```c
/****************************************************************************
****

1. Particle Motion Simulator
Description:
Simulate the motion of particles in a two-dimensional space under the influence of forces.
Specifications:
Structure: Represents particle properties (mass, position, velocity).
Array: Stores the position and velocity vectors of multiple particles.
Union: Handles force types (gravitational, electric, or magnetic).
Strings: Define force types applied to particles.
const Pointers: Protect particle properties.
Double Pointers: Dynamically allocate memory for the particle system.

****************************************************************************
****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>


typedef struct {
    const char *particleID;
    float mass;
    float position[2];
    float velocity[2];
} Particle;
```

```c
typedef union {
    float gravitational;
    float electric;
    float magnetic;
} ForceType;


typedef struct {
    Particle particle;
    ForceType force;
    int forceType;
    char forceTypeName[20];
} ParticleSystem;


void addParticle(ParticleSystem **system, int *size, int *capacity, const char *particleID,
float mass, float *position, float *velocity, ForceType force, int forceType, const char
*forceTypeName);
void updateParticlePositions(ParticleSystem **system, int size, float deltaTime);
void displayParticleSystem(ParticleSystem **system, int size);

int main() {
    int size = 0, capacity = 2;
    ParticleSystem *system = (ParticleSystem *)malloc(capacity * sizeof(ParticleSystem));

    // Define some particles
    float position1[] = {0.0, 0.0};
    float velocity1[] = {1.0, 1.5};
```

```
    ForceType force1;
    force1.gravitational = 9.8;

    float position2[] = {2.0, 3.0};
    float velocity2[] = {0.5, 0.7};
    ForceType force2;
    force2.electric = 1.6e-19;

    // Add particles to the system
    addParticle(&system, &size, &capacity, "P001", 1.0, position1, velocity1, force1, 0,
"Gravitational");
    addParticle(&system, &size, &capacity, "P002", 2.0, position2, velocity2, force2, 1,
"Electric");

    // Update particle positions
    float deltaTime = 0.01; // time step in seconds
    updateParticlePositions(&system, size, deltaTime);

    // Display the particle system
    displayParticleSystem(&system, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)system[i].particle.particleID);
    }
    free(system);

    return 0;
}
```

```
void addParticle(ParticleSystem **system, int *size, int *capacity, const char *particleID,
float mass, float *position, float *velocity, ForceType force, int forceType, const char
*forceTypeName) {

    if (*size == *capacity) {
        *capacity *= 2;
        ParticleSystem *newSystem = (ParticleSystem *)malloc(*capacity *
sizeof(ParticleSystem));


        for (int i = 0; i < *size; i++) {
            newSystem[i] = (*system)[i];
        }


        free(*system);
        *system = newSystem;
    }


    (*system)[*size].particle.particleID = strdup(particleID);
    (*system)[*size].particle.mass = mass;
    (*system)[*size].particle.position[0] = position[0];
    (*system)[*size].particle.position[1] = position[1];
    (*system)[*size].particle.velocity[0] = velocity[0];
    (*system)[*size].particle.velocity[1] = velocity[1];
```

```
    (*system)[*size].force = force;
    (*system)[*size].forceType = forceType;
    strcpy((*system)[*size].forceTypeName, forceTypeName);

    (*size)++;
}


void updateParticlePositions(ParticleSystem **system, int size, float deltaTime) {
    for (int i = 0; i < size; i++) {
        (*system)[i].particle.position[0] += (*system)[i].particle.velocity[0] * deltaTime;
        (*system)[i].particle.position[1] += (*system)[i].particle.velocity[1] * deltaTime;
    }
}


void displayParticleSystem(ParticleSystem **system, int size) {
    printf("Particle System:\n");
    for (int i = 0; i < size; i++) {
        printf("Particle ID: %s\n", (*system)[i].particle.particleID);
        printf("Mass: %.2f kg\n", (*system)[i].particle.mass);
        printf("Position: (%.2f, %.2f)\n", (*system)[i].particle.position[0],
(*system)[i].particle.position[1]);
        printf("Velocity: (%.2f, %.2f)\n", (*system)[i].particle.velocity[0],
(*system)[i].particle.velocity[1]);
        printf("Force Type: %s\n", (*system)[i].forceTypeName);
        switch ((*system)[i].forceType) {
            case 0:
                printf("Gravitational Force: %.2f N\n", (*system)[i].force.gravitational);
```

```
            break;
          case 1:
            printf("Electric Force: %.2e N\n", (*system)[i].force.electric);
            break;
          case 2:
            printf("Magnetic Force: %.2e T\n", (*system)[i].force.magnetic);
            break;
        }
        printf("\n");
    }
}
```

/**

2. Electromagnetic Field Calculator

Description:

Calculate the electromagnetic field intensity at various points in space.

Specifications:

Structure: Stores field parameters (electric field, magnetic field, and position).

Array: Holds field values at discrete points.

Union: Represents either electric or magnetic field components.

Strings: Represent coordinate systems (Cartesian, cylindrical, spherical).

const Pointers: Prevent modification of field parameters.

Double Pointers: Manage memory for field grid allocation dynamically.

**/

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


typedef struct {
   const char *pointID;
   float position[3];
   union {
      float electricField[3];
      float magneticField[3];
   } fieldComponents;
   int isElectricField;
   char coordinateSystem[20];
} FieldParameters;


void addFieldPoint(FieldParameters **fields, int *size, int *capacity, const char *pointID, float *position, float *fieldComponents, int isElectricField, const char *coordinateSystem);
void displayFieldPoints(FieldParameters **fields, int size);

int main() {
   int size = 0, capacity = 2;
   FieldParameters *fields = (FieldParameters *)malloc(capacity * sizeof(FieldParameters));


   float position1[] = {0.0, 0.0, 0.0};
```

```
    float electricField1[] = {1.0, 2.0, 3.0};
    addFieldPoint(&fields, &size, &capacity, "P001", position1, electricField1, 1, "Cartesian");

    float position2[] = {1.0, 1.0, 1.0};
    float magneticField1[] = {0.1, 0.2, 0.3};
    addFieldPoint(&fields, &size, &capacity, "P002", position2, magneticField1, 0,
"Cylindrical");

    displayFieldPoints(&fields, size);



    for (int i = 0; i < size; i++) {
        free((void *)fields[i].pointID);
    }
    free(fields);

    return 0;
}



void addFieldPoint(FieldParameters **fields, int *size, int *capacity, const char *pointID,
float *position, float *fieldComponents, int isElectricField, const char *coordinateSystem) {

    if (*size == *capacity) {
        *capacity *= 2;
        FieldParameters *newFields = (FieldParameters *)malloc(*capacity *
sizeof(FieldParameters));
```

```
    for (int i = 0; i < *size; i++) {
        newFields[i] = (*fields)[i];
    }


    free(*fields);
    *fields = newFields;
}


(*fields)[*size].pointID = strdup(pointID);
(*fields)[*size].position[0] = position[0];
(*fields)[*size].position[1] = position[1];
(*fields)[*size].position[2] = position[2];

if (isElectricField) {
    (*fields)[*size].fieldComponents.electricField[0] = fieldComponents[0];
    (*fields)[*size].fieldComponents.electricField[1] = fieldComponents[1];
    (*fields)[*size].fieldComponents.electricField[2] = fieldComponents[2];
} else {
    (*fields)[*size].fieldComponents.magneticField[0] = fieldComponents[0];
    (*fields)[*size].fieldComponents.magneticField[1] = fieldComponents[1];
    (*fields)[*size].fieldComponents.magneticField[2] = fieldComponents[2];
}

