Programmieren II

Zeiger und Vektoren
Dateien
Spezielle Datentypen
Dynamische Speicherverwaltung
Rekursion
Verkettete Listen

HAW-Hamburg
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Prof. Robert Heß

17. Februar 2020

Inhaltsverzeichnis

Kapitel 1

Zeiger (pointer) und Vektoren

1.1 Einleitung

Ein Zeiger (engl. *pointer*) ist eine Variable, die auf eine Stelle im Speicher zeigt. Dabei ist der Zeiger eine Variable, in die eine Adresse gespeichert wird. Dagegen ist ein Vektor (engl. *array*) eine Verkettung von Variablen. Zeiger und Vektoren sind in C eng miteinander verwandt, und müssen daher im Zusammenhang betrachtet werden. Vektoren haben wir bereits im ersten Semester kennengelernt (Programmieren 1, www.rrhess.de). Hier wenden wir uns den Zeigern und deren Zusammenspiel mit Vektoren zu.

Zur Erinnerung: Eine Variable wird häufig mit vier Eigenschaften beschrieben:

Typ. Eine Variable hat in C einen definierten Datentyp wie z.B. *int* oder *char*.

Name. Um Variablen von anderen zu unterscheiden, wird ihnen ein eindeutiger Name zugewiesen.

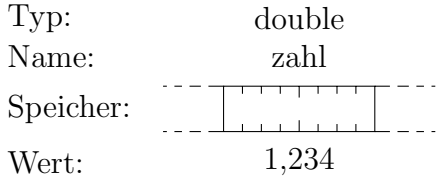
Adresse/Speicher. Eine Variable wird an einer Stelle im Speicher abgelegt, die durch eine eindeutige Adresse definiert ist.

Wert. Schließlich hat die Variable auch einen Wert.

Wir betrachten z.B. folgende Definition:

```
double zahl=1.234;
```

Die vier Eigenschaften der Variable *zahl* können so dargestellt werden:



Um das Thema Zeiger (und Vektoren) besser zu verstehen, werden wir uns in diesem Kapitel immer wieder die vier Eigenschaften der Variablen vergegenwärtigen.

Jede Speicherzelle im Arbeitsspeicher eines Computers hat eine eindeutige Adresse. Eine Information, z.B. der Zahlenwert einer Zahl vom Typ *int*, wird unter einer

Adresse abgelegt, und kann über diese Adresse wieder abgerufen oder verändert werden. Ein Großteil der Handhabung von Adressen läuft für den Programmierer unsichtbar im Hintergrund ab. C bietet die Möglichkeit, mit diesen Adressen gezielt zu arbeiten.

Wird in C mit einer Adresse gearbeitet, ist sie nicht selbst die Information, sondern die Adresse *zeigt* auf die Stelle im Speicher, an der die Information steht. Von daher wird in C nicht von Adressen, sondern von *Zeigern* (engl. *pointer*) gesprochen.

In diesem Abschnitt betrachten wir zunächst einen einfachen Zeiger, danach verketten wir Zeiger zu einem Zeigervektor und lassen schließlich einen Zeiger auf einen Zeiger zeigen.

1.2 Einfache Zeiger

1.2.1 Definition eines einfachen Zeigers

Die allgemeine Syntax zur Definition eines Zeigers lautet:

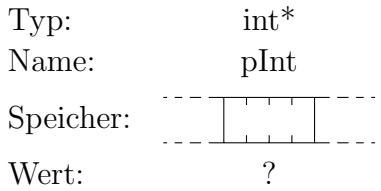
```
<Datentyp> *<Variablenname>;
```

Beispiel: Es soll ein Zeiger mit Namen *pInt* definiert werden, der auf den Datentyp *int* zeigen soll.

```
int *pInt;
```

Der Datentyp *int* deutet an, dass der Speicher, auf den der Zeiger zeigt, als *int* behandelt werden soll. Der Stern gibt an, dass es sich um einen einfachen Zeiger handelt. Der Datentyp *int* muss zusammen mit dem Stern gelesen werden, *int**, und steht für „Zeiger auf Datentyp *int*“. Der Name des Zeigers, hier *pInt*, ist frei wählbar.

Ein Zeiger belegt bei einem 32 Bit System vier Byte. Auch wenn der Zeiger auf größere oder kleinere Datentypen zeigt, benötigt er immer vier Byte. Die folgende Skizze zeigt die vier Eigenschaft des eben definierten Zeigers *pInt*:



1.2.2 Initialisierung eines einfachen Zeigers

Nachdem ein Zeiger definiert wurde, zeigt er an eine unbekannte Stelle im Speicher. Um ihn auf eine sinnvolle Stelle zeigen zu lassen, benötigen wir den Adressoperator &. Das Symbol & haben Sie schon bei den bitweisen Operatoren kennengelernt. Dort handelt es sich um einen *binären* Operator, der zwei Operanden erwartet, und als Ergebnis einen bitweise UND-verknüpften Wert liefert. Hier verwenden wir die *unäre* Variante des Operators, der als einzigen Operanden eine Variable erwartet, und als Ergebnis die Adresse dieser Variable im Speicher liefert. Dazu ein Beispiel:

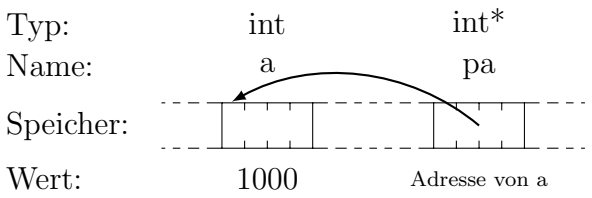
```
int a=1000;
int *pa;
pa = &a;
```

In der ersten Zeile wird die Variable *a* vom Typ *int* definiert und es wird ihr gleich der Wert 1000 zugewiesen. In der zweiten Zeile erstellen wir uns einen Zeiger mit Namen *pa*, der auf den Datentyp *int* zeigt. In der dritten Zeile wird mit dem Operator & die Speicheradresse der Variable *a* ermittelt, und dem Zeiger *pa* zugewiesen.

Definition und Initialisierung kann auch hier in einer Zeile erfolgen:

```
int a=1000;
int *pa=&a;
```

Es ergeben sich folgende Eigenschaften für die Variablen.



In Tabelle 1.1 ist beispielhaft ein Ausschnitt des Speichers nach Ausführung der eben gezeigten Programmzeilen dargestellt. Ab Adresse 01C0 7D04 ist die *int*-Variable mit Namen *a* und dem Wert 1 000 gespeichert. Weiter unten im Speicher, ab Adresse 01C4 E358, findet sich die Variable *pa* vom Typ *int** mit dem Wert 01C0 7D04, welches die Adresse der Variable *a* ist. Hier spricht man von einem Zeiger, der auf die Zahl 1 000 zeigt. Für den Zeiger spielt der Name der Variablen, auf die er zeigt, keine Rolle, sondern mit ihm kann direkt der Inhalt manipuliert werden.

Adresse	**Datum**	**Datentyp**	**Inhalt**	**Beispiel**
...				
01C0 7D04	E8	int	die Zahl 1.000	**int** a=1000;
01C0 7D05	03			
01C0 7D06	00			
01C0 7D07	00			
...				
01C4 E358	04	Zeiger auf int	die Adresse 01C0 7D04	**int** *pa=&a;
01C4 E359	7D			
01C4 E35A	C0			
01C4 E35B	01			
...				

Tabelle 1.1: Beispiel, wie ein Zeiger auf *int* im Speicher platziert wird.

1.2.3 Arbeiten mit einfachen Zeigern

Das Arbeiten mit Zeigern soll anhand eines Programmstücks erläutert werden. Achten Sie vor allem auf die Verwendung der unären Operatoren * und &. (*Unäre* Operatoren erwarten nur einen Operanden. Im Gegensatz dazu benötigen *binäre* Operatoren zwei Operanden.)

```
1  int a=4, b;      /* zwei int-Variablen */
2  int z[6];        /* ein Vektor mit sechs int-Variablen */
3  int *pi;         /* ein Zeiger auf int */
4
5  pi = &a;         /* jetzt zeigt pi auf a */
6  b = *pi;         /* jetzt hat b den Wert 4 */
7  *pi = 7;         /* jetzt hat a den Wert 7 */
8  pi = &z[2];      /* nun zeigt pi auf z[2] */
9  *pi = 9;         /* jetzt hat z[2] den Wert 9 */
10 pi++;            /* ... und jetzt zeigt pi auf z[3] */
```

Die ersten drei Zeilen sollten Ihnen geläufig sein: Die erste Zeile definiert die *int*-Variablen *a* und *b* und weist der ersten den Wert 4 zu. Die zweite Zeile definiert die Variable *z* als Vektor von sechs *int*-Variablen. Schließlich, die dritte Zeile definiert einen Zeiger auf den Datentyp *int* mit dem Namen *pi*.

In der 5. Zeile wird dafür gesorgt, dass der Zeiger *pi* (Zeiger auf *int*) auf die Variable *a* zeigt. Dies geschieht mit dem Adressoperator &, den wir bereits beim initialisieren von einfachen Zeigern kennengelernt haben.

Um die Werte zu verwenden, auf die der Zeiger zeigt, ist das Gegenstück zum Adressoperator nötig: Der Verweisoperator * ist ein *unärer* Operator, der als einzigen Operanden einen Zeiger erwartet, und als Ergebnis das liefert, worauf der Zeiger zeigt. Sie kennen bereits die *binäre* Variante des Operators * zum Multiplizieren von zwei Zahlen. Der Compiler erkennt aus dem Kontext heraus, um welchen der beiden Operatoren es sich handelt.

In der 6. Zeile wird der Wert, auf den der Zeiger *pi* zeigt, ausgelesen und in die Variable *b* kopiert. Da der Zeiger als ein Zeiger auf *int* definiert wurde, ist das Ergebnis des Verweisoperators vom Typ *int*. Das heißt, die Speicherstelle, auf die der Zeiger zeigt, wird als *int* interpretiert. In dem Beispiel wird in der Zeile `b = *pi;` die Stelle, auf die der Zeiger *pi* zeigt, als ein *int* bewertet und der Variable *b* zugewiesen. Da in der 5. Zeile dafür gesorgt wurde, dass der Zeiger *pi* auf die Variable *a* zeigt, wird hier der Wert 4 der Variablen *a* in die Variable *b* kopiert.

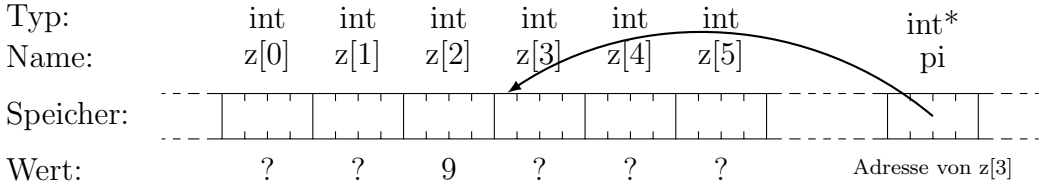
Das Gleiche kann auch andersherum geschehen. In der 7. Zeile wird mit `*pi = 7;` der Stelle, auf die der Zeiger *pi* zeigt (das ist im Beispiel immer noch die Variable *a*) der Wert 7 zugewiesen.

Ein Zeiger kann auch auf ein beliebiges Element eines Vektors zeigen. Die 8. Zeile bewirkt mit `pi = &z[2];`, dass der Zeiger *pi* nun auf das dritte Element des Vektors *z* zeigt. Wichtig: In dieser Anweisung wird wieder nur eine Adresse übergeben, und nicht der Inhalt einer Speicherstelle. (Hinweis: In der Praxis würde man `pi = z+2;` schreiben. Erläuterungen dazu folgen im Abschnitt 1.5.1)

Die Anweisung `*pi = 9;` (9. Zeile) weist der Stelle, auf den der Zeiger *pi* zeigt den Wert 9 zu. In unserem Beispiel hat das den Effekt, dass das dritte Element des Vektors *z*, also `z[2]`, diesen Wert 9 zugewiesen bekommt.

In der letzten Zeile, `pi++;` wird nun der Zeiger *pi* erhöht. Von den ganzzahligen Variablen wissen wir, dass der Operator ++ eine Variable um eins erhöht. Wird dieser Operator auf einen Zeiger angewandt, so wird die in ihm abgelegte Adresse um die Größe des Typs, auf den der Zeiger zeigt, erhöht. Wenn wir davon ausgehen, das auf unserem System die *int*-Variablen vier Byte groß sind, dann wird in dem Ausdruck `pi++;` der Zeiger *pi* um vier Speicherstellen erhöht. Da der Zeiger in unserem Beispiel vorher auf das dritte Element des Vektors `z` gezeigt hat, zeigt er nun auf das vierte Element dieses Vektors.

Nach dem hier gezeigten Programmcode haben die Variablen *pi* und *z* die folgenden vier Eigenschaften:



1.2.4 Beispiel: Zwei Zahlen austauschen, *SwapInt*

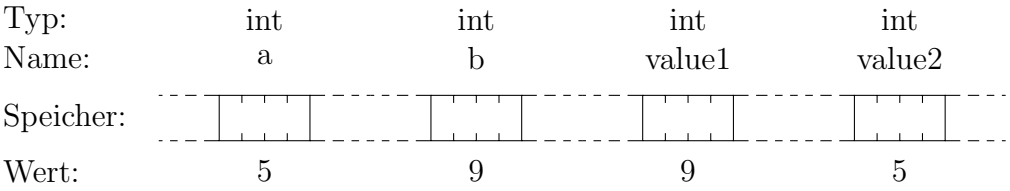
Es kommt häufig vor, dass in einem Programm zwei Zahlen ausgetauscht werden müssen. Nehmen wir an, dass die Werte der Variablen *a* und *b* vom Typ *int* vertauscht werden sollen, so sind dafür eine temporäre Variable und drei Programmzeilen nötig:

```
tmp = a;
a = b;
b = tmp;
```

Wenn in einem Programm mehrfach solche Vertauschungen vorgenommen werden, ist es sinnvoll, eine geeignete Funktion dafür zu schreiben. Nun liegen aber die übergebenen Argumente innerhalb der Funktion nur als Kopie vor. Das heißt, auch wenn die Funktion die Parameter bearbeitet, so bleiben die Werte im aufrufenden Programm unverändert. Ein solcher falscher Ansatz zum Vertauschen von *int*-Zahlen würde wie folgt aussehen:

```
void SwapInt(int value1, int value2)
/* falscher Ansatz!!! */
{
    int tmp;

    tmp = value1;
    value1 = value2;
    value2 = tmp;
}
```

Angenommen, diese falsche Funktion würde mit den beiden *int*-Variablen *a* und *b* mit den Werten fünf und neun aufgerufen, so ergibt sich am Ende der Funktion folgendes Bild für die Variablen:



Die Variablen *a* und *b* wurden in die Parameter *value1* und *value2* kopiert. Ein Austauschen der Parameter hat daher keinen Einfluss auf die übergebenen Variablen.

Die Lösung liegt in der Verwendung von Zeigern. Der Funktion werden beim Aufruf die Variablen nicht direkt übergeben, sondern Zeiger auf die Variablen. Zwar

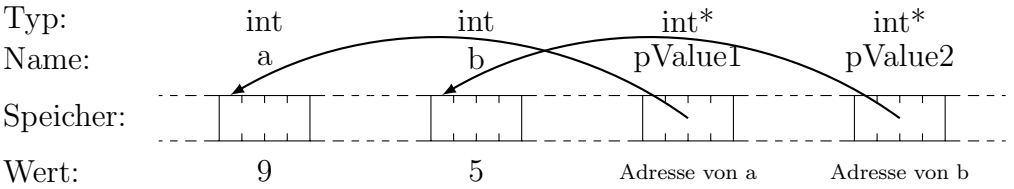
werden die Zeiger beim Aufruf wieder kopiert, doch zeigen auch die Kopien auf die Werte, die es gilt zu vertauschen. Eine korrekte Lösung der Funktion *SwapInt* könnte folgendermaßen aussehen:

```
void SwapInt(int *pValue1, int *pValue2)
/* korrekte Lösung */
{
    int tmp;

    tmp = *pValue1;
    *pValue1 = *pValue2;
    *pValue2 = tmp;
}
```

Der Aufruf der Funktion *SwapInt* erfolgt mit Zeigern auf die zu vertauschenden Werte:

```
int a=5;
int b=9;

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

Am Ende der Funktion ergeben sich folgende Eigenschaften für die Variablen *a* und *b* und die Parameter *pValue1* und *pValue2*:

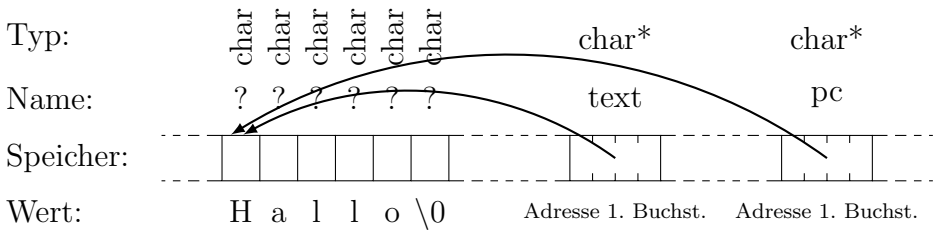


1.2.5 Beispiel: Implementierung der Funktion *strlen*

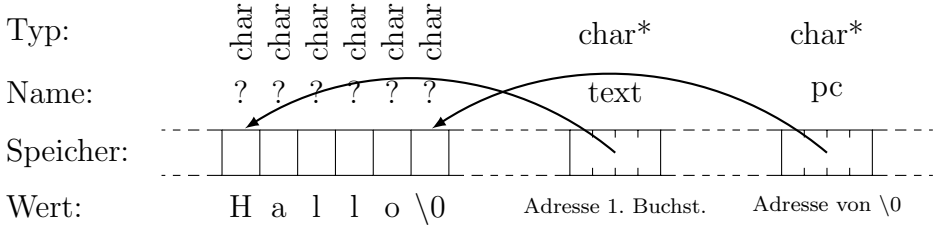
Betrachten Sie nun folgende Funktion, welche die Anzahl der Zeichen in einer Zeichenkette ermittelt. Erinnern Sie sich, am Ende einer Zeichenkette steht immer eine null.

```
int strlen(char *text)
{
    char *pc=text;
    while(*pc != '\0') pc++;
    return pc-text;
}
```

Die Funktion erhält als Parameter einen Zeiger auf eine Zeichenkette, die mit dem Code null ('\0') abgeschlossen wird. Die lokale Variable *pc* ist ein Zeiger auf den Typ *char* und erhält den gleichen Wert wie der übergebene Zeiger *text*. Das heißt, der übergebene Parameter *text* und die lokale Variable *pc* zeigen nun beide auf den Anfang der Zeichenkette, dessen Länge ermittelt werden soll. Nehmen wir an, die Funktion wird mit strlen("Hallo"); aufgerufen, so ergeben sich zu Beginn der Funktion folgende Eigenschaften der Variablen:



In der *while*-Schleife wird nun geprüft, ob der Zeiger *pc* auf das Zeichen '\0' zeigt. Ist das nicht der Fall, wird der Zeiger um eins erhöht. Damit zeigt jetzt die Variable *pc* auf das nächste Zeichen der Zeichenkette. Die Prüfung und die Erhöhung des Zeigers *pc* wird nun solange wiederholt, bis das Zeichen '\0' und damit das Ende der Zeichenkette gefunden wurde. Nach dem Durchlaufen der *while*-Schleife zeigt die Variable *pc* auf das Zeichen '\0' am Ende der Zeichenkette, während der Parameter *text* immer noch auf den Anfang der Zeichenkette zeigt. Die Differenz zwischen den Zeigern ist nun die Länge der Zeichenkette. (Der Code null am Ende der Zeichenkette wird nicht mitgezählt.) Am Ende er Funktion ergeben sich bei einem Aufruf mit `strlen("Hallo");` folgende Eigenschaften für die Variablen:



Die Funktion *strlen* ist eines der Standardbefehle von C und kann mit der Headerdatei <string.h> eingebunden werden, siehe auch den Anhang im Skript Programmieren 1.

1.3 Zeigervektoren

Wie alle anderen Datentypen können auch Zeiger zu Vektoren verkettet werden. Der Begriff *Zeigervektor* hat seine eigentliche Bedeutung im zweiten Teil, dem *Vektor*. Es handelt sich um einen *Vektor*, dessen Elemente *Zeiger* sind.

1.3.1 Definition von Zeigervektoren

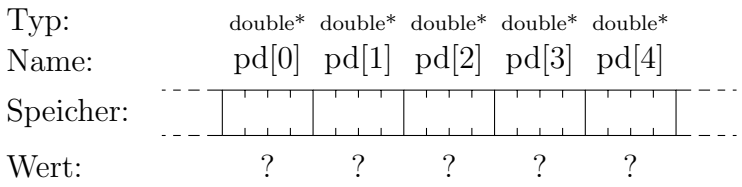
Allgemein wird ein Zeigervektor wie folgt defineirt:

```
<Datentyp> *<Variablenname>[<Anzahl>];
```

Beispiel: Es soll ein Zeigervektor mit Namen *pd* definiert werden, der fünf Zeiger beinhaltet, die auf den Datentyp *double* zeigen.

```
double *pd[5];
```

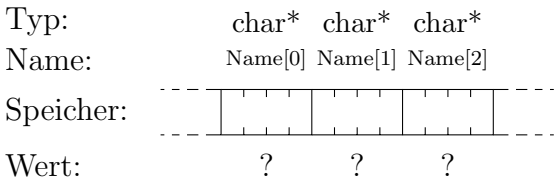
Es ergeben sich folgende Eigenschaften für diesen Vektor:



Auch wenn der Datentyp *double* acht Bytes beansprucht, benötigt ein Zeiger auf *double* nach wie vor vier Byte. Der eben definierte Zeigervektor benötigt für seine fünf Zeiger insgesamt 20 Bytes. Das folgende Beispiel definiert einen Vektor von drei Zeigern auf den Datentyp *char*:

```
char *Name[3];
```

Es ergeben sich folgende Eigenschaften für diesen Vektor:



Wieder benötigt jeder Zeiger jeweils vier Byte, auch wenn der Datentyp, auf den gezeigt wird, nur ein Byte beansprucht. Ein Zeiger speichert eine Adresse, und die ist in einem 32 Bit System immer vier Byte groß.

1.3.2 Initialisierung eines Zeigervektors

Als nächstes muss ein Zeigervektor initialisiert werden. Als Beispiel sollen die drei Zeiger im Zeigervektor *Name* auf sinnvolle Stellen im Speicher zeigen:

```
char a[] = "Hans";
char b[] = "Jutta";
char c[] = "Lu";
char *Name[3];

Name[0] = &a[0];
Name[1] = &b[0];
Name[2] = &c[0];
```

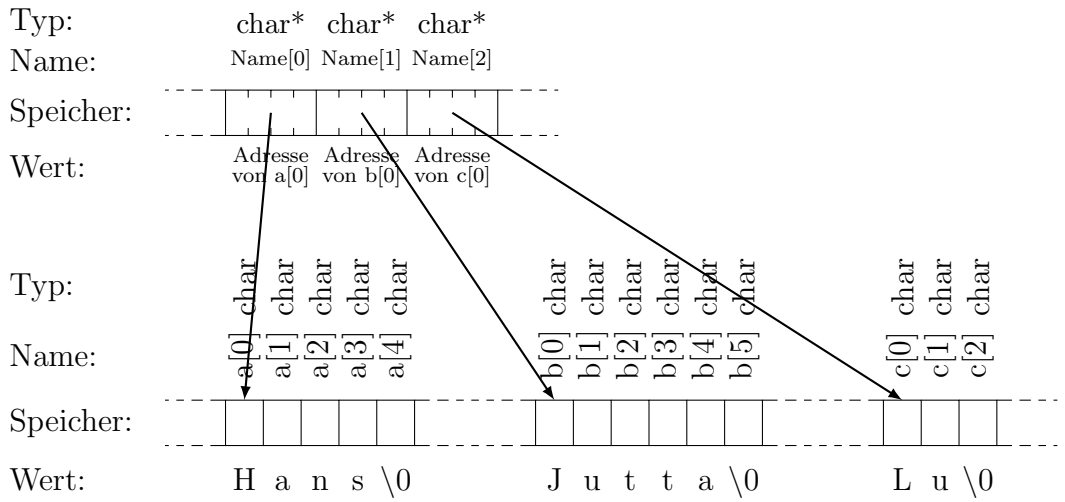
Es ist möglich Definition und Initialisierung in einem Schritt vorzunehmen:

```
char a[] = "Hans";
char b[] = "Jutta";
char c[] = "Lu";
char *Name[3] = { &a[0], &b[0], &c[0] };
```

Im Abschnitt 1.5.1 werden wir sehen, dass die Adresse des ersten Elementes eines eindimensionalen Vektors noch einfacher ermittelt werden kann:

```
char a[] = "Hans";
char b[] = "Jutta";
char c[] = "Lu";
char *Name[] = { a, b, c };
```

Bezogen auf die vier Eigenschaften der Variablen führen alle drei Beispiele zu dem selben Ergebnis:



1.3.3 Arbeiten mit Zeigervektoren

Ausgehend von der eben durchgeführten Initialisierung soll nun mit dem Zeigervektor gearbeitet werden. Was passiert zum Beispiel bei der folgenden Zeile?

```
*Name[0] = 'G';
```

Auf die Variable *Name* werden nacheinander zwei Operatoren angesetzt: Erst wird mit den eckigen Klammern das erste Element des Vektors gewählt, dann wird mit dem Stern auf den Inhalt, auf den der Zeiger zeigt, zugegriffen. Die Reihenfolge ergibt sich aus der Rangfolge der Operatoren, siehe Anhang A. Mit anderen Worten, der Ausdruck links vom Gleichheitszeichen spricht den Buchstaben an, auf den der erste Zeiger im Zeigervektor zeigt. Der Buchstabe *H* wird durch *G* ersetzt und aus *Hans* wird *Gans*.

Betrachten Sie nun folgende Zeile. Versuchen Sie, bevor Sie weiter lesen, selbst herauszubekommen, was die Zeile bewirkt.

