7 Zeiger

Die Länge von mehrdimensionalen Feldern muß zur Compilierzeit bekannt sein, es sei denn sie werden dynamisch über Zeiger initialisiert. Die interne Darstellung zwei- und höherdimensionaler Felder erfolgt zeilenweise als langer Vektor.

7.1 Vektoren

Vektoren:

***typ var* [10];** vereinbart einen Vektor der Länge 10 vom Typ *typ*. Die Elemente sind *var*[0], *var*[1], ..., *var*[9].

Initialisierung von Vektoren:

***typ var* [3] = {1, 2, 3 };**

***typ var* [] = {1, 2, 3, 4, 7 };** werden bei vorgegebener Dimension zu wenig Initialisierungswerte angegeben, so werden die fehlenden auf 0 gesetzt. Fehlt die Dimensionsangabe, so wird sie aus der Initialisierung berechnet. Zu viele Initialisierungswerte bewirken einen Fehler, ebenso der Zugriff auf Elemente außerhalb des definierten Dimensionsbereichs.

Listing 33: Vektoren (vec1.c)

```
#include <stdio.h>

int main(void) {
int i, n = 3;
int a2[2] = { 7 };
int a1[2] = { 12, 4};
for ( i = 0; i <= n; i++ )
        printf("%2d: %5d, %5d\n", i, a1[i], a2[i]);
return 0;
}
```

liefert:

```
 0:    12,     7
 1:     4,     0
 2:     3,    12
 3:     3,     4
```

7.2 Matrizen

Matrizen:

***typ var*[10][15];** vereinbart eine 10 x 15 Matrix vom Typ *typ*. Die Feldelemente beginnen mit *var*[0][0].

Initialisierung von Matrizen:

***typ var* [3][2] = { { 1, 2}, { 3, 4}, { 3, 6}};**

7.3 Höherdimensionale Felder

Höherdimensionale Felder:

***typ var* [10][15] ... [23];** vereinbart wird ein Feld vom Typ *typ*, mit Indizes i_1: 0..9, i_2: 0 ... 14, ..., i_n: 0 ... 22. Die Feldelemente werden mit *var*[i_1][i_2]...[i_n] angesprochen.

Dimension in bytes:

sizeof(*array*); Größe eines Feldes *array* (oder einer Variablen) in bytes. Der Typ von `sizeof` ist `size_t`, dieser Typ wird im header <`stddef.h`> vereinbart und kann in `int` umgewandelt werden.

7.4 Interne Darstellung

Ein Feld *v*`[11]` wird in aufeinenderfolgenden Speicherplätzen abgelegt. Dabei ist *v* ein Pointer, der die Adresse des ersten Elements *v*`[0]` enthält; *v*`[10]` entspricht $*$(*v*`+10`). Mit *v*`[11] = 1;` wird die Adresse hinter (*v*`+10`) mit 1 belegt. So etwas kann zu Fehlern führen. Ein Matrixelement *m*`[2][10]` wird durch $*$($*$(*m*`+2`)`+10`) dargestellt. Der Typ von *m* ist Zeiger auf einen Zeiger; denn $*$ erscheint zweimal. Demnach ist $*$(*m*`+2`) ein einfacher Zeiger von dem Datentyp, der mit *m* verbunden ist. Nehmen wir `int` $**$*m*; an. Dann ist $*$(*m*`+2`) ein Zeiger auf `int` und *m* ein Vektor mit Zeigern. Die Bedeutung des ganzen Ausdrucks ist: in dem Vektor von Zeigern *m* wird der dritte (die Indizierung beginnt ja mit 0!) Zeiger ausgewählt, aus dem Vektor, auf den er zeigt wird das 11-te Element angesprochen.

```
/* short belegen  2 bytes */
short mat[2][5] = { { 10, 11, 1, 3, 4},
                    { 20, 21, 2, 4, 5}};
```

