Lab 3: Common C Memory Management Errors

Execute the following code snippets to identify the common C Memory Management errors. Submit the observations along with snapshots of the program and output.

1. **Forget to check return value of malloc**

In below code, everything is fine until the malloc function doesn't return the null pointer. If malloc returns the NULL, the code will crash.
Could you practically demonstrate under which condition malloc function fail??

```
#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 * 3;
	}

	//Print the value
	for (i = 0; i < n; ++i)
	{
		printf("%d\n", piBuffer[i]);
	}
	//free up allocated memory

	free(piBuffer);
	return 0;
}
```

We can resolve the above problem to verify the return value of malloc function. If malloc returns the null pointer, the code will display an error message and terminate the execution.

```
#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));

	//make sure pcBuffer is valid or not
	if (piBuffer == NULL)
	{
		// allocation failed, exit from the program
		fprintf(stderr, "Out of memory!\n");
		exit(1);
	}

	//Assigned value to allocated memory
	for (i = 0; i < n; ++i)
	{
		piBuffer[i] = i * 3;
	}

	//Print the value
	for (i = 0; i < n; ++i)
	{
		printf("%d\n", piBuffer[i]);
	}

	//free up allocated memory
	free(piBuffer);

	return 0;
}
```

2. Initialization errors

Generally, c programmer uses malloc to allocate the block of memory. Some programmers assume that malloc allocated memory is initialized by the zero and they use the block of memory without any initialization. In some scenario, it does not reflect the bad effect but sometimes it creates hidden issues.
In below code, programmer incorrectly assumes that value of the allocated memory is zero and performs some arithmetical operation.
Implement the code snippet as a complete program and discuss about the outcome of this code.

```
int * Foo(int *x, int n)
{
	int *piBuffer = NULL;
	int i = 0;

	//creating an integer array of size n.
	piBuffer = malloc(n * sizeof(int));
```

```
        //make sure piBuffer is valid or not
        if (piBuffer == NULL)
        {
                // allocation failed, exit from the program
                fprintf(stderr, "Out of memory!\n");
                exit(1);
        }
        //Add the value of the arrays
        for (i = 0; i < n; ++i)
        {
                piBuffer[i] = piBuffer[i] + x[i];
        }
                //Return allocated memory
        return piBuffer;
}
```

Note: *If you have required initialized memory, use the memset() along with malloc or called the calloc() that allocate the initialized memory.*

3. **Access the already freed memory**

When you freed the allocated memory then still **pointer** pointing to the same address. if you attempt to read or write the freed pointer then might be you succeed but it is illegal and can be the cause of the code crashing. It is also a reason to born the **dangling pointer**.

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

int main()
{
int *piData = NULL;
piData = malloc(sizeof(int) * 10); //creating integer of size 10.
free(piData); //free the allocated memory
*piData = 10; //piData is dangling pointer
return 0;
}
```

4. **Freeing the same memory multiple times**

A free function is used to deallocate the allocated memory. If piData (arguments of free) is pointing to a memory that has been deallocated (using the free or realloc function), the behavior of free function would be undefined.
The freeing the memory twice is more dangerous then memory leak, so it is a good habit to assign NULL to the deallocated pointer because the free function does not perform anything with the **null pointer**.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int *piData = NULL;
piData = malloc(sizeof(int) * 10); //creating integer of size 10.
```

```
free(piData); //free the allocated memory
free(piData); //free the allocated memory twice
return 0;
}
```

5. **Freeing memory that was not allocated**

The free function only deallocates the allocated memory. If piData is not pointing to a memory that is allocated by the memory management function, the behavior of the free function will be undefined.

In below code, do you get segmentation fault? Justify

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

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

6. **Forget to free the allocated memory**

The dynamically allocated memory only destroy by the programmer to calling the free function explicitly. If the programmer forgets to deallocate the allocated memory, then the allocated memory is not available to another process and it is reserved for the entire life of the program. It is the one of the important cause of the memory leaks. The **memory leak** is a common and dangerous problem and type of resource leak.

```
int main ()
{

  char * pBuffer = malloc(sizeof(char) * 20);

  /* Do some work */

  return 0; /*Not freeing the allocated memory*/
}
```

Note: *once you allocate a memory than allocated memory does not allocate to another program or process until it gets free.*

7. **Using malloc () in the wrong place**

Declaration of the normal **array** is easy and fast. The allocated memory of the normal array is automatically released by the compiler when the control comes out from the function. In the other hand, dynamic memory allocation is slow and only released by the developer explicitly to call the free function. So it is beneficial to use the normal array when the array is not needed after the function returns.

