4.12 Datenstrukturen/String-Bibliothek

4.12.1 Vergleich von Strings: `strcmp()`

Ein Programm soll zu einem eingegebenen Bauelement die Betriebsmittelkennzeichnung ausgeben. Ein erster Entwurf sieht so aus (`betriebsmittel0.c`):

```c
#include <stdio.h>
#define LEN 80
int main(void)
{
    char wort[LEN+1]="";
    printf("Bauelement eingeben:");
    scanf("%80s", wort);
    while(getchar()!='\n'){}
    printf("Betriebsmittelkennzeichnung: ");
    if      (wort=="Transistor")  printf("V\n");
    else if(wort=="Widerstand")   printf("R\n");
    else if(wort=="Kondensator")  printf("C\n");
    else if(wort=="Spule")        printf("L\n");
    else                          printf("unbekannt\n");
    return 0;
}
```

Nun wird getestet:

Terminal

```
schueler@debian964:~$ gcc -o betriebsmittel0 betriebsmittel0.c
schueler@debian964:~$ betriebsmittel0
Bauelement eingeben:Transistor
Betriebsmittelkennzeichnung: unbekannt
```

Warum wird keine Eingabe richtig erkannt? Warum werden gleiche Strings nicht als gleich angesehen? Die Lösung liegt darin, dass mit dem Zugriff auf den Namen der Arrays nur die Adressen verglichen werden und nicht die Inhalte. Hier noch ein Beispiel (`vergleichsoperator_demo.c`):

```c
#include <stdio.h>
int main(void)
{
    char frucht1[]="Kirsche";
    char frucht2[]="Kirsche";

    if(frucht1==frucht2)
        printf("gleich\n");
    else
        printf("ungleich\n");

    if("Zitrone"=="Zitrone")
        printf("gleich\n");
    else
        printf("ungleich\n");
    return 0;
}
```

Terminal

```
schueler@debian964:~$ gcc -o vergleichsop_demo vergleichsop_demo.c
schueler@debian964:~$ ./vergleichsop_demo
```

```
ungleich
gleich
```

Die beiden Variablen, die zuerst verglichen werden, liegen an unterschiedliche Adressen, also sind sie ungleich. Die beiden literalen (wörtlichen) Stringkonstanten, die danach verglichen werden, liegen interessanterweise *bei diesem Compiler* (GNU-C) an derselben Adresse. Dadurch werden sie als gleich angesehen.[1]

Wenn man die Inhalte zweier Strings vergleichen will, braucht man also eine andere Lösung. Sie besteht aus der Funktion `strcmp()` und einigen verwandten Funktionen. `strcmp()` wird in der C-Standard-Bibliothek mitgeliefert. Über `<string.h>` wird sie eingebunden. `strcmp()` erhält zwei Strings (dürfen auch konstant sein) als Parameter und gibt bei Gleichheit eine Null aus (Beispiel: `strcmp_demo.c`):

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char string1[21]="";
    char string2[21]="";
    printf("Zeichenkette 1 eingeben:");
    scanf("%20[^\n]", string1);
    while(getchar()!='\n');

    printf("Zeichenkette 2 eingeben:");
    scanf("%20[^\n]", string2);
    while(getchar()!='\n');

    printf("strcmp(\"%s\",\"%s\") ergibt %i\n",
           string1, string2,strcmp(string1,string2));
    return 0;
}
```

Bei Ungleichheit wird eine Zahl größer oder kleiner als null ausgegeben[2].

Das Programm vom Beginn des Abschnitts kann damit so geschrieben werden (`betriebsmittel1.c`):

```c
#include <stdio.h>
#include <string.h>
#define LEN 80
int main(void)
{
    char wort[LEN+1]="";
    printf("Bauelement eingeben:");
    scanf("%80s", wort);
    while(getchar()!='\n'){}
    printf("Betriebsmittelkennzeichnung:");
    if      (strcmp(wort,"Transistor")==0)  printf("V\n");
    else if(strcmp(wort,"Widerstand")==0)   printf("R\n");
    else if(strcmp(wort,"Kondensator")==0) printf("C\n");
    else if(strcmp(wort,"Spule")==0)        printf("L\n");
    else                                    printf("unbekannt\n");
    return 0;
```

[1] Ein anderer Compiler, der vielleicht nicht merkt, dass die beiden Konstanten gleich sind, legt die beiden Strings einzeln an zwei verschiedene Adressen. Bei ihm ergibt der Vergleich dann eine Ungleichheit.

[2] Je nachdem, ob der erste String in alphabetischer Sortierung hinter oder vor dem zweiten String kommt

