4.7 Datenstrukturen/Rechnen mit Zeigern

4.7.1 Unglaubliche Konstruktionen

Im folgenden Beispielprogramm wird die Länge eines Strings ermittelt (`src/strlen.c`):

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char s[81]="";
5      char *p;
6      printf("Bitte einen kurzen Text eingeben: ");
7      scanf("%80[^\n]", s);
8      p=s;
9      while(*p!='\0')
10     {
11         ++p;
12     }
13     printf("Laenge des Texts: %u Zeichen\n", p-s);
14     return 0;
15 }
```

Das Programm wirkt sehr merkwürdig. Offenbar wird auf die Array-Elemente zugegriffen. Aber das passiert ganz ohne eckige Klammern und auch nicht über s, sondern über p (Zeile 9). Mittendrin wird die Zeiger-Variable um eins hochgezählt (Zeile 11). Und zum Schluss wird auch noch die Differenz aus einem Zeiger und einem Array gebildet (Zeile 13). In vielen anderen Programmiersprachen wäre das nicht erlaubt. In C dagegen geht es. Was aber bedeuten diese Ausdrücke in C?

4.7.2 Bekanntes: Arrayname als Startadresse

In Zeile 8 findet man eine bekannte C-Tatsache: Wenn auf der rechten Seite einer Zuweisung (oder eine Initialisierung) wirkt der Arrayname allein (also ohne eckige Index-Klammern) steht, wird er interpretiert als die Startadresse des Arrays. Daher ist die Zuweisung `p=s` erlaubt: Links steht eine Zeiger-Variable, rechts eine Zeiger-Konstante.

4.7.3 Zugriff auf das Start-Element per Zeiger?

Wie kann es eigentlich sein, dass der Arrayname `s` verschiedene Bedeutungen haben kann, je nachdem, ob er mit oder ohne eckige Index-Klammern verwendet wird? Mal soll er ein Array sein, mal ein Zeiger?

Wenn `s` ein konstanter Zeiger auf das erste Element sein soll (s = &s[0]), dann müsste `*s` der Inhalt des ersten Elements sein (*s = s[0]). Dann gibt das folgende Programmstück den Buchstaben `'H'` aus (`src/zugriffstartelement.c`):

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char s[]="Hallo";
5      putchar(*s);
6      putchar('\n');
7      return 0;
8  }
```

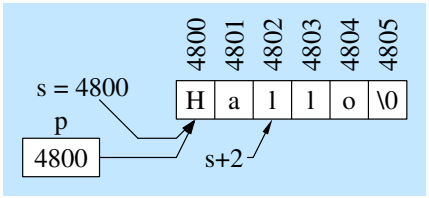

Abbildung 1: Zugriff auf Array-Elemente

Terminal

```
schueler@debian964:~$ gcc zugriffstartelement.c
schueler@debian964:~$ ./a.out
H
```

`s` kann hier wirklich wie ein ganz normaler Zeiger benutzt werden, ganz genau wie `p`.

4.7.4 Zugriff auf weitere Elemente per Zeiger?

Wenn man mit `*s` an das erste Element herankommt, wie kommt man dann an die anderen Elemente? Und zwar ohne Index-Klammern, nur mit dem Inhalts-Operator? Dazu verrät das Standardwerk von Kernighan und Ritchie ein Geheimnis des C-Compilers: Jeder Zugriff auf ein Array-Element mit eckigen Index-Klammern wird vom Compiler sofort zu einem Zugriff mit Inhalts-Operator umgewandelt:

```
s[0] ----> *s
s[1] ----> *(s+1)
s[2] ----> *(s+2)
...
s[i] ----> *(s+i)
```

Wenn also `s` an Adresse 4800 liegt, wird es so:

```
s[0] ----> *(4800)
s[1] ----> *(4801)
s[2] ----> *(4802)
...
s[i] ----> *(4800+i)
```

Man könnte sagen, in C gibt es in Wirklichkeit gar keine Index-Klammern, sondern nur Zeiger, das Rechnen mit Zeigern und (drittens) Inhalts-Operatoren (Abbildung 1).

