Due: Mon, Mar 27, 2017 @ 4pm

1. Solve question 5 from chapter 14.

```
#include <stdlib.h>

int main(){
        int * data = malloc(100 * sizeof(int));
        data[100] = 0;
}
```

Analysis of runtime behavior:

The program appears to run when compiled. No output is printed since no print statements were given in the directions.

This is misbehaving code as directed, however, since 'malloc(100 * sizeof(int))' does not allocate any memory for data[100] which is the 101'th item (index overflow, also known as buffer overrun).

C allows this since it does not have limits on buffer overrun for dynamically allocated memory, it is up to the user to properly access arrays created dynamically with some flavor of malloc or new, which this problem most likely did intentionally make a point about buffer overrun in c.

Valgrind output:

```
--34949-- ./malloc_test:
--34949-- dSYM directory is missing; consider using --dsymutil=yes
--34949-- UNKNOWN fcntl 97!
--34949-- UNKNOWN fcntl 97! (repeated 2 times)
--34949-- UNKNOWN fcntl 97! (repeated 4 times)
--34949-- UNKNOWN fcntl 97! (repeated 8 times)
--34949-- UNKNOWN fcntl 97! (repeated 16 times)
--34949-- UNKNOWN fcntl 97! (repeated 32 times)
--34949-- UNKNOWN task message [id 3406, to mach_task_self(), reply 0x60f]
--34949-- UNKNOWN host message [id 412, to mach_host_self(), reply 0x60f]
--34949-- UNKNOWN host message [id 222, to mach_host_self(), reply 0x60f]
--34949-- UNKNOWN task message [id 3410, to mach_task_self(), reply 0x60f]
==34949== Invalid write of size 4
==34949==    at 0x100000F7E: main (in ./malloc_test)
==34949==  Address 0x100a80450 is 0 bytes after a block of size 400 alloc'd
==34949==    at 0x1000073B1: malloc (vg_replace_malloc.c:303)
==34949==    by 0x100000F73: main (in ./malloc_test)
==34949==
```

'Invalid write of size 4 … Address 0x100a80450 is 0 bytes after a block of size 400' shows the invalid buffer read index. An integer is 4 bytes on my implementation of c, and so the block of 400 bytes is the 100 item integer array that was allocated.

```
#include <stdlib.h>
#include <stdio.h>
int main(){
int * data = malloc(100 * sizeof(int));
printf("pointer_before: %p\n", data);
data[0] = 500;
free(data);
printf("%p:%d\n", data, data[0]);
```

This outputs 500 for the value of data[0], showing the backing memory is not nulled when it is freed by free().

Assuming “frees them” in this question refers to freeing the pointer to the array of integers, since freeing non-pointer types causes unexpected behavior, and calling free on the pointer will free the entire underlying array pointed to by the pointer.

I believe this behavior exhibits that of a dangling pointer. The pointer was allocated to a 100 item integer array at first, and then free is called on it. Free does not null the pointer, and does not reset the memory pointed to to 0, so it is still possible to dereference the pointer and access the (non-zeroed) integer values stored within the array. Free only returns the memory to the process’s heap, it does not have any guarantees for nulling pointers or zeroing memory values.

```
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

Base   : 0x00003082 (decimal 12418)
Limit  : 472

Virtual Address Trace
VA  0: 0x000001ae (decimal:  430) --> 12418 + 430 = 0x3230
VA  1: 0x00000109 (decimal:  265) --> 12418 + 265 = 0x318B
VA  2: 0x0000020b (decimal:  523) --> seg fault
VA  3: 0x0000019e (decimal:  414) --> 12418 + 414 = 0x3220
VA  4: 0x00000322 (decimal:  802) --> seg fault
VA  5: 0x00000136 (decimal:  310) --> 12418 + 310 =0x 31B8
VA  6: 0x000001e8 (decimal:  488) --> seg fault
VA  7: 0x00000255 (decimal:  597) --> seg fault
VA  8: 0x000003a1 (decimal:  929) --> seg fault
VA  9: 0x00000204 (decimal:  516) --> seg fault
```

The limit register must be set to –l 930 for all addresses to be within the boundaries of the virtual address space.

I ran this question. It does not ask for the ./relocation output so I am omitting it for copy pasting reasons.

The maximum base register that can be used follows the following relation:

base + limit <= phys mem size

in these terms:

16284 + 100 == 16384 == 2^16 == 16KB

I confirmed this to be true by testing with the ‘-c’ switch, with a base of 16285. With a base of 16285, the address space fell out of the boundaries of physical memory, which does not change at 2^16Kb.

This confirms that 16384 must be the maximum for the case presented by this problem.