

# Hardwarenahe Programmierung

## Musterlösung zu den Übungsaufgaben – 5. Dezember 2022

### Aufgabe 1: Objektorientierte Tier-Datenbank (Ergänzung)

Diese Aufgabe ist eine Neuauflage der Übungsaufgabe 3 vom 28. November 2022, ergänzt um die Teilaufgaben (e) bis (g).

Das auf der nächsten Seite dargestellte Programm (Datei: [aufgabe-1a.c](#)) soll Daten von Tieren verwalten.

Beim Compilieren erscheinen die folgende Fehlermeldungen:

```
$ gcc -std=c99 -Wall -O aufgabe-1a.c -o aufgabe-1a
aufgabe-1a.c: In function 'main':
aufgabe-1a.c:31: error: 'animal' has no member named 'wings'
aufgabe-1a.c:37: error: 'animal' has no member named 'legs'
```

Der Programmierer nimmt die auf der nächsten Seite in Rot dargestellten Ersetzungen vor (Datei: [aufgabe-1b.c](#)). Daraufhin gelingt das Compilieren, und die Ausgabe des Programms lautet:

```
$ gcc -std=c99 -Wall -O aufgabe-1b.c -o aufgabe-1b
$ ./aufgabe-1b
A duck has 2 legs.
Error in animal: cow
```

- (a) Erklären Sie die o. a. Compiler-Fehlermeldungen.
- (b) Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen?
- (c) Erklären Sie die Ausgabe des Programms.
- (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.
- (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

- (e) Schreiben Sie das Programm so um, daß es keine expliziten Typumwandlungen mehr benötigt.

**Hinweis:** Verwenden Sie **union**.

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

Diese Lösung basiert auf [loesung-3d-2.c](#), da diese bereits weniger explizite Typumwandlungen enthält als [loesung-3d-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 Beispielprogramm 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-1f.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-1g.c](#).

## Aufgabe 2: Iterationsfunktionen

Wir betrachten das folgende Programm ([aufgabe-2.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)
- (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-2.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-2.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 3: Stack-Operationen

Das folgende Programm ([aufgabe-3.c](#)) implementiert einen Stapelspeicher (Stack). Dies ist ein Array, das nur bis zu einer variablen Obergrenze (Stack-Pointer) tatsächlich genutzt wird. An dieser Obergrenze kann man Elemente hinzufügen (push).

In dieser Aufgabe sollen zusätzlich Elemente in der Mitte eingefügt werden (insert). Die dafür bereits existierenden Funktionen `insert()` und `insert_sorted()` sind jedoch fehlerhaft.

```

#include <stdio.h>

#define STACK_SIZE 10

int stack[STACK_SIZE];
int stack_pointer = 0;

void push (int x)
{
    stack[stack_pointer++] = x;
}

void show (void)
{
    printf ("stack_content:");
    for (int i = 0; i < stack_pointer; i++)
        printf ("_%d", stack[i]);
    if (stack_pointer)
        printf ("\n");
    else
        printf ("_(empty)\n");
}

void insert (int x, int pos)
{
    for (int i = pos; i < stack_pointer; i++)
        stack[i + 1] = stack[i];
    stack[pos] = x;
    stack_pointer++;
}

void insert_sorted (int x)
{
    int i = 0;
    while (i < stack_pointer && x < stack[i])
        i++;
    insert (x, i);
}

int main (void)
{
    push (3);
    push (7);
    push (137);
    show ();
    insert (5, 1);
    show ();
    insert_sorted (42);
    show ();
    insert_sorted (2);
    show ();
    return 0;
}

