Assignment -26

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.
- **const Pointers:** Protect material properties.
- **Double Pointers:** Allocate stress-strain data dynamically.

```
//1.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>



#define GRAVITY 9.8



typedef struct {

    double x, y;
```

```c
} Vector2D;

typedef struct {
    double mass;
    Vector2D position;
    Vector2D velocity;
} Particle;

typedef union {
    double gravitationalForce;
    double electricForce;
    double magneticForce;
} ForceType;

typedef enum {
    GRAVITATIONAL,
    ELECTRIC,
    MAGNETIC
} Force;

void applyForce(Particle *p, ForceType force, Force type) {
    switch (type) {
        case GRAVITATIONAL:
```

```
            p->velocity.y -= force.gravitationalForce / p->mass;
            break;
        case ELECTRIC:
            p->velocity.x += force.electricForce / p->mass;
            break;
        case MAGNETIC:
            p->velocity.y -= force.magneticForce / p->mass;
            break;
        default:
            printf("Unknown force type!\n");
            break;
    }
}

void updateParticle(Particle *p, double deltaTime) {
    p->position.x += p->velocity.x * deltaTime;
    p->position.y += p->velocity.y * deltaTime;
}

void printParticleState(Particle *p) {
    printf("Position: (%.2f, %.2f)  Velocity: (%.2f, %.2f)\n",
           p->position.x, p->position.y, p->velocity.x, p->velocity.y);
}
```

```
int main() {
    int numParticles = 3;
    Particle *particleSystem = (Particle *)malloc(numParticles * sizeof(Particle *));
    if (particleSystem == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numParticles; i++) {
        particleSystem[i] = (Particle *)malloc(sizeof(Particle));
        if (particleSystem[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }
        particleSystem[i]->mass = 1.0;
        particleSystem[i]->position.x = 0.0;
        particleSystem[i]->position.y = 0.0;
        particleSystem[i]->velocity.x = 0.0;
        particleSystem[i]->velocity.y = 0.0;
    }

    ForceType force;
```

```c
    force.gravitationalForce = GRAVITY;


    double deltaTime = 0.1;
    for (int t = 0; t <= 5; t++) {  // Time steps from 0 to 5
        printf("\nTime step %d:\n", t);
        for (int i = 0; i < numParticles; i++) {
            applyForce(particleSystem[i], force, GRAVITATIONAL);
            updateParticle(particleSystem[i], deltaTime);
            printParticleState(particleSystem[i]);
        }
    }


    for (int i = 0; i < numParticles; i++) {
        free(particleSystem[i]);
    }
    free(particleSystem);


    return 0;
}


//2.


#include <stdio.h>
```

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

typedef struct {
    double Ex, Ey, Ez;
    double Bx, By, Bz;
    double x, y, z;
} FieldPoint;

typedef union {
    double electricField[3];
    double magneticField[3];
} FieldComponents;

typedef enum {
    CARTESIAN,
    CYLINDRICAL,
    SPHERICAL
} CoordinateSystem;

void printField(FieldPoint *point, CoordinateSystem coordSys) {
    printf("Position: (%.2f, %.2f, %.2f)\n", point->x, point->y, point->z);
    if (coordSys == CARTESIAN) {
```

```
        printf("Electric Field: (%.2f, %.2f, %.2f)\n", point->Ex, point->Ey, point->Ez);
        printf("Magnetic Field: (%.2f, %.2f, %.2f)\n", point->Bx, point->By, point->Bz);
    } else if (coordSys == CYLINDRICAL) {
    } else if (coordSys == SPHERICAL) {
    }
}

int main() {
    int numPoints = 3;
    FieldPoint *fieldGrid = (FieldPoint *)malloc(numPoints * sizeof(FieldPoint *));
    if (fieldGrid == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numPoints; i++) {
        fieldGrid[i] = (FieldPoint *)malloc(sizeof(FieldPoint));
        if (fieldGrid[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }

        fieldGrid[i]->x = i * 1.0;
```

```
        fieldGrid[i]->y = i * 1.0;
        fieldGrid[i]->z = i * 1.0;
        fieldGrid[i]->Ex = 0.5 * i;
        fieldGrid[i]->Ey = 0.5 * i;
        fieldGrid[i]->Ez = 0.5 * i;
        fieldGrid[i]->Bx = 0.2 * i;
        fieldGrid[i]->By = 0.2 * i;
        fieldGrid[i]->Bz = 0.2 * i;
    }