(*fields)[*size].isElectricField = isElectricField;
strcpy((*fields)[*size].coordinateSystem, coordinateSystem);
```

```
    (*size)++;
}



void displayFieldPoints(FieldParameters **fields, int size) {
    printf("Electromagnetic Field Points:\n");
    for (int i = 0; i < size; i++) {
        printf("Point ID: %s\n", (*fields)[i].pointID);
        printf("Position: (%.2f, %.2f, %.2f)\n", (*fields)[i].position[0], (*fields)[i].position[1], (*fields)[i].position[2]);
        printf("Coordinate System: %s\n", (*fields)[i].coordinateSystem);
        if ((*fields)[i].isElectricField) {
            printf("Electric Field: (%.2f, %.2f, %.2f) V/m\n", (*fields)[i].fieldComponents.electricField[0], (*fields)[i].fieldComponents.electricField[1], (*fields)[i].fieldComponents.electricField[2]);
        } else {
            printf("Magnetic Field: (%.2f, %.2f, %.2f) T\n", (*fields)[i].fieldComponents.magneticField[0], (*fields)[i].fieldComponents.magneticField[1], (*fields)[i].fieldComponents.magneticField[2]);
        }
        printf("\n");
    }
}
```

```
/**************************************************************************
****

3. Atomic Energy Level Tracker

Description:

Track the energy levels of atoms and the transitions between them.

Specifications:

Structure: Contains atomic details (element name, energy levels, and transition probabilities).

Array: Stores energy levels for different atoms.

Union: Represents different energy states.

Strings: Represent element names.

const Pointers: Protect atomic data.

Double Pointers: Allocate memory for dynamically adding new elements.

**************************************************************************
****/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>



typedef struct {

   const char *elementName;

   float energyLevels[10];

   float transitionProbabilities[10];

   int numLevels;

} AtomicDetails;
```

```c
typedef union {
    float groundState;
    float excitedState;
} EnergyState;


typedef struct {
    AtomicDetails atomicDetails;
    EnergyState energyState;
    int isGroundState;
} AtomicEnergyLevelRecord;


void addAtomicEnergyLevelRecord(AtomicEnergyLevelRecord **records, int *size, int
*capacity, const char *elementName, float *energyLevels, float *transitionProbabilities, int
numLevels, EnergyState energyState, int isGroundState);
void displayAtomicEnergyLevelRecords(AtomicEnergyLevelRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    AtomicEnergyLevelRecord *records = (AtomicEnergyLevelRecord *)malloc(capacity *
sizeof(AtomicEnergyLevelRecord));


    float energyLevels1[] = {0.0, 1.0, 2.5, 4.0};
    float transitionProbabilities1[] = {0.0, 0.2, 0.15, 0.1};
```

```
    float energyLevels2[] = {0.0, 0.8, 1.7, 3.0};
    float transitionProbabilities2[] = {0.0, 0.25, 0.18, 0.12};


    EnergyState state1;
    state1.groundState = 0.0;

    EnergyState state2;
    state2.excitedState = 3.0;


    addAtomicEnergyLevelRecord(&records, &size, &capacity, "Hydrogen", energyLevels1, transitionProbabilities1, 4, state1, 1);
    addAtomicEnergyLevelRecord(&records, &size, &capacity, "Helium", energyLevels2, transitionProbabilities2, 4, state2, 0);

    displayAtomicEnergyLevelRecords(&records, size);


    for (int i = 0; i < size; i++) {
        free((void *)records[i].atomicDetails.elementName);
    }
    free(records);

    return 0;
}
```

```c
void addAtomicEnergyLevelRecord(AtomicEnergyLevelRecord **records, int *size, int
*capacity, const char *elementName, float *energyLevels, float *transitionProbabilities, int
numLevels, EnergyState energyState, int isGroundState) {

    if (*size == *capacity) {
        *capacity *= 2;
        AtomicEnergyLevelRecord *newRecords = (AtomicEnergyLevelRecord
*)malloc(*capacity * sizeof(AtomicEnergyLevelRecord));


        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }


        free(*records);
        *records = newRecords;
    }


    (*records)[*size].atomicDetails.elementName = strdup(elementName);
    (*records)[*size].atomicDetails.numLevels = numLevels;


    for (int i = 0; i < numLevels; i++) {
        (*records)[*size].atomicDetails.energyLevels[i] = energyLevels[i];
        (*records)[*size].atomicDetails.transitionProbabilities[i] = transitionProbabilities[i];
    }
```

```
    (*records)[*size].energyState = energyState;
    (*records)[*size].isGroundState = isGroundState;



    (*size)++;
}

void displayAtomicEnergyLevelRecords(AtomicEnergyLevelRecord **records, int size) {
    printf("Atomic Energy Level Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Element Name: %s\n", (*records)[i].atomicDetails.elementName);
        printf("Energy Levels: ");
        for (int j = 0; j < (*records)[i].atomicDetails.numLevels; j++) {
            printf("%.2f ", (*records)[i].atomicDetails.energyLevels[j]);
        }
        printf("\nTransition Probabilities: ");
        for (int j = 0; j < (*records)[i].atomicDetails.numLevels; j++) {
            printf("%.2f ", (*records)[i].atomicDetails.transitionProbabilities[j]);
        }
        printf("\nEnergy State: ");
        if ((*records)[i].isGroundState) {
            printf("Ground State: %.2f eV\n", (*records)[i].energyState.groundState);
        } else {
            printf("Excited State: %.2f eV\n", (*records)[i].energyState.excitedState);
        }
        printf("\n\n");
    }
}
```

```c
/**************************************************************************
****

4. Quantum State Representation System

Description:

Develop a program to represent quantum states and their evolution over time.

Specifications:

Structure: Holds state properties (wavefunction amplitude, phase, and energy).

Array: Represents the wavefunction across multiple points.

Union: Stores amplitude or phase information.

Strings: Describe state labels (e.g., "ground state," "excited state").

const Pointers: Protect state properties.

Double Pointers: Manage quantum states dynamically.

***************************************************************************
****/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>



typedef struct {

   const char *stateLabel;

   float wavefunctionAmplitude[100];

   float energy;

} QuantumState;



typedef union {
```

```
    float amplitude;
    float phase;
} StateProperties;


typedef struct {
    QuantumState state;
    StateProperties properties;
    int isAmplitude;
} QuantumStateRecord;


void addQuantumState(QuantumStateRecord **records, int *size, int *capacity, const char
*stateLabel, float *wavefunctionAmplitude, float energy, StateProperties properties, int
isAmplitude);
void displayQuantumStates(QuantumStateRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    QuantumStateRecord *records = (QuantumStateRecord *)malloc(capacity *
sizeof(QuantumStateRecord));

    // Define some wavefunction amplitudes
    float wavefunctionAmplitude1[] = {0.0, 0.1, 0.2, 0.3, 0.4};
    float wavefunctionAmplitude2[] = {0.0, 0.2, 0.4, 0.6, 0.8};

    // Define some state properties
    StateProperties properties1;
    properties1.amplitude = 0.5;
```

```
    StateProperties properties2;
    properties2.phase = 0.25;

