Programmieren in C/C++

Dipl.-Ing. Dr.techn. Maximilian Mayr

Inhaltsverzeichnis

Kapitel 1

Elementare Datentypen

Datentype	Länge	Wertebereich	Genauigkeit
Character			
char signed char unsigned char	1	$-2^7 ... + 2^7 - 1$ $-2^7 ... + 2^7 - 1$ $0 ... 2^8 - 1$	2
Integer			
int signed int unsigned int	2/4	$-2^{15} ... + 2^{15} - 1 / -2^{31} ... + 2^{31} - 1$ $-2^{15} ... + 2^{15} - 1 / -2^{31} ... + 2^{31} - 1$ $0 ... 2^{16} - 1 / 0 ... 2^{32} - 1$	4/9
Short Integer			
short signed short unsigned short	2	$-2^{15} ... + 2^{15} - 1$ $-2^{15} ... + 2^{15} - 1$ $0 ... 2^{16} - 1$	4
Long Integer			
long signed long unsigned long	2	$-2^{31} ... + 2^{31} - 1$ $-2^{31} ... + 2^{31} - 1$ $0 ... 2^{32} - 1$	9
Aufzählungstypen			
enum {list} enum id {list} enum id	2/4	wie int	

Tabelle 1.1: Ganzzahltypen: Die angegebenen Werte stellen Beispiele für 16/32-Bit-Umgebungen dar, sie sind implementierungsabhängig!

Datentype	Länge	Wertebereich	Genauigkeit
Gleitkommatypen			
float	4	$\approx -10^{\pm 38}...10^{\pm 38}$	7
double	8	$\approx -10^{\pm 308}...10^{\pm 308}$	15
long double	10	$\approx -10^{\pm 4932}...10^{\pm 4932}$	19
Typisierte und nicht typisierte Zeiger			
type *	2/4	$0...2^{16}-1$ / $0...+2^{32}-1$	
void *	2/4	$0...2^{16}-1$ / $0...+2^{32}-1$	4/9

Tabelle 1.2: Fließkommazahlen und Zeiger: Die angegebenen Werte stellen Beispiele für 16/32-Bit-Umgebungen dar, sie sind implementierungsabhängig!

1.1 Namen für Variable (und andere Sprachkonstrukte)

Für die Wahl von Variablennamen gilt folgende Regel:

- Ein Name besteht aus beliebigen Folgen von Buchstaben, Ziffern und Unterstrichen. Das erste Zeichen muss ein Buchstabe sein oder ein Unterstrich sein.
- Bezüglich der Namenslänge kann es implementierungsabhängige Einschränken geben.
- Es wird zwischen Klein- und Großschreibung unterschieden.
- Ein Name darf kein Schlüsselwort sein.

Die Regel für die Bildung von Namen für Variablen gilt auch für die Bildung von Namen für andere Sprachkonstrukte, wie z.B. Typen, Konstanten und Funktionen.

1.2 Darstellung von Zahlen

In C können ganze Zahlen nicht nur dezimal, sondern auch oktal und hexadezimal dargestellt werden, z.B.:

1023 123 5 027033 01111 0xab 0x3f 0xff

Die ersten drei Zahlen sind dezimal, die nächsten drei oktal, die letzten drei hexadezimal:

- dezimal zahlen beginnen stets mit einer Ziffer ungleich 0.
- oktale Zahlen beginnen stets mit 0.
- hexadezimale Zahlen beginnen mit 0x oder mit 0X.

Durch Anhängen von Suffixen (u, U, l, L) kann der Typ der Konstanten präzisiert werden, und zwar stehen u und U für *unsigned*, d.h. vorzeichenlos sowie l und L für long, z.B.:

2023L 2023UL 2023LU 023u 023ul 0xfful 0xffL

1.2.1 Darstellung von Fließkommazahlen

Zur Darstellung nicht ganzer Zahlen gibt es eine Vielzahl von Schreibweisen, z.B.:

12.2345 123.0 123. 12.45e-3 123.0e20 0.23e12 1e30

Auch hier können zusätzliche Suffixe (f, F, l, L) verwendet werden, um explizit den exakten Typ zu spezifizieren, und zwar stehen f und F für *float* sowie l und L für *long double*. Eine nicht ganze Zahl ohne Suffix hat den Typ double, d.h. double ist der *natürliche* Typ für Fließkommazahlen.

1.3 Darstellung von Zeichen

Druckbare Zeichen werden üblicherweise wie folgt dargestellt:

'a' 'X' 'O' '3' ' ' '0' '+' '?' '&'

Zeichen können durch ihre Ordnungszahl in dem zugrundeliegenden Zeichensatz (z.B. ASCII[1]-Code) bezeichnet werden:

$97 \rightarrow' a'$, $88 \rightarrow' X'$, $48 \rightarrow' 0'$, $33 \rightarrow'!'$

Dabei ist aber zu bedenken, dass das zugrundeliegende Alphabet und die entsprechenden Ordnungszahlen inplementiertungsunabhängig sind! Darüber sind auch die folgendenDarstellungen möglich:

'\07'	→	Klingelzeichen	'\011'	→	horiz. Tabulator
'\013'	→	vert. Tabulator	'\075'	→	'A'
'\x7'	→	Klingelzeichen	'\x9'	→	horiz. Tabulator
'\xb'	→	vert. Tabulator	'\x41'	→	'A'

Hinter dem Backslash \ (Rückschrägstrich) wird oktal oder hexadezimal die Ordnungszahl angegeben. Diese Form heißt *oktale* bzw. *hexadezimale Escape-Sequenz*, sie wird insbesondere zur Bezeichnung von Steuerzeichen verwendet.

Als letztes gibt es die sogenannte *einfache Escape-Sequenz* zur Bezeichnung einiger spezieller Zeichen:

'\' '	einfaches Anführungszeichen
'\" '	doppeltes Anführungszeichen
'\?'	Backslash
'\a'	Klingelzeichen
'\b'	Backspace
'\f'	Seitenvorschub
'\n'	neue Zeile, new Line
'\r'	Begin der Zeile setzen, carriage return
'\t'	horizontaler Tabulator
'\v'	vertikaler Tabulator

[1] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

Kapitel 2

Operatoren für elementare Datentypen

Die Tabellen 2.1, 2.2 und 2.3 geben eine Übersicht über die in C verfügbaren Operatoren und ihre jeweilige Bedeutung.

2.1 Binäre Operatoren

Arithmetik:

+, -, ∗, / und %: Die ersten vier Operatoren betreffen die aus der Mathematik bekannten Grundrechnungsarten für Zahlen und sind auf alle Integer-Typen, auf Character-Typen und auf Fließkommatypen anwendbar. Der Operator % ist nicht auf Fließkommatypen anwendbar und steht für die Modulo-Operation, d.h. er liefert den Rest einer ganzzahligen Division ($5\%2 \equiv 1$, $27\%10 \equiv 7$, $35\%5 \equiv 0$). Die Division zweier ganzzahliger Operanden liefert ein ganzzahliges Ergebnis ($\frac{5}{2} \equiv 2$). Eine Operation mit einem ganzzahligen und einem Gleitkommatyp wird im Gleitkommabereich durchgeführt ($\frac{5.0}{2} \equiv 2.5$). Die Operatoren + und - können auch im begrenzten Umfang auf Adressen und Zeiger angewendet werden (*Zeigerarithmetik*).

Vergleich:

<, <=, ==, !=, >= und >: Die Vergleichsoperatoren sind im wesentlichen auf alle elementaren Datentypen anwendbar, auf Zeiger allerdings nur der Test auf Gleichheit sowie auf Ungleichheit. Das Ergebnis einer Vergleichsoperation ist vom Typ int, der Wert ist 0 oder 1 (für FALSE bzw. TRUE). Die Vergleichsoperanden sollten möglichst dem gleichen Datentyp angehören, es können aber auch verwandte Datentypen - wie z.B. int und long int - verglichen werden.

Operator	Bedeutung	Art	anwendbar auf
+	Addition		
-	Subtraktion		
*	Multiplikation	Arithmetik	Zahlen
/	Division		
%	Divisionsrest		nur ganze Zahlen
<	Vergl. auf *kleiner*		
<=	Vergl. auf *kleiner gleich*		
==	Vergl. auf *gleich*	Vergleich	alle Typen
!=	Vergl. auf *ungleich*		
>=	Vergl. auf *größer gleich*		
>	Vergl. auf *größer*		
&	bitw. UND-Verknüpfung		
|	bitw. ODER-Verknüpfung		
∧	bitw. EXOR-Verknüpfung	Bitoperatoren	ganzzahlige
< <	bitw. Linksschieben		Typen
> >	bitw. Rechtsschieben		
&&	log. UND-Verkn. in Ausdr.	Logische	Boolsche
||	log. ODER-Verkn. in Ausdr.	Verknüpfung	Werte

Tabelle 2.1: Binäre Operatoren für elementare Datentypen

Bitoperationen:

&, |, ˜, < < und > >: Die Operatoren &, |, und ˜ beziehen sich auf die binäre Darstellung ganzzahliger Typen und führen jeweils bitweise UND-, ODER- bzw. EXOR-Verknüpfungen durch, das sieht dann bei der üblichen Verwendung des Dualcodes am Beispiel der Zahlen $5_d = 101_b$ und $4_d = 100_b$ wie folgt aus:

- 5 & 4 → 000 ..... 00101 & 000 ..... 00100 ≡ 000 ..... 00100 → 4
- 5 | 4 → 000 ..... 00101 | 000 ..... 00100 ≡ 000 ..... 00101 → 5
- 5 ˜ 4 → 000 ..... 00101 ˜ 000 ..... 00100 ≡ 000 ..... 00001 → 1

Die Operatoren < < und > > beziehen sich ebenfalls auf die binäre Darstellung ganzzahliger Typen und führen Links- bzw. Rechtsverschiebungen auf den linken Operanden um eine durch den rechten Operanden definierte Stellenzahl durch, z.B.:

- 5 < < 2 → 000 ..... 00101 < < 2 ≡ 000 ..... 10100 → 20
- 20 > > 2 → 000 ..... 10100 > > 2 ≡ 000 ..... 00101 → 4

Beim Links- und Rechtsverschieben werden jeweils von rechts bzw. von links binäre Nullen nachgeschoben. Eine Linksverschiebung um n Stellen entspricht einer Multiplikation mit 2^n, eine Rechtsverschiebung um n Stellen entsprechend einer Division durch 2^n.

Logische Verknüpfung:

&& und | |: Die Operatoren && und | l verknüpfen logische Werte und Ausdrücke durch ein logisches UND bzw. durch ein logisches ODER, z.B.:

```
int Okey, NotOkey;
double x;
.....
Okey = (x >= 0) && (x < 100);
NotOkey = (x < 0) || (x >= 100);
if(Okey && NotOkey)
  printf("Das kann nicht sein!\n");
```

2.2 Unäre Operatoren

Referenzierung/Dereferenzierung:

& und *: Der Operator & liefert die Adresse im Arbeitsspeicher, unter der das durch den Operanden bezeichnete Datum abgelegt ist, entsprechend liefert der Operator $*$ das Datum, das unter der durch den Operanden bezeichneten Adresse abgelegt ist. Eine Adresse ist typgebunden und wird z.B. wie folgt vereinbart:

```
double *x;
```

Das bedeutet: der Inhalt von x ist vom Typ double, also ist x eine Adresse, die auf einen Speicherbereich zeigt, in dem ein double-Wert steht. Das folgende kleine Beispiel demonstriert den prinzipiellen Umgang mit den beiden Operatoren & und $*$:

```
double x;
double *adresse;

x=1.23;
adresse = &x;
printf("%f %f\n", x,*adresse); //Ausgabe: 1.23 1.23
```

Operator	Bedeutung	Art	anwendbar auf
&	Adressse von	Refernzierung	alle Typen
*	Inhalt von	Drefernzierung	Adresssen
+	pos. Vorzeichen	Arithmetik	Zahlen
-	neg. Vorzeichen		
~	bitw. Invertierung	Bitoperation	ganzz. Typen
!	log. Invertierung	Log. Verkn.	Bool Werte
(*type*)	Typkonvertierung	Typisierung	alle Typen
sizeof	expr.: Speicherbedarf		Ausdrücke
sizeof	sizeof(*type*): speicherbedarf		Typen

Tabelle 2.2: Unäre Operatoren für elementare Datentypen

Arithmetik:

+ und -: Die beiden unären Operatoren werden als positives bzw. als negatives Vorzeichen für Zahlen verwendet.

Bitoperation:

~: Der unäre Operator ~ gehört inhaltlich zu den oben behandelten Bitoperationen, er bewirkt, dass in der binären Darstellung des entsprechenden ganzzahligen Datentyps alle Bit invertiert werden.

Logische Verknüpfung:

!: Der unäre Operator ! gehört inhaltlich zu den oben behandelten logischen Verknüpfungen, er invertiert einen Boole'schen Ausdruck,
z.B. !((x<0) || (x>=100)) = !(x<0) && !(x>=100) = (x>=0) && (x<100).

Typkonvertierung:

(type) dazu später mehr

Speicherbedarf:

sizeof: Der Operator sizeof wird in zwei unterschiedlichen Zusammenhängen verwendet, er liefert den Speicherbedarf eines konkreten Objektes oder eines Datentyps in Byte, z.B.:

```
double x;
int xLength = sizeof x;
int doubleLength = sizeof(double);
```

Wenn, wie im zweiten Fall, der Operand ein Typ ist, dann muss er geklammert werden!

2.3 Postfix- und Präfix-Operatoren

Operator	Bedeutung	Art	anwendbar auf
++ - -	Inkrementierung Dekrementierung	Arithmethik	ganzz. Typen und Zeiger

Tabelle 2.3: Postfix- und Präfix-Operatoren

Inkrementierung/Dekrementierung:

++ und - -: Durch Voranstellen oder Anhängen des Operators ++ oder - -wird der zugehörige Ausdruck jeweils um den Wert 1 erhöht bzw. um den Wert 1 erniedrigt, dabei muss der Operand einem ganzzahligen Typ angehören und er muss ein L-Wert sein (d.h. ein konkretes Objekt repräsentieren). Bei einfachen Anwendungen führen Voranstellen und Anhängen zum gleichen Ergebnis, z.B.:

```
int i=0, j=0;

i++;
++j; //i und j erhalten beide den Wert 1
```

i++ und ++j sind Kurzschreibweisen für i = i+1 bzw. j = j+1. In vielen Fällen führen Voranstellen und Anhängen auch zu verschiedenen Ergebnissen, z.B.:

Beispiel:

```
int x=2, y=3;
int z;

z= x++ + ++y;   // z = 2 + 4 = 6
// x=3 und y=4 für den weiteren Verlauf
```

2.4 Zuweisungs- und sonstige Operatoren

Zuweisung:

=: Der Zuweisungsoperator wird meist in der Form

- x = 4;
- y=3+a;

im Rahmen einer Anweisung verwendet. Dabei wird der Ausdruck auf der rechten Seite berechnet und dem Objekt der linken Seite zugeordnet (d.h. der Wert des Ausdrucks wird in den zu dem Objekt gehörenden Speicherbereich kopiert).

Verknüpfung und Zuweisung:

+=, -=, ∗=, /=, %=, &=, |=, ˜=, «= und »=: Diese Operatoren dienen wieder der verkürzten Schreibweise, und zwar gilt allgemein

- x+=3; → zu x wird 3 addiert, x=x+3;
- x∗=5; → x wird 5 multipliziert, x=x∗5;

bedingte Auswertung:

Bedeutung:

Beispiel: x < y ? x : y; → Wenn x < y, dann hat der Ausdruck den Wert x, sonst den Wert y.

Der erste Ausdruck wird ausgewertet, wenn er den Wert $\neq 0$ (d.h. true) hat, dann ergibt sich der Wert des zweiten Ausdrucks, sonst der des dritten Ausdrucks für den Gesamtausdruck.

Beispiel: z=x < y ? x : y;

Der bedingte Ausdruck wird ausgewertet und entsprechend der Bedeutung des Zuweisungsoperators dem L-Wert zugewiesen. Die bedingte Anweisung ist eine Kurznotation für die if/else-Anweisung.

Aufzählung in Klammerausdrücken:

,: Die sicher verbreitetste Verwendung der Operators , (Kommaoperator genannt) findet in der for-Anweisung statt, siehe dazu 4.1.3, auf weitere Erläuterungen wird hier verzichtet.

Zuweisungsoperatoren			
Operator	Bedeutung	Art	anwendbar auf
=	Wertzuweisung	Zuweisung	alle Typen
+=	Addition	Arithmetik und Zuweisung	Zahlen
-=	Subtraktion		
*=	Mutiplikation		
/=	Division		
%=	Divisionrest		nur ganze Z.
&=	bitw. UND-Verknüpfung	Bitopertion und Zuweisung	ganzz. Typen
|=	bitw. ODER-Verknüpfung		
∧ =	bitw. EXOR-Verknüpfung		
< <=	bitw. Linksschieben		
> >=	bitw. rechtsschieben		
sonstige Operatoren			
Operator	Bedeutung	Art	anwendbar auf
?:	bedingte Auswertung		Ausdrücke
,	Aufzählung in Klammerausdr.		Ausdrücke

Tabelle 2.4: Zuweisungs- und sonstige Operatoren

2.5 Ausdrücke

Ausdrücke sind Konstrukte - Folgen von Operatoren und Operanden - die dazu dienen, Verarbeitungsvorgänge zu formulieren. Die entsprechenden Möglichkeiten in der Programmiersprache C sind sehr vielfältig, der syntaktische Rahmen ist entsprechend komplex, deshalb besschränken wier uns an dieser Stelle auf einige Beispiel und Hinweise:

Beispiele:

- $\frac{a-b}{a+b}$ ⇒ (a−b)/a+b);
- $a^2 + b^2$ ⇒ a∗a + b∗b;
- $x^n - y^n$ ⇒ pow(x,n) + pow(y,n);
- $x^3 + 3x^2y + 3xy^2 + y^3$ ⇒ pow(x,3)+3∗pow(x,2)∗y+3∗x∗pow(y,2)+pow(y,3);
- $\sqrt[2]{a^2 + b^2}$ ⇒ sqrt(a∗a + b∗b);
- $\sqrt[n]{\frac{x^n-y^n}{1+u^{2n}}} = (\frac{x^n-y^n}{1+u^{2n}})^{\frac{1}{n}}$ ⇒ pow((pow(x,y)−pow(y,n))/(1+pow(u,2∗n)),1.0/(**double**)n);
- $sin(x + \frac{\pi}{2})$ ⇒ sin(x+pi/2);

pow, sqrt und sin sind Bezeichner für Funktionen aus der C-Standardbibliothek.

2.6 Objekte und L-Werte

Ein Objekt ist ein Bereich im Arbeitsspeicher, der einem Namen zugeordnet ist, d.h. eine Variable. Ein L-Wert (Ivalue) ist ein Ausdruck, der sich auf ein solches Objekt bezieht, z.B. ein Variablenname. In

der Zuweisung a1 = a2 muss der linke Ausdruck a1 ein L-Wert - also z.B. ein Variablenname - sein. Entsprechendes gilt auch bei der Verwendung der übrigen Zuweisungsoperatoren +=, ..., ||=.