```
17 }
```

4.12.2 Kopieren von Strings: `strcpy()`

Das Programm von oben soll eleganter gestaltet werden. Die Antwort soll in einen dritten String gegeben werden, der erst zum Schluss ausgegeben werden soll (`betriebsmittel2.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  #define LEN 80
4  int main(void)
5  {
6      char wort[LEN+1]="";
7      char antwort[LEN+1]="unbekannt";
8      printf("Bauelement␣eingeben:");
9      scanf("%80s", wort);
10     while(getchar()!='\n'){}
11     printf("Betriebsmittelkennzeichnung:␣");
12     if      (strcmp(wort,"Transistor")==0)  antwort="V";
13     else if(strcmp(wort,"Widerstand")==0)  antwort="R";
14     else if(strcmp(wort,"Kondensator")==0) antwort="C";
15     else if(strcmp(wort,"Spule")==0)       antwort="L";
16     printf("%s\n", antwort);
17     return 0;
18 }
```

Der Test ergibt:

Terminal

```
schueler@debian964:~$ gcc -o betriebsmittel2 betriebsmittel2.c
betriebsmittel2.c: In function 'main':
betriebsmittel2.c:11: error: incompatible types when assigning to type
'char[81]' from type 'char *'
```

Stimmt, man darf an ein Array (also auch an einen String) nichts zuweisen. Aber auch für dieses Problem gibt es eine Lösung aus `<string.h>`, nämlich die Funktion `strcpy()`. Auch `strcpy()` bekommt zwei Strings als Parameter, der erst davon muss eine Variable sein. Dann wird der zweite String in den ersten kopiert (Beispiel: `strcpy_demo.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(void)
4  {
5      int c;
6      char s2[4001]="";
7      char s1[5]="";
8
9      printf("Zeichenkette␣nach␣s2␣eingeben␣(max.␣4␣Zeichen):");
10     scanf("%4000[^\n]", s2);
11     while(getchar()!='\n'){}
12
13     printf("kopiere␣s2␣nach␣s1␣mit␣Aufruf␣strcpy(s1,s2);\n");
14     strcpy(s1,s2);
15     printf("Inhalt␣von␣s1:␣>>%s<<\n", s1);
16     printf("Inhalt␣von␣s2:␣>>%s<<\n", s2);
```

```
17     return 0;
18 }
```

Terminal

```
schueler@debian964:~$ gcc -o strcpy_demo strcpy_demo.c
schueler@debian964:~$ ./strcpy_demo
Zeichenkette nach s2 eingeben (max. 4 Zeichen):Lala
kopiere s2 nach s1 mit Aufruf strcpy(s1,s2);
Inhalt von s1: >>Lala<<
Inhalt von s2: >>Lala<<
```

Nach Aufruf von `strcpy()` haben also beide Strings den gleichen Inhalt.

Die neue Version des Betriebsmittel-Programms sieht nun so aus (`betriebsmittel3.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  #define LEN 80
4  int main(void)
5  {
6      char wort[LEN+1]="";
7      char antwort[LEN+1]="unbekannt";
8      printf("Bauelement␣eingeben:");
9      scanf("%80s", wort);
10     while(getchar()!='\n'){}
11     printf("Betriebsmittelkennzeichnung:␣");
12     if      (strcmp(wort,"Transistor")==0)  strcpy(antwort,"V");
13     else if(strcmp(wort,"Widerstand")==0)  strcpy(antwort,"R");
14     else if(strcmp(wort,"Kondensator")==0) strcpy(antwort,"C");
15     else if(strcmp(wort,"Spule")==0)       strcpy(antwort,"L");
16     printf("%s\n", antwort);
17     return 0;
18 }
```

