

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben 2 – 31. Oktober 2024

Aufgabe 1: Seltsame Programme

Unter <https://gitlab.cvh-server.de/pgerwinski/hp/tree/2024ws/20241031> finden Sie (unter anderem) die Programme `test-1.c`, `test-2.c` und `test-3.c`.

Was bewirken diese Programme, und warum verhalten sie sich so?

Lösung

- `test-1.c`

Hinter `return` steht ein Ausdruck mit dem Komma-Operator. Dieser bewirkt, daß der Wert vor dem Komma berechnet und ignoriert und danach der Wert nach dem Komma zurückgegeben wird.

In diesem Fall wird vor dem Komma der Wert des `printf()`-Aufrufs berechnet und ignoriert. Als Seiteneffekt gibt das Programm die Zeile `Hello, world!` aus. Anschließend wird der Wert `0` an `return` übergeben und daher `return 0` ausgeführt.

- `test-2.c`

Das Programm gibt die Zeile `Die Antwort lautet: 42` aus.

Die `if`-Bedingung ist eine Zuweisung `b = 42`, die den zugewiesenen Wert `42` zurückgibt. Weil dieser Wert ungleich Null ist, interpretiert `if` ihn als Wahrheitswert „wahr“, führt also den `if`-Zweig aus und überspringt den `else`-Zweig.

- `test-3.c`

Das Programm stürzt mit einer Fehlermeldung „Speicherzugriffsfehler“ oder „Schutzverletzung“ ab.

Der Funktionsaufruf `printf (42)` übergibt den Zahlenwert `42` als String, also als einen Zeiger auf `char`-Variable, an die Funktion `printf()`. Diese versucht, auf den Speicher ab Adresse `42` zuzugreifen, wofür aber das Programm keine Zugriffsrechte hat. Das Betriebssystem beendet daraufhin das Programm mit der o. a. Fehlermeldung.

Der String `"Die_Antwort_lautet:_"` wird nicht ausgegeben, weil Schreiboperationen aus Effizienzgründen erst nach einer abgeschlossenen Zeile (`"\n"`) durchgeführt werden.

Aufgabe 2: Kalender-Berechnung

Am 3. 1. 2009 meldete *heise online*:

Kunden des ersten mobilen Media-Players von Microsoft erlebten zum Jahresende eine böse Überraschung: Am 31. Dezember 2008 fielen weltweit alle Zune-Geräte der ersten Generation aus. Ursache war ein interner Fehler bei der Handhabung von Schaltjahren.

