1. Statistical Analysis Tool

o Function Prototype: void computeStats(const double *array, int size, double *average, double *variance)

o Data Types: const double*, int, double*

o Concepts: Pointers, arrays, functions, passing constant data, pass by reference.

o Details: Compute the average and variance of an array of experimental results, ensuring the function uses pointers for accessing the data and modifying the results.

Sol:

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

// Function prototype
void computeStats(const double *array, int size, double *average, double *variance);

int main() {
    double data[] = {10.5, 20.3, 30.8, 25.4, 15.9};
    int size = sizeof(data) / sizeof(data[0]);
    double avg = 0.0, var = 0.0;

    // Compute the average and variance
    computeStats(data, size, &avg, &var);

    // Display the results
    printf("Average: %.2f\n", avg);
    printf("Variance: %.2f\n", var);

    return 0;
}

// Function to compute average and variance
void computeStats(const double *array, int size, double *average, double *variance) {
    double sum = 0.0;
    double sumSquaredDiffs = 0.0;
```

```
    // Compute sum of elements
    for (int i = 0; i < size; i++) {
        sum += array[i];
    }

    // Compute average
    *average = sum / size;

    // Compute variance
    for (int i = 0; i < size; i++) {
        sumSquaredDiffs += pow(array[i] - *average, 2);
    }
    *variance = sumSquaredDiffs / size;
}
```

O/p:

Average: 20.58

Variance: 50.25

2. Data Normalization

- Function Prototype: double* normalizeData(const double *array, int size)
- Data Types: const double*, int, double*
- Concepts: Arrays, functions returning pointers, loops.
- Details: Normalize data points in an array, returning a pointer to the new normalized array.

Sol:

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

// Function prototype
double* normalizeData(const double *array, int size);

int main() {
```

```
    double data[] = {10.0, 20.0, 30.0, 40.0, 50.0};
    int size = sizeof(data) / sizeof(data[0]);

    // Normalize the data
    double *normalizedData = normalizeData(data, size);

    // Display the normalized data
    if (normalizedData != NULL) {
        printf("Normalized data:\n");
        for (int i = 0; i < size; i++) {
            printf("%.2f ", normalizedData[i]);
        }
        printf("\n");

        // Free the allocated memory
        free(normalizedData);
    }

    return 0;
}

// Function to normalize data
double* normalizeData(const double *array, int size) {
    if (size <= 0) return NULL;

    double min = array[0];
    double max = array[0];

    // Find min and max values
    for (int i = 1; i < size; i++) {
        if (array[i] < min) min = array[i];
```

```
    if (array[i] > max) max = array[i];
  }

  if (max == min) return NULL;  // Avoid division by zero if all values are the same

  // Allocate memory for normalized data
  double normalizedArray = (double)malloc(size * sizeof(double));
  if (normalizedArray == NULL) return NULL; // Memory allocation check

  // Normalize data
  for (int i = 0; i < size; i++) {
    normalizedArray[i] = (array[i] - min) / (max - min);
  }

  return normalizedArray;
}
```

O/p:

Normalized data:

0.00 0.25 0.50 0.75 1.00

3. Experimental Report Generator

o Function Prototype: void generateReport(const double *results, const char *descriptions[], int size)

o Data Types: const double*, const char*[], int

o Concepts: Strings, arrays, functions, passing constant data.

o Details: Generate a report summarizing experimental results and their descriptions, using constant data to ensure the input is not modified.

Sol:

```
#include <stdio.h>

// Function prototype
void generateReport(const double *results, const char *descriptions[], int size);

int main() {
```

```
    const char *descriptions[] = {
        "Temperature measurement",
        "Pressure reading",
        "Humidity level",
        "Wind speed"
    };
    double results[] = {23.5, 101.2, 45.8, 12.3};
    int size = sizeof(results) / sizeof(results[0]);

    // Generate the report
    generateReport(results, descriptions, size);

    return 0;
}

// Function to generate a report of results with descriptions
void generateReport(const double *results, const char *descriptions[], int size) {
    printf("\nExperimental Report\n");
    printf("---------------------\n");
    for (int i = 0; i < size; i++) {
        printf("%s: %.2f\n", descriptions[i], results[i]);
    }
}
```

O/p:

Experimental Report

Temperature measurement: 23.50

Pressure reading: 101.20

Humidity level: 45.80

Wind speed: 12.30

4. Data Anomaly Detector

o Function Prototype: void detectAnomalies(const double *data, int size, double threshold, int *anomalyCount)

o Data Types: const double*, int, double, int*

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

o Details: Detect anomalies in a dataset based on a threshold, updating the anomaly count by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void detectAnomalies(const double *data, int size, double threshold, int *anomalyCount);

int main() {
    double data[] = {10.5, 20.8, 5.1, 50.3, 12.0, 70.6};
    int size = sizeof(data) / sizeof(data[0]);
    double threshold = 25.0;
    int anomalyCount = 0;

    // Detect anomalies
    detectAnomalies(data, size, threshold, &anomalyCount);

    // Display the result
    printf("Number of anomalies detected: %d\n", anomalyCount);

    return 0;
}

// Function to detect anomalies in data
void detectAnomalies(const double *data, int size, double threshold, int *anomalyCount) {
    *anomalyCount = 0;

    for (int i = 0; i < size; i++) {
        if (data[i] > threshold) {
```

```
            (*anomalyCount)++;
            printf("Anomaly detected: data[%d] = %.2f\n", i, data[i]);
        }
    }
}
```

O/p: Anomaly detected: data[3] = 50.30

Anomaly detected: data[5] = 70.60

Number of anomalies detected: 2

5. Data Classifier

o Function Prototype: void classifyData(const double *data, int size, char *labels[], double threshold)

o Data Types: const double*, int, char*[], double

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

o Details: Classify data points into categories based on a threshold, updating an array of labels.

Sol:

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

// Function prototype
void classifyData(const double *data, int size, char *labels[], double threshold);

int main() {
    double data[] = {10.0, 30.5, 15.2, 50.1, 25.0};
    int size = sizeof(data) / sizeof(data[0]);
    double threshold = 20.0;
    char *labels[size];

    // Allocate memory for labels and initialize them
    for (int i = 0; i < size; i++) {
        labels[i] = (char *)malloc(20 * sizeof(char));
        if (labels[i] != NULL) {
            strcpy(labels[i], "");
```

```
        }
    }

    // Classify data
    classifyData(data, size, labels, threshold);

    // Display the classification results
    printf("Data Classification:\n");
    for (int i = 0; i < size; i++) {
        printf("data[%d] = %.2f -> %s\n", i, data[i], labels[i]);
        free(labels[i]);  // Free allocated memory
    }

    return 0;
}

// Function to classify data based on a threshold
void classifyData(const double *data, int size, char *labels[], double threshold) {
    for (int i = 0; i < size; i++) {
        if (data[i] > threshold) {
            strcpy(labels[i], "High");
        } else {
            strcpy(labels[i], "Low");
        }
    }
}
```

O/p: Data Classification:

data[0] = 10.00 -> Low

data[1] = 30.50 -> High

data[2] = 15.20 -> Low

data[3] = 50.10 -> High

data[4] = 25.00 -> High

Artificial Intelligence

6. Neural Network Weight Adjuster

o Function Prototype: void adjustWeights(double *weights, int size, double learningRate)

o Data Types: double*, int, double

o Concepts: Pointers, arrays, functions, loops.

o Details: Adjust neural network weights using a given learning rate, with weights passed by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void adjustWeights(double *weights, int size, double learningRate);

int main() {
    double weights[] = {0.1, 0.5, -0.3, 0.8, -0.6};
    int size = sizeof(weights) / sizeof(weights[0]);
    double learningRate = 0.05;

    printf("Original Weights:\n");
    for (int i = 0; i < size; i++) {
        printf("weights[%d] = %.2f\n", i, weights[i]);
    }

    // Adjust weights
    adjustWeights(weights, size, learningRate);

    printf("\nAdjusted Weights:\n");
    for (int i = 0; i < size; i++) {
        printf("weights[%d] = %.2f\n", i, weights[i]);
    }
```

```
    return 0;
}

// Function to adjust neural network weights
void adjustWeights(double *weights, int size, double learningRate) {
    for (int i = 0; i < size; i++) {
        weights[i] += learningRate * weights[i];  // Simple weight adjustment example: w = w + learningRate * w
    }
}
```

O/p: Original Weights:

weights[0] = 0.10

weights[1] = 0.50

weights[2] = -0.30

weights[3] = 0.80

weights[4] = -0.60

Adjusted Weights:

weights[0] = 0.11

weights[1] = 0.53

weights[2] = -0.32

weights[3] = 0.84

weights[4] = -0.63

7. AI Model Evaluator

o Function Prototype: void evaluateModels(const double *accuracies, int size, double *bestAccuracy)

o Data Types: const double*, int, double*

o Concepts: Loops, arrays, functions, pointers.

o Details: Evaluate multiple AI models, determining the best accuracy and updating it by reference.

Sol: #include <stdio.h>

```
// Function prototype
void evaluateModels(const double *accuracies, int size, double *bestAccuracy);

int main() {
    double accuracies[] = {85.5, 90.2, 78.9, 92.5, 88.0};
    int size = sizeof(accuracies) / sizeof(accuracies[0]);
    double bestAccuracy = 0.0;

    // Evaluate models
    evaluateModels(accuracies, size, &bestAccuracy);