4.12.3 Verketten von Strings: `strcat()`

Nun soll das Programm – ganz nach dem Anspruch des E-V-A-Prinzips – alle Ausgaben im Antwortstring sammeln und erst zum Schluss mit einem `printf()`-Befehl ausgeben. Hier ist die erste Version (`betriebsmittel4.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  #define LEN 80
4  int main(void)
5  {
6      char wort[LEN+1]="";
7      char antwort[LEN+1]="unbekannt";
8      printf("Bauelement␣eingeben:");
9      scanf("%80s", wort);
10     while(getchar()!='\n'){}
11     strcpy(antwort,"Betriebsmittelkennzeichnung:␣");
12     antwort=antwort+wort;
13     if      (strcmp(wort,"Transistor")==0)  antwort=antwort+"V";
14     else if(strcmp(wort,"Widerstand")==0)  antwort=antwort+"R";
15     else if(strcmp(wort,"Kondensator")==0) antwort=antwort+"C";
```

```
16      else if(strcmp(wort,"Spule")==0)        antwort=antwort+"L";
17      else                                    antwort=antwort+"unbekannt";
18      printf("%s\n", antwort);
19      return 0;
20  }
```

Dieses Programm enthält offensichtlich an mehreren Stellen denselben Fehler: Man kann Strings in C nicht einfach addieren. Gesucht wird eine Funktion, die einen String an einen anderen anhängt. Wieder gibt es eine Funktion aus `<string.h>`, die diese Aufgabe löst: `strcat()` heißt sie. Sie erhält (wie `strcpy()`) zwei Strings als Parameter, von denen der erste variabel sein muss. Dann wird der zweite String an das Ende des ersten Strings kopiert (Beispiel: `strcat_demo.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void)
5  {
6      int c;
7      char s2[11]="";
8      char s1[11]="";
9
10     printf("Zeichenkette␣nach␣s1␣eingeben␣(max.␣10␣Zeichen):");
11     scanf("%10[^\n]", s1);
12     while(getchar()!='\n'){}
13
14     printf("Zeichenkette␣nach␣s2␣eingeben␣(max.␣10␣Zeichen):");
15     scanf("%10[^\n]", s2);
16     while(getchar()!='\n'){}
17
18     printf("haenge␣s2␣an␣s1␣an␣mit␣Aufruf␣strcat(s1,s2);\n");
19     strcat(s1,s2);
20     printf("Inhalt␣von␣s1:␣>>%s<<\n", s1);
21     printf("Inhalt␣von␣s2:␣>>%s<<\n", s2);
22     return 0;
23 }
```

Terminal

```
schueler@debian964:~$ gcc -o strcat_demo strcat_demo.c
schueler@debian964:~$ ./strcat_demo
Zeichenkette nach s1 eingeben (max. 10 Zeichen):Toast
Zeichenkette nach s2 eingeben (max. 10 Zeichen):Brot
haenge s2 an s1 an mit Aufruf strcat(s1,s2);
Inhalt von s1: >>ToastBrot<<
Inhalt von s2: >>Brot<<
```

Nach Aufruf von `strcat()` ist `s1` um `s2` verlängert.

Die letzte Version des Betriebsmittel-Programm sieht damit so aus (`betriebsmittel5.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  #define LEN 80
4  int main(void)
5  {
6      char wort[LEN+1]="";
```

```
7     char antwort[LEN+1];
8     printf("Bauelement eingeben:");
9     scanf("%80s", wort);
10    while(getchar()!='\n'){}
11    strcpy(antwort,"Betriebsmittelkennzeichnung:");
12    strcat(antwort,wort);
13    if      (strcmp(wort,"Transistor")==0)   strcat(antwort,"V");
14    else if(strcmp(wort,"Widerstand")==0)    strcat(antwort,"R");
15    else if(strcmp(wort,"Kondensator")==0)   strcat(antwort,"C");
16    else if(strcmp(wort,"Spule")==0)         strcat(antwort,"L");
17    else                                     strcat(antwort,"unbekannt");
18    printf("%s\n", antwort);
19    return 0;
20 }
```

4.12.4 Anfänge vergleichen mit `strncmp()`

Die Funktion `strncmp()` ist eine Erweiterung zu `strcmp()`: Man kann damit die Anzahl der Zeichen, die verglichen werden sollen, begrenzen:

```
1     x=strncmp("Eiszeit","Eisenbahn",3);
```

In diesem Beispiel werden nur die ersten drei Buchstaben verglichen, und damit erhält `x` den Wert null (=gleich).

4.12.5 Anfang kopieren mit `strncpy()`

Wesentlich häufiger wird eine Funktion benötigt, die nur den Anfang von `s2` nach `s1` kopiert. Die bisherige Funktion `strcpy()` hat nämlich einen Nachteil, der nicht sofort auffällt:

Terminal

```
schueler@debian964:~$ ./strcpy_demo
Zeichenkette nach s2 eingeben (max. 4 Zeichen):123456789
kopiere s2 nach s1 mit Aufruf strcpy(s1,s2);
Inhalt von s1: >>123456789<<
Inhalt von s2: >>6789<<
```

Hier ist `s2` offenbar länger als die maximale Länge von `s1`. Die Funktion `strcpy()` kopiert jedoch einfach so viele Zeichen nach `s1`, wie in `s2` enthalten sind. In diesem Beispiel wird `s2` selbst davon überschrieben, in anderen Umgebungen könnten wichtige Daten betroffen sein.

Hier kann man nun die Funktion `strncpy()` benutzen, indem man die Zeile 14 ersetzt durch (siehe `strncpy_demo.c`):

```
1     strncpy(s1,s2,4);
```

Hier das Ergebnis:

Terminal

```
schueler@debian964:~$ gcc -o strncpy strncpy_demo.c
schueler@debian964:~$ ./strncpy_demo
Zeichenkette nach s2 eingeben (max. 4 Zeichen):123456789
kopiere s2 nach s1 mit Aufruf strncpy(s1,s2);
Inhalt von s1: >>1234_123456789<<
Inhalt von s2: >>123456789<<
```

Offenbar tut `strncpy()` seine Arbeit anders als gedacht. Und richtig, die Beschreibung sagt ausdrücklich:

- Hat `s2` genau `n` oder mehr Elemente, werden genau `n` Elemente nach `s1` kopiert. Es wird kein `\0`-Zeichen angehängt.
- Hat `s2` weniger als `n` Elemente, werden zuerst alle Elemente von `s2` kopiert. Anschließend werden so viele `\0`-Zeichen angehängt, dass zusammen `n` Zeichen geschrieben wurden.

Besonders der erste Punkt (keine Terminierung) ist gefährlich. Zeile 14 ist also zu ergänzen (siehe `strncpy_demo2.c`):

```
1    strncpy(s1,s2,4); /* s2 nach s1 kopieren */
2    s1[4]='\0';       /* s1 terminieren (kann auch vorher gemacht werden) */
```

Merksatz: **Bei Aufruf der Funktion `strncpy()` muss der Zielstring in jedem Fall von Hand terminiert werden.** Eine andere Möglichkeit besteht darin, auf `strncpy()` zu verzichten und stattdessen `strncat()` (s.u.) zu verwenden:

```
1    s1[0]='\0';       /* s1 leeren */
2    strncat(s1,s2,4); /* s2 an s1 anhaengen */
```

4.12.6 Anfang anhängen mit `strncat()`

Es gibt auch eine Funktion, die maximal `n` Zeichen von `s2` an `s1` anhängt: `strncat()` erhält wie `strncpy()` einen dritten Parameter `n`, der die maximal anzuhängende Zahl an Elementen angibt. Die Zeile 19 aus `strcat_demo.c` wird zu (siehe `strncat_demo.c`):

