## 5.1 Einführung und Klassen/Neue Möglichkeiten für structs

### 5.1.1 Problem

Bei der Programmierung mit Records ist es oft so, dass es zu bestimmten Datentypen (z. B. `struct persontyp`) passende Funktionen gibt (z. B. `printperson(struct persontyp p)`). Diese Funktionen benutzen die passenden Records als Parameter (wie bei `printperson()`) oder als Rückgabetyp (wie bei `scanperson()`).

Da wäre es sehr bequem, wenn man die Records und die dazu gehörenden Funktionen zusammenfassen könnte, so dass sie räumlich zusammen im Quelltext stehen.

Noch besser wäre es, wenn sie so als Einheit zusammengefasst wären, dass man sie nur gemeinsam benutzen kann. Die Records sollten für andere Funktionen nicht sichtbar sein und die Funktionen sollten nur mit den richtigen Records aufgerufen werden können.

Theoretisch gäbe es da zwei Möglichkeiten:

a) Man legt das Record als `static`-Variable in einer Funktion( z. B. `printperson()`) an. Dann kann nur die Funktion auf das Record zugreifen. Für alle anderen ist es nicht sichtbar. – Schade nur, dass das allein mit einer Funktion klappt. Was ist mit `scanperson()`? Also müsste man alle Funktionalität für das Record in eine einzige Funktion hineinkopieren.

b) Man legt die Funktionen in das Record[1]. – Das sieht am Anfang sehr ungewöhnlich aus, und es stellt an manchen Stellen (Top-Down-Programmierung) die Vorgänge auf den Kopf – aber es funktioniert!

### 5.1.2 So wird's gemacht

Das folgende Beispiel ist einmal in C und einmal in C++ programmiert. Die C++-Version zeigt, wie in der Objektorientierung Daten und Funktionen zusammengefasst werden.

#### 5.1.2.1 C-Version ohne Objektorientierung

Zuerst die „herkömmliche" C-Version (`pkw.c`):

```
1  #include <stdio.h>
2  #include <string.h>
3  struct pkwtyp
4  {
5      char kennz[20];
6      int kmstand;
7  };
8  /*********************************************/
9  void printpkw(struct pkwtyp p)
10 {
11     printf("%s, %d\n", p.kennz, p.kmstand);
12 }
13 /*********************************************/
14 int main(void)
15 {
16     struct pkwtyp fw;
17     strcpy(fw.kennz,"BI-XY123");
18     fw.kmstand=20000;
19     printpkw(fw);
20     return 0;
21 }
```

[1] In C ginge das mit Funktionszeigern. Sehr flexibel, aber nicht ganz einfach

**Zeile 3–7** Zu einem Fahrzeug eines Fuhrparks werden Kennzeichen und Kilometerstand in einem Record `struct pkwtyp` erfasst.

**Zeile 9–12** Mit `printpkw(x)` kann man diese Daten des Records `x` ausgeben.

**Zeile 16–18** In `main()` wird ein Record angelegt. Danach Zugriffe auf Komponenten des Records

**Zeile 19** Ausgabe der Daten des Records `fw`

**5.1.2.2 C++-Version mit Objektorientierung** Und nun die für uns neue C++-Version (`pkw.cpp`):

```
1  #include <stdio.h>
2  #include <string.h>
3  struct pkwtyp
4  {
5      char kennz[20];
6      int kmstand;
7      void printpkw(void)
8      {
9          printf("%s, %d\n", kennz, kmstand);
10     }
11 };
12 /*********************************************/
13 int main(void)
14 {
15     int x;
16     struct pkwtyp fw;
17     strcpy(fw.kennz,"BI-XY123");
18     fw.kmstand=20000;
19     fw.printpkw();
20     return 0;
21 }
```

**Zeile 3–11** Auch hier ein Record mit Kennzeichen und Kilometerstand

**Zeile 7–10** `printpkw()` liegt im Record selbst!

**Zeile 9** `kennz` und `kmstand` sind global zur Funktion `printpkw()`, deshalb hier sichtbar.

**Zeile 19** Der Zugriff auf `printpkw()` jetzt mit Angabe der Record-Variablen und Punkt-Operator – wie bei jeder anderen Record-Komponente

Was ist anders?

a) Neu ist, dass die Funktion `printpkw()`, die auf dieses Record zugreift, *im Record selbst* definiert ist: Die Funktion liegt im Record.

b) Daraus folgt die Einschränkung: Die Funktion ist also auch nur im Record sichtbar.

c) Andererseits sind für diese Funktion die Variablen `kennz` und `kmstand` quasi global: In der Funktion `printpkw()` kann – unabhängig von Parameter und Rückgabewert – jederzeit auf `kennz` und `kmstand` zugegriffen werden. Deshalb hat `printpkw()` hier eine leere Parameterliste (Zeile 9).

d) Und es ändert sich jetzt die Art, wie man die Funktion aufruft: Sie liegt im Record, also wird sie nicht mehr mit `printpkw(fw)` aufgerufen, sondern mit dem Zugriffsoperator: `fw.printpkw()`.

Durch diese kleine Änderung ergeben sich viele Konsequenzen (dazu später).

### 5.1.3 Begriffe der Objektorientierung

Es ist sinnvoll, im Zusammenhang mit den neuen, erweiterten Records neue Begriffe einzuführen, die aus der Objektorientierung stammen:

a) Der Datentyp `struct pkwtyp` heißt jetzt statt Datentyp *Klasse*.

b) Die Variable `fw` heißt nicht mehr Variable, sondern *Objekt*. Man sagt auch, `fw` ist eine *Instanz* (=Realisierung) der Klasse `struct pkwtyp`.

c) Die Mitgliedsvariablen der Klasse (`kennz` und `kmstand`) heißen jetzt *Attribute*.

d) Die Mitgliedsfunktion `printpkw()` heißt *Methode*.

### 5.1.4 UML

UML heißt *unified modelling language* und ist eine ganze Gruppe von Diagrammformen in der objektorientierten Programmierung[2].

Eine dieser (momentan 14) Diagrammformen ist das UML-Klassendiagramm. Abbildung 1 zeigt das UML-Klassendiagramm für `struct pkwtyp`. Das Klassendiagramm ist ein dreifach

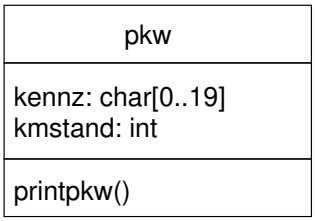


Abbildung 1: Beispiel für ein UML-Klassendiagramm

unterteiltes Rechteck. Im oberen Teil des Rechtecks ist der Klassen- oder Objektname verzeichnet. Falls es sich um ein konkretes Objekt (also um eine Variable) handelt, ist der Name unterstrichen, sonst nicht.

Im mittleren Teil des Rechtecks findet man die Namen der Attribute. Der Datentyp wird, falls angegeben, durch einen Doppelpunkt getrennt hinter den Namen gestellt. Die Notation ist ähnlich wie bei ST/SCL (oder auch Pascal/Delphi).

Im unteren Teil des Rechtecks findet man die Namen der Methoden mit angehängter Parameterliste (Parameter wieder in der Form Name, Doppelpunkt, Typ). Eine leere Parameterliste wird hier (anders als in C) als leeres Klammerpaar dargestellt. Rückgabetypen werden hinter einem Doppelpunkt genannt (außer bei `void`).

In UML-Klassendiagrammen kann man auch mehrere Klassen und Objekte und ihre Beziehungen untereinander darstellen.

[2]Manche dieser Formen gab es schon vorher, aber in UML sind sie zusammengefasst und vereinheitlicht worden.