    // Display the best accuracy
    printf("Best model accuracy: %.2f%%\n", bestAccuracy);

    return 0;
}

// Function to evaluate models and find the best accuracy
void evaluateModels(const double *accuracies, int size, double *bestAccuracy) {
    *bestAccuracy = accuracies[0];

    for (int i = 1; i < size; i++) {
        if (accuracies[i] > *bestAccuracy) {
            *bestAccuracy = accuracies[i];
        }
    }
}
```

O/p:

Best model accuracy: 92.50%

8. Decision Tree Constructor

o Function Prototype: void constructDecisionTree(const double *features, int size, int *treeStructure)

o Data Types: const double*, int, int*

o Concepts: Decision-making, arrays, functions.

o Details: Construct a decision tree based on feature data, updating the tree structure by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void constructDecisionTree(const double *features, int size, int *treeStructure);

int main() {
    double features[] = {2.5, 7.0, 4.8, 9.1, 3.6};
    int size = sizeof(features) / sizeof(features[0]);
    int treeStructure[size];

    // Construct the decision tree
    constructDecisionTree(features, size, treeStructure);

    // Display the tree structure
    printf("Decision Tree Structure (Simplified):\n");
    for (int i = 0; i < size; i++) {
        printf("Node %d: Feature %.2f -> Decision %d\n", i, features[i], treeStructure[i]);
    }

    return 0;
}

// Function to construct a simple decision tree based on features
void constructDecisionTree(const double *features, int size, int *treeStructure) {
    for (int i = 0; i < size; i++) {
        if (features[i] > 5.0) {
```

```
            treeStructure[i] = 1;  // Example decision: 1 for features > 5.0
        } else {
            treeStructure[i] = 0;  // 0 for features <= 5.0
        }
    }
}
```

O/p: Decision Tree Structure (Simplified):

Node 0: Feature 2.50 -> Decision 0

Node 1: Feature 7.00 -> Decision 1

Node 2: Feature 4.80 -> Decision 0

Node 3: Feature 9.10 -> Decision 1

Node 4: Feature 3.60 -> Decision 0

9. Sentiment Analysis Processor

o Function Prototype: void processSentiments(const char *sentences[], int size, int *sentimentScores)

o Data Types: const char*[], int, int*

o Concepts: Strings, arrays, functions, pointers.

o Details: Analyze sentiments of sentences, updating sentiment scores by reference.

Sol:

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

// Function prototype
void processSentiments(const char *sentences[], int size, int *sentimentScores);

int main() {
    const char *sentences[] = {
        "The product is excellent and works perfectly.",
        "I am very disappointed with the service.",
        "It was an average experience, nothing special.",
        "Absolutely fantastic, I love it!",
        "Not good, I wouldn't recommend it."
```

```
    };
    int size = sizeof(sentences) / sizeof(sentences[0]);
    int sentimentScores[size];

    // Process sentiments
    processSentiments(sentences, size, sentimentScores);

    // Display sentiment scores
    printf("Sentiment Analysis Results:\n");
    for (int i = 0; i < size; i++) {
        printf("Sentence %d: %s\nSentiment Score: %d\n\n", i + 1, sentences[i], sentimentScores[i]);
    }

    return 0;
}

// Function to process sentiments and assign scores
void processSentiments(const char *sentences[], int size, int *sentimentScores) {
    for (int i = 0; i < size; i++) {
        if (strstr(sentences[i], "excellent") || strstr(sentences[i], "fantastic") || strstr(sentences[i],
"love")) {
            sentimentScores[i] = 1;  // Positive sentiment
        } else if (strstr(sentences[i], "disappointed") || strstr(sentences[i], "not good") ||
strstr(sentences[i], "recommend")) {
            sentimentScores[i] = -1; // Negative sentiment
        } else {
            sentimentScores[i] = 0;  // Neutral sentiment
        }
    }
}
```

O/p: Sentiment Analysis Results:

Sentence 1: The product is excellent and works perfectly.

Sentiment Score: 1

Sentence 2: I am very disappointed with the service.

Sentiment Score: -1

Sentence 3: It was an average experience, nothing special.

Sentiment Score: 0

Sentence 4: Absolutely fantastic, I love it!

Sentiment Score: 1

Sentence 5: Not good, I wouldn't recommend it.

Sentiment Score: -1

10. Training Data Generator

o Function Prototype: double* generateTrainingData(const double *baseData, int size, int multiplier)

o Data Types: const double*, int, double*

o Concepts: Arrays, functions returning pointers, loops.

o Details: Generate training data by applying a multiplier to base data, returning a pointer to the new data array.

Sol: #include <stdio.h>

#include <stdlib.h>

// Function prototype

double* generateTrainingData(const double *baseData, int size, int multiplier);

int main() {

  double baseData[] = {1.5, 2.0, 3.5, 4.0, 5.5};

  int size = sizeof(baseData) / sizeof(baseData[0]);

  int multiplier = 2;

```
    // Generate training data
    double *trainingData = generateTrainingData(baseData, size, multiplier);

    // Display the generated training data
    if (trainingData != NULL) {
        printf("Generated Training Data:\n");
        for (int i = 0; i < size; i++) {
            printf("%.2f ", trainingData[i]);
        }
        printf("\n");

        // Free the allocated memory
        free(trainingData);
    } else {
        printf("Failed to generate training data.\n");
    }

    return 0;
}

// Function to generate training data by applying a multiplier
double* generateTrainingData(const double *baseData, int size, int multiplier) {
    if (size <= 0 || multiplier <= 0) return NULL;

    double newData = (double)malloc(size * sizeof(double));
    if (newData == NULL) return NULL; // Memory allocation check

    for (int i = 0; i < size; i++) {
        newData[i] = baseData[i] * multiplier;
    }
```

```
    return newData;
}
```

O/p: Generated Training Data:

3.00 4.00 7.00 8.00 11.00

Computer Vision

11. Image Filter Application

o Function Prototype: void applyFilter(const unsigned char *image, unsigned char *filteredImage, int width, int height)

o Data Types: const unsigned char*, unsigned char*, int

o Concepts: Arrays, pointers, functions.

o Details: Apply a filter to an image, modifying the filtered image by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void applyFilter(const unsigned char *image, unsigned char *filteredImage, int width, int height);

int main() {
    // Example image of 3x3 pixels (in grayscale)
    unsigned char image[9] = {255, 100, 50, 200, 150, 75, 0, 50, 200};  // 3x3 grayscale image
    unsigned char filteredImage[9];  // Array to store the filtered image

    int width = 3, height = 3;  // Image dimensions

    // Apply filter to the image
    applyFilter(image, filteredImage, width, height);

    // Print the filtered image
    for(int i = 0; i < width * height; i++) {
        printf("%d ", filteredImage[i]);
        if ((i + 1) % width == 0) {
            printf("\n");
```

```
        }
    }

    return 0;
}

// Function to apply filter (invert the image in this case)
void applyFilter(const unsigned char *image, unsigned char *filteredImage, int width, int height) {
    for (int i = 0; i < width * height; i++) {
        // Inversion filter: new value = 255 - original value
        filteredImage[i] = 255 - image[i];
    }
}
```

O/p: 0 155 205

55 105 180

255 205 55

12. Edge Detection Algorithm

- Function Prototype: void detectEdges(const unsigned char *image, unsigned char *edges, int width, int height)
- Data Types: const unsigned char*, unsigned char*, int
- Concepts: Loops, arrays, decision-making, functions.
- Details: Detect edges in an image, updating the edges array by reference.

Sol:

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

// Function prototype
void detectEdges(const unsigned char *image, unsigned char *edges, int width, int height);

int main() {
    // Example 5x5 grayscale image
```

```
    unsigned char image[25] = {
        10, 10, 10, 10, 10,
        10, 50, 50, 50, 10,
        10, 50, 100, 50, 10,
        10, 50, 50, 50, 10,
        10, 10, 10, 10, 10
    };

    unsigned char edges[25];  // Array to store edge-detected image

    int width = 5, height = 5;  // Image dimensions

    // Apply edge detection to the image
    detectEdges(image, edges, width, height);

    // Print the edge-detected image
    for(int i = 0; i < width * height; i++) {
        printf("%d ", edges[i]);
        if ((i + 1) % width == 0) {
            printf("\n");
        }
    }

    return 0;
}

// Function to detect edges using Sobel operator
void detectEdges(const unsigned char *image, unsigned char *edges, int width, int height) {
    // Sobel operators for edge detection (Gx and Gy)
    int Gx[3][3] = {
        {-1, 0, 1},
```

```
    {-2, 0, 2},
    {-1, 0, 1}
};

int Gy[3][3] = {
    {-1, -2, -1},
    { 0,  0,  0},
    { 1,  2,  1}
};

// Apply Sobel edge detection filter
for (int y = 1; y < height - 1; y++) {
    for (int x = 1; x < width - 1; x++) {
        int gradX = 0;
        int gradY = 0;

        // Apply Gx and Gy to the 3x3 neighborhood of each pixel
        for (int ky = -1; ky <= 1; ky++) {
            for (int kx = -1; kx <= 1; kx++) {
                int pixel = image[(y + ky) * width + (x + kx)];
                gradX += pixel * Gx[ky + 1][kx + 1];
                gradY += pixel * Gy[ky + 1][kx + 1];
            }
        }

        // Calculate the magnitude of the gradient
        int magnitude = (int)sqrt(gradX * gradX + gradY * gradY);
        // Threshold to detect edges (you can adjust the threshold value)
        edges[y * width + x] = (magnitude > 255) ? 255 : magnitude;
    }
}
```

}

O/p: 0 0 0 0 0

0 240 255 240 0

0 255 0 255 0

0 240 255 240 0

0 0 0 0 0

13. Object Recognition System

o Function Prototype: void recognizeObjects(const double *features, int size, char *objectLabels[])

o Data Types: const double*, int, char*[]

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

o Details: Recognize objects based on feature vectors, updating an array of object labels.

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

// Function prototype
void recognizeObjects(const double *features, int size, char *objectLabels[]);

int main() {
    // Example feature vectors for objects
    // Each object has 3 features: [size, color, shape]
    double features1[] = {3.0, 2.5, 1.0};  // Example: object 1 features
    double features2[] = {1.0, 1.0, 0.5};  // Example: object 2 features
    double features3[] = {5.0, 3.5, 4.0};  // Example: object 3 features

    // Array of feature vectors
    double *featuresArray[] = {features1, features2, features3};

    // Corresponding object labels (for comparison)
    char *objectLabels[] = {"Object 1", "Object 2", "Object 3"};
```

```
    // Call the recognizeObjects function for each object
    for (int i = 0; i < 3; i++) {
        printf("Recognizing object %d based on features: [%.2f, %.2f, %.2f] -> ",
            i + 1, featuresArray[i][0], featuresArray[i][1], featuresArray[i][2]);
        recognizeObjects(featuresArray[i], 3, objectLabels);
    }

    return 0;
}

// Function to recognize objects based on feature vectors
void recognizeObjects(const double *features, int size, char *objectLabels[]) {
    // Predefined feature vectors for each object (for simplicity)
    double object1[] = {3.0, 2.5, 1.0};  // Object 1
    double object2[] = {1.0, 1.0, 0.5};  // Object 2
    double object3[] = {5.0, 3.5, 4.0};  // Object 3

    // Calculate the Euclidean distance between the given feature vector and each object
    double minDistance = INFINITY;
    int closestObjectIndex = -1;

    double *predefinedObjects[] = {object1, object2, object3};

    for (int i = 0; i < 3; i++) {
        double distance = 0.0;

        // Calculate the Euclidean distance between the feature vector and the predefined object
        for (int j = 0; j < size; j++) {
            distance += (features[j] - predefinedObjects[i][j]) * (features[j] - predefinedObjects[i][j]);
        }
```

```
        distance = sqrt(distance);

        // If the current object is closer, update the closest object
        if (distance < minDistance) {
            minDistance = distance;
            closestObjectIndex = i;
        }
    }

    // Output the recognized object label based on the closest match
    printf("Recognized as: %s\n", objectLabels[closestObjectIndex]);
}
```

