4.6 Datenstrukturen/Zuweisung von Arrays an Zeiger

4.6.1 Eine merkwürdige Zuweisung ...

Der folgende Abschnitt eines C-Programms soll analysiert werden (`src/feld_an_pointer0.c`).

```
1 #include <stdio.h>
2 int main(void)
3 {
4     char string[80]="Dies ist eine Textzeile.";
5     char *p;
6     p = string;
7     printf("%s\n", p);
8     return 0;
9 }
```

In Zeile 6 fällt auf: Hier steht auf der rechten Seite der Zuweisung der Name eines Arrays. Auf der linken Seite steht eine Adressvariable (Zeiger). Es scheint so, dass der Inhalt des Arrays mit dem Namen `string` in den Zeiger `p` gefüllt wird. Aber geht das überhaupt? Zwei Argumente sprechen dagegen:

a) `string` ist ein Array, `p` eine Adressvariable (Zeiger). C ist eine typsichere Sprache, damit kann man nicht einfach dermaßen verschiedene Variablen einander zuweisen. Mindestens eine Typkonversion wäre nötig.

b) `string` ist 80 Bytes groß, `p` nur 4 Bytes (oder 8). Das kann gar nicht passen (andernfalls wäre das eine Revolution in der Technik der Datenkompression).

4.6.2 ... und die Auflösung des Rätsels

Die Lösung liegt in einer Besonderheit der Sprache C: An `p` wird in Wirklichkeit nur die Startadresse von `string` übergeben.

Mit diesem Programm kann man es überprüfen (`src/feld_an_pointer0ausg.c`):

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char string[80]="Dies ist eine Textzeile.";
5      char *p;
6      p = string;
7      printf("%s\n", p);
8      printf("Inhalt von p: %p\n", p);
9      printf("Startadresse von string: %p\n", &string[0]);
10     return 0;
11 }
```

Compilieren und Ausführen ergibt:

Terminal

```
schueler@debian964:~$ gcc src/feld_an_pointer0ausg.c
schueler@debian964:~$ ./a.out
Dies ist eine Textzeile.
Inhalt von p: 0xbf87c28c
Startadresse von string: 0xbf87c28c
```

Das Ergebnis lässt sich so formulieren: Wenn man einem Zeiger ein Array zuweist, dann erhält der Zeiger als Inhalt anschließend die Adresse des ersten Array-Elements. Der Array-Name (ohne die Index-Klammern) bezeichnet hier offenbar die Adresse des vordersten Elements. Dann sind die beiden folgenden Zeilen in C identisch:

```
1    p = &string[0]; /* <-- dies geht in jeder Sprache                */
2    p = string;     /* <-- diese Schreibweise funktioniert nur in C */
```

Allgemein gilt nur in C (und C++ und einigen verwandten Sprachen):
Nachdem in C ein Array vereinbart wurde, steht der Arrayname nur noch für die konstante Startadresse des Arrays.

Der Datentyp des Arraynamens ist damit der Datentyp der Startadresse: `string` hat (nach der Vereinbarung!) den Typ `char *` (eigentlich wäre es der Typ `const char *`; die Bedeutung von `const` ist *konstant* und wird später behandelt). Der Compiler hat keine Schwierigkeiten, die konstante Startadresse in die Adressvariable `p` zu kopieren, denn beide Datentypen sind ja (abgesehen vom Wort `const`) gleich.

4.6.3 Geht es auch umgekehrt?

Die umgekehrte Zuweisung ist übrigens nicht möglich: `string=p;` funktioniert nicht. Man kann die konstante Startadresse (die Adresskonstante) `string` in die Variable `p` kopieren. Aber man kann den Inhalt der Variablen `p` nicht in die Adresskonstante `string` übertragen. Denn das hieße, dass das Array `string` an eine neue Adresse verschoben würde. Aber das ist in C nicht vorgesehen: Jede Variable bleibt nach ihrer Vereinbarung brav an derselben Stelle[1].

4.6.4 Array an Zeiger – schon bei der Initialisierung

Man kann übrigens die Startadresse des Arrays schon bei der Initialisierung an den Zeiger übergeben (`src/feld_an_pointer0.c`), damit spart man sich eine Zeile:

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char string[80]="Dies ist eine Textzeile.";
5      char *p = string;
6      printf("%s\n", p);
7      return 0;
8  }
```

4.6.5 Anonymes Array an Zeiger (Zuweisung)

Das obige Programm wurde optimiert (`src/feld_an_pointer2.c`).

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char *p;
5      p = "Hallo";
6      printf("Inhalt von p: %p\n", p);
7      printf("Worauf p zeigt: %c\n", *p);
8      printf("Startadresse von \"Hallo\": %p\n", "Hallo");
9
10     char c="Hallo"[2];
11     printf("Zeichen Nr. 2 von Hallo: %c\n", c);
12     return 0;
13 }
```

[1] Die einzige Ausnahme sind Speicherbereiche, die mit `malloc` dynamisch alloziert wurden; sie können mit `realloc` verschoben werden; dazu später mehr

Was ist nun passiert? Offenbar enthält auch hier `p` die Startadresse eines Strings.

Wo aber steht der String? In `p` ist kein Platz für so viele Zeichen. Außerdem soll in `p` ja die Adresse stehen, nicht die Zeichen selbst.

Die Lösung lautet: Der String ist ein *anonymes Array*. Anonym heißt: Es hat keinen eigenen Namen. Dieses anonyme Array ist außerdem konstant, es liegt in einem Speicherbereich für Konstanten. Bei Mikrocontroller-Systemen kann das zum Beispiel das Flash-EPROM sein. Bei PCs kann das ein schreibgeschützter Bereich sein.

Ein solches anonymes Array wurde bereits ganz zu Anfang des C-Kurses im Funktionsaufruf `printf("hello, world!\n");` benutzt.

Sobald ich `p` auf etwas Anderes zeigen lasse, also eine andere Adresse in `p` lade, ist der Zugriff auf das anonyme Array nicht mehr möglich (es sei denn, ich habe mir vorher den alten Wert von `p` in einer anderen Variablen gemerkt).

4.6.6 Anonymes Array an Zeiger (Initialisierung)

Wenn man einen Zeiger mit einem Array initialisiert, kann man dazu auch ein anonymes Array verwenden (`src/feld_an_pointer3.c`):

