2.5 Funktionen/Parameter

2.5.1 Ziel

Hier soll gezeigt werden, wie man eine Funktion *parametrierbar* machen kann, δ , wie man sie an die Wünsche der Benutzer anpassbar machen kann.

Als Beispiel dient die Funktion `linie()`. Zur Erinnerung: Mit `linie()` soll einfach eine Linie aus ASCII-Zeichen quer auf den Bildschirm oder den Drucker gezeichnet werden.

Dazu werden die folgenden drei Funktionen vorgestellt:

- a) `alinie()`: Die Länge der Linie soll wählbar sein
- b) `blinie()`: Die Art der Linie soll wählbar sein (durchgezogen, gestrichelt, gepunktet)
- c) `clinie()`: Beides soll wählbar sein

2.5.2 Funktion mit einem Parameter

```
1 #include <stdio.h>
2 void alinie(int anzahl);
3
4 int main(void)
5 {
6     int len=30;
7     alinie(len);
8     return 0;
9 }
10 void alinie(int anzahl)
11 {
12     int lauf;
13     for(lauf=0; lauf<anzahl; ++lauf)
14     {
15         printf("*");
16     }
17     printf("\n");
18 }
```

- In Zeile 7 findet man den Aufruf von `alinie()` mit dem Wert `len` in der Klammer. Dieser Wert heißt *aktueller Parameter*. Der aktuelle Parameter wird beim Aufruf an `alinie()` übergeben.
- In Zeile 10 steht der Funktionskopf von `alinie()`. Hier gibt es in den Klammern nicht mehr wie bisher das Wort `void`, sondern eine Variable `anzahl` vom Typ `int`. Diese Variable heißt *formaler Parameter*. Sie hat nämlich eine ähnliche Aufgabe wie ein Feld in einem *Formular*, in das man etwas eintragen kann.
- In Zeile 13 wird der formale Parameter wie eine ganz normale Variable benutzt, um die Länge der Linie festzulegen.

Und was passiert genau beim Aufruf von `alinie()`? Bisher passierten bei einem Aufruf zwei Dinge: Erstens das Speichern der Rücksprungadresse und zweitens der Sprung an den Anfang von `alinie()`.

Jetzt kommt dazu noch das Kopieren des Inhalts von `len` (von Zeile 7) nach `anzahl` (in Zeile 10). Also vom aktuellen in den formalen Parameter.

In C wird übrigens immer nur die Kopie, also der Wert des aktuellen Parameters übergeben. Man nennt das *call by value*. Aus diesem Grund darf man als aktuellen Parameter, wenn man will, eine Konstante übergeben:

```
1  alinie(30);
```

Es wird dann der Wert 30 dieser Konstante in den formalen Parameter kopiert.

Im Funktionsrumpf können wir den formalen Parameter `anzahl` überall wie eine Variable benutzen: Er ist in der ganzen Funktion sichtbar (wenn man ihn nicht verdeckt) und lebt so lange, wie die Funktion läuft (also bis die schließende geschweifte Klammer erreicht wird). Nach Ende der Funktion hat der formale Parameter keinen reservierten Speicher mehr; die Funktion „vergisst“ seinen Wert.

Wir dürfen den formalen Parameter `anzahl` im Funktionsrumpf sogar verändern. Er ist ja nur eine Kopie des aktuellen Parameters `len`. Dieser ändert sich dadurch nicht. Deshalb kann man `alinie()` auch auf folgende Weise realisieren:

```
1  void alinie(int anzahl)
2  {
3      while(anzahl > 0)
4      {
5          printf("*");
6          --anzahl;
7      }
8      printf("\n");
9  }
```

Wichtig ist aber: Man darf den formalen Parameter nicht noch einmal im Funktionsrumpf mit dem gleichen Namen als Variable vereinbaren. Wenn man es doch tut, verdeckt diese lokale Variable den formalen Parameter. Man merkt das daran, dass der aktuelle Parameter keinen Einfluss darauf hat, wie die Funktion sich verhält.

```
1  void alinie(int anzahl) // Formaler Par. heisst anzahl
2  {
3      int anzahl; //<= falsch! Weitere Var. heisst auch anzahl
4      printf("%d\n", anzahl);
5      /* ... */
6  }
```

In diesem Fall wird in Zeile 4 immer irgendein Zufallswert ausgegeben (oft 0), egal, was in Zeile 1 übergeben wurde. Klar, denn `anzahl` in Zeile 4 bezieht sich auf die lokale Variable `anzahl` aus Zeile 3 und nicht auf den formalen Parameter aus Zeile 1.

