ARRAY AND POINTERS

```
#include <stdio.h>

int main()
{
  int a[10];
  int *ptr;
  ptr=a;
  for(int i=0;i<10;i++){
      scanf("%d \n",(ptr+i));
  }
  for(int k=0;k<10;k++){
      printf("%d =>",a[k]);
  }
    return 0;
}
```

2.

```
#include <stdio.h>

int main()
{
  int a[10];
  int *ptr;
  ptr=a;
  for(int i=0;i<10;i++){
      scanf("%d \n",(ptr+i));
  }
  for(int k=0;k<10;k++){
      printf("%d =>",*(ptr+k));//DEREFERENCING
  }
    return 0;
}
```

```
#include <stdio.h>

int main()
{
  int a[10];
  int *ptr;
  ptr=a;
  for(int i=0;i<10;i++){
      scanf("%d \n",&a[i]);
  }
  for(int k=0;k<10;k++){
      printf("%d =>",*(ptr+k));
  }
    return 0;
}
```

```
#include <stdio.h>

int main()
{
  int a[5];
  int *ptr;
  ptr=a;
  for(int i=0;i<5;i++){
      scanf("%d \n",(a+i));
  }
  for(int k=0;k<5;k++){
      printf("%d =>",*(a+k));//dereferencing
  }
    return 0;
}
```

RETURNING A POINTER

```
#include <stdio.h>
int* sum();//return an address
int main()
{

  int *p;
  p=sum();
  printf("sum=%d ",*p);
 return 0;
}
int* sum(){
    int sum=20;
    int *ptr = &sum;
    return ptr;
}
```

BY USING CHAR

```
#include <stdio.h>
char* sum();//return an address
int main()
{

  char *p;
  p=sum();
  printf("sum=%c ",*p);
 return 0;
}
char* sum(){
    char sum='A';
    char *ptr = &sum;
    return ptr;
```

}

--|

```
#include <stdio.h>
void PrintArray(int a[],int n);
int main()
{
    int a[5]={1,2,3,4,5};
    PrintArray(a,5);//a represent the address of first element
  return 0;
}
void PrintArray(int a[],int n){
    for(int i=0;i<n;i++){
        printf("%d \n",a[i]);
    }
}
```

```
#include <stdio.h>
void PrintArray(int a[],int n);
int main()
{
    int a[5]={1,2,3,4,5};
    PrintArray(a,5);//a represent the address of first element
  return 0;
}
void PrintArray(int *ptr,int n){
    for(int i=0;i<n;i++){
        printf("%d \n",*(ptr+i));//using pointer
    }
}
```

```
#include <stdio.h>
void PrintArray(int a[],int n);
int main()
{
    int a[5]={1,2,3,4,5};
    PrintArray(&a[0],5);//By Using &
  return 0;
}
void PrintArray(int *ptr,int n){
    for(int i=0;i<n;i++){
        printf("%d \n",*(ptr+i));//using pointer
    }
}
```

--|

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char multiple[]="a string";
    char *p=multiple;
    for(int i=0;i<strnlen(multiple,sizeof(multiple));++i)
    printf("multiple[%d]=%c *(p+%d)=%c &multiple[%d]=%p p+%d=%p
\n",i,multiple[i],i,*(p+i),i,&multiple[i],i,p+i);
  return 0;
}
```

SET OF PROBLEMS

1.Pointers: Use to traverse the trajectory array.
Arrays: Store trajectory points (x, y, z) at discrete time intervals.
Functions:
void calculate_trajectory(const double *parameters, double *trajectory, int size): Takes the initial velocity, angle, and an array to store trajectory points.
void print_trajectory(const double *trajectory, int size): Prints the stored trajectory points.
Pass Arrays as Pointers: Pass the trajectory array as a pointer to the calculation function

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

#define GRAVITY 9.81  // Acceleration due to gravity (m/s^2)

void calculate_trajectory(const double *parameters, double *trajectory, int size) {
    // parameters[0] = initial velocity, parameters[1] = angle in degrees
    double velocity = parameters[0];
    double angle_deg = parameters[1];
    double angle_rad = angle_deg * (M_PI / 180.0);  // Convert angle to radians

    double time_of_flight = (2 * velocity * sin(angle_rad)) / GRAVITY;
    double time_interval = time_of_flight / (size - 1);  // Divide flight time into 'size' intervals

    // Calculate trajectory points (x, y, z)
    for (int i = 0; i < size; ++i) {
        double t = i * time_interval;  // Current time
        double x = velocity * cos(angle_rad) * t;  // Horizontal distance
        double y = velocity * sin(angle_rad) * t - 0.5 * GRAVITY * t * t;  // Vertical distance

        trajectory[2 * i] = x;   // x-coordinate
        trajectory[2 * i + 1] = y; // y-coordinate
    }
}

void print_trajectory(const double *trajectory, int size) {
    for (int i = 0; i < size; ++i) {
        printf("Point %d: (x = %.2f, y = %.2f)\n", i, trajectory[2 * i], trajectory[2 * i + 1]);
    }
}

int main() {
    double parameters[2] = {50.0, 45.0};  // Initial velocity (m/s), angle (degrees)
```

```c
    int size = 10;  // Number of trajectory points

    double trajectory[2 * size];  // Array to store x, y points for each time step

    // Calculate the trajectory
    calculate_trajectory(parameters, trajectory, size);

    // Print the trajectory
    print_trajectory(trajectory, size);

    return 0;
}
```

