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

union Force {
    char gravitational[50];
    char electric[50];
    char magnetic[50];
};

typedef struct {
    float mass;
    int position[2];
    float velocity[2];
    const char *properties;
    union Force force;
} Motion;

void add(Motion **motion, int *count);
void display(Motion *motion, int count);

int main() {
    Motion *motion = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Particle\n");
        printf("2. Display Particles\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&motion, &count);
                break;
            case 2:
                display(motion, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(motion);
    return 0;
}

void add(Motion **motion, int *count) {
    (*count)++;
    *motion = (Motion *)realloc(*motion, (*count) * sizeof(Motion));
    if (*motion == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    Motion *newParticle = &(*motion)[*count - 1];

    printf("\nEnter the mass: ");
    scanf("%f", &newParticle->mass);

    printf("Enter the position (x y): ");
    scanf("%d %d", &newParticle->position[0], &newParticle->position[1]);

    printf("Enter velocity (vx vy): ");
    scanf("%f %f", &newParticle->velocity[0], &newParticle->velocity[1]);

    printf("Enter force type (gravitational/electric/magnetic): ");
    char forceType[50];
    scanf("%s", forceType);

    if (strcmp(forceType, "gravitational") == 0) {
        strcpy(newParticle->force.gravitational, "Gravitational Force");
    } else if (strcmp(forceType, "electric") == 0) {
        strcpy(newParticle->force.electric, "Electric Force");
    } else if (strcmp(forceType, "magnetic") == 0) {
        strcpy(newParticle->force.magnetic, "Magnetic Force");
    } else {
        printf("Invalid force type! Setting to 'Unknown Force'.\n");
        strcpy(newParticle->force.gravitational, "Unknown Force");
    }

    newParticle->properties = "Particle Properties";

    printf("Particles Added Sucessfully\n");
}

void display(Motion *motion, int count) {
    if (count == 0) {
        printf("\nNo particles to display.\n");
        return;
    }

    printf("\nParticle Information:\n");
    printf("--------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nParticle %d:\n", i + 1);
        printf("Mass: %.2f\n", motion[i].mass);
        printf("Position: (%d, %d)\n", motion[i].position[0], motion[i].position[1]);
        printf("Velocity: (%.2f, %.2f)\n", motion[i].velocity[0], motion[i].velocity[1]);
        if (strlen(motion[i].force.gravitational) > 0) {
            printf("Force: %s\n", motion[i].force.gravitational);
        } else if (strlen(motion[i].force.electric) > 0) {
            printf("Force: %s\n", motion[i].force.electric);
        } else if (strlen(motion[i].force.magnetic) > 0) {
            printf("Force: %s\n", motion[i].force.magnetic);
        }

        printf("Properties: %s\n", motion[i].properties);
    }
}
```

2. Electromagnetic Field Calculator

Description:

Calculate the electromagnetic field intensity at various points in space.

Specifications:

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

Array: Holds field values at discrete points.

Union: Represents either electric or magnetic field components.

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

const Pointers: Prevent modification of field parameters.

Double Pointers: Manage memory for field grid allocation dynamically.

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

union FieldComponent {
    float electric[3];
    float magnetic[3];
};

typedef struct {
    const float position[3];
    union FieldComponent field;
    const char *coordinateSystem;
} Field;

void allocateFieldGrid(Field ***grid, int *rows, int *cols);
void deallocateFieldGrid(Field **grid, int rows);
void inputFieldValues(Field **grid, int rows, int cols);
void displayFieldGrid(Field **grid, int rows, int cols);

int main() {
    Field **fieldGrid = NULL;
    int rows, cols, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Allocate Field Grid\n");
        printf("2. Input Field Values\n");
        printf("3. Display Field Grid\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter number of rows: ");
                scanf("%d", &rows);
                printf("Enter number of columns: ");
                scanf("%d", &cols);
                allocateFieldGrid(&fieldGrid, &rows, &cols);
                break;
            case 2:
                inputFieldValues(fieldGrid, rows, cols);
                break;
            case 3:
                displayFieldGrid(fieldGrid, rows, cols);
                break;
            case 4:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 4);

    deallocateFieldGrid(fieldGrid, rows);
    return 0;
}

void allocateFieldGrid(Field ***grid, int *rows, int *cols) {
    *grid = (Field **)malloc(*rows * sizeof(Field *));
    for (int i = 0; i < *rows; i++) {
        (*grid)[i] = (Field *)malloc(*cols * sizeof(Field));
    }
    printf("Field grid allocated successfully.\n");
}

void deallocateFieldGrid(Field **grid, int rows) {
    for (int i = 0; i < rows; i++) {
        free(grid[i]);
    }
    free(grid);
    printf("Field grid deallocated successfully.\n");
}

void inputFieldValues(Field **grid, int rows, int cols) {
    char coordSystem[20];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("Enter position (x y z) for point (%d, %d): ", i + 1, j + 1);
            scanf("%f %f %f", (float *)&grid[i][j].position[0],
                              (float *)&grid[i][j].position[1],
                              (float *)&grid[i][j].position[2]);

            printf("Enter field type (electric/magnetic): ");
            scanf("%s", coordSystem);

            if (strcmp(coordSystem, "electric") == 0) {
                printf("Enter electric field components (Ex Ey Ez): ");
                scanf("%f %f %f", &grid[i][j].field.electric[0],
                                  &grid[i][j].field.electric[1],
                                  &grid[i][j].field.electric[2]);
                grid[i][j].coordinateSystem = "Electric Field";
            } else if (strcmp(coordSystem, "magnetic") == 0) {
                printf("Enter magnetic field components (Bx By Bz): ");
                scanf("%f %f %f", &grid[i][j].field.magnetic[0],
                                  &grid[i][j].field.magnetic[1],
                                  &grid[i][j].field.magnetic[2]);
                grid[i][j].coordinateSystem = "Magnetic Field";
            } else {
                printf("Invalid field type.\n");
            }
        }
    }
```

```
}

void displayFieldGrid(Field **grid, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("\nPoint (%d, %d):\n", i + 1, j + 1);
            printf("Position: (%.2f, %.2f, %.2f)\n", grid[i][j].position[0],
                                                     grid[i][j].position[1],
                                                     grid[i][j].position[2]);
            if (strcmp(grid[i][j].coordinateSystem, "Electric Field") == 0) {
                printf("Electric Field: (%.2f, %.2f, %.2f)\n",
                        grid[i][j].field.electric[0],
                        grid[i][j].field.electric[1],
                        grid[i][j].field.electric[2]);
            } else if (strcmp(grid[i][j].coordinateSystem, "Magnetic Field") == 0) {
                printf("Magnetic Field: (%.2f, %.2f, %.2f)\n",
                        grid[i][j].field.magnetic[0],
                        grid[i][j].field.magnetic[1],
                        grid[i][j].field.magnetic[2]);
            }
        }
    }
}
```

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 <string.h>
```

```
#include <stdlib.h>

union EnergyState {
    float ground_state;
    float excited_state;
};

typedef struct {
    char element_name[50];
    float *energy_levels;
    float *transition_probabilities;
    const char *atomic_data;
    union EnergyState state;
} Atom;

void add_atom(Atom **atoms, int *count);
void display_atoms(Atom *atoms, int count);

int main() {
    Atom *atoms = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n1. Add Atom\n2. Display Atoms\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_atom(&atoms, &count);
                break;
            case 2:
                display_atoms(atoms, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 3);
    free(atoms);
    return 0;
}

void add_atom(Atom **atoms, int *count) {
    *atoms = (Atom *)realloc(*atoms, (*count + 1) * sizeof(Atom));
    Atom *new_atom = &(*atoms)[*count];

    printf("Enter element name: ");
    scanf("%s", new_atom->element_name);

    new_atom->energy_levels = (float *)malloc(2 * sizeof(float));
    new_atom->transition_probabilities = (float *)malloc(1 * sizeof(float));

    printf("Enter ground state energy: ");
    scanf("%f", &new_atom->energy_levels[0]);

    printf("Enter excited state energy: ");
    scanf("%f", &new_atom->energy_levels[1]);

    printf("Enter transition probability: ");
    scanf("%f", &new_atom->transition_probabilities[0]);

    new_atom->state.ground_state = new_atom->energy_levels[0];
    new_atom->atomic_data = "Atomic Data";
    (*count)++;
}

void display_atoms(Atom *atoms, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nAtom %d:\n", i + 1);
        printf("Element: %s\n", atoms[i].element_name);
        printf("Ground State Energy: %.2f\n", atoms[i].energy_levels[0]);
        printf("Excited State Energy: %.2f\n", atoms[i].energy_levels[1]);
        printf("Transition Probability: %.2f\n", atoms[i].transition_probabilities[0]);
        printf("Atomic Data: %s\n", atoms[i].atomic_data);
    }
}
```

4. Quantum State Representation System

Description:

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

Specifications:

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

Array: Represents the wavefunction across multiple points.

Union: Stores amplitude or phase information.

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

const Pointers: Protect state properties.

Double Pointers: Manage quantum states dynamically.

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

union StateComponent {
    float amplitude;
    float phase;
};

typedef struct {
    float wavefunction[2];
    const char *state_properties;
    union StateComponent component;
    char label[50];
} QuantumState;

void add_state(QuantumState **states, int *count);
void display_states(QuantumState *states, int count);

int main() {
    QuantumState *states = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n1. Add Quantum State\n2. Display Quantum States\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_state(&states, &count);
                break;
            case 2:
                display_states(states, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 3);

    free(states);
    return 0;
}

void add_state(QuantumState **states, int *count) {
    *states = (QuantumState *)realloc(*states, (*count + 1) * sizeof(QuantumState));
    QuantumState *new_state = &(*states)[*count];

    printf("Enter wavefunction amplitude: ");
    scanf("%f", &new_state->wavefunction[0]);

    printf("Enter wavefunction phase: ");
    scanf("%f", &new_state->wavefunction[1]);

    printf("Enter state label (e.g., 'ground state'): ");
    scanf("%s", new_state->label);

    new_state->component.amplitude = new_state->wavefunction[0];
    new_state->state_properties = "State Properties";
    (*count)++;
}

void display_states(QuantumState *states, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nQuantum State %d:\n", i + 1);
        printf("Wavefunction Amplitude: %.2f\n", states[i].wavefunction[0]);
        printf("Wavefunction Phase: %.2f\n", states[i].wavefunction[1]);
        printf("Label: %s\n", states[i].label);
        printf("State Properties: %s\n", states[i].state_properties);
    }
}
```

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>

union OpticalElement {
    float refractive_index;
    float focal_length;
};

typedef struct {
    float ray_path[2];
    const char *optical_properties;
    union OpticalElement element;
    char element_type[50];
} Optics;

void add_optical_element(Optics **elements, int *count);
void display_optical_elements(Optics *elements, int count);

int main() {
    Optics *elements = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n1. Add Optical Element\n2. Display Optical Elements\n3.
Exit\nEnter choice: ");
```

```
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_optical_element(&elements, &count);
                break;
            case 2:
                display_optical_elements(elements, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 3);

    free(elements);
    return 0;
}

void add_optical_element(Optics **elements, int *count) {
    *elements = (Optics *)realloc(*elements, (*count + 1) * sizeof(Optics));
    Optics *new_element = &(*elements)[*count];

    printf("Enter ray path coordinates (x y): ");
    scanf("%f %f", &new_element->ray_path[0], &new_element->ray_path[1]);

    printf("Enter optical element type (lens/mirror): ");
    scanf("%s", new_element->element_type);

    if (strcmp(new_element->element_type, "lens") == 0) {
        printf("Enter focal length: ");
        scanf("%f", &new_element->element.focal_length);
    } else if (strcmp(new_element->element_type, "mirror") == 0) {
        printf("Enter refractive index: ");
        scanf("%f", &new_element->element.refractive_index);
    }

    new_element->optical_properties = "Optical Properties";
    (*count)++;
```

```
}

void display_optical_elements(Optics *elements, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nOptical Element %d:\n", i + 1);
        printf("Ray Path: (%.2f, %.2f)\n", elements[i].ray_path[0], elements[i].ray_path[1]);
        printf("Element Type: %s\n", elements[i].element_type);
        if (strcmp(elements[i].element_type, "lens") == 0) {
            printf("Focal Length: %.2f\n", elements[i].element.focal_length);
        } else if (strcmp(elements[i].element_type, "mirror") == 0) {
            printf("Refractive Index: %.2f\n", elements[i].element.refractive_index);
        }
        printf("Optical Properties: %s\n", elements[i].optical_properties);
    }
}
```

6. Thermodynamics State Calculator

Description:
Calculate thermodynamic states of a system based on input parameters like pressure, volume, and temperature.
Specifications:
Structure: Represents thermodynamic properties (P, V, T, and entropy).
Array: Stores states over a range of conditions.
Union: Handles dependent properties like energy or entropy.
Strings: Represent state descriptions.
const Pointers: Protect thermodynamic data.
Double Pointers: Allocate state data dynamically for simulation.

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

union OpticalElement {
    float refractive_index;
    float focal_length;
};

typedef struct {
    float ray_path[2];
    const char *optical_properties;
    union OpticalElement element;
    char element_type[50];
} Optics;

void add_optical_element(Optics **elements, int *count);
void display_optical_elements(Optics *elements, int count);

int main() {
    Optics *elements = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n1. Add Optical Element\n2. Display Optical Elements\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_optical_element(&elements, &count);
                break;
            case 2:
                display_optical_elements(elements, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (choice != 3);

    free(elements);
    return 0;
}

void add_optical_element(Optics **elements, int *count) {
    *elements = (Optics *)realloc(*elements, (*count + 1) * sizeof(Optics));
    Optics *new_element = &(*elements)[*count];

    printf("Enter ray path coordinates (x y): ");
    scanf("%f %f", &new_element->ray_path[0], &new_element->ray_path[1]);

    printf("Enter optical element type (lens/mirror): ");
    scanf("%s", new_element->element_type);

    if (strcmp(new_element->element_type, "lens") == 0) {
        printf("Enter focal length: ");
        scanf("%f", &new_element->element.focal_length);
    } else if (strcmp(new_element->element_type, "mirror") == 0) {
        printf("Enter refractive index: ");
        scanf("%f", &new_element->element.refractive_index);
    }

    new_element->optical_properties = "Optical Properties";
    (*count)++;
}

void display_optical_elements(Optics *elements, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nOptical Element %d:\n", i + 1);
        printf("Ray Path: (%.2f, %.2f)\n", elements[i].ray_path[0], elements[i].ray_path[1]);
        printf("Element Type: %s\n", elements[i].element_type);
        if (strcmp(elements[i].element_type, "lens") == 0) {
            printf("Focal Length: %.2f\n", elements[i].element.focal_length);
        } else if (strcmp(elements[i].element_type, "mirror") == 0) {
            printf("Refractive Index: %.2f\n", elements[i].element.refractive_index);
        }
        printf("Optical Properties: %s\n", elements[i].optical_properties);
    }
}
```

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>

union ReactionDetail {
    float energy_released;
    char product_details[50];
};

typedef struct {
    char reactant[50];
    char product[50];
    const char *reaction_description;
    union ReactionDetail detail;
    char reaction_type[50];
} NuclearReaction;

void add_reaction(NuclearReaction **reactions, int *count);
void display_reactions(NuclearReaction *reactions, int count);

int main() {
    NuclearReaction *reactions = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n1. Add Reaction\n2. Display Reactions\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_reaction(&reactions, &count);
                break;
            case 2:
                display_reactions(reactions, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
```

```
            printf("Invalid choice.\n");
        }
    } while (choice != 3);

    free(reactions);
    return 0;
}

void add_reaction(NuclearReaction **reactions, int *count) {
    *reactions = (NuclearReaction *)realloc(*reactions, (*count + 1) *
sizeof(NuclearReaction));
    NuclearReaction *new_reaction = &(*reactions)[*count];

    printf("Enter reactant name: ");
    scanf("%s", new_reaction->reactant);

    printf("Enter product name: ");
    scanf("%s", new_reaction->product);

    printf("Enter reaction type (fission/fusion): ");
    scanf("%s", new_reaction->reaction_type);

    if (strcmp(new_reaction->reaction_type, "fission") == 0) {
        printf("Enter energy released: ");
        scanf("%f", &new_reaction->detail.energy_released);
    } else if (strcmp(new_reaction->reaction_type, "fusion") == 0) {
        printf("Enter product details: ");
        scanf("%s", new_reaction->detail.product_details);
    }

    new_reaction->reaction_description = "Nuclear Reaction Data";
    (*count)++;
}

void display_reactions(NuclearReaction *reactions, int count) {
    for (int i = 0; i < count; i++) {
        printf("\nReaction %d:\n", i + 1);
        printf("Reactant: %s\n", reactions[i].reactant);
        printf("Product: %s\n", reactions[i].product);
        printf("Reaction Type: %s\n", reactions[i].reaction_type);
        if (strcmp(reactions[i].reaction_type, "fission") == 0) {
            printf("Energy Released: %.2f\n", reactions[i].detail.energy_released);
        } else if (strcmp(reactions[i].reaction_type, "fusion") == 0) {
            printf("Product Details: %s\n", reactions[i].detail.product_details);
        }
        printf("Description: %s\n", reactions[i].reaction_description);
    }
}
```

8. Gravitational Field Simulation

Description:
Simulate the gravitational field of massive objects in a system.
Specifications:
Structure: Contains object properties (mass, position, field strength).
Array: Stores field values at different points.
Union: Handles either mass or field strength as parameters.
Strings: Represent object labels (e.g., "Planet A," "Star B").
const Pointers: Protect object properties.
Double Pointers: Dynamically allocate memory for gravitational field data.

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

typedef union {
    float mass;
    float field_strength;
} Parameter;

typedef struct {
    char object_label[50];
    float position[2];
    const char *field_description;
    Parameter parameter;
} GravitationalField;

void add_field(GravitationalField **fields, int *count);
void display_fields(GravitationalField *fields, int count);

int main() {
    GravitationalField *fields = NULL;
    int count = 0, choice;
    do {
        printf("\nMenu:\n");
        printf("1. Add Field\n");
        printf("2. Display Fields\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_field(&fields, &count);
                break;
            case 2:
                display_fields(fields, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(fields);
    return 0;
}

void add_field(GravitationalField **fields, int *count) {
    *fields = (GravitationalField *)realloc(*fields, (*count + 1) *
sizeof(GravitationalField));
    GravitationalField *new_field = &(*fields)[*count];

    printf("Enter object label: ");
    scanf("%s", new_field->object_label);

    printf("Enter position (x y): ");
    scanf("%f %f", &new_field->position[0], &new_field->position[1]);

    char parameter_type[50];
    int valid = 0;
    while (!valid) {
        printf("Enter parameter type (mass/field_strength): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "mass") == 0) {
            printf("Enter mass: ");
            scanf("%f", &new_field->parameter.mass);
            valid = 1;
        } else if (strcmp(parameter_type, "field_strength") == 0) {
            printf("Enter field strength: ");
            scanf("%f", &new_field->parameter.field_strength);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'mass' or 'field_strength'.\n");
            while (getchar() != '\n');  // Clear input buffer
        }
    }

    new_field->field_description = "Gravitational Field Properties";
    (*count)++;
}

void display_fields(GravitationalField *fields, int count) {
    if (count == 0) {
        printf("\nNo fields to display.\n");
        return;
    }

    printf("\nField Information:\n");
    printf("-----------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nField %d:\n", i + 1);
        printf("Object Label: %s\n", fields[i].object_label);
        printf("Position: (%.2f, %.2f)\n", fields[i].position[0], fields[i].position[1]);
        if (fields[i].field_description != NULL) {
            printf("Field Description: %s\n", fields[i].field_description);
        }

        if (fields[i].parameter.mass > 0) {
            printf("Mass: %.2f\n", fields[i].parameter.mass);
        } else if (fields[i].parameter.field_strength > 0) {
            printf("Field Strength: %.2f\n", fields[i].parameter.field_strength);
        }
    }
}
```

9. Wave Interference Analyzer

Description:

Analyze interference patterns produced by waves from multiple sources.

Specifications:

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

Array: Stores wave interference data at discrete points.

Union: Handles either amplitude or phase information.

Strings: Represent wave source labels.

const Pointers: Protect wave properties.

Double Pointers: Manage dynamic allocation of wave sources.

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

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

typedef struct {
    char wave_label[50];
    float wavelength;
    const char *wave_description;
    WaveParameter parameter;
} Wave;

void add_wave(Wave **waves, int *count);
void display_waves(Wave *waves, int count);

int main() {
    Wave *waves = NULL;
    int count = 0, choice;
    do {
        printf("\nMenu:\n");
        printf("1. Add Wave\n");
        printf("2. Display Waves\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_wave(&waves, &count);
                break;
            case 2:
                display_waves(waves, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(waves);
    return 0;
}

void add_wave(Wave **waves, int *count) {
    *waves = (Wave *)realloc(*waves, (*count + 1) * sizeof(Wave));
    Wave *new_wave = &(*waves)[*count];

    printf("Enter wave label: ");
    scanf("%s", new_wave->wave_label);

    printf("Enter wavelength: ");
    scanf("%f", &new_wave->wavelength);

    char parameter_type[50];
    int valid = 0;
    while (!valid) {
        printf("Enter parameter type (amplitude/phase): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "amplitude") == 0) {
            printf("Enter amplitude: ");
            scanf("%f", &new_wave->parameter.amplitude);
            valid = 1;
        } else if (strcmp(parameter_type, "phase") == 0) {
            printf("Enter phase: ");
            scanf("%f", &new_wave->parameter.phase);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'amplitude' or 'phase'.\n");
            while (getchar() != '\n');
        }
    }

    new_wave->wave_description = "Wave Interference Properties";
    (*count)++;
}

void display_waves(Wave *waves, int count) {
    if (count == 0) {
        printf("\nNo waves to display.\n");
        return;
    }

    printf("\nWave Information:\n");
    printf("----------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nWave %d:\n", i + 1);
        printf("Wave Label: %s\n", waves[i].wave_label);
        printf("Wavelength: %.2f\n", waves[i].wavelength);
        printf("Wave Description: %s\n", waves[i].wave_description);

        if (waves[i].parameter.amplitude > 0) {
            printf("Amplitude: %.2f\n", waves[i].parameter.amplitude);
        } else if (waves[i].parameter.phase > 0) {
            printf("Phase: %.2f\n", waves[i].parameter.phase);
        }
    }
}
```

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>

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

typedef struct {
    char material_name[50];
    const char *material_description;
    MaterialProperty property;
} MagneticMaterial;

void add_material(MagneticMaterial **materials, int *count);
void display_materials(MagneticMaterial *materials, int count);

int main() {
    MagneticMaterial *materials = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Material\n");
        printf("2. Display Materials\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_material(&materials, &count);
                break;
            case 2:
                display_materials(materials, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(materials);
    return 0;
}

void add_material(MagneticMaterial **materials, int *count) {
    *materials = (MagneticMaterial *)realloc(*materials, (*count + 1) *
sizeof(MagneticMaterial));
    MagneticMaterial *new_material = &(*materials)[*count];

    printf("Enter material name: ");
    scanf("%s", new_material->material_name);

    char property_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter property type (permeability/saturation): ");
        scanf("%s", property_type);

        if (strcmp(property_type, "permeability") == 0) {
            printf("Enter permeability: ");
            scanf("%f", &new_material->property.permeability);
            valid = 1;
        } else if (strcmp(property_type, "saturation") == 0) {
            printf("Enter saturation: ");
            scanf("%f", &new_material->property.saturation);
            valid = 1;
        } else {
            printf("Invalid property type. Please enter 'permeability' or 'saturation'.\n");
            while (getchar() != '\n');
        }
    }

    new_material->material_description = "Magnetic Material Properties";
    (*count)++;
}

void display_materials(MagneticMaterial *materials, int count) {
    if (count == 0) {
        printf("\nNo materials to display.\n");
        return;
    }

    printf("\nMaterial Information:\n");
    printf("-------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nMaterial %d:\n", i + 1);
        printf("Material Name: %s\n", materials[i].material_name);
        printf("Material Description: %s\n", materials[i].material_description);

        if (materials[i].property.permeability > 0) {
            printf("Permeability: %.2f\n", materials[i].property.permeability);
        } else if (materials[i].property.saturation > 0) {
            printf("Saturation: %.2f\n", materials[i].property.saturation);
        }
    }
}
```

11. Plasma Dynamics Simulator
Description:
Simulate the behavior of plasma under various conditions.

Specifications:
Structure: Represents plasma parameters (density, temperature, and electric field).
Array: Stores simulation results.
Union: Handles either density or temperature data.
Strings: Represent plasma types.
const Pointers: Protect plasma parameters.
Double Pointers: Manage dynamic allocation for simulation data.

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

typedef union {
    float density;
    float temperature;
} PlasmaParameter;

typedef struct {
    char plasma_type[50];
    const char *description;
    PlasmaParameter parameter;
} Plasma;

void add_plasma(Plasma **plasmas, int *count);
void display_plasmas(Plasma *plasmas, int count);

int main() {
    Plasma *plasmas = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Plasma Data\n");
        printf("2. Display Plasma Data\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_plasma(&plasmas, &count);
                break;
            case 2:
                display_plasmas(plasmas, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(plasmas);
    return 0;
}

void add_plasma(Plasma **plasmas, int *count) {
    *plasmas = (Plasma *)realloc(*plasmas, (*count + 1) * sizeof(Plasma));
    Plasma *new_plasma = &(*plasmas)[*count];

    printf("Enter plasma type: ");
    scanf("%s", new_plasma->plasma_type);

    char parameter_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter parameter type (density/temperature): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "density") == 0) {
            printf("Enter density: ");
            scanf("%f", &new_plasma->parameter.density);
            valid = 1;
        } else if (strcmp(parameter_type, "temperature") == 0) {
            printf("Enter temperature: ");
            scanf("%f", &new_plasma->parameter.temperature);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'density' or 'temperature'.\n");
            while (getchar() != '\n');
        }
    }

    new_plasma->description = "Plasma Dynamics Parameters";
    (*count)++;
}

void display_plasmas(Plasma *plasmas, int count) {
    if (count == 0) {
        printf("\nNo plasma data to display.\n");
        return;
    }

    printf("\nPlasma Data:\n");
    printf("----------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nPlasma %d:\n", i + 1);
        printf("Plasma Type: %s\n", plasmas[i].plasma_type);
        printf("Description: %s\n", plasmas[i].description);

        if (plasmas[i].parameter.density > 0) {
            printf("Density: %.2f\n", plasmas[i].parameter.density);
        } else if (plasmas[i].parameter.temperature > 0) {
            printf("Temperature: %.2f\n", plasmas[i].parameter.temperature);
        }
    }
}
```

12. Kinematics Equation Solver

Description:

Solve complex kinematics problems for objects in motion.

Specifications:

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

Array: Stores time-dependent motion data.

Union: Handles either velocity or displacement equations.

Strings: Represent motion descriptions.

const Pointers: Protect object properties.

Double Pointers: Dynamically allocate memory for motion data.

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

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

typedef union {
    float velocity;
    float displacement;
} MotionParameter;

typedef struct {
    char motion_description[100];
    const char *equation_type;
    MotionParameter parameter;
} Kinematics;

void add_motion(Kinematics **motions, int *count);
void display_motions(Kinematics *motions, int count);

int main() {
    Kinematics *motions = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Motion Data\n");
        printf("2. Display Motion Data\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_motion(&motions, &count);
                break;
            case 2:
                display_motions(motions, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(motions);
    return 0;
}

void add_motion(Kinematics **motions, int *count) {
    *motions = (Kinematics *)realloc(*motions, (*count + 1) * sizeof(Kinematics));
    Kinematics *new_motion = &(*motions)[*count];

    printf("Enter motion description: ");
    scanf(" %[^\n]", new_motion->motion_description);

    char parameter_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter parameter type (velocity/displacement): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "velocity") == 0) {
            printf("Enter velocity: ");
            scanf("%f", &new_motion->parameter.velocity);
            valid = 1;
        } else if (strcmp(parameter_type, "displacement") == 0) {
            printf("Enter displacement: ");
            scanf("%f", &new_motion->parameter.displacement);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'velocity' or 'displacement'.\n");
            while (getchar() != '\n');
        }
    }

    new_motion->equation_type = "Kinematics Equation Solver";
    (*count)++;
}

void display_motions(Kinematics *motions, int count) {
    if (count == 0) {
        printf("\nNo motion data to display.\n");
        return;
    }

    printf("\nMotion Data:\n");
    printf("----------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nMotion %d:\n", i + 1);
        printf("Motion Description: %s\n", motions[i].motion_description);
        printf("Equation Type: %s\n", motions[i].equation_type);

        if (motions[i].parameter.velocity > 0) {
            printf("Velocity: %.2f\n", motions[i].parameter.velocity);
        } else if (motions[i].parameter.displacement > 0) {
            printf("Displacement: %.2f\n", motions[i].parameter.displacement);
        }
    }
}
```

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

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

typedef union {
    float intensity;
    float wavelength;
} SpectralLineParameter;
```

```
typedef struct {
    char element[50];
    const char *line_description;
    SpectralLineParameter parameter;
} SpectralLine;

void add_spectral_line(SpectralLine **lines, int *count);
void display_spectral_lines(SpectralLine *lines, int count);

int main() {
    SpectralLine *lines = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Spectral Line Data\n");
        printf("2. Display Spectral Line Data\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_spectral_line(&lines, &count);
                break;
            case 2:
                display_spectral_lines(lines, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(lines);
    return 0;
}
```

```
void add_spectral_line(SpectralLine **lines, int *count) {
    *lines = (SpectralLine *)realloc(*lines, (*count + 1) * sizeof(SpectralLine));
    SpectralLine *new_line = &(*lines)[*count];

    printf("Enter element name: ");
    scanf(" %[^\n]", new_line->element);

    char parameter_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter parameter type (intensity/wavelength): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "intensity") == 0) {
            printf("Enter intensity: ");
            scanf("%f", &new_line->parameter.intensity);
            valid = 1;
        } else if (strcmp(parameter_type, "wavelength") == 0) {
            printf("Enter wavelength: ");
            scanf("%f", &new_line->parameter.wavelength);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'intensity' or 'wavelength'.\n");
            while (getchar() != '\n');
        }
    }

    new_line->line_description = "Spectral Line Data";
    (*count)++;
}

void display_spectral_lines(SpectralLine *lines, int count) {
    if (count == 0) {
        printf("\nNo spectral line data to display.\n");
        return;
    }

    printf("\nSpectral Line Data:\n");
    printf("----------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nSpectral Line %d:\n", i + 1);
        printf("Element: %s\n", lines[i].element);
        printf("Line Description: %s\n", lines[i].line_description);

        if (lines[i].parameter.intensity > 0) {
            printf("Intensity: %.2f\n", lines[i].parameter.intensity);
        } else if (lines[i].parameter.wavelength > 0) {
            printf("Wavelength: %.2f\n", lines[i].parameter.wavelength);
        }
    }
}
```

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

typedef union {
    float velocity;
    float displacement;
} ProjectileMotionParameter;

typedef struct {
    char trajectory_description[100];
    const char *motion_type;
    ProjectileMotionParameter parameter;
} ProjectileMotion;

void add_projectile_motion(ProjectileMotion **motions, int *count);
void display_projectile_motions(ProjectileMotion *motions, int count);

int main() {
    ProjectileMotion *motions = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Projectile Motion Data\n");
        printf("2. Display Projectile Motion Data\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_projectile_motion(&motions, &count);
                break;
            case 2:
                display_projectile_motions(motions, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(motions);
    return 0;
}

void add_projectile_motion(ProjectileMotion **motions, int *count) {
    *motions = (ProjectileMotion *)realloc(*motions, (*count + 1) *
sizeof(ProjectileMotion));
    ProjectileMotion *new_motion = &(*motions)[*count];

    printf("Enter trajectory description: ");
    scanf(" %[^\n]", new_motion->trajectory_description);
    char parameter_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter parameter type (velocity/displacement): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "velocity") == 0) {
            printf("Enter velocity: ");
            scanf("%f", &new_motion->parameter.velocity);
            valid = 1;
        } else if (strcmp(parameter_type, "displacement") == 0) {
            printf("Enter displacement: ");
            scanf("%f", &new_motion->parameter.displacement);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'velocity' or 'displacement'.\n");
            while (getchar() != '\n');
        }
    }

    new_motion->motion_type = "Projectile Motion Simulator";
    (*count)++;
}

void display_projectile_motions(ProjectileMotion *motions, int count) {
    if (count == 0) {
        printf("\nNo projectile motion data to display.\n");
        return;
    }

    printf("\nProjectile Motion Data:\n");
    printf("------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nProjectile Motion %d:\n", i + 1);
        printf("Trajectory Description: %s\n", motions[i].trajectory_description);
        printf("Motion Type: %s\n", motions[i].motion_type);

        if (motions[i].parameter.velocity > 0) {
            printf("Velocity: %.2f\n", motions[i].parameter.velocity);
        } else if (motions[i].parameter.displacement > 0) {
            printf("Displacement: %.2f\n", motions[i].parameter.displacement);
        }
    }
}
```

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>

typedef union {
    float yield_stress;
    float elastic_modulus;
} StressStrainParameter;

typedef struct {
    char material_name[100];
    const char *material_type;
    StressStrainParameter parameter;
} StressStrainData;

void add_stress_strain_data(StressStrainData **data, int *count);
void display_stress_strain_data(StressStrainData *data, int count);

int main() {
    StressStrainData *data = NULL;
    int count = 0, choice;

    do {
        printf("\nMenu:\n");
        printf("1. Add Stress-Strain Data\n");
        printf("2. Display Stress-Strain Data\n");
        printf("3. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_stress_strain_data(&data, &count);
                break;
            case 2:
                display_stress_strain_data(data, count);
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    free(data);
    return 0;
}

void add_stress_strain_data(StressStrainData **data, int *count) {
    *data = (StressStrainData *)realloc(*data, (*count + 1) * sizeof(StressStrainData));
    StressStrainData *new_data = &(*data)[*count];

    printf("Enter material name: ");
    scanf(" %[^\n]", new_data->material_name);

    char parameter_type[50];
    int valid = 0;

    while (!valid) {
        printf("Enter parameter type (yield_stress/elastic_modulus): ");
        scanf("%s", parameter_type);

        if (strcmp(parameter_type, "yield_stress") == 0) {
            printf("Enter yield stress: ");
            scanf("%f", &new_data->parameter.yield_stress);
            valid = 1;
        } else if (strcmp(parameter_type, "elastic_modulus") == 0) {
            printf("Enter elastic modulus: ");
            scanf("%f", &new_data->parameter.elastic_modulus);
            valid = 1;
        } else {
            printf("Invalid parameter type. Please enter 'yield_stress' or 'elastic_modulus'.\n");
            while (getchar() != '\n');
        }
    }

    new_data->material_type = "Material Stress-Strain Analyzer";
    (*count)++;
}

void display_stress_strain_data(StressStrainData *data, int count) {
    if (count == 0) {
        printf("\nNo stress-strain data to display.\n");
        return;
    }

    printf("\nStress-Strain Data:\n");
    printf("--------------------------------------\n");
    for (int i = 0; i < count; i++) {
        printf("\nMaterial %d:\n", i + 1);
        printf("Material Name: %s\n", data[i].material_name);
        printf("Material Type: %s\n", data[i].material_type);

        if (data[i].parameter.yield_stress > 0) {
            printf("Yield Stress: %.2f\n", data[i].parameter.yield_stress);
        } else if (data[i].parameter.elastic_modulus > 0) {
            printf("Elastic Modulus: %.2f\n", data[i].parameter.elastic_modulus);
        }
    }
}
```