

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 7. Januar 2019

Aufgabe 1: Speicherformate von Zahlen

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

```
1  #include <stdio.h>
2  #include <stdint.h>
3
4  typedef struct
5  {
6      uint32_t a;
7      uint64_t b;
8      uint8_t c;
9  } three_numbers;
10
11 int main (void)
12 {
13     three_numbers xyz = { 1819042120, 2410670883059281007, 0 };
14     printf ("%s\n", &xyz);
15     return 0;
16 }
```

Das Programm wird für einen 32-Bit-Rechner kompiliert und ausgeführt.

(Die `gcc`-Option `-m32` sorgt dafür, daß `gcc` Code für einen 32-Bit-Prozessor erzeugt.)

```
$ gcc -Wall -m32 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *", but
argument 2 has type "three_numbers * {aka struct <anonymous> *}" [-Wformat=]
    printf ("%s\n", &xyz);
                   ^
$ ./aufgabe-2
Hallo, Welt!
```

- (a) Erklären Sie die beim Compilieren auftretende Warnung. (2 Punkte)
- (b) Erklären Sie die Ausgabe des Programms. (4 Punkte)
- (c) Welche Endianness hat der verwendete Rechner? Wie sähe die Ausgabe auf einem Rechner mit entgegengesetzter Endianness aus? (2 Punkte)
- (d) Dasselbe Programm wird nun für einen 64-Bit-Rechner kompiliert und ausgeführt.
(Die `gcc`-Option `-m64` sorgt dafür, daß `gcc` Code für einen 64-Bit-Prozessor erzeugt.)

```
$ gcc -Wall -m64 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *",
but argument 2 has type "three_numbers * {aka struct <anonymous> *}"
[-Wformat=]
    printf ("%s\n", &xyz);
                   ^
$ ./aufgabe-2
Hall5V
```

(Es ist möglich, daß die konkrete Ausgabe auf Ihrem Rechner anders aussieht.)

Erklären Sie die geänderte Ausgabe des Programms. (3 Punkte)

Lösung

(a) Erklären Sie die beim Compilieren auftretende Warnung.

Die Funktion `printf()` mit der Formatspezifikation `%s` erwartet als Parameter einen String, d. h. einen Zeiger auf `char`. Die Adresse (`&`) der Variablen `xyz` ist zwar ein Zeiger, aber nicht auf `char`, sondern auf einen `struct` vom Typ `three_numbers`. Eine implizite Umwandlung des Zeigertyps ist zwar möglich, aber normalerweise nicht das, was man beabsichtigt.




(b) Erklären Sie die Ausgabe des Programms.

Ein String in C ist ein Array von **chars** bzw. ein Zeiger auf **char**. Da die Funktion **printf()** mit der Formatspezifikation **%s** einen String erwartet, wird sie das, worauf der übergebene Zeiger zeigt, als ein Array von **chars** interpretieren. Ein **char** entspricht einer 8-Bit-Speicherzelle. Um die Ausgabe des Programms zu erklären, müssen wir daher die Speicherung der Zahlen in den einzelnen 8-Bit-Speicherzellen betrachten.




Hierfür wandeln wir zunächst die Zahlen von Dezimal nach Hexadezimal um. Sofern nötig (hier nicht der Fall) füllen wir von links mit Nullen auf, um den gesamten von der Variablen belegten Speicherplatz zu füllen (hier: 32 Bit, 64 Bit, 8 Bit). Jeweils 2 Hex-Ziffern stehen für 8 Bit.

dezimal		hexadezimal
1 819 042 120	=	6C 6C 61 48
2 410 670 883 059 281 007	=	21 74 6C 65 57 20 2C 6F
0	=	00

Die Anordnung dieser 8-Bit-Zellen im Speicher lautet **auf einem Big-Endian-Rechner** wie folgt:

6C	6C	61	48	21	74	6C	65	57	20	2C	6F	00
												

Auf einem Little-Endian-Rechner lautet sie hingegen:

48	61	6C	6C	6F	2C	20	57	65	6C	74	21	00
												

Anhand einer ASCII-Tabelle erkennt man, daß die Big-Endian-Variante dem String "llaHtleW_o" und die Little-Endian-Variante dem String "Hallo,_Welt!" entspricht – jeweils mit einem Null-Symbol am Ende, das von der Variablen `c` herrührt.