O/p: Recognizing object 1 based on features: [3.00, 2.50, 1.00] -> Recognized as: Object 1

Recognizing object 2 based on features: [1.00, 1.00, 0.50] -> Recognized as: Object 2

Recognizing object 3 based on features: [5.00, 3.50, 4.00] -> Recognized as: Object 3

14. Image Resizing Function

- Function Prototype: void resizeImage(const unsigned char *inputImage, unsigned char *outputImage, int originalWidth, int originalHeight, int newWidth, int newHeight)
- Data Types: const unsigned char*, unsigned char*, int
- Concepts: Arrays, functions, pointers.
- Details: Resize an image to new dimensions, modifying the output image by reference.

Sol:

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

// Function prototype
void resizeImage(const unsigned char *inputImage, unsigned char *outputImage, int originalWidth, int originalHeight, int newWidth, int newHeight);

int main() {
    // Example 3x3 image (grayscale values for simplicity)
    unsigned char inputImage[9] = {255, 100, 50, 200, 150, 75, 0, 50, 200}; // 3x3 grayscale image
    unsigned char outputImage[16];  // Output image for resizing (for 4x4 image)
```

```
    int originalWidth = 3, originalHeight = 3;
    int newWidth = 4, newHeight = 4;  // Resize to 4x4 image

    // Resize the image
    resizeImage(inputImage, outputImage, originalWidth, originalHeight, newWidth, newHeight);

    // Print the resized image
    for (int i = 0; i < newHeight; i++) {
        for (int j = 0; j < newWidth; j++) {
            printf("%d ", outputImage[i * newWidth + j]);
        }
        printf("\n");
    }

    return 0;
}

// Function to resize the image using nearest-neighbor interpolation
void resizeImage(const unsigned char *inputImage, unsigned char *outputImage, int originalWidth,
int originalHeight, int newWidth, int newHeight) {
    // Scale factors for width and height
    float xScale = (float)originalWidth / newWidth;
    float yScale = (float)originalHeight / newHeight;

    // Iterate over each pixel in the new image and find the corresponding pixel in the original image
    for (int y = 0; y < newHeight; y++) {
        for (int x = 0; x < newWidth; x++) {
            // Calculate the position of the corresponding pixel in the original image
            int origX = (int)(x * xScale);
            int origY = (int)(y * yScale);
```

```
            // Get the pixel value from the original image
            outputImage[y * newWidth + x] = inputImage[origY * originalWidth + origX];
        }
    }
}
```

O/p: 255 255 100 50

255 255 100 50

200 200 150 75

0 0 50 200

15. Color Balance Adjuster

o Function Prototype: void balanceColors(const unsigned char *image, unsigned char *balancedImage, int width, int height)

o Data Types: const unsigned char*, unsigned char*, int

o Concepts: Arrays, functions, pointers, loops.

o Details: Adjust the color balance of an image, updating the balanced image by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void balanceColors(const unsigned char *image, unsigned char *balancedImage, int width, int height);

int main() {
    // Example 3x3 color image (RGB format: [R, G, B])
    // Each pixel is represented by 3 values: [R, G, B]
    unsigned char image[27] = {
        255, 100, 50,   200, 150, 75,   0, 50, 200,   // Row 1
        150, 200, 50,   100, 100, 100,  75, 75, 75,    // Row 2
        255, 255, 0,    50, 100, 150,  200, 50, 100     // Row 3
    };

    unsigned char balancedImage[27];  // Array to store the balanced image
```

```
    int width = 3, height = 3;  // Image dimensions

    // Apply color balance adjustment to the image
    balanceColors(image, balancedImage, width, height);

    // Print the balanced image
    for (int i = 0; i < width * height; i++) {
        printf("[%d, %d, %d] ", balancedImage[i * 3], balancedImage[i * 3 + 1], balancedImage[i * 3 + 2]);
        if ((i + 1) % width == 0) {
            printf("\n");
        }
    }

    return 0;
}

// Function to adjust the color balance of the image
void balanceColors(const unsigned char *image, unsigned char *balancedImage, int width, int
height) {
    // Color adjustment factors (modify these values for different color balances)
    float redFactor = 1.2f;    // Increase red channel by 20%
    float greenFactor = 1.0f;  // No change to green channel
    float blueFactor = 0.8f;   // Decrease blue channel by 20%

    // Iterate over all pixels in the image
    for (int i = 0; i < width * height; i++) {
        int r = image[i * 3];     // Red channel
        int g = image[i * 3 + 1]; // Green channel
        int b = image[i * 3 + 2]; // Blue channel
```

```
        // Adjust the color channels based on the factors
        r = (int)(r * redFactor);
        g = (int)(g * greenFactor);
        b = (int)(b * blueFactor);

        // Ensure that values are within the 0-255 range (clamp if necessary)
        if (r > 255) r = 255;
        if (r < 0) r = 0;
        if (g > 255) g = 255;
        if (g < 0) g = 0;
        if (b > 255) b = 255;
        if (b < 0) b = 0;

        // Store the adjusted values in the balanced image
        balancedImage[i * 3] = (unsigned char)r;
        balancedImage[i * 3 + 1] = (unsigned char)g;
        balancedImage[i * 3 + 2] = (unsigned char)b;
    }
}
```

O/p: [255, 100, 40] [240, 150, 60] [0, 50, 160]

[180, 200, 40] [120, 100, 80] [90, 75, 60]

[255, 255, 0] [60, 100, 120] [240, 50, 80]

16. Pattern Recognition Algorithm

- Function Prototype: void recognizePatterns(const char *patterns[], int size, int *matchCounts)
- Data Types: const char*[], int, int*
- Concepts: Strings, arrays, decision-making, pointers.
- Details: Recognize patterns in a dataset, updating match counts by reference.

Sol:

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

```
// Function prototype
void recognizePatterns(const char *patterns[], int size, int *matchCounts);

int main() {
    // Example dataset (array of strings)
    const char *dataset[] = {
        "apple", "banana", "cherry", "apple", "date", "banana", "apple"
    };
    int datasetSize = 7; // Size of the dataset

    // Example patterns to match
    const char *patterns[] = {
        "apple", "banana", "cherry"
    };
    int patternSize = 3; // Number of patterns to match

    // Array to store the match counts for each pattern
    int matchCounts[patternSize];

    // Call the recognizePatterns function to count matches
    recognizePatterns(patterns, patternSize, matchCounts);

    // Print the match counts for each pattern
    for (int i = 0; i < patternSize; i++) {
        printf("Pattern: '%s' matched %d times\n", patterns[i], matchCounts[i]);
    }

    return 0;
}

// Function to recognize patterns and update match counts
```

```
void recognizePatterns(const char *patterns[], int size, int *matchCounts) {
   // Example dataset (array of strings)
   const char *dataset[] = {
      "apple", "banana", "cherry", "apple", "date", "banana", "apple"
   };
   int datasetSize = 7; // Size of the dataset

   // Initialize the match counts to 0
   for (int i = 0; i < size; i++) {
      matchCounts[i] = 0;
   }

   // For each pattern, count how many times it appears in the dataset
   for (int i = 0; i < size; i++) {
      for (int j = 0; j < datasetSize; j++) {
         if (strcmp(patterns[i], dataset[j]) == 0) {
            matchCounts[i]++;
         }
      }
   }
}
```

O/p: Pattern: 'apple' matched 3 times

Pattern: 'banana' matched 2 times

Pattern: 'cherry' matched 1 times

17. Climate Data Analyzer

o Function Prototype: void analyzeClimateData(const double *temperatureReadings, int size, double *minTemp, double *maxTemp)

o Data Types: const double*, int, double*

o Concepts: Decision-making, arrays, functions.

o Details: Analyze climate data to find minimum and maximum temperatures, updating these values by reference.

