Fragen zur Programmiersprache C

Fragen entnommen aus SPiC [1]

1. Ein Hauptprogramm und eine Interruptbehandlung greifen nebenläufig auf die Variable `uint16_t foo` zu. Das Hauptprogramm verwendet `foo` in der Anweisung `uint16_t bar = foo;` der Interrupthandler verwendet die Variable `foo` im Vergleich `if(foo == 5)`. Welches Nebenläufigkeitsproblem kann auftreten?
   - ☐ Lost-Update
   - ☐ Lost-Wakeup
   - ☐ keines
   - ☐ Das Hauptprogramm könnte einen inkonsistenten Wert lesen, da `foo` aus 2 Bytes besteht und nicht mit einer Instruktion geladen werden kann.
2. Welche Aussage zu Zeigern ist richtig?
   - ☐ Die Speicherstelle, auf die ein Zeiger verweist, kann niemals selbst einen Zeiger enthalten.
   - ☐ Beim Rechnen mit Zeigern muss immer der Typ des Zeigers beachtet werden.
   - ☐ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
   - ☐ Zeiger vom Typ `void*` benötigen weniger Speicher als andere Zeiger, da bei anderen Zeigertypen zusätzlich die Größe gespeichert werden muss.
3. Was ist ein Stack-Frame?
   - ☐ Der Speicherbereich, in dem der Programmcode einer Funktion abgelegt ist.
   - ☐ Ein spezieller Registersatz des Prozessors zur Bearbeitung von Funktionen.
   - ☐ Ein Fehler, der bei unberechtigten Zugriffen auf den Stack-Speicher entsteht.
   - ☐ Ein Bereich des Speichers, in dem u.a. lokale automatic-Variablen einer Funktion abgelegt sind.
4. Welche Aussage zu `volatile` ist richtig?
   - ☐ Das Schlüsselwort `volatile` unterbindet Optimierungen an einer damit deklarierten Variable.
   - ☐ Das Schlüsselwort `volatile` unterbindet alle Nebenläufigkeitsprobleme.
   - ☐ Das Schlüsselwort `volatile` hat nur noch eine historische Bedeutung und ist in heutigen Programmen nicht mehr nötig.

[1] `https://sys.cs.fau.de/lehre/ss/spic/`

- ☐ Das Schlüsselwort `volatile` erlaubt dem Compiler bessere Optimierungen durchzuführen.

5. Welchen Wert hat die Variable `a` nach Ausführung der folgenden Zeilen Code:
```
uint8_t a = 3 | 16;
a &= (1 « 4);
```
   - ☐ 1
   - ☐ 3
   - ☐ 16
   - ☐ 0

6. Worin liegt der Unterschied zwischen den wie folgt in der Datei `prog.c` deklarierten Funktionen?
```
uint8_t testA(void);
static uint8_t testB(uint8_t param);
```
   - ☐ Die Funktion `testA` ist nur für Funktionen in der Datei `prog.c` aufrufbar.
   - ☐ Die Funktion `testB` wird vom Linker statisch in das Programm gebunden, `testA` wird hingegen dynamisch (zur Laufzeit) eingebunden.
   - ☐ Die Funktion `testB` ist nur für Funktionen in der Datei `prog.c` aufrufbar.
   - ☐ Die Funktion `testB` ist nur über einen Funktionszeiger aufrufbar.

7. Gegeben ist folgender Programmcode:
```
int32_t x[] = {3, 8, -13, 5, 4};
int32_t *y = &x[4];
y -= 3;
```
   Welchen Wert liefert die Dereferenzierung von `y` (also `*y`)?
   - ☐ Zur Laufzeit tritt ein Fehler auf.
   - ☐ 8
   - ☐ 4
   - ☐ 1

8. Was bewirkt folgende Funktion?
```
void func(int a, int b) {
  int c = a; a = b; b = c;
}
```
   - ☐ Bei einem Aufruf vertauscht die Funktion die Inhalte der an die formalen Parameter `a` und `b` übergebenen Variablen.
   - ☐ Da in C Funktionen mit “call by value” aufgerufen werden, erhält die Funktion Kopien der Aufrufparameter, die sie vertauscht. Beim Aufrufer hat dies allerdings keine Auswirkungen.

- ☐ Der Compiler meldet bei der Übersetzung einen Fehler, weil die Funktionsparameter nicht verändert werden dürfen.
- ☐ Die von `b` adressierte Speicherzelle enthält nach dem Aufruf die in der Variablen `a` abgelegte Speicheradresse.