    // Add quantum state records
    addQuantumState(&records, &size, &capacity, "Ground State", wavefunctionAmplitude1, 1.0, properties1, 1);
    addQuantumState(&records, &size, &capacity, "Excited State", wavefunctionAmplitude2, 2.0, properties2, 0);

    displayQuantumStates(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].state.stateLabel); // Cast to void* to free const char*
    }
    free(records);

    return 0;
}

// Function to add a quantum state to the list
void addQuantumState(QuantumStateRecord **records, int *size, int *capacity, const char *stateLabel, float *wavefunctionAmplitude, float energy, StateProperties properties, int isAmplitude) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
```

```
        QuantumStateRecord *newRecords = (QuantumStateRecord *)malloc(*capacity *
sizeof(QuantumStateRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }

    // Initialize the new quantum state
    (*records)[*size].state.stateLabel = strdup(stateLabel);
    (*records)[*size].state.energy = energy;

    // Copy wavefunction amplitudes
    for (int i = 0; i < 100; i++) {
        (*records)[*size].state.wavefunctionAmplitude[i] = (i < 5) ? wavefunctionAmplitude[i] :
0.0;
    }

    (*records)[*size].properties = properties;
    (*records)[*size].isAmplitude = isAmplitude;

    // Increment the size
    (*size)++;
}
```

```c
// Function to display the quantum states
void displayQuantumStates(QuantumStateRecord **records, int size) {
    printf("Quantum State Records:\n");
    for (int i = 0; i < size; i++) {
        printf("State Label: %s\n", (*records)[i].state.stateLabel);
        printf("Energy: %.2f eV\n", (*records)[i].state.energy);
        printf("Wavefunction Amplitudes: ");
        for (int j = 0; j < 5; j++) {
            printf("%.2f ", (*records)[i].state.wavefunctionAmplitude[j]);
        }
        printf("\nProperty: ");
        if ((*records)[i].isAmplitude) {
            printf("Amplitude: %.2f\n", (*records)[i].properties.amplitude);
        } else {
            printf("Phase: %.2f\n", (*records)[i].properties.phase);
        }
        printf("\n\n");
    }
}
```

```c
/***************************************************************************
****

5. Optics Simulation Tool

Description:

Simulate light rays passing through different optical elements.

Specifications:

Structure: Represents optical properties (refractive index, focal length).

Array: Stores light ray paths.

Union: Handles lens or mirror parameters.

Strings: Represent optical element types.

const Pointers: Protect optical properties.

Double Pointers: Manage arrays of optical elements dynamically.

****************************************************************************
****/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>



typedef struct {

    const char *elementID;

    char elementType[20];

    float refractiveIndex;

    float focalLength;

} OpticalProperties;
```

```
typedef union {
    struct {
        float curvatureRadius;
    } lens;
    struct {
        float curvatureRadius;
        float reflectivity;
    } mirror;
} OpticalParameters;


typedef struct {
    OpticalProperties properties;
    OpticalParameters parameters;
    int isLens;
} OpticalElement;


void addOpticalElement(OpticalElement **elements, int *size, int *capacity, const char
*elementID, const char *elementType, float refractiveIndex, float focalLength,
OpticalParameters parameters, int isLens);
void simulateLightRayPaths(OpticalElement **elements, int size);
void displayOpticalElements(OpticalElement **elements, int size);

int main() {
    int size = 0, capacity = 2;
    OpticalElement *elements = (OpticalElement *)malloc(capacity * sizeof(OpticalElement));
```

```
    OpticalParameters lensParameters;
    lensParameters.lens.curvatureRadius = 10.0;

    OpticalParameters mirrorParameters;
    mirrorParameters.mirror.curvatureRadius = 20.0;
    mirrorParameters.mirror.reflectivity = 0.9;


    addOpticalElement(&elements, &size, &capacity, "E001", "Lens", 1.5, 50.0,
lensParameters, 1);
    addOpticalElement(&elements, &size, &capacity, "E002", "Mirror", 0.0, 100.0,
mirrorParameters, 0);


    simulateLightRayPaths(&elements, size);


    displayOpticalElements(&elements, size);


    for (int i = 0; i < size; i++) {
        free((void *)elements[i].properties.elementID);
    }
    free(elements);

    return 0;
}
```

```c
void addOpticalElement(OpticalElement **elements, int *size, int *capacity, const char
*elementID, const char *elementType, float refractiveIndex, float focalLength,
OpticalParameters parameters, int isLens) {

    if (*size == *capacity) {
        *capacity *= 2;
        OpticalElement *newElements = (OpticalElement *)malloc(*capacity *
sizeof(OpticalElement));

        for (int i = 0; i < *size; i++) {
            newElements[i] = (*elements)[i];
        }

        free(*elements);
        *elements = newElements;
    }

    (*elements)[*size].properties.elementID = strdup(elementID);
    strcpy((*elements)[*size].properties.elementType, elementType);
    (*elements)[*size].properties.refractiveIndex = refractiveIndex;
    (*elements)[*size].properties.focalLength = focalLength;

    (*elements)[*size].parameters = parameters;
    (*elements)[*size].isLens = isLens;
```

```
    (*size)++;
}


void simulateLightRayPaths(OpticalElement **elements, int size) {

    printf("Simulating light ray paths through the optical elements...\n");
}

// Function to display the optical elements
void displayOpticalElements(OpticalElement **elements, int size) {
    printf("Optical Elements:\n");
    for (int i = 0; i < size; i++) {
        printf("Element ID: %s\n", (*elements)[i].properties.elementID);
        printf("Element Type: %s\n", (*elements)[i].properties.elementType);
        if ((*elements)[i].isLens) {
            printf("Lens Curvature Radius: %.2f mm\n",
(*elements)[i].parameters.lens.curvatureRadius);
        } else {
            printf("Mirror Curvature Radius: %.2f mm\n",
(*elements)[i].parameters.mirror.curvatureRadius);
            printf("Mirror Reflectivity: %.2f\n", (*elements)[i].parameters.mirror.reflectivity);
        }
        printf("Refractive Index: %.2f\n", (*elements)[i].properties.refractiveIndex);
        printf("Focal Length: %.2f mm\n", (*elements)[i].properties.focalLength);
        printf("\n");
    }
}
```

```
/**************************************************************************
****

6. Thermodynamics State Calculator

Description:

Calculate thermodynamic states of a system based on input parameters like pressure,
volume, and temperature.

Specifications:

Structure: Represents thermodynamic properties (P, V, T, and entropy).

Array: Stores states over a range of conditions.

Union: Handles dependent properties like energy or entropy.

Strings: Represent state descriptions.

const Pointers: Protect thermodynamic data.

Double Pointers: Allocate state data dynamically for simulation.

