c_programming_practice_files

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

Author: Fares Hesham Mahmoud
Tags: c programming
Status: #Adult

c_programming_practice_files

Heading1

C Programming Practice Files Documentation

Practice File 1: sun_6_jul_part2.c - Ternary Operator

Overview

Demonstrates complex ternary operator usage and nested conditional expressions.

Code Analysis

```c
#include <stdio.h>

int main() {
    int var = 75;
    int var2 = 56;
    int num;

    // Complex nested ternary operation
    num = sizeof(var) ? (var2 > 23 ? ((var == 75) ? 'a' : 0) : 0) : 0;
    printf("%d", num);

    return 0;
}
```

Step-by-Step Evaluation

1. `sizeof(var)` returns 4 (size of int), which is truthy
2. `var2 > 23` → `56 > 23` → true
3. `var == 75` → `75 == 75` → true
4. Final result: `'a'` (ASCII value 97)

Output: `97`

Alternative Switch Implementation (Commented)

```c
// ID-based welcome system
int id;
printf("Enter ID: ");
scanf("%d", &id);

switch (id) {
    case 1:
        printf("Welcome Fares");
        break;
    case 2:
        printf("Welcome Ahmed");
        break;
    case 3:
        printf("Welcome Mohammed");
        break;
    default:
        printf("Invalid ID");
        break;
}
```

Practice File 2: mon_7_jul_practice.c - Factorial with Recursion

Overview

Recursive factorial calculation demonstrating function calls and base cases.

Code Analysis

```c
#include <stdio.h>

int fact(int x);
```

```c
int x, n;

int main() {
    int y = 0;
    printf("Enter x value: ");
    scanf("%d", &n);
    y = fact(n);
    printf("%d", y);
    return 0;
}

int fact(int n) {
    if (n == 0 || n == 1) {
        x = 1;
    } else {
        x = n * fact(n - 1);
    }
    return x;
}
```

Execution Flow

For input `n = 5`:

```
fact(5) → 5 * fact(4) → 5 * 4 * fact(3) → 5 * 4 * 3 * fact(2) → 5 * 4 * 3 * 2 * fact(1) → 5 * 4 * 3 * 2 * 1 = 120
```

Issues and Improvements

- Global variable `x`: Should use local variables or direct return
- Variable naming: `n` is shadowed in function parameter

Improved version:

```c
int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Practice File 3: thu_8_jul_practice.c - Array Sum with Pointers

Overview

Calculates sum of array elements using pointer arithmetic.

Code Analysis

```c
#include <stdio.h>

int sum_array(int y);
int x, y, sum = 0;
int array[10] = {1, 2, 7, 8, 19, 32, 45, 67, 69, 1000};

int main() {
    x = sum_array(y);
    printf("Sum is %d", x);
    return 0;
}

int sum_array(int y) {
    int *ptr1 = array;
    for (int i = 0; i < 10; i++) {
        sum = sum + ptr1[i];
    }
    return sum;  // Missing return statement
}
```

Issues and Fixes

1. Missing return statement
2. Unused parameter `y`
3. Global `sum` variable - should be local

Corrected version:

```c
int sum_array(int arr[], int size) {
    int *ptr = arr;
    int total = 0;

    for (int i = 0; i < size; i++) {
```

```
        total += ptr[i];  // or total += *(ptr + i);
    }

    return total;
}

int main() {
    int array[10] = {1, 2, 7, 8, 19, 32, 45, 67, 69, 1000};
    int result = sum_array(array, 10);
    printf("Sum is %d\n", result);
    return 0;
}
```

Practice File 4: thu_10_jul_practice.c - Bit Manipulation Menu

Overview

Interactive program for bit manipulation operations (set, clear, toggle).

Code Analysis

```
#include <stdio.h>
#include "math_Fares.h"  // Contains bit manipulation macros

int main() {
    do {
        int x, bit, operation, result;

        printf("Enter number: ");
        scanf("%d", &x);
        printf("Enter bit position: ");
        scanf("%d", &bit);
        printf("Choose operation (1:set, 2:clear, 3:toggle): ");
        scanf("%d", &operation);

        switch (operation) {
            case 1:
                result = SET_BIT(x, bit);
                printf("Number after SET: %d\n", result);
                break;
            case 2:
                result = CLEAR_BIT(x, bit);
```

```
                printf("Number after CLEAR: %d\n", result);
                break;
            case 3:
                result = TOGGLE_BIT(x, bit);
                printf("Number after TOGGLE: %d\n", result);
                break;
            default:
                printf("Invalid operation code.\n");
                break;
        }
    } while (1);

    return 0;
}
```

Required Header File (math_Fares.h)

