

Hardwarenahe Programmierung / Angewandte Informatik

Musterlösung zu den Übungsaufgaben – 12. Dezember 2016

Aufgabe 1: Daten im Speicher

Das folgende C-Programm `aufgabe-1.c` gibt den Speicherbereich, in dem sich seine Variablen befinden, als eine Folge von 8-Bit-Zahlen aus:

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

int16_t a = -1;
int32_t b = 8320;

int main (void)
{
    uint8_t *p = &a;
    for (int i = 0; i < 8; i++)
        printf ("%d", p[i]);
    printf ("\n");
}
```

Das Programm wird ohne Optimierung auf einem 32-Bit-Rechner kompiliert (mit Warnung) und gestartet:

```
$ gcc -std=c99 -Wall aufgabe-1.c -o aufgabe-1
aufgabe-1.c: In function "main":
aufgabe-1.c:9:16: warning: initialization from incompatible pointer type [...]
$ ./aufgabe-1
255 255 0 0 128 32 0 0
```

- (a) Welche Endianness hat der verwendete Rechner und warum? (2 Punkte)
- (b) Erklären Sie die ausgegebenen Zahlen.
Zu welcher Variablen gehört jeweils die Zahl? (4 Punkte)
- (c) Wie würde die Ausgabe auf einem 16-Bit-Rechner mit entgegengesetzter Endianness lauten und warum? (3 Punkte)
- (x) **Freiwillige Zusatzaufgabe:** Erklären Sie, was sich ändert und warum, wenn das Programm auf einem 32-Bit-Rechner *mit* Optimierung (`-O`) kompiliert wird. (5 Extrapunkte)

Hinweis: Aus anderen Lehrveranstaltungen sollte Ihnen der Begriff des Zweierkomplements bekannt sein. Falls nicht, gilt eine Recherche nach diesem Begriff als zugelassenes Hilfsmittel.

Lösung

Vorüberlegung: Der Zeiger `p` zeigt auf `uint8_t`-Variable, also auf einzelne Speicherzellen. In der `for`-Schleife wird achtmal der Inhalt der Speicherzelle ausgegeben, auf die `p` zeigt, und anschließend `p` inkrementiert, also eine Speicherzelle weiter vorgerückt. Auf diese Weise werden 8 fortlaufende Speicherzellen des Rechners ausgegeben, beginnend mit der Adresse der Variablen `a`. Die Ausgabe zeigt uns also die Inhalte der einzelnen Speicherzellen, in denen die Variablen des Programms gespeichert sind.

- (a) **Welche Endianness hat der verwendete Rechner und warum?**

Die 32-Bit-Dezimalzahl 8320 hat die Hexadezimaldarstellung 00002080. Auf 8-Bit-Speicherzellen aufgeteilt, sind dies die Bytes 00, 00, 20 und 80 in Big-Endian-Darstellung – bzw. 80, 20, 00 und 00 in Little-Endian-Darstellung. Dezimal notiert, sind dies die Zahlen 0, 0, 32 und 128 für Big-Endian bzw. 128, 32, 0 und 0 für Little-Endian.

In der Ausgabe des Programms erscheinen die Zahlen 128 und 32 in dieser Reihenfolge, also die 128 zuerst. Daher kann es sich nur um einen Little-Endian-Rechner handeln.

(b) **Erklären Sie die ausgegebenen Zahlen.**

Zu welcher Variablen gehört jeweils die Zahl?

Die ersten beiden Zahlen – zweimal 255 – sind die Binärdarstellung der vorzeichenbehafteten 16-Bit-Zahl -1 (Zweierkomplement von 1), also die Variable **a**.

Die nächsten zwei Zahlen sind 0, gehören aber *nicht* zu der Variablen **b**, sondern werden vom Compiler freigelassen, damit die 32-Bit-Variable **b** auf einer durch 32 Bit (= 4 Bytes) teilbaren Speicherzelle liegt (32-Bit-Alignment).

Es folgen die vier Zahlen der Variablen **b** in Little-Endian-Reihenfolge (siehe Aufgabenteil a).

(c) **Wie würde die Ausgabe auf einem 16-Bit-Rechner mit entgegengesetzter Endianness lauten und warum?**

Auf einem 16-Bit-Rechner findet kein 32-Bit-Alignment statt, sondern nur 16-Bit-Alignment. Es genügt, dann, wenn die Variable **b** auf einer durch 16 Bit (= 2 Bytes) teilbaren Speicherzelle liegt. Daher folgt auf einem 16-Bit-Rechner die Variable im Speicher **b** direkt auf die Variable **a**.

Die Bytes der Variablen **a** lauten zweimal 255, daher ist die Änderung der Byte-Reihenfolge hier nicht sichtbar.

Die Bytes der Variablen **b** in Big-Endian-Reihenfolge lauten 0, 0, 32 und 128 (siehe Aufgabenteil a).

Die nächsten zwei Bytes im Speicher nach der Variablen **b** enthalten zufällige Werte.

Wenn wir für die zufälligen Werte die Zahl 0 annehmen, lautet die Ausgabe:

