Assignment -12

1.Inventory Update System

Input: An array of integers representing inventory levels and an array of changes in stock.

Process: Pass the arrays to a function by reference to update inventory levels.

Output: Print the updated inventory levels and flag items below the restocking threshold.

Concepts: Arrays, functions, pass by reference, decision-making (if-else).

Sol:
```
#include <stdio.h>

#define THRESHOLD 5  // Restocking threshold

// Function to update inventory levels
void updateInventory(int inventory[], int changes[], int size) {
    for (int i = 0; i < size; i++) {
        inventory[i] += changes[i];  // Update inventory with stock changes
        if (inventory[i] < THRESHOLD) {
            printf("Item %d needs restocking. Current level: %d\n", i + 1, inventory[i]);
        }
    }
}

int main() {
    int inventory[] = {10, 3, 7, 2, 15};  // Initial inventory levels
    int changes[] = {-3, 2, -1, -5, 4};   // Stock changes (increase or decrease)
    int size = sizeof(inventory) / sizeof(inventory[0]);
```

```
    printf("Initial Inventory Levels:\n");
    for (int i = 0; i < size; i++) {
        printf("Item %d: %d\n", i + 1, inventory[i]);
    }

    updateInventory(inventory, changes, size);  // Call function to update inventory

    printf("\nUpdated Inventory Levels:\n");
    for (int i = 0; i < size; i++) {
        printf("Item %d: %d\n", i + 1, inventory[i]);
    }

    return 0;
}
```

O/p: Initial Inventory Levels:

Item 1: 10

Item 2: 3

Item 3: 7

Item 4: 2

Item 5: 15

Item 4 needs restocking. Current level: -3

Updated Inventory Levels:

Item 1: 7

Item 2: 5

Item 3: 6

Item 4: -3

Item 5: 19

2.Product Price Adjustment

Input: An array of demand levels (constant) and an array of product prices.

Process: Use a function to calculate new prices based on demand levels. The function should return a pointer to an array of adjusted prices.

Output: Display the original and adjusted prices.

Concepts: Passing constant data, functions, pointers, arrays.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of products

// Function to calculate adjusted prices based on demand
float* adjustPrices(const int demand[], float prices[], int size) {
    static float adjustedPrices[SIZE];  // Array to store adjusted prices
    for (int i = 0; i < size; i++) {
        if (demand[i] > 10) {
            adjustedPrices[i] = prices[i] * 1.10;  // Increase price by 10% if demand > 10
        } else {
            adjustedPrices[i] = prices[i] * 0.90;  // Decrease price by 10% otherwise
        }
    }
```

```
    return adjustedPrices;  // Return pointer to adjusted prices
}

int main() {
    const int demand[] = {12, 8, 15, 5, 10};  // Demand levels (constant)
    float prices[] = {100.0, 150.0, 200.0, 80.0, 120.0};  // Original product prices

    printf("Original Prices:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Product %d: $%.2f\n", i + 1, prices[i]);
    }

    float* adjustedPrices = adjustPrices(demand, prices, SIZE);  // Calculate
adjusted prices

    printf("\nAdjusted Prices:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Product %d: $%.2f\n", i + 1, adjustedPrices[i]);
    }

    return 0;
}
```

o/p: Original Prices:

Product 1: $100.00

Product 2: $150.00

Product 3: $200.00

Product 4: $80.00

Product 5: $120.00

Adjusted Prices:

Product 1: $110.00

Product 2: $135.00

Product 3: $220.00

Product 4: $72.00

Product 5: $108.00

3.Daily Sales Tracker

Input: Array of daily sales amounts.

Process: Use do-while to validate sales data input. Use a function to calculate total sales using pointers.

Output: Display total sales for the day.

Concepts: Loops, arrays, pointers, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of sales entries

// Function to calculate total sales using pointers
float calculateTotalSales(float *sales, int size) {
    float total = 0;
    for (int i = 0; i < size; i++) {
        total += *(sales + i);  // Access elements using pointer arithmetic
    }
```

```c
    return total;
}

int main() {
    float sales[SIZE];
    int i = 0;

    printf("Enter sales data for %d entries:\n", SIZE);

    // Input validation using do-while loop
    do {
        printf("Sales amount for entry %d: ", i + 1);
        scanf("%f", &sales[i]);
        if (sales[i] < 0) {
            printf("Invalid input. Sales amount cannot be negative. Try again.\n");
        } else {
            i++;
        }
    } while (i < SIZE);

    // Calculate total sales
    float totalSales = calculateTotalSales(sales, SIZE);

    // Display total sales
    printf("Total sales for the day: $%.2f\n", totalSales);
```

```
    return 0;
}
```

O/p:

Enter sales data for 5 entries:

Sales amount for entry 1: 100

Sales amount for entry 2: 300

Sales amount for entry 3: 200

Sales amount for entry 4: 334

Sales amount for entry 5: 500

Total sales for the day: $1434.00

4. Discount Decision System

Input: Array of sales volumes.