Note: *This technique is supported by the C99 or C11 compilers.*

In below code, I am describing the place where is the normal array is better than the dynamic array**. Execute a program which uses these following two kind of implementation and compare the execution time**

```
void Foo(int n)
{
	int *piBuffer = NULL;
	//creating an integer array of size n.
	piBuffer = malloc(n * sizeof(int));
	//make sure piBuffer is valid or not
	if (piBuffer == NULL)
	{
		// allocation failed, exit from the program
		fprintf(stderr, "Out of memory!\n");
		exit(1);
	}
	free(piBuffer);
}
void Foo(int n)
{
	int piBuffer[n]; //normal array

}
```

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.
```

```
        piBuffer = malloc(n * sizeof(int));
        //make sure piBuffer is valid or not
        if (piBuffer == NULL)
        {
                // allocation failed, exit from the program
                fprintf(stderr, "Out of memory!\n");
                exit(1);
        }
        printf("%d\n",sizeof(piBuffer));
        free(piBuffer);
}
```

It is a great idea to carry the length of the dynamic array. Whenever you have required the length of the array, you need to read the stored length. To implement this idea in the program we need to allocate some extra space to store the length. Whenever you use the technique then check that length of the array should not exceed the type of the array.

For example,
Suppose you need to create an integer array whose size is n. So to carry the array length of the array, you need to allocate the memory for n+1

int *piArray = malloc (sizeof(int) * (n+1));

If memory is allocated successfully, assign n (size of the array) its 0 places.

piArray[0] = n;
or
*** piArray = n;**

Now it's time to create a copy of original pointer but to left one location from the beginning.
int * pTmpArray = piArray +1;
Now, whenever in a program you ever required the size of the array then you can get from copy pointer.

ArraySize = pTmpArray[-1];

After using the allocated memory don't forget to deallocate the allocated memory.

free (piArray);

9. Improper use of the memory management function

It is very important to use the memory management function in proper ways. Some developer uses the zero size malloc in their program. It is very dangerous because if the size of the requested space is zero, the behavior will be implementation-defined. The return value of the malloc could be a null pointer or it shows the behavior like that size is some nonzero value.

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));

	//make sure piBuffer is valid or not
	if (piBuffer == NULL)
	{
		// allocation failed, exit from the program
		fprintf(stderr, "Out of memory!\n");
		exit(1);
	}

	printf("%d\n",sizeof(piBuffer));
	free(piBuffer);
	return 0;
}
```

10. Memory Leak

It is good habits to count the number of allocated memory in the program, this method prevents the memory leak and freed the memory multiple times.

In this technique, we will create two global counters and initialize them with 0. In every successful allocation, we will increment the value of the counter1 (Allocate_Counter) and after the deallocating the memory we will increment the counter2 (Deallocate_Counter). In 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;


void *Memory_Allocate (size_t size)
{
```

```
    void *pvHandle = NULL;

    pvHandle = malloc(size);
    if (NULL == pvHandle)
    {
        ++Allocate_Counter;
    }
    else
 {
 //Log error
 }
    return (pvHandle);
}


void Memory_Deallocate (void *pvHandle)
{
    if(pvHandle != NULL)
    {
        free(pvHandle);
        ++Deallocate_Counter;
    }
}


int Check_Memory_Leak(void)
{
 int iRet = 0;
    if (Allocate_Counter != Deallocate_Counter)
 {
 //Log error
 iRet = Memory_Leak_Exception;
 }
 else
 {
 iRet = OK;
 }
  return iRet;
}
```

11. Accessing a dynamic array out of boundaries

It is a common mistake that is done by the developers. When you access the dynamic array out of the boundary then the behavior of your program can be undefined. We can resolve this problem to put a check condition before to access the array.

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

```
int main()
{
        int *piData = NULL;
        int n = 10; //size of the array
        int pos = 0;

        //allocate memory
        piData = malloc(sizeof(int) * n);
        for(pos = 0; pos < n; pos++)
        {
        piData[pos] = 10;
        }

        printf("Enter the array index = ");
        // Enter the index which you want to read
        scanf("%d",&pos);

         if( pos < n) //put the boundary check condition
           {
           printf("%d\n",piData[pos]);
           }
        //deallocate memory
        free(piData);
        return 0;
}
```