# 13 Strukturen, Unionen und Bitfelder

Programme verarbeiten häufig **Datenstrukturen**, welche in sich eine Semantik enthalten. Beispielsweise könnte ein Programm für die Personalabteilung für jede Person einen Eintrag mit den folgenden Elementen speichern:

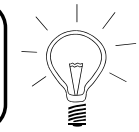
| Personal-nummer | Nachname | Vorname | Straße | Haus-nummer | Postleit-zahl | Wohnort | Gehalt |
|---|---|---|---|---|---|---|---|

Die einzelnen Elemente können beispielsweise die folgenden Typen haben:

| `int` | `char[20]` | `char[20]` | `char[20]` | `int` | `int` | `char[20]` | `float` |
|---|---|---|---|---|---|---|---|

Dies wird als eine zusammengesetzte Datenstruktur, oder in C schlicht als „**Struktur**“ bezeichnet. Eine solche Struktur enthält unterschiedliche Typen und fasst sie unter einem einzigen Typ zusammen, welcher eine durch die Teile festgelegte Anzahl an Bytes besitzt.

> Eine Struktur enthält semantisch zusammengehörige Daten in einem zusammengesetzten Datentyp.



Diese Art der Datenstruktur konnte man bereits in COBOL definieren. Als ein eigenständiger Datentyp wurde sie dann von N. Wirth in Pascal eingeführt und dort „Record“ genannt. Der Name Record wurde nach dem Satz einer Datei auf der Festplatte gewählt, welcher im Englischen ebenfalls „record“ genannt wird. Ein solcher „record“ auf der Platte konnte Komponenten verschiedener Typen enthalten.

In Anlehnung an den Plattensatz werden die Komponenten eines Records oder einer Struktur oft als „**Feld**“ beziehungsweise „Datenfeld“ bezeichnet. Vor allem in der Java-Literatur wird der Begriff „Datenfeld“ bevorzugt, während in C jedoch auch die Begriffe „Komponente“ oder gerade in C++ auch „**Member**“ gängig sind.

Nebst Strukturen gibt es auch noch andere zusammengesetzte Typen wie Unionen und Bitfelder. Auf diese wird in den folgenden Unterkapiteln ebenfalls eingegangen, sie werden jedoch in der modernen Programmierung äußerst selten verwendet.

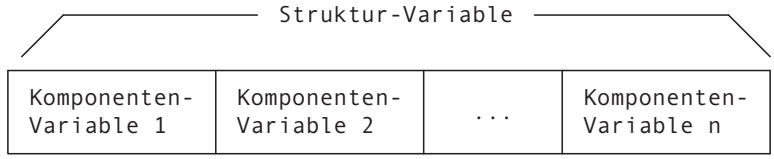
---

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_13.


J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_13

## 13.1 Strukturen

In der Programmiersprache C werden zusammengehörige Variablen in einer sogenannten „**Struktur**“ zusammengefasst. Eine solche Struktur wird mittels eines sogenannten „Struktur-Typs“ definiert und die daraus resultierende Variable wird „Struktur-Variable“ genannt.

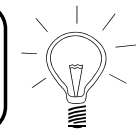



Ein Struktur-Typ ist ein selbst definierter zusammengesetzter Datentyp, welcher aus einer festen Anzahl von Komponenten besteht, die jeweils einen Namen haben. Um einen Struktur-Typ festzulegen, wird das Schlüsselwort **`struct`** verwendet. Die allgemeine Form für einen Struktur-Typ lautet:

```
struct Strukturname {
  KomponentenTyp1 komponente1;
  KomponentenTyp2 komponente2;
  ...
  KomponentenTypN komponenteN;
};
```

Im Unterschied zu den Komponenten eines Arrays können die Komponenten einer Struktur verschiedene Typen haben.

In der Typ-Definition einer Struktur muss für jede Komponente deren Namen und Typ angegeben werden.



Der hier selbst definierte Typ-Bezeichner ist `struct Strukturname`, wobei `struct` ein Schlüsselwort ist und `Strukturname` als das sogenannte „**Etikett**“ der Struktur (auf englisch „**structure tag**“) bezeichnet wird. Dieser Datentyp ist definiert durch den Inhalt der geschweiften Klammern.

Hier ein einfaches Beispiel für eine Struktur:

```
struct Adresse {
  char strasse[20];
  int  hausnummer;
  int  postleitzahl;
  char stadt[20];
};
```

Hier ist zu beachten, dass alle Namen der Komponenten einer Struktur verschieden sein müssen. In verschiedenen Strukturen dürfen jedoch Komponenten gleichen Namens auftreten, da jede Struktur einen eigenen Namensraum hat (siehe Kapitel → 14.3). Im Beispiel ist auch zu sehen, dass Strukturen ganze Arrays beinhalten können.

Ein weiteres Beispiel:

```
struct Student {
  int matrikelnummer;
  char name[20];
  char vorname[20];
  struct Adresse wohnort;
};
```

Hier wurde nun eine Struktur deklariert, bei welcher die Komponente `wohnort` wiederum vom Typ einer Struktur ist. Die Definition des Struktur-Typs `struct Adresse` muss dabei zwingend seiner Verwendung im Datentyp `struct Student` vorausgehen.

Um Variablen mit diesen Struktur-Typen zu definieren, schreibt man folgendes:

```
struct Adresse zuhause;
struct Student meyer;
struct Student studenten[50];  // ein Array aus 50 Studenten
```

Hierbei wird ersichtlich, dass das Schlüsselwort `struct` zum Typ dazugehört. Ohne das Schlüsselwort würde der Compiler das Etikett nicht finden.

Um die Komponenten einer Struktur-Variablen anzusprechen, werden sie nicht wie ein Array mittels eines Index, sondern mittels des **Punkt-Operators** und der Angabe des gewünschten Feldnamens dereferenziert:

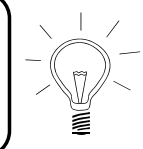
```
meyer.matrikelnummer = 716347;
```

Dabei wird der Punkt-Operator sowohl für lesenden wie auch schreibenden Zugriff verwenden. Hier wurde die Komponente `matrikelnummer` mit dem Wert `716347` belegt.

Durch mehrfaches Verwenden des Punkt-Operators können direkt auch Komponenten von Komponenten angesprochen werden:

```
meyer.wohnort.postleitzahl = 73733;
```

> Bei mehrfach zusammengesetzten Strukturen kann man über das mehrfache Anwenden des Punkt-Operators auf geschachtelte Komponenten zugreifen.



Einzelheiten zum Punkt-Operator folgen in Kapitel → 13.1.7.

