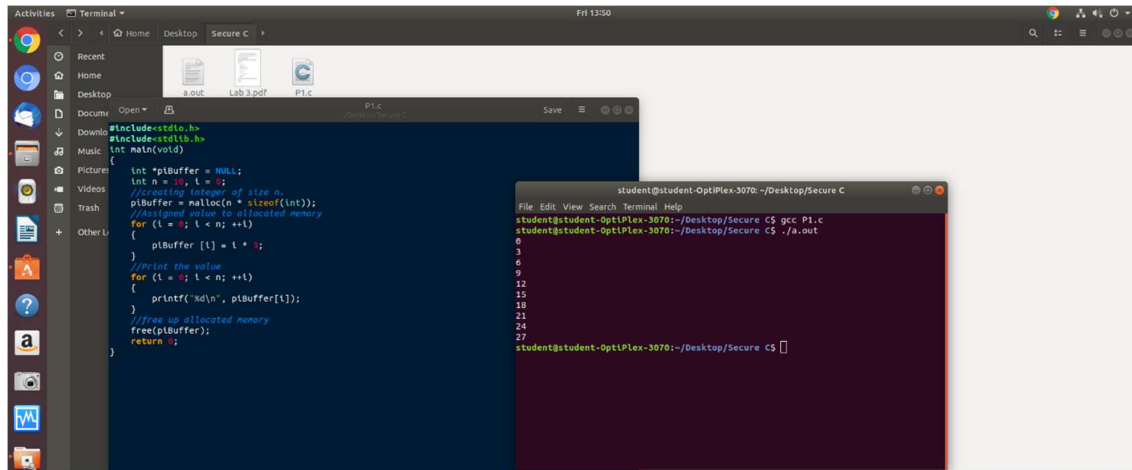


# Common C Memory Management Errors

## Exercise 1: Forget to check return value of malloc

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

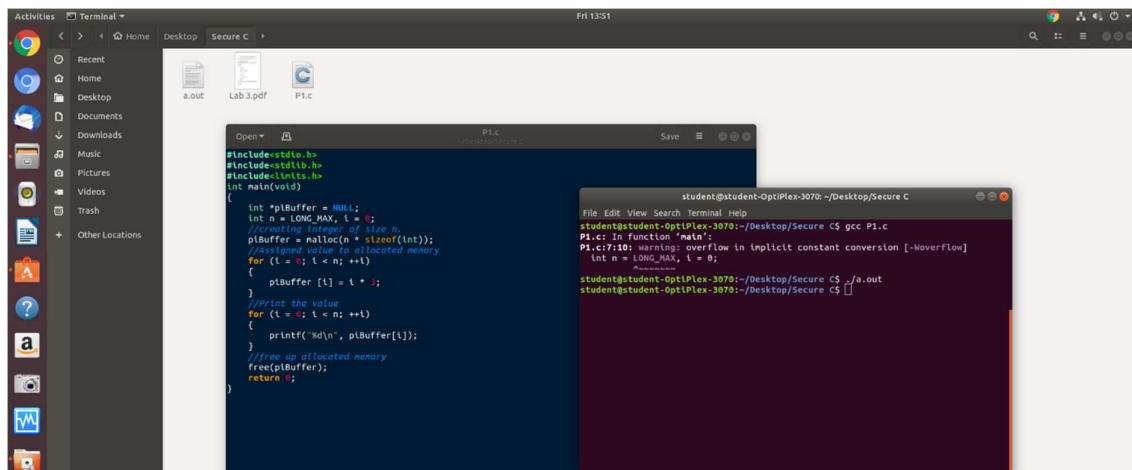


The screenshot shows a Linux desktop with a file manager and a terminal. The file manager displays a directory named 'Secure C' containing files 'a.out', 'Lab 3.pdf', and 'P1.c'. The terminal window shows the execution of the program 'P1.c', which outputs the values 0 through 27, indicating successful memory allocation and usage.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *piBuffer = NULL;
    int n = 10, i = 0;
    //creating integer of size n
    piBuffer = malloc(n * sizeof(int));
    //Assigned value to allocated memory
    for (i = 0; i < n; ++i)
    {
        piBuffer[i] = i * 2;
    }
    //Print the value
    for (i = 0; i < n; ++i)
    {
        printf("%d\n", piBuffer[i]);
    }
    //free up allocated memory
    free(piBuffer);
    return 0;
}
```

