Day 23 programs

----

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.

```c
#include <stdio.h>

#include <stdlib.h>

#include <math.h>


// Define constants for forces

#define GRAVITY 9.8

#define ELECTRIC_CONSTANT 8.99e9

#define MAGNETIC_CONSTANT 1.0


// Force types enumeration

typedef enum {

   GRAVITATIONAL,

   ELECTRIC,

   MAGNETIC
```

```
} ForceType;

// Particle structure
typedef struct {
    double mass;            // Mass of the particle (kg)
    double position[2];     // Position in 2D (x, y)
    double velocity[2];     // Velocity in 2D (vx, vy)
} Particle;

// Union for force types
typedef union {
    double gravitational_force;
    double electric_force;
    double magnetic_force;
} Force;

// Function prototypes
void updateParticlePosition(Particle* particle, double time_interval);
void applyForce(Particle* particle, ForceType force_type, double force_magnitude);
void printParticle(Particle* particle);

int main() {
    // Dynamically allocate memory for an array of particles using double pointers
    int num_particles = 3;  // Example for 3 particles
    Particle** particles = (Particle*) malloc(num_particles * sizeof(Particle));

    // Initialize particles
    for (int i = 0; i < num_particles; i++) {
        particles[i] = (Particle*) malloc(sizeof(Particle));
        particles[i]->mass = 1.0 + i;  // Assign mass of each particle
        particles[i]->position[0] = i * 1.0;  // Assign initial position
        particles[i]->position[1] = 0.0;
        particles[i]->velocity[0] = 0.0;  // Initial velocity
        particles[i]->velocity[1] = 0.0;
    }

    // Apply forces to particles and update positions
    applyForce(particles[0], GRAVITATIONAL, 9.8);  // Gravitational force on particle 0
    applyForce(particles[1], ELECTRIC, 5.0);  // Electric force on particle 1
    applyForce(particles[2], MAGNETIC, 3.0);  // Magnetic force on particle 2

    // Update particle positions based on forces and time intervals
    double time_interval = 1.0;  // 1 second time step
    for (int i = 0; i < num_particles; i++) {
        updateParticlePosition(particles[i], time_interval);
        printParticle(particles[i]);
    }

    // Free allocated memory
    for (int i = 0; i < num_particles; i++) {
        free(particles[i]);
    }
    free(particles);

    return 0;
}

// Function to update particle position based on velocity
```

```
void updateParticlePosition(Particle* particle, double time_interval) {
    particle->position[0] += particle->velocity[0] * time_interval;
    particle->position[1] += particle->velocity[1] * time_interval;
}

// Function to apply a force to a particle
void applyForce(Particle* particle, ForceType force_type, double force_magnitude) {
    switch (force_type) {
        case GRAVITATIONAL:
            particle->velocity[1] += force_magnitude / particle->mass;  // Simplified
gravity
            break;
        case ELECTRIC:
            // Simple electric force, assuming particle charge = 1 (for simplicity)
            particle->velocity[0] += force_magnitude / particle->mass;
            break;
        case MAGNETIC:
            // Simplified magnetic force, assuming perpendicular motion to magnetic field
            particle->velocity[0] += force_magnitude / particle->mass;  // Simplification
            break;
    }
}

// Function to print the particle details
void printParticle(Particle* particle) {
    printf("Particle - Mass: %.2f, Position: (%.2f, %.2f), Velocity: (%.2f, %.2f)\n",
           particle->mass, particle->position[0], particle->position[1],
           particle->velocity[0], particle->velocity[1]);
}
```

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.

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

#define MAX_POINTS 100

// Define a union to hold either electric or magnetic field components
union FieldComponent {
    double electric[3];  // Electric field components (Ex, Ey, Ez)
    double magnetic[3];  // Magnetic field components (Bx, By, Bz)
};

// Struct to hold field parameters and the position
struct Field {
    union FieldComponent field;  // Electric or magnetic field
    double position[3];          // Coordinates (x, y, z)
};
```

```
// Function to print the field at a given point
void printField(struct Field *field, const char *fieldType, const char *coordSys) {
    printf("%s field at position (%lf, %lf, %lf) in %s coordinates:\n", fieldType, field-
>position[0], field->position[1], field->position[2], coordSys);
    for (int i = 0; i < 3; i++) {
        if (fieldType == "Electric") {
            printf("E%d: %lf\n", i+1, field->field.electric[i]);
        } else if (fieldType == "Magnetic") {
            printf("B%d: %lf\n", i+1, field->field.magnetic[i]);
        }
    }
}

// Function to calculate the intensity of the electromagnetic field at a given point
double calculateIntensity(struct Field *field) {
    double intensity = 0.0;
    // Calculate intensity as the magnitude of the field vector (E or B)
    for (int i = 0; i < 3; i++) {
        intensity += field->field.electric[i] * field->field.electric[i]; // For electric field
        // intensity += field->field.magnetic[i] * field->field.magnetic[i]; // Uncomment for
magnetic field
    }
    return sqrt(intensity);
}

int main() {
    // Array of field data at discrete points
    struct Field *fields = malloc(MAX_POINTS * sizeof(struct Field));
    // Example: Set electric and magnetic fields at points (this is just for illustration)
    fields[0].position[0] = 1.0; fields[0].position[1] = 2.0; fields[0].position[2] = 3.0;
    fields[0].field.electric[0] = 5.0; fields[0].field.electric[1] = 3.0;
fields[0].field.electric[2] = 1.0;