### 13.1.1 Definition von Struktur-Variablen

Es ist möglich, gleichzeitig den Typ einer Struktur festzulegen sowie auch Variablen dieses neuen Typs zu definieren. Beispielsweise:

```
struct Student {
  ...
} tina;
```

Hier wird sowohl der Typ `struct Student` festgelegt als auch eine Variable `tina` vom Typ dieser Struktur definiert. Weitere Variablen dieses Typs können später über den Typnamen `struct Student` definiert werden.

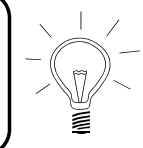
Das Etikett der Struktur kann auch weggelassen werden. Dies ist jedoch nur dann sinnvoll, wenn sofort alle Variablen eines Typs definiert werden, da ohne ein Etikett später keine Variablen dieses Typs mehr vereinbart werden können.

```
struct {
  ...
} heinz, heinrich, anita;
```

Man beachte, dass der Name eines Etiketts, der Name einer Komponente und der Name einer Struktur-Variablen identisch sein dürfen, denn sie liegen in verschiedenen Namensräumen (siehe Kapitel → 14.3). Dies macht dem Compiler keine Probleme, da er aus dem Kontext schließt, um welche Größe es sich handelt. Eine solche Namensgleichheit ist jedoch dennoch nicht zu empfehlen, da auch Menschen den Code lesen müssen, was leicht zu Missverständnissen führen kann.

### 13.1.2 Initialisierung einer Struktur mit einer Initialisierungsliste

Eine Initialisierung einer Struktur-Variablen kann direkt bei der Definition der Struktur-Variablen mit Hilfe einer **Initialisierungsliste** durchgeführt werden.

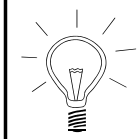


Dies zeigt das folgende Beispiel:

```
struct Student mueller =
{
  66202,
  "Mueller",
  "Herbert",
  {
    "Schillerplatz",
    20,
    73730,
    "Esslingen"
  }
};
```

Die Initialisierungsliste enthält die Werte für die einzelnen Komponenten getrennt durch Kommas. Da die Komponenten-Variable `wohnort` selbst eine Struktur ist, erfolgt die Initialisierung der Komponenten-Variable `wohnort` wieder über eine Initialisierungsliste.

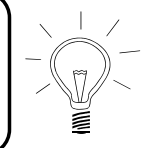
> Array- und Struktur-Typen werden in C auch als Aggregat-Typen bezeichnet. Ein Aggregat-Typ ist ein anderes Wort für „zusammengesetzter Typ“. Wegen dieser Gemeinsamkeit erfolgt die Initialisierung von Strukturen und Arrays analog.



Struktur-Variablen können auch durch die Zuweisung einer Struktur-Variablen des gleichen Typs oder durch einen beliebigen Ausdruck, der zu einer Struktur-Variablen auswertet (beispielsweise ein Funktionsaufruf), initialisiert werden. In C11 kann eine Struktur-Variable auch mittels eines sogenannten compound literals (siehe Kapitel → 13.2.2) initialisiert werden.

### 13.1.3 Initialisierung einzelner Komponenten einer Struktur

> Seit C99 können einzelne Komponenten einer Struktur-Variablen initialisiert werden. Dies wird im Englischen als „**designated initializer**“ bezeichnet.



Beispielsweise kann eine Variable vom Typ `struct Student` wie im folgenden Beispiel initialisiert werden:

```
struct Student petra = {
  .matrikelnummer = 4711,
  .vorname = "Petra"
};
```

Man muss dabei nicht auf die Reihenfolge der Komponenten achten. Man hätte auch schreiben können:

```
struct Student petra = {
  .vorname = "Petra",
  .matrikelnummer = 4711
};
```

Werden die Elemente einer Struktur in der gleichen Reihenfolge initialisiert, wie sie bei der Definition des Typs angegeben wurden, so muss man den Elementnamen nicht angeben wie im folgenden Beispiel:

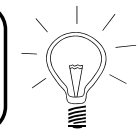
```
struct Student petra = {
  4711,        // Initialisierung der Matrikelnummer
  "Welsch",    // Initialisierung des Namens
  "Petra"      // Initialisierung des Vornamens
};
```

Unvollständige Initialisierungen und Initialisierungen ohne Angabe des Elementnamens sind eine große Fehlerquelle und führen zu Unleserlichkeit. Das zu initialisierende Element sollte stets angegeben werden. Unvollständige Initialisierungen sollten vermieden werden.


Will man auf eine bestimmte Komponente zugreifen und anschließend in derselben Reihenfolge weiterinitialisieren, so braucht man die Namen der folgenden Komponenten nicht anzugeben, wie im folgenden Beispiel:

```
struct Student petra = {
  .name = "Welsch",    // Initialisierung des Namens
  "Petra"              // Initialisierung des Vornamens
};
```

Werden nicht alle Komponenten initialisiert, so werden die nicht aufgeführten Elemente automatisch mit null initialisiert.



Hierfür eine detaillierte Aufstellung:

```
struct Student petra = {
                     // Initialisierung der Matrikelnummer mit 0
  .name = "Welsch",  // Initialisierung des Namens
  .vorname = "Petra" // Initialisierung des Vornamens
                     // Initialisierung der Adresse: Der Compiler
                     // schreibt an die Speicherstelle der
                     // Struktur genau sizeof(struct Adresse) Nullen.
};
```

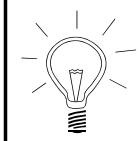
### 13.1.4 Strukturen und Pointer auf Strukturen in Strukturen

Wie im Beispiel gezeigt, kann eine Struktur selbst wiederum Strukturen beinhalten: Ein Student besitzt eine Adresse. Damit die Struktur `Student` jedoch definiert werden kann, muss die Struktur `Adresse` bereits definiert sein. Es ist somit erforderlich, dass die Definition von `Adresse` vor `Student` kommt.

Wird hingegen anstelle der Struktur selbst nur ein **Pointer auf die Struktur** verwendet, so benötigt der Compiler nur den Namen der Struktur.

```
struct Adresse* wohnort;
```

> Für die Definition eines Pointers auf eine Struktur müssen die Größe und der Aufbau der Struktur nicht bekannt sein, da alle Pointer-Typen dieselbe Anzahl Bytes beinhalten. Damit kann in der Definition einer Struktur eine Pointer-Variable als Komponente definiert werden, die auf eine Variable vom Typ der Struktur zeigt.



Die Verwendung von Pointern auf Strukturen wird insbesondere bei dynamischen Datenstrukturen verwendet. Siehe dazu Kapitel → 19.