```
255 255 0 0 32 128 0 0
```

Ebenso richtig wäre aber z. B. auch:

```
255 255 0 0 32 128 42 137
```

(x) **Freiwillige Zusatzaufgabe: Erklären Sie, was sich ändert und warum, wenn das Programm auf einem 32-Bit-Rechner *mit* Optimierung (-O) compiliert wird.**

Wie man durch Ausprobieren direkt sieht, lautet die Ausgabe jetzt:

```
255 255 0 0 0 0 0 0
```

Eine mögliche Erklärung dafür wäre, daß der Compiler die Variable **b** wegoptimiert hätte. Tatsächlich ist dies aber nicht der Fall.

Um dies zu prüfen, lohnt es sich, das Programm nach Assembler zu übersetzen

```
$ gcc -std=c99 -m32 -Wall -O aufgabe-1.c -S -o aufgabe-1-O.s
```

und den Assembler Quelltext [aufgabe-1-O.s](#) zu untersuchen.

Dort stellen wir fest, daß die Variable **b** noch vorhanden ist, im Assembler-Quelltext aber *vor* der Variablen **a** steht.

Ein Vergleich mit dem Assembler-Quelltext [aufgabe-1.s](#) *ohne* Optimierung

```
$ gcc -std=c99 -m32 -Wall aufgabe-1.c -S -o aufgabe-1.s
```

zeigt, daß dort die Reihenfolge der Variablen dieselbe ist wie im C-Programm.

Eine Optimierung des Programms durch den Compiler besteht also darin, daß er die Reihenfolge der globalen Variablen so umsortiert, daß bei korrektem Alignment kein Speicherplatz verschwendet wird.

Die Bytes, die im Speicher hinter der Variablen **a** liegen, sind allesamt *nicht* von C-Variablen belegt, sondern enthalten zufällige Werte. Die Variable **b** befindet sich im Speicher *vor* **a** und wird daher nicht mit ausgegeben.

Aufgabe 2: Zeigerarithmetik

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

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

void output (uint16_t *a)
{
    for (int i = 0; a[i]; i++)
        printf ("_%d", a[i]);
    printf ("\n");
}

int main (void)
{
    uint16_t prime_numbers[] = { 2, 3, 5, 7, 11, 13, 17, 0 };

    uint16_t *p1 = prime_numbers;
    output (p1);
    p1++;
    output (p1);

    char *p2 = prime_numbers;
    output (p2);
    p2++;
    output (p2);

    return 0;
}
```

Das Programm wird kompiliert und ausgeführt:

```
$ gcc -Wall -std=c99 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function 'main':
aufgabe-2.c:20:13: warning: initialization from
                  incompatible pointer type [enabled by default]
aufgabe-2.c:21:3: warning: passing argument 1 of 'output' from
                  incompatible pointer type [enabled by default]
aufgabe-2.c:4:6: note: expected 'uint16_t *' but argument is of type 'char *'
aufgabe-2.c:23:3: warning: passing argument 1 of 'output' from
                  incompatible pointer type [enabled by default]
aufgabe-2.c:4:6: note: expected 'uint16_t *' but argument is of type 'char *'
$ ./aufgabe-2
2 3 5 7 11 13 17
3 5 7 11 13 17
2 3 5 7 11 13 17
768 1280 1792 2816 3328 4352
```