Sol: #include <stdio.h>

```
// Function prototype
void analyzeClimateData(const double *temperatureReadings, int size, double *minTemp, double *maxTemp);

int main() {
    // Example temperature readings (in Celsius)
    double temperatureReadings[] = {15.2, 22.5, 19.8, 30.0, 10.5, 25.3, 18.4};
    int size = 7;  // Number of temperature readings

    double minTemp, maxTemp;  // Variables to store the min and max temperatures

    // Call the function to analyze the climate data
    analyzeClimateData(temperatureReadings, size, &minTemp, &maxTemp);

    // Print the results
    printf("Minimum Temperature: %.2f°C\n", minTemp);
    printf("Maximum Temperature: %.2f°C\n", maxTemp);

    return 0;
}

// Function to analyze the climate data and find the min and max temperatures
void analyzeClimateData(const double *temperatureReadings, int size, double *minTemp, double *maxTemp) {
    // Initialize min and max temperatures to the first reading
    *minTemp = temperatureReadings[0];
    *maxTemp = temperatureReadings[0];

    // Loop through the temperature readings and find the min and max values
    for (int i = 1; i < size; i++) {
        if (temperatureReadings[i] < *minTemp) {
```

```
            *minTemp = temperatureReadings[i];  // Update min temperature
        }
        if (temperatureReadings[i] > *maxTemp) {
            *maxTemp = temperatureReadings[i];  // Update max temperature
        }
    }
}
```

O/p: Minimum Temperature: 10.50°C

Maximum Temperature: 30.00°C

18. Quantum Data Processor

o Function Prototype: void processQuantumData(const double *measurements, int size, double *processedData)

o Data Types: const double*, int, double*

o Concepts: Arrays, functions, pointers, loops.

o Details: Process quantum measurement data, updating the processed data array by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void processQuantumData(const double *measurements, int size, double *processedData);

int main() {
    // Example quantum measurements (could represent some observed quantum values)
    double measurements[] = {1.5, 2.0, 3.1, 2.5, 1.8, 3.3, 2.9};
    int size = 7;  // Number of measurements

    double processedData[size];  // Array to store the processed data

    // Call the function to process the quantum data
    processQuantumData(measurements, size, processedData);

    // Print the processed data
```

```
    printf("Processed Quantum Data (Normalized):\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f ", processedData[i]);
    }
    printf("\n");

    return 0;
}

// Function to process the quantum data (normalization in this case)
void processQuantumData(const double *measurements, int size, double *processedData) {
    // Find the minimum and maximum values in the measurements
    double minVal = measurements[0];
    double maxVal = measurements[0];

    for (int i = 1; i < size; i++) {
        if (measurements[i] < minVal) {
            minVal = measurements[i];  // Update min value
        }
        if (measurements[i] > maxVal) {
            maxVal = measurements[i];  // Update max value
        }
    }

    // Normalize the measurements to the range [0, 1]
    for (int i = 0; i < size; i++) {
        processedData[i] = (measurements[i] - minVal) / (maxVal - minVal);
    }
}
```

O/p: Processed Quantum Data (Normalized):

0.00 0.28 0.89 0.56 0.17 1.00 0.78

19. Scientific Data Visualization

- Function Prototype: void visualizeData(const double *data, int size, const char *title)
- Data Types: const double*, int, const char*
- Concepts: Arrays, functions, strings.
- Details: Visualize scientific data with a given title, using constant data for the title.

Sol:

```
#include <stdio.h>

// Function prototype
void visualizeData(const double *data, int size, const char *title);

int main() {
    // Example scientific data (e.g., measurements, values, etc.)
    double data[] = {12.5, 20.8, 15.3, 30.1, 18.7};
    int size = 5;  // Number of data points

    // Title for the visualization
    const char *title = "Scientific Data Visualization";

    // Call the function to visualize the data
    visualizeData(data, size, title);

    return 0;
}

// Function to visualize scientific data (text-based bar chart)
void visualizeData(const double *data, int size, const char *title) {
    // Print the title of the visualization
    printf("%s\n\n", title);

    // Find the maximum value in the data to normalize the bars
    double maxDataValue = data[0];
```

```
    for (int i = 1; i < size; i++) {
        if (data[i] > maxDataValue) {
            maxDataValue = data[i];
        }
    }

    // Scale factor to control the maximum bar length (max length of 50)
    double scaleFactor = 50.0 / maxDataValue;

    // Print each data point as a bar chart
    for (int i = 0; i < size; i++) {
        int barLength = (int)(data[i] * scaleFactor); // Calculate the length of the bar

        // Print the data label (value) and a bar of appropriate length
        printf("%.2f | ", data[i]);
        for (int j = 0; j < barLength; j++) {
            printf(".");
        }
        printf("\n");
    }
}
```

O/p: Scientific Data Visualization

```
12.50 | ..................
20.80 | ..............................
15.30 | ......................
30.10 | ..........................................
18.70 | ...........................
```

20. Genetic Data Simulator

o Function Prototype: double* simulateGeneticData(const double *initialData, int size, double mutationRate)

o Data Types: const double*, int, double

o Concepts: Arrays, functions returning pointers, loops.

o Details: Simulate genetic data evolution by applying a mutation rate, returning a pointer to the simulated data.

Sol:

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

// Function prototype
double* simulateGeneticData(const double *initialData, int size, double mutationRate);

int main() {
    // Example initial genetic data (could represent genetic markers or other values)
    double initialData[] = {0.5, 0.8, 1.0, 0.7, 0.3};
    int size = 5;  // Size of the genetic data array
    double mutationRate = 0.1;  // 10% mutation rate

    // Simulate the genetic data evolution
    double* simulatedData = simulateGeneticData(initialData, size, mutationRate);

    // Print the simulated data
    printf("Simulated Genetic Data:\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f ", simulatedData[i]);
    }
    printf("\n");

    // Free the allocated memory for simulated data
    free(simulatedData);

    return 0;
```

```
}

// Function to simulate genetic data evolution
double* simulateGeneticData(const double *initialData, int size, double mutationRate) {
    // Dynamically allocate memory for the simulated data
    double* simulatedData = (double*)malloc(size * sizeof(double));

    if (simulatedData == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);  // Exit if memory allocation fails
    }

    // Seed the random number generator
    srand(time(NULL));

    // Simulate the genetic data evolution
    for (int i = 0; i < size; i++) {
        simulatedData[i] = initialData[i];

        // Apply mutation with a certain probability (based on mutationRate)
        if ((rand() / (double)RAND_MAX) < mutationRate) {
            // Apply a random mutation (e.g., adding a small random change)
            simulatedData[i] += (rand() / (double)RAND_MAX) * 0.2 - 0.1;  // Small mutation
        }

        // Ensure the simulated data stays within the range [0, 1]
        if (simulatedData[i] < 0) simulatedData[i] = 0;
        if (simulatedData[i] > 1) simulatedData[i] = 1;
    }

    return simulatedData;  // Return the pointer to the simulated data
```

}

O/p: Simulated Genetic Data:

0.50 0.80 1.00 0.70 0.30

21. AI Performance Tracker

o Function Prototype: void trackPerformance(const double *performanceData, int size, double *maxPerformance, double *minPerformance)

o Data Types: const double*, int, double*

o Concepts: Arrays, functions, pointers.

o Details: Track AI performance data, updating maximum and minimum performance by reference.

Sol: #include <stdio.h>

```
// Function prototype
void trackPerformance(const double *performanceData, int size, double *maxPerformance, double *minPerformance);

int main() {
    // Example AI performance data (e.g., accuracy, F1 score, etc.)
    double performanceData[] = {0.85, 0.92, 0.78, 0.95, 0.89, 0.80};
    int size = 6;  // Number of performance data points

    double maxPerformance, minPerformance;  // Variables to store the max and min performance

    // Call the function to track performance
    trackPerformance(performanceData, size, &maxPerformance, &minPerformance);

    // Print the tracked performance
    printf("Maximum Performance: %.2f\n", maxPerformance);
    printf("Minimum Performance: %.2f\n", minPerformance);

    return 0;
}
```

```
// Function to track the AI performance data
void trackPerformance(const double *performanceData, int size, double *maxPerformance, double *minPerformance) {
    // Initialize max and min performance to the first data point
    *maxPerformance = performanceData[0];
    *minPerformance = performanceData[0];

    // Loop through the performance data and update max and min values
    for (int i = 1; i < size; i++) {
        if (performanceData[i] > *maxPerformance) {
            *maxPerformance = performanceData[i];  // Update max performance
        }
        if (performanceData[i] < *minPerformance) {
            *minPerformance = performanceData[i];  // Update min performance
        }
    }
}
```

O/p: Maximum Performance: 0.95

Minimum Performance: 0.78

22. Sensor Data Filter

o Function Prototype: void filterSensorData(const double *sensorData, double *filteredData, int size, double filterThreshold)

o Data Types: const double*, double*, int, double

o Concepts: Arrays, functions, decision-making.

o Details: Filter sensor data based on a threshold, updating the filtered data array by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void filterSensorData(const double *sensorData, double *filteredData, int size, double filterThreshold);
```

```
int main() {
    // Example sensor data (could represent sensor readings like temperature, pressure, etc.)
    double sensorData[] = {1.5, 3.2, 0.8, 5.0, 2.3, 1.0};
    int size = 6;  // Number of sensor data points
    double filterThreshold = 2.0;  // Filter threshold value

    double filteredData[size];  // Array to store filtered data

    // Call the function to filter the sensor data
    filterSensorData(sensorData, filteredData, size, filterThreshold);

    // Print the filtered data
    printf("Filtered Sensor Data:\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f ", filteredData[i]);
    }
    printf("\n");

    return 0;
}

// Function to filter sensor data based on the given threshold
void filterSensorData(const double *sensorData, double *filteredData, int size, double
filterThreshold) {
    // Loop through the sensor data and apply the filter
    for (int i = 0; i < size; i++) {
        if (sensorData[i] > filterThreshold) {
            filteredData[i] = sensorData[i];  // Keep values above the threshold
        } else {
            filteredData[i] = 0.0;  // Set values below the threshold to 0 (or another value)
        }
```

```
    }
}
```

O/p: Filtered Sensor Data:

0.00 3.20 0.00 5.00 2.30 0.00

23. Logistics Data Planner

o Function Prototype: void planLogistics(const double *resourceLevels, double *logisticsPlan, int size)

o Data Types: const double*, double*, int

o Concepts: Arrays, functions, pointers, loops.

o Details: Plan logistics based on resource levels, updating the logistics plan array by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void planLogistics(const double *resourceLevels, double *logisticsPlan, int size);

int main() {
    // Example resource levels (e.g., available resources at different locations)
    double resourceLevels[] = {50.0, 20.5, 100.0, 30.2, 75.3};
    int size = 5;  // Number of resource levels

    double logisticsPlan[size];  // Array to store the logistics plan

    // Call the function to plan the logistics based on resource levels
    planLogistics(resourceLevels, logisticsPlan, size);

    // Print the logistics plan
    printf("Logistics Plan:\n");
    for (int i = 0; i < size; i++) {
        printf("Location %d: %.2f\n", i + 1, logisticsPlan[i]);
    }
```

```
    return 0;
}

// Function to plan logistics based on resource levels
void planLogistics(const double *resourceLevels, double *logisticsPlan, int size) {
    // Determine the maximum resource level to scale the logistics plan
    double maxResourceLevel = resourceLevels[0];
    for (int i = 1; i < size; i++) {
        if (resourceLevels[i] > maxResourceLevel) {
            maxResourceLevel = resourceLevels[i];
        }
    }

    // Plan logistics by allocating resources proportional to their levels
    for (int i = 0; i < size; i++) {
        logisticsPlan[i] = (resourceLevels[i] / maxResourceLevel) * 100.0;  // Example: scale to a range of 0-100
    }
}
```

