Kapitel 8

Zeiger (Pointer)

Bislang war beim Zugriff auf eine Variable nur ihr Inhalt von Interesse. Dabei war es unwichtig, wo (an welcher Speicheradresse) der Inhalt abgelegt wurde. Ein neuer Variablentyp, der Pointer (Zeiger), speichert Adressen unter Berücksichtigung des dort abgelegten Datentyps.

8.1 Adressen

Das folgende Programm demonstriert, wie man Speicheradressen von Variablen ermittelt.

Beispiel 8.1 Adressen von Variablen.

```
#include <stdio.h>

int main()
{
        int a=16;
        int b=4;
        double f=1.23;
        float g=5.23;

        /* Formatangabe %u steht f\"ur unsigned int */
        /* &a = Adresse von a */
        printf("Wert von a = %i, \t Adresse von a = %u\n",a,(unsigned int)&a);
        printf("Wert von b = %i, \t Adresse von b = %u\n",b,(unsigned int)&b);
        printf("Wert von f = %f, \t Adresse von f = %u\n",f,(unsigned int)&f);
        printf("Wert von g = %f, \t Adresse von g = %u\n",g,(unsigned int)&g);
        return 0;
}
```

Nach dem Start des Programms erscheint folgende Ausgabe:

```
Wert von a = 16,       Adresse von a = 2289604
Wert von b = 4,        Adresse von b = 2289600
Wert von f = 1.230000,  Adresse von f = 2289592
Wert von g = 5.230000,  Adresse von g = 2289588
```

□

Bemerkung 8.2 Zu Beispiel 8.1.

1.) Dieses Programm zeigt die Werte und die Adressen der Variablen `a,b,f,g` an. (Die Adressangaben sind abhängig vom System und Compiler und variieren dementsprechend).

2.) Der Adressoperator im `printf()`–Befehl sorgt dafür, dass nicht der Inhalt der jeweiligen Variable ausgegeben wird, sondern die Adresse der Variable im Speicher. Die Formatangabe `%u` dient zur Ausgabe von vorzeichenlosen Integerzahlen (`unsigned int`). Dieser Platzhalter ist hier nötig, da der gesamte Wertebereich der Ganzzahl ausgeschöpft werden soll und negative Adressen nicht sinnvoll sind.

3.) In der letzten Zeile wird angegeben, dass die Variable `g` auf der Speicheradresse 2289588 liegt und die Variable `f` auf der Adresse 2289592. Die Differenz beruht auf der Tatsache, dass die Variable `g` vom Typ `float` zur Speicherung `sizeof(float)=4` Bytes benötigt. Auch bei den anderen Variablen kann man erkennen, wieviel Speicherplatz sie aufgrund ihres Datentyps benötigen.

□

8.2 Pointervariablen

Eine Pointervariable (Zeigervariable) ist eine Variable, deren Wert (Inhalt) eine Adresse ist. Die Deklaration erfolgt durch:

```
Datentyp *Variablenname;
```

Das nächste Programm veranschaulicht diese Schreibweise.

Beispiel 8.3

```
#include <stdio.h>

int main()
{
       int a=16;
       int *pa;       /* Deklaration von int Zeiger pa - pa ist ein Zeiger auf
                       * eine Integer*/
       double f=1.23;
       double *pf;    /* Deklaration von double Zeiger pf */

       pa=&a; /* Zeiger pa wird die Adresse von a zugewiesen */
       pf=&f; /* Zeiger pf wird die Adresse von f zugewiesen */

       printf("Variable a : Inhalt = %i\t Adresse = %u\t Gr\"o\3e %i \n"
       ,a,(unsigned int)&a,sizeof(a));
       printf("Variable pa : Inhalt = %u\t Adresse = %u\t Gr\"o\3e %i \n"
       ,(unsigned int)pa,(unsigned int)&pa,sizeof(pa));
       printf("Variable f : Inhalt = %f\t Adresse = %u\t Gr\"o\3e %i \n"
       ,f,(unsigned int)&f,sizeof(f));
       printf("Variable pf : Inhalt = %u\t Adresse = %u\t Gr\"o\3e %i \n"
       ,(unsigned int)pf,(unsigned int)&pf,sizeof(pf));
       return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Variable a : Inhalt = 16      Adresse = 2289604     Gr\"o\3e 4
Variable pa : Inhalt = 2289604 Adresse = 2289600    Gr\"o\3e 4
Variable f : Inhalt = 1.230000 Adresse = 2289592    Gr\"o\3e 8
Variable pf : Inhalt = 2289592 Adresse = 2289588    Gr\"o\3e 4
```

□

Bemerkung 8.4 Zu Beispiel 8.3.