- (a) Erklären Sie die Funktionsweise der Funktion `output ()`. (2 Punkte)
- (b) Begründen Sie den Unterschied zwischen der ersten (2 3 5 7 11 13 17) und der zweiten Zeile (3 5 7 11 13 17) der Ausgabe des Programms. (2 Punkte)
- (c) Erklären Sie die beim Compilieren auftretenden Warnungen und die dritte Zeile (2 3 5 7 11 13 17) der Ausgabe des Programms. (3 Punkte)
- (d) Erklären Sie die vierte Zeile (768 1280 1792 2816 3328 4352) der Ausgabe des Programms. Sie dürfen einen Little-Endian-Rechner voraussetzen. (4 Punkte)

Lösung

- (a) Erklären Sie die Funktionsweise der Funktion `output()`.

Die Funktion bekommt ein Array von ganzen Zahlen als Zeiger übergeben und gibt dessen Inhalt auf den Bildschirm aus, **bis es auf die Zahl 0 stößt**. (Die Ende-Markierung 0 wird nicht mit ausgegeben.)

(Die Erwähnung der Abbruchbedingung der Schleife („bis es auf die Zahl 0 stößt“) ist ein wichtiger Bestandteil der richtigen Lösung. Wenn man nämlich einen Zeiger auf ein Array übergibt, das am Ende keine 0 enthält, liest die Funktion über das Array hinaus zufällige Werte aus dem Speicher. Dies kann zu einem Absturz führen.)

- (b) Begründen Sie den Unterschied zwischen der ersten (2 3 5 7 11 13 17) und der zweiten Zeile (3 5 7 11 13 17) der Ausgabe des Programms.

Zwischen der Ausgabe der ersten und der zweiten Zeile wurde der Zeiger `p1` um 1 inkrementiert; er zeigt danach auf die nächste ganze Zahl im Array (also die zweite Zahl im Array – mit Index 1 statt 0). Als Folge davon wird beim zweiten Aufruf von `output()` die erste Zahl des Arrays nicht mehr mit ausgegeben.

- (c) Erklären Sie die beim Compilieren auftretenden Warnungen und die dritte Zeile (2 3 5 7 11 13 17) der Ausgabe des Programms.

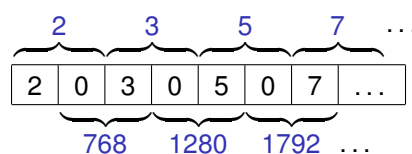
Die Warnungen kommen daher, daß die Variable `p2` ein Zeiger auf `char`-Variable ist, ihr jedoch ein anderer Zeigertyp, nämlich ein Zeiger auf `uint16_t`-Variable, zugewiesen wird. Beim Aufruf der Funktion `output()` wird umgekehrt dem Parameter `a`, der auf `uint16_t`-Variable zeigt, ein Zeiger auf `char` zugewiesen, was dieselbe Warnung hervorruft.

Trotz des anderen Zeigertyps und trotz der Warnungen zeigt `p2` auf die Speicheradresse der ersten Zahl im Array, so daß die Funktion `output()` normal arbeitet.

- (d) Erklären Sie die vierte Zeile (768 1280 1792 2816 3328 4352) der Ausgabe des Programms. Sie dürfen einen Little-Endian-Rechner voraussetzen.

Zwischen der Ausgabe der ersten und der zweiten Zeile wurde der Zeiger `p2` um 1 inkrementiert. Da `p2` auf `char`-Variable zeigt und nicht auf `uint16_t`-Variable, zeigt er danach *nicht* auf die nächste ganze Zahl im Array, sondern auf die *nächste Speicherzelle*. Da eine `uint16_t`-Variable zwei Speicherzellen belegt, zeigt `p2` nach dem Inkrementieren auf das zweite Byte der ersten Variablen im Array. Die Funktion `output()` liest immer ganze `uint16_t`-Variablen und nimmt für die Ausgabe noch das erste Byte der zweiten Zahl im Array hinzu.