O/p:

```
Logistics Plan:
Location 1: 50.00
Location 2: 20.50
Location 3: 100.00
Location 4: 30.20
Location 5: 75.30
```

24. Satellite Image Processor

- Function Prototype: void processSatelliteImage(const unsigned char *imageData, unsigned char *processedImage, int width, int height)
- Data Types: const unsigned char*, unsigned char*, int
- Concepts: Arrays, functions, pointers, loops.
- Details: Process satellite image data, updating the processed image by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void processSatelliteImage(const unsigned char *imageData, unsigned char *processedImage, int width, int height);

int main() {
    // Example image data (RGB format: 3 channels per pixel)
    int width = 3, height = 2;  // Example image size (3x2 pixels)

    // Sample image with RGB values (each pixel has 3 values: R, G, B)
    unsigned char imageData[3 * 2 * 3] = {
        255, 0, 0,   // Red
        0, 255, 0,   // Green
        0, 0, 255,   // Blue
        255, 255, 0, // Yellow
        0, 255, 255, // Cyan
        255, 0, 255  // Magenta
    };

    unsigned char processedImage[3 * 2];  // Processed grayscale image (1 channel per pixel)

    // Call the function to process the satellite image (convert to grayscale)
    processSatelliteImage(imageData, processedImage, width, height);

    // Print the processed grayscale image data
    printf("Processed Grayscale Image Data:\n");
    for (int i = 0; i < width * height; i++) {
        printf("%d ", processedImage[i]);
    }
    printf("\n");
```

```
    return 0;
}

// Function to process satellite image and convert it to grayscale
void processSatelliteImage(const unsigned char *imageData, unsigned char *processedImage, int width, int height) {
    // Loop through each pixel and apply the grayscale conversion
    for (int i = 0; i < width * height; i++) {
        // Calculate the index for RGB values
        int rIndex = i * 3;        // Red channel index
        int gIndex = rIndex + 1;   // Green channel index
        int bIndex = rIndex + 2;   // Blue channel index

        // Convert RGB to grayscale using the luminosity method
        unsigned char r = imageData[rIndex];
        unsigned char g = imageData[gIndex];
        unsigned char b = imageData[bIndex];

        // Luminosity formula: Y = 0.299 * R + 0.587 * G + 0.114 * B
        unsigned char gray = (unsigned char)(0.299 * r + 0.587 * g + 0.114 * b);

        // Store the grayscale value in the processed image
        processedImage[i] = gray;
    }
}
```

O/p: Processed Grayscale Image Data:

76 149 29 225 178 105

25. Flight Path Analyzer

o Function Prototype: void analyzeFlightPath(const double *pathCoordinates, double *optimizedPath, int size)

o Data Types: const double*, double*, int

o Concepts: Arrays, functions, pointers, loops.

o Details: Analyze and optimize flight path coordinates, updating the optimized path by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void analyzeFlightPath(const double *pathCoordinates, double *optimizedPath, int size);

int main() {
    // Example path coordinates (each coordinate consists of x, y, z values)
    // In this example, we assume each coordinate has 3 values (x, y, z)
    int size = 6;  // Number of path coordinates (each consisting of 3 values)

    // Sample flight path coordinates (x, y, z)
    double pathCoordinates[] = {
        0.0, 0.0, 0.0,   // Point 1
        1.0, 1.0, 1.0,   // Point 2
        2.0, 2.0, 2.0,   // Point 3 (potentially redundant point)
        3.0, 3.0, 3.0,   // Point 4
        4.0, 4.0, 4.0,   // Point 5 (another potentially redundant point)
        5.0, 5.0, 5.0    // Point 6
    };

    double optimizedPath[size];  // Array to store the optimized path

    // Call the function to analyze and optimize the flight path
    analyzeFlightPath(pathCoordinates, optimizedPath, size);

    // Print the optimized flight path coordinates
    printf("Optimized Flight Path Coordinates:\n");
    for (int i = 0; i < size; i += 3) {
```

```
        printf("(%f, %f, %f)\n", optimizedPath[i], optimizedPath[i + 1], optimizedPath[i + 2]);
    }

    return 0;
}

// Function to analyze and optimize flight path coordinates
void analyzeFlightPath(const double *pathCoordinates, double *optimizedPath, int size) {
    // For simplicity, let's remove redundant consecutive points
    // We'll keep only every second point for demonstration purposes (optimization example)

    int j = 0;  // Index for the optimized path
    for (int i = 0; i < size; i += 3) {
        if (i == 0 || (pathCoordinates[i] != pathCoordinates[i - 3] || pathCoordinates[i + 1] !=
pathCoordinates[i - 2] || pathCoordinates[i + 2] != pathCoordinates[i - 1])) {
            // If it's the first point or the current point is different from the previous one
            optimizedPath[j++] = pathCoordinates[i];     // Copy x-coordinate
            optimizedPath[j++] = pathCoordinates[i + 1]; // Copy y-coordinate
            optimizedPath[j++] = pathCoordinates[i + 2]; // Copy z-coordinate
        }
    }
}
```

O/p: Optimized Flight Path Coordinates:

(0.000000, 0.000000, 0.000000)

(1.000000, 1.000000, 1.000000)

26. AI Data Augmenter

- Function Prototype: void augmentData(const double *originalData, double *augmentedData, int size, double augmentationFactor)
- Data Types: const double*, double*, int, double
- Concepts: Arrays, functions, pointers, loops.
- Details: Augment AI data by applying an augmentation factor, updating the augmented data array by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void augmentData(const double *originalData, double *augmentedData, int size, double augmentationFactor);

int main() {
    // Example original data (e.g., AI model input data)
    double originalData[] = {1.0, 2.5, 3.7, 4.2, 5.6};
    int size = 5;  // Number of data points
    double augmentationFactor = 1.5;  // Augmentation factor (e.g., scaling factor)

    double augmentedData[size];  // Array to store the augmented data

    // Call the function to augment the data
    augmentData(originalData, augmentedData, size, augmentationFactor);

    // Print the augmented data
    printf("Augmented Data:\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f ", augmentedData[i]);
    }
    printf("\n");

    return 0;
}

// Function to augment AI data by applying the augmentation factor
void augmentData(const double *originalData, double *augmentedData, int size, double augmentationFactor) {
    // Loop through the original data and apply the augmentation factor
    for (int i = 0; i < size; i++) {
```

```
    augmentedData[i] = originalData[i] * augmentationFactor;  // Multiply each element by the factor
  }
}
```

O/p: Augmented Data:

1.50 3.75 5.55 6.30 8.40

27. Medical Image Analyzer

o Function Prototype: void analyzeMedicalImage(const unsigned char *imageData, unsigned char *analysisResults, int width, int height)

o Data Types: const unsigned char*, unsigned char*, int

o Concepts: Arrays, functions, pointers, loops.

o Details: Analyze medical image data, updating analysis results by reference.

Sol:

```
#include <stdio.h>

// Function prototype
void analyzeMedicalImage(const unsigned char *imageData, unsigned char *analysisResults, int width, int height);

int main() {
  // Example medical image data (grayscale image)
  // The image is assumed to be grayscale, so each pixel is a single byte (0 to 255)
  int width = 3, height = 2;  // Example image size (3x2 pixels)

  // Sample grayscale image data (each pixel value is between 0 and 255)
  unsigned char imageData[] = {
    100, 150, 200,   // Row 1: Pixels (100, 150, 200)
    50, 75, 125      // Row 2: Pixels (50, 75, 125)
  };

  unsigned char analysisResults[1];  // Array to store analysis results (e.g., average brightness)

  // Call the function to analyze the medical image
```

```
    analyzeMedicalImage(imageData, analysisResults, width, height);

    // Print the analysis result (average brightness)
    printf("Average Brightness of Image: %d\n", analysisResults[0]);

    return 0;
}

// Function to analyze medical image and calculate average brightness
void analyzeMedicalImage(const unsigned char *imageData, unsigned char *analysisResults, int width, int height) {
    int totalBrightness = 0;
    int totalPixels = width * height;

    // Loop through each pixel to calculate total brightness
    for (int i = 0; i < totalPixels; i++) {
        totalBrightness += imageData[i];  // Sum all pixel values
    }

    // Calculate the average brightness and store it in analysisResults
    unsigned char averageBrightness = totalBrightness / totalPixels;
    analysisResults[0] = averageBrightness;  // Store the result
}
```

O/p: Average Brightness of Image: 116

28. Object Tracking System

o Function Prototype: void trackObjects(const double *objectData, double *trackingResults, int size)

o Data Types: const double*, double*, int

o Concepts: Arrays, functions, pointers, loops.

o Details: Track objects based on data, updating tracking results by reference.

Sol:

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

```
// Function prototype
void trackObjects(const double *objectData, double *trackingResults, int size);

int main() {
    // Example object data (each object has x and y coordinates)
    // Assume we have 3 objects, and each has an (x, y) position
    int size = 6;  // 3 objects, each with x and y (so 6 values)

    // Sample object data (x, y positions for 3 objects)
    double objectData[] = {
        1.0, 2.0,   // Object 1: Position (1.0, 2.0)
        3.0, 4.0,   // Object 2: Position (3.0, 4.0)
        5.0, 6.0    // Object 3: Position (5.0, 6.0)
    };

    double trackingResults[size];  // Array to store tracking results (updated positions)

