

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben 3 – 7. November 2024

Aufgabe 1: Arrays mit Zahlen

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

```
#include <stdio.h>

void f (int *s0, int *s1)
{
    while (*s0 >= 0)
    {
        int *s = s1;
        while (*s >= 0)
            if (*s0 == *s++)
                printf ("%d_", *s0);
        s0++;
    }
    printf ("\n");
}

int main (void)
{
    int a[] = { 10, 4, 3, 7, 12, 0, 1, -1 };
    int b[] = { 7, 14, 0, 8, 9, 22, 10, -1 };
    f (a, b);
    return 0;
}
```

- (a) Was bewirkt die Funktion `f`, und wie funktioniert sie? (4 Punkte)
- (b) Was passiert, wenn Sie beim Aufruf der Funktion für einen der Parameter den Wert `NULL` übergeben? Begründen Sie Ihre Antwort. (2 Punkte)
- (c) Was kann passieren, wenn Sie das Hauptprogramm wie folgt abändern ([aufgabe-1c.c](#))? Begründen Sie Ihre Antwort.

```
int main (void)
{
    int a[] = { 10, 4, 3, 7, 12, 0, 1 };
    int b[] = { 7, 14, 0, 8, 9, 22, 10 };
    f (a, b);
    return 0;
}
```

(2 Punkte)

Lösung

- (a) Was bewirkt die Funktion `f`, und wie funktioniert sie?

Die Funktion gibt alle Zahlen aus, die sowohl im Array `s0` als auch im Array `s1` vorkommen (Schnittmenge).

Dies geschieht, indem der Zeiger `s0` das gesamte Array durchläuft (äußere Schleife). Für jedes Element des ersten Arrays durchläuft der Zeiger `s` das gesamte zweite Array (innere Schleife). Auf diese Weise wird jedes Element von `s0` mit jedem von `s1` verglichen und bei Gleichheit ausgegeben.

Um die Schleifen abbrechen zu können, enthalten beide Arrays als Ende-Markierung eine negative Zahl (`-1`).

- (b) Was passiert, wenn Sie beim Aufruf der Funktion für einen der Parameter den Wert `NULL` übergeben? Begründen Sie Ihre Antwort.

In dem Moment, wo auf den jeweiligen Parameter-Zeiger zugegriffen wird (`while (*s0 >= 0)` für `s0` bzw. `int *s = s1; while (*s >= 0)` für `s1`), kommt es zu einem Absturz (Speicherzugriffsfehler). Die Dereferenzierung eines Zeigers mit dem Wert `NULL` ist nicht zulässig.

- (c) Was kann passieren, wenn Sie das Hauptprogramm wie folgt abändern ([aufgabe-1c.c](#))? Begründen Sie Ihre Antwort.

```
int main (void)
{
    int a[] = { 10, 4, 3, 7, 12, 0, 1 };
    int b[] = { 7, 14, 0, 8, 9, 22, 10 };
    f (a, b);
    return 0;
}
```

Durch die fehlenden Ende-Markierungen der Arrays laufen die Schleifen immer weiter, bis sie irgendwann zufällig auf Speicherzellen stoßen, die sich als Ende-Markierungen interpretieren lassen (negative Zahlen). Dadurch kann es zu einem Lesezugriff auf Speicher kommen, für den das Programm kein Lesezugriffsrecht hat, also zu einem Absturz (Speicherzugriffsfehler).

Aufgabe 2: Einfügen in Strings

Wir betrachten das folgende Programm (aufgabe-2.c):

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

void insert_into_string (char src, char *target, int pos)
{
    int len = strlen (target);
    for (int i = pos; i < len; i++)
        target[i+1] = target[i];
    target[pos] = src;
}

int main (void)
{
    char test[100] = "Hochshule_Bochum";
    insert_into_string ('c', test, 5);
    printf ("%s\n", test);
    return 0;
}
```

Die Ausgabe des Programms lautet: `Hochschhhhhhhhhhh`

- (a) Erklären Sie, wie die Ausgabe zustandekommt. (3 Punkte)
- (b) Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstaben `src` an der Stelle `pos` in den String `target` einfügt.
Die Ausgabe des Programms müßte dann `Hochschhule Bochum` lauten. (2 Punkte)
- (c) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)
- (d) Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen? Begründen Sie Ihre Antwort. (2 Punkte)

Lösung

- (a) **Erklären Sie, wie die Ausgabe zustandekommt.**
In der Schleife wird *zuerst* der nächste Buchstabe `target[i + 1]` gleich dem aktuellen gesetzt und *danach* der Zähler `i` erhöht. Dadurch wird im nächsten Schleifendurchlauf der bereits verschobene Buchstabe noch weiter geschoben und letztlich alle Buchstaben in `target[]` durch den an der Stelle `pos` ersetzt.
- (b) **Schreiben Sie die Funktion `insert_into_string()` so um, daß sie den Buchstben `src` an der Stelle `pos` in den String `target` einfügt.**
Die Ausgabe des Programms müßte dann `Hochschhule Bochum` lauten.
Um im String „Platz zu schaffen“, muß man von hinten beginnen, also die Schleife umdrehen (siehe: `loesung-2.c`):

```
for (int i = len; i >= pos; i--)
    target[i + 1] = target[i];
```
- (c) **Was kann passieren, wenn Sie die Zeile `char test[100] = "Hochshule_Bochum";` durch `char test[] = "Hochshule_Bochum";` ersetzen und warum?**
Die Schreibweise `test[]` bedeutet, daß der Compiler selbst zählt, wieviel Speicherplatz der String benötigt, un dann genau die richtige Menge Speicher reserviert (anstatt, wie wir es manuell getan haben, pauschal Platz für 100 Zeichen).

Wenn wir nun in den String ein zusätzliches Zeichen einfügen, ist dafür kein Speicherplatz reserviert worden, und wir **überschreiben** dann Speicher, an dem sich andere Variable befinden, was zu einem **Absturz** führen kann.

Da wir hier nur ein einziges Zeichen schreiben, wird dieser Fehler nicht sofort auffallen. Dies ist schlimmer, als wenn das Programm direkt beim ersten Test abstürzt, denn dadurch entsteht bei uns der Eindruck, es sei in Ordnung. Wenn danach der Fehler in einer Produktivumgebung auftritt, kann dadurch Schaden entstehen – je nach Einsatzgebiet der Software u. U. erheblicher Vermögens-, Sach- und/oder Personenschaden (z. B. Absturz eines Raumflugkörpers).

- (d) Was kann passieren, wenn Sie `char test[100] = "Hochshule_Bochum";` durch `char *test = "Hochshule_Bochum";` ersetzen und warum?

In diesem Fall wird der Speicher für den eigentlichen String in einem unbenannten, **nicht schreibbaren** Teil des Speichers reserviert. Unser Versuch, dorthin ein zusätzliches Zeichen zu schreiben, führt dann normalerweise zu einem **Absturz**.

In manchen Systemen (Betriebssystem, Compiler, ...) ist der Speicherbereich tatsächlich sehr wohl schreibbar. In diesem Fall tritt der Absturz nicht immer und nicht immer sofort auf – genau wie in Aufgabenteil (c).

Aufgabe 3: Fehlerhaftes Primzahl-Programm

Das nebenstehende Primzahlsuchprogramm (Datei: [aufgabe-3.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. (5 Punkte)

```
#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-3-1.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-3-2.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-3-3.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-3-4.c](#).

Nach der Korrektur dieses Fehlers verhält sich das Programm korrekt.

Die Datei [loesung-3-4.c](#) enthält somit das korrigierte Programm.