4.8 Datenstrukturen/Übergabe eines Arraynamens an eine Funktion

4.8.1 Übergabe eines Arrays?

Es gibt viele Funktionen, die mit Zeichenketten und anderen Arrays arbeiten. Da stellt sich die Frage: Wie kann man eigentlich eine Zeichenkette (oder ein anderes Array) als Parameter an eine Funktion übergeben?

4.8.2 Übergabe eines Zeigers

Das folgende erste Beispiel zeigt, wie es geht:

```
zaehle_bs1.c
```

```

1 #include <stdio.h>
2 int zaehle_bs(char *z)
3 {
4     int i, erg=0;
5     for(i=0; z[i]!='\0'; ++i)
6     {
7         if(z[i]>='A' && z[i]<='Z') ++erg;
8         if(z[i]>='a' && z[i]<='z') ++erg;
9     }
10    return erg;
11 }
12 int main(void)
13 {
14     int anzahl;
15     char s[801]="";
16     printf("Bitte_eine_Zeile_(max._800_Zeichen)_eingeben:_");
17     scanf("%800[^\n]", s);
18     anzahl = zaehle_bs(s);
19     printf("Die_Zeile_enthielt_%i_Buchstaben.\n", anzahl);
20     return 0;
21 }
```

Zur Wiederholung noch einmal die bisher bekannten Tatsachen:

- a) Mit dem Namen `s` wird in C (und nur dort, es ist eine Besonderheit von C) die Adresse des ersten Elements von `s` angesprochen: *s ist identisch mit &s[0]*.
- b) Damit bezeichnet `*s` den Inhalt des ersten Elements von `s`; **s ist identisch mit s[0]*.
- c) Mit dem Operator `s[i]` wird der Inhalt des `i`-ten Elements des Arrays `s` angesprochen.

Wenn die Aussage a) stimmt, dann wird in Zeile 18 als Parameter ein Zeiger an die Funktion übergeben, nämlich die Startadresse von `s`.

Beim Aufruf der Funktion wird dieser Zeiger in den formalen Parameter `z` kopiert, der ebenfalls einen Zeiger-Datentyp hat. In den Zeilen 5, 7 und 8 wird mit diesem Zeiger dann so gearbeitet, als wenn es sich um ein Array handelt – im Sinne der Zeigerarithmetik ist das in C ja erlaubt.

Das Ergebnis lautet also:

- a) Wenn man in C einen Array-Namen an eine Funktion übergibt, dann übergibt man in Wirklichkeit nur die Startadresse.
- b) Innerhalb der aufgerufenen Funktion kann man mit dieser Startadresse genauso arbeiten wie sonst mit dem Arraynamen.

4.8.3 Eine besondere Schreibweise

Im folgenden Beispiel ist eine Neuigkeit versteckt:

zaehle_bs2.c

```

1 #include <stdio.h>
2 int zaehle_bs(char z[])
3 {
4     int i, erg=0;
5     for(i=0; z[i]!='\0'; ++i)
6     {
7         if(z[i]>='A' && z[i]<='Z') ++erg;
8         if(z[i]>='a' && z[i]<='z') ++erg;
9     }
10    return erg;
11 }
12 int main(void)
13 {
14     int anzahl;
15     char s[801]="";
16     printf("Bitte_eine_Zeile_(max._800_Zeichen)_eingeben:_");
17     scanf("%800[^\n]", s);
18     anzahl = zaehle_bs(s);
19     printf("Die_Zeile_enthielt_%i_Buchstaben.\n", anzahl);
20     return 0;
21 }
```

In diesem Beispiel gibt es nur eine einzige neue Stelle, und zwar in der Parameterliste, also in Zeile 2¹.

Soeben war noch klar, es kann immer nur ein Zeiger übergeben werden, niemals ein ganzes Array. *Hier* aber steht als Parameter `char z[]`! Wie kann das sein?

Kurz gesagt: Das ist nur eine neue Schreibweise für einen Zeiger. Es macht hier also nichts aus, ob der Parameter als Array oder als Zeiger beschrieben wird: **In der Parameterliste wird die Vereinbarung eines Arrays umgeschrieben in die Vereinbarung eines Zeigers.** Das bedeutet:

`void zaehle_bs(char *z)` ist identisch mit

`void zaehle_bs(char z[])` und sogar mit

`void zaehle_bs(char z[2015])`. Egal, welche Variante man wählt – der Compiler macht eine Zeiger-Vereinbarung daraus.

In Lehrbüchern heißt es dazu oft sinngemäß: Ein Array „zerfällt“ in der Parameterliste einer Funktion (und nur dort!) zu einem Zeiger (*decay*). Das ist etwas irreführend – ein Array kann nicht von selbst zu irgendetwas zerfallen. Sondern der Compiler verändert die **Vereinbarung** eines Arrays zu der eines Zeigers.