2.Pointers: Manipulate position and velocity vectors.
Arrays: Represent the satellite's position over time as an array of 3D vectors.
Functions:
void update_position(const double *velocity, double *position, int size): Updates the position based on velocity.
void simulate_orbit(const double *initial_conditions, double *positions, int steps): Simulates orbit over a specified number of steps.
Pass Arrays as Pointers: Use pointers for both velocity and position arrays.

```c
#include <stdio.h>

// Function to update the position based on velocity
void update_position(const double *velocity, double *position, int size) {
    // Assuming size is 3, for a 3D vector (x, y, z)
    for (int i = 0; i < size; i++) {
        position[i] += velocity[i]; // Update the position: position = position + velocity
    }
}

// Function to simulate the orbit over a specified number of steps
void simulate_orbit(const double *initial_conditions, double *positions, int steps) {
    // Initial conditions: {x, y, z, vx, vy, vz}
    double position[3] = {initial_conditions[0], initial_conditions[1], initial_conditions[2]};
    double velocity[3] = {initial_conditions[3], initial_conditions[4], initial_conditions[5]};

    // Array to store the positions over time (output)
    for (int step = 0; step < steps; step++) {
        // Store the current position in the positions array (flattens the 3D vector)
        positions[step * 3 + 0] = position[0];
        positions[step * 3 + 1] = position[1];
        positions[step * 3 + 2] = position[2];

        // Update the position based on velocity
        update_position(velocity, position, 3);
    }
}

int main() {
    // Initial conditions: {x, y, z, vx, vy, vz}
    double initial_conditions[6] = {0.0, 0.0, 0.0, 1.0, 0.0, 0.0}; // Initial position and velocity

    // Number of steps for simulation
```

```
    int steps = 10;

    // Array to store the positions at each step (flattens the 3D vector over time)
    double positions[steps * 3];

    // Call the simulate_orbit function to update the positions
    simulate_orbit(initial_conditions, positions, steps);

    // Print the positions over time
    printf("Positions over time:\n");
    for (int step = 0; step < steps; step++) {
        printf("Step %d: (%f, %f, %f)\n", step, positions[step * 3 + 0], positions[step * 3 + 1], positions[step *
3 + 2]);
    }

    return 0;
}
```

3.Pointers: Traverse weather data arrays efficiently.
Arrays: Store hourly temperature, wind speed, and pressure.
Functions:
void calculate_daily_averages(const double *data, int size, double *averages): Computes daily averages for each parameter.
void display_weather_data(const double *data, int size): Displays data for monitoring purposes.
Pass Arrays as Pointers: Pass weather data as pointers to processing functions.

```
#include <stdio.h>

// Function to calculate daily averages for each parameter
void calculate_daily_averages(const double *data, int size, double *averages) {
    double sum = 0.0;

    // Calculate the sum of the data
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }

    // Calculate the average
    *averages = sum / size;
}

// Function to display weather data for monitoring purposes
void display_weather_data(const double *data, int size) {
    printf("Weather Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Hour %d: %.2f\n", i + 1, data[i]);
    }
}

int main() {
    // Sample weather data for 24 hours
    double temperature[24] = {20.5, 21.3, 22.1, 22.8, 23.4, 24.0, 24.5, 24.7, 24.8, 24.9, 25.0, 25.1,
                              25.2, 25.3, 25.4, 25.4, 25.3, 25.2, 25.0, 24.7, 24.5, 24.2, 23.8, 23.2, 22.5};
    double wind_speed[24] = {5.0, 5.2, 5.3, 5.5, 5.6, 5.8, 6.0, 6.1, 6.3, 6.4, 6.5, 6.6,
                             6.7, 6.8, 6.9, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 8.0};
    double pressure[24] = {1013.5, 1013.4, 1013.3, 1013.1, 1013.0, 1012.9, 1012.8, 1012.7, 1012.6,
```

```
1012.5,
                        1012.4, 1012.3, 1012.2, 1012.1, 1012.0, 1011.9, 1011.8, 1011.7, 1011.6, 1011.5,
                        1011.4, 1011.3, 1011.2, 1011.1, 1011.0};

    double avg_temperature, avg_wind_speed, avg_pressure;

    // Calculate daily averages for each parameter
    calculate_daily_averages(temperature, 24, &avg_temperature);
    calculate_daily_averages(wind_speed, 24, &avg_wind_speed);
    calculate_daily_averages(pressure, 24, &avg_pressure);

    // Display hourly data for monitoring
    printf("Hourly Weather Data:\n");
    display_weather_data(temperature, 24);
    display_weather_data(wind_speed, 24);
    display_weather_data(pressure, 24);

    // Display daily averages
    printf("\nDaily Averages:\n");
    printf("Average Temperature: %.2f°C\n", avg_temperature);
    printf("Average Wind Speed: %.2f km/h\n", avg_wind_speed);
    printf("Average Pressure: %.2f hPa\n", avg_pressure);

    return 0;
}
```