1.) Da Pointervariablen wieder eine Speicheradresse besitzen, ist die Definition eines Pointers auf einen Pointer nicht nur sinnvoll, sondern auch nützlich (siehe Beispiel 8.9).

2.) Die Größe des benötigten Speicherplatzes für einen Pointer ist unabhängig vom Typ der ihm zu Grunde liegt, da der Inhalt stets eine Adresse ist. Der hier verwendete Rechner (32-Bit-System) hat einen Speicherplatzbedarf von 4 Byte (= 32 Bit).

□

8.3 Adressoperator und Zugriffsoperator

Der unäre Adressoperator & (Referenzoperator)

```
& Variablenname
```

bestimmt die Adresse der Variable. Der unäre Zugriffsoperator * (Dereferenzoperator)

```
* pointer
```

erlaubt den (indirekten) Zugriff auf den Inhalt, auf den der Pointer zeigt. Die Daten können wie Variablen manipuliert werden.

Beispiel 8.5

```
#include <stdio.h>

int main()
{
       int a=16;
       int b;
       int *p; /* Deklaration von int Zeiger p */

       p=&a;  /* Zeiger p wird die Adresse von a zugewiesen */
       b=*p;  /* b = Wert unter Adresse p = a = 16 */

       printf("Wert von b = %i = %i = Wert von *p \n",b,*p);
       printf("Wert von a = %i = %i = Wert von *p \n",a,*p);
       printf("Adresse von a = %u = %u = Wert von p\n"
       ,(unsigned int)&a,(unsigned int)p);
       printf("Adresse von b = %u != %u = Wert von p\n\n"
       ,(unsigned int)&b,(unsigned int)p);

       *p=*p+2; /* Wert unter Adresse p wird um 2 erh\"oht */
              /* d.h. a=a+2                          */

       printf("Wert von b = %i != %i = Wert von *p \n",b,*p);
       printf("Wert von a = %i = %i = Wert von *p \n",a,*p);
       printf("Adresse von a = %u = %u = Wert von p\n"
       ,(unsigned int)&a,(unsigned int)p);
       printf("Adresse von b = %u != %u = Wert von p\n\n"
       ,(unsigned int)&b,(unsigned int)p);
       return 0;
}
```

Das Programm erzeugt folgende Ausgabe:

```
Wert von b = 16 = 16 = Wert von *p
Wert von a = 16 = 16 = Wert von *p
Adresse von a = 2289604 = 2289604 = Wert von p
Adresse von b = 2289600 != 2289604 = Wert von p

Wert von b = 16 != 18 = Wert von *p
Wert von a = 18 = 18 = Wert von *p
Adresse von a = 2289604 = 2289604 = Wert von p
Adresse von b = 2289600 != 2289604 = Wert von p
```

□

8.4 Zusammenhang zwischen Zeigern und Feldern

Felder nutzen das Modell des linearen Speichers, d.h. ein im Index nachfolgendes Element ist auch physisch im unmittelbar nachfolgenden Speicherbereich abgelegt. Dieser Fakt erlaubt die Interpretation von Zeigervariablen als Feldbezeichner und umgekehrt.

Beispiel 8.6 Zeiger und Felder.

```
#include <stdio.h>

int main()
{
        float ausgabe;
        float f[4]={1,2,3,4};
        float *pf;


        pf=f;  /* \"Aquivalent w\"are die Zuweisung pf=&f[0] */

        /* Nicht Zul\"assige Operationen mit Feldern */
        /* f=g;
         * f=f+1; */

        /* \"Aquivalente Zugriffe auf Feldelemente */
        ausgabe=f[3];
        ausgabe=*(f+3);
        ausgabe=pf[3];
        ausgabe=*(pf+3);
        return 0;
}
```

□

Bemerkung 8.7 Zu Beispiel 8.6.

