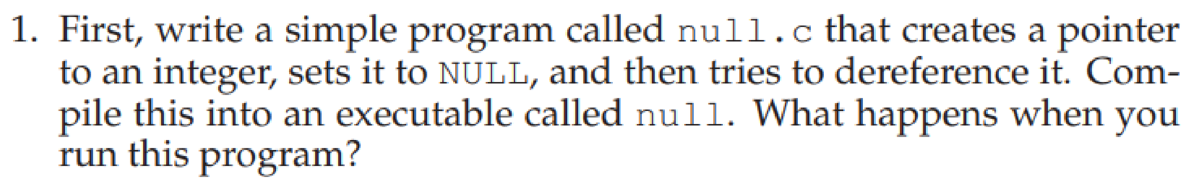
【OS】Day17(2)

【Ch14】Memory API Homework

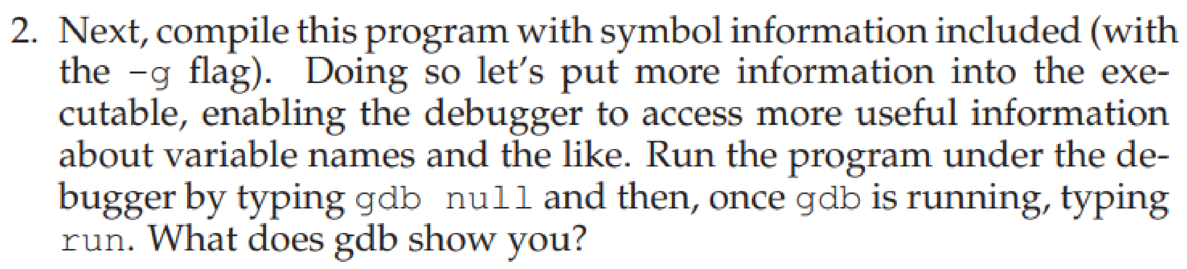
Question 1



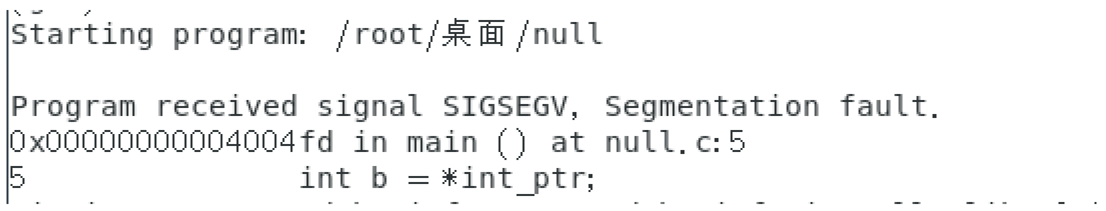
1. First, write a simple program called `null.c` that creates a pointer to an integer, sets it to `NULL`, and then tries to dereference it. Compile this into an executable called `null`. What happens when you run this program?

A segmentation fault is reported because we did not allocate memory space for the pointer and then dereferences it.

Question 2



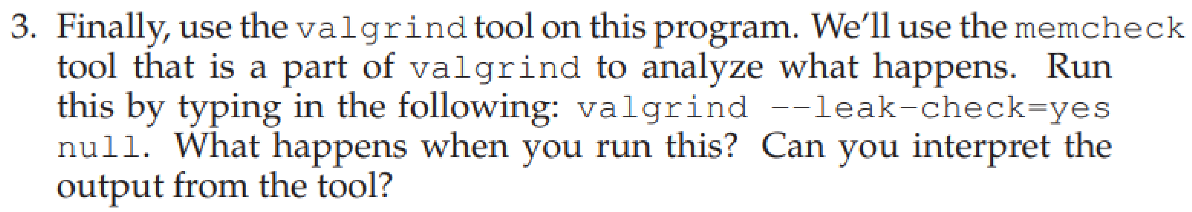
2. Next, compile this program with symbol information included (with the `-g` flag). Doing so let's put more information into the executable, enabling the debugger to access more useful information about variable names and the like. Run the program under the debugger by typing `gdb null` and then, once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



