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

1. Alloy Composition Analysis System

Description:

Design a system to analyze alloy compositions using structures

for composition details, arrays for storing multiple samples,

and unions to represent percentage compositions of different

metals.

Specifications:

Structure: Stores sample ID, name, and composition details.

Union: Represents variable percentage compositions of metals.

Array: Stores multiple alloy samples.

const Pointers: Protect composition details.

Double Pointers: Manage dynamic allocation of alloy samples.

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

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

typedef struct {
    const char *sampleID;
```

```
    char name[50];
    union {
        float iron;
        float copper;
        float zinc;
        float aluminum;
    } composition;
    int metalType;
} CompositionDetails;

void addSample(CompositionDetails **samples, int *size, int *capacity, const char *sampleID, const char *name, float percentage, int metalType);
void displaySamples(CompositionDetails **samples, int size);

int main() {
    int size = 0, capacity = 2;
    CompositionDetails *samples = (CompositionDetails *)malloc(capacity * sizeof(CompositionDetails));

    addSample(&samples, &size, &capacity, "S001", "Sample A", 55.0, 0); // Iron
    addSample(&samples, &size, &capacity, "S002", "Sample B", 45.0, 1); // Copper
    addSample(&samples, &size, &capacity, "S003", "Sample C", 35.0, 2); // Zinc
    addSample(&samples, &size, &capacity, "S004", "Sample D", 25.0, 3); // Aluminum

    displaySamples(&samples, size);
```

```
    for (int i = 0; i < size; i++) {
        free((void *)samples[i].sampleID); // Cast to void* to free const char*
    }
    free(samples);

    return 0;
}

// Function to add a sample to the list
void addSample(CompositionDetails **samples, int *size, int *capacity, const char *sampleID, const char *name, float percentage, int metalType) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        CompositionDetails *newSamples = (CompositionDetails *)malloc(*capacity * sizeof(CompositionDetails));

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

        // Free old array and update pointer
        free(*samples);
        *samples = newSamples;
    }

    // Initialize the new sample
    (*samples)[*size].sampleID = strdup(sampleID);
```

```
    strcpy((*samples)[*size].name, name);

    // Set the percentage composition based on the metal type
    switch (metalType) {
        case 0:
            (*samples)[*size].composition.iron = percentage;
            break;
        case 1:
            (*samples)[*size].composition.copper = percentage;
            break;
        case 2:
            (*samples)[*size].composition.zinc = percentage;
            break;
        case 3:
            (*samples)[*size].composition.aluminum = percentage;
            break;
    }

    (*samples)[*size].metalType = metalType;

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

// Function to display the samples
void displaySamples(CompositionDetails **samples, int size) {
    printf("Alloy Composition Samples:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Sample ID: %s\n", (*samples)[i].sampleID);
        printf("Name: %s\n", (*samples)[i].name);
        printf("Composition: ");
        switch ((*samples)[i].metalType) {
            case 0:
                printf("Iron: %.2f%%\n", (*samples)[i].composition.iron);
                break;
            case 1:
                printf("Copper: %.2f%%\n", (*samples)[i].composition.copper);
                break;
            case 2:
                printf("Zinc: %.2f%%\n", (*samples)[i].composition.zinc);
                break;
            case 3:
                printf("Aluminum: %.2f%%\n", (*samples)[i].composition.aluminum);
                break;
        }
        printf("\n");
    }
}
```

```
/*******************************
2. Heat Treatment Process Manager
Description:
Develop a program to manage heat treatment processes for
metals using structures for process details, arrays for
treatment parameters, and strings for process names.
Specifications:
Structure: Holds process ID, temperature, duration, and
cooling rate.
Array: Stores treatment parameter sets.
Strings: Process names.
const Pointers: Protect process data.
Double Pointers: Allocate and manage dynamic process data.