```
student@student-OptiPlex-3070: ~/Desktop/Secure C
student@student-OptiPlex-3070:~/Desktop/Secure C$ gcc P1.c
student@student-OptiPlex-3070:~/Desktop/Secure C$ ./a.out
0
2
4
6
8
10
12
14
16
18
20
22
24
26
27
student@student-OptiPlex-3070:~/Desktop/Secure C$
```

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.

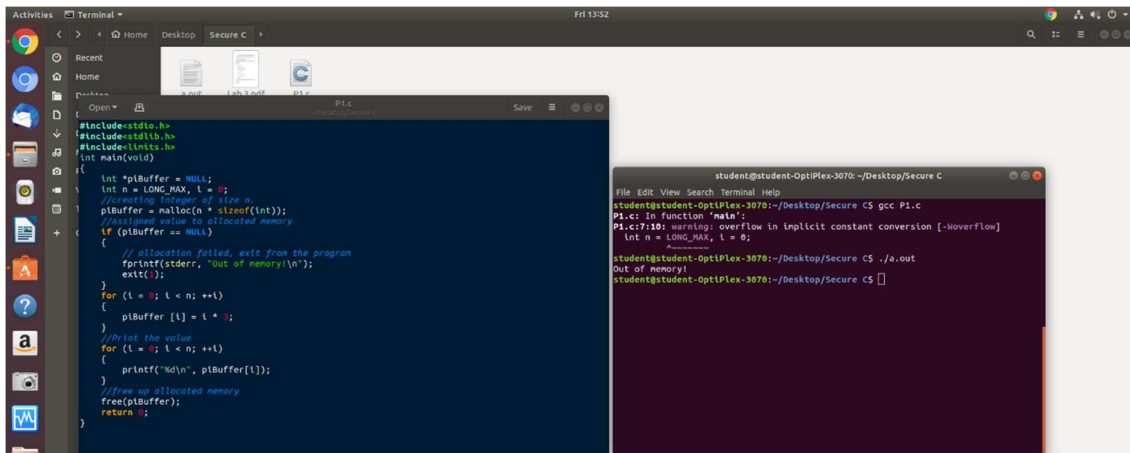


The screenshot shows the same Linux desktop environment. The file manager still shows the 'Secure C' directory. The terminal window shows the execution of the program 'P1.c' with a warning message: 'P1.c:17:10: warning: overflow in implicit constant conversion [-Woverflow]'. The program then crashes with a segmentation fault, indicated by the 'P1.c:17:10: warning: overflow in implicit constant conversion [-Woverflow]' message and the 'P1.c:17:10: warning: overflow in implicit constant conversion [-Woverflow]' message.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
int main(void)
{
    int *piBuffer = NULL;
    int n = LONG_MAX, i = 0;
    //creating integer of size n
    piBuffer = malloc(n * sizeof(int));
    //Assigned value to allocated memory
    for (i = 0; i < n; ++i)
    {
        piBuffer[i] = i * 2;
    }
    //Print the value
    for (i = 0; i < n; ++i)
    {
        printf("%d\n", piBuffer[i]);
    }
    //free up allocated memory
    free(piBuffer);
    return 0;
}
```

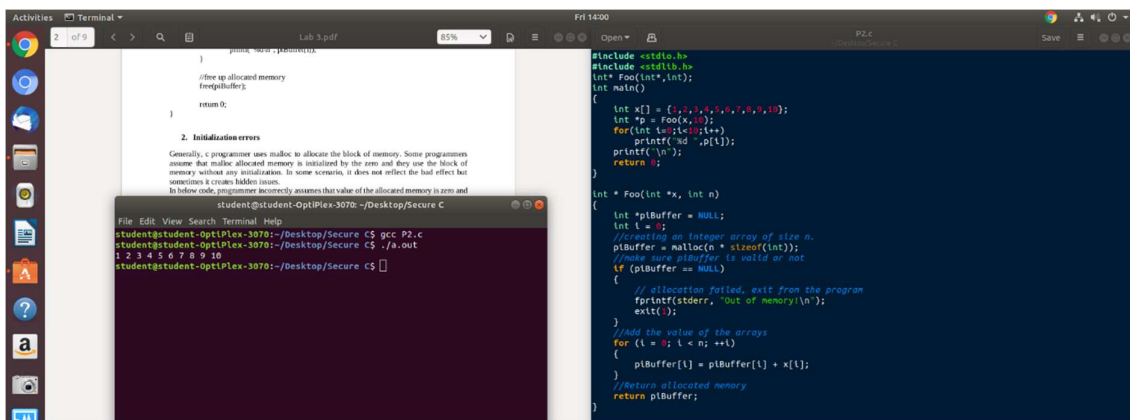
```
student@student-OptiPlex-3070: ~/Desktop/Secure C
student@student-OptiPlex-3070:~/Desktop/Secure C$ gcc P1.c
P1.c:17:10: warning: overflow in implicit constant conversion [-Woverflow]
    int n = LONG_MAX, i = 0;
           ^
student@student-OptiPlex-3070:~/Desktop/Secure C$ ./a.out
student@student-OptiPlex-3070:~/Desktop/Secure C$
```

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.