```
*(Name[2]+1) = 'a';
```

In diesem Ausdruck finden wir auf der linken Seite vom Gleichheitszeichen drei Operatoren: eckige Klammern, das Pluszeichen und den Stern. Als erstes wird mit der eckigen Klammer der dritte Zeiger aus dem Zeigervektor gewählt, der auf den Buchstaben *L* des Wortes *Lu* zeigt. Danach wird zu dem Zeiger eins dazu addiert, sodass das Ergebnis der runden Klammer auf den Buchstaben *u* im Wort *Lu* zeigt. Schließlich wird mit dem Stern auf den Inhalt der Speicherstelle zugegriffen, auf den der ermittelte Zeiger verweist. Das heißt, das Ergebnis der linken Seite vom Gleichheitszeichen ist ein Zeichen (*char*) mit dem Inhalt *u*. Dieser Buchstabe wird nun mit dem Buchstaben a überschrieben, so dass der Name *Lu* zu *La* verunstaltet wird.

Der letzte Ausdruck kann auch wie folgt vereinfacht werden:

```
Name[2][1] = 'a';
```

Mehr Details zu dieser Schreibweise im Abschnitt 1.5.1.

1.3.4 Beispiel: Auswertung der Kommandozeilenparameter

Was verbirgt sich hinter diesem gewaltigen Wort *Kommandozeilenparameter*? Wenn Sie unter MS-DOS, Unix oder Linux ein Programm aufrufen, so können Sie diesem *Parameter* mit auf den Weg geben. Das Programm wird als ein *Kommando* aufgerufen und empfängt dabei bei Bedarf *Parameter*. Das gleiche gilt für Betriebssysteme mit grafischen Oberflächen: Wenn Sie ein Icon (z.B. ein Text-Dokument) auf ein anderes Icon (z.B. ein Textverarbeitungsprogramm) schieben, so wird das zweite Icon aufgerufen und bekommt Name und Pfad des ersten Icons als Kommandozeilenparameter übergeben.

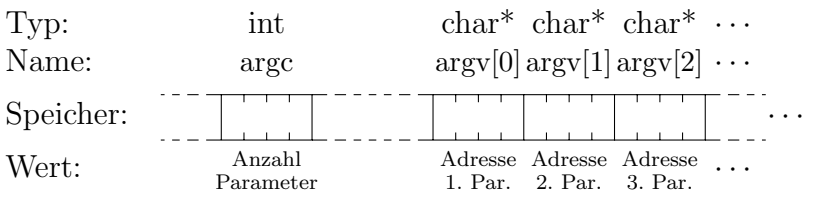
Wie werden die Kommandozeilenparameter von einem Programm ausgewertet? C bietet dazu im Hauptprogramm *main* die beiden Parameter *argc* und *argv*:

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

Der Parameter *argc* liefert die Anzahl der übergebenen Parameter '*ARGument Count*'. Da der eigene Programmname, mit dem das Programm aufgerufen wurde, immer als erster Parameter übergeben wird, ist der Wert von *argc* größer oder gleich eins.

Der Parameter *argv* liefert die Werte der Parameter, '*ARGument Values*'. Es handelt sich dabei um einen Zeigervektor auf *char*; eine Kette von Zeigern, die jeweils auf den Anfang von Zeichenketten zeigen, in denen die Parameter gespeichert sind.

Die folgende Skizze zeigt die vier Eigenschaften der Kommandozeilenparameter:

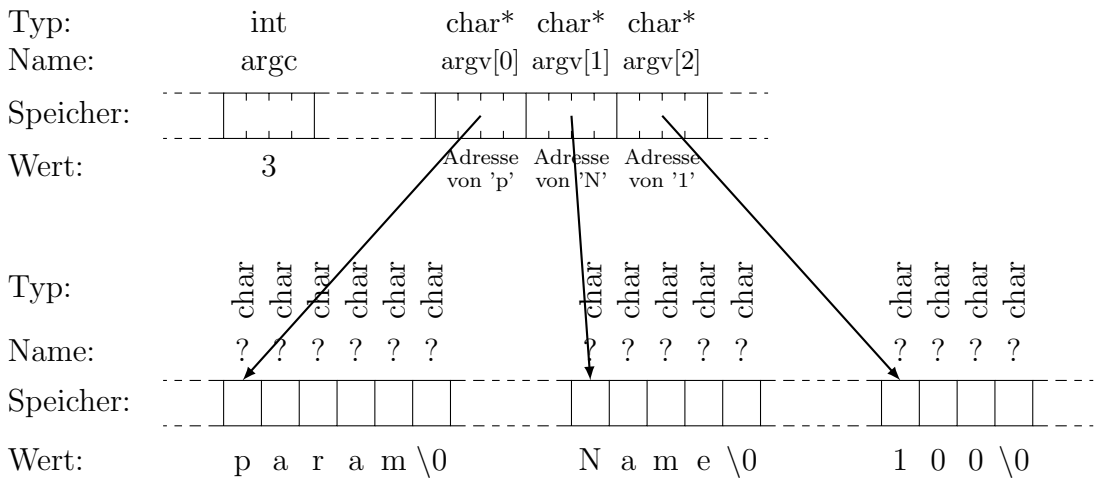


Zu beachten ist, dass nur argv[0] auf jeden Fall vorhanden ist. Bevor auf die weiteren Elemente des Zeigervektors zugegriffen wird, muss erst die Anzahl der Parameter in *argc* überprüft werden.

Das folgende Listing zeigt ein kleines vollständiges Programm, das alle empfangenen Parameter auf dem Bildschirm ausgibt.

```
#include <stdio.h>

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

    for(i=0; i<argc; i++)
        printf("%d. Parameter: %s\n", i+1, argv[i]);
}
```

Angenommen, das übersetzte Programm hat den Namen „*param.exe*“ und Sie rufen es mit „*param Name 100*“ auf, so ergeben sich folgende Eigenschaften für die Komandozeilenparameter:



Es ist wichtig zu beachten, dass alle Parameter als Zeichenketten empfangen werden. Das heißt, dass auch Zahlen als Zeichenfolgen übergeben werden. Soll eine solche Zahl im Programm als Zahl weiterverarbeitet werden, so muss der Parameter zunächst in eine Zahl umgewandelt werden. (Mögliche Funktionen sind z.B. *atoi()*, *atof()* oder *sscanf()*, siehe Anhang B.)

1.4 Zeiger auf Zeiger

Wie fast alles in C können auch Zeiger ineinander verschachtelt werden. Ein einfacher Zeiger zeigt auf Datentypen wie *int* oder *float*. Es ist nun möglich einen Zeiger zu definieren, der wieder auf einen Zeiger zeigt.

1.4.1 Definition eines Zeigers auf einen Zeiger

Die allgemeine Syntax zur Definition eines Zeigers auf einen Zeiger lautet:

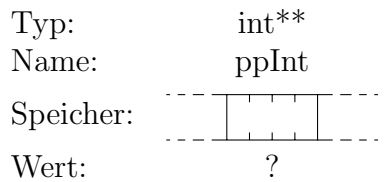
```
<Datentyp> **<Variablenname>;
```

Beispiel: Es soll ein Zeiger mit Namen *ppInt* definiert werden, der auf einen Zeiger zeigen soll, der seinerseits auf den Datentyp *int* zeigt.

```
int **ppInt;
```

Die Definition ähnelt stark der eines einfachen Zeigers. Die Anzahl der Sterne gibt an, wie stark der Zeiger verschachtelt ist. Gedanklich kann man auch **int*** *ppInt schreiben: Die Variable *ppInt* ist ein Zeiger auf den Datentyp *int**. Der Name *ppInt* wurde frei gewählt und kann durch einen beliebigen anderen Namen ersetzt werden.

Ein Zeiger auf einen Zeiger belegt genauso wie ein einfacher Zeiger in einem 32 Bit System vier Byte. Die folgende Skizze zeigt die vier Eigenschaft des eben definierten Zeigers auf einen Zeiger:

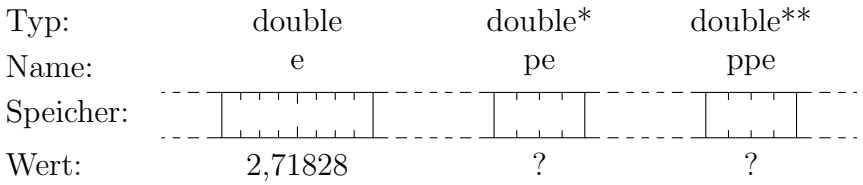


1.4.2 Initialisierung eines Zeigers auf einen Zeiger

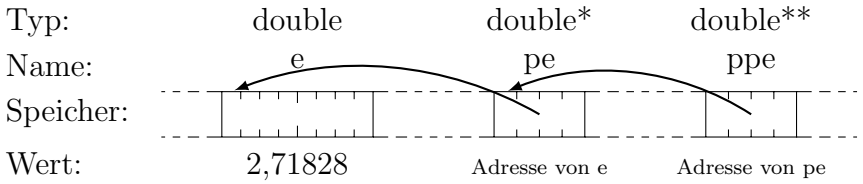
Nachdem ein Zeiger definiert wurde, zeigt er an eine unbekannte Stelle im Speicher. Um ihn an eine sinnvolle Stelle zeigen zu lassen, wird wieder der Adressoperator & benötigt. Beispiel:

```
1  double e=2.71828;
2  double *pe;
3  double **ppe;
4
5  pe = &e;
6  ppe=&pe;
```

In der ersten Zeile wird die Variable *e* vom Typ *double* definiert und es wird ihr gleich der Wert 2,71828 zugewiesen. In der zweiten Zeile erstellen wir uns einen einfachen Zeiger mit Namen *pe*, der auf den Datentyp *double* zeigt. In der dritten Zeile erzeugen wir uns einen Zeiger auf einen Zeiger auf *double* mit Namen *ppe*. Nach diesen drei Zeilen ergeben sich folgende Eigenschaften für die drei Variablen:



In der fünften Zeile des Beispielcodes wird nun zunächst der einfache Zeiger mit pe=&e initialisiert. Danach wird in der sechsten Zeile mit ppe=&pe dafür gesorgt, dass der Zeiger *ppe* auf die Variable *pe* zeigt. Es ergeben sich folgende Eigenschaften:



Definition und Initialisierung kann wieder in einer Zeile erfolgen:

```
double e=2.71828;
double *pe=&e;
double **ppe=&pe;
```

Im Folgenden soll der Speicher etwas genauer betrachtet werden. In Tabelle 1.2 sind die drei Variablen *e*, *pe* und *ppe* aus dem vorherigen Beispielcode dargestellt.

Die Variable *e* vom Typ *double* belegt acht Byte im Speicher und ist in diesem Beispiel unter Adresse 0012 FF78 abgelegt.

Die Variable *pe* vom Typ *double** benötigt wie alle 32-Bit-Zeiger vier Byte im Speicher und ist unter Adresse 0012 FF74 abgelegt. Der Wert der Variable *pe* ist die Adresse der Variable *e*, also 0012 FF78.

Schließlich, die Variable *ppe* vom Typ *double*** benötigt wieder vier Byte im Speicher und ist unter der Adresse 0012 FF70 abgelegt. Der Wert von *ppe* ist die Adresse von *pe*, also 0012 FF74.

Adresse	Datum	Datentyp	Inhalt	Beispiel
...				
0012 FF78	90	double	Zahl 2,71828	**double** e=2.71828;
0012 FF79	F7			
0012 FF7A	AA			
0012 FF7B	95			
0012 FF7C	09			
0012 FF7D	BF			
0012 FF7E	05			
0012 FF7F	40			
...				
0012 FF74	78	Zeiger auf double	Adresse 0012 FF78	**double** *pe=&e;
0012 FF75	FF			
0012 FF76	12			
0012 FF77	00			
...				
0012 FF70	74	Zeiger auf Zeiger auf double	Adresse 0012 FF74	**double** **ppe=&pe;
0012 FF71	FF			
0012 FF72	12			
0012 FF73	00			
...				

Tabelle 1.2: Prinzip eines Zeigers auf einen Zeiger, der auf double zeigt.

1.4.3 Arbeiten mit Zeigern auf Zeiger

Um mit dem Inhalt von Zeigern auf Zeiger zu arbeiten, wird wieder der Verweisoperator * verwendet. Ausgehend von dem Beispiel in Tabelle 1.2 soll der Wert der drei Variablen mittels des Zeigers auf Zeiger auf *double*, *ppe*, ausgegeben werden:

```
printf("Wert der Variable ppe: %p\n", ppe);
printf("Wert der Variable pe:  %p\n", *ppe);
printf("Wert der Variable e:   %lf\n", **ppe);
```

In der ersten Zeile wird die Adresse, die in der Variable *ppe* gespeichert ist, direkt ausgegeben. In der zweiten Zeile wird mit dem Verweisoperator auf die Variable *pe* zugegriffen und dessen Inhalt auf dem Bildschirm gebracht.

In der dritten Zeile finden wir vor der Variable *ppe* zwei Verweisoperatoren. Zunächst greift der rechte Stern mit der Variable *ppe* indirekt auf die Variable *pe* zu. Danach ermittelt der linke Stern mit dem Ergebnis des rechten Sterns den Inhalt der Variable *e*.

Es ergibt sich folgende Ausgabe:

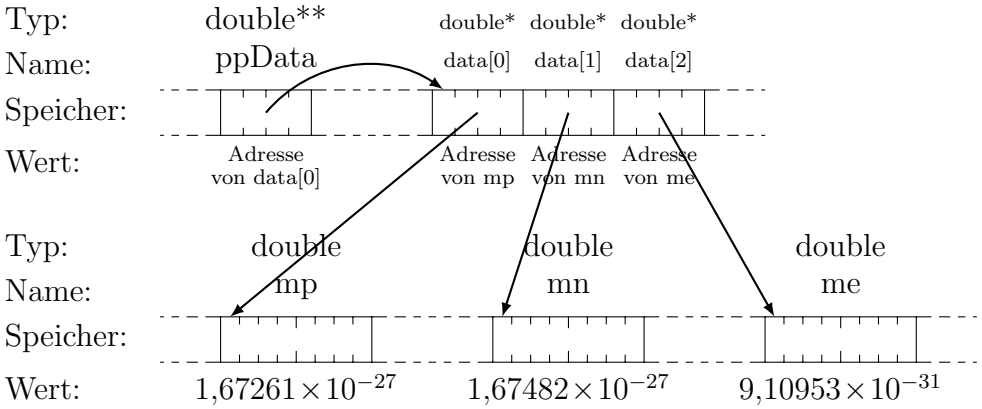
```
Wert der Variable ppe: 0012FF74
Wert der Variable pe:  0012FF78
Wert der Variable e:   2.718280
```

1.4.4 Beispiel: Zeiger auf Zeiger inkrementieren

Wir betrachten ein weiteres, etwas verschachteltes Beispiel:

```
double mp=1.67261e-27;
double mn=1.67482e-27;
double me=9.10953e-31;
double *data[] = { &mp, &mn, &me };
double **ppData=&data[0];
```

In den ersten drei Zeilen werden drei *double*-Variablen definiert und initialisiert (Masse von Proton, Neutron und Elektron in kg). In der vierten Zeile wird ein Zeigervektor mit drei Zeigern auf den Typ *double* definiert und mit den Adressen der drei zuvor definierten *double*-Variablen initialisiert. In der fünften Zeile wird ein Zeiger auf Zeiger auf *double* definiert und mit der Adresse des ersten Zeigers im zuvor definierten Zeigervektor initialisiert. Es ergeben sich folgende Eigenschaften für die Varaiablen:



Jetzt soll mittels des Zeigers auf Zeiger auf *double*, *ppData*, auf die drei *double*-Variablen zugegriffen werden indem der Verweisoperator * mehrfach angewendet wird. Die folgenden drei Zeilen erledigen das:

```
printf("Protonenmasse:   %lg kg\n", **ppData);
printf("Neutronenmasse:  %lg kg\n", **(ppData+1));
printf("Elektronenmasse: %lg kg\n", **(ppData+2));
```

In der ersten Zeile wird nur der Verweisoperator zweimal angewendet. Zunächst wird mit dem rechten Stern auf das erste Element des Zeigervektors *data* zugegriffen. Danach wird mit dem zweiten Stern auf die Stelle im Speicher zugegriffen, auf die das erste Element des Zeigervektors zeigt; das ist die Variable *mp*.

In der zweiten Zeile wird die Variable *ppData* zunächst um eins erhöht, bevor der Verweisoperator zweimal angewendet wird. Da der Zeiger auf einen Zeiger zeigt, und alle Zeiger vier Byte Speicherplatz benötigen, wird der addierte Wert mit vier multipliziert und die Adresse um vier erhöht. Der Ausdruck ppData+1 ist somit ein Zeiger auf data[1]. Mit den beiden Sternen wird dann die Variable *mp* angesprochen.

Die dritte Zeile ähnelt der zweiten, nur das die Variable *ppData* um zwei erhöht wird. Multipliziert mit vier wird die Adresse um acht erhöht und der Ausdruck ppData+2 zeigt auf data[2]. Zusammen mit den beiden Verweisoperatoren wird die Variable *me* ausgelesen.

Es ist auch möglich die Variable *ppData* jeweils auf das gesuchte Element im Zeigervektor zu „verbiegen“. Betrachten Sie folgende Zeilen:

```
printf("Protonenmasse:   %lg kg\n", **ppData++);
```

```
printf("Neutronenmasse:  %lg kg\n", **ppData++);
printf("Elektronenmasse: %lg kg\n", **ppData++);
ppData = &data[0];
```

Hier wird die Variable *ppData* inkrementiert nachdem sie zum Auslesen verwendet wurde. (Zur Erinnerung: Das ++ *hinter* der Variable bedeutet, dass die Variable erst *nach* der Verwendung im Ausdruck erhöht wird.) In jeder der drei ersten Zeilen wird erst die Variable ausgelesen, und danach der Zeiger um eins erhöht. Da es sich um einen Zeiger auf einen Zeiger handelt, wird die Adresse um den Wert vier erhöht, und der Zeiger zeigt auf den nächsten Zeiger.

Das einzige Problem bei dieser Methode ist, dass der Zeiger auf Zeiger auf *double* danach nicht mehr zu gebrauchen ist, da er im Speicher hinter den Zeigervektor zeigt. Mit der vierten Zeile wird er daher auf seinen ursprünglichen Wert zurückgesetzt.

Eine einfachere Schreibweise, die im Abschnitt 1.5.1 erläutert wird, ist diese:

```
printf("Protonenmasse:   %lg kg\n", *ppData[0]);
printf("Neutronenmasse:  %lg kg\n", *ppData[1]);
printf("Elektronenmasse: %lg kg\n", *ppData[2]);
```

Alle drei Beispielcodes erzeugen folgende Ausgabe:

```
Protonenmasse:    1.67261e-027 kg
Neutronenmasse:   1.67482e-027 kg
Elektronenmasse:  9.10953e-031 kg
```

1.4.5 Beispiel: Funktion zum Öffnen von Dateien

Wo können Zeiger auf Zeiger sinnvoll verwendet werden? Als Beispiel soll eine Datei mit Hilfe einer selbst erstellten Funktion geöffnet werden. Das ist ein Vorgriff auf Kapitel 2, der Fokus liegt aber auf den Zeigern.

Beim Arbeiten mit Dateien wird durchgängig ein Zeiger auf die Struktur *FILE* verwendet. Beim Öffnen der Datei wird der Zeiger auf eine Stelle im Speicher gesetzt, an der eine Instanz der Struktur *FILE* abgelegt ist. Alle weiteren Funktionen zum Arbeiten mit Dateien benötigen danach diesen Zeiger, um die geöffnete Datei zu benutzen.

Die hier zu erstellende Funktion soll eine Datei öffnen und eventuelle Fehler abfangen. Wir nennen die Funktion *openFile* und wollen die Datei *Beispiel.txt* öffnen. Der Aufruf erfolgt mit:

```
FILE *inp;
if(openFile("Beispiel.txt", "rt", &inp)) return -1;
```

Die Funktion *openFile* soll bei einem Fehler einen Wert ungleich null zurückgeben, so dass das Programm ggf. abgebrochen werden kann. Damit der Zeiger FILE *inp innerhalb der Funktion manipuliert werden kann, muss ein Zeiger auf diesen Zeiger übergeben werden. Die Implementierung erfolgt folgendermaßen:

```
int openFile(char fileName[], char mode[], FILE **file)
{
    *file = fopen(fileName, mode);
    if(*file==NULL) {
        printf("Fehler beim Öffnen der Datei '%s'!\n", fileName);
        return -1;
    }
```

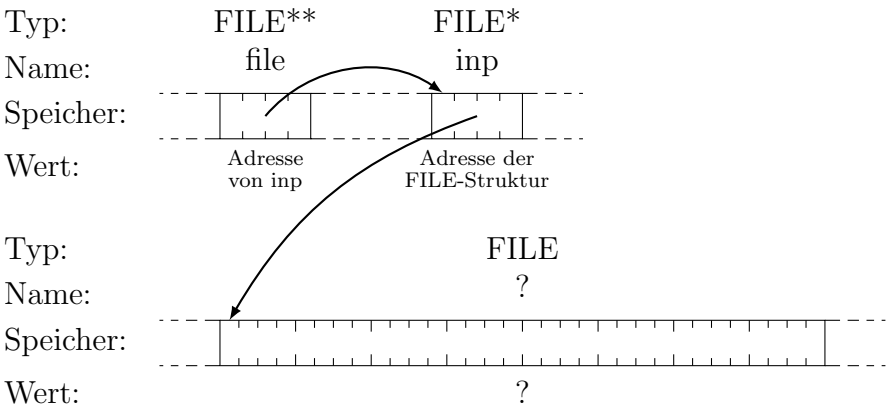
```
    return 0;
}
```

In der Funktion ist *file* ein Zeiger auf einen Zeiger, der auf den Datentyp *FILE* zeigt. Im ersten Schritt wird die Datei mit *fopen()* geöffnet. Die Funktion liefert einen Zeiger auf die Struktur *FILE* zurück. Das Ergebnis soll aber nicht dem Parameter *file* zugewiesen werden, sondern der Stelle im Speicher, auf den der Zeiger *file* zeigt. Das wird erreicht, in dem vorher der Verweisoperator * angewendet wird.

Die Funktion *fopen()* gibt beim erfolgreichen Öffnen der Datei eine sinnvolle Adresse im Speicher zurück. Tritt ein Fehler auf, so wird ein Zeiger auf die Adresse null zurückgegeben. In C wird ein Zeiger auf die Adresse null mit *NULL* bezeichnet.

Zur Fehlerabfrage wird geprüft, ob der Zeiger *NULL* zurückgegeben wurde. Ist dies der Fall, so wird eine Fehlermeldung ausgegeben und es wird der Wert -1 zurückgegeben. Tritt kein Fehler auf, so wird ohne eine Meldung der Wert null zurückgegeben.

Am Ende der Funktion *openFile()* haben die Variablen *file*, *inp* und die neue Instanz von der Struktur *FILE* folgende Eigenschaften:



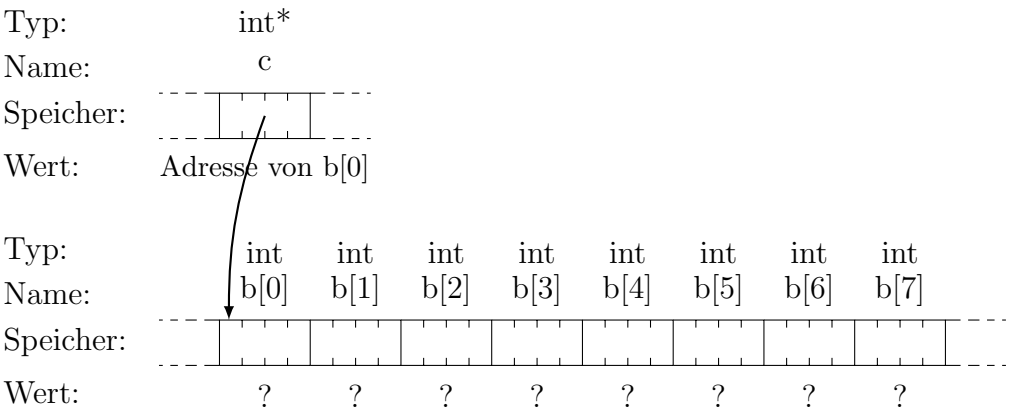
1.5 Vergleich Zeiger – Vektoren

1.5.1 Einfache Zeiger – eindimensionale Vektoren

Zeiger und Vektoren sind eng miteinander verwandt. Betrachten Sie folgende Definitionen:

```
int b[8];
int *c=&b[0];
```

Es ergeben sich für die beiden Variablen b und c folgende Eigenschaften:



Mit dem bisher gelernten sollten Sie die folgenden Zeilen nachvollziehen können, wobei jeweils zwei Zeilen die selbe Wirkung haben:

```
b[0] = 2;
*c = 2;        /* selbe Wirkung wie vorherige Zeile */
b[2] = 4;
*(c+2) = 4;    /* selbe Wirkung wie vorherige Zeile */
```

In der ersten und dritten Zeile wird direkt auf das erste, bzw. dritte Element im Vektor b zugegriffen. In der zweiten und vierten Zeile werden mit dem Zeiger c und dem Verweisoperator * indirekt die selben Elemente angesprochen.

Neu ist, dass in diesem Fall die Syntaxen für Zeiger Vektoren vertauscht werden können. Betrachten Sie folgende vier Zeilen, die alle die selbe Wirkung haben:

```
b[0] = 2;      /* alle vier Zeilen haben die selbe Wirkung */
*b = 2;
c[0] = 2;
*c = 2;
```

Erste und letzte Zeile sehen so aus, wie wir es bisher gewohnt sind. In der zweiten Zeile wird der Verweisoperator * auf den Vektor angesetzt, der in diesem Fall als Ergebnis das erste Element des Vektors zurück gibt. In der dritten Zeile wird mit den eckigen Klammern auf den Zeiger zugegriffen. In diesem Fall wird zunächst die in den Klammern angegebene Zahl zu dem Zeiger addiert, und dann die Speicherstelle, auf die der neue Zeiger zeigt, zurückgegeben.

Damit gibt es von dem Verweisoperator * zwei Varianten: Die erste Variante erwartet als Operanden einen Zeiger und liefert als Ergebnis die Speicherstelle, auf die der Zeiger zeigt. Die zweite Variante erwartet als Operanden einen Vektor und gibt das erste Element des Vektors zurück.

Bei den eckigen Klammern handelt es sich auch um einen Operator, der in zwei Varianten vorliegt. Die erste Variante erwartet als Operanden einen Vektor vor den Klammern und eine ganze Zahl in den Klammern, und liefert als Ergebnis das entsprechende Element aus dem Vektor. Die zweite Variante erwartet vor den Klammern einen Zeiger und in den Klammern wieder eine ganze Zahl; als Ergebnis wird zunächst die Zahl in den Klammern auf den Zeiger addiert und dann die Speicherstelle, auf die der neue Zeiger zeigt, zurückgegeben.

Auch die folgenden vier Zeilen haben alle die selbe Wirkung:

```
b[2] = 4;      /* alle vier Zeilen haben die selbe Wirkung */
*(b+2) = 4;
c[2] = 4;
```

```
*(c+2) = 4;
```

Hier kommt eine weitere Besonderheit hinzu: In der zweiten Zeile wird zu dem Vektor *b* die Zahl 2 addiert bevor mit dem Verweisoperator * das Element angesprochen wird. Mit anderen Worten, auch die Addition kann mit Zeigen und Vektoren arbeiten. Bei einem Zeiger wird die Adresse um den zweiten Summanden mal die Elementgröße erhöht. Bei einem Vektor wird das entsprechende Element angesprochen.

Wo kann dieser Mechanismus verwendet werden? Das folgende Programmstück gibt nur den rechten Teil einer Zeichenkette aus, in dem eine Zahl zum Vektor addiert wird:

```
char Text[] = "links rechts";
printf("Text: %s\n", Text+6);
```

Es wird folgende Ausgabe erzeugt:

```
Text: rechts
```

Bei den Parametern einer Funktion wird gerne von der Verwandtschaft zwischen Zeigern und Vektoren Gebrauch gemacht. Am Ende des Abschnitts 1.4 haben wir uns als Beispiel eine Funktion zum Öffnen einer Datei betrachtet. Wir haben dabei den Funktionskopf wie folgt gestaltet:

```
int openFile(char fileName[], char mode[], FILE **file)
```

Die ersten beiden Parameter *fileName* und *mode* werden hier als Vektor entgegengenommen. Sie könnten aber genauso als Zeigern übernommen werden:

```
int openFile(char *fileName, char *mode, FILE **file)
```

Egal für welche der beiden Formen Sie sich entscheiden, in beiden Fällen wird im Hintergrund der Parameter als Zeiger entgegengenommen. D.h. immer, wenn Sie einer Funktion einen Vektor als Parameter übergeben, wird dieser als Zeiger auf das erste Element entgegen genommen.

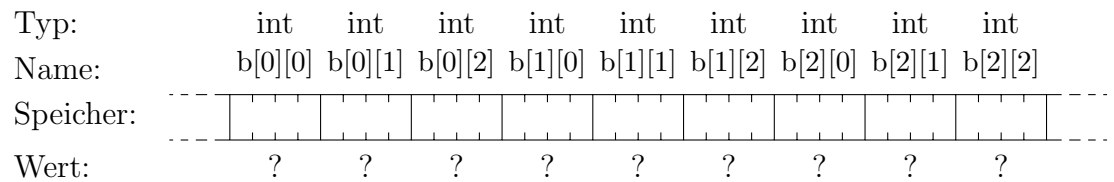
Vor diesem Hintergrund wird auch deutlich, dass bei der Übergabe eines Vektors keine Kopie übergeben wird. Tatsächlich wird ein Zeiger als Kopie übergeben, der aber auf das erste Element des Vektors der aufrufenden Funktion zeigt. Wird jetzt der Zeiger mit der Syntax eines Vektors angesprochen, wird direkt der Vektor der aufrufenden Funktion verwendet und nicht eine Kopie.

1.5.2 Zeigervektor – zweidimensionale Vektoren

Als nächstes wenden wir uns der zweiten Dimension zu. Es gibt zwei Möglichkeiten ein zweidimensionales Feld zu definieren. Der einfache Weg ist es, einen zweidimensionalen Vektor zu verwenden. Beispiel für ein Feld mit 3x3 Elementen:

```
int b[3][3];
```

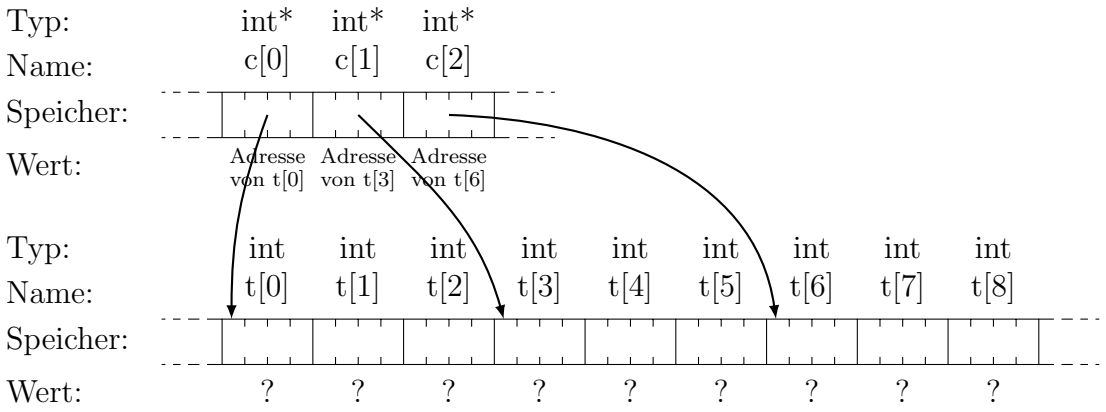
Die Variable *b* hat folgende Eigenschaften:



Als zweite Möglichkeit soll hier ein Zeigervektor verwendet werden:

```
int t[3*3];
int *c[3] = { t, t+3, t+6 };
```

Die Variable t wird nur zum Reservieren des Speichers verwendet. (Nach den beiden Zeilen hat der Variablenname t keine Bedeutung mehr.) Die Variable c ist ein Vektor mit drei Zeigern auf den Typ *int*. Die drei Zeiger werden bei der Definition gleich mit den Adressen jedes dritten Elements von t initialisiert. Wir verwenden hier jetzt die kürzere Schreibweise, bei der ein Vektor als ein Zeiger interpretiert wird. Es ergeben sich für c und t folgende Eigenschaften:



Nach diesem Programmstück können die Variablen b und c wieder gleichermaßen verwendet werden. (In diesem Beispiel haben die beiden Variablen aber unterschiedliche Speicherbereiche für sich reserviert.) Die zwei Zeilen

```
b[1][2] = 23;
printf("Wert bei (1;2): %d\n", b[1][2]);
```

würden mit der Variable c statt b zum selben Ergebnis führen.

Was sind nun die Unterschiede zwischen der Vektor- und der Zeiger-Version? Wir betrachten fünf Aspekte, die in Tabelle 1.3 zusammengefasst sind.

Speicherbedarf. Der zweidimensionale Vektor reserviert Speicher (nur) für seine 9 Elemente. Bei der Variante mit dem Zeigervektor wird extra Speicher für den Zeigervektor benötigt. Bei großen Feldern wird dieser Nachteil aber prozentual immer unbedeutender.

Geschwindigkeit. Bei der Vektor-Variante muss die Position im Speicher berechnet werden: Der erste Index mal die Länge einer Zeile plus dem zweiten Index. Bei der Vektor-Variante fällt die Multiplikation weg, was zu einer geringeren Verarbeitungszeit führt. Bei sehr rechenintensiven Aufgaben (z.B. Lösen eines Gleichungssystems mit vielen Unbekannten) wird dieser Effekt spürbar.

Erstellung. Die Vektor-Variante ist leichter zu erstellen.

Flexibilität. Bei einem zweidimensionalen Vektor sind alle Zeilen gleich lang. Bei der Variante mit dem Zeigervektor können die Zeiger auf beliebige Stellen im Speicher gesetzt werden. So können unterschiedliche Zeilenlängen realisiert werden (z.B. für das Dreieck von Pascal).

Speicherverteilung. Bei einem Vektor werden alle Elemente hintereinander im Speicher abgelegt, was bei sehr großen Datenmengen zu Problemen führen kann. Bei der Variante mit einem Zeigervektor können die Zeiger auf beliebige Stellen im Speicher zeigen, Dadurch können auch die Zeilen beliebig im Speicher verteilt werden.

	2d Vektor	**Zeigervektor**
Speicherbedarf	nur 9 Elemente (+)	9 Elemente + Zeigervektor (−)
Geschwindigkeit	extra Multiplikation (−)	keine extra Multiplikation (+)
Erstellung	simpel (+)	aufwändig (−)
Flexibilität	alle Zeilen gleich lang (−)	variable Zeilenlänge möglich (+)
Speicherverteilung	als ein Block (−)	kann verteilt werden (+)

Tabelle 1.3: Vergleich eines 3×3 Vektors mit einem entsprechenden Zeigervektor.

Eine allgemeine Bewertung des Vergleichs Zeigervektor contra 2d-Vektor lässt sich nicht treffen. Je nach Anwendungsgebiet ist das Eine dem Anderen vorzuziehen.

1.6 Prinzip beim Definieren von Variablen

Allgemein folgt die Definition von Variablen und Parametern in C einer Systematik: Auf der linken Seite findet sich ein elementarer Datentyp wie z.B. *int* oder *double*. Auf der rechten Seite steht der Name der neuen Variable umgeben von Operatoren. Die Variable auf der rechten Seite hat den Datentyp, der nach Anwendung der Operatoren den elementaren Datentyp auf der linken Seite ergibt. Beispiele:

```
int *a;
```

Auf die Variable *a* wird der Verweisoperator * angewendet, und das Ergebnis soll den Datentyp *int* haben. Das ist nur der Fall, wenn *a* den Datentyp Zeiger auf *int* hat. Von daher hat *a* den Typ *Zeiger auf int*.

```
double b[7];
```

Auf die Variable *b* werden die eckigen Klammern angesetzt und das Ergebnis hat den Typ *double*. Die eckigen Klammern werden auf einen Vektor angesetzt, so dass *b* ein *Vektor von double-Elementen* ist. Achtung: In den Klammern steht die *Anzahl* der Elemente und nicht der maximale Index, welcher immer um eins kleiner ist.

```
char *c[5];
```

Auf die Variable *c* werden hier zwei Operatoren angewendet. Hier ist es wichtig zu wissen, in welcher Reihenfolge die Operatoren angewendet werden. Im Anhang A sind alle Operatoren von C mit ihrer Rangfolge aufgelistet. In diesem Fall werden erst die eckigen Klammern angewendet und danach der Verweisoperator. Damit dabei der Datentyp *char* entsteht muss *c* ein *Vektor von Zeigern auf char* sein.

```
int (*d)[6];
```

In diesem Beispiel finden wir wieder die eckigen Klammern und den Verweisoperator. Durch die runden Klammern wird aber explizit angegeben, dass erst der Verweisoperator angewendet werden soll. Die Variable *d* ist von daher ein *Zeiger auf einen Vektor mit sechs int-Elementen*. Diese Variable benötigt nur vier Byte für einen Zeiger. Wird dieser Zeiger inkrementiert, so wird die gespeicherte Adresse um 6*4=24 Byte erhöht.

Später lernen wir noch weitere Datentypen und Strukturen kennen. Auch da kann sinngemäß diese Technik angewendet werden.

Hinweis: Sie können das Leerzeichen zwischen Datentyp und Variable auch an einer anderen Stelle platzieren oder ganz weglassen. In diesem Skript finden Sie aber durchgehend bei den Definitionen hinter dem elementaren Datentyp ein Leerzeichen. Damit soll der hier beschriebene Mechanismus verdeutlicht werden.

1.7 Aufgaben

Aufgabe 1.1: Definieren Sie Zeiger auf die Typen *int*, *double* und *char* mit den entsprechenden Namen *a*, *b* und *c*.

Aufgabe 1.2: Definieren Sie einen Zeiger auf einen Zeiger auf *double*.

Aufgabe 1.3: Ein Zeiger auf *int* (4 Byte) zeigt auf die Adresse 0123 3210_{16}. Auf welche Adresse zeigt der Zeiger, nachdem auf ihn zweimal der Inkrementoperator angewendet wurde?

Aufgabe 1.4: Ein Zeiger auf Zeiger auf *char* zeigt auf die Adresse 01ff $3e58_{16}$. Auf welche Adresse zeigt der Zeiger, nachdem auf ihn einmal der Dekrementoperator angewendet wurde?

Aufgabe 1.5: Ein Zeiger auf *char* mit Namen *text* zeigt auf den ersten Buchstaben der Zeichenkette „Guten Tag“. Wie wird der Buchstabe 'n' einzeln am sinnvollsten angesprochen?

Aufgabe 1.6: Die Elemente eines Zeigervektors mit Namen *Zahlen* zeigen auf den Typ *double*. Wie wird der Wert, auf den der dritte Zeiger zeigt, mit 3,141 überschrieben?

Aufgabe 1.7: Welchen Wert haben die Variablen x und y nach dem folgenden Programmstück?

```
double x=1.2, y=1.5, *px=&x, *py=&y;
*py = *px+0.8;
*px = *px**py;
```

Aufgabe 1.8: Ein Zeiger auf *int* mit Namen *pInt* zeigt auf das erste Element eines eindimensionalen Vektors gleichen Typs. Wie wird das dritte Element des Vektors am sinnvollsten angesprochen.

Aufgabe 1.9: Definieren Sie einen Zeigervektor mit neun Elementen. Der Vektor soll später den Typ *unsigned* verarbeiten.

Aufgabe 1.10: Definieren Sie einen Zeiger auf einen Vektor von acht *int*-Variablen.

Aufgabe 1.11: Betrachten Sie folgende Definitionen:

```
long Zahlen[] = { 7, 4, 5, 2, 7, 8 };
char *Name = "Werner Kleinhansen";
```

Was ist mit dieser Definition das Ergebnis folgender Ausdrücke?

Ausdruck	**Ergebnis**
Zahlen[3]	
Name[3]	
*Zahlen	
*Name	
*(Zahlen+2)	
*(Name+7)	

Kapitel 2

Umgang mit Dateien

2.1 Einleitung

In diesem Kapitel befassen wir uns mit zwei Arten von Dateien:

Text-Dateien. Im engeren Sinne handelt es sich bei einer Text-Datei um eine ASCII-Datei. Damit wird eine Datei bezeichnet, die nur Zeichen des *American Standard Code for Information Interchange*-Zeichensatzes enthält. Das sind bis auf wenige Steuerzeichen nur druckbare Zeichen die mit 7 Bit ($0...127_{10}$) kodiert sind. (Dabei wird Bit 7 von einem Byte immer auf 0 gesetzt.) Allgemeiner bezeichnet der Begriff Text-Datei jene Dateien, die nur lesbare Zeichen enthalten, auch wenn diese über den Kodierungsbereich von ASCII hinaus gehen. Text-Dateien lassen sich leicht erstellen und der Inhalt kann mit einem Texteditor gelesen und bearbeitet werden. Das Auslesen ist etwas aufwändiger und die Dateien werden üblicherweise größer.

Binär-Dateien. In eine Binärdatei können beliebige Zahlenwerte geschrieben werden, egal ob sie ein druckbares Zeichen repräsentieren oder nicht. Zum Beispiel können Dateien mit der Endung .exe mit einem normalen Texteditor nicht betrachtet oder gar bearbeitet werden, da diese zu weiten Teilen aus CPU-Befehlen für den Computer bestehen. Wird eine solche Datei als Binär-Datei behandelt, so kann sie ohne Probleme gelesen werden. In Binär-Dateien können Daten kompakter geschrieben werden und sind mit einfachen Editoren nicht lesbar.

Alle Dateien können binär gelesen werden. Soll eine Datei als Text-Datei gelesen werden, so muss diese auch eine Text-Datei sein. Im Folgenden betrachten wir zunächst Text-Dateien; danach wenden wir uns den Binär-Dateien zu. Im Anhang finden Sie eine ausführliche Beschreibung der für die Verarbeitung von Dateien verfügbaren Funktionen, siehe Abschnitt B.2.

2.2 Verarbeiten von Text-Dateien

2.2.1 Beispiel: Kopieren einer Text-Datei

Wir nähern uns dem Thema *Dateien* mit einem Beispielprogramm, das eine Text-Datei unverändert kopiert und somit eine neue Text-Datei erstellt.

```c
#include <stdio.h>

int main()
{
    char InpFileName[] = "Beispiel.txt";  /* Eingabedateiname */
    char OutFileName[] = "Beispiel.bak";  /* Ausgabedateiname */
    FILE *inp=NULL; /* Zeiger auf Eingabedatei-Struktur */
    FILE *out=NULL; /* Zeiger auf Ausgabedatei-Struktur */
    char ch;        /* aktuelles Zeichen */

    /* Erster Schritt: Dateien öffnen */
    inp = fopen(InpFileName, "rt");
    if(inp==NULL)
        printf("Konnte Eingabedatei nicht oeffnen: %s\n", InpFileName);
    out = fopen(OutFileName, "wt");
    if(out==NULL)
        printf("Konnte Ausgabedatei nicht oeffnen: %s\n", OutFileName);

    /* Zweiter Schritt: Dateien verarbeiten */
    if(inp && out) {
        ch = fgetc(inp);
        while(!feof(inp)) {
            fputc(ch, out);
            ch = fgetc(inp);
        }
    }

    /* Dritter Schritt: Dateien schließen */
    if(inp) fclose(inp);
    if(out) fclose(out);

    /* Schlussmeldung */
    if(inp && out) printf("Datei erfolgreich kopiert.\n");

    return 0;
}
```

Für das Arbeiten mit Dateien wird die Struktur *FILE* benötigt. Das Bearbeiten einer Datei erfolgt in drei Schritten: Als erstes muss eine Datei geöffnet werden. Danach kann sie in einem zweiten Schritt bearbeitet werden. Schließlich muss sie in einem dritten Schritt wieder geschlossen werden. Später sind Schritt 1 und 3 einfaches Handwerk, und die eigentliche Arbeit geschieht in Schritt 2. Im Folgenden gehen wir auf die drei Schritte ausführlicher ein.

2.2.2 Struktur zum Bearbeiten von Dateien: FILE

Alle Interaktionen mit einer Datei erfolgen über die Struktur *FILE*. Beim Öffnen der Datei wird diese Struktur erstellt und ein Zeiger darauf zurückgegeben. Alle anderen Datei-Funktionen erwarten diesen Zeiger auf die Dateistruktur.

Neben einigen anderen Daten wird in der Struktur *FILE* ein Puffer für die Ein- und Ausgabe zur Verfügung gestellt. Mit dem Puffer wird dafür gesorgt, dass nicht für jedes Byte, das gelesen oder geschrieben wird, ein eigener Zugriff auf den Datenträger erfolgt. Durch den Puffer wird die Programmlaufzeit z.T. deutlich reduziert.

2.2.3 Erster Schritt: Dateien öffnen

Das Öffnen einer Datei erfolgt mit der Funktion *fopen()*. Der erste Parameter gibt den Dateinamen mit oder ohne Pfad an. Der Dateiname kann dierekt als Zeichenkette oder indirekt, wie in dem Beispiel, mit einer Variablen angegeben werden.

Der zweite Parameter ist eine zweite Zeichenkette, die den Modus der Datei angibt, wobei jeder Buchstabe eine eigene Bedeutung hat: Der erste Buchstabe definiert, ob die Datei zum Lesen ('r' für *read*) oder Schreiben ('w' für *write*) geöffnet werden soll. Der Buchstabe 't' steht für *text* und gibt an, dass die Datei als eine Text-Datei behandelt werden soll.

Die Funktion gibt einen Zeiger auf die Struktur *FILE* zurück, mit der alle weiteren Dateizugriffe getätigt werden können. Tritt beim Öffnen der Datei ein Fehler auf (z.B. eine Datei zum Lesen ist nicht vorhanden) wird der Wert *NULL* zurückgegeben. *NULL* ist als ein Zeiger auf die Adresse null definiert. Um Fehler abzufangen, sollte nach dem Versuch eine Datei zu öffnen immer geprüft werden, ob der zurückgegebene Zeiger auf einen Wert ungleich *NULL* zeigt.

2.2.4 Zweiter Schritt: Dateien verarbeiten

In dem Beispielprogramm werden die Zeichen einzeln gelesen und geschrieben. Die Funktion *fgetc()* liest ein Zeichen aus der Datei. Sie bekommt als Argument den Zeiger auf die Dateistruktur *FILE* und liefert im Gegenzug ein Zeichen aus der Datei. Die Funktion *fputc()* schreibt ein Zeichen in eine Datei und erwartet als Parameter ein Zeichen und den Zeiger auf die Dateistruktur *FILE*.

Die Daten sind in der Datei sequentiell abgelegt. Stellen Sie sich zwei alte Tonbänder vor, auf denen die Daten hintereinander gespeichert werden. Mit jedem Aufruf von *fgetc()* und *fputc()* wir ein Zeichen gelesen, bzw. geschrieben und beide Bänder laufen ein kleines Stück weiter.

Damit die gedachten Bänder nicht bei jedem Dateizugriff gestartet und gestoppt werden müssen, werden die Daten blockweise in einen Puffer zwischengespeichert. Beim ersten Lesen werden z.B. 4096 Byte gelesen und im Arbeitsspeicher abgelegt. Erst wenn das 4097. Byte angefragt wird, erfolgt ein erneuter Zugriff auf die Datei. Beim Schreiben wird zunächst in den Arbeitsspeicher geschrieben, und erst wenn der Puffer voll ist, wird der ganze Block in die Datei geschrieben. Von außen bekommen Sie davon nichts mit, nur dass die Verarbeitung mit den Puffern viel schneller abläuft.

Wie wird das Dateiende erkannt? Die Funktion *feof()* püft nach, ob der letzte Lesevorgang über das Dateiende hinausgegangen ist. Ist dies der Fall, so gibt die Funktion *feof()* einen Wert ungleich null (true) zurück. War der letzte Lesezugriff aber noch erfolgreich, so gibt die Funktion den Wert null (false) zurück. Als Argument muss der Funktion der Zeiger auf die Dateistruktur *FILE* übergeben werden. In dem Beispiel prüft die *while*-Schleife nach jedem Lesevorgang nach, ob das Dateiende erreicht wurde. In Abhängigkeit von dem Ergebnis wird das Zeichen ausgegeben und das nächste gelesen, bzw. die Schleife abgebrochen.

2.2.5 Dritter Schritt: Dateien schließen

Nachdem eine Datei verarbeitet wurde, muss sie wieder geschlossen werden. Dazu dient die Funktion *fclose()* welche als Argument einen Zeiger auf eine Dateistruktur erwartet.

In vielen kleinen Programmen würden Sie es nicht bemerken, wenn Sie diese Anweisung weglassen. Das liegt daran, das C am Ende eines Programms alle offenen Dateien automatisch schließt. In der Praxis ist das aber der Ausnahmefall. Programme laufen länger und eine nicht geschlossene Datei macht bei einem späteren Zugriff Probleme. Daher sollte eine Datei nach der Verarbeitung *immer* geschlossen werden.

2.2.6 Umgang mit Fehlern

Beim Verarbeiten von Dateien kann es zur Laufzeit zu Fehlern kommen: Eine zu lesende Datei existiert nicht. Eine zu schreibende Datei existiert bereits und ist schreibgeschützt oder ist durch eine andere Anwendung blockiert.

In dem gezeigten Beispiel wurde dazu eine Konvention definiert: Hat der Zeiger für eine Datei einen Wert ungleich *NULL*, so wurde die Datei erfolgreich geöffnet. Hat er den Wert *NULL*, so wurde entweder die Datei noch nicht geöffnet oder es ist ein Fehler aufgetreten.

Um anzudeuten, dass die Dateien noch nicht geöffnet wurden, werden zu Beginn die Zeiger für die beiden Dateien mit *NULL* initialisiert. Wird eine Datei erfolgreich geöffnet, so nimmt der entsprechende Zeiger einen Wert ungleich *NULL* an. Konnte die Datei nicht geöffnet werden, so behält der Zeiger den Wert *NULL*.

Vor dem eigentlichen Kopieren wird zunächst geprüft, ob beide Zeiger einen Wert ungleich *NULL* haben, und nur dann wird der Kopiervorgang gestartet. Auch beim Schließen der Dateien wird erst geprüft, ob der entsprechende Zeiger ungleich *NULL* ist, bevor die Datei geschlossen wird. Und schließlich, die Schlussmeldung erfolgt nur, wenn beide Zeiger ungleich *NULL* sind.

2.3 Verarbeiten von Binärdateien

2.3.1 Beispiel: Lesen und Schreiben einer Binärdatei

In diesem Abschnitt sind zwei Beispiele aufgeführt: Das erste Beispiel erstellt und schreibt in eine Binärdatei. Das zweite Beispiel liest die im ersten Beispiel erstellte Binärdatei wieder aus.

Schreiben einer Binär-Datei:

```
#include <stdio.h>

int main()
{
    int i = 27;              /* ganze Zahl zum Schreiben */
    double e = 2.718281828; /* Gleitkommazahl zum Schreiben */
    int Prim[6] = {          /* Vektor zum Schreiben */
        2, 3, 5, 7, 11, 13 };
    FILE *out = NULL;        /* Zeiger für Ausgabedatei */

    /* Erster Schritt: Datei öffnen */
    out = fopen("Beispiel.dat", "wb");
    if(out==NULL)
    printf("Konnte Ausgabedatei nicht \224ffnen: Beispiel.dat\n");

    /* Zweiter Schritt: Datei verarbeiten */
```

```
    if(out) {
        fwrite(&i, sizeof(int), 1, out);
        fwrite(&e, sizeof(double), 1, out);
        fwrite(Prim, sizeof(int), 6, out);
    }

    /* Dritter Schritt: Datei schließen */
    if(out) fclose(out);

    /* Schlussmedung */
    printf("Datei erfolgreich erstellt\n");

    return 0;
}
```

Lesen einer Binär-Datei:

```
#include <stdio.h>

int main()
{
    int i;          /* ganze Zahl zum Lesen */
    double e;       /* Gleitkommazahlzum Lesen */
    int Prim[6];    /* Vektorzum Lesen */
    FILE *inp=NULL; /* Zeiger für Eingabedatei */

    /* Erster Schritt: Datei öffnen */
    inp = fopen("Beispiel.dat", "rb");
    if(inp==NULL)
    printf("Konnte Eingabedatei nicht \224ffnen: Beispiel.dat\n");

    /* Zweiter Schritt: Datei verarbeiten */
    if(inp) {
        fread(&i, sizeof(int), 1, inp);
        fread(&e, sizeof(double), 1, inp);
        fread(Prim, sizeof(int), 6, inp);
    }

    /* Dritter Schritt: Datei schließen */
    if(inp) fclose(inp);

    /* Ergebnisse auusgeben */
    if(inp) {
        printf("i: %d\n", i);
        printf("e: %.15g\n", e);
        printf("Prim[0]: %d\n", Prim[0]);
        printf("Prim[1]: %d\n", Prim[1]);
        printf("Prim[2]: %d\n", Prim[2]);
        printf("Prim[3]: %d\n", Prim[3]);
        printf("Prim[4]: %d\n", Prim[4]);
        printf("Prim[5]: %d\n", Prim[5]);
    }

    return 0;
}
```

2.3.2 Erster Schritt: Datei öffnen

Eine Binär-Datei wird wieder mit der Funktion *fopen()* geöffnet. Die Funktion übernimmt als ersten Parameter den Dateinamen als Zeichenkette. Der zweite Parameter gibt den Modus an, mit der die Datei geöffnet werden soll. Die Buchstaben 'w' und 'r' stehen für schreiben (engl. *write*) und lesen (engl. *read*). Der Buchstabe 'b' gibt an, dass es sich um eine Binär-Datei (engl. *binary*) handelt.

2.3.3 Zweiter Schritt: Datei verarbeiten

Zum Verarbeiten von Binärdateien stehen die Funktionen *fwrite()* und *fread()* zur Verfügung. Beide erwarten vier Parameter mit annähernd gleicher Bedeutung:

Der erste Parameter ist ein Zeiger auf eine Stelle im Speicher. Für *fwrite()* müssen dort die Daten stehen, die in die Datei geschrieben werden sollen. Die Funktion *fread()* schreibt an die Stelle, auf die der Zeiger zeigt, die Daten, die aus der Datei gelesen werden.

Der zweite Parameter gibt an, wie groß eine Einheit ist, die gelesen oder geschrieben werden soll. Hilfreich ist dafür die Funktion *sizeof()*, die ermittelt, wieviel Speicher die Variable oder der Typ in der Klammer benötigt.

Der dritte Parameter steht für die Anzahl der Einheiten, die gelesen werden sollen. Wenn ein Vektor (array) gelesen oder geschrieben werden soll, muss hier die Anzahl der Elemente eingetragen werden.

Der vierte Parameter schließlich ist ein Zeiger auf die Dateistruktur der Datei *FILE*, in die geschrieben bzw. aus der gelesen werden soll.

Die beiden Funktionen geben die Anzahl der erfolgreich gelesenen Einheiten zurück. Der Wert kann für eine Überprüfung des Schreib- und Lesevorgangs verwendet werden. Weicht dieser von der angegebenen Anzahl der Einheiten ab (3. Parameter), so liegt ein Fehler vor.

2.3.4 Dritter Schritt: Datei schließen

Wie immer muss auch hier die Datei am Ende geschlossen werden. Dies geschieht wieder mit der Funktion *fclose()* die als Parameter einen Zeiger auf die Dateistruktur *FILE* erwartet.

2.4 Aufgaben

Aufgabe 2.1: Definieren Sie eine Variable für die Verarbeitung einer Datei.

Aufgabe 2.2: Es soll eine Textdatei mit Namen *Brief.txt* gelesen werden. Wie sieht der Befehl zum Öffnen der Datei aus?

Aufgabe 2.3: Worin unterscheiden sich Binär- von Textdateien?

Aufgabe 2.4: Wozu wird die Struktur *FILE* benötigt?

Aufgabe 2.5: Öffnen Sie eine Binärdatei mit Namen *Address.dat* zum Schreiben.

Aufgabe 2.6: Wie lauten die beiden Funktionen zum Lesen und Schreiben von Binärdateien?

Aufgabe 2.7: Was muss am Ende mit einer geöffneten Datei immer geschehen?

Aufgabe 2.8: Geben Sie die drei *int*-Variablen *a*, *b* und *c* jeweils mit acht Zeichen Breite ohne führende nullen mit einem Zeilenumbruch am Ende in eine Textdatei aus. Der Stream der geöffneten Datei lautet *out*.

Aufgabe 2.9: Mit welcher Funktion können Sie prüfen, ob das Ende einer Datei überschritten wurde?

Aufgabe 2.10: Mit welcher Funktion können Sie zwischendurch dafür sorgen, dass alle Daten im Puffer sofort in die Datei geschrieben werden? (siehe Anhang)

Aufgabe 2.11: Mit welcher Funktion kann eine Datei gelöscht werden? (siehe Anhang)

Aufgabe 2.12: Erstellen Sie ein Programm, welches eine Tabelle für Temperaturen in Fahrenheit und Celsius in eine CSV-Datei (Comma Separated Values) schreibt. Die Umrechnung erfolgt mit $\frac{\vartheta_C}{^\circ\mathrm{C}} = \frac{5}{9}(\frac{\vartheta_F}{^\circ\mathrm{F}} - 32)$. Die Datei soll wie folgt aussehen:

```
Fahrenheit,Celsius
0.00,−17.78
50.00,10.00
100.00,37.78
150.00,65.56
```

Kapitel 3

Spezielle Datentypen

Neben den elementaren Datentypen wie z.B. *char* und *float* ist es möglich, eigene Datentypen zu definieren. Dieses Kapitel behandelt einige spezielle Datenkonstrukte, die dem Programmierer helfen, seine Daten besser zu strukturieren.

3.1 Strukturen mit `struct`

In einer Struktur werden mehrere Datenelemente zusammengefasst. Zum Beispiel können die Elemente einer Adresse mit Straße, Hausnummer, Postleitzahl und Ort in einer Struktur gebündelt werden. Auf solche Strukturen gehen die folgenden Abschnitte ein.

3.1.1 Deklaration von Strukturen

Das folgende Beispiel bündelt die drei Elemente eines Datums (Tag, Monat und Jahr) zu einer Struktur. Zur Darstellung unterschiedlicher Datentypen innerhalb einer Struktur wird der Monat hier als Text gespeichert.