    // Print electric field at point (1,2,3) in Cartesian coordinates
    printField(&fields[0], "Electric", "Cartesian");

    // Calculate and print the intensity of the electric field at point (1,2,3)
    double intensity = calculateIntensity(&fields[0]);
    printf("Intensity of Electric field: %lf\n", intensity);

    // Free dynamically allocated memory
    free(fields);
    return 0;
}
```

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>

// Define a constant for maximum number of energy levels
#define MAX_ENERGY_LEVELS 10

// Define a structure to store energy level information
typedef struct {
    double energy[MAX_ENERGY_LEVELS];  // Energy levels for the atom
    double transitionProbabilities[MAX_ENERGY_LEVELS - 1]; // Transition
probabilities between levels
    int numLevels; // Number of energy levels for this atom
} EnergyStates;

// Define a structure to represent atomic details
typedef struct {
    const char *elementName; // Element name (string is constant)
    EnergyStates *energyStates; // Pointer to energy state data (dynamically
allocated)
} Atom;

// Function to create a new atom
Atom *createAtom(const char *elementName, int numLevels) {
    Atom *newAtom = (Atom *)malloc(sizeof(Atom));
    newAtom->elementName = elementName; // Constant string, no need to allocate
memory

    // Allocate memory for energy levels
    newAtom->energyStates = (EnergyStates *)malloc(sizeof(EnergyStates));
    newAtom->energyStates->numLevels = numLevels;
    // Initialize energy levels and transition probabilities to zero
    for (int i = 0; i < numLevels; i++) {
        newAtom->energyStates->energy[i] = 0.0;
        if (i < numLevels - 1) {
            newAtom->energyStates->transitionProbabilities[i] = 0.0;
        }
    }

    return newAtom;
}

// Function to free memory used by an atom
void freeAtom(Atom *atom) {
    if (atom) {
        free(atom->energyStates); // Free energy states memory
        free(atom); // Free atom memory
    }
}

// Function to set energy levels for a given atom
void setEnergyLevels(Atom *atom, double *energies, int numLevels) {
    for (int i = 0; i < numLevels; i++) {
        atom->energyStates->energy[i] = energies[i];
    }
}

// Function to set transition probabilities between energy levels
void setTransitionProbabilities(Atom *atom, double *probabilities, int numLevels) {
    for (int i = 0; i < numLevels - 1; i++) {
        atom->energyStates->transitionProbabilities[i] = probabilities[i];
    }
}

// Function to print atom details (energy levels and transition probabilities)
void printAtomDetails(Atom *atom) {
    printf("Element: %s\n", atom->elementName);
    printf("Energy Levels: \n");
    for (int i = 0; i < atom->energyStates->numLevels; i++) {
        printf("Level %d: %.2f eV\n", i + 1, atom->energyStates->energy[i]);
    }

    printf("Transition Probabilities: \n");
    for (int i = 0; i < atom->energyStates->numLevels - 1; i++) {
        printf("Transition %d -> %d: %.4f\n", i + 1, i + 2, atom->energyStates-
>transitionProbabilities[i]);
    }
}

// Main function to test the implementation
int main() {
    // Example of creating an atom and setting its energy levels and transition
probabilities
    Atom *hydrogen = createAtom("Hydrogen", 3);

    double hydrogenEnergies[] = {13.6, 3.4, 1.5}; // Energy levels in eV
    setEnergyLevels(hydrogen, hydrogenEnergies, 3);

    double hydrogenTransitions[] = {0.98, 0.95}; // Transition probabilities
    setTransitionProbabilities(hydrogen, hydrogenTransitions, 3);

    // Print details of the hydrogen atom
    printAtomDetails(hydrogen);

    // Free memory
    freeAtom(hydrogen);

    return 0;
}
```

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

// Define constants for quantum states
#define NUM_POINTS 10  // Number of points for wavefunction representation
```

```
// Union to store amplitude or phase
union QuantumInfo {
    double amplitude;  // Wavefunction amplitude
    double phase;      // Phase of the wavefunction
};

// Structure to represent a quantum state
struct QuantumState {
    union QuantumInfo *wavefunction;  // Pointer to an array of quantum information
(amplitude or phase)
    double energy;                    // Energy of the state
    const char *stateLabel;           // Label describing the quantum state
};

// Function to create a quantum state
struct QuantumState *createQuantumState(const char *label, double energy, double
*amplitudeData, double *phaseData) {
    struct QuantumState *state = (struct QuantumState *)malloc(sizeof(struct
QuantumState));
    state->stateLabel = label;
    state->energy = energy;

    // Dynamically allocate memory for wavefunction data (amplitude or phase)
    state->wavefunction = (union QuantumInfo *)malloc(NUM_POINTS * sizeof(union
QuantumInfo));

    for (int i = 0; i < NUM_POINTS; i++) {
        // Assign amplitude and phase to the wavefunction based on data passed in
        state->wavefunction[i].amplitude = amplitudeData[i];
        state->wavefunction[i].phase = phaseData[i];
    }

    return state;
}

// Function to update quantum state dynamically (e.g., time evolution)
void evolveQuantumState(struct QuantumState *state, double time) {
    // For simplicity, we simulate a basic time evolution where phase shifts with time
    for (int i = 0; i < NUM_POINTS; i++) {
        state->wavefunction[i].phase += 0.1 * time;  // Update phase (just a simple
model)
    }
}

