

1 Erkennen von undefiniertem Verhalten

Analysieren Sie folgendes Beispielprogramm und beschreiben Sie, an welchen Stellen es UB enthält. Hinweis: Es enthält an vier Stellen UB.

Den Code finden Sie in der Datei UBIdentification.c:

```
1 #include <stdio.h>
2
3 char* formatTemperature(double value, const char* unit) {
4     char result[10];
5     sprintf(result, "%f degree %s", value, unit);
6 }
7
8 int main() {
9     printf("enter a temperature in degree F: ");
10    double fahrenheit;
11    scanf("%lf", &fahrenheit);
12    double celsius = (fahrenheit - 32)/1.8;
13    char* formattedTemperature = formatTemperature(celsius, "C");
14    printf("the temperatures is %s\n", formattedTemperature);
15 }
```

2 Vermeiden von undefiniertem Verhalten

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

1. Die erste Variante `formatMoney1()` verhält 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%"PRIu64".%02"PRIu64" euros", (isNegative ? "-"
    ↪ : ""), euros, cents);
23    return result;
24 }
25
26 void printMoney(uint64_t cents) {
27     char* string = formatMoney(cents);
28     printf("formatMoney(%"PRIu64") = %s\n", cents, string);
29     free(string);
30 }
31
32 int main() {
33     printMoney(-1967200000000000);
34     printMoney(-314);
35     printMoney(-31);
36     printMoney(-3);
37     printMoney(0);
38     printMoney(2);
39     printMoney(27);
40     printMoney(271);
41 }
```

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. Allozieren 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.

```
1 #include <assert.h>
2 #include <stdbool.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 typedef struct {
7     size_t curCount, allocatedSize;
8     double* data;
9 } Stack;
10
11 void Stack_construct(Stack* me) { assert(0 && "TODO"); }
12 bool Stack_isEmpty(Stack* me) { assert(0 && "TODO"); }
13 void Stack_push(Stack* me, double value) { assert(0 && "TODO"); }
14 double Stack_pop(Stack* me) { assert(0 && "TODO"); }
15 void Stack_destruct(Stack* me) { assert(0 && "TODO"); }
16
17 double calculate(char* expression) {
18     Stack myStack;
19     Stack_construct(&myStack);
20     while(*expression) {
21         switch(*expression) {
22             case ' ': {
23                 expression++;
24             } break;
25
26             case '+':
27             case '-':
28             case '*':
29             case '/': {
30                 assert(!Stack_isEmpty(&myStack));
31                 double argB = Stack_pop(&myStack);
32                 assert(!Stack_isEmpty(&myStack));
33                 double argA = Stack_pop(&myStack);
34                 Stack_push(&myStack, *expression == '+' ? argA + argB :
35                             *expression == '-' ? argA - argB :
36                             *expression == '*' ? argA * argB :
37                             argA / argB);
38                 expression++;
39             } break;
40
```

```

41         default: {
42             char* nextToken;
43             double value = strtod(expression, &nextToken);
44             assert(nextToken != expression);
45             Stack_push(&myStack, value);
46             expression = nextToken;
47         } break;
48     }
49 }
50 assert(!Stack_isEmpty(&myStack));
51 double result = Stack_pop(&myStack);
52 assert(Stack_isEmpty(&myStack));
53 return result;
54 }
55
56 int main() {
57     printf("%f\n", calculate("1 1 1 1 + + + 1 - 1 1 1 1 + + + 1 1 + / *
58         ↪ 1 1 + 1 1 + * 1 1 + * 1 - *"));

```