Es ist zu beachten, dass, obschon die Struktur selbst nicht definiert sein muss, die **Vorwärtsdeklaration** des Namens der Struktur zwingend notwendig ist:

```
struct Adresse;

struct Student {
  ...
  struct Adresse* wohnort;
};

struct Adresse {
  ...
};
```

Demzufolge darf eine Struktur somit auch nur einen Pointer auf sich selbst beinhalten, niemals sich selbst. Folgender Code erzeugt somit einen Fehler:

```
struct Test {
  struct Test testStruktur;  // Fehler
};
```

Dies würde voraussetzen, dass im Definitionsteil von `struct Test` die Definition von `struct Test` bereits bekannt ist, was nicht der Fall ist, da die Definition noch nicht abgeschlossen ist. Außerdem würde dies zu einer Rekursion ohne Abbruchkriterium führen und somit eine Struktur unendlicher Größe entstehen.

### 13.1.5 String-Variablen in Strukturen

String-Variablen in Strukturen können `char`-Arrays oder Pointer-Variablen vom Typ `char*` sein.

Dies zeigt das folgende Beispiel:

```
struct Name {
  char name[20];
  char* vorname;
};
```

Im Falle des `char`-Arrays „gehören“ alle Zeichen zu der entsprechenden Struktur-Variablen. Im Falle der Pointer-Variablen vom Typ `char*` steht in der entsprechenden Komponente nur ein Pointer auf einen String, das heißt auf ein `char`-Array, das sich nicht in der Struktur selbst befindet.

In beiden Fällen kann die Initialisierung mit String-Konstanten erfolgen:

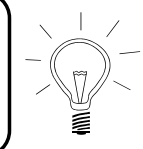
```
struct Name maier = {"Maier", "Herbert"};
```

Bei Änderungen des Strings ist im Falle, dass die Komponente ein `char`-Array ist, die Funktion `strcpy()` (siehe Kapitel → 12.9.1) zu verwenden. Im Falle der Pointer-Variablen kann einfach ein neuer Pointer zugewiesen werden. Über die Unterschiede zwischen `char`-Arrays und Pointer wurde bereits in Kapitel → 12.5 geschrieben.

### 13.1.6 Operationen auf Komponenten und ganzen Struktur-Variablen

Die Komponenten einer Struktur-Variablen können mittels des Punkt-Operators angesprochen werden. Für sie gelten dieselben Regeln, wie wenn sie eigenständige Variablen wären.

> Auf Komponenten-Variablen sind diejenigen Operationen zugelassen, die für den entsprechenden Komponenten-Typ möglich sind.



Beispielsweise können die Adressen von Komponenten-Variablen mit Hilfe des Adress-Operators bestimmt werden wie im folgenden Beispiel:

```
char* namePtr = &meyer.name;
```

Doch auch auf ganzen Strukturen, sprich den Struktur-Variablen selbst, können Operationen durchgeführt werden.

So kann der Inhalt einer Struktur-Variablen einer anderen Struktur-Variablen mittels des Zuweisungs-Operators zugewiesen werden, sofern sie denn vom gleichen Struktur-Typ ist:

```
struct Adresse adresse1 = {"Ambrosiastrasse", 52, 68759, "Borrheim"};
struct Adresse adresse2;
adresse2 = adresse1;
```

Hierbei werden sämtliche Elemente der Struktur von `adresse1` nach `adresse2` kopiert.

> Vorsicht ist geboten bei Komponenten, die Pointer sind: Bei einer Zuweisung wird nur der Pointer kopiert, nicht aber der Inhalt, auf den der Pointer zeigt.


Im Gegensatz zum Zuweisungs-Operator ist ein Vergleich von zwei Struktur-Variablen mit Vergleichs-Operatoren hingegen nicht möglich. Strukturen muss man immer komponentenweise vergleichen.

Die Größe eines Struktur-Typs oder einer Struktur-Variablen im Arbeitsspeicher kann nicht aus der Größe der einzelnen Komponenten berechnet werden, da Compiler die Komponenten oft auf bestimmte Wortgrenzen (Alignment) legen. Zur Ermittlung der Größe einer Struktur im Arbeitsspeicher muss der Operator `sizeof` (siehe Kapitel → 9.9.1) verwendet werden, wie beispielsweise `sizeof(struct Student)`. Mit Hilfe des `_Alignof`-Operators kann seit dem C11-Standard das Alignment der ganzen Struktur ermittelt werden.

Die Adresse einer Struktur-Variablen `a` wird wie bei Variablen von einfachen Datentypen mit Hilfe des Adress-Operators ermittelt, das heißt durch `&a`. Der resultierende Pointer zeigt dabei auf das allererste Element der Struktur.

### 13.1.7 Selektion der Komponenten: Punkt- und Pfeil-Operator

Für den Zugriff auf Komponenten steht grundsätzlich der **Punkt-Operator** zur Verfügung. Sehr häufig jedoch sind Strukturen mittels Pointer verfügbar, beispielsweise, da sie als solche an Funktionen übergeben werden. Im folgenden Beispiel wird mittels des Adress-Operators ein solcher Pointer definiert:

```
struct Adresse* adressePtr = &adresse1;
```

Um eine solche Pointer-Struktur-Variable nun korrekt zu dereferenzieren, muss man schreiben:

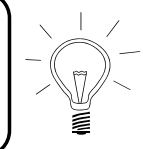
```
(*adressePtr).hausnummer = 55;
```

Die Klammern sind notwendig, da der Punkt-Operator höher priorisiert wird, wie der Dereferenzierungs-Operator. Da diese Schreibweise sehr umständlich ist, wurde in C der sogenannte „**Pfeil-Operator**“ eingeführt, welcher genau dasselbe tut:

```
adressePtr->hausnummer = 55;
```

Der Pfeil-Operator `->` wird an der Tastatur durch ein Minuszeichen und ein Größerzeichen geschrieben.

Die Selektions-Operatoren (Auswahl-Operatoren) `.` und `->` haben die gleiche Vorrangstufe. Sie werden von links nach rechts abgearbeitet.



Beachten Sie, dass zwar der Operand des Punkt-Operators `.` ein L-Wert sein muss, nicht aber unbedingt der Operand des Pfeil-Operators `->`. Dieser benötigt nur einen Pointer, sprich eine Adresse, welche aus einem beliebigen Ausdruck entstehen kann. Dies wird insbesondere wichtig bei mehrfach zusammengesetzten Strukturen.