// Function to print the quantum state
void printQuantumState(struct QuantumState *state) {
    printf("Quantum State: %s\n", state->stateLabel);
    printf("Energy: %.2f\n", state->energy);
    printf("Wavefunction:\n");

    for (int i = 0; i < NUM_POINTS; i++) {
        printf("Point %d: Amplitude = %.2f, Phase = %.2f\n", i, state-
>wavefunction[i].amplitude, state->wavefunction[i].phase);
    }
}

int main() {
    // Sample data for quantum states
    double amplitudeData[NUM_POINTS] = {0.5, 0.7, 1.0, 0.8, 0.6, 0.3, 0.9, 0.4, 0.2, 0.1};
    double phaseData[NUM_POINTS] = {0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5};

    // Create a quantum state for the "ground state"
    struct QuantumState *groundState = createQuantumState("Ground State", 0.0, amplitudeData, phaseData);

    // Print the initial state
    printQuantumState(groundState);

    // Simulate the evolution of the quantum state over time (e.g., t = 2.0)
    double time = 2.0;
    evolveQuantumState(groundState, time);

    // Print the evolved state
    printf("\nAfter %f units of time:\n", time);
    printQuantumState(groundState);

    // Free allocated memory
    free(groundState->wavefunction);
    free(groundState);

    return 0;
}
```

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>

#define MAX_RAYS 100

// Optical Element Type: Lens or Mirror
typedef enum {
    LENS,
    MIRROR
} OpticalElementType;

// Structure to represent an optical element
typedef struct {
    OpticalElementType type;
    double focal_length; // Focal length of lens or mirror
    double refractive_index; // Refractive index for lenses
    char name[50]; // Name or type of the optical element (for display purposes)
} OpticalElement;

// Structure to represent a light ray
typedef struct {
    double x; // Position along the x-axis
    double y; // Position along the y-axis
    double angle; // Angle of the ray in degrees
} LightRay;

// Array of light rays
LightRay lightRays[MAX_RAYS];

// Function to simulate light ray passing through optical elements
void simulateLightRays(OpticalElement* element, LightRay* rays, int numRays) {
    for (int i = 0; i < numRays; i++) {
        // Simple simulation logic based on type of optical element (Lens or Mirror)
        if (element->type == LENS) {
            // Ray passing through a lens
            printf("Ray %d passing through Lens %s with focal length %.2f\n", i+1,
element->name, element->focal_length);
            // Modify ray position and angle based on lens properties (simplified)
            rays[i].angle = rays[i].angle - (1 / element->focal_length);
        } else if (element->type == MIRROR) {
            // Ray reflecting from a mirror
            printf("Ray %d reflecting from Mirror %s with focal length %.2f\n", i+1,
element->name, element->focal_length);
            // Modify ray angle based on mirror properties (simplified)
            rays[i].angle = -rays[i].angle;
        }
        // Output the updated ray angle
        printf("Updated ray angle: %.2f degrees\n", rays[i].angle);
    }
}
```

```
// Function to dynamically allocate memory for optical elements
OpticalElement* createOpticalElement(OpticalElementType type, double
focalLength, double refractiveIndex, const char* name) {
    OpticalElement* newElement = (OpticalElement*)malloc(sizeof(OpticalElement));
    newElement->type = type;
    newElement->focal_length = focalLength;
    newElement->refractive_index = refractiveIndex;
    strncpy(newElement->name, name, sizeof(newElement->name) - 1);
    newElement->name[sizeof(newElement->name) - 1] = '\0'; // Ensure null-
termination
    return newElement;
}

// Function to simulate the entire optics system
void simulateOpticsSystem(OpticalElement** elements, int numElements) {
    for (int i = 0; i < numElements; i++) {
        simulateLightRays(elements[i], lightRays, MAX_RAYS);
    }
}

int main() {
    // Create some optical elements (Lens and Mirror)
    OpticalElement* lens1 = createOpticalElement(LENS, 50.0, 1.5, "Convex Lens");
    OpticalElement* mirror1 = createOpticalElement(MIRROR, 100.0, 1.0, "Concave
Mirror");

    // Array of pointers to optical elements
    OpticalElement* elements[] = {lens1, mirror1};

    // Initialize some light rays
    for (int i = 0; i < MAX_RAYS; i++) {
        lightRays[i].x = 0.0;
        lightRays[i].y = 0.0;
        lightRays[i].angle = i * 1.0; // Rays with different angles
    }

    // Simulate the optics system
    simulateOpticsSystem(elements, 2);

    // Free dynamically allocated memory
    free(lens1);
    free(mirror1);

    return 0;
}
```

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>

// Define a structure for storing thermodynamic properties
typedef struct {
    double pressure;     // Pressure (P)
    double volume;       // Volume (V)
    double temperature;  // Temperature (T)
    double entropy;      // Entropy (S), dependent property
    union {
        double energy;   // Internal energy (U), dependent property
        double enthalpy; // Enthalpy (H), dependent property
    };
} ThermoState;

// Function to calculate entropy based on P, V, T (simplified model)
double calculate_entropy(double P, double V, double T) {
    return (P * V) / T; // Example relationship (not physically accurate, for
demonstration)
}

// Function to calculate internal energy based on P, V, T (simplified model)
double calculate_energy(double P, double V, double T) {
    return P * V / T; // Example relationship (not physically accurate, for
demonstration)
}

// Function to create a state dynamically
ThermoState* create_state(double P, double V, double T) {
    ThermoState* state = (ThermoState*)malloc(sizeof(ThermoState));
    if (state == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    state->pressure = P;
    state->volume = V;
    state->temperature = T;
    state->entropy = calculate_entropy(P, V, T);
    state->energy = calculate_energy(P, V, T);
    return state;
}

