4.2 Datenstrukturen/Zeiger und Funktionen

4.2.1 Funktion mit zwei Ergebnissen

Es ist sicher sehr interessant, dass es Adress- bzw. Zeigervariablen gibt. Aber wozu braucht man sie? Im folgenden Programm kommt man nicht ohne sie aus.

Die Funktion `rechteck_au()` soll für ein Rechteck den Flächeninhalt A und den Umfang U berechnen. Gegeben sind wie immer die Seitenlängen h (Höhe) und b (Breite). Hier haben wir also eine Funktion, die mit Hilfe der Eingabewerte nicht nur eine, sondern zwei Größen berechnen und zurückgeben soll.

Die beiden Größen nur *auszugeben*, wäre kein Problem: Zweimal `printf()`, und alles ist gelöst (`gcc rechteck_au0.c`):

```
1  #include <stdio.h>
2  void calc_rechteck_au(double s1, double s2)
3  {
4      double a;
5      double u;
6      a=s1*s2;
7      u=2*s1+2*s2;
8      printf("%f %f\n", a, u);
9  }
10 int main(void)
11 {
12     double h=30.0;
13     double b=20.0;
14     calc_rechteck_au(h, b);
15     return 0;
16 }
```

Hier aber sollen beide Ergebnisse an die aufrufende Funktion (z. B. `main()`) *zurückgegeben* werden.

Der `return`-Operator in C kann aber nur einen einzigen Wert zurückgeben, z. B. `0` bei der Zeile `return 0;` am Ende von `main()`. Die folgende Lösung ist daher **nicht** möglich (`gcc rechteck_au1.c`):

```
1  #include <stdio.h>
2  double,double calc_rechteck_au(double s1, double s2)
3  {
4      double a;
5      double u;
6      a=s1*s2;
7      u=2*s1+2*s2;
8      return a,u;
9  }
10 int main(void)
11 {
12     double h=30.0;
13     double b=20.0;
14     double a;
15     double u;
16     a,u=calc_rechteck_au(h, b);
17     printf("%f, %f\n", a, u);
18     return 0;
19 }
```

Nun ist es naheliegend, dass man `a` und `u` als dritten und vierten Parameter an die Funktion übergibt:

```c
#include <stdio.h>
void calc_rechteck_au(double s1, double s2, double a, double u)
{
    a=s1*s2;
    u=2*s1+2*s2;
}
int main(void)
{
    double h=30.0;
    double b=20.0;
    double a;
    double u;
    calc_rechteck_au(h, b, a, u);
    printf("%f %f\n", a, u);
    return 0;
}
```

Das lässt sich – im Gegensatz zum vorigen Beispiel – einwandfrei übersetzen. Wenn man das Programm aufruft, funktioniert es aber nicht:

Terminal

```
schueler@debian964:~$ gcc rechteck_au2.c
schueler@debian964:~$ ./a.out
-0.016440 -0.016443
```

Woran liegt das? In C werden Parameter immer als Kopien übergeben (*call by value*). Das heißt, der Inhalt der Variablen `h` wird in den Parameter `s1` kopiert, ebenso der Inhalt von `b` nach `s2`, der Inhalt der Variablen `a` in den Parameter `a` und der Inhalt der Variablen `u` in den Parameter `u`. Danach beginnt die Funktion: Hier werden die Kopien, also die Parameter `a` und `u`, verändert. Die in `main()` vereinbarten Originale (die Variablen `a` und `u` in `main()`) dagegen bleiben unverändert.

In manchen anderen Programmiersprachen gibt es noch wahlweise eine weitere Möglichkeit der Übergabe von Variablen an Parameter, nämlich *call by reference*: Dort erhält die Funktion den Zugriff auf das Original.

4.2.2 Der Trick mit den Adressen

Und wie löst man das Problem in C? Im Programm `rechteck_au3.c` ist es zu sehen:

```
1  #include <stdio.h>
2  void calc_rechteck_au(double s1, double s2, double *pa, double *pu)
3  {
4      *pa = s1*s2;
5      *pu = 2*s1+2*s2;
6  }
7
8  int main(void)
9  {
10     double h=30.0;
11     double b=20.0;
12     double a;
13     double u;
14     double *aa;
15     double *au;
16     aa = &a;
17     au = &u;
18     calc_rechteck_au(h, b, aa, au);
19     printf("%f %f\n", a, u);
20     return 0;
21 }
```

Zuerst zur `main()`-Funktion:

- Zeile 14: Die Variable `aa` ist vom Typ `double *` (Adresse einer `double`-Variablen).
- Zeile 16: Als Inhalt bekommt sie die Adresse von `a`. In der Variablen `a` soll nämlich später die Rechteckfläche stehen.
- Zeile 18: Der Inhalt von `aa` wird in den dritten Parameter `pa` kopiert. Das ist kein Problem, weil `aa` und `pa` denselben Datentyp haben (nämlich `double *`).