    // Call the function to track the objects
    trackObjects(objectData, trackingResults, size);

    // Print the tracking results (updated positions)
    printf("Tracked Object Positions:\n");
    for (int i = 0; i < size; i += 2) {
        printf("Object %d: (%f, %f)\n", (i / 2) + 1, trackingResults[i], trackingResults[i + 1]);
    }

    return 0;
}

// Function to track objects based on data (e.g., updating their positions)
void trackObjects(const double *objectData, double *trackingResults, int size) {
```

```
    // Assuming a simple movement model: move each object by a fixed amount (e.g., +1.0 in both x and y directions)
    for (int i = 0; i < size; i += 2) {
        // Update the x and y positions of each object (move by +1.0)
        trackingResults[i] = objectData[i] + 1.0;     // Update x-coordinate
        trackingResults[i + 1] = objectData[i + 1] + 1.0; // Update y-coordinate
    }
}
```

O/p: Tracked Object Positions:

Object 1: (2.000000, 3.000000)

Object 2: (4.000000, 5.000000)

Object 3: (6.000000, 7.000000)

29. Defense Strategy Optimizer

o Function Prototype: void optimizeDefenseStrategy(const double *threatLevels, double *optimizedStrategies, int size)

Sol:

```
#include <stdio.h>

// Function prototype
void optimizeDefenseStrategy(const double *threatLevels, double *optimizedStrategies, int size);

int main() {
    // Example threat levels (e.g., threat intensities or risk factors)
    int size = 5;  // Number of different threats
    double threatLevels[] = {2.5, 7.8, 3.4, 9.0, 5.1};  // Example threat levels

    double optimizedStrategies[size];  // Array to store optimized defense strategies

    // Call the function to optimize the defense strategy
    optimizeDefenseStrategy(threatLevels, optimizedStrategies, size);

    // Print the optimized strategies
    printf("Optimized Defense Strategies:\n");
```

```
    for (int i = 0; i < size; i++) {
        printf("Threat %d: %.2f -> Optimized Strategy: %.2f\n", i + 1, threatLevels[i],
optimizedStrategies[i]);
    }

    return 0;
}

// Function to optimize defense strategy based on threat levels
void optimizeDefenseStrategy(const double *threatLevels, double *optimizedStrategies, int size) {
    double scalingFactor = 1.5;  // Assume we apply a scaling factor to optimize the strategy

    // Loop through each threat level and calculate the corresponding optimized defense strategy
    for (int i = 0; i < size; i++) {
        optimizedStrategies[i] = threatLevels[i] * scalingFactor;  // Apply scaling factor
    }
}
```

O/p: Optimized Defense Strategies:

Threat 1: 2.50 -> Optimized Strategy: 3.75

Threat 2: 7.80 -> Optimized Strategy: 11.70

Threat 3: 3.40 -> Optimized Strategy: 5.10

Threat 4: 9.00 -> Optimized Strategy: 13.50

Threat 5: 5.10 -> Optimized Strategy: 7.65

1. **String Length Calculation**
    - **Requirement**: Write a program that takes a string input and calculates its length using strlen(). The program should handle empty strings and output appropriate messages.
    - **Input**: A string from the user.
    - **Output**: Length of the string.

Sol:

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

```
int main() {
    char input[100];  // Array to store the input string

    // Prompting the user to enter a string
    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);  // Using fgets to handle spaces in input

    // Remove newline character if present
    size_t len = strlen(input);
    if (len > 0 && input[len - 1] == '\n') {
        input[len - 1] = '\0';
    }

    // Calculate and display the length of the string
    len = strlen(input);
    if (len == 0) {
        printf("The string is empty.\n");
    } else {
        printf("The length of the string is: %zu\n", len);
    }

    return 0;
}
```

O/p: Enter a string: Sofia

The length of the string is: 5

2. **String Copy**
    - **Requirement**: Implement a program that copies one string to another using strcpy(). The program should validate if the source string fits into the destination buffer.
    - **Input**: Two strings from the user (source and destination).
    - **Output**: The copied string.

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

int main() {
   char source[100];   // Buffer for the source string
   char destination[100];  // Buffer for the destination string

   // Prompting the user to enter the source string
   printf("Enter the source string: ");
   fgets(source, sizeof(source), stdin);

   // Remove newline character if present
   size_t len = strlen(source);
   if (len > 0 && source[len - 1] == '\n') {
      source[len - 1] = '\0';
   }

   // Check if the source string fits into the destination buffer
   if (strlen(source) >= sizeof(destination)) {
      printf("Error: The source string is too large to fit into the destination buffer.\n");
   } else {
      strcpy(destination, source);  // Copying the source string into destination
      printf("The copied string is: %s\n", destination);
   }

   return 0;
}
```
O/p:
```
Enter the source string: hello sofia
The copied string is: hello sofia
```

3. **String Concatenation**

- **Requirement**: Create a program that concatenates two strings using strcat(). Ensure the destination string has enough space to hold the result.
- **Input**: Two strings from the user.
- **Output**: The concatenated string.

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

int main() {
    char str1[200];  // Buffer for the first string (destination)
    char str2[100];  // Buffer for the second string

    // Prompting the user to enter the first string
    printf("Enter the first string: ");
    fgets(str1, sizeof(str1), stdin);

    // Remove newline character from the first string if present
    size_t len1 = strlen(str1);
    if (len1 > 0 && str1[len1 - 1] == '\n') {
        str1[len1 - 1] = '\0';
    }

    // Prompting the user to enter the second string
    printf("Enter the second string: ");
    fgets(str2, sizeof(str2), stdin);

    // Remove newline character from the second string if present
    size_t len2 = strlen(str2);
    if (len2 > 0 && str2[len2 - 1] == '\n') {
        str2[len2 - 1] = '\0';
    }
```

```
    // Check if the concatenated string will fit in the destination buffer
    if (strlen(str1) + strlen(str2) + 1 > sizeof(str1)) {
        printf("Error: Not enough space to concatenate the strings.\n");
    } else {
        strcat(str1, str2);  // Concatenating str2 to str1
        printf("The concatenated string is: %s\n", str1);
    }

    return 0;
}
```

O/p: Enter the first string: Hello

Enter the second string: Sofia

The concatenated string is: HelloSofia

4. **String Comparison**
    - **Requirement**: Develop a program that compares two strings using strcmp(). It should indicate if they are equal or which one is greater.
    - **Input**: Two strings from the user.
    - **Output**: Comparison result.

Sol:

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

int main() {
    char str1[100];  // Buffer for the first string
    char str2[100];  // Buffer for the second string

    // Prompting the user to enter the first string
    printf("Enter the first string: ");
    fgets(str1, sizeof(str1), stdin);

    // Remove newline character from the first string if present
    size_t len1 = strlen(str1);
```

```
    if (len1 > 0 && str1[len1 - 1] == '\n') {
        str1[len1 - 1] = '\0';
    }

    // Prompting the user to enter the second string
    printf("Enter the second string: ");
    fgets(str2, sizeof(str2), stdin);

    // Remove newline character from the second string if present
    size_t len2 = strlen(str2);
    if (len2 > 0 && str2[len2 - 1] == '\n') {
        str2[len2 - 1] = '\0';
    }

    // Compare the two strings
    int result = strcmp(str1, str2);
    if (result == 0) {
        printf("The strings are equal.\n");
    } else if (result < 0) {
        printf("The first string is less than the second string.\n");
    } else {
        printf("The first string is greater than the second string.\n");
    }

    return 0;
}
```

O/p: Enter the first string: Sofia

Enter the second string: Mickelen

The first string is less than the second string.

5. **Convert to Uppercase**

- **Requirement**: Write a program that converts all characters in a string to uppercase using strupr().
- **Input**: A string from the user.
- **Output**: The uppercase version of the string.

Sol:

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

void toUpperCase(char *str) {
    while (*str) {
        *str = toupper(*str);  // Convert each character to uppercase
        str++;
    }
}

int main() {
    char input[100];  // Buffer for the input string

    // Prompting the user to enter a string
    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);

    // Remove newline character if present
    size_t len = strlen(input);
    if (len > 0 && input[len - 1] == '\n') {
        input[len - 1] = '\0';
    }

    // Convert to uppercase
    toUpperCase(input);
```

```
    // Display the result
    printf("The uppercase string is: %s\n", input);

    return 0;
}
```

O/p: Enter a string: likitha

The uppercase string is: LIKITHA

6. **Convert to Lowercase**
    - **Requirement**: Implement a program that converts all characters in a string to lowercase using strlwr().
    - **Input**: A string from the user.
    - **Output**: The lowercase version of the string.

Sol:

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

void toLowerCase(char *str) {
    while (*str) {
        *str = tolower(*str);  // Convert each character to lowercase
        str++;
    }
}

int main() {
    char input[100];  // Buffer for the input string

    // Prompting the user to enter a string
    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);

    // Remove newline character if present
```

```
    size_t len = strlen(input);
    if (len > 0 && input[len - 1] == '\n') {
        input[len - 1] = '\0';
    }

    // Convert to lowercase
    toLowerCase(input);

    // Display the result
    printf("The lowercase string is: %s\n", input);

    return 0;
}
```

O/p: [?2004l

Enter a string: SOFIA MICKELEN

The lowercase string is: sofia mickelen

7. **Substring Search**
    - **Requirement**: Create a program that searches for a substring within a given string using strstr() and returns its starting index or an appropriate message if not found.
    - **Input**: A main string and a substring from the user.
    - **Output**: Starting index or not found message.