Beispiele:

```
int a,b,*p;

a=27;    // richtig
a+=27;   // richtig
b++=27   // falsch, b++ ist kein L-Wert
a+b=27;  // falsch, a+b ist kein L-Wert

*p=27;           // richtig, *p ist ein L-Wert
(p++)++;         // falsch; p++ ist kein L-Wert
```

Hinweis:

- Die Reihenfolge der Auswertungen wird durch die Prioritäten der Operatoren und durch die Gruppierung der Unterausdrücke bestimmt, darüber hinaus können aber Freiheitsgrade bleiben (siehe weiter unten).
- Im Zweifelsfall und zur Erhöhung der Lesbarkeit sollten ggf. auch redundante Klammerpaare verwendet werden.
- Als Folge begrenzter Genauigkeit der Rechnerarithmetik können - auch bei gleichrangigen Operatoren - verschiedene Reihenfolgen der Auswertung zu unterschiedlichen Ergebnissen führen.
- Das bedeutet, dass auch z.B. das Kommutative Gesetz nicht immer anwendbar ist (z.B. bei Überlauf oder Unterlauf).
- Entsprechendes gilt für das Distributive Gesetz und für das Assoziative Gesetz. (Punkt- vor Strichrechnung).
- Bei einem Ausdruck oder Unterausdruck mit gleichrangigen Operatoren ist die Reihenfolge der Auswertung Undefiniert und damit implementierungsabhängig, hier können Klammern ggf. helfen, die Gruppierung - und damit die Reihenfolge der Auswertung - zu definieren.
- Das Verhalten bei Überlauf und Unterlauf ist implementierungsabhängig, üblicherweise wird in den Fällen kein Fehler angezeigt.

2.7 Prioritäten von Operatoren

Bezüglich der Prioritäten aller C-Operatoren gilt die in Tabelle 2.5 angegebene Reihenfolge.

Priorität	Operator	Bemerkung
0	(), [], ->, .	siehe spätere Abschnitte
1	!, ~, ++, - -, +, -, ∗, sizeof, (*type*)	unäre, postfix, präfix
2	∗, /, %	binäre arithmetische
3	+, -	Operatoren
4	< <, > >	Shift-Operatoren
5	<, <=, >, >=	Vergleichs-
6	==, !=	operatoren
7	&	Bitoperationen
8	∧	
9	|	
10	&&	Logische
11	||	Verknüpfungen
12	=, +=, -=, ∗ =, /=, %=, &=, ∧ =, |=, < <=, > >=	Zuweisungsoperatoren
13	,	Kommaoperator

Tabelle 2.5: Priorität von Operatoren

Die höchste Priorität haben die in Zeile (0) aufgeführten Operatoren, die erst in späteren Abschnitten behandelt werden. Die zweithöchste Priorität haben die in Zeile (1) notierten unären Operatoren sowie der Postfix- und der Präfixoperator. Die zweitniedrigste Priorität haben die in Zeile (12) angegebenen Zuweisungsoperatoren. Die Abstufungen sind mit Bedacht so festgelegt worden, damit in vielen Fällen auf Klammerungen verzichtet werden kann, wie es das folgende Beispiel zeigt, in dem die Darstellung ohne zusätzliche Klammern der Darstellung mit zusätzlichen Klammern gleichwertig ist:

Beispiel:

```
int a,b,c,d,y1,y2,y3;

y1=a < b && !(c == d);
y2=a & ~b | ~(c | d);
y3=a << b & c >> a;
```

Beispiel:

```
int a,b,c,d,y1,y2,y3;

y1=(a < b) && (!(c == d));
y2=(a & ~b) | (~(c | d));
y3=(a << b) & (c >> a);
```

2.8 Übungen zu Operatoren

1. Eine Bedingung ist wahr oder falsch. Die Eigenschaften “wahr” und “falsch” müssen einem bestimmten Wert entsprechen. Versuche mit Hilfe eines Programms herauszufinden, welche Werte die Programmiersprache C für “wahr” und “falsch” benützt:

```
int x;

// Für welche(n) Wert(e) ist diese Bedingung
// erfüllt bzw. nicht erfüllt?
if(x)
{
                    ...
}
```

2. Schreibe ein Programm, das eine Zahl einliest und den dazugehörigen Kehrwert ausgibt. Zusätzlich soll überprüft werden, ob die eingegebnen Zahl Null war. In diesem Fall soll eine Meldung ausgegeben werden, dass keine Kehrwert berechnet werden kann.

3. Schreibe ein Programm, welches zwei int-Werte von der Tastatur einliest und die größere (kleinere) der beiden Zahlen am Bildschirm ausgibt.

4. Schreibe ein Programm, welches drei int-Werte von der Tastatur einliest und die größte (kleinste) der drei Zahlen am Bildschirm ausgibt.

5. Schreibe ein Programm, welches einen int-Wert von der Tastatur einliest und am Bildschirm ausgibt, ob die Zahl gerade(even) oder ungerade(odd) ist.

6. Schreibe ein Programm, welches zwei int-Werte von der Tastatur einliest und die Summe der beiden Zahlen als int-Wert bildet. Das Programm soll nun prüfen, ob die Summe auch wirklich stimmt, das heißt ob kein Wertebereichsüberlauf aufgetreten ist.

7. Schreibe ein Programm, welches prüft, ob eine eingegebene Zahl durch einen bestimmten Wert (z.B. 3) teilbar ist. Die Programmiersprache C bietet dafür den %(Modulo)-Operator. Versuche ohne diesen Operator auszukommen!

8. Schreibe ein Programm, das die Einer-, Zehner- und Hunderterstelle einer eingegebenen dreistelligen Zahl ermittelt und diese untereinander am Bildschirm ausgibt.

9. Schreibe ein Programm, welches eine Jahreszahl von der Tastatur einliest und ermittelt, ob es sich um ein Schaltjahr handelt.

   (a) Ein Jahr ist kein Schaltjahr, wenn die Jahreszahl nicht durch 4 teilbar ist.

   (b) Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4, nicht aber durch 100 teilbar ist.

   (c) Ein Jahr ist ebenfalls ein Schaltjahr, wenn die Jahreszahl durch 4, durch 100 und durch 400 teilbar ist.

10. Schreibe eine einfache Variante eines Zahlenratespiels. Der Benutzer muss eine zufällige Zahl errraten und hat eine bestimmte Anzahl von Versuchen. Das Programm zeigt dem Benutzer nach jeder Eingabe an, ob die eingegebene Zahl größer oder kleiner als die gesuchte Zahl ist.

11. Schreibe ein Programm, welches einen 1-Byte-Wert (0..255) in eine int-Variable von der Tastatur einliest und als Dualzahl in Textform ausgibt.

    Überlege, wie die Werte der einzelnen Bits ermitteln können!

    Beispiel: Eingabe: 150 ⇒ Ausgabe 10010110

12. Wie kann man das folgende Listing vereinfachen? Beseht die Möglichkeit, beide *if*-Anweisung in einer zusammenfassen?

```
if(i > -5)
{
  if (i < 5)
    printf("i liegt zwischen -5 und +5.\n");
}
```

13. Schreibe ein Programm, das von der Tastatur eine Zahl einliest und danach prüft, ob eine Zahl größer, kleiner oder gleich Null ist, und gleichzeitig azsgibt, ob es sich um eine gerade oder ungerade Zahl handelt.

14. Welche Ausgabe erzeugt das folgende Programm? Gib eine Lösung an, so dass das Programm ordungsgemäß funktioniert.

```
#include <stdio.h>

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

  a=5;
  b=7;
  c=4;
  if(a<b)
    if (a<c)
      printf("a ist die kleinste Zahl.\n");
  else
      printf("a ist nicht die kleiner als b.\n");
}
```

15. Handelt es sich bei dem folgenden Programm um syntaktisch korrekten Code? Falls ja, welche Ausgabe erzeugt das Programm?

```
#include <stdio.h>

void main()
{
  int x=2, y=0;

  if(x>2 && (x/y))
    printf("Hallo Welt!\n");
}
```

16. Mit welchen Datentypen kann ein **switch−case**−Konstrukt arbeiten?

17. Erzeuge eine Programm, das von der Tastatur ein beliebiges Zeichen einliest und prüft, ob das eingegeben Zeichen ein Zahl zwischen 1 und 5 ist. Liegt die Zahl in diesem Intervall, ist sie auszugeben, ansonsten soll die Meldung erscheinen, dass die Zahl nicht zwischen 1 und 5 liegt. Löse die Aufgabe

    (a) mit einem **if**−**else**−Konstrukt.

    (b) mit einem **switch**−**case**−Konstrukt.

18. Welche Ausgabe erzeugt das folgende Programm, wenn der Anwender die 1 drückt? Ändere das Programm so, dass die richtige Ausgabe erfolgt.

```
#include <stdio.h>

void main()
{
  int iMenunummer;

  printf("Nummer eingeben: ");
  scanf("%i",&iMenunummer);
  fflush(stdin);

  switch(iMenunummer)
  {
  case 1:
    printf("Menü 1 ist ausgewählt worden\n");
  case 2:
    printf("Menü 2 ist ausgewählt worden\n");
    break;
  }
}
```

Kapitel 3

Datenformatierung

Alle Anwendungen zur formatierten Eingabe und Ausgabe enthalten als wesentliche Parameter Zeichenketten, die *Formatstring*, z.B.:

- **int** printf(**const char**∗ format, ...);
- **int** scanf (**const char** ∗ format, ...);

Die Formatstrings können Konvertierungssequenzen enthalten, mit denen die Form gelesenen bzw. geschriebenen Daten genau spezifiziert wird. Die Konvertierungssequenzen im Formatstring werden jeweils mit dem Zeichen % eingeleitet.
Beispiel: printf("Zahl = %i\n",x)

3.1 Konvertierungssequenzen für Ausgabe

Die Konvertierungssequenzen für die Ausgaben im Formatstring haben folgende allgemeine Form:

$$\%Flags_{opt}Feldbreite_{opt}Genauigkeit_{opt}Konvertierungsangabe$$

- *Flags (Steuerzeichen):*
  Eines der fünf Zeichen "- + *Leerzeichen* # 0" mit folgenden Bedeutungen:
  - \- bedeutet, dass die Ausgabe linksbündig erfolgt (default: rechtsbündig)
  - \+ bedeutet, dass bei der Ausgabe von Zahlen auch das positive Vorzeichen stets mit ausgegeben wird (default = es entfällt)
  - *Leerzeichen* bedeutet, dass bei der Ausgabe von Zahlen anstelle des positiven Vorzeichen ein Leerzeichen ausgegeben wird
  - \# bedeutet, "alternative Form" der Ausgabe, auf die hier nicht eingegangen wird
  - 0 bedeutet, dass bei der Ausgabe von Zahlen das Ausgabefeld ggf. mit Nullen (0) aufgefüllt wird

- *Feldbreite:*

  Eine ganze Zahl, die die Breite des Ausgabefeldes angibt.Wenn die Breite zu groß ist, wird mit Leerzeichen aufgefüllt, wenn sie zu klein ist, wird sie überschritten.

- *Genauigkeit:*

  ein Punkt (.) gefolgt von einer Zahl, die bei Fließkommazahlen die Anzahl der Nachkommastellen (bei Verwendung der Konvertierungszeichen f,e, und E) angibt bzw. der signifikanten Stellen (bei Verwendung der Konvertierungszeichen g und G) bzw. der MIndeststellenanzahl (bei VErwendung der Konvertierungszeichen d, i, o, u, x und X). Im letzten Falle, d.h. bei der Ausgabe ganzer Zahlen, wird die Differenz zwischen Mindeststellenanzahl und realer Stellenanzahl ggf. durch Nullen (0) aufgefüllt.

- *Konvertierungsangabe:*

  Die Konvertierungsangabe besteht aus einem der folgenden Konvertierungsspezifizierer:

  d i o x X u f e E g G c s p n %

  dem ggf. noch eines der drei Zeichen h l L vorangestellt werden kann. Die folgenden Tabellen 3.1, 3.2 und 3.3 stellt die entsprechenden Zeichen und Zeichenkombinationen jeweils mit einer kurzen Beschreibung zusammen.

Zeichen	Typ des Ausgabewertes	Form der Ausgabe
Integer		
d, i	signed int	dezimal, d.h. Zahl zur Basis 10
o, u	unsigned int	oktal bzw. dezimal
x, X	unsigned int	hexadez. mit 0...9a..f bzw. 0...9A...F
Short Integer		
hd, hi	signed short	
ho, hu	unsigned short	wie bei Integer
hx, hX	unsigned short	
Long Integer		
ld, li	signed long	
lo, lu	unsigned long	wie bei Integer
lx, lX	unsigned long	

Tabelle 3.1: Ausgaben von Ganzzahltypen mit printf, u.ä.

Zeichen	Typ des Ausgabewertes	Form der Ausgabe
float		
f	float	[-]ddd.ddd
e, E	float	[-]d.ddd**e**±dd bzw. [-]d.ddd**E**±dd
g, G	float	je nach Ggröße im e- oder f-Format
double		
lf	float	
le, lE	float	wie bei float
lg, lG	float	
long double		
Lf	float	
Le, LE	float	wie bei float
Lg, LG	float	

Tabelle 3.2: Ausgaben von Fließkommatypen mit printf, u.ä.

Zeichen	Typ des Ausgabewertes	Form der Ausgabe
Zeichen und Strings		
c	int, char	als Zeichen (unsigned char), ASCII-Code
s	char*	als String, Zeichenkette
%		das Zeichen% wird ausgegeben

Tabelle 3.3: sonstige Typen mit printf, u.ä.

Beispiel:

```
int x = 12;

printf("Zahl = %i\n", x);
```

Ausgabe: Zahl = 12

Beispiel:

```
double x = 12.3;

printf("Zahl = %lf\n", x);
```

Ausgabe: Zahl = 12.300000

Beispiel:

```
int tag = 9;
int monat = 5;
int jahr = 10;

printf("Datum = %2i.%2i.%4i\n",tag,monat,jahr);
```

Ausgabe:

Datum = 9. 5. 10

Beispiel:

```
int h = 9;
int min = 5;
int sec = 23;

printf("Uhrzeit = %02i:%02i:02i\n",h,min,sec);
```

Ausgabe: Uhrzeit = 09:05:23

Beispiel:

```
double x=17.2345

printf("Uhrzeit = %6.3lf\n",x);
```

Ausgabe: Ausgabe = 17.234

3.2 Konvertierungssequenzen für Eingaben

Die Konvertierungssequnz für Eingaben im Formatstring haben folgende allgemeine Form:

$$\%Stern_{opt} Feldbreite_{opt} Konvertierungsangabe$$

- *Stern:*

  Mit dem Zeichen * (Stern) wird die Zuweisung des entsprechenden Wertes an die zugehörige Variable unterdrückt.

- *Feldbreite:*

  Eine ganze Zahl, die die maximale Breite des Eingabefeldes angibt.

- *Konvertierungsangabe:*

  Die Konvertierungsangabe besteht aus einem der folgenden Konvertierungsspezifizierer,

  d i o x X u f e g c s p n %

  dem ggf. noch eines der drei Zeichen h l L vorangestellt werden kann. Die folgenden Tabellen 3.4 und 3.5 stellt die entsprechenden Zeichen und Zeichenkombinationen jeweils mit einer kurzen Besschreibung zusammen.

Zeichen	Typ des Eingabewertes	Form der Eingabe
Integer		
d	signed int *	dezimal, oktal oder hexadezimal
i	unsigned int *	oktal bzw. dezimal
o	unsigned int *	oktal
u	unsigned int *	dezimal, vorzeichenlos
x	signed int *	hexadezimal
Short Integer		
hd	signed short *	
hi	unsigned short *	
ho	unsigned short *	wie bei Integer
hu	unsigned short *	
hx	signed short *	
Long Integer		
ld	signed long *	
li	unsigned long *	
lo	unsigned long *	wie bei Integer
lu	unsigned long *	
lx	signed long *	
float, double und long double		
e, f, g	float *	
le, lf, lg	double *	gültiges Gleitkommaformat
Le, Lf, Lg	long double *	

Tabelle 3.4: Eingaben von Zahltypen mit scanf, u.ä.

Zeichen	Typ des Eingabewertes	Form der Eingabe
Integer		
c	char *	als Zeichen oder Zeichenfolge (ohne Abschluss mit ASCII-0)
s	char *	als String (ASCII-0 wird als Abschlusszeichen angefügt)

Tabelle 3.5: Eingaben von Zeichentypen mit scanf, u.ä.

Beispiel:

```
int x;

printf("Zahl = ");
scanf("%i",&x);
fflush(stdin);

printf("eingegebene Zahl = %i",x);
```

Ausgabe:

```
Zahl = 3
eingegebene Zahl = 3
```

Beispiel:

```
double x;

printf("Zahl = ");
scanf("%lf",&x);
fflush(stdin);

printf("eingegebene Zahl = %lf",x);
```

Ausgabe:

```
Zahl = 7.89
eingegebene Zahl = 7.890000
```

3.3 Übungen zu Datenformate

1. Gib mittels printf() folgende Zeichen am Bildschirm aus: \ ”
2. Schreib ein Programm, welches eine Zahl zwischen 0 und 255 von der Tastatur einliest und das entsprechende ASCII-Zeichen am Bildschirm ausgibt.
3. Schreib ein Programm, welches die 3 Werte eines Datums (Tag, Monat, Jahr) im Format TT.MM.JJJJ auf einmal einliest und mittels der eingelesenen Werte im Format MM.TT.JJ wieder ausgibt.
4. Schreib ein Programm, das zwei short-Werte von der Tastatur einliest, die Summe der beiden Zahlen ermittelt und das Ergebnis am Bildschirm ausgibt. Prüfe mit diesem Programm den Wertebereichsüberlauf, falls das Ergebnis nicht mehr im jeweiligen Wertebereich liegt.

   Beispiel: 30000 und 10000 werden addiert → Ergebnis = -25536!

   Ändere den Datentyp der Ergebnisvariablen so ab, dass der Wertebereich passen müsste und prüfe nach, dass dies alleine an der Problematik noch nichts ändert! Achte auch auf den richtigen Formatbezeichner für die Ausgabe!
5. Schreib ein Programm, welches eine Temperatur in °Celsius einliest und diese in °Fahrenheit umgerechnet am Bildschirm ausgibt. Das Ergebnisses soll mit 3 Kommastellen ausgegeben werden.

   Umrechnungsformel:

   $$\text{fahr} = ((\text{celsius} * 9) / 5) + 32$$

   Verwende zunächst als Eingabe einen ganzzahligen Wert und prüfe die Ergebnisse mittels Taschenrechner nach (Problematik der ganzzahligen Division!). Änder die Eingabe dahingehend, dass eine Kommazahl eingegeben wird.

   Schreibe auch die inverse Variante, so dass eine Temperatur in °Fahrenheit eingegeben wird und diese in °Celsius umgerechnet wird.