    CoordinateSystem coordSys = CARTESIAN;
    for (int i = 0; i < numPoints; i++) {
        printField(fieldGrid[i], coordSys);
    }

    for (int i = 0; i < numPoints; i++) {
        free(fieldGrid[i]);
    }
    free(fieldGrid);

    return 0;
}
```

```
//3.

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

typedef struct {
    double Ex, Ey, Ez;
    double Bx, By, Bz;
    double x, y, z;
} FieldPoint;

typedef union {
    double electricField[3];
    double magneticField[3];
} FieldComponents;

typedef enum {
    CARTESIAN,
    CYLINDRICAL,
    SPHERICAL
} CoordinateSystem;
```

```
void printField(FieldPoint *point, CoordinateSystem coordSys) {
    printf("Position: (%.2f, %.2f, %.2f)\n", point->x, point->y, point->z);
    if (coordSys == CARTESIAN) {
        printf("Electric Field: (%.2f, %.2f, %.2f)\n", point->Ex, point->Ey, point->Ez);
        printf("Magnetic Field: (%.2f, %.2f, %.2f)\n", point->Bx, point->By, point->Bz);
    } else if (coordSys == CYLINDRICAL) {
    } else if (coordSys == SPHERICAL) {
    }
}

int main() {
    int numPoints = 3;
    FieldPoint *fieldGrid = (FieldPoint *)malloc(numPoints * sizeof(FieldPoint *));
    if (fieldGrid == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numPoints; i++) {
        fieldGrid[i] = (FieldPoint *)malloc(sizeof(FieldPoint));
        if (fieldGrid[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
```

```
    }

    fieldGrid[i]->x = i * 1.0;
    fieldGrid[i]->y = i * 1.0;
    fieldGrid[i]->z = i * 1.0;
    fieldGrid[i]->Ex = 0.5 * i;
    fieldGrid[i]->Ey = 0.5 * i;
    fieldGrid[i]->Ez = 0.5 * i;
    fieldGrid[i]->Bx = 0.2 * i;
    fieldGrid[i]->By = 0.2 * i;
    fieldGrid[i]->Bz = 0.2 * i;
}

CoordinateSystem coordSys = CARTESIAN;
for (int i = 0; i < numPoints; i++) {
    printField(fieldGrid[i], coordSys);
}

for (int i = 0; i < numPoints; i++) {
    free(fieldGrid[i]);
}
free(fieldGrid);
```

```c
    return 0;
}

//4.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct {
    double amplitude;
    double phase;
    double energy;
} QuantumState;

typedef union {
    double amplitude;
    double phase;
} StateProperty;

typedef enum {
    GROUND_STATE,
    EXCITED_STATE
} StateType;
```

```c
void printQuantumState(QuantumState *state, StateType type) {
    printf("State Type: %s\n", type == GROUND_STATE ? "Ground State" : "Excited State");
    printf("Amplitude: %.2f\n", state->amplitude);
    printf("Phase: %.2f rad\n", state->phase);
    printf("Energy: %.2f eV\n", state->energy);
}


int main() {
    int numStates = 3;
    QuantumState *quantumStates = (QuantumState *)malloc(numStates * sizeof(QuantumState
*));
    if (quantumStates == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }


    for (int i = 0; i < numStates; i++) {
        quantumStates[i] = (QuantumState *)malloc(sizeof(QuantumState));
        if (quantumStates[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }
```

```c
        quantumStates[i]->amplitude = (i + 1) * 0.5;
        quantumStates[i]->phase = (i + 1) * 0.2;
        quantumStates[i]->energy = (i + 1) * 1.0;
    }

    StateType type = GROUND_STATE;
    for (int i = 0; i < numStates; i++) {
        printQuantumState(quantumStates[i], type);
        type = EXCITED_STATE;
    }

    for (int i = 0; i < numStates; i++) {
        free(quantumStates[i]);
    }
    free(quantumStates);

    return 0;
}

//5.

