9. Common Errors and Debugging

9.1 Memory Leaks

Problem:

```
void createList() {
    struct Node *head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = NULL;
    // MEMORY LEAK! head is lost when function returns
}
```

Solution:

```
struct Node* createList() {
    struct Node *head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = NULL;
    return head;  // Return pointer to caller
}

// In main:
struct Node *myList = createList();
// ... use list ...
freeList(&myList);  // Free memory when done
```

9.2 Dereferencing NULL Pointer

Problem:

```
void insertAtEnd(struct Node **head, int value) {
    struct Node *temp = *head;
    while (temp->next != NULL) {  // CRASH if head is NULL!
```

```
        temp = temp->next;
    }
    // ...
}
```

Solution:

```
void insertAtEnd(struct Node **head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (*head == NULL) {  // Check for NULL first!
        *head = newNode;
        return;
    }

    struct Node *temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

9.3 Lost Node References

Problem:

```
void deleteNode(struct Node **head, int position) {
    struct Node *temp = *head;
    for (int i = 0; i < position - 1; i++) {
        temp = temp->next;
    }
    temp->next = temp->next->next;  // MEMORY LEAK! Node not freed
}
```

Solution:

```
void deleteNode(struct Node **head, int position) {
    struct Node *temp = *head;
```

```
    for (int i = 0; i < position - 1; i++) {
        temp = temp->next;
    }
    struct Node *nodeToDelete = temp->next;  // Save reference
    temp->next = nodeToDelete->next;
    free(nodeToDelete);  // Free memory
}
```

9.4 Infinite Loop in Circular List

Problem:

```
void display(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {  // Never NULL in circular list!
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

Solution:

```
void displayCircular(struct Node *head) {
    if (head == NULL) return;

    struct Node *temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);  // Check if back to head
}
```

9.5 Incorrect Pointer Updates

Problem:

```
void insertAtBeginning(struct Node *head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
```

```
    head = newNode;  // Only changes local copy!
}
```

Solution:

```
void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;  // Updates actual head pointer
}
```

9.6 Debugging Techniques

Print Node Addresses

```
void debugList(struct Node *head) {
    printf("\n=== Debug Info ===\n");
    struct Node *temp = head;
    int count = 0;

    while (temp != NULL) {
        printf("Node %d:\n", count++);
        printf("  Address: %p\n", (void*)temp);
        printf("  Data: %d\n", temp->data);
        printf("  Next: %p\n", (void*)temp->next);
        temp = temp->next;
    }
    printf("==================\n\n");
}
```

Check List Integrity

```
int checkListIntegrity(struct Node *head) {
    if (head == NULL) {
        return 1;  // Empty list is valid
    }

    struct Node *slow = head;
    struct Node *fast = head;
```

```
    // Check for cycles
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            printf("ERROR: Cycle detected!\n");
            return 0;
        }
    }

    printf("List integrity: OK\n");
    return 1;
}
```

Visualize List

```
void visualizeList(struct Node *head) {
    if (head == NULL) {
        printf("NULL\n");
        return;
    }

    struct Node *temp = head;
    printf("HEAD → ");

    while (temp != NULL) {
        printf("[%d]", temp->data);
        temp = temp->next;
        if (temp != NULL) {
            printf(" → ");
        }
    }
    printf(" → NULL\n");
}
```

9.7 Common Error Messages

Segmentation Fault:

- Dereferencing NULL pointer
- Accessing freed memory
- Buffer overflow

Memory Leak:

- Not freeing allocated memory
- Losing references to nodes
- Not calling free() on all nodes

Infinite Loop:

- Wrong termination condition
- Circular list without proper check
- Pointer not advancing

9.8 Preventive Measures

```
// Always check malloc return value
struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
if (newNode == NULL) {
    fprintf(stderr, "Memory allocation failed!\n");
    return NULL;
}

// Check for NULL before dereferencing
if (head != NULL) {
    // Safe to use head->data
}

// Free all nodes before exiting
void freeList(struct Node **head) {
    struct Node *temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

// Set pointers to NULL after freeing
free(node);
```

```
node = NULL;


// Use assertions for debugging
#include <assert.h>
assert(head != NULL);  // Program stops if false
```

Revision #1
Created 2025-10-27 05:07:26 UTC by DS
Updated 2025-10-27 05:07:49 UTC by DS