Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO DATA STRUCTURES IN C

CHAPTER 1: UNDERSTANDING DATA STRUCTURES

What are Data Structures?

A **data structure** is a specialized format for organizing, managing, and storing data efficiently. In **C programming**, data structures enable developers to **process data effectively** for different computing tasks. They are fundamental building blocks of software applications and are widely used in areas such as **database management, operating systems, and artificial intelligence**.

Data structures help in **structuring data logically** so that it can be manipulated efficiently. For example, imagine a program that keeps track of student records. Using an **array** allows storing multiple records, while a **linked list** enables dynamic storage. By selecting the right data structure, programs can **execute faster and use memory efficiently**.

Why Are Data Structures Important?

1. **Efficiency** – They help in storing, accessing, and processing large amounts of data quickly.
2. **Memory Management** – Optimize memory usage by allocating memory dynamically.
3. **Real-world Applications** – Used in databases, web development, and networking.

4. **Algorithm Optimization** – The right data structure improves algorithm performance.

By mastering data structures, a programmer **gains deeper insights into memory management, problem-solving, and software optimization**.

CHAPTER 2: CLASSIFICATION OF DATA STRUCTURES

Types of Data Structures

Data structures in **C** are categorized into two major types:

1. **Primitive Data Structures** – These include int, char, float, and double, which hold a single value at a time.
2. **Non-Primitive Data Structures** – These store multiple values and are further divided into:
   - **Linear Data Structures** – Arrays, Linked Lists, Stacks, and Queues.
   - **Non-Linear Data Structures** – Trees and Graphs.

Comparison of Linear vs. Non-Linear Structures

Feature	Linear Data Structure	Non-Linear Data Structure
Memory Usage	Contiguous or sequential memory allocation	Uses linked memory representation
Complexity	Easier to implement	More complex in nature
Data Relation	Elements are arranged sequentially	Elements are arranged hierarchically

Examples	Arrays, Stacks, Queues, Linked Lists	Trees, Graphs

Selecting the right data structure depends on **problem requirements, memory constraints, and efficiency considerations**.

CHAPTER 3: ARRAYS – THE BASIC DATA STRUCTURE

What is an Array?

An **array** is a **collection of elements of the same data type**, stored at contiguous memory locations. It provides a way to store multiple values in a **single variable**, making it efficient for handling large amounts of data.

Example: Declaring and Using an Array

```
#include <stdio.h>


int main() {

  int numbers[5] = {10, 20, 30, 40, 50}; // Initializing an array


  for (int i = 0; i < 5; i++) {

    printf("Element at index %d: %d\n", i, numbers[i]);

  }


  return 0;
```

}

Output

Element at index 0: 10

Element at index 1: 20

Element at index 2: 30

Element at index 3: 40

Element at index 4: 50

Advantages of Arrays

- **Fast access** to elements using an index.
- **Efficient for storing** large data sets.
- **Easy implementation** of linear structures like stacks and queues.

Disadvantages of Arrays

- **Fixed size**, making dynamic storage inefficient.
- **Insertion and deletion** operations are costly.
- **Wastes memory** if the array is underutilized.

CHAPTER 4: LINKED LISTS – A DYNAMIC ALTERNATIVE

What is a Linked List?

A **linked list** is a collection of **nodes**, where each node contains:

1. **Data** – The actual value stored.

2. **Pointer** – A reference to the next node in the list.

Unlike arrays, **linked lists** are **dynamic in size** and allow **efficient insertion and deletion**.

Example: Implementing a Simple Linked List

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

   int data;

   struct Node* next;

};


void printList(struct Node* head) {

   struct Node* temp = head;

   while (temp != NULL) {

      printf("%d -> ", temp->data);

      temp = temp->next;

   }

   printf("NULL\n");

}
```

```
int main() {
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 10;
    head->next = second;

    second->data = 20;
    second->next = third;

    third->data = 30;
    third->next = NULL;

    printList(head);

    return 0;
}
```

Output

10 -> 20 -> 30 -> NULL

Advantages of Linked Lists

- **Dynamic memory allocation** – No fixed size.
- **Fast insertions/deletions** compared to arrays.
- **Efficient memory usage** – No unnecessary memory allocation.

Disadvantages of Linked Lists

- **Extra memory** for storing pointers.
- **Slower access** since elements are not contiguous in memory.

Chapter 5: Stacks and Queues – Special Linear Structures

Stack: Last In, First Out (LIFO)

A **stack** is a data structure that follows the **LIFO (Last In, First Out) principle**. Operations include:

- **Push** – Adding an element to the stack.
- **Pop** – Removing an element from the stack.
- **Peek** – Viewing the top element without removing it.

Example: Implementing a Stack Using Arrays

```
#include <stdio.h>

#define SIZE 5


int stack[SIZE], top = -1;


void push(int value) {
```

```
    if (top == SIZE - 1) {
        printf("Stack Overflow!\n");
    } else {
        stack[++top] = value;
    }
}

void pop() {
    if (top == -1) {
        printf("Stack Underflow!\n");
    } else {
        top--;
    }
}

void display() {
    if (top == -1) {
        printf("Stack is empty!\n");
    } else {
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
```

```
        }
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

Output

```
30
20
10
Stack after pop:
20
10
```

Chapter 6: Case Study – Using Data Structures in a Library Management System

Problem Statement

A library wants to maintain **book records** and allow:

1. **Adding a new book.**
2. **Removing a book.**
3. **Viewing the list of available books.**

Solution Using Linked Lists

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


struct Book {

   char title[100];

   struct Book* next;

};


void addBook(struct Book** head, char title[]) {

   struct Book* newBook = (struct Book*)malloc(sizeof(struct Book));

   strcpy(newBook->title, title);

   newBook->next = *head;
```

```
    *head = newBook;
}

void displayBooks(struct Book* head) {
    while (head != NULL) {
        printf("Book: %s\n", head->title);
        head = head->next;
    }
}

int main() {
    struct Book* library = NULL;
    addBook(&library, "C Programming");
    addBook(&library, "Data Structures");
    displayBooks(library);
    return 0;
}
```

Output

Book: Data Structures

Book: C Programming

CHAPTER 7: EXERCISES

1. **Implement a Queue Using Arrays.**
2. **Write a Program to Reverse a Linked List.**
3. **Implement a Stack Using Linked Lists.**
4. **Create a Menu-Driven Program for a Student Database Using Structures.**

CONCLUSION

Understanding **data structures in C** is essential for efficient programming. By mastering **arrays, linked lists, stacks, and queues**, programmers can **optimize performance, manage memory effectively, and solve complex problems efficiently**.

Implementation of Stack, Queue, and Linked List in C

Chapter 1: Introduction to Stack, Queue, and Linked List

Understanding Stack, Queue, and Linked List

Data structures play a crucial role in organizing and managing data efficiently in **C programming**. Among the fundamental data structures, **Stacks, Queues, and Linked Lists** provide flexible ways to store, access, and manipulate data dynamically. These structures are widely used in areas such as **memory management, scheduling algorithms, and real-time applications**.

- **Stack** follows the **LIFO (Last In, First Out) principle**, making it useful in scenarios such as expression evaluation and backtracking algorithms.
- **Queue** follows the **FIFO (First In, First Out) principle**, making it ideal for task scheduling and buffer management.
- **Linked List** is a **dynamic data structure** that allows efficient memory allocation and flexible data storage without predefined sizes.

Understanding these data structures **enhances problem-solving skills** and **optimizes program performance** by utilizing the right data structure for a given problem.

Chapter 2: Implementation of Stack in C

What is a Stack?

A **stack** is a linear data structure that allows **insertion and deletion** of elements only from the **top**. It follows the **LIFO (Last In, First Out) principle**, meaning that the last element added to the stack is the first one to be removed.

Operations on a Stack

1. **Push** – Adds an element to the top of the stack.
2. **Pop** – Removes an element from the top of the stack.
3. **Peek (Top)** – Retrieves the top element without removing it.
4. **isEmpty** – Checks if the stack is empty.

Example: Implementing a Stack Using Arrays

```
#include <stdio.h>

#define SIZE 5


int stack[SIZE], top = -1;


void push(int value) {

  if (top == SIZE - 1) {

    printf("Stack Overflow!\n");

  } else {

    stack[++top] = value;

  }

}
```

```
void pop() {
    if (top == -1) {
        printf("Stack Underflow!\n");
    } else {
        printf("Popped element: %d\n", stack[top--]);
    }
}

void display() {
    if (top == -1) {
        printf("Stack is empty!\n");
    } else {
        printf("Stack elements:\n");
        for (int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}

int main() {
```

```
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

Expected Output

```
Stack elements:
30
20
10
Popped element: 30
Stack elements:
20
10
```

Advantages of Stack

- Simple implementation.
- Efficient for managing **function calls (recursion)** and **undo operations**.

Disadvantages of Stack

- Fixed size in the array implementation.
- Limited operations only from one end.