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

```
typedef struct {
    double refractiveIndex;
    double focalLength;
} OpticalElement;

typedef union {
    double lensParameters[2];
    double mirrorParameters[1];
} ElementParameters;

typedef enum {
    LENS,
    MIRROR
} ElementType;

void printOpticalElement(OpticalElement *element, ElementType type) {
    if (type == LENS) {
        printf("Element Type: Lens\n");
        printf("Refractive Index: %.2f\n", element->refractiveIndex);
        printf("Focal Length: %.2f\n", element->focalLength);
    } else if (type == MIRROR) {
        printf("Element Type: Mirror\n");
```

```
        printf("Refractive Index: %.2f\n", element->refractiveIndex);
    }
}


int main() {
    int numElements = 2;
    OpticalElement *opticalElements = (OpticalElement *)malloc(numElements *
sizeof(OpticalElement *));
    if (opticalElements == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }


    for (int i = 0; i < numElements; i++) {
        opticalElements[i] = (OpticalElement *)malloc(sizeof(OpticalElement));
        if (opticalElements[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }


        opticalElements[i]->refractiveIndex = 1.5;
        opticalElements[i]->focalLength = 5.0;
    }
```

```
    ElementType type = LENS;
    for (int i = 0; i < numElements; i++) {
        printOpticalElement(opticalElements[i], type);
        type = MIRROR;
    }

    for (int i = 0; i < numElements; i++) {
        free(opticalElements[i]);
    }
    free(opticalElements);

    return 0;
}

//6.

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

typedef struct {
    double pressure;
    double volume;
    double temperature;
```

```
    double entropy;
} ThermodynamicState;

typedef union {
    double energy;
    double entropy;
} StateProperties;

typedef enum {
    GAS,
    LIQUID
} StateType;

void printThermodynamicState(ThermodynamicState *state, StateType type) {
    printf("State Type: %s\n", type == GAS ? "Gas" : "Liquid");
    printf("Pressure: %.2f Pa\n", state->pressure);
    printf("Volume: %.2f m^3\n", state->volume);
    printf("Temperature: %.2f K\n", state->temperature);
    printf("Entropy: %.2f J/K\n", state->entropy);
}

int main() {
    int numStates = 3;
```

```
    ThermodynamicState *states = (ThermodynamicState *)malloc(numStates *
sizeof(ThermodynamicState *));

    if (states == NULL) {

        printf("Memory allocation failed!\n");

        return 1;

    }


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

        states[i] = (ThermodynamicState *)malloc(sizeof(ThermodynamicState));

        if (states[i] == NULL) {

            printf("Memory allocation failed!\n");

            return 1;

        }


        states[i]->pressure = 1.0e5;

        states[i]->volume = 1.0;

        states[i]->temperature = 300.0;

        states[i]->entropy = 100.0;

    }


    StateType type = GAS;

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

        printThermodynamicState(states[i], type);

        type = LIQUID;
```

```
    }

    for (int i = 0; i < numStates; i++) {
        free(states[i]);
    }
    free(states);

    return 0;
}

//7.

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

typedef struct {
    char reactant[50];
    char product[50];
    double energyReleased;
} NuclearReaction;

typedef union {
```

```
    double energyReleased;

    char product[50];

} ReactionDetails;


void printReactionDetails(NuclearReaction *reaction) {

    printf("Reactant: %s\n", reaction->reactant);

    printf("Product: %s\n", reaction->product);

    printf("Energy Released: %.2f MeV\n", reaction->energyReleased);

}


int main() {

    int numReactions = 2;

    NuclearReaction *reactions = (NuclearReaction *)malloc(numReactions *
sizeof(NuclearReaction *));

    if (reactions == NULL) {

        printf("Memory allocation failed!\n");

        return 1;

    }


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

        reactions[i] = (NuclearReaction *)malloc(sizeof(NuclearReaction));

        if (reactions[i] == NULL) {

            printf("Memory allocation failed!\n");

            return 1;
```

```
    }

    if (i == 0) {
        strcpy(reactions[i]->reactant, "Uranium-235");
        strcpy(reactions[i]->product, "Krypton + Barium");
        reactions[i]->energyReleased = 200.0;
    } else {
        strcpy(reactions[i]->reactant, "Deuterium");
        strcpy(reactions[i]->product, "Helium");
        reactions[i]->energyReleased = 17.6;
    }
}

for (int i = 0; i < numReactions; i++) {
    printReactionDetails(reactions[i]);
}

for (int i = 0; i < numReactions; i++) {
    free(reactions[i]);
}
free(reactions);

return 0;
```

```c
}

//8.

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

#define G 6.67430e-11

typedef struct {
    double mass;
    double x, y, z;
    double fieldStrength;
} GravitationalObject;

typedef union {
    double mass;
    double fieldStrength;
} FieldParameters;

void calculateFieldStrength(GravitationalObject *object) {
    double r = sqrt(object->x * object->x + object->y * object->y + object->z * object->z);
```

```
    object->fieldStrength = G * object->mass / (r * r);
}

void printGravitationalObject(GravitationalObject *object, const char *label) {
    printf("Object: %s\n", label);
    printf("Mass: %.2e kg\n", object->mass);
    printf("Position: (%.2f, %.2f, %.2f)\n", object->x, object->y, object->z);
    printf("Gravitational Field Strength: %.2e N/kg\n", object->fieldStrength);
}

int main() {
    int numObjects = 2;
    GravitationalObject *objects = (GravitationalObject *)malloc(numObjects *
sizeof(GravitationalObject *));
    if (objects == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numObjects; i++) {
        objects[i] = (GravitationalObject *)malloc(sizeof(GravitationalObject));
        if (objects[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
```

```
    }

    if (i == 0) {
        objects[i]->mass = 5.97e24; // Mass of Earth
        objects[i]->x = 0;
        objects[i]->y = 0;
        objects[i]->z = 0;
    } else {
        objects[i]->mass = 1.99e30; // Mass of Sun
        objects[i]->x = 1.496e11; // Distance from Earth (1 AU)
        objects[i]->y = 0;
        objects[i]->z = 0;
    }

    calculateFieldStrength(objects[i]);
}

const char *labels[] = {"Earth", "Sun"};
for (int i = 0; i < numObjects; i++) {
    printGravitationalObject(objects[i], labels[i]);
}

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