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

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



typedef struct {
    const char *processID;
    char processName[50];
    float temperature;
    int duration;
    float coolingRate;
} ProcessDetails;
```

```
void addProcess(ProcessDetails **processes, int *size, int *capacity, const char *processID,
const char *processName, float temperature, int duration, float coolingRate);
void displayProcesses(ProcessDetails **processes, int size);

int main() {
    int size = 0, capacity = 2;
    ProcessDetails *processes = (ProcessDetails *)malloc(capacity * sizeof(ProcessDetails));

    addProcess(&processes, &size, &capacity, "P001", "Annealing", 900.0, 120, 5.0);
    addProcess(&processes, &size, &capacity, "P002", "Quenching", 850.0, 30, 15.0);
    addProcess(&processes, &size, &capacity, "P003", "Tempering", 600.0, 60, 10.0);

    displayProcesses(&processes, size);

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

    return 0;
}

void addProcess(ProcessDetails **processes, int *size, int *capacity, const char *processID,
const char *processName, float temperature, int duration, float coolingRate) {
```

```
    if (*size == *capacity) {
        *capacity *= 2;
        ProcessDetails *newProcesses = (ProcessDetails *)malloc(*capacity *
sizeof(ProcessDetails));


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


        free(*processes);
        *processes = newProcesses;
    }


    (*processes)[*size].processID = strdup(processID);
    strcpy((*processes)[*size].processName, processName);
    (*processes)[*size].temperature = temperature;
    (*processes)[*size].duration = duration;
    (*processes)[*size].coolingRate = coolingRate;


    (*size)++;
}
```

```
void displayProcesses(ProcessDetails **processes, int size) {
    printf("Heat Treatment Processes:\n");
    for (int i = 0; i < size; i++) {
        printf("Process ID: %s\n", (*processes)[i].processID);
        printf("Process Name: %s\n", (*processes)[i].processName);
        printf("Temperature: %.2f°C\n", (*processes)[i].temperature);
        printf("Duration: %d minutes\n", (*processes)[i].duration);
        printf("Cooling Rate: %.2f°C per minute\n", (*processes)[i].coolingRate);
        printf("\n");
    }
}
```

Union: Represents tensile strength, hardness, or elongation.

Array: Test data for multiple samples.

const Pointers: Protect test IDs.

Double Pointers: Manage dynamic test records.

**********/

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

typedef struct {
    const char *testID;
    char testType[50];
    union {
        float tensileStrength;
        float hardness;
        float elongation;
    } result;
    int resultType;
} TestResult;
```

void addTestResult(TestResult **results, int *size, int *capacity, const char *testID, const char *testType, float resultValue, int resultType);

void displayTestResults(TestResult **results, int size);

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

    addTestResult(&results, &size, &capacity, "T001", "Tensile Test", 550.0, 0); // Tensile strength
    addTestResult(&results, &size, &capacity, "T002", "Hardness Test", 200.0, 1); // Hardness
    addTestResult(&results, &size, &capacity, "T003", "Elongation Test", 25.0, 2); // Elongation

    displayTestResults(&results, size);

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

    return 0;
}

void addTestResult(TestResult **results, int *size, int *capacity, const char *testID, const char *testType, float resultValue, int resultType) {

    if (*size == *capacity) {
        *capacity *= 2;
        TestResult *newResults = (TestResult *)malloc(*capacity * sizeof(TestResult));
```

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


    free(*results);
    *results = newResults;
}


(*results)[*size].testID = strdup(testID);
strcpy((*results)[*size].testType, testType);


switch (resultType) {
    case 0:
        (*results)[*size].result.tensileStrength = resultValue;
        break;
    case 1:
        (*results)[*size].result.hardness = resultValue;
        break;
    case 2:
        (*results)[*size].result.elongation = resultValue;
        break;
}

(*results)[*size].resultType = resultType;
```

```
    (*size)++;
}



void displayTestResults(TestResult **results, int size) {
    printf("Steel Quality Test Results:\n");
    for (int i = 0; i < size; i++) {
        printf("Test ID: %s\n", (*results)[i].testID);
        printf("Test Type: %s\n", (*results)[i].testType);
        printf("Result: ");
        switch ((*results)[i].resultType) {
            case 0:
                printf("Tensile Strength: %.2f MPa\n", (*results)[i].result.tensileStrength);
                break;
            case 1:
                printf("Hardness: %.2f HB\n", (*results)[i].result.hardness);
                break;
            case 2:
                printf("Elongation: %.2f%%\n", (*results)[i].result.elongation);
                break;
        }
        printf("\n");
    }
}
```

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

4. Metal Fatigue Analysis

Description:

Develop a program to analyze metal fatigue using arrays for stress cycle data, structures for material details, and strings for material names.

