

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 14. Januar 2019

Aufgabe 1: Iterationsfunktionen

Wir betrachten das folgende Programm ([aufgabe-1.c](#)):

```
#include <stdio.h>

void foreach (int *a, void (*fun) (int x))
{
    for (int *p = a; *p >= 0; p++)
        fun (*p);
}

void even_or_odd (int x)
{
    if (x % 2)
        printf ("%d_ist_ungerade.\n", x);
    else
        printf ("%d_ist_gerade.\n", x);
}

int main (void)
{
    int numbers[] = { 12, 17, 32, 1, 3, 16, 19, 18, -1 };
    foreach (numbers, even_or_odd);
    return 0;
}
```

(a) Was bedeutet **void (*fun) (int x)**, und welchen Sinn hat seine Verwendung in der Funktion **foreach()**? (2 Punkte)

(b) Schreiben Sie das Hauptprogramm **main()** so um, daß es unter Verwendung der Funktion **foreach()** die Summe aller positiven Zahlen in dem Array berechnet. Sie dürfen dabei weitere Funktionen sowie globale Variable einführen. (4 Punkte)

Lösung

- (a) Was bedeutet **void (*fun) (int x)**, und welchen Sinn hat seine Verwendung in der Funktion **foreach()**?

void (*fun) (int x) deklariert einen Zeiger **fun**, der auf Funktionen zeigen kann, die einen Parameter **x** vom Typ **int** erwarten und keinen Wert zurückgeben (**void**).

Durch die Übergabe eines derartigen Parameters an die Funktion **foreach()** lassen wir dem Aufrufer die Wahl, welche Aktion für alle Elemente des Arrays aufgerufen werden soll.

- (b) Schreiben Sie das Hauptprogramm **main()** so um, daß es unter Verwendung der Funktion **foreach()** die Summe aller positiven Zahlen in dem Array berechnet. Sie dürfen dabei weitere Funktionen sowie globale Variable einführen.

Siehe: [loesung-1.c](#)

Damit die Funktion **add_up()** Zugriff auf die Variable **sum** hat, muß diese global sein und vor der Funktion **add_up()** deklariert werden.

Die Bedingung, daß nur positive Zahlen summiert werden sollen, ist durch die Arbeitsweise der Funktion **foreach()** bereits gewährleistet, da negative Zahlen als Ende-Markierungen dienen.

Wichtig ist, daß die Variable **sum** vor dem Aufruf der Funktion **foreach()** auf den Wert **0** gesetzt wird. In [loesung-1.c](#) geschieht dies durch die Initialisierung von **sum**. Wenn mehrere Summen berechnet werden sollen, muß dies durch explizite Zuweisungen **sum = 0** vor den Aufrufen von **foreach()** erfolgen.

Aufgabe 2: Objektorientierte Tier-Datenbank

Das auf der nächsten Seite in Blau dargestellte Programm (Datei: [aufgabe-2a.c](#)) soll Daten von Tieren verwalten.

Beim Compilieren erscheinen die folgende Fehlermeldungen:

```
$ gcc -std=c99 -Wall -O aufgabe-2a.c -o aufgabe-2a
aufgabe-2a.c: In function 'main':
aufgabe-2a.c:31: error: 'animal' has no member named 'wings'
aufgabe-2a.c:37: error: 'animal' has no member named 'legs'
```

Der Programmierer nimmt die auf der nächsten Seite in Rot dargestellten Ersetzungen vor (Datei: [aufgabe-2b.c](#)). Daraufhin gelingt das Compilieren, und die Ausgabe des Programms lautet:

```
$ gcc -std=c99 -Wall -O aufgabe-2b.c -o aufgabe-2b
$ ./aufgabe-2b
A duck has 2 legs.
Error in animal: cow
```

