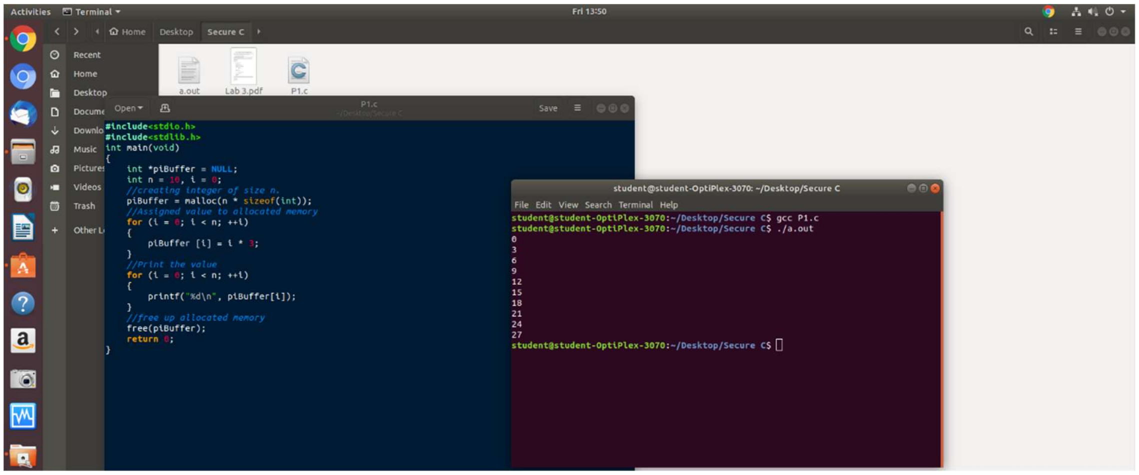
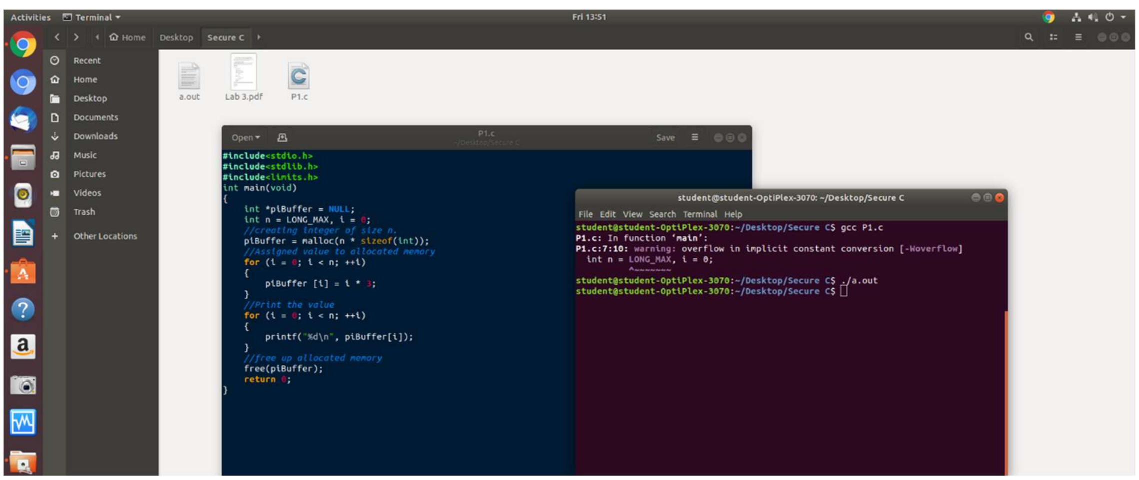
Common C Memory Management Errors

Exercise 1: Forget to check return value of malloc

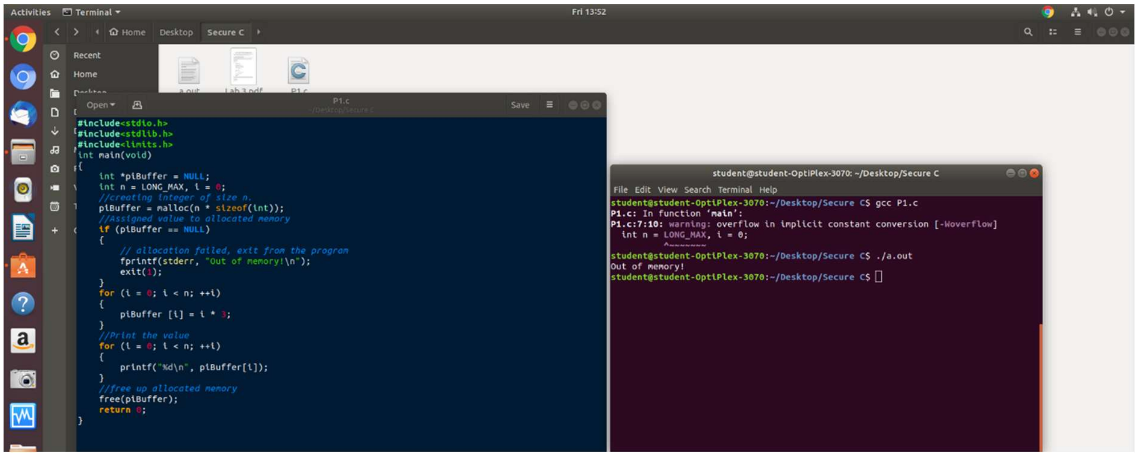
In the below example, the value of n is small, and hence no error occurs.