Kapitel 4

Steueranweisung

4.1 Übersicht

Es gibt in C/C++ folgende Arten von Steueranweisungen:

- if-, if/else-Anweisung
- switch/case-Anweisung
- while-, do/while- und for-Anweisung

4.1.1 if-, if/else-Anweisung

if-Anweisung:

Beispiel:

```
if (x<y)
{
        printf("x ist kleiner als y\n");
}
```

Listing 4.1: *if*-Anweisung

```
if (Bedingung)
        Anweisung
```

Der Ausdruck (Bedingung) wird ausgewertet. Wenn das Ergebnis $\neq 0$ ist, dann wird es als logisch wahr interpretiert, und die Anweisung wird ausgeführt, an sonst wird die Anweisung nicht durchgeführt.

if/else-Anwewisung:

Beispiel 1:

```
if (x<y)
{
        printf("x ist kleiner als y\n");
```

```
}
else
{
        printf("y ist kleiner als x\n");
}
```

Listing 4.2: *if/else*-Anweisung

```
if (Bedingung)
        Anweisung
else
        Anweisung
```

Wenn die Auswertung des Ausdrucks einen Wert $\neq 0$ (d.h. true) ergibt, dann wird die auf die *if*-Bedingung folgende Anweisung, sonst wird die auf *else* folgende Anweisung durchgeführt.

Beispiel 2:

```
int zahl1, zahl2, zahl3;

printf("Zahl1=");
scanf("%i",&zahl1);
fflush(stdin);

printf("Zahl2=");
scanf("%i",&zahl2);
fflush(stdin);

printf("Zahl3=");
scanf("%i",&zahl3);
fflush(stdin);

if (zahl1 < zahl2 && zahl1 < zahl3)
{
        printf("Zahl1 ist die kleinste Zahl\n");
}
else if (zahl2 < zahl1 && zahl2 < zahl3)
{
        printf("Zahl2 ist die kleinste Zahl\n");
}
else
{
        printf("Zahl3 ist die kleinste Zahl\n");
}
```

Listing 4.3: Mehrfachverzweigung

```
if (Bedingung)
        Anweisung
else if (Bedingung)
        Anweisung
else
```

Anweisung

Die Anweisung im *else*-Teil einer *if/else*-Anweisung kann natürlich auch wieder eine *if/else*-Anweisung sein, was dann zu Mehrfachverzweigungen wie in Listing 4.3 führt. Das Beispiel zeigt auch, wie man durch logische Verknüpfung in der Bedingung die Anzahl der Verzweigungen reduzieren kann.

4.1.2 switch/case-Anweisung

Beispiel:

```
switch (art)
{
        case 0:
                ...........
                break;
        case 1:
                ...........
                break;
        case 2:
                ...........
                break;
        case 3:
                ...........
                break;
        case 4:
        case 5:
                ...........
                break;
        default:
                printf("falscher Wert!\n");
                break;
}
```

Listing 4.4: switch-/case-Anweisung

Bedeutung: Die switch-Anweisung ist speziell zur Formulierung von Mehrfachverzweigungen vorgesehen. Die allgemeine Syntax lässt viele Konstrukte zu, die übliche Verwendung enthält aber im Rumpf nur Anweisungen oder Anweisungsfolgen, die jeweils durch eine "case-Marke" eingeleitet werden, sowie maximal eine Anweisung oder Anweisungsfolge, die durch eine "default-Marke" eingeleitet wird.

Der Ausdruck wird ausgewertet, und es wird geprüft, ob eine dem errechneten Wert entsprechende Sprungmarke vorhanden ist. Ist das der Fall, dann wird dorthin verzweigt, ist das nicht der Fall, dann wird - falls vorhanden - zum "default-Teil" verzweigt. Wenn weder eine entsprechende Sprungmarke explizit angegeben, noch ein "default-Teil" vorhanden ist, dann wird keine Anweisung durchgeführt!

Die "case-Teile" werden üblicherweise mit einer break- Anweisung abgeschlossen, die die switch-Anweisung an der Stelle beenden. Wenn das "break" entfällt, dann wird sequentiell mit dem folgenden "case-Teil" oder mit dem "default-Teil" weitergemacht.

Der Ausdruck muss einen ganzzahligen Typ liefern (char, int, ... oder enum), die auf "case" folgenden Marken müssen ganzzahlige Konstanten sein. Mit der switch-Anweisung kann natürlich auch die *if/else*-Anweisung simuliert werden, z.B.:

```
switch (i < j)
{
        case 1:
                printf("kleiner\n");
                break;
        case 0:
                printf("größer\n");
                break;
}
```

Listing 4.5: if/else-Anweisung mit switch-/case-Anweisung realisiert

4.1.3 Wiederholungen

while-Anweisung:

```
int sum=0;
int count=1;

while(count <= 100)
{
        sum+=count;
        count++;
}
```

Listing 4.6: Beispiel: while-Anweisung

```
while(Bedingung)
        Anweisung
```

Die Anweisung wird so lange wiederholt, wie die Auswertung des Ausdrucks einen Wert zngleich Null (d.h. wahr) ergibt. Hat der Ausdruck den Anfangswert Null (d.h. falsch), so wird die Anweisung gar nicht ausgeführt. Dementsprechend ist **while**(1) ... also eine Endlosschleife (repeat forever)!

do/while-Anweisung:

```
int sum=0;
int count=1;

do
{
        sum+=count;
        count++;
} while(count <= 101);
```

Listing 4.7: Beispiel: do/while-Anweisung

```
do
        Anweisung
while(Bedingung)
```

Die Anweisung wird so lange wiederholt, wie die Auswertung des Ausdrucks einen Wert zngleich Null (d.h. wahr) ergibt. Im Unterschied zur *while*-Anweisung erfolgt die Auswertung nicht zu Beginn, sondern zum Schluss, und somit wird die Anweisung mindestens einmal ausgeführt.

Dementsprechend ist **do** ... **while**(1) ebenfalls eine Endlosschleife.

for-Anweisung:

Beispiel 1:

```
int sum=0;
int count;

for(count = 1; count <= 100; count++)
{
        sum+=count;
}
```

Listing 4.8: Beispiel 1: for-Anweisung

Beispiel 2:

```
int sum=0;
int i,j;

for(i = 1,j = 100; i <= 50 && j > 50; i++,j--)
{
        sum=sum + i + j;
}
```

Listing 4.9: Beispiel 2: for-Anweisung

```
for(Initialisierung; Bedingung; Operation)
        Anweisung
```

Im ersten Ausdruck wird die Initialisierung spezifiziert, im zweiten wird die Bedingung spezifiziert, die erfüllt sein muss, bevor eine weitere Iteration durchgeführt wird, und im dritten Ausdruck wird eien Operation spezifiziert, die bei jeder Iteration durchgeführt wird.

Wie das Beispiel 2 demonstriert, können die Ausdrücke in der for- Anweisung sich aus jeweils mehreren durch Kommata getrennten Teilausdrücken zusammensetzen. Das Beispiel kann wie folgt durch Verwendung einer *while*-Anweisung nachgebildet werden:

```
int sum=0,i=0,j=100;

while(i <= 50 && j > 50)
{
        sum=sum + i + j;
        i++;
        j++;
}
```

Dementsprechend sind **for** (;1;) ... und auch **for** (;;) ... weitere Formulierungen für Endlosschleifen.

Bezüglich der Entscheidung, welche der drei Anweisungen while, do/while oder for jeweils zur Formulierung einer bestimmten Iteration zu verwenden ist, gibt es unterschiedliche Ansichten. Puristen verwenden stets die sichere while-Anweisung, typische C-Programmierer alter Schule haben eine ausgeprägte Vorliebe für die mächtige for-Anweisung. Sinnvoll ist sicher eine differenzierte Auswahl:

Verwendung der *while*-Anweisung, wenn die Anzahl der Iterationen n unbekannt ist mit $n \geq 0$ (auch null Durchläufe möglich!), Verwendung der *do/while*-Anweisung, wenn im Gegensatz dazu $n \geq 1$ (mindestens ein Durchlauf!), und Bevorzugung der *for*-Schleife insbesondere dann, wenn die Anzahl der Iterationen schon bekannt ist.

4.2 Übungen zu Schleifen

1. Schreibe ein Programm, dass mittels einer for-Schleife exakt folgende Bildschirmausgabe erzeugt: 1,2,3,4,5,6,7,8,9,10
2. Schreibe die Übung 1 so um, dass die Start- und Stoppwert eingegeben werden kann. Die Ausgabe soll dabei auch den Start und den Stoppwert beinhalten. Wenn Start- und Stoppwert identisch sind, so soll nur eine Zahl ausgegeben werden!
3. Schreibe die Übung 2 so um, dass von einem größeren Startwert zu einem kleineren Stoppwert heruntergezählt wird.
4. Schreibe die Übung 2 so um, dass anstelle der for-Schleife eine while-Schleife bzw. eine do-while-Schleife verwendet wird.
5. Schaue dir die folgenden Schleifenkonstrukte an und erkläre, was sie machen:
   (a) `for(x=1;x<100;x++);`
   (b) `for(x=2;x>1;x++);`
   (c) `for(x=1;x<20;);`
   (d) `for (;1;);`
   (e) `while(1);`
   (f)
   ```
   do
   {}while(0);
   ```
   (g) `for(;x<=8;x++) x--;`
6. Welche Ausgabe produziert das folgende Programm? Ändere das Programm so, dass die erwartet Ausgabe erfolgt.

```
#include <stdio.h>

void main()
{
  int i;

  for(i=0;i<5;i++);
  {
    printf("Hallo Welt!\n");
  }
}
```

7. Welche Ausgabe produziert das folgende Programm?

```
#include <stdio.h>

void main()
{
  int x,y;
```

```
    y=2;
    for(x=0;x<y;x++)
    {
       printf("Hallo Welt!\n");
    }
}
```

8. Was passiert, wenn in dem Programm aus der Aufgabe zuvor y auf den Wert 0 gesetzt wird? Wie oft wird die Schleife durchlaufen?

9. Schreibe ein Programm, dass eine Folge von positiven Zahlen einliest. Mit der Zahl 0 wird die Eingabe beendet. Anschließend soll die Anzahl und die Summe der eingelesenen Zahlen ausgegeben werden.

10. Schreibe ein Programm, welches mittels einer Schleife folgende Zahlenfolge erzeugt: 2 4 6 8 10 12 14 16 18 20

11. Schreibe ein Programm, welches mittels einer Schleife folgende Zahlenfolge erzeugt: 1 1 2 4 7 11 16 22 29 37

12. Schreibe ein einfaches Zahlenratespiel, wobei der Benutzer den Zahlenbereich und die maximale Anzahl der Versuche selbst einstellen kann.

    Es soll ein einfaches Menü mit Hilfe der switch-case-Anweisung realisiert werden:

    Aktuelle Einstellungen: Zahlenbereich 0...10, Versuche: 3
    1...Spiel starten
    2...Zahlenbereich eingeben
    3...Anzahl der Versuche eingeben
    4...Ende

    Der Benutzer soll immer wieder zu diesem Menü zurückkehren. Durch Eingabe der entsprechenden Zahl wird der jeweilige Menüpunkt ausgeführt.

    Für Fortgeschrittene:

13. Schreibe ein Programm, dass die sogenannten Fibonacci-Zahlen bis zu einer einzugebenden Höchstgrenze am Bildschirm ausgibt. Eine Fibonacci-Zahl wird als Summe der beiden vorhergehenden Fibonacci-Zahlen gebildet. Die erste und die zweite Fibonacci-Zahl sind gleich 1.

    Zahlenfolge: 1 1 2 3 5 8 13 ...

14. Gib in einer **for**−Schleife die Buchstaben ‘a’ bis ‘z’ aus. Verwende dabei die Funktion putchar().

    Beispiel:

```
char Buchstabe;

printf("Zeichen: ");
scanf("%c",&Buchstabe);
fflush(stdin);

putchar(Buchstabe);
```

15. Worin besteht der Unterschied zwischen **for**(i=0;i<10;i++) und **for**(i=0;i<10;++i)

16. Schreibe ein Programm, das beliebig viele Zeichen von der Tastatur einliest und gleichzeitig wieder ausgibt. die Schleife soll erst beendet werden, wenn ein großes 'X' eingegeben worden ist.

17. Folgende Ausgabe soll erzeugt werden: `1 3 5 7 9`. Löse die Aufgabe

    (a) mit Hilfe einer **while**−Schleife

    (b) mit Hilfe einer **do**−**while**−Schleife

    (c) mit Hilfe einer **for**−Schleife und des Mudulo-Operators

    (d) mit Hilfe einer **for**−Schleife ohne den Modulo-Operators.

18. Folgende Ausgabe soll erzeugt werden: `1 2 4 7 11 16 22 29 37`. Löse die Aufgabe

    (a) mit Hilfe einer **do**−**while**−Schleife

    (b) mit Hilfe einer **for**−Schleife.

4.3 Geschicklichkeitsspiel

1. Schreibe ein Programm, mit welchem ein beliebiges Textzeichen, das sich von selbst fortbewegt, am Bildschirm gesteuert werden kann (z.B. mit den Pfeiltasten). Wenn das Zeichen auf einer Seite aus dem Bildschirm fährt, so soll es von der anderen Seite wieder hereinkommen.
2. Positioniere eine gewisse Anzahl von Zeichen auf zufälligen Positionen als Hindernisse auf dem Bildschirm, welche umfahren werden müssen. Falls ein solches Zeichen getroffen wird, ist das Spiel zu Ende.
3. Positioniere eine gewisse Anzahl von Zeichen auf zufälligen Positionen auf dem Bildschirm, welche gefressen werden sollen. Für jedes gefressene Zeichen wird ein Zähler inkrementiert. Ein gefressenes Zeichen verschwindet vom Bildschirm und taucht auf einer zufälligen Position wieder auf (in einem bestimmten Mindestabstand zur aktuellen Position des zu steuernden Zeichens!).

Anmerkung:

- Füge die beiden Dateien HTL_Konsole.h sowie HTL_Konsole.cpp deinem Projekt hinzu.
- Tastenabfrage

```
// Prüfung, ob irgend eine Taste gedrückt wurde
if (kbhit())
{
  //Taste einlesen
  taste=getch();

  // prüfen,ob Taste mit erweiterten Tastencode
  if (taste == 224)
  {
    // erweiterten Tastencode einlesen
    taste == getch();

    // Tastencode auswerten
    switch(taste)
    {
      // Pfeiltaste nach oben gedrückt
      case UP:
        ....
        break;
    }
  }
```

Erweiterte Tastencodes für Pfeiltasten:

```
#define UP      72
#define DOWN    80
#define LEFT    75
#define RIGHT   77
```

- Bildschirmsteuerung:
  Der Bildschirm besteht aus 80x25 Textzeichen, wobei die Positionierung mit der Funktion gotoxy() erfolgt.

  Beispiel: gotoxy(10,12);... Setzt den Cursor an die Position x=10 und y=12!

  ACHTUNG: Das Koordinatensystem startet in der linken oberen Bildschirmecke mit x=1 und y=1!

- Geschwindigkeit:
  Variiere die Geschwindigkeit durch den Einbau einer Zeitverzögerung mittels der Funktion Sleep() bzw. einer entsprechende lang dauernden Verzögerungsschleife.

  Beispiel: Sleep(30).... Programm bleibt für 30ms stehen!

  Die Geschwindigkeit des zu steuernden Zeichens soll mit steigendem Zählerstand langsam vergrößert werden.

Kapitel 5

Funktionen

5.1 Grundlagen

Unterprogramme heißen in C Funktionen; und zwar gibt es zwei Varianten von C-Funktionen:

- Echte Funktionen, die ihr Ergebnis über ihren Funktionsnamen zurückgeben
- Anweisungs-Funktionen, die Ergebnisse ggf. über Parameter zurückgeben, aber kein Ergebnis über ihren Funktionsnamen.

```
int Summe(int a, int b)
{
  int i, sum;

  if((a < 0) || (b < 0) || (a > b))
    return -1;

  sum=0;
  for(i=a; i < b; i++)
    sum+=i;

  return sum;
}
```

Der Aufruf erfolgt z.B. in der Form:

```
void main()
{
  int s,start, ende;

  printf("Startwert: ");
  scanf("%i",&start);
  fflush(stdin);

  printf("Endwert: ");
```

```
  scanf("%i",&ende);
  fflush(stdin);

  s=Summe(start,ende);

  printf("Summe = %i\n",s);
}
```

Beispiel für eine Anweisungs-Funktion:

```
int Summe(int a, int b, int & sum)
{
  int i;

  if((a < 0) || (b < 0) || (a > b))
  {
    sum=-1;
    return;
  }

  sum=0;
  for(i=a; i < b; i++)
    sum+=i;
}
```

Der Aufruf erfolgt z.B. in der Form:

```
void main()
{
  int s,start, ende;

  printf("Startwert: ");
  scanf("%i",&start);
  fflush(stdin);
  printf("Endwert: ");
  scanf("%i",&ende);
  fflush(stdin);

  Summe(start,ende,s);

  printf("Summe = %i\n",s);
}
```

Der Parameter &sum ist im Gegensatz zu den Werteparameter (**"Called by Value"**) a und b ein Parameter, der **"Called by Refence"** bezeichnet wird. Der Unterschied zwischen "Called by Value"-Parameter und "Called by Reference"-Parameter ist, dass er ein Ergebnis an den aufrufenden Programmteil zurückgeben kann. Genau genommen gibt es in C nur "Called by Value"-Parameter, und der Mechanismus "Called by Refernce" wird dadurch simuliert, dass man nicht den Wert selbst, sondern seine Adresse übergibt.

Die Angabe **void** heisst leer und besagt, dass die Funktion selbst keinen Wert zurückgibt.

5.1.1 Prototypen

Eine Funktion muss in C bei ihrem Aufruf bezüglich ihrer Schnittstelle bekannt sein, damit der Compiler den Aufruf auf korrekte Syntax überprüfen kann. Dafür gibt es zwei Möglichkeiten:

- **entweder** wird die Funktion vor ihrem Aufruf *definiert*, d.h. implementiert (d.h. vollständig realisiert),
- **oder** sie wird in Form eines sogenannten *Prototypen4* vorher *deklariert*

Der Prototyp umfasst den Funktionskopf und damit die Schnittstelle der Funktion. Das folgende Beispiel zeigt einen entsprechenden Programmtext:

```
int Summe(int a, int b);        //Prototyp der Funktion

void main()
{
  int s;
  ....
  s = Summe(1,100);
  ....
}

int Summe(int a, int b)
{
  int i, sum;

  if((a < 0) || (b < 0) || (a > b))
    return -1;

  sum=0;
  for(i=a; i < b; i++)
    sum+=i;

  return sum;
}
```

Sonstiges:

- Das Hauptprogramm **void** main() ist ebenfalls eine Funktion (aus der Sicht des Betriebssystems), sie hat in dem obigen Beispiel keine Parameter und liefert keinen Wert zurück.
- Funktionen können nicht statisch geschachtelt werden, d.h. eine Funktion kann nicht innerhalb einer anderen Funktion definiert werden, und das bedeutet wiederum, dass alle Funktionen auf der gleichen Ebene liegen.
- Rekursionen sind möglich, d.h. Funktionen können sich selbst direkt oder indirekt aufrufen.

