1 Erkennen von undefiniertem Verhalten

Bewertete Aufgabe!
Analysieren Sie folgendes Beispielprogram und beschreiben Sie, an welchen Stellen es UB enthällt. Hinweis: Es enthällt an sechs Stellen UB, und nicht alles, was ein Fehler ist, ist sofort UB.
Den Code finden Sie in der Datei UBIdentification.c:

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

double convertTemperature(double* fahrenheit) {
    double celsius = (*fahrenheit - 32)/1.8;
}

char* formatTemperature(double value, const char* unit) {
    char result[10];
    sprintf(result, "%f degree %s", value, unit);
    return result;
}

int main() {
    void* buffer = (void*)malloc(33);
    strncpy(buffer, "enter a temperature in degree F: ", 33);
    printf("%s", buffer);

    scanf("%f", buffer);
    double celsius = convertTemperature(buffer);
    printf("the temperatures is %s\n", formatTemperature(celsius, "C"));
}
```

Geben Sie den mit Kommentaren versehenen, unmodifizierten Programmcode ab.

2 Vermeiden von undefiniertem Verhalten

Bewertete Aufgabe!
Der Programmcode für diese Aufgabe enthällt zwar kein undefiniertes Verhalten, ist aber unnötig fehleranfällig geschrieben. Schreiben Sie drei Varianten der Funktion `formatMoney()`:

1. **(Keine Bewertung, ist unter Windows nicht lösbar.)**
   Die erste Variante `formatMoney1()` verhällt sich identisch zu ‘formatMoney()‘, verwendet aber `asprintf()` um den Ergebnisstring zu generieren.

   Achtung: Hierfür ist ein GNU basiertes System nötig, auf Windows geht’s nicht.

2. Die zweite Variante `formatMoney2()` hat die Signatur

```
1 void formatMoney2(int64_t cents, FILE* stream);
```

und benutzt einen Aufruf von ‘fprintf()‘ um das Ergebnis direkt auf den ‘stream‘ auszugeben.

3. Die dritte Variante `formatMoney3()` hat wieder die Signatur

```
1 char* formatMoney3(int64_t cents);
```

Diese Variante soll nicht selbst den String zusammensetzen, sondern auf `formatMoney2()` zurückgreifen indem sie zunächst einen `FILE*` mit `open_memstream()` erzeugt, dann mit `formatMoney2()` den String in diesen Stream reinschreibt, und zum Schluss mit `fclose()` den Stream schließt und den Ergebnisstring zurückgibt.

Der vorgegebene Programmcode in seiner ganzen Hässlichkeit ist in der Datei safeCoding.c zu finden:

```
1  #include <inttypes.h>
2  #include <stdbool.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  char* formatMoney(int64_t cents) {
9      bool isNegative = cents < 0;
10     cents = (isNegative ? -cents : cents);
11     int64_t euros = cents/100;
12     cents -= 100*euros;
13
14     size_t characterCount = 0;
15     if(isNegative) characterCount++;    //space for the sign
16     for(int64_t i = euros; i; i /= 10) characterCount++;    //space for
           ↪ the full euros
17     if(!euros) characterCount++;    //space for the zero if we don't
           ↪ have full euros
18     characterCount += 3;    //space for the dot and the two cent digits
19     characterCount += strlen(" euros"); //space for unit string
20
21     char* result = malloc(characterCount + 1);  //+1 for the
           ↪ terminating null byte
22     sprintf(result, "%s%"PRId64".%02"PRId64" euros", (isNegative ? "-"
           ↪ : ""), euros, cents);
23     return result;
24 }
25
26 char* formatMoney1(int64_t cents) { /*TODO*/ }
27 void formatMoney2(int64_t cents, FILE* stream) { /*TODO*/ }
28 char* formatMoney3(int64_t cents) { /*TODO*/ }
29
30 void printMoney(uint64_t cents, char* (*formatter)(int64_t cents)) {
31     char* string = formatter(cents);
```

