Report 1

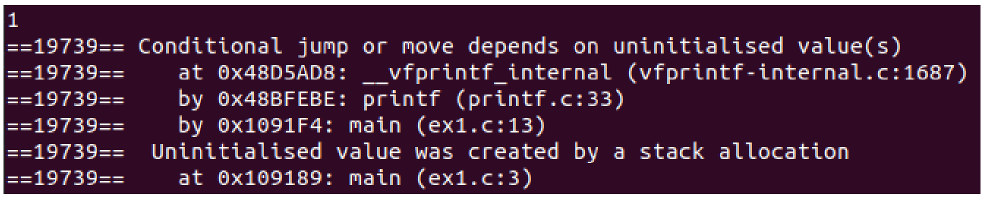
Sofya Aksenyuk, 150284

Example 1.

Executed: 2 Result: 3 4 5 6 7 8 9 10 11 12

Executed: 1 Result: -16777151 -16777150 -16777149 -16777148 -16777147 -16777146 -16777145 -16777144 -16777143 -16777142

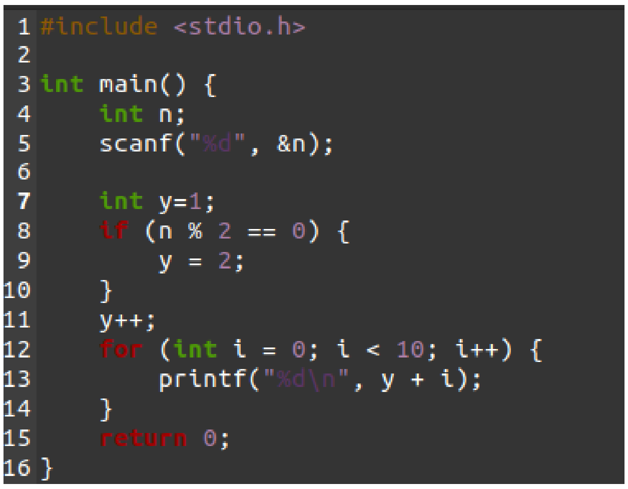
The example of an error running a given unchanged code for the input equal to 1 under Valgrind:



```
1
==19739== Conditional jump or move depends on uninitialised value(s)
==19739==    at 0x48D5AD8: __vfprintf_internal (vfprintf-internal.c:1687)
==19739==    by 0x48BFEBE: printf (printf.c:33)
==19739==    by 0x1091F4: main (ex1.c:13)
==19739==  Uninitialised value was created by a stack allocation
==19739==    at 0x109189: main (ex1.c:3)
```

After running unchanged code for odd input, all the errors showed by Valgrind were caused by uninitialized value of variable y that leads to conditional jump right to the 13th line (i.e. print statement). In other words, if condition written in the 8th line is not met, then variable y keeps being uninitialized and therefore causes an error in print statement where y is required.

The example of code that works correctly:



```
 1 #include <stdio.h>
 2 
 3 int main() {
 4     int n;
 5     scanf("%d", &n);
 6 
 7     int y=1;
 8     if (n % 2 == 0) {
 9         y = 2;
10     }
11     y++;
12     for (int i = 0; i < 10; i++) {
13         printf("%d\n", y + i);
14     }
15     return 0;
16 }
```

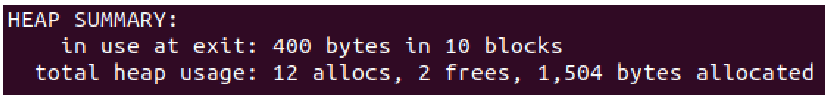
Therefore, to fix the code it was necessary to initialize y variable equal to 1 in the 7th line to make it readable for an odd input as well.

Executed: 2 Result: 3 4 5 6 7 8 9 10 11 12

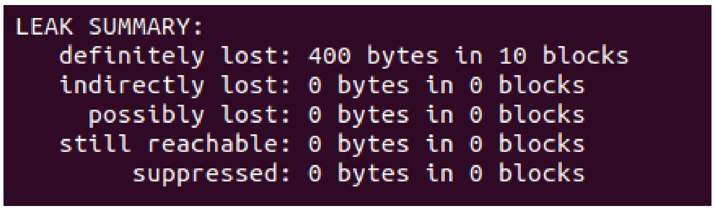
Executed: 1 Result: 2 3 4 5 6 7 8 9 10 11

Example 2.