Auf einem Little-Endian-Rechner hat das zweite Byte der Zahl 2 den Wert 0 und das erste Byte der Zahl 3 den Wert 3. Wenn man dies zu einer Little-Endian-16-Bit-Zahl zusammensetzt, entsteht die Zahl $256 \cdot 3 + 0 = 768$. Entsprechendes gilt für alle anderen Zahlen im Array.



Die Schleife in der Funktion `output()` bricht ab, sobald sie auf den Zahlenwert 0 trifft. Dies ist jetzt nur noch zufällig der Fall; anscheinend hat die Speicherzelle hinter dem Array zufällig den Wert 0. Ansonsten wäre es auch möglich gewesen, daß die Schleife über das Array hinaus immer weiter liest, was zu einem Absturz führen kann.

Aufgabe 3: XBM-Grafik

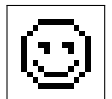
Bei einer XBM-Grafikdatei handelt es sich um ein als C-Quelltext abgespeichertes Array, das die Bildinformationen enthält:

- Jedes Bit entspricht einem Pixel.
- Nullen stehen für Weiß, Einsen für Schwarz.
- LSB first.
- Jede Zeile des Bildes wird auf ganze Bytes aufgefüllt.
- Breite und Höhe des Bildes sind als Konstantendefinitionen (**#define**) in der Datei enthalten.

Sie können eine XBM-Datei sowohl mit einem Texteditor als auch mit vielen Grafikprogrammen öffnen und bearbeiten.

Beispiel ([aufgabe-3.xbm](#)):

```
#define aufgabe_3_width 14
#define aufgabe_3_height 14
static unsigned char aufgabe_3_bits[] = {
    0x00, 0x00, 0xf0, 0x03, 0x08, 0x04, 0x04, 0x08, 0x02, 0x10, 0x32, 0x13,
    0x22, 0x12, 0x02, 0x10, 0x0a, 0x14, 0x12, 0x12, 0xe4, 0x09, 0x08, 0x04,
    0xf0, 0x03, 0x00, 0x00 };
```



Ein C-Programm, das eine XBM-Grafik nutzen will, kann die [.xbm](#)-Datei mit **#include "..."** direkt einbinden.

Schreiben Sie ein Programm, das die XBM-Datei als ASCII-Grafik ausgibt, z. B.:

```

  * * * * *
 *           *
*           *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
*   *   *   *
  * * * * *
```

(8 Punkte)

(Hinweis für die Klausur: Abgabe auf Datenträger ist erlaubt und erwünscht, aber nicht zwingend.)

Lösung

Siehe die Datei [loesung-3.c](#).

Der Ausdruck $(\text{aufgabe_3_width} + 7) / 8$ ist die auf ganze Bytes aufgerundete Breite des Bildes in Bytes. Ohne das „+ 7“ in der Klammer würde stets abgerundet; die Breite von Bildern, deren Breite kein Vielfaches von 8 ist, wäre dann immer um 1 zu klein. Dadurch daß wir $n - 1$ addieren, bevor wir durch n dividieren, machen wir aus dem Abrunden ein Aufrunden.

In jedem Durchlauf der äußeren Schleife wird der Zeiger `p` auf den Anfang der Zeile `i` des Bildes gesetzt.

Innerhalb der Zeile gehen wir mit einer Bit-Maske `mask` die Bits innerhalb des Bytes `*p` von rechts nach links durch (LSB first). Wenn das Bit aus der Maske links herausgeschoben wird, setzen wir die Maske auf `0x01` zurück und setzen den Zeiger `p` auf das nächste Byte.

(Der Datentyp **unsigned char** ist eine vorzeichenlose ganze Zahl von der Größe einer Speicherzelle und normalerweise identisch mit `uint8_t`. Da das XBM-Dateiformat den Datentyp **unsigned char** verwendet, geschieht dies auch im Programm [loesung-3.c](#); ansonsten wäre `uint8_t` eine bessere Wahl.)

(Anmerkung: In einer früheren Fassung enthielt die Datei [aufgabe-3.xbm](#) die Bezeichner `aufgabe_4_width`, `aufgabe_4_height` und `aufgabe_4_bits` anstelle von `aufgabe_3_width`, `aufgabe_3_height` und `aufgabe_3_bits`. Dieser Fehler wurde korrigiert und das Lösungsprogramm entsprechend angepaßt.)