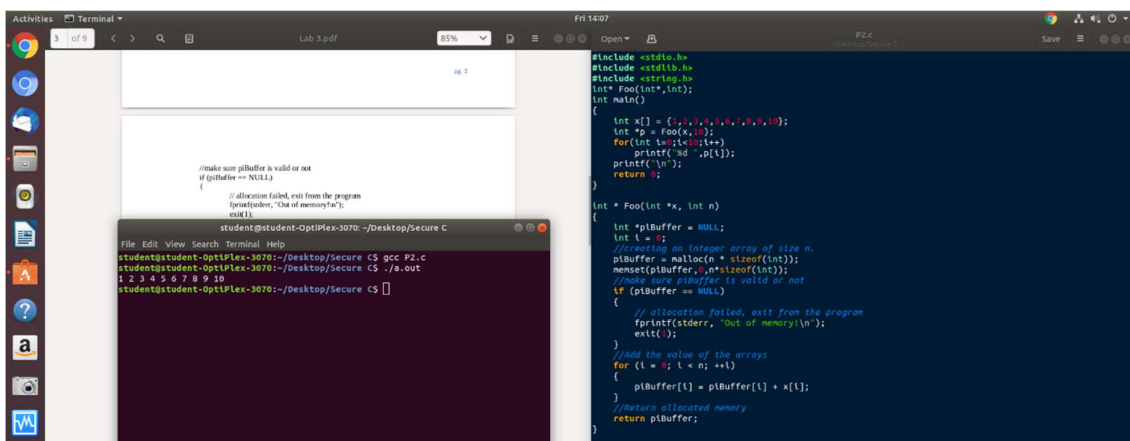


## Exercise 2: Initialization Errors

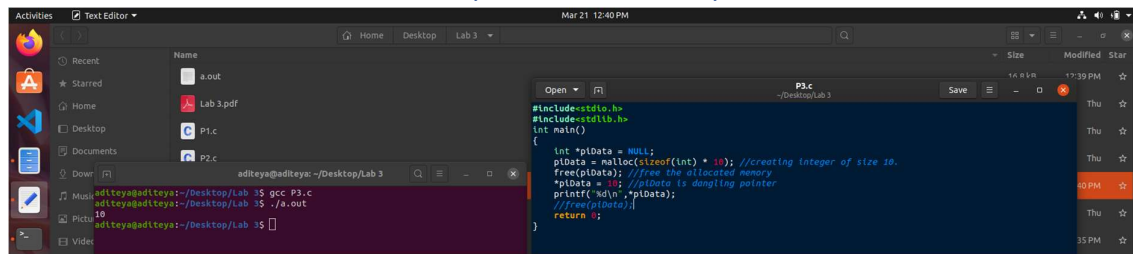
The piBuffer array was not initialised before its values were accessed. It is possible that it can hold junk values. However, due to different specific implementation-based behaviours, it was initialised with zeroes.



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[].

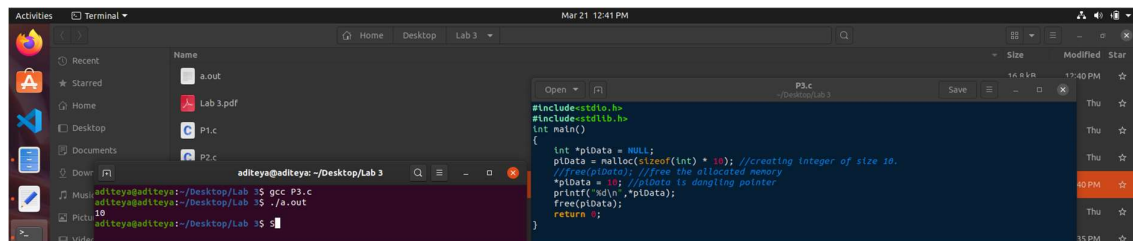


## Exercise 3: Access the already freed memory



```
aditeya@aditeya:~/Desktop/Lab 3$ gcc P3.c
aditeya@aditeya:~/Desktop/Lab 3$ ./a.out
16.91n
17:39 PM
```

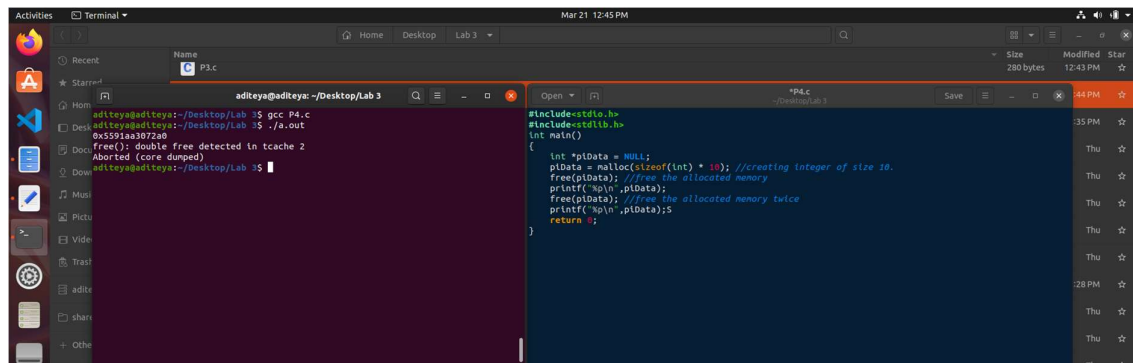
piData is a dangling pointer since it has been freed. Accessing dangling pointer leads to undefined behaviour and may cause the program to crash. To avoid such undefined behaviour, we should ensure that our pointers point to some valid memory address. We can then choose to free it.



```
aditeya@aditeya:~/Desktop/Lab 3$ gcc P3.c
aditeya@aditeya:~/Desktop/Lab 3$ ./a.out
16.91n
17:40 PM
```

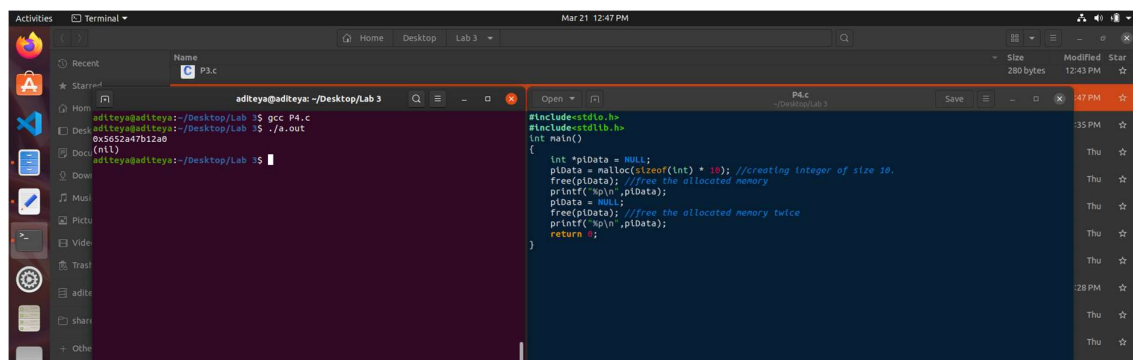
## Exercise 4: Freeing the same memory multiple times