<http://heise.de/-193332>,

Der Artikel verweist auf ein Quelltextfragment (Datei: [aufgabe-2.c](#)), das für einen gegebenen Wert `days` das Jahr und den Tag innerhalb des Jahres für den `days`-ten Tag nach dem 1. 1. 1980 berechnen soll:

```
year = ORIGINYEAR; /* = 1980 */

while (days > 365)
{
    if (IsLeapYear (year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

Dieses Quelltextfragment enthält schlechten Programmierstil, nämlich mehrere Code-Verdopplungen:

- Die Anweisung `year += 1` taucht an zwei Stellen auf.
- Es gibt zwei unabhängige Abfragen `days > 365` und `days > 366`: eine in einer `while`- und die andere in einer `if`-Bedingung.
- Die Länge eines Jahres wird nicht durch eine Funktion berechnet oder in einer Variablen gespeichert; stattdessen werden an mehreren Stellen die expliziten numerischen Konstanten 365 und 366 verwendet.

Diese Probleme führten am 31. Dezember 2008 zu einer Endlosschleife.

Gut hingegen ist die Verwendung einer Konstanten `ORIGINYEAR` anstelle der Zahl 1980 sowie die Kapselung der Berechnung der Schaltjahr-Bedingung in einer Funktion `IsLeapYear()`.

- (a) Erklären Sie das Zustandekommen der Endlosschleife.
- (b) Schreiben Sie das Quelltextfragment so um, daß es die beschriebenen Probleme nicht mehr enthält.

Hinweis 1: Verwenden Sie für `IsLeapYear()` Ihre eigene Funktion aus Aufgabe 1 der letzten Übung.

Hinweis 2: Schreiben Sie zusätzlich eine Funktion `DaysInYear()`.

Lösung

(a) **Erklären Sie das Zustandekommen der Endlosschleife.**

Das Programm startet mit demjenigen Wert für `days`, der der Anzahl der Tage vom 1. 1. 1980 bis zum 31. 12. 2008 entspricht. Die `while`-Schleife läuft zunächst solange korrekt durch, bis `year` den Wert 2008 und `days` den Wert 366 hat. (Der 31. 12. des Schaltjahres 2008 ist der 366. Tag seines Jahres.)

Die Bedingung der `while`-Schleife ist damit weiterhin erfüllt; das Programm läuft weiter.

Da 2008 ein Schaltjahr ist, ist auch die Bedingung der äußeren `if`-Anweisung erfüllt.

Da `days` den Wert 366 hat und dieser nicht größer als 366 ist, ist die innere `if`-Bedingung nicht erfüllt. Somit wird innerhalb der `while`-Schleife kein weiterer Code ausgeführt, die `while`-Bedingung bleibt erfüllt, und das Programm führt eine Endlosschleife aus.

(b) **Schreiben Sie das Quelltextfragment so um, daß es die beschriebenen Probleme nicht mehr enthält.**

Um das Programm zu testen, genügt es, das Datum auf den 31. 12. 1980 zu stellen, also `days` auf den Wert 366 zu setzen. Darüberhinaus muß man die Funktion `isLeapYear()` bereitstellen (vgl. Aufgabe 1 vom 11. 10. 2021).

Der Quelltext `loesung-2-f1.c` ist eine lauffähige Version des Programms, die den Fehler (Endlosschleife) reproduziert.

Es liegt nahe, den Fehler in der `while`-Bedingung zu korrigieren, so daß diese Schaltjahre berücksichtigt. Der Quelltext `loesung-2-f2.c` behebt den Fehler auf diese Weise mit Hilfe von Und- (`&&`) und Oder-Verknüpfungen (`||`) in der `while`-Bedingung.

Der Quelltext `loesung-2-f3.c` vermeidet die umständliche Formulierung mit `&&` und `||` durch Verwendung des ternären Operators `?:`. Dieser stellt eine „`if`-Anweisung für Ausdrücke“ bereit. In diesem Fall liefert er für die rechte Seite des Vergleichs `days > den Wert 366` im Falle eines Schaltjahrs bzw. ansonsten den Wert 365.

Beide Lösungen `loesung-2-f2.c` und `loesung-2-f3.c` sind jedoch im Sinne der Aufgabenstellung **falsch**. Diese lautet: „Schreiben Sie das Quelltextfragment so um, daß es die beschriebenen Probleme nicht mehr enthält.“ Mit den beschriebenen Problemen sind die genannten drei Code-Verdopplungen gemeint, und diese befinden sich weiterhin im Quelltext. Damit ist der Fehler zwar „korrigiert“, aber das Programm ist eher noch unübersichtlicher geworden, so daß nicht klar ist, ob es nicht noch weitere Fehler enthält.

Eine richtige Lösung liefert `loesung-2-4.c`. Dieses Programm speichert den Wert der Tage im Jahr in einer Variablen `DaysInYear`. Damit erübrigen sich die `if`-Anweisungen innerhalb der `while`-Schleife, und die damit verbundenen Code-Verdopplungen verschwinden.

Etwas unschön ist hierbei die neu hinzugekommene Code-Verdopplung bei der Berechnung von `DaysInYear`. Diese ist allerdings weniger kritisch als die vorherigen, da sie nur einmal innerhalb der `while`-Schleife vorkommt und das andere Mal außerhalb derselben.

Um diese Code-Verdopplung loszuwerden, kann man das `if` durch den `?:`-Operator ersetzen und die Zuweisung innerhalb der `while`-Bedingung vornehmen – siehe `loesung-2-5.c`. Dies ist einer der seltenen Fälle, in denen ein Programm *übersichtlicher* wird, wenn eine Zuweisung innerhalb einer Bedingung stattfindet.

Alternativ kann `DaysInYear()` auch eine Funktion sein – siehe `loesung-2-6.c`. Diese Version ist wahrscheinlich die übersichtlichste, hat jedoch den Nachteil, daß die Berechnung von `DaysInYear()` zweimal statt nur einmal pro Schleifendurchlauf erfolgt, wodurch Rechenzeit verschwendet wird.

`loesung-2-7.c` und `loesung-2-8.c` beseitigen dieses Problem durch eine Zuweisung des Funktionsergebnisses an eine Variable – einmal innerhalb der `while`-Bedingung und einmal außerhalb. Der zweimalige Aufruf der Funktion `DaysInYear()` in `loesung-2-8.c` zählt nicht als Code-Verdopplung, denn der Code ist ja in einer Funktion gekapselt. (Genau dazu sind Funktionen ja da: daß man sie mehrfach aufrufen kann.)

Fazit: Wenn Sie sich beim Programmieren bei Cut-And-Paste-Aktionen erwischen, sollten Sie die Struktur Ihres Programms noch einmal überdenken.

Wahrscheinlich gibt es dann eine elegantere Lösung, deren Korrektheit man auf den ersten Blick sieht.

Aufgabe 3: Strings

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

Wir betrachten die folgende Funktion (Datei: [aufgabe-3.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.

Alternative Formulierung: 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-3.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 4: Programm analysieren

Wir betrachten das folgende C-Programm (Datei: `aufgabe-4.c`):

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%;c";main(){printf(f,34,f,34,10);}
```

