzen der Verbindungen */
        (lst->next)->prev=lst->prev;
        /* des Vorg"angers auf den */
        /* Nachfolger und umgekehrt. */
        free((void *) lst);
        return head;
    }

```

Das Ausgeben der Liste erfolgt mit dem iterativen Analogon der Print_List-Funktion aus den Beispiel für lineare Listen.

```

void PrintList(s_dlst *lst, int print_komma)
{ static char *fmt_c="%s, ";
  static char *fmt_woc ="%s ";
  char *fmt;

  if (!lst) {
    printf("()\n");
    return;
  }
  fmt= (print_komma ? fmt_c : fmt_woc);
  printf("(");
  BeginList(lst);
  while (lst->next) {
    printf(fmt, lst->str);
    lst=lst->next;
  }
  printf("%s)\n", lst->str);
}

```

Das Sortieren der Listeneinträge erfordert eine etwas andere Strategie als etwa bei Feldern. Als erstes soll der asymptotisch sehr ineffiziente *bubble sort* Algorithmus implementiert werden. *bubble sort* nutzt zum Sortieren nur jeweils die nächsten Nachbarn. Zum Sortieren benötigt man zwei Operationen, die für die jeweiligen Inhalte der Liste charakteristisch sind. Man benötigt eine Vergleichsrelation und eine Routine, die jeweils die Inhalte vertauscht. Im vorliegenden Beispiele soll die Vergleichsroutine DComp(arg1, arg2) einen Wert kleiner Null liefern für $arg1 < arg2$; Null für $arg1 == arg2$ und sonst einen Wert größer als Null. Beim Vertauschen ist es nicht sinnvoll, die Listenelemente aus dem Verband zu lösen; es genügt, die Zeiger auf die Zeichenketten zu vertauschen. Hier nun die Vergleichs- und Austauschfunktion

```

int DComp(void *p1, void *p2)
{ s_dlst *l1, *l2;

  l1=(s_dlst *)p1;
  l2=(s_dlst *)p2;
  return strcmp(l1->str, l2->str);
}

```

```

void DSwap(void *p1, void *p2)
{ s_dlst *l1,*l2;
  char *tmp;

  l1=(s_dlst *)p1;
  l2=(s_dlst *)p2;
  tmp=l1->str;
  l1->str=l2->str;
  l2->str=tmp;
}

```

Die Funktion DComp realisiert dann das Sortieren in alphabetisch aufsteigender Folge. Der eigentliche *bubble sort* Algorithmus sieht dann so aus

```

s_dlst *DBubble_Sort(s_dlst *lst,
                    int (*compare)(void *, void *),
                    void (*swap)(void *, void *))
{ s_dlst *last, *endp, *p;

  last=lst;
  EndList(last);
  endp=last;
  while (last!=lst) {
    p=last=lst;
    while ((p->next) && (p!=endp)) {
      if (compare(p,p->next)>0) {
        swap(p,p->next);
        last=p;
      }
      p=p->next;
    }
    endp=last;
  }
  return lst;
}

```

Da *bubble sort* beim k -ten Durchlauf den $N - k + 1$ Größten findet, genügt es, die Position der letzten Vertauschung *last* zu speichern. Mit dem Aufruf:

```

l=DList("Zebra","Gnu","Lion","Tiger","Ape","Horse",
        "Cat","Mouse","Dog","Rabbit","Bunny","Elephant","");
l=DBubble_Sort(l,DComp,DSwap);

```

erhält man nach jeweils einem Durchlauf der while (last!=lst) {...} Schleife die Ergebnisse:

```

1 ( Zebra , Gnu , Lion , Tiger , Ape , Horse , Cat , Mouse , Dog , Rabbit , Bunny , Elephant )
2 ( Gnu , Lion , Tiger , Ape , Horse , Cat , Mouse , Dog , Rabbit , Bunny , Elephant , Zebra )
3 ( Gnu , Lion , Ape , Horse , Cat , Mouse , Dog , Rabbit , Bunny , Elephant , Tiger , Zebra )
4 ( Gnu , Ape , Horse , Cat , Lion , Dog , Mouse , Bunny , Elephant , Rabbit , Tiger , Zebra )
5 ( Ape , Gnu , Cat , Horse , Dog , Lion , Bunny , Elephant , Mouse , Rabbit , Tiger , Zebra )
6 ( Ape , Cat , Gnu , Dog , Horse , Bunny , Elephant , Lion , Mouse , Rabbit , Tiger , Zebra )
7 ( Ape , Cat , Dog , Gnu , Bunny , Elephant , Horse , Lion , Mouse , Rabbit , Tiger , Zebra )
8 ( Ape , Cat , Dog , Bunny , Elephant , Gnu , Horse , Lion , Mouse , Rabbit , Tiger , Zebra )
9 ( Ape , Cat , Bunny , Dog , Elephant , Gnu , Horse , Lion , Mouse , Rabbit , Tiger , Zebra )
10 ( Ape , Bunny , Cat , Dog , Elephant , Gnu , Horse , Lion , Mouse , Rabbit , Tiger , Zebra )

```

Der gemeinsame Weg von Hase und Elefant nach vorn unterstreicht nochmals die Wirkungsweise des Algorithmus’.

Die Implementation des *quick sort* Algorithmus’ erfordert eine Indizierung der Liste, den das bessere asymptotische Laufzeitverhalten von *quick sort* basiert darauf, daß die Einträge über einen größeren Abstand, als den der nächsten Nachbarn, getauscht werden können. Um beim rekursiven Aufruf von `qsort_aux` Stack zu sparen, werden die Tausch- und die Vergleichsfunktionen in den globalen Variablen `qcmp_` und `qswap_` abgelegt. `qsort_aux` realisiert den eigentlichen *quick sort* Algorithmus. In der Funktion `DQuick_Sort` wird das Feld der Zeiger auf die Listenelemente initialisiert und die Rekursion gestartet.