```c
        free(objects[i]);
    }
    free(objects);

    return 0;
}

//9.

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

typedef struct {
    double amplitude;
    double wavelength;
    double phase;
} Wave;

typedef union {
    double amplitude;
    double phase;
} WaveProperty;
```

```
void printWaveProperties(Wave *wave, const char *label) {
    printf("Wave Source: %s\n", label);
    printf("Amplitude: %.2f\n", wave->amplitude);
    printf("Wavelength: %.2f m\n", wave->wavelength);
    printf("Phase: %.2f rad\n", wave->phase);
}

int main() {
    int numWaves = 2;
    Wave *waves = (Wave *)malloc(numWaves * sizeof(Wave *));
    if (waves == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numWaves; i++) {
        waves[i] = (Wave *)malloc(sizeof(Wave));
        if (waves[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }
```

```
        if (i == 0) {
            waves[i]->amplitude = 1.0;
            waves[i]->wavelength = 500.0;
            waves[i]->phase = 0.0;
        } else {
            waves[i]->amplitude = 0.5;
            waves[i]->wavelength = 600.0;
            waves[i]->phase = M_PI / 2;
        }
    }

    const char *labels[] = {"Wave 1", "Wave 2"};
    for (int i = 0; i < numWaves; i++) {
        printWaveProperties(waves[i], labels[i]);
    }

    for (int i = 0; i < numWaves; i++) {
        free(waves[i]);
    }
    free(waves);

    return 0;
}
```

```c
//10.

