5.1 Datenstrukturen II/Records

5.1.1 Wozu Records?

Eine Buch-Verwaltung soll zu jedem Buch folgende Daten verwalten:

```
1 char autor[31];
2 char titel[31];
3 unsigned short auflage;
4 int erscheinungsjahr;
5 char verlag[31];
6 char isbn[14];
```

Alle diese Daten gehören zu einem einzigen Gegenstand. Sie sollen im ganzen Programm gemeinsam benutzt werden (z.B. in einer Bestellung).

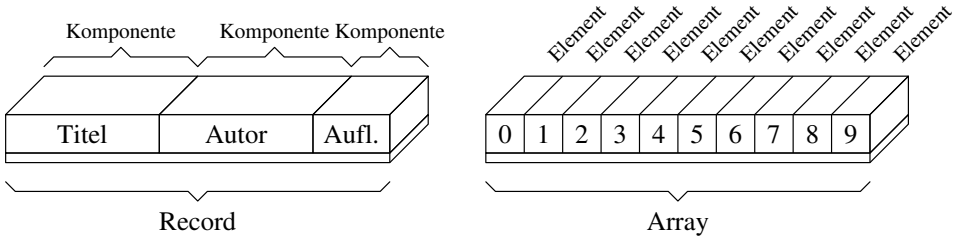

Abbildung 1: Unterschied zwischen Record und Array

Es liegt nahe, diese Daten zu einer Variablen zusammenzufassen, wie man auch Anweisungen, die zusammengehören, zu einer Funktion oder zumindest zu einem Block zusammenfasst. Da liegt es nahe, ein (eventuell zweidimensionales) Array zu benutzen (Abbildung 1 rechts).

Bei näherer Betrachtung merkt man, dass ein *Array* für diesen Zweck ungeeignet ist. Bei einem Array sind nämlich alle Elemente gleich groß und haben den gleichen Datentyp. Außerdem erfolgt der Zugriff auf jedes Array-Element über eine Nummer, so dass man beim Programmieren immer wieder nachsehen muss: War der Autor nun die Nummer `[0]` oder die Nummer `[1]`?

Es gibt aber eine andere Datenstruktur, die auch verschiedenartige Daten in einer Variablen zusammenfassen kann. Diese Datenstruktur heißt *Record* oder auch *Verbund* (Abbildung 1 links).

5.1.2 Beispiel

Im folgenden Programm wird ein Record benutzt, um die wichtigsten Daten eines Buches zu beschreiben.

```
1  #include <stdio.h>
2  /* Neuen Typ erstellen */
3  struct buchtyp
4  {
5      char autor[31];
6      char titel[31];
7      unsigned short auflage;
8  };
9  int main(void)
10 {
11     struct buchtyp bucha;
12     struct buchtyp buchb;
13     printf("A:");scanf("%30[^\n]",buchb.autor);while(getchar()!='\n'){}
14     printf("T:");scanf("%30[^\n]",buchb.titel);while(getchar()!='\n'){}
```

```
15      printf("A:");scanf("%hu",&buchb.auflage);while(getchar()!='\n'){}
16      bucha = buchb;
17      bucha.auflage=123;
18      printf("Autor:%s\n", buchb.autor);
19      printf("Titel:%s\n", buchb.titel);
20      printf("Aufl.:%hu\n",buchb.auflage);
21      return 0;
22  }
```

5.1.3 Vereinbarung

Die Typdefinition eines Records erfolgt durch Definition ihrer Bestandteile, der so genannten *Komponenten*. Anschließend können Variablen dieses neuen Typs angelegt werden.

```
1   /* Am Anfang: */
2   struct buchtyp
3   {
4       char autor[31];
5       char titel[31];
6       unsigned short auflage;
7   };
8   /* Spaeter in main() oder einer anderen Funktion: */
9   struct buchtyp bucha;
10  struct buchtyp buchb;
```

Zuerst muss also immer ein neuer Typ, z.B. **struct** buchtyp vereinbart werden (Zeilen 2 bis 7). Dieser neue Typ ist nun gleichwertig mit Typen wie **int** oder **unsigned long**. Die Typdefinition kann in main() erfolgen. Sie ist dann im ganzen main() verfügbar. Besser aber ist es, sie bereits am Anfang des Quelltextes, also vor main(), vorzunehmen, denn dann ist sie in allen folgenden Funktionen bekannt. *Danach*, also sobald der Typ **struct** buchtyp bekannt ist, können Variablen dieses Typs angelegt werden (siehe Zeilen 9 und 10 im Beispiel).