***************************************************************************
****/
#include <stdio.h>

#include <stdlib.h>

#include <string.h>



typedef struct {

   const char *stateDescription;

   float pressure;

   float volume;

   float temperature;

   float entropy;

} ThermodynamicState;
```

```
typedef union {
    float energy;
    float entropy;
} DependentProperties;


typedef struct {
    ThermodynamicState state;
    DependentProperties properties;
    int isEnergy;
} ThermodynamicStateRecord;


void addThermodynamicState(ThermodynamicStateRecord **records, int *size, int
*capacity, const char *stateDescription, float pressure, float volume, float temperature, float
entropy, DependentProperties properties, int isEnergy);
void displayThermodynamicStates(ThermodynamicStateRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    ThermodynamicStateRecord *records = (ThermodynamicStateRecord *)malloc(capacity *
sizeof(ThermodynamicStateRecord));

    DependentProperties properties1;
    properties1.energy = 500.0; // in J

    DependentProperties properties2;
```

```
    properties2.entropy = 300.0; // in J/K



    addThermodynamicState(&records, &size, &capacity, "State 1", 101325.0, 1.0, 300.0, 
250.0, properties1, 1);
    addThermodynamicState(&records, &size, &capacity, "State 2", 202650.0, 0.5, 350.0, 
270.0, properties2, 0);

    displayThermodynamicStates(&records, size);



    for (int i = 0; i < size; i++) {
        free((void *)records[i].state.stateDescription);
    }
    free(records);

    return 0;
}



void addThermodynamicState(ThermodynamicStateRecord **records, int *size, int 
*capacity, const char *stateDescription, float pressure, float volume, float temperature, float 
entropy, DependentProperties properties, int isEnergy) {

    if (*size == *capacity) {
        *capacity *= 2;
        ThermodynamicStateRecord *newRecords = (ThermodynamicStateRecord 
*)malloc(*capacity * sizeof(ThermodynamicStateRecord));
```

```
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }


        free(*records);
        *records = newRecords;
    }


    (*records)[*size].state.stateDescription = strdup(stateDescription);
    (*records)[*size].state.pressure = pressure;
    (*records)[*size].state.volume = volume;
    (*records)[*size].state.temperature = temperature;
    (*records)[*size].state.entropy = entropy;

    (*records)[*size].properties = properties;
    (*records)[*size].isEnergy = isEnergy;


    (*size)++;
}


void displayThermodynamicStates(ThermodynamicStateRecord **records, int size) {
    printf("Thermodynamic State Records:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("State Description: %s\n", (*records)[i].state.stateDescription);
        printf("Pressure: %.2f Pa\n", (*records)[i].state.pressure);
        printf("Volume: %.2f m^3\n", (*records)[i].state.volume);
        printf("Temperature: %.2f K\n", (*records)[i].state.temperature);
        printf("Entropy: %.2f J/K\n", (*records)[i].state.entropy);
        printf("Dependent Property: ");
        if ((*records)[i].isEnergy) {
            printf("Energy: %.2f J\n", (*records)[i].properties.energy);
        } else {
            printf("Entropy: %.2f J/K\n", (*records)[i].properties.entropy);
        }
        printf("\n");
    }
}
```

/**
****

7. Nuclear Reaction Tracker

Description:

Track the parameters of nuclear reactions like fission and fusion processes.

Specifications:

Structure: Represents reaction details (reactants, products, energy released).

Array: Holds data for multiple reactions.

Union: Represents either energy release or product details.

Strings: Represent reactant and product names.

const Pointers: Protect reaction details.

```
Double Pointers: Dynamically allocate memory for reaction data.

*******************************************************************************
****/

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


typedef struct {
    const char *reactionID;
    char reactants[100];
    char products[100];
    float energyReleased; // in MeV
} ReactionDetails;


typedef union {
    float energyReleased;
    struct {
        char productDetails[100];
    } productInfo;
} ReactionProperties;


typedef struct {
```

```c
    ReactionDetails details;
    ReactionProperties properties;
    int isEnergyRelease;
} ReactionRecord;


void addReaction(ReactionRecord **records, int *size, int *capacity, const char *reactionID, const char *reactants, const char *products, float energyReleased, ReactionProperties properties, int isEnergyRelease);
void displayReactions(ReactionRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    ReactionRecord *records = (ReactionRecord *)malloc(capacity * sizeof(ReactionRecord));


    ReactionProperties properties1;
    properties1.energyReleased = 200.0;

    ReactionProperties properties2;
    strcpy(properties2.productInfo.productDetails, "2 Helium nuclei");


    addReaction(&records, &size, &capacity, "R001", "Uranium-235 + neutron", "Barium-141 + Krypton-92 + 3 neutrons", 200.0, properties1, 1);
    addReaction(&records, &size, &capacity, "R002", "Deuterium + Tritium", "Helium-4 + neutron", 17.6, properties2, 0);

    displayReactions(&records, size);
```

```
    for (int i = 0; i < size; i++) {
        free((void *)records[i].details.reactionID);
    }
    free(records);

    return 0;
}

void addReaction(ReactionRecord **records, int *size, int *capacity, const char *reactionID,
const char *reactants, const char *products, float energyReleased, ReactionProperties
properties, int isEnergyRelease) {

    if (*size == *capacity) {
        *capacity *= 2;
        ReactionRecord *newRecords = (ReactionRecord *)malloc(*capacity *
sizeof(ReactionRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }
```

```
    // Initialize the new reaction record
    (*records)[*size].details.reactionID = strdup(reactionID);
    strcpy((*records)[*size].details.reactants, reactants);
    strcpy((*records)[*size].details.products, products);
    (*records)[*size].details.energyReleased = energyReleased;

    (*records)[*size].properties = properties;
    (*records)[*size].isEnergyRelease = isEnergyRelease;

    // Increment the size
    (*size)++;
}

// Function to display the reaction records
void displayReactions(ReactionRecord **records, int size) {
    printf("Nuclear Reaction Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Reaction ID: %s\n", (*records)[i].details.reactionID);
        printf("Reactants: %s\n", (*records)[i].details.reactants);
        printf("Products: %s\n", (*records)[i].details.products);
        printf("Energy Released: %.2f MeV\n", (*records)[i].details.energyReleased);
        printf("Additional Properties: ");
        if ((*records)[i].isEnergyRelease) {
            printf("Energy Released: %.2f MeV\n", (*records)[i].properties.energyReleased);
        } else {
            printf("Product Details: %s\n", (*records)[i].properties.productInfo.productDetails);
        }
        printf("\n");
```

```
    }
}/*************************************************************************
****

8. Gravitational Field Simulation

Description:

Simulate the gravitational field of massive objects in a system.

Specifications:

Structure: Contains object properties (mass, position, field strength).

Array: Stores field values at different points.

Union: Handles either mass or field strength as parameters.

Strings: Represent object labels (e.g., "Planet A," "Star B").

const Pointers: Protect object properties.

Double Pointers: Dynamically allocate memory for gravitational field data.