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.



```
aditeya@aditeya:~/Desktop/Lab 3$ gcc P4.c
aditeya@aditeya:~/Desktop/Lab 3$ ./a.out
0x5591aa3072a0
free(): double free detected in tcache 2
Aborted (core dumped)
aditeya@aditeya:~/Desktop/Lab 3$
```

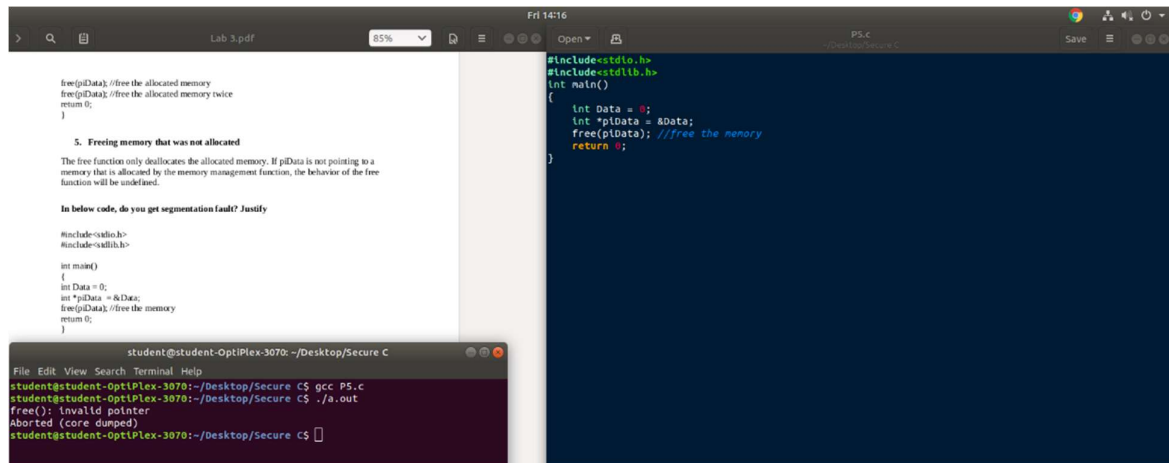
Here, we free the pointer and then assign it to NULL and attempt to free it. The second free function doesn't result in a segmentation fault since the pointer is a NULL pointer.



```
aditeya@aditeya:~/Desktop/Lab 3$ gcc P4.c
aditeya@aditeya:~/Desktop/Lab 3$ ./a.out
0x5652a47b12a0
aditeya@aditeya:~/Desktop/Lab 3$
```

## 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.



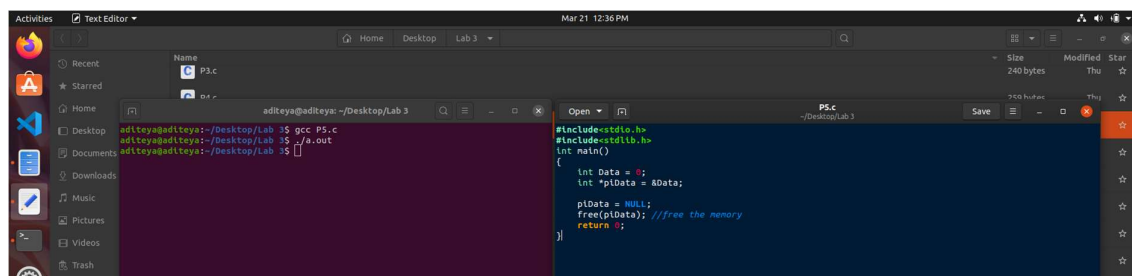
The screenshot shows a code editor with a C program and a terminal window. The C program defines a variable `Data` and a pointer `piData` pointing to `Data`. It then attempts to free `piData` twice, which is incorrect because `piData` is not a dynamically allocated memory block. The terminal output shows a segmentation fault and a core dump.

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int Data = 0;
    int *piData = &Data;
    free(piData); //free the memory
    return 0;
}
```

```
student@student-OptiPlex-3070: ~/Desktop/Secure C
student@student-OptiPlex-3070:~/Desktop/Secure C$ gcc P5.c
student@student-OptiPlex-3070:~/Desktop/Secure C$ ./a.out
free(): invalid pointer
Aborted (core dumped)
student@student-OptiPlex-3070:~/Desktop/Secure C$
```

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.



The screenshot shows the same C program as before, but with a modification: the pointer `piData` is set to `NULL` before being passed to the `free` function. The terminal output shows that the program runs successfully without any errors.