CHAPTER 3: IMPLEMENTATION OF QUEUE IN C

What is a Queue?

A **queue** is a linear data structure that follows the **FIFO (First In, First Out) principle**, meaning the **first element added is the first one removed**. It is widely used in **task scheduling, CPU scheduling, and data buffering**.

Operations on a Queue

1. **Enqueue** – Adds an element to the **rear**.
2. **Dequeue** – Removes an element from the **front**.
3. **Peek (Front)** – Retrieves the first element without removing it.
4. **isEmpty** – Checks if the queue is empty.

Example: Implementing a Queue Using Arrays

```
#include <stdio.h>

#define SIZE 5


int queue[SIZE], front = -1, rear = -1;


void enqueue(int value) {
```

```
    if (rear == SIZE - 1) {
        printf("Queue Overflow!\n");
    } else {
        if (front == -1) front = 0;
        queue[++rear] = value;
    }
}

void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow!\n");
    } else {
        printf("Dequeued element: %d\n", queue[front++]);
        if (front > rear) front = rear = -1; // Reset when empty
    }
}

void display() {
    if (front == -1 || front > rear) {
        printf("Queue is empty!\n");
    } else {
```

```
        printf("Queue elements:\n");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

Expected Output

Queue elements:

10 20 30

Dequeued element: 10

Queue elements:

20 30

Advantages of Queue

- Efficient for **task scheduling and buffer management**.
- Maintains **order of execution**.

Disadvantages of Queue

- In **static arrays**, memory wastage occurs after dequeuing elements.
- Insertion and deletion **can be slow** in large queues.

CHAPTER 4: IMPLEMENTATION OF LINKED LIST IN C

What is a Linked List?

A **linked list** is a dynamic data structure where **each node contains data and a pointer to the next node**. Unlike arrays, linked lists **do not require contiguous memory allocation**, making them efficient for dynamic applications.

Operations on a Linked List

1. **Insertion** – Adds a node at the beginning, middle, or end.
2. **Deletion** – Removes a node from the list.
3. **Traversal** – Moves through the list to display elements.

Example: Implementing a Singly Linked List

#include <stdio.h>

```
#include <stdlib.h>


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


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

   if (*head == NULL) {
      *head = newNode;
      return;
   }

   while (temp->next != NULL)
      temp = temp->next;
```

```
    temp->next = newNode;
}

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

int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
    displayList(head);
    return 0;
}
```

Expected Output

10 -> 20 -> 30 -> NULL

Advantages of Linked List

- **Dynamic memory allocation**, avoiding wastage.
- **Efficient insertions and deletions**.

Disadvantages of Linked List

- **Extra memory required** for pointers.
- **Slower access** compared to arrays.

CHAPTER 5: CASE STUDY – TASK SCHEDULING SYSTEM

Problem Statement

A **task scheduler** manages different tasks in **FIFO order**, ensuring **oldest tasks are executed first**.

Solution Using Queues

- **Enqueue new tasks** when added.
- **Dequeue tasks** when completed.

This model is used in **CPU scheduling, print job scheduling, and event handling systems**.

CHAPTER 6: EXERCISES

1. **Implement a stack using linked lists.**
2. **Create a circular queue using arrays.**
3. **Write a program to merge two linked lists.**
4. **Implement priority queue using linked list.**

CONCLUSION

The implementation of **Stacks, Queues, and Linked Lists** in C **enhances efficiency, flexibility, and memory management** in programming. Mastering these data structures provides a **solid foundation for advanced computing** and **real-world applications**.

Searching and Sorting Algorithms in C (Bubble Sort, Merge Sort, Quick Sort)

Chapter 1: Introduction to Searching and Sorting Algorithms

Understanding Searching and Sorting

Searching and sorting are fundamental operations in computer science. **Searching** allows locating an element in a dataset, while **sorting** organizes data in a particular order to enhance searching efficiency and data processing.

- **Searching Algorithms** find elements efficiently, reducing execution time.
- **Sorting Algorithms** arrange data in ascending or descending order for easy retrieval.

Why Are Searching and Sorting Important?

1. **Optimized Data Processing** – Organizing data speeds up searching and retrieval.
2. **Enhances Algorithm Efficiency** – Faster algorithms improve performance in large datasets.
3. **Real-World Applications** – Used in **databases, search engines, and data analytics**.

Sorting techniques differ based on time complexity and efficiency. In this chapter, we explore three widely used sorting algorithms: **Bubble Sort, Merge Sort, and Quick Sort**.

Chapter 2: Bubble Sort – A Simple Sorting Algorithm

What is Bubble Sort?

Bubble Sort is a simple comparison-based algorithm that **repeatedly swaps adjacent elements** if they are in the wrong order. The process continues until the array is fully sorted.

How Does Bubble Sort Work?

1. Compare **adjacent elements**.
2. Swap them if they are in the wrong order.
3. Repeat the process until the entire array is sorted.

Example: Implementing Bubble Sort in C

```
#include <stdio.h>


// Bubble Sort Function
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 if out of order
        int temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
      }
    }
  }
}
```

```
// Function to Print an Array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}


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

    bubbleSort(arr, n);
    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output

Sorted array: 11 12 22 25 34 64 90

Advantages of Bubble Sort

- Simple to implement.
- Works well for **small datasets**.

Disadvantages of Bubble Sort

- **Time Complexity: O(n^2)**, making it inefficient for large datasets.
- Requires **multiple passes**, leading to slow execution.

Chapter 3: Merge Sort – A Divide and Conquer Algorithm

What is Merge Sort?

Merge Sort is a **divide and conquer algorithm** that:

1. **Divides** the array into two halves.
2. **Recursively sorts** both halves.
3. **Merges** the sorted halves to produce a sorted array.

Example: Implementing Merge Sort in C

```
#include <stdio.h>


// Merge Function

void merge(int arr[], int left, int mid, int right) {

   int n1 = mid - left + 1;

   int n2 = right - mid;
```

```
    int L[n1], R[n2]; // Temporary arrays

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

// Merge Sort Function
```

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Function to Print an Array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Main Function
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int size = sizeof(arr) / sizeof(arr[0]);
```

```
    mergeSort(arr, 0, size - 1);
    printf("Sorted array: ");
    printArray(arr, size);

    return 0;
}
```

Output

Sorted array: 3 9 10 27 38 43 82

Advantages of Merge Sort

- **Time Complexity: O(n log n)** – Efficient for large datasets.
- **Stable Sort** – Maintains order of duplicate elements.

Disadvantages of Merge Sort

- **Uses extra memory** for temporary arrays.
- **Slower for small arrays** compared to simpler sorting algorithms.

Chapter 4: Quick Sort – The Fastest Sorting Algorithm

What is Quick Sort?

Quick Sort is a **divide and conquer algorithm** that:

1. **Selects a pivot element**.

2. **Partitions** the array into two parts:
    - Elements **less than** the pivot.
    - Elements **greater than** the pivot.
3. **Recursively sorts** both partitions.

Example: Implementing Quick Sort in C

```
#include <stdio.h>


// Partition Function

int partition(int arr[], int low, int high) {

  int pivot = arr[high];

  int i = (low - 1);


  for (int j = low; j < high; j++) {

    if (arr[j] < pivot) {

      i++;

      int temp = arr[i];

      arr[i] = arr[j];

      arr[j] = temp;

    }

  }
```

```
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return (i + 1);
}

// Quick Sort Function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

// Function to Print an Array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
```

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

// Main Function
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, size - 1);
    printf("Sorted array: ");
    printArray(arr, size);

    return 0;
}
```

Output

Sorted array: 1 5 7 8 9 10

Advantages of Quick Sort

- **Time Complexity: O(n log n)** in the average case.
- **In-place sorting** – Requires little extra memory.

Disadvantages of Quick Sort

- **Worst case O(n²)** if the pivot is chosen poorly.

- **Unstable sort**, meaning duplicate elements may lose their relative order.

Chapter 5: Exercises

1. **Implement Selection Sort and compare its efficiency with Bubble Sort.**
2. **Modify Merge Sort to sort in descending order.**
3. **Use Quick Sort to sort an array of strings.**
4. **Implement a binary search algorithm to find an element in a sorted array.**

Conclusion

Sorting algorithms enhance data organization and retrieval. **Bubble Sort** is easy to implement but inefficient for large data. **Merge Sort** provides **stability and efficiency**, while **Quick Sort** offers **fast in-place sorting**. Understanding and implementing these algorithms helps programmers **build optimized, real-world applications**.

INTRODUCTION TO SYSTEM CALLS IN C

CHAPTER 1: UNDERSTANDING SYSTEM CALLS

What are System Calls?

System calls are the **interface between user programs and the operating system (OS)**. In **C programming**, system calls allow a program to **request services** from the OS, such as:

- **File handling**
- **Process creation**
- **Memory management**
- **Device control**
- **Inter-process communication**

Every modern operating system, such as **Linux, macOS, and Windows**, provides system calls to interact with **hardware and kernel-level resources**.

