

# Hardwarenahe Programmierung

## Musterlösung zu den Übungsaufgaben – 12. Oktober 2023

### Aufgabe 1: Schaltjahr ermitteln

Schreiben Sie ein C-Programm, das eine Jahreszahl erfragt und ausgibt, ob es sich um ein Schaltjahr handelt.

- Wenn die Jahreszahl durch 4 teilbar ist, ist das Jahr zunächst einmal ein Schaltjahr.
- Ausnahme: Wenn die Jahreszahl durch 100 teilbar ist, ist das Jahr kein Schaltjahr.
- Ausnahme von der Ausnahme: Wenn die Jahreszahl durch 400 teilbar ist, ist das Jahr doch wieder ein Schaltjahr.

### Lösung

Am einfachsten ist es, die Aufgabenstellung in geschachtelte **if**-Verzweigungen zu übersetzen. Im folgenden finden Sie eine Funktion `is_leap_year()`, der man das Jahr übergibt und die für Schaltjahre **1** zurückgibt und für Nicht-Schaltjahre **0**.

```
#include <stdio.h>

int is_leap_year (int year)
{
    int leap_year = 0;
    if (year % 4 == 0)
    {
        leap_year = 1;
        if (year % 100 == 0)
        {
            leap_year = 0;
            if (year % 400 == 0)
                leap_year = 1;
        }
    }
    return leap_year;
}
```

(In C steht **0** für den Wahrheitswert „falsch“ und jeder Wert ungleich **0** für den Wahrheitswert „wahr“; die Zeile `leap_year = 0` steht daher wörtlich und selbsterklärend für „ist kein Schaltjahr“.)

Unter Verwendung von **else** lässt sich dies verkürzen zu:

```
#include <stdio.h>

int is_leap_year (int year)
{
    if (year % 4 == 0)
    {
        if (year % 100 == 0)
        {
            if (year % 400 == 0)
                return 1;
            else
                return 0;
        }
        else
            return 1;
    }
    else
        return 0;
}
```

Eine andere Möglichkeit ist es, die Schaltjahr-Bedingung in eine Kette von „und“- und „oder“-Verknüpfungen (C-Operatoren `&&` und `||`) zu übersetzen:

```
int is_leap_year (int year)
{
    if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
        return 1;
    else
        return 0;
}
```

Dies ist zwar kürzer, aber nicht unbedingt übersichtlicher. Der erzeugte Code ist übrigens *nicht* kürzer und/oder effizienter als bei der Verwendung mehrerer `if`-Verzweigungen. Wir empfehlen, daß Sie immer so programmieren, daß Sie selbst den maximalen Überblick über Ihr Programm behalten.

Ein Hauptprogramm, das die o. a. Funktion aufruft, könnte dann wie folgt aussehen:

```
int main (void)
{
    int year;
    printf ("Bitte geben Sie eine Jahreszahl ein: ");
    scanf ("%d", &year);
    if (is_leap_year (year))
        printf ("Das Jahr %d ist ein Schaltjahr.\n", year);
    else
        printf ("Das Jahr %d ist kein Schaltjahr.\n", year);
    return 0;
}
```

In den Dateien [loesung-1-1.c](#) bis [loesung-1-3.c](#) finden Sie lauffähige Programme, die die o. a. Funktionen aufrufen. Beachten Sie, daß die Funktion *vor* dem Hauptprogramm deklariert werden muß, damit das Hauptprogramm sie kennt. (Es gibt Tricks, mit denen es auch anders geht, aber was hätten wir in diesem Zusammenhang davon?)

In [loesung-1-4.c](#) und [loesung-1-5.c](#) findet die Schaltjahr-Prüfung direkt im Hauptprogramm statt. Dies ist ebenfalls eine richtige Lösung der Aufgabe, schränkt aber die Wiederverwertbarkeit des Codes ein.

Die Datei [loesung-1-4.c](#) enthält darüberhinaus Codeverdopplungen, nämlich mehrere identische `printf()`-Aufrufe an unterschiedlichen Stellen. Dies ist schlechter Programmierstil („Cut-and-paste-Programmierung“).