### 13.1.8 Mehrfach zusammengesetzte Strukturen

Im Folgenden werden Struktur-Variablen als Komponenten eines Datentyps betrachtet. Mit anderen Worten, es sollen mehrfach zusammengesetzte Strukturen eingeführt werden. Als Anwendungsbeispiel soll der Fitnessplan einer Person dienen. Wir definieren folgende beiden Datenstrukturen:

```
struct Lauf {
  float km;           // Anzahl gerannte Kilometer
  int   hoehenmeter;  // Zurueckgelegter Hoehenunterschied
  int   s;            // Benoetigte Sekunden fuer die Strecke
};

struct Trainingsplan {
  struct Lauf morgenrunde;
  struct Lauf mittagsrunde;
  struct Lauf abendrunde;
};
```

Nehmen wir an, die Person habe am Montag ihr Training absolviert. So sei `montag` eine Variable vom Typ `struct Trainingsplan`:

```
struct Trainingsplan montag;
```

Die Person absolviert ihr Training und läuft folgende Strecken:

```
montag.morgenrunde.km = 5;
montag.morgenrunde.hoehenmeter = 60;
montag.morgenrunde.s = 30 * 60;
montag.mittagsrunde.km = 3;
montag.mittagsrunde.hoehenmeter = 0;
montag.mittagsrunde.s = 15 * 60;
montag.abendrunde.km = 20;
montag.abendrunde.hoehenmeter = 200;
montag.abendrunde.s = 80 * 60;
```

In der Datenstruktur können somit die entsprechenden Felder direkt mittels des Punkt-Operators verschachtelt angesprochen werden.

Da die Person täglich trainiert und sie es leid ist, immer alle Daten einzutragen, fügt sie einen zusätzlichen Datentyp hinzu und schreibt den Datentyp `Lauf` um:

```
struct Strecke {
  float km;                // Anzahl gerannte Kilometer
  int   hoehenmeter;       // Zurueckgelegter Hoehenunterschied
};

struct Lauf {
  struct Strecke* strecke; // Pointer auf die Strecke
  int             s;       // Benoetigte Sekunden fuer die Strecke
};
```

So kann die Person ihre Lieblingsstrecken folgendermaßen definieren:

```
struct Strecke finnenbahn = {5, 60};
struct Strecke seeweg     = {3, 0};
struct Strecke waldweg    = {20, 200};
```

Dadurch kann sie ihre Runden folgendermaßen aufschreiben:

```
montag.morgenrunde.strecke = &finnenbahn;
montag.morgenrunde.s = 30 * 60;
montag.mittagsrunde.strecke = &seeweg;
montag.mittagsrunde.s = 15 * 60;
montag.abendrunde.strecke = &waldweg;
montag.abendrunde.s = 80 * 60;
```

Und wenn sie am Dienstag dieselben Strecken nochmals abläuft, kann sie dieselben Pointer nochmals verwenden.

```
dienstag.morgenrunde.strecke = &finnenbahn;
dienstag.morgenrunde.s = 25 * 60;
...
```

Um nun beispielsweise die Statistik vom Training am Dienstagmorgen auszurechnen, kann folgendes geschrieben werden:

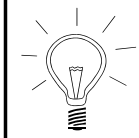
```
printf("%.0f Meter in %d:%02d Minuten = %.1f km/h\n",
  dienstag.morgenrunde.strecke->km * 1000.f,
  dienstag.morgenrunde.s / 60,
  dienstag.morgenrunde.s % 60,
  dienstag.morgenrunde.strecke->km / dienstag.morgenrunde.s * 3600);
```

Dieses gesamte Beispiel findet sich in der Datei `sportler.c` und erzeugt folgende Ausgabe:

```
5000 Meter in 30:00 Minuten = 10.0 km/h
```

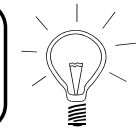
### 13.1.9 Übergabe von Struktur-Variablen an Funktionen und Rückgabe

Strukturen werden als zusammengesetzte Variablen komplett an Funktionen übergeben. Es gibt hier keinen Unterschied zu Variablen von einfachen Datentypen wie beispielsweise `float` oder `int`.



Man muss nur einen formalen Parameter vom Typ der Struktur einführen und als aktuellen Parameter eine Struktur-Variable dieses Typs übergeben.

Auch die Rückgabe einer Struktur-Variablen unterscheidet sich nicht von der Rückgabe einer einfachen Variablen.



Der Rückgabetyp der Funktion muss selbstverständlich vom Typ der Struktur-Variablen sein, die zurückgegeben werden soll.

Bei der Übergabe von Strukturen werden sämtliche Komponenten genau gleich wie bei einer Zuweisung (siehe Kapitel → 13.1.6) kopiert. Dies kann sehr ineffizient sein, da Strukturen häufig vergleichsweise große Datenmengen speichern. Somit ist die Übergabe von ganzen Strukturen eher selten anzutreffen.

Viel häufiger werden Strukturen per Pointer übergeben. Beispielsweise könnte die Statistik der Person aus dem vorherigen Unterkapitel mittels eines Funktionsaufrufs viel einfacher gestaltet werden:

```
void laufStatistik(struct Lauf* lauf) {
  printf("%.03f Kilometer in %d:%02d Minuten = %.1f km/h\n",
    lauf->strecke->km,
    lauf->s / 60,
    lauf->s % 60,
    lauf->strecke->km / lauf->s * 3600);
}
```

Dadurch kann diese Funktion ganz praktisch aufgerufen werden mit:

```
laufStatistik(&dienstag.morgenrunde);
```

Hierbei muss jedoch beachtet werden, dass in der aufgerufenen Funktion keine lokale Kopie, sondern nur der Pointer auf die Struktur der aufrufenden Funktion verfügbar ist. Jede Änderung innerhalb der aufgerufenen Funktion ändert somit die Struktur außerhalb.

Um Änderungen innerhalb der aufgerufenen Funktion zu verhindern, kann der Parameter mittels `const` deklariert werden (siehe Kapitel → 7.5.1). Dadurch verhindert der Compiler jeglichen Schreibzugriff auf eine Komponente der Struktur.

```
void laufStatistik(const struct Lauf* lauf);
```

Die Angabe `const` schützt nur die übergebene Struktur-Variable und deren Komponenten, nicht aber Strukturen, welche als Pointer über eine Komponente dereferenziert werden!

So wäre es beispielsweise möglich, innerhalb der Funktion `laufStatistik()` die Kilometeranzahl einer Strecke zu manipulieren:

```
lauf->strecke->km = 100;
```

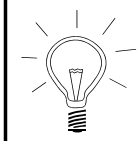
Um dies ebenfalls zu verhindern, müsste die Pointer-Komponente `strecke` der Struktur `Lauf` ebenfalls mit `const` deklariert werden:

```
const struct Strecke* strecke; // Pointer auf die Strecke
```

Diese durchgängige Verwendung des `const`-Schlüsselwortes wird **const-safe-Programmierung** genannt und kann im Detail in Kapitel → 11.5.2 nachgelesen werden.