```
struct sDatum {
    int Tag;
    char Monat[10];
    int Jahr;
};
```

Die Deklaration einer Struktur beginnt mit dem Schlüsselwort *struct* gefolgt von einem Bezeichner, dem Namen der Struktur, der frei gewählt werden kann. Danach folgt in geschweiften Klammern die Liste der Elemente, die zu der Struktur gehören. Die Deklaration der Elemente erfolgt nach dem gleichen Schema wie sonst Variablen definiert werden, nur muss auf die zusätzliche Festlegungen mit *static* und *register* verzichtet werden. Die Deklaration einer Struktur wird mit einem Semikolon abgeschlossen.

Abbildung 3.1 zeigt wie die Elemente der Struktur *sDatum* im Speicher angeordnet werden. Die Elemente werden nacheinander im Speicher abgelegt nur dass zwischen den Elementen ggf. kleine Lücken bleiben. (Die Lücken entstehen, wenn der Compiler alle Elemente auf ein festgelegtes Vielfaches an Bytes ausrichtet. In der Praxis hat das aber keine Auswirkung auf die Arbeitsweise mit Strukturen.) Die tatsächliche Größe einer Struktur kann mit *sizeof* ermittelt werden.

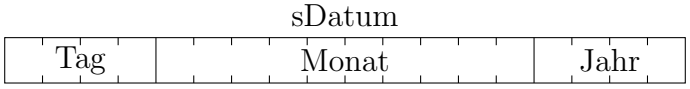


Abbildung 3.1: Anordnung der Struktur `sDatum` im Speicher.

Mit dieser *Deklaration* wird dem Compiler die neue Struktur mitgeteilt, aber es wird noch keine Variable mit dieser Struktur erzeugt. Das geschieht mit der *Definition*.

3.1.2 Definition von Strukturen

Für die Definition einer Struktur wird wieder der Begriff *struct* verwendet, gefolgt von dem Namen, der bei der Deklaration vergeben wurde. Danach werden die Varaiablennamen aufgeführt. In unserem Beispiel führen wir eine Variable mit dem Namen *Datum* ein:

```
struct sDatum Datum;
```

Die Initialisierung erfolgt ähnlich wie bei Vektoren mit geschweiften Klammern hinter der Definition:

```
struct sDatum Datum = { 12, "Februar", 2020 };
```

Die Deklaration und Definition kann kombiniert werden:

```
struct sDatum {
    int Tag;
    char Monat[10];
    int Jahr;
} Datum;
```

Wenn eine Struktur nur an einer Stelle verwendet wird, kann der bei der Deklaration vergebene Name weggelassen werden:

```
struct {
    int Tag;
    char Monat[10];
    int Jahr;
} Datum;
```

Es können auch Zeiger auf Strukturen und Vektoren von Strukturen gebildet werden:

```
struct sDatum Fussball[7], *pDatum;
```

3.1.3 Arbeiten mit Strukturen

Um auf ein Element einer Struktur zuzugreifen, wird ein Punkt verwendet. Folgende Programmzeile gibt ein formatiertes Datum aus:

```
printf("Datum: %d. %s %d\n", Datum.Tag, Datum.Monat, Datum.Jahr);
```

Das Ergebnis dieser Zeile sieht so aus:

```
Datum: 12. Februar 2020
```

Die Elemente können wie einzelne Variablen behandelt werden. Um den 3. Oktober 2019 zu erzeugen, könnten folgende Zeilen verwendet werden:

```
Datum.Tag = Datum.Tag-9;
strcpy(Datum.Monat, "Oktober");
Datum.Jahr--;
```

Anders als Vektoren können Strukturen auch als Ganzes kopiert werden:

```
struct sDatum Kopie, Datum = { 13, "April", 2007 };

Kopie = Datum;
```

Auch Vektoren innerhalb von Strukturen werden komplett kopiert, was in dem Beispiel bei dem Zeichenvektor *Monat* passiert ist.

Wird mit einem Zeiger auf eine Struktur gearbeitet, so gibt es einen speziellen Operator, der etwas Tipparbeit einspart. Wir gehen von folgender Definition aus:

```
struct sDatum Datum, *pDatum=&Datum;
```

Mit den uns bekannten Mitteln kann der Zeiger wie folgt verwendet werden:

```
(*pDatum).Tag = 17; /* nicht so schön */
```

Einfacher ist aber folgende Zeile, die zum selben Ergebnis führt:

```
pDatum->Tag = 17;    /* besser */
```

Links von dem Operator '->' steht ein Zeiger auf eine Struktur, rechts davon wird das Element eingetragen, das angesprochen werden soll.

Wenn mit einem Zeiger auf eine Struktur gerechnet wird, so wird bei jeder Erhöhung um eins der Zeiger um die Größe der Struktur erhöht.

3.1.4 Bitweise Strukturen

Alle Datentypen, die wir bisher behandelt haben, benötigen im Speicher ein ganzzahliges Vielfaches von einem Byte. Der Typ *char* benötigt ein Byte, *int* vier Byte, *double* acht Byte etc. Hier wollen wir uns einer Struktur zuwenden, die Elemente beliebiger Größe beinhaltet.

Die Struktur im folgenden Beispiel ist gerade mal vier Byte groß:

```
struct sDatum {
    unsigned Tag:5;
    unsigned Monat:4;
    unsigned Jahr:12;
    int FreierTag:1;
    int Sommer:1;
    int Reserviert:9;
};
```

Die Größe der Elemente wird hinter dem Namen des Elementes mit einem Doppelpunkt in Bit (nicht Byte) angegeben. In dem Beispiel werden für das Speichern eines Tages (Werte von 1 bis 31) nur fünf Bit benötigt (Werte von 0 bis 31), der Monat (Werte von 1 bis 12) benötigt nur 4 Bit (Werte von 0 bis 15) etc.

Die Elemente können wie gewohnt mit einem Punkt '.' oder Pfeil '->' angesprochen werden. Die folgenden Zeilen zeigen, wie eine Instanz der oben deklarierten Struktur verwendet wird:

```
struct sDatum Datum = { 10, 4, 2020, 1, 0 };

printf("Datum: %d.%d.%d\n", Datum.Tag, Datum.Monat, Datum.Jahr);
printf("Es ist%s Sommer.\n", Datum.Sommer?"":" kein");
printf("Heute haben wir%s frei.\n", Datum.FreierTag?"":" nicht");
printf("Anzahl der Bytes: %d\n", sizeof(Datum));
```

Es wird folgende Ausgabe erzeugt, und die Anordnung im Speicher wird in Abbildung 3.2 dargestellt.

```
Datum: 10.4.2020
Es ist kein Sommer.
Heute haben wir frei.
Anzahl der Bytes: 4
```

Werden die insgesamt 32 Bit der Struktur als ein 32-Bit Integer betrachtet, so ergibt sich binär das Muster wie in Abbildung 3.2 dargestellt.

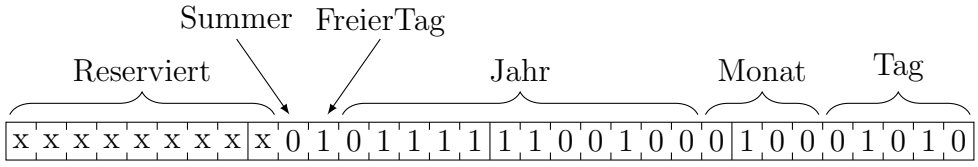


Abbildung 3.2: Anordnung der bitweisen Struktur *sDatum* im Typ *int*.

Die Möglichkeit, einen Speicherbereich gleichzeitig als Struktur und als Integer zu behandeln, lernen wir im Abschnitt 3.3 kennen. Ausgehend für den Wert 0 im Element *Reserviert* ergibt sich ein Dezimalwert von 3131530. Es ist zu beachten, dass die vier Bytes auf unseren Rechnern im Labor in umgekehrter Reihenfolge gespeichert werden.

Wo kann von bitweisen Strukturen Gebrauch gemacht werden? Immer wenn es darum geht, ein Byte in seine Bestandteile zu zerlegen, bietet sich dieses Verfahren an. Wenn Sie in die Verlegenheit geraten, einen digitalen Signalprozessor, DSP, oder einen Controller zu programmieren, so werden die Funktionen des Bauteils häufig über die einzelnen Bits einer definierten Speicheradresse gesteuert. Wenn Ihr Programm genau an diese Stelle eine Variable mit einer passenden Struktur wie hier beschreiben platziert, so können Sie die einzelnen Funktionen über die Struktur ansprechen.

3.1.5 Verschachtelte Strukturen

Es ist möglich, innerhalb einer Struktur eine weitere Struktur zu verwenden. Dazu folgendes Beispiel:

```
struct sDatum {
    int Tag, Monat, Jahr;
};

struct sPerson {
    char Nachname[31];
    char Vorname[31];
    struct sDatum Geburt;
};
```

```
struct sSchulKlasse {
    char KlassenName[11];
    int AnzahlSchueler;
    struct sPerson Schueler[50];
};
```

Die Struktur *sSchulKlasse* beinhaltet neben zwei Variablen für den Namen der Klasse und die Anzahl der Schüler einen Vektor für alle Schüler der Klasse. Für jeden Schüler wurde eine Struktur mit Nachnamen, Vornamen und Geburtsdatum angelegt. Das Geburtsdatum ist wiederum eine Struktur mit den Elementen *Tag*, *Monat* und *Jahr*.

Eine Struktur kann auch vollständig innerhalb einer anderen Struktur deklariert werden. So kann die Struktur *sPerson* aus dem vorherigen Beispiel auch wie folgt beschrieben werden:

```
struct sPerson {
    char Nachname[31];
    char Vorname[31];
    struct sDatum { int Tag, Monat, Jahr; } Geburt;
};
```

Wird die Struktur *sDatum* an keiner anderen Stelle benötigt, so kann ihr Bezeichner auch weggelassen werden:

```
struct sPerson {
    char Nachname[31];
    char Vorname[31];
    struct { int Tag, Monat, Jahr; } Geburt;
};
```

Die Elemente einer verschachtelten Struktur werden entsprechend verschachtelt angesprochen. In dem folgenden Beispiel wird eine Instanz der eben erstellten Struktur definiert und das Geburtsjahr des vierten Schülers auf 1995 gesetzt. Danach bekommt der sechste Schüler den Nachnamen *Mustermann* und die Anzahl der Schüler wird auf 23 gesetzt:

```
struct sSchulKlasse Klasse;
Klasse.Schueler[3].Geburt.Jahr = 1995;
strcpy(Klasse.Schueler[5].Nachname, "Mustermann");
Klasse.AnzahlSchueler = 23;
```

Strukturen können auch mit den unten aufgeführten Datentypen *enum* und *union* kombiniert werden.

3.2 Aufzählungen mit `enum`

3.2.1 Beispiel für eine Aufzählung (enum)

Häufig wird in einer Variablen ein Wert aus einer begrenzten Anzahl von Möglichkeiten gepeichert. Angenommen, es soll die aktuelle Jahreszeit ermittelt und gespeichert werden, so gibt es nur die vier Möglichkeiten *Frühling*, *Sommer*, *Herbst* und *Winter*. (Die fünfte Jahreszeit kennen wir hier im Norden nicht...) Es handelt sich dabei um eine *Aufzählung* der Jahreszeiten.

Eine mögliche Lösung ist, eine *int*-Variable zu verwenden, den Jahreszeiten die Werte null bis drei zu geben und zur besseren Lesbarkeit des Programms mit *#define* Namen zu vergeben:

```
#include <stdio.h>

#define FRUEHL 0
#define SOMMER 1
#define HERBST 2
#define WINTER 3

int main()
{
    int Jahreszeit;

    Jahreszeit = HERBST;

    switch(Jahreszeit) {
    case FRUEHL: printf("Frühling\n"); break;
    case SOMMER: printf("Sommer\n"); break;
    case HERBST: printf("Herbst\n"); break;
    case WINTER: printf("Winter\n"); break;
    }

    return 0;
}
```

C bietet speziell für *Aufzählungen* den Datentyp *enum*. Das Beispiel mit *enum* realisiert ergäbe folgendes Programm:

```
#include <stdio.h>

int main()
{
    enum eJahreszeit { Fruehl, Sommer, Herbst, Winter };
    enum eJahreszeit Jahreszeit;

    Jahreszeit = Herbst;

    switch(Jahreszeit) {
    case Fruehl: printf("Frühling\n"); break;
    case Sommer: printf("Sommer\n"); break;
    case Herbst: printf("Herbst\n"); break;
    case Winter: printf("Winter\n"); break;
    }

    return 0;
}
```

3.2.2 Deklaration einer Aufzählung (enum)

Ähnlich wie bei Strukturen wird zunächst der neue *enum*-Typ *deklariert*. Nach dem Schlüsselwort *enum* und dem frei wählbaren Namen (hier *sJahreszeit*) folgt in geschweiften Klammern die Liste der Namen, die zu dieser Aufzählung gehören. Es können beliebige Namen in beliebiger Anzahl angegeben werden. Es gelten die Regeln, die auch für Variablennamen gelten. Die Deklaration wird mit einem Semikolon

abgeschlossen.

Die Elemente der Aufzählung werden beginnend mit null durchnummeriert. Es können auch gezielt andere Werte vergeben werden, indem hinter dem Elementnamen mit einem Gleichheitszeichen getrennt eine ganze Zahl eingetragen wird. Wird ein Wert für ein Element vorgegeben, so erhalten die folgenden Elemente ohne Angabe eines Wertes die nächst höheren Werte. Beispiel:

```
enum eJahreszeit { Fruehl=1, Sommer, Herbst, Winter };
```

Hier bekommt *Fruehl* explizit den Wert 1, und damit implizit *Sommer* den Wert 2, *Herbst* den Wert 3 und *Winter* den Wert 4.

Es können auch Zahlenwerte doppelt vergeben werden. Im folgenden Beispiel wird mit *enum* ein Typ deklariert, der nur die Werte *Falsch* und *Wahr* annehmen kann. Damit der gleiche Typ auch in englischsprachigen Programmen verwendet werden kann, sollen parallel die englischen Begriffe *False* und *True* verwendet werden:

```
enum eBool { Falsch, Wahr, False=0, True };
```

3.2.3 Definition einer Aufzählung (enum)

Die Definition erfolgt wie bei den Strukturen. Nach dem Schlüsselwort *enum* folgt der zuvor vergebene Name (in unserem Beispiel *eJahreszeit*) und ein frei wählbarer Variablenname (im Beispiel *Jahreszeit*).

```
enum eJahreszeit Jahreszeit;
```

Wie bei anderen Variablen kann bei der Definition gleich ein Wert zugewiesen werden:

```
enum eJahreszeit Jahreszeit=Herbst;
```

Deklaration und Definition kann in einem Schritt erfolgen:

```
enum eJahreszeit { Fruehl, Sommer, Herbst, Winter } Jahreszeit;
```

Wird eine Aufzählung nur an einer Stelle verwendet, so kann der Name des Typs entfallen:

```
enum { Fruehl, Sommer, Herbst, Winter } Jahreszeit;
```

Eine *enum*-Variable wird intern als *int* gespeichert und benötigt auf 32 Bit-Systemen 4 Byte.

Variablen vom Typ *enum* können wie andere Variablen verkettet werden, oder Teil einer Struktur sein.

```
enum eBool Logik[10];
struct sDatum {
    int Tag, Monat, Jahr;
    enum eJahreszeit Jahreszeit;
    enum eBool Feiertag;
} Datum;
```

3.2.4 Arbeiten mit Aufzählungen (enum)

Eine Variable vom Typ *enum* wird wie jede andere Variable angesprochen. Während z.B. eine Variable vom Typ *char* als Zustand einen ganzzahligen Wert zwischen -128 und +127 annehmen kann, kann eine Variable vom Typ *enum* nur die Zustände einnehmen, die bei der Deklaration angegeben wurden.

```
enum eJahreszeit { Fruehl, Sommer, Herbst, Winter } a, b, c[8];

a = Sommer;                          /* Zuweisung einer Konstanten */
if(a==Sommer) printf("Sommer!\n");       /* Verwendung von a */
b = a;                        /* Zuweisung des Inhalts einer Variable */
c[3] = Herbst;                        /* Zuweisung in einen Vektor */
```

Im Gegensatz zu C++ ist es bei C noch möglich, einer Variable vom Typ *enum* neben den bei der Deklaration aufgeführten Werten einen beliebigen Zahlenwert vom Typ *int* zuzuweisen. Somit könnte die Variable gleichzeitig als normale *int*-Variable verwendet werden. Diese Vermischung der Datentypen ist bei C++ nicht mehr möglich und soll von daher hier nicht weiter vertieft werden.

3.3 Mehrfach genutzter Speicher mit `union`

Mit dem Schlüsselwort *union* ist es möglich, mehrere Variablen auf die selbe Stelle im Speicher zu legen. Die Elemente der *union*-Struktur haben üblicherweise unterschiedliche Datentypen, sie können aber auch den selben Datentyp haben. Wird der Inhalt einer der Elemente geändert, so werden die anderen Elemente, die den selben Speicherbereich nutzen, gleichzeitig verändert.

Der Vorteil liegt darin, dass die selbe Stelle im Speicher auf unterschiedliche Weise betrachtet werden kann. Zum Beispiel kann eine *double*-Variable gleichzeitig als ein *char*-Vektor mit acht Elementen betrachtet werden. Eine andere typische Anwendung sind Variablen, in die mehrere Ja/Nein-Informationen gespeichert werden, die einerseits einzeln angesprochen werden sollen, dann aber auch als *int*-Variable zusammen verfügbar sein sollen.

3.3.1 Beispiel für eine `union`-Struktur

Das folgende Beispiel verwendet für eine *double*-Variable und einen *unsigned char*-Vektor mit acht Elementen den selben Speicher. Das Progamm speichert in diese Variable den Wert $\pi = 3{,}141...$ und gibt die acht Bytes des Typs *double* Hexadezimal auf dem Bildschirm aus.

```
#include <stdio.h>

int main()
{
    union uZahl {
        double Wert;
        unsigned char Byte[8];
    };
    union uZahl Zahl;
    int i;
```

```
    Zahl.Wert = 3.141592653589793;
    printf("Speicherbelegung des Wertes %.15f:\n", Zahl.Wert);
    for(i=0; i<8; i++)
        printf("Byte %d: %02x\n", i, (int)Zahl.Byte[i]);

    return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Speicherbelegung des Wertes 3.141592653589793:
Byte 0: 18
Byte 1: 2d
Byte 2: 44
Byte 3: 54
Byte 4: fb
Byte 5: 21
Byte 6: 09
Byte 7: 40
```

3.3.2 Deklaration einer `union`-Struktur

Die Deklaration einer *union*-Struktur erfolgt in gleicher Weise wie eine Struktur mit *struct*. Nach dem Schlüsselwort *union* folgt der frei wählbare Bezeichner (hier *uZahl*). Danach folgt in geschweiften Klammern die Liste der Elemente, die zu der Struktur gehören. Es gelten die gleichen Regeln wie bei einer Struktur, die mit dem Schlüsselwort *struct* deklariert wurde, nur dass bei *union* für alle aufgeführten Elemente der selbe Speicher verwentet wird, siehe Abbildung 3.3. Die Größe der *union*-Struktur ergibt sich durch das größte enthaltene Element.

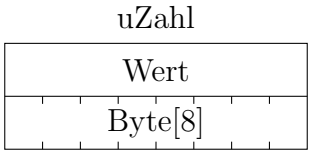


Abbildung 3.3: Anordnung der `union`-Struktur `uZahl` im Speicher.

3.3.3 Definition einer `union`-Struktur

Die Definition erfolgt nach dem gleichen Muster wie bei einer Struktur, die mit *struct* deklariert wurde. Nach dem Schlüsselwort *union* folgt der zuvor vergebene Name (in unserem Beispiel *uZahl*) und ein frei wählbarer Variablenname (im Beispiel *Zahl*).

Deklaration und Definition kann in einem Schritt erfolgen:

```
union uZahl {
    double Wert;
    unsigned char Byte[8];
} Zahl;
```

Wird eine *union*-Struktur nur an einer Stelle verwendet, so kann der Name des Typs weggelassen werden:

```
union {
    double Wert;
    unsigned char Byte[8];
} Zahl;
```

3.3.4 Arbeiten mit `union`-Strukturen

Mit einer *union*-Struktur wird genauso gearbeitet wie mit einer Struktur, die mit *struct* definiert wurde. Hinter dem Namen, der bei der Definition festegelegt wurde, wird mit einem Punkt das gewünschte Element angesprochen, siehe Beispiel.

Wird eine *union*-Struktur als Ganzes kopiert, so werden alle in ihr enthaltenen Elemente kopiert. Auch ein Vektor innerhalb der *union*-Struktur wird mit allen Elementen kopiert.

Der Operator '–>' kann wie bei normalen Strukturen für einen Zeiger auf eine *union*-Struktur zum Ansprechen eines Elementes verwendet werden:

```
union uZahl Zahl, *pZahl=&Zahl;

pZahl->Wert = 3.141592653589793;
```

3.3.5 Verschachtelte `union`-Strukturen

Es ist möglich, *union*-Strukturen zu verschachteln. Betrachten Sie dazu folgendes Beispiel und versuchen Sie, bevor Sie weiterlesen, die Strukturen zu verstehen:

```
union uZahl {
    double Wert;
    unsigned char Byte[8];
};

struct sZahlen {
    union uZahl x;
    union uZahl y;
    union uZahl z;
};

union uZahlen {
    struct sZahlen Zahlen;
    unsigned char Byte[24];
};
union uZahlen a, b;
```

Die *union*-Struktur mit dem Namen *uZahl* ist uns bereits aus dem vorherigen Beispiel bekannt. Sie sogt dafür, dass die Elemente *Wert* und *Byte* auf die selbe Stelle im Speicher zugreifen. Da beide Elemente für sich acht Byte Speicherplatz benötigen, hat auch die *union*-Struktur eine Größe von acht Byte.

Die Struktur *sZahlen* bündelt drei *uZahl*-Strukturen zusammen, wobei jedes Element separaten Speicher zugewiesen bekommt. Die Größe dieser Struktur ergibt sich aus der Summe der Größe der Elemente, hier also 24 Byte.

Schließlich, die *union*-Struktur *uZahlen* vergibt für die beiden Elemente *Zahlen* und *Byte* einen gemeinsamen Speicherbereich von 24 Byte.

Bei der Deklaration wird dem C-Compiler nur die Struktur bekannt gegeben, und es wird noch kein Speicher reserviert. Erst bei der Definition in der letzten Zeile des Beispiels wird für die Variablen *a* und *b* jeweils 24 Byte Speicher reserviert.

Auch ein *enum*-Datentyp kann als Element einer *union*-Struktur verwendet werden. Da sich hinter einem *enum* der Typ *int* verbirgt, werden für *enum* die gleiche Anzahl an Bytes benötigt wie für den Typ *int*.

3.4 Eigene Datentypen mit `typedef`

Mit *typedef* können neue Datentypen erzeugt werden. Zunächst ein Beispiel:

```
typedef unsigned char tByte;
tByte Byte;

Byte = 192;
```

Hier wurde der neue Datentyp *tByte* kreiert. Hinter *tByte* verbirgt sich der Datentyp *unsigned char*. Anstatt den relativ langen Bezeichner *unsigned char* zu verwenden, kann jetzt mit dem neuen Typ *tByte* gearbeitet werden.

Neue Datentypen können an verschiedenen Stellen verwendet werden. Zunächst kann der Quellcode mit neuen Datentypen übersichtlicher gestaltet werden. Durch eine gute Namensvergabe kann schon am Datentyp ersehen werden, wofür eine Variable gedacht ist. Beispiel hierfür ist der Typ *size_t*, der unter anderem in der Header-Datei <stdio.h> deklariert ist, hinter dem sich der Typ *unsigned int* verbirgt. Dieser Typ wird z.B. von der Funktion *sizeof()* zurückgegeben.