```
#ifndef MATH_FARES_H
#define MATH_FARES_H

#define SET_BIT(var, bit)    ((var) |= (1 << (bit)))
#define CLEAR_BIT(var, bit)  ((var) &= ~(1 << (bit)))
#define TOGGLE_BIT(var, bit) ((var) ^= (1 << (bit)))

#endif
```

Example Usage

```
Input: number = 5 (binary: 101), bit = 1, operation = 1 (set)
Output: 7 (binary: 111)

Input: number = 7 (binary: 111), bit = 2, operation = 2 (clear)
Output: 3 (binary: 011)

Input: number = 5 (binary: 101), bit = 0, operation = 3 (toggle)
Output: 4 (binary: 100)
```

Practice File 5: thu_10_jul_practice2.c - Bit Manipulation Demo

Overview

Demonstrates bit manipulation operations with fixed values.

Code Analysis

```c
#include <stdio.h>
#include "math_Fares.h"

int main() {
    int value = 7;  // Binary: 0111

    printf("Initial value     = %d\n", value);

    SET_BIT(value, 2);
    printf("After SET_BIT(2)  = %d\n", value);    // 0111 | 0100 = 0111
(no change)

    CLEAR_BIT(value, 2);
    printf("After CLEAR_BIT(2)= %d\n", value);    // 0111 & ~0100 = 0011
= 3

    TOGGLE_BIT(value, 3);
    printf("After TOGGLE_BIT(3)= %d\n", value);   // 0011 ^ 1000 = 1011
= 11

    TOGGLE_BIT(value, 3);
    printf("After TOGGLE_BIT(3)= %d\n", value);   // 1011 ^ 1000 = 0011
= 3

    return 0;
}
```

Expected Output

```
Initial value     = 7
After SET_BIT(2)  = 7
After CLEAR_BIT(2)= 3
After TOGGLE_BIT(3)= 11
After TOGGLE_BIT(3)= 3
```

Practice File 6: Fri_11_jul_practice.c - Bubble Sort with Dynamic Memory

Overview

Implements bubble sort on dynamically allocated array with proper memory management.

Code Analysis

```c
#include <stdio.h>
#include <stdlib.h>
#include "(NTI)standard.h"  // Custom standard definitions

void bubble_sort(uint32 *point, int size);
void print_sorted(uint32 *point, int size);

int main() {
    uint32 *ptr;
    ptr = (uint32 *)calloc(10, sizeof(uint32));

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

    // Input phase
    for (int i = 0; i < 10; i++) {
        printf("Enter number %d: ", i + 1);
        scanf("%d", &ptr[i]);
    }

    bubble_sort(ptr, 10);
    printf("The sorted array is: ");
    print_sorted(ptr, 10);

    free(ptr);
    return 0;
}

void bubble_sort(uint32 *point, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int k = 0; k < size - i - 1; k++) {
```

```
            if (point[k] > point[k + 1]) {
                uint32 t = point[k];
                point[k] = point[k + 1];
                point[k + 1] = t;
            }
        }
    }
}

void print_sorted(uint32 *point, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", point[i]);
    }
    printf("\n");
}
```

Algorithm Analysis

- Time Complexity: $O(n^2)$ - nested loops
- Space Complexity: O(1) - in-place sorting
- Memory Management: Uses `calloc()` for zero-initialization

Example Execution

```
Input: 64, 34, 25, 12, 22, 11, 90, 88, 76, 50
Output: 11 12 22 25 34 50 64 76 88 90
```

Practice File 7: Fri_11_jul_practice_2.c - Linked List Implementation (Incomplete)

Overview

Incomplete linked list implementation with menu system structure.

Code Analysis

```
#include <stdio.h>
#include <stdlib.h>
#include "(NTI)standard.h"

typedef struct Node_type Node;
```

```
struct Node_type {
    uint32 data;
    Node *Next;
};

Node *Start = NULL;

int main() {
    do {
        int choice, data;
        printf("Choice (0:New node, 1:Print list, 2:Exit): ");
        scanf("%d", &choice);

        switch (choice) {
            case 0:
                printf("Enter data: ");
                scanf("%d", &data);
                // Missing: insert_node(data);
                printf("Node added\n");
                break;
            case 1:
                // Missing: print_list();
                printf("List printed\n");
                break;
            case 2:
                printf("\nThank you\nGood Bye");
                return 0;
            default:
                printf("Invalid choice, please try again.\n");
                break;
        }
    } while (1);
}