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

int main() {
    char mainStr[200];  // Buffer for the main string
    char subStr[100];   // Buffer for the substring

    // Prompting the user to enter the main string
    printf("Enter the main string: ");
    fgets(mainStr, sizeof(mainStr), stdin);
```

```
    // Remove newline character from the main string if present
    size_t len1 = strlen(mainStr);
    if (len1 > 0 && mainStr[len1 - 1] == '\n') {
        mainStr[len1 - 1] = '\0';
    }

    // Prompting the user to enter the substring
    printf("Enter the substring: ");
    fgets(subStr, sizeof(subStr), stdin);

    // Remove newline character from the substring if present
    size_t len2 = strlen(subStr);
    if (len2 > 0 && subStr[len2 - 1] == '\n') {
        subStr[len2 - 1] = '\0';
    }

    // Search for the substring
    char *position = strstr(mainStr, subStr);
    if (position != NULL) {
        // Calculate and display the starting index
        int index = position - mainStr;
        printf("Substring found at index: %d\n", index);
    } else {
        printf("Substring not found.\n");
    }

    return 0;
}
```

O/p: Enter the main string: hello world

Enter the substring: world

Substring found at index: 6

8. **Character Search**
    - **Requirement**: Write a program that finds the first occurrence of a character in a string using strchr() and returns its index or indicates if not found.
    - **Input**: A string and a character from the user.
    - **Output**: Index of first occurrence or not found message.

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

int main() {
    char str[100];  // Buffer for the input string
    char ch;        // Character to be searched

    // Prompting the user to enter the string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    // Remove newline character if present
    size_t len = strlen(str);
    if (len > 0 && str[len - 1] == '\n') {
        str[len - 1] = '\0';
    }

    // Prompting the user to enter the character
    printf("Enter the character to search for: ");
    scanf("%c", &ch);

    // Search for the character
    char *position = strchr(str, ch);
    if (position != NULL) {
        int index = position - str;  // Calculate index of the character
```

```
        printf("Character '%c' found at index: %d\n", ch, index);
    } else {
        printf("Character '%c' not found.\n", ch);
    }

    return 0;
}
```

O/p: Enter a string: Likitha Sirigowni

Enter the character to search for: a

Character 'a' found at index: 6

9. **String Reversal**
    - **Requirement**: Implement a function that reverses a given string in place without using additional memory, leveraging strlen() for length determination.
    - **Input**: A string from the user.
    - **Output**: The reversed string.

Sol:

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

void reverseString(char str[]) {
    int start = 0, end = strlen(str) - 1;
    while (start < end) {
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;
        start++;
        end--;
    }
}

int main() {
    char str[100];
```

```
    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    reverseString(str);
    printf("Reversed string: %s\n", str);

    return 0;
}
```

O/p: Enter a string: Hello

Reversed string: olleH

10. **String Tokenization**
    - **Requirement**: Create a program that tokenizes an input string into words using strtok() and counts how many tokens were found.
    - **Input**: A sentence from the user.
    - **Output**: Number of words (tokens).

Sol:

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

int main() {
    char str[100];

    // Taking input string
    printf("Enter a sentence: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    int count = 0;
    char *token = strtok(str, " ");  // Tokenize by spaces
```

```
    while (token != NULL) {
        count++;
        token = strtok(NULL, " ");
    }

    printf("Number of words: %d\n", count);

    return 0;
}
```

O/p: Enter a sentence: this is correct question

Number of words: 4

11. **String Duplication**
    - **Requirement**: Write a function that duplicates an input string (allocating new memory) using strdup() and displays both original and duplicated strings.
    - **Input**: A string from the user.
    - **Output**: Original and duplicated strings.

Sol:

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

int main() {
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    char *duplicatedStr = strdup(str);
    if (duplicatedStr != NULL) {
```

```
    printf("Original string: %s\n", str);
    printf("Duplicated string: %s\n", duplicatedStr);
    free(duplicatedStr);  // Don't forget to free allocated memory
  }

  return 0;
}
```

O/p: Enter a string: hello world!

Original string: hello world!

Duplicated string: hello world!

12. **Case-Insensitive Comparison**
    - **Requirement**: Develop a program to compare two strings without case sensitivity using strcasecmp() and report equality or differences.
    - **Input**: Two strings from the user.
    - **Output**: Comparison result.

Sol:

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

int main() {
  char str1[100], str2[100];

  // Taking input strings
  printf("Enter first string: ");
  fgets(str1, sizeof(str1), stdin);
  str1[strcspn(str1, "\n")] = '\0';  // Remove newline

  printf("Enter second string: ");
  fgets(str2, sizeof(str2), stdin);
  str2[strcspn(str2, "\n")] = '\0';  // Remove newline

  int result = strcasecmp(str1, str2);
```

```
    if (result == 0) {
        printf("The strings are equal (case-insensitive).\n");
    } else {
        printf("The strings are not equal (case-insensitive).\n");
    }

    return 0;
}
```

O/p: Enter first string: hello

Enter second string: likitha

The strings are not equal (case-insensitive).

13. **String Trimming**
    - **Requirement**: Implement functionality to trim leading and trailing whitespace from a given string, utilizing pointer arithmetic with strlen().
    - **Input**: A string with extra spaces from the user.
    - **Output**: Trimmed version of the string.

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

void trimWhitespace(char str[]) {
    int start = 0;
    int end = strlen(str) - 1;

    while (isspace(str[start])) start++;
    while (end >= start && isspace(str[end])) end--;

    str[end + 1] = '\0';  // Null-terminate after the last character
    memmove(str, &str[start], end - start + 2);  // Shift the string
}
```

```
int main() {
    char str[100];

    // Taking input string
    printf("Enter a string with spaces: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    trimWhitespace(str);
    printf("Trimmed string: '%s'\n", str);

    return 0;
}
```

O/p: Enter a string with spaces: N i k i t h a

Trimmed string: 'N i k i t h a'

14. **Find Last Occurrence of Character**
    - **Requirement**: Write a program that finds the last occurrence of a character in a string using manual iteration instead of library functions, returning its index.
    - **Input**: A string and a character from the user.
    - **Output**: Index of last occurrence or not found message.

Sol:

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

int main() {
    char str[100];
    char ch;

    // Taking input string and character
    printf("Enter the string: ");
    fgets(str, sizeof(str), stdin);
```

```
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter the character to search for: ");
    scanf("%c", &ch);

    int lastIndex = -1;

    for (int i = 0; i < strlen(str); i++) {
        if (str[i] == ch) {
            lastIndex = i;
        }
    }

    if (lastIndex != -1) {
        printf("Last occurrence of character '%c' is at index %d\n", ch, lastIndex);
    } else {
        printf("Character not found.\n");
    }

    return 0;
}
```

O/p:

Enter the string: hello india

Enter the character to search for: d

Last occurrence of character 'd' is at index 8

15. **Count Vowels in String**
    - **Requirement**: Create a program that counts how many vowels are present in an input string by iterating through each character.
    - **Input**: A string from the user.
    - **Output**: Count of vowels.

Sol: #include <stdio.h>

```
#include <ctype.h>

int countVowels(char str[]) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (strchr("aeiouAEIOU", str[i]) != NULL) {
            count++;
        }
    }
    return count;
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    int vowelCount = countVowels(str);
    printf("Number of vowels: %d\n", vowelCount);

    return 0;
}
```

O/p:

Enter a string: hello

Number of vowels: 2

16. **Count Specific Characters**
    - **Requirement**: Implement functionality to count how many times a specific character appears in an input string, allowing for case sensitivity options.

- **Input**: A string and a character from the user.
- **Output**: Count of occurrences.

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

int countCharacter(char str[], char ch) {
    int count = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == ch) {
            count++;
        }
    }
    return count;
}

int main() {
    char str[100], ch;

    // Taking input string and character
    printf("Enter the string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter the character to count: ");
    scanf("%c", &ch);

    int count = countCharacter(str, ch);
    printf("Character '%c' appears %d times.\n", ch, count);

    return 0;
}
```

O/p:
```
Enter the string: nikitha
```

Enter the character to count: i

Character 'i' appears 2 times.

17. **Remove All Occurrences of Character**
    - **Requirement**: Write a function that removes all occurrences of a specified character from an input string, modifying it in place.
    - **Input**: A string and a character to remove from it.
    - **Output**: Modified string without specified characters.

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

void removeCharacter(char str[], char ch) {
    int i = 0, j = 0;
    while (str[i] != '\0') {
        if (str[i] != ch) {
            str[j++] = str[i];
        }
        i++;
    }
    str[j] = '\0';
}

int main() {
    char str[100], ch;

    // Taking input string and character
    printf("Enter the string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter the character to remove: ");
    scanf("%c", &ch);
```

```
    removeCharacter(str, ch);
    printf("Modified string: %s\n", str);

    return 0;
}
```

O/p: Enter the string: hello

Enter the character to remove: o

Modified string: hell

18. **Check for Palindrome**
    - **Requirement**: Develop an algorithm to check if an input string is a palindrome by comparing characters from both ends towards the center, ignoring case and spaces.
    - **Input**: A potential palindrome from the user.
    - **Output**: Whether it is or isn't a palindrome.

Sol:

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

int isPalindrome(char str[]) {
    int start = 0, end = strlen(str) - 1;
    while (start < end) {
        if (tolower(str[start]) != tolower(str[end])) {
            return 0;  // Not a palindrome
        }
        start++;
        end--;
    }
    return 1;  // Palindrome
}

int main() {
```

```
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    if (isPalindrome(str)) {
        printf("The string is a palindrome.\n");
    } else {
        printf("The string is not a palindrome.\n");
    }

    return 0;
}
```

O/p: Enter a string: wow

The string is a palindrome.

19. **Extract Substring**
    - **Requirement**: Create functionality to extract a substring based on specified start index and length parameters, ensuring valid indices are provided by users.
    - **Input**: A main string, start index, and length from the user.
    - **Output**: Extracted substring or error message for invalid indices.

Sol:

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

void extractSubstring(char str[], int start, int length) {
    char substr[100];
    strncpy(substr, &str[start], length);
    substr[length] = '\0';
    printf("Extracted substring: %s\n", substr);
}
```

```
int main() {
    char str[100];
    int start, length;

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter start index and length: ");
    scanf("%d %d", &start, &length);

    if (start >= 0 && start < strlen(str)) {
        extractSubstring(str, start, length);
    } else {
        printf("Invalid start index.\n");
    }
    return 0;
}
```

O/p:

Enter a string: Sofia Mickelen

Enter start index and length: 3 7

Extracted substring:ia mick

20. **Sort Characters in String**

- **Requirement**: Implement functionality to sort characters in an input string alphabetically, demonstrating usage of nested loops for comparison without library sorting functions.
- **Input**: A string from the user.
- **Output**: Sorted version of the characters in the string.