Jetzt geht es weiter in der Funktion `calc_rechteck_au()`:

- Zeile 4: Das Ergebnis (Höhe mal Breite) wird nach `*pa` übergeben. Was ist `*pa`? Es handelt sich um die `double`-Variable, deren Adresse in `pa` steht. Diese Adresse stammt (Zeile 18) von `aa` und gehört der Variablen `a` (Zeile 16).

Zurück in `main()`:

- Zeile 19: In `a` steht nun das Ergebnis und kann benutzt werden.

4.2.3 Kleine Vereinfachung

Im nächsten Quelltext (`rechteck_au4.c`) sieht man, dass man die beiden Zwischenvariablen `aa` und `au` einsparen kann:

```
1  #include <stdio.h>
2  void calc_rechteck_au(double s1, double s2, double *pa, double *pu)
3  {
4      *pa = s1*s2;
5      *pu = 2*s1+2*s2;
6  }
7
8  int main(void)
9  {
10     double h=30.0;
11     double b=20.0;
12     double a;
13     double u;
14     calc_rechteck_au(h, b, &a, &u);
15     printf("%f %f\n", a, u);
16     return 0;
17 }
```

In Zeile 14 wird anstelle der Variablen `aa`, die im vorigen Programm die Adresse von `a` enthalten hat, einfach direkt die Adresse von `a` übergeben. Man übergibt also einfach eine Konstante (eben die konstante Adresse von `a`) anstelle der Variablen. Und diese Konstante, die Adresse von `a`, ist vom gleichen Datentyp wie die Variable (nämlich: Adresse einer `double`-Variablen). Damit wird das Programm etwas kürzer und übersichtlicher.

4.2.4 Zusammenfassung und Kochrezept

Für jede Variable `v`, die von der Funktion aus verändert werden soll, muss vom aufrufenden Programm ihre Adresse an die Funktion übergeben werden:

```
1  int v;
2  funktion(&v);
```

Alle anderen Variablen können (und sollten auch) übergeben werden wie bisher.

In der Parameterliste im Funktionskopf wird die Adresse dieser Variablen entsprechend als Zeiger vereinbart:

```
1  void funktion(int *pv);
```

Mit dem Namen (hier `pv`) kann man sich deutlich machen, dass es sich um eine Adresse (Zeiger bzw. Pointer) handelt, indem man ihn mit dem Buchstaben `p` beginnen lässt.

Im Rumpf der Funktion greift man auf die Variable `v` zu, indem man auf den Zeiger `pv` immer den Inhaltsoperator (`*`-Zeichen) verwendet:

```
1  void funktion(int *pv)
2  {
3      *pv = 90; /* jetzt ist v=90 */
4  }
```

Nach der Funktion kann man mit der Variablen `v` wieder ganz normal umgehen:

```
1  printf("%i\n", v);
```

4.2.5 Ungültige Adresse abfangen

Der einzige Adressewert, der ungültig ist, ist der Wert 0, ausgedrückt durch die symbolische Konstante `NULL`. Falls ein Parameter diesen Wert erhält, darf die Funktion den Inhaltsoperator nicht verwenden: Dem NULL-Zeiger darf man nicht folgen. Manche Funktionsbausteine nutzen diesen Wert, um eine Unterscheidung zu treffen (zum Beispiel: nutze diesen Parameter nicht[1]). In allen anderen Fällen ist es sinnvoll, wenn der Funktionsbaustein den Parameter auf den `NULL`-Wert abfragt und in dem Fall abbricht. Im obigen Beispiel kann das so aussehen:

```
1 void funktion(int *pv)
2 {
3     if(pv==NULL)
4     {
5         return;
6     }
7     *pv = 90; /* jetzt ist v=90 */
8 }
```

Auch ein Programmabbruch ist denkbar, schließlich sollte hier die Adresse einer Variablen ankommen:

```
1 #include <assert.h>
2 void funktion(int *pv)
3 {
4     assert(pv!=NULL);
5     *pv = 90; /* jetzt ist v=90 */
6 }
```

Wenn jetzt durch einen Programmierfehler ein `NULL`-Wert ankommt, wird das Programm abgebrochen, und der Programmierer kann auf die Suche nach dem Fehler gehen.

[1] Die Funktion `strtok()` aus `string.h` macht es so.