C Programs for Learning Basics, Data Structures, and Pointers

DS2040, Satyajit Das

January 9, 2025

1 Introduction

This document contains 15 C programs designed to help you understand fundamental C programming concepts, basic data structures, and pointers. Each program is accompanied by:

- A short description
- The source code
- Instructions for compilation and execution
- Key concepts

2 Programs

2.1 1. Hello World

Purpose: The classic introductory program—verifies your environment is set up and shows basic C syntax.

Listing 1: hello.c

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Instructions:

- Compile: `gcc hello.c -o hello`
- Run: `./hello`

Key Concepts:

- Basic structure of a C program (`#include`, `main()`, `return 0;`)
- Printing output to the console.

2.2 2. Simple Input and Output

Purpose: Demonstrates reading integers from the user and printing results.

Listing 2: simple_io.c

```c
#include <stdio.h>

int main(void) {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);
    printf("You entered: %d\n", number);
    return 0;
}
```

Instructions:

- Compile: `gcc simple_io.c -o simple_io`
- Run: `./simple_io`

Key Concepts:

- Using `scanf()` and `printf()`.
- Relationship between format specifiers (e.g., `%d`) and variable types.

2.3 3. If-Else and Loops

Purpose: Illustrates conditional logic and basic loops in C.

Listing 3: control_flow.c

```c
#include <stdio.h>

int main(void) {
    int i, sum = 0;

    for (i = 1; i <= 5; i++) {
        if (i % 2 == 0) {
            printf("%d is even\n", i);
        } else {
            printf("%d is odd\n", i);
        }
        sum += i;
    }

    printf("Sum of numbers 1 to 5 is %d\n", sum);
    return 0;
}
```

Instructions:

- Compile: `gcc control_flow.c -o control_flow`
- Run: `./control_flow`

Key Concepts:

- `for` loop, `if` statement, % operator for checking even/odd.
- Accumulating a sum in a loop.

2.4 4. Arrays and Summation

Purpose: Shows how to handle arrays, sum their elements, and print results.

Listing 4: array_sum.c

```c
#include <stdio.h>

#define SIZE 5

int main(void) {
    int arr[SIZE] = {1, 2, 3, 4, 5};
    int sum = 0;

    for (int i = 0; i < SIZE; i++) {
        sum += arr[i];
    }

    printf("Sum of array elements = %d\n", sum);
    return 0;
}
```

Instructions:

- Compile: `gcc array_sum.c -o array_sum`
- Run: `./array_sum`

Key Concepts:

- Declaring and initializing arrays.
- Iterating through arrays with a loop.
- Accumulating values.

2.5 5. Pointer Basics

Purpose: Demonstrates how to use pointers to print variable addresses and values.

Listing 5: pointer_basics.c

```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *ptr = &x; // pointer to x

    printf("Value of x: %d\n", x);
```

```c
    printf("Address of x: %p\n", (void*)&x);
    printf("Value of ptr: %p\n", (void*)ptr);
    printf("Value pointed by ptr: %d\n", *ptr);

    return 0;
}
```

Instructions:

- Compile: `gcc pointer_basics.c -o pointer_basics`
- Run: `./pointer_basics`

Key Concepts:

- Declaring pointers (`int *ptr`).
- Dereferencing (`*ptr`).
- Difference between address and value.

2.6 6. Pointer Arithmetic

Purpose: Explores how incrementing pointers relates to the size of their data type.

Listing 6: pointer_arithmetic.c

```c
#include <stdio.h>

int main(void) {
    int arr[3] = {10, 20, 30};
    int *p = arr; // points to arr[0]

    printf("p points to arr[0]: value = %d\n", *p);
    p++; // Move to arr[1]
    printf("p now points to arr[1]: value = %d\n", *p);
    p++; // Move to arr[2]
    printf("p now points to arr[2]: value = %d\n", *p);

    return 0;
}
```

Instructions:

- Compile: `gcc pointer_arithmetic.c -o pointer_arithmetic`
- Run: `./pointer_arithmetic`

Key Concepts:

- Relationship between pointer increments and array indexing.
- Each increment of an `int *` moves `sizeof(int)` bytes in memory.