Auf einem Little-Endian-Rechner wird daher `Hallo, Welt!` ausgegeben.

(c) Welche Endianness hat der verwendete Rechner?

Little-Endian (Begründung siehe oben)

Wie sähe die Ausgabe auf einem Rechner mit entgegengesetzter Endianness aus?

llaH!tleW ,o (Begründung siehe oben)

(d) Dasselbe Programm wird nun für einen 64-Bit-Rechner kompiliert und ausgeführt. (Die `gcc`-Option `-m64` sorgt dafür, daß `gcc` Code für einen 64-Bit-Prozessor erzeugt.)

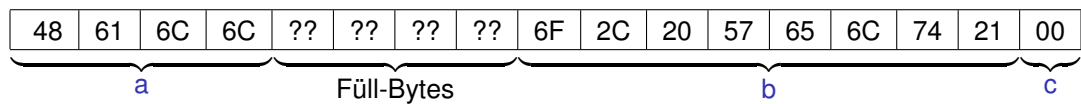
```
$ gcc -Wall -m64 aufgabe-2.c -o aufgabe-2
aufgabe-2.c: In function "main":
aufgabe-2.c:14:13: warning: format "%s" expects argument of type "char *",
but argument 2 has type "three_numbers * {aka struct <anonymous> *}"
[-Wformat=]
    printf ("%s\n", &xyz);
                ^
$ ./aufgabe-2
Hall15V
```

(Es ist möglich, daß die konkrete Ausgabe auf Ihrem Rechner anders aussieht.)

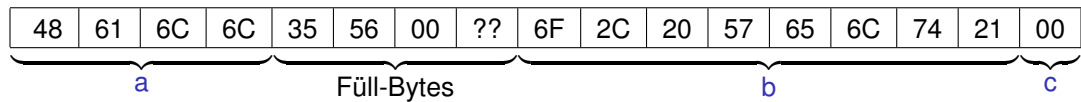
Erklären Sie die geänderte Ausgabe des Programms.

Auf einem 64-Bit-Rechner hat eine 64-Bit-Variable ein **64-Bit-Alignment**, d. h. ihre Speicheradresse muß durch 8 teilbar sein.

Der Compiler legt die Variablen daher wie folgt im Speicher ab (Little-Endian):



Der Inhalt der Füll-Bytes ist undefiniert. Im Beispiel aus der Aufgabenstellung entsteht hier die Ausgabe **5v**, was den (zufälligen) hexadezimalen Werten 35 56 entspricht:



Da danach die Aufgabe aufhört, muß an der nächsten Stelle ein Null-Symbol stehen, das das Ende des Strings anzeigt. Der Inhalt der darauf folgenden Speicherzelle bleibt unbekannt.

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 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.**

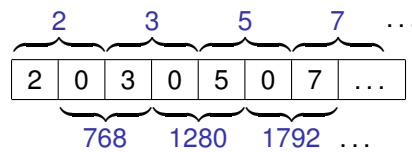
Die Warnungen kommen daher, daß die Variable `p2` ein Zeiger auf `char`-Variable ist, ihr jedoch ein anderer Zeigertyp, nämlich ein Zeiger auf `uint16_t`-Variable, zugewiesen wird. Beim Aufruf der Funktion `output()` wird umgekehrt dem Parameter `a`, der auf `uint16_t`-Variable zeigt, ein Zeiger auf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

Zusatzübung: Auf einem Big-Endian-Rechner verhält sich das Programm genauso. Versuchen Sie, auch dies zu erklären.