Why Are System Calls Important?

1. **Direct Communication with the OS** – Programs use system calls to access hardware features and OS services.
2. **Resource Management** – System calls handle files, processes, and memory efficiently.
3. **Security and Protection** – OS ensures that system calls follow strict permission policies.
4. **Cross-Platform Functionality** – Standard system calls allow portability between Unix-based systems.

In C, system calls are invoked using **library functions from unistd.h, fcntl.h, and sys/types.h**, which provide access to the OS services.

Chapter 2: Classification of System Calls

System calls are broadly classified into five categories:

1. Process Control

- fork() – Creates a new process.
- exec() – Replaces the current process with a new one.
- exit() – Terminates a process.

2. File Management

- open() – Opens a file.
- read() – Reads data from a file.
- write() – Writes data to a file.
- close() – Closes a file.

3. Device Management

- ioctl() – Controls device parameters.
- read() / write() – Used for I/O operations.

4. Information Maintenance

- getpid() – Gets process ID.
- getcwd() – Gets the current working directory.

5. Communication

- pipe() – Creates a communication channel.
- send() / recv() – Used in networking.

Selecting the appropriate system call depends on the **type of task and the operating system requirements**.

CHAPTER 3: FILE MANAGEMENT SYSTEM CALLS IN C

What Are File System Calls?

File system calls provide an interface for **reading, writing, opening, and closing files** in the OS. They enable programs to handle files **without using high-level C functions like fopen() and fprintf()**.

Example: Using open(), read(), and write() System Calls

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>


int main() {

  int file = open("example.txt", O_WRONLY | O_CREAT, 0644);


  if (file < 0) {

    perror("File opening failed");

    return 1;

  }
```

```
    char data[] = "System Call Example in C\n";
    write(file, data, sizeof(data));

    close(file);
    printf("File written successfully using system calls!\n");

    return 0;
}
```

Output

File written successfully using system calls!

Explanation

- open() creates or opens a file.
- write() writes data to the file.
- close() closes the file after writing.

Advantages of File System Calls

- **Lower-level access** to files compared to C standard I/O functions.
- **Better control over file operations** using file descriptors.

Disadvantages

- **More complex** than standard library functions.

- Requires **manual error handling**.

Chapter 4: Process Management System Calls

What is Process Control?

Process control system calls manage processes in an operating system. They allow:

- **Creating a new process**
- **Replacing an existing process**
- **Terminating a process**

Example: Using fork() System Call

```
#include <stdio.h>

#include <unistd.h>


int main() {

   int pid = fork();


   if (pid < 0) {

      printf("Fork failed!\n");

      return 1;

   } else if (pid == 0) {

      printf("Child process: PID = %d\n", getpid());
```

```
    } else {
        printf("Parent process: PID = %d, Child PID = %d\n", getpid(), pid);
    }

    return 0;
}
```

Output

Parent process: PID = 1234, Child PID = 1235

Child process: PID = 1235

Explanation

- fork() creates a new child process.
- getpid() retrieves the process ID.
- **Parent and child processes execute independently.**

Advantages of Process System Calls

- **Efficient multitasking** by creating multiple processes.
- **Resource sharing** between processes.

Disadvantages

- Requires **proper synchronization**.
- **Debugging is complex** in concurrent execution.

CHAPTER 5: INTER-PROCESS COMMUNICATION (IPC) USING PIPES

What is IPC?

Inter-process communication (IPC) allows processes to **exchange data**. Pipes (pipe()) enable unidirectional communication between a **parent and child process**.

Example: Using pipe() for Parent-Child Communication

```
#include <stdio.h>

#include <unistd.h>


int main() {

  int fd[2];

  pipe(fd);


  int pid = fork();


  if (pid == 0) {  // Child Process

    close(fd[0]);  // Close read end

    char message[] = "Hello from Child!";

    write(fd[1], message, sizeof(message));

    close(fd[1]);

  } else {  // Parent Process

    close(fd[1]);  // Close write end
```

```
        char buffer[50];
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
        close(fd[0]);
    }

    return 0;
}
```

Output

Parent received: Hello from Child!

Explanation

- pipe(fd) creates a communication channel.
- **Child writes** to the pipe.
- **Parent reads** the message from the pipe.

Advantages of IPC

- **Processes can share information efficiently**.
- **Used in client-server models** for inter-process data exchange.

Disadvantages

- **Complexity increases** when multiple processes communicate.
- Requires **proper synchronization**.

CHAPTER 6: CASE STUDY – PROCESS MANAGEMENT IN AN OPERATING SYSTEM

Problem Statement

An **operating system** manages multiple running processes. It needs to:

1. **Create new processes** for running programs.
2. **Assign process IDs** for tracking.
3. **Terminate completed processes**.

Solution Using fork() and exec()

```
#include <stdio.h>

#include <unistd.h>


int main() {

  int pid = fork();


  if (pid == 0) {

    printf("Child Process Running...\n");

    execlp("/bin/ls", "ls", NULL);

  } else {

    printf("Parent Process Waiting...\n");

    wait(NULL);

    printf("Child Process Completed!\n");
```

```
    }

    return 0;
}
```

Expected Output

Parent Process Waiting...

Child Process Running...

(list of files)

Child Process Completed!

Key Learnings

- **execlp() replaces the child process** with another program (ls command).
- **wait() ensures the parent waits** for the child to complete.

CHAPTER 7: EXERCISES

1. **Modify the fork() example** to create **two child processes**.
2. **Implement a logging system** where a **parent process writes logs to a file**, and the **child process reads them**.
3. **Use pipe() to create two-way communication** between processes.
4. **Write a program that uses exec()** to execute the date command.

CONCLUSION

System calls in C provide **direct interaction with the operating system** for managing files, processes, and inter-process communication. Mastering system calls is crucial for:

- **Building system-level applications**
- **Developing operating systems**
- **Creating efficient multitasking programs**

MULTI-THREADING IN C (PTHREAD LIBRARY)

CHAPTER 1: INTRODUCTION TO MULTI-THREADING

What is Multi-Threading?

Multi-threading is a technique that allows a program to execute **multiple tasks concurrently**. Instead of running **sequentially**, a program can create **multiple threads**, each performing a different task simultaneously. This leads to:

- **Faster execution**
- **Efficient CPU utilization**
- **Better responsiveness in applications**

How Does Multi-Threading Work?

A thread is a **lightweight process** that shares the **same memory space** as other threads within the same program. In C, multi-threading is implemented using the **pthread (POSIX thread) library**, which provides functions to **create, manage, and synchronize threads**.

Why Use Multi-Threading?

1. **Improves Performance** – Enables parallel execution of tasks.
2. **Efficient Resource Utilization** – Multiple threads share the same memory.
3. **Better Responsiveness** – Ideal for **GUI applications and servers**.
4. **Concurrency** – Enables running multiple independent operations simultaneously.

Multi-threading is widely used in **operating systems, web servers, gaming, and real-time applications**.

Chapter 2: Basics of the pthread Library

What is the pthread Library?

The **pthread** library is a POSIX standard for multi-threading in C. It provides functions for:

- **Creating Threads (pthread_create())**
- **Waiting for Threads (pthread_join())**
- **Terminating Threads (pthread_exit())**
- **Synchronizing Threads (pthread_mutex_t)**

To use pthreads, **include the pthread.h library** and compile programs with the -pthread flag:

```
gcc -pthread program.c -o program
```

Basic Multi-Threading Example

```
#include <stdio.h>

#include <pthread.h>


// Thread function

void *printMessage(void *arg) {

  printf("Hello from Thread!\n");

  return NULL;
```

```
}

int main() {
    pthread_t thread;

    // Creating a thread
    pthread_create(&thread, NULL, printMessage, NULL);

    // Waiting for the thread to finish
    pthread_join(thread, NULL);

    printf("Main Thread Finished.\n");
    return 0;
}
```

Output

Hello from Thread!

Main Thread Finished.

Explanation

1. **pthread_create()** starts a new thread.
2. **pthread_join()** waits for the thread to complete execution.

CHAPTER 3: CREATING AND MANAGING THREADS

How to Create Threads in C?

Threads are created using the pthread_create() function.

Syntax

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);

- **thread** – Thread ID.
- **attr** – Attributes (NULL for default).
- **start_routine** – Function executed by the thread.
- **arg** – Arguments passed to the function.

Example: Creating Multiple Threads

```
#include <stdio.h>

#include <pthread.h>


void *printThread(void *arg) {

   int id = *(int *)arg;

   printf("Thread %d is running\n", id);

   return NULL;

}


int main() {
```

```
    pthread_t threads[3];