Man kann gleichzeitig einen neuen Record-Typ definieren und schon dabei auch Variablen definieren:

```
1   struct buchtyp
2   {
3       char autor[31];
4       char titel[31];
5       unsigned short auflage;
6   } bucha, buchb, buchc;
```

Das erklärt, warum in C hinter der Typdefinition eines Records ein Semikolon folgen muss[1] – das Semikolon beendet die (meistens leere) Liste der Variablen.

Das Weglassen des sogenannten Etiketts buchtyp ist zwar möglich, gibt aber hier wenig Sinn:

```
1   struct
2   {
3       char autor[31];
4       char titel[31];
5       unsigned short auflage;
6   } bucha, buchb, buchc;
```

Man kann dann keine weiteren Variablen dieses Typs mehr anlegen, **struct** als Typname reicht nämlich nicht.

[1] in Java ist das anders, daher müssen Java-Programmierer hier aufpassen

5.1.4 Initialisierung I

Records können wie Arrays initialisiert werden mit Hilfe einer Komma-getrennten Liste.

```
1  struct buchtyp buchb={"Verschiedene","Telefonbuch", 95};
```

Wie bei einem Array müssen auch hier nicht alle Komponenten initialisiert werden. Und auch hier gilt wie beim Array: Wenn man mindestens eine Komponente initialisiert, werden auch *alle anderen* Komponenten initialisiert, und zwar (je nach Typ) mit `0`, `0.0`, `NULL` oder `""`.

5.1.5 Zugriff auf Komponenten

Wie bei Arrays ist auch bei Records ein Zugriff auf den einzelnen Bestandteil (hier Komponente genannt) möglich:

```
1  bucha.auflage = 2;              /* Komponente mit Wert fuellen     */
2  bucha.auflage = buchb.auflage; /* Inhalt der Komponente kopieren */
3  scanf("%hu", &bucha.auflage);  /* Inhalt von Tastatur einlesen    */
4  printf("%hu", bucha.auflage);  /* Inhalt auf Bildschirm ausgeben */
```

Beim Umgang mit den Komponenten, die selbst Arrays sind (z.B. Strings), muss man nur das beachten, was man bei Arrays immer beachten muss:

```
1  strcpy(bucha.titel, "CMOS–Kochbuch"); /* strcpy statt =       */
2  strcpy(bucha.titel, buchb.titel);     /* strcpy statt =       */
3  scanf("%30[^\n]", bucha.titel);       /* kein &, weil Zeiger */
4  printf("%s", bucha.titel);
```

5.1.6 Initialisierung II

Schon seit C99 kann man die Initialisierung auch mit dem Punkt-Operator vornehmen:

```
1  struct buchtyp buchb={.autor="Verschiedene",
2                        .auflage=95,
3                        .titel="Telefonbuch"};
```

Auch hier gilt: Wenn man ein Record initialisiert, dann werden alle *nicht explizit genannten* Komponenten mit `0`, `0.0`, `NULL` oder `""` initialisiert.

5.1.7 Zugriff auf den ganzen Record

Über den Variablennamen ist ein Zugriff auf alle Komponenten miteinander möglich. Dabei kann man – anders als bei Arrays – einen Record komplett an einen anderen zuweisen:

```
1  bucha = buchb; /* alle Komponenten, auch die Arrays darin! */
```

Übrigens kann man an einen Record auch eine Art Record-Konstante (nennt sich *Compound Literal*, seit C99) zuweisen. Sie sieht aus wie die Werteliste bei der Initialisierung. Hier man muss nur dafür sorgen, dass der Compiler den Typ der Konstante richtig ermittelt. Das geschieht mit Hilfe der expliziten Typkonvertierung:

```
1  bucha = (struct buchtyp){"Meier","Kochbuch", 20};
```

Zum Schluss noch eine kleine Einschränkung von Records: Der Vergleich zweier Records mit dem Vergleichsoperator ist nicht erlaubt.

```
1    if(bucha == buchb) /* wird nicht compiliert! */
2      printf("gleich");
3    else
4      printf("verschieden");
```