```

static int  (*qcmp_)(void *,void *);
static void (*qswap_)(void *, void *);

int DLength(s_dlst *lst)
{ int len;

  if (!lst)
    return 0;
  for (len=0; lst; lst=lst->next, len++);
  return len;
}

void qsort_aux(s_dlst **v, int l,int r)
{ s_dlst *key;
  int i,j;

  key= *(v + ((l+r) >> 1));
  i=l;
  j=r;
  do {
    while (qcmp_(*(v+i),key)<0) i++;
    while (qcmp_(*(v+j),key)>0) j--;
    if (i<j)
      qswap_(*(v+i),*(v+j));
    if (i<=j) {
      i++;
      j--;
    }
  }
}

```

```

    while (i<=j);
    if (l<j)
        qsort_aux(v,l,j);
    if (i<r)
        qsort_aux(v,i,r);
}

s_dlst *DQuick_Sort(s_dlst *lst,
    int (*compare)(void *, void *),
    void (*swap)(void *, void *))
{ s_dlst **hlp, *l;
  int i,n;

  BeginList(lst);
  n=DLength(lst);
  hlp=(s_dlst **) malloc(n*sizeof(s_dlst *));
  for (i=0,l=lst; l; i++,l=l->next)
      *(hlp+i)=l;
  qcmp__=compare;
  qswap__=swap;
  qsort_aux(hlp,0,n-1);
  qcmp__=NULL;
  qswap__=NULL;
  free(hlp);
  return lst;
}

```

7 Sortieren

Im Beispiel für die doppelt verketteten Listen wurden zwei Sortierverfahren vorgestellt. Die relativ kurzen Listen von Zeichenketten sind für die Diskussion des günstigsten Sortieralgorithmus schlechte Beispiele. Am Beispiel des Sortierens von Feldern reeller Zahlen sei ein Vergleich des Laufzeitverhaltens von *straight insertion*, *bubble sort*, *quick sort* und *heap sort* durchgeführt. Die Funktion *insertion* implementiert den *straight insertion*-Algorithmus für ein Feld aus double Zahlen:

```

void insertion(long int n, double *v)
{ long int i,j;
  double t;

  for (i=1; i<n; i++) {
      j=i;
      t=v[i];
      while (v[j-1]>t) {
          v[j]=v[j-1];
          if (--j <1)
              break;
      } /* while */
      v[j]=t;
  } /* for i */
}

```

Eine Implementation von *bubble sort* kann zum Beispiel so aussehen

```
void dswap(double *x, double *y)
{ double tmp;

  tmp=(*x);
  (*x)=(*y);
  (*y)=tmp;
}

void bubble(long int n, double *v)
{ int exch=1;
  long int i,last,up;

  last=n;
  while (exch) {
    exch=0;
    up=last;
    for (i=1; i<up; i++)      /* suchen unsortierter
                              Nachbarn */
      if (v[i-1] > v[i]) {
        dswap(v+i-1,v+i);  /* Austauschen */
        exch=1;
        last=i;
      } /* if */
    } /* while */
  }
}
```

Die etwas übersichtlichere Variante von *quick sort* hat die Form:

```
void quick_aux(long int left, long int right, double *v)
{ long int i,j;
  double k;

  i=left;
  j=right;
  k=v[(left + right) / 2];
  do {
    while (v[i]<k) i++;
    while (v[j]>k) j--;
    if (i < j)
      dswap(v+i, v+j);
    if (i<=j) {
      i++;
      j--;
    }
  }
  while (i<=j);
  if (left<j)
    quick_aux(left,j,v);
  if (i<right)
    quick_aux(i,right,v);
}
```

<i>N</i>	<i>straight insertion</i>	<i>bubble sort</i>	<i>quick sort</i>	<i>heap sort</i>
50	0.00	0.05	0.00	0.00
100	0.06	0.05	0.00	0.00
200	0.11	0.39	0.00	0.05
400	0.50	1.43	0.05	0.11
800	1.92	5.83	0.11	0.27
1600	7.19	22.90	0.27	0.61
3200	28.45	91.40	0.49	1.26
6400	114.47	367.18	1.10	2.74

Tabelle 4: Verschiedene Sortierverfahren im Vergleich. Zeiten für einen 486-er in Sekunden.