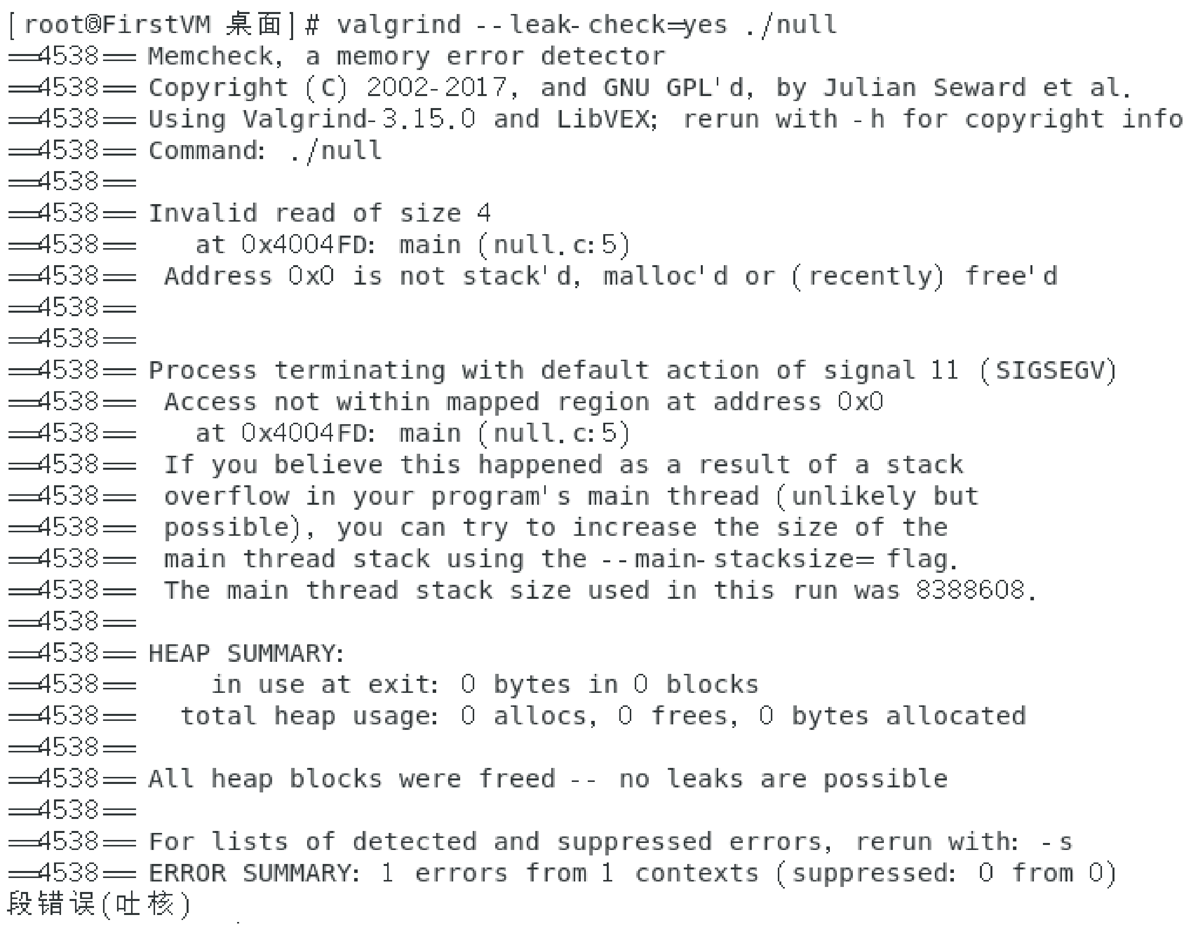
```
Starting program: /root/桌面/null

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004fd in main () at null.c:5
5                  int b = *int_ptr;
```

- Signal: SIGSEGV
- Value: 11
- Comment: Invalid Memory Dereference

Question 3

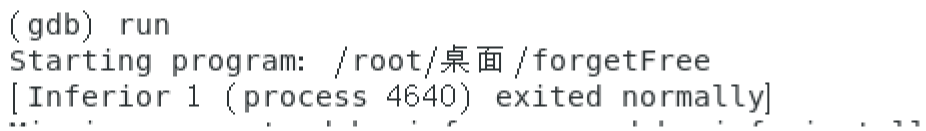


3. Finally, use the `valgrind` tool on this program. We'll use the `memcheck` tool that is a part of `valgrind` to analyze what happens. Run this by typing in the following: `valgrind --leak-check=yes null`. What happens when you run this? Can you interpret the output from the tool?



