Code wurde auf dem HTWG Container ausgeführt

14.1

Das Programm wird mit clang -g -Wall -Wpedantic -Wextra -std=gnu17 erfolgreich kompiliert. Wenn man es allerdings ausführt wird ein Segmentation Fault Error geworfen. Was nicht verwunderlich ist, da man ja einen „leeren“ pointer dereferenziert. Die Dereferenziert Variable kann nun mal mit einer „leeren“ Adresse nicht viel anfangen.

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

int main() {
    int n;
    int *p = (int *) NULL;
    n = *p;
    return EXIT_SUCCESS;
}

```

14.2

Wie zu erwarten zeigt der Debugger, den gleichen Fehler nur mit etwas mehr Information. Er gibt z.B. die Stelle im Code an, wodurch der Fehler zum Vorschein kommt. Man bemerke allerdings, dass das nicht die Stelle ist wo dem pointer NULL zugewiesen wird, sondern erst die Stelle, wo es dereferenziert wird. Auch gibt er eine Adresse, wahrscheinlich die physikalische, im Speicher, wo das Programm wahrscheinlich untergebracht wurde, an.

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401129 in main () at null.c:6
6       n = *p;
```

14.3

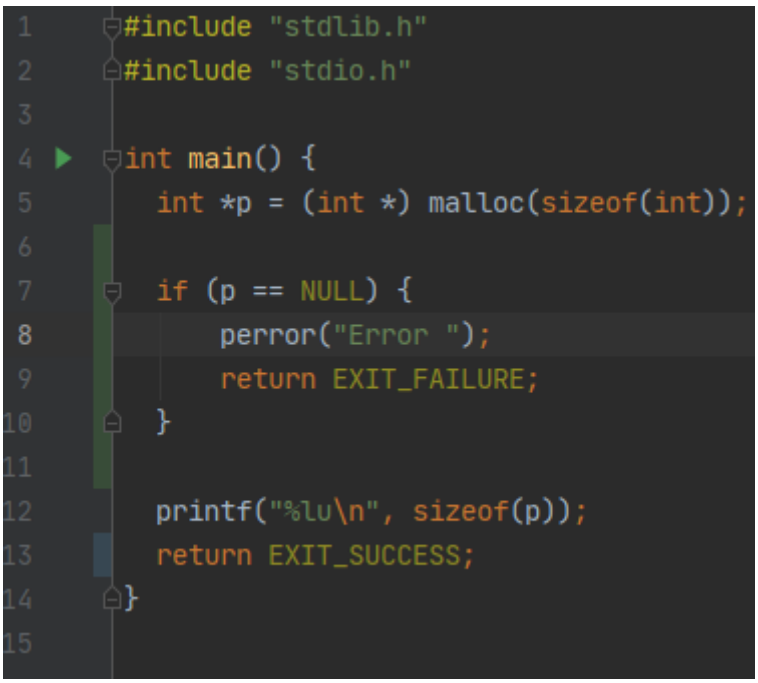
Auch Valgrind gibt logischer weise auch einen SIGSSEGV oder Segmentation fault zurück. Allerdings sagt er, dass es daran liegt, dass es eine invalid read of Size 4 gibt. Dies wird wahrscheinlich aus dem Grund hervorgerufen, da wir einen INT pointer haben, die normalerweise eine 4 Byte Größe haben, aber eine NULL Byte Größe zurückkommt. Auch sagt er das im HEAP nichts allokiert wurde, was normalerweise der Fall sein sollte, wenn der Pointer richtig erstellt würde.

Die Option –leak-check=yes schaltet das detaillierte Memory leak detector ein. Dieser verlangsamt das Programm und führt dazu das mehr Speicher verbraucht wird, aber er gibt auch mehr Fehler und Meldung bezgl. Speicherfehler und Speicherleaks aus.

```
Invalid read of size 4
   at 0x401129: main (null.c:6)
 Address 0x0 is not stack'd, malloc'd or (recently) free'd


Process terminating with default action of signal 11 (SIGSEGV)
 Access not within mapped region at address 0x0
   at 0x401129: main (null.c:6)
```