### 13.1.10 Anwendungsmöglichkeiten von Strukturen

Die Verwendung von Strukturen empfiehlt sich immer bei semantisch zusammengehörenden strukturierten Daten. Einzelkomponenten mit unterschiedlichen Datentypen können als zusammengehörige Struktur wesentlich bequemer gehandhabt werden.



Hier ein einfaches Beispiel für eine Filmdatenbank:

moviedatabase.c

```
#include <stdio.h>

struct Movie {
  char        name[50];
  int         dauer;
  const char* tags[10];
};

void printMovie(const struct Movie* movie) {
  printf("\"%s\", ", movie->name);
  printf("%d Minuten\nTags:\n", movie->dauer);

  int tagindex = 0;
  while (movie->tags[tagindex]) {
    printf("\"%s\"\n", movie->tags[tagindex]);
    ++tagindex;
  }

  printf("\n");
}

int main(void) {
  struct Movie film1 = {
    "Lion King",
    85,
    {"Simba", "Circle of Life"}
  };
  struct Movie film2 = {
    "Terminator 2",
    147,
    {"Bad to the bone", "T-1000", "Hasta la vista"}
  };

  printMovie(&film1);
  printMovie(&film2);
  return 0;
}
```

Die Ausgabe dieses Programmes lautet

```
Lion King", 85 Minuten
Tags:
"Simba"
"Circle of Life"

Terminator 2", 147 Minuten
Tags:
"Bad to the bone"
"T-1000"
"Hasta la vista"
```

Auch viele Systemfunktionen liefern logisch zusammengehörige Daten in Form von Strukturen oder Pointern auf Strukturen zurück. Die benötigten Daten erhält man dann durch Zugriff auf die einzelnen Komponenten der Struktur.

So enthält beispielsweise die im Folgenden gezeigte Struktur `tm` (siehe `<time.h>`) alle wichtigen Daten, die sich aus der Sekundenanzahl berechnen lassen:

```
struct tm {
  int tm_sec;      // Sekunden             [0,59]
  int tm_min;      // Minuten              [0,59]
  int tm_hour;     // Stunden              [0,23]
  int tm_mday;     // Tag                  [1,31]
  int tm_mon;      // Monat                [0,11]
  int tm_year;     // Jahre seit 1900
  int tm_wday;     // Tage seit Sonntag    [0,6]
  int tm_yday;     // Tage seit 1. Januar  [0,365]
  int tm_isdst;    // Daylight Saving Time
};
```

Dieser Datentyp wird von der Funktion `localtime()` genutzt, um die Sekundenanzahl, die von der Funktion `time()` berechnet wird, strukturiert zurückzugeben.

Die Systemzeit eines Computers wird in einem Unix-System als Anzahl der Sekunden seit Mitternacht des 1. Januars 1970 ausgegeben. Da sich aus der Anzahl der Sekunden sowohl die Uhrzeit, das Datum und weitere Zusatzinformationen ableiten lassen, müssten mehrere Funktionen geschrieben werden, die jeweils die gesuchte Information in einem bestimmten Format liefern. Um dies zu vermeiden, nutzt man Strukturen, um dort alle zusammengehörigen Informationen abzuspeichern.

### 13.1.11 Beispielprogramm Systemzeit

Das folgende Beispiel zeigt die Verwendung der Struktur `tm` in Verbindung mit der Funktion `localtime()`. Dieses Beispiel gibt für die Startzeit des Programms die Uhrzeit, das Datum und den Wochentag auf dem Bildschirm aus:

datum.c

```
#include <time.h>
#include <stdio.h>

int main(void) {
  time_t sekunden;
  time(&sekunden);
  struct tm* zeit = localtime(&sekunden);

  printf("Aktuelle Zeitangabe:\n\n");
  printf("%02d:%02d:%02d\n",
    zeit->tm_hour, zeit->tm_min, zeit->tm_sec);
  printf("%02d.%02d.%d \n",
    zeit->tm_mday, zeit->tm_mon + 1, 1900 + (zeit->tm_year));

  char* tag[] = {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                 "Donnerstag", "Freitag", "Samstag"};
  printf("Wochentag: %s\n", tag[zeit->tm_wday]);

  return 0;
}
```

Eine mögliche Ausgabe des Programms ist:

```
Aktuelle Zeitangabe:

21:11:47
30.03.2023
Wochentag: Donnerstag
```

Das Programm `datum.c` ruft mit der Funktion `time()` die aktuelle Systemzeit in Sekunden ab. Dann wird die Sekundenanzahl durch die Funktion `localtime()` in Uhrzeit, Datum und vergangene Tage umgerechnet und in der Struktur `zeit` vom Typ `tm` gespeichert. Die einzelnen Werte der Struktur werden dann als Uhrzeit, Datum und als Wochentag ausgegeben. Da die Struktur-Variable `tm_wday` die Anzahl der Tage seit Sonntag zurückliefert, kann man dies zur Ausgabe eines Wochentages nutzen, indem man `tm_wday` als Index eines `char`-Arrays benutzt.

### 13.1.12 Beispielprogramm Jahr-2038-Problem

Bei der Zeitermittlung mittels Sekunden seit 01.01.1970 ergibt sich im Jahr 2038 ein Problem, das Jahr-2038-Problem: Da `size_t` in der Regel eine 32-Bit Integer-Zahl ist, ergibt sich daraus ein maximaler Wert von 2147483647 für die Anzahl der Sekunden. Dies entspricht einem Datum vom 19.01.2038, wie das folgende Beispiel zeigt:

maxdatum.c

```
#include <limits.h>
#include <time.h>
#include <stdio.h>

int main(void) {
  time_t maxDatum = INT_MAX;
  struct tm* ptrMaxDatum;
  ptrMaxDatum = gmtime(&maxDatum);

  printf("Die 32-Bit-Zeit laeuft ab\n");
  printf("am %02d.%02d.%d\n",
    ptrMaxDatum->tm_mday,
    ptrMaxDatum->tm_mon + 1,
    1900+(ptrMaxDatum->tm_year));
  printf("um %02d:%02d:%02d Uhr\n",
    ptrMaxDatum->tm_hour,
    ptrMaxDatum->tm_min,
    ptrMaxDatum->tm_sec);
  return 0;
}
```

Als Sekundenanzahl wird hier einfach der größtmögliche Wert einer `int`-Zahl verwendet. Dann wird mit diesem Wert das Datum und die Uhrzeit berechnet.