```
32      printf("formatMoney(%"PRId64", %p) = %s\n", cents, formatter,
            ↪ string);
33      free(string);
34  }
35
36  int main() {
37      printMoney(-196720000000000, formatMoney);
38      printMoney(-314, formatMoney);
39      printMoney(-31, formatMoney);
40      printMoney(-3, formatMoney);
41      printMoney(0, formatMoney);
42      printMoney(2, formatMoney);
43      printMoney(27, formatMoney);
44      printMoney(271, formatMoney);
45
46      //XXX Bitte auskommentieren, falls Ihr auf Windows arbeitet und
            ↪ formatMoney1 nicht implementiert.
47      printMoney(-196720000000000, formatMoney1);
48      printMoney(-314, formatMoney1);
49      printMoney(-31, formatMoney1);
50      printMoney(-3, formatMoney1);
51      printMoney(0, formatMoney1);
52      printMoney(2, formatMoney1);
53      printMoney(27, formatMoney1);
54      printMoney(271, formatMoney1);
55
56      printMoney(-196720000000000, formatMoney3);
57      printMoney(-314, formatMoney3);
58      printMoney(-31, formatMoney3);
59      printMoney(-3, formatMoney3);
60      printMoney(0, formatMoney3);
61      printMoney(2, formatMoney3);
62      printMoney(27, formatMoney3);
63      printMoney(271, formatMoney3);
64  }
```

Abzugeben ist, wie üblich, der erweiterte Programmcode.

3 Vermeiden von undefiniertem Verhalten

Das Program zu dieser Aufgabe ist eine Funktion, die Ausdrücke in inverser polnischer Notation verarbeiten kann. Dazu benötigt sie eine Stack, den Sie implementieren sollen. Die Datenstruktur `typedef struct { ... } Stack` ist bereits definiert, sowie die Signaturen der fünf Funktionen, mit denen auf den Stack zugegriffen wird.
Implementieren Sie alle fünf Stack-Funktionen, wobei Sie darauf achten sollen, dass immer ausreichend Speicher vorhanden ist. Alloziieren Sie den Speicher mit `malloc()`, und vergrößern Sie ihn bei Bedarf mit `realloc()`. Wenn Sie den Speicher vergrößern, achten Sie darauf, dass Sie jeweils die Speichergröße verdoppeln, um ein quadratisches Laufzeitverhalten zu vermeiden.
Sie finden den Programmcode, den Sie erweitern sollen, in der Datei calculator.c, die auszufüllenden Stellen sind mit `assert(0 && "TODO")`-Ausdrücken markiert.

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    size_t curCount, allocatedSize;
    double* data;
} Stack;

void Stack_construct(Stack* me) { assert(0 && "TODO"); }
bool Stack_isEmpty(Stack* me) { assert(0 && "TODO"); }
void Stack_push(Stack* me, double value) { assert(0 && "TODO"); }
double Stack_pop(Stack* me) { assert(0 && "TODO"); }
void Stack_destruct(Stack* me) { assert(0 && "TODO"); }

double calculate(char* expression) {
    Stack myStack;
    Stack_construct(&myStack);
    while(*expression) {
        switch(*expression) {
            case ' ': {
                expression++;
            } break;

            case '+':
            case '-':
            case '*':
            case '/': {
                assert(!Stack_isEmpty(&myStack));
                double argB = Stack_pop(&myStack);
                assert(!Stack_isEmpty(&myStack));
                double argA = Stack_pop(&myStack);
                Stack_push(&myStack, *expression == '+' ? argA + argB :
                                     *expression == '-' ? argA - argB :
                                     *expression == '*' ? argA * argB :
                                                          argA / argB);
                expression++;
            } break;

            default: {
                char* nextToken;
                double value = strtod(expression, &nextToken);
                assert(nextToken != expression);
                Stack_push(&myStack, value);
                expression = nextToken;
            } break;
        }
    }
    assert(!Stack_isEmpty(&myStack));
    double result = Stack_pop(&myStack);
    assert(Stack_isEmpty(&myStack));
    return result;
}
```

```
55
56 int main() {
57     printf("%f\n", calculate("1 1 1 1 + + + 1 - 1 1 1 1 + + + 1 1 + / *
       ↪ 1 1 + 1 1 + * 1 1 + * 1 - *"));
58 }
```

4 Abgabe

Abzugeben ist ein gemäß den bekannten Richtlinien erstelltes und benanntes Archiv. Das enthaltene und gewohnt benannte Verzeichnis soll folgenden Inhalt haben:

- Alle Quellen, aus denen Ihr Programm besteht (exercise08-1.c und exercise08-2.c); gut dokumentiert(Kommentare im Code!)

Achtung: Diesmal sind zwei Aufgaben abzugeben.
Senden Sie Ihre Abgabe an cp-abgabe@wr.informatik.uni-hamburg.de .