Listing 34: Speicher (tt.c)

```
#include <stdio.h>
int main(void) {
short mat[2][5] = { { 10, 11, 1, 3, 4},
                    {20, 21, 2, 4, 5}};
printf("------------------------\n");
printf("    Adresse \t | Inhalt\n");
printf("------------------------\n");
printf("  %lu\t | %2hi \n", *mat, *(*mat));
printf("  %lu\t | %2hi \n", *mat+1, *((*mat)+1));
printf("  %lu\t | %2hi \n", *mat+2, *((*mat)+2));
printf("  %lu\t | %2hi \n", *mat+3, *((*mat)+3));
printf("  %lu\t | %2hi \n", *mat+4, *((*mat)+4));
printf("  %lu\t | %2hi \n", *(mat+1), *(*(mat+1)));
printf("  %lu\t | %2hi \n", *(mat+1)+1, *(*(mat+1)+1));
printf("  %lu\t | %2hi \n", *(mat+1)+2, *(*(mat+1)+2));
printf("  %lu\t | %2hi \n", *(mat+1)+3, *(*(mat+1)+3));
printf("  %lu\t | %2hi \n", *(mat+1)+4, *(*(mat+1)+4));
printf("------------------------\n");
return 0; }
```

Adresse		Inhalt	
*mat, mat[0]	4290768148	10	mat[0][0], *(*mat)
(*mat)+1	4290768150	11	mat[0][1], *((*mat)+1)
	4290768152	1	
	4290768154	3	
	4290768156	4	
(mat+1), mat[1]	4290768158	20	mat[1][0], *(*(mat+1))
(mat+1)+1	4290768160	21	mat[1][1], *(*(mat+1)+1)
(mat+1)+2	4290768162	2	mat[1][2], *(*(mat+1)+2)
	4290768164	4	
	4290768166	5	

7.5 Zeiger

Ein Zeiger (pointer) ist eine Variable, die auf einen Speicherplatz zeigt, dessen Datentyp bekannt ist. Sei i eine Variable vom Typ `int`. Dann gibt es

irgendwo im Speicher des Rechners einen Ort, an dem der Wert von *i* gespeichert wird. Die Position (Speicheradresse, Zeigerinformation) dieser Wertezelle erhält man durch Anwendung des Adreßoperator auf *i*: `&`*i*.
Hat man die Zeigerinformation zur Verfügung, so erhält man den an der entsprechenden Position abgespeicherten Wert durch Anwendung des Zeiger- oder Inhaltsoperators `*`: `*(&`*i*`)`.
Zeigerwertige Variablen werden kurz als Zeiger (pointer) bezeichnet. Ein Zeiger enthält die Adresse einer Variablen vom gleichen Typ. Ein Zeiger ist also ein Objekt mit dessen Hilfe auf andere Objekte zugegriffen werden kann. Ist etwa *c* ein Objekt vom Typ *typ* und *pc* ein Objekt vom Typ Zeiger auf *typ*, so bewirkt die Zuweisung *pc* `= &`*c*`;` daß nun *pc* auf *c* weist.
Mit der symbolischen Konstanten `NULL` kann ein Zeiger belegt werden, wenn er auf keine Adresse verweist.

Zeigervereinbarung:

typ *var; typ **qar; vereinbart die Variable *var* als Zeiger, *var* enthält somit eine Adresse, mit `*`*var* erhält man den Inhalt dieser Adresse, der vom Typ *typ* ist, entsprechend ist *qar* ein Zeiger auf einen Zeiger, `*`*qar* ist somit eine Adresse, deren Inhalt vom Typ *typ* ist, dieser Wert kann mit `**`*qar* abgerufen werden.

Adreßoperator &:

&var gibt die Speicheradresse von *var* zurück.

Inhaltsoperator *:

****var*** Inhalt der Adresse, auf die *var* zeigt.

Beispiele:

```
int x = 4711, y= 2;
int *px;
px = &x;        /* px zeigt auf x */
y = *px;        /* y wird der Inhalt der Adresse px
                   zugewiesen, y = 4711 */
*px = 0;        /* Unter der Adresse px wird 0 abgelegt,
                   also x = 0 */
```

Das Programm

Listing 35: Pointer (pointer1.c)

```
#include <stdio.h>
#include <stddef.h>
#define UI (unsigned)

int main(void)
{
short a = 29, b = 47;
short *pa, *pb, **ppa;
int c = 47114711;
int *pc;
pa = &a;
pb = &b;
pc = &c;
ppa = &pa;
printf("   pa: Hex-Adr. %x, bytes: %2d,  *pa: %9d \n",
       UI pa, (int) sizeof(*pa), *pa);
printf("   pb: Hex-Adr. %x, bytes: %2d,  *pb: %9d \n",
       UI pb, (int) sizeof(*pb), *pb);
printf("   pc: Hex-Adr. %x, bytes: %2d,  *pc: %9d \n",
       UI pc, (int) sizeof(*pc), *pc);
printf("  ppa: Hex-Adr. %x, bytes: %2d, *ppa: %9x,"
           "**ppa: %5d\n",
           UI &ppa, (int) sizeof(ppa), UI *ppa, **ppa);
printf("  &pa: Hex-Adr. %x, bytes: %2d\n", UI &pa,
       (int) sizeof(&pa));
printf("  &pb: Hex-Adr. %x, bytes: %2d\n", UI &pb,
       (int) sizeof(&pb));
printf("  &pc: Hex-Adr. %x, bytes: %2d\n", UI &pc,
       (int) sizeof(&pc));
return 0;
}
```