```
#include<stdio.h>
#include<stdlib.h>

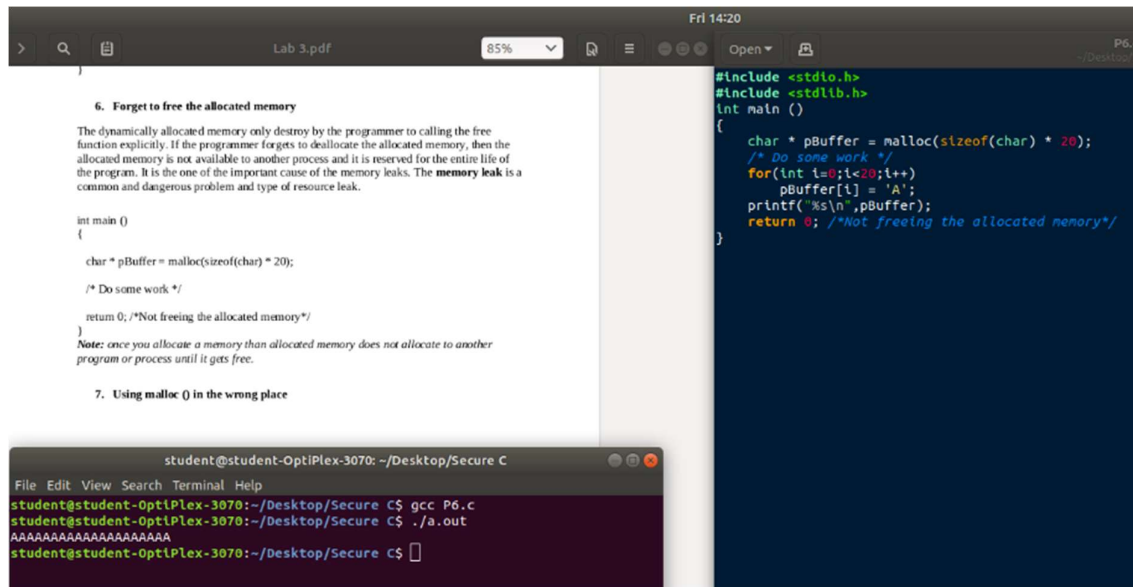
int main()
{
    int Data = 0;
    int *piData = &Data;