```
typedef unsigned int size_t;
```

Des Weiteren wird eine Typendefinition für neue Datenstrukturen verwendet. Beispiel: Beim Arbeiten mit Dateien wird der Datentyp *FILE* verwendet, der mit *typedef* erstellt worden ist. Die folgende Deklaration findet sich so oder ähnlich in der Header-Datei <stdio.h>:

```
struct _iobuf {
    char *_ptr;
    int   _cnt;
    char *_base;
    int   _flag;
    int   _file;
    int   _charbuf;
    int   _bufsiz;
    char *_tmpfname;
};
typedef struct _iobuf FILE;
```

Schließlich hilft eine Typendefinition beim Schreiben von kompatiblen Programmen. Wenn Sie ein Programm schreiben, das auf möglichst vielen Plattformen laufen soll, so ist Ihnen nicht bekannt, wie groß z.B. eine *int*-Variable ist. Unter Visual C hat *int* einen Wertebereich von -2^{31} bis $2^{31}-1$ (4 Byte) während ein Z80-Compiler nur zwei Bytes mit einem Wertebereich von -2^{15} bis $2^{15}-1$ spendiert. Wie lösen Sie dieses Problem? Sie erstellen sich für jede Plattform eine eigene Header-Datei, in der Sie elementare Datentypen deklarieren. Danach arbeiten Sie dann nur noch mit den

neu kreierten Datentypen. Die Header-Datei für Visual C könnte folgendermaßen aussehen:

```
/* Header-Datei für Visual C */
typedef short Int2;
typedef int Int4;
```

Die Header-Datei für einen Z80-Compiler könnte dagegen wie folgt aussehen:

```
/* Header-Datei für Z80-Compiler */
typedef int Int2;
typedef long Int4;
```

Ihr Programm schließlich verwendet dann die neuen Datentypen *Int2* und *Int4*:

```
/* Hauptprogramm */
#include "DatenTypen.h"

int main()
{
    Int2 Klein=12345;
    Int4 Gross=1234567890;
}
```

3.5 Abschließende Bemerkungen

Bei Variablen haben Sie gelernt, dass globale Variablen vermieden werden sollen. Bei der Deklaration von speziellen Datentypen ist das nicht so kritisch, da bei der Deklaration ja noch kein Speicher vergeben wird. (Das erfolgt erst bei der Definition.) Aus Gründen der Übersicht sollten Sie aber dennoch versuchen, auch die Deklarationen nur in den Bereichen bekannt zu machen, in denen sie auch verwendet werden.

Zur besseren Lesbarkeit Ihrer Programme sollten Sie bei der Verwendung von *struct*, *enum*, *union* oder *typedef* jeweils entsprechend den Buchstaben *s*, *e*, *u* oder *t* anfügen. Sie werden keine Vorschrift finden, die genau regelt, wie diese Kennung erfolgen soll (vorne, hinten, Groß-, oder Kleinbuchstabe, durch Tiefstrich getrennt etc.), aber innerhalb eines Programms oder Projektes sollten Regeln definiert und eingehalten werden. Für größere Projekte werden zu diesem Zweck Regeln für den Programmierstil im sogenannten *Coding Styles* festgelegt.

3.6 Aufgaben

Aufgabe 3.1: Erstellen Sie eine Struktur mit Namen *sDatum* und den drei *int*-Elementen *Tag*, *Monat* und *Jahr*.

Aufgabe 3.2: Definieren Sie von der soeben deklarierten Struktur eine Variable und weisen Sie ihr gleich das Datum 12.9.2007 zu.

Aufgabe 3.3: Weisen Sie der soeben definierten Variable den Monat Mai zu.

Aufgabe 3.4: Definieren Sie einen Zeiger auf die deklarierte Struktur *sDatum*, und lassen Sie diesen Zeiger auf die vorher definierte Variable zeigen.

Aufgabe 3.5: Verwenden Sie den soeben definierten Zeiger, um den Tag auf den 1. zu setzen.

Aufgabe 3.6: Ergänzen Sie die in Abschnitt 3.1.5 aufgeführten Strukturen einer Schulklasse und erstellen Sie eine Struktur für eine ganze Schule mit maximal 20 Klassen.

Aufgabe 3.7: Erstellen Sie den Aufzählungstyp *Monat* für die zwölf Monate, welche die Werte eins bis zwölf haben.

Aufgabe 3.8: Deklarieren Sie einen Aufzählungstyp für die Farben *rot*, *grün* und *blau*.

Aufgabe 3.9: Erweitern Sie den soeben erstellen Aufzählungstyp um die englischen Namen *red*, *green* und *blue*. Sorgen Sie dafür, dass gleiche Farben auch gleiche Werte haben.

Aufgabe 3.10: Erstellen Sie eine Struktur für drei *double*-Variablen, die den selben Speicher belegen.

Aufgabe 3.11: Definieren Sie eine Struktur, bei der eine *int*-Variable gleichzeitig als vierfacher *char* betrachtet werden kann.

Aufgabe 3.12: Erstellen Sie eine *union*-Struktur, in der sich ein *double*, zwei *int*, vier *short* und acht *char* den selben Speicher teilen.

Aufgabe 3.13: Kreieren Sie sich einen neuen Datentyp mit Namen *DWORD*, der aus einem *unsigned* besteht.

Aufgabe 3.14: Erstellen Sie einen neuen Datentyp mit Namen *tDatum* mit den Elementen *Tag*, *Monat* und *Jahr*.

Kapitel 4

Dynamische Speicherverwaltung

Häufig ist es zum Zeitpunkt der Programmerstellung nicht bekannt, wie viele Daten zur Laufzeit im Arbeitsspeicher gespeichert werden sollen. Stellen Sie sich ein elektronisches Telefonbuch vor, in das regelmäßig Einträge eingefügt oder gelöscht werden. Eine unelegante Möglichkeit ist, eine große Menge an Elementen vorzusehen. Wenn Sie für das Telefonbuch z.B. 1.000.000 Einträge zulassen, so sollte dass für den Privatbereich ausreichend sein. Nachteil ist aber, dass Sie unnötig viel Speicher verbrauchen. Bei nur 1kB pro Adresse wären das bei dem genannten Beispiel 1GB! Und was ist, wenn sich eine Versicherungsgesellschaft mit einem Kundenstamm von über eine Million für Ihr Programm interessiert...?

Die Antwort liegt in der dynamischen Speicherverwaltung. Es wird nicht Speicher für eine feste Menge an Daten reserviert, sondern die Größe des verwendeten Speichers wird dynamisch angepasst.

Die nächsten drei Abschnitte beschäftigen sich mit der Arbeitsweise und den vorhandenen Funktionen für die dynamische Speicherverwaltung.

4.1 Speicherreservierung mit malloc()

Wir beginnen erste Versuche mit einem einfachen Beispiel, in dem Speicher für sechs *int*-Variablen zum Speichern eines Notenspiegels für die Noten 1 bis 6 reserviert, verwendet und wieder freigegeben wird:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *Noten=NULL; /* Zeiger für reservierten Speicher */
7      int i;           /* lokale Laufvariable */
8
9      /* 1. Schritt: Speicher reservieren */
10     Noten = (int*)malloc(6*sizeof(int));
11     if(Noten==NULL) printf("Nicht genug Speicher vorhanden\n");
12
13     /* 2. Schritt: Speicher verwenden */
14     if(Noten) {
15         for(i=0; i<6; i++) Noten[i] = 0;
16         /* ... */
17     }
18
```

```
19      /* 3. Schritt: Speicher freigeben */
20      if(Noten) free(Noten);
21
22      /* Schlussmeldung */
23      if(Noten) printf("Programm erfolgreich beendet.\n");
24
25      return 0;
26  }
```

4.1.1 Zeiger auf den zu reservierenden Speicher

Für die dynamische Verwendung von Speicher benötigen wir zunächst einen Zeiger auf den Datentyp, mit dem gearbeitet werden soll. Im Beispiel ist dies in Zeile 6 ein Zeiger auf *int*.

Dieser Zeiger wird für die folgenden drei Schritte benötigt: 1. Speicher reservieren, 2. Speicher verwenden und 3. Speicher wieder freigeben.

4.1.2 Erster Schritt: Speicher reservieren

Mit der Funktion *malloc()* wird in Zeile 10 der benötigte Speicher angefordert. Die Funktion erwartet als Parameter die Größe des geforderten Speichers in Byte. Als Ergebnis liefert die Funktion die Adresse im Speicher, an der Platz reserviert wurde.

Da die Funktion *malloc()* nicht wissen kann, für welchen Datentyp der Speicher gedacht ist, wird zunächst ein Zeiger auf den Datentyp *void* zurückgegeben. In unserem Beispiel wandeln wir ihn mit *(int*)* in einen Zeiger auf *int* um. Dies Umwandlung ist nicht zwingend notwendig, sollte aber immer erfolgen, um evtl. Fehlern vorzubeugen. So kann der Compiler prüfen, ob die Umwandlung mit dem Typ des Zeigers übereinstimmt.

Der Speicher wird nur reserviert aber nicht initialisiert. Das heißt, in dem reservierten Speicher steht noch irgend etwas undefiniertes, was in dem Programm überschrieben werden muss. (Im nächsten Beispielprogramm lernen wir die Funktion *calloc()* kennen, die den Speicher mit null initialisiert.)

Es kann passieren, dass nicht genug Speicher zur Verfügung steht und die Funktion den Wunsch nach Speicher nicht erfüllen kann. In diesem Fall wird eine *NULL* (Zeiger auf die Adresse null) zurückgegeben. Damit ein Programm sicher läuft, muss nach einer Speicherreservierung zunächst geprüft werden, ob der zurückgegebene Zeiger nicht auf null zeigt. Ist das doch der Fall, muss eine Fehlerbehandlung erfolgen. In dem Beispiel wird dafür in Zeile 11 eine Fehlermeldung ausgegeben.

4.1.3 Zweiter Schritt: Speicher verwenden

War die Speicherreservierung erfolgreich, so kann mit dem Speicher gearbeitet werden. Der Zugriff auf den Speicher erfolgt dabei mit dem Zeiger, in dem die Adresse des reservierten Speichers gespeichert wurde.

Durch die Verwandtschaft zwischen Zeigern und Vektoren können in dem Beispiel in Zeile 15 die sechs *int*-Variablen über einen Index angesprochen werden.

Für die Einhaltung der Grenzen des reservierten Speichers sind Sie als Programmierer verantwortlich. In dem Beispiel bedeutet dies, dass der Indexbereich auf null bis fünf beschränkt ist, was durch die *for*-Schliefe in Zeile 15 sichergestellt ist. Ein

überschreiten führt zu unvorhersehbaren Effekten, die leider häufig erst an ganz anderen Stellen sichtbar werden. Hier ist gründliches Arbeiten geboten.

4.1.4 Dritte Schritt: Speicher freigeben

Nachdem die Verwendung des Speichers abgeschlossen wurde, muss der Speicher wieder freigegeben werden. Das geschieht mit der Funktion *free()*, die als einzigen Parameter einen Zeiger auf den zuvor reservierten Speicher erwartet, siehe Zeile 20.

Bei größeren Programmen, die viel mit dynamischen Speicher arbeiten, wäre ein Weglassen der Funktion *free()* fatal. Der verfügbare Speicher würde immer weiter abnehmen, bis das ganze System instabil wird. Dieses Verhalten wird als *Speicherleckage* (engl. *memory leakage*) bezeichnet.

4.1.5 Umgang mit Fehlern

Ähnlich wie beim Verarbeiten von Dateien kann es auch hier beim Reservieren von Speicher zu Fehlern kommen: Es wird zu viel Speicher angefordert und die Funktion *malloc()* gibt den Wert *NULL* zurück.

In dem gezeigten Beispiel wurde dazu eine Konvention definiert: Hat der Zeiger für den dynamisch reservierten Speicher einen Wert ungleich *NULL*, so wurde Speicher reserviert. Hat er dagegen den Wert *NULL*, so wurde kein Speicher reserviert, und der Zeiger kann nicht für die weitere Verarbeitung verwendet werden.

Zu diesem Zweck wird in Zeile 6 der Zeiger bei der Definition mit dem Wert *NULL* initialisiert. Wurde der Speicher erfolgreich reserviert, erhält der Zeiger einen Wert ungleich *NULL*. Gab es dagegen ein Problem beim Reservieren des Speichers, erhält der Zeiger den Wert *NULL*.

Vor der Verwendung des Speichers wird in Zeile 14 zunächst geprüft, ob der Zeiger einen Wert ungleich *NULL* hat, und nur dann wird mit dem Speicher gearbeitet. Auch bei der Freigabe des Speichers wird in Zeile 20 zunächst geprüft, ob der Zeiger ungleich *NULL* ist, und nur dann der Speicher wieder freigegeben. Auch die Schlussmeldung in Zeile 23 erfolgt nur, wenn der Zeiger ungleich *NULL* ist.

4.2 Speicherreservierung mit calloc()

Die Funktion *malloc()* im vorherigen Beispiel hat Speicher reserviert ohne ihn zu initialisieren. Im folgenden Beispiel führen wir die Funktion *calloc()* ein, die ebenfalls Speicher reserviert, ihn aber dann mit null initialisiert.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *Note=NULL; /* Zeiger für reservierten Speicher */
7      int Anzahl=6;   /* Anzahl unterschiedlicher Noten */
8      int i;          /* lokale Laufvariable */
9
10     /* 1. Schritt: Speicher reservieren */
11     Note = (int*)calloc(Anzahl, sizeof(int));
12     if(Note==NULL) printf("Nicht genug Speicher vorhanden\n");
13
```

```
14      /* 2. Schritt: Speicher verwenden */
15      if(Note) {
16          for(i=0; i<Anzahl; i++)
17          printf("%2d mal Note %d\n", Note[i], i+1);
18          /* ... */
19      }
20
21      /* 3.. Schritt: Speicher freigeben */
22      if(Note) free(Note);
23
24      /* Schlussmeldung */
25      if(Note) printf("Programm erfolgreich beendet.\n");
26
27      return 0;
28  }
```

4.2.1 Speicher mittels calloc() reservieren

Die Funktion *calloc()* bietet sich an, wenn mit Zahlen gearbeitet wird, die mit nullen initialisiert werden sollen. In diesem Beispiel soll wieder ein Notenspiegel erstellt werden, bei dem die erzielten Noten einer Schulklasse gezählt werden. Da das Zählen immer mit null beginnt, ist es hilfreich, wenn die sechs Zähler für die Noten mit null initialisiert werden.

In Zeile 11 wird als erstes Argument der Funktion *calloc()* die Anzahl der Elemente übergeben, für die Speicher reserviert werden soll. Das zweite Argument gibt die Größe eines Elements an. Ohne zu wissen, was mit dem Speicher gemacht werden soll, initialisiert *calloc()* den reservierten Speicher mit null. Für alle Ganzzahl- und Gleitkommatypen hat das zur Folge, dass die Elemente des jeweiligen Typs auch den Wert null haben. Für Zeichenketten ergibt sich die Länge null, da das erste Zeichen auch auf null gesetzt wird; aber der Speicher ist trotzdem für die angegebene Länge reserviert.

4.2.2 Größe des Speichers durch eine Variable

Im Gegensatz zu dem ersten Beispiel wurde hier in Zeile 7 die Anzahl der zu reservierenden Variablen in die Variable *Anzahl* gespeichert. Diese Variable wird sowohl bei der Definition des Notenvektors als auch in der *for*-Schleife für die Ausgabe verwendet. Der Vorteil ist, dass eine evtl. Änderung der Anzahl der Noten nur an einer Zeile erfolgen muss.

Später kann ein einem Programm zunächst die Anzahl der Noten vom Benutzer abgefragt werden, und erst dann erfolgt die Reservierung des Speichers und das Verarbeiten der Daten. Dadurch wird das Wesen der dynamischen Speicherreservierung deutlich: Erst zur Laufzeit wird klar, wie viel Speicher benötigt wird und entsprechend Platz reserviert.

4.3 Speicherreservierung mit realloc()

Wenn Sie die Größe eines reservierten Speicherblocks ändern wollen kommt die Funktion *realloc()* zum Tragen. Dafür wieder ein Beispiel:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      char *text=NULL; /* Zeiger für reservierten Speicher */
8      char *tmp;       /* Zeiger zur Erweiterung des Speichers */
9
10     /* Speicher für 9 Buchstaben reservieren */
11     text = (char*)malloc(10);
12     if(text==NULL) printf("Nicht genug Speicher vorhanden\n");
13
14     /* reservierten Speicher verwenden */
15     if(text) {
16         strcpy(text, "Guten Tag");
17         printf("1. Text: %s\n", text);
18     }
19
20     /* Speicher für 33 Buchstaben reservieren */
21     if(text) {
22         tmp = (char*)realloc(text, 34);
23         if(tmp==NULL) {
24             free(text);
25             printf("Nicht genug Speicher vorhanden\n");
26         }
27         text = tmp;
28     }
29
30     /* reservierten Speicher verwenden */
31     if(text) {
32         strcat(text, " meine Damen und Herren!");
33         printf("2. Text: %s\n", text);
34     }
35
36     /* Speicher freigeben */
37     if(text) free(text);
38
39     return 0;
40 }
```

Das Programm erzeugt folgende Ausgabe:

```
1. Text: Guten Tag
2. Text: Guten Tag meine Damen und Herren!
```

4.3.1 Geänderte Speichergröße mittels realloc()

Wenn von *dynamischer* Speicherverwaltung gesprochen wird, meint das nicht nur, dass Speicher reserviert und freigegeben wird, sondern auch, dass die Größe eines Speicherblocks hin und wieder angepasst werden muss.

In Zeile 22 erwartet die Funktion *realloc()* einen Zeiger auf einen zuvor reservierten Speicherblock und die gewünschte neue Größe. Als Ergebnis wird ein Zeiger auf den neuen Speicherblock zurückgegeben.

Die Funktion versucht zunächst den Speicherblock an der aktuellen Stelle zu erweitern. Ist dies nicht möglich, so wird eine neue Stelle im Speicher gesucht. Im zweiten Fall wird der alte Speicherinhalt an die neue Stelle kopiert bis der kleinere Wert der alten und neuen Speichergröße erreicht ist.

Tritt ein Problem auf, so wird der vorherige Speicherblock nicht freigegeben und kann noch weiter verwendet werden. Das heißt aber auch, dass der Speicherblock in diesem Fall auch freigegeben werden muss. Im Beispiel erfolgt das bei der Fehlerbehandlung nach *realloc()* in Zeile 24.

4.3.2 Umgang mit Fehlern

Da im Fehlerfall der vorige Speicher nicht freigegeben wird, wird bei Anwendung von *realloc()* ein zweiter Zeiger benötigt. Nur dadurch kann eine Speicherleckage (memory leakage) vermieden werden.

In C erhält der Programmierer anders als bei anderen Sprachen mehr Verantwortung zur Vermeidung von Fehlern zur Laufzeit. Das ist zunächst etwas lästig, hat aber den Vorteil, dass die Programme zur Laufzeit weniger Rechenzeit benötigen.

4.4 Aufgaben

Aufgabe 4.1: Worin unterscheiden sich *malloc()* und *calloc()*?

Aufgabe 4.2: Reservieren Sie für 16 *int*-Variablen Speicher. Verwenden Sie dafür die Variable *Note*, die ein Zeiger auf *int* ist.

Aufgabe 4.3: Weisen Sie dem Zeiger auf *double* mit Namen *pDouble* Speicher für 100 Temperaturen zu und initialisieren Sie diese mit null.

Aufgabe 4.4: Was fehlt bei der folgenden Programmzeile?

```
char *Text = malloc(100*sizeof(char));
```

Aufgabe 4.5: Wozu dient die Funktion *free()*?

Aufgabe 4.6: Wie melden die Funktionen *malloc()*, *calloc()* und *realloc()* einen Fehler zurück?

Aufgabe 4.7: Was wird mit *Speicherleckage* (*memory leakage*) bezeichnet?

Aufgabe 4.8: Was darf nach der Verwendung von dynamischem Speicher nicht vergessen werden?

Kapitel 5

Rekursion

5.1 Einleitung

Wir beginnen das Thema mit einer Übung. Führen Sie die folgenden Folgen fort:

a) 3 - 4 - 5 - 6 - ...

b) 1 - 1 - 2 - 3 - 5 - 8 - ...

c) 1 - 2 - 4 - 8 - 16 ...

d) 18.1. - 25.1. - 1.2. - 8.2. - ...

e) 2 - 11 - 3 - 12 - 4 - 13 - ...

f) 1 - 2 - 6 - 24 - 120 - ...

(Die Lösungen finden Sie in der Fußnote[1].) Nein, es soll hier kein IQ-Test durchgeführt werden, sondern wir wollen untersuchen, wie Sie eine solche Lösung angehen.

Die erste Aufgabe ist einfach: Die Zahlen erhöhen sich bei jedem Schritt um eins. Mit z_i als die aktuell zu findende Zahl können wir diese Erkenntnis mathematisch mit $z_i = z_{i-1} + 1$ ausdrücken. Hinzu kommt eine Startbedingung, die wir mit $z_1 = 3$ bezeichnen.

Die zweite Aufgabe sind die Fibonacci-Zahlen: Eine Folgezahl ergibt sich aus der Summe der beiden vorherigen Zahlen, wobei die ersten zwei Zahlen den Wert 1 haben. Die Fibonacci-Zahlen können mit $z_i = z_{i-1} + z_{i-2}$ und $z_1 = z_2 = 1$ ausgedrückt werden.

Die mathematische Beschreibung der sechs Aufgaben ist in Tabelle 5.1 zusammengefasst.

Im Folgenden betrachten wir die Berechnung der Fakultät (Aufgabe f) etwas näher. Mit den in Tabelle 5.1 angegebenen Gleichungen kann die Fakultät für eine beliebige positive Zahl schrittweise berechnet werden. Abbildung 5.1 illustriert die nötigen fünf Schritte zur Berechnung der Fakultät von 5. Dieser Mechanismus lässt sich in mit einem rekursiven Programm realisieren. Die folgende kleine Funktion übernimmt die Berechnung der Fakultät.

[1] A) 7 8; B) 13 21 (Fibonacci); C) 32 64; D) 15.2. 22.2.; E) 5 14; F) 720 5040

Nr.	inkrementelle Gleichung	Startbedingung	Bezeichnung
a)	$z_i = z_{i-1} + 1$	$z_1 = 3$	Einer-Reihe
b)	$z_i = z_{i-1} + z_{i-2}$	$z_1 = z_2 = 1$	Fibonacci-Zahlen
c)	$z_i = 2z_{i-1}$	$z_1 = 1$	Zweier-Potenz
d)	$z_i = z_{i-1} + 7$ Tage	$z_1 = 18.1.$	Daten im Wochentakt
e)	$z_i = z_{i-2} + 1$	$z_1 = 2$ und $z_2 = 11$	doppelte Einer-Reihe
f)	$z_i = i \cdot z_{i-1}$	$z_1 = 1$	Fakultät

Tabelle 5.1: Die Zahlenfolgen der sechs Aufgaben mathematisch ausgedrückt.

$$
\begin{array}{lcl}
5! & = & 5 \cdot 4! \\
 & & \Updownarrow \\
 & & 4! \quad = \quad 4 \cdot 3! \\
 & & \qquad\qquad \Updownarrow \\
 & & \qquad\qquad 3! \quad = \quad 3 \cdot 2! \\
 & & \qquad\qquad\qquad\qquad \Updownarrow \\
 & & \qquad\qquad\qquad\qquad 2! \quad = \quad 2 \cdot 1! \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \Updownarrow \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad 1! \quad = \quad 1
\end{array}
$$

Abbildung 5.1: Schrittweise Berechnung von $n!$.

```
1 double Fakultaet(unsigned n)
2 {
3     if(n<=1) return 1;
4     return n*Fakultaet(n-1);
5 }
```

In Zeile 3 ist die Startbedingung $1! = 1$ (und $0! = 1$) implementiert. In Zeile 4 findet sich die inkrementelle Gleichung der Fakultät $n! = n \cdot (n-1)!$. Wird nun die Funktion mit einem Wert von 5 aufgerufen, so führt diese die Multiplikation von 5 und der Fakultät von 4 aus.

Für die Berechnung der Fakultät von 4 ruft die zuerst aufgerufene Funktion sich selbst mit dem Wert von 4 auf. Damit dieser Mechanismus funktioniert, legt C im Speicher wieder eine Kopie des Argumentes an. Dadurch kann die Funktion mehrfach zur gleichen Zeit aktiv sein, ohne dass sich die Parameter n ins Gehege kommen.

5.2 Erstellung eines rekursiven Programms

Viele Programmieraufgaben lassen sich rekursiv lösen. Immer wenn es gelingt, eine größere Aufgabe in mehrere gleichartige, kleinere Aufgaben zu zerlegen, kann ein rekursiver Ansatz sinnvoll sein. Bei einem rekursiven Ansatz müssen drei Fragen geklärt werden:

Festlegung der inkrementellen Aufgabe. Die Inkrementelle Aufgabe ist das, was die Funktion selbst erledigt. Ohne eine inkrementelle Aufgabe könnte die Rekursion noch so tief verschachtelt sein – es würde nichts passieren.

Tiefe der Verschachtlung begrenzen. Damit die Rekursion ein Ende finden kann, muss dafür gesorgt werden, dass die Verschachtlung nicht unbegrenzt tief erfolgen kann.

Reihenfolge von rekursivem Aufruf und inkrementeller Aufgabe. Die Anordnung der inkrementellen Aufgabe relativ zum rekursivem Aufruf, bzw. zu den rekursiven Aufrufen hat einen wesentlichen Einfluss auf das Verhalten der Funktion. Es wird von *Traversierung* gesprochen.

Im Abschnitt 5.3 werden für einige Beispiele diese drei Aspekte betrachtet. Ein kleiner Fehler in einer rekursiven Funktion kann verheerende Folgen haben. Auf der anderen Seite kann es sein, dass bei völlig unerwarteten, fehlerhaften Ergebnissen nur ein kleiner Fehler die Ursache ist.

5.2.1 Inkrementelle Aufgabe

Die inkrementelle Aufgabe ist das, was in einer rekursiven Funktion direkt erledigt wird. Bei der Fakultät ist das die Multiplikation mit einer ganzen Zahl. Bei den Zahlen von Fibonacci ist es die Addition von zwei Zahlen. In Abschnitt 5.3 sind einige Beispiele Aufgelistet. Durch eine geschickte Festlegung der inkrementellen Aufgabe kann die rekursive Funktion z.T. sehr klein ausfallen.

5.2.2 Tiefe der Verschachtlung begrenzen

Es muss sicher gestellt werden, dass sich eine rekursive Funktion nicht unbegrenzt oft aufruft. Bei vielen Fragestellungen ergibt sich eine Begrenzung von selbst. Bei anderen Aufgaben sollte ein Zähler als Parameter übergeben werden, der bei jeder Ebene der Rekursion um eins reduziert wird.

Eine unbegrenzte Tiefe der Verschachtlung hat zur Folge, dass ein Programm nicht zum Ende kommt, und dass der Speicher überläuft. Und natürlich erzeugt die Funktion nicht das gewünschte Ergebnis.

5.2.3 Traversierung

Bei der Erstellung einer rekursiven Funktion muss festgelegt werden, wann die inkrementelle Aufgabe relativ zu den rekursiven Aufrufen erfolgen soll. In den meisten Fällen hat die Reihenfolge einen maßgeblichen Einfluss auf das Ergebnis der Rekursion. Es wird von *Traversierung* gesprochen und es wird zwischen drei Arten der Traversierung unterschieden:

Preorder-Traversierung. Die Verarbeitung findet *vor* dem rekursiven Aufruf, bzw. den rekursiven Aufrufen statt.

Inorder-Traversierung. Die Verarbeitung findet *zwischen* zwei oder mehr rekursiven Aufrufen statt.

Postorder-Traversierung. Die Verarbeitung findet *nach* dem rekursiven Aufruf, bzw. den rekursiven Aufrufen statt.

Bei den Beispielen in Abschnitt 5.3 wird jeweils die Traversierung betrachtet. Um ein Gefühl für die Traversierung zu entwickeln wird empfohlen, bei den Beispielen mit der Reihenfolge zu experimentieren.

5.3 Beispiele

Bei den folgenden Beispielen wird nach den folgenden Schritten vorgegangen:

a) Ermittlung der *inkrementellen Aufgabe*.

b) Definition der *Abbruchbedingung*.

c) Festlegung ob Pre-, In- oder Postorder-*Traversierung* angewendet wird.

Mit dieser Erkenntnis wird die rekursive Funktion erstellt.

5.3.1 Fibonacci-Zahlen

Die Fibonacci Zahlen wurden schon in der Einführung zu diesem Thema erwähnt. Die i-te Zahl wird nach der Formel $z_i = z_{i-1} + z_{i-2}$ mit den Anfangswerten $z_1 = z_2 = 1$ berechnet.

a) Inkrementelle Aufgabe. Addition der beiden vorherigen Zahlen.

b) Abbruchbedingung. Die erste und zweite Zahl hat den Wert eins.

c) Traversierung. Da die Addition erst *nach* der Berechnung der beiden Summanden erfolgen kann, handelt es sich hier um *Postorder-Traversierung*.

Die Implementierung sieht wie folgt aus:

```
1 unsigned Fibonacci(unsigned Zahl)
2 {
3     if(Zahl==0) return 0;
4     if(Zahl<=2) return 1;
5     return Fibonacci(Zahl-1) + Fibonacci(Zahl-2);
6 }
```

In Zeile 3 wird der Vollständigkeit halber für z_0 der Wert null zurückgegeben. Zeile 4 legt die Abbruchbedingung fest und gibt für z_1 und z_2 den Wert eins zurück. In Zeile 5 finden sich die beiden rekursiven Aufrufe und die inkrementelle Aufgabe. Die beiden Funktionsaufrufe finden vor der Addition statt, so dass die Postorder-Traversierung sichergestellt ist.

In diesem Beispiel hat die rekursive Variante einen Nachteil: Für die Berechnung einer Fibonacci-Zahl wird die Funktion mehrfach mit dem gleichen Argument aufgerufen. (Probieren Sie es aus!) Die Anzahl der Aufrufe geht exponentiell mit der Übergebenen Zahl hoch! Von daher wäre hier eine Variante ohne Rekursion geeigneter.

5.3.2 Fakultät

Die Fakultät wurde bereits in der Einleitung vorgestellt. Sie soll aber hier nochmal systematisch behandelt werden.

a) Inkrementelle Aufgabe. Multiplikation der Übergebenen Zahl mit der Fakultät der um eins reduzierten Zahl.

b) Abbruchbedingung. Hat der Parameter den Wert null oder eins, ist das Ergebnis eins.

c) Traversierung. Die Multiplikation kann erst erfolgen, wenn die Fakultät der um eins reduzierten Zahl berechnet wurde. Daher handelt es sich um eine *Postorder-Traversierung.*

Die Implementierung sieht wie folgt aus:

```
1 double Fakultaet(unsigned n)
2 {
3     if(n<=1) return 1;
4     return n*Fakultaet(n-1);
5 }
```

Die *if*-Verzweigungen in Zeile 3 stellt die Abbruchbedingung dar. Bevor in Zeile 4 die Multiplikation mit dem Parameter n erfolgt, wird die Fakultät von $n-1$ ermittelt und damit auch der rekursive Aufruf durchgeführt. Dadurch ist die Postorder-Traversierung sichergestellt.

