c_programming_session_6

اللهمَّ علِّمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علمًا. وافتح علينا فتحًا عظيمًا.

Tags: c programming
Status: #Adult

C Programming Session 6 - Dynamic Memory and Data Structures

Dynamic Memory Management

Memory Allocation Functions Overview

more on this at Dynamic Memory Allocation in C

Function	Purpose	Initialization	Parameters
`malloc()`	Allocate single block	Uninitialized (garbage)	Size in bytes
`calloc()`	Allocate multiple blocks	Zero-initialized	Number of blocks, size per block
`realloc()`	Resize existing block	Preserves existing data	Pointer, new size
`free()`	Deallocate memory	N/A	Pointer to allocated memory

malloc() - Memory Allocation

Syntax: `pointer = (cast_type*) malloc(size_in_bytes)`

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

int main() {
    uint8_t *ptr;
```

```c
    // Allocate 10 bytes
    ptr = (uint8_t*) malloc(10 * sizeof(uint8_t));

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Use allocated memory
    for (int i = 0; i < 10; i++) {
        ptr[i] = i * 2;
    }

    // Print values
    for (int i = 0; i < 10; i++) {
        printf("ptr[%d] = %d\n", i, ptr[i]);
    }

    free(ptr);  // Always free allocated memory
    return 0;
}
```

calloc() - Contiguous Allocation

Syntax: `pointer = (cast_type*) calloc(num_blocks, size_of_each_block)`

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

int main() {
    uint32_t *ptr;

    // Allocate array of 12 uint32_t elements (all initialized to 0)
    ptr = (uint32_t*) calloc(12, sizeof(uint32_t));

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Values are already initialized to 0
    printf("Initial values (should all be 0):\n");
    for (int i = 0; i < 5; i++) {
```

```c
        printf("ptr[%d] = %u\n", i, ptr[i]);
    }

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

malloc() vs calloc() Comparison

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

int main() {
    int *malloc_ptr, *calloc_ptr;

    // malloc - uninitialized memory
    malloc_ptr = (int*) malloc(5 * sizeof(int));
    printf("malloc values (garbage):\n");
    for (int i = 0; i < 5; i++) {
        printf("malloc_ptr[%d] = %d\n", i, malloc_ptr[i]);
    }

    // calloc - zero-initialized memory
    calloc_ptr = (int*) calloc(5, sizeof(int));
    printf("\ncalloc values (initialized to 0):\n");
    for (int i = 0; i < 5; i++) {
        printf("calloc_ptr[%d] = %d\n", i, calloc_ptr[i]);
    }

    free(malloc_ptr);
    free(calloc_ptr);
    return 0;
}
```

realloc() - Resize Memory Block

Syntax: `new_pointer = realloc(old_pointer, new_size_in_bytes)`

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

int main() {
    int *ptr;
```

```c
    // Initial allocation: 5 integers
    ptr = (int*) malloc(5 * sizeof(int));

    // Fill with data
    for (int i = 0; i < 5; i++) {
        ptr[i] = i + 1;
    }

    printf("Original array (5 elements):\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Expand to 10 integers
    ptr = (int*) realloc(ptr, 10 * sizeof(int));

    if (ptr == NULL) {
        printf("Reallocation failed!\n");
        return 1;
    }

    // Fill new elements
    for (int i = 5; i < 10; i++) {
        ptr[i] = i + 1;
    }

    printf("Expanded array (10 elements):\n");
    for (int i = 0; i < 10; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

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

free() - Memory Deallocation

```c
void free(void *pointer);
```

Important Notes:

- Always call `free()` for every `malloc()`, `calloc()`, or `realloc()`
- After `free()`, the pointer becomes invalid (dangling pointer)
- Never use freed memory
- Set pointer to NULL after freeing (good practice)

```
int *ptr = (int*) malloc(10 * sizeof(int));
// ... use ptr ...
free(ptr);
ptr = NULL;  // Prevent accidental use of freed memory
```