    piData = NULL;
    free(piData); //free the memory
    return 0;
}
```

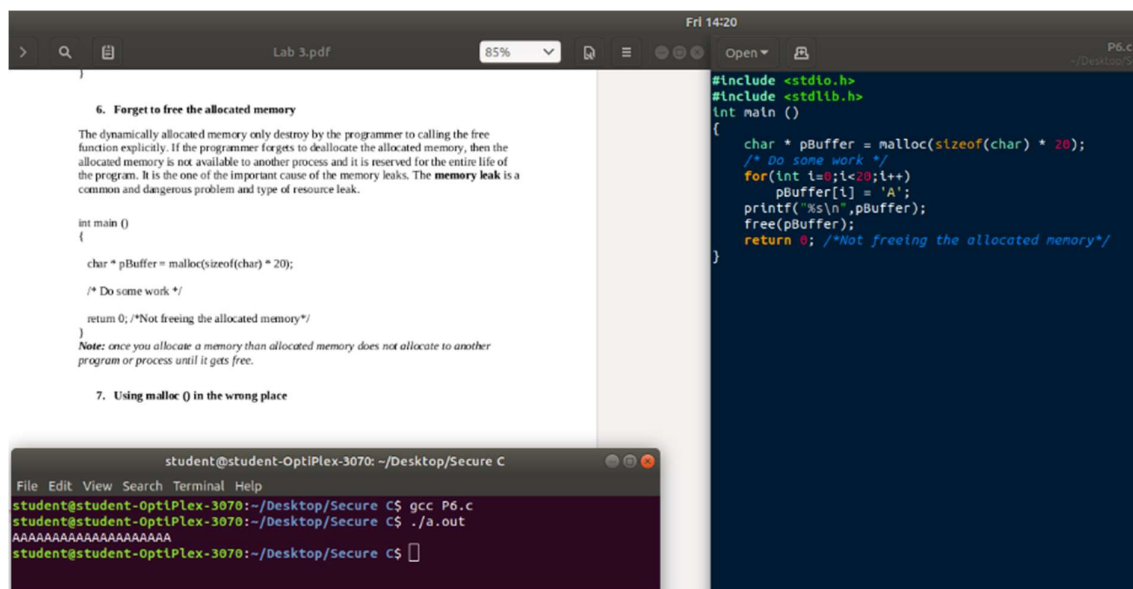
```
aditeya@aditeya: ~/Desktop/Lab 3
aditeya@aditeya:~/Desktop/Lab 3$ gcc P5.c
aditeya@aditeya:~/Desktop/Lab 3$ ./a.out
aditeya@aditeya:~/Desktop/Lab 3$
```

## 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.

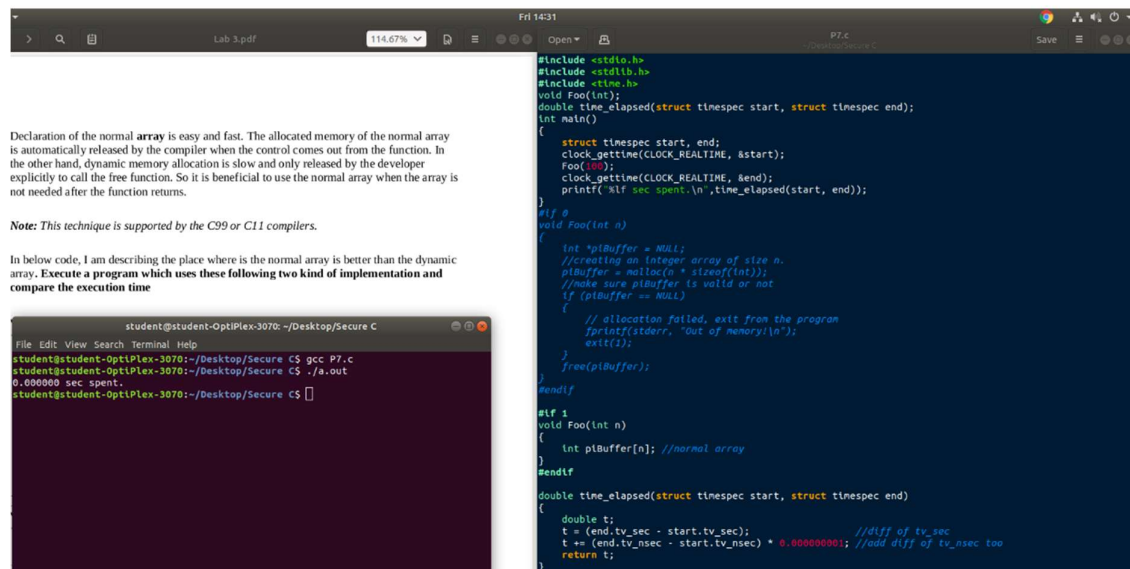
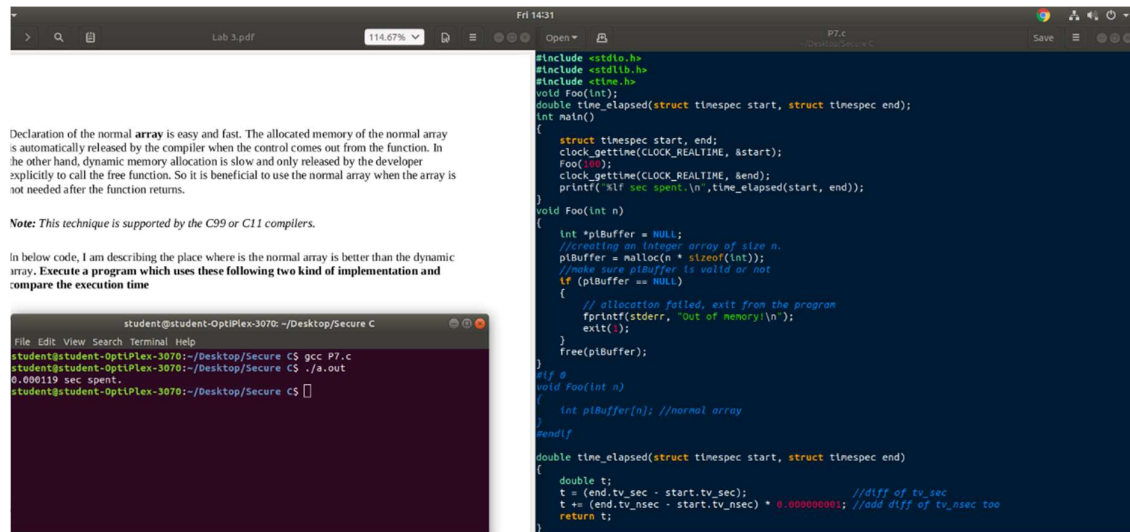


To handle memory leaks, we must always ensure to free any memory that is not in use.



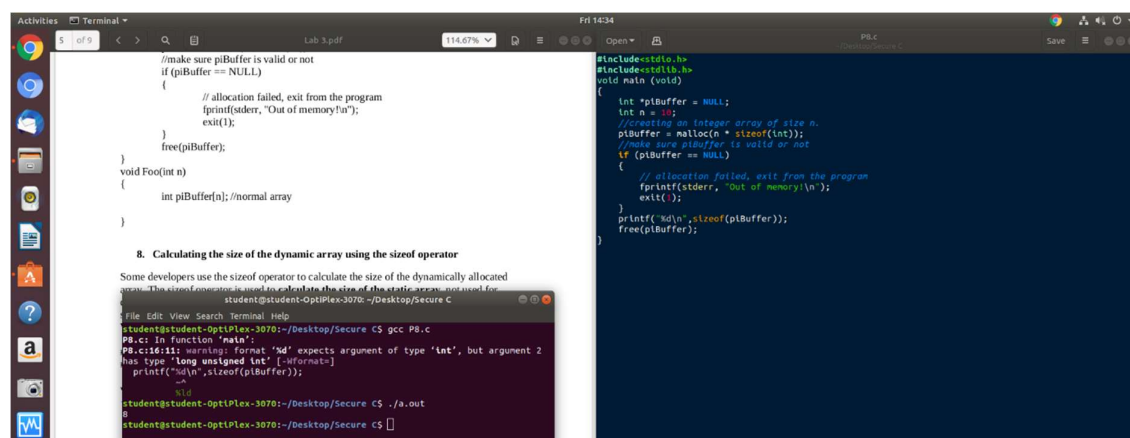
## Exercise 7: Using malloc in the wrong place

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 (0<sup>th</sup> index) and use a separate pointer to point to the second element (1<sup>st</sup> index). Hence, we will have access to both the size of the array and the elements.

