

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 29. Oktober 2018

Aufgabe 1: Strings

Strings werden in der Programmiersprache C durch Zeiger auf **char**-Variable realisiert.

Wir betrachten die folgende Funktion (Datei: [aufgabe-1.c](#)):

```
int fun_1 (char *s1, char *s2)
{
    int result = 1;
    for (int i = 0; s1[i] && s2[i]; i++)
        if (s1[i] != s2[i])
            result = 0;
    return result;
}
```

- (a) Was bewirkt die Funktion?
- (b) Welchen Sinn hat die Bedingung „s1[i] && s2[i]“ in der **for**-Schleife?
- (c) Was würde sich ändern, wenn die Bedingung „s1[i] && s2[i]“ in der **for**-Schleife zu „s1[i]“ verkürzt würde?
- (d) Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie **fun_1()**, nur effizienter.

Lösung

- (a) **Was bewirkt die Funktion?**

Sie vergleicht zwei Strings miteinander bis zur Länge des kürzeren Strings und gibt bei Gleichheit 1 zurück, ansonsten 0.

Mit anderen Worten: Die Funktion prüft, ob zwei Strings bis zur Länge des kürzeren übereinstimmen, und gibt bei Gleichheit 1 zurück, ansonsten 0.

Die Funktion prüft insbesondere **nicht** zwei Strings auf Gleichheit, und sie ist **nicht** funktionsgleich zur Standard-Bibliotheksfunktion **strcmp()**.

- (b) **Welchen Sinn hat die Bedingung „s1[i] && s2[i]“ in der for-Schleife?**

Die Bedingung prüft, ob *bei einem der beiden Strings* die Ende-Markierung (Null-Symbol) erreicht ist. Falls ja, wird die Schleife beendet.

- (c) **Was würde sich ändern, wenn die Bedingung „s1[i] && s2[i]“ in der for-Schleife zu „s1[i]“ verkürzt würde?**

In diesem Fall würde nur für **s1** geprüft, ob das Ende erreicht ist. Wenn **s1** länger ist als **s2**, würde **s2** über sein Ende hinaus ausgelesen. Dies kann zu Lesezugriffen auf Speicher außerhalb des Programms und damit zu einem Absturz führen („Speicherzugriffsfehler“, „Schutzverletzung“).

- (d) **Schreiben Sie eine eigene Funktion, die dieselbe Aufgabe erledigt wie fun_1(), nur effizienter.**

Die Effizienz lässt sich steigern, indem man die Schleife abbricht, sobald das Ergebnis feststeht. Es folgen drei Möglichkeiten, dies zu realisieren.

Erweiterung der Schleifenbedingung:

```
int fun_2 (char *s1, char *s2)
{
    int result = 1;
    for (int i = 0; s1[i] && s2[i] && result; i++)
        if (s1[i] != s2[i])
            result = 0;
    return result;
}
```

Verwendung von **return**:

```
int fun_3 (char *s1, char *s2)
{
    for (int i = 0; s1[i] && s2[i]; i++)
        if (s1[i] != s2[i])
            return 0;
    return 1;
}
```

Die nebenstehende Lösung unter Verwendung von **break** ist zwar ebenfalls richtig, aber länger und weniger übersichtlich als die beiden anderen Lösungen.

Die Datei **loesung-1.c** enthält ein Testprogramm für alle o. a. Lösungen. Das Programm testet nur die offensichtlichsten Fälle; für den Einsatz der Funktionen in einer Produktivumgebung wären weitaus umfassendere Tests erforderlich.

Das Testprogramm enthält String-Zuweisungen wie z. B. **s2 = "Apfel"**. Dies funktioniert, weil wir damit einen Zeiger (**char *s2**) auf einen neuen Speicherbereich ("Apfel") zeigen lassen. Eine entsprechende Zuweisung zwischen Arrays (**char s3[] = "Birne"; s3 = "Pfirsich";**) funktioniert *nicht*.

```
int fun_4 (char *s1, char *s2)
{
    int result = 1;
    for (int i = 0; s1[i] && s2[i]; i++)
        if (s1[i] != s2[i])
        {
            result = 0;
            break;
        }
    return result;
}
```

Aufgabe 2: Fehlerhaftes Programm: Primzahlen

Das nebenstehende Primzahlprogramm (Datei: **aufgabe-2.c**) soll Zahlen ausgeben, die genau zwei Teiler haben, ist aber fehlerhaft.

Korrigieren Sie das Programm derart, daß ein Programm entsteht, welches alle Primzahlen kleiner 100 ausgibt.

```
#include <stdio.h>

int main (void)
{
    int n, i, divisors;
    for (n = 0; n < 100; n++)
        divisors = 0;
        for (i = 0; i < n; i++)
            if (n % i == 0)
                divisors++;
            if (divisors = 2)
                printf ("%d ist eine Primzahl.\n", n);
    return 0;
}
```

Lösung

Beim Compilieren des Beispiel-Programms mit **gcc -Wall** erhalten wir die folgende Warnung:

```
aufgabe-2.c:11:5: warning: suggest parentheses around assignment
used as truth value [-Wparentheses]
```

Beim Ausführen gibt das Programm die folgende (falsche) Behauptung aus:

```
100 ist eine Primzahl.
```

Einen ersten Hinweis auf den Fehler im Programm liefert die Warnung. Die Bedingung **if (divisors = 2)** in Zeile 11 steht *nicht* für einen Vergleich der Variablen **divisors** mit der Zahl 2, sondern für eine Zuweisung der Zahl 2 an die Variable **divisors**. Neben dem *Seiteneffekt* der Zuweisung gibt **divisors = 2** den Wert 2 zurück. Als Bedingung interpretiert, hat 2 den Wahrheitswert „wahr“ („true“); die **printf()**-Anweisung wird daher in jedem Fall ausgeführt.