5.3.3 Duale Zahlen darstellen

Es soll eine ganze Zahl vom Typ *unsigned* dual auf dem Bildschirm ausgegeben werden. Für die Umwandlung wird eine Division mit 2 durchgeführt. Der Rest der Division wird für die Darstellung der letzten Ziffer verwendet. Um den Rest der Stellen darzustellen wird der abgerundete Quotient rekursiv übergeben.

a) Inkrementelle Aufgabe. Rest der Division mit 2 zur Darstellung der letzten Ziffer verwenden.

b) Abbruchbedingung. Wenn die Zahl null ist, wird der Vorgang abgebrochen.

c) Traversierung. Da bei der Verarbeitung die letzte Ziffer der übergebenen Zahl dargestellt wird, muss die Verarbeitung *nach* dem rekursiven Aufruf erfolgen. Es handelt sich daher um *Postorder-Traversierung.*

Die Implementierung sieht wie folgt aus:

```
1 void Binaer(unsigned Zahl)
2 {
3     if(Zahl) {
4         Binaer(Zahl/2);
5         printf("%c", Zahl%2?'1':'0');
6     }
7 }
```

Die *if*-Verzweigung in Zeile 3 realisiert die Abbruchbedingung. Innerhalb der Verzweigung erfolgt in Zeile 4 zunächst der rekursive Aufruf mit dem Quotienten aus der Zahl und 2. Danach findet in Zeile 5 die Darstellung der aktuellen dualen Ziffer statt.

Die Funktion hat einen Schönheitsfehler: Die Null wird nicht dargestellt. Hierfür ist eine Modifikation der aufrufenden Funktion nötig, oder der rekursiven Funktion wird mit einem extra Parameter mitgeteilt, ob sie als erstes aufgerufen wird.

5.3.4 Hexadezimale Zahlen darstellen

Auch wenn die Funktion *printf()* mit dem Platzhalter %x bereits die Option bietet, soll hier eine rekursive Funktion zur Darstellung hexadezimaler Zahlen erstellt werden. Die Funktion ist der vorherigen sehr ähnlich, nur dass hier statt mit dem Divisor 2 mit dem Divisor 16 gearbeitet wird.

a) Inkrementelle Aufgabe. Rest der Division mit 16 zur Darstellung der letzten Ziffer verwenden.

b) Abbruchbedingung. Wenn die Zahl null ist, wird der Vorgang abgebrochen.

c) Traversierung. Da bei der Verarbeitung die letzte Ziffer der übergebenen Zahl dargestellt wird, muss die Verarbeitung *nach* dem rekursiven Aufruf erfolgen. Es handelt sich um eine *Postorder-Traversierung*.

Die Implementierung sieht wie folgt aus:

```
1  void Hexadezimal(unsigned Zahl)
2  {
3      char ch;
4
5      if(Zahl) {
6          Hexadezimal(Zahl/16);
7          ch = Zahl%16;
8          if(ch<=9) ch += '0';
9          else ch += 'A'-10;
10         printf("%c", ch);
11     }
12 }
```

Die *if*-Verzweigung in Zeile 5 realisiert die Abbruchbedingung. Innerhalb der Verzweigung erfolgt in Zeile 6 zunächst der rekursive Aufruf mit dem Quotienten aus der Zahl und 16. Danach findet in Zeile 7 bis 10 die Darstellung der aktuellen hexadezimalen Ziffer statt.

5.3.5 Die Türme von Hanoi

Die Türme von Hanoi sind ein altes Knobelspiel, bei dem es darum geht, einen Turm von Scheiben von einem Stab auf den anderen Stab zu bewegen, siehe Abbildung 5.2. Dabei gelten zwei Regeln: a) Die Scheiben dürfen nur einzeln von einem Stab auf den anderen bewegt werden. b) Es darf nie eine größere Scheibe auf eine kleinere gelegt werden.

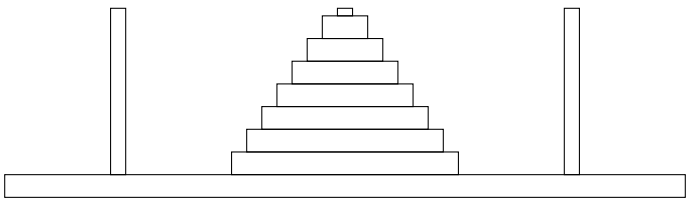


Abbildung 5.2: Die Türme von Hanoi.

Das Bewegen von n Scheiben von Stapel a nach Stapel b lässt sich rekursiv beschreiben: Erstens, bewege $n-1$ Scheiben von a nach c, zweitens, bewege Scheibe n von a nach b, drittens, bewege $n-1$ Scheiben von c nach b.

a) Inkrementelle Aufgabe. Bewegen einer Scheibe von einem Stapel zu einem anderen Stapel.

b) Abbruchbedingung. Die kleinste Scheibe wird direkt ohne rekursivem Aufruf verschoben.

c) Traversierung. Die Verschiebung der einzelnen Scheibe erfolgt zwischen zwei rekursiven Aufrufen. Es handelt sich von daher um *Inorder-Traversierung.*

Die Implementierung sieht wie folgt aus:

```
1 void Hanoi(int Anzahl, int woher, int wohin)
2 {
3     if(Anzahl>1) Hanoi(Anzahl-1, woher, 6-woher-wohin);
4     printf("Scheibe %d: Stapel %d => Stapel %d\n",
5             Anzahl, woher, wohin);
6     if(Anzahl>1) Hanoi(Anzahl-1, 6-woher-wohin, wohin);
7 }
```

Bei einem Aufruf mit *Hanoi(3, 1, 2)* erzeugt die Funktion folgende Ausgabe:

```
Scheibe 1: Stapel 1 => Stapel 2
Scheibe 2: Stapel 1 => Stapel 3
Scheibe 1: Stapel 2 => Stapel 3
Scheibe 3: Stapel 1 => Stapel 2
Scheibe 1: Stapel 3 => Stapel 1
Scheibe 2: Stapel 3 => Stapel 2
Scheibe 1: Stapel 1 => Stapel 2
```

Die Funktion erwartet als ersten Parameter die *Anzahl* der Scheiben, die verschoben werden sollen. Der zweite und dritte Parameter sind die Nummern der Stapel, *woher* und *wohin* die Scheiben verschoben werden sollen. Die Stapel werden dabei mit den Nummern 1 bis 3 bezeichnet. Die Funktion liefert als Ergebnis eine genaue Anleitung, wann welche Scheibe verschoben werden soll.

Die inkrementelle Aufgabe findet sich in den Zeilen 4 und 5 mit dem *printf*-Befehl. Die Abbruchbedingung ist mit den beiden *if*-Befehlen in Zeile 3 und 6 realisiert. Die Inorder-Traversierung lässt sich direkt aus der Reihenfolge der inkrementellen Augabe und den rekursiven Aufrufen ablesen.

Die Türme von Hanoi ist eine Anwendung, die geradezu nach Rekursion schreit. Eine Implementation ohne Rekursion wäre deutlich aufwändiger.

5.4 Rekursion auflösen

Jede rekursiv gelöste Aufgabe lässt sich auch ohne Rekursion beschreiben und implementieren. Das mag manchen Leser überraschen und wirft die Frage auf, wozu dann Rekursion überhaupt noch nötig ist. Dieser Abschnitt zeigt, wie Rekursion aufgelöst werden kann, und zeigt einige Gründe, warum Rekursion dennoch sinnvoll ist.

5.4.1 Auflösung der Rekursion: Fakultät

Unser erstes Beispiel in diesem Kapitel war eine rekursive Funktion zur Berechnung der Fakultät. Wenn Sie an dieser Stelle des Skripts angekommen sind, sollte es

Ihnen keine Schwierigkeit bereiten, die Funktion ohne Rekursion zu erstellen. Hier eine mögliche Implementation, zur Übersicht mit und ohne Rekursion:

```
double FakultaetMit(unsigned n)    /* Fakultät mit Rekursion */
{
    if(n<=1) return 1;
    return n*FakultaetMit(n-1);
}

double FakultaetOhne(unsigned n)   /* Fakultät ohne Rekursion */
{
    double Zahl=1;
    while(n>1) Zahl *= n--;
    return Zahl;
}
```

Beide Funktionen liefern exakt das gleiche Ergebnis. Die Funktion ohne Rekursion hat zwei Vorteile: sie läuft schneller und benötigt weniger Speicher. Und in der Tat, die Berechnung der Fakultät ist zwar ein schönes Beispiel, um Rekursion zu beschreiben, ist aber besser ohne Rekursion zu realisieren. Gerade wenn es um rechenintensive Programme geht, sollte auf solche Details geachtet werden. Die Auflösung der Rekursion ist hier recht einfach, da keine lokalen Variablen verwendet werden.

5.4.2 Auflösung der Rekursion: Binär-/Dual-Zahlen

Etwas umständlicher wird es bei der rekursiven Funktion zur Darstellung einer ganzen Zahl im Binär-/Dual-Format. Das Problem ist, das bei der angewandten Technik die Zahl von hinten nach vorne konvertiert wird, aber die Ausgabe von vorne nach hinten erfolgen muss. Um die Ausgabe dennoch korrekt zu erzeugen, müssen die Zahlen zwischengespeichert werden. Die beiden folgenden Implementationen zeigen die Funktion zur Umwandlung ins Binär/Dual-Format mit und ohne Rekursion:

```
void BinaerMit(unsigned Zahl)  /* Binärwandlung mit Rekursion */
{
    if(Zahl) {
        BinaerMit(Zahl/2);
        printf("%c", Zahl%2?'1':'0');
    }
}

void BinaerOhne(unsigned Zahl)  /* Binärwandlung ohne Rekursion */
{
    unsigned Temp[32];
    int i=0;

    while(Zahl) {
        Temp[i++] = Zahl;
        Zahl /= 2;
    }
    while(i) printf("%c", Temp[--i]%2?'1':'0');
}
```

Wieder liefern beide Funktionen exakt das gleiche Ergebnis. Die Variante ohne Rekursion stellt einen Vektor für alle Zahlen zur Verfügung, die sich bei der Variante mit Rekursion temporär ergeben. Es genügen 32 Elemente, da eine ganze Zahl von

vier Byte max. 32 relevante Binär-Ziffern haben kann. Bei der Variante mit Rekursion werden diese Zwischenergebnisse unsichtbar auf dem Stack abgelegt, da ja bei jedem Funktionsaufruf die Parameter der Funktion in Kopie übergeben werden.

5.4.3 Auflösung der Rekursion: Allgemeiner Ansatz

Nach diesem Prinzip kann jede Rekursion aufgelöst werden. Es müssen für alle relevanten Parameter und lokalen Variablen Vektoren zur Verfügung gestellt werden, in die alle temporären Ergebnisse abgelegt werden. Die Anzahl der Elemente der Vektoren richtet sich nach der maximalen Rekursionstiefe. Ist die maximale Rekursionstiefe beim Erstellen des Programms unbekannt, so muss dynamisch Speicher reserviert und bei bedarf erweitert werden.

5.4.4 Warum Rekursion dennoch sinnvoll ist

Wozu brauchen wir Rekursion, wenn doch alle Rekursionen aufgelöst werden können? Viele praktische Fragestellungen rufen quasi nach einer rekursiven Implementation. (Siehe das Beispiel *Türme von Hanoi.*) Mit Rekursion ist die Realisierung einfacher und übersichtlicher.

Schon bei der Darstellung einer beliebigen ganzen Zahl im Binär-Format haben wir gesehen, das die rekursive Formulierung übersichtlicher ausgefallen ist. Bei komplexeren Fragestellungen wird der Unterschied noch deutlicher.

Rekursion gehört zum Standard-Werkzeug eines Programmierers. Programme können durch Rekursion häufig übersichtlicher, kompakter und weniger fehleranfällig implementiert werden.

5.5 Aufgaben

Aufgabe 5.1: Was ist das charakteristische Merkmal einer rekursiven Funktion?

Aufgabe 5.2: Beschreiben Sie mit jeweils einem Satz die Begriffe *inkrementelle Aufgabe*, *Abbruchbedingung* und *Traversierung*.

Aufgabe 5.3: Schreiben Sie eine rekursive Funktion, die eine Zahl vom Typ *unsigned* oktal auf dem Bildschirm ausgibt.

Aufgabe 5.4: Welche Art der Traversierung liegt in der vorherigen Aufgabe vor, *Preorder*, *Inorder* oder *Postorder*?

Aufgabe 5.5: Lösen Sie bei der eben erstellten Funktion die Rekursion auf.

Aufgabe 5.6: Warum sollte die Berechnung der Fibonacci-Zahlen besser nicht rekursiv angegangen werden?

Aufgabe 5.7: Erstellen Sie eine rekursive Funktion, die alle ganzen Zahlen von a bis b aufaddiert.

Aufgabe 5.8: Erstellen Sie eine rekursive Funktion, mit der eine ganze Zahl in einem beliebigen Zahlensystem dargestellt werden kann. Es sollen Zahlensysteme mit Basis 2 bis 36 möglich sein. Als Zeichen dienen zunächst die Ziffern '0' bis '9' und dann die Buchstaben 'A' bis 'Z'.

Kapitel 6

Listen

In diesem Kapitel beschäftigen wir uns mit zwei Arten von Listen: Zum einen geht es um Listen in Form von Vektoren, die Sie in statischer Form schon im letzten Semester kennengelernt haben, die wir hier auch dynamisch anlegen wollen. Zum anderen geht es in diesem Kapitel um verkettete Listen, bei dem jedes Element einen oder mehrere Zeiger erhält, die auf Folgeelemente Zeigen.

6.1 Listen als Vektor

6.1.1 Eindimensionale Listen als Vektor

Eine einfache eindimensionale Liste kann in C als ein Vektor behandelt werden. Jeder Datentyp wie *int* oder *double*, aber auch neue vom Programmierer definierten Datentypen können zu einem Vektor verkettet werden.

Eine einfache Möglichkeit ist bei der Definition der Variablen in eckigen Klammern die Anzahl der Listenelemente anzugeben. Das folgende Beispiel zeigt eine Liste von zehn (Kalender-) Daten, die als neuer Datentyp definiert wurden:

```
typedef struct {
    int Tag;
    int Monat;
    int Jahr;
} tDat;

tDat Datum[10];
```

Eine flexiblere Möglichkeit ist die dynamische Speicherverwaltung. Das Verfahren hat den Vorteil, dass die Länge der Liste zum Zeitpunkt der Programmierung noch nicht bekannt sein muss. Dazu nochmal ein Beispiel für zehn Daten:

```
1  typedef struct {
2      int Tag;
3      int Monat;
4      int Jahr;
5  } tDat;
6
7  int Anzahl=10;
8  tDat *pDatum=NULL;
9
10 /* 1. Schritt: Speicher reservieren */
11 pDatum = (tDat*) malloc(Anzahl*sizeof(tDat));
```

```
12      if(pDatum==NULL) printf("Zu wenig Speicher vorhanden\n");
13
14      /* 2. Schritt: Eindimensionalen Vektor verwenden */
15      if(pDatum) {
16          /* ... */
17      }
18
19      /* 3. Schritt: Speicher freigeben */
20      if(pDatum) free(pDatum);
```

Die Gestaltung des Programms läuft, wie in Kapitel 4 beschreiben, in drei Schritten: Nachdem in Zeile 1 bis 5 ein neuer Typ angelegt und in Zeile 8 ein Zeiger auf den neuen Datentyp definiert wurde wird in Zeile 11 in einem ersten Schritt Speicher für den Vektor reserviert. In Zeile 15 bis 17 wird als zweiter Schritt die Verwendung des Vektors angedeutet und Schritt 3 erfolgt in Zeile 20 mit der Freigabe des Speichers.

6.1.2 Mehrdimensionale Listen als Vektor

Soll mit einer mehrdimensionalen Liste gearbeitet werden, so kann entsprechend ein mehrdimensionaler Vektor verwendet werden.

Die einfachste Implementierung ist wieder die Erstellung der Liste bei der Definition der Variable. Hinter dem Variablennamen werden für jede Dimension jeweils in eckigen Klammern die Anzahl der Elemente angegeben. Das folgende Beispiel erstellt eine zweidimensionale Liste für $10 \times 10 = 100$ (Kalender-) Daten.

```
typedef struct {
    int Tag;
    int Monat;
    int Jahr;
} tDat;

tDat Datum[10][10];
```

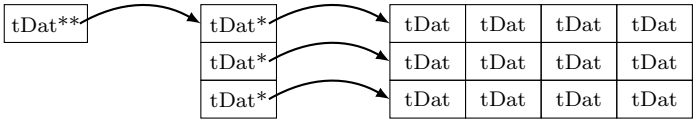
Eine mehrdimensionale Liste als Vektor dynamisch zu erzeugen ist etwas aufwändiger. Für eine zweidimensionale Liste kann z.B. wie folgt vorgegangen werden: Zunächst wird ein Zeigervektor (Vektor von Zeigern) benötigt, dessen Elemente auf die einzelnen Zeilen der Liste zeigen. Danach wird Speicher für die Daten selbst reserviert. Schließlich müssen die Zeiger initialisiert werden. Das folgende Programmstück erzeugt eine zweidimensionale Liste für $3 \times 4 = 12$ (Kalender-) Daten.

```
1       typedef struct {
2           int Tag;
3           int Monat;
4           int Jahr;
5       } tDat;
6
7       int Zeilen=3;
8       int Spalten=4;
9       tDat **Datum=NULL;
10      int i;
11      int Fehler=0;
12
13      /* Speicher für Zeigervektor */
14      Datum = (tDat**) malloc(Zeilen*sizeof(tDat*));
15      if(Datum==NULL) Fehler = -1;
16
```

```
17      /* Speicher für Daten */
18      if(!Fehler) {
19          Datum[0] = (tDat*) malloc(Zeilen*Spalten*sizeof(tDat));
20          if(Datum[0]==NULL) Fehler = -2;
21      }
22
23      /* Zeigervektor initialisieren */
24      if(!Fehler)
25          for(i=1; i<Zeilen; i++) Datum[i] = Datum[i-1]+Spalten;
```

In dem Programm wird in Zeile 9 zunächst ein Doppelzeiger auf *tDat* definiert. In Zeile 14 wird als nächstes Speicher für einen Zeigervektor und in Zeile 19 Speicher für die eigentlichen Daten reserviert. Schließlich wird in Zeile 25 der Zeigervektor initialisiert.

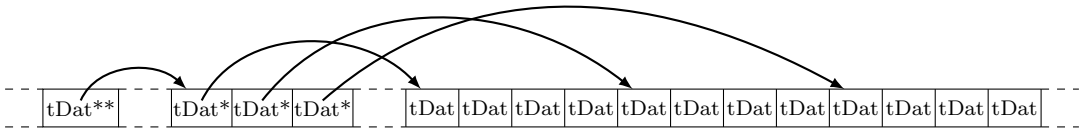
Gedanklich können wir uns den Vektor wie in der folgenden Skizze vorstellen, wobei in den Feldern der jeweilige Datentyp angegeben ist.



Der Zeiger auf Zeiger vom Typ *tDat*** ist die Variable *Datum*, mit dem später die Daten angesprochen werden. Der zweite Block mit den drei Elementen vom Typ *tDat** ist der Zeigervektor, für den mit dem ersten *malloc()*-Befehl in Zeile 14 Speicher reserviert wurde. Schließlich, in dem rechten Block, mit den 12 Elementen vom Typ *tDat*, werden die eigentlichen Daten gespeichert und es wurde in Zeile 19 mit dem zweiten *malloc()*-Befehl der benötigte Speicher reserviert.

Durch die beiden *malloc()* Anweisungen werden der Doppelzeiger und des erste Element des Zeigervektors initialisiert. In der Schleife in Zeile 25 werden die verbleibenden Elemente des Zeigervektors auf die passenden Elemente des Rechten Blocks gelenkt.

Ob die Daten Zeilenweise oder Spaltenweise angeordnet sind, kann vom Programmierer entschieden werden und bzw. ist reine Interpretation. Im Speicher sind die Daten sequentiell angeordnet, so dass sich, anders dargestellt, folgende Skizze ergibt:



Als nächstes soll das Datenfeld (der Vektor) initialisiert werden. Das folgende Programmstück setzt alle Daten auf den 1. Januar 2000. (Bei einem statischen Vektor kann die Fehlerabfrage vor den Schleifen weggelassen werden.)

```
    int z, s;  /* Index für Zeile und Spalte */

    /* Daten initialisieren */
    if(!Fehler) {
        for(z=0; z<Zeilen; z++) {
            for(s=0; s<Spalten; s++) {
                Datum[z][s].Tag = 1;
```

```
                Datum[z][s].Monat = 1;
                Datum[z][s].Jahr = 2000;
            }
        }
    }
```

Wurde der Speicher dynamisch reserviert, so muss er am Ende wieder freigegeben werden:

```
    /* Speicher freigeben */
    if(Datum) {
        if(Datum[0]) free(Datum[0]);
        free(Datum);
    }
```

Für jede weitere Dimension müssen weitere Ebenen von Zeigervektoren eingefügt werden.

6.1.3 Liste mit Blöcken unterschiedlicher Größe

Bei vielen mehrdimensionalen Listen sind die Blöcke unterschiedlich groß. So können z.B. die Zeilen einer zweidimensionalen Liste unterschiedliche Längen haben. Damit nicht unnötig Speicher verwendet wird, sollten auch die Zeilen eines zweidimensionalen Vektors entsprechend unterschiedlich lang sein.

Als Beispiel wollen wir das Dreieck von Pascal berechnen und dafür dynamisch Speicher reservieren. Beim Dreieck von Pascal wird die linke und rechte Kante des Dreiecks mit Einsen gefüllt, die anderen Zahlen ergeben sich jeweils aus der Summe der beiden darüber liegenden Zahlen.

$$
\begin{array}{ccccccc}
 & & & 1 & & & \\
 & & 1 & & 1 & & \\
 & 1 & & 2 & & 1 & \\
1 & & 3 & & 3 & & 1 \\
 & 4 & & 6 & & 4 & \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots &
\end{array}
$$

Das folgende Programm stellt Speicher für das Dreieck von Pascal mit fünf Zeilen dynamisch zur Verfügung. Für den Zeigervektor wird, wie im vorherigen Beispiel, Speicher reserviert. Die erste Zeile des Dreiecks hat ein Element, die zweite Zeile zwei Elemente, die dritte drei Elemente u.s.w. Gemäß der Gaußschen Summenformel

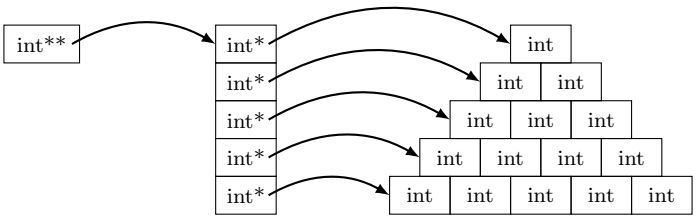
$$1 + 2 + 3 + \ldots + n = \sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

wird die Anzahl der Elemente ermittelt und Speicher dynamisch reserviert. Schließlich werden die Zeiger des Zeigervektors entsprechend initialisiert.

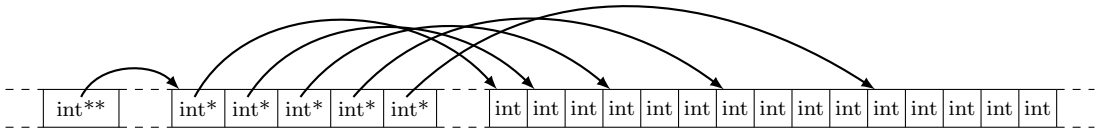
```
1    int Zeilen=5;       /* Anzal der Zeilen in Pascalschen Dreieck */
2    int z, s;           /* Index für Zeile und Spalte */
3    int **Pascal=NULL; /* Doppelzeiger für Datenstruktur */
4    int Fehler=0;       /* Fehlerindikator */
5
```

```
6     /* Speicher für Zeigervektor */
7     Pascal = (int**) malloc(Zeilen*sizeof(int*));
8     if(Pascal==NULL) Fehler = -1;
9
10    /* Speicher für Daten */
11    if(!Fehler) {
12        Pascal[0] = (int*) malloc(Zeilen*(Zeilen+1)/2*sizeof(int));
13        if(Pascal[0]==NULL) Fehler = -2;
14    }
15
16    /* Zeigervektor initialisieren */
17    if(!Fehler)
18        for(z=1; z<Zeilen; z++) Pascal[z] = Pascal[z-1]+z;
19
20    /* Pascalsches Dreieck füllen */
21    for(z=0; !Fehler && z<Zeilen; z++) {
22        Pascal[z][0] = Pascal[z][z] = 1;
23        for(s=1; s<z; s++)
24            Pascal[z][s] = Pascal[z-1][s-1]+Pascal[z-1][s];
25    }
```

Grafisch können wir uns das gemäß folgender Skizze vorstellen. Der Zeiger auf Zeiger links in der Skizze ist die Variable *Pascal* in unserem Programm. Der Block in der Mitte entspricht dem Zeigervektor, für den mit dem ersten *malloc()*-Befehl Speicher reserviert wurde. Die Pyramide rechts enthält die Daten des Pascalschen Dreiecks, für den mit dem zweiten *malloc()*-Befehl Speicher reserviert wurde.



Die Daten sind im Speicher sequentiell angeordnet, so dass sich folgende Skizze ergibt:



Die Unterschiedlichen Abstände der Zeiger im rechten Datenfeld wurde in Zeile 18 durch die variable Addition von z.

Nachdem mit dem Vektor gearbeitet wurde, muss zum Schluss wieder der Speicher freigegeben werden:

```
    /* Speicher freigeben */
    if(Pascal) {
        if(Pascal[0]) free(Pascal[0]);
        free(Pascal);
    }
```

6.1.4 Stärken und Schwächen von Listen als Vektoren

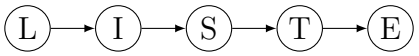
Der Vorteil einer Liste als Vektor ist, dass die Elemente zu jeder Zeit direkt angesprochen werden können. Dieser Vorteil wird im nächsten Abschnitt durch die verketteten Listen deutlich, die diesen Vorteil nicht haben.

Wird bei einer Vektor-Liste ein Element eingefügt oder entfernt, so müssen alle folgenden Elemente verschoben werden. Bei großen Listen kann das u.U. recht zeitaufwändig werden. Dieser Nachteil wird durch die verketteten Listen behoben.

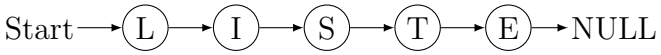
6.2 Verkettete Listen

6.2.1 Prinzip von verketteten Listen

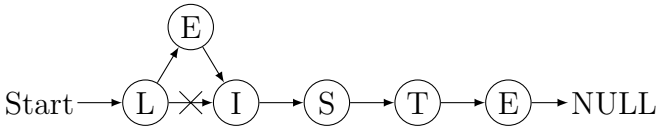
Eine Liste von Elementen kann als eine Kette dargestellt werden. Jedes Element steht nur mit seinem unmittelbaren Nachbarn in Verbindung. In der folgenden Abbildung ist das Wort *Liste* als eine Kette dargestellt.