5.1.2 Variable Anzahl von Parametern

In C gibtes die Möglichkeit, Funktionen mit variabler Parameterzahl auch selbst zu definieren. In der Standard-Bibliothek sind

int printf(**const char**∗ format, ...) und
int scanf(**const char**∗ format, ...)

die bekanntesten Vertreter von Funktionen mit variabler Parameterliste. Aus der Interpretation der Formatstrings in der Implementierung dieser Funktionen ergibt sich die Anzahl der relevanten zusätzlichen Argumente: zu jeder mit % eingeleiteten Formatspezifikation gehört ein Argument.Ein etwas einfacheres Beispiel zeigt das Prinzip:

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

int summe(int count, ...);

void main()
{
  printf("%i\n", summe(3,11,22,33));
}

int summe(int count, ...)
{
  int s=0;

  va_list ap;
  va_start(ap,count);

  while(count > 0)
  {
    s=s+va_arg(ap,int);
    count--;
  }

  va_end(sp);
  return s;
}
```

Die erforderlichen Hilfsmittel - der Typ va_list sowie die drei Makros va_start, va_arg und va_end - werden über die Definitionsdtei stdarg.h zur Verfügung gestellt:

Zunächst wird mit va_list ap; ein Zeiger (argument pointer) definiert und mit va_start(ap,count); auf das erste Argument der variablen Parameterliste gesetzt. Bei jedem Schleifendurchlauf wird der Wert des Arguments gemäß dem als zweiten Parameter angegebenen Typs geliefert, und der Zeiger ap wird auf das nächste Argument gesetzt. Mit va_end(ap); werden erforderliche Abschlussaktionen durchgeführen.

5.1.3 Definitionen, Deklaration und Aufruf von Funktionen

Eine typische Funktionsdefinition gemäss dem C-Standard sieht wie folgt aus:

```
int f(int a, int b)
```

```
{
  return (a+b)*(a+b);
}
```

Funktionsdefinition

Die Definition einer C-Funktion sieht wie folgt aus:

Rückgabetype *Funktionsname*(Parameterliste)

Bedeutung:

- Rückgabetyp: umfasst die Spezifikation des Rückgabetyps (z.B. **int**, **float**, **double**, **char**, **void**)
- Funktionsname: dieser Name muss eindeutig sein
- Parameterliste: enthält eine oder mehrere Variablen-Deklarationen und ggf. ... am Ende der Liste, wobei die Elemente der Liste durch Kommata getrennt sind. Eine leere Parametrliste wird durch **void** bezeichnet.

Der Rückgabetyp einer Funktion kann auch eine C-Struktur oder eine C-Union sein.

Funktionsdeklaration (Funktionsprototyp)

Die Deklaration einer C-Funktion, der Funktionsprototyp, ist in erster Näherung mit dem Kopf der Funktionsdefinition und einem abschließenden Semikolon anstelle des Funktionsrumpfes identisch. Ein wesentlicher Unterschied besteht darin, dass in der Parameterliste des Prototyps auf die Namen der Objekte verzichtet werden **kann**, also z.B.:

```
int f(int, int);
```

oder

```
int f(int a, int b);
```

Der Prototyp dient ja nur dazu, dem Compiler die Kontrolle der richtigen Verwendung der Funktion zu ermöglichen, und dafür benötigt er jeweils nur die Parametertypen, aber nicht ihre Namen. Trotzdem ist es im allgemeinen sinnvoll, auch bei den Prototypen die Parameternamen anzugeben.

Funktionsaufruf:

```
x = f(3,2);

y = f(a,b);
```

5.1.4 C-Bibliotheksfunktionen

Der Gesamtstandard ANSI-C beinhaltet neben der eigentlichen Sprache noch eine Standard-Bibliothek, bestehend aus fünfzehn Bibliotheksmodulen. Wesentliche Module dieser Biblothek sind:

- <stdio.h>: Eingabe- und Ausgabe-Funktionen
- <string.h>: Textbearbeitungsfunktionen

- <math.h>: mathematische Funktionen
- <stddef.h>: Allgemeine Definitionen
- <stdlib.h>: Hilfsfunktionen

In diesen Modulen stehen entsprechende Funktionsdeklarationen (Prototypen) sowie andere Deklarationen und Definitionen, die durch entsprechende #**include**-Anweisungen in Programme eingebunden werden und damit dort anwendbar sind.

5.1.5 Funktionsaufruf und Argumentübergabe

5.1.5.1 "call by value"

Bei der Übergabe mit "call by value" wird eine Kopie der Werte an die Funktion übergeben. Das stellt sicher, dass die Funktion das Original nicht ändern kann.

Beispiel 1:

```
int mult(int a, int b);

void main()
{
        int x=3;
        int y=2;
        int c;

        c=mult(x,y);
}

int mult(int a, int b)
{
        a=2*a;
        return a*b;
}
```

Welche Werte stehen in x, y und c nach dem Funktionsaufruf?
x=3, y=2 und c=12. Warum?

Beispiel 2:

```
void swap(int a, int b);

void main()
{
        int x=3;
        int y=2;

        swap(x,y);
}
```

```
void swap(int a, int b)
{
        int h=a;

        a=b;
        b=h;
}
```

Welche Werte stehen in a und b nach dem Funktionsaufruf?
x=3 und y=2. Warum?

5.1.5.2 "call by refernce"

Die Wertübergabe kann aus verschiedenen Gründen unerwünscht ein. Ein Hauptgrund: Eine Funktion kann zwar beliebig viele Parameter übernehmen, aber nur genau einen Wert zurückgeben (oder keinen Wert, wenn **void** gesetzt ist).

Sollen mehrere Werte übermittelt werden, kann man alternativ mit den Adressen der Variablen arbeiten, also "call by reference". In diesem Fall werden die Werte der Originale innerhalb der Funktion verändert, sodass die veränderten Werte auch nach Abarbeitung der Funktion zur Verfügung stehen (an der Stelle im Programm, die die Funktion aufgerufen hat).

Beispiel:

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

void main()
{
        int x=3;
        int y=2;

        swap(x,y);
}

void swap(int &a, int &b)
{
        int h=a;

        a=b;
        b=h;
}
```

Welche Werte stehen in x und y nach dem Funktionsaufruf?
x=2 und y=3. Warum?

5.1.5.3 Felder als Argumente

Wenn in einer Funktion ein Feld als Parameter verwendet wird, nimmt C/C++ eine Standardkonversion in einen Zeiger auf die erste Feldkomponente vor. Argumente des Typs **int** arr [] werden also implizit in

den Typ **int** ∗arr umgewandelt, und die Adresse der ersten Feldkomponente wird als Wert übergeben. Dies bedeutet, dass die Funktion nicht mit eiiner lokalen Kopie des aktuellen Feldarguments arbeitet, sondern dass mit Veränderungen der Feldkomponenten über den Funktionsaufruf hinaus wirksam sind ("call by reference").

```
#define N 10

void IntArrayIn(int werte[], int anzahl);

void main()
{
        int w[N];

        IntArrayIn(w,N);

        ...
}

int IntArrayIn(int werte[], int anzahl)
{
        int i;

        for(i = 0 ; i < anzahl; i++)
        {
                printf("%i. Zahl= ",i+1);
                scanf("%i",&werte[i];)
                fflush(stdin);
        }
}
```

5.1.5.4 const Parameter

Wenn ein Funktionsparameter **const** ist, wird im Rumpf der Funktionsdefinition, also bei der Implementation der Funktion, jedoch kontrolliert, dass er nicht modifiziert wird. Beispielsweise wird das folgende nicht übersetzt:

```
int h(const int i)
{
        return ++i;
}
```

5.2 Übungen zu Funktionen

1. Schreibe eine Funktion **int** Sub(**int** a, **int** b), welcher 2 int-Werte übergeben werden und welche die Differenz der beiden Werte als int-Wert zurück liefert.
2. Schreibe eine Funktion **int** Max(**int** a, **int** b), welcher 2 int-Werte übergeben werden und welche den größeren der beiden Werte zurück liefert.
3. Schreibe eine Funktionen zur Umrechnung von Temperaturen zwischen Fahrenheit und Celsius.
4. Schreibe eine Funktion, die ein Datum auf Gültigkeit prüft. Tag, Monat und Jahr werden als Integer-Werte übergeben. Die Funktion liefert 1 zurück, falls es sich um ein gültiges Datum handelt, 0 sonst. Achte auch auf Schaltjahre! siehe Kapizel 2.8
5. Schreibe eine Funktion NSum() zur Ermittlung der Summe der ersten n natürlichen Zahlen (1...n), n ist der Übergabeparameter.
6. Schreibe eine Funktion Potenz(), der zwei ganzzahlige Werte a und b übergeben werden und welche dann a hoch b berechnet und zurück liefert.
7. Schreibe eine Funktion IsPrim(), der ein ganzzahliger Wert > 1 übergeben wird und welche 1 zurück liefert, falls die übergebene Zahl eine Primzahl ist und 0 sonst.
8. Schreibe eine Funktion Swap(), der 2 Integer-Variablen übergeben werden und die die Werte der beiden Variablen vertauscht. Dies soll sich natürlich auf die Originalwerte auswirken.
9. Schreibe eine Funktion Line() zum Zeichnen einer horizontalen bzw. vertikalen Linie am Textbildschirm. Übergeben werden x,y-Positionen von Start- und Endpunkt.
   ANMERKUNG: Das Zeichnen einer Linie zwischen 2 beliebigen Punkten (auch diagonal) ist eine ungleich schwierigere Aufgabe!
10. Schreibe eine Funktion Rectangle() zum Zeichnen eines Rechtecks am Textbildschirm. Übergeben werden die x,y-Position der linken oberen Ecke sowie die x,y-Position der rechten unteren Ecke. Verwende die vorhandene Funktion Line()!
11. Schreibe das Programm Zahlenratespiel entsprechend um, so dass für alle Menüpunkte Funktionen verwendet werden.
12. Versuche das Programm Geschicklichkeitsspiel sinnvoll in einzelne Teilaufgaben zu unterteilen, für welche Funktionen verwendet werden. Beispiele für Teilaufgaben:
    - Tastaturabfrage
    - Positionsprüfungen
    - Kollisionsabfragen
    - ...

Anmerkungen:

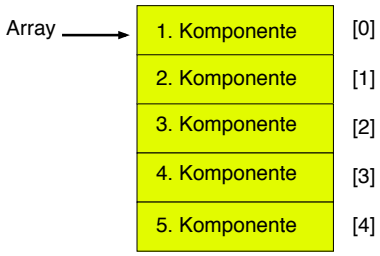
- Verwende nirgends globale Variablen
- Teste alle Funktionen innerhalb der Funktion **void** main( ) aus
- Es dürfen nur selbstgeschriebene Funktionen verwendet werden

Kapitel 6

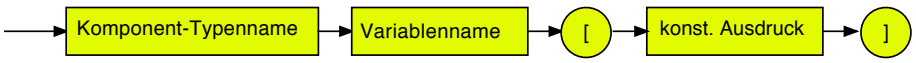
eindimensionale Arrays

6.1 Eindimensionale Arrays in C

- Ein Array (Vektor) ist eine Datenstruktur (= zusammengesetzter Datentyp), deren Elemente (= Komponenten) alle vom gleichen Typ (= Basistyp) sind.
- Die einzelnen Elemente sind in einer festliegenden Reihenfolge angeordnet und werden über einen Index angesprochen.
- C kennt nur Arrays feststehender - d.h. zur Compile-Zeit ermittelbarer - Länge. Die Abspeicherung eines Arrays erfolgt in einem zusammenhängenden Speicherblock; die einzelnen Array-Elemente sind in aufeinanderfolgenden Speicherplätzen abgelegt.
  Beipiel : Array mit 5 Komponenten



- Array-Variablenvereinbarung
  Festlegung von
  - Name der Array-Variablen
  - Typ der Array-Komponenten (Basistyp)
  - Anzahl der Array-Komponenten (= Länge des Arrays), (nur bei Definition notwendig)
    Definition:



    Beispiele :

```
* char kette[50];

  int zfeld[10];

* #define ANZ 80

  char k1[ANZ], k2[2*ANZ];
```

- Die Indexgrenzen sind in C nicht frei wählbar.

  Untere Grenze: 0
  Obere Grenze : Komponentenzahl - 1
  Beispiel : `float a[5]` → Indexbereich 0 .. 4

- Zugriff zu einer Array-Komponente :

  Angabe des Index (als int-Ausdruck) in eckigen Klammern nach dem Array- Variablennamen.

  Beispiele :

  a[3], a[i+2*j]

- Array-Variable können in einer Vereinbarung gemischt mit einfachen Variablen vereinbart werden, wenn die Array-Komponenten vom gleichen Typ wie die einfachen Variablen sind.

  Beispiel :

  int zfeld [10], laenge;

- Die Längenangabe bei Array-Vereinbarungen darf fehlen :

  - bei der Definition initialisierter Arrays
  - in Parameterdeklarationen
  - in Array-Variablen-Deklarationen

- Achtung !

  Es gibt keine Laufzeitüberprüfung auf Überschreitung des max. Array- Index.

  Beispiel :

```
int f[10];
...
f[10] = ... ;
// Fehler, der aber nicht als Laufzeitfehler angezeigt wird.
```

- Geschlossene Wertzuweisungen zwischen ganzen Arrays sind nicht zulässig.

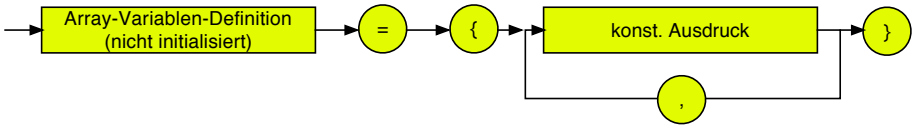
  Beispiel :

```
int feld1[10], feld2[10];
...
feld1 = feld2;      /* Unzulässig, Compilerfehler ! */
```

6.1.1 Initialisierung von Arrays

- In C können auch Arrays bei der Definition initialisiert werden.
- Syntax :

  Die Initialisierungswerte für die einzelnen Komponenten sind als konstante Ausdrücke anzugeben und durch Kommata getrennt in geschweiften Klammern zusammenzufassen



- Beispiel :

```
#define AWERT 5

int ifeld[4] = { 2, AWERT+3, 2*AWERT, -8};
```

- Bei der Definition initialisierter Arrays kann die Angabe der Array- Komponentenzahl (Arraylänge) weggelassen werden:
- Das Array bekommt genau soviel Komponenten, wie Initialisierungswerte angegeben werden.

  Wird bei der Definition initialisierter Arrays die Arraylänge angegeben und werden weniger Initialisierungswerte als es der Arraylänge entspricht angegeben, so werden die ersten Komponenten mit den angegebenen Werten und die restlichen Komponenten mit 0 initialisiert.

  Beispiel :

```
int ifeld[5] = {1, 2, 3};

ifeld[0] = 1;
ifeld[1] = 2;
ifeld[2] = 3;
ifeld[3] = ifeld[4] = 0;
```

- Die Angabe von mehr Initialisierungswerten als der Arraylänge entspricht ist ein Fehler!
- Soll ein Element in der Mitte des Arrays initialisiert werden, so müssen auch alle vorhergehenden Elemente initialisiert werden.
- Es gibt keine Möglichkeit zur einfachen Angabe einer wiederholten Initialisierung aufeinanderfolgender Komponenten mit gleichen Werten.
- Die Komponenten nichtinitialisierter Arrays
  - werden bei globalen und statisch-lokalen Arrays defaultmäßig mit 0 initialisiert,
  - bzw sind bei lokalen Arrays der Speicherklasse auto undefiniert
- Programmbeispiel zur Initialisierung von Arrays in C

```
// ------------------------------------------------
// Programm INITAR1
// ------------------------------------------------
// Demonstration der Initialisierung
// von Arrays
// ------------------------------------------------

#include <stdio.h>

#define ANZ 4


/* Funktionsdeklaration */
void aus_array(float feld[], int laenge);

void main()
{
  double a[]   = {3.5, -7.2, -4.3, 1.8};
  double b[ANZ] = {2.9, 1.3, 5.1};
  double c[ANZ];
  int i;

  for (i=0; i<ANZ; i++)
    c[i]=a[i]+b[i];

  aus_array(c,ANZ);
}

 /* Funktionsdefinition */
void aus_array(double feld[], int laenge)
{
  int i;

  printf(''\n');

  for (i=0; i<(laenge-1); i++)
    printf(''%5.2f, '',feld[i]);

  printf(''%5.2f \n'',feld[laenge-1]);
}

// ------------------------------------------------
```

6.2 Übungen zu eindimensionalen Arrays

1. Schreibe ein Programm, welches eine Folge von 10 Zahlen einliest und anschließend in umgekehrter Reihenfolge wieder ausgibt.
2. Schreibe eine Funktion IntArrayIn() für die Werteingabe in ein Integer-Array. Übergabeparameter sind das Array sowie die Anzahl der Elemente. Die Funktion liest die Werte von der Tastatur ein und speichert Sie in das Array.
3. Schreibe eine Funktion IntArrayOut(), die die gleichen Parameter wie IntArrayInt() besitzt und die Arrayelemente untereinander auf dem Bildschirm ausgibt. Die Arrayelemente sollen in der Funktion nicht verändert werden können!
4. Schreibe die Funktionen IntArraySum(), IntArrayMax() und IntArrayPosMin(). Welche die Summe, das Maximum sowie die Position des kleinsten Elements eines Integer-Arrays bilden und zurückgeben. Übergabeparameter sind das Array sowie die Anzahl der Elemente. Die Arrayelemente sollen in allen Funktionen nicht verändert werden können!
5. Schreibe ein Programm zur Ermittlung einer Testnotenstatistik.
6. Erweitern Sie das Geschicklichkeitsspiel von Übungsblatt 4 in der folgenden Weise:
   - Es soll eine Schlange über den Bildschirm bewegt werden (gesteuert wird nur der Kopf)
   - Mit jedem gefressenen Zeichen soll die Schlange länger werden!
   - Die Schlange darf sich nicht selbst fressen
   - Bei bestimmten gefressenen Zeichen kann die Schlange auch wieder schrumpfen
7. Für Fortgeschrittene: Schreiben Sie eine Funktion IntArraySort() zum Sortieren eines Integer-Arrays. Übergabeparameter sind das Array sowie die Anzahl der Elemente.

Aufgabe:

Für eine Schulklasse werden Testnoten eingegeben und daraufhin folgende Werte ermittelt:

- Notendurchschnitt
- Wieviele 1er, 2er, ...

Die Anzahl der Schüler soll zu Beginn des Programms eingegeben werden (maximal 36).

Folgende Funktionen sind erforderlich:

1. **Funktion zur Eingabe der Testnoten**
   1. Übergabeparameter: Array für Testnoten
   2. Übergabeparameter: Schüleranzahl
   Rückgabewert: keiner

   ANMERKUNG:
   Es sollen nur ganze Zahlen zwischen 1 und 5 eingegeben werden können !

2. **Funktion zur Ermittlung des Notendurchschnitts**
   1. Übergabeparameter: Array mit Testnoten
   2. Übergabeparameter: Schüleranzahl
   Rückgabewert: Notendurchschnitt (Kommazahl !)

