8. Common Pointer Pitfalls and Best Practices

8.1 Uninitialized Pointers

WRONG:

```
int *ptr;      // Uninitialized - contains garbage address
*ptr = 42;     // DANGER! Writing to unknown memory location
               // May cause segmentation fault
```

CORRECT:

```
int *ptr = NULL;   // Initialize to NULL
int num = 0;
ptr = &num;        // Assign valid address before use
*ptr = 42;         // Now safe to dereference
```

8.2 Dangling Pointers

A **dangling pointer** points to memory that has been freed or is no longer valid:

WRONG:

```
int *ptr;
{
    int num = 42;
    ptr = &num;
}  // num goes out of scope here
// ptr is now dangling - points to invalid memory
printf("%d", *ptr);  // DANGER! Undefined behavior
```

CORRECT:

```
int num = 42;
int *ptr = &num;
```

```
// Use ptr while num is in scope
printf("%d", *ptr);  // Safe
```

Dangling Pointer with free():

```
int *ptr = (int*)malloc(sizeof(int));
*ptr = 42;
free(ptr);
// ptr is now dangling
printf("%d", *ptr);  // DANGER! Undefined behavior

// Better:
free(ptr);
ptr = NULL;  // Set to NULL after freeing
if (ptr != NULL) {
    printf("%d", *ptr);  // This check prevents the error
}
```

8.3 NULL Pointer Dereference

WRONG:

```
int *ptr = NULL;
*ptr = 42;  // CRASH! Cannot dereference NULL pointer
```

CORRECT:

```
int *ptr = NULL;

if (ptr != NULL) {  // Always check before dereferencing
    *ptr = 42;
} else {
    printf("Error: NULL pointer\n");
}
```

8.4 Array Bounds with Pointers

WRONG:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
int value = *(ptr + 10);  // Out of bounds! Undefined behavior
```

CORRECT:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
int size = 5;

for (int i = 0; i < size; i++) {
    printf("%d ", *(ptr + i));  // Safe: within bounds
}
```

8.5 Returning Pointer to Local Variable

WRONG:

```
int* createNumber() {
    int num = 42;
    return &num;  // DANGER! num is destroyed after function returns
}

int main() {
    int *ptr = createNumber();
    printf("%d", *ptr);  // Undefined behavior - dangling pointer
    return 0;
}
```

CORRECT - Using Dynamic Allocation:

```
int* createNumber() {
    int *num = (int*)malloc(sizeof(int));
    *num = 42;
    return num;  // Safe - memory persists
}

int main() {
    int *ptr = createNumber();
    printf("%d", *ptr);
```

```
    free(ptr);  // Don't forget to free!
    return 0;
}
```

CORRECT - Using Static Variable:

```
int* createNumber() {
    static int num = 42;  // Static - persists after function returns
    return &num;
}
```

8.6 Best Practices Summary

1. **Always initialize pointers**

   ```
   int *ptr = NULL;  // Good
   int *ptr;         // Bad
   ```

2. **Check for NULL before dereferencing**

   ```
   if (ptr != NULL) {
       *ptr = value;
   }
   ```

3. **Set pointers to NULL after freeing**

   ```
   free(ptr);
   ptr = NULL;
   ```

4. **Be careful with pointer arithmetic**

   ```
   // Ensure you don't go out of array bounds
   if (ptr + i < arr + size) {
       // Safe to access
   }
   ```

5. **Use const for pointers that shouldn't modify data**

   ```
   void printString(const char *str) {
       // str cannot be used to modify the string
   }
   ```

6. **Match every malloc with free**

```
int *ptr = (int*)malloc(100 * sizeof(int));
// Use ptr...
free(ptr);
```

Revision #1
Created 2025-10-01 03:30:10 UTC by DS
Updated 2025-10-01 03:30:25 UTC by DS