4.Pointers: Traverse and manipulate error values in arrays.
Arrays: Store historical error values for proportional, integral, and derivative calculations.
Functions:
double compute_pid(const double *errors, int size, const double *gains): Calculates control output using PID logic.
void update_errors(double *errors, double new_error): Updates the error array with the latest value.
Pass Arrays as Pointers: Use pointers for the errors array and the gains array.

```
#include <stdio.h>

#define PID_HISTORY_SIZE 3  // For storing the last 3 errors (Proportional, Integral, Derivative)

// Function to compute PID control output based on error values and gains
double compute_pid(const double *errors, int size, const double *gains) {
    if (size != PID_HISTORY_SIZE) {
        printf("Error: Array size mismatch.\n");
        return -1;
    }

    // Proportional, Integral, and Derivative calculations
    double proportional = errors[0];  // P error (current error)
    double integral = 0;  // I error (sum of previous errors)
    double derivative = 0;  // D error (change in error)

    for (int i = 1; i < size; i++) {
        integral += errors[i];  // Accumulate error for integral term
        derivative = errors[i] - errors[i - 1];  // Difference between current and previous error for derivative
term
    }
```

```c
    // PID formula: P + I + D
    double pid_output = gains[0] * proportional + gains[1] * integral + gains[2] * derivative;

    return pid_output;
}

// Function to update the error array with the latest error value
void update_errors(double *errors, double new_error) {
    // Shift the historical error values to the right
    for (int i = PID_HISTORY_SIZE - 1; i > 0; i--) {
        errors[i] = errors[i - 1];
    }

    // Insert the latest error value at the front
    errors[0] = new_error;
}

int main() {
    double errors[PID_HISTORY_SIZE] = {0.0, 0.0, 0.0};  // Array to store historical errors
    double gains[3] = {1.0, 0.1, 0.01};  // PID gains: P, I, D

    // Simulate receiving new errors
    update_errors(errors, 0.5);  // New error: 0.5
    update_errors(errors, 0.4);  // New error: 0.4
    update_errors(errors, 0.3);  // New error: 0.3

    // Compute PID control output
    double output = compute_pid(errors, PID_HISTORY_SIZE, gains);

    // Output the result
    printf("PID Control Output: %f\n", output);

    return 0;
}
```

5. Aircraft Sensor Data Fusion
Pointers: Handle sensor readings and fusion results.
Arrays: Store data from multiple sensors.
Functions:
void fuse_data(const double *sensor1, const double *sensor2, double *result, int size): Merges two sensor datasets into a single result array.
void calibrate_data(double *data, int size): Adjusts sensor readings based on calibration data.
Pass Arrays as Pointers: Pass sensor arrays as pointers to fusion and calibration functions

```c
#include <stdio.h>

// Function to fuse data from two sensors into one result array
void fuse_data(const double *sensor1, const double *sensor2, double *result, int size) {
    for (int i = 0; i < size; i++) {
        result[i] = (sensor1[i] + sensor2[i]) / 2;  // Simple averaging of sensor data
    }
}

// Function to calibrate sensor data by applying some offset or scaling factor
```

```
void calibrate_data(double *data, int size) {
    for (int i = 0; i < size; i++) {
        // Applying a simple calibration factor, for example, subtracting an offset
        // In real cases, the calibration logic may involve more complex mathematical operations
        data[i] -= 1.0;  // Example calibration: subtract 1.0 from each sensor reading
    }
}

int main() {
    int size = 5;  // Example array size

    // Example sensor readings (arrays)
    double sensor1[5] = {10.0, 20.0, 30.0, 40.0, 50.0};
    double sensor2[5] = {12.0, 22.0, 32.0, 42.0, 52.0};

    // Array to store fused data
    double result[5];

    // Fuse sensor data
    fuse_data(sensor1, sensor2, result, size);

    // Print fused data
    printf("Fused Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Result[%d]: %.2f\n", i, result[i]);
    }

    // Calibrate sensor1 data
    calibrate_data(sensor1, size);

    // Print calibrated data
    printf("\nCalibrated Sensor1 Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Sensor1[%d]: %.2f\n", i, sensor1[i]);
    }

    return 0;
}
```

