**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>
* typedef struct {
* const int id;
* const double mass;
* double position[2];
* double velocity[2];
* } Particle;
* typedef union {
* double gravitational;
* double electric;
* double magnetic;
* } Force;
* void allocateParticles(Particle \*\*particles, int count) {
* \*particles = (Particle \*)malloc(count \* sizeof(Particle));
* }
* void initializeParticle(Particle \*particle, int id, double mass, double x, double y, double vx, double vy) {
* Particle temp = {id, mass, {x, y}, {vx, vy}};
* \*particle = temp; // Initialize a particle with given properties.
* }
* void freeParticles(Particle \*\*particles) {
* free(\*particles);
* \*particles = NULL;
* }
* int main() {
* int numParticles = 3;
* Particle \*particles = NULL;
* allocateParticles(&particles, numParticles);
* for (int i = 0; i < numParticles; i++) {
* initializeParticle(&particles[i], i + 1, 1.0 + i, i \* 1.0, i \* 1.5, 0.5 \* i, 0.3 \* i);
* }
* const char \*forceTypes[] = {"Gravitational", "Electric", "Magnetic"};
* Force force;
* force.gravitational = 9.8; // Example force value
* printf("Particle System:\n");
* for (int i = 0; i < numParticles; i++) {
* printf("ID: %d, Mass: %.2f, Position: (%.2f, %.2f), Velocity: (%.2f, %.2f)\n",
* particles[i].id, particles[i].mass, particles[i].position[0], particles[i].position[1],
* particles[i].velocity[0], particles[i].velocity[1]);
* }
* printf("\nForce Type: %s, Value: %.2f\n", forceTypes[0], force.gravitational);
* freeParticles(&particles);
* return 0;
* }

**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 <math.h>
* typedef struct {
* const double electricField[3];
* const double magneticField[3];
* const double position[3];
* } FieldPoint;
* typedef union {
* double electric[3];
* double magnetic[3];
* } FieldComponent;
* void allocateFieldGrid(FieldPoint \*\*fieldGrid, int numPoints) {
* \*fieldGrid = (FieldPoint \*)malloc(numPoints \* sizeof(FieldPoint));
* }
* void initializeFieldPoint(FieldPoint \*point, double ex, double ey, double ez, double bx, double by, double bz, double x, double y, double z) {
* FieldPoint temp = {{ex, ey, ez}, {bx, by, bz}, {x, y, z}};
* \*point = temp;
* }
* void freeFieldGrid(FieldPoint \*\*fieldGrid) {
* free(\*fieldGrid);
* \*fieldGrid = NULL;
* }
* int main() {
* int numPoints = 3;
* FieldPoint \*fieldGrid = NULL;
* allocateFieldGrid(&fieldGrid, numPoints);
* for (int i = 0; i < numPoints; i++) {
* initializeFieldPoint(
* &fieldGrid[i],
* i \* 1.0, i \* 2.0, i \* 3.0, // Electric field (Ex, Ey, Ez)
* i \* 0.5, i \* 1.5, i \* 2.5, // Magnetic field (Bx, By, Bz)
* i \* 1.0, i \* 2.0, i \* 3.0  // Position (x, y, z)
* );
* }
* const char \*coordinateSystems[] = {"Cartesian", "Cylindrical", "Spherical"};
* FieldComponent field;
* field.electric[0] = 10.0;
* field.electric[1] = 5.0;
* field.electric[2] = 0.0;
* printf("Electromagnetic Field Grid:\n");
* for (int i = 0; i < numPoints; i++) {
* printf("Point %d: Position(%.2f, %.2f, %.2f), Electric(%.2f, %.2f, %.2f), Magnetic(%.2f, %.2f, %.2f)\n",
* i + 1,
* fieldGrid[i].position[0], fieldGrid[i].position[1], fieldGrid[i].position[2],
* fieldGrid[i].electricField[0], fieldGrid[i].electricField[1], fieldGrid[i].electricField[2],
* fieldGrid[i].magneticField[0], fieldGrid[i].magneticField[1], fieldGrid[i].magneticField[2]);
* }
* printf("\nCoordinate System: %s\n", coordinateSystems[0]);
* printf("Sample Electric Field Component: (%.2f, %.2f, %.2f)\n", field.electric[0], field.electric[1], field.electric[2]);