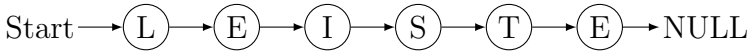


Die Enden einer verketteten Liste müssen etwas anders behandelt werden. Zum einen, damit die Kette angesprochen werden kann, muss von Außerhalb auf die Kette verwiesen werden. Zum andern muss das hintere Ende markiert werden. In der folgenden Abbildung wurde die Kette mit einem Anfang versehen, das auf das erste Element zeigt. Dieser Anfang kann ein Zeiger in dem Programm sein, über den die Kette angesprochen wird. Um das Ende der Kette zu markieren kann z.B. der Zeiger des letzten Elementes auf sich selbst zeigen. Eine andere Möglichkeit ist, das letzte Element auf eine definierte Stelle zeigen zu lassen, z.B. auf die Speicheradresse *NULL*. Diese Methode wird hier im Weiteren verwendet.

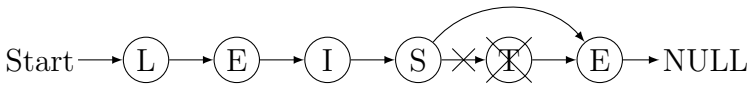


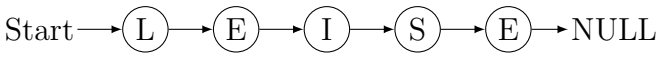
Soll in eine verkettete Liste ein Element eingefügt werden, so müssen nur die Zeiger der benachbarten Elemente angepasst werden. Alle anderen Elemente bleiben unverändert.



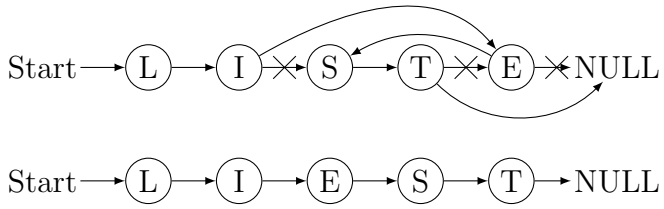


Soll ein Element aus einer verketteten Liste entfernt werden, so müssen wieder nur die Zeiger der benachbarten Elemente verändert werden. Alle anderen Elemente bleiben unverändert.



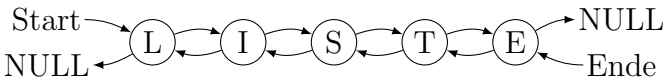


Um ein Element zu verschieben muss das Element an der Herkunftsstelle entfernt und an der Zielstelle eingefügt werden. In der folgenden Abbildung wird der Buchstabe E vom hinteren Ende in die Mitte verschoben.



Alle bisher aufgeführten verketteten Listen sind einfach verkettet. Durch die Zeiger ist es möglich, sich vom ersten Element schrittweise bis zum letzten Element durchzuhangeln. Um zu einem vorherigen Element zu gelangen, muss die Liste wieder von vorne durchgegangen werden.

Damit eine verkettete Liste in beide Richtungen durchlaufen werden kann, muss sie doppelt verkettet werden. Jedes Element beinhaltet nun zwei Zeiger, einen für das nächste Element und einen für das vorherige Element.



6.2.2 Erste Ansätze zur Implementierung

Wir wollen hier als Beispiel eine Kette für das Wort „Tag“ erstellen, die Kette füllen, sie ausgeben und danach wieder löschen (freigeben). Die Programmstücke in diesem Abschnitt ergeben ein lauffähiges Programm und können zu Übungszwecken nachprogrammiert werden. Allerdings stellt das Programm nur eine erste Einführung dar, die so nicht in Anwendungen verwendet werden sollte...

Zunächst muss eine Struktur für die Elemente der Kette erstellt werden. Die Struktur muss die Information des Kettenelementes und die Zeiger zum Verketten der Elemente beinhalten. In unserem Fall ist die zu speichernde Information ein Buchstabe und für die *einfache* Verkettung wird entsprechend nur *ein* Zeiger benötigt. Um die Verwendung der Kettenelemente zu vereinfachen, wird mit *typedef* ein neuer Datentyp definiert.

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

/* Struktur und Typ für die Elemente der verketteten Liste */
struct sKette {
    char Buchstabe;
    struct sKette *next;
};
typedef struct sKette tKette;
```

Die verkettete Liste braucht einen Startpunkt, von dem aus sie angesprochen werden kann. In diesem Beispiel wird das Ende der Kette mit einem Zeiger auf NULL markiert. Wir erstellen einen Zeiger auf ein Kettenelement, der mit NULL initialisiert wird. Des Weiteren wird ein Fehlerindikator definiert und mit *null* (was für „kein Fehler“ steht) initialisiert.

```
int main()
{
    tKette *Start=NULL; /* Zeiger auf erstes Element */
    int Fehler=0;       /* Fehlerindikator */
```

Als nächstes wird das erste Kettenelement erstellt und mit dem Buchstaben *T* gefüllt. Es wird dafür der Zeiger *Start* verwendet.

```
    /* Buchstabe 'T' */
    if(!Fehler) {
        Start = (tKette*)malloc(sizeof(tKette));
        if(Start==NULL) Fehler = -1;
    }
    if(!Fehler) Start->Buchstabe = 'T';
```

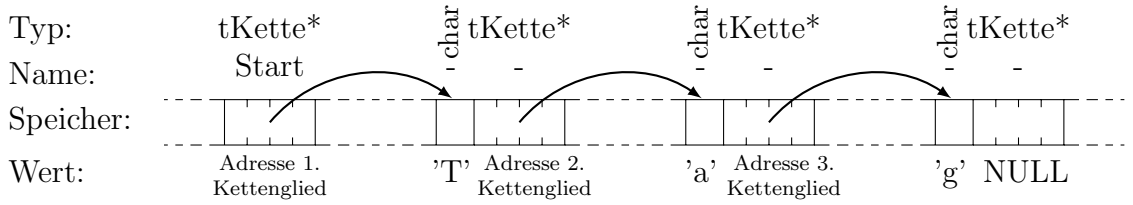
Jetzt zeigt *Start* auf das erste Kettenelement mit dem Buchstaben *T*. Der Zeiger *naechstes* soll nun auf ein Kettenelement mit dem zweiten Buchstaben zeigen. Also wird wieder Speicher reserviert und ein Buchstabe gesetzt:

```
    /* Buchstabe 'a' */
    if(!Fehler) {
        Start->next = (tKette*)malloc(sizeof(tKette));
        if(Start->next==NULL) Fehler = -2;
    }
    if(!Fehler) Start->next->Buchstabe = 'a';
```

Das gleiche geschieht für den dritten Buchstaben. Da dies zugleich der letzte Buchstabe ist, wird in diesem letzten Kettenelement der Zeiger *naechstes* auf NULL gesetzt.

```
    /* Buchstabe 'g' */
    if(!Fehler) {
        Start->next->next = (tKette*)malloc(sizeof(tKette));
        if(Start->next->next==NULL) Fehler = -3;
    }
    if(!Fehler) Start->next->next->Buchstabe = 'g';
    /* Zeiger im letzten Element auf NULL setzen */
    if(!Fehler) Start->next->next->next = NULL;
```

Damit ist die Kette vollständig erstellt und gefüllt. Für die vier Eigenschaften der Kette ergibt sich folgendes Diagramm:



Als nächstes soll der Text (hier das Wort *Tag*) auf dem Bildschirm ausgegeben werden. Wir sprechen dafür der Reihe nach die drei Kettenelemente an.

```
    /* gespeicherten Text ausgeben */
    if(!Fehler) {
        printf("%c", Start->Buchstabe);
        printf("%c", Start->next->Buchstabe);
        printf("%c", Start->next->next->Buchstabe);
```

```
        }
        printf("\n");
```

Am Ende muss natürlich der Speicher wieder freigegeben werden. Da für jeden Buchstaben einzeln Speicher reserviert wurde, muss dieser auch einzeln freigeben werden.

```
        /* Speicher freigeben */
        if(Start) {
            if(Start->next) {
                if(Start->next->next)
                    free(Start->next->next);
                free(Start->next);
            }
            free(Start);
        }

        return 0;
    }
```

Diese Art der Implementierung einer einfach verketteten Liste wird für längere Ketten zunehmend umständlich, ist für variable Kettenlängen ungeeignet und sollte hier nur der Veranschaulichung dienen. Der folgende Abschnitt stellt eine bessere Implementierung vor.

6.2.3 Verbesserte Implementierung einer einfach verketteten Liste

In der hier vorgestellten Implementierung einer einfach verketteten Liste wird das *Füllen*, *Ausgeben* und *Löschen* der Kette jeweils in einer Funktion erledigt. Die Kette soll wieder einen Text speichern, diesmal aber mit beliebiger Länge. Die einzelnen Programmstücke in diesem Abschnitt ergeben wieder ein lauffähiges Programm.

Zunächst wird wieder die Struktur und der Datentyp für die Kettenelemente erstellt und die Funktionen zum Füllen, Ausgeben und Löschen der Kette deklariert.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   struct sKette {
5       char Buchstabe;
6       struct sKette *next;
7   };
8   typedef struct sKette tKette;
9
10  int KetteFuellen(char *Text, tKette **Start);
11  void KetteAusgeben(tKette *Start);
12  void KetteLoeschen(tKette **Start);
```

In dem Hauptprogramm *main()* definieren wir die Variable *Start* als Zeiger auf den neuen Datentyp, der im Folgenden als Startpunkt für die verkettete Liste dient. Das Ende der Liste wird in diesem Beispiel wieder mit einem Zeiger auf *NULL* markiert. Da zunächst noch keine Liste existiert, wird der Start-Zeiger mit *NULL* initialisiert.

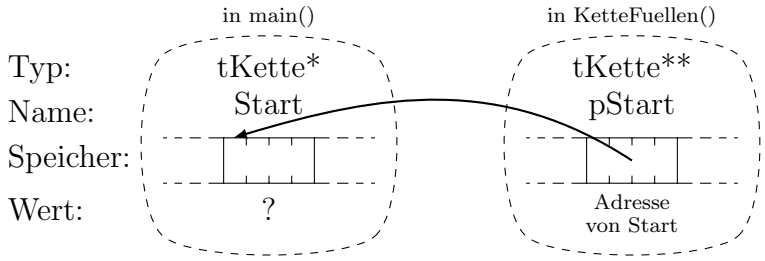
Danach rufen wir die vorher deklarierten Funktionen auf, um einen Text zu speichern, auszugeben und schließlich wieder zu löschen.

```
14  int main()
15  {
16      tKette *Start=NULL; /* Zeiger auf erstes Element */
17
18      /* Einen Text speichern, ausgeben und wieder freigeben */
19      KetteFuellen("Dies ist eine verkettete Liste.\n", &Start);
20      KetteAusgeben(Start);
21      KetteLoeschen(&Start);
22
23      return 0;
24  }
```

Mit der Funktion *KetteFuellen()* soll die Kette erstellt und gefüllt werden. Die Zeichenkette wird als Zeiger auf *char* entgegengenommen. Da die Funktion den Zeiger *Start* aus dem Hauptprogramm manipulieren soll, muss die Adresse des Zeigers übergeben werden und als Zeiger auf Zeiger entgegengenommen werden.

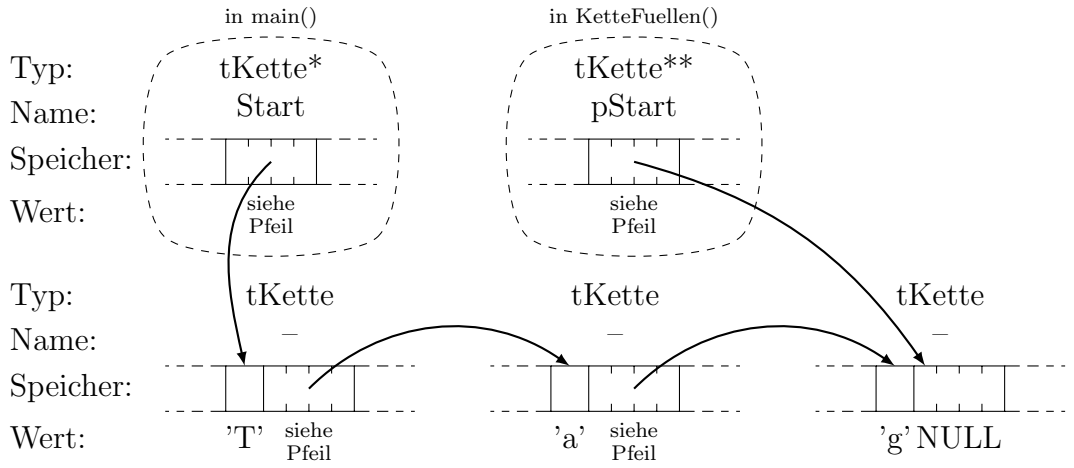
```
26  int KetteFuellen(char *Text, tKette **pStart)
27  {
28      /* ggf. alte Kette löschen */
29      KetteLoeschen(pStart);
30
31      /* solange noch Buchstaben vorhanden */
32      while(*Text) {
33          /* Speicher reservieren */
34          *pStart = (tKette*)malloc(sizeof(tKette));
35          if(*pStart==NULL) return -1;
36          /* Buchstaben kopieren */
37          (*pStart)->Buchstabe = *Text;
38          /* Zeiger setzen */
39          pStart = &(*pStart)->next;
40          /* Textzeiger um eins verschieben */
41          Text++;
42      }
43      *pStart = NULL;
44
45      return 0;
46  }
```

Beim Aufruf wird der Funktion *KetteFuellen()* die Adresse des Zeigers *Start* übergeben. Dieser Zeiger auf einen Zeiger erhält in der Funktion den Nahmen *pStart*. Direkt nach dem Funktionsaufruf ergibt sich folgendes Bild (ohne den Zeiger *Text*):



Als erstes wird in Zeile 29 die Funktion *KetteLoeschen()* aufgerufen, um eine eventuell bereits existierende Kette zu löschen. Mit der *while*-Schleife in Zeile 32 bis 42 werden alle Buchstaben des übergebenen Textes durchgegangen. Bei jedem Schleifendurchgang wird Speicher in Zeile 34 für ein neues Element reserviert. Konnte kein

Speicher reserviert werden, wird in Zeile 35 die Schleife vorzeitig abgebrochen. Bei Erfolg wird in Zeile 37 der aktuelle Buchstabe kopiert und in Zeile 39 der Doppelzeiger *pStart* auf den Zeiger für das nächste Element gesetzt. Wurde das Ende des Textes erreicht, so wird in Zeile 43 der Zeiger im letzten Kettenelement auf NULL gesetzt. Angenommen, die Funktion *KetteFuellen()* wurde mit dem Text "Tagäufgerufen, so ergibt sich am Ende der Funktion folgendes Bild (wieder ohne den Zeiger *Text*):



Der Text der verketteten Liste wird mit der Funktion *KetteAusgeben()* auf dem Bildschirm dargestellt. Da der Zeiger *Start* aus dem Hauptprogramm in dieser Funktion nicht verändert wird, reicht es eine Kopie dieses Zeigers zu übergeben.

```
48  void KetteAusgeben(tKette *Start)
49  {
50      /* solange Elemente vorhanden */
51      while(Start) {
52          /* Buchstabe ausgeben */
53          printf("%c", Start->Buchstabe);
54          /* Zeiger setzen */
55          Start = Start->next;
56      }
57  }
```

Die Schleife in Zeile 51 bis 56 wird solange wiederholt, bis der Zeiger *Start* auf *NULL* zeigt. Da es sich bei dem Parameter *Start* nur um eine Kopie des Zeigers aus dem Hauptprogramm handelt, kann er innerhalb der Funktion ohne Bedenken verändert werden. Innerhalb der Schleife wird in Zeile 53 der Buchstabe des aktuellen Elements ausgegeben und in Zeile 55 der Zeiger *Start* auf das nächste Element gesetzt.

Bei der Einrichtung der Kette wurde für jedes Zeichen einmal Speicher reserviert. Bevor das Programm beendet wird, muss dieser Speicher wieder freigegeben werden. Dafür erstellen wir die Funktion *KetteLoeschen()*, die als einzigen Parameter einen Zeiger auf den Zeiger *Start* erwartet, der seinerseits auf das erste Element der Kette zeigt.

```
59  void KetteLoeschen(tKette **pStart)
60  {
61      tKette *next;       /* Zeiger auf nächstes Element */
```

```
62
63          /* solange noch Elemente vorhanden */
64          while(*pStart) {
65              /* nächstes Element merken */
66              next = (*pStart)->next;
67              /* aktuelles Element freigeben */
68              free(*pStart);
69              /* nächstes Element übernehmen */
70              *pStart = next;
71          }
72      }
```

Innerhalb der Funktion wird in Zeile 61 zunächst ein Zeiger auf ein Element der Kette definiert. Die Schleife in Zeile 64 bis 71 wird solange wiederholt, bis der Zeiger, auf den *pStart* zeigt, auf *NULL* zeigt. Bevor innerhalb der Schleife in Zeile 68 das aktuelle Element freigegeben wird, muss in Zeile 66 die Adresse des nächsten Elements kopiert werden. (Das ist wichtig, da nach der Freigabe des Speichers dieser unter Umständen gleich wieder überschrieben wird.) Nach Freigabe des Speichers wird om Zeile 70 der Zeiger, auf den der Parameter *pStart* zeigt, auf das nächste Element gesetzt, und die Schleife beginnt von vorne.

In dem hier aufgeführten Beispiel wird in jedem Kettenglied nur ein Buchstabe gespeichert. Sinnvoller ist eine solche Kette bei größeren Datenpaketen. So könnten z.B. in einem Geschäft alle Artikel über eine verkettete Liste verwaltet werden. Jeder Artikel hat einen Namen, eine Gruppe, einen Preis, eine Barcode-Nummer etc. Wenn jetzt ein Element eingefügt, entfernt oder verschoben werden soll, so müssen nur die entsprechenden Zeiger angepasst werden.

6.2.4 Implementierung einer doppelt verketteten Liste

Für eine doppelt verkettete Liste sollen hier nur die wichtigsten Programmstücke erläutert werden. Das „Drumherum“ muss vom Leser ergänzt werden.

Im Prinzip muss bei der doppelt verketteten Liste einfach alles nur doppelt gemacht werden. Insgesamt bietet die Liste aber mehr Flexibilität, da in beide Richtungen gearbeitet werden kann. Zunächst benötigen wir wieder eine Struktur, diesmal aber mit zwei Zeigern. Der Einfachheit halber verwenden wir für die beiden Zeiger die englischen Begriffe *prev* (previous, vorheriger) und *next* (nächster). Was in die verkettete Liste gespeichert werden soll, lassen wir hier offen.

```
struct sKette {
    struct sKette *prev;
    struct sKette *next;
    /* Hier die spezifischen Daten einfügen */
};
typedef struct sKette tKette;
```

Diese Liste wird an beiden Enden aufgehängt. Dafür werden die Zeiger *Start* und *Ende* definiert. Um anzudeuten, dass zunächst noch kein Kettenglied existiert, werden beide Zeiger mit *NULL* initialisiert.

```
tKette *Start=NULL;
tKette *Ende=NULL;
```

Nehmen wir nun an, es existiert bereits eine Liste, und es soll ein Element, auf das der Zeiger *pNeu* zeigt, hinter das Element, auf das der Zeiger *pAktuell* zeigt, eingefügt werden, so geschieht das mit den folgenden Zeilen.

```
pNeu->prev = pAktuell;
pNeu->next = pAktuell->next;
pNeu->prev->next = pNeu;
pNeu->next->prev = pNeu;
```

An den beiden Enden der Liste ist Achtung geboten. Zum einen, der *next*-Zeiger des letzten Elementes zeigt wieder auf *NULL* und muss gesondert behandelt werden. Zum anderen, um ein Element am vorderen Ende einzufügen, muss anders vorgegangen werden, da es ja kein Element gibt, *hinter* das Sie das neue einfügen könnten.

Soll ein Element, auf das der Zeiger *pAktuell* zeigt, aus einer existierenden Liste entfernt werden, so sind nur zwei Zeilen nötig.

```
pAktuell->prev->next = pAktuell->next;
pAktuell->next->prev = pAktuell->prev;
```

Wieder müssen die beiden Enden der Liste gesondert behandelt werden.

6.3 Vergleich von Vektoren mit Ketten

Der Vergleich von Ketten mit Vektoren muss vielschichtig durchgeführt werden. In Tabelle 6.1 werden einige Punkte betrachten und verglichen. Am Ende muss bei jeder Fragestellung entschieden werden, welche Art von Liste zu bevorzugen ist.

	Liste als Vektor	**verkettete Liste**
Aufwand	gering (+)	höher (−)
Flexibilität	schlecht (−)	besser (+)
schneller bei ... Elementen	kleinen	großen
für große Datenmengen	nein (−)	ja (+)
extra Speicher für Zeiger	nein (+)	ja (−)
Liste verlängerbar	nein (−)	ja (+)

Tabelle 6.1: Vergleich von Listen als Vektor und als Kette

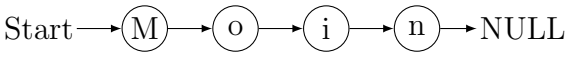
6.4 Aufgaben

Aufgabe 6.1: Was ist eine verkettete Liste?

Aufgabe 6.2: Erstellen Sie die Struktur eines Elementes einer zweifach verketteten Liste. Jedes Element soll eine komplexe Zahl beinhalten.

Aufgabe 6.3: Erstellen Sie eine Liste als Vektor für 100 Studenten mit Vornamen, Nachnamen und Matrikelnummer.

Aufgabe 6.4: Erstellen Sie ein Programm, welches folgende verkettete Liste realisiert.



Aufgabe 6.5: Sie wollen aus einer Liste mit 1000 Elementen das zweite Element entfernen. Bei welcher Liste geht das schneller, bei der Liste als Vektor oder bei der verketteten Liste.

Aufgabe 6.6: Sie wollen eine Liste mit genau 20 Elementen gleicher Größe erstellen. Welche Liste ist zu bevorzugen, die Liste als Vektor oder die verkettete Liste?

Aufgabe 6.7: Eine Liste besteht aus folgenden Elementen:

```
struct sElement {
    struct sElement *next;
    struct sElement *prev;
    int number;
};
```

Fügen Sie das Element *pElementNew* zwischen die Elemente *pElement1* und *pElement2* ein (das sind jeweils Zeiger auf die Elemente).

Anhang A

Rangfolge der Operatoren

In der folgenden Tabelle sind alle Operatoren von standard C mit ihrer Rangfolge zusammengefasst. Die letzte Spalte gibt die Verarbeitungsreihenfolge innerhalb einer Befehlszeile mit mehreren Operatoren des Ranges an.

Rang	**Operatoren**	**Reihenfolge**
1	() [] –> .	von links nach rechts
2	! ~ ++ –– + – * & (type) **sizeof**	*von rechts nach links*
3	* / %	von links nach rechts
4	+ –	von links nach rechts
5	<< >>	von links nach rechts
6	< <= > >=	von links nach rechts
7	== !=	von links nach rechts
8	&	von links nach rechts
9	^	von links nach rechts
10	|	von links nach rechts
11	&&	von links nach rechts
12	||	von links nach rechts
13	?:	*von rechts nach links*
14	= += –= *= /= %= &= ^= |= <<= >>=	*von rechts nach links*
15	,	von links nach rechts

Bei den Operatoren +, –, * und & im Rang 2 handelt es sich um die unären Varianten (positives und negatives Vorzeichen, Verweis- und Adressoperator).

Anhang B

Weitere nützliche Funktionen

In diesem Abschnitt sind einige weitere Funktionen aufgeführt. Die Auswahl richtet sich nach den im Praktikum benötigten Funktionen. Alle Funktionen, die im Laufe des Skripts erläutert wurden, können über den Index gefunden werden. Natürlich gibt es ein vielfaches an weiteren Funktionen. Verwenden Sie hierfür die allgemein verfügbaren Sprachreferenzen im Internet oder die dem Compiler beigefügte Hilfe.

B.1 Umwandlung von Text in Zahlen

B.1.1 atoi()

Die Funktion atoi() wandelt eine ASCII-Zeichenkette in eine ganze Zahl um. Die Syntax lautet:

```
int atoi(const char *text);
```

Headerdatei: <stdlib.h>

text. Zeiger auf eine mit null abgeschlossene Zeichenkette.

Rückgabewert. Bei Erfolg wird die umgewandelte Zahl, bei einem Fehler der Wert null zurückgegeben. Wird der Wertebereich überschritten, so ist das Ergebnis undefiniert.

B.1.2 atof()

Die Funktion atof() wandelt eine ASCII-Zeichenkette in eine Gleitkommazahl um. Die Syntax lautet:

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

Headerdatei: <stdlib.h> und <math.h>

text. Zeiger auf eine mit null abgeschlossene Zeichenkette.

Rückgabewert. Bei Erfolg wird die umgewandelte Zahl, bei einem Fehler der Wert null zurückgegeben. Wird der Wertebereich überschritten, so ist das Ergebnis undefiniert.

B.1.3 sscanf()

Die Funktion sscanf() wertet eine ASCII-Zeichenkette gemäß einer Formatanweisung um, und speichert die Ergebnisse in eine variable Anzahl an Variablen. Die Syntax lautet:

int sscanf(**const char** *text, **const char** *format [, argument] ...);

Headerdatei: <stdio.h>

text. Zeiger auf eine mit null abgeschlossene Zeichenkette.

format. Format-Zeichenkette wie bei der Funktion scanf().

argument. Zeiger auf eine Variable zum Speichern eines umgewandelten Werts. Je nach Formatanweisung werden hier mehrere Zeiger auf Variablen aufgeführt.

Rückgabewert. Bei Erfolg wird die Anzahl der korrekt umgewandelten Werte zurückgegeben. Wurde das Ende der Zeichenkette erreicht, bevor ein Wert umgewandelt werden konnte, ist der Rückgabewert EOF (*end of file*, Wert -1).

B.2 Hilfsmittel zum Verarbeiten von Dateien

In diesem Abschnitt werden die wichtigsten Werkzeuge für das Arbeiten mit Dateien zusammengefasst. Für eine vollständige Beschreibung sollten Sie in einer Sprachreferenz nachschlagen.

B.2.1 Die Datei-Struktur FILE

Hinter dem Wort *FILE* verbirgt sich eine Datenstruktur mit einigen Daten, die für das Bearbeiten von Dateien nötig sind. Auf die einzelnen Datenelemente braucht hier nicht eingegangen werden, da normalerweise nur mit der Struktur als Ganzes gearbeitet wird. Die Funktion *fopen()* liefert als Ergebnis einen Zeiger auf diese Struktur, mit dem dann alle Dateioperationen durchgeführt werden können.

Es stehen drei globale *FILE*-Variablen zur Verfügung:

stdout ist der Ausgabekanal für den Bildschirm.

stdin ist der Eingabekanal für die Tastatur.

stderr ist der Ausgabekanal für Fehlermeldungen. Standardmäßig erfolgt die Ausgabe auch auf dem Bildschirm. Sie kann aber mit *fopen* in eine Fehlerdatei umgeleitet werden.

Zu vielen der Datei-Funktionen stehen identische Funktionen für die Bildschrimausgabe und Tastaturabfrage zur verfügung. Beispiel:

fprintf(stdout, FormatString, ...)
führt exakt zu dem selben Ergebnis wie
printf(FormatString, ...).

B.2.2 fopen()

Der Befehl *fopen()* öffnet eine Datei. Die Systax lautet:

FILE *fopen(**const char** *name, **const char** *mode);

name. Zeiger auf eine mit null abgeschlossene Zeichenkette mit dem Dateinamen (z.B. "Brief.txt"). Dem Dateinamen kann auch eine Pfadangabe hinzugefügt werden (z.B. "C:\Texte\Brief.txt"). Ohne Pfadangabe wird die Datei im aktuellen Verzeiuchnis geöffnet.

mode. Zeiger auf eine mit null abgeschlossene Zeichenkette mit der der Modus der zu öffnenden Datei festgelegt wird. Der Modus setzt sich aus zwei Teilen zusammen. Der erste Teil legt fest *wie* auf die Datei zugegriffen werden soll:

"r" Öffnet eine Datei nur zum Lesen (engl. *read*). Die Datei muss bereits existieren.