Specifications:

Structure: Contains material ID, name, and endurance limit.

Array: Stress cycle data.

Strings: Material names.

const Pointers: Protect material details.

Double Pointers: Allocate dynamic material test data.

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

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

typedef struct {
    const char *materialID;
    char name[50];
    float enduranceLimit;
} MaterialDetails;

typedef struct {
```

```
    MaterialDetails material;
    int stressCycles[100];
    int numCycles;
} StressCycleData;


void addMaterial(StressCycleData **data, int *size, int *capacity, const char *materialID,
const char *name, float enduranceLimit, int *cycles, int numCycles);
void displayMaterials(StressCycleData **data, int size);

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


    int cycles1[] = {1000, 2000, 3000, 4000};
    int cycles2[] = {1500, 2500, 3500, 4500, 5500};
    int cycles3[] = {2000, 3000, 4000, 5000, 6000};

    // Add materials and their stress cycle data
    addMaterial(&data, &size, &capacity, "M001", "Steel A", 400.0, cycles1, 4);
    addMaterial(&data, &size, &capacity, "M002", "Aluminum B", 250.0, cycles2, 5);
    addMaterial(&data, &size, &capacity, "M003", "Titanium C", 500.0, cycles3, 5);

    displayMaterials(&data, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
```

```
    free((void *)data[i].material.materialID); // Cast to void* to free const char*
  }
  free(data);

  return 0;
}

void addMaterial(StressCycleData **data, int *size, int *capacity, const char *materialID,
const char *name, float enduranceLimit, int *cycles, int numCycles) {

  if (*size == *capacity) {
    *capacity *= 2;
    StressCycleData *newData = (StressCycleData *)malloc(*capacity *
sizeof(StressCycleData));

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

    // Free old array and update pointer
    free(*data);
    *data = newData;
  }

  // Initialize the new material
  (*data)[*size].material.materialID = strdup(materialID);
  strcpy((*data)[*size].material.name, name);
```

```
    (*data)[*size].material.enduranceLimit = enduranceLimit;
    (*data)[*size].numCycles = numCycles;

    // Copy stress cycle data
    for (int i = 0; i < numCycles; i++) {
        (*data)[*size].stressCycles[i] = cycles[i];
    }



    (*size)++;
}

// Function to display the materials and their stress cycle data
void displayMaterials(StressCycleData **data, int size) {
    printf("Metal Fatigue Analysis Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Material ID: %s\n", (*data)[i].material.materialID);
        printf("Material Name: %s\n", (*data)[i].material.name);
        printf("Endurance Limit: %.2f MPa\n", (*data)[i].material.enduranceLimit);
        printf("Stress Cycles: ");
        for (int j = 0; j < (*data)[i].numCycles; j++) {
            printf("%d ", (*data)[i].stressCycles[j]);
        }
        printf("\n\n");
    }
}
```

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

5. Foundry Management System

Description:

Create a system for managing foundry operations using arrays for equipment data, structures for casting details, and unions for variable mold properties.

Specifications:

Structure: Stores casting ID, weight, and material.

Union: Represents mold properties (dimensions or thermal conductivity).

Array: Equipment data.

const Pointers: Protect equipment details.

Double Pointers: Dynamic allocation of casting records

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

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

typedef struct {
    const char *castingID;
    float weight;
    char material[50];
    union {
        struct {
            float length;
            float width;
```

```
            float height;
        } dimensions;
        float thermalConductivity;
    } moldProperties;
    int isDimensions;
} CastingDetails;

// Define the structure for equipment data
typedef struct {
    const char *equipmentID;
    char name[50];
    char type[50];
} EquipmentData;

void addCastingRecord(CastingDetails **castings, int *size, int *capacity, const char *castingID, float weight, const char *material, void *moldProperties, int isDimensions);
void displayCastingRecords(CastingDetails **castings, int size);
void addEquipmentRecord(EquipmentData **equipment, int *size, int *capacity, const char *equipmentID, const char *name, const char *type);
void displayEquipmentRecords(EquipmentData **equipment, int size);

int main() {
    int castingSize = 0, castingCapacity = 2;
    CastingDetails *castings = (CastingDetails *)malloc(castingCapacity * sizeof(CastingDetails));

    int equipmentSize = 0, equipmentCapacity = 2;
    EquipmentData *equipment = (EquipmentData *)malloc(equipmentCapacity * sizeof(EquipmentData));
```

```
    // Define some mold properties
    float thermalConductivity = 120.5;
    struct {
        float length;
        float width;
        float height;
    } dimensions = {10.0, 5.0, 2.0};

    // Add some casting records
    addCastingRecord(&castings, &castingSize, &castingCapacity, "C001", 150.0, "Steel", &thermalConductivity, 0);
    addCastingRecord(&castings, &castingSize, &castingCapacity, "C002", 200.0, "Aluminum", &dimensions, 1);

    // Add some equipment records
    addEquipmentRecord(&equipment, &equipmentSize, &equipmentCapacity, "E001", "Furnace", "Heating");
    addEquipmentRecord(&equipment, &equipmentSize, &equipmentCapacity, "E002", "Crane", "Lifting");

    // Display records
    displayCastingRecords(&castings, castingSize);
    displayEquipmentRecords(&equipment, equipmentSize);

    // Free allocated memory
    for (int i = 0; i < castingSize; i++) {
        free((void *)castings[i].castingID); // Cast to void* to free const char*
    }
```

```
    free(castings);

    for (int i = 0; i < equipmentSize; i++) {
        free((void *)equipment[i].equipmentID); // Cast to void* to free const char*
    }
    free(equipment);

    return 0;
}

// Function to add a casting record to the list
void addCastingRecord(CastingDetails **castings, int *size, int *capacity, const char
*castingID, float weight, const char *material, void *moldProperties, int isDimensions) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        CastingDetails *newCastings = (CastingDetails *)malloc(*capacity *
sizeof(CastingDetails));

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

        // Free old array and update pointer
        free(*castings);
        *castings = newCastings;
    }
```

```
    // Initialize the new casting record
    (*castings)[*size].castingID = strdup(castingID);
    (*castings)[*size].weight = weight;
    strcpy((*castings)[*size].material, material);

    if (isDimensions) {
        (*castings)[*size].moldProperties.dimensions =
*(typeof((*castings)[*size].moldProperties.dimensions) *)moldProperties;
    } else {
        (*castings)[*size].moldProperties.thermalConductivity = *(float *)moldProperties;
    }
    (*castings)[*size].isDimensions = isDimensions;

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

// Function to display casting records
void displayCastingRecords(CastingDetails **castings, int size) {
    printf("Casting Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Casting ID: %s\n", (*castings)[i].castingID);
        printf("Weight: %.2f kg\n", (*castings)[i].weight);
        printf("Material: %s\n", (*castings)[i].material);
        if ((*castings)[i].isDimensions) {
            printf("Dimensions: %.2f x %.2f x %.2f cm\n",
(*castings)[i].moldProperties.dimensions.length,
(*castings)[i].moldProperties.dimensions.width,
(*castings)[i].moldProperties.dimensions.height);
```

```
        } else {
            printf("Thermal Conductivity: %.2f W/mK\n",
(*castings)[i].moldProperties.thermalConductivity);
        }
        printf("\n");
    }
}

// Function to add an equipment record to the list
void addEquipmentRecord(EquipmentData **equipment, int *size, int *capacity, const char
*equipmentID, const char *name, const char *type) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        EquipmentData *newEquipment = (EquipmentData *)malloc(*capacity *
sizeof(EquipmentData));

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

        // Free old array and update pointer
        free(*equipment);
        *equipment = newEquipment;
    }

    // Initialize the new equipment record
    (*equipment)[*size].equipmentID = strdup(equipmentID);
```

```
    strcpy((*equipment)[*size].name, name);
    strcpy((*equipment)[*size].type, type);

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

// Function to display equipment records
void displayEquipmentRecords(EquipmentData **equipment, int size) {
    printf("Equipment Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Equipment ID: %s\n", (*equipment)[i].equipmentID);
        printf("Name: %s\n", (*equipment)[i].name);
        printf("Type: %s\n", (*equipment)[i].type);
        printf("\n");
    }
}
```

```
/*******************************
6. Metal Purity Analysis
Description:
Develop a system for metal purity analysis using structures for sample data, arrays for impurity percentages, and unions for variable impurity types.
Specifications:
Structure: Contains sample ID, type, and purity.
Union: Represents impurity type (trace elements or oxides).
Array: Impurity percentages.
const Pointers: Protect purity data.
Double Pointers: Manage dynamic impurity records.

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

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


typedef struct {
    const char *sampleID;
    char type[50];
    float purity;
} SampleData;


typedef union {
    float traceElements[5];
```

```
    float oxides[5];
} ImpurityType;


typedef struct {
    SampleData sample;
    ImpurityType impurities;
    int isTraceElements; // 1 if trace elements, 0 if oxides
    int numImpurities; // Number of impurities
} ImpurityRecord;


void addImpurityRecord(ImpurityRecord **records, int *size, int *capacity, const char *sampleID, const char *type, float purity, void *impurities, int isTraceElements, int numImpurities);
void displayImpurityRecords(ImpurityRecord **records, int size);

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

    // Define some impurities
    float traceElements1[] = {0.02, 0.01, 0.03, 0.04, 0.05};
    float oxides1[] = {0.1, 0.2, 0.3, 0.4, 0.5};

    // Add some impurity records
    addImpurityRecord(&records, &size, &capacity, "S001", "Steel", 99.5, traceElements1, 1, 5);
    addImpurityRecord(&records, &size, &capacity, "S002", "Aluminum", 98.7, oxides1, 0, 5);
```

```
    displayImpurityRecords(&records, size);

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

    return 0;
}

// Function to add an impurity record to the list
void addImpurityRecord(ImpurityRecord **records, int *size, int *capacity, const char *sampleID, const char *type, float purity, void *impurities, int isTraceElements, int numImpurities) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        ImpurityRecord *newRecords = (ImpurityRecord *)malloc(*capacity * sizeof(ImpurityRecord));

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

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

```
        *records = newRecords;
    }

    // Initialize the new impurity record
    (*records)[*size].sample.sampleID = strdup(sampleID);
    strcpy((*records)[*size].sample.type, type);
    (*records)[*size].sample.purity = purity;

    if (isTraceElements) {
        for (int i = 0; i < numImpurities; i++) {
            (*records)[*size].impurities.traceElements[i] = ((float *)impurities)[i];
        }
    } else {
        for (int i = 0; i < numImpurities; i++) {
            (*records)[*size].impurities.oxides[i] = ((float *)impurities)[i];
        }
    }

    (*records)[*size].isTraceElements = isTraceElements;
    (*records)[*size].numImpurities = numImpurities;

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

// Function to display the impurity records
void displayImpurityRecords(ImpurityRecord **records, int size) {
    printf("Metal Purity Analysis Records:\n");
```

```
    for (int i = 0; i < size; i++) {
        printf("Sample ID: %s\n", (*records)[i].sample.sampleID);
        printf("Type: %s\n", (*records)[i].sample.type);
        printf("Purity: %.2f%%\n", (*records)[i].sample.purity);
        printf("Impurities: ");
        if ((*records)[i].isTraceElements) {
            for (int j = 0; j < (*records)[i].numImpurities; j++) {
                printf("%.2f%% (Trace Element) ", (*records)[i].impurities.traceElements[j]);
            }
        } else {
            for (int j = 0; j < (*records)[i].numImpurities; j++) {
                printf("%.2f%% (Oxide) ", (*records)[i].impurities.oxides[j]);
            }
        }
        printf("\n\n");
    }
}
```

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

7. Corrosion Testing System

Description:

Create a program to track corrosion tests using structures for test details, arrays for test results, and strings for test conditions.

Specifications:

Structure: Holds test ID, duration, and environment.

Array: Test results.

Strings: Test conditions.

const Pointers: Protect test configurations.

Double Pointers: Dynamic allocation of test records.

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

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

typedef struct {
    const char *testID;
    int duration; // in hours
    char environment[100];
} TestDetails;

typedef struct {
    TestDetails details;
```

```
    float results[100];
    int numResults;
} TestRecord;

void addTestRecord(TestRecord **records, int *size, int *capacity, const char *testID, int duration, const char *environment, float *results, int numResults);
void displayTestRecords(TestRecord **records, int size);

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

    // Define some test results
    float results1[] = {0.1, 0.2, 0.15, 0.18};
    float results2[] = {0.05, 0.07, 0.1, 0.12, 0.11};
    float results3[] = {0.2, 0.25, 0.22, 0.3};

    // Add test records
    addTestRecord(&records, &size, &capacity, "T001", 24, "Salt Water", results1, 4);
    addTestRecord(&records, &size, &capacity, "T002", 48, "Acidic Solution", results2, 5);
    addTestRecord(&records, &size, &capacity, "T003", 72, "High Humidity", results3, 4);

    displayTestRecords(&records, size);

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

```
    }
    free(records);

    return 0;
}

// Function to add a test record to the list
void addTestRecord(TestRecord **records, int *size, int *capacity, const char *testID, int duration, const char *environment, float *results, int numResults) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        TestRecord *newRecords = (TestRecord *)malloc(*capacity * sizeof(TestRecord));

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

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

    // Initialize the new test record
    (*records)[*size].details.testID = strdup(testID);
    (*records)[*size].details.duration = duration;
    strcpy((*records)[*size].details.environment, environment);
```

```
    // Copy test results
    for (int i = 0; i < numResults; i++) {
        (*records)[*size].results[i] = results[i];
    }
    (*records)[*size].numResults = numResults;

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

// Function to display the test records
void displayTestRecords(TestRecord **records, int size) {
    printf("Corrosion Test Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Test ID: %s\n", (*records)[i].details.testID);
        printf("Duration: %d hours\n", (*records)[i].details.duration);
        printf("Environment: %s\n", (*records)[i].details.environment);
        printf("Results: ");
        for (int j = 0; j < (*records)[i].numResults; j++) {
            printf("%.2f ", (*records)[i].results[j]);
        }
        printf("\n\n");
    }
}
```

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

8. Welding Parameter Optimization

Description:

Develop a program to optimize welding parameters using structures for parameter sets, arrays for test outcomes, and unions for variable welding types.

Specifications:

Structure: Stores parameter ID, voltage, current, and speed.

Union: Represents welding types (MIG, TIG, or Arc).

Array: Test outcomes.

const Pointers: Protect parameter configurations.

Double Pointers: Manage dynamic parameter sets.

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

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



typedef union {
    char MIG[50];
    char TIG[50];
    char Arc[50];
} WeldingType;



typedef struct {
    const char *parameterID;
```

```
    float voltage;
    float current;
    float speed;
    WeldingType type;
    int weldingType; // 0 for MIG, 1 for TIG, 2 for Arc
} ParameterSet;


typedef struct {
    ParameterSet parameters;
    float outcomes[100];
    int numOutcomes;
} TestOutcome;


void addParameterSet(TestOutcome **outcomes, int *size, int *capacity, const char
*parameterID, float voltage, float current, float speed, WeldingType type, int weldingType);
void displayParameterSets(TestOutcome **outcomes, int size);

int main() {
    int size = 0, capacity = 2;
    TestOutcome *outcomes = (TestOutcome *)malloc(capacity * sizeof(TestOutcome));


    WeldingType type1;
    strcpy(type1.MIG, "MIG Welding");

    WeldingType type2;
```

```
    strcpy(type2.TIG, "TIG Welding");

    WeldingType type3;
    strcpy(type3.Arc, "Arc Welding");

    // Add parameter sets
    addParameterSet(&outcomes, &size, &capacity, "P001", 24.0, 200.0, 5.0, type1, 0);
    addParameterSet(&outcomes, &size, &capacity, "P002", 22.0, 180.0, 4.5, type2, 1);
    addParameterSet(&outcomes, &size, &capacity, "P003", 26.0, 220.0, 6.0, type3, 2);

    displayParameterSets(&outcomes, size);

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

    return 0;
}

// Function to add a parameter set to the list
void addParameterSet(TestOutcome **outcomes, int *size, int *capacity, const char
*parameterID, float voltage, float current, float speed, WeldingType type, int weldingType) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        TestOutcome *newOutcomes = (TestOutcome *)malloc(*capacity *
sizeof(TestOutcome));
```

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

        // Free old array and update pointer
        free(*outcomes);
        *outcomes = newOutcomes;
    }

    // Initialize the new parameter set
    (*outcomes)[*size].parameters.parameterID = strdup(parameterID);
    (*outcomes)[*size].parameters.voltage = voltage;
    (*outcomes)[*size].parameters.current = current;
    (*outcomes)[*size].parameters.speed = speed;
    (*outcomes)[*size].parameters.type = type;
    (*outcomes)[*size].parameters.weldingType = weldingType;
    (*outcomes)[*size].numOutcomes = 0; // Initialize the number of outcomes to zero

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

// Function to display the parameter sets
void displayParameterSets(TestOutcome **outcomes, int size) {
    printf("Welding Parameter Sets:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Parameter ID: %s\n", (*outcomes)[i].parameters.parameterID);
        printf("Voltage: %.2f V\n", (*outcomes)[i].parameters.voltage);
        printf("Current: %.2f A\n", (*outcomes)[i].parameters.current);
        printf("Speed: %.2f mm/s\n", (*outcomes)[i].parameters.speed);
        printf("Welding Type: ");
        switch ((*outcomes)[i].parameters.weldingType) {
            case 0:
                printf("%s\n", (*outcomes)[i].parameters.type.MIG);
                break;
            case 1:
                printf("%s\n", (*outcomes)[i].parameters.type.TIG);
                break;
            case 2:
                printf("%s\n", (*outcomes)[i].parameters.type.Arc);
                break;
        }
        printf("Test Outcomes: ");
        for (int j = 0; j < (*outcomes)[i].numOutcomes; j++) {
            printf("%.2f ", (*outcomes)[i].outcomes[j]);
        }
        printf("\n\n");
    }
}
```

```
/*******************************
9. Metal Surface Finish Analysis
Description:
Design a program to analyze surface finishes using arrays for measurement data, structures for test configurations, and strings for surface types.
Specifications:
Structure: Holds configuration ID, material, and measurement units.
Array: Surface finish measurements.
Strings: Surface types.
const Pointers: Protect configuration details.
Double Pointers: Allocate and manage measurement data.

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

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



typedef union {
   char MIG[50];
   char TIG[50];
   char Arc[50];
} WeldingType;



typedef struct {
   const char *parameterID;
```

```
    float voltage;
    float current;
    float speed;
    WeldingType type;
    int weldingType; // 0 for MIG, 1 for TIG, 2 for Arc
} ParameterSet;


typedef struct {
    ParameterSet parameters;
    float outcomes[100];
    int numOutcomes;
} TestOutcome;


void addParameterSet(TestOutcome **outcomes, int *size, int *capacity, const char
*parameterID, float voltage, float current, float speed, WeldingType type, int weldingType);
void displayParameterSets(TestOutcome **outcomes, int size);

int main() {
    int size = 0, capacity = 2;
    TestOutcome *outcomes = (TestOutcome *)malloc(capacity * sizeof(TestOutcome));


    WeldingType type1;
    strcpy(type1.MIG, "MIG Welding");

    WeldingType type2;
```

```
    strcpy(type2.TIG, "TIG Welding");

    WeldingType type3;
    strcpy(type3.Arc, "Arc Welding");

    // Add parameter sets
    addParameterSet(&outcomes, &size, &capacity, "P001", 24.0, 200.0, 5.0, type1, 0);
    addParameterSet(&outcomes, &size, &capacity, "P002", 22.0, 180.0, 4.5, type2, 1);
    addParameterSet(&outcomes, &size, &capacity, "P003", 26.0, 220.0, 6.0, type3, 2);

    displayParameterSets(&outcomes, size);

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

    return 0;
}

// Function to add a parameter set to the list
void addParameterSet(TestOutcome **outcomes, int *size, int *capacity, const char
*parameterID, float voltage, float current, float speed, WeldingType type, int weldingType) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        TestOutcome *newOutcomes = (TestOutcome *)malloc(*capacity *
sizeof(TestOutcome));
```

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

        // Free old array and