3. **Funktion zur Ermittlung der Anzahl einer bestimmten Note**
   1. Übergabeparameter: Array mit Testnoten
   2. Übergabeparameter: Schüleranzahl
   3. Übergabeparameter: Gesuchte Note (1..5)
   Rückgabewert: Anzahl der Arrayelemente mit der gesuchten Note

4. **Funktion zur Ausgabe der Notenstatistik**
   1. Übergabeparameter: Notendurchschnitt
   2. Übergabeparameter: Anzahl der 1er
   3. Übergabeparameter: Anzahl der 2er
   4. Übergabeparameter: Anzahl der 3er
   5. Übergabeparameter: Anzahl der 4er
   6. Übergabeparameter: Anzahl der 5er
   Rückgabewert: keiner
   Diese Funktion gibt alle gesuchten Parameter aus

   Der Notendurchschnitt ist mit 2 Kommastellen auszugeben.

Anmerkungen:

- Deklariere für alle Funktionen Prototypen
- Es dürfen nur selbstgeschriebene Funktionen verwendet werden
- Teste alle Funktionen innerhalb der Funktion main() aus
- Verwenden Sie nirgends globale Variablen

Kapitel 7

Zeichenkette

Der Begriff Computer oder Rechner legt nahe, dass sich die Datenverarbeitung vornehmlich mit numerischen Problemen beschäftigt. Dies war in den Anfängen der Datenverarbeitung auch richtig. Heute ist aber sicherlich nur noch ein sehr geringer Teil der Computeranwendungen numerischer Natur. Die meisten Anwendungen beschäftigen sich mit Daten- und Informationsverarbeitung im weitesten Sinne. Von daher ist die Verarbeitung von Zeichen und Zeichenketten (Strings) von großer Bedeutung. Glücklicherweise haben wir eine Schriftsprache, die auf sehr wenigen Grundelementen (Buchstaben, Zeichen) basiert, so dass wir unsere Rechner mit einem überschaubaren Zeichensatz ausstatten können. Andere Schriftsysteme (z. ß. Japanisch oder Chinesisch) sind sehr viel komplexer angelegt, und entsprechend aufwendig ist dort die Speicherung und Verarbeitung von Texten.

Zeichenketten (Strings) sind Reihungen von Zeichen. Naheliegenderweise wird ein String in einem Array abgelegt. Um dem String Raum für Veränderungen zu lassen, ist der Array in der Regel um einiges größer als der eigentliche String. Um das Ende des Strings im Array zu markieren, wird der String durch ein Terminatorzeichen abgeschlossen. Als Terminator darf natürlich nur ein Zeichen verwendet werden, das ansonsten in einem String nicht vorkommen darf. In C dient die Null (eine richtige Null, nicht das Zeichen '0') als Kennung für das String-Ende:

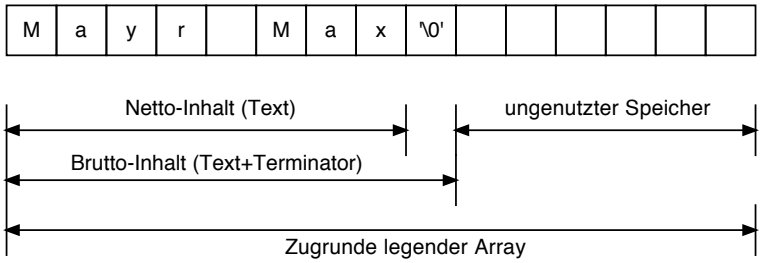


Abbildung 7.1: Speicherbelegung eines Strings

Grundsätzlich ist eine Zeichenkette also eine eindimensionaler Array von Zeichen (**char**) mit der zusätzlichen Eigenschaften, dass das letzte Zeichen NULL ist. Damit ist eigentlich alles gesagt. Wir können Strings mit den im letzten Abschnitt vorgestellten Array-Zugriff in jeder erdenklichen Weise bearbeiten. Beachte dabei aber die folgenden Warnung:

- Der String befindet sich in einem Array fester Länge. Sie müssen darauf achten, dass bei Manipu-

lationen des Strings (z.B. Anfügen von Buchstaben) die Grenzen des zugrunde liegenden Arrays nicht überschritten werden.

- Wegen des Terminators muss der Array, der den String beherbergt, mindestens ein Element mehr haben als der String Zeichen enthält.
- Der String muss nach eventuellen Manipulationen immer konsistent sein. Insbesondere bedeutet das, dass das Terminator-Zeichen korrekt positioniert werden muss.

Um einen String verwenden zu können, legt man zunächst einen Array an, der groß genug sein muss, um den zu erwartenden Text aufzunehmen:

```
void main()
{
  char Vorname[21];   // 20 Zeichen + Terminator
  char name[21];      // 20 Zeichen + Terminator
  ....
}
```

Zum Einlesen von Strings können wir die Funktion scanf verwenden. Die zugehörige Formatanweisung lautet %s.

```
void main()
{
  char vorname[21];   // 20 Zeichen + Terminator
  char name[21];      // 20 Zeichen + Terminator

  printf("Vorname: ");
  scanf("%s",vorname);
  fflush(stdin);
}
```

Achtung: Es wird dabei nicht geprüft, ob der Array groß genug ist, um den String aufzunehmen. Werden in unserem Beispiel mehr als 20 Zeichen eingegeben, so wird rücksichtslos außerhalb des Arrays geschrieben. Beachte auch, dass dem Variablennamen in diesem Fall kein & vorangestellt wird. Die Adressoperation ist hier überflüssig, da der Name eines Arrays in C automatisch wie die Adresse des ersten Elements verwendet wird. Diesen Zusammenhang werden wir im Abschnitt über Zeiger näher untersuchen.

Mit scanf können Sie über das Format %s nur einzelne Wörter jeweils bis zum nächsten Trennzeichen (Leerzeichen, Tabulator oder Zeilenumbruch) einlesen. Soll ein kompletter Text gegebenenfalls mit Leerzeichen in einen Array eingelesen werden, so verwende die Funktion gets, die das Einlesen der gesamten Eingabe bis zum Zeilenende ermöglicht:

```
void main()
{
  char vorname[21];   // 20 Zeichen + Terminator
  char name[21];      // 20 Zeichen + Terminator
  char adresse[100];

  printf("Vorname und Nachname: ");
  scanf("%s %s",vorname,name);
  fflush(stdin);
```

```
  printf("Adresse: ");
  gets(adresse);
  fflush(stdin);
}
```

Aus dem jeweils ersten Buchstaben des Vornamens und des Nachnamens stellen wir in unserem Beispiel jetzt ein Kürzel zusammen. Das Kürzel ist also ein 2-buchstabiger String, für den wir einen Array mit 3 Elementen reservieren müssen. Anschließend kopieren wir die gewünschten Buchstaben aus Name und Vorname in das Kürzel und ergänzen das Terminatorzeichen, damit das Kürzel zu einem korrekten String wird:

```
void main()
{
  char vorname[21];   // 20 Zeichen + Terminator
  char name[21];      // 20 Zeichen + Terminator
  char adresse[100];
  char kuerzel[3];

  printf("Vorname und Nachname: ");
  scanf("%s %s",vorname,name);
  fflush(stdin);

  printf("Adresse: ");
  gets(adresse);
  fflush(stdin);

  kuerzel[0]=vorname[0];
  kuerzel[1]=name[0];
  kuerzel[2]='\0';
}
```

Ungeachtet gegebenenfalls vorhandener Leerzeichen können alle Strings mit printf ausgegeben werden. Als Formatanweiser dient %s:

```
void main()
{
  char vorname[21];   // 20 Zeichen + Terminator
  char name[21];      // 20 Zeichen + Terminator
  char adresse[100];
  char kuerzel[3];

  printf("Vorname und Nachname: ");
  scanf("%s %s",vorname,name);
  fflush(stdin);

  printf("Adresse: ");
  gets(adresse);
  fflush(stdin);

  kuerzel[0]=vorname[0];
```

```
  kuerzel[1]=name[0];
  kuerzel[2]='\0';

  printf("\nName: %s, Vorname: %s, Kuerzel: %s\n",name,vorname,kuerzel);
  printf("Adresse : %s\n",adresse);
}
```

Das vollständige Programm erfragt Vorname, Name und die Adresse, erzeugt das Kürzel und gibt alle Daten anschließend auf dem Bildschirm aus:

```
Vorname und Nachname: Max Mayr
Adresse: Braunau am Inn, Osternbergerstraße 55

Name: Max, Vorname: Max Kuerzel: MM
Adresse: Braunau am Inn, Osternbergerstraße 55
```

Zur Initialisierung von Strings oder Bildschirmausgaben benötigen wir Stringkonstanten.

```
"Dies ist ein String!"
"a"
```

Beachte den Unterschied zwischen 'a' und "a"!

- 'a' ist das ASCII-Zeichen a
- "a" ist eine Zeichenkette, die nur das Zeichen 'a' enthält.

Dies ist mehr als eine Spitzfindigkeit, da Zeichen bzw. Strings im Rechner unterschiedlich dargestellt werden und demzufolge auch unterschiedlich zu verarbeiten sind.

Auch in Zeichenketten verwenden wir die bereits bekannten Escape-Sequenzen, um die nicht druckbaren bzw. mit einer Sonderbedeutung belegten Zeichen einzubauen:

```
printf("\tAlles\tklar!\n");
printf("Ein \"String\" im String\n");
printf("Dies ist ein Backslash: \\ \n");
printf("Dies ist ein \'A\'; \101 und noch eins: \x41\n");
```

Der Compiler löst die Escape-Sequenz auf und speichert den zugehörigen Zeichencode ab. In dem Beispiel ergeben sich die folgenden Bildschirmausgaben:

```
        Alles     klar!
Ein "String" im String
Dies ist ein Backslash: \
Dies ist ein 'A': A und noch eins: A
```

Ich hatte oben bereits erwähnt, dass der Name eines Arrays in C wie ein Zeiger auf das erste Element verwendet wird. Das hat zur Folge, dass zwei Strings nicht mit dem Operator = kopiert und nicht mit dem Operator == verglichen werden können.

So verlockend es auch ist, das folgende Programm zu schreiben,

```
char w1[20];
char w2[20];
```

```
printf("Erstes Wort: ");
scanf("%s",w1);
fflush(stdin);

printf("Zweites Wort: ");
scanf("%s",w2);
fflush(stdin);

if(w1 == w2)
  printf(Die Woerter sind gleich\n");
else
  printf("Die Woerter sind verscshieden\n");
```

so muss man doch feststellen, dass dieses Programm lediglich die Adressen der beiden Strings miteinander vergleicht. Diese sind im Übrigen stets verschieden, da sie ja an verschiedenen Stellen im Speicher liegen. Will man die Strings inhaltlich miteinander vergleichen, so muss man in einer Schleife Buchstaben für Buchstaben die beiden Wörter miteinander vergleichen:

```
char w1[20];
char w2[20];
int i;

... Einlesen der beiden Woerter ...

for( i=0; (wl[i] == w2[i]) && (wl[i] != 0);i=i+1)

if( wl[i] == w2[i])
  printf( "Die Woerter sind gleich\n");
eise
  printf( "Die Woerter sind verschieden\n");
```

Das Programm ist durchaus trickreich, sodass wir uns noch einmal anschauen wollen, was genau passiert. In der Schleife werden die beiden Wörter Buchstabe für Buchstabe durchlaufen. Die Schleife wird abgebrochen, sobald die Bedingung w1[i] = = w2[i] verletzt ist, d.h., sobald man eine Stelle erreicht, an der sich beide Wörter unterscheiden.

Dies kann in drei verschiedenen Situationen auftreten:

1. Das erste Wort ist zu Ende, das zweite aber noch nicht.
2. Das zweite Wort ist zu Ende, das erste aber noch nicht.
3. Keins der beiden Wörter ist zu Ende, aber sie unterscheiden sich an der betrachteten Stelle.

Offen bleibt jetzt nur noch der Fall, dass beide Wörter gleich sind. Hier besteht die Gefahr, dass die Schleife über das Wortende hinausläuft und hinter dem Wortende stehende, als zufällig anzusehende Zeichen in den Vergleich einbezieht. Hier ziehen wir die Notbremse, indem wir zusätzlich festlegen, dass nur weitergemacht werden soll, solange das erste Wort noch nicht beendet ist (w1[i] ! = 0). Ob die Wörter insgesamt gleich waren, erkennen wir, egal mit welcher Bedingung die Schleife abgebrochen wurde, an den zuletzt verglichenen Buchstaben.

Auch beim Kopieren eines Strings s2 in einen anderen s l müssen wir Buchstabe für Buchstabe vorgehen:

```
for(i=0; s[i] != '\0'; i++)
  s1[i]=s2[i];
s1[i]='\0';
```

Beachte, dass wir den kopierten String noch mit '\0' abschließen müssen, da in der Schleife der Terminator nicht mitkopiert wird. Bevor man einen solchen Kopiervorgang durchführt, muss man natürlich sicher sein, dass der String s2 in den Array s1 hineinpasst. Dazu benötigt man u.U. eine Funktion, die die Länge eines Strings s ermittelt, indem Sie die Buchstaben zählt:

```
for(i=0; s[i] != '\0'; i++);
```

Die Netto-Länge des Strings, also ohne Einbeziehung des Terminators, steht anschließend in der Variablen i .

Alle hier vorgestellten Algorithmen zur Stringverarbeitung funktionieren nur, wenn die zu bearbeitenden Strings immer sauber mit dem Terminatorzeichen abgeschlossen sind. Ist das nicht der Fall, können unvorhersagbare Fehler eintreten.

Den Bedarf an solchen immer wiederkehrenden Operationen auf Strings haben natürlich auch die Macher von C erkannt und deshalb eine Reihe von Funktionen für die Stringverarbeitung bereitgestellt. Diese Funktionen werden wir später in einem Abschnitt über die sogenannte Runtime-Library besprechen. Für das Erlernen der Sprache C ist es besser, wenn Sie solche Operationen vorerst einmal selbst programmieren.

7.1 Programmierbeispiel mit Arrays und Strings

7.1.1 Buchstaben zählen

Wir wollen ein Programm erstellen, das eine Reihe von Wörtern einliest und dann eine Statistik über die Häufigkeit des Vorkommens von Buchstaben erstellt. Der Einfachheit halber beschränken wir uns auf die Kleinbuchstaben a-z.

Als Vorgabe zur Pogrammerstellung dient uns die folgende Eingabe:

```
Anzahl: 9
1. Wort: the
2. Wort: quick
3. Wort: brown
4. Wort: fox
5. Wort: jumps
6. Wort: over
7. Wort: the
8. Wort: lazy
9. Wort: dog
```

Nach dem Einlesen der Zeichenketten soll die Buchstaben-Statistik in der folgenden Form ausgegeben werden.:

```
Auswertung:
a: 1
b: 1
c: 1
d: 1
e: 1
f: 1
g: 1
h: 1
i: 1
j: 1
k: 1
l: 1
m: 1
n: 1
o: 1
p: 1
q: 1
r: 1
s: 1
t: 1
u: 1
v: 1
w: 1
x: 1
y: 1
z: 1
```

Nach diesen Vorgaben entwickeln wir das Programm.

Um die Buchstabenhäufigkeiten zu zählen, legen wir unter dem Namen count einen Array mit 26 Integer-Feldem an. Im Feld count[0] zählen wir, wie oft 'a', im Feld count[1] zählen wir, wie oft 'b' vorgekommen ist usw. Zwischen dem Arrayindex und dem zugehörigen Buchstaben bestehen dann die folgenden Umrechnungsvorschriften:

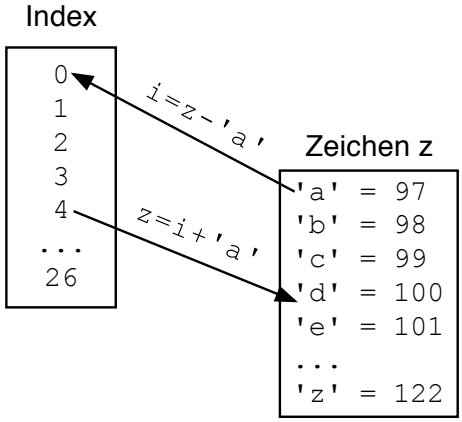


Abbildung 7.2: Zusammenhang zwischen Zeichen und Index

```
void main()
{
        unsigned char buf[80];
        int count[26];
        int i,k,anz;

        for(i=0;i<26;i++)
                count[i]=0;

        printf("Anzahl: ");
        scanf("%i",&anz);
        fflush(stdin);

        for(k=1;k<=anz;k++)
        {
                printf("%i. Wort: ",k);
                gets(buf);
                fflush(stdin);

                for(i=0;buf[i] != '\0'; i++)
                        count[buf[i]-'a']++;
        }

        printf("\nAuswertung:\n");
        for(i=0;i < 26; i++)
                printf("%c: %i\n",'a'+i,count[i]);
}
```

7.1.2 Übungen

1. Schreibe folgende Stringfunktionen:

   - **unsigned int** strlen(**const char** string[ ]);
   - **void** strcpy(**char** dest[ ], **const char** src[ ]);
   - **int** strchr(**const char** s[ ], **int** c);
   - **void** strcat(**char** dest[ ], **const char** src[ ]);
   - Für Fortgeschrittene:
     int strcmp(**const char** s1[ ], **const char** s2[ ]);
     int strstr(**const char** s1[ ], **const char** s2[ ]);

2. Schreibe folgende Stringfunktionen:

   - ReverseStr(): Umkehren eines Strings (z:B. aus "Hallo" ==> "ollaH")
   - LeftStr(): Kopieren der ersten anz Zeichen eines Quellstrings in einen Zielstring
   - RightStr(): Kopieren der letzten anz Zeichen eines Quellstrings in einen Zielstring
   - MidStr(): Kopieren von anz Zeichen ab einem Index pos aus einem Quellstring in einen Zielstring
     Überlege ,wie die Übergabeparameter aussehen sollten (Arrays verwenden!).

3. Schreibe eine Funktion mygets(), welche das Verhalten der Funktion gets() zum Einlesen eines Strings von der Tastatur nachbildet.

4. Schreibe eine Funktion, die eine arabische Zahl in eine römische Zahl umwandelt.
   Anleitung:
   M...1000, D...500, C...100, L...50, X...10, V...5, I...1
   900 kann durch CM oder DCCCC ausgedrückt werden, analog dazu 90 und 9. 400 kann durch CD oder CCCC ausgedrückt werden, analog dazu 40 und 4.

5. Schreiben Sie ein Programm, das die Anzahl der Selbstlaute eines Textes ermittelt und in folgender Form am Bildschirm ausgibt:

```
          Anz.          %
-------------------------------------------------
A:          5       18.52 ##########
E:          9       33.33 ##################
I:          8       29.63 ################
O:          2        7.41 ###
U:          3       11.11 ######
-------------------------------------------------
Ges.:      27
```

   Der zu untersuchende Text soll in einem konstanten Character-Array gespeichert werden. Die Ausgabe des Balkens soll mit Hilfe einer Funktion realisiert werden. Die Balkenlänge soll für 100% 50 Zeichen entsprechen.