Hier die Ausgabe des Programms:

```
Die 32-Bit-Zeit laeuft ab
am 19.01.2038
um 03:14:07 Uhr
```

Bis zum Jahr 2038 dauert es zwar noch ein paar Jahre, aber die Probleme, die dann kommen werden, dürften denen des Jahr-2000-Problemes nur um wenig nachstehen.

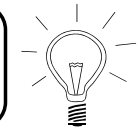
## 13.2 Anonyme Strukturen

Normalerweise werden Felder von Strukturen genauso wie Variablen stets mit einem eindeutigen Namen versehen. Dadurch sind diese Komponenten klar ansprechbar. Nicht immer jedoch will man Variablen oder Felder explizit ansprechen, beispielsweise wenn Variablen nur einen einmaligen Übergabewert für eine Funktion darstellen oder die Felder eines Struktur-Typs direkt einer übergeordneten Struktur angehören sollen.

Im Standard C11 wurde genau aus diesem Grund festgelegt, dass Struktur-Variablen auch anonym – ohne Namen oder Etikett – vereinbart werden können.

### 13.2.1 Anonyme Struktur-Typen

**Anonyme Strukturen** können als Teil einer Definition von anderen Strukturen auftreten.



Im folgenden Beispiel hat die Struktur innerhalb der Struktur keinen Namen:

```
struct MeineStruktur {
  int komponente1;
  struct {
    int komponente2;
  };
};
```

Da die Struktur keinen Variablennamen besitzt, lassen sich die Komponenten innerhalb der namenlosen Struktur ansprechen, als wären die Komponenten direkt Teil der übergeordneten benannten Struktur:

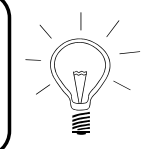
```
struct MeineStruktur strukturVariable;
strukturVariable.komponente1;
strukturVariable.komponente2;
```

Dabei ist zu beachten, dass die Namen der einzelnen Komponenten in der Struktur für eine eindeutige Zuordnung verschieden sein müssen. Der Aufruf einer Komponente innerhalb einer eingebetteten Struktur wird damit einfacher und kürzer, der Programmcode wird übersichtlicher.

### 13.2.2 Compound Literals

In C11 kann eine Struktur auch mittels eines sogenannten „compound literals“ direkt, also ohne Vereinbarung einer Variablen angegeben werden.

> Ein **compound literal** ist ein Ausdruck, der ein anonymes Objekt (Objekt ohne Namen) erzeugt, dessen Wert durch eine Initialisierungsliste gegeben ist.



Im Folgenden ein erstes Beispiel für ein compound literal:

compoundliteral.c

```
#include <stdio.h>

void printArray(int array[4]) {
  for (int i = 0; i < 4; ++i) {
    printf("%d ", array[i]);
  }
}

int main(void) {
  printArray((int[4]) {1, 2, 3, 4});
  return 0;
}
```

Hier wird ein compound literal an die Funktion `printArray()` übergeben. Der Typ des Objektes ist `int[4]` und steht in runden Klammern vor der Initialisierungsliste. Der Typname kann ein vollständiger Datentyp oder ein Array von unbekannter Länge sei, nicht aber ein Array-Typ variabler Länge.

Beispiele:

```
(int){1}
(const int){2}
(struct Point2D){0, 0}
```

Compound literals erlauben eine Notation ähnlich wie literale Konstanten für Arrays, Strukturen und andere Typen (außer den Arrays variabler Länge). Ein compound literal kann an jeder Stelle benutzt werden, an der ein Objekt desselben Typs wie das compound literal benutzt werden kann. Beispielsweise:

```
int x = (int){1} + (int){4710};
```

Ein compound literal bezeichnet dabei stets einen L-Wert. Die Adresse eines compound literals ist die Adresse des anonymen Objekts, das durch das compound literal erklärt wird. Wenn das compound literal keinen mit `const` qualifizierten Typ hat, kann das compound literal über einen Pointer auf das compound literal abgeändert werden.

Beispiele:

```
int* p = (int[3]){1, 2, 3};
*(p + 1) = 40;   // setzt die Komponente mit Index 1 auf 40
```

```
struct Point2D* p = &(struct Point2D){0, 0};
p->x = 1;
p->y = 1;
// der Punkt hat jetzt die Koordinaten (1, 1)
```

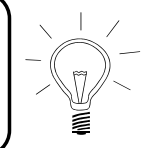
Wird ein anonymes Objekt außerhalb einer Funktion angelegt, ist die Lebensdauer statisch während des Programmlaufs. Da die Initialisierung stattfindet, bevor ein Programm läuft, müssen die Initialisierer in der Initialisierungsliste konstante Ausdrücke sein. Innerhalb einer Funktion erfolgt die Initialisierung, wenn der entsprechende Block im Programmablauf erreicht wird. Die Ausdrücke in der Initialisierungsliste können in diesem Fall beliebige zur Laufzeit zu berechnende Ausdrücke sein.

Alle Einschränkungen für die „normale“ Initialisierungsliste gelten auch für compound literals. Wenn man in der Initialisierungsliste für Struktur-Typen und Arrays mit fester Elementanzahl somit nur einige Initialisierer liefert, werden die ersten Elemente initialisiert und die anderen mit null des entsprechenden Typs. Wie bei jeder anderen Initialisierungsliste mit geschweiften Klammern können „designated inititializers“ verwendet werden, siehe Kapitel → 13.1.3. Bei einem anonymen Array ohne Längenangabe bestimmt sich die Länge des Arrays aus der Zahl der Elemente des Arrays.

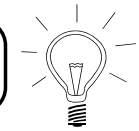
## 13.3 Unionen 

Eine **Union** besteht wie eine Struktur aus einer Reihe von Komponenten mit unterschiedlichen Datentypen. Im Gegensatz zur Struktur jedoch stellen die einzelnen Komponenten nicht Teile einer Gesamtstruktur dar, sondern nur Alternativen, welche unter dem Strukturnamen angesprochen werden können. Die Komponenten einer Union werden also nicht hintereinander im Speicher abgebildet, sondern beginnen alle an derselben Adresse. Der Speicherplatz wird vom Compiler so groß angelegt, dass der Speicherplatz auch für die größte Alternative reicht.

Bei einer Union ist zu einem bestimmten Zeitpunkt jeweils nur eine einzige Komponente einer Reihe von alternativen Komponenten gespeichert.



Eine Union wird mit dem Schlüsselwort **`union`** eingeleitet.