```

- Korrigieren Sie das Programm so, daß die Funktion `insert()` ihren Parameter `x` an der Stelle `pos` in den Stack einfügt und den sonstigen Inhalt des Stacks verschiebt, aber nicht zerstört. (3 Punkte)
- Korrigieren Sie das Programm so, daß die Funktion `insert_sorted()` ihren Parameter `x` an derjenigen Stelle einfügt, an die er von der Sortierung her gehört. (Der Stack wird hierbei vor dem Funktionsaufruf als sortiert vorausgesetzt.) (2 Punkte)
- Schreiben Sie eine zusätzliche Funktion `int search (int x)`, die die Position (Index) des Elements `x` innerhalb des Stack zurückgibt – oder die Zahl `-1`, wenn `x` nicht im Stack enthalten ist. Der Rechenaufwand darf höchstens  $\mathcal{O}(n)$  betragen. (3 Punkte)
- Wie (c), aber der Rechenaufwand darf höchstens  $\mathcal{O}(\log n)$  betragen. (4 Punkte)

## Lösung

- Korrigieren Sie das Programm so, daß die Funktion `insert()` ihren Parameter `x` an der Stelle `pos` in den Stack einfügt, und den sonstigen Inhalt des Stacks verschiebt, aber nicht zerstört.**  
Die `for`-Schleife in der Funktion `insert()` durchläuft das Array von unten nach oben. Um den Inhalt des Arrays von unten nach oben zu verschieben, muß man die Schleife jedoch von oben nach unten durchlaufen.

Um die Funktion zu reparieren, ersetze man also

```
for (int i = pos; i < stack_pointer; i++)
```

durch:

```
for (int i = stack_pointer - 1; i >= pos; i--)
```

(Siehe auch: [loesung-3.c](#))

- (b) Korrigieren Sie das Programm so, daß die Funktion `insert_sorted()` ihren Parameter `x` an derjenigen Stelle einfügt, an die er von der Sortierung her gehört. (Der Stack wird hierbei vor dem Funktionsaufruf als sortiert vorausgesetzt.)

Der Vergleich `x < stack[j]` als Bestandteil der `while`-Bedingung paßt nicht zur Durchlaufrichtung der Schleife (von unten nach oben).

Um die Funktion zu reparieren, kann man daher entweder das Kleinerzeichen durch ein Größerzeichen ersetzen (`x > stack[j]`) – siehe [loesung-3b-1.c](#)) oder die Schleife von oben nach unten durchlaufen (siehe [loesung-3b-2.c](#)).

Eine weitere Möglichkeit besteht darin, das Suchen nach der Einfügeposition mit dem Verschieben des Arrays zu kombinieren (siehe [loesung-3.c](#)). Hierdurch spart man sich eine Schleife; das Programm wird schneller. (Es bleibt allerdings bei  $\mathcal{O}(n)$ .)

- (c) Schreiben Sie eine zusätzliche Funktion `int search(int x)`, die die Position (Index) des Elements `x` innerhalb des Stack zurückgibt – oder `-1`, wenn `x` nicht im Stack enthalten ist. Der Rechenaufwand darf höchstens  $\mathcal{O}(n)$  betragen.

Man geht in einer Schleife den Stack (= den genutzten Teil des Arrays) durch. Bei Gleichheit gibt man direkt mit `return` den Index zurück. Nach dem Schleifendurchlauf steht fest, daß `x` nicht im Stack vorhanden ist; man kann dann direkt `-1` zurückgeben (siehe [loesung-3c.c](#)).

Da es sich um eine einzelne Schleife handelt, ist die Ordnung  $\mathcal{O}(n)$ .

- (d) Wie (c), aber der Rechenaufwand darf höchstens  $\mathcal{O}(\log n)$  betragen.

Um  $\mathcal{O}(\log n)$  zu erreichen, halbiert man fortwährend das Intervall von (einschließlich) `0` bis (ausschließlich) `stack_pointer` (siehe [loesung-3d.c](#)).

Wichtig ist, daß man nach dem Durchlauf der Schleife noch auf die Gleichheit `x == stack[left]` (insbesondere nicht: `stack[right]`) prüfen und ggf. `left` bzw. `-1` zurückgeben muß.