Next, we try to assign the value of n to a huge value (LONG_MAX). This size cannot be allocated by malloc and hence it returns a NULL pointer. As we can see, it crashes when we try to access piBuffer.

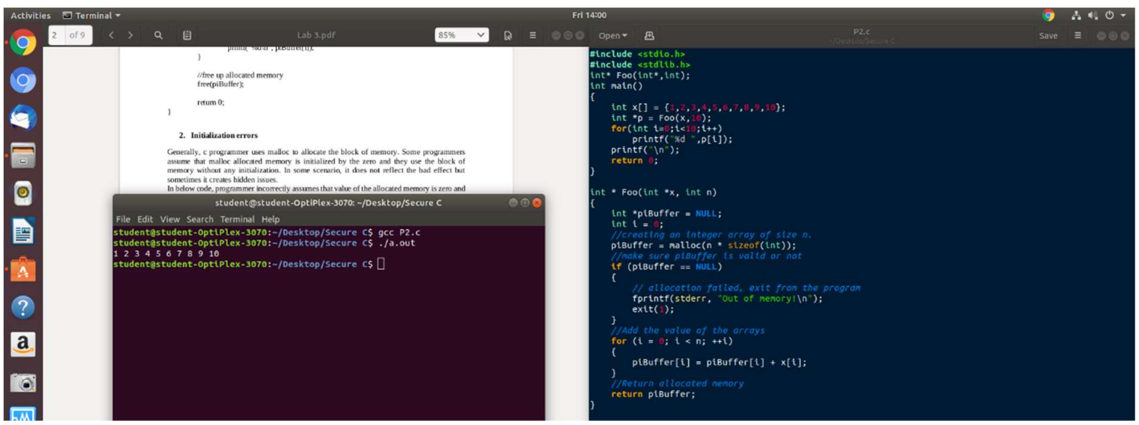


To prevent this, we first check if piBuffer points to NULL. If it does not, we continue execution, else, we display that there is no memory that has been allocated and we terminate the program.

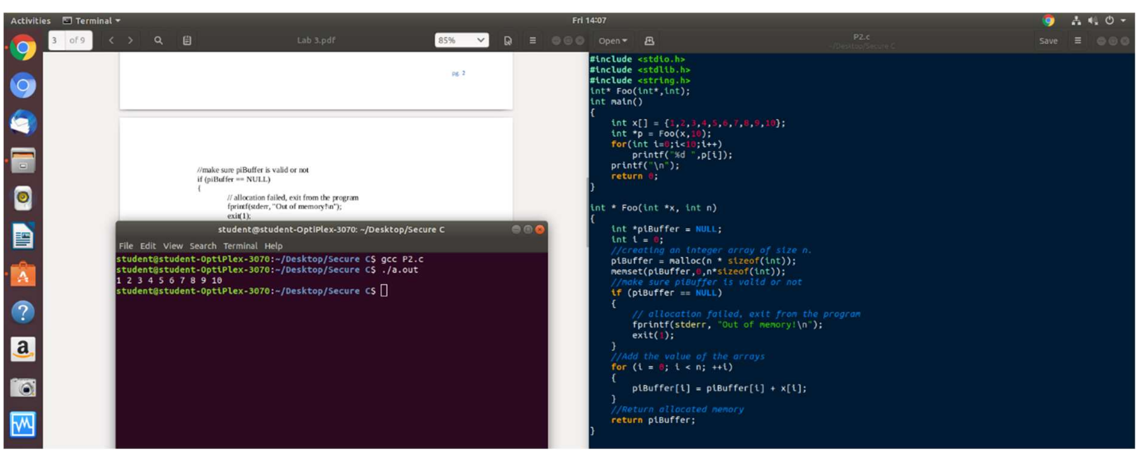


Exercise 2: Initialization Errors

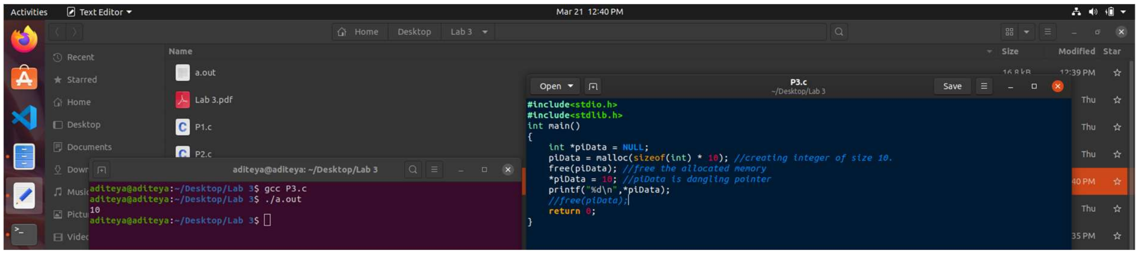
The piBuffer array was not initialised before its values were accessed. It is possible that it can hold junk values. However, due to different specific implementation-based behaviours, it was initialised with zeroes.