Korrektur dieses Fehlers: **if (divisors == 2)** – siehe die Datei **loesung-2-1.c**.

Nach der Korrektur dieses Fehlers compiliert das Programm ohne Warnung, gibt aber beim Ausführen die folgende Fehlermeldung aus:

```
Gleitkomma-Ausnahme
```

(Bemerkung: Bei ausgeschalteter Optimierung – **gcc** ohne **-O** – kommt diese Fehlermeldung bereits beim ersten Versuch, das Programm auszuführen. Der Grund für dieses Verhalten ist, daß bei eingeschalteter Optimierung irrelevante Teile des Programms entfernt und gar nicht ausgeführt werden, so daß der Fehler nicht zum Tragen kommt. In diesem Fall wurde die Berechnung von **divisors** komplett wegoptimiert, da der Wert dieser Variablen nirgendwo abgefragt, sondern durch die Zuweisung **if (divisors = 2)** sofort wieder überschrieben wurde.)

Die Fehlermeldung „Gleitkomma-Ausnahme“ ist insofern irreführend, als daß hier gar keine Gleitkommazahlen im Spiel sind; andererseits deutet sie auf einen Rechenfehler hin, was auch tatsächlich zutrifft. Durch Untersuchen aller Rechenoperationen – z. B. durch das Einfügen zusätzlicher `printf()` – finden wir den Fehler in Zeile 9: Die Modulo-Operation `n % i` ist eine Division, die dann fehlschlägt, wenn der Divisor `i` den Wert 0 hat. Die Fehlerursache ist die bei 0 beginnende `for`-Schleife in Zeile 8: `for (i = 0; i < n; i++)`.

Korrektur dieses Fehlers: Beginn der Schleife mit `i = 1` statt `i = 0` – siehe die Datei [loesung-2-2.c](#).

Nach der Korrektur dieses Fehlers gibt das Programm überhaupt nichts mehr aus.

Durch Untersuchen des Verhaltens des Programms – z. B. durch das Einfügen zusätzlicher `printf()` – stellen wir fest, daß die Zeilen 8 bis 12 des Programms nur einmal ausgeführt werden und nicht, wie die `for`-Schleife in Zeile 6 vermuten ließe, 100mal. Der Grund dafür ist, daß sich die `for`-Schleife nur auf die unmittelbar folgende Anweisung `divisors = 0` bezieht. Nur diese Zuweisung wird 100mal ausgeführt; alles andere befindet sich außerhalb der `for`-Schleife. (Die Einrückung hat in C keine inhaltliche Bedeutung, sondern dient nur zur Verdeutlichung der Struktur des Programms. In diesem Fall entsprach die tatsächliche Struktur nicht der beabsichtigten.)

Korrektur dieses Fehlers: geschweifte Klammern um den Inhalt der äußeren `for`-Schleife – siehe die Datei [loesung-2-3.c](#).

Nach der Korrektur dieses Fehlers gibt das Programm folgendes aus:

```
4 ist eine Primzahl.  
9 ist eine Primzahl.  
25 ist eine Primzahl.  
49 ist eine Primzahl.
```

Diese Zahlen sind keine Primzahlen (mit zwei Teilern), sondern sie haben drei Teiler. Demnach findet das Programm einen Teiler zu wenig. (Um diesen Fehler zu finden, kann man sich zu jeder Zahl die gefundene Anzahl der Teiler `divisors` ausgeben lassen.)

Der nicht gefundene Teiler ist jeweils die Zahl selbst. Dies kommt daher, daß die Schleife `for (i = 1; i < n; i++)` nur bis `n - 1` geht, also keine Division durch `n` stattfindet.

Korrektur dieses Fehlers: Schleifenbedingung `i <= n` statt `i < n` – siehe die Datei [loesung-2-4.c](#).

Nach der Korrektur dieses Fehlers verhält sich das Programm korrekt.

Die Datei [loesung-2-4.c](#) enthält somit das korrigierte Programm.

Aufgabe 3: Datum-Bibliothek

Schreiben Sie eine Bibliothek (= Sammlung von Deklarationen und Funktionen) zur Behandlung von Datumsangaben.

Diese soll enthalten:

- einen `struct`-Datentyp `date`, der eine Datumsangabe speichert,
- eine Funktion `void date_print (date *d)`, die ein Datum ausgibt,
- eine Funktion `int date_set (date *d, int day, int month, int year)`, die ein Datum auf einen gegebenen Tag setzt und zurückgibt, ob es sich um ein gültiges Datum handelt (0 = nein, 1 = ja),
- eine Funktion `void date_next (date *d)`, die ein Datum auf den nächsten Tag vorrückt.

Lösung

Die Datei [loesung-3.c](#) enthält die Bibliothek zusammen mit einem Test-Programm.

Eine detaillierte Anleitung, wie man auf die Funktion `date_next()` kommt, finden Sie im Skript zur Lehrveranstaltung, Datei [hp-2018ws.pdf](#), ab Seite 29.

(Die Vorgehensweise, die Bibliothek und das Hauptprogramm in dieselbe Datei zu schreiben, hat den Nachteil, daß man die Bibliothek in jedes weitere Programm, das sie benutzt, kopieren und auch dort aktuell halten muß. Eine sinnvollere Lösung wird demnächst in der Vorlesung vorgestellt werden.)