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

typedef struct {
    char materialName[50];
    double permeability;
    double saturation;
} MagneticMaterial;

typedef union {
    double permeability;
    double saturation;
} MaterialProperty;

void printMaterialProperties(MagneticMaterial *material) {
    printf("Material: %s\n", material->materialName);
    printf("Permeability: %.2e H/m\n", material->permeability);
    printf("Saturation: %.2e A/m\n", material->saturation);
}
```

```
int main() {
    int numMaterials = 2;
    MagneticMaterial *materials = (MagneticMaterial *)malloc(numMaterials *
sizeof(MagneticMaterial *));
    if (materials == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numMaterials; i++) {
        materials[i] = (MagneticMaterial *)malloc(sizeof(MagneticMaterial));
        if (materials[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
        }

        if (i == 0) {
            strcpy(materials[i]->materialName, "Iron");
            materials[i]->permeability = 1.26e-6;
            materials[i]->saturation = 2.2e6;
        } else {
            strcpy(materials[i]->materialName, "Nickel");
            materials[i]->permeability = 6.5e-6;
```

```
            materials[i]->saturation = 0.48e6;
        }
    }

    for (int i = 0; i < numMaterials; i++) {
        printMaterialProperties(materials[i]);
    }

    for (int i = 0; i < numMaterials; i++) {
        free(materials[i]);
    }
    free(materials);

    return 0;
}

//11.

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

typedef struct {
    double density;
```

```c
    double temperature;
    double electricField;
} Plasma;

typedef union {
    double density;
    double temperature;
} PlasmaData;

void printPlasmaData(Plasma *plasma, const char *type) {
    printf("Plasma Type: %s\n", type);
    printf("Density: %.2f particles/m^3\n", plasma->density);
    printf("Temperature: %.2f K\n", plasma->temperature);
    printf("Electric Field: %.2f V/m\n", plasma->electricField);
}

int main() {
    int numPlasmaTypes = 2;
    Plasma *plasmas = (Plasma *)malloc(numPlasmaTypes * sizeof(Plasma *));
    if (plasmas == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
```

```c
for (int i = 0; i < numPlasmaTypes; i++) {
    plasmas[i] = (Plasma *)malloc(sizeof(Plasma));
    if (plasmas[i] == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    if (i == 0) {
        plasmas[i]->density = 1.0e19;
        plasmas[i]->temperature = 15000;
        plasmas[i]->electricField = 2.5e3;
    } else {
        plasmas[i]->density = 5.0e18;
        plasmas[i]->temperature = 10000;
        plasmas[i]->electricField = 3.0e3;
    }
}

const char *types[] = {"Ionized Plasma", "Neutral Plasma"};
for (int i = 0; i < numPlasmaTypes; i++) {
    printPlasmaData(plasmas[i], types[i]);
}
```

```
    for (int i = 0; i < numPlasmaTypes; i++) {
        free(plasmas[i]);
    }
    free(plasmas);

    return 0;
}

//12.

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

typedef struct {
    double initialVelocity;
    double acceleration;
    double displacement;
} Kinematics;

typedef union {
    double velocity;
```

```
    double displacement;
} MotionData;

void printKinematicsData(Kinematics *kinematics, const char *description) {
    printf("Motion Description: %s\n", description);
    printf("Initial Velocity: %.2f m/s\n", kinematics->initialVelocity);
    printf("Acceleration: %.2f m/s^2\n", kinematics->acceleration);
    printf("Displacement: %.2f m\n", kinematics->displacement);
}

int main() {
    int numObjects = 2;
    Kinematics *objects = (Kinematics *)malloc(numObjects * sizeof(Kinematics *));
    if (objects == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < numObjects; i++) {
        objects[i] = (Kinematics *)malloc(sizeof(Kinematics));
        if (objects[i] == NULL) {
            printf("Memory allocation failed!\n");
            return 1;
```

```
}

if (i == 0) {
    objects[i]->initialVelocity = 0.0;
    objects[i]->acceleration = 9.8;
```