Als Beispiel wird hier eine Union vom Typ `union vario` eingeführt:

```
union vario {
  int     intVar;
  double  doubleVar;
  float   floatVar;
} variant;
```

Hierbei ist `vario` das **Etikett** (auf Englisch das „**union tag**“) des Union-Typs. Ein Wert aus dem Wertebereich eines jeden der drei in der Union enthaltenen Datentypen kann an die Variable `variant` zugewiesen und in Ausdrücken benutzt werden. Man muss jedoch aufpassen, dass die Benutzung konsistent bleibt.

Beim Schreiben von Code mit Unionen muss verfolgt werden, welcher Typ jeweils in der Union gespeichert ist. Der Datentyp, der entnommen wird, muss derjenige sein, der zuletzt gespeichert wurde.

Die Auswahl von Komponenten erfolgt wie bei Strukturen über den Punkt- beziehungsweise den Pfeil-Operator:

```
variant.intVar = 123;
int x = variant.intVar;
```

oder

```
union vario* ptr = &variant;
double y = ptr->doubleVar;
```

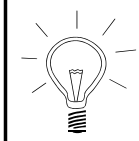
Da weder beim Compilieren noch zur Laufzeit die Zugriffe auf die jeweiligen Komponenten überprüft werden, gelten Unionen heutzutage als verpönt. Unionen werden außer in der Systemprogrammierung kaum mehr verwendet.

### 13.3.1 Detaillierte Informationen zu Unionen

Obschon Unionen heute kaum mehr genutzt werden, werden hier dennoch einige fortgeschrittene Informationen und ein Beispiel gezeigt.

- Unionen können in Strukturen und Arrays auftreten und umgekehrt.
- Unions-Variablen können mittels des Zuweisungs-Operators gegenseitig sich zugewiesen werden.
- Die Ermittlung der Größe und Speicheranordnung einer Union erfolgt mit Hilfe des `sizeof`- und (seit C11) des `_Alignof`-Operators.
- Es ist möglich, die Adresse einer Unions-Variablen mittels des Adress-Operators zu ermitteln.

Wenn ein Pointer auf eine Union in den Typ eines Pointers auf eine Alternative umgewandelt wird, so verweist das Resultat auf diese Alternative. Voraussetzung ist natürlich, dass diese Alternative die gerade aktuell gespeicherte Alternative darstellt.



Dies ist unter anderem im folgenden Beispiel zu sehen:

union.c

```
#include <stdio.h>

int main(void) {
  union Zahl {
    int    intVar;
    double doubleVar;
    float  floatVar;
  };

  union Zahl feld[2];
  union Zahl* ptr;
  float* floatPtr;

  printf("Groesse der Union: %d\n", (int)sizeof(union Zahl));
  printf("Groesse der Komponenten: %d\n", (int)sizeof(feld[1]));
  printf("Groesse von int   : %d\n", (int)sizeof(int));
  printf("Groesse von double: %d\n", (int)sizeof(double));
  printf("Groesse von float : %d\n", (int)sizeof(float));

  printf("\n");

  feld[0].doubleVar = 5.;
  printf("Inhalt von feld[0]: %f\n", feld[0].doubleVar);
  feld[0].intVar = 10;
  printf("Inhalt von feld[0]: %d\n", feld[0].intVar);
  feld[0].floatVar = 100.0;
  printf("Inhalt von feld[0]: %6.2f\n", feld[0].floatVar);
  printf("-------------------------------------\n");

  feld[1] = feld[0];
  printf("Inhalt von feld[1]: %6.2f\n", feld[1].floatVar);
  feld[1].floatVar += 25.;
  ptr = &feld[1];
  floatPtr = (float*)ptr;

  printf("Inhalt von feld[1]: %6.2f\n", ptr ->floatVar);
  printf("Inhalt von feld[1]: %6.2f\n", *floatPtr);

  return 0;
}
```

Hier die Ausgabe des Programms:

```
Groesse der Union: 8
Groesse der Komponenten: 8
Groesse von int   : 4
Groesse von double: 8
Groesse von float : 4

Inhalt von feld[0]: 5.000000
Inhalt von feld[0]: 10
Inhalt von feld[0]: 100.00
-------------------------------------
Inhalt von feld[1]: 100.00
Inhalt von feld[1]: 125.00
Inhalt von feld[1]: 125.00
```

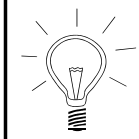
Bei einer Union kann nur eine Initialisierung der ersten Alternative erfolgen. Diese Initialisierung erfolgt durch einen in geschweiften Klammern stehenden konstanten Ausdruck.


Eine Variable vom Typ `union Zahl` aus obigem Beispiel kann also nur mit einem konstanten `int`-Ausdruck initialisiert werden.

## 13.4 Bitfelder 

Bis zu dieser Stelle wurden lediglich die Bit-Operationen | (bitweises ODER), & (bitweises UND), ^ (Exklusives-ODER) und ~ (Einer-Komplement) als Möglichkeit zur Bit-Manipulation in der Programmiersprache C vorgestellt (siehe dazu Kapitel → 9.8). Hierbei kann man durch gezieltes Verknüpfen der Operanden mit einem entsprechenden Bitmuster Bits setzen oder löschen. Eine weitere Möglichkeit, um mit Bits zu arbeiten, stellen die Bitfelder dar. Bitfelder ermöglichen es, Bits zu gruppieren.

Ein **Bitfeld** besteht aus einer angegebenen Zahl von Bits (einschließlich eines eventuellen Vorzeichenbits) und wird als Integer-Typ betrachtet. Die Länge des Bitfeldes wird vom Bitfeld-Namen durch einen Doppelpunkt getrennt.



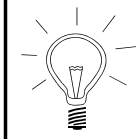
```
struct {
  ...
  BitfeldTyp BitfeldName : BitAnzahl;
  ...
};
```

Dabei ist zu beachten, dass ein Bitfeld nur als Teil einer Struktur oder einer Union existieren kann.

Beispielsweise deklariert folgende Zeile ein Bitfeld a bestehend aus 4 Bits, welches als `unsigned int` interpretiert wird:

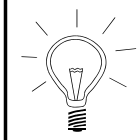
```
unsigned a : 4;
```

Bitfelder sind von der jeweiligen Implementierung des Compilers abhängig. So sind beispielsweise die zulässigen Typen für ein Bitfeld und die Anordnung der verschiedenen Bitfelder im Speicher je nach Compiler unterschiedlich.