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, printThread, &i);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All Threads Finished\n");
    return 0;
}
```

Expected Output

Thread 2 is running

Thread 2 is running

Thread 2 is running

All Threads Finished

(The output may vary due to race conditions on variable i)

Key Learnings

- **Race conditions occur** when multiple threads access the same variable (i).
- **Use proper synchronization techniques** to avoid unexpected behavior.

Chapter 4: Synchronizing Threads Using Mutex

What is a Mutex?

A **mutex (mutual exclusion lock)** prevents multiple threads from **accessing shared resources simultaneously**, avoiding **data corruption and race conditions**.

Example: Using Mutex to Prevent Race Conditions

```
#include <stdio.h>

#include <pthread.h>


pthread_mutex_t lock;

int counter = 0;


void *incrementCounter(void *arg) {

   pthread_mutex_lock(&lock);  // Lock the mutex

   counter++;

   printf("Counter: %d\n", counter);

   pthread_mutex_unlock(&lock);  // Unlock the mutex
```

```
    return NULL;
}


int main() {
    pthread_t threads[5];

    pthread_mutex_init(&lock, NULL);

    for (int i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, incrementCounter, NULL);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    return 0;
}
```

Expected Output

Counter: 1

```
Counter: 2

Counter: 3

Counter: 4

Counter: 5
```

Key Takeaways

- **pthread_mutex_lock() ensures only one thread updates counter at a time.**
- **pthread_mutex_unlock() releases the lock, allowing the next thread to proceed.**

Chapter 5: Thread Synchronization Using Condition Variables

What is a Condition Variable?

Condition variables allow threads to **wait for a specific condition to be met** before proceeding.

Example: Using Condition Variables

```c
#include <stdio.h>

#include <pthread.h>


pthread_mutex_t lock;

pthread_cond_t cond;

int dataReady = 0;
```

```
void *producer(void *arg) {
   pthread_mutex_lock(&lock);
   dataReady = 1;
   printf("Producer: Data ready\n");
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&lock);
   return NULL;
}

void *consumer(void *arg) {
   pthread_mutex_lock(&lock);
   while (dataReady == 0) {
      pthread_cond_wait(&cond, &lock);
   }
   printf("Consumer: Data processed\n");
   pthread_mutex_unlock(&lock);
   return NULL;
}

int main() {
```

```
    pthread_t prod, cons;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);

    return 0;
}
```

Expected Output

Producer: Data ready

Consumer: Data processed

Explanation

- **pthread_cond_wait() makes the consumer wait until data is ready.**
- **pthread_cond_signal() notifies the consumer that data is available.**

CHAPTER 6: CASE STUDY – MULTI-THREADED WEB SERVER

Problem Statement

A **web server** must handle multiple **client requests simultaneously**. A single-threaded server will process one request at a time, leading to **delays**.

Solution Using Multi-Threading

1. **Each request is handled by a separate thread**.
2. **Mutex locks** ensure thread-safe access to shared resources.
3. **Condition variables synchronize request handling**.

Implementation

```
#include <stdio.h>

#include <pthread.h>

#include <unistd.h>


void *handleRequest(void *arg) {

   int clientID = *(int *)arg;

   printf("Handling request from client %d\n", clientID);
```

```
    sleep(2);
    printf("Request from client %d completed\n", clientID);
    return NULL;
}

int main() {
    pthread_t threads[3];

    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, handleRequest, &i);
        sleep(1);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All requests handled\n");
    return 0;
}
```

Expected Output

Handling request from client 2

Request from client 2 completed

All requests handled

Chapter 7: Exercises

1. **Modify the producer-consumer problem to handle multiple producers and consumers.**
2. **Implement a multi-threaded sorting program using Quick Sort.**
3. **Create a multi-threaded logging system using mutex for synchronized file access.**

Conclusion

Multi-threading in C using the **pthread library** improves **program efficiency and responsiveness**. By mastering:

- **Thread creation and management**
- **Synchronization using mutex and condition variables**
- **Concurrency handling techniques**

INTER-PROCESS COMMUNICATION (IPC) IN C

CHAPTER 1: INTRODUCTION TO INTER-PROCESS COMMUNICATION (IPC)

What is Inter-Process Communication (IPC)?

Inter-Process Communication (IPC) is a mechanism that allows **multiple processes to communicate and share data**. Since processes in an operating system **run independently**, IPC provides a way for them to **exchange information** efficiently.

IPC is essential for **multi-processing applications, operating systems, and distributed systems**, enabling processes to:

- **Exchange messages**
- **Share memory**
- **Synchronize execution**

Why is IPC Important?

1. **Facilitates Data Sharing** – Allows processes to share information dynamically.
2. **Enables Parallel Processing** – Enhances performance in multi-processing environments.
3. **Supports Distributed Computing** – Helps in networked applications and cloud computing.
4. **Efficient Synchronization** – Manages concurrent process execution.

IPC is widely used in **multi-core processors, databases, web servers, and real-time applications**.

CHAPTER 2: TYPES OF INTER-PROCESS COMMUNICATION

IPC can be categorized into **two main types**:

1. **Message Passing** – Processes communicate using messages.
   - **Pipes**
   - **Message Queues**
   - **Sockets**
2. **Shared Memory** – Processes share a common memory segment.
   - **Shared Memory**
   - **Semaphores**

Comparison of IPC Methods

IPC Method	Data Transfer	Speed	Complexity
Pipes	Unidirectional	Medium	Easy
Message Queues	Message-based	Medium	Moderate
Shared Memory	Memory-based	Fast	Complex
Sockets	Network-based	Slow	High

Choosing the right IPC mechanism depends on the **type of application and efficiency requirements**.

CHAPTER 3: PIPES – BASIC IPC MECHANISM

What is a Pipe?

A **pipe** is a unidirectional communication channel used to **transfer data between parent and child processes**.

Example: Using pipe() for Parent-Child Communication

```
#include <stdio.h>

#include <unistd.h>


int main() {

   int fd[2]; // File descriptors for pipe

   pipe(fd);


   int pid = fork();


   if (pid == 0) {  // Child Process

      close(fd[0]);  // Close read end

      char message[] = "Hello from Child!";

      write(fd[1], message, sizeof(message));

      close(fd[1]);

   } else {  // Parent Process

      close(fd[1]);  // Close write end
```

```
        char buffer[50];
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);
        close(fd[0]);
    }

    return 0;
}
```

Expected Output

Parent received: Hello from Child!

Explanation

- **pipe(fd)** creates a communication channel.
- **Child writes** to the pipe.
- **Parent reads** the message from the pipe.

Advantages of Pipes

- **Simple to implement**.
- **Efficient for related processes**.

Disadvantages

- **Unidirectional communication**.
- **Limited to parent-child process communication**.

CHAPTER 4: MESSAGE QUEUES – QUEUE-BASED COMMUNICATION

What is a Message Queue?

A **message queue** allows processes to **exchange messages asynchronously**. Unlike pipes, message queues **persist even after process termination**.

Example: Using msgget(), msgsnd(), and msgrcv()

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>


struct message {

   long msg_type;

   char msg_text[100];

};


int main() {

   key_t key = ftok("progfile", 65);

   int msgid = msgget(key, 0666 | IPC_CREAT);


   struct message msg;

   msg.msg_type = 1;
```

```
    strcpy(msg.msg_text, "Hello from Message Queue!");

    msgsnd(msgid, &msg, sizeof(msg), 0);
    printf("Message sent: %s\n", msg.msg_text);

    return 0;
}
```

Expected Output

Message sent: Hello from Message Queue!

Advantages of Message Queues

- **Supports asynchronous communication**.
- **Persists beyond process termination**.

Disadvantages

- **Slower compared to shared memory**.
- **Requires explicit message formatting**.

Chapter 5: Shared Memory – Fastest IPC Mechanism

What is Shared Memory?

Shared memory is an IPC mechanism where multiple processes **access the same memory segment**, enabling the **fastest data exchange**.

Example: Using shmget(), shmat(), and shmdt()

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main() {
   key_t key = ftok("shmfile", 65);
   int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

   char *str = (char*) shmat(shmid, (void*)0, 0);
   strcpy(str, "Hello from Shared Memory!");

   printf("Data written in memory: %s\n", str);
   shmdt(str);

   return 0;
}
```

Expected Output

Data written in memory: Hello from Shared Memory!

Advantages of Shared Memory

- **Fastest IPC mechanism**.

- **Efficient for large data transfers**.

Disadvantages

- **Requires synchronization mechanisms** (like semaphores).
- **More complex to implement** than pipes and message queues.

CHAPTER 6: SYNCHRONIZATION USING SEMAPHORES

What is a Semaphore?

A **semaphore** is used to control access to shared resources **by multiple processes**.

Example: Using sem_init(), sem_wait(), and sem_post()

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>


sem_t sem;


void *process(void *arg) {

   sem_wait(&sem);  // Wait (lock)

   printf("Process %d is executing\n", *(int *)arg);

   sem_post(&sem);  // Signal (unlock)

   return NULL;
```

```
}

int main() {
   pthread_t threads[3];
   sem_init(&sem, 0, 1);

   int ids[] = {1, 2, 3};
   for (int i = 0; i < 3; i++) {
      pthread_create(&threads[i], NULL, process, &ids[i]);
   }

   for (int i = 0; i < 3; i++) {
      pthread_join(threads[i], NULL);
   }

   sem_destroy(&sem);
   return 0;
}
```

Expected Output

Process 1 is executing

Process 2 is executing

Process 3 is executing

Key Takeaways

- **Semaphores prevent race conditions**.
- **Ensure that only one process accesses shared resources at a time**.