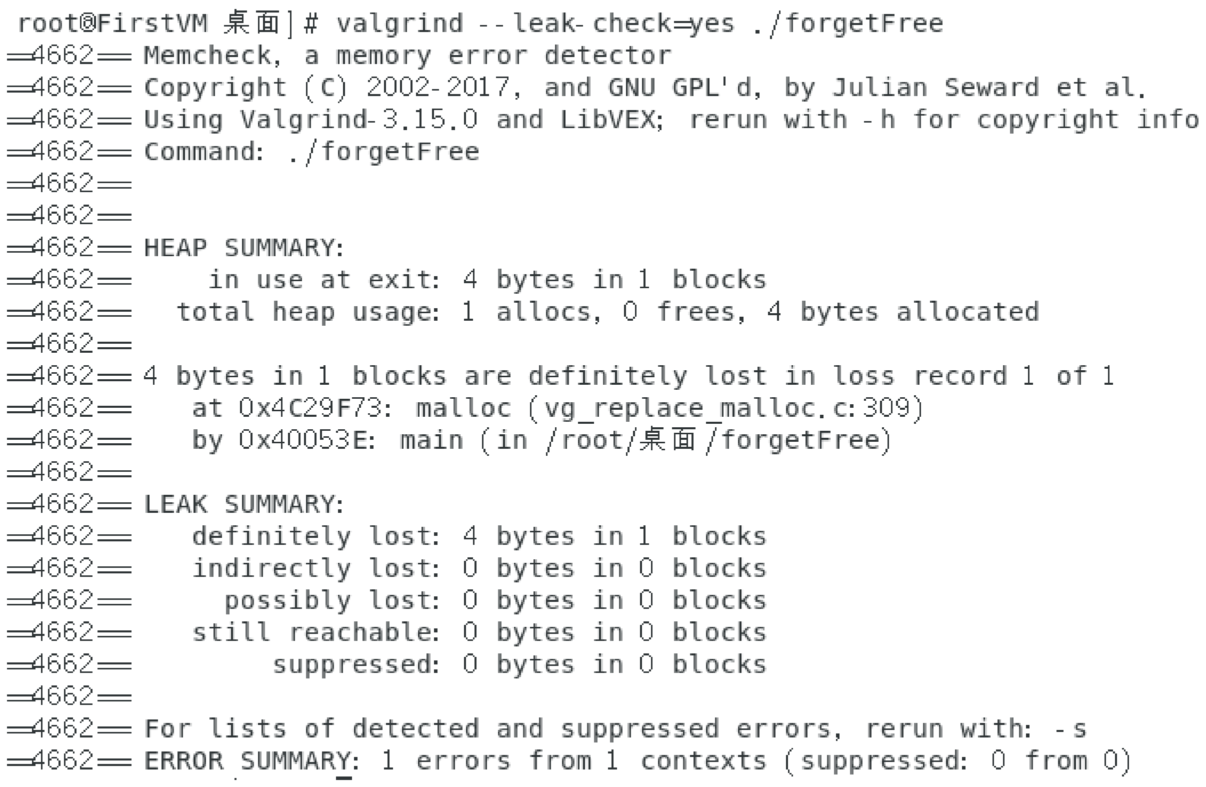
```
[root@FirstVM 桌面]# valgrind --leak-check=yes ./null
==4538== Memcheck, a memory error detector
==4538== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4538== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4538== Command: ./null
==4538==
==4538== Invalid read of size 4
==4538==    at 0x4004FD: main (null.c:5)
==4538==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==4538==
==4538==
==4538== Process terminating with default action of signal 11 (SIGSEGV)
==4538==  Access not within mapped region at address 0x0
==4538==    at 0x4004FD: main (null.c:5)
==4538==  If you believe this happened as a result of a stack
==4538==  overflow in your program's main thread (unlikely but
==4538==  possible), you can try to increase the size of the
==4538==  main thread stack using the --main-stacksize= flag.
==4538==  The main thread stack size used in this run was 8388608.
==4538==
==4538== HEAP SUMMARY:
==4538==     in use at exit: 0 bytes in 0 blocks
==4538==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==4538==
==4538== All heap blocks were freed -- no leaks are possible
==4538==
==4538== For lists of detected and suppressed errors, rerun with: -s
==4538== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
段错误(吐核)
```

```
Linux> gdb forgetFree
```