```

8. Calculating the size of the dynamic array using the sizeof operator

Some developers use the sizeof operator to calculate the size of the dynamically allocated array. The sizeof operator is used to calculate the size of the static array, not used for dynamic array. If you tried to calculate the size of the dynamic array then you will get the size of the pointer.
What is the output of the following program?

#include<stdio.h>
#include<stdlib.h>

void main (void)
{
    int *piBuffer = NULL;
    int n = 10;
    //creating an integer array of size n.

    int *piBuffer = NULL;
    int n = 10;
    //creating an integer array of size n.

    student@student-OptiPlex-3070: ~/Desktop/Secure C
    student@student-OptiPlex-3070:~/Desktop/Secure C$ gcc P8.c
    student@student-OptiPlex-3070:~/Desktop/Secure C$ ./a.out
    10
    student@student-OptiPlex-3070:~/Desktop/Secure C$
  
```

## 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.

```

In below program, zero size malloc is used. The output of the zero size malloc is implementation defined, so it will be dangerous to use the returned value of the malloc.

#include<stdio.h>
#include<stdlib.h>

int main (void)
{
    int *piBuffer = NULL;
    //creating an integer array of size n.
    piBuffer = malloc(0 * sizeof(int));

    student@student-OptiPlex-3070: ~/Desktop/Secure C
    student@student-OptiPlex-3070:~/Desktop/Secure C$ gcc P9.c
    P9.c: in function 'main':
    P9.c:11:11: warning: format '%d' expects argument of type 'int', but argument 2 has type 'long unsigned int' [-Wformat=]
    printf("%d\n", sizeof(piBuffer));
    ^~
    student@student-OptiPlex-3070:~/Desktop/Secure C$ ./a.out
    0
    student@student-OptiPlex-3070:~/Desktop/Secure C$

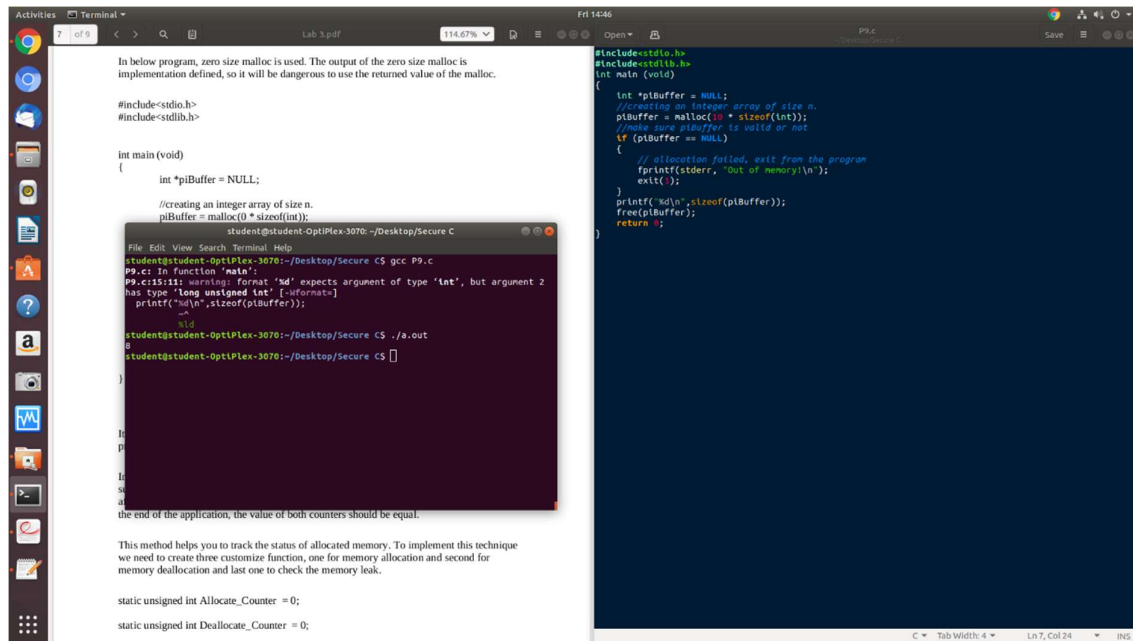
    At the end of the application, the value of both counters should be equal.

    This method helps you to track the status of allocated memory. To implement this technique we need to create three customize function, one for memory allocation and second for memory deallocation and last one to check the memory leak.

    static unsigned int Allocate_Counter = 0;
    static unsigned int Deallocate_Counter = 0;