Chapter 7: Case Study – Multi-Process Chat Application

Problem Statement

A **multi-process chat application** where:

1. **One process sends messages**.
2. **Another process receives messages**.

Solution Using Message Queues

- **msgsnd() for sending messages**.
- **msgrcv() for receiving messages**.
- **Processes run independently**.

Implementation includes:

1. **Sender Process** – Reads user input and sends it to the queue.
2. **Receiver Process** – Continuously reads from the queue and displays messages.

Expected Features

- **Real-time communication between processes**.
- **Efficient messaging using IPC mechanisms**.

CHAPTER 8: EXERCISES

1. **Modify the pipe example to allow bidirectional communication.**
2. **Implement a multi-client chat system using sockets.**
3. **Use shared memory to transfer a large array between two processes.**
4. **Implement process synchronization using semaphores in a bank transaction system.**

CONCLUSION

Inter-Process Communication (IPC) enables **processes to share data efficiently**. By mastering:

- **Pipes for simple communication**
- **Message Queues for structured messaging**
- **Shared Memory for high-speed data transfer**
- **Semaphores for synchronization**

INTRODUCTION TO EMBEDDED C PROGRAMMING

CHAPTER 1: UNDERSTANDING EMBEDDED C PROGRAMMING

What is Embedded C?

Embedded C is an extension of the **C programming language** that is specifically designed for **microcontrollers and embedded systems**. Unlike standard C, which runs on general-purpose computers, **Embedded C is optimized for direct hardware interaction**.

Why is Embedded C Important?

1. **Low-Level Hardware Access** – Directly interacts with registers, memory, and peripherals.
2. **Efficiency and Speed** – Optimized for performance with minimal overhead.
3. **Real-Time Processing** – Supports real-time applications in robotics, automotive, and IoT.
4. **Portability** – Can run on multiple embedded platforms with minimal changes.

Embedded C is widely used in **automotive control systems, medical devices, industrial automation, consumer electronics, and IoT applications**.

CHAPTER 2: DIFFERENCES BETWEEN STANDARD C AND EMBEDDED C

Feature	Standard C	Embedded C

Execution Platform	General-purpose computers	Microcontrollers and embedded systems
Hardware Control	Limited access	Direct hardware manipulation
Memory Constraints	Large memory available	Limited memory resources
Performance	Optimized for software	Optimized for speed and efficiency
Standard Libraries	Uses standard C libraries	Uses hardware-specific libraries

Unlike traditional C, **Embedded C often lacks standard libraries like <stdio.h>** because most microcontrollers do not have displays or file systems.

CHAPTER 3: BASIC STRUCTURE OF AN EMBEDDED C PROGRAM

Essential Components of Embedded C

An Embedded C program typically consists of:

1. **Preprocessor Directives** – Includes headers for hardware-specific functions.
2. **Global Variables and Constants** – Defines memory-mapped register addresses.
3. **Main Function (main())** – The entry point of execution.
4. **Peripheral Access** – Directly interacts with registers and hardware.

5. **Interrupt Service Routines (ISR)** – Handles real-time events.

Example: Simple LED Blinking Program (Using GPIO)

```
#include <avr/io.h> // AVR Microcontroller Register Definitions

#include <util/delay.h> // Delay Function


int main() {

  DDRB |= (1 << PB0); // Set PB0 as output


  while (1) {

    PORTB |= (1 << PB0);  // Turn LED ON

    _delay_ms(1000);      // 1-second delay

    PORTB &= ~(1 << PB0); // Turn LED OFF

    _delay_ms(1000);      // 1-second delay

  }


  return 0;

}
```

Explanation

- DDRB |= (1 << PB0); – Configures **Pin PB0 as an output**.
- PORTB |= (1 << PB0); – Sets PB0 **HIGH** to turn ON the LED.
- PORTB &= ~(1 << PB0); – Sets PB0 **LOW** to turn OFF the LED.

- _delay_ms(1000); – Adds a **1-second delay** between LED toggles.

Key Takeaways

- **Direct hardware control** using **registers**.
- **Uses memory-mapped I/O** instead of printf().

CHAPTER 4: WORKING WITH MICROCONTROLLERS IN EMBEDDED C

What is a Microcontroller?

A **microcontroller** is a compact computer-on-a-chip that includes:

1. **CPU (Processor)**
2. **Memory (RAM, ROM)**
3. **I/O Ports (GPIO, UART, SPI, I2C)**
4. **Timers, ADC, PWM**

Popular microcontrollers used in Embedded C:

- **AVR (ATmega328, ATmega2560)**
- **PIC (PIC16F877A, PIC18F4550)**
- **ARM Cortex (STM32, LPC2148)**

Example: Reading a Button Press

```
#include <avr/io.h>


int main() {
```

```
  DDRD &= ~(1 << PD0); // Set PD0 as input (Button)
  DDRB |= (1 << PB0);  // Set PB0 as output (LED)

  while (1) {
    if (PIND & (1 << PD0)) // Check if button is pressed
      PORTB |= (1 << PB0); // Turn ON LED
    else
      PORTB &= ~(1 << PB0); // Turn OFF LED
  }

  return 0;
}
```

Key Concepts

- **Polling Mechanism** – The program continuously checks for button presses.
- **Bitwise Operations** – Used to set and clear bits efficiently.

CHAPTER 5: INTERRUPTS IN EMBEDDED C

What are Interrupts?

Interrupts are **signals that pause the normal execution flow** of a program to handle high-priority events.

Example: Handling a Button Press with Interrupts

```
#include <avr/io.h>

#include <avr/interrupt.h>


ISR(INT0_vect) {
   PORTB ^= (1 << PB0); // Toggle LED on interrupt
}


int main() {
   DDRB |= (1 << PB0);  // Set PB0 as output
   EIMSK |= (1 << INT0); // Enable external interrupt INT0
   EICRA |= (1 << ISC01); // Falling edge trigger
   sei(); // Enable global interrupts


   while (1);
}
```

Explanation

- **ISR(INT0_vect)** – Interrupt Service Routine executes when **INT0 is triggered**.
- **sei();** – Enables global interrupts.

Key Advantages

- **No need for polling** – Reduces CPU usage.

- **Handles real-time events efficiently**.

CHAPTER 6: SERIAL COMMUNICATION (UART) IN EMBEDDED C

What is UART?

UART (**Universal Asynchronous Receiver-Transmitter**) allows communication between a microcontroller and a **computer or another device**.

Example: Sending Data Over UART

```
#include <avr/io.h>


void UART_init(unsigned int baud) {

   UBRR0L = baud;

   UCSR0B = (1 << TXEN0); // Enable transmitter

}


void UART_transmit(char data) {

   while (!(UCSR0A & (1 << UDRE0))); // Wait until buffer is empty

   UDR0 = data; // Send data

}


int main() {

   UART_init(9600);
```

```
  while (1) {
    UART_transmit('A'); // Send 'A' character
  }
}
```

Expected Output

- The letter **'A'** is continuously sent over **UART**.

Key Takeaways

- **UART is essential for debugging**.
- **Used in Bluetooth, Wi-Fi, and GPS communication**.

CHAPTER 7: CASE STUDY – SMART HOME AUTOMATION SYSTEM

Problem Statement

A **smart home system** requires:

1. **Controlling an LED (light) via a button press**.
2. **Sending system status to a serial monitor**.

Solution Using Embedded C

- **LED controlled by a button**
- **Status messages sent over UART**
- **Interrupts handle real-time responses**

Features Implemented:

- **Polling-free button detection**
- **Live system feedback through serial communication**

Code Implementation

```
#include <avr/io.h>

#include <avr/interrupt.h>


void UART_init(unsigned int baud) {

   UBRR0L = baud;

   UCSR0B = (1 << TXEN0);

}


void UART_transmit(char data) {

   while (!(UCSR0A & (1 << UDRE0)));

   UDR0 = data;

}


ISR(INT0_vect) {

   PORTB ^= (1 << PB0);

   UART_transmit('T'); // Send status update

}
```

```
int main() {
  DDRB |= (1 << PB0);
  EIMSK |= (1 << INT0);
  EICRA |= (1 << ISC01);
  sei();
  UART_init(9600);

  while (1);
}
```

Expected Behavior

- **LED toggles when button is pressed**.
- **System sends a status update ('T') via UART**.