6.Air Traffic Management
Pointers: Traverse the array of flight structures.
Arrays: Store details of active flights (e.g., ID, altitude, coordinates).
Functions:
void add_flight(flight_t *flights, int *flight_count, const flight_t *new_flight): Adds a new flight to the system.
void remove_flight(flight_t *flights, int *flight_count, int flight_id): Removes a flight by ID.
Pass Arrays as Pointers: Use pointers to manipulate the array of flight structures.

```
#include <stdio.h>

#define MAX_FLIGHTS 5

// Function to add a new flight
void add_flight(int *flight_ids, double *altitudes, int *flight_count, int id, double altitude) {
    if (*flight_count < MAX_FLIGHTS) {
```

```
        flight_ids[*flight_count] = id;
        altitudes[*flight_count] = altitude;
        (*flight_count)++;
        printf("Flight %d added.\n", id);
    } else {
        printf("Error: Maximum flight limit reached.\n");
    }
}

// Function to display all flights
void display_flights(const int *flight_ids, const double *altitudes, int flight_count) {
    printf("Active Flights:\n");
    for (int i = 0; i < flight_count; i++) {
        printf("ID: %d, Altitude: %.2f\n", flight_ids[i], altitudes[i]);
    }
}

int main() {
    int flight_ids[MAX_FLIGHTS];    // Array to store flight IDs
    double altitudes[MAX_FLIGHTS];  // Array to store altitudes
    int flight_count = 0;           // Number of flights

    // Adding flights
    add_flight(flight_ids, altitudes, &flight_count, 101, 35000.0);
    add_flight(flight_ids, altitudes, &flight_count, 102, 30000.0);

    // Display flights
    display_flights(flight_ids, altitudes, flight_count);

    return 0;
}
```

7.Pointers: Traverse telemetry data arrays.
Arrays: Store telemetry parameters (e.g., power, temperature, voltage).
Functions:
void analyze_telemetry(const double *data, int size): Computes statistical metrics for telemetry data.
void filter_outliers(double *data, int size): Removes outliers from the telemetry data array.
Pass Arrays as Pointers: Pass telemetry data arrays to both functions.

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

#define MAX_SIZE 100

// Function to compute mean and standard deviation
void analyze_telemetry(const double *data, int size) {
    if (size <= 0) return;

    double sum = 0.0;
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }

    double mean = sum / size;
```

```c
    // Compute standard deviation
    double variance = 0.0;
    for (int i = 0; i < size; i++) {
        variance += pow(data[i] - mean, 2);
    }
    double stddev = sqrt(variance / size);

    printf("Mean: %.2f\n", mean);
    printf("Standard Deviation: %.2f\n", stddev);
}

// Function to filter outliers based on 2 standard deviations
void filter_outliers(double *data, int *size) {
    if (*size <= 0) return;

    // Calculate mean and standard deviation
    double sum = 0.0;
    for (int i = 0; i < *size; i++) {
        sum += data[i];
    }
    double mean = sum / *size;

    double variance = 0.0;
    for (int i = 0; i < *size; i++) {
        variance += pow(data[i] - mean, 2);
    }
    double stddev = sqrt(variance / *size);

    // Filter outliers (more than 2 standard deviations from the mean)
    int new_size = 0;
    for (int i = 0; i < *size; i++) {
        if (fabs(data[i] - mean) <= 2 * stddev) {
            data[new_size++] = data[i];
        }
    }
    *size = new_size;  // Update size after filtering
}

int main() {
    // Example telemetry data arrays
    double power_data[MAX_SIZE] = {3.2, 4.0, 5.1, 3.9, 2.5, 15.0, 4.3}; // Example with an outlier (15.0)
    double temperature_data[MAX_SIZE] = {22.1, 23.5, 20.0, 21.5, 23.0};
    double voltage_data[MAX_SIZE] = {12.4, 12.7, 12.2, 11.9, 12.0};

    int power_size = 7;  // Number of elements in the power data array
    int temp_size = 5;   // Number of elements in the temperature data array
    int voltage_size = 5; // Number of elements in the voltage data array