2.7 7. Swapping Values with Pointers

Purpose: Demonstrates using pointers in function calls to swap two variables.

Listing 7: swap.c

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int x = 5, y = 10;
    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

Instructions:

- Compile: `gcc swap.c -o swap`
- Run: `./swap`

Key Concepts:

- Passing addresses to a function.
- Dereferencing to modify original variables.

2.8 8. Dynamic Memory Allocation

Purpose: Explores `malloc()` and `free()` to dynamically manage memory for arrays.

Listing 8: dynamic_allocation.c

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

int main(void) {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Store values
```

```c
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    // Print values
    printf("Allocated array elements:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr); // Release the allocated memory
    return 0;
}
```

Instructions:

- Compile: `gcc dynamic_allocation.c -o dynamic_allocation`
- Run: `./dynamic_allocation`

Key Concepts:

- `malloc()` for dynamic memory allocation.
- Checking for `NULL` to avoid segmentation faults.
- `free()` to release allocated memory.

2.9 9. Structure Basics

Purpose: Shows how to define and use a `struct` in C.

Listing 9: structure_basics.c

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

struct Student {
    char name[50];
    int age;
};

int main(void) {
    struct Student s1;

    strcpy(s1.name, "Alice");
    s1.age = 20;

    printf("Student Name: %s, Age: %d\n", s1.name, s1.age);

    return 0;
}
```

Instructions:

- Compile: `gcc structure_basics.c -o structure_basics`
- Run: `./structure_basics`

Key Concepts:

- Declaring a structure.
- Accessing structure members using the dot (`.`) operator.
- Using library functions (`strcpy`) to manipulate strings within structures.

2.10 10. Singly Linked List (Insertion & Print)

Purpose: Provides a foundational data structure using pointers—linked lists.

Listing 10: linked_list.c

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

struct Node {
    int data;
    struct Node *next;
};

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

void printList(struct Node *head) {
    struct Node *current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main(void) {
    struct Node *head = NULL;

    insertAtHead(&head, 30);
    insertAtHead(&head, 20);
    insertAtHead(&head, 10);

    printList(head);
    return 0;
}
```

Instructions:

- Compile: `gcc linked_list.c -o linked_list`
- Run: `./linked_list`

Key Concepts:

- Dynamic allocation for nodes.
- Maintaining a `head` pointer to a linked list.
- Inserting new nodes at the head of the list.

2.11 11. Stack (Array Implementation)

Purpose: Implements a simple stack using an array.

Listing 11: stack_array.c

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

#define MAX_SIZE 5

typedef struct {
    int top;
    int arr[MAX_SIZE];
} Stack;

void initStack(Stack *s) {
    s->top = -1;
}

int isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack overflow!\n");
        return;
    }
    s->arr[++(s->top)] = value;
}

int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow!\n");
        return -1;
    }
```

```c
    return s->arr[(s->top)--];
}

int main(void) {
    Stack myStack;
    initStack(&myStack);

    push(&myStack, 10);
    push(&myStack, 20);
    push(&myStack, 30);

    printf("Popped: %d\n", pop(&myStack));
    printf("Popped: %d\n", pop(&myStack));

    return 0;
}
```

Instructions:

- Compile: `gcc stack_array.c -o stack_array`
- Run: `./stack_array`

Key Concepts:

- LIFO (Last-In, First-Out) principle.
- Array-based stack with `top` index.
- Overflow and underflow checks.

2.12 12. Queue (Array Implementation)

Purpose: Implements a simple queue using an array.