CHAPTER 8: EXERCISES

1. **Modify the LED blinking code to use timers instead of delays.**
2. **Implement an ultrasonic sensor using Embedded C.**
3. **Create a temperature monitoring system using ADC (Analog-to-Digital Converter).**
4. **Develop an LCD-based display system using Embedded C.**

CONCLUSION

Embedded C **enables direct hardware interaction** and is crucial for **microcontroller-based systems**. Mastering:

- **GPIO operations**
- **Interrupt handling**
- **Serial communication**

REAL-WORLD APPLICATIONS OF C LANGUAGE

CHAPTER 1: INTRODUCTION TO C LANGUAGE IN THE REAL WORLD

Why is C Still Widely Used?

The **C programming language** has been around for decades and continues to be a foundational language in modern computing. It provides:

- **High performance** – Ideal for low-level system programming.
- **Portability** – Runs on different hardware with minimal modifications.
- **Efficient memory management** – Enables direct memory access for speed optimization.
- **Hardware control** – Used in firmware, microcontrollers, and embedded systems.

C is used across a **wide range of industries**, from **operating systems and game development to high-frequency trading and robotics**.

Key Features Making C Suitable for Real-World Applications

1. **Speed and Efficiency** – Runs **close to hardware**, making it suitable for performance-critical applications.
2. **Portability** – Runs on **Windows, Linux, macOS, embedded systems, and supercomputers**.
3. **Structured and Modular** – Enables large-scale software development.

Due to its balance of **control, efficiency, and flexibility**, C remains a **dominant force** in software development.

Chapter 2: C in Operating System Development

How is C Used in Operating Systems?

Operating systems rely heavily on **C language** because of:

- **Low-level hardware access** – Direct memory manipulation.
- **Efficient resource management** – Handles CPU scheduling, memory allocation, and multitasking.
- **Portability** – Runs on multiple architectures.

Popular Operating Systems Written in C

Operating System	Description
Windows	The kernel of Microsoft Windows is written in C.
Linux	The entire Linux kernel is written in C.
MacOS	The macOS and iOS kernels are based on C.
UNIX	The original UNIX OS was developed in C.

Example: Kernel-Level Programming in C

```
#include <stdio.h>

#include <sys/sysinfo.h>
```

```
int main() {
   struct sysinfo info;
   sysinfo(&info);


   printf("Uptime: %ld seconds\n", info.uptime);
   printf("Total RAM: %ld MB\n", info.totalram / (1024 * 1024));


   return 0;
}
```

Expected Output

Uptime: 15000 seconds

Total RAM: 4096 MB

Key Takeaways

- **C provides direct access to system resources.**
- **OS kernels use C for efficient memory and process management.**

CHAPTER 3: C IN EMBEDDED SYSTEMS AND IOT

How is C Used in Embedded Systems?

Embedded systems **control hardware devices** and require low-level programming for:

- **Microcontrollers**

- **Automobiles (ECUs)**
- **Smart home devices (IoT)**
- **Medical devices**

Example: LED Control Using Embedded C

```
#include <avr/io.h>

#include <util/delay.h>



int main() {

   DDRB |= (1 << PB0); // Set PB0 as output



   while (1) {

      PORTB ^= (1 << PB0);  // Toggle LED

      _delay_ms(1000);      // 1-second delay

   }



   return 0;

}
```

Real-World Applications

- **Automobile ECU (Electronic Control Units)**
- **Smart Home Systems (IoT)**
- **Medical Devices (Pacemakers, MRI Machines)**

Key Takeaways

- **C directly interacts with hardware** through memory-mapped registers.
- **Embedded C ensures optimized performance** with minimal power usage.

Chapter 4: C in Game Development

Why is C Used in Game Development?

1. **Fast Execution** – Provides high performance with **low latency**.
2. **Direct Hardware Access** – Optimized graphics and memory management.
3. **Cross-Platform Compatibility** – Games run on multiple OS with minimal changes.

Popular Game Engines Using C

Game Engine	Use Case
Unity	Uses C for performance-critical parts.
Unreal Engine	Core engine written in C.
id Tech Engine	Used in games like Doom and Quake.

Example: Simple Game Loop in C

```
#include <stdio.h>

#include <stdlib.h>
```

```
int main() {
   int score = 0;
   char input;

   while (1) {
      printf("Press 's' to score, 'q' to quit: ");
      scanf(" %c", &input);

      if (input == 's') {
         score++;
         printf("Score: %d\n", score);
      } else if (input == 'q') {
         printf("Final Score: %d\n", score);
         break;
      }
   }

   return 0;
}
```

Expected Output

Press 's' to score, 'q' to quit: s

Score: 1

Press 's' to score, 'q' to quit: s

Score: 2

Press 's' to score, 'q' to quit: q

Final Score: 2

Key Takeaways

- **C is used in game engines for its speed and efficiency.**
- **Game loops rely on C for handling real-time inputs and rendering.**

CHAPTER 5: C IN DATABASE MANAGEMENT SYSTEMS (DBMS)

Why is C Used in Databases?

1. **High Performance** – C allows efficient **query execution**.
2. **Memory Optimization** – Databases require fast memory access.
3. **Scalability** – Handles **large datasets efficiently**.

Popular Databases Written in C

Database	Description
MySQL	One of the most popular open-source databases.
PostgreSQL	Advanced database system with high scalability.
SQLite	Lightweight, embedded database used in mobile applications.

Example: SQLite Database Access in C

```
#include <stdio.h>

#include <sqlite3.h>


int main() {

   sqlite3 *db;

   int result = sqlite3_open("test.db", &db);


   if (result) {

      printf("Database opening failed!\n");

   } else {

      printf("Database opened successfully!\n");

   }


   sqlite3_close(db);

   return 0;

}
```

Expected Output

Database opened successfully!

Key Takeaways

- **C provides high-speed database operations.**

- **Many database engines rely on C for efficient query execution.**

Chapter 6: C in Cybersecurity and Cryptography

Why is C Used in Cybersecurity?

1. **Fast Execution** – Critical for encryption/decryption operations.
2. **Memory Management** – Enables secure buffer handling.
3. **Low-Level Access** – Required for network security applications.

Example: Simple Caesar Cipher Encryption in C

```
#include <stdio.h>


void caesarEncrypt(char *text, int shift) {
    for (int i = 0; text[i] != '\0'; i++) {
        text[i] = text[i] + shift;
    }
}


int main() {
    char text[] = "HELLO";
    caesarEncrypt(text, 3);
    printf("Encrypted Text: %s\n", text);
```

```
    return 0;
}
```

Expected Output

Encrypted Text: KHOOR

Real-World Applications

- **SSL/TLS Encryption**
- **Network Security Tools (Wireshark)**
- **Secure Authentication Systems**

CHAPTER 7: CASE STUDY – C IN HIGH-FREQUENCY TRADING SYSTEMS

Problem Statement

A **high-frequency trading (HFT) system** requires:

1. **Ultra-low latency execution**
2. **High-speed networking**
3. **Efficient memory management**

Solution Using C

- **Optimized algorithms for real-time processing**
- **Direct hardware interaction for speed**
- **Efficient data structures for market analysis**

HFT firms like Goldman Sachs, Morgan Stanley, and Citadel use C for financial trading systems.

Code Implementation

```
#include <stdio.h>


int main() {
    double stock_price = 150.75;
    double trade_price = stock_price * 1.02; // 2% markup
    printf("Trade Executed at: $%.2f\n", trade_price);
    return 0;
}
```

Expected Output

Trade Executed at: $153.77

CHAPTER 8: EXERCISES

1. **Modify the SQLite example to create a table and insert data.**
2. **Develop a simple encryption/decryption tool using C.**
3. **Create a multi-threaded web server using C.**
4. **Implement a real-time stock price simulator using C.**

CONCLUSION

C continues to be **one of the most powerful programming languages** for real-world applications. It is widely used in:

- **Operating Systems**
- **Embedded Systems & IoT**
- **Game Development**
- **Database Systems**
- **Cybersecurity**
- **Financial Trading**

Assignment Solution: Develop a Library Management System Using File Handling and Data Structures in C

Objective

The objective of this assignment is to develop a **Library Management System** using **file handling** and **data structures (linked lists and structures)** in **C programming**. This system will allow users to:

1. **Add new books**
2. **View available books**
3. **Search for books**
4. **Issue books**
5. **Return books**
6. **Save and retrieve book records using file handling**

Step-by-Step Guide

Step 1: Define the Problem

A **Library Management System** should allow:

- Storing book records efficiently.
- Searching for books by **title, author, or book ID**.
- Borrowing and returning books while updating the record.

- Saving and retrieving data using **file handling**.

Step 2: Plan the Data Structures

Structure for Book Details

A **structure (struct Book)** is used to store book information:

```
struct Book {
    int bookID;
    char title[100];
    char author[100];
    int available; // 1 = Available, 0 = Issued
    struct Book *next; // Pointer for Linked List
};
```

- **bookID** – Unique identifier for each book.
- **title** – Book title.
- **author** – Author name.
- **available** – Status of book (1 = Available, 0 = Issued).
- **next** – Pointer to the next book in the linked list.