Anmerkungen:

- Verwende nirgends globale Variablen

- Teste alle Funktionen innerhalb der Funktion **void** main( ) aus
- Es dürfen nur selbstgeschriebene Funktionen verwendet werden

Kapitel 8

Datenstrukturen, Wert- und Zeigersemantik

8.1 Zeiger, Adressen, Vektoren

Das unten angeführte Programm zeigt zum einen am Beispiel des Arrays a (in C spricht man meist von Vektoren), dass man in C mit linearen Arrays arbeiten kann. Gleichzeitig zeigt es aber eine alternative - und zwar sehr C-typische - Möglichkeiten, nämlich das Arbeiten mit Zeigern (Adressen).

```
#include <stdio.h>

void main()
{
        int i;
        int a[10], b[10]; // zwei Arrays mit je 10 int-Werten
        int *pb;          // ein Zeiger auf einen int-Wert
        int c[9];         // ein Array für 9 int-Werte
        int *pc;          // Zeiger auf einen int-Wert

        pb = &b[0];       // oder auch pb=b
        pc=c;    // oder auch pc = &c[0];

        for (i=0;i<=9;i++)
        {
                a[i]=i*i;
                *pb=i*i;
                pb++;
                *pc=i*I;
                pc++;
                // Achtung!! es ist nur Speicherplatz für
                // c[0] bis c[8] reserviert
        }

        pb =&pb[0];
```

```
        pc=c;

        for(i=0;i<=9;i++)
        {
                printf("%4d %4d %4d %4d %4d %4\n",i,a[i],b[i],*pb,c[i],*pc);
                pb++;
                pc++;
        }
}
```

Erläuterungen:

In Abbildung 8.1 wird der Zusammenhang zwischen Vektoren und Adressen sowie die jeweiligen Zugriffsmöglichkeiten über Indizierung und über Adressrechnung dargestellt

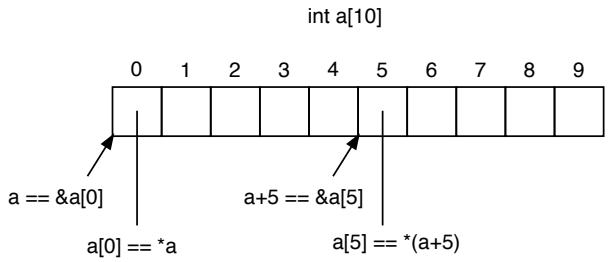


Abbildung 8.1: Die Dualität zwischen Vektoren und Adresssen

- Bei der Vereinbarung von Arrays (Vektoren) wird die Anzahl der Elemente angegeben, der Compiler reserviert entsprechend Speicherplatz.
- Der Indexbereich beginnt stets per Definition bei 0.
- Die Vereinbarungen **int** ∗pb, ∗pc; bedeuten: die Inhalte von pb und pc sind vom Typ int, d.h. pb und pc selbst sind die entsprechenden Adressen (Zeiger) auf diese Werte.
- Mit pb = &b [0] und pc = &c [0] werden die Zeiger jeweils auf das erste Element des Vektors b bzw. des Vektors c gesetzt.
- Das gleiche kann durch die Anweisungen pb = b bzw. pc = c erreicht werden!
- Das liegt daran, dass ein Vektor (Array) durch seine Adresse dargestellt wird!
- Somit kann der Zugriff z.B. auf das i-te Element des Vektors b entweder über die Notation b[i] oder über die Notation ∗(b+i) erfolgen oder, wegen der Zuweisung pb = b, natürlich auch über ∗(pb+i) bzw. pb[i].
- Bei der Adressrechnung b+i, z.B. b+3, wird die Adresse typgerecht um drei Schritte, also nicht etwa um drei Byte erhöht; das gilt allgemein für Adressrechnungen in C.
- In C wird nicht geprüft, ob ein Index innerhalb des erlaubten (d.h. vereinbarten) Bereiches liegt, das erkennt man ganz deutlich an der Benutzung des Vektors c, für den ja nur die Größe 9 vereinbart wurde!
- Adressrechnungen werden ohnehin nicht überprüft, also Vorsicht!

- Das oben angegebene Programm "funktioniert zwar meistens", obgleich es mit nicht reserviertem Speicher arbeitet, aber das liegt nur daran, dass das Programm so klein ist und sonst nichts macht!

8.1.1 Initialisierung von Vektoren

Vektoren können an der Stelle der Vereinbarung auch gleich in folgender Form initialisiert werden:

```
int table[10]={0,1,2,3,4,5,6,7,8,9};
```

In dem Falle kann die explizite Angabe der Vektorgröße auch entfallen, da sie sich implizit aus der Anzahl der Initialisierungselemente ergibt:

```
int table[]={0,1,2,3,4,5,6,7,8,9};
```

8.1.2 Adressarithmetik, NULL-Zeiger

Wie schon in den obigen Erläuterungen gesagt, können Adressen inkrementiert und dekrementiert werden, z.B.

```
double dx[10];
/*Es wird Speicher für einen Vektor mit 10 Elementen vom Typ
double reserviert, dx ist ein typisierter Zeiger, der auf den
Anfang des ersten Elements zeigt.
Achtung: dx ist kein L-Wert, d.h. dx kann nicht verändert werden ! */

double* dy = dx;    // dy ist im Gegensatz zu dx ein L-Wert !

dy++;
++dy;   // dy zeigt jetzt auf dx[2]
dy += 7;        // dy zeigt jetzt auf dx[9], das letzte Element
dy -= 9;        // dy zeigt jetzt auf dx[0], das erste Element.
```

Die angegebenen Inkremente und Dekremente sind ganzzahlige Werte, die als typisierte Schritte interpretiert werden, d.h. z.B., dass dx += 7 bedeutet: plus sieben Schritte mit der Schrittweite sizeof (double). Adressen können nicht addiert, aber sie können subtrahiert werden. Voraussetzung dabei ist natürlich, dass beide Adresswerte vom gleichen Typ sind;

Beispiel:

```
int i, j, n;
double dx[100], *p, *q;

p = &dx [...];
q = &dx [...];
if (p < q)
        n = q - p;
else
        n = p - q;
```

Damit wird der Abstand zwischen p und q berechnet, z.B. p — q = 5 bedeutet, dass in dem Bereich von Adresse p bis Adresse q sechs Elemente liegen, allgemein gilt demnach

$$n = \begin{cases} p - q + 1 & : \quad p \geq q \\ q - p + 1 & : \quad sonst \end{cases} \tag{8.1}$$

Insgesamt sind folgende Operationen mit Zeigern möglich:

- Dereferenzieren mit Lesen und Schreiben: x = *p, *p = x
- Vergleichen: p < q, p <= q, p == q, p ! = q, p >= q, p > q
- Inkrementieren und Dekrementieren: p++, ++p, p- - , - -p
- Addieren, Subtrahieren: p + n, p - n. n muss ein ganzzahliger Typ sein und p - n darf nicht negativ werden.
- Abstände bestimmen: n = p - q, wenn p >= q, n ist ganzzahlig.

8.1.3 NULL-Zeiger:

Es gibt einen besonderen Zeigerwert, der als Konstante mit dem Namen NULL definiert ist. Die interne Darstellung der NULL ist implementierungsabhängig, aber kompatibel zur ganzzahligen Null (0). Der NULL-Zeiger kann jeder Zeigervariablen zugewiesen werden, und jede Zeigervariable kann auf den Wert NULL getestet werden;

Beispiele:

```
p=NULL; // Beide Anweisungen sind
p=0;    // korrekt und bedeutungsgleich

// Alle drei Abfragen sind korrekt und bedeutungsgleich
if (p == NULL)
if (p == 0)
if (!p)
```

Achtung, wichtiger Hinweis:

Adressrechnungen sind besonders gefährliche Operationen, weil keine Bereichsüberprüfungen gemacht werden. Dadurch passiert es leicht, dass nicht reservierte Speicherbereiche adressiert werden!

8.1.4 Adressparameter

In C gibt es für Funktionen nur Werteparameter, keine Variablenparameter wie z.B. in Pascal, Modula oder Oberon. Das bedeutet aber, dass der Funktion zwar von außen ein Wert übergeben, aber nicht umgekehrt ein Resultat von innen nach außen transportiert werden kann. Hier muss der Umweg über die Adresse gegangen werden: Man übergibt also nicht das Resultat direkt, sondern man übergibt die Adresse des Speicherplatzes (den Zeiger auf das Objekt), wo das Resultat abzulegen ist. Mit den oben angegebenen Erläuterungen werden die Hintergründe bei der Benutzung des Adressoperators beim Aufruf (aktuelle Parameter) und des Inhalts Operators bei der Deklaration und der Definition der Funktion (formale Parameter) klar, das folgende kleine Beispiel demonstriert die Zusammenhänge:

```
#include <stdio.h>

void pi(double *x);  // called by reference
```

```
void main()
{
        double y;

        // Es wird nicht die Variable y, sondern ein Zeiger auf y
        // übergeben.
        pi(&y);

        printf("%lf\n2,y);
}

void pi(double *x)
{
        *x=3.141592653;
}
```

Wenn das Resultat aber ein Vektor (Array) ist, dann ist zu beachten, dass auf Vektoren ja sowieso per Adresse zugegriffen wird (siehe obige Ausführung), d.h. eine explizite *Referenzierung* und *Dereferenzierung* ist in diesem Fall nicht angebracht, wie es auch durch das folgende kleine Beispiel demonstriert wird:

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

void Hallo(char s[6]);

void main()
{
        char str[10];

        Hallo(str);

        printf("%s\n",str);
}

void Hallo(char s[6])
{
        strcpy(s,"Hallo");
}
```

8.1.5 Strings

Das letzte Beispiel - die Funktion Hallo - definiert einen String (Zeichenkette). Ein String ist in C eine Folge von Zeichen, die durch einen Stringterminator = Stringendezeichen ('\0') abgeschlossen wird. Eine Stringkonstante kann z.B. wie folgt vereinbart werden:

```
const char Hallo[6]   =  "Hallo";
```

Entsprechend lässt sich eine Stringvariable als Zeichenvektor vereinbaren, der dann ein String zugewiesen werden kann:

```
char Name[20];
Name = "Peter"; // falsch, Name ist kein L-Wert
strcpy(Name,"Peter");    // der ganze String wird kopiert
```

Die Zuweisung mit dem Zuweisungsoperator ist schon syntaktisch falsch, weil Name ein konstanter Zeiger und damit kein L-Wert ist, d.h. Name darf nicht links vom Zuweisungsoperator stehen! Mit der Funktion strcpy aus string.h wird dagegen der String in den Zeichenvektor Name hineinkopiert, und zwar einschließlich des Stringendezeichens. Die Länge des String Peterïst fünf, der String wird aber jeweils automatisch mit dem zusätzlichen Stringendezeichen abgelegt.

Das folgende kleine Programm demonstriert die formatierte Eingabe und Ausgabe von Strings unter Verwendung des Formatschlüssels s:

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

int main(void)
{
        char Vorname[33];

        do
        {
                printf("Vorname    (<= 32 Zeichen)    :    ");
                scanf("°/,s",Vorname);
                fflush(stdin);

                if    (Vorname[0] != '.')
                        printf ("Hallo,  %s\n",Vorname);
        } while    (Vorname[0]    !=    '.');
}
```

Da eine Stringvariable ein Vektor ist, kann auch über Indizes auf einzelne Zeichen lesend und schreibend zugegriffen werden. Das Bibliotheksmodul string.h stellt dem Benutzer eine ganze Reihe von Funktionen zum Arbeiten mit Strings zur Verfügung, die wichtigsten sind:

- **char*** strcpy(**char** *s1, **char** *s2);
  den String s2 nach s1 kopieren, incl. Stringterminator
- **char*** strcat(**char** *s1, **char** *s2);
  den String s2 an den String s1 anhängen
- size_t strlen(**const char** *s);
  liefert dei Länge des Strings s, d.h. Anzahl der Zeichen ohne Stringterminator
- **int** strcmp(**const char** *s1, **const char** *s2);
  vergleicht beide Strings, liefert 0, wenn s1 = = s2, liefert einen Wert < 0, wenn s1 $<$ s2 und einen Wert > 0, wenn s1 $>$ s2. Das zugrunde liegende Alphabet ist der ASCII-Zeichensatz

Bei den beiden Funktionen strcpy und strcat wird das Ergebnis jeweils unter s1 abgelegt und zusätzlich als Funktionswert ein Zeiger auf den Ergebnis-String geliefert.

In Anlehnung an das Standardwerk von Kernighan/Ritchie sollen im folgenden am Beispiel einer selbst zu schreibenden Funktion strcpy das Arbeiten mit Vektoren und Zeigern sowie C-typische Notationen an Hand verschiedener Alternativen demonstriert werden, wobei hier der Einfachheit halber kein Funktionswert zurückgeliefert wird:

Version 1:

```
void strcpy(char s1[], char s2[])
{
        int i=0;

        while (s2[i] != '\0')
        {
                s1[i]=s2[i];
                i++;
        }
        s1[i]=s2[i];
}
```

Version 2:

```
void strcpy(char *s1, char *s2)
{
        int i=0;

        while (*(s2+i) != '\0')
        {
                *(s1+i)=*(s2+i);
                i++;
        }
        *(s1+i)=*(s2+i);
}
```

Version 3:

```
void strcpy(char *s1, char *s2)
{
        int i=0;

        while (*s2 != '\0')
        {
                *s1=*s2;
                s1++;
                s2++;
        }
        *s1=*s2;
}
```

Version 4:

```
void strcpy(char *s1, char *s2)
{
        while((*s1=*s2) != '\0')
```

```
        {
                s1++;
                s2++;
        }
}
```

Version 5:

```
void strcpy(char *s1, char *s2)
{
        while((*s1++=*s2++) != '\0');
}
```

Version 6:

```
void strcpy(char *s1, char *s2)
{
        while(*s1++=*s2++);
}
```

Die entsprechende Funktion strcpy aus der C-Standardbibliothek (string.h) hat folgenden Prototyp:

char* strcpy(**char*** sl, **const char*** s2);

Das Attribut **const** dokumentiert und garantiert, dass der Zeiger s2 von der Funktion nicht verändert wird. Die Funktion liefert zusätzlich als Funktionswert einen Zeiger auf den Ergebnisstring s1.

Sie Abbildung 8.2 stellt die C-typische Anweisung aus Version 6 zum Kopieren einer Zeichenkette graphisch dar. Es wird jeweils kopiert entsprechend *s1 = *s2, danach werden dann beide Zeiger inkrementiert. Wenn das Stringende-Zeichen '\0' kopiert ist, dann erhält auch der Ausdruck den Wert '\0', der als false interpretiert wird, und damit terminiert die while-Schleife.

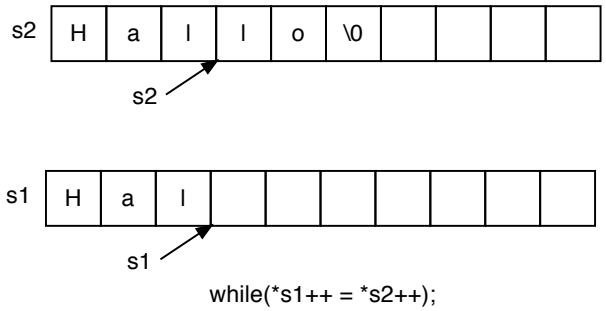


Abbildung 8.2: Kopieren einer Zeichenkette (String)

8.1.6 Alternative Darstellungen bei der Verwendung von Strings

In dem folgenden Beispiel sind die Definitionen von s1 und s2 gleichwertig, in beiden Fällen wird ein Vektor der Größe 6 bereitgestellt und mit dem String "hallo" gefüllt (incl. Stringterminator):

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

```
void main()
{
        char s1[6] = "hallo";
        char s2[] = "hallo";
        char *s3 = "hallo";

        strcpy(s1,    "AAAAA");   /* richtig */
        strcpy(s2,    "BBBBB");   /* richtig */
        strcpy(s3,    "CCCCC");   /*???*/
        printf("%s %s %s\n",s1,s2,s3);
}
```

Anders verhält es sich dagegen mit der Definition von s3, hier wird ein Zeiger bereitgestellt, der auf einen nicht veränderbaren String “hallo” zeigt. Nun kann zwar der Zeiger s3 auf einen anderen String gesetzt werden, aber wenn versucht wird, mit strcpy(s3, “CCCCC”) den String “hallo” - auf den s3 zeigt - zu verändern, dann ist das Ergebnis Undefiniert!

Man lässt sich leicht dadurch täuschen, dass das angegebene Programm doch “richtig” abläuft und das erwartete Ergebnis erzeugt, aber es ist keinesfalls gewährleistet, dass es mit einem anderen Compiler oder in einer anderen Umgebung auch richtig läuft, gerade darum sind solche Fehler, die nur “ab und zu” auftreten, besonders gefährlich!!!

8.1.7 Nichttypisierte Zeiger

In der folgenden Deklaration ist p ein *nichttypisierter* - man kann auch sagen: ein *neutraler* - Zeiger:

void *p;

Nichttypisierte Zeiger sind mit allen typisierten Zeigern kompatibel in dem Sinne, dass man ihnen typisierte Zeigerwerte zuweisen kann und umgekehrt sowie, dass man beim Aufruf einer Funktion für formale nichttypisierte Parameter aktuelle typisierte Parameter einsetzen kann und umgekehrt, z.B.:

```
double *x;
void *p;

void f(void *pp)          // nichttypisierter formaler Parameter
{
        ...
}

p=x;              //richting
x=p;              // richtig aber unschön, in C erlaubt, C++ nicht
x=(double*)p;     // richtig und schön

f(x);             //richtig
```

Die Zuweisung von **void***-Zeigern an typisierte Zeiger ohne explizites “tasten” (explizite Typumwandlung, Typecast) ist in ANSI/ISO-C erlaubt, in ANSI/ISO-C++ aber nicht, sie sollte deshalb vermieden werden! Nichttypisierte Zeiger können nicht dereferenziert werden, das ist eine einleuchtende und wichtige Einschränkung im Gebrauch von **void***-Zeigern:

```
double *x,y,z;
void *p;
x=&z;
*x=12.34;
y=*x;    // klar
p=x;     // in Ordnung
y=*p;    // falsch, p bleibt ein void*-Zeiger und kann nicht derefenziert werden!
```

Die Nichtdereferenzierbarkeit ist einleuchtend, denn was und wie sollte als Inhalt interpretiert werden, das nächste Byte als Zeichen, das nächste Wort als ganze Zahl, die nächsten acht Byte als Gleitkommazahl, ... ? Das Themengebiet typisierte Zeiger, nichttypisierte Zeiger, Zeigerkonvertierungen wird später gründlich behandelt.

8.1.7.1 Typnamen, Synonyme für Typnamen

In C ist es möglich, z.B. Zweitnamen (Synonyme) für vorbelegte Typnamen festzulegen, z.B.:

```
typedef double Real;
typedef unsigned int Uint;
```

Die Synonyme können dann genauso verwendet werden wie die originalen Namen, z.B.:

```
Real x,y,z;      // wie double x,y,z
Uint a,b,c;      // wie unsigned int a,b,c
```

Besonders praktisch ist die Möglichkeit, sich Typnamen zu definieren, für die bis dahin keine expliziten Bezeichner existieren, um sie dann als aussagekräftige Typnamen bei der Vereinbarung von Variablen zu benutzen, z.B.:

```
typedef double Vector[100] ;
typedef char       String[20];

Vector x;        //wie double x[100];
String name;     //wie char name[20];
```

Übungen:

Interpretiere folgende Konstrukte:

- **double** x,y,*p,*q;
- ++p=q++;
- y=x**p;
- *p++ = *q++;
- *++p=*++q;
- y=*--x;
- y=+++x;
- y=+x++;

- y=++x++;
- y=++(++x);
- y=++++x;
- z=x+++++y;

8.2 Programmierbeispiel mit Zeigern

8.2.1 Übungen

1. Bei txt soll es sich um eine Stringvariable handeln, die mit dem Text “abcdefg” initialisiert wurde. Es soll ein Zeiger pt definiert und auf das Zeichen ’c’ in der Zeichenkette txt gesetzt werden. Wie wird die Zeigervariable pt definiert, und wie lautet die Zuweisung an die Zeigervariable?