// Function to free dynamically allocated state
void free_state(ThermoState* state) {
    free(state);
}

// Function to print thermodynamic state
void print_state(ThermoState* state) {
    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);
    printf("Energy: %.2f J\n", state->energy);
}

int main() {
    // Example of dynamically allocating memory for a state
    ThermoState* state1 = create_state(101325, 0.1, 300.0); // P = 101325 Pa, V =
0.1 m^3, T = 300K
```

```
    // Print the state information
    print_state(state1);

    // Free allocated memory
    free_state(state1);

    return 0;
}
```

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>

// Define a Union to store either energy released or product details
union ReactionDetails {
    double energyReleased;  // Energy released in the reaction
    char *productDetails;   // Product details (name of the product)
```

```
};

// Define a Structure for a nuclear reaction
struct NuclearReaction {
    char *reactant1;           // Name of the first reactant
    char *reactant2;           // Name of the second reactant (for fusion)
    union ReactionDetails details;  // Details (either energy or products)
    int isFission;            // Flag to differentiate between fission (0) or fusion (1)
};

// Function to create a new reaction dynamically
struct NuclearReaction* createReaction(char *reactant1, char *reactant2, double
energyReleased, int isFission) {
    struct NuclearReaction *reaction = (struct NuclearReaction *) malloc(sizeof(struct
NuclearReaction));

    reaction->reactant1 = (char *) malloc(strlen(reactant1) + 1);
    strcpy(reaction->reactant1, reactant1);

    if (isFission == 0) {
        reaction->reactant2 = NULL;  // Fission doesn't require a second reactant
    } else {
        reaction->reactant2 = (char *) malloc(strlen(reactant2) + 1);
        strcpy(reaction->reactant2, reactant2);
    }

    // Set energy released or product details
    if (isFission == 0) {
        reaction->details.energyReleased = energyReleased;
    } else {
        reaction->details.productDetails = (char *) malloc(100 * sizeof(char));  // Assume
product name is <100 chars
        strcpy(reaction->details.productDetails, "Helium, Neutron, Energy");
    }

    reaction->isFission = isFission;

    return reaction;
}

// Function to print the details of a reaction
void printReactionDetails(struct NuclearReaction *reaction) {
    if (reaction->isFission == 0) {
        printf("Fission Reaction:\n");
        printf("Reactant: %s + %s\n", reaction->reactant1, reaction->reactant2 ?
reaction->reactant2 : "N/A");
        printf("Energy Released: %.2lf MeV\n", reaction->details.energyReleased);
    } else {
        printf("Fusion Reaction:\n");
        printf("Reactant: %s + %s\n", reaction->reactant1, reaction->reactant2);
        printf("Products: %s\n", reaction->details.productDetails);
    }
}

// Function to free the memory allocated for a reaction
void freeReaction(struct NuclearReaction *reaction) {
    free(reaction->reactant1);
    if (reaction->reactant2) {
        free(reaction->reactant2);
    }
    if (reaction->isFission == 1) {
        free(reaction->details.productDetails);
    }
    free(reaction);
}

int main() {
    // Create a few example nuclear reactions
    struct NuclearReaction *fissionReaction = createReaction("Uranium-235",
"Neutron", 200.0, 0);
    struct NuclearReaction *fusionReaction = createReaction("Deuterium", "Tritium",
0.0, 1);

    // Print the reaction details
    printReactionDetails(fissionReaction);
    printf("\n");
    printReactionDetails(fusionReaction);

    // Free allocated memory
    freeReaction(fissionReaction);
    freeReaction(fusionReaction);

    return 0;
}
```

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 <math.h>

// Constants
#define G 6.67430e-11 // Gravitational constant (m^3 kg^−1 s^−2)

// Structure to store object properties (mass, position, field strength)
typedef struct {
    char label[50];        // Name/label of the object (e.g., "Planet A")
    double mass;           // Mass of the object in kilograms
    double position[3];    // Position of the object in space (x, y, z coordinates)
    double field_strength[3]; // Gravitational field strength at the object's position (x, y, z components)
} GravitationalObject;

// Union to handle either mass or field strength as parameters
typedef union {
    double mass;
    double field_strength[3];
} GravitationalParameter;

// Function to calculate gravitational field at a given point due to an object
```

```
void calculate_gravitational_field(GravitationalObject* obj, double* point, double* field) {
    // Vector from the object to the point
    double dx = point[0] - obj->position[0];
    double dy = point[1] - obj->position[1];
    double dz = point[2] - obj->position[2];

    // Calculate the distance between the object and the point
    double r = sqrt(dx * dx + dy * dy + dz * dz);

    // Gravitational field strength calculation
    if (r != 0) {
        double field_magnitude = G * obj->mass / (r * r);

        // Field components (directional)
        field[0] = field_magnitude * dx / r;
        field[1] = field_magnitude * dy / r;
        field[2] = field_magnitude * dz / r;
    } else {
        field[0] = field[1] = field[2] = 0;
    }
}

// Function to simulate and display the gravitational field at various points in space
void simulate_gravitational_field(GravitationalObject* objects, int num_objects, double** points, int num_points) {
    // Allocate memory for field values (dynamically)
    double** field_values = (double*)malloc(num_points * sizeof(double));
    for (int i = 0; i < num_points; i++) {
        field_values[i] = (double*)malloc(3 * sizeof(double)); // 3 components for x, y, z
    }