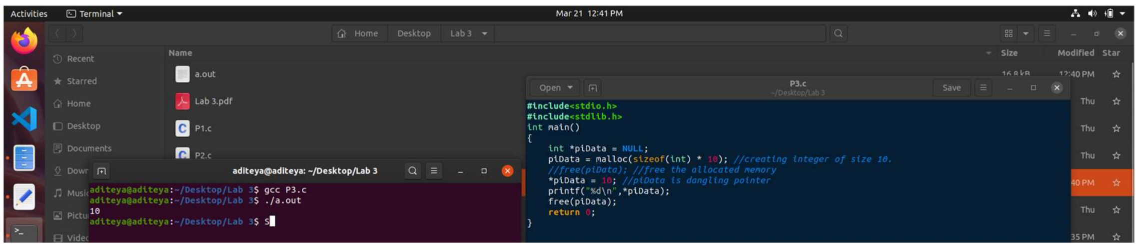
We add the memset function to explicitly initialize all values to 0. Now the piBuffer array will hold only 0's initially, and will hold the same values as the original array a[].



Exercise 3: Access the already freed memory

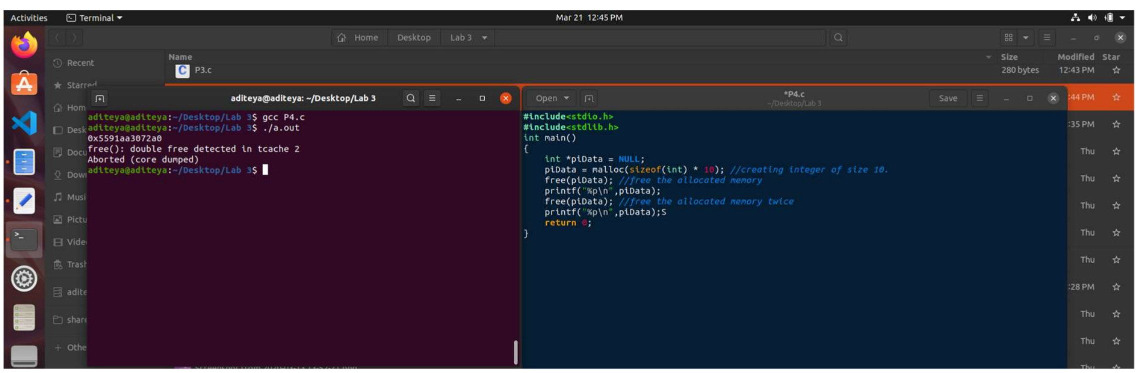


piData is a dangling pointer since it has been freed. Accessing dangling pointer leads to undefined behaviour and may cause the program to crash. To avoid such undefined behaviour, we should ensure that our pointers point to some valid memory address. We can then choose to free it.

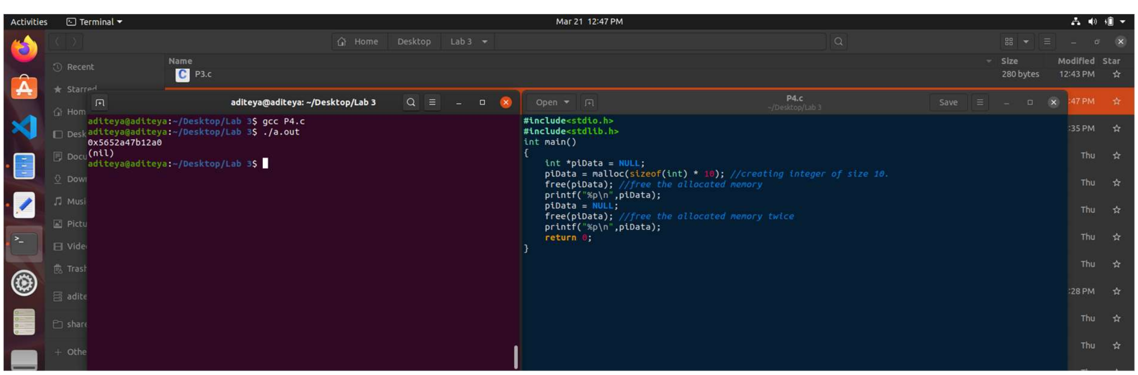


Exercise 4: Freeing the same memory multiple times

Attempting to free the same memory twice leads to undefined behaviour. Here it resulted in a segmentation fault and a program crash. After freeing once, the pointer becomes a dangling pointer, and cannot be freed again.

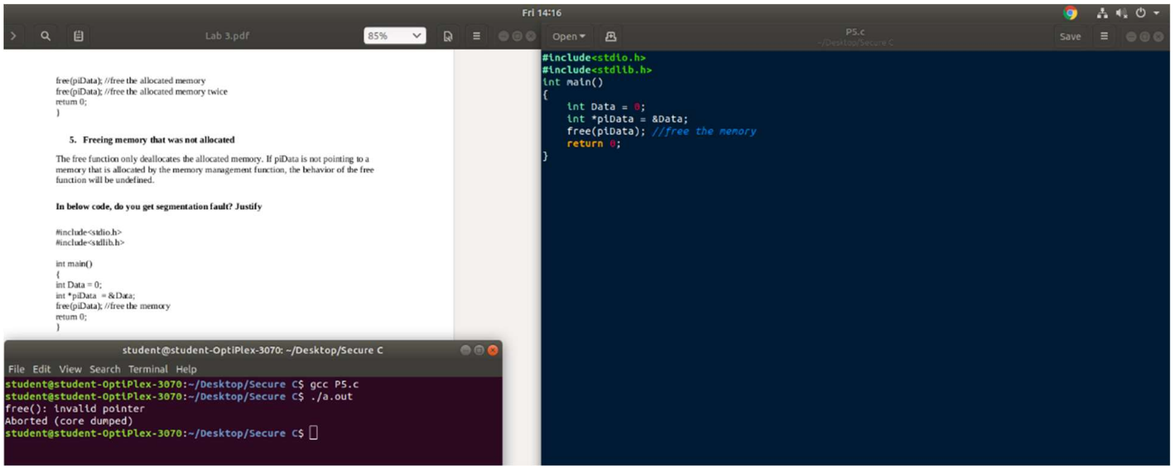


Here, we free the pointer and then assign it to NULL and attempt to free it. The second free function doesn’t result in a segmentation fault since the pointer is a NULL pointer.