    // Analyze telemetry data
    printf("Power Data Analysis:\n");
    analyze_telemetry(power_data, power_size);

    printf("\nTemperature Data Analysis:\n");
    analyze_telemetry(temperature_data, temp_size);
```

```
    printf("\nVoltage Data Analysis:\n");
    analyze_telemetry(voltage_data, voltage_size);

    // Filter outliers from the power data
    printf("\nFiltering Outliers from Power Data...\n");
    filter_outliers(power_data, &power_size);

    printf("\nPower Data after Filtering:\n");
    for (int i = 0; i < power_size; i++) {
        printf("%.2f ", power_data[i]);
    }
    printf("\n");

    return 0;
}
```

8.Rocket Thrust Calculation
Pointers: Traverse thrust arrays.
Arrays: Store thrust values for each stage of the rocket.
Functions:
double compute_total_thrust(const double *stages, int size): Calculates cumulative thrust across all stages.
void update_stage_thrust(double *stages, int stage, double new_thrust): Updates thrust for a specific stage.
Pass Arrays as Pointers: Use pointers for thrust arrays.

```
#include <stdio.h>

#define NUM_STAGES 5  // Define the number of stages in the rocket

// Function to compute the total thrust
double compute_total_thrust(const double *stages, int size) {
    double total_thrust = 0.0;

    // Traverse through the stages array and sum the thrust values
    for (int i = 0; i < size; i++) {
        total_thrust += stages[i];
    }

    return total_thrust;
}

// Function to update the thrust of a specific stage
void update_stage_thrust(double *stages, int stage, double new_thrust) {
    // Ensure the stage is within bounds
    if (stage >= 0 && stage < NUM_STAGES) {
        stages[stage] = new_thrust;
        printf("Stage %d thrust updated to %.2f\n", stage, new_thrust);
    } else {
        printf("Invalid stage number!\n");
    }
}

int main() {
    // Example array for rocket thrust across 5 stages
```

```
    double rocket_thrust[NUM_STAGES] = {500.0, 700.0, 600.0, 450.0, 800.0};

    // Displaying initial thrust array
    printf("Initial Thrust values for each stage:\n");
    for (int i = 0; i < NUM_STAGES; i++) {
        printf("Stage %d thrust: %.2f\n", i, rocket_thrust[i]);
    }

    // Calculate the total thrust of the rocket
    double total_thrust = compute_total_thrust(rocket_thrust, NUM_STAGES);
    printf("\nTotal rocket thrust: %.2f\n", total_thrust);

    // Updating the thrust of stage 2 (index 1)
    update_stage_thrust(rocket_thrust, 1, 750.0);

    // Displaying updated thrust array
    printf("\nUpdated Thrust values for each stage:\n");
    for (int i = 0; i < NUM_STAGES; i++) {
        printf("Stage %d thrust: %.2f\n", i, rocket_thrust[i]);
    }

    // Recalculate total thrust after update
    total_thrust = compute_total_thrust(rocket_thrust, NUM_STAGES);
    printf("\nUpdated total rocket thrust: %.2f\n", total_thrust);

    return 0;
}
```

9.Pointers: Access stress values at various points.
Arrays: Store stress values for discrete wing sections.
Functions:
void compute_stress_distribution(const double *forces, double *stress, int size): Computes stress values based on applied forces.
void display_stress(const double *stress, int size): Displays the stress distribution.
Pass Arrays as Pointers: Pass stress arrays to computation functions.