- (a) Erklären Sie die o. a. Compiler-Fehlermeldungen. (2 Punkte)
- (b) Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen? (3 Punkte)
- (c) Erklären Sie die Ausgabe des Programms. (5 Punkte)
- (d) Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt. (4 Punkte)
- (e) Schreiben Sie das Programm so um, daß es keine expliziten Typumwandlungen mehr benötigt. Hinweis: Verwenden Sie **union**. (4 Punkte)
- (f) Schreiben Sie das Programm weiter um, so daß es die Objektinstanzen **duck** und **cow** dynamisch erzeugt. Hinweis: Verwenden Sie **malloc()** und schreiben Sie Konstruktoren. (4 Punkte)
- (g) Schreiben Sie das Programm weiter um, so daß die Ausgabe nicht mehr direkt im Hauptprogramm erfolgt, sondern stattdessen eine virtuelle Methode **print()** aufgerufen wird. Hinweis: Verwenden Sie in den Objekten Zeiger auf Funktionen, und initialisieren Sie diese in den Konstruktoren. (4 Punkte)

```
#include <stdio.h>

#define ANIMAL 0
#define WITH_WINGS 1
#define WITH_LEGS 2

typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int wings;
} with_wings;

typedef struct with_legs
{
    int legs;
} with_legs;

int main (void)
{
    animal *a[2];

    animal duck;
    a[0] = &duck;
    a[0]—>type = WITH_WINGS;
    a[0]—>name = "duck";
    a[0]—>wings = 2;  ← ((with_wings *) a[0])—>wings = 2;

    animal cow;
    a[1] = &cow;
    a[1]—>type = WITH_LEGS;
    a[1]—>name = "cow";
    a[1]—>legs = 4;  ← ((with_legs *) a[1])—>legs = 4;

    for (int i = 0; i < 2; i++)
        if (a[i]—>type == WITH_LEGS)
            printf ("A_%s_has_%d_legs.\n", a[i]—>name,
                    ((with_legs *) a[i])—>legs);
        else if (a[i]—>type == WITH_WINGS)
            printf ("A_%s_has_%d_wings.\n", a[i]—>name,
                    ((with_wings *) a[i])—>wings);
        else
            printf ("Error_in_animal:_%s\n", a[i]—>name);

    return 0;
}
```

Lösung

(a) **Erklären Sie die o. a. Compiler-Fehlermeldungen.**

`a[0]` und `a[1]` sind gemäß der Deklaration `animal *a[2]` Zeiger auf Variablen vom Typ `animal` (ein `struct`). Wenn man diesen Zeiger dereferenziert (`->`), erhält man eine `animal`-Variable. Diese enthält keine Datenfelder `wings` bzw. `legs`.

(b) **Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen?**

Durch die *explizite Typumwandlung des Zeigers* erhalten wir einen Zeiger auf eine `with_wings`- bzw. auf eine `with_legs`-Variable. Diese enthalten die Datenfelder `wings` bzw. `legs`.

(c) **Erklären Sie die Ausgabe des Programms.**

Durch die explizite Typumwandlung des Zeigers zeigt `a[0]` auf eine `with_wings`-Variable. Diese enthält nur ein einziges Datenfeld `wings`, das an genau derselben Stelle im Speicher liegt wie `a[0]->type`, also das Datenfeld `type` der `animal`-Variable, auf die der Zeiger `a[0]` zeigt. Durch die Zuweisung der Zahl 2 an `((with_wings *) a[0])->wings` überschreiben wir also `a[0]->type`, so daß das `if` in der `for`-Schleife `a[0]` als `WITH_LEGS` erkennt.

Bei der Ausgabe `A duck has 2 legs.` wird das Datenfeld `((with_legs *) a[0])->legs` als Zahl ausgegeben. Dieses Datenfeld befindet sich in denselben Speicherzellen wie `a[0]->type` und `((with_wings *) a[0])->wings` und hat daher ebenfalls den Wert 2.

Auf die gleiche Weise überschreiben wir durch die Zuweisung der Zahl 4 an `((with_legs *) a[1])->legs` das Datenfeld `a[0]->type`, so daß das `if` in der `for`-Schleife `a[1]` als unbekanntes Tier (Nr. 4) erkennt und `Error in animal: cow` ausgibt.