```
void quick(long int n, double *v)
{
    quick_aux(0,n-1,v);
}
```

Für *heap sort* wird die Implementation

```
void heap_arrange(long int i, long int m, double *v)
{ long int j;

    while (i * 2 <= m) {
        j = i * 2;
        if (j < m && v[j - 1] < v[j])
            j++;
        if (v[i-1] < v[j-1]) {
            dswap(v+i-1, v+j-1);
            i = j;
        }
        else
            i = m;
    }
}

void heapsort(long int n, double *v)
{ long int i;

    for (i = n / 2; i >= 1; i--)
        heap_arrange(i, n, v);
    for (i = n - 1; i >= 1; i--) {
        dswap(v+i, v);
        heap_arrange(1, i, v);
    }
} /* heapsort */
```

benutzt.

Die Tabelle zeigt das Laufzeitverhalten der so implementierten Verfahren für verschiedene Anzahl *N* der zu sortierenden Daten.

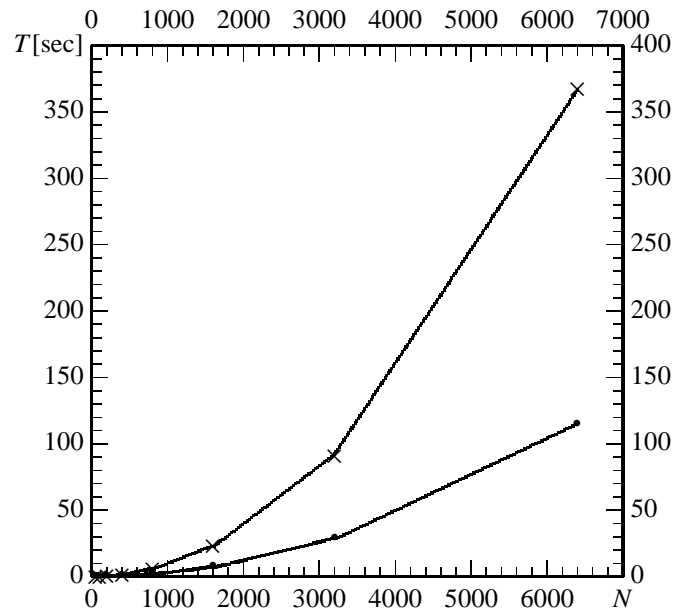


Abbildung 1: Vergleich von *straight insertion* (•) mit *bubble sort* (×).

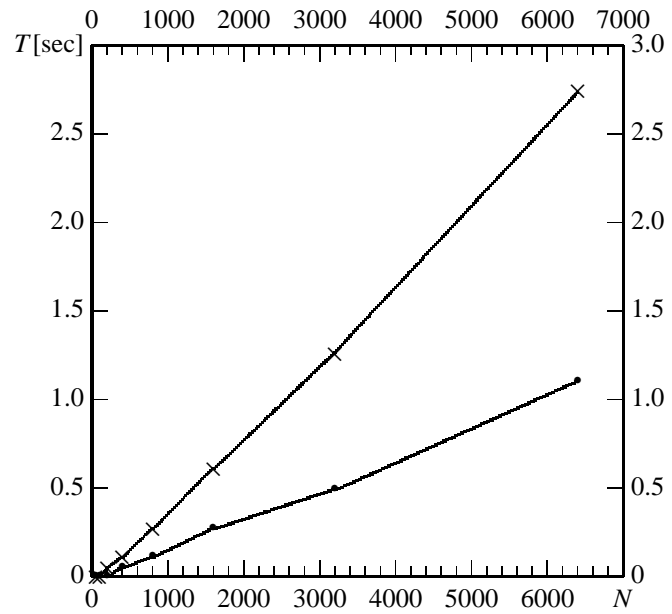


Abbildung 2: Vergleich von *quick sort* (•) mit *heap sort* (×).

Als zu sortierende Folgen wurden gleichverteilte Zufallszahlen im Bereich $[0, 1]$ gewählt. Jedes Verfahren hatte die gleiche Folge von Zahlen zu sortieren.

Wie man aus der Tabelle ersieht, ist *bubble sort* dreimal langsamer als *straight insertion*, beide Verfahren liefern ein Laufzeitverhalten $\approx N^2$. *quick sort* ist nur ungefähr doppelt so schnell wie *heap sort*. Der verwendete Datensatz ist allerdings auch optimal für *quick sort*, die Laufzeit beider Verfahren bestätigt die $\approx N \ln N$ Abhängigkeit. Während *heap sort* stets ein echter $N \ln N$ Prozeß ist, kann *quick sort* im schlimmsten Fall ein N^2 Prozeß werden.