**************************************************************************
****/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>



typedef struct {

    const char *objectLabel;

    float mass;

    float position[2];
```

```
    float fieldStrength;
} ObjectProperties;



typedef union {
    float mass;
    float fieldStrength;
} VariableProperties;



typedef struct {
    ObjectProperties properties;
    VariableProperties variables;
    int isMass;
} GravitationalFieldRecord;

// Function prototypes
void addObject(GravitationalFieldRecord **records, int *size, int *capacity, const char
*objectLabel, float mass, float *position, float fieldStrength, VariableProperties variables, int
isMass);
void calculateGravitationalField(GravitationalFieldRecord **records, int size, float *point);
void displayGravitationalFieldRecords(GravitationalFieldRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    GravitationalFieldRecord *records = (GravitationalFieldRecord *)malloc(capacity *
sizeof(GravitationalFieldRecord));
```

```
    float position1[] = {0.0, 0.0};
    float position2[] = {1.0, 2.0};

    VariableProperties variables1;
    variables1.mass = 5.97e24;

    VariableProperties variables2;
    variables2.fieldStrength = 9.8;



    addObject(&records, &size, &capacity, "Planet A", 5.97e24, position1, 9.8, variables1, 1);
    addObject(&records, &size, &capacity, "Object B", 100.0, position2, 0.0, variables2, 0);



    float point[] = {0.5, 0.5};
    calculateGravitationalField(&records, size, point);

    // Display the gravitational field records
    displayGravitationalFieldRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].properties.objectLabel);
    }
    free(records);

    return 0;
}
```

```c
void addObject(GravitationalFieldRecord **records, int *size, int *capacity, const char
*objectLabel, float mass, float *position, float fieldStrength, VariableProperties variables, int
isMass) {

    if (*size == *capacity) {
        *capacity *= 2;
        GravitationalFieldRecord *newRecords = (GravitationalFieldRecord *)malloc(*capacity *
sizeof(GravitationalFieldRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }

    // Initialize the new object
    (*records)[*size].properties.objectLabel = strdup(objectLabel);
    (*records)[*size].properties.mass = mass;
    (*records)[*size].properties.position[0] = position[0];
    (*records)[*size].properties.position[1] = position[1];
    (*records)[*size].properties.fieldStrength = fieldStrength;

    (*records)[*size].variables = variables;
```

```
    (*records)[*size].isMass = isMass;

    // Increment the size
    (*size)++;
}

// Function to calculate the gravitational field at a specific point
void calculateGravitationalField(GravitationalFieldRecord **records, int size, float *point) {
    const float G = 6.67430e-11; // gravitational constant in m^3 kg^-1 s^-2

    for (int i = 0; i < size; i++) {
        float dx = point[0] - (*records)[i].properties.position[0];
        float dy = point[1] - (*records)[i].properties.position[1];
        float distanceSquared = dx * dx + dy * dy;
        float distance = sqrt(distanceSquared);

        if ((*records)[i].isMass) {
            (*records)[i].properties.fieldStrength = (G * (*records)[i].properties.mass) /
distanceSquared;
        }

        printf("Gravitational field at point (%.2f, %.2f) due to %s: %.2e N/kg\n", point[0],
point[1], (*records)[i].properties.objectLabel, (*records)[i].properties.fieldStrength);
    }
}

// Function to display the gravitational field records
void displayGravitationalFieldRecords(GravitationalFieldRecord **records, int size) {
    printf("Gravitational Field Records:\n");
```

```
    for (int i = 0; i < size; i++) {
        printf("Object Label: %s\n", (*records)[i].properties.objectLabel);
        printf("Mass: %.2e kg\n", (*records)[i].properties.mass);
        printf("Position: (%.2f, %.2f)\n", (*records)[i].properties.position[0],
(*records)[i].properties.position[1]);
        if ((*records)[i].isMass) {
            printf("Gravitational Field Strength: %.2e N/kg\n",
(*records)[i].properties.fieldStrength);
        } else {
            printf("Field Strength: %.2e N/kg\n", (*records)[i].properties.fieldStrength);
        }
        printf("\n");
    }
}
```

/**

9. Wave Interference Analyzer

Description:

Analyze interference patterns produced by waves from multiple sources.

Specifications:

Structure: Represents wave properties (amplitude, wavelength, and phase).

Array: Stores wave interference data at discrete points.

Union: Handles either amplitude or phase information.

Strings: Represent wave source labels.

const Pointers: Protect wave properties.

Double Pointers: Manage dynamic allocation of wave sources.

```
***************************************************************************
****/

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


typedef struct {
   const char *sourceLabel;
   float amplitude;
   float wavelength;
   float phase;
} WaveProperties;


typedef union {
   float amplitude;
   float phase;
} WaveInfo;


typedef struct {
   WaveProperties properties;
   WaveInfo info;
   int isAmplitude;
} WaveSource;
```

```
void addWaveSource(WaveSource **sources, int *size, int *capacity, const char *sourceLabel, float amplitude, float wavelength, float phase, WaveInfo info, int isAmplitude);

void calculateInterferencePattern(WaveSource **sources, int size, float *points, int numPoints);

void displayWaveSources(WaveSource **sources, int size);


int main() {

    int size = 0, capacity = 2;

    WaveSource *sources = (WaveSource *)malloc(capacity * sizeof(WaveSource));



    WaveInfo info1;

    info1.amplitude = 1.0;


    WaveInfo info2;

    info2.phase = 0.0;



    addWaveSource(&sources, &size, &capacity, "Source A", 1.0, 0.5, 0.0, info1, 1);

    addWaveSource(&sources, &size, &capacity, "Source B", 1.5, 0.4, 1.57, info2, 0);



    float points[] = {0.0, 0.5, 1.0, 1.5, 2.0};

    int numPoints = sizeof(points) / sizeof(points[0]);


    calculateInterferencePattern(&sources, size, points, numPoints);
```

```
    displayWaveSources(&sources, size);



    for (int i = 0; i < size; i++) {
        free((void *)sources[i].properties.sourceLabel);
    }
    free(sources);

    return 0;
}


void addWaveSource(WaveSource **sources, int *size, int *capacity, const char
*sourceLabel, float amplitude, float wavelength, float phase, WaveInfo info, int isAmplitude)
{

    if (*size == *capacity) {
        *capacity *= 2;
        WaveSource *newSources = (WaveSource *)malloc(*capacity * sizeof(WaveSource));



        for (int i = 0; i < *size; i++) {
            newSources[i] = (*sources)[i];
        }



        free(*sources);
        *sources = newSources;
    }
```

```
    (*sources)[*size].properties.sourceLabel = strdup(sourceLabel);
    (*sources)[*size].properties.amplitude = amplitude;
    (*sources)[*size].properties.wavelength = wavelength;
    (*sources)[*size].properties.phase = phase;

    (*sources)[*size].info = info;
    (*sources)[*size].isAmplitude = isAmplitude;



    (*size)++;
}



void calculateInterferencePattern(WaveSource **sources, int size, float *points, int numPoints) {
    printf("Interference Pattern:\n");
    for (int i = 0; i < numPoints; i++) {
        float totalAmplitude = 0.0;
        for (int j = 0; j < size; j++) {
            float waveComponent = (*sources)[j].properties.amplitude * cos(2 * M_PI * points[i] / (*sources)[j].properties.wavelength + (*sources)[j].properties.phase);
            totalAmplitude += waveComponent;
        }
        printf("Point %.2f: Total Amplitude = %.2f\n", points[i], totalAmplitude);
    }
}
```

```
void displayWaveSources(WaveSource **sources, int size) {
    printf("Wave Sources:\n");
    for (int i = 0; i < size; i++) {
        printf("Source Label: %s\n", (*sources)[i].properties.sourceLabel);
        printf("Amplitude: %.2f\n", (*sources)[i].properties.amplitude);
        printf("Wavelength: %.2f\n", (*sources)[i].properties.wavelength);
        printf("Phase: %.2f\n", (*sources)[i].properties.phase);
        printf("\n");
    }
}
```

```
/***********************************************************************
****
10. Magnetic Material Property Database
Description:
Create a database to store and retrieve properties of magnetic materials.
Specifications:
Structure: Represents material properties (permeability, saturation).
Array: Stores data for multiple materials.
Union: Handles temperature-dependent properties.
Strings: Represent material names.
const Pointers: Protect material data.
Double Pointers: Allocate material records dynamically.