    // For each point, calculate the gravitational field due to all objects
    for (int i = 0; i < num_points; i++) {
        for (int j = 0; j < 3; j++) {
            field_values[i][j] = 0; // Initialize field to zero
        }

        // Sum the contributions of all objects to the field at this point
        for (int j = 0; j < num_objects; j++) {
            double field[3];
            calculate_gravitational_field(&objects[j], points[i], field);

            // Accumulate the field contributions
            for (int k = 0; k < 3; k++) {
                field_values[i][k] += field[k];
            }
        }

        // Display the resulting field at the current point
        printf("Gravitational field at point (%f, %f, %f): (%f, %f, %f)\n",
                points[i][0], points[i][1], points[i][2],
                field_values[i][0], field_values[i][1], field_values[i][2]);
    }

    // Free dynamically allocated memory
    for (int i = 0; i < num_points; i++) {
        free(field_values[i]);
    }
```

```
    free(field_values);
}


int main() {
    // Define some sample objects (e.g., planets, stars)
    GravitationalObject objects[2] = {
        {"Planet A", 5.972e24, {0, 0, 0}, {0, 0, 0}}, // Earth-like object
        {"Star B", 1.989e30, {1000, 1000, 1000}, {0, 0, 0}} // Sun-like object at (1000,
1000, 1000)
    };

    // Define some points in space where we want to calculate the gravitational field
    double points[2][3] = {
        {10, 10, 10},  // Point 1 (x, y, z)
        {200, 200, 200} // Point 2 (x, y, z)
    };

    // Call the simulation function
    simulate_gravitational_field(objects, 2, (double**)points, 2);

    return 0;
}
```

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 <math.h>

#define MAX_SOURCES 10  // You can define a limit for the number of sources

// Structure to hold wave properties (amplitude, wavelength, phase)
typedef struct {
    double amplitude;
    double wavelength;
    double phase;
} WaveProperties;

// Union to store either amplitude or phase information
typedef union {
    double amplitude;
    double phase;
} WaveUnion;

// Structure to represent a single wave source
typedef struct {
    char* label;              // Label for the wave source
    WaveProperties properties; // Wave properties: amplitude, wavelength, phase
    WaveUnion wave_info;      // Union for amplitude or phase information
```

```
} WaveSource;

// Double pointer to manage dynamic allocation for an array of wave sources
typedef struct {
    WaveSource** sources;      // Array of wave sources
    int num_sources;           // Number of sources
} WaveSourceManager;

// Function to initialize the WaveSourceManager
WaveSourceManager* createWaveSourceManager(int num_sources) {
    WaveSourceManager* manager =
(WaveSourceManager*)malloc(sizeof(WaveSourceManager));
    manager->sources = (WaveSource*)malloc(num_sources * sizeof(WaveSource));
    manager->num_sources = num_sources;
    return manager;
}

// Function to initialize a single wave source
void initWaveSource(WaveSource* source, const char* label, double amplitude,
double wavelength, double phase) {
    source->label = label;
    source->properties.amplitude = amplitude;
    source->properties.wavelength = wavelength;
    source->properties.phase = phase;
}

// Function to calculate interference pattern at a point (for simplicity)
double calculateInterferencePattern(WaveSourceManager* manager, double
position) {
    double total_amplitude = 0.0;
    double total_phase = 0.0;

    for (int i = 0; i < manager->num_sources; i++) {
        WaveSource* source = manager->sources[i];

        // Calculate interference for each wave based on its properties
        total_amplitude += source->properties.amplitude * cos(source-
>properties.phase + (2 * M_PI * position / source->properties.wavelength));
        total_phase += source->properties.phase;
    }

    // In this simple case, we return the total amplitude
    return total_amplitude;
}

// Clean up memory for wave sources
void freeWaveSourceManager(WaveSourceManager* manager) {
    free(manager->sources);
    free(manager);
}

int main() {
    // Create a manager for 2 wave sources
    WaveSourceManager* manager = createWaveSourceManager(2);

    // Initialize wave sources
    WaveSource wave1;
    initWaveSource(&wave1, "Wave1", 1.0, 2.0, 0.0);
    WaveSource wave2;
    initWaveSource(&wave2, "Wave2", 0.5, 3.0, M_PI / 2);

    // Assign sources to the manager
    manager->sources[0] = &wave1;
    manager->sources[1] = &wave2;

    // Analyze interference pattern at position x = 5.0
    double interference = calculateInterferencePattern(manager, 5.0);
    printf("Interference pattern at x = 5.0: %f\n", interference);

    // Clean up
    freeWaveSourceManager(manager);

    return 0;
}
```

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.