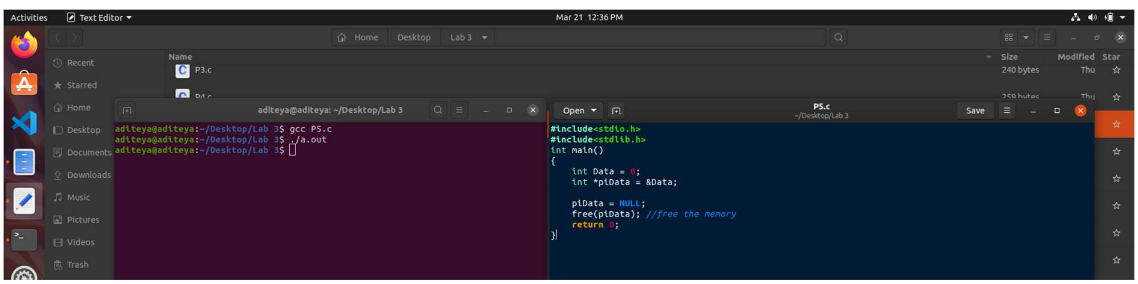


Exercise 5: Freeing memory that wasn't allocated

We cannot free memory that hasn't been allocated already using any of the memory management functions like malloc, calloc or realloc. piData doesn't point to any memory location and hence attempting to free it will lead to undefined behaviour. Here, we get a segmentation fault since we are freeing a non-initialised pointer.

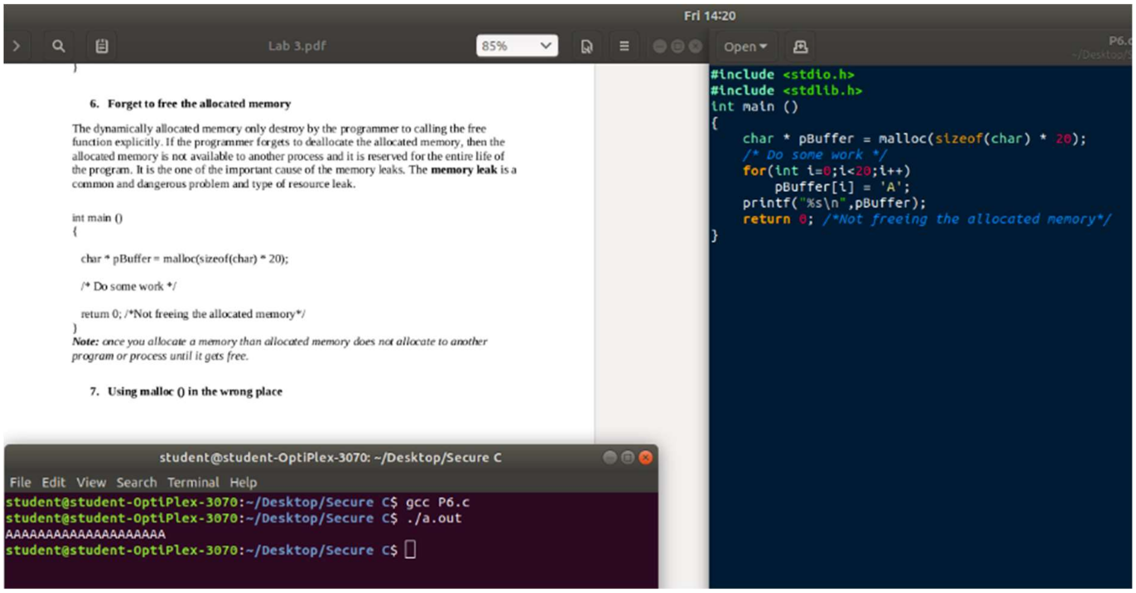


To prevent the undefined behaviour, we first assign the pointer to NULL and then free it. WE can alternatively choose to not call the free() function too.

