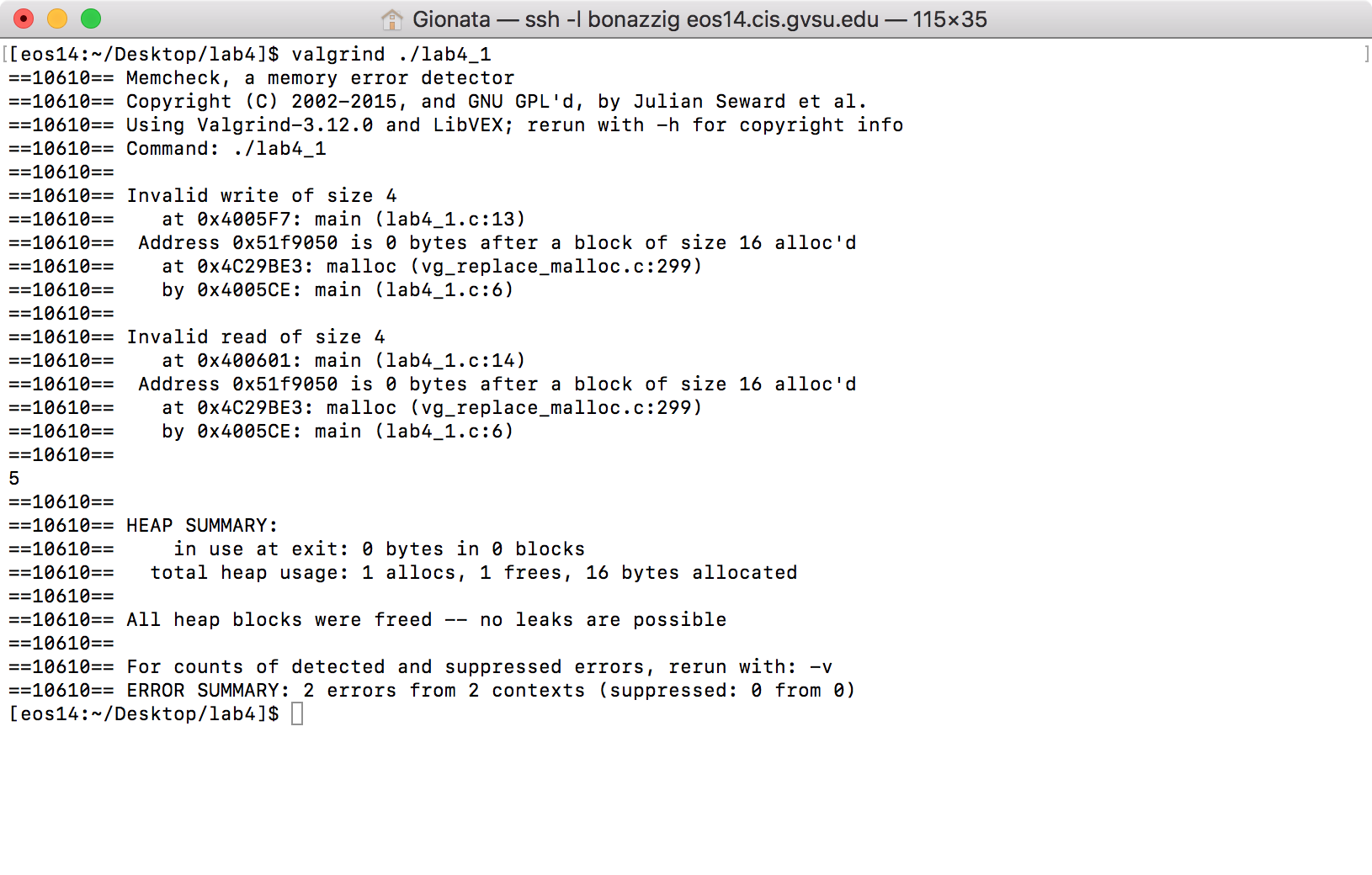
**Lab4\_1.c**

**Valgrind report:**

**Problem:**

In general, memory leaks are a result of invoking a method in C that allocate memory in the heap memory. C does not have a “garbage collector” like Java or Python, so every chunk of heap memory allocated when the program is running has to be freed, or it will stay allocated (occupied) even after the program stops running.