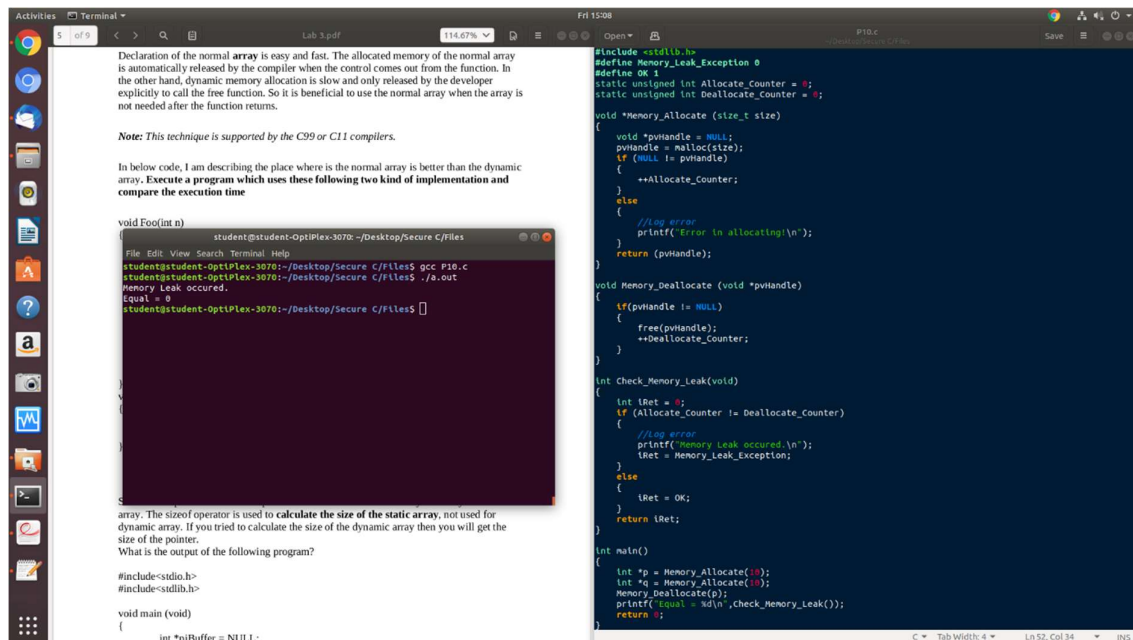
    student@student-OptiPlex-3070:~/Desktop/Secure C$
  
```

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.



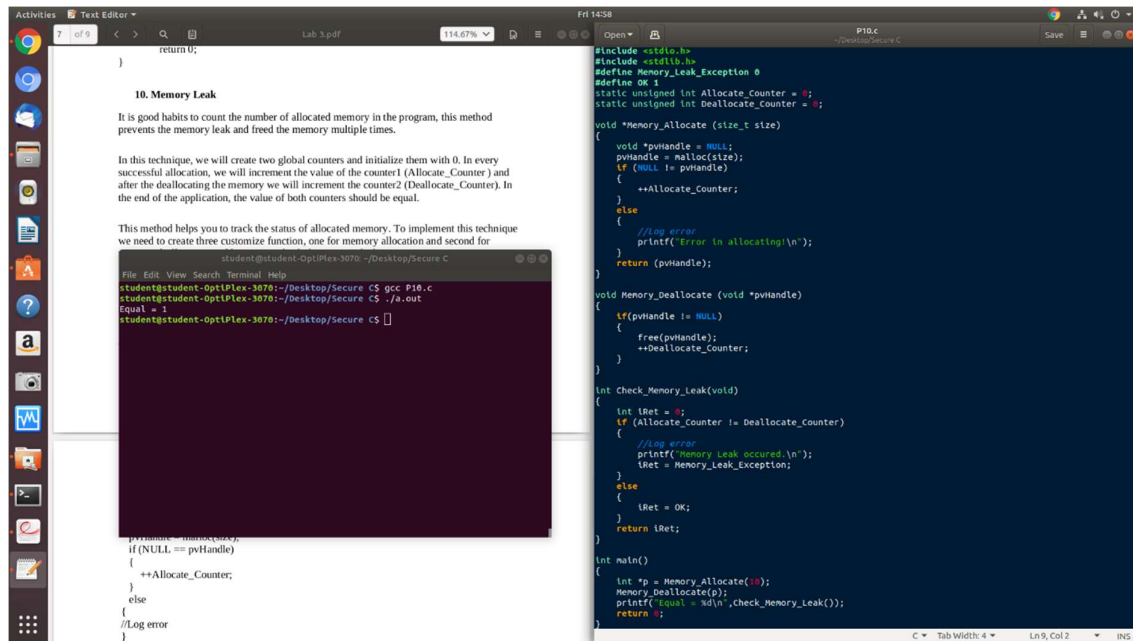
## Exercise 10: Memory Leak

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.



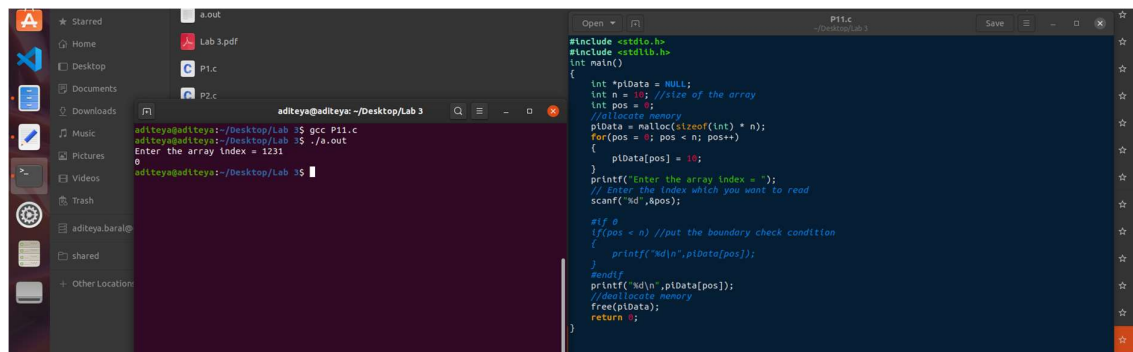
To prevent this, we ensure that memory was allocated and deallocated equal number of times. On doing so, we see that we get the value of 1, which stands for OK, implying no memory leak occurred.





## 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.