Sol:

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

```
void sortString(char str[]) {
    int len = strlen(str);
    for (int i = 0; i < len - 1; i++) {
        for (int j = i + 1; j < len; j++) {
            if (str[i] > str[j]) {
                // Swap characters
                char temp = str[i];
                str[i] = str[j];
                str[j] = temp;
            }
        }
    }
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    sortString(str);
    printf("Sorted string: %s\n", str);

    return 0;
}
```

O/p:

```
Enter a string: likitha
Sorted string: ahiiklt
```

20.

21. **Count Words in String**

- **Requirement:** Write code to count how many words are present in an input sentence by identifying spaces as delimiters, utilizing strtok().
- **Input:** A sentence from the user.
  - **Output:** Number of words counted.

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

int countWords(char str[]) {
    int count = 0;
    char *token = strtok(str, " ");

    while (token != NULL) {
        count++;
        token = strtok(NULL, " ");
    }

    return count;
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a sentence: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    int wordCount = countWords(str);
    printf("Number of words: %d\n", wordCount);
```

```
    return 0;
}
```

O/p: Enter a sentence: this is india

Number of words: 3

22. **Remove Duplicates from String**
    - **Requirement:** Develop an algorithm to remove duplicate characters while maintaining their first occurrence order in an input string.
    - **Input:** A string with potential duplicate characters.
    - **Output:** Modified version of the original without duplicates.

Sol:

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

void removeDuplicates(char str[]) {
    int i, j, len = strlen(str);

    for (i = 0; i < len; i++) {
        for (j = i + 1; j < len; j++) {
            if (str[i] == str[j]) {
                for (int k = j; k < len; k++) {
                    str[k] = str[k + 1];
                }
                len--;
                j--;
            }
        }
    }
}

int main() {
    char str[100];

    // Taking input string
```

```
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    removeDuplicates(str);
    printf("String after removing duplicates: %s\n", str);

    return 0;
}
```

O/p:

Enter a string: 'likitha'

String after removing duplicates: 'liktha

23. **Find First Non-Repeating Character**
    - **Requirement:** Create functionality to find the first non-repeating character in an input string, demonstrating effective use of arrays for counting occurrences.
    - **Input:** A sample input from the user.
    - **Output:** The first non-repeating character or indication if all are repeating.

Sol:

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

char findFirstNonRepeatingChar(char str[]) {
    int count[256] = {0};

    for (int i = 0; str[i] != '\0'; i++) {
        count[str[i]]++;
    }

    for (int i = 0; str[i] != '\0'; i++) {
        if (count[str[i]] == 1) {
            return str[i];
        }
    }
```

```
    return '\0';  // No non-repeating character
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    char result = findFirstNonRepeatingChar(str);
    if (result != '\0') {
        printf("First non-repeating character: %c\n", result);
    } else {
        printf("No non-repeating character found.\n");
    }

    return 0;
}
```

O/p: Enter a string: red is green

First non-repeating character: d

24. **Convert String to Integer**
    - **Requirement:** Implement functionality to convert numeric strings into integer values without using standard conversion functions like atoi(), handling invalid inputs gracefully.
    - **Input:** A numeric string.
    - **Output:** Converted integer value or error message.

Sol:

```
#include <stdio.h>

int convertToInt(char str[]) {
    int result = 0;
```

```
    int i = 0;

    // Handle negative numbers
    int sign = 1;
    if (str[i] == '-') {
        sign = -1;
        i++;
    }

    for (; str[i] != '\0'; i++) {
        result = result * 10 + (str[i] - '0');
    }

    return sign * result;
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a number string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    int number = convertToInt(str);
    printf("Converted integer: %d\n", number);

    return 0;
}
```

o/p: Enter a number string: 12345

Converted integer: 12345

25. **Check Anagram Status Between Two Strings**
    - **Requirement:** Write code to check if two strings are anagrams by sorting their characters and comparing them.
    - **Input:** Two strings.
    - **Output:** Whether they are anagrams.

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

int areAnagrams(char str1[], char str2[]) {
    if (strlen(str1) != strlen(str2)) {
        return 0;  // Not anagrams
    }

    int count[256] = {0};

    for (int i = 0; str1[i] != '\0'; i++) {
        count[str1[i]]++;
        count[str2[i]]--;
    }

    for (int i = 0; i < 256; i++) {
        if (count[i] != 0) {
            return 0;  // Not anagrams
        }
    }

    return 1;  // Anagrams
}

int main() {
    char str1[100], str2[100];
```

```
    // Taking input strings
    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    str1[strcspn(str1, "\n")] = '\0';  // Remove newline

    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str2[strcspn(str2, "\n")] = '\0';  // Remove newline

    if (areAnagrams(str1, str2)) {
        printf("The strings are anagrams.\n");
    } else {
        printf("The strings are not anagrams.\n");
    }

    return 0;
}
```

O/p: Enter first string: hello

Enter second string: world

The strings are not anagrams.

26. **Merge Two Strings Alternately**
    - **Requirement:** Create functionality to merge two strings alternately into one while handling cases where strings may be of different lengths.
    - **Input:** Two strings.
    - **Output:** Merged alternating characters.

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

void mergeAlternately(char str1[], char str2[], char result[]) {
    int i = 0, j = 0, k = 0;
```

```
    // Merge both strings alternately
    while (str1[i] != '\0' && str2[j] != '\0') {
        result[k++] = str1[i++];
        result[k++] = str2[j++];
    }

    // Append remaining characters
    while (str1[i] != '\0') result[k++] = str1[i++];
    while (str2[j] != '\0') result[k++] = str2[j++];

    result[k] = '\0';
}

int main() {
    char str1[100], str2[100], result[200];

    // Taking input strings
    printf("Enter first string: ");
    fgets(str1, sizeof(str1), stdin);
    str1[strcspn(str1, "\n")] = '\0';  // Remove newline

    printf("Enter second string: ");
    fgets(str2, sizeof(str2), stdin);
    str2[strcspn(str2, "\n")] = '\0';  // Remove newline

    mergeAlternately(str1, str2, result);
    printf("Merged string: %s\n", result);

    return 0;
}
```

O/p: Enter first string: hello

Enter second string: world

Merged string: hweolrllod

27. **Count Consonants in String**
    - **Requirement:** Develop code to count consonants while ignoring vowels and whitespace characters.
    - **Input:** Any input text.
    - **Output:** Count of consonants.

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

int countConsonants(char str[]) {
    int count = 0;

    for (int i = 0; str[i] != '\0'; i++) {
        if (isalpha(str[i]) && !strchr("aeiouAEIOU", str[i])) {
            count++;
        }
    }

    return count;
}

int main() {
    char str[100];

    // Taking input string
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    int consonantCount = countConsonants(str);
    printf("Number of consonants: %d\n", consonantCount);
```

```
    return 0;
}
```

O/p: Enter a string: hello

Number of consonants: 3

28. **Replace Substring with Another String**
    - **Requirement:** Write functionality to replace all occurrences of one substring with another within a given main string.
    - **Input:** Main text, target substring, replacement substring.
    - **Output:** Modified main text after replacements.

Sol:

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

void replaceSubstring(char str[], char oldSub[], char newSub[]) {
    char temp[1000];
    int i = 0, j = 0;

    while (str[i] != '\0') {
        if (strncmp(&str[i], oldSub, strlen(oldSub)) == 0) {
            strcpy(&temp[j], newSub);
            j += strlen(newSub);
            i += strlen(oldSub);
        } else {
            temp[j++] = str[i++];
        }
    }

    temp[j] = '\0';
    strcpy(str, temp);
}

int main() {
```

```
    char str[100], oldSub[100], newSub[100];

    // Taking input strings
    printf("Enter main string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter substring to replace: ");
    fgets(oldSub, sizeof(oldSub), stdin);
    oldSub[strcspn(oldSub, "\n")] = '\0';  // Remove newline

    printf("Enter new substring: ");
    fgets(newSub, sizeof(newSub), stdin);
    newSub[strcspn(newSub, "\n")] = '\0';  // Remove newline

    replaceSubstring(str, oldSub, newSub);
    printf("Updated string: %s\n", str);

    return 0;
}
```

O/p: Enter main string: this is Sofi

Enter substring to replace: mickey

Enter new substring: ria

Updated string: this is ria

29. **Count Occurrences of Substring**
    - **Requirement:** Create code that counts how many times one substring appears within another larger main text without overlapping occurrences.
    - **Input:** Main text and target substring.
    - **Output:** Count of occurrences.

Sol: #include <stdio.h>

#include <string.h>

```
int countOccurrences(char str[], char sub[]) {
    int count = 0;
    char *pos = str;

    while ((pos = strstr(pos, sub)) != NULL) {
        count++;
        pos++;  // Move to next character after found substring
    }

    return count;
}

int main() {
    char str[100], sub[100];

    // Taking input strings
    printf("Enter main string: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';  // Remove newline

    printf("Enter substring to count: ");
    fgets(sub, sizeof(sub), stdin);
    sub[strcspn(sub, "\n")] = '\0';  // Remove newline

    int count = countOccurrences(str, sub);
    printf("Occurrences of '%s': %d\n", sub, count);

    return 0;
}
```

Sol: Enter main string: hello hello Mickelen

Enter substring to count: hello

Occurrences of 'hello': 2

30. **Implement Custom String Length Function**
    - **Requirement:** Finally, write your own implementation of strlen() function from scratch, demonstrating pointer manipulation techniques.
    - **Input:** Any input text.
    - **Output:** Length calculated by custom function.

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


int customStrlen(char str[]) {
   int length = 0;
   while (str[length] != '\0') {
      length++;
   }
   return length;
}


int main() {
   char str[100];


   // Taking input string
   printf("Enter a string: ");
   fgets(str, sizeof(str), stdin);
   str[strcspn(str, "\n")] = '\0';  // Remove newline


   int length = customStrlen(str);
   printf("Length of string: %d\n", length);


   return 0;
}
```

o/p:

```
Enter a string: Sofia
Length of string: 7
```