Now, specific to this file:

Looking at the report from valgrind, every block of heap memory has been freed, so there is no danger of memory leak. However, there are some other memory allocation faults that need to be solved. The variable ‘p’, after line 7, points to pt[0]. The for loop starting at line 9 iterates a total of 4 times (i = 0, i = 1, i = 2, i = 3), so at the end of the loop ‘p’ will point to pt[4]. Here is the problem: we allocated only 4 int-sized chunks of heap memory to pt, so the highest index of pt is pt[3]. At line 13, the program assigns 5 to pt[4], but that space of memory is not owned by pt. This is the “Invalid write of size 4” from the valgrind report. At line 14, the program tries to print the content of pt[4], incurring in a index out of bounds error, trying to access a space of memory not owned by pt, which is the second error in the valgrind report “Invalid read of size 4”.

**Solution:**

The solution I decided to go with is changing line 10, the for loop, from:

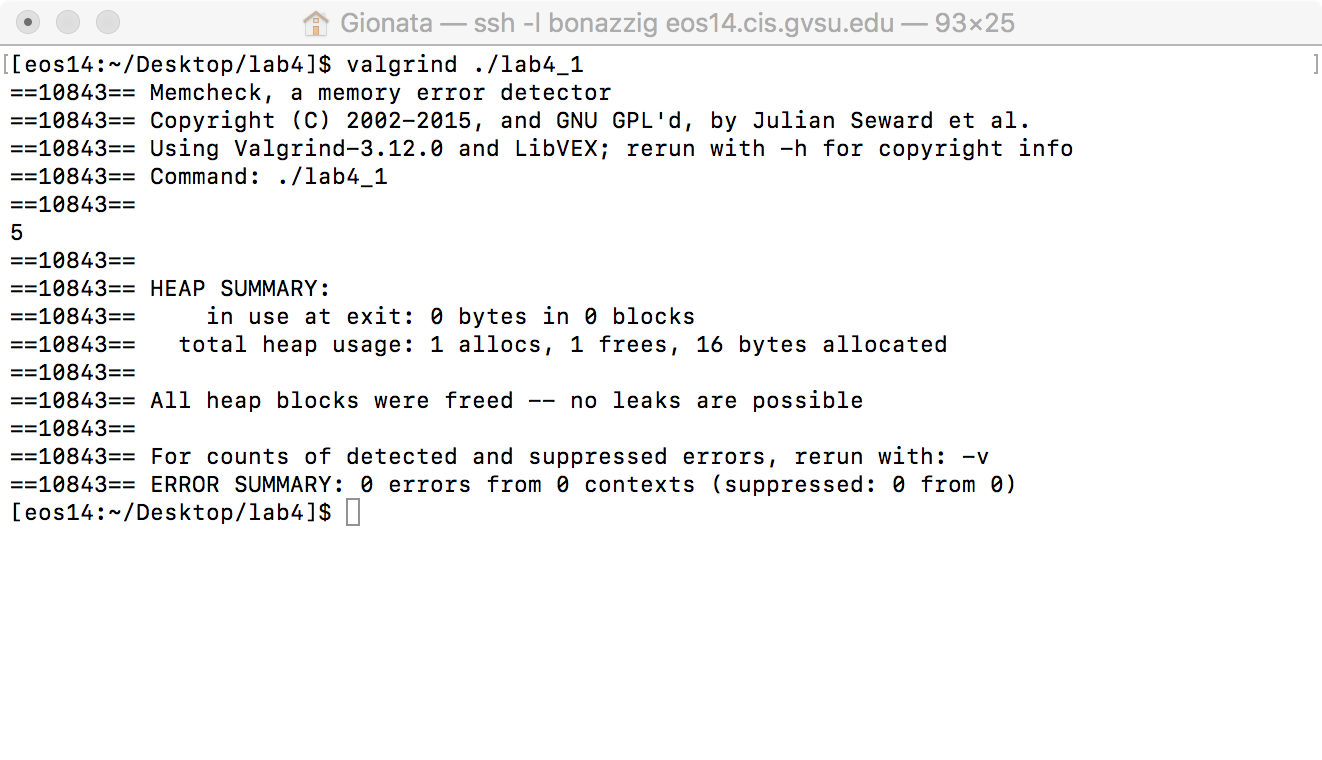
for(i=0;i<4;i++)

to:

for(i=0;i<3;i++)