```
1 strncat(s1,s2,sizeof(s1)-strlen(s1)-1);
```

Hier das Ergebnis:

Terminal

```
schueler@debian964:~$ gcc -o strncat strncat_demo.c
schueler@debian964:~$ ./strncat_demo
Zeichenkette nach s1 eingeben (max. 10 Zeichen):ABCDE
Zeichenkette nach s2 eingeben (max. 10 Zeichen):1234567890
haenge s2 an s1 an mit Aufruf strcat(s1,s2);
Inhalt von s1: >>ABCDE12345<<
Inhalt von s2: >>1234567890<<
```

Hier lauert im Gegensatz zu `strncpy()` keine Überraschung. `strncat()` tut, was es soll; es fügt in jedem Fall genau ein `\0`-Zeichen an. Dafür ist die Berechnung der Anzahl der Zeichen, die angefügt werden sollen, nicht ganz einfach: Der Puffer für `s1` hat 11 Zeichen, davon muss ein Zeichen für den Terminator abgezogen werden; darüberhinaus muss die aktuelle Länge von `s1` abgezogen werden; übrig bleibt die Zahl der Zeichen, die noch angehängt werden dürfen.

4.12.7 Strings zerteilen mit `strtok()`

Häufig besteht die Aufgabe, einen String in Teile (so genannte *token*)zu zerlegen:

- Ein Satz besteht aus Wörtern, getrennt durch Leerzeichen, Komma und Semikolon
- Ein Pfadname besteht aus Verzeichnis- und Dateinamen, getrennt durch Pfadnamentrenner (`/` bei Linux, `\` bei Windows)
- Jede Zeile einer CSV-Datei besteht aus Werten, die durch Komma getrennt sind (*comma separated values*)

Zu diesen Zwecken bietet sich die Funktion `strtok()` an. Die Funktion bekommt zwei Parameter, nämlich erstens den zu zerlegenden String und zweitens eine Zusammenstellung der Trennzeichen. Diese Zusammenstellung ist ebenfalls ein String. Wenn im ersten Beispiel als Trennzeichen das Leerzeichen, Komma und Semikolon erlaubt sind, lautet der Aufruf:

```
1     char *p;
2     char satz[]="Das ist das Haus, welches dem Nikolaus gehoert";
3     p=strtok(satz, " ,;");
```

Anschließend enthält `p` das erste Wort `Das\0` (mit Terminator). Mit diesem Wort kann jetzt gearbeitet werden.

Der Preis dafür ist, dass `strtok` vor das Wort `ist` einen Terminator gesetzt hat. Das heißt, dass `strtok` die Original-Zeichenkette durch seine Arbeit zerstört.

Gleichzeitig hat `strtok` eine interne `static`-Variable auf die Adresse hinter dem Terminator gesetzt.

Der nächste Aufruf sieht anders aus:

```
1     p=strtok(NULL, " ,;");
```

Der Wert `NULL` als erster Parameter gibt `strtok` die Anweisung, mit der Stelle weiter zu arbeiten, auf die die interne `static`-Variable zeigt. Als Ergebnis zeigt `p` nun auf das Wort `ist`. Hinter diesem Wort ist nun ein weiterer Terminator gesetzt worden, und der interne Zeiger zeigt auf das Wort `das`.

Mit den weiteren Aufrufen arbeiten sich der Rückgabewert und der interne Zeiger immer weiter bis zum Ende der Zeichenkette fort. Irgendwann trifft `strtok` auf das Ende der Zeichenkette. Der interne Zeiger merkt sich das (wahrscheinlich wird er auf `NULL` gesetzt), und beim nächsten Aufruf, wenn also kein weiteres Wort mehr vorhanden ist, erhält `p` den Wert `NULL`.

Insgesamt sieht der Ablauf also so aus (`strtok_demoA.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  int main(void)
4  {
5      char *p;
6      char string[]="Das ist das Haus, welches dem Nikolaus gehoert";
7      char trenner[]=" ,;";
8      printf("strtok(\"%s\",\"%s\") ergibt:", string, trenner);
9      p=strtok(string, trenner);
10     while(p!=NULL)
11     {
12         printf(">>%s<<\n", p);
13         printf("strtok(NULL,\"%s\") ergibt: ", trenner);
14         p=strtok(NULL, trenner);
15     }
16     printf("p=%li\n", p);
17     return 0;
18 }
```

Leider hat `strtok` den Nachteil, dass es mit einer internen statischen Variable arbeitet. Für kleine Programme kann man es verwenden; sobald man aber eine Zeichenkette in Felder und Unterfelder zerteilen will, kann man mit `strtok` schon nicht mehr effizient arbeiten.[3] In diesen Fällen bieten sich andere Funktionen aus `<string.h>` an:

- `p=strchr(string, c)` liefert nach `p` die Adresse, des ersten Trennzeichens `c` in `string` (einfach zu benutzen, wenn man nur ein Trennzeichen hat)

[3] Funktionen, die `strtok` benutzen, sind nicht wiedereintrittsfähig (*reentrant*). Das bedeutet, man kann nicht mehrere Versionen davon gleichzeitig auf dem Stack haben. Sie scheiden damit ebenso aus für dynamische Bibliotheken, für Rekursion sowie für Programme, die Multithreading nutzen.