Und hier ein „Beweis“: Die beiden Programme (Beispiel 1 und Beispiel 2) sind ja im Quelltext fast gleich; die Funktion `zaehle_bs()` hat als Parameter in der einen Version `char z[]` und in der anderen Version `char *z`:

```

----- Terminal -----
schueler@debian964:~$ diff zaehle_bs1.c zaehle_bs2.c
3c3
< int zaehle_bs(char z[])
---
> int zaehle_bs(char *z)
```

¹Zur Erinnerung: Parameter sind die Variablen innerhalb der Funktionsklammer, also die Daten, die an eine Funktion übergeben werden.

Das Compilieren bringt folgendes Ergebnis:

```

Terminal
schueler@debian964:~$ gcc -o zaehle_bs1 zaehle_bs1.c
schueler@debian964:~$ gcc -o zaehle_bs2 zaehle_bs2.c
schueler@debian964:~$ ls -l zaehle_bs1 zaehle_bs2
-rwxr-xr-x 1 user group 5456 Okt 28 16:39 zaehle_bs1
-rwxr-xr-x 1 user group 5456 Okt 28 16:39 zaehle_bs2

```

Beide Objektdateien (ausführbare Programme) sind schon einmal gleich lang. Aber sind sie auch gleich? Dazu muss man zuerst alle Reste an Quelltext aus den ausführbaren Dateien entfernen:

```

Terminal
schueler@debian964:~$ strip zaehle_bs1
schueler@debian964:~$ strip zaehle_bs2

```

Dann kann man überprüfen, ob beide Programme auch die gleichen CPU-Befehle haben:

```

Terminal
schueler@debian964:~$ diff -s zaehle_bs1 zaehle_bs2
Binärdateien zaehle_bs1 und zaehle_bs2 sind identisch.
schueler@debian964:~$

```

Das Programm diff zeigt es: Beide Programme sind bis auf das letzte Bit identisch.

4.8.4 Wird wirklich *nur* ein Zeiger übergeben?

Offenbar ist es egal, ob in der Parameterliste `char s*`, `char s[]` oder gar `char s[80]` steht. Aber wie kann man erkennen, dass tatsächlich ein *Zeiger* übergeben wird?

Hier hilft der `sizeof()`-Operator weiter. Zur Compilierzeit setzt er bereits die Größe des Objekts ein. Wir definieren einfach ein großes Array mit 2000 Bytes und übergeben es an eine Funktion. Dort lassen wir uns durch `sizeof()` die Größe ermitteln. Durch `printf()` soll sie dann ausgegeben werden:

```

sizeof.c
1 #include <string.h>
2 #include <stdio.h>
3 void demo(char x []);
4
5 /******
6 int main(void)
7 {
8     char x[2000]="BLABLABLABLABLE";
9
10    printf("main():_sizeof(x)=%lu\n", sizeof x );
11    printf("main():_strlen(x)=%lu\n", strlen(x));
12    demo(x);
13    return 0;
14 }
15 /******
16 void demo(char x [])
17 {
18    printf("demo():_sizeof(x)=%lu\n", sizeof x );
19    printf("demo():_strlen(x)=%lu\n", strlen(x));
20 }

```

Das Programm ergibt:

```

Terminal
schueler@debian964:~$ gcc sizeof.c
schueler@debian964:~$ ./a.out
main(): sizeof(x)=2000
main(): strlen(x)=15
demo(): sizeof(x)=4
demo(): strlen(x)=15

```

Man sieht, dass `x` in der Funktion `demo()` nur 4 Bytes groß ist – das ist auf diesem System genau die Größe einer Adresse (Adressbusbreite ist 32 Bit). Der Wert 2000 ist in der Funktion `demo()` dagegen unbekannt. **Es wird wirklich nur ein Zeiger übergeben.**

4.8.5 Und wo bleibt die Array-Größe?

Die Größeninformation des Arrays ist beim obigen Beispiel verlorengegangen - nur die Startadresse und der Elementtyp sind bekannt. Wie kann man nun die Größe an das Array übergeben? Dazu sind mehrere Möglichkeiten denkbar:

- a) Man nimmt einen Extra-Parameter, der die Größe des Arrays angibt:

```

printarray.c
1 #include <stdio.h>
2 void printarray(int a[], unsigned int anzahl)
3 {
4     unsigned int lauf;
5     for(lauf=0; lauf<anzahl; ++lauf)
6         printf("%d_", a[lauf]);
7 }
8 int main(void)
9 {
10     int arr[3]={19,22,25};
11     printarray(arr,3);
12 }

```

- b) Man legt einen besonderen Element-Wert als Terminator fest; bei Strings ist dies meistens `'\0'`:

```

printstr.c
1 #include <stdio.h>
2 void printstr(char s[])
3 {
4     unsigned int lauf;
5     for(lauf=0; s[lauf]!='\0'; ++lauf)
6         printf("%c", s[lauf]);
7 }
8 int main(void)
9 {
10     char s[]={ 'H', 'a', 'l', 'l', 'o', '\0' };
11     printstr(s);
12 }

```

Damit spart man in diesem Fall den zweiten Parameter ein.