Die besten Lösungen sind [loesung-1-2.c](#) und [loesung-1-3.c](#).

Zum Testen:

- 1900 ist kein Schaltjahr.
- 1902 ist kein Schaltjahr.
- 1904 ist ein Schaltjahr.
- 1996 ist ein Schaltjahr.
- 1998 ist kein Schaltjahr.
- 2000 ist ein Schaltjahr.
- 2002 ist kein Schaltjahr.
- 2004 ist ein Schaltjahr.
- 2020 ist ein Schaltjahr.
- 2021 ist kein Schaltjahr.
- 2022 ist kein Schaltjahr.
- 2023 ist kein Schaltjahr.

Hier noch ein Hinweis für Unix-Shell-Experten:

```
for y in 1 2 3 4 5; do
    clear
    for x in 1900 1902 1904 1996 1998 2000 2002 2004 2020 2021 2022 2023; do
        echo $x | ./loesung-1-$y
    done
    sleep 2s
done
```

## Aufgabe 2: Multiplikationstabelle

Geben Sie mit Hilfe einer Schleife ein „Einmaleins“ aus.

Dabei sollen die Faktoren und Ergebnisse rechtsbündig untereinander stehen:

```
1 * 7 = 7
2 * 7 = 14
...
10 * 7 = 70
```

Hinweis: Verwenden Sie Formatspezifikationen wie z. B. `%3d`  
(siehe dazu die Dokumentation zu `printf()`, z. B. `man 3 printf`)

### Lösung

Drei verschiedene richtige Lösungen finden Sie in den Dateien [loesung-2-1.c](#), [loesung-2-2.c](#) und [loesung-2-3.c](#). (Zum Compilieren von [loesung-2-2.c](#) und [loesung-2-3.c](#) ist mindestens der C99-Standard erforderlich; bitte nötigenfalls in `gcc` die Option `-std=c99` mit angeben.)

Die Lösung in [loesung-2-3.c](#) ist zwar richtig, aber unnötig kompliziert und daher nicht empfohlen.

Eine **falsche** Lösung finden Sie in der Datei [loesung-2-f4.c](#): In der Ausgabe dieses Programms stehen die Faktoren und Ergebnisse nicht rechtsbündig untereinander.

## Aufgabe 3: Fibonacci-Zahlen

Die Folge der Fibonacci-Zahlen ist definiert durch:

- 1. Zahl: 0
- 2. Zahl: 1
- nächste Zahl = Summe der beiden vorherigen

Schreiben Sie ein Programm, das die ersten 50 Fibonacci-Zahlen ausgibt.

Falls Ihnen dabei irgendwelche Besonderheiten auffallen und/oder Sie irgendwelche besondere Maßnahmen treffen, dokumentieren Sie diese.

(Wem dies zu einfach ist, kann auch gerne die ersten 100 Fibonacci-Zahlen ausgeben.)

### Lösung

Zwei verschiedene richtige Lösungen finden Sie in den Dateien [loesung-3-1.c](#) und [loesung-3-2.c](#).

Die Lösung in [loesung-3-2.c](#) speichert alle berechneten Zahlen in einem Array, die in [loesung-3-1.c](#) hingegen speichert immer nur maximal drei Zahlen gleichzeitig. Sofern nicht alle berechneten Zahlen später noch benötigt werden, ist daher [loesung-3-1.c](#) zu bevorzugen.

Wichtig in [loesung-3-1.c](#) ist, daß `f0 + f1` berechnet wird, *bevor* `f0` oder `f1` ein neuer Wert zugewiesen wird. Dies ist nur möglich, weil das Programm eine zusätzliche Variable (hier: `f2`) verwendet.

Eine „Besonderheit“ besteht darin, daß das Ergebnis ab der Fibonacci-Zahl Nr. 47 **falsch** ist:

```
f[45] = 1134903170
f[46] = 1836311903
f[47] = -1323752223
f[48] = 512559680
f[49] = -811192543
```

Die Summe zweier positiver Zahlen darf keine negative Zahl sein.