***********************************************************************
****/
```

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


typedef struct {
    const char *materialName;
    float permeability;
    float saturation;
} MaterialProperties;


typedef union {
    float temperature;
    struct {
        float lowTempValue;
        float highTempValue;
    } tempRange;
} TempDependentProperties;


typedef struct {
    MaterialProperties properties;
    TempDependentProperties tempProperties;
    int isSingleTemp;
} MaterialRecord;
```

```
void addMaterialRecord(MaterialRecord **records, int *size, int *capacity, const char
*materialName, float permeability, float saturation, TempDependentProperties
tempProperties, int isSingleTemp);

void displayMaterialRecords(MaterialRecord **records, int size);


int main() {

    int size = 0, capacity = 2;

    MaterialRecord *records = (MaterialRecord *)malloc(capacity * sizeof(MaterialRecord));



    TempDependentProperties tempProperties1;

    tempProperties1.temperature = 25.0;


    TempDependentProperties tempProperties2;

    tempProperties2.tempRange.lowTempValue = -20.0;

    tempProperties2.tempRange.highTempValue = 150.0;


    // Add material records

    addMaterialRecord(&records, &size, &capacity, "Material A", 4.0e-7, 1.2,
tempProperties1, 1);

    addMaterialRecord(&records, &size, &capacity, "Material B", 3.5e-7, 1.8,
tempProperties2, 0);


    displayMaterialRecords(&records, size);


    // Free allocated memory

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

        free((void *)records[i].properties.materialName);

    }
```

```
    free(records);

    return 0;
}

void addMaterialRecord(MaterialRecord **records, int *size, int *capacity, const char
*materialName, float permeability, float saturation, TempDependentProperties
tempProperties, int isSingleTemp) {

    if (*size == *capacity) {
        *capacity *= 2;
        MaterialRecord *newRecords = (MaterialRecord *)malloc(*capacity *
sizeof(MaterialRecord));

        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        free(*records);
        *records = newRecords;
    }

    (*records)[*size].properties.materialName = strdup(materialName);
    (*records)[*size].properties.permeability = permeability;
    (*records)[*size].properties.saturation = saturation;
```

```
    (*records)[*size].tempProperties = tempProperties;
    (*records)[*size].isSingleTemp = isSingleTemp;



    (*size)++;
}



void displayMaterialRecords(MaterialRecord **records, int size) {
    printf("Magnetic Material Property Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Material Name: %s\n", (*records)[i].properties.materialName);
        printf("Permeability: %.2e H/m\n", (*records)[i].properties.permeability);
        printf("Saturation: %.2f T\n", (*records)[i].properties.saturation);
        if ((*records)[i].isSingleTemp) {
            printf("Temperature: %.2f °C\n", (*records)[i].tempProperties.temperature);
        } else {
            printf("Temperature Range: %.2f to %.2f °C\n",
(*records)[i].tempProperties.tempRange.lowTempValue,
(*records)[i].tempProperties.tempRange.highTempValue);
        }
        printf("\n");
    }
}
```

```c
/***************************************************************************
****
11. Plasma Dynamics Simulator
Description:
Simulate the behavior of plasma under various conditions.
Specifications:
Structure: Represents plasma parameters (density, temperature, and electric field).
Array: Stores simulation results.
Union: Handles either density or temperature data.
Strings: Represent plasma types.
const Pointers: Protect plasma parameters.
Double Pointers: Manage dynamic allocation for simulation data.

****************************************************************************
****/

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

typedef struct {
    const char *plasmaType;
    float density;
    float temperature;
    float electricField;
} PlasmaParameters;


typedef union {
```

```
    float density;
    float temperature;
} VariableProperties;


typedef struct {
    PlasmaParameters parameters;
    VariableProperties properties;
    int isDensity;
} PlasmaSimulationRecord;


void addPlasmaSimulationRecord(PlasmaSimulationRecord **records, int *size, int
*capacity, const char *plasmaType, float density, float temperature, float electricField,
VariableProperties properties, int isDensity);
void simulatePlasmaBehavior(PlasmaSimulationRecord **records, int size);
void displayPlasmaSimulationRecords(PlasmaSimulationRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    PlasmaSimulationRecord *records = (PlasmaSimulationRecord *)malloc(capacity *
sizeof(PlasmaSimulationRecord));


    VariableProperties properties1;
    properties1.density = 1.0e20;

    VariableProperties properties2;
    properties2.temperature = 1.5e6; // in Kelvin
```

```
    // Add plasma simulation records
    addPlasmaSimulationRecord(&records, &size, &capacity, "Plasma Type A", 1.0e20, 1.0e6, 
100.0, properties1, 1);
    addPlasmaSimulationRecord(&records, &size, &capacity, "Plasma Type B", 2.0e20, 1.5e6, 
200.0, properties2, 0);

    // Simulate plasma behavior
    simulatePlasmaBehavior(&records, size);

    // Display the plasma simulation records
    displayPlasmaSimulationRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].parameters.plasmaType);
    }
    free(records);

    return 0;
}


void addPlasmaSimulationRecord(PlasmaSimulationRecord **records, int *size, int 
*capacity, const char *plasmaType, float density, float temperature, float electricField, 
VariableProperties properties, int isDensity) {

    if (*size == *capacity) {
        *capacity *= 2;
```

```
        PlasmaSimulationRecord *newRecords = (PlasmaSimulationRecord *)malloc(*capacity *
sizeof(PlasmaSimulationRecord));


        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }


        free(*records);
        *records = newRecords;
    }

    (*records)[*size].parameters.plasmaType = strdup(plasmaType);
    (*records)[*size].parameters.density = density;
    (*records)[*size].parameters.temperature = temperature;
    (*records)[*size].parameters.electricField = electricField;

    (*records)[*size].properties = properties;
    (*records)[*size].isDensity = isDensity;

    // Increment the size
    (*size)++;
}

// Function to simulate the behavior of plasma
void simulatePlasmaBehavior(PlasmaSimulationRecord **records, int size) {
    // Placeholder function for simulating plasma behavior
```

```
    // Actual simulation logic will depend on the specific requirements of the plasma
dynamics simulation
    printf("Simulating plasma behavior...\n");
}

// Function to display the plasma simulation records
void displayPlasmaSimulationRecords(PlasmaSimulationRecord **records, int size) {
    printf("Plasma Simulation Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Plasma Type: %s\n", (*records)[i].parameters.plasmaType);
        printf("Density: %.2e particles/m^3\n", (*records)[i].parameters.density);
        printf("Temperature: %.2e K\n", (*records)[i].parameters.temperature);
        printf("Electric Field: %.2f V/m\n", (*records)[i].parameters.electricField);
        if ((*records)[i].isDensity) {
            printf("Variable Property: Density = %.2e particles/m^3\n",
(*records)[i].properties.density);
        } else {
            printf("Variable Property: Temperature = %.2e K\n",
(*records)[i].properties.temperature);
        }
        printf("\n");
    }
}
```

```c
/****************************************************************************
****

12. Kinematics Equation Solver

Description:

Solve complex kinematics problems for objects in motion.

Specifications:

Structure: Represents object properties (initial velocity, acceleration, displacement).

Array: Stores time-dependent motion data.

Union: Handles either velocity or displacement equations.

Strings: Represent motion descriptions.

const Pointers: Protect object properties.

Double Pointers: Dynamically allocate memory for motion data.


*****************************************************************************
****/


#include <stdio.h>

#include <stdlib.h>

#include <string.h>



typedef struct {

   const char *description;

   float initialVelocity; // in m/s

   float acceleration; // in m/s^2

   float displacement; // in m

} ObjectProperties;
```

```c
typedef union {
    float velocity; // in m/s
    float displacement; // in m
} KinematicEquations;

// Define the structure for kinematics data
typedef struct {
    ObjectProperties properties;
    KinematicEquations equations;
    int isVelocity; // 1 if velocity, 0 if displacement
} KinematicsData;



void addKinematicsData(KinematicsData **data, int *size, int *capacity, const char *description, float initialVelocity, float acceleration, float displacement, KinematicEquations equations, int isVelocity);
void calculateMotion(KinematicsData **data, int size, float time);
void displayKinematicsData(KinematicsData **data, int size);

int main() {
    int size = 0, capacity = 2;
    KinematicsData *data = (KinematicsData *)malloc(capacity * sizeof(KinematicsData));



    KinematicEquations equations1;
    equations1.velocity = 20.0;

    KinematicEquations equations2;
    equations2.displacement = 50.0;
```

```
    addKinematicsData(&data, &size, &capacity, "Object A", 10.0, 2.0, 0.0, equations1, 1);
    addKinematicsData(&data, &size, &capacity, "Object B", 15.0, 1.5, 0.0, equations2, 0);


    float time = 5.0;
    calculateMotion(&data, size, time);

    // Display the kinematics data
    displayKinematicsData(&data, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)data[i].properties.description);
    }
    free(data);

    return 0;
}


void addKinematicsData(KinematicsData **data, int *size, int *capacity, const char *description, float initialVelocity, float acceleration, float displacement, KinematicEquations equations, int isVelocity) {

    if (*size == *capacity) {
        *capacity *= 2;
```

```
        KinematicsData *newData = (KinematicsData *)malloc(*capacity *
sizeof(KinematicsData));

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


        free(*data);
        *data = newData;
    }


    (*data)[*size].properties.description = strdup(description);
    (*data)[*size].properties.initialVelocity = initialVelocity;
    (*data)[*size].properties.acceleration = acceleration;
    (*data)[*size].properties.displacement = displacement;

    (*data)[*size].equations = equations;
    (*data)[*size].isVelocity = isVelocity;

    // Increment the size
    (*size)++;
}

// Function to calculate motion based on time
void calculateMotion(KinematicsData **data, int size, float time) {
    for (int i = 0; i < size; i++) {
```

```
        if ((*data)[i].isVelocity) {
            // Calculate velocity using v = u + at
            (*data)[i].equations.velocity = (*data)[i].properties.initialVelocity +
(*data)[i].properties.acceleration * time;
        } else {
            // Calculate displacement using s = ut + 0.5at^2
            (*data)[i].equations.displacement = (*data)[i].properties.initialVelocity * time + 0.5 *
(*data)[i].properties.acceleration * time * time;
        }
    }
}

// Function to display the kinematics data
void displayKinematicsData(KinematicsData **data, int size) {
    printf("Kinematics Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Description: %s\n", (*data)[i].properties.description);
        printf("Initial Velocity: %.2f m/s\n", (*data)[i].properties.initialVelocity);
        printf("Acceleration: %.2f m/s^2\n", (*data)[i].properties.acceleration);
        if ((*data)[i].isVelocity) {
            printf("Velocity after given time: %.2f m/s\n", (*data)[i].equations.velocity);
        } else {
            printf("Displacement after given time: %.2f m\n", (*data)[i].equations.displacement);
        }
        printf("\n");
    }
}
```

```c
/**************************************************************************
****
13. Spectral Line Database
Description:
Develop a database to store and analyze spectral lines of elements.
Specifications:
Structure: Represents line properties (wavelength, intensity, and element).
Array: Stores spectral line data.
Union: Handles either intensity or wavelength information.
Strings: Represent element names.
const Pointers: Protect spectral line data.
Double Pointers: Allocate spectral line records dynamically.

**************************************************************************
****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>



typedef struct {
   const char *elementName;
   float wavelength; // in nm
   float intensity; // arbitrary units
} LineProperties;

// Define the union for variable line properties
typedef union {
   float wavelength; // in nm
```

```
    float intensity; // arbitrary units
} LineInfo;

// Define the structure for spectral line records
typedef struct {
    LineProperties properties;
    LineInfo info;
    int isWavelength; // 1 if wavelength, 0 if intensity
} SpectralLineRecord;



void addSpectralLineRecord(SpectralLineRecord **records, int *size, int *capacity, const char
*elementName, float wavelength, float intensity, LineInfo info, int isWavelength);
void displaySpectralLineRecords(SpectralLineRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    SpectralLineRecord *records = (SpectralLineRecord *)malloc(capacity *
sizeof(SpectralLineRecord));



    LineInfo info1;
    info1.wavelength = 656.3; // in nm

    LineInfo info2;
    info2.intensity = 100.0;

    // Add spectral line records
    addSpectralLineRecord(&records, &size, &capacity, "Hydrogen", 656.3, 100.0, info1, 1);
```

```
    addSpectralLineRecord(&records, &size, &capacity, "Helium", 587.6, 200.0, info2, 0);

    displaySpectralLineRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].properties.elementName); // Cast to void* to free const char*
    }
    free(records);

    return 0;
}

// Function to add a spectral line record to the database
void addSpectralLineRecord(SpectralLineRecord **records, int *size, int *capacity, const char
*elementName, float wavelength, float intensity, LineInfo info, int isWavelength) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        SpectralLineRecord *newRecords = (SpectralLineRecord *)malloc(*capacity *
sizeof(SpectralLineRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
```

```
        *records = newRecords;
    }

    // Initialize the new spectral line record
    (*records)[*size].properties.elementName = strdup(elementName);
    (*records)[*size].properties.wavelength = wavelength;
    (*records)[*size].properties.intensity = intensity;

    (*records)[*size].info = info;
    (*records)[*size].isWavelength = isWavelength;

    // Increment the size
    (*size)++;
}

// Function to display the spectral line records
void displaySpectralLineRecords(SpectralLineRecord **records, int size) {
    printf("Spectral Line Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Element Name: %s\n", (*records)[i].properties.elementName);
        printf("Wavelength: %.2f nm\n", (*records)[i].properties.wavelength);
        printf("Intensity: %.2f (arbitrary units)\n", (*records)[i].properties.intensity);
        if ((*records)[i].isWavelength) {
            printf("Variable Property: Wavelength = %.2f nm\n", (*records)[i].info.wavelength);
        } else {
            printf("Variable Property: Intensity = %.2f (arbitrary units)\n",
(*records)[i].info.intensity);
        }
```