1.) Das Beispiel zeigt, dass der Zugriff auf einzelne Feldelemente für Zeiger und Felder identisch ist, obwohl es sich um unterschiedliche Datentypen handelt.

2.) Die Arithmetischen Operatoren + und - haben bei Zeigern und Feldern auch den gleichen Effekt. Der Ausdruck pf + 3 liefert als Wert

(Adresse in pf) + 3 x sizeof(Typ)
(und nicht (Adresse in pf) + 3 !!!)

3.) Der Zuweisungsoperator = ist für Felder nicht anwendbar, d.h. f=ausdruck ist nicht zulässig. Einzelne Feldelemente können jedoch wie gewohnt manipuliert werden (z.B. f[2]=g[3] ist zulässig).
Die Zuweisung pf=pf+1 hingegen bewirkt, dass pf nun auf f[1] zeigt.

4.) Ein weiterer Unterschied zwischen Feldvariablen und Pointervariablen ist der benötigte Speicherplatz. Im Beipiel liefert sizeof(pf) den Wert 4 und sizeof(f) den Wert 16 (=4 x sizeof(float)).

Die folgenden Operatoren sind auf Zeiger anwendbar :

- Vergleichsoperatoren: ==, !=, <, >, <=, >=
- Addition + und Subtraktion -
- Inkrement ++, Dekrement -- und zusammengesetzte Operatoren +=, -=

8.5 Dynamische Felder mittels Zeiger

Bisher wurde die Länge von Feldern bereits bei der Deklaration bzw. Definition angegeben. Da viele Aufgaben und Probleme stets nach dem selben Prinzip ausgeführt werden können, möchte man die Feldlänge gerne als Parameter und nicht als feste Größe in die Programmierung einbeziehen. Die benötigten Datenobjekte werden dann in der entsprechenden Größe und damit mit entsprechend optimalem Speicherbedarf erzeugt.
Für Probleme dieser Art bietet C mehrere Funktionen (in der Headerdatei `malloc.h`), die den notwendigen Speicherplatz zur Laufzeit verwalten. Dazu zählen:

malloc()	reserviert Speicher einer bestimmten Größe
calloc()	reserviert Speicher einer bestimmten Größe und initialisiert die Feldelemente mit 0
realloc()	erweitert einen reservierten Speicherbereich
free()	gibt den Speicherbereich wieder frei

Die Funktionen `malloc()`, `calloc()` und `realloc()` versuchen, den angeforderten Speicher bereitzustellen (allokieren), und liefern einen Pointer auf diesen Bereich zurück. Konnte die Speicheranforderung nicht erfüllt werden, wird ein Null-Pointer (NULL in C, d.h. Pointer zeigt auf die 0 Adresse im Speicher) zurückliefert. Die Funktion `free()` enthält als Argument einen so definierten Pointer und gibt den zugehörigen Speicherbereich wieder frei.

Beispiel 8.8 Norm des Vektors (1,...,n). Das folgende Programm demonstriert die Reservierung von Speicher durch die Funktion `malloc()` und die Freigabe durch `free()`.

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
int main()
{
       int n, i;
       float *vektor, norm=0.0;

       printf("Geben Sie die Dimension n des Vektorraums an: ");
       scanf("%i",&n);

       /* Dynamische Speicher Reservierung */
```

```
        vektor = (float *) malloc(n*sizeof(float));

        if (vektor == NULL) /* Genuegend Speicher vorhanden? */
        {
                printf("Nicht genuegend Speicher vorhanden \n");
                return 1;
                /* Programm beendet sich und gibt einen Fehlerwert zurueck */
        }
        else
        {
                /* Initialisierung des Vektors */
                /* Norm des Vektors */
                for (i=0;i<n;i=i+1)
                {
                        vektor[i]=i+1;
                        norm=norm+vektor[i]*vektor[i];
                }
                norm=sqrt(norm);