Step 3: Implement File Handling

File handling will **store and retrieve** book records in a file (library.dat).

- **Writing to File (saveBooksToFile())** – Saves all books to a file.
- **Reading from File (loadBooksFromFile())** – Loads books into memory when the system starts.

STEP 4: IMPLEMENT THE C PROGRAM

Complete Code Implementation

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


struct Book {

   int bookID;

   char title[100];

   char author[100];

   int available; // 1 = Available, 0 = Issued

   struct Book *next;

};


struct Book *head = NULL;


// Function to Save Books to File
```

```
void saveBooksToFile() {
    FILE *file = fopen("library.dat", "wb");
    if (!file) {
        printf("Error saving file!\n");
        return;
    }

    struct Book *current = head;
    while (current) {
        fwrite(current, sizeof(struct Book), 1, file);
        current = current->next;
    }

    fclose(file);
}

// Function to Load Books from File
void loadBooksFromFile() {
    FILE *file = fopen("library.dat", "rb");
    if (!file) {
        return;
```

```
    }

    struct Book temp;
    while (fread(&temp, sizeof(struct Book), 1, file)) {
        struct Book *newBook = (struct Book *)malloc(sizeof(struct Book));
        *newBook = temp;
        newBook->next = head;
        head = newBook;
    }

    fclose(file);
}

// Function to Add a Book
void addBook() {
    struct Book *newBook = (struct Book *)malloc(sizeof(struct Book));

    printf("Enter Book ID: ");
    scanf("%d", &newBook->bookID);
    printf("Enter Title: ");
```

```
    getchar();
    fgets(newBook->title, 100, stdin);
    printf("Enter Author: ");
    fgets(newBook->author, 100, stdin);
    newBook->available = 1; // Book is available initially
    newBook->next = head;
    head = newBook;

    saveBooksToFile();
    printf("Book added successfully!\n");
}

// Function to Display All Books
void displayBooks() {
    struct Book *current = head;

    if (!current) {
        printf("No books available!\n");
        return;
    }
```

```
    printf("\nAvailable Books:\n");
    while (current) {
        printf("Book ID: %d\nTitle: %sAuthor: %sStatus: %s\n\n",
               current->bookID, current->title, current->author,
               current->available ? "Available" : "Issued");
        current = current->next;
    }
}

// Function to Search for a Book
void searchBook() {
    int id;
    printf("Enter Book ID to search: ");
    scanf("%d", &id);

    struct Book *current = head;
    while (current) {
        if (current->bookID == id) {
            printf("\nBook Found:\nBook ID: %d\nTitle: %sAuthor:
%sStatus: %s\n",
                   current->bookID, current->title, current->author,
```

```
            current->available ? "Available" : "Issued");
        return;
      }
      current = current->next;
    }
    printf("Book not found!\n");
}

// Function to Issue a Book
void issueBook() {
    int id;
    printf("Enter Book ID to issue: ");
    scanf("%d", &id);

    struct Book *current = head;
    while (current) {
      if (current->bookID == id) {
        if (current->available) {
          current->available = 0;
          saveBooksToFile();
          printf("Book issued successfully!\n");
```

```
        } else {
            printf("Book is already issued!\n");
        }
        return;
    }
    current = current->next;
  }
  printf("Book not found!\n");
}

// Function to Return a Book
void returnBook() {
  int id;
  printf("Enter Book ID to return: ");
  scanf("%d", &id);

  struct Book *current = head;
  while (current) {
    if (current->bookID == id) {
      if (!current->available) {
        current->available = 1;
```

```
          saveBooksToFile();
          printf("Book returned successfully!\n");
        } else {
          printf("This book was not issued!\n");
        }
        return;
      }
      current = current->next;
    }
    printf("Book not found!\n");
}

// Function to Display the Menu
void menu() {
    int choice;
    while (1) {
      printf("\nLibrary Management System\n");
      printf("1. Add Book\n");
      printf("2. Display Books\n");
      printf("3. Search Book\n");
      printf("4. Issue Book\n");
```

```
    printf("5. Return Book\n");
    printf("6. Exit\n");
    printf("Enter choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1: addBook(); break;
      case 2: displayBooks(); break;
      case 3: searchBook(); break;
      case 4: issueBook(); break;
      case 5: returnBook(); break;
      case 6: exit(0);
      default: printf("Invalid choice! Try again.\n");
    }
  }
}

// Main Function
int main() {
  loadBooksFromFile(); // Load books from file on startup
  menu();
```

```
    return 0;
}
```

Step 5: Explanation of the Code

1. **File Handling**
   - fopen(), fwrite(), fread(), and fclose() are used to store and retrieve book data persistently.
2. **Linked List**
   - Used to dynamically manage book records without size limitations.
3. **Functions Implemented**
   - addBook() – Adds new books.
   - displayBooks() – Shows available books.
   - searchBook() – Searches for a book by ID.
   - issueBook() – Issues a book to a user.
   - returnBook() – Marks a book as returned.
   - saveBooksToFile() – Saves books to a file.
   - loadBooksFromFile() – Loads books from a file.

Step 6: Example Runs

Adding a Book

Enter Book ID: 101

Enter Title: C Programming Basics

Enter Author: Dennis Ritchie

Book added successfully!

Displaying Books

Book ID: 101

Title: C Programming Basics

Author: Dennis Ritchie

Status: Available

Issuing a Book

Enter Book ID to issue: 101

Book issued successfully!

CONCLUSION

This **Library Management System** uses:

- **Linked Lists** for dynamic book storage.
- **File Handling** for persistent book records.
- **Efficient Searching & Updates** for user-friendly book management.

ASSIGNMENT SOLUTION: IMPLEMENT A SIMPLE OS-LEVEL PROCESS MANAGER IN C

OBJECTIVE

The goal of this assignment is to develop a **simple process manager** in **C** that:

- **Creates and manages multiple processes**
- **Monitors running processes**
- **Allows users to terminate processes**
- **Displays process information (PID, status, etc.)**

This process manager will use **system calls** such as fork(), exec(), wait(), and kill() to **interact with the OS**.

Step-by-Step Guide

STEP 1: UNDERSTANDING THE PROCESS MANAGER

A process manager must:

1. **Create new processes**
2. **List currently running processes**
3. **Terminate processes**
4. **Display process statuses**

Step 2: System Calls Required

System Call	Purpose

fork()	Creates a new child process
exec()	Replaces process image with a new program
getpid()	Gets the Process ID (PID)
wait()	Waits for a child process to finish
kill()	Terminates a process
getppid()	Gets the Parent Process ID (PPID)

STEP 3: DESIGN THE PROCESS MANAGER

- The program will display a **menu-driven interface**.
- The user can **create, list, or terminate processes**.
- **Processes are stored in a linked list** for easy management.

STEP 4: IMPLEMENTING THE PROCESS MANAGER

Complete Code Implementation

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <signal.h>
```

```
// Structure to store process information
struct Process {
   pid_t pid;
   char name[50];
   struct Process *next;
};

struct Process *head = NULL;

// Function to Add Process to Linked List
void addProcess(pid_t pid, char *name) {
   struct Process *newProcess = (struct Process*)malloc(sizeof(struct Process));
   newProcess->pid = pid;
   snprintf(newProcess->name, sizeof(newProcess->name), "%s", name);
   newProcess->next = head;
   head = newProcess;
}

// Function to Remove Process from Linked List
```

```
void removeProcess(pid_t pid) {
    struct Process *current = head, *prev = NULL;

    while (current != NULL) {
        if (current->pid == pid) {
            if (prev == NULL)
                head = current->next;
            else
                prev->next = current->next;

            free(current);
            return;
        }
        prev = current;
        current = current->next;
    }
}

// Function to List Active Processes
void listProcesses() {
    struct Process *current = head;
```

```
    if (!current) {
        printf("No active processes.\n");
        return;
    }

    printf("\nActive Processes:\n");
    printf("PID\tName\n");
    while (current) {
        printf("%d\t%s\n", current->pid, current->name);
        current = current->next;
    }
}

// Function to Create a New Process
void createProcess() {
    pid_t pid = fork();

    if (pid < 0) {
        printf("Process creation failed!\n");
    } else if (pid == 0) {
```

```
        // Child process
        printf("New process created: PID = %d\n", getpid());
        execlp("sleep", "sleep", "60", NULL); // Simulate a running process
        exit(0);
    } else {
        // Parent process
        addProcess(pid, "sleep");
        printf("Process added: PID = %d\n", pid);
    }
}

// Function to Terminate a Process
void terminateProcess() {
    int pid;
    printf("Enter PID to terminate: ");
    scanf("%d", &pid);

    if (kill(pid, SIGTERM) == 0) {
        removeProcess(pid);
        printf("Process %d terminated successfully.\n", pid);
```

```
    } else {
        printf("Failed to terminate process %d. Process may not exist.\n", pid);
    }
}

// Menu for Process Manager
void menu() {
    int choice;
    while (1) {
        printf("\nSimple OS-Level Process Manager\n");
        printf("1. Create Process\n");
        printf("2. List Processes\n");
        printf("3. Terminate Process\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: createProcess(); break;
            case 2: listProcesses(); break;
```

```
            case 3: terminateProcess(); break;
            case 4: exit(0);
            default: printf("Invalid choice! Try again.\n");
        }
    }
}

// Main Function
int main() {
    menu();
    return 0;
}
```

STEP 5: EXPLANATION OF THE CODE

1. Process Storage Using a Linked List

- Each process is stored in a **linked list**, containing:
  - **PID** (Process ID)
  - **Process Name**
  - **Pointer to next process**

2. Process Creation (createProcess())

- Uses fork() to **create a child process**.