Listing 12: queue_array.c

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

#define MAX_SIZE 5

typedef struct {
    int front, rear;
    int arr[MAX_SIZE];
} Queue;

void initQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isEmpty(Queue *q) {
    return q->front == -1;
```

```c
}

int isFull(Queue *q) {
    return (q->rear + 1) % MAX_SIZE == q->front;
}

void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full!\n");
        return;
    }
    if (q->front == -1)
        q->front = 0;
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->arr[q->rear] = value;
}

int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return -1;
    }
    int result = q->arr[q->front];
    if (q->front == q->rear) {
        // Only one element
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % MAX_SIZE;
    }
    return result;
}

int main(void) {
    Queue myQueue;
    initQueue(&myQueue);

    enqueue(&myQueue, 1);
    enqueue(&myQueue, 2);
    enqueue(&myQueue, 3);

    printf("Dequeued: %d\n", dequeue(&myQueue));
    printf("Dequeued: %d\n", dequeue(&myQueue));

    return 0;
}
```

Instructions:

- Compile: `gcc queue_array.c -o queue_array`
- Run: `./queue_array`

Key Concepts:

- FIFO (First-In, First-Out) principle.
- Circular indexing using `(rear + 1) % MAX_SIZE`.
- Checking for empty and full conditions.

2.13 13. Bubble Sort

Purpose: Illustrates a simple sorting algorithm operating on an array in-place.

Listing 13: bubble_sort.c

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main(void) {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    bubbleSort(arr, n);

    printf("Sorted array:\n");
    for(int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Instructions:

- Compile: `gcc bubble_sort.c -o bubble_sort`

- Run: `./bubble_sort`

Key Concepts:

- Nested loops and swapping adjacent elements.
- Time complexity: $\mathcal{O}(n^2)$ average/worst case.

2.14 14. Binary Search (Iterative)

Purpose: Demonstrates searching a sorted array in $\mathcal{O}(\log n)$ time.

Listing 14: binary_search.c

```c
#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int mid = (low + high) / 2;

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

int main(void) {
    int arr[] = {2, 4, 6, 8, 10, 12};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 8;

    int result = binarySearch(arr, size, target);
    if (result != -1) {
        printf("Element %d found at index %d.\n", target, result);
    } else {
        printf("Element %d not found.\n", target);
    }

    return 0;
}
```

Instructions:

- Compile: `gcc binary_search.c -o binary_search`
- Run: `./binary_search`

Key Concepts:

- Requires **sorted array** for correctness.
- Iterative approach to finding a target in $\mathcal{O}(\log n)$.

2.15 15. File I/O (Read and Write)

Purpose: Demonstrates reading from and writing to a file in C.

Listing 15: file_io.c

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

int main(void) {
    FILE *fp = fopen("output.txt", "w");
    if (fp == NULL) {
        printf("Error opening file for writing.\n");
        return 1;
    }
    fprintf(fp, "Hello, file!\n");
    fclose(fp);

    fp = fopen("output.txt", "r");
    if (fp == NULL) {
        printf("Error opening file for reading.\n");
        return 1;
    }

    char buffer[100];
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        printf("%s", buffer);
    }
    fclose(fp);

    return 0;
}
```

Instructions:

1. Compile: `gcc file_io.c -o file_io`
2. Run: `./file_io`
3. This will write `"Hello, file!"` to `output.txt` and then read it back.

Key Concepts:

- Using `FILE *` to handle file streams.
- `fopen`, `fprintf`, `fgets`, `fclose`.
- Error checking when opening/reading/writing files.

3 Tips for Running and Understanding the Programs

- **Compilation:** Use `gcc filename.c -o outputname` on Linux or macOS. On Windows, the command is similar, though you may end up with an `.exe` file.
- **Execution:** On UNIX-like systems, run `./outputname`. On Windows, run `outputname.exe` or `outputname`.
- **Experimentation:** Modify values, change array sizes, or add additional `printf` statements to see how behavior changes. Insert debugging printouts to trace loops or pointer operations.
- **Common Pitfalls:**
  - Not checking the return of `malloc()` for `NULL`.
  - Off-by-one errors in loops and array indexing.
  - Forgetting to free dynamically allocated memory, leading to leaks.
  - Mixing `scanf` and `fgets` without clearing input buffers.
- **Further Exploration:**
  - Explore advanced data structures (trees, graphs, hash tables).
  - Implement additional sorting algorithms (insertion sort, merge sort, quicksort).
  - Add error handling and input validation.
  - Use debuggers like `gdb` to step through code line by line.