```
#include <stdio.h>

// Function to compute stress distribution
void compute_stress_distribution(const double *forces, double *stress, int size) {
    // Assuming a simple linear relation between force and stress for illustration
    // Stress = Force / Area; we assume constant area for simplicity.
    // For now, let's just set stress as a function of the force at each section.
    const double area = 10.0; // Example constant area
    for (int i = 0; i < size; i++) {
        stress[i] = forces[i] / area; // Simple calculation for stress
    }
}

// Function to display stress distribution
void display_stress(const double *stress, int size) {
    printf("Stress Distribution: \n");
    for (int i = 0; i < size; i++) {
        printf("Section %d: Stress = %.2f\n", i+1, stress[i]);
    }
}
```

```
int main() {
    // Number of sections (discrete points on the wing)
    int size = 5;

    // Array to store applied forces at different sections (e.g., in Newtons)
    double forces[] = {100.0, 150.0, 200.0, 250.0, 300.0};

    // Array to store computed stress values (in Pascals)
    double stress[size];

    // Call the function to compute stress distribution
    compute_stress_distribution(forces, stress, size);

    // Call the function to display stress values
    display_stress(stress, size);

    return 0;
}
```

10.Drone Path Optimization
Pointers: Traverse waypoint arrays.
Arrays: Store coordinates of waypoints.
Functions:
double optimize_path(const double *waypoints, int size): Reduces the total path length.
void add_waypoint(double *waypoints, int *size, double x, double y): Adds a new waypoint.
Pass Arrays as Pointers: Use pointers to access and modify waypoints.

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

// Function to calculate the Euclidean distance between two waypoints
double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}

// Function to optimize the path by reordering the waypoints
double optimize_path(double *waypoints, int size) {
    double total_distance = 0.0;

    // A simple heuristic: calculate the total distance from start to end
    for (int i = 0; i < size - 1; i++) {
        total_distance += distance(waypoints[i * 2], waypoints[i * 2 + 1],
                                   waypoints[(i + 1) * 2], waypoints[(i + 1) * 2 + 1]);
    }

    return total_distance;
}

// Function to add a waypoint to the array
void add_waypoint(double *waypoints, int *size, double x, double y) {
    waypoints[*size * 2] = x;
    waypoints[*size * 2 + 1] = y;
    (*size)++;
}
```

```
// Function to print the waypoints for debugging
void print_waypoints(double *waypoints, int size) {
    for (int i = 0; i < size; i++) {
        printf("Waypoint %d: (%.2f, %.2f)\n", i + 1, waypoints[i * 2], waypoints[i * 2 + 1]);
    }
}

int main() {
    int size = 0;
    double waypoints[100 * 2];  // Array to store up to 100 waypoints, each with (x, y)

    // Adding some waypoints
    add_waypoint(waypoints, &size, 0.0, 0.0);
    add_waypoint(waypoints, &size, 1.0, 1.0);
    add_waypoint(waypoints, &size, 2.0, 2.0);
    add_waypoint(waypoints, &size, 3.0, 3.0);

    // Print current waypoints
    printf("Current Waypoints:\n");
    print_waypoints(waypoints, size);

    // Optimize the path
    double total_distance = optimize_path(waypoints, size);
    printf("\nTotal path length after optimization: %.2f\n", total_distance);

    return 0;
}
```

11. Satellite Attitude Control
Pointers: Manipulate quaternion arrays.
Arrays: Store quaternion values for attitude control.
Functions:
void update_attitude(const double *quaternion, double *new_attitude): Updates the satellite's attitude.
void normalize_quaternion(double *quaternion): Ensures quaternion normalization.
Pass Arrays as Pointers: Pass quaternion arrays as pointers.

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

// Function to normalize a quaternion
void normalize_quaternion(double *quaternion) {
    // Calculate the magnitude (norm) of the quaternion
    double norm = sqrt(quaternion[0] * quaternion[0] +
                       quaternion[1] * quaternion[1] +
                       quaternion[2] * quaternion[2] +
                       quaternion[3] * quaternion[3]);

    // Avoid division by zero (in case of very small values)
    if (norm > 0.0) {
        // Normalize the quaternion by dividing each component by the norm
        quaternion[0] /= norm;
        quaternion[1] /= norm;
        quaternion[2] /= norm;
        quaternion[3] /= norm;
    } else {
        // If the quaternion is degenerate (norm is 0), reset it to the identity quaternion
```

```
        quaternion[0] = 1.0;
        quaternion[1] = 0.0;
        quaternion[2] = 0.0;
        quaternion[3] = 0.0;
    }
}

// Function to update satellite's attitude based on the quaternion
void update_attitude(const double *quaternion, double *new_attitude) {
    // For this example, we just copy the quaternion to the new attitude array.
    // In a real system, you would use the quaternion to perform matrix transformations or other operations.
    new_attitude[0] = quaternion[0];
    new_attitude[1] = quaternion[1];
    new_attitude[2] = quaternion[2];
    new_attitude[3] = quaternion[3];

    // Optionally, normalize the new attitude to ensure it's a valid quaternion
    normalize_quaternion(new_attitude);
}

int main() {
    // Example quaternion representing satellite's current attitude (initial)
    double quaternion[4] = {1.0, 0.0, 0.0, 0.0};