Memory Management Best Practices

Error Handling

```
int *create_array(int size) {
    int *arr = (int*) malloc(size * sizeof(int));

    if (arr == NULL) {
        fprintf(stderr, "Error: Memory allocation failed for %d
integers\n", size);
        return NULL;
    }

    return arr;
}

int main() {
    int *numbers = create_array(1000);

    if (numbers == NULL) {
        return 1;  // Exit if allocation failed
    }

    // Use the array...

    free(numbers);
    numbers = NULL;
    return 0;
}
```

Memory Leaks Prevention

```c
// WRONG: Memory leak
int *allocate_memory() {
    int *ptr = (int*) malloc(100 * sizeof(int));
    return ptr;  // Caller must remember to free
}

// BETTER: Clear ownership
int *create_buffer(int size, int **buffer) {
    *buffer = (int*) malloc(size * sizeof(int));
    return *buffer;  // Caller knows they own the memory
}

// BEST: RAII-style with cleanup function
typedef struct {
    int *data;
    int size;
} IntArray;

IntArray* create_int_array(int size) {
    IntArray *arr = (IntArray*) malloc(sizeof(IntArray));
    if (!arr) return NULL;

    arr->data = (int*) malloc(size * sizeof(int));
    if (!arr->data) {
        free(arr);
        return NULL;
    }

    arr->size = size;
    return arr;
}

void destroy_int_array(IntArray *arr) {
    if (arr) {
        free(arr->data);
        free(arr);
    }
}
```

Bubble Sort with Dynamic Memory

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

void bubble_sort(uint32_t *array, int size);
void print_array(uint32_t *array, int size);

int main() {
    uint32_t *ptr;
    const int SIZE = 10;

    // Allocate memory for 10 integers
    ptr = (uint32_t*) calloc(SIZE, sizeof(uint32_t));
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Get input from user
    printf("Enter %d numbers:\n", SIZE);
    for (int i = 0; i < SIZE; i++) {
        printf("Number %d: ", i + 1);
        scanf("%u", &ptr[i]);
    }

    printf("\nBefore sorting:\n");
    print_array(ptr, SIZE);

    // Sort the array
    bubble_sort(ptr, SIZE);

    printf("\nAfter sorting:\n");
    print_array(ptr, SIZE);

    free(ptr);
    return 0;
}

void bubble_sort(uint32_t *array, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                // Swap elements
                uint32_t temp = array[j];
```

```c
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

void print_array(uint32_t *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%u ", array[i]);
    }
    printf("\n");
}
```

Linked Lists

more on this at Linked List in C

Basic Node Structure

```c
typedef struct Node_type Node;
struct Node_type {
    uint32_t data;
    Node *next;
};
```

Linked List Operations

Node Creation

```c
Node* create_node(uint32_t data) {
    Node *new_node = (Node*) malloc(sizeof(Node));

    if (new_node == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }

    new_node->data = data;
    new_node->next = NULL;
```

```
    return new_node;
}
```

Insert at End

```
Node* insert_at_end(Node *head, uint32_t data) {
    Node *new_node = create_node(data);
    if (new_node == NULL) {
        return head;  // Failed to create node
    }

    if (head == NULL) {
        return new_node;  // First node
    }

    // Traverse to end
    Node *current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = new_node;
    return head;
}
```

Print List

```
void print_list(Node *head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    Node *current = head;
    printf("List contents: ");
    while (current != NULL) {
        printf("%u ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

Free List

```c
void free_list(Node *head) {
    Node *current = head;
    Node *next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}
```

Complete Linked List Program

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

typedef struct Node_type {
    uint32_t data;
    struct Node_type *next;
} Node;

Node* create_node(uint32_t data);
Node* insert_at_end(Node *head, uint32_t data);
void print_list(Node *head);
void free_list(Node *head);

int main() {
    Node *head = NULL;
    int choice, data;

    while (1) {
        printf("\nLinked List Menu:\n");
        printf("0: Add new node\n");
        printf("1: Print list\n");
        printf("2: Exit\n");
        printf("Choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 0:
                printf("Enter data: ");
```