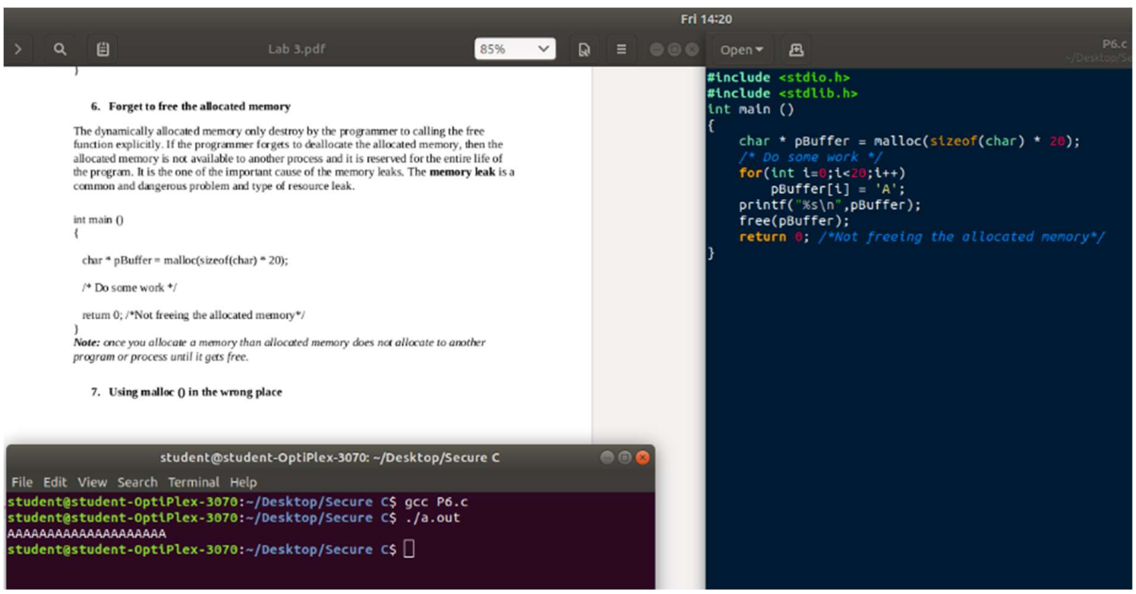


Exercise 6: Forget to free the allocated memory

Dynamically allocated memory using memory management functions is freed only when it is explicitly done so by calling the free() function. If memory is not freed, then this memory remains for the entire lifetime of the program and is not available for other processes. In such cases, we lose memory from the heap and is termed as a memory leak.

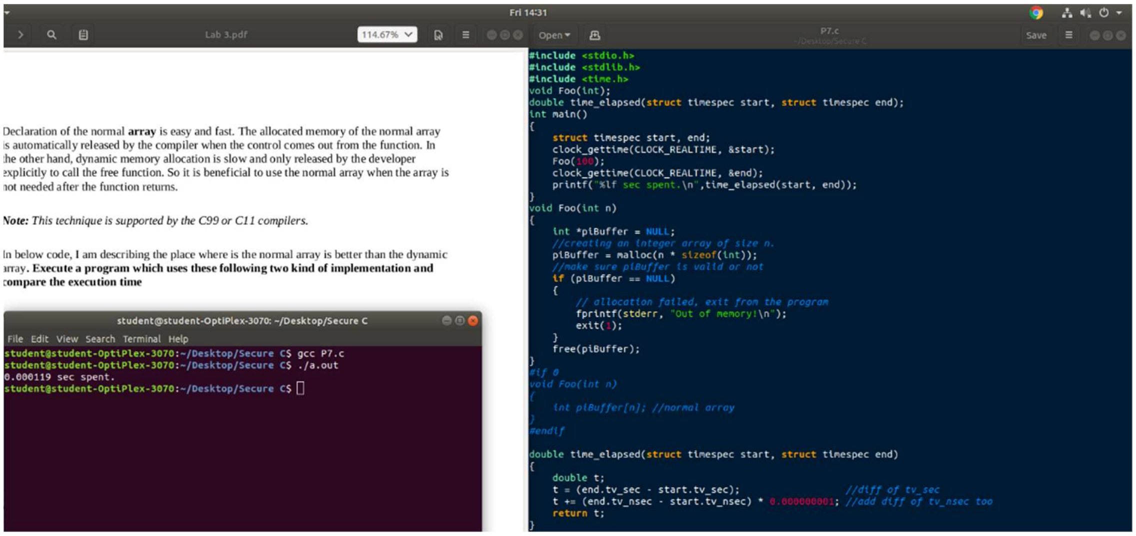


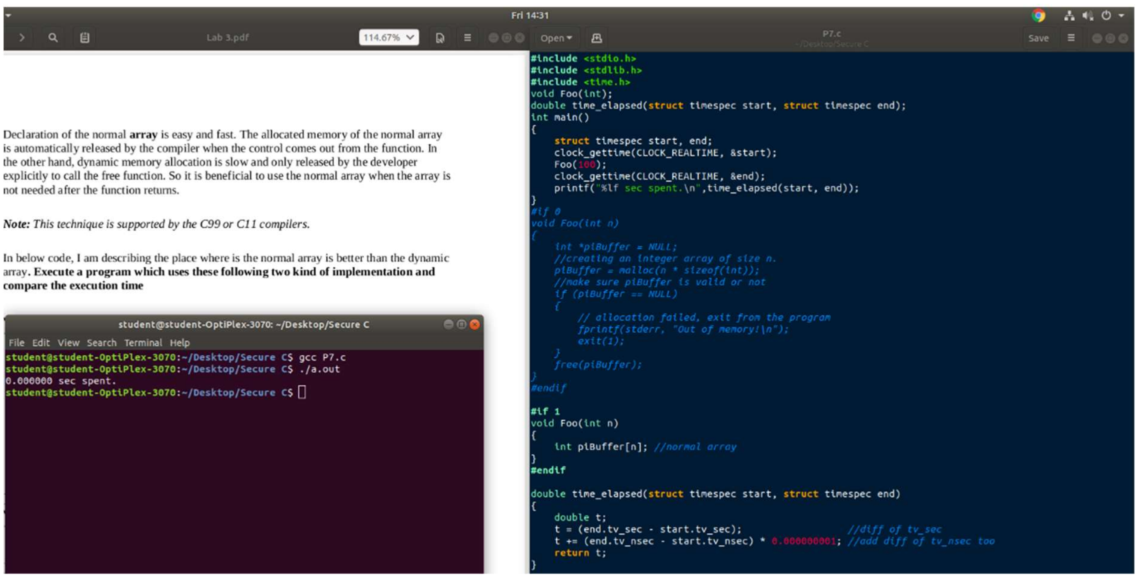
To handle memory leaks, we must always ensure to free any memory that is not in use.