2. Eine Stringvariable der Länge 80 ist mit der Zeichenkette “abcdefg” initialisiert worden. Nun soll sie um das Zeichen ’h’ am Ende verlängert werden. Definiere dazu einen Zeiger pt, der unter Benutzung der Funktion strlen auf das Ende der Zeichenkette gesetzt wird. Wie lautet die Zuweisung an die Zeigervariable pt, und welche Schritte sind notwendig, um das ’h’ an die Zeichenkette anzuhängen?

3. Es seien pt1 und pt2 zwei char-Zeiger, sh eine **int**-Variable. Welche der folgenden Zuweisungen wird der Compiler akzeptieren, welche nicht?

   (a) pt1=pt2+sh;
   (b) pt1=sh+pt2;
   (c) sh=pt1∗pt2;
   (d) sh=pt1−pt2;
   (e) sh=pt1+pt2;

4. Es seien pt1, pt2 und pt3 drei **int**-Zeiger. Wie muss die Variable res definiert werden, damit der Compiler die Zuweisung

   res = pt1 + (pt2 - pt3);

   akzeptiert? Was wird in diesem Fall berechnet?Ist die Zuweisung sinnvoll?

5. Schreib ein Programm, dass jedes ’E’ oder ’e’ in einer Stringvariablen mit dem Namen txt durch einen Bindestrich ersetzt wird.

6. Eine Kunstausstellung hatte an 28 Tagen geöffnet. In einem C-Programm werden einige statistische Daten über die Besucherzahlen ausgewertet. Es sei definiert:

   ```
   #define TAGE 28
   short anz_bes[TAGE], summe, *pt;
   ```

   Im Array anz_bes sind die Besucherzahlen an den einzelnen Tagen gespeichert. Welchem Trugschluss ist der Programmierer unterlegen, der mit folgender Schleife die Gesamtzahl der Besucher ermitteln wollte:

   ```
   for (summe=0, pt=anz_bes; *pt; pt++)
           summe = summe + *pt;
   ```

   Und wie könnte man dem durch eine kleine Änderung abhelfen?

7. In einem physikalischen Programm werden Abweichungen von Messwerten verarbeitet. Positive Werte bedeuten eine Abweichung nach oben, negative Werte eine Abweichung nach unten. In einem **double**-Array mit dem Namen Abw mit 500 Elementen sind diese Abweichungen gespeichert. In einer Schleife sollen alle Abweichungen auf dem Bildschirm ausgegeben werden, die größer als 10.0 oder kleiner als -10.0 sind. Warum ist es schwierig, dieses Array mit Zeigern zu bearbeiten? Schreibe für die Auswertung eine Funktion.

8. Im **char**-Array txt sei eine Zeichenkette gespeichert. Es soll nun das erste Auftreten des Zeichens 'a' in txt ermittelt werden. Resultat soll ein Zeiger auf das entsprechende Element in txt sein. Schreibe eine Funktion.

9. Es ist Wert ein **int**-Array und pt ein **int**-Zeiger. Welche der folgenden Zuweisungen sind zulässig, welche nicht?

   (a) pt = Wert;
   (b) Wert = pt;
   (c) pt = &Wert[3];
   (d) Wert[2] = pt[5];

10. Ein Programmierer probiert die Initialisierung einer Stringvariablen mittels geschweifter Klammern aus. Er schreibt:

```
char txt[]={'a', 'b', 'c','d','0'};
```

    Beim Aufruf der strlen-Funktion zur Ermittlung der Länge der Zeichenkette in der Form

```
len=strlen(txt);
```

    erhält er nun einen völlig unsinnigen Wert. Er wechselt daher sofort wieder auf die gewohnte Initialisierung:

```
char txt[]="abcd";
```

    Was hat der Programmierer falsch gemacht?

11. In der folgenden Funktion wird die Länge einer Zeichenkette berechnet:

```
void txtlen(char *txt, int len)
{
        char *pt;

        for(pt=txt;*pt;pt++);

        len=pt-txt;
}
```

    Welchem Denkfehler ist der Programmierer unterlegen, und welche Änderungen sind möglich (mehrere Möglichkeiten)?

12. Schreibe die Funktionen aus den Übungen zu Strings (siehe 7.1.2) unter ausschließlicher Verwendung von Zeigern.

13. Schreibe ein Programm, dass einen Satz einliest und jedes Wort in einer eigenen Zeile wieder ausgibt. Als Wortbegrenzung dient das Leerzeichen. Realisieren Sie das Programm ohne bzw. unter Verwendung der Stringfunktion strtok().

14. Jedes Buch hat eine ISBN (Internationale Standard-Buchnummer). Diese ISBN besteht aus 9 Ziffern (Z1-Z9) und einer Prüfsumme (Z0). Die Ziffern liegen jeweils im Bereich von 0 bis 9. Das Prüfzeichen ist eine Ziffer (0-9) oder ein X, welches für die Zahl 10 steht. Die Nummer ist durch drei Bindestriche

gegliedert, wobei die Position der Bindestriche nicht exakt festgelegt ist. Die Prüfsumme ist aber in jedem Fall das letzte Zeichen der ISBN.

Das folgende Beispiel zeigt eine gültige ISBN:

Z_9		Z_8	Z_7	Z_6	Z_5	Z_4		Z_3	Z_2	Z_1		Z_0
3	-	8	9	3	1	9	-	3	8	6	-	3

Das Prüfzeichen dient dazu, festzustellen, ob eine ISBN (z.B. bei einer Buchbestellung) korrekt übermittelt wurde. Eine ISBN gilt als korrekt übertragen, wenn die Summe:

$$\sum_{i=0}^{9}(i+1)\cdot Z_i \tag{8.2}$$

ohne Rest durch 11 teilbar ist.

Schreibe eine Funktion ISBNTest(), welcher eine vermeintliche ISBN als String übergeben wird und die feststellt, ob es sich dabei um eine korrekte ISBN handelt. Der Rückgabewert soll 1 sein, falls die vermeintliche ISBN richtig ist, an sonst 0.

Anmerkungen:

- Verwende nirgends globale Variablen
- Teste alle Funktionen innerhalb der Funktion **void** main( ) aus

Kapitel 9

Mehrdimensionale Vektoren

9.1 Deklaration, Initialisierung, Zugriff auf Elemente

Als erstes Beispiel für einen mehrdimensionalen Vektor betrachten wir eine zweidimensionale Matrix mit **double**-Zahlen der Größe drei Zeilen und vier Spalten:

```
double matrix[3][4];      //3 Zeilen, 4 Spalten
```

Die einzelnen Elemente der Matrix werden in der Form matrix [i] [j] angesprochen.

Formal gesehen ist die Variable matrix ein linearer Vektor, dessen Elemente auch wieder lineare Vektoren sind, d.h. die folgende Deklaration ist mit der vorangehenden kompatibel:

```
typedef double  Zeile[4];
Zeile matrix [3];
```

Die Matrix kann an der Stelle der Vereinbarung auch gleich initialisiert (mit Werten belegt) werden, z.B.:

```
double matrix[3][4] = { {1.1,2.2,3.3,4.4},
                        {5.5,6.6,7.7,8.8},
                        {9.9,0.0,1.1,2.2}};
```

Die inneren geschweiften Klammern sind redundant, sie dienen nur der optischen Strukturierung:

```
double matrix[3][4] = {1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,0.0,1.1,2.2};
```

Die Elemente werden im Arbeitsspeicher in der folgenden Reihenfolge abgelegt:

```
matrix[0][0]=1.1;
matrix[0][1]=2.2;
matrix[0][2]=3.3;
matrix[0][3]=4.4;
matrix[1][0]=5.5;
matrix[1][1]=6.6;
matrix[1][2]=7.7;
...
matrix[2][3]=2.2;
```

Das ist leicht nachprüfbar, wenn man sich das Speicherabbild wie folgt in linear aufsteigender Reihenfolge ausgeben lässt:

```
double* p;

p=&matrix[0][0];

for(int i=0;i<3*4;i++)
  printf("%lf ",*p++);
```

Exakt die gleiche Ausgabe erhält man natürlich bei folgender, an den Indizes orientierten Ausgabe:

```
for(i=0;i<3;i++)   // Zeilenindex
{
  for(j=0;j<4;j++)   // Spaltenindex
  {
    printf("%lf ",matrix[i][j]);
  }
}
```

9.1.1 Alternativen für den Zugriff auf Elemente

Beim lesenden oder schreibenden Zugriff auf einzelne Elemente eines Vektors gibt es wegen der vorhandenen Dualität Indizierung/Adressrechnung generell die Alternativen, entweder wie oben über Indizes oder über Adressen und Dereferenzierung oder über eine gemischte Form

```
x=matrix[i][j];          // indizierter Zugriff
x=(matrix[i])[j];        // indizierter Zugriff
x=*(matrix[i]+j);        // Zugriff über Index und Adressberechnung
x=*(*(matrix+i)+j);      // Zugriff über Adressberechnung
```

Beispiel:

```
#include <stdio.h>

void main()
{
        int mat1[5][2]={{0,1},{2,3},{4,5},{6,7},{8,9}};
        int x1,x2,x3,x4,i,j;

        for(i=0;i<5;i++)
        {
                for(j=0;j<2;j++)
                {
                        x1=mat1[i][j];
                        x2=(mat1[i])[j];
                        x3=*(mat1[i]+j);
                        x4=*(*(mat1+i)+j);

                        printf("%i, %i, %i, %i ",x1,x2,x3,x4);
                }
```

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

Ausgabe des Programms:

```
0000 1111
2222 3333
...
8888 9999
```

9.1.2 Übergabe mehrdimensionaler Vektoren als Parameter

Die Übergabe eindimensionaler Vektoren variabler Größe bereitet keine Probleme, z.B.:

```
#define SIZE 5
double vector[SIZE];

void f(double* arr);              // Zeiger
void g(double arr[]);             // als Vektor
```

Die entsprechende Übergabe eines zweidimensionalen Arrays in der Form

```
#define ROWS 5
#define COLS 2

double vector[ROWS][COLS];

void f(double matrix[][]); // falsch!!!!!
```

ist leider nicht möglich. Als Lösung bietet sich folgende Form an:

```
#define ROWS 5
#define COLS 2

double vector[ROWS][COLS];

void f(double matrix[ROWS][COLS]);
```

9.1.3 Listen mit Strings

Für die Ablage von Strings in Listen und Tabellen gibt es verschiedene Alternativen, die hier am Beispiel einer Tabelle der Monatsnamen gegenübergestellt werden. **Alternative** 1 sieht so aus:

```
char Monat1[13][10]= {  "???       ",
                        "Jänner    ","Februar   ","März      ",
                        "April     ","Mai       ","Juni      ",
                        "Juli      ","August    ","September ",
                        "Oktober   ","November  ","Dezember  "};
```

Hier wird von dem Sonderfall Gebrauch gemacht, dass in ANSI-C in dem Falle Stringlänge = Vektorlänge kein Stringendezeichen erforderlich ist (im Gegensatz zu ANSI-C++). Aus leicht nachvollziehbaren Gründen enthält die Liste dreizehn statt nur zwölf Einträge (damit Jänner an Position 1 und Dezember an Position 12 steht), der erste ist ein Platzhalter (Dummy). Jeder Monat wird im Speicher durch zehn Zeichen dargestellt, d.h. die Liste wird in folgender Darstellung auf den Arbeitsspeicher abgebildet:

??? Jänner Februar März ... Dezember

Das lässt sich ebenfalls wieder leicht nachprüfen, und zwar durch folgende linear arbeitende Ausgabe:

```
char * p;
int i;

p=&Monat1[0][0];

for(i=0;i<13*10;i++)
        printf("%c ",*p++);
```

Wenn bei der Initialisierung jeweils die Blanks am Ende weggelassen werden, dann wird bei der internen Darstellung jeder String mit dem Stringterminator abgeschlossen, **Alternative 2:**

```
char Monat2[13][10]= {  "???",
                        "Jänner","Februar","März",
                        "April","Mai","Juni",
                        "Juli","August","September",
                        "Oktober","November","Dezember"};
```

Die entsprechende Ausgabe kann dann z.B. so aussehen:

```
for(i=0;i<13;i++)
        printf("%s ",Monat1[i]);
```

Ganz anders sieht die interne Struktur aus, wenn die Liste als **Alternative 3** wie folgt definiert wird:

```
char *Monat3[13]= {     "???",
                        "Jänner","Februar","März",
                        "April","Mai","Juni",
                        "Juli","August","September",
                        "Oktober","November","Dezember"};
```

In diesem Falle gibt es einen Speicherbereich, in dem nacheinander dreizehn char*-Zeiger abgelegt sind und andere Speicherbereiche, in denen die dreizehn konstanten Strings abgelegt sind, siehe dazu auch Abschn. 1.8.1. Ein praktischer Unterschied der beiden Darstellungen Monat [13] [10] und *Monat [13] besteht z.B. darin, dass mit strcpy(Monat2[0] , " !!! "); der nullte Eintrag neu geschrieben wird, wohingegen strcpy(Monat3[0] , " !!! "); zu Undefiniertem Verhalten führt. Das Ziel der Kopieroperation ist ein String, der nicht verändert werden darf:

- Ein Versuch den String doch zu verändern, führt zu **undefiniertem Verhalten**!

Als Alternative 4 bietet sich natürlich auch die Wahl eines linearen Vektors an:

```
char Monat4[13*10]={"???       Jänner    Februar   März      ... Dezember"};
```

Das Abbildung 9.1 stellt die genannten vier Alternativen zur Speicherung einer Liste mit Strings am verkürzten Beispiel einer Liste für vier Namen dar.

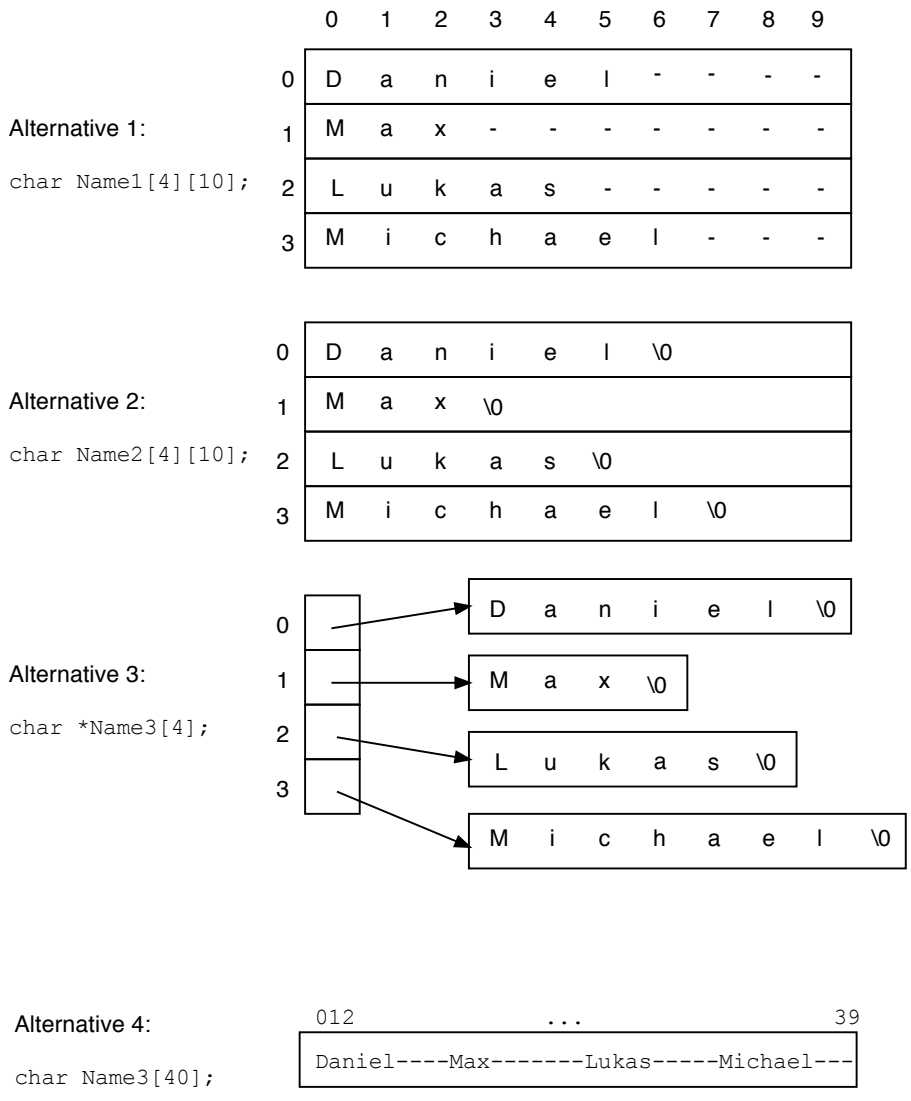


Abbildung 9.1: 4 Alternativen zur Speicherung einer Liste mit Strings

9.2 Programmierbeispiel mit Mehrdimensionale Vektoren

9.2.1 Übungen

Schreibe sämtliche folgenden Funktionen unter Verwendung des []-Operators: (Zeiger sind verboten!)

1. Schreibe eine Funktion lnt2DArrayln( ) für die Werteingabe In ein zweidimensionales Integer-Array. Die Funktion liest die Werte von der Tastatur ein und speichert Sie in das Array. Schreibe 2 Versionen für eine zeilen- bzw. spaltenweise Eingabe.

2. Schreibe eine Funktion lnt2DArrayOut( ), welche die Elemente eines zweidimensionalen Integer-Arrays in Tabellenform auf dem Bildschirm ausgibt. Schreibe 2 Versionen für eine zeilenweise sowie spaltenweise Ausgabe. Die Arrayelemente sollen in der Funktion nicht verändert werden können!

3. Schreibe eine Funktion Spur( ), welche die Spur einer quadratischen Matrix berechnet. Die Spur ergibt sich aus der Summe aller Elemente der Hauptdiagonale. Die Matrixelemente sollen in der Funktion nicht verändert werden können!

4. Schreibe eine Funktion Hauptdiagonalen( ), welche alle Hauptdiagonalen einer quadratischen Matrix auf dem Bildschirm ausgibt. Die Matrixelemente sollen in der Funktion nicht verändert werden können!

5. Schreibe eine Funktion NebendiagonaIen( ), welche alle Nebendiagonalen einer quadratischen Matrix auf dem Bildschirm ausgibt. Die Matrixelemente sollen in der Funktion nicht verändert werden können!

6. Schreibe eine Funktion MulMatVect( ), welche eine quadratische Matrix mit einem Vektor entsprechender Größe multipliziert. Übergabeparameter sind die beiden Multiplikanden sowie ein Zielvektor, in welchem das Ergebnis abgelegt wird. Die Ausgangsmatrix sowie der Ausgangsvektor sollen in der Funktion nicht verändert werden können!