Process: Pass the sales volume array by reference to a function. Use a switch statement to assign discount rates.

Output: Print discount rates for each product.

Concepts: Decision-making (switch), arrays, pass by reference, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of products

// Function to assign discount rates based on sales volumes
void assignDiscounts(int sales[], float discounts[], int size) {
    for (int i = 0; i < size; i++) {
        switch (sales[i] / 100) {
```

```
            case 0:
            case 1:  // Sales between 0 and 199
                discounts[i] = 0.05;  // 5% discount
                break;
            case 2:  // Sales between 200 and 299
                discounts[i] = 0.10;  // 10% discount
                break;
            case 3:  // Sales between 300 and 399
                discounts[i] = 0.15;  // 15% discount
                break;
            default:  // Sales 400 and above
                discounts[i] = 0.20;  // 20% discount
                break;
        }
    }
}

int main() {
    int sales[SIZE] = {150, 250, 320, 180, 450};  // Sales volumes
    float discounts[SIZE];  // Array to store discount rates

    assignDiscounts(sales, discounts, SIZE);  // Pass arrays by reference

    printf("Sales Volume and Discount Rates:\n");
    for (int i = 0; i < SIZE; i++) {
```

```
        printf("Product %d: Sales = %d, Discount = %.0f%%\n", i + 1, sales[i],
discounts[i] * 100);
    }

    return 0;
}
```

O/p: Sales Volume and Discount Rates:

Product 1: Sales = 150, Discount = 5%

Product 2: Sales = 250, Discount = 10%

Product 3: Sales = 320, Discount = 15%

Product 4: Sales = 180, Discount = 5%

Product 5: Sales = 450, Discount = 20%

5. Transaction Anomaly Detector

Input: Array of transaction amounts.

Process: Use pointers to traverse the array. Classify transactions as "Normal" or "Suspicious" based on thresholds using if-else.

Output: Print classification for each transaction.

Concepts: Arrays, pointers, loops, decision-making.

Sol:

```
#include <stdio.h>

#define SIZE 5  // Number of transactions
#define SUSPICIOUS_THRESHOLD 1000  // Threshold for suspicious
transactions

// Function to classify transactions using pointers
```

```
void classifyTransactions(float *transactions, int size) {
    for (int i = 0; i < size; i++) {
        if (*(transactions + i) > SUSPICIOUS_THRESHOLD) {
            printf("Transaction %d: $%.2f - Suspicious\n", i + 1, *(transactions + i));
        } else {
            printf("Transaction %d: $%.2f - Normal\n", i + 1, *(transactions + i));
        }
    }
}

int main() {
    float transactions[SIZE] = {200.50, 1500.75, 750.30, 1200.40, 450.60};  // 
Transaction amounts

    classifyTransactions(transactions, SIZE);  // Call function to classify transactions

    return 0;
}
```

O/p: Transaction 1: $200.50 - Normal

Transaction 2: $1500.75 - Suspicious

Transaction 3: $750.30 - Normal

Transaction 4: $1200.40 - Suspicious

Transaction 5: $450.60 - Normal

6. Account Balance Operations

Input: Array of account balances.

Process: Pass the balances array to a function that calculates interest. Return a pointer to the updated balances array.

Output: Display updated balances.

Concepts: Functions, arrays, pointers, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of accounts
#define INTEREST_RATE 0.05  // 5% interest rate

// Function to calculate interest on account balances
float* applyInterest(float *balances, int size) {
    static float updatedBalances[SIZE];  // Array to store updated balances
    for (int i = 0; i < size; i++) {
        updatedBalances[i] = *(balances + i) * (1 + INTEREST_RATE);  // Apply interest
    }
    return updatedBalances;  // Return pointer to the updated balances array
}

int main() {
    float balances[SIZE] = {1000.50, 2000.75, 1500.20, 3000.00, 1200.10};  // Initial account balances

    // Call the function to apply interest and get the updated balances
    float* updatedBalances = applyInterest(balances, SIZE);
```

```
    // Display the updated balances
    printf("Updated Account Balances after Interest:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Account %d: $%.2f\n", i + 1, *(updatedBalances + i));
    }

    return 0;
}
```

O/p:

```
Updated Account Balances after Interest:
Account 1: $1050.53
Account 2: $2100.79
Account 3: $1575.21
Account 4: $3150.00
Account 5: $1260.10
```

7.Bank Statement Generator

Input: Array of transaction types (e.g., 1 for Deposit, 2 for Withdrawal) and amounts.