// Incomplete function implementations
Node* create_node(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    newNode->data = data;
    newNode->Next = NULL;  // Note: should be NULL, not Null
    return newNode;  // Missing semicolon
```

```
}

void insert_node(int data) {
    // Implementation missing
}
```

Issues Found

1. Typo: `Null` should be `NULL`
2. Missing semicolon in `create_node` return statement
3. Incomplete functions: `insert_node()` and `print_list()`
4. Wrong case number in switch (case 3 should be case 2)

Complete Implementation

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

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

Node *head = NULL;

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;
}

void insert_node(uint32_t data) {
    Node* new_node = create_node(data);
    if (new_node == NULL) return;

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

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

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

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

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

int main() {
    int choice, data;

    do {
        printf("\nMenu:\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: ");
                scanf("%d", &data);
                insert_node(data);
                printf("Node added successfully\n");
                break;
            case 1:
                print_list();
                break;
            case 2:
                free_list();
                printf("\nThank you\nGood Bye\n");
                return 0;
            default:
                printf("Invalid choice, please try again.\n");
                break;
        }
    } while (1);
}
```

Common Issues and Solutions

Issue 1: Memory Management

Problem: Not checking for allocation failure

```
// Wrong
int *ptr = malloc(size);
ptr[0] = 10;  // Potential segmentation fault
```

Solution: Always check for NULL

```
// Correct
int *ptr = malloc(size);
if (ptr == NULL) {
    printf("Allocation failed\n");
    return -1;
}
ptr[0] = 10;  // Safe to use
```

Issue 2: Global Variables

Problem: Excessive use of global variables

```
// Problematic
int sum = 0;  // Global variable

int calculate() {
    sum += 10;  // Modifies global state
    return sum;
}
```

Solution: Use local variables and parameters

```
// Better
int calculate(int current_sum) {
    return current_sum + 10;
}
```

Issue 3: Missing Return Statements

Problem: Functions that should return values don't

```
// Wrong
int add(int a, int b) {
    int result = a + b;
    // Missing return statement
}
```

Solution: Always return appropriate values

```
// Correct
int add(int a, int b) {
    int result = a + b;
    return result;
}
```

Issue 4: Bit Manipulation Macros

Problem: Unprotected macro parameters

```
// Dangerous
#define SET_BIT(var, bit) var |= 1 << bit

// Can cause issues with: SET_BIT(x + y, 2)
// Expands to: x + y |= 1 << 2  (wrong precedence)
```

Solution: Always parenthesize macro parameters

```
// Safe
#define SET_BIT(var, bit) ((var) |= (1 << (bit)))
```

Best Practices Demonstrated

1. Error Handling

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

2. Memory Cleanup

```
free(ptr);
ptr = NULL;  // Prevent dangling pointer
```

3. Function Modularity

```
// Separate concerns into different functions
void bubble_sort(uint32 *array, int size);
void print_array(uint32 *array, int size);
```

4. Input Validation

```
if (choice < 0 || choice > 2) {
    printf("Invalid choice\n");
    continue;
}
```

5. Clear Variable Names

```c
// Good naming
int student_count;
float average_grade;
Node *current_node;

// Poor naming
int n;
float avg;
Node *p;
```

Performance Analysis

Time Complexity Summary

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Linear Search	O(1)	O(n)	O(n)
Linked List Insert	O(1)	O(n)	O(n)
Array Access	O(1)	O(1)	O(1)

Space Complexity Summary

Data Structure	Space Complexity	Notes
Static Array	O(n)	Stack allocation
Dynamic Array	O(n)	Heap allocation
Linked List	O(n)	Extra pointer overhead

Learning Outcomes

From these practice files, students learn:

1. **Memory Management:** Dynamic allocation, deallocation, error handling
2. **Data Structures:** Arrays, linked lists, basic operations
3. **Algorithms:** Sorting, searching, basic complexity analysis
4. **Bit Manipulation:** Setting, clearing, toggling bits

5. **Function Design:** Modular programming, parameter passing
6. **Error Handling:** Defensive programming practices
7. **Code Organization:** Header files, function prototypes

Recommendations for Improvement

1. **Use consistent naming conventions**
2. **Add comprehensive error checking**
3. **Implement complete function definitions**
4. **Add input validation**
5. **Use proper memory cleanup**
6. **Follow single responsibility principle**
7. **Add meaningful comments**
8. **Test edge cases**

These practice files serve as excellent examples of both common programming patterns and typical mistakes made by beginning C programmers, providing valuable learning opportunities for embedded systems development.