(d) **Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt.**

Damit die *Vererbung* zwischen den Objekten `animal`, `with_wings` und `with_legs` funktioniert, müssen die abgeleiteten Klassen `with_wings` und `with_legs` alle Datenfelder der Basisklasse `animal` erben. In C geschieht dies explizit; die Datenfelder müssen in den abgeleiteten Klassen neu angegeben werden (siehe `loesung-2d-1.c`):

```
typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int type;
    char *name;
    int wings;
} with_wings;

typedef struct with_legs
{
    int type;
    char *name;
    int legs;
} with_legs;
```

Zusätzlich ist es notwendig, die Instanzen `duck` und `cow` der abgeleiteten Klassen `with_wings` und `with_legs` auch als solche zu deklarieren, damit für sie genügend Speicher reserviert wird:

```
animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
```

```

a[0]—>type = WITH_WINGS;
a[0]—>name = "duck";
((with_wings *) a[0])—>wings = 2;

with_legs cow;
a[1] = (animal *) &cow;
a[1]—>type = WITH_LEGS;
a[1]—>name = "cow";
((with_legs *) a[1])—>legs = 4;

```

Wenn man dies vergißt und sie nur als `animal` deklariert, wird auch nur Speicherplatz für (kleinere) `animal`-Variable angelegt. Dadurch kommt es zu Speicherzugriffen außerhalb der deklarierten Variablen, was letztlich zu einem Absturz führt (siehe [loesung-2d-0f.c](#)).

Für die Zuweisung eines Zeigers auf `duck` an `a[0]`, also an einen Zeiger auf `animal` wird eine weitere explizite Typumwandlung notwendig. Entsprechendes gilt für die Zuweisung eines Zeigers auf `cow` an `a[1]`.

Es ist sinnvoll, explizite Typumwandlungen so weit wie möglich zu vermeiden. Es ist einfacher und gleichzeitig sicherer, direkt in die Variablen `duck` und `cow` zu schreiben, anstatt dies über die Zeiger `a[0]` und `a[1]` zu tun (siehe [loesung-2d-2.c](#)):

```

animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
duck.type = WITH_WINGS;
duck.name = "duck";
duck.wings = 2;

with_legs cow;
a[1] = (animal *) &cow;
cow.type = WITH_LEGS;
cow.name = "cow";
cow.legs = 4;

```

- (e) **Schreiben Sie das Programm so um, daß es keine expliziten Typumwandlungen mehr benötigt.**
Hinweis: Verwenden Sie `union`.

Siehe [loesung-2e.c](#).

Diese Lösung basiert auf [loesung-2d-2.c](#), da diese bereits weniger explizite Typumwandlungen enthält als [loesung-2d-1.c](#).

Arbeitsschritte:

- Umbenennen des Basistyps `animal` in `base`, damit wir den Bezeichner `animal` für die `union` verwenden können
- Schreiben einer `union animal`, die die drei Klassen `base`, `with_wings` und `with_legs` als Datenfelder enthält
- Umschreiben der Initialisierungen: Zugriff auf Datenfelder erfolgt nun durch z. B. `a[0]—>b.name`. Hierbei ist `b` der Name des `base`-Datenfelds innerhalb der `union animal`.
- Auf gleiche Weise schreiben wir die `if`-Bedingungen innerhalb der `for`-Schleife sowie die Parameter der `printf()`-Aufrufe um.

Explizite Typumwandlungen sind nun nicht mehr nötig.

Nachteil dieser Lösung: Jede Objekt-Variable belegt nun Speicherplatz für die gesamte `union animal`, anstatt nur für die benötigte Variable vom Typ `with_wings` oder `with_legs`. Dies kann zu einer Verschwendung von Speicherplatz führen, auch wenn dies in diesem Beispielpogramm tatsächlich nicht der Fall ist.

- (f) **Schreiben Sie das Programm weiter um, so daß es die Objektinstanzen `duck` und `cow` dynamisch erzeugt.**

Hinweis: Verwenden Sie `malloc()` und schreiben Sie Konstruktoren.

Siehe [loesung-2f.c](#).

- (g) Schreiben Sie das Programm weiter um, so daß die Ausgabe nicht mehr direkt im Hauptprogramm erfolgt, sondern stattdessen eine virtuelle Methode `print()` aufgerufen wird.

Hinweis: Verwenden Sie in den Objekten Zeiger auf Funktionen, und initialisieren Sie diese in den Konstruktoren.

Siehe [loesung-2g.c](#).