Process: Use a switch statement to classify transactions. Pass the array as a constant parameter to a function.

Output: Summarize total deposits and withdrawals.

Concepts: Decision-making, passing constant data, arrays, functions.

Sol:

```
#include <stdio.h>

#define SIZE 5  // Number of transactions
```

```
// Function to classify transactions and summarize totals
void summarizeTransactions(const int types[], const float amounts[], int size) {
    float totalDeposits = 0, totalWithdrawals = 0;

    for (int i = 0; i < size; i++) {
        switch (types[i]) {
            case 1:  // Deposit
                totalDeposits += amounts[i];
                break;
            case 2:  // Withdrawal
                totalWithdrawals += amounts[i];
                break;
            default:  // Invalid transaction type
                printf("Transaction %d: Invalid transaction type\n", i + 1);
                break;
        }
    }

    printf("Total Deposits: $%.2f\n", totalDeposits);
    printf("Total Withdrawals: $%.2f\n", totalWithdrawals);
}

int main() {
    const int transactionTypes[SIZE] = {1, 2, 1, 2, 1};  // 1 for Deposit, 2 for
Withdrawal
```

```
    const float transactionAmounts[SIZE] = {500.0, 200.0, 150.0, 300.0, 400.0};  // Transaction amounts

    summarizeTransactions(transactionTypes, transactionAmounts, SIZE);  // Call function

    return 0;
}
```

O/p: Total Deposits: $1050.00

Total Withdrawals: $500.00

8.Loan Eligibility Check

Input: Array of customer credit scores.

Process: Use if-else to check eligibility criteria. Use pointers to update eligibility status.

Output: Print customer eligibility statuses.

Concepts: Decision-making, arrays, pointers, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of customers
#define ELIGIBILITY_THRESHOLD 650  // Minimum credit score for loan eligibility

// Function to check loan eligibility
void checkEligibility(const int *scores, char *status, int size) {
    for (int i = 0; i < size; i++) {
        if (*(scores + i) >= ELIGIBILITY_THRESHOLD) {
```

```
            *(status + i) = 'Y';  // Eligible
        } else {
            *(status + i) = 'N';  // Not eligible
        }
    }
}

int main() {
    const int creditScores[SIZE] = {720, 580, 680, 600, 750};  // Customer credit
scores
    char eligibilityStatus[SIZE];  // Array to store eligibility status ('Y' for Yes, 'N'
for No)

    checkEligibility(creditScores, eligibilityStatus, SIZE);  // Call function to check
eligibility

    // Display eligibility statuses
    printf("Customer Loan Eligibility Status:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Customer %d: Credit Score = %d, Eligible = %c\n", i + 1,
creditScores[i], eligibilityStatus[i]);
    }

    return 0;
}
```

O/p: Customer Loan Eligibility Status:

Customer 1: Credit Score = 720, Eligible = Y

Customer 2: Credit Score = 580, Eligible = N

Customer 3: Credit Score = 680, Eligible = Y

Customer 4: Credit Score = 600, Eligible = N

Customer 5: Credit Score = 750, Eligible = Y

9.Order Total Calculator

Input: Array of item prices.

Process: Pass the array to a function. Use pointers to calculate the total cost.

Output: Display the total order value.

Concepts: Arrays, pointers, functions, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of items

// Function to calculate total order value using pointers
float calculateTotal(const float *prices, int size) {
    float total = 0;
    for (int i = 0; i < size; i++) {
        total += *(prices + i);  // Access array elements using pointer
    }
    return total;
}

int main() {
    float itemPrices[SIZE] = {10.50, 20.75, 15.30, 40.60, 25.90};  // Array of item prices
```

```
    // Call function to calculate the total order value
    float totalOrderValue = calculateTotal(itemPrices, SIZE);

    // Display the total order value
    printf("Total Order Value: $%.2f\n", totalOrderValue);

    return 0;
}
```

O/p: Total Order Value: $113.05

10.Stock Replenishment Alert

Input: Array of inventory levels.