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


/************************************************************************
****
14. Projectile Motion Simulator
Description:
Simulate and analyze projectile motion under varying conditions.
Specifications:
Structure: Stores projectile properties (mass, velocity, and angle).
Array: Stores motion trajectory data.
Union: Handles either velocity or displacement parameters.
Strings: Represent trajectory descriptions.
const Pointers: Protect projectile properties.
Double Pointers: Manage trajectory records dynamically.

*************************************************************************
****/

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


typedef struct {
    const char *description;
```

```
    float mass; // in kg
    float velocity; // in m/s
    float angle; // in degrees
} ProjectileProperties;


typedef union {
    float velocity; // in m/s
    float displacement; // in m
} MotionParameters;


typedef struct {
    ProjectileProperties properties;
    MotionParameters parameters;
    int isVelocity;
} ProjectileMotionRecord;


void addProjectileMotionRecord(ProjectileMotionRecord **records, int *size, int *capacity, const char *description, float mass, float velocity, float angle, MotionParameters parameters, int isVelocity);
void calculateTrajectory(ProjectileMotionRecord **records, int size, float time);
void displayProjectileMotionRecords(ProjectileMotionRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    ProjectileMotionRecord *records = (ProjectileMotionRecord *)malloc(capacity * sizeof(ProjectileMotionRecord));
```

```
    MotionParameters parameters1;
    parameters1.velocity = 50.0; // in m/s

    MotionParameters parameters2;
    parameters2.displacement = 100.0; // in m

    // Add projectile motion records
    addProjectileMotionRecord(&records, &size, &capacity, "Projectile A", 2.0, 50.0, 45.0,
parameters1, 1);
    addProjectileMotionRecord(&records, &size, &capacity, "Projectile B", 1.5, 40.0, 60.0,
parameters2, 0);



    float time = 5.0; // time in seconds
    calculateTrajectory(&records, size, time);

    // Display the projectile motion records
    displayProjectileMotionRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].properties.description); // Cast to void* to free const char*
    }
    free(records);

    return 0;
}
```

```
void addProjectileMotionRecord(ProjectileMotionRecord **records, int *size, int *capacity,
const char *description, float mass, float velocity, float angle, MotionParameters
parameters, int isVelocity) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        ProjectileMotionRecord *newRecords = (ProjectileMotionRecord *)malloc(*capacity *
sizeof(ProjectileMotionRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }

    // Initialize the new projectile motion record
    (*records)[*size].properties.description = strdup(description);
    (*records)[*size].properties.mass = mass;
    (*records)[*size].properties.velocity = velocity;
    (*records)[*size].properties.angle = angle;

    (*records)[*size].parameters = parameters;
    (*records)[*size].isVelocity = isVelocity;
```

```
    // Increment the size
    (*size)++;
}

// Function to calculate the trajectory of the projectiles
void calculateTrajectory(ProjectileMotionRecord **records, int size, float time) {
    const float g = 9.81;

    for (int i = 0; i < size; i++) {
        float angleRad = (*records)[i].properties.angle * M_PI / 180.0; // convert angle to
radians
        float initialVelocityX = (*records)[i].properties.velocity * cos(angleRad);
        float initialVelocityY = (*records)[i].properties.velocity * sin(angleRad);
        float displacementX = initialVelocityX * time;
        float displacementY = initialVelocityY * time - 0.5 * g * time * time;

        if ((*records)[i].isVelocity) {
            // Calculate velocity at given time
            float velocityX = initialVelocityX;
            float velocityY = initialVelocityY - g * time;
            (*records)[i].parameters.velocity = sqrt(velocityX * velocityX + velocityY * velocityY);
        } else {
            // Store displacement at given time
            (*records)[i].parameters.displacement = sqrt(displacementX * displacementX +
displacementY * displacementY);
        }
    }
}
```

```
// Function to display the projectile motion records
void displayProjectileMotionRecords(ProjectileMotionRecord **records, int size) {
    printf("Projectile Motion Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Description: %s\n", (*records)[i].properties.description);
        printf("Mass: %.2f kg\n", (*records)[i].properties.mass);
        printf("Initial Velocity: %.2f m/s\n", (*records)[i].properties.velocity);
        printf("Angle: %.2f degrees\n", (*records)[i].properties.angle);
        if ((*records)[i].isVelocity) {
            printf("Velocity after given time: %.2f m/s\n", (*records)[i].parameters.velocity);
        } else {
            printf("Displacement after given time: %.2f m\n",
(*records)[i].parameters.displacement);
        }
        printf("\n");
    }
}

/**************************************************************************
****
```

15. Material Stress-Strain Analyzer

Description:

Analyze the stress-strain behavior of materials under different loads.

Specifications:

Structure: Represents material properties (stress, strain, modulus).

Array: Stores stress-strain data.

Union: Handles dependent properties like yield stress or elastic modulus.

Strings: Represent material names.

```c
const Pointers: Protect material properties.

Double Pointers: Allocate stress-strain data dynamically.

***************************************************************************
****/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct {

    const char *materialName;

    float stress; // in MPa

    float strain; // in percentage

    float modulus; // in GPa

} MaterialProperties;

// Define the union for dependent properties

typedef union {

    float yieldStress; // in MPa

    float elasticModulus; // in GPa

} DependentProperties;

// Define the structure for stress-strain records

typedef struct {

    MaterialProperties properties;
```

```
    DependentProperties dependent;
    int isYieldStress; // 1 if yield stress, 0 if elastic modulus
} StressStrainRecord;


void addStressStrainRecord(StressStrainRecord **records, int *size, int *capacity, const char
*materialName, float stress, float strain, float modulus, DependentProperties dependent, int
isYieldStress);
void displayStressStrainRecords(StressStrainRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    StressStrainRecord *records = (StressStrainRecord *)malloc(capacity *
sizeof(StressStrainRecord));

    // Define some material properties
    DependentProperties dependent1;
    dependent1.yieldStress = 250.0; // in MPa

    DependentProperties dependent2;
    dependent2.elasticModulus = 200.0;

    // Add stress-strain records
    addStressStrainRecord(&records, &size, &capacity, "Steel", 300.0, 0.01, 210.0,
dependent1, 1);
    addStressStrainRecord(&records, &size, &capacity, "Aluminum", 150.0, 0.02, 70.0,
dependent2, 0);

    displayStressStrainRecords(&records, size);
```

```
    for (int i = 0; i < size; i++) {
        free((void *)records[i].properties.materialName);
    }
    free(records);

    return 0;
}

// Function to add a stress-strain record to the database
void addStressStrainRecord(StressStrainRecord **records, int *size, int *capacity, const char
*materialName, float stress, float strain, float modulus, DependentProperties dependent, int
isYieldStress) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        StressStrainRecord *newRecords = (StressStrainRecord *)malloc(*capacity *
sizeof(StressStrainRecord));

        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }



        free(*records);
        *records = newRecords;
    }
```

```
    (*records)[*size].properties.materialName = strdup(materialName);
    (*records)[*size].properties.stress = stress;
    (*records)[*size].properties.strain = strain;
    (*records)[*size].properties.modulus = modulus;

    (*records)[*size].dependent = dependent;
    (*records)[*size].isYieldStress = isYieldStress;



    (*size)++;
}

void displayStressStrainRecords(StressStrainRecord **records, int size) {
    printf("Material Stress-Strain Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Material Name: %s\n", (*records)[i].properties.materialName);
        printf("Stress: %.2f MPa\n", (*records)[i].properties.stress);
        printf("Strain: %.2f%%\n", (*records)[i].properties.strain);
        printf("Modulus: %.2f GPa\n", (*records)[i].properties.modulus);
        if ((*records)[i].isYieldStress) {
            printf("Dependent Property: Yield Stress = %.2f MPa\n",
(*records)[i].dependent.yieldStress);
        } else {
            printf("Dependent Property: Elastic Modulus = %.2f GPa\n",
(*records)[i].dependent.elasticModulus);
        }
        printf("\n");
    }
}
```