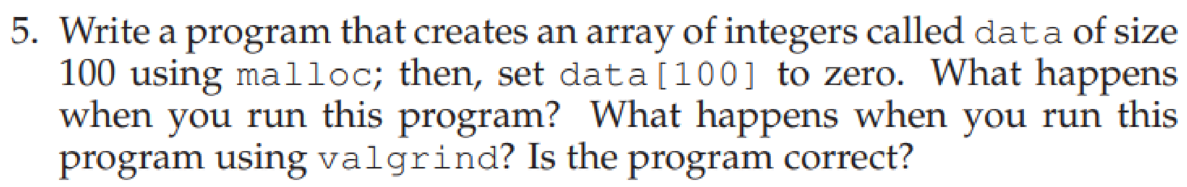
```
(gdb) run
Starting program: /root/桌面/forgetFree
[Inferior 1 (process 4640) exited normally]
```

```
Linux> valgrind --leak-check=yes forgetFree
```



```
 root@FirstVM 桌面]# valgrind --leak-check=yes ./forgetFree
==4662== Memcheck, a memory error detector
==4662== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4662== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4662== Command: ./forgetFree
==4662==
==4662==
==4662== HEAP SUMMARY:
==4662==     in use at exit: 4 bytes in 1 blocks
==4662==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==4662==
==4662== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4662==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4662==    by 0x40053E: main (in /root/桌面/forgetFree)
==4662==
==4662== LEAK SUMMARY:
==4662==    definitely lost: 4 bytes in 1 blocks
==4662==    indirectly lost: 0 bytes in 0 blocks
==4662==      possibly lost: 0 bytes in 0 blocks
==4662==    still reachable: 0 bytes in 0 blocks
==4662==         suppressed: 0 bytes in 0 blocks
==4662==
==4662== For lists of detected and suppressed errors, rerun with: -s
==4662== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

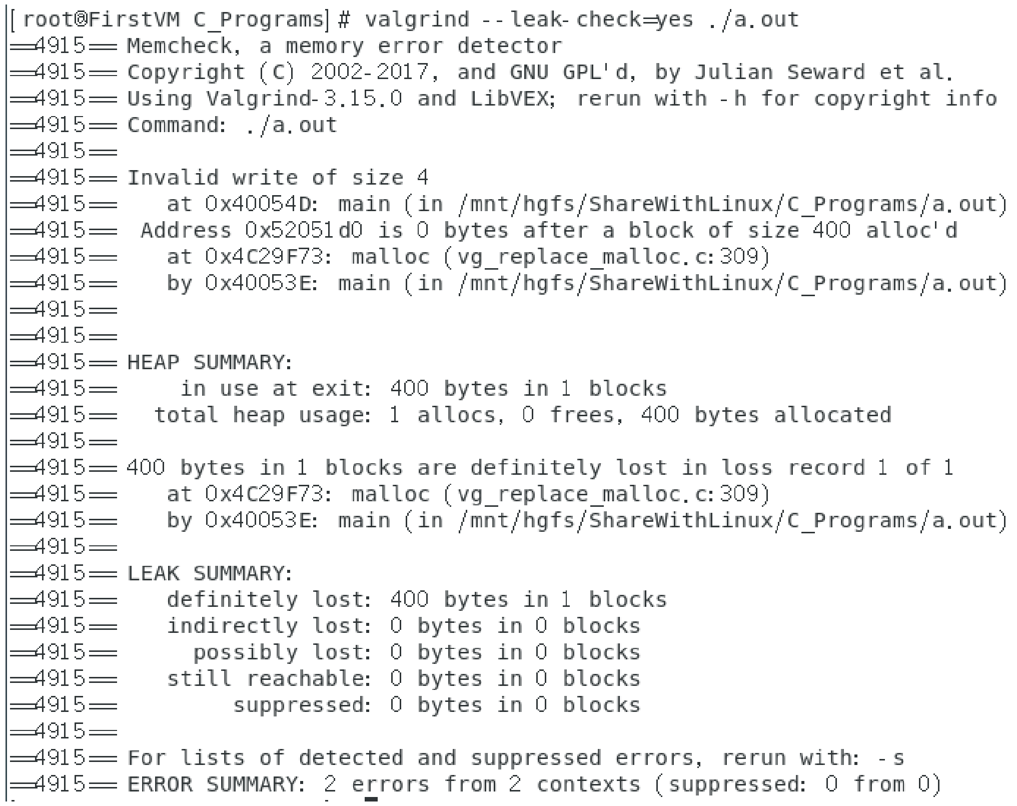
Question 5



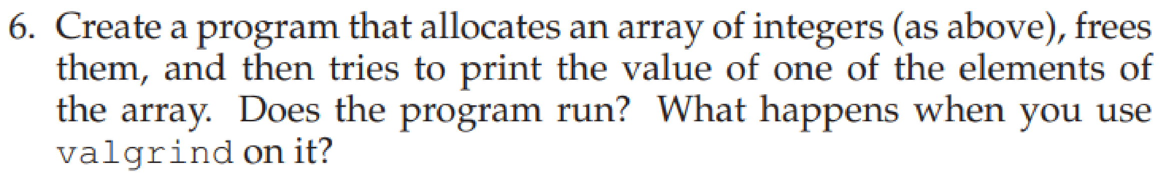
5. Write a program that creates an array of integers called `data` of size 100 using `malloc`; then, set `data[100]` to zero. What happens when you run this program? What happens when you run this program using `valgrind`? Is the program correct?

Running `forgetFree` works is fine.

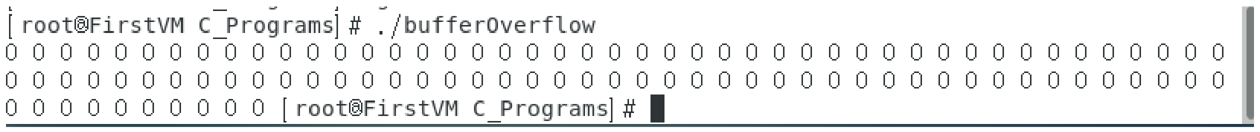
However, 400 bytes of memory are lost and an invalid write occurs.