Process: Use a function to flag products below a threshold. Return a pointer to flagged indices.

Output: Display flagged product indices.

Concepts: Arrays, functions returning pointers, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of products
#define THRESHOLD 50  // Replenishment threshold

// Function to find indices of products below threshold
int* findLowStockIndices(const int *inventory, int size, int *count) {
    static int flaggedIndices[SIZE];  // Static array to store flagged indices
    *count = 0;  // Initialize count of flagged items
```

```
    for (int i = 0; i < size; i++) {
        if (*(inventory + i) < THRESHOLD) {
            flaggedIndices[*count] = i;  // Store index of low-stock product
            (*count)++;  // Increment count
        }
    }
    return flaggedIndices;  // Return pointer to array of flagged indices
}

int main() {
    const int inventoryLevels[SIZE] = {30, 60, 20, 80, 10};  // Inventory levels
    int flaggedCount = 0;  // To store the number of flagged products

    // Call function to find low stock indices
    int *lowStockIndices = findLowStockIndices(inventoryLevels, SIZE,
&flaggedCount);

    // Display flagged product indices
    printf("Products below replenishment threshold:\n");
    for (int i = 0; i < flaggedCount; i++) {
        printf("Product %d (Index %d)\n", i + 1, lowStockIndices[i]);
    }

    return 0;
}
```

O/P: Products below replenishment threshold:

Product 1 (Index 0)

Product 2 (Index 2)

Product 3 (Index 4)

11.Customer Reward Points

Input: Array of customer purchase amounts.

Process: Pass the purchase array by reference to a function that calculates reward points using if-else.

Output: Display reward points for each customer.

Concepts: Arrays, functions, pass by reference, decision-making.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of customers

// Function to calculate reward points based on purchase amounts
void calculateRewardPoints(const float *purchases, int *rewardPoints, int size) {
    for (int i = 0; i < size; i++) {
        if (*(purchases + i) >= 500) {
            rewardPoints[i] = 50;  // 50 points for purchases >= 500
        } else if (*(purchases + i) >= 200) {
            rewardPoints[i] = 20;  // 20 points for purchases between 200 and 499
        } else {
            rewardPoints[i] = 5;   // 5 points for purchases below 200
        }
    }
```

```
}

int main() {
    float purchases[SIZE] = {150.0, 300.0, 550.0, 100.0, 700.0};  // Customer purchase amounts
    int rewardPoints[SIZE];  // Array to store reward points for each customer

    // Call function to calculate reward points
    calculateRewardPoints(purchases, rewardPoints, SIZE);

    // Display reward points for each customer
    printf("Customer Reward Points:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Customer %d: Purchase = $%.2f, Reward Points = %d\n", i + 1, purchases[i], rewardPoints[i]);
    }

    return 0;
}
```

O/p: Customer Reward Points:

Customer 1: Purchase = $150.00, Reward Points = 5

Customer 2: Purchase = $300.00, Reward Points = 20

Customer 3: Purchase = $550.00, Reward Points = 50

Customer 4: Purchase = $100.00, Reward Points = 5

Customer 5: Purchase = $700.00, Reward Points = 50

12.Shipping Cost Estimator

Input: Array of order weights and shipping zones.

Process: Use a switch statement to calculate shipping costs based on zones. Pass the weight array as a constant parameter.

Output: Print the shipping cost for each order.

Concepts: Decision-making, passing constant data, arrays, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of orders

// Function to calculate shipping cost
void calculateShippingCosts(const float *weights, const int *zones, float *shippingCosts, int size) {
    for (int i = 0; i < size; i++) {
        switch (zones[i]) {
            case 1:
                shippingCosts[i] = weights[i] * 5.0;  // Zone 1: $5 per unit weight
                break;
            case 2:
                shippingCosts[i] = weights[i] * 7.5;  // Zone 2: $7.5 per unit weight
                break;
            case 3:
                shippingCosts[i] = weights[i] * 10.0;  // Zone 3: $10 per unit weight
                break;
            default:
                shippingCosts[i] = 0;  // Invalid zone
```

```
                printf("Order %d: Invalid shipping zone\n", i + 1);
                break;
        }
    }
}

int main() {
    const float orderWeights[SIZE] = {2.5, 4.0, 1.2, 5.0, 3.5};  // Order weights
    const int shippingZones[SIZE] = {1, 2, 3, 1, 2};  // Shipping zones
    float shippingCosts[SIZE];  // Array to store shipping costs