Exercise 7: Using malloc in the wrong place

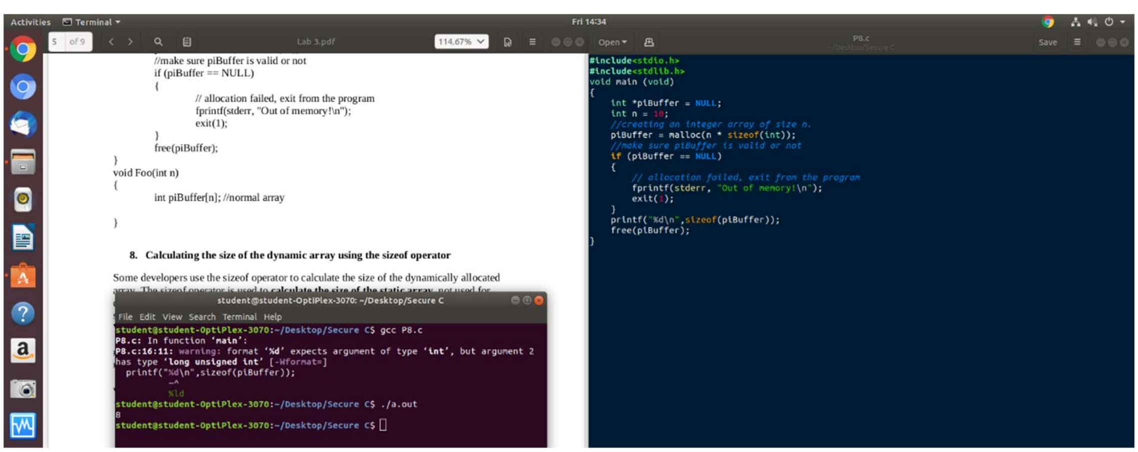
Static allocation is always faster than dynamic allocation, since for dynamic allocation, memory is always taken from the heap. We must use static allocation wherever possible to make our program run faster.



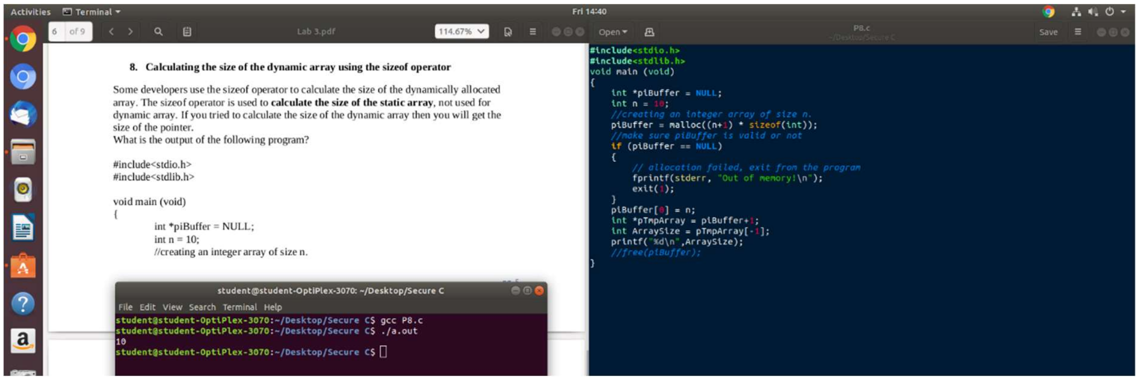


Exercise 8: Calculating size of dynamic array using sizeof() operator

If we directly try to display the size of pBuffer, we will only get the size of the variable and not the entire array of values that it holds.