liefert

```
   pa: Hex-Adr. ffbfed46, bytes:  2,  *pa:        29
   pb: Hex-Adr. ffbfed44, bytes:  2,  *pb:        47
   pc: Hex-Adr. ffbfed34, bytes:  4,  *pc:  47114711
  ppa: Hex-Adr. ffbfed38, bytes:  4, *ppa:  ffbfed46,**ppa: 29
  &pa: Hex-Adr. ffbfed40, bytes:  4
  &pb: Hex-Adr. ffbfed3c, bytes:  4
  &pc: Hex-Adr. ffbfed30, bytes:  4
```

wobei die Speicheradressen auch anders lauten können.
Mit der Typenvereinbarung ist die Adresse auf die ein Zeiger verweist nicht belegt. Die Initialisierung darf nicht so erfolgen.

```
# include <stdio.h>
int main(void)
{
int *pholz = 10;
printf("%d\n", *pholz);
return 0;
}
```

Es wird ein bus-error gemeldet. Aber auch nicht so

```
#include <stdio.h>

int main(void)
{
int holz = 10, *pholz;
*pholz = holz; /* falsche Initialisierung */
printf("xylos = %d, Adresse = %d\n", *pholz, holz);
return 0;
}
```

Hier wird man einen segmentation fault erhalten, falls der Zeiger in den Speicherbereich des Betriebssystems reicht, kann ein Systemabsturz erfolgen.

Ein zulässige Initialisierung kann z.B. so erfolgen.

```
#include <stdio.h>

int main(void)
{
int holz = 10, *pholz;
pholz = &holz; /* Initialisierung */
printf("xylos = %d, Adresse = %x\n", *pholz, (unsigned int) pholz);
return 0;
}
```

7.6 Felder und Zeiger

Felder vom Typ *typ* sind kein neuer Typ von Objekten, vielmehr sind sie Zeiger vom Typ *typ*. So sind für einen Vektor *a a[10]* und **(a+10)* äquivalent. Weiter entspricht **a* der Schreibweise *a[0]*.
Ein Feld *a*`[10]` wird in eine Funktion übergeben, indem der Feldname *a*, ein Pointer, übergeben wird. Die Elemente können dann entweder durch *a[i]* oder durch **(a+i)* angesprochen werden. Da über den Feldnamen ein direkter Zugriff auf die Feldelemente möglich ist, sollte man Felder, die nicht verändert werden sollen, mit der zusätzlichen Option `constant` vom Typ her vereinbaren, um ungewollte Änderungen zu vermeiden.

Listing 36: Pointer (pointer2.c)

```
#include <stdio.h>
int main() {
void square(int[], int[], int);
int i, n = 3;
int arr[3] = { 4, 7, 3 }, arr2[3];
square(arr, arr2, n);
for (i = 0; i <= n - 1; i++)
        printf("%i: %i, %i\n", i, arr[i], arr2[i]);
return 0; }

void square(int arr[], int s[], int n) {
        int i;
        for (i = 0; i <= n - 1; i++) {
                s[i] = arr[i] * arr[i];
        } }
```

erhält in reiner Zeiger-Schreibweise die Form

Listing 37: Pointer (pointer3.c)

```
#include <stdio.h>
int main(void) {
void square(int *, int *, int);
int i, n = 3;
int arr[3] = { 4, 7, 3 }, arr2[3];
square(arr, arr2, n);
for (i = 0; i <= n - 1; i++)
        printf("%i: %i, %i\n", i, *(arr+i), *(arr2+i));
return 0;
}

void square(int *arr, int *s, int n) {
int i;
for (i = 0; i <= n - 1; i++) {
        *(s + i) = *(arr + i) * *(arr + i);
} }
```

7.7 Dynamische Speicherbelegung

Für Vektoren und Matrizen ist eine dynamische Speicherbelegung zur Laufzeit vorteilhaft. Dazu benötigt man folgende Funktionen aus stdlib.h.

Belegen:

void *malloc(size_t *n*); liefert einen Zeiger auf einen (noch nicht initialisierten) Speicherbereich von n bytes, falls das nicht möglich ist wird `NULL` zurückgegeben. Der Zeiger muß noch durch ein casting auf den richtigen Datentyp umgewandelt werden.

Belegen:

void *calloc(size_t *n*, size_t *m*); liefert einen Zeiger auf einen Speicherbereich von n Objekten mit m bytes, der mit 0 initialisiert wird. Falls das nicht möglich ist, wird `NULL` zurück gegeben. Der Zeiger muß noch durch ein casting auf den richtigen Datentyp umgewandelt werden.

Zurückgeben:

free(*p*); der Speicher, auf den p zeigt und der mit `malloc` oder `calloc` angelegt wurde, wird freigegeben.

Listing 38: Random (rand2.c)

```
#include <stdlib.h>
#include <stdio.h>
int *dvector(int low,int high) {
int *v;
v = (int *) calloc((size_t) (high -low+1),
                sizeof(int));
if (!v) {
    printf("allocation failure\n");
        exit(0); }
v -= low;
return v;
}

void printdvector(int n, int *v, char c) {
int i;
printf("%c = (",c);
for (i = 1; i <= n; i++)
        printf("%d, ",v[i]);
printf("\b\b)\n"); }

int main() {
int *v, i, n, erz;
printf("Eingabe Dimension (int): ");
scanf("%d", &n);
printf("Erzeuger der Zufallszahlen (int): ");
scanf("%d", &erz);
srand(erz);
/* Zufallszahlen zwischen 1 und 49 */
v = dvector(1,n);
for (i = 1; i <= n; i++)
        v[i] = (rand() % 49+1);
printdvector(n,v,'v');
free(v);
return 0; }
```

7.8 Nochmals Matrizen

Um mit Matrizen einfach arbeiten zu können, kann man eine dynamische Speicherverwaltung vornehmen, bei der Matrizen als Zeiger von Zeigern behandelt werden.

Listing 39: Matrix Multiplikation (mat1.c)

```
#include <stdio.h>
#include <stdlib.h>
#define UI (unsigned)
#define DB double
#define SO sizeof

int main() {

DB **dmatrix(int, int, int, int);
DB **dmatmult(int, DB**, DB**);
void printmat(int, DB**, char);

DB a[2][2] = { {1.0, 1.0}, {2.0, 1.0} };
DB b[2][2] = { {0.0, 1.0}, {1.0, 1.0} };
DB **w, **m, **n;
int i, j;

m = (DB **) dmatrix(1,2,1,2);
n = (DB **) dmatrix(1,2,1,2);
for (i = 1; i <= 2; i++) {
        for (j = 1; j <= 2; j++) {
                m[i][j] = a[i-1][j-1];
                n[i][j] = b[i-1][j-1];
        }
}
printmat(2,m,'m');
printmat(2,n,'n');
w = (DB **) dmatmult(2,m,n);
printmat(2,w,'w');
return 0;
}
```

```
DB **dmatrix(int rlow,int rhigh,int clow,int chigh)
{
void* malloc(size_t);
int i;
DB **m;
m = (DB **) malloc( UI (rhigh -rlow+1)*SO(DB*));
if ( !m ) {
        printf("allocation failure");
        exit(0);
        }
m -= rlow;
for (i = rlow; i <= rhigh; i++) {
        m[i] = (DB *) malloc(UI (chigh-clow+1)*SO(DB));
        if ( (!m[i]) ) {
                printf("allocation failure");
                exit(0);
                }
        m[i] -= clow;
}
return m;
}

DB **dmatmult(int n, DB **a, DB**b) {
int i, j, k;
DB **s, t;
s = (DB **) dmatrix(1,n,1,n);
for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
                t = 0;
                for (k=1; k <= n; k++) {
                        t += a[i][k]*b[k][j];
                }
                s[i][j] = t;
        }
}
return s; }
```

```
void printmat(int n, DB **a, char c) {
int i, j;

printf("\n%c =\n",c);
for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
                printf("%f ",a[i][j]);
        }
        printf("\n");
        }
}
```