- (a) Was bewirkt dieses Programm?
- (b) Wofür stehen die Zahlen?
- (c) Ergänzen Sie das Programm derart, daß seine `main()`-Funktion `int main (void)` lautet und eine **return**-Anweisung hat, wobei die in Aufgabenteil (a) festgestellte Eigenschaft erhalten bleiben soll.

Lösung

- (a) **Was bewirkt dieses Programm?**

Es gibt *seinen eigenen Quelltext* aus.

(Wichtig ist die Bezugnahme auf den eigenen Quelltext. Die Angabe

„Es gibt `char*f="char*f=%c%s%c;main(){printf(f,34,f,34,10);}%;c";main(){printf(f,34,f,34,10);}` aus“ genügt insbesondere nicht.)

- (b) **Wofür stehen die Zahlen?**

Die 34 steht für ein Anführungszeichen und die 10 für ein Zeilenendezeichen (`\n`).

Hintergrund: Um den eigenen Quelltext ausgeben zu können, muß das Programm auch Anführungszeichen und Zeilenendezeichen ausgeben. Dies geschieht normalerweise mit vorangestelltem Backslash: `\` bzw. `\n`. Um dann aber den Backslash ausgeben zu können, müßte man diesem ebenfalls einen Backslash voranstellen: `\\`. Damit dies nicht zu einer Endlosschleife wird, verwendet der Programmierer dieses Programms den Trick mit den Zahlen, die durch `%c` als Zeichen ausgegeben werden.

- (c) **Ergänzen Sie das Programm derart, daß seine `main()`-Funktion `int main (void)` lautet und eine **return**-Anweisung hat, wobei die in Aufgabenteil (a) festgestellte Eigenschaft erhalten bleiben soll.**

Datei: `loesung-4.c`

```
char*f="char*f=%c%s%c;int_main(void){printf(f,34,f,34,10);return_0;}%;c";
int main(void){printf(f,34,f,34,10);return 0;}
```

Das Programm ist eine einzige, lange Zeile, die hier nur aus Platzgründen als zwei Zeilen abgedruckt wird. Auf das Semikolon am Ende der „ersten Zeile“ folgt unmittelbar – ohne Leerzeichen – das Schlüsselwort `int` am Anfang der „zweiten Zeile“.

Mit „die in Aufgabenteil (a) festgestellte Eigenschaft“ ist gemeint, daß das Programm weiterhin seinen eigenen Quelltext ausgeben soll. Die Herausforderung dieser Aufgabe besteht darin, das Programm zu modifizieren, ohne diese Eigenschaft zu verlieren.

Zusatzaufgabe für Interessierte: Ergänzen Sie das Programm so, daß es auch mit `-Wall` ohne Warnungen kompiliert werden kann.

Hinweis dazu: `#include<stdio.h>` (ohne Leerzeichen, um Platz zu sparen)

Lösung der Zusatzaufgabe: `loesung-4x.c`