* freeFieldGrid(&fieldGrid);
* 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>
* typedef struct {
* const char \*elementName;
* const double \*energyLevels;
* const double \*transitionProbabilities;
* int numLevels;
* } Atom;
* typedef union {
* double groundState;
* double excitedState;
* } EnergyState;
* void addAtom(Atom \*\*atoms, int \*count, const char \*name, const double \*levels, const double \*probabilities, int numLevels) {
* \*atoms = (Atom \*)realloc(\*atoms, (\*count + 1) \* sizeof(Atom));
* Atom newAtom = {name, levels, probabilities, numLevels};
* (\*atoms)[\*count] = newAtom;
* (\*count)++;
* }
* void displayAtoms(const Atom \*atoms, int count) {
* printf("Atomic Energy Levels:\n");
* for (int i = 0; i < count; i++) {
* printf("Element: %s\n", atoms[i].elementName);
* printf("Energy Levels: ");
* for (int j = 0; j < atoms[i].numLevels; j++) {
* printf("%.2f ", atoms[i].energyLevels[j]);
* }
* printf("\nTransition Probabilities: ");
* for (int j = 0; j < atoms[i].numLevels - 1; j++) {
* printf("%.2f ", atoms[i].transitionProbabilities[j]);
* }
* printf("\n");
* }
* }
* int main() {
* Atom \*atoms = NULL;
* int count = 0;
* double energyLevels1[] = {0.0, 10.2, 20.4};
* double transitionProbabilities1[] = {0.5, 0.7};
* double energyLevels2[] = {0.0, 15.0, 30.0};
* double transitionProbabilities2[] = {0.6, 0.8};
* addAtom(&atoms, &count, "Hydrogen", energyLevels1, transitionProbabilities1, 3);
* addAtom(&atoms, &count, "Helium", energyLevels2, transitionProbabilities2, 3);
* displayAtoms(atoms, count);
* free(atoms);
* 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>
* typedef struct {
* const char \*stateLabel;
* const double \*wavefunction;
* double energy;
* int numPoints;
* } QuantumState;
* typedef union {
* double amplitude;
* double phase;
* } StateInfo;
* void addQuantumState(QuantumState \*\*states, int \*count, const char \*label, const double \*wavefunction, double energy, int numPoints) {
* \*states = (QuantumState \*)realloc(\*states, (\*count + 1) \* sizeof(QuantumState));
* QuantumState newState = {label, wavefunction, energy, numPoints};
* (\*states)[\*count] = newState;
* (\*count)++;
* }
* void displayQuantumStates(const QuantumState \*states, int count) {
* printf("Quantum States:\n");
* for (int i = 0; i < count; i++) {
* printf("State Label: %s\n", states[i].stateLabel);
* printf("Wavefunction: ");
* for (int j = 0; j < states[i].numPoints; j++) {
* printf("%.2f ", states[i].wavefunction[j]);
* }
* printf("\nEnergy: %.2f\n", states[i].energy);
* }
* }
* int main() {
* QuantumState \*states = NULL;
* int count = 0;
* double wavefunction1[] = {0.1, 0.2, 0.3, 0.4};
* double wavefunction2[] = {0.5, 0.6, 0.7, 0.8};
* addQuantumState(&states, &count, "Ground State", wavefunction1, 1.0, 4);
* addQuantumState(&states, &count, "Excited State", wavefunction2, 2.0, 4);
* displayQuantumStates(states, count);
* free(states);
* 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>
* typedef struct {
* const char \*elementType;
* double refractiveIndex;
* double focalLength;
* } OpticalElement;
* typedef union {
* double lensFocalLength;
* double mirrorRadiusOfCurvature;
* } ElementParameters;
* void addOpticalElement(OpticalElement \*\*elements, int \*count, const char \*type, double refractiveIndex, double focalLength) {
* \*elements = (OpticalElement \*)realloc(\*elements, (\*count + 1) \* sizeof(OpticalElement));
* OpticalElement newElement = {type, refractiveIndex, focalLength};
* (\*elements)[\*count] = newElement;