```
1 #include <stdio.h>
2 int main(void)
3 {
4     char *p="Dies ist eine Textzeile.";
5     printf("%s\n", p);
6     return 0;
7 }
```

4.6.7 Anonymes Array in anderer Schreibweise

Wenn man ein anonymes Array nicht mit Anführungszeichen, sondern mit geschweiften Klammern schreibt, muss man dem Compiler mit Hilfe einer expliziten Typangabe (per Typumwandlungs-Operator) mitteilen, welchen Datentyp das anonyme Array haben soll:

```
1     char *p;
2     p = {'A', 'B', 'C', '\0'};            /* falsch   */
3     p = (char []){'A', 'B', 'C', '\0'}; /* richtig */
```

Damit kann man anonyme Arrays von beliebigen Elementtypen erzeugen (`src/feld_an_pointer22.c`):

```
1 #include <stdio.h>
2 int main(void)
3 {
4     double *p;
5     p = (double []){3.0,4.0,5.0};
6     printf("%f,%f,%f\n", p[0],p[1],p[2]);
7     return 0;
8 }
```

4.6.8 Unterschiede

Die beiden folgenden Variablen sehen fast gleich aus:

```
1 #include <stdio.h>
2 int main(void)
3 {
```

```
4     char string[] = "Guten Morgen1";
5     char *p       = "Guten Morgen2";
6     printf("%s,%s\n", p, string);
7     return 0;
8  }
```

In Zeile 4 wird ein Array angelegt; es wird so initialisiert, dass es anschließend den Inhalt `"Guten Morgen1"` hat.

In Zeile 5 wird ein Zeiger angelegt; er wird so initialisiert, dass er anschließend auf ein konstantes anonymes Array zeigt, welches den Inhalt `"Guten Morgen2"` zeigt.

Zeile 6 suggeriert, dass beide – `p` und `string` – gleichartig sind. Das stimmt aber nicht:

a) Mit dem Namen `string` greift man auf die *konstante Adresse* eines *variablen Arrays* zu. `string[4]` kann man ändern, die Startadresse von `string` dagegen nicht.

b) Mit dem Namen `p` greift man auf die *variable Adresse* zu, die momentan zu einem *konstanten (anonymen) Array* gehört. `p` kann man ändern, `p[4]` nicht.

4.6.9 Wozu ist das gut?

Häufig muss man in Programmen Elemente sortieren, z.B. Zahlen. Zur Sortierung braucht man eine kluge Anleitung (einen so genannten Algorithmus) und viel Arbeit. Die Arbeit besteht vor allem aus dem Vertauschen zweier Elemente. Dieses Vertauschen kommt sehr oft vor:

```
1     int c;
2     c = a;
3     a = b;
4     b = c;
```

Wenn man jetzt nicht Zahlen sortiert, sondern Strings, z.B. Kundennamen, dann braucht man für das Vertauschen zweier Elemente, also zweier Strings, viel Zeit: Man muss jedes Zeichen des einen Strings mit dem entsprechenden Zeichen des anderen Strings vertauschen (`src/vertauschen_arrays.c`):

```
1  #include <stdio.h>
2  int main(void)
3  {
4     int lauf;
5     char a[2000000]="Dies ist die erste Textzeile.";
6     char b[2000000]="Dies ist die zweite Textzeile.";
7     char c[2000000];
8     printf("Inhalt von a: %s\n", a);
9     printf("Inhalt von b: %s\n", b);
10    for(lauf=0; lauf<2000000; ++lauf)
11    {
12       c[lauf]=a[lauf];
13       a[lauf]=b[lauf];
14       b[lauf]=c[lauf];
15    }
16    printf("Inhalt von a: %s\n", a);
17    printf("Inhalt von b: %s\n", b);
18    return 0;
19 }
```

Und diese Prozedur muss nun bei jeder Vertauschung einmal durchlaufen werden.

Mit Zeigern kann man sich das Leben einfacher machen. Zu jedem String legt man einen Zeiger an. Und anstelle der String-Inhalte vertauscht man einfach die Zeiger-Inhalte (`src/vertauschen_pointer.c`):

```c
#include <stdio.h>
int main(void)
{
    char a[2000000]="Dies ist die erste Textzeile.";
    char b[2000000]="Dies ist die zweite Textzeile.";
    char *p=a;
    char *q=b;
    char *r;
    printf("Inhalt von p: %s\n", p);
    printf("Inhalt von q: %s\n", q);
    r=p;
    p=q;
    q=r;
    printf("Inhalt von p: %s\n", p);
    printf("Inhalt von q: %s\n", q);
    return 0;
}
```

Jetzt müssen pro Vertauschung nur noch 4 oder 8 Bytes statt 2000000 Bytes vertauscht werden.