The example of an error running a given unchanged code for the input equal to 10 (Executed: 10 Result) under valgrind –leak-check=full main main.c:



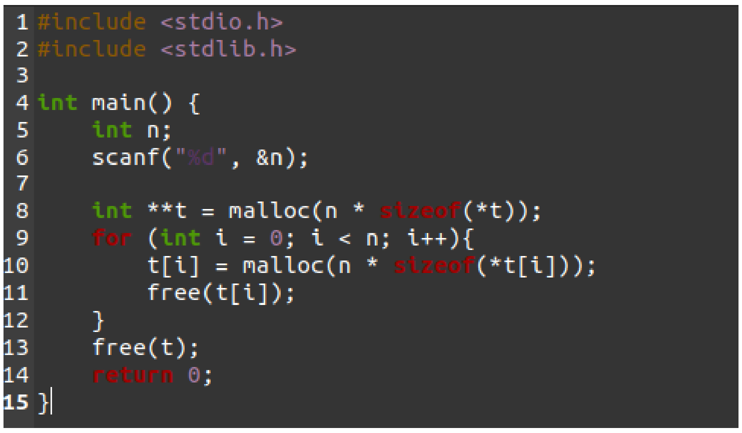
```
HEAP SUMMARY:
    in use at exit: 400 bytes in 10 blocks
  total heap usage: 12 allocs, 2 frees, 1,504 bytes allocated
```



```
LEAK SUMMARY:
   definitely lost: 400 bytes in 10 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

It can be noticed that a great amount of bytes were lost (i.e. not all heap blocks were freed).

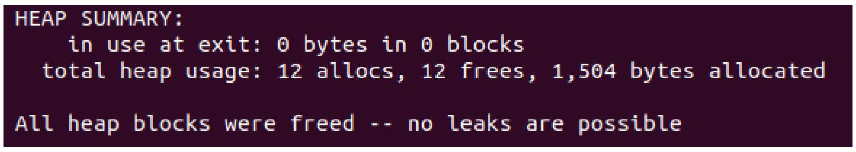
The example of code that works correctly:



```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 int main() {
 5     int n;
 6     scanf("%d", &n);
 7 
 8     int **t = malloc(n * sizeof(*t));
 9     for (int i = 0; i < n; i++){
10         t[i] = malloc(n * sizeof(*t[i]));
11         free(t[i]);
12     }
13     free(t);
14     return 0;
15 }
```

To fix that, it is enough to free each allocated element of the matrix during the work of for statement simply because it is already allocated and there are no reasons to keep that data in the memory. Apart from that, it is obligatory to enclose in braces for statement operations according to syntaxis of C language. Otherwise, for statement operations would not be readable at all that would lead to the undeclared i variable in the 10th line.

Executed: 10   Result:



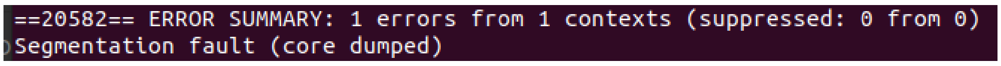
```
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 12 allocs, 12 frees, 1,504 bytes allocated

All heap blocks were freed -- no leaks are possible
```

Example 3.

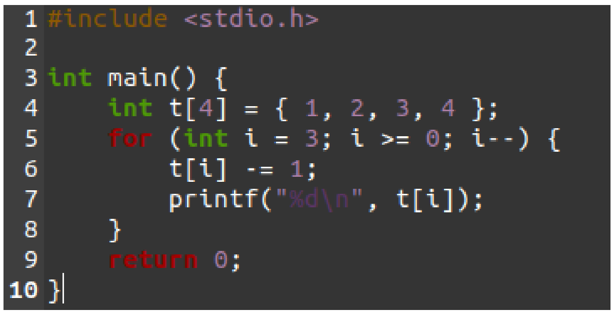
Result: 3 2 1 0 Bus error (core dumped)

The example of an error running a given unchanged code:



```
==20582== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

The example of code that works correctly:



```
 1 #include <stdio.h>
 2 
 3 int main() {
 4     int t[4] = { 1, 2, 3, 4 };
 5     for (int i = 3; i >= 0; i--) {
 6         t[i] -= 1;
 7         printf("%d\n", t[i]);
 8     }
 9     return 0;
10 }
```

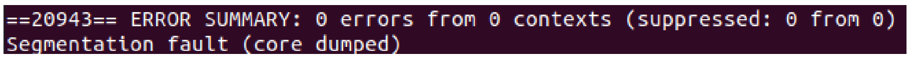
Error was caused by the "unsigned" statement. We cannot use "unsigned" type for “int i” in the 5th line, because "-1" that gets executed from the last iteration is not positive. Program faces the type of data that is not initially supposed to appear, data that does not belong to the specified “unsigned int” type which causes so-called core dump. Therefore, it is possible to fix the program by changing the type of variable i to int in the 5th line.