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

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

// Union to handle temperature-dependent properties
union TempDependentProperties {
    double curieTemperature;  // Curie temperature (for ferromagnetic materials)
    double temperatureSaturation;  // Temperature-dependent saturation value
};

// Structure to represent a magnetic material's properties
typedef struct {
    char name[50];                       // Material name (e.g., "Iron")
    double permeability;                 // Permeability (μ)
    double saturation;                   // Saturation (B_s)
    union TempDependentProperties tempProperties;  // Union for temperature-related
properties
} Material;

// Function to create a new material
Material* createMaterial(const char* name, double permeability, double saturation,
double curieTemp) {
    Material* newMaterial = (Material*)malloc(sizeof(Material));  // Dynamically
allocate memory for a new material
    if (newMaterial == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    strncpy(newMaterial->name, name, sizeof(newMaterial->name));  // Copy the
name
    newMaterial->permeability = permeability;                    // Set permeability
    newMaterial->saturation = saturation;                        // Set saturation
    newMaterial->tempProperties.curieTemperature = curieTemp;     // Set Curie
temperature
    return newMaterial;
}

// Function to display material information
void displayMaterial(const Material* material) {
    printf("Material: %s\n", material->name);
    printf("Permeability: %.2lf\n", material->permeability);
    printf("Saturation: %.2lf\n", material->saturation);
    printf("Curie Temperature: %.2lf K\n", material-
>tempProperties.curieTemperature);
}

// Function to free dynamically allocated memory for materials
void freeMaterial(Material* material) {
    free(material);  // Free the memory allocated for the material
}

int main() {
    // Dynamically allocate memory for an array of materials (using double pointer)
    Material** materials = (Material*)malloc(3 * sizeof(Material));  // Example with 3
materials
    if (materials == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Create materials and store in the array
    materials[0] = createMaterial("Iron", 1.26e-6, 2.2, 1043.0);
    materials[1] = createMaterial("Nickel", 6.0e-6, 0.6, 358.0);
    materials[2] = createMaterial("Cobalt", 1.2e-5, 1.4, 1388.0);

    // Display all materials
    for (int i = 0; i < 3; i++) {
        displayMaterial(materials[i]);
        printf("\n");
    }

    // Free the dynamically allocated memory for each material
    for (int i = 0; i < 3; i++) {
        freeMaterial(materials[i]);
    }

    // Free the array of pointers
    free(materials);

    return 0;
}
```

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>

#define MAX_PLASMA_TYPES 5

// Define a union for storing either density or temperature data.
typedef union {
    double density;
    double temperature;
} PlasmaData;

// Structure to represent plasma parameters.
typedef struct {
    char type[50];        // Plasma type (e.g., "Ionized Gas", "Hot Plasma")
    double electricField; // Electric field in the plasma
    PlasmaData data;      // Either density or temperature
} Plasma;

// Structure to hold simulation results.
typedef struct {
    Plasma* plasmas;      // Array of plasma objects
    size_t numPlasmas;    // Number of plasmas in the simulation
} PlasmaSimulation;

// Function to initialize the plasma array.
```

```
void initPlasmaSimulation(PlasmaSimulation* sim, size_t numPlasmas) {
    sim->plasmas = (Plasma*)malloc(numPlasmas * sizeof(Plasma));
    sim->numPlasmas = numPlasmas;
    if (!sim->plasmas) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}

// Function to set plasma data.
void setPlasmaData(Plasma* plasma, const char* type, double electricField, double
value, int isDensity) {
    strncpy(plasma->type, type, sizeof(plasma->type) - 1);
    plasma->electricField = electricField;
    if (isDensity) {
        plasma->data.density = value;
    } else {
        plasma->data.temperature = value;
    }
}

// Function to display the plasma simulation results.
void displaySimulationResults(const PlasmaSimulation* sim) {
    for (size_t i = 0; i < sim->numPlasmas; ++i) {
        printf("Plasma Type: %s\n", sim->plasmas[i].type);
        printf("Electric Field: %.2f\n", sim->plasmas[i].electricField);
        if (sim->plasmas[i].data.density) {
            printf("Density: %.2f\n", sim->plasmas[i].data.density);
        } else {
            printf("Temperature: %.2f\n", sim->plasmas[i].data.temperature);
        }
        printf("----------------------\n");
    }
}

// Function to free the memory used by the simulation.
void freePlasmaSimulation(PlasmaSimulation* sim) {
    free(sim->plasmas);
}

int main() {
    PlasmaSimulation sim;
    size_t numPlasmas = 3;

    // Initialize simulation with 3 plasma types.
    initPlasmaSimulation(&sim, numPlasmas);

    // Set data for each plasma (density for first and temperature for others).
    setPlasmaData(&sim.plasmas[0], "Ionized Gas", 5.0, 1.2, 1); // Set density
    setPlasmaData(&sim.plasmas[1], "Hot Plasma", 10.0, 15000.0, 0); // Set
temperature
    setPlasmaData(&sim.plasmas[2], "Cold Plasma", 8.0, 1000.0, 0); // Set
temperature

    // Display simulation results.
    displaySimulationResults(&sim);

    // Free the dynamically allocated memory.
    freePlasmaSimulation(&sim);
    return 0;
}
```

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>

// Structure to represent object properties (initial velocity, acceleration, displacement)
typedef struct {
    double initial_velocity;  // Initial velocity (m/s)
    double acceleration;      // Acceleration (m/s^2)
    double displacement;      // Displacement (m)
} ObjectProperties;

// Union to handle either velocity or displacement equations
typedef union {
    double velocity;         // Final velocity (m/s)
    double displacement;     // Displacement (m)
```

```c
} MotionData;

// Structure to represent kinematics equations (using both structure and union)
typedef struct {
    ObjectProperties obj_props;  // Object properties
    double time;                 // Time (s)
    MotionData motion;           // Union for either velocity or displacement
} KinematicsSolver;

// Function to solve for final velocity (v = u + at)
double solve_velocity(KinematicsSolver *ks) {
    return ks->obj_props.initial_velocity + ks->obj_props.acceleration * ks->time;
}

// Function to solve for displacement (s = ut + 0.5 * a * t^2)
double solve_displacement(KinematicsSolver *ks) {
    return ks->obj_props.initial_velocity * ks->time + 0.5 * ks->obj_props.acceleration
* ks->time * ks->time;
}