Ein Programm muss zeigen, ob das alles stimmt (`src/zugriffweitere.c`):

```
1  #include <stdio.h>
2  int main(void)
3  {
4      char s[]="Hallo";
5      char *p;
6      putchar(  *(s+2)  );
7      putchar(  s[2]    );
8      p=s;
9      putchar(  *(p+2)  );
10     putchar(  p[2]    );
11     putchar(  '\n'    );
12     return 0;
13 }
```

Terminal
```
schueler@debian964:~$ gcc zugriffweitere.c
schueler@debian964:~$ ./a.out
llll
```

4.7.5 Rechnen mit Zeigern

Man kann also in C mit Zeigern rechnen wie mit Zahlen. Diese Tätigkeit nennt man *Zeiger-Arithmetik*[1]. Dabei sind ganz bestimmte Rechenoperationen erlaubt:

- `s+100` – man darf einen Zeiger mit einer Zahl addieren. Ergebnis ist hier die Adresse von Array-Element 100.
- `s-1` – man darf auch eine Zahl von einem Zeiger subtrahieren. Ergebnis ist hier die Adresse des ersten Elements vor dem ersten gültigen Element im Array
- `p-s` – man darf auch zwei Zeiger voneinander subtrahieren, vorausgesetzt, sie zeigen auf denselben Elementtyp. Wenn sie auf dasselbe Array zeigen (z.B. `s` auf Element Nr. 0 und `p` auf Element Nr. 7), erhält man als Ergebnis die Differenz der Indizes (hier wäre das 7-0=7), und das ist eine einfache Zahl.

Mit variablen Zeigern sind naturgemäß weitere arithmetische Operationen möglich, die den Inhalt des Zeiger verändern:

- `p+=2` – erhöht `p`, so dass es auf das übernächste Element zeigt.
- `p-=2` – verringert `p`, so dass es auf das vorvorige Element zeigt.
- `++p` – erhöht `p`, so dass es auf das nächste Element zeigt.
- `--p` – verringert `p`, so dass es auf das vorige Element zeigt.

Manche Operationen sind für Zeiger nicht zugelassen, weil den Autoren der Programmiersprache dazu nichts Sinnvolles eingefallen ist:

- `s*2` – gibt keinen Sinn
- `s/3` – gibt keinen Sinn
- `p+s` – gibt auf den ersten Blick auch keinen Sinn

Das Addieren zweier Zeiger scheint zunächst so sinnvoll, als wenn man zwei Uhrzeiten oder zwei Datumswerte addiert. Es gibt aber einen Zusammenhang, indem man es gebrauchen könnte, nämlich die Mittelwertbildung („gib mir das Element in der Mitte eines Strings“). Egal, ob man `(p+s)/2` oder `p/2+s/2` schreibt, es muss immer addiert werden. Man kann aber `(p+s)/2` ersetzen durch `s+(p-s)/2`, und dann braucht man nur einen Zeiger (nämlich `x`) und eine Zahl (nämlich `(p-s)/2`) zu addieren.

```c
#include <stdio.h>
int main(void)
{
    char s[]="WESER";
    char *p;
    p=s;
    p=p+4; /* erhoehe p um 4, so dass p auf Element Nr. 4 zeigt */
    putchar(*p); putchar('\n');
```

[1] *Arithmetik* ist das Rechnen mit Zahlen, wie man es in der Grundschule lernt. *Algebra* dagegen ist das Rechnen mit Buchstaben, wie man es in der Sekundarstufe lernt.

```
9     printf("Abstand zwischen %c und %c: %d Elemente\n", *p, *s, p-s);
10    printf("Zeichen in der Mitte: %c\n", *(s+(p-s)/2));
11    return 0;
12 }
```

4.7.6 Klappt das mit allen Arrays?

Bisher haben wir das alles nur mit Arrays probiert, deren Elemente genau ein Byte groß waren, nämlich mit Strings. Das ist auch der häufigste Anwendungsfall. Die Frage ist nur: Wie klappt das Rechnen mit Zeigern, soll heißen mit Adressen, wenn die Elemente so groß sind, das dass sie nicht mehr an aufeinanderfolgenden Adressen liegen?

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5     char c[8]="BMW 123";
6     int i[3]={47, 3900, 220};
7
8     char *pc =c;
9     int *pi =i;
10
11    printf("Adresse in pc=%u\n", (void *)pc);
12    printf("*pc=%c\n", *pc);
13    printf("Erhoehe pc um 1.\n");
14    ++pc;
15    printf("Adresse in pc=%u\n", (void *)pc);
16    printf("*pc=%c\n", *pc);
17
18    printf("Adresse in pi=%u\n", (void *)pi);
19    printf("*pi=%d\n", *pi);
20    printf("Erhoehe pi um 1.\n");
21    ++pi;
22    printf("Adresse in pi=%u\n", (void *)pi);
23    printf("*pi=%d\n", *pi);
24    return 0;
25 }
```

Terminal