Result: 3 2 1 0

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

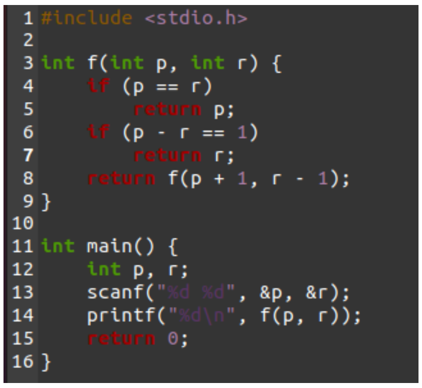
Example 4.

The example of an error running a given unchanged code for the input equal to “0 5” (Executed: 0 5 Result):



```
==20943== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

The example of code that works correctly:



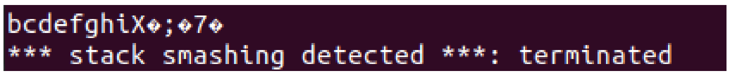
```
 1 #include <stdio.h>
 2 
 3 int f(int p, int r) {
 4     if (p == r)
 5         return p;
 6     if (p - r == 1)
 7         return r;
 8     return f(p + 1, r - 1);
 9 }
10 
11 int main() {
12     int p, r;
13     scanf("%d %d", &p, &r);
14     printf("%d\n", f(p, r));
15     return 0;
16 }
```

Executing an odd input, errors appeared because of impossibility of reaching “(p == r)” if-condition in the 4th line that led to an infinite running of the cycle (=> core dumped). To fix that we need to add additional condition for such situations when an input numbers would never be equal to each other (as it is presented in the 6th and 7th lines in the code above).

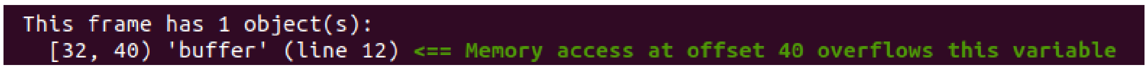
Executed: 0 5 Result: 2

Example 5.

The example of an error running a given unchanged code (Result) under Valgrind:

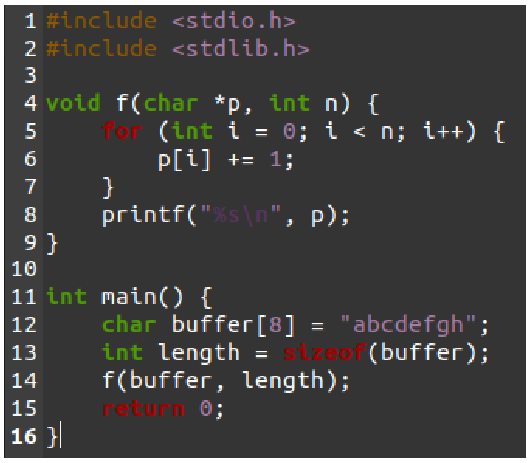


```
bcdefghiX�;�7�
*** stack smashing detected ***: terminated
```



```
 This frame has 1 object(s):
   [32, 40) 'buffer' (line 12) <== Memory access at offset 40 overflows this variable
```

The example of code that works correctly:



```c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 void f(char *p, int n) {
 5     for (int i = 0; i < n; i++) {
 6         p[i] += 1;
 7     }
 8     printf("%s\n", p);
 9 }
10 
11 int main() {
12     char buffer[8] = "abcdefgh";
13     int length = sizeof(buffer);
14     f(buffer, length);
15     return 0;
16 }
```

The problem was in the 14th line that was changed from “length + 1” to “length”. That caused as error because of overflowed buffer. The initial buffer sequence consists of 8 characters so that if we would use “length + 1” value of variable n in void function as it was in the unchanged code, the program would interpret n equal to 9 (8+1). Thus, that would exceed initial size of buffer. Thereby, to fix this we simply need to get rid of “+ 1” as that was demonstrated in the first sentence (14th line of the code above).

Result: bcdefghi