// Function to display motion data
void display_motion_data(KinematicsSolver *ks) {
    printf("Initial Velocity (u): %.2f m/s\n", ks->obj_props.initial_velocity);
    printf("Acceleration (a): %.2f m/s^2\n", ks->obj_props.acceleration);
    printf("Time (t): %.2f s\n", ks->time);
}

int main() {
    // Dynamically allocate memory for motion data (array for storing times and
velocities)
```

```
    int num_samples = 5;
    double velocities = (double)malloc(num_samples * sizeof(double));
    if (velocities == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Sample kinematic values for the object
    KinematicsSolver ks = {
        .obj_props = {50.0, 9.8, 0.0},  // Initial velocity = 50 m/s, acceleration = 9.8
m/s^2, displacement = 0
        .time = 10.0
    };

    // Display initial object properties
    display_motion_data(&ks);

    // Calculate final velocity and displacement
    ks.motion.velocity = solve_velocity(&ks);
    ks.motion.displacement = solve_displacement(&ks);

    printf("\nFinal velocity (v): %.2f m/s\n", ks.motion.velocity);
    printf("Displacement (s): %.2f m\n", ks.motion.displacement);

    // Store velocity data at different time intervals
    for (int i = 0; i < num_samples; i++) {
        ks.time = (i + 1) * 2.0;  // Time steps at 2s intervals
        velocities[i] = solve_velocity(&ks);
    }
    // Display velocities at different time intervals
    printf("\nVelocities at different time intervals:\n");
    for (int i = 0; i < num_samples; i++) {
        printf("t = %.2f s, v = %.2f m/s\n", (i + 1) * 2.0, velocities[i]);
    }

    // Free dynamically allocated memory
    free(velocities);

    return 0;
}
```

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>

// Define the maximum size for element names
```

```
#define MAX_ELEMENT_NAME 50


// Union to hold either wavelength or intensity information
union SpectralData {
    double wavelength;
    double intensity;
};


// Structure to store spectral line data
struct SpectralLine {
    char element[MAX_ELEMENT_NAME];  // Element name (e.g., "Hydrogen")
    union SpectralData data;         // Wavelength or Intensity
    int isWavelength;                // Flag to check if the data is wavelength (1) or intensity
(0)
};


// Function to create a new spectral line record dynamically
void createSpectralLine(struct SpectralLine **line, const char *element, double value,
int isWavelength) {
    *line = (struct SpectralLine *)malloc(sizeof(struct SpectralLine));  // Allocate
memory for a new record
    if (*line == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }


    // Set the element name
    strncpy((*line)->element, element, MAX_ELEMENT_NAME - 1);
    (*line)->element[MAX_ELEMENT_NAME - 1] = '\0'; // Ensure null termination
    // Store either wavelength or intensity based on the flag
    if (isWavelength) {
        (*line)->data.wavelength = value;
        (*line)->isWavelength = 1;
    } else {
        (*line)->data.intensity = value;
        (*line)->isWavelength = 0;
    }
}

// Function to display a spectral line record
void displaySpectralLine(const struct SpectralLine *line) {
    if (line->isWavelength) {
        printf("Element: %s, Wavelength: %.2f nm\n", line->element, line-
>data.wavelength);
    } else {
        printf("Element: %s, Intensity: %.2f\n", line->element, line->data.intensity);
    }
}

// Function to free allocated memory for spectral line records
void freeSpectralLine(struct SpectralLine *line) {
    free(line);
}

int main() {
    struct SpectralLine *line1, *line2;

    // Create a spectral line for Hydrogen with wavelength
    createSpectralLine(&line1, "Hydrogen", 656.3, 1); // Wavelength in nm
    // Create a spectral line for Oxygen with intensity
    createSpectralLine(&line2, "Oxygen", 12345.67, 0); // Intensity in arbitrary units

    // Display the spectral lines
    displaySpectralLine(line1);
    displaySpectralLine(line2);

    // Free allocated memory
    freeSpectralLine(line1);
    freeSpectralLine(line2);

    return 0;
}
```

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 <math.h>
```

```
#include <stdlib.h>

#define GRAVITY 9.81 // m/s^2

// Structure to hold projectile properties
typedef struct {
    double mass;     // Mass of the projectile (kg)
    double velocity; // Initial velocity (m/s)
    double angle;    // Launch angle (degrees)
} Projectile;

// Union to handle velocity or displacement data
typedef union {
    double velocity;
    double displacement;
} TrajectoryData;

// Structure to store trajectory data
typedef struct {
    double time;         // Time at this point in the trajectory
    double height;       // Height of the projectile at this time
    double horizontal;   // Horizontal displacement at this time
} TrajectoryPoint;

// Function to calculate the trajectory
TrajectoryPoint *calculateTrajectory(Projectile *p, int *pointsCount) {
    // Calculate the initial velocity components
    double angle_rad = p->angle * (M_PI / 180.0);
    double v_x = p->velocity * cos(angle_rad);
    }

    double v_y = p->velocity * sin(angle_rad);

    // Time of flight (T)
    double flightTime = (2 * v_y) / GRAVITY;

    // Max number of points
    int maxPoints = 100;
    *pointsCount = maxPoints;

    // Dynamically allocate memory for trajectory points
    TrajectoryPoint *trajectory = (TrajectoryPoint *)malloc(maxPoints *
sizeof(TrajectoryPoint));