```
[root@FirstVM C_Programs]# valgrind --leak-check=yes ./a.out
==4915== Memcheck, a memory error detector
==4915== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4915== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4915== Command: ./a.out
==4915==
==4915== Invalid write of size 4
==4915==    at 0x40054D: main (in /mnt/hgfs/ShareWithLinux/C_Programs/a.out)
==4915==  Address 0x52051d0 is 0 bytes after a block of size 400 alloc'd
==4915==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4915==    by 0x40053E: main (in /mnt/hgfs/ShareWithLinux/C_Programs/a.out)
==4915==
==4915==
==4915== HEAP SUMMARY:
==4915==     in use at exit: 400 bytes in 1 blocks
==4915==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==4915==
==4915== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4915==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==4915==    by 0x40053E: main (in /mnt/hgfs/ShareWithLinux/C_Programs/a.out)
==4915==
==4915== LEAK SUMMARY:
==4915==    definitely lost: 400 bytes in 1 blocks
==4915==    indirectly lost: 0 bytes in 0 blocks
==4915==      possibly lost: 0 bytes in 0 blocks
==4915==    still reachable: 0 bytes in 0 blocks
==4915==         suppressed: 0 bytes in 0 blocks
==4915==
==4915== For lists of detected and suppressed errors, rerun with: -s
==4915== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Question 6