"w" Erstellt eine neue Datei nur zum Schreiben (engl. *write*). Existiert die Datei bereits, so wird der alte Inhalt beim Öffnen gelöscht.

"a" Öffnet eine Datei nur zum Schreiben. Existiert die Datei bereits, so wird der vorherige Inhalt nicht gelöscht, sondern die neuen Daten an das Ende angehängt (engl. *append*).

"r+" Öffnet eine Datei zum Lesen und Schreiben. Die Datei muss bereits existieren.

"w+" Erstellt eine neue Datei zum Lesen und Schreiben. Existiert die Datei bereits, so wird der alte Inhalt beim Öffnen gelöscht.

"a+" Öffnet eine Datei zum Lesen und Schreiben. Alle geschreibenen Daten werden ans Ende angehängt. Existiert noch keine Datei, so wird eine neue erstellt.

Der zweite Teil der Zeichenkette *mode* legt fest, ob es sich um eine Text- oder Binärdatei handelt.

"t" Die Datei wird als Text-, bzw. ASCII-Datei behandelt. In diesem Modus wird das Ende einer Zeile besonders behandelt.

"b" Die Datei wird als Binär-Datei behandelt. In diesem Modus werden alle Daten ungefiltert geschrieben und gelesen.

Die Angabe, ob es sich um eine Text- oder Binär-Datei handelt kann auch weggelassen werden. In diesem Fall wird die Art der Datei durch die globale Variable *_fmode* festgelegt, die standardmäßig die Art *Text-Datei* vorgibt. Es ist aber empfehlenswert, immer die Dateiart beim Öffnen festzulegen.

Rückgabewert. Konnte die Datei erfolgreich geöffnet werden, wird ein Zeiger auf eine Dateistruktur vom Typ *FILE* zurückgegeben. Tritt ein Fehler auf, so wird der Zeiger *NULL* (Zeiger auf die Adresse 0) zurückgegeben. Es sollte bei jedem Öffnen einer Datei dieser Rückgabewert abgefragt werden und Fehler entsprechend bearbeitet werden.

B.2.3 fclose()

Die Funktion *fclose()* schließt eine offene Datei. Die Syntax lautet:

```
int fclose(FILE *stream);
```

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss geöffnet sein.

Rückgabewert. Wurde die Datei erfolgreich geschlossen, so gibt *fclose()* den Wert null zurück. Tritt ein Fehler auf, so wird der Wert *EOF* zurückgegeben.

B.2.4 feof()

Die Funktion *feof()* prüft, ob versucht wurde, über das Ende einer Datei hinaus zu lesen. Die Syntax lautet:

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

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Lesen geöffnet sein.

Rückgabewert. Wurde über das Dateiende hinaus gelesen, so gibt *feof()* einen Wert ungleich null (das entspricht einem logischen *wahr*) zurück. War der letzte Lesevorgang erfolgreich in den Grenzen der Datei, so wird der Wert null (logisch *nicht wahr*) zurückgegeben.

B.2.5 fputc()

Die Funktion *fputc()* schreibt ein Zeichen in eine Textdatei. Die Syntax lautet:

```
int fputc(int ch, FILE *stream);
```

ch. Zeichen, das in die Datei geschrieben werden soll. Auch wenn der Parameter den Typ *int* hat, wird nur ein Byte geschrieben. Es kann auch der Typ *char* übergeben werden.

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Schreiben geöffnet sein.

Rückgabewert. Die Funktion liefert als Ergebnis das geschriebene Zeichen zurück. Tritt ein Fehler auf, so wird das Zeichen für Dateiende *EOF* (engl. *end of file*, Wert -1) zurückgegeben.

B.2.6 fgetc()

Die Funktion *fgetc()* liest ein Zeichen aus einer Textdatei. Die Syntax lautet:

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

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Lesen geöffnet sein.

Rückgabewert. Die Funktion liefert als Ergebnis das gelesene Zeichen im Wertebereich 0-255 zurück. (Es kann aber direkt in eine *char*-Variable geschrieben werden.) Wurde das Dateiende erreicht, oder ist ein Fehler aufgetreten, so wird das Zeichen für Dateiende *EOF* (engl. *end of file*, Wert -1) zurückgegeben.

B.2.7 fputs()

Die Funktion *fputs()* schreibt eine Textzeile in eine Textdatei. Die Syntax lautet:

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

string. Zeiger auf die Zeichenkette, die in die Datei geschrieben werden soll.

stream. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Schreiben geöffnet sein.

Rückgabewert. Beim erfolgreichen Schreiben der Zeile wird das zuletzt geschriebene Zeichen (Wert 0-255) zurückgegeben. Tritt ein Fehler auf, so wird das *EOF*-Zeichen (Wert -1) zurückgegeben.

B.2.8 fgets()

Die Funktion *fgets()* liest eine Textzeile aus einer Textdatei. Die Syntax lautet:

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

string. Zeiger auf eine Zeichenkette, in die die gelesene Zeile geschrieben werden soll.

n. Maximale länge der Zeichenkette inklusive der abschließenden null.

stream. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Lesen geöffnet sein.

Rückgabewert. Bei einem erfolgreichen einlesen der Zeile wird eine Kopie des Zeigers auf die Zeichenkette zurückgegeben. Tritt ein Fehler auf, oder wurde das Ende der Datei erreicht, so wird eine *NULL* (Zeiger auf die Adresse 0) zurückgegeben.

Die Funktion *fgets()* liest eine Textzeile aus einer geöffneten Datei. Die Länge der eingelesenen Zeile ist begrenzt durch das nächste Zeilenumbruchzeichen, das Ende der Datei oder die maximale Länge *n*.

Die Funktion *fgets()* kann mit *stdin* auch sinnvoll zum begrenzten Einlesen von der Tastatur verwendet werden, ohne dass bei einer zu langen Eingabe der Speicherbereich überschritten wird. Beispiel:

```
char Text[21];
fgets(Text, 21, stdin);
```

B.2.9 fprintf()

Die Funktion *fprintf()* dient der formatierten Ausgabe von Text in eine Textdatei. Die Syntax lautet:

```
int fprintf(FILE *stream, const char *format [, argument] ...);
```

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Schreiben geöffnet sein.

format. Die Format-Zeichenkette folgt den selben Regeln wie die Format-Zeichenkette in der Funktion *printf()*. Es können wie gewohnt alle Platzhalter (z.B. "%d" oder "%f") und Formatanweisungen (z.B. "%5d" oder "%8.3f") verwendet werden.

argument. Hinter der Format-Zeichenkette müssen die Werte angehängt werden, für die in der Format-Zeichenkette Platzhalter vorgesehen sind.

Rückgabewert. Die Funktion liefert als Ergebnis die Anzahl der geschriebenen Zeichen (Bytes). Tritt ein Fehler auf, so wird ein negativer Wert zurückgegeben.

Die Funktion *fprintf()* ist die allgemeine Version zur Funktion *printf()*. Die Funktion *fprintf()* erwartet zusätzlich als ersten Parameter einen Zeiger auf die Struktur *FILE*, ansonsten sind alle Parameter identisch.

B.2.10 fscanf()

Die Funktion *fscanf()* dient dem formatierten Einlesen von Text aus einer Textdatei. Die Syntax lautet:

```
int fscanf(FILE *stream, const char *format [, argument] ...);
```

stream. Die Funktion erwartet einen Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Lesen geöffnet sein.

format. Die Format-Zeichenkette folgt den selben Regeln wie die Format-Zeichenkette in der Funktion *scanf()*. Es können wie gewohnt alle Platzhalter (z.B. "%d" oder "%f") verwendet werden.

argument. Hinter der Format-Zeichenkette müssen Zeiger auf die Variablen angehängt werden, für die in der Format-Zeichenkette Platzhalter vorgesehen sind.

Rückgabewert. Die Funktion liefert als Ergebnis die Anzahl der erfolgreich gelesenen und konvertierten Variablen zurück. Wurde kein Wert eingelesen, so ist der Rückgabewert null. Tritt ein Fehler auf, so wird das Zeichen *EOF* (Wert -1) zurückgegeben.

Die Funktion *fscanf()* ist die allgemeine Version zur Funktion *scanf()*. Die Funktion *fscanf()* erwartet zusätzlich als ersten Parameter einen Zeiger auf die Struktur *FILE*, ansonsten sind alle Parameter identisch.

B.2.11 fwrite()

Die Funktion *fwrite()* schreibt Daten in eine Binärdatei. Die Syntax lautet:

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *str);
```

ptr. Zeiger auf die Stelle im Speicher, an der die Daten zum Schreiben stehen.

size. Größe einer Einheit, die geschreiben werden soll. Auch wenn hier der Datentyp *size_t* erwartet wird, kann hier ein *short*-, *int*- oder *long*-Wert übergeben werden.

n. Anzahl der Einheiten, die geschrieben werden sollen. Auch wenn hier der Datentyp *size_t* erwartet wird, kann hier ein *short*-, *int*- oder *long*-Wert übergeben werden.

str. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Schreiben geöffnet sein.

Rückgabewert. Es wird die Anzahl der erfolgreich geschriebenen Einheiten zurückgegeben. Wurde der Schreibvorgang erfolgreich abgeschlossen, dann entspricht der Rückgabewert dem Parameter *n*. Tritt ein Fehler auf, so wird die Anzahl der Einheiten zurückgegeben, die noch vor dem Fehler geschrieben wurden. Wurde keine Einheit geschrieben, so wird eine null zurückgegeben.

B.2.12 fread()

Die Funktion *fread()* liest Daten aus einer Binärdatei. Die Syntax lautet:

```
size_t fread(const void *ptr, size_t size, size_t n, FILE *str);
```

ptr. Zeiger auf die Stelle im Speicher, an die die gelesenen Daten gespeichert werden sollen.

size. Größe einer Einheit, die gelesen werden soll. Auch wenn hier der Datentyp *size_t* erwartet wird, kann hier ein *short*-, *int*- oder *long*-Wert übergeben werden.

n. Anzahl der Einheiten, die gelesen werden sollen. Auch wenn hier der Datentyp *size_t* erwartet wird, kann hier ein *short*-, *int*- oder *long*-Wert übergeben werden.

str. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Lesen geöffnet sein.

Rückgabewert. Es wird die Anzahl der erfolgreich gelesenen Einheiten zurückgegeben. Wurde der Lesevorgang erfolgreich abgeschlossen, dann entspricht der Rückgabewert dem Parameter *n*. Tritt ein Fehler auf, so wird die Anzahl der Einheiten zurückgegeben, die noch vor dem Fehler gelesen wurden. Wurde keine Einheit gelesen, so wird eine null zurückgegeben.

B.2.13 fflush()

Die Funktion *fflush()* erzwingt das physikalische Schreiben des Dateipuffers. Die Syntax lautet:

```
int fflush(FILE *stream);
```

stream. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss zum Schreiben geöffnet sein.

Rückgabewert. Bei Erfolg gibt *fflush()* den Wert null zurück. Tritt ein Fehler auf, so wird der Wert -1 zurückgegeben.

Beim Lesen und Schreiben von Dateien wird ein Puffer verwendet, damit der Zugriff beschleunigt wird. Erst wenn beim Lesen der Puffer leer, bzw. beim Schreiben der Puffer voll ist, wird auf den Datenträger zugegriffen.

Bei Dateien zum Schreiben sorgt die Funktion *fflush()* dafür, dass der aktuelle Inhalt des Puffers in die Datei geschrieben wird, auch wenn dieser noch nicht voll ist. Die Funktion kann bei Dateien sinnvoll sein, die gleichzeitig zum Lesen und Schreiben geöffnet wurden.

B.2.14 ftell()

Die Funktion *ftell()* ermittelt die aktuelle Position innerhalb einer geöffneten Datei. Sie lässt sich sinnvoll zusammen mit der Funktion *fseek()* einsetzen. Die Syntax lautet:

```
long ftell(FILE *stream);
```

stream. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss geöffnet sein.

Rückgabewert. Bei Erfolg gibt die Funktion die aktuelle Position in der geöffneten Datei zurück. Tritt ein Fehler auf, so wird der Wert -1 zurückgegeben.

B.2.15 fseek()

Die Funktion *fseek()* beeinflusst die aktuelle Position in einer offenen Datei. Sie kann verwendet werden, um beim Lesen einer Datei einen Block zu überspringen, oder um zu einer Stelle in der Datei zurück zu springen. Die Syntax lautet:

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

stream. Zeiger auf eine Dateistruktur vom Typ *FILE*. Die Datei muss geöffnet sein.

offset. Position in der Datei relativ zu der mit *origin* angegebenen Position.

origin. Position, von der aus innerhalb der Datei gesprungen werden soll. Der Parameter kann einen der folgenden drei Werte annehmen:

SEEK_SET. Relativ zum Anfang der Datei.

SEEK_CUR. Relativ zur aktuellen Position.

SEEK_END. Relativ zum Ende der Datei.

Rückgabewert. Bei Erfolg ist der Rückgabewert null. Tritt ein Fehler auf, so wird ein Wert ungleich null zurückgegeben.

B.2.16 remove()

Die Funktion *remove()* löscht eine Datei. Die Syntax lautet:

int remove(**const char** *filename);

filename. Zeichenkette mit Dateinamen und ggf. Pfadnamen der zu löschenden Datei.

Rückgabewert. Wurde die Datei erfolgreich gelöscht, so gibt *remove()* den Wert null zurück. Tritt ein Fehler auf, so wird der Wert -1 zurückgegeben.

B.3 Hilfsmittel zur dynamischen Speicherverwaltung

Für die dynamische Speicherverwaltung stehen in C die folgenden Funktionen zur Verfügung. Die Deklarationen befinden sich in der Include-Datei *stdlib.h*. Zunächst gehen wir auf den Datentyp *size_t* ein, bevor wir vier Funktionen genauer betrachten.

B.3.1 Datentyp size_t

Der Datentyp *size_t* wird verwendet, um die Größe von Objekten zu speichern. Hinter dem Typ *size_t* verbirgt sich ein *unsigned int*. Hat ein Parameter einer Funktion den Typ *size_t*, so kann auch der Type *int* oder *short* (*signed* oder *unsigned*) übergeben werden. Soll der Typ *long* übergeben werden, muss evtl. eine Typumwandlung vorgenommen werden. Beispiel:

```
ptr = malloc((size_t)LongVariable);
```

B.3.2 malloc()

Die Funktion *malloc()* reserviert einen Speicherblock ohne ihn zu initialisieren. Die Syntax lautet:

void *malloc(size_t size);

size Anzahl der Bytes, für die Speicher reserviert werden soll.

Rückgabewert Bei Erfolg wird ein Zeiger auf die Stelle im Speicher zurückgegeben, an der der Speicherblock reserviert wurde. Tritt ein Fehler auf, wird der Zeiger *NULL* (Zeiger auf die Adresse null) zurückgegeben.

Die Funktion *malloc()* reserviert *size* Bytes Speicher ohne ihn zu initialisieren. Zurückgegeben wird ein Zeiger auf *void*, der auf den reservierten Speicherblock zeigt. Bei der Zuweisung des Zeigers sollte eine Typumwandlung in den gewünschten Datentyp erfolgen. Tritt ein Fehler auf (z.B. es ist nicht genügend Speicher vorhanden) wird der Zeiger NULL (Zeiger auf die Adresse null) zurückgegeben. Nach jeder Speicherreservierung sollte geprüft werden, ob ein Fehler aufgetreten ist. Beispiel:

```
int *data;

data = (int*)malloc(10*sizeof(int));
if(data==NULL) {
    /* Fehlerbehandlung */
}
```

B.3.3 calloc()

Die Funktion *calloc()* reserviert Speicher für einen Vektor und initialisiert ihn mit null. Die Syntax lautet:

```
void *calloc( size_t  num, size_t  size );
```

num Anzahl der Elemente im Vektor, für die Speicher reserviert werden soll.

size Anzahl der Bytes für ein Element des Vektors.

Rückgabewert Bei Erfolg wird ein Zeiger auf die Stelle im Speicher zurückgegeben, an der der Speicherblock reserviert wurde. Tritt ein Fehler auf, wird der Zeiger *NULL* (Zeiger auf die Adresse null) zurückgegeben.

Die Funktion *calloc()* reserviert Speicher für einen Vektor und initialisiert den Speicher mit null. Zurückgegeben wird ein Zeiger auf *void*, der auf den reservierten Speicher zeigt. Bei der Zuweisung des Zeigers sollte eine Typumwandlung in den gewünschten Datentyp erfolgen. Tritt ein Fehler auf (z.B. es ist nicht genügend Speicher vorhanden) wird der Zeiger NULL (Zeiger auf die Adresse null) zurückgegeben. Nach jeder Speicherreservierung sollte geprüft werden, ob ein Fehler aufgetreten ist. Beispiel:

```
int *data;

data = (int*)calloc(10, sizeof(int));
if(data==NULL) {
    /* Fehlerbehandlung */
}
```

B.3.4 realloc()

Die Funktion *realloc()* reserviert einen Speicherblock mit veränderter Größe. Die Syntax lautet:

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

memblock Zeiger auf den zuvor reservierten Speicherblock. Wird der Zeiger *NULL* übergeben, so wird ein neuer Speicherblock reserviert.

size Anzahl der Bytes, für die Speicher reserviert werden soll.

Rückgabewert Bei Erfolg wird ein Zeiger auf die Stelle im Speicher zurückgegeben, an der der Speicherblock reserviert wurde. Tritt ein Fehler auf, wird der Zeiger *NULL* (Zeiger auf die Adresse null) zurückgegeben.

Die Funktion *realloc()* reserviert einen Speicherblock mit veränderter Größe. Wird als erster Parameter ein Zeiger auf null übergeben, so wird neuer Speicher reserviert und die Funktion verhält sich wie die Funktion *malloc()*.

Zeigt der erste Parameter von *realloc()* auf einen zuvor reservierten Speicherblock, so wird die Größe des Blockes entsprechend angepasst. Der zurückgegebene Zeiger kann dabei auf dieselbe, oder auf eine andere Stelle im Speicher zeigen. Der Inhalt des Speicherblockes bleibt bis zu der kleineren der alten und neuen Größe unverändert. Das gilt auch, wenn sich der Speicherblock an einer neuen Position befindet.

Zurückgegeben wird ein Zeiger auf *void*, der auf den reservierten Speicher zeigt. Bei der Zuweisung des Zeigers sollte eine Typumwandlung in den gewünschten Datentyp erfolgen. Tritt ein Fehler auf (z.B. es ist nicht genügend Speicher vorhanden) wird der Zeiger NULL (Zeiger auf die Adresse null) zurückgegeben. Nach jeder Speicherreservierung sollte geprüft werden, ob ein Fehler aufgetreten ist. Beispiel:

```
int *data;
int *tmp;

data = (int*)malloc(10*sizeof(int));          /* 10 int-Werte */
if(data==NULL) {
    /* Fehlerbehandlung */
}
tmp = (int*)realloc(data, 20*sizeof(int));    /* 20 int-Werte */
if(tmp==NULL) {
    /* Fehlerbehandlung */
}
data = tmp;
```

Im Fehlerfall wird der zuvor reservierte Speicher nicht freigegeben. Der Speicher mit der vorigen Größe kann weiter verwendet werden bzw. muss freigegeben werden.

B.3.5 free()

Die Funktion *free()* gibt einen zuvor mit *malloc()*, *calloc()* oder *realloc()* reservierten Speicherblock wieder frei. Die Syntax lautet:

void free(**void** *memblock);

memblock Zeiger auf den zuvor reservierten Speicher. Ein Zeiger auf null führt zu keiner Aktion. Ein Zeiger auf eine Stelle, die nicht zuvor reserviert wurde, führt zu einem undefinierten Ergebnis und kann zu erheblichen Problemen führen!

Rückgabewert Es wird kein Wert zurückgegeben.

Die Funktion *free()* gibt zuvor reservierten Speicher wieder frei. Bei kleinen kurzlebigen Programmen fällt es kaum auf, wenn dieser Befehl fehlt, da beim Beenden des Progamms automatisch der reservierte Speicher freigegeben wird. Bei langlebigen Programmen führt dass allerdings dazu, dass der verfügbare Speicher immer

weiter abnimmt. Es wird von *Speicherleckage* (engl. *memory leakage*) gesprochen. Es gibt leider auch namenhafte Betriebssysteme, die Speicherlecks aufweisen. Solche Betriebssysteme müssen von Zeit zu Zeit neu gestartet werden.

Anhang C

Lösungen zu den Aufgaben

C.1 Lösungen zu Kapitel 1

1. **int** *a;
   double *b;
   char *c;

2. **double** **ppZahl;

3. $0123\,3218_{16}$

4. $01FF\,3E54_{16}$

5. text[4]

6. *Zahlen[2] = 3.141;

7. Die Variablen x und y haben die Werte 2.4 und 2.

8. pInt[2]

9. **unsigned** *ZeigerVerktor[9];

10. **int** (*pVektor)[8];

11. Die Ergebnisse der Reihe nach: 2, 'n', 7, 'W', 5, 'K'

C.2 Lösungen zu Kapitel 2

1. FILE *inp;

2. fopen("Brief.txt", "rt")

3. In einer Binärdatei wird jedes Byte auf gleiche Weise, unabhängig von dessen Inhalt, als Zahl interpretiert und gelesen oder geschrieben. Bei einer Textdatei werden einige Zeichen besonders behandelt (z.B. Zeilen- oder Dateiende). Jede Textdatei kann auch als Binärdatei verarbeitet werden, aber nicht jede Binärdatei als Textdatei.

4. Mit einer Variable vom Typ Zeiger auf FILE finden alle Dateizugriffe statt.

5.

```
FILE *out;
out = fopen("Address.dat", "wb");
if(out==NULL) {
    printf("Could not open file Address.dat\n");
    return -1;
}
```

6. fread() und fwrite()

7. Die Datei muss geschlossen werden.

8. fprintf(out, "%8d %8d %8d\n", a, b, c);

9. feof()

10. fflush()

11. remove()

12.

```
#include <stdio.h>

int main()
{
    FILE *out=NULL; /* Zeiger für Ausgabedatei */
    double f;       /* Temperatur in Fahrenheit */

    /* Überschrift */
    printf("Erstellen einer Tabelle zur Umrechnung von "
        "Fahrenheit in Celsius.\n");

    /* Datei öffnen */
    out = fopen("Temperatur.csv", "wt");
    if(out==NULL) {
        printf("Konnte Ausgabe Datei nicht öffnen!\n");
        return -1;
    }

    /* Tabelle erstellen */
    fprintf(out, "Fahrenheit,Celsius\n");
    for(f=0; f<160; f+=50)
        fprintf(out, "%.2lf,%.2lf\n", f, 5*(f-32)/9);

    /* Datei schließen */
    if(out) fclose(out);

    /* Schlussmeldung */
    printf("Die Tabelle wurde in die Datei "
        "'Temperatur.csv' geschreiben.\n");

    return 0;
}
```

C.3 Lösungen zu Kapitel 3

1.
```
struct sDatum {
    int Tag;
    int Monat;
    int Jahr;
};
```

2. **struct** sDatum Datum = { 12, 9, 2007 };

3. Datum.Monat = 5;

4. **struct** sDatum *pDatum = &Datum;

5. pDatum->Tag = 1;

6.
```
struct sSchule {
    char Name[31];
    int AnzahlKlassen;
    struct sSchulKlasse Klasse[20];
};
```

7.
```
enum eMonat { Januar=1, Februar, Maerz, April, Mai, Juni,
    Juli, August, September, Oktober, November, Dezember };
```

8. **enum** eFarbe { rot, gruen, blau };

9. **enum** eFarbe { rot, gruen, blau, red=0, green, blue };

10.
```
union uZahl {
    double Zahl1;
    double Zahl2;
    double Zahl3;
};
```

11.
```
union uInt {
    int Zahl;
    char Byte[4];
};
```

12.
```
union uDouble {
    double dZahl;
    int iZahl[2];
    short sZahl[4];
    char cZahl[8];
};
```

13. **typedef unsigned** DWORD;

14.
```
typedef struct {
    int Tag;
    int Monat;
    int Jahr;
} tDatum;
```

C.4 Lösungen zu Kapitel 4

1. Die Funktion malloc() reserviert Speicher ohne ihn zu initialisieren, während die Funktion calloc() alle Bytes des reservierten Speichers auf null setzt.

2.
```
Note = (int*)malloc(16*sizeof(int));
if(Note==NULL) {
    printf("Not enough memory available!\n");
    return -1;
}
```

3.
```
pDouble = (double*)calloc(100, sizeof(double));
if(pDouble==NULL) {
    printf("Not enough memory available!\n");
    return -1;
}
```

4. Es fehlt die Typumwandlung: **char** *Text = (**char***)malloc(100***sizeof**(**char**));

5. Mit der Funktion free() wird der zuvor reservierte Speicher wieder freigegeben.

6. Die Funktionen malloc(), calloc() und realloc() melden einen Fehler, indem sie einen Zeiger auf NULL zurückgeben.

7. Wenn Speicher dynamisch reserviert und nicht wieder freigegeben wird, steht dem Rest des Systems danach weniger Speicher zur Verfügung. Dieser Vorgang wird *Speicherleckage* (*memory leakage*) genannt.

8. Nach der Verwendung dynamischen Speichers muss der Speicher wieder freigegeben werden.

C.5 Lösungen zu Kapitel 5

1. Charakteristisch für eine rekursive Funktion ist, dass sie sich selbst aufruft.

2. Die *inkrementelle Aufgabe* ist das, was die rekursive Funktion neben den rekursiven Aufrufen selbst durchführt. Die *Abbruchbedingung* sorgt dafür, dass Die Rekursion nur bis zu einer begrenzten Verschachtelungstiefe erfolgt. Die *Traversierung* gibt an, in welcher Reihenfolge die inkrementelle Aufgabe relativ zu den rekursiven Aufrufen erfolgt.

3.

```
void Oktal(unsigned Zahl)
{
    if(Zahl) {
        Oktal(Zahl/8);
        printf("%c", Zahl%8+'0');
    }
}
```

4. Bei der vorherigen Aufgabe liegt eine *Postorder-Traversierung* vor.

5.

```
void Oktal(unsigned Zahl)
{
    unsigned Temp[11];
    int i=0;

    while(Zahl) {
        Temp[i++] = Zahl;
        Zahl /= 8;
    }
    while(i) printf("%c", Temp[--i]%8+'0');
}
```

6. Die Berechnung der Fibonacci-Zahlen sollte nicht rekursiv erfolgen, da hier die Anzahl der rekursiven Aufrufe exponentiell ansteigt. Bei großen Zahlen ergeben sich so sehr lange Rechenzeiten.

7.

```
int Summe(int a, int b)
{
    if(a>b) return 0;
    return a+Summe(a+1, b);
}
```

8.

```
void VariableBasis(unsigned Zahl, unsigned Basis)
{
    char ch;
    if(Zahl) {
        VariableBasis(Zahl/Basis, Basis);
        ch = Zahl%Basis+'0';
        printf("%c", ch>'9'?ch+7:ch);
    }
}
```

C.6 Lösungen zu Kapitel 6

1. Eine verkettete Liste ist eine Menge von Elementen mit Zeigern, die jeweils auf ihre Nachbarelemente zeigen.

2.

```
struct sElement {
    struct sElement *prev
    struct sElement *next;
    double re;
    double im;
};
```

3.

```
struct sStudent {
    char Vorname[51];
    char Nachname[51];
    char MatrNr[8];
} Student[100];
```

4. Siehe Abschnitt 6.2.2 oder 6.2.3

5. Aus einer *verketteten Liste* lässt sich leichter ein Element entfernen als aus einer Liste, die als Vektor implementiert wurde.

6. Kleine Listen mit Elementen konstanter Größe lassen sich besser als *Vektor* implementieren.

7.

```
pElementNew->prev = pElement1;
pElementNew->next = pElement2;
pElement1->next = pElementNew;
pElement2->prev = pElementNew;
```

Stichwortverzeichnis