* (\*count)++;
* }
* void displayOpticalElements(const OpticalElement \*elements, int count) {
* printf("Optical Elements:\n");
* for (int i = 0; i < count; i++) {
* printf("Element Type: %s\n", elements[i].elementType);
* printf("Refractive Index: %.2f\n", elements[i].refractiveIndex);
* printf("Focal Length: %.2f\n", elements[i].focalLength);
* }
* }
* int main() {
* OpticalElement \*elements = NULL;
* int count = 0;
* addOpticalElement(&elements, &count, "Lens", 1.5, 10.0);
* addOpticalElement(&elements, &count, "Mirror", 0.0, 15.0);
* displayOpticalElements(elements, count);
* free(elements);
* 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>
* typedef struct {
* const char \*description;
* double pressure;
* double volume;
* double temperature;
* double entropy;
* } ThermodynamicState;
* typedef union {
* double internalEnergy;
* double entropy;
* } DependentProperties;
* void addState(ThermodynamicState \*\*states, int \*count, const char \*desc, double pressure, double volume, double temperature, double entropy) {
* \*states = (ThermodynamicState \*)realloc(\*states, (\*count + 1) \* sizeof(ThermodynamicState));
* ThermodynamicState newState = {desc, pressure, volume, temperature, entropy};
* (\*states)[\*count] = newState;
* (\*count)++;
* }
* void displayStates(const ThermodynamicState \*states, int count) {
* printf("Thermodynamic States:\n");
* for (int i = 0; i < count; i++) {
* printf("Description: %s\n", states[i].description);
* printf("Pressure: %.2f\n", states[i].pressure);
* printf("Volume: %.2f\n", states[i].volume);
* printf("Temperature: %.2f\n", states[i].temperature);
* printf("Entropy: %.2f\n", states[i].entropy);
* }
* }
* int main() {
* ThermodynamicState \*states = NULL;
* int count = 0;
* addState(&states, &count, "State 1", 101.3, 1.0, 300.0, 1.5);
* addState(&states, &count, "State 2", 202.6, 2.0, 400.0, 2.0);
* displayStates(states, count);
* free(states);
* 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>
* typedef struct {
* const char \*reactants;
* const char \*products;
* double energyReleased;
* } NuclearReaction;
* typedef union {
* double energy;
* const char \*details;
* } ReactionInfo;
* void addReaction(NuclearReaction \*\*reactions, int \*count, const char \*reactants, const char \*products, double energy) {
* \*reactions = (NuclearReaction \*)realloc(\*reactions, (\*count + 1) \* sizeof(NuclearReaction));
* NuclearReaction newReaction = {reactants, products, energy};
* (\*reactions)[\*count] = newReaction;
* (\*count)++;
* }
* void displayReactions(const NuclearReaction \*reactions, int count) {
* printf("Nuclear Reactions:\n");
* for (int i = 0; i < count; i++) {
* printf("Reactants: %s\n", reactions[i].reactants);
* printf("Products: %s\n", reactions[i].products);
* printf("Energy Released: %.2f MeV\n", reactions[i].energyReleased);
* }
* }
* int main() {
* NuclearReaction \*reactions = NULL;
* int count = 0;
* addReaction(&reactions, &count, "U-235 + n", "Ba-141 + Kr-92 + 3n", 200.0);
* addReaction(&reactions, &count, "D + T", "He-4 + n", 17.6);
* displayReactions(reactions, count);
* free(reactions);
* 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>
* #define G 6.67430e-11
* typedef struct {
* const char \*objectLabel;
* double mass;
* double position[3];  // x, y, z coordinates
* double fieldStrength;
* } GravitationalObject;
* typedef union {
* double mass;
* double fieldStrength;
* } FieldParams;
* void calculateGravitationalField(GravitationalObject \*obj, double \*field, int numPoints) {
* for (int i = 0; i < numPoints; i++) {
* double distance = obj->position[i];
* if (distance != 0) {
* field[i] = G \* obj->mass / (distance \* distance);
* } else {
* field[i] = 0;
* }
* }
* }
* void displayGravitationalField(const GravitationalObject \*obj, double \*field, int numPoints) {
* printf("Gravitational Field of %s:\n", obj->objectLabel);
* for (int i = 0; i < numPoints; i++) {
* printf("Point %d: Field Strength = %.2e N\n", i + 1, field[i]);
* }
* }
* int main() {
* GravitationalObject \*obj = (GravitationalObject \*)malloc(sizeof(GravitationalObject));
* obj->objectLabel = "Planet A";
* obj->mass = 5.972e24;  // Mass of Earth in kg
* obj->position[0] = 1.0e7; // Distance 1
* obj->position[1] = 1.5e7; // Distance 2
* obj->position[2] = 2.0e7; // Distance 3
* int numPoints = 3;
* double \*field = (double \*)malloc(numPoints \* sizeof(double));
* calculateGravitationalField(obj, field, numPoints);
* displayGravitationalField(obj, field, numPoints);
* free(obj);
* free(field);
* 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>
* typedef struct {
* const char \*sourceLabel;
* double amplitude;
* double wavelength;
* double phase;
* } WaveSource;
* typedef union {
* double amplitude;
* double phase;
* } WaveProperty;
* void calculateInterferencePattern(WaveSource \*sources, double \*interferenceData, int numSources, int numPoints) {
* for (int i = 0; i < numPoints; i++) {
* interferenceData[i] = 0.0;
* for (int j = 0; j < numSources; j++) {
* double waveEffect = sources[j].amplitude \* cos((2 \* M\_PI / sources[j].wavelength) \* i + sources[j].phase);
* interferenceData[i] += waveEffect;
* }
* }
* }
* void displayInterferencePattern(double \*interferenceData, int numPoints) {
* printf("Wave Interference Pattern:\n");
* for (int i = 0; i < numPoints; i++) {
* printf("Point %d: Interference = %.2f\n", i + 1, interferenceData[i]);
* }
* }
* int main() {
* int numSources = 2;
* int numPoints = 10;
* WaveSource \*sources = (WaveSource \*)malloc(numSources \* sizeof(WaveSource));
* sources[0].sourceLabel = "Source 1";
* sources[0].amplitude = 1.0;
* sources[0].wavelength = 2.0;
* sources[0].phase = 0.0;
* sources[1].sourceLabel = "Source 2";
* sources[1].amplitude = 1.0;
* sources[1].wavelength = 2.5;
* sources[1].phase = M\_PI / 2.0;
* double \*interferenceData = (double \*)malloc(numPoints \* sizeof(double));
* calculateInterferencePattern(sources, interferenceData, numSources, numPoints);
* displayInterferencePattern(interferenceData, numPoints);
* free(sources);
* free(interferenceData);
* 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>
* typedef struct {
* const char \*materialName;
* double permeability;
* double saturation;
* } MagneticMaterial;
* typedef union {
* double temperature;
* double propertyValue;
* } TemperatureDependentProperty;
* void displayMaterialProperties(MagneticMaterial \*materials, int numMaterials) {
* for (int i = 0; i < numMaterials; i++) {
* printf("Material: %s\n", materials[i].materialName);
* printf("Permeability: %.2f\n", materials[i].permeability);
* printf("Saturation: %.2f\n", materials[i].saturation);
* printf("-----------------------------\n");
* }
* }
* int main() {
* int numMaterials = 3;
* MagneticMaterial \*materials = (MagneticMaterial \*)malloc(numMaterials \* sizeof(MagneticMaterial));
* materials[0].materialName = "Material A";
* materials[0].permeability = 1.5;
* materials[0].saturation = 2.5;
* materials[1].materialName = "Material B";
* materials[1].permeability = 3.0;
* materials[1].saturation = 4.5;
* materials[2].materialName = "Material C";
* materials[2].permeability = 2.1;
* materials[2].saturation = 3.6;
* displayMaterialProperties(materials, numMaterials);
* 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>
* typedef struct {
* double density;
* double temperature;
* double electricField;
* } Plasma;
* typedef union {
* double density;
* double temperature;
* } PlasmaData;
* void displayPlasmaSimulationResults(Plasma \*plasmas, int numPlasmas) {
* for (int i = 0; i < numPlasmas; i++) {
* printf("Plasma %d:\n", i + 1);
* printf("Density: %.2f\n", plasmas[i].density);
* printf("Temperature: %.2f\n", plasmas[i].temperature);
* printf("Electric Field: %.2f\n", plasmas[i].electricField);
* printf("------------------------------\n");
* }
* }
* int main() {
* int numPlasmas = 3;
* Plasma \*plasmas = (Plasma \*)malloc(numPlasmas \* sizeof(Plasma));
* plasmas[0].density = 1.5;
* plasmas[0].temperature = 5000;
* plasmas[0].electricField = 200.0;
* plasmas[1].density = 2.3;
* plasmas[1].temperature = 7000;
* plasmas[1].electricField = 250.0;
* plasmas[2].density = 1.8;
* plasmas[2].temperature = 6000;
* plasmas[2].electricField = 220.0;
* displayPlasmaSimulationResults(plasmas, numPlasmas);
* free(plasmas);
* 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>
* typedef struct {
* double initialVelocity;
* double acceleration;
* double displacement;
* } ObjectMotion;
* typedef union {
* double velocity;
* double displacement;
* } MotionEquation;
* void calculateKinematics(ObjectMotion \*object, double time) {
* object->displacement = object->initialVelocity \* time + 0.5 \* object->acceleration \* time \* time;
* }
* void displayMotionData(ObjectMotion \*objects, int numObjects) {
* for (int i = 0; i < numObjects; i++) {
* printf("Object %d:\n", i + 1);
* printf("Initial Velocity: %.2f m/s\n", objects[i].initialVelocity);
* printf("Acceleration: %.2f m/s^2\n", objects[i].acceleration);
* printf("Displacement after time: %.2f m\n", objects[i].displacement);
* printf("------------------------------\n");
* }
* }
* int main() {
* int numObjects = 3;
* double time = 5.0;
* ObjectMotion \*objects = (ObjectMotion \*)malloc(numObjects \* sizeof(ObjectMotion));
* objects[0].initialVelocity = 10.0;
* objects[0].acceleration = 2.0;
* objects[1].initialVelocity = 15.0;
* objects[1].acceleration = 3.0;
* objects[2].initialVelocity = 20.0;
* objects[2].acceleration = 1.5;
* for (int i = 0; i < numObjects; i++) {
* calculateKinematics(&objects[i], time);
* }
* displayMotionData(objects, numObjects);
* free(objects);
* 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>
* #define MAX\_ELEMENTS 5
* #define MAX\_NAME\_LENGTH 20
* // Structure to store spectral line properties
* struct SpectralLine {
* float wavelength;
* float intensity;
* char element[MAX\_NAME\_LENGTH];
* };
* // Union to store either intensity or wavelength
* union LineData {
* float intensity;
* float wavelength;
* };
* // Function to dynamically allocate memory for spectral lines
* void allocateSpectralLines(struct SpectralLine \*\*lines, int count) {
* \*lines = (struct SpectralLine \*)malloc(count \* sizeof(struct SpectralLine));
* }
* // Function to free the dynamically allocated memory
* void freeSpectralLines(struct SpectralLine \*lines) {
* free(lines);
* }
* int main() {
* struct SpectralLine \*spectralLines;
* int lineCount = 3;
* // Dynamically allocate memory for the spectral line records
* allocateSpectralLines(&spectralLines, lineCount);
* // Fill in spectral line data
* spectralLines[0].wavelength = 650.0f;
* spectralLines[0].intensity = 500.0f;
* // Manually assign element names without snprintf
* char hydrogen[] = "Hydrogen";
* for (int i = 0; i < MAX\_NAME\_LENGTH && hydrogen[i] != '\0'; i++) {
* spectralLines[0].element[i] = hydrogen[i];
* }
* spectralLines[1].wavelength = 485.0f;
* spectralLines[1].intensity = 400.0f;
* // Manually assign element names without snprintf
* char oxygen[] = "Oxygen";
* for (int i = 0; i < MAX\_NAME\_LENGTH && oxygen[i] != '\0'; i++) {
* spectralLines[1].element[i] = oxygen[i];
* }
* spectralLines[2].wavelength = 589.0f;
* spectralLines[2].intensity = 600.0f;
* // Manually assign element names without snprintf
* char sodium[] = "Sodium";
* for (int i = 0; i < MAX\_NAME\_LENGTH && sodium[i] != '\0'; i++) {
* spectralLines[2].element[i] = sodium[i];
* }
* // Output the spectral line data
* for (int i = 0; i < lineCount; i++) {
* printf("Element: %s, Wavelength: %.2f nm, Intensity: %.2f\n",
* spectralLines[i].element,
* spectralLines[i].wavelength,
* spectralLines[i].intensity);
* }
* // Free the dynamically allocated memory
* freeSpectralLines(spectralLines);
* 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 <stdlib.h>
* #include <math.h>
* #define MAX\_TRAJECTORIES 5
* struct Projectile {
* float mass;
* float velocity;
* float angle;
* };
* union MotionData {
* float velocity;
* float displacement;
* };
* void calculateTrajectory(struct Projectile \*projectile, float \*trajectoryData, int numPoints) {
* float g = 9.81;
* float angleRad = projectile->angle \* M\_PI / 180.0;
* for (int i = 0; i < numPoints; i++) {
* float time = i \* 0.1;
* float x = projectile->velocity \* cos(angleRad) \* time;
* float y = projectile->velocity \* sin(angleRad) \* time - 0.5 \* g \* time \* time;
* trajectoryData[i] = y;
* if (y < 0) {
* break;
* }
* }
* }
* int main() {
* struct Projectile \*projectiles;
* int numProjectiles = 1;
* int numPoints = 50;
* projectiles = (struct Projectile \*)malloc(numProjectiles \* sizeof(struct Projectile));
* projectiles[0].mass = 1.0f;
* projectiles[0].velocity = 50.0f;
* projectiles[0].angle = 45.0f;
* float \*trajectoryData = (float \*)malloc(numPoints \* sizeof(float));
* calculateTrajectory(&projectiles[0], trajectoryData, numPoints);
* for (int i = 0; i < numPoints; i++) {
* if (trajectoryData[i] < 0) break;
* printf("Time: %.1fs, Displacement: %.2fm\n", i \* 0.1, trajectoryData[i]);
* }
* free(projectiles);
* free(trajectoryData);
* 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>
* struct Material {
* char name[50];
* float stress;
* float strain;
* float modulus;
* };
* union MaterialProperties {
* float yieldStress;
* float elasticModulus;
* };
* void analyzeStressStrain(struct Material \*material, float \*stressStrainData, int dataPoints) {
* for (int i = 0; i < dataPoints; i++) {
* stressStrainData[i] = material->stress \* material->strain / material->modulus;
* }
* }
* int main() {
* struct Material \*materials;
* int numMaterials = 1;
* int dataPoints = 5;
* materials = (struct Material \*)malloc(numMaterials \* sizeof(struct Material));
* materials[0].stress = 100.0f;
* materials[0].strain = 0.01f;
* materials[0].modulus = 200.0f;
* float \*stressStrainData = (float \*)malloc(dataPoints \* sizeof(float));
* analyzeStressStrain(&materials[0], stressStrainData, dataPoints);
* for (int i = 0; i < dataPoints; i++) {
* printf("Stress-Strain Ratio at Point %d: %.2f\n", i + 1, stressStrainData[i]);
* }
* free(materials);
* free(stressStrainData);
* return 0;
* }