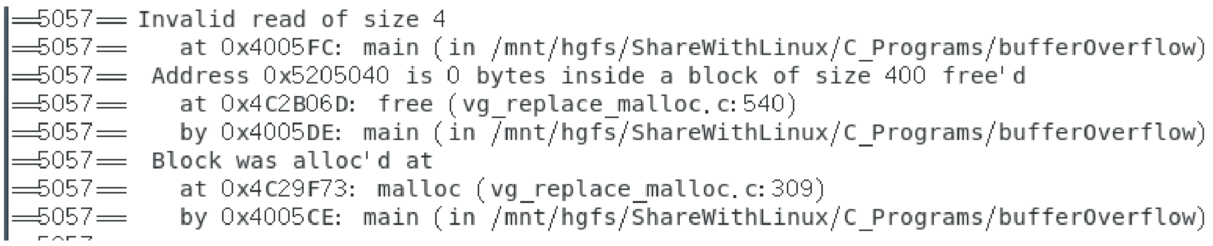

6. Create a program that allocates an array of integers (as above), frees them, and then tries to print the value of one of the elements of the array. Does the program run? What happens when you use `valgrind` on it?

Running the program:


```
[root@FirstVM C_Programs]# ./bufferOverflow
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 [root@FirstVM C_Programs]#
```

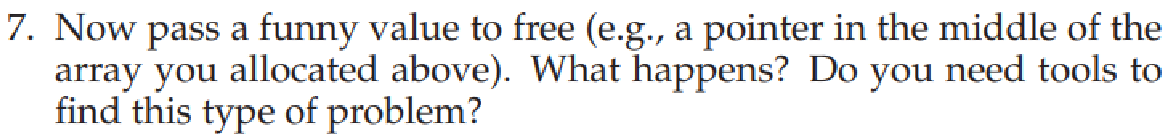
It runs well.

Running with valgrind: Invalid Read

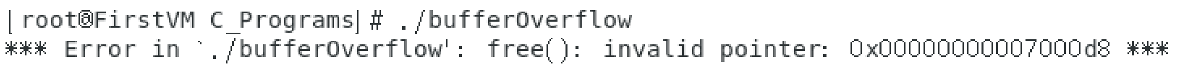


```
==5057== Invalid read of size 4
==5057==    at 0x4005FC: main (in /mnt/hgfs/ShareWithLinux/C_Programs/bufferOverflow)
==5057==  Address 0x5205040 is 0 bytes inside a block of size 400 free'd
==5057==    at 0x4C2B06D: free (vg_replace_malloc.c:540)
==5057==    by 0x4005DE: main (in /mnt/hgfs/ShareWithLinux/C_Programs/bufferOverflow)
==5057==  Block was alloc'd at
==5057==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
==5057==    by 0x4005CE: main (in /mnt/hgfs/ShareWithLinux/C_Programs/bufferOverflow)
```

Question 7

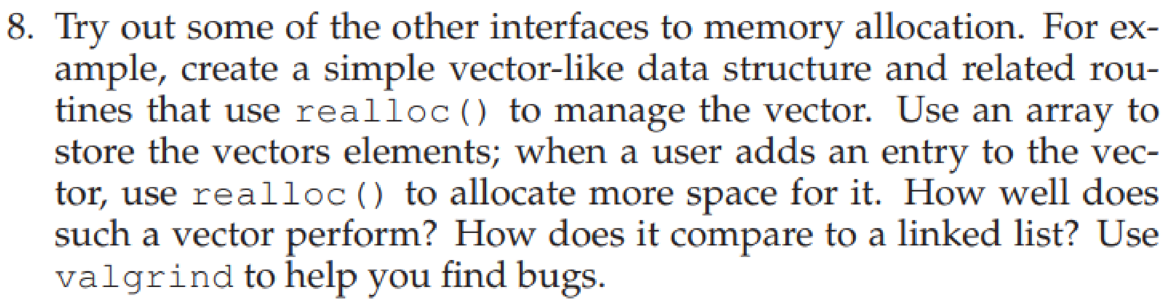


7. Now pass a funny value to free (e.g., a pointer in the middle of the array you allocated above). What happens? Do you need tools to find this type of problem?



```
[root@FirstVM C_Programs]# ./bufferOverflow
*** Error in `./bufferOverflow': free(): invalid pointer: 0x00000000007000d8 ***
```

Question 8



8. Try out some of the other interfaces to memory allocation. For example, 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.

See `Dynamic Array.c`