```c
                scanf("%d", &data);
                head = insert_at_end(head, data);
                printf("Node added successfully\n");
                break;

            case 1:
                print_list(head);
                break;

            case 2:
                printf("Freeing memory and exiting...\n");
                free_list(head);
                printf("Thank you! Goodbye.\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
                break;
        }
    }

    return 0;
}

Node* create_node(uint32_t data) {
    Node *new_node = (Node*) malloc(sizeof(Node));
    if (new_node == NULL) {
        printf("Memory allocation failed!\n");
        return NULL;
    }

    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

Node* insert_at_end(Node *head, uint32_t data) {
    Node *new_node = create_node(data);
    if (new_node == NULL) {
        return head;
    }

    if (head == NULL) {
        return new_node;
```

```
    }

    Node *current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    current->next = new_node;
    return head;
}

void print_list(Node *head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    Node *current = head;
    printf("List contents: ");
    while (current != NULL) {
        printf("%u ", current->data);
        current = current->next;
    }
    printf("\n");
}

void free_list(Node *head) {
    Node *current = head;
    Node *next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}
```

Advanced Linked List Operations

Insert at Beginning

```
Node* insert_at_beginning(Node *head, uint32_t data) {
    Node *new_node = create_node(data);
```

```c
    if (new_node == NULL) {
        return head;
    }

    new_node->next = head;
    return new_node;  // New head
}
```

Delete by Value

```c
Node* delete_by_value(Node *head, uint32_t value) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }

    // Delete head node
    if (head->data == value) {
        Node *temp = head;
        head = head->next;
        free(temp);
        printf("Node with value %u deleted\n", value);
        return head;
    }

    // Search for node to delete
    Node *current = head;
    while (current->next != NULL && current->next->data != value) {
        current = current->next;
    }

    if (current->next == NULL) {
        printf("Value %u not found in list\n", value);
        return head;
    }

    Node *to_delete = current->next;
    current->next = to_delete->next;
    free(to_delete);
    printf("Node with value %u deleted\n", value);

    return head;
}
```

Count Nodes

```
int count_nodes(Node *head) {
    int count = 0;
    Node *current = head;

    while (current != NULL) {
        count++;
        current = current->next;
    }

    return count;
}
```

Search for Value

```
int search_value(Node *head, uint32_t value) {
    Node *current = head;
    int position = 0;

    while (current != NULL) {
        if (current->data == value) {
            return position;  // Found at this position
        }
        current = current->next;
        position++;
    }

    return -1;  // Not found
}
```

Memory Allocation Patterns

Dynamic Array Resizing

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

typedef struct {
    int *data;
    int size;
    int capacity;
```

```c
} DynamicArray;

DynamicArray* create_dynamic_array(int initial_capacity) {
    DynamicArray *arr = (DynamicArray*) malloc(sizeof(DynamicArray));
    if (!arr) return NULL;

    arr->data = (int*) malloc(initial_capacity * sizeof(int));
    if (!arr->data) {
        free(arr);
        return NULL;
    }

    arr->size = 0;
    arr->capacity = initial_capacity;
    return arr;
}

int push_back(DynamicArray *arr, int value) {
    if (arr->size >= arr->capacity) {
        // Need to resize
        int new_capacity = arr->capacity * 2;
        int *new_data = (int*) realloc(arr->data, new_capacity *
sizeof(int));

        if (!new_data) {
            printf("Failed to resize array\n");
            return 0;  // Failed
        }

        arr->data = new_data;
        arr->capacity = new_capacity;
        printf("Array resized to capacity %d\n", new_capacity);
    }

    arr->data[arr->size] = value;
    arr->size++;
    return 1;  // Success
}

void print_dynamic_array(DynamicArray *arr) {
    printf("Array (size %d, capacity %d): ", arr->size, arr->capacity);
    for (int i = 0; i < arr->size; i++) {
        printf("%d ", arr->data[i]);
    }
```

```c
    printf("\n");
}

void destroy_dynamic_array(DynamicArray *arr) {
    if (arr) {
        free(arr->data);
        free(arr);
    }
}

int main() {
    DynamicArray *arr = create_dynamic_array(2);
    if (!arr) {
        printf("Failed to create array\n");
        return 1;
    }

    // Add elements, triggering resizes
    for (int i = 1; i <= 10; i++) {
        push_back(arr, i);
        print_dynamic_array(arr);
    }

    destroy_dynamic_array(arr);
    return 0;
}
```

