

# Handling Pointers and Memory Management in C Efficiently

Guru Swarupa      Prof. Megha Sharma

October 3, 2024

## 1 Introduction

In the field of computer programming, efficient memory management is crucial. This technical report discusses a practical experience with dynamic memory allocation and pointer handling in C, which highlights the importance of avoiding dangling pointers and understanding their impact on program stability. The narrative unfolds around a problem assigned by my department professor, the debugging process, and a real-world example of how dangling pointers can have significant consequences.

## 2 The Problem Statement

The assignment was to write a program that inputs elements into an array using dynamic memory allocation with the ‘malloc’ function, without relying on static arrays or array syntax. The program should then display the array both in its original order and in reverse. This exercise was designed to enhance understanding of pointers and memory management in C, without prior knowledge that it would lead to specific issues, allowing me to encounter and learn from these challenges firsthand.

## 3 Implementation and Issue Encountered

The initial implementation involved dynamically allocating memory for an array and then allowing user input to populate this array. However, a critical mistake was made: after freeing the allocated memory with the ‘free’ command, the pointer to this memory was not set to ‘NULL’. This oversight led to a segmentation fault, which occurs when a program tries to access a memory location that is not allowed.

Here is a simplified version of the problematic code:

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

```

int display(int *arr, int size) {
    for(int i = 0; i < size; i++) {
        printf("%d", *arr++);
    }
}

int main() {
    int *arr, size;

    printf("Enter the number of elements: ");
    scanf("%d", &size);

    // Allocate memory
    arr = (int *)malloc(size * sizeof(int));

    // Issue with CrowdStrike outage was with not checking it
    // I forgot to add it too initially
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Input elements
    for (int i = 0; i < size; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", arr++);
    }

    display(arr, size); // This line causes a segmentation fault

    // Free allocated memory
    free(arr); // Dangling pointer issue

    // arr = NULL; //this was the cause for segmentation fault!
    return 0;
}

```

## 4 Debugging Process

During the debugging phase, it was identified that accessing the memory after it was freed led to the segmentation fault. The issue was traced back to the fact that the pointer 'arr' was not set to 'NULL' after freeing the memory. As a result, 'arr' became a dangling pointer, pointing to an invalid memory location.

## 5 Explanation of Dangling Pointers

A dangling pointer is a pointer that continues to reference a memory location after the memory has been freed. Accessing or modifying memory through a dangling pointer can lead to undefined behavior, crashes, or security vulnerabilities. In the example provided, attempting to access the freed memory caused a segmentation fault, a common symptom of accessing invalid memory.

## 6 Real-World Implications

The importance of proper pointer management was further highlighted by a recent global outage involving Microsoft and the security company CrowdStrike.[1] In this case, the issue arose from a missing NULL check, which led the program to attempt access to memory locations it was not authorized to access. Windows recognized this unauthorized access as a potential security threat. To protect the system, Windows triggered a system crash, resulting in the Blue Screen of Death (BSOD) and causing an outage. This real-world example illustrates the severe consequences of pointer-related errors and highlights the critical need for rigorous memory management practices.

## 7 Conclusion

This experience highlights how crucial it is to understand and manage pointers when programming. Handling memory correctly and avoiding dangling pointers are key to creating reliable and secure software. By learning from both classroom exercises and real-world examples, programmers can see why careful memory management is so important.

## References

- [1] Rahul Raj. Analysis of the crowdstrike. [https://www.linkedin.com/posts/rahulraj90\\_crowdstrike-crowdstrike-microsoft-activity-7220297385292259328-B7JN](https://www.linkedin.com/posts/rahulraj90_crowdstrike-crowdstrike-microsoft-activity-7220297385292259328-B7JN), 2024. Available on LinkedIn.