    // Array to store updated attitude
    double new_attitude[4];

    // Update the attitude using the quaternion
    update_attitude(quaternion, new_attitude);

    // Print the updated attitude
    printf("Updated Attitude: [%f, %f, %f, %f]\n", new_attitude[0], new_attitude[1], new_attitude[2],
new_attitude[3]);

    return 0;
}
```

12. Aerospace Material Thermal Analysis
Pointers: Access temperature arrays for computation.
Arrays: Store temperature values at discrete points.
Functions:
void simulate_heat_transfer(const double *material_properties, double *temperatures, int size): Simulates heat transfer across the material.
void display_temperatures(const double *temperatures, int size): Outputs temperature distribution.
Pass Arrays as Pointers: Use pointers for temperature arrays

```
#include <stdio.h>

#define SIZE 10  // Number of temperature points (can adjust as needed)

// Function to simulate heat transfer across the material
void simulate_heat_transfer(const double *material_properties, double *temperatures, int size) {
    // Assume the first index holds initial conditions (e.g., heat source at one end)
    double thermal_conductivity = material_properties[0];  // First property could be conductivity
    double heat_capacity = material_properties[1];         // Second property could be specific heat
```

```
    double density = material_properties[2];              // Third property could be density

    // A simple heat transfer simulation based on conduction
    // For simplicity, we will just do a basic numerical approach (not realistic for real-world simulation)
    for (int i = 1; i < size - 1; i++) {
        // Heat transfer equation (simple one-dimensional)
        temperatures[i] = temperatures[i] + thermal_conductivity * (temperatures[i - 1] - 2 * temperatures[i] +
temperatures[i + 1]) / heat_capacity;
    }
}

// Function to display temperature distribution
void display_temperatures(const double *temperatures, int size) {
    printf("Temperature Distribution:\n");
    for (int i = 0; i < size; i++) {
        printf("Point %d: %.2f\n", i, temperatures[i]);
    }
}

int main() {
    // Example material properties: {thermal conductivity, specific heat, density}
    double material_properties[3] = {100.0, 500.0, 7000.0};  // Example values

    // Array of temperatures at discrete points
    double temperatures[SIZE];

    // Initial conditions: Assume a linear temperature gradient
    for (int i = 0; i < SIZE; i++) {
        temperatures[i] = (double)i * 10.0;  // Simple linear temperature distribution
    }

    // Simulate heat transfer
    simulate_heat_transfer(material_properties, temperatures, SIZE);

    // Display the temperature distribution
    display_temperatures(temperatures, SIZE);

    return 0;
}
```

13. Aircraft Fuel Efficiency
Pointers: Traverse fuel consumption arrays.
Arrays: Store fuel consumption at different time intervals.
Functions:
double compute_efficiency(const double *fuel_data, int size): Calculates overall fuel efficiency.
void update_fuel_data(double *fuel_data, int interval, double consumption): Updates fuel data for a specific interval.
Pass Arrays as Pointers: Pass fuel data arrays as pointers.

```
#include <stdio.h>

#define MAX_INTERVALS 10 // Max number of intervals (can be adjusted)

// Function to compute overall fuel efficiency
double compute_efficiency(const double *fuel_data, int size) {
    double total_fuel = 0;
```

```
    // Traverse the fuel data array and sum the fuel consumption
    for (int i = 0; i < size; i++) {
        total_fuel += fuel_data[i];
    }

    // For the purpose of the example, let's assume efficiency is
    // total distance per total fuel, so we return the total fuel
    return total_fuel;
}

// Function to update the fuel data for a specific interval
void update_fuel_data(double *fuel_data, int interval, double consumption) {
    if (interval >= 0) {
        fuel_data[interval] = consumption;  // Update the fuel consumption for the specified interval
    } else {
        printf("Invalid interval!\n");
    }
}

int main() {
    // Array to store fuel consumption data (in liters or another unit)
    double fuel_data[MAX_INTERVALS] = {0};

    // Update fuel data for different intervals
    update_fuel_data(fuel_data, 0, 5.5);  // Interval 0, 5.5 liters consumed
    update_fuel_data(fuel_data, 1, 6.0);  // Interval 1, 6.0 liters consumed
    update_fuel_data(fuel_data, 2, 4.8);  // Interval 2, 4.8 liters consumed
    update_fuel_data(fuel_data, 3, 7.2);  // Interval 3, 7.2 liters consumed

    // Compute the overall fuel efficiency (just total consumption for simplicity)
    double total_fuel = compute_efficiency(fuel_data, MAX_INTERVALS);
    printf("Total fuel consumed: %.2f liters\n", total_fuel);

    return 0;
}
```

14. Satellite Communication Link Budget
Pointers: Handle parameter arrays for computation.
Arrays: Store communication parameters like power and losses.
Functions:
double compute_link_budget(const double *parameters, int size): Calculates the total link budget.
void update_parameters(double *parameters, int index, double value): Updates a specific parameter.
Pass Arrays as Pointers: Pass parameter arrays as pointers.