7. Schreibe die beiden Ein-/Ausgabefunktionen lnt2Darrayln() sowie lnt2DarrayOut() unter ausschließlicher Verwendung von Zeigern (der [ ]-Operator ist verboten!).

8. Schreibe eine einfache Version des Spieles Vier gewinnt. Verwende dazu den einfachen Textmodus sowie eine Spielfeldgröße von 7x6.

9. Das Spiel des Lebens:

   Wir denken uns ein unbegrenztes 2-dimensionales Raster, in dem eine primitive Lebensform ihr Leben fristet. Diese Lebensform setzt sich aus einzelnen Zellen zusammen. Jede einzelne Zelle entsteht und vergeht nach Prinzipien, die durch zwei Regeln beschrieben werden können:

   (a) Eine Zelle wird in einem Feld des Rasters geboren, wenn sie genau drei Eltern hat, d.h., wenn es genau drei angrenzende Felder mit Zellen hat.

   (b) Eine Zelle stirbt, wenn ihre Umgebung über- oder unterbevölkert ist, d.h., wenn es mehr als drei oder weniger als zwei angrenzende Felder mit lebenden Zellen gibt.

   Die Lebensform insgesamt entwickelt sich von Generation zu Generation. Ausgehend von einer bestehenden Generation wird nach den obigen Regeln eine neue Generation vollständig erzeugt. Erst dann ersetzt die neue Generation die alte Generation.

Anmerkungen:

- Definieren Sie für die Matrixdimensionen globale Konstanten
- Teste alle Funktionen innerhalb der Funktion main() aus.
- Verwende nirgends globale Variablen.
- Es dürfen nur selbstgeschriebene Funktionen verwendet werden.

Kapitel 10

Strukturen

Der Begriff Struktur hat für uns im folgenden drei Bedeutungen, die sich zwar stark überschneiden, die aber doch auseinanderzuhalten sind:

- die allgemeinste Bedeutung betrifft die Struktur als Oberbegriff für die vielfältigen Strukturierungsansätze in der Informatik (Programmstrukturen, Datenstrukturen, Kontrollstrukturen, ...).
- die speziellere betrifft die Datenstrukturen allgemein, für verschiedene Ansätze der Datenstrukturierung.
- die speziellste betrifft die spezielle Datenstruktur mit dem Namen Struktur (auch Record oder Verbund genannt) als eine Alternative zur Datenstruktur Vektor.

Ein **Vektor** (Array) ist eine homogene Datenstruktur, die verschiedene Datenelemente gleichen Typs zusammenfasst, ein einzelnes Datenelement wird dann über einen Index als Selektor ausgewählt:

```
double tab[20]   // Def. einer Vektor-Variablen

tab[i]= 3.2      //Zugriff auf das Element i
```

Eine Struktur ist dagegen eine im allgemeinen inhomogene Datenstruktur, die verschiedene Datenelemente unterschiedlichen Typs zusammenfassen kann, ein einzelnes Datenelement (häufig *Komponente* oder *Mitglied [engl, member]* genannt) wird dann über einen symbolischen Komponentennamen als Selektor ausgewählt:

Folgende Definition einer Struktur in C:

```
// Definition der Struktur
typedef struct
{
        long MatrikelNr:
        char Name[20];
        char Fachbereich;
} Student;

// Anwendung
Student s1,s2,s3;
```

```
s1.MatrikelNr=12345;
```

Als Alternative kann man auch folgende Definitionen in C verwenden:

```
// Definition der Struktur
struct S
{
        long MatrikelNr:
        char Name[20];
        char Fachbereich;
};

typedef struct S Student;

// Anwendung
struct Student s1,s2,s3;

s1.MatrikelNr=12345;

// Definition der Struktur
struct Student
{
        long MatrikelNr:
        char Name[20];
        char Fachbereich;
};

// Anwendung
struct Student s1,s2,s3;

s1.MatrikelNr=12345;
```

Folgende Definition einer Struktur gilt für C++:

```
// Definition der Struktur
struct Student
{
        long MatrikelNr:
        char Name[20];
        char Fachbereich;
};

// Anwendung
Student s1,s2,s3;

s1.MatrikelNr=12345;
```

Dabei ist **Student** das *sogenannte structure tag (Etikette)*, die Kombination **struct** Student kann als Typenbezeichner verwendet werden. IIN diesem Fall muss das Schlüsselwort **struct** bei der Definition der Variable mit verwendet werden, in C++ ist das überflüssig.

10.1 Arbeiten mit Strukturen

Strukturen können initialisiert werden, als Ganzes zugewiesen, als Parameter von Funktionen übergeben und als Funktionswert zurückgegeben werden:

```
struct complex
{
        double re,im;
}

complex x = {1.1, 2.2};   // x.re=1.1, x.im=2.2
complex y;
complex z;

y=x;
z.re=3.12;
z.im=4.5;
z=Add(x+y);

complex Add(complex c1, complex c2)
{
        complex temp;

        temp.re=c1.re+c2.re;
        temp.im=c1.im+c2.im;

        return temp;
}
```

Darüber hinaus ist es natürlich möglich, Zeiger auf Strukturen zu definieren und über Zeiger auf Strukturen zuzugreifen (Zeigersemantik siehe 8):

```
complex x;
complex *px;

px = & x;
(*px).re=1.1;     // d.h. x.re=1.1
(*px).im=2.2;     // d.h. x.im=2.2
```

Die Klammern sind wegen der hohen Priorität des Punktoperators erforderlich, der Ausdruck *px. re=1.1 wäre falsch! Da die geklammerte Schreibweise etwas umständlich ist, hat man dafür eine Abkürzung unter Verwendung des Operators *−> (Pfeiloperator)* eingeführt:

- (*Strukturzeiger).Komponente ≡ Strukturzeiger−>Komponente

Strukturzeiger wird hier als Kurzbezeichnung für *Zeiger auf Struktur* verwendet. Damit lassen sich die Zuweisungen wie folgt formulieren:

```
px->re=1.1;       // == (*px).re=1.1
px->im=2.2;       // == (*px).im=2.2
```

10.2 Rekursive Struktur

Strukturen, die eine oder mehrere Komponenten enthalten, deren Typ direkt oder indirekt über den eigenen Strukturtyp definiert sind, heißen rekursive Strukturen.

Das klassische Beispiel hierfür ist ein *Listenknoten*, der aus dem eigentlichen Inhalt - z.B. einem Namen - und aus einem Zeiger auf einen Listenknoten gleichen Typs besteht:

Folgendes Beispiel gibt die Definition eines Listenknoten in C++ an:

```
struct ListNode
{
        char Name[20];

        ListNode* pNext;
```

Folgendes Beispiel gibt die Definition eines Listenknoten in C an:

```
typedef struct ListNode PointerToListNode;

struct ListNode
{
        char Name[20];

        PointerToListNode* pNext;
```

Solche rekursive Strukturen spielen in der Informatik bei der Definition von Listen- und Baumstrukturen eine wichtige Rolle, das Thema wird später ausführlich behandelt und wird hier deshalb nicht weiter vertieft.

10.3 Schachtelung von Datenstrukturen

Das Prinzip der Schachtelung spielt in der Informatik eine große Rolle, und auch bei der Beschreibung der Elemente der Programmiersprache C sind wir schon mehrfach auf das Prinzip gestoßen: Ausdrücke und Anweisungen können geschachtelt werden. Das gleiche Prinzip gilt nun auch bei der Definition von Datenstrukturen:

- Datenstrukturen können geschachtelt werden, d.h. z.B.
  - die Elemente eines Vektors können auch Vektoren oder Strukturen sein.
  - Sie Elemente einer Struktur können auch Strukturen oder Vektoren sein.
- Die Schachtelungstiefe ist formal nicht begrenzt (sondern nur durch die Größe des verfügbaren Arbeitsspeichers begrenzt):

Hier einige typische Beispiele:

```
// Vektor aus Vektoren
typedef int vector1[10];          // eindim. Vektor
typedef vector1 vector2[5];       // zweidim. Vektor
vector2 vector3[20];              // dreidim. Vektor
```

```
vector3[i][j][k]= ....;

// Vektor aus Strukturen
struct complex
{
        double re,im;
};

complex vector[100];

vector[25].re=....;

// Struktur aus Vektoren und Strukturen
struct complex
{
        double re,im;
};

struct Datum
{
        int tag,monat,jahr;
}

struct Messwert
{
        char Name[20];
        Datum, dat;
        complex wert;
}

Messwert m;

strcpy(m.Name,"3. Messwert");
m.dat.tag=29;
m.dat.monat=8;
m.dat.jahr=2012;
m.wert.re=1.1;
m.wert.im=2.2;
```

10.4 Initialisierung von Strukturen

Bei Strukturen, deren Komponenten nichtstrukturierte Datentypen sind, erfolgt die Zuordnung der Anfangswerte zu den Komponenten gemäß der Reihenfolge, d.h. die i-te Komponente wird mit dem i-ten Anfangswert initialisiert:

```
struct datum{int tag,monat,jahr;};
datum d={29,8,2012};
```

Und wie ist das bei geschachtelten Datenstrukturen? Es gilt das gleiche Prinzip, wobei zusätzlich zur Verbesserung der Lesbarkeit geschweifte Klammerpaare verwendet werden können, z.B.

```
struct complex
{
        double re,im;
};

struct  Datum
{
        int tag,monat,jahr;
};

struct Messwert
{
        char Name [20] ;
        Datum dat;
        complex wert;
};

Messwert m = {"1. Messung",{29,8,2012 },{ 0.0,0.0}};
```

Ganz entsprechend kann auch ein Vektor initialisiert werden, dessen Elemente Strukturen sind:

```
struct complex
{
        double re,im;
};

complex x[5]= {{0.0,1.1},{2.2,3.3},{4.4,5.5},{6.6,7.7},{8.8,9.9},};
```

10.5 Programmbeispiel Wortliste, Version 2

Das Programm Wortliste soll jetzt etwas erweitert werden, und zwar wird in der neuen Version nicht nur eine sortierte Liste aller in der gelesenen Textdatei gefundenen Wörter erzeugt, sondern zusätzlich zu jedem Wort wird auch angegeben, wie häufig das Wort im Text vorkommt. Die Datenstrukturen für die Einträge und für die Gesamtliste können dann wie folgt definiert und benutzt werden:

```
struct Entry
{
        Word wo;
        int  count;
};

Entry list[ListLen];

strcpy(list[i].wo,"Hallo");
list[i].count=2;
```

10.6 Bitfelder

Die Komponenten von Strukturen können elementare und selbstdefinierte Datentypen sein. Darüber hinaus gibt es - z.B. aus Gründen einer möglichst effektiven Speicherausnutzung - auch die Möglichkeit, dass Komponenten einzelne Bits oder Bitgruppen repräsentieren:

```
struct Datum
{
        unsigned int tag        : 5;    //0 ... 31
        unsigned int monat      : 4;    // 0 ... 15
        unsigned int jahr       : 12;   // 0 ... 4095
};

Datum d;
d.tag=29;
d.monat=8;
d.jahr=2012;
```

Die entsprechenden Komponenten müssen vom Typ **int**, **unsigned int** ≡ **unsigned** oder **signed int** ≡ **signed** sein, hinter dem Doppelpunkt wird die Anzahl der benutzbaren Bits angegeben.

Achtung: Die Implementierung von Bitfeldern, d.h. ihre Abbildung auf den Speicher, ist durch den ANSI/ISO-Standard nicht definiert, sie ist also implementierungsabhängig!

10.7 Programmierbeispiel mit Strukturen

10.7.1 Übungen

Definieren Sie einen neuen Datentyp SCHUELER mit zumindest folgenden Elementen:

- Vorname
- Nachname
- Geschlecht (am besten Aufzähltyp verwenden, siehe Schlüsselwort enum)
- Geburtsdatum (eigene Struktur GEBDATUM mit Elementen Jahr, Monat, Tag)
- Klasse

Weitere Elemente nach eigenem Ermessen. Schreibe die Definition des neuen Datentyps für C sowie C++.

1. Schreibe eine Funktion SchuelerEingabe() zur Eingabe der Elemente einer Struktur des Typs SCHUELER über die Tastatur. Die mit Werten zu belegende Struktur soll entsprechend als Parameter übergeben werden. Schreiben Sie 2 Varianten dieser Funktion:
   - Übergabe der Struktur unter Verwendung des Referenzoperators
   - Übergabe eines Zeigers auf die Struktur
2. Schreibe eine Funktion SchuelerAusgabe(), welche die Elemente einer Struktur am Bildschirm entsprechend ausgibt. Die auszugebende Struktur wird als gewöhnlicher Parameter übergeben.
3. Schreibe eine Funktion SchuelerLoeschen(), welche die Elemente einer Struktur löscht. Wähle sinnvolle Werte als Kennung dafür, dass eine Struktur sozusagen "leer" ist. Versuche die Funktion memset() zu verwenden (siehe Hilfe).
4. Schreibe ein einfaches Programm zur Verwaltung einer Schülerliste, welches mit einem eindimensionalen Array von Elementen des DatenTyps SCHUELER (max. 100 Einträge) arbeitet. Verlagere alle Teilaufgaben in eigene Funktionen.

   Folgende Operationen sollen über ein einfaches Menü aufgerufen werden können:
   - Einfügen eines neuen Schülers in die Schülerliste:
     Das Einfügen soll an der ersten Stelle im Array erfolgen, an der eine "leere" Schülerstruktur gefunden wird
   - Suchen nach einem bestimmten Schüler in der Schülerliste (Suchkriterium Nachname)
   - Löschen eines bestimmten Schülers aus der Schülerliste (Durch "Leeren" der Struktur)
   - Durchblättern der Schülerliste

Sonderaufgaben:

- Schreiben Sie eine Funktion zum Sortieren der Schülerliste!
- Erweiterte Suchfunktionalität (Wildcards?), wobei innerhalb der gefundenen Schüler geblättert werden kann

Anmerkungen:

- Teste die Funktionen 1.-3. innerhalb der Funktion main() aus
- Verwende keine globalen Variablen
- Versuche alle Funktionen so allgemeingültig wie möglich zu schreiben
- Definiere falls möglich Konstanten
- Achte auf eine gute Formatierung des Sourcecodes

Kapitel 11

Vereinigungen (Unionen)

Vereinigungen (Unionen) gleichen syntaktisch den Strukturen, sie unterscheiden sich in der Semantik:

```
union XYZ
{
        typeA a;
        typeB b;

        typeN n;
};

XYZ x;
typeA aa;
typeB bb;

typeN nn;

// Im Falle einer Struktur werden hier alle
// Werte gespeichert werden, in diesem Falle
// wird bei jeder Zuweisung der vorherige
// Wert überschrieben
x.a=aa;
x.b=bb;

x.n=nn;
```

Es seien typeA, typeB und typeN irgendwelche vordefinierte oder selbstdefinierte Typbezeichner. Im Falle einer Struktur hätte eine Variable dieses Typs entsprechend viele Komponenten, alle können unabhängig voneinander mit einen Wert belegt werden. Im Falle einer Union gibt es nur eine gemeinsame Komponente, die zu jeder Zeit auch nur einen Wert haben kann, wobei aber zu verschiedenen Zeitpunkten jeweils unterschiedliche Datentypen abgelegt sein können. Bei der Abbildung der gemeinsamen Komponente auf den Arbeitsspeicher wird der Speicherplatz so ausgelegt, dass jeder der angegebenen Datentypen dort abgelegt werden kann. Eine Union entspricht einem Varianten Record der Sprachen Pascal und Modula-2.

Kapitel 12

ASCII-Tabelle

dez.	hex.	oct.	char	dez.	hex.	oct.	char	dez.	hex.	oct.	char	dez.	hex.	oct.	char
0	0x00	000	NUL	32	0x20	040	SP	64	0x40	100	@	96	0x60	140	‘
1	0x01	001	SOH	33	0x21	041	!	65	0x41	101	A	97	0x61	141	a
2	0x02	002	STX	34	0x22	042	“	66	0x42	102	B	98	0x62	142	b
3	0x03	003	ETX	35	0x23	043	#	67	0x43	103	C	99	0x63	143	c
4	0x04	004	EOT	36	0x24	044	$	68	0x44	104	D	100	0x64	144	d
5	0x05	005	ENQ	37	0x25	045	%	69	0x45	105	E	101	0x65	145	e
6	0x06	006	ACK	38	0x26	046	&	70	0x46	106	F	102	0x66	146	f
7	0x07	007	BEL	39	0x27	047	’	71	0x47	107	G	103	0x67	147	g
8	0x08	010	BS	40	0x28	050	(	72	0x48	110	H	104	0x68	150	h
9	0x09	011	TAB	41	0x29	051	)	73	0x49	111	I	105	0x69	151	i
10	0x0A	012	LF	42	0x2A	052	*	74	0x4A	112	J	106	0x6A	152	j
11	0x0B	013	VT	43	0x2B	053	+	75	0x4B	113	K	107	0x6B	153	k
12	0x0C	014	FF	44	0x2C	054	,	76	0x4C	114	L	108	0x6C	154	l
13	0x0D	015	CR	45	0x2D	055	-	77	0x4D	115	M	109	0x6D	155	m
14	0x0E	016	SO	46	0x2E	056	.	78	0x4E	116	N	110	0x6E	156	n
15	0x0F	017	SI	47	0x2F	057	/	79	0x4F	117	O	111	0x6F	157	o
16	0x10	020	DLE	48	0x30	060	0	80	0x50	120	P	112	0x70	160	p
17	0x11	021	DC1	49	0x31	061	1	81	0x51	121	Q	113	0x71	161	q
18	0x12	022	DC2	50	0x32	062	2	82	0x52	122	R	114	0x72	162	r
19	0x13	023	DC3	51	0x33	063	3	83	0x53	123	S	115	0x73	163	s
20	0x14	024	DC4	52	0x34	064	4	84	0x54	124	T	116	0x74	164	t
21	0x15	025	NAK	53	0x35	065	5	85	0x55	125	U	117	0x75	165	u
22	0x16	026	SYN	54	0x36	066	6	86	0x56	126	V	118	0x76	166	v
23	0x17	027	ETB	55	0x37	067	7	87	0x57	127	W	119	0x77	167	w
24	0x18	030	CAN	56	0x38	070	8	88	0x58	130	X	120	0x78	170	x
25	0x19	031	EM	57	0x39	071	9	89	0x59	131	Y	121	0x79	171	y
26	0x1A	032	SUB	58	0x3A	072	:	90	0x5A	132	Z	122	0x7A	172	z
27	0x1B	033	ESC	59	0x3B	073	;	91	0x5B	133	[	123	0x7B	173	{
28	0x1C	034	FS	60	0x3C	074	«	92	0x5C	134	\	124	0x7C	174	|
29	0x1D	035	GS	61	0x3D	075	=	93	0x5D	135	]	125	0x7D	175	}
30	0x1E	036	RS	62	0x3E	076	»	94	0x5E	136	ˆ	126	0x7E	176	-
31	0x1F	037	US	63	0x3F	077	?	95	0x5F	137	_	127	0x7F	177	DEL