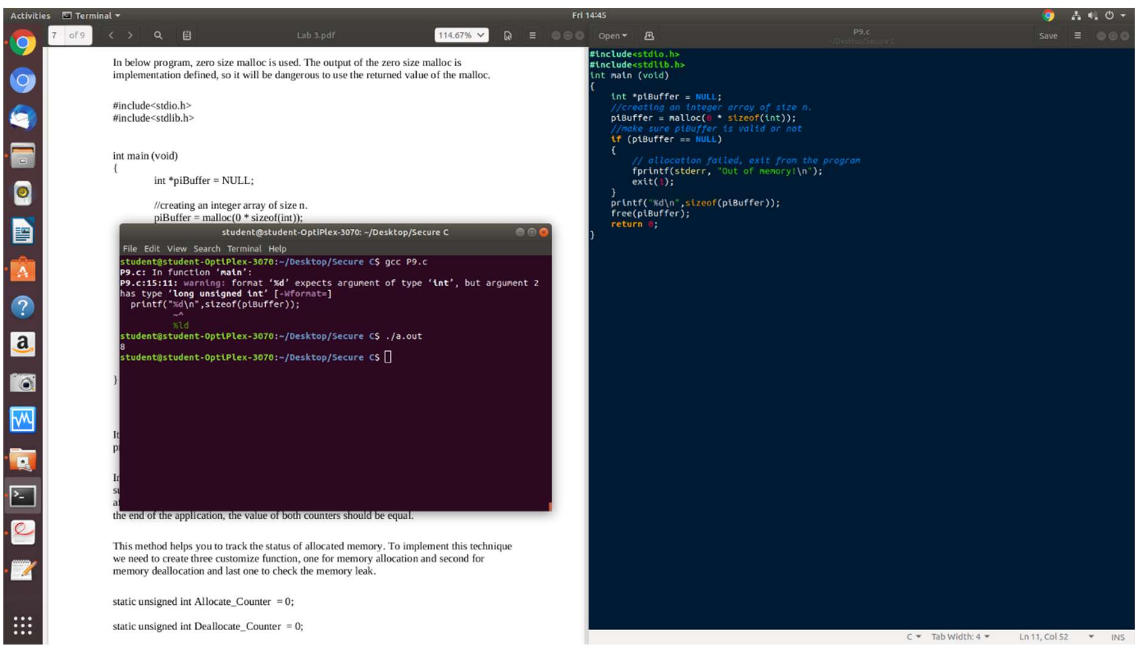


A good practice is to increase the size of the array by one and store the size of the array as the first element (0th index) and use a separate pointer to point to the second element (1st index). Hence, we will have access to both the size of the array and the elements.

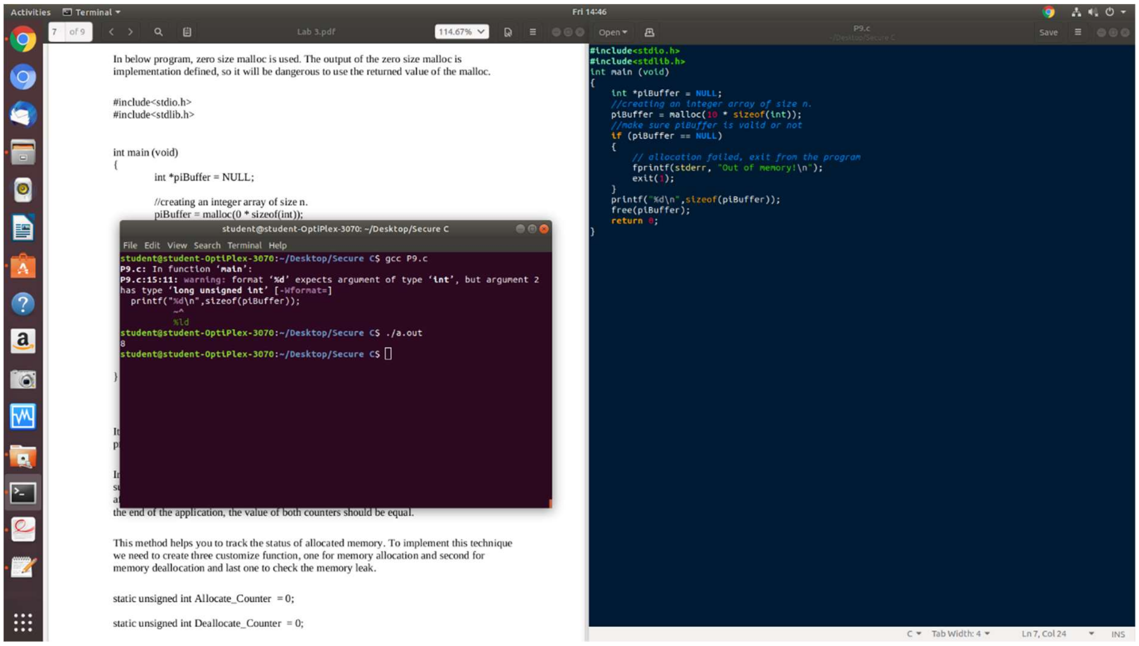


Exercise 9: Improper use of the memory management function

In the following scenario malloc returned a non-NULL pointer. This behaviour is implementation defined and leads to undefined behaviour.