9. Welchen Wert enthält die Variable a nach Ausführung der folgenden Code-Zeilen?

```
int a = 2;
a ^= a;
a = a | (1 << 2);
```

   - ☐ 0
   - ☐ 2
   - ☐ 4
   - ☐ 6

10. Welche Aussage zu folgender Funktion ist richtig?

```
int *foo() {
static int bar = 0;
bar++;
return &bar;
}
```

    - ☐ Die Funktion liefert einen Zeiger auf die lokale Variable `bar` zurück. Dies ist in C nicht zulässig und führt zu einem Übersetzungsfehler.
    - ☐ Die Variable `bar` ist über die Laufzeit der `foo()`- Funktion hinaus gültig. Daher kann der zurückgelieferte Zeiger sicher vom Aufrufer verwendet werden.
    - ☐ Die Variable `bar` enthält beim Verlassen der `foo()`- Funktion immer den Wert 1, da `bar` bei jedem Aufruf von `foo()` mit 0 initialisiert wird.
    - ☐ Beim Verlassen der Funktion `foo()` wird die automatic-Variable `bar` vom Stack entfernt und der Zeiger verliert seine Gültigkeit. Ein Zugriff durch den Aufrufer führt zu zufälligen Ergebnissen.

11. Gegeben sei folgende Enumeration: `enum SPRACHE {Deutsch, Englisch, Russisch};` Welche Aussage ist richtig?
    - ☐ Der Compiler meldet einen Fehler, weil den enum-Elementen kein Wert zugewiesen wurde.
    - ☐ Der Wert von Russisch ist 2.
    - ☐ Der Wert von Russisch ist 3.
    - ☐ Der Wert von Russisch ist unbekannt;
    - ☐ der Compiler weist zur Übersetzungszeit jedem enum-Element einen zufälligen, aber eindeutigen Wert zu

12. Praktische Programmierung in C

In einer größeren Industriesteuerungsanlage wird Eisen in mehreren Hochöfen verhüttet. Leider ist die Automatisierung noch nicht ausreichend fortgeschritten und der Abstich muss hier manuell durch die Mitarbeiter vorgenommen werden. Dies ist möglich, sobald der Schmelzpunkt von 1800°K erreicht wurde.

Um die Mitarbeiter zu benachrichtigen läuft in regelmäßigen Abständen hierzu das von Ihnen zu verfassende Programm. Es filtert mittels der zu implementierenden Funktion `filterSensors` die Namen der Öfen, welche „abstichreif“ sind und gibt die zugehörigen Namen dann mittels der (ebenfalls zu implementierenden) Funktion `printSensors` für die Stahlkocher aus.

Hinweis: Die vollständige Quelltextdatei (mit Testeingaben) kann auch auf der Übungswebsite [2] heruntergeladen werden.

sensors_print.c

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

struct dentry {
  const char *key;
  uint32_t *value;
};

static uint32_t vals[20] = {
  /* ... */
};

static struct dentry data[20] = {
  /* ... */
};

const char **filterSensors(struct dentry *data, unsigned int num, uint32_t cutoff) {
  // Eingabeparameter:
  //   - data:   Array mit num Sensorwerten und Bezeichnern
  //   - num:    Anzahl der Öfen
  //   - cutoff: Die Schmelztemperatur ab der der Kessel abstichreif ist
  // Rückgabewert:
  //   - Array der Ofennamen (struct dentry.key) derjenigen Öfen, deren Wert (struct
    dentry.value) größer
  //     als die Temperatur cutoff ist.

  /* Hier müsst ihr selbst Code vervollständigen */
}

void printSensors(const char **sensors) {
  // Eingabeparameter:
  //   - sensors: Die Namen der Öfen, welche den Mitarbeitern als abstichreif
    gemeldet werden müssen
```

[2] https://sys.cs.fau.de/extern/lehre/ss22/ezs/uebung/ezs-test/sensors.c

```
  // Funktionalität:
  //   - Ausgabe der Ofennamen auf der Kommandozeile (stdout).

  /* Hier müsst ihr selbst Code vervollständigen */
}

int main(void)
{
  const char **res = filterSensors(data, sizeof(data)/sizeof(*data), 1800);
  printSensors(res);

  /* Hier müsst ihr selbst Code vervollständigen */

  return 0;
}
```

Aufgabe:

- Implementieren Sie die Funktion `filterSensors`
- Implementieren Sie die Funktion `printSensors`
- Denken Sie daran, alle möglicherweise allokierten Ressourcen in `main` an den gekennzeichneten Stellen wieder freizugeben
- Übersetzen sie Ihr Programm

```
$  gcc sensors.c -ggdb -Wall -Wextra -pedantic -pedantic-errors
   -Werror=pedantic -o sensors

```

- Überprüfen Sie die funktionale Korrektheit (in erster Näherung) per Vergleich mit der folgenden Ausgabe

```
$ ./sensors
Ofen1
Ofen2
Ofen3
Ofen6
Ofen10
Ofen11
Ofen12
Ofen19
Ofen20
```

- Überprüfen Sie sowohl die Speicherzugriffe per Zeiger als auch die Freigabe aller allokierten Ressourcen. Eine erste gute Einschätzung kann hier das Programm valgrind liefern, es sollten in Ihrer Lösung keine Fehler detektiert werden:

```
$ valgrind --leak-check=full ./sensors
...
==....== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
...
```