Question 2

Next, compile this program with symbol information included (with the -g flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the de- bugger by typing gdb null and then, once gdb is running, typing run. What does gdb show you?

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555161 in main () at null.c:6
6       int v = *ptr; //dereference the pointer
```

Question 3

Finally, use the valgrind tool on this program. We'll use the memcheck tool that is a part of valgrind to analyse what happens. Run this by typing in the following:
valgrind --leak-check=yes null
What happens when you run this? Can you interpret the output from the tool?

```
==2940153== Invalid read of size 4
==2940153==    at 0x109161: main (null.c:6)
==2940153==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2940153==
==2940153==
==2940153== Process terminating with default action of signal 11 (SIGSEGV)
==2940153==  Access not within mapped region at address 0x0
==2940153==    at 0x109161: main (null.c:6)
==2940153==  If you believe this happened as a result of a stack
==2940153==  overflow in your program's main thread (unlikely but
==2940153==  possible), you can try to increase the size of the
==2940153==  main thread stack using the --main-stacksize= flag.
==2940153==  The main thread stack size used in this run was 8388608.
==2940153==
==2940153== HEAP SUMMARY:
==2940153==       in use at exit: 0 bytes in 0 blocks
==2940153==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2940153==
==2940153== All heap blocks were freed -- no leaks are possible
==2940153==
==2940153== For lists of detected and suppressed errors, rerun with: -s
==2940153== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault
```

Even though we don't have any leaks (since we never allocated any memory and never needed to) We have a segmentation fault because the dereferencing is trying to read the data from Address NULL which prohibited by the OS.

Question 4

Write a simple program that allocates memory using malloc() but forgets to free it before exiting. What happens when this program runs? Can you use gdb to find any problems with it? How about valgrind (again with the --leak-check=yes flag)?

Gdb can't find any issues and states: [Inferior 1 (process 2941681) exited normally]

```
==2941862== HEAP SUMMARY:
==2941862==     in use at exit: 4 bytes in 1 blocks
==2941862==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==2941862==
==2941862== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2941862==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2941862==    by 0x10917E: main (never-free.c:5)
==2941862==
==2941862== LEAK SUMMARY:
==2941862==    definitely lost: 4 bytes in 1 blocks
==2941862==    indirectly lost: 0 bytes in 0 blocks
==2941862==      possibly lost: 0 bytes in 0 blocks
==2941862==    still reachable: 0 bytes in 0 blocks
==2941862==         suppressed: 0 bytes in 0 blocks
==2941862==
==2941862== For lists of detected and suppressed errors, rerun with: -s
==2941862== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind finds the lost memory.
(This is actually not a big deal in this program. Since the OS frees everything the program used. Since the job is doing so little, the lost memory does not really matter)

Question 5

Write a program that creates an array of integers called data of size 100 using malloc; then, set data[100] to zero. What happens when you run this program? What happens when you run this program using valgrind? Is the program correct?

```
==2944129== Invalid write of size 4
==2944129==    at 0x10918D: main (zero-array.c:5)
==2944129==  Address 0x4a911d0 is 0 bytes after a block of size 400 alloc'd
==2944129==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2944129==    by 0x10917E: main (zero-array.c:4)
==2944129==
==2944129==
==2944129== HEAP SUMMARY:
==2944129==     in use at exit: 0 bytes in 0 blocks
==2944129==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==2944129==
==2944129== All heap blocks were freed -- no leaks are possible
==2944129==
==2944129== For lists of detected and suppressed errors, rerun with: -s
==2944129== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Even though we used free() we did not initialise all values of the array. This is technically not wrong, but can give in surprising results. Reading from uninitialised memory is not predictable since it could have any value still assigned to it. Thus valgrind is telling us there is an error.

Question 6

Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use valgrind on it?

The Programm runs! (Yay!) But the data printed out is not necessarily correct anymore since the memory is freed and could
be already allocated again. Valgrind now has 2 problems:
1. Some uninitialises values
2. Reading from a not allocated memory.

Question 7

Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?

```
free-funny.c: In function ‘main’:
free-funny.c:7:5: warning: ‘free’ called on pointer ‘array’ with nonzero offset 80 [-Wfree-nonheap-
object]
    7 |     free(array);
      |     ^~~~~~~~~~~
free-funny.c:5:18: note: returned from ‘malloc’
    5 |     int *array = malloc(100* sizeof(int));
      |                  ^~~~~~~~~~~~~~~~~~~~~~~~
free(): invalid pointer
Aborted
```

The code does not run.

Question 8

Try out some of the other interfaces to memory allocation. For ex- ample, create a simple vector-like data structure and related routines that use realloc() to manage the vector. Use an array to store the vectors elements; when a user adds an entry to the vector, use realloc() to allocate more space for it. How well does such a vector perform? How does it compare to a linked list? Use valgrind to help

you find bugs.