```
schueler@debian964:~$ gcc zeigerarith_demo.c
schueler@debian964:~$ ./a.out
Adresse in pc=0xbf932130
*pc=B
Erhoehe pc um 1.
Adresse in pc=0xbf932131
*pc=M
Adresse in pi=0xbf932124
*pi=47
Erhoehe pi um 1.
Adresse in pi=0xbf932128
*pi=3900
```

Man sieht: Beim Rechnen mit `char`-Zeigern gilt $130 + 1 = 131$. Beim Rechnen mit `int`-Zeigern dagegen gilt $124 + 1 = 128$ (siehe Abbildung 2).

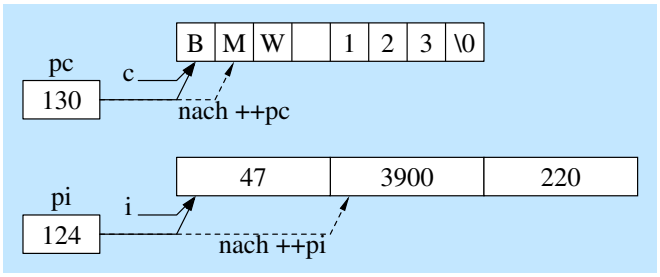

Abbildung 2: Rechnen mit Zeigern auf verschieden große Elemente

Das heißt, wenn man einen Zeiger auf ein x Byte großes Element um eins erhöht, erhöht man die im Zeiger gespeicherte Adresse um den Wert x.

Zeigerarithmetik ist also immer auf ein fiktives Array bezogen, auf dessen Elemente man zugreifen möchte, selbst dann, wenn da gar kein Array ist, weil man zum Beispiel vergessen hat, den Zeiger zu initialisieren.

Nun sieht man auch einen weiteren Grund dafür, dass es in C verschiedene Arten von Zeigern gibt: Gäbe es nur einen einzigen Datentyp `address` und nicht die Vielfalt von `char*`, `int*`, `double*`, dann wüsste der Compiler bei der Zuweisung `++p` gar nicht, um wieviel er die im Zeiger gespeicherte Adresse hochzählen müsste.

4.7.7 Arrays sind auch nur Zeiger?

Nun ist der Eindruck entstanden, Arrays und Zeiger seien gleich. Daher sollen hier ein paar Unterschiede aufgeführt werden:

- Beim Array wird Speicherplatz für die Array-Elemente belegt, beim Zeiger nicht. Der Zeiger muss auf fremden Speicherplatz zeigen.
- Wenn man in einer Funktion das Array `s` und den Zeiger `p` anlegt und in dieser Funktion mit dem sizeof-Operator ihre Größe (=den belegten Speicherplatz) überprüft, erhält man deshalb unterschiedliche Ergebnisse (`src/speicherarrayzeiger.c`):

```c
#include <stdio.h>
int main(void)
{
    char s[80];
    char *p;
    printf("s belegt %lu Bytes.\n", sizeof(s));
    printf("p belegt %lu Bytes.\n", sizeof(p));
    return 0;
}
```

Terminal

```
schueler@debian964:~$ gcc speicherarrayzeiger.c
schueler@debian964:~$ ./a.out
s belegt 80 Bytes
p belegt 4 Bytes
```

`s` belegt Platz für 80 Zeichen, `p` belegt Platz für eine Adresse (hier 4 Byte, kann aber je nach System auch einen anderen Wert haben).

- Ein Zeiger ist eine Variable (es sei denn, man legt es anders fest, dazu benötigte man dann das Schlüsselwort `const`). Das Array dagegen liegt fest an einer konstanten Adresse.

4.7.8 Gebrauch der Zeigerarithmetik für eigene Programme

Wenn man bei Operationen an Arrays Zeigerarithmetik benutzt, kann man oft auf sehr einfache, kurze und schnelle Lösungen kommen, wie es das folgende Beispiel zeigt, in der ein String umgekehrt in einen anderen kopiert wird (`src/umkehr.c`).

```
1  #include <stdio.h>
2  #define LEN 70
3
4
5  int main(void)
6  {
7      char s[LEN]="Das ist das Haus vom Nikolaus.";
8      char t[LEN];
9      char *p =s;
10     char *q =t;
11     while(*p) ++p;
12     while(p>s)
13     {
14         --p;
15         *q=*p;
16         ++q;
17     }
18     *q='\0';
19     printf("%s\n", s);
20     printf("%s\n", t);
21     return 0;
22 }
```

Erfahrungsgemäß fühlen sich viele Anfänger zunächst wohler damit, Arrays mit Zählschleife und Index zu bearbeiten und schwenken erst mit zunehmender Erfahrung auf die Arbeit mit Zeigern um.