14.4

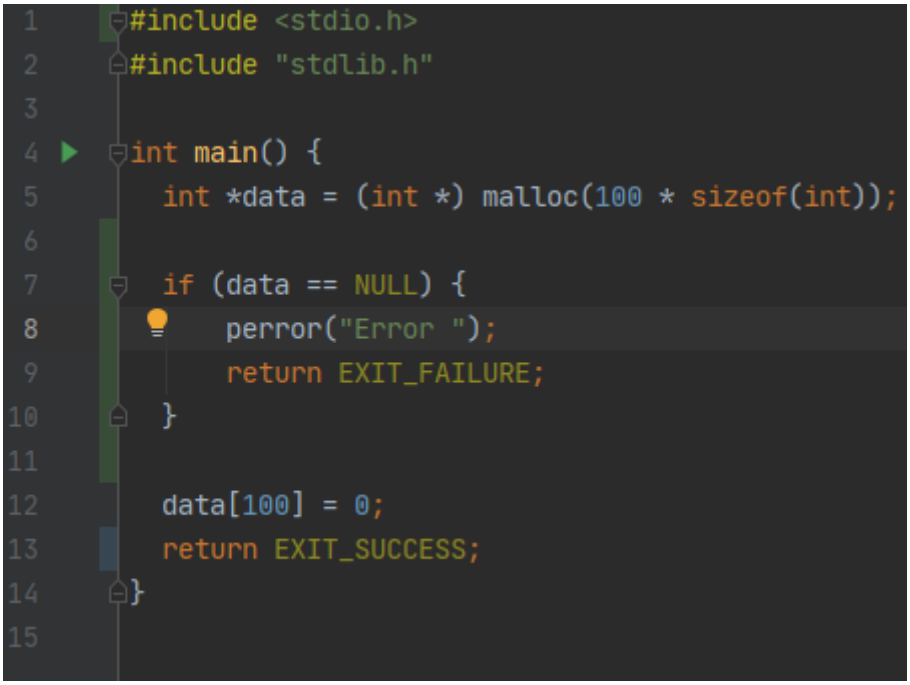
Wenn das Programm ausgeführt wird, wird kein Fehler geworfen. Auch mit GDB kann kein Fehler gefunden werden, da der Code an sich richtig ist. Wenn man dann allerdings mit Valgrind mit der leak-check=yes Flag drüber geht, kann man natürlich feststellen das 4 Bytes verloren gehen. Diese 4 Bytes kommen von den nicht befreiten INT pointer, den wir hier wieder benutzen. Warum die anderen beiden „verfahren“ keinen Fehler werfen kommt daher, dass sobald das Programm beendet ist, der Speicher eh wieder freigestellt wird. Es ist in diesem Fall also nur eine Code Unschönheit.



```c
#include "stdlib.h"
#include "stdio.h"

int main() {
    int *p = (int *) malloc(sizeof(int));

    if (p == NULL) {
        perror("Error ");
        return EXIT_FAILURE;
    }

    printf("%lu\n", sizeof(p));
    return EXIT_SUCCESS;
}

```

14.5

Wie vorher lässt sich das Programm normal per Command Line sowie unter GDB ohne Fehler ausführen. Wenn man nun wieder Valgrind mit der gleicher Flag ausführt bekommt man 2 Fehler. Zum einer wieder der gleiche wie oben. Also das 400 Bytes verloren gehen, da kein Free gemacht wird. Dieser war zu erwarten. Zum anderen wird ein invalid write of 4 Bytes geworfen. Dies dürfte dadurch versucht sein da wird data [100] = 0 machen. Denn unser Array ist zwar 100 Elemente groß doch der höchste Index dürfte 99 sein, da Arrays bei 0 starten.

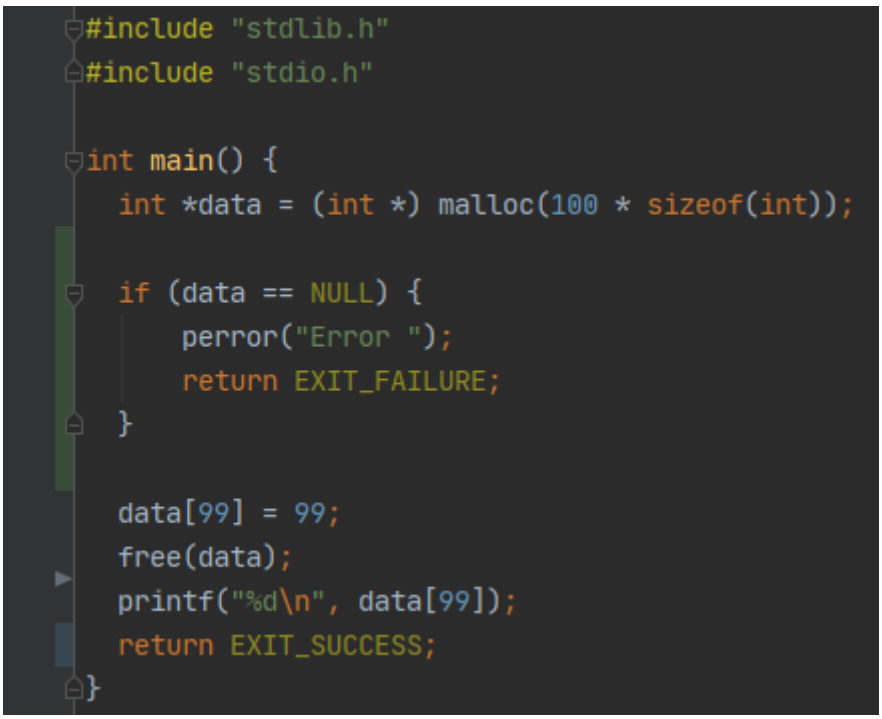


```c
#include <stdio.h>
#include "stdlib.h"

int main() {
    int *data = (int *) malloc(100 * sizeof(int));

    if (data == NULL) {
        perror("Error ");
        return EXIT_FAILURE;
    }

    data[100] = 0;
    return EXIT_SUCCESS;
}
```

```
Invalid write of size 4
   at 0x401155: main (array.c:5)
 Address 0x4a181d0 is 0 bytes after a block of size 400 alloc'd
   at 0x483577F: malloc (vg_replace_malloc.c:299)
   by 0x40114A: main (array.c:4)


HEAP SUMMARY:
    in use at exit: 400 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 400 bytes allocated
```