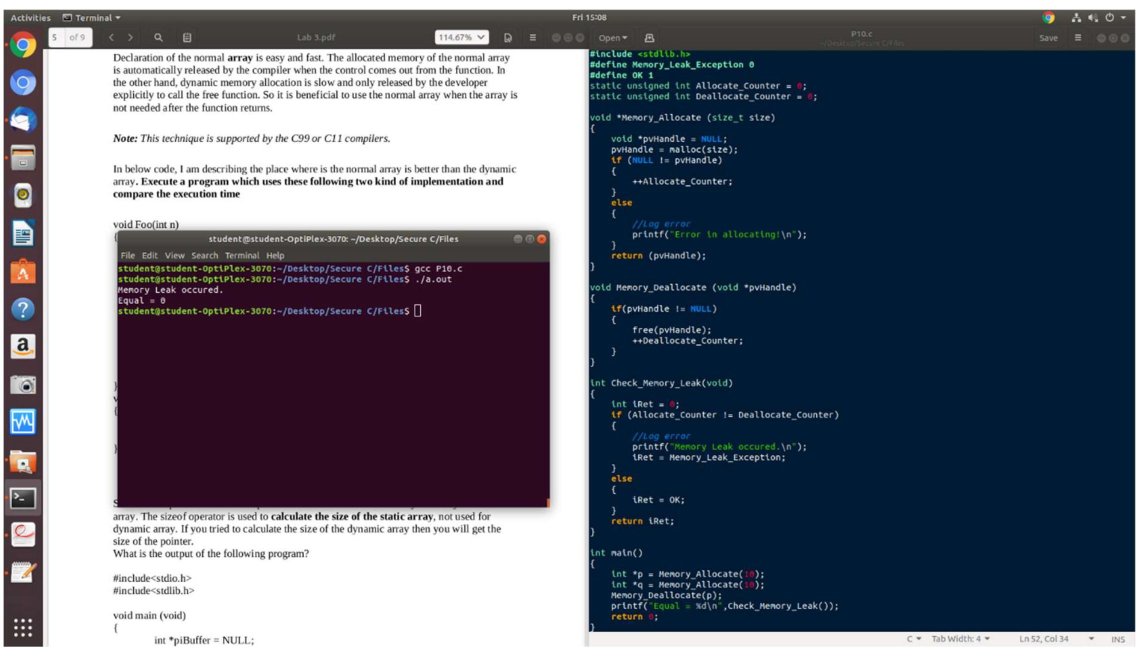


To prevent this, we must always pass a non-zero value to malloc and other memory management functions to ensure that a valid address is returned.

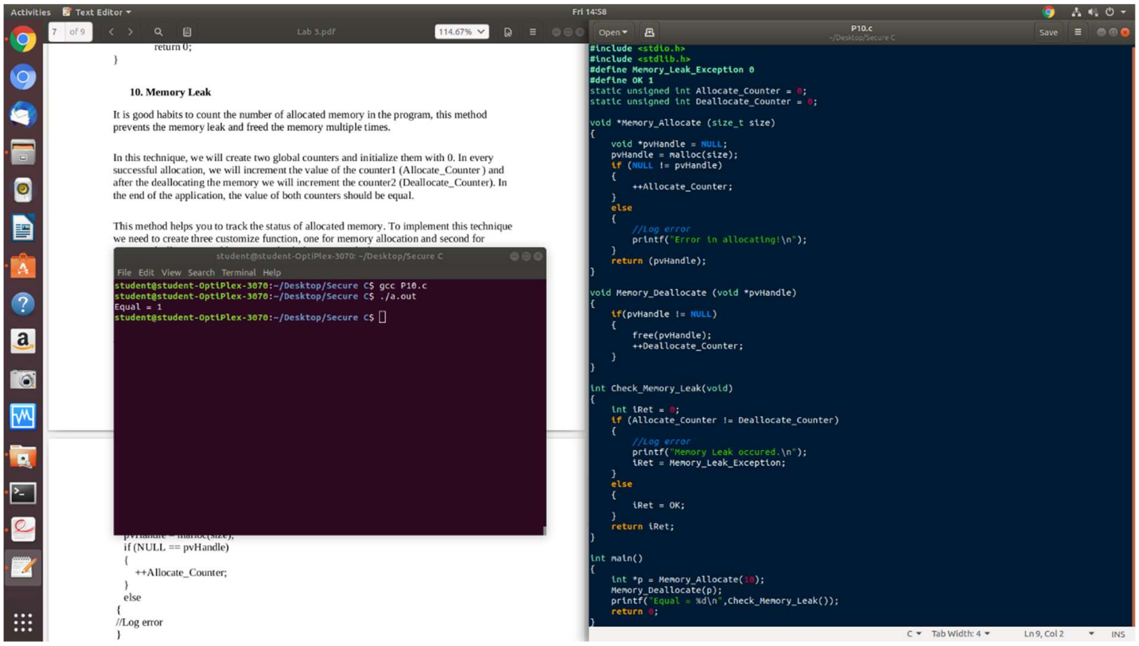


Exercise 10: Memory Leak

We define two macros, Memory_Leak_Exception as 0 and OK as 1 and allocate and deallocate memory unequal number of times. Hence memory is allocated more times and the values of Allocate_Counter and Deallocate_Counter aren't the same. Hence we get the value as 0, which indicates there was a memory leak.

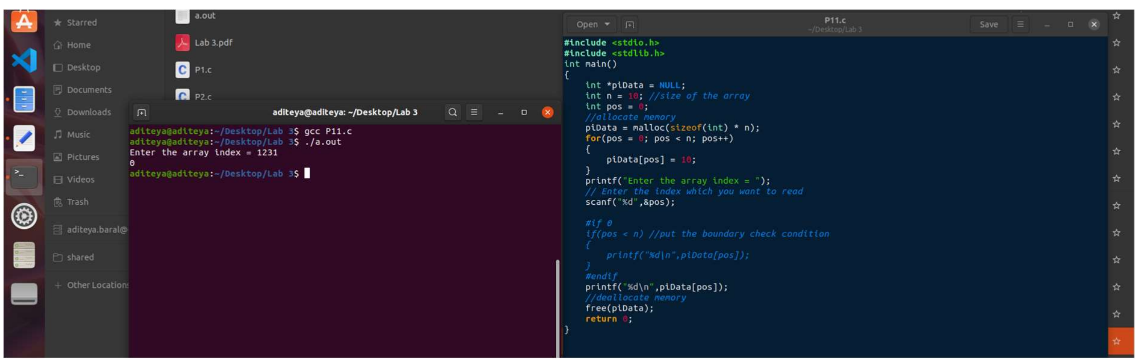


To prevent this, we ensure that memory was allocated and deallocated equal number of times. On doing so, we see that we get the value of 1, which stands for OK, implying no memory leak occurred.



Exercise 11: Accessing a dynamic array out of boundaries

Accessing a dynamic array's contents beyond what has been allocated leads to undefined behaviour. Here we got a 0, but it can be dangerous.



To prevent this, we must always ensure to check whether the index we are trying to access is within the allotted size.