2.5.2.1 Noch eine Funktion mit einem Parameter Im letzten Programm diente der Parameter dazu, dass die Funktion weiß, wie viel getan werden muss.

Im folgenden Programm, bei `blinie()`, soll das anders aussehen. Die Linie soll immer 30 Zeichen lang sein.

- `blinie(1)` soll eine durchgehende Linie ausgeben, also dreißigmal einen Unterstrich
- `blinie(2)` soll eine gestrichelte Linie ausgeben, also dreißigmal ein Minuszeichen
- `blinie(3)` soll eine gepunktete Linie ausgeben, also dreißigmal einen Punkt

```
1  #include <stdio.h>
2  void blinie(int art);
3
4  int main(void)
5  {
6      blinie(1); blinie(2); blinie(3);
```

```
7     return 0;
8 }
9 void blinie(int art)
10 {
11     int lauf;
12     for(lauf=0; lauf<30; ++lauf)
13     {
14         switch(art)
15         {
16             case 1: printf("_"); break;
17             case 2: printf("-"); break;
18             case 3: printf("."); break;
19             default: printf("?");
20         }
21     }
22     printf("\n");
23 }
```

Aufruf und Funktionskopf der Funktion sind vergleichbar mit dem ersten Beispiel. Interessant ist hier, dass man einen Parameter auch dazu einsetzen kann, der Funktion zu sagen, *was* sie machen soll.

2.5.3 Funktion mit zwei Parametern

Nun soll beides kombiniert werden, das ist für `clinie()` gefordert. Dazu braucht man zwei Parameter. Wie man sieht, geht auch das:

```
1 #include <stdio.h>
2 void clinie(int anzahl, int art);
3
4 int main(void)
5 {
6     clinie(10, 1); clinie(20, 2); clinie(30, 3);
7     return 0;
8 }
9 void clinie(int anzahl, int art)
10 {
11     int lauf;
12     for(lauf=0; lauf<anzahl; ++lauf)
13     {
14         switch(art)
15         {
16             case 1: printf("_"); break;
17             case 2: printf("-"); break;
18             case 3: printf("."); break;
19             default: printf("?");
20         }
21     }
22     printf("\n");
23 }
```

Hier gibt es zwei aktuelle Parameter (Zeile 6), die beim Aufruf übergeben werden. Sie sind durch Komma voneinander getrennt. Man spricht von einer *Parameterliste*.

Im Kopf der Funktion `clinie()` (Zeile 9) sind zwei formale Parameter vorhanden, und auch sie sind durch Komma voneinander getrennt.

Beim Aufruf wird der erste aktuelle Parameter in den ersten formalen Parameter kopiert. Genauso wird der zweite aktuelle Parameter in den zweiten formalen Parameter kopiert. Die Sprache C hat sogenannte *Positionsparameter*.

Der Aufbau einer (nicht-leeren) Parameterliste sieht immer so aus:

Typ1 Name1, Typ2 Name2, Typ3 Name3 usw.

Das ist anders als bei der Vereinbarung normaler Variablen am Blockanfang, bei der man mehrere Variablen des gleichen Typs zusammenfassen kann. Dort ist es auch so möglich:

```

1 int main(void)
2 {
3     int a, b, c; // 3 int-Variablen
4     double d, e; // 2 double-Variablen
5 }
```

Hier dagegen (in der Parameterliste) geht es nur so:

```

1 void beispiel(int a, int b, int c, double d, double e) // 5 Var.
2 { /* ... */ }
```

2.5.4 Prototyp einer Funktion mit Parametern

Auch bei einer Funktion mit Parametern gilt: Der Prototyp sieht aus wie der Funktionskopf plus ein Semikolon:

```

1 void clinie(int len, int anzahl); // Das ist der Prototyp
2
3 void clinie(int len, int anzahl) // Definition, Funktionskopf
4 { /* ... */ // Definition, Funktionsrumpf
```

Aber: Beim Prototyp dürfen die Namen der formalen Parameter fehlen. Der Prototyp sieht dann so aus:

```

1 void clinie(int, int); // Das ist der Prototyp (Kurzform)
```

Es wird aber empfohlen, die Namen trotzdem zu schreiben. Beim Benutzen der Funktion freut man sich, wenn man sich anhand der Namen erinnert, wozu die einzelnen Parameter gut sind.