    // Call function to calculate shipping costs
    calculateShippingCosts(orderWeights, shippingZones, shippingCosts, SIZE);

    // Display shipping costs
    printf("Shipping Costs for Orders:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Order %d: Weight = %.2f, Zone = %d, Shipping Cost = $%.2f\n", i + 1, orderWeights[i], shippingZones[i], shippingCosts[i]);
    }

    return 0;
}
```

O/p:

```
Shipping Costs for Orders:
Order 1: Weight = 2.50, Zone = 1, Shipping Cost = $12.50
```

Order 2: Weight = 4.00, Zone = 2, Shipping Cost = $30.00

Order 3: Weight = 1.20, Zone = 3, Shipping Cost = $12.00

Order 4: Weight = 5.00, Zone = 1, Shipping Cost = $25.00

Order 5: Weight = 3.50, Zone = 2, Shipping Cost = $26.25

13. Missile Trajectory Analysis

Input: Array of trajectory data points.

Process: Use functions to find maximum and minimum altitudes. Use pointers to access data.

Output: Display maximum and minimum altitudes.

Concepts: Arrays, pointers, functions.

Sol:
```
#include <stdio.h>

#define SIZE 6  // Number of trajectory data points

// Function to find the maximum altitude
float findMaxAltitude(const float *altitudes, int size) {
    float maxAltitude = *altitudes;  // Initialize with the first element
    for (int i = 1; i < size; i++) {
        if (*(altitudes + i) > maxAltitude) {
            maxAltitude = *(altitudes + i);
        }
    }
    return maxAltitude;
}
```

```
// Function to find the minimum altitude
float findMinAltitude(const float *altitudes, int size) {
    float minAltitude = *altitudes;  // Initialize with the first element
    for (int i = 1; i < size; i++) {
        if (*(altitudes + i) < minAltitude) {
            minAltitude = *(altitudes + i);
        }
    }
    return minAltitude;
}

int main() {
    float trajectoryData[SIZE] = {500.0, 750.0, 1200.5, 980.3, 450.0, 1300.0};  //
Trajectory altitudes

    // Find and display maximum and minimum altitudes
    float maxAltitude = findMaxAltitude(trajectoryData, SIZE);
    float minAltitude = findMinAltitude(trajectoryData, SIZE);

    printf("Maximum Altitude: %.2f meters\n", maxAltitude);
    printf("Minimum Altitude: %.2f meters\n", minAltitude);

    return 0;
}
```

O/p: Maximum Altitude: 1300.00 meters

Minimum Altitude: 450.00 meters

14. Target Identification System

Input: Array of radar signal intensities.

Process: Classify signals into categories using a switch statement. Return a pointer to the array of classifications.

Output: Display classified signal types.

Concepts: Decision-making, functions returning pointers, arrays.

Sol:
```
#include <stdio.h>

#define SIZE 6  // Number of radar signals

// Function to classify signal intensities
const char* classifySignal(float intensity) {
    switch ((int)intensity / 100) {
        case 0: return "Weak";
        case 1: return "Moderate";
        case 2: return "Strong";
        default: return "Very Strong";
    }
}

// Function to classify all signals and return a pointer to the classifications
void classifySignals(const float *signals, const char *classifications[], int size) {
    for (int i = 0; i < size; i++) {
        classifications[i] = classifySignal(signals[i]);
    }
```

```
}

int main() {
    float radarSignals[SIZE] = {50.0, 120.5, 250.0, 300.3, 180.0, 400.0};  // Radar
signal intensities
    const char *signalClassifications[SIZE];  // Array of classification strings

    // Classify the signals
    classifySignals(radarSignals, signalClassifications, SIZE);

    // Display the classified signal types
    printf("Radar Signal Classifications:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Signal %d: Intensity = %.2f, Classification = %s\n", i + 1,
radarSignals[i], signalClassifications[i]);
    }