```
#include <stdio.h>

// Function to compute the link budget
// Parameters array will contain values like transmitter power, receiver gain, etc.
double compute_link_budget(const double *parameters, int size) {
    double link_budget = 0.0;

    // For the sake of this example, assume:
    // parameters[0] -> Transmitter Power (in dB)
    // parameters[1] -> Receiver Gain (in dB)
```

```
    // parameters[2] -> Path Loss (in dB)
    // parameters[3] -> Free space path loss (in dB)
    // parameters[4] -> Additional losses (in dB)

    // Link budget formula (simplified for this example):
    link_budget = parameters[0] + parameters[1] - parameters[2] - parameters[3] - parameters[4];

    return link_budget;
}

// Function to update a specific parameter in the array
// Takes an index and the new value to update the parameter
void update_parameters(double *parameters, int index, double value) {
    // Check if the index is valid
    if (index >= 0 && index < 5) { // Assume we have 5 parameters
        parameters[index] = value;
    } else {
        printf("Error: Index out of range.\n");
    }
}

int main() {
    // Array to store the communication parameters
    // Example: [Transmitter Power, Receiver Gain, Path Loss, Free Space Path Loss, Additional Losses]
    double parameters[5] = {50.0, 40.0, 150.0, 200.0, 10.0}; // dB values

    // Compute the initial link budget
    double initial_link_budget = compute_link_budget(parameters, 5);
    printf("Initial Link Budget: %.2f dB\n", initial_link_budget);

    // Update a parameter (for example, update Transmitter Power)
    update_parameters(parameters, 0, 55.0); // New Transmitter Power

    // Compute the link budget after the update
    double updated_link_budget = compute_link_budget(parameters, 5);
    printf("Updated Link Budget: %.2f dB\n", updated_link_budget);

    return 0;
}
```

15. Turbulence Detection in Aircraft
Pointers: Traverse acceleration arrays.
Arrays: Store acceleration data from sensors.
Functions:
void detect_turbulence(const double *accelerations, int size, double *output): Detects turbulence based on frequency analysis.
void log_turbulence(double *turbulence_log, const double *detection_output, int size): Logs detected turbulence events.
Pass Arrays as Pointers: Pass acceleration and log arrays to functions.

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

// Function to perform frequency analysis and detect turbulence
void detect_turbulence(const double *accelerations, int size, double *output) {
```

```c
    // Placeholder: Here we assume that turbulence is detected if the acceleration
    // exceeds a certain threshold (e.g., 1.5g). In a real-world application, frequency
    // analysis or more advanced algorithms would be used to detect turbulence.
    for (int i = 0; i < size; i++) {
        if (accelerations[i] > 1.5) { // Threshold for turbulence detection
            output[i] = 1.0; // Mark turbulence detected
        } else {
            output[i] = 0.0; // No turbulence
        }
    }
}

// Function to log the turbulence detection results
void log_turbulence(double *turbulence_log, const double *detection_output, int size) {
    for (int i = 0; i < size; i++) {
        if (detection_output[i] == 1.0) {
            turbulence_log[i] = 1.0; // Log turbulence event
        } else {
            turbulence_log[i] = 0.0; // No event
        }
    }
}

int main() {
    // Example data: acceleration readings from the aircraft sensors (in g)
    double accelerations[] = {0.9, 1.2, 1.6, 0.8, 2.0, 1.1, 1.7, 0.5};
    int size = sizeof(accelerations) / sizeof(accelerations[0]);

    // Output arrays
    double detection_output[size];
    double turbulence_log[size];

    // Detect turbulence based on acceleration data
    detect_turbulence(accelerations, size, detection_output);

    // Log the turbulence events
    log_turbulence(turbulence_log, detection_output, size);

    // Print results
    printf("Turbulence Detection Log:\n");
    for (int i = 0; i < size; i++) {
        if (turbulence_log[i] == 1.0) {
            printf("Turbulence detected at index %d\n", i);
        } else {
            printf("No turbulence at index %d\n", i);
        }
    }

    return 0;
}
```