    // Calculate the trajectory points
    for (int i = 0; i < maxPoints; i++) {
        double t = flightTime * i / (maxPoints - 1); // Normalize time

        // Calculate position at time t
        trajectory[i].time = t;
        trajectory[i].horizontal = v_x * t;
        trajectory[i].height = v_y * t - 0.5 * GRAVITY * t * t;

        // If the projectile hits the ground, stop calculating
        if (trajectory[i].height < 0) {
            *pointsCount = i;
            break;
        }
    }
    return trajectory;
}

// Function to print trajectory points
void printTrajectory(TrajectoryPoint *trajectory, int pointsCount) {
    printf("Time (s)\tHorizontal Displacement (m)\tHeight (m)\n");
    for (int i = 0; i < pointsCount; i++) {
        printf("%.2f\t\t%.2f\t\t\t\t%.2f\n", trajectory[i].time, trajectory[i].horizontal,
trajectory[i].height);
    }
}

int main() {
    // Initialize projectile properties
    Projectile p;
    p.mass = 1.0; // Mass (kg)
    p.velocity = 50.0; // Initial velocity (m/s)
    p.angle = 45.0; // Launch angle (degrees)

    // Calculate the trajectory
    int pointsCount = 0;
    TrajectoryPoint *trajectory = calculateTrajectory(&p, &pointsCount);

    // Print the results
    printTrajectory(trajectory, pointsCount);

    // Free dynamically allocated memory
    free(trajectory);
    return 0;
}
```

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.

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

#define MAX_MATERIALS 10

// Union to store dependent properties like yield stress or elastic modulus
union MaterialProperties {
    double yieldStress;
    double elasticModulus;
};

// Structure to represent the material's stress and strain data
typedef struct {
    char name[50];              // Material name (String)
    double* stress;             // Pointer to stress data (dynamically allocated)
    double* strain;             // Pointer to strain data (dynamically allocated)
    int dataPoints;             // Number of data points for stress-strain data
    union MaterialProperties properties; // Union to hold dependent material properties
} Material;

// Function to allocate and initialize a material's stress-strain data
void initMaterial(Material* material, const char* name, int dataPoints) {
    strncpy(material->name, name, sizeof(material->name) - 1);
    material->name[sizeof(material->name) - 1] = '\0'; // Ensure null-terminated string
    material->dataPoints = dataPoints;

    // Dynamically allocate memory for stress and strain data
    material->stress = (double*)malloc(dataPoints * sizeof(double));
    material->strain = (double*)malloc(dataPoints * sizeof(double));

    if (!material->stress || !material->strain) {
        printf("Memory allocation failed for stress or strain data.\n");
        exit(1);
    }
}

// Function to deallocate material data
void freeMaterial(Material* material) {
    free(material->stress);
    free(material->strain);
}
```

```
// Function to display stress-strain data
void displayStressStrainData(Material* material) {
    printf("Material: %s\n", material->name);
    printf("Stress (Pa) | Strain\n");
    for (int i = 0; i < material->dataPoints; i++) {
        printf("%.2f        | %.5f\n", material->stress[i], material->strain[i]);
    }
}

// Function to set yield stress or elastic modulus
void setMaterialProperties(Material* material, double value, int isElasticModulus) {
    if (isElasticModulus) {
        material->properties.elasticModulus = value;
    } else {
        material->properties.yieldStress = value;
    }
}

// Function to analyze stress-strain behavior (basic analysis: display yield stress or
modulus)
void analyzeStressStrain(Material* material) {
    printf("Analyzing stress-strain for material: %s\n", material->name);
    printf("Elastic Modulus: %.2f MPa\n", material->properties.elasticModulus);
    printf("Yield Stress: %.2f MPa\n", material->properties.yieldStress);
}

int main() {
    Material materials[MAX_MATERIALS];
```

```
// Example material 1
initMaterial(&materials[0], "Steel", 5);
materials[0].stress[0] = 100.0; materials[0].strain[0] = 0.002;
materials[0].stress[1] = 200.0; materials[0].strain[1] = 0.004;
materials[0].stress[2] = 300.0; materials[0].strain[2] = 0.006;
materials[0].stress[3] = 400.0; materials[0].strain[3] = 0.008;
materials[0].stress[4] = 500.0; materials[0].strain[4] = 0.01;

setMaterialProperties(&materials[0], 210000, 1);  // Elastic Modulus (MPa)
setMaterialProperties(&materials[0], 250, 0);  // Yield Stress (MPa)

displayStressStrainData(&materials[0]);
analyzeStressStrain(&materials[0]);

// Example material 2
initMaterial(&materials[1], "Aluminum", 5);
materials[1].stress[0] = 50.0; materials[1].strain[0] = 0.001;
materials[1].stress[1] = 100.0; materials[1].strain[1] = 0.002;
materials[1].stress[2] = 150.0; materials[1].strain[2] = 0.003;
materials[1].stress[3] = 200.0; materials[1].strain[3] = 0.004;
materials[1].stress[4] = 250.0; materials[1].strain[4] = 0.005;

setMaterialProperties(&materials[1], 70000, 1);  // Elastic Modulus (MPa)
setMaterialProperties(&materials[1], 150, 0);  // Yield Stress (MPa)

displayStressStrainData(&materials[1]);
analyzeStressStrain(&materials[1]);

// Clean up dynamic memory
```

```
    freeMaterial(&materials[0]);
    freeMaterial(&materials[1]);


    return 0;
}
```