    return 0;
}
```

O/p: Radar Signal Classifications:

Signal 1: Intensity = 50.00, Classification = Weak

Signal 2: Intensity = 120.50, Classification = Moderate

Signal 3: Intensity = 250.00, Classification = Strong

Signal 4: Intensity = 300.30, Classification = Very Strong

Signal 5: Intensity = 180.00, Classification = Moderate

Signal 6: Intensity = 400.00, Classification = Very Strong

15.Threat Level Assessment

Input: Array of sensor readings.

Process: Pass the array by reference to a function that uses if-else to categorize threats.

Output: Display categorized threat levels.

Concepts: Arrays, functions, pass by reference, decision-making.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of sensor readings

// Function to categorize threat levels
void assessThreatLevels(const float *readings, char *levels[], int size) {
    for (int i = 0; i < size; i++) {
        if (readings[i] < 50.0) {
            levels[i] = "Low";
        } else if (readings[i] < 150.0) {
            levels[i] = "Medium";
        } else if (readings[i] < 300.0) {
            levels[i] = "High";
        } else {
            levels[i] = "Critical";
        }
    }
}
```

```
int main() {
    float sensorReadings[SIZE] = {30.5, 100.0, 200.0, 350.0, 75.5};  // Sensor readings
    const char *threatLevels[SIZE];  // Array to store threat levels


    // Call function to assess threat levels
    assessThreatLevels(sensorReadings, threatLevels, SIZE);


    // Display categorized threat levels
    printf("Threat Level Assessment:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Sensor Reading %.2f: Threat Level = %s\n", sensorReadings[i], threatLevels[i]);
    }


    return 0;
}
```

O/p: Threat Level Assessment:

Sensor Reading 30.50: Threat Level = Low

Sensor Reading 100.00: Threat Level = Medium

Sensor Reading 200.00: Threat Level = High

Sensor Reading 350.00: Threat Level = Critical

Sensor Reading 75.50: Threat Level = Medium

16. Signal Calibration

Input: Array of raw signal data.

Process: Use a function to adjust signal values by reference. Use pointers for data traversal.

Output: Print calibrated signal values.

Concepts: Arrays, pointers, functions, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of signals

// Function to calibrate signal values
void calibrateSignals(float *signals, int size, float adjustmentFactor) {
    for (int i = 0; i < size; i++) {
        *(signals + i) *= adjustmentFactor;  // Adjust signal value using pointer
arithmetic
    }
}

int main() {
    float rawSignals[SIZE] = {10.0, 20.5, 15.3, 30.0, 25.7};  // Raw signal data
    float adjustmentFactor = 1.1;  // Calibration factor (example)

    // Call function to calibrate signals
    calibrateSignals(rawSignals, SIZE, adjustmentFactor);

    // Display calibrated signal values
    printf("Calibrated Signal Values:\n");
    for (int i = 0; i < SIZE; i++) {
```

```
        printf("Signal %d: %.2f\n", i + 1, rawSignals[i]);
    }

    return 0;
}
```

O/p: Calibrated Signal Values:

Signal 1: 11.00

Signal 2: 22.55

Signal 3: 16.83

Signal 4: 33.00

Signal 5: 28.27

17.Matrix Row Sum

Input: 2D array representing a matrix.

Process: Write a function that calculates the sum of each row. The function returns a pointer to an array of row sums.

Output: Display the row sums.

Concepts: Arrays, functions returning pointers, loops.

Sol:
```
#include <stdio.h>

#define ROWS 3  // Number of rows
#define COLS 4  // Number of columns

// Function to calculate row sums
int* rowSum(int matrix[ROWS][COLS], int size) {
    static int sums[ROWS];  // Array to store row sums
```

```
    for (int i = 0; i < size; i++) {
        sums[i] = 0;
        for (int j = 0; j < COLS; j++) {
            sums[i] += matrix[i][j];  // Add elements of each row
        }
    }

    return sums;
}

int main() {
    int matrix[ROWS][COLS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };  // A 3x4 matrix

    // Call function to get row sums
    int *sums = rowSum(matrix, ROWS);

    // Display row sums
    printf("Row Sums:\n");
    for (int i = 0; i < ROWS; i++) {
        printf("Row %d Sum: %d\n", i + 1, sums[i]);
```

```
    }

    return 0;
}
```

O/p: Row Sums:

Row 1 Sum: 10

Row 2 Sum: 26

Row 3 Sum: 42

18.Statistical Mean Calculator

Input: Array of data points.

Process: Pass the data array as a constant parameter. Use pointers to calculate the mean.

Output: Print the mean value.

Concepts: Passing constant data, pointers, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of data points


// Function to calculate the mean of the data points
float calculateMean(const float *data, int size) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *(data + i);  // Access elements using pointer arithmetic
    }
    return sum / size;  // Return the mean
```

```
}

int main() {
    const float dataPoints[SIZE] = {12.5, 15.3, 18.2, 10.8, 14.4};  // Data points
array

    // Call function to calculate the mean
    float mean = calculateMean(dataPoints, SIZE);

    // Display the mean value
    printf("Mean of the Data Points: %.2f\n", mean);

    return 0;
}
```

O/p: Mean of the Data Points: 14.24

19. Temperature Gradient Analysis

Input: Array of temperature readings.