(Mit der Dokumentation dieser Beobachtung ist die Aufgabenstellung bereits erfüllt.)

Der Grund für diese fehlerhafte Rechnung ist die begrenzte Rechengenauigkeit unserer Rechner, in diesem Fall ein sogenannter *Integer-Überlauf*. Details dazu sind Gegenstand von Kapitel 5 der Lehrveranstaltung.

Eine „besondere Maßnahme“ besteht darin, anstelle des „normalen“ Ganzzahl-Datentyps **int** „lange“ ganze Zahlen (**long int**) zu verwenden – siehe [loesung-3-3.c](#). Damit nicht nur die Rechnung, sondern auch die Ausgabe funktioniert, muß die Formatspezifikation in `printf()` von `%d` zu `%ld` angepaßt werden – siehe [loesung-3-4.c](#). (Anstelle von **long int** verwendet man üblicherweise abkürzend **long** – siehe [loesung-3-5.c](#).)

Wenn die genaue Rechengenauigkeit wichtig ist, sind die Definitionen der Datentypen **int** und **long** zu unpräzise. Für derartige Fälle gibt es spezielle Datentypen, z. B. `uint64_t` für vorzeichenlose 64-Bit-Ganzzahlen. Für die Ausgabe derartiger Datentypen sind spezielle Formatspezifikationen erforderlich. Die Beispiel-Lösungen [loesung-3-5.c](#) und [loesung-3-6.c](#) funktionieren zwar, zeigen aber *noch nicht* die bestmögliche Lösung.

**Achtung:** Die Verwendung von Fließkommazahlen für ganzzahlige Berechnungen wie bei den Fibonacci-Zahlen ist **nicht** zielführend! Die in [loesung-3-7f.c](#) bis [loesung-3-12f.c](#) vorgestellte „Lösung“ ist **falsch**.

## Aufgabe 4: Fehlerhaftes Programm

Wir betrachten das nebenstehende C-Programm (Datei: [aufgabe-4.c](#)).

- (a) Was bewirkt dieses Programm? Begründen Sie Ihre Antwort.

Schreiben Sie Ihre Begründung so auf, daß man sie auch dann versteht, wenn man gerade nicht die Möglichkeit hat, bei Ihnen persönlich nachzufragen (z. B. weil man gerade eine Klausur korrigiert).

Die Schwierigkeit dieser Aufgabe besteht nicht allein darin, die Problematik zu verstehen, sondern auch darin, dieses Verständnis für andere aufzuschreiben.

- (b) Ändern Sie das Programm so um, daß es einen „Countdown“ von 10 bis 0 ausgibt.

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

```
int main (void)
```

```
{
    for (int i = 10; i = 0; i - 1)
        printf ("%d\n", i);
    return 0;
}
```

## Lösung

- (a) **Was bewirkt dieses Programm? Begründen Sie Ihre Antwort.**

Dieses Programm bewirkt nichts. Die **for**-Schleife wird nicht ausgeführt.

Begründung: Die **for**-Bedingung ist eine Zuweisung des Werts **0** an die Variable **i**. Neben dem Seiteneffekt der Zuweisung liefert der Ausdruck einen Wert zurück, nämlich den zugewiesenen Wert **0**. Dieser wird von **for** als eine Bedingung mit dem konstanten Wert „falsch“ interpretiert.

(Hinweis: Ohne diese Begründung ist die Aufgabe nur zu einem kleinen Teil gelöst.)

Darüberhinaus ist die Zähl-Anwendung unwirksam: Sie berechnet den Wert **i - 1** und vergißt ihn wieder, ohne ihn einer Variablen (z. B. **i**) zuzuweisen.

- (b) **Ändern Sie das Programm so, daß es einen „Countdown“ von 10 bis 0 ausgibt.**

Datei `loesung-4.c`:

```
#include <stdio.h>

int main (void)
{
    for (int i = 10; i >= 0; i--)
        printf ("%d\n", i);
    return 0;
}
```