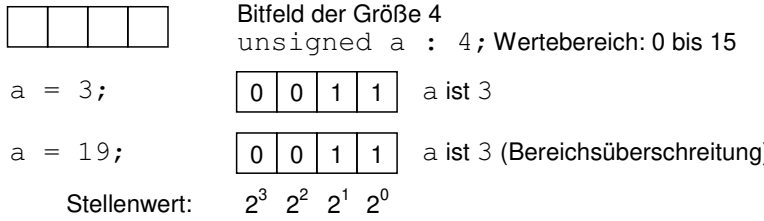


Ein Bitfeld kann wie eine normale Variable mit dem definierten Typ gelesen und gesetzt werden, allerdings wird nur die angegebene Anzahl Bits wirklich gespeichert.

> Ein Bitfeld wird wie eine normale Komponente einer Struktur mit dem Punkt-Operator . angesprochen. Der Pfeil-Operator -> ist je nach Compilerhersteller zugelassen oder auch nicht, da ein Bitfeld nicht immer eine Adresse hat.



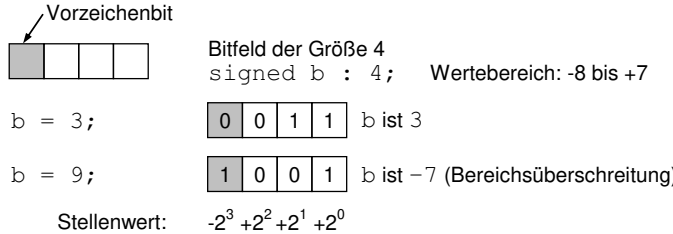
Werden einem Bitfeld Werte zugewiesen, die außerhalb des Wertebereichs des Bitfeldes liegen, so wird mit der Modulo-Arithmetik (siehe Kapitel → 7.3.2) ein Überlauf vermieden. Weist man somit der Variablen a den Wert 19 zu, so werden nur die untersten 4 Bits verwendet, was den gespeicherten Wert entsprechend verändert:




Bei vorzeichenbehafteten Typen kann dies zu weiteren Problemen führen. Beim Typ `signed` wird normalerweise das Most Significant Bit (MSB) für die Darstellung des Vorzeichenbits in Zweierkomplement-Form benutzt. Ist das MSB gleich 1, so wird die Zahl als negative Zahl interpretiert.

```
signed b : 4;
```

Wenn wie im folgenden Beispiel der Variablen b der Wert 9 zugewiesen wird, so tritt eine Bereichsüberschreitung auf:



Das Bitfeld `b` hat eigentlich einen Zahlenbereich von `-8` bis `+7`. Bei einer Zuweisung einer Zahl außerhalb des Zahlenbereichs werden die Bits den Stellen entsprechend hart zugewiesen, ohne dass vom Compiler auf einen Überlauf hingewiesen wird. Entsprechend der Darstellung im Zweierkomplement wird bei `7+1` das höchste Bit gesetzt, die anderen Bits sind 0. Daraus ergibt sich der Wert `-8`. Aus der Zahl `9` wird dann in dem Bitfeld entsprechend eine `-7`, `10` entspricht `-6`, und so fort.

Die Datentypen, die in einem Bitfeld verwendet werden dürfen, sind eingeschränkt. Nach dem Standard dürfen lediglich die Typen `int`, `signed int` oder `unsigned int` und unter C11 auch der Typ `_Bool` verwendet werden. Bei manchen Compilern wie beispielsweise beim Visual C++ Compiler sind auch die Typen `char`, `short` und `long` jeweils `signed` und `unsigned` erlaubt. Letztendlich ist der Datentyp eines Bitfeldes für die Interpretation der einzelnen Bits ausschlaggebend. Hierbei spielt auch eine entscheidende Rolle, ob das Bitfeld `signed` oder `unsigned` ist.

Bitfelder sind sehr implementierungsabhängig, was durch den ISO-Standard gewünscht wurde. Dies führt dazu, dass bei der Portierung von Programmen, die Bitfelder enthalten, große Vorsicht angeraten ist.


Aufgrund der enormen Fortschritte betreffend verfügbarem Speicherplatz werden Bitfelder heutzutage kaum mehr eingesetzt. Sie werden jedoch manchmal in der hardwarenahen Programmierung verwendet, wo Hardware-Bausteine über Bitmasken programmiert werden können.

## 13.5 Übungsaufgaben

### Aufgabe 1: Strings

Studieren Sie das folgende Programm.

a) Welche Codezeilen werden einen Fehler erzeugen? Überprüfen Sie Ihre Annahmen mit Hilfe des Compilers.

b) Korrigieren Sie die Fehler, sodass das Programm lauffähig wird.

c) Was wird nun ausgegeben werden? Überprüfen Sie Ihre Annahmen mit einem Programmdurchlauf.

structErrors.c

```
#include <stdio.h>

struct Person {
  char vorname[30];
  char* nachname;
};

int main(void) {
  struct Person person1 = {"Kathrin", "Knoll"};
  struct Person person2 = {"Thorsten", "Powlov"};

  person1.vorname = "Maria";
  person1.nachname = "Hanse";
  printf("%s %s\n", person1.vorname, person1.nachname);

  person1.vorname = person2.vorname;
  person1.nachname = person2.nachname;
  printf("%s %s\n", person1.vorname, person1.nachname);

  person1 = person2;
  printf("%s %s\n", person1.vorname, person1.nachname);

  person2.nachname = "Frick";
  printf("%s %s\n", person2.vorname, person2.nachname);

  return 0;
}
```

**Aufgabe 2: Compound literal**

1. Verändern und erweitern Sie das Programm aus Aufgabe 1 so, dass die Ausgabe des vollständigen Namens in einer Funktion mit folgendem Funktionskopf passiert:

```
void printPerson(struct Person myPerson);
```

2. Geben Sie mittels dieser Funktion mehrere neue Namen bestehend aus Vorname und Nachname aus, jedoch ohne Variablen zu verwenden. Nutzen Sie hierfür compound literals.

3. Verändern Sie die Funktion und die Aufrufe derselben so, dass ein Pointer als Argument erwartet wird:

```
void printPersonPointer(struct Person* myPerson);
```

**Aufgabe 3: Strukturen. Pointer auf Strukturen.**

a) Schreiben Sie ein Programm, welches eine Münzsammlung speichern kann. Eine Münze soll Eigenschaften speichern wie Währung, Münzwert, Jahrgang, Tauschwert, Größe, Kaufdatum.

b) Erweitern Sie das Programm mit einer Struktur „Münzkategorie“, welche die Währung und den Münzwert speichert. Münzen sollen sodann nur noch per Pointer auf diese Münzkategorien verweisen.

c) Schreiben Sie eine Funktion, welche eine Münze als Pointer erwartet und deren Eigenschaften schön formatiert ausgibt.