Process: Compute the gradient using a function that returns a pointer to the array of gradients.

Output: Display temperature gradients.

Concepts: Arrays, functions returning pointers, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of temperature readings
```

```
// Function to compute the temperature gradients
float* computeGradients(const float *temperatures, int size) {
    static float gradients[SIZE - 1];  // Array to store the temperature gradients

    for (int i = 0; i < size - 1; i++) {
        gradients[i] = temperatures[i + 1] - temperatures[i];  // Compute gradient
between consecutive readings
    }

    return gradients;
}

int main() {
    float temperatureReadings[SIZE] = {22.5, 24.0, 25.5, 23.0, 26.5};  //
Temperature readings

    // Call function to compute the gradients
    float *gradients = computeGradients(temperatureReadings, SIZE);

    // Display temperature gradients
    printf("Temperature Gradients:\n");
    for (int i = 0; i < SIZE - 1; i++) {
        printf("Gradient between T%d and T%d: %.2f°C\n", i + 1, i + 2, gradients[i]);
    }

    return 0;
```

}

O/p: Temperature Gradients:

Gradient between T1 and T2: 1.50°C

Gradient between T2 and T3: 1.50°C

Gradient between T3 and T4: -2.50°C

Gradient between T4 and T5: 3.50°C

20.Data Normalization

Input: Array of data points.

Process: Pass the array by reference to a function that normalizes values to a range of 0–1 using pointers.

Output: Display normalized values.

Concepts: Arrays, pointers, pass by reference, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of data points

// Function to normalize the data points
void normalizeData(float *data, int size) {
    float min = data[0], max = data[0];

    // Find the min and max values in the array
    for (int i = 1; i < size; i++) {
        if (*(data + i) < min) {
            min = *(data + i);
        }
```

```
        if (*(data + i) > max) {
            max = *(data + i);
        }
    }

    // Normalize the data to the range [0, 1]
    for (int i = 0; i < size; i++) {
        *(data + i) = (*(data + i) - min) / (max - min);
    }
}

int main() {
    float dataPoints[SIZE] = {12.5, 18.3, 14.7, 20.1, 16.8};  // Data points

    // Call function to normalize the data
    normalizeData(dataPoints, SIZE);

    // Display the normalized values
    printf("Normalized Data Points:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("Normalized Data %d: %.2f\n", i + 1, dataPoints[i]);
    }

    return 0;
}
```

O/p: Normalized Data Points:

Normalized Data 1: 0.00

Normalized Data 2: 0.76

Normalized Data 3: 0.29

Normalized Data 4: 1.00

Normalized Data 5: 0.57

21.Exam Score Analysis

Input: Array of student scores.

Process: Write a function that returns a pointer to the highest score. Use loops to calculate the average score.

Output: Display the highest and average scores.

Concepts: Arrays, functions returning pointers, loops.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of students

// Function to find the highest score
int* findHighestScore(int scores[], int size) {
    int *highest = &scores[0];  // Pointer to the first element

    for (int i = 1; i < size; i++) {
        if (scores[i] > *highest) {
            highest = &scores[i];  // Update pointer to the highest score
        }
    }
```

```
    return highest;
}

// Function to calculate the average score
float calculateAverage(int scores[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += scores[i];
    }
    return (float)sum / size;
}

int main() {
    int studentScores[SIZE] = {85, 90, 78, 88, 92};  // Array of student scores

    // Call function to find the highest score
    int *highestScore = findHighestScore(studentScores, SIZE);

    // Call function to calculate the average score
    float averageScore = calculateAverage(studentScores, SIZE);

    // Display the highest and average scores
    printf("Highest Score: %d\n", *highestScore);
    printf("Average Score: %.2f\n", averageScore);
```