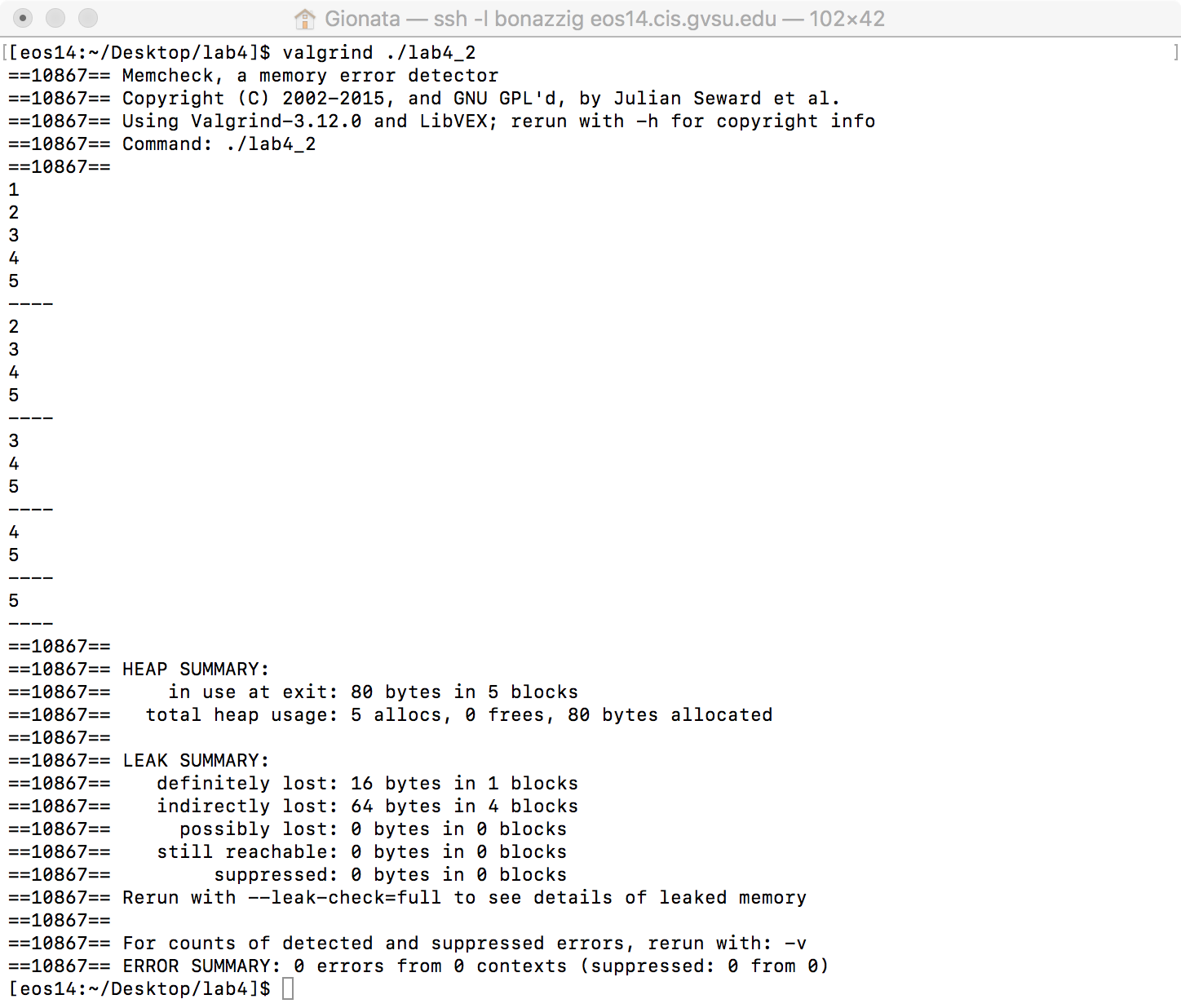
So that line 13 and 14 will access pt[3], which is a part of memory allocated to pt.

**Valgrind report after Fix:**



**Lab4\_2.c**

**Valgrind report:**



**Problem:**

From the report, we can see that the program has no compiling errors, but has some memory leaks, caused by an omission of a free() statement for each part of heap memory allocated.

addFront() is a method to add a node element to the front of the linked list. The main() method calls addFront() 5 times, so the program allocates memory for a node 5 times. However, the removeFromFront() method removes the pointer to the current head of the list, without freeing the node itself. At the end of the program, we should have 5 node-sized chunks of memory heap permanently occupied (memory leak), which is in line with the valgrind report:

The 4 indirectly lost elements are the 4 four elements deleted from the list (lost because there is no ‘active’ pointer to the pointer pointing to the memory space), and the 1 definitely lost is the last element removed, because there is not a single pointer that still points to that memory space.

**Solution:**

The solution here is to free() the node removed, starting at line 28;

**before:**

else {

head = head->next;

return head;

}

**After:**

else {

struct node\* temp = head;

head = head->next;

free(temp);

return head;

}

**Valgrind report after Fix:**