Common Memory Management Errors

Double Free

```c
// WRONG: Double free error
int *ptr = (int*) malloc(sizeof(int));
free(ptr);
free(ptr);  // ERROR: Already freed!

// CORRECT: Set to NULL after free
int *ptr = (int*) malloc(sizeof(int));
free(ptr);
ptr = NULL;
if (ptr != NULL) {  // Safe check
```

```c
    free(ptr);
}
```

Use After Free

```c
// WRONG: Using freed memory
int *ptr = (int*) malloc(sizeof(int));
*ptr = 42;
free(ptr);
printf("%d\n", *ptr);  // ERROR: Using freed memory!

// CORRECT: Don't use after free
int *ptr = (int*) malloc(sizeof(int));
*ptr = 42;
printf("%d\n", *ptr);  // Use before free
free(ptr);
ptr = NULL;  // Prevent accidental use
```

Memory Leak

```c
// WRONG: Memory leak
void function() {
    int *ptr = (int*) malloc(100 * sizeof(int));
    // ... some code ...
    return;  // ERROR: Never freed ptr!
}

// CORRECT: Always free allocated memory
void function() {
    int *ptr = (int*) malloc(100 * sizeof(int));
    if (!ptr) return;

    // ... some code ...

    free(ptr);  // Always free before return
}
```

Performance Considerations

Allocation Performance Comparison

Operation	Time Complexity	Notes
`malloc()`	O(1) - O(log n)	Depends on allocator implementation
`calloc()`	O(n)	Must initialize all bytes to zero
`realloc()`	O(1) - O(n)	May need to copy data if block moves
`free()`	O(1)	Usually constant time

Memory Usage Comparison

```
// Static array: compile-time allocation, stack memory
int static_array[1000];  // 4000 bytes on stack

// Dynamic array: runtime allocation, heap memory
int *dynamic_array = (int*) malloc(1000 * sizeof(int));  // 4000 bytes on heap
```

Aspect	Static Array	Dynamic Array
Memory Location	Stack	Heap
Size Determination	Compile time	Runtime
Lifetime	Automatic cleanup	Manual cleanup required
Performance	Faster access	Slightly slower access
Flexibility	Fixed size	Variable size

Best Practices Summary

1. **Always check for NULL** after allocation
2. **Free every allocated block** exactly once
3. **Set pointers to NULL** after freeing
4. **Match every malloc/calloc with free**
5. **Use valgrind or similar tools** to detect memory errors
6. **Consider using static arrays** for fixed-size data
7. **Initialize allocated memory** if needed
8. **Handle allocation failures gracefully**

Practice Exercises

Exercise 1: Dynamic String Array

Create a program that dynamically allocates an array of strings, allowing the user to input names and then sorts them alphabetically.

Exercise 2: Linked List with Search

Extend the linked list implementation to include search, delete, and insert at specific position functions.

Exercise 3: Memory Pool

Implement a simple memory pool allocator that pre-allocates a large block and manages smaller allocations within it.

Exercise 4: Stack Implementation

Use dynamic memory to implement a stack data structure with push, pop, and peek operations.

Next Session Preview

This concludes the C programming fundamentals sessions. The knowledge gained here forms the foundation for:

- **Embedded Systems Programming:** Direct hardware control
- **Real-time Systems:** Timing-critical applications
- **Microcontroller Programming:** AVR, ARM, PIC development
- **Device Driver Development:** Low-level system programming
- **IoT Applications:** Connected embedded devices

The skills in memory management, data structures, and low-level programming are essential for embedded systems development where resources are constrained and efficiency is paramount.