- The child process runs sleep 60 using execlp().
- Parent stores the new process in the linked list.

3. Process Termination (terminateProcess())

- Takes **PID as input** and sends SIGTERM signal using kill().
- The process is removed from the linked list.

4. Listing Processes (listProcesses())

- Iterates over the linked list to **display all active processes**.

STEP 6: EXAMPLE RUNS

1. Creating a New Process

Simple OS-Level Process Manager

1. Create Process

2. List Processes

3. Terminate Process

4. Exit

Enter choice: 1

New process created: PID = 3456

Process added: PID = 3456

2. Listing Active Processes

Enter choice: 2

Active Processes:

PID Name

3456 sleep

3. Terminating a Process

Enter choice: 3

Enter PID to terminate: 3456

Process 3456 terminated successfully.

Step 7: Enhancements and Future Scope

1. Enhancements

- **Display process status** (Running, Suspended, Stopped).
- **Implement process priorities**.
- **Allow resuming and suspending processes**.

2. Real-World Applications

- **Task Managers in Operating Systems** (Linux top command).
- **Process Monitoring in Embedded Systems**.
- **Job Scheduling in Cloud Computing**.

Step 8: Additional Exercises

1. **Modify the program to suspend (SIGSTOP) and resume (SIGCONT) processes.**

2. **Implement process scheduling to prioritize tasks.**
3. **Enhance the program to allow running custom commands instead of sleep.**
4. **Track CPU usage and memory consumption for each process.**

CONCLUSION

This **OS-Level Process Manager** in C:

- **Creates, lists, and terminates processes**.
- **Uses system calls (fork(), exec(), kill(), wait())**.
- **Stores process details dynamically using linked lists**.

Assignment Solution: Build a Multi-Threaded Chat Application Using Socket Programming in C

Objective

The objective of this assignment is to develop a **multi-threaded chat application** in **C** using **socket programming**. The chat system will:

- **Allow multiple clients to connect** to a server.
- **Enable real-time messaging** between clients via the server.
- **Utilize multi-threading (pthread)** to handle multiple client connections.

Step-by-Step Guide

Step 1: Understanding the Chat Application

The chat application consists of:

1. **A Server**
   - Listens for incoming client connections.
   - Uses **threads** to handle multiple clients concurrently.
   - Forwards messages from one client to others.
2. **Multiple Clients**
   - Connect to the server using sockets.

- Send messages to the server, which then distributes them to all connected clients.

Step 2: Required Libraries

Library	Usage
sys/socket.h	Provides socket functions (socket(), bind(), listen(), accept()).
netinet/in.h	Defines internet address structures.
pthread.h	Enables multi-threading for handling multiple clients.
string.h	Provides string manipulation functions.
unistd.h	Enables system calls like close().

Step 3: Implement the Chat Server

The **server**:

- Listens for **client connections**.
- Uses **threads** to handle multiple clients.
- Broadcasts messages to **all connected clients**.

Server Code Implementation

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>
```

```
#include <unistd.h>

#include <pthread.h>

#include <arpa/inet.h>


#define PORT 8080

#define MAX_CLIENTS 10

#define BUFFER_SIZE 1024


// Client structure

typedef struct {

   int socket;

   struct sockaddr_in address;

} Client;


Client *clients[MAX_CLIENTS]; // Array to store client information

pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;


// Broadcast message to all clients

void broadcastMessage(char *message, int sender_socket) {

   pthread_mutex_lock(&clients_mutex);

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

```
        if (clients[i] != NULL && clients[i]->socket != sender_socket) {
            send(clients[i]->socket, message, strlen(message), 0);
        }
    }
    pthread_mutex_unlock(&clients_mutex);
}

// Handle client communication
void *handleClient(void *arg) {
    Client *client = (Client *)arg;
    char buffer[BUFFER_SIZE];

    while (1) {
        memset(buffer, 0, BUFFER_SIZE);
        int bytes_received = recv(client->socket, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) {
            printf("Client disconnected.\n");
            close(client->socket);

            pthread_mutex_lock(&clients_mutex);
```

```
            for (int i = 0; i < MAX_CLIENTS; i++) {
                if (clients[i] == client) {
                    clients[i] = NULL;
                    break;
                }
            }
            pthread_mutex_unlock(&clients_mutex);
            free(client);
            pthread_exit(NULL);
        }

        printf("Client: %s\n", buffer);
        broadcastMessage(buffer, client->socket);
    }
}

// Main function to start the server
int main() {
    int server_socket, new_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t addr_size = sizeof(client_addr);
```

```
    // Create socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Socket creation failed");
        return EXIT_FAILURE;
    }

    // Set up server address structure
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind socket
    if (bind(server_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("Binding failed");
        return EXIT_FAILURE;
    }

    // Listen for clients
```

```
    if (listen(server_socket, MAX_CLIENTS) < 0) {
        perror("Listening failed");
        return EXIT_FAILURE;
    }

    printf("Chat Server Started on Port %d...\n", PORT);

    while (1) {
        // Accept a new client connection
        new_socket = accept(server_socket, (struct sockaddr
*)&client_addr, &addr_size);
        if (new_socket < 0) {
            perror("Client connection failed");
            continue;
        }

        // Allocate memory for the new client
        Client *new_client = (Client *)malloc(sizeof(Client));
        new_client->socket = new_socket;
        new_client->address = client_addr;
```

```
    pthread_mutex_lock(&clients_mutex);
    for (int i = 0; i < MAX_CLIENTS; i++) {
      if (clients[i] == NULL) {
        clients[i] = new_client;
        pthread_t thread;
        pthread_create(&thread, NULL, handleClient, (void *)new_client);
        pthread_detach(thread);
        break;
      }
    }
    pthread_mutex_unlock(&clients_mutex);

    printf("New client connected.\n");
  }

  return 0;
}
```

Step 4: Implement the Chat Client

The **client**:

- Connects to the **server**.
- Reads user input and sends it to the server.
- Receives and displays messages from other clients.

Client Code Implementation

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <arpa/inet.h>


#define PORT 8080

#define BUFFER_SIZE 1024


int client_socket;


// Function to receive messages from server

void *receiveMessages(void *arg) {

  char buffer[BUFFER_SIZE];

  while (1) {

    memset(buffer, 0, BUFFER_SIZE);
```

```
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE,
0);
        if (bytes_received <= 0) {
            printf("Server disconnected.\n");
            close(client_socket);
            exit(0);
        }
        printf("\nMessage: %s\n", buffer);
    }
}

// Main function for client
int main() {
    struct sockaddr_in server_addr;
    char message[BUFFER_SIZE];

    // Create socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1) {
        perror("Socket creation failed");
        return EXIT_FAILURE;
```

```
}

// Set up server address structure
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

// Connect to server
if (connect(client_socket, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
    perror("Connection to server failed");
    return EXIT_FAILURE;
}

printf("Connected to the chat server...\n");

// Create thread to receive messages
pthread_t receive_thread;
pthread_create(&receive_thread, NULL, receiveMessages, NULL);

// Send messages to server
```

```
  while (1) {
    printf("Enter message: ");
    fgets(message, BUFFER_SIZE, stdin);
    send(client_socket, message, strlen(message), 0);
  }

  return 0;
}
```

Step 5: Running the Application

1. Compile the Programs

```
gcc server.c -o server -pthread
gcc client.c -o client -pthread
```

2. Start the Server

```
./server
```

Expected Output:

Chat Server Started on Port 8080...

3. Start Multiple Clients

```
./client
```

Expected Output on Each Client:

Connected to the chat server...

Enter message:

4. Chat Example

Client 1 Sends:

Hello, everyone!

Client 2 Receives:

Message: Hello, everyone!

STEP 6: ENHANCEMENTS

1. **Implement private messaging** by allowing clients to send messages to specific users.
2. **Encrypt messages** for secure communication.
3. **Add user authentication** (username/password).
4. **Allow file sharing** over the chat system.

STEP 7: CONCLUSION

This **multi-threaded chat application**:

- Uses **sockets** for network communication.
- Implements **multi-threading** to handle multiple clients.
- Enables **real-time message broadcasting**.