```
    return 0;
}
```

O/p: Highest Score: 92

Average Score: 86.60

22.Grade Assignment

Input: Array of student marks.

Process: Pass the marks array by reference to a function. Use a switch statement to assign grades.

Output: Display grades for each student.

Concepts: Arrays, decision-making, pass by reference, functions.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of students

// Function to assign grades based on marks
void assignGrades(int *marks, char *grades, int size) {
    for (int i = 0; i < size; i++) {
        // Using a switch statement to assign grades
        switch (*(marks + i) / 10) {
            case 10:
            case 9:
                *(grades + i) = 'A';  // Grade A for 90 and above
                break;
            case 8:
```

```
            *(grades + i) = 'B';  // Grade B for 80-89
            break;
        case 7:
            *(grades + i) = 'C';  // Grade C for 70-79
            break;
        case 6:
            *(grades + i) = 'D';  // Grade D for 60-69
            break;
        default:
            *(grades + i) = 'F';  // Grade F for below 60
            break;
        }
    }
}

int main() {
    int studentMarks[SIZE] = {85, 92, 70, 55, 68};  // Array of student marks
    char studentGrades[SIZE];  // Array to store assigned grades

    // Call function to assign grades
    assignGrades(studentMarks, studentGrades, SIZE);

    // Display grades for each student
    printf("Student Grades:\n");
    for (int i = 0; i < SIZE; i++) {
```

```
        printf("Student %d: Marks = %d, Grade = %c\n", i + 1, studentMarks[i], studentGrades[i]);
    }

    return 0;
}
```

O/p: Student Grades:

Student 1: Marks = 85, Grade = B

Student 2: Marks = 92, Grade = A

Student 3: Marks = 70, Grade = C

Student 4: Marks = 55, Grade = F

Student 5: Marks = 68, Grade = D

23.Student Attendance Tracker

Input: Array of attendance percentages.

Process: Use pointers to traverse the array. Return a pointer to an array of defaulters.

Output: Display defaulters' indices.

Concepts: Arrays, pointers, functions returning pointers.

Sol:
```
#include <stdio.h>

#define SIZE 5  // Number of students

// Function to find the defaulters (students with attendance < 75%)
int* findDefaulters(float *attendance, int size) {
    static int defaulters[SIZE];  // Array to store indices of defaulters
```

```
    int defaulterCount = 0;

    // Traverse the attendance array and identify defaulters
    for (int i = 0; i < size; i++) {
        if (*(attendance + i) < 75.0) {
            defaulters[defaulterCount] = i;  // Store index of defaulter
            defaulterCount++;
        }
    }
    return defaulters;  // Return pointer to the defaulters array
}

int main() {
    float attendancePercentages[SIZE] = {80.5, 72.3, 60.4, 85.7, 74.9};  // Array of
attendance percentages

    // Call function to find defaulters
    int *defaulters = findDefaulters(attendancePercentages, SIZE);

    // Display indices of defaulters
    printf("Defaulters (attendance < 75%%):\n");
    for (int i = 0; i < SIZE; i++) {
        if (defaulters[i] != 0 || (attendancePercentages[defaulters[i]] < 75.0)) {
            printf("Student %d is a defaulter (Attendance: %.2f%%)\n", defaulters[i] +
1, attendancePercentages[defaulters[i]]);
        }
```

```
    }

    return 0;
}
```

O/p: Defaulters (attendance < 75%):

Student 2 is a defaulter (Attendance: 72.30%)

Student 3 is a defaulter (Attendance: 60.40%)

Student 5 is a defaulter (Attendance: 74.90%)

24.Quiz Performance Analyzer

Input: Array of quiz scores.

Process: Pass the array as a constant parameter to a function that uses if-else for performance categorization.

Output: Print categorized performance.

Concepts: Arrays, passing constant data, functions, decision-making

```
Sol: #include <stdio.h>

#define SIZE 5  // Number of students

// Function to categorize quiz performance
void categorizePerformance(const int *scores, int size) {
    for (int i = 0; i < size; i++) {
        // Categorize performance based on the score
        if (*(scores + i) >= 90) {
            printf("Student %d: Excellent Performance (Score: %d)\n", i + 1, *(scores + i));
```

```c
        } else if (*(scores + i) >= 75) {
            printf("Student %d: Good Performance (Score: %d)\n", i + 1, *(scores + i));
        } else if (*(scores + i) >= 50) {
            printf("Student %d: Average Performance (Score: %d)\n", i + 1, *(scores +
i));
        } else {
            printf("Student %d: Poor Performance (Score: %d)\n", i + 1, *(scores + i));
        }
    }
}

int main() {
    int quizScores[SIZE] = {85, 92, 78, 60, 45};  // Array of quiz scores

    // Call function to categorize performance
    categorizePerformance(quizScores, SIZE);

    return 0;
}
```

O/p: Student 1: Good Performance (Score: 85)

Student 2: Excellent Performance (Score: 92)

Student 3: Good Performance (Score: 78)

Student 4: Average Performance (Score: 60)

Student 5: Poor Performance (Score: 45)