- `p=strpbrk(string, trenner)` liefert nach `p` die Adresse, an der erstmalig ein Zeichen aus `trenner` in `string` vorkommt (geeignet, wenn man mehrere Trennzeichen hat)
- `p=strtok_r(string, trenner, &q)` ist die verbesserte Version von `strtok`. Hier wird die interne statische Variable durch die Adresse des Zeigers `q` ersetzt. Diese Funktion ist nur im POSIX-Standard definiert und damit zwar auf Linux und verwandten Systemen, aber nicht überall verfügbar

4.12.8 Strings formatiert schreiben mit `snprintf()`

Will man Zahlen in Strings schreiben, bietet sich die Funktion `snprintf()` an:

```
1  int erg;
2  char s[21];
3  erg = snprintf(s, 21,"%f", 1.0/7.0);
```

Dieser Programmausschnitt schreibt die Zahl $\frac{1}{7}$ mit maximal 21 Zeichen (einschließlich Terminator) in den String `s`.

Falls der Platz ausreicht, gibt der Rückgabewert `erg` die Anzahl der Zeichen zurück, die geschrieben worden sind; andernfalls erhält `erg` die Anzahl der Zeichen, die bei ausreichendem Platz ausgegeben worden wären. $erg \geq 21$ zeigt also an, dass gekürzt werden musste.

Anders als bei `printf()` sorgt `snprintf()` nicht dafür, dass ein Zeiger oder ein Cursor oder irgendetwas im String weiterbewegt wird. Wenn man also mehrere Male mit `snprintf()` auf denselben String schreibt, überschreibt man immer wieder den bisherigen Inhalt. Möchte man also einen laufenden Text mit `snprintf()` ergänzen, muss man einen eigenen Zeiger verwenden:

```
1  int erg;
2  char s[200];
3  char *p=s;
4  platz=sizeof(s);
5  erg = snprintf(p, platz,"%f\n", 1.0/17.0);
6  if(erg>=platz) return;
7  platz=platz-erg;
8  p=p+erg;
9  erg = snprintf(p, platz, "%f\n", 1.0/19.0);
10 ...
```

4.12.9 Aus Zeichenketten formatiert lesen mit `sscanf()`

Ebenso kann man Zahlen aus Zeichenketten einlesen mit Hilfe der Funktion `sscanf()`:

```
1  int erg;
2  char s[21]="4.7e-6";
3  double kap;
4  erg = sscanf(s,"%lf", &kap);
```

Dieser Programmausschnitt liest die Zeichenkette `s`, interpretiert ihren Inhalt als Gleitkommazahl und schreibt die Zahl nach `kap`.

Der Rückgabewert ist wie bei `scanf()`. Ein Unterschied zu `scanf()` liegt allerdings darin, dass bei `sscanf()` kein Datei- oder sonstiger Zeiger weiterbewegt wird. Will man mehrere Zahlen (oder andere Daten) aus demselben String lesen, muss man selbst dafür sorgen, dass man bis zum nächsten Trennzeichen weitergeht (z.B. mit Hilfe eines Zeigers):

```
1  int erg;
2  char s[21]="4.7e-6!6.8e-6!8.2e-6";
3  char *p=s;
```

```
4     double kap1, kap2, kap3;
5     erg = sscanf(p,"%lf", &kap1);
6     p=strchr(p,'!');
7     if(p!=NULL) ++p; else return;
8     erg = sscanf(p,"%lf", &kap2);
9     ...
```

`sscanf()` wird oft zusammen mit `fgets()` benutzt, um Daten aus Dateien zu entnehmen. Zuerst wird mit `fgets()` eine einzelne Zeile eingelesen; die Zeile wird dann mit `sscanf()` weiter zerlegt.