14.6

Wie vorher lässt sich das Programm normal per Command Line sowie unter GDB ohne Fehler ausführen. Was eigentlich unterwartet war denn die Werte des Arrays sollten eigentlich nicht mehr vorhanden sein da sie vorher gefreet werden. Dies spiegelt sich auch in dem Valgrind Ergebnis wider. Diesmal haben wir durch den Free immerhin keine Bytes lost allerdings haben wir einen invalid read. Den wie vorher schon gesagt wird vor dem Print der Array gefreet, allerdings wird trotzdem der richtige Wert geprintet.



```c
#include "stdlib.h"
#include "stdio.h"

int main() {
    int *data = (int *) malloc(100 * sizeof(int));

    if (data == NULL) {
        perror("Error ");
        return EXIT_FAILURE;
    }

    data[99] = 99;
    free(data);
    printf("%d\n", data[99]);
    return EXIT_SUCCESS;
}
```

```
Invalid read of size 4
   at 0x40118D: main (freearray.c:10)
 Address 0x4a181cc is 396 bytes inside a block of size 400 free'd
   at 0x48369AB: free (vg_replace_malloc.c:530)
   by 0x401188: main (freearray.c:7)
 Block was alloc'd at
   at 0x483577F: malloc (vg_replace_malloc.c:299)
   by 0x40116A: main (freearray.c:5)
```

14.7

In unserem Fall haben wir wie im Buch beschrieben einen Pointer gefreed der auf eine bestimmte Stelle von unserem Array zeigt. Kompiliert wird das Ganze noch, allerdings sobald man das Programm ausführt wird ein invalid pointer bei free () geworfen. Laut Manpage verhält sich free () ungewiss, wenn man fehlerhafte Werte übergibt, daher ist es gut, dass das Programm durch sowas aborted wird.

```c
#include "stdlib.h"
#include "stdio.h"

int main() {
    int *data = (int *) malloc(100 * sizeof(int));

    if (data == NULL) {
        perror("Error ");
        return EXIT_FAILURE;
    }

    data[99] = 99;
    //free(data);
    int *funnyPointer = &data[99];
    free(funnyPointer);
    printf("%d\n", data[99]);
    return EXIT_SUCCESS;
}
```

14.8

Die Aufgabe wurde so verstanden, dass ein Programm geschrieben werden soll, dass einen Paramter übergeben bekommt um so viel ein Array reallociert wird. Dann sollte die Performance gemessen werden. Dies haben wir mit dem Aufruf valgrind –tool = callgrind. /vector 2 gemacht. Laut Internet soll mit callgrind die Performance messbar sein. Dieser Aufruf generiert ein out.txt File die man mithilfe von callgrind_annotate –auto=yes filename auslesen kann. Hier bekommt man dann unter anderem die Info wie viele Assembler Befehle ausgeführt werden mussten. Das waren bei uns: 204238 aufrufe. Das Ganze listet auch wie viele aufrufe von welcher Zeile gemacht wurden bzw. wie viele Assembler aufrufe gemacht werden. Um nun den Vergleich zu einer LinkedList zu ziehen, denken wir, dass eine LinkedList wahrscheinlich mehr Assembler befehle benötigt, da diese ja aus N Nodes besteht und diese immer die daten und einen Pointer enthält.

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


int main(int argc, char *argv[]) {
    struct vec {
        int arrSize;
        int arr[];
    };

    struct vec *vecArr;
    vecArr = malloc(sizeof(struct vec));

    if (vecArr == NULL) {
        perror("Error ");
        return EXIT_FAILURE;
    }

    if (argc != 2) {
        fprintf(stderr, "Parameter missing or too many");
        return EXIT_FAILURE;
    }

    int amountOfElements = atoi(argv[1]);

    if (amountOfElements < 0) {
        fprintf(stderr, "Parameter needs to be >= 0");
        return EXIT_FAILURE;
    }

    vecArr->arrSize = vecArr->arrSize + amountOfElements;
    vecArr = realloc(vecArr, amountOfElements * sizeof(int));

    if (vecArr == NULL) {
        perror("Error");
        return EXIT_FAILURE;
    }

    for (int i = 0; i < vecArr->arrSize; i++) {
        printf("%u\n", vecArr->arr[i]);
    }
    free(vecArr);
    return EXIT_SUCCESS;
}
```