5.1.8 Graphische Darstellung

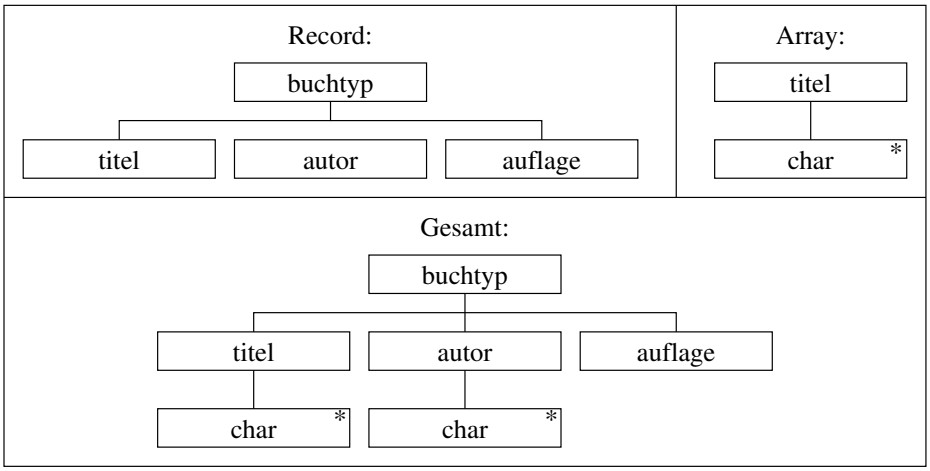

Abbildung 2: Jackson-Diagramm

Mit der Datenstruktur Record kann man sehr große Datentypen zusammenstellen. Deshalb ist es sinnvoll, solch einen Datentyp für Nicht-Programmierer als Bild darzustellen. Dafür dient das Jackson-Diagramm[2]. In Abbildung 2 ist das für den Datentyp `struct buchtyp` gezeichnet.

Oben links wird deutlich, wie ein Record symbolisiert wird, nämlich durch einen Baum. Man erkennt: `buchtyp` enthält `titel`, `autor` und `auflage`.

Oben rechts ist zu sehen, wie ein Array symbolisiert wird: Ein Baum mit nur einem Zweig, der aber im unteren Kasten einen Stern enthält. Der Stern steht hier für „viele“. `titel` kann also viele `char`-Elemente enthalten.

Im unteren Teil der Abbildung sieht man, wie die beiden Einzelheiten zu einem Gesamtdiagramm vereinigt werden.

5.1.9 Ein Record enthält einen anderen Record

Ein Record-Typ kann einen anderen Record als Komponente enthalten:

```
1  struct adresstyp{
2     char strasse[30];
3     int hausnummer;
4  };
5  struct buchtyp{
6     char autor[31];
7     char titel[31];
8     unsigned short auflage;
9     struct adresstyp verlagadr; // <-- hier
10 };
```

[2] gibt es auch für Programmstrukturen

Der Zugriff auf ein inneres Element erfolgt dann mit zwei Punkt-Operatoren (Zeile 4):

```
1 struct adresstyp adr={"Papierstrasse",39};
2 struct buchtyp telbuch;
3 telbuch.adresse=adr;
4 telbuch.adresse.hausnummer=39; // hier
```

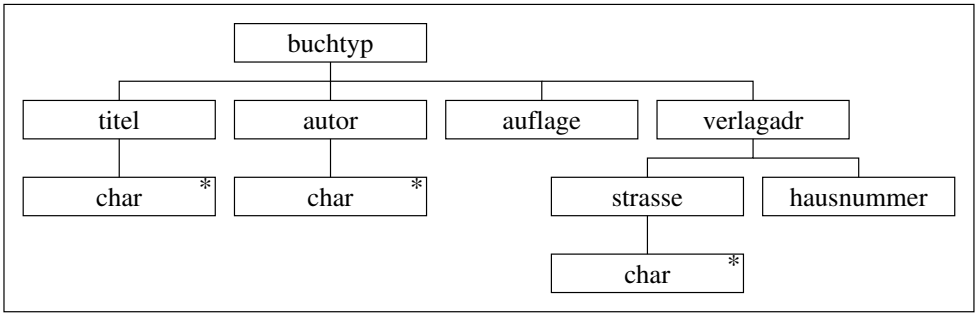

Abbildung 3: Jackson-Diagramm: Record enthält Record

Das Jackson-Diagramm muss nun entsprechend ergänzt werden (Abbildung 3).