                /* Freigabe des Speichers */
                free(vektor);
                printf("Die Norm des eingegebenen Vektors (1,...,%i) ist : %f \n"
                ,n,norm);
        }
        return 0;
}
```

□

Ein zweidimensionales dynamisches Feld läßt sich einerseits durch ein eindimensionales dynamisches Feld darstellen, als auch durch einen Zeiger auf ein Feld von Zeigern. Dies sieht für eine Matrix von m Zeilen und n Spalten wie folgt aus:

Beispiel 8.9 Dynamisches 2D Feld.

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

int main()
{
        int n,m,i,j;
        double **p;    /* Zeiger auf Zeiger vom Typ double */

        printf("Geben Sie die Anzahl der Zeilen der Matrix an: ");
        scanf("%i",&m);
        printf("Geben Sie die Anzahl der Spalten der Matrix an: ");
        scanf("%i",&n);

        /* Allokiert Speicher fuer Zeiger auf die Zeilen der Matrix */
        p=(double **) malloc(m*sizeof(double*));

        for (i=0;i<m;i++)
        {
                /* Allokiert Speicher fuer die Zeilen der Matrix */
                p[i]= (double *) malloc(n*sizeof(double));
        }

        for (i=0;i<m;i++) /* Initialisierung von Matrix p */
        {
```

```
                for (j=0;j<n;j++)
                {
                        p[i][j]=(i+1)*(j+1);
                        printf("%f ",p[i][j]);
                }
                printf("\n");
        }

        for (i=0;i<m;i++)
        {
                free(p[i]); /* Freigabe der Zeilen */
        }
        free(p);        /* Freigabe der Zeilenzeiger */
        return 0;
}
```

□

Zuerst muss der Speicher auf die Zeilenpointer allokiert werden, erst danach kann der Speicher für die einzelnen Zeilen angefordert werden. Beim Deallokieren des Speichers müssen ebenfalls alle Spalten und danach alle Zeilen wieder freigegeben werden. Für den Fall m=3 und n=4 veranschaulicht das Bild die Ablage der Daten im Speicher.

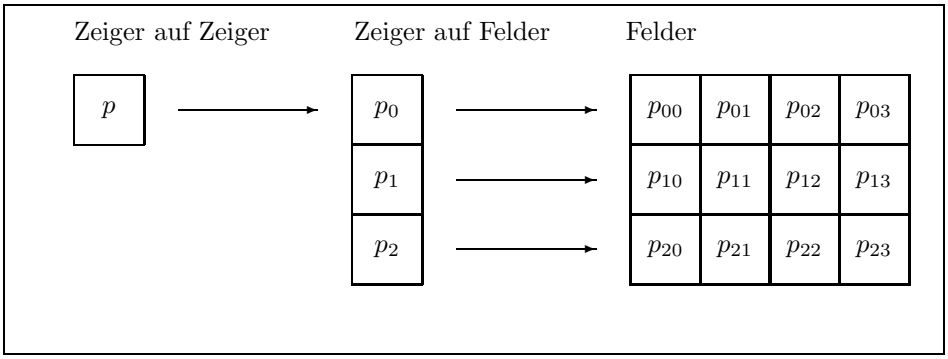

Abbildung 8.1: Dynamisches 2D Feld mit Zeiger auf Zeiger

Achtung ! Es gibt keine Garantie, dass die einzelnen Zeilen der Matrix hintereinander im Speicher angeordnet sind. Somit unterscheidet sich die Speicherung des dynamischen 2D-Feldes von der Speicherung des statischen 2D-Feldes (siehe Kapitel 6.3.2), obwohl die Syntax des Elementzugriffes p[i][j] identisch ist. Dafür ist diese Matrixspeicherung flexibler, da die Zeilen auch unterschiedliche Längen haben dürfen. Insbesondere findet das dynamische 2D-Feld Anwendung zur Speicherreservierung bei der Bearbeitung von dünnbesetzten Matrizen.

Will man sich sicher sein, dass die Matrixeinträge im Speicher hintereinander kommen, so allokiere man zuerst den Speicher für die gesamte Matrix und weise dann den Zeigern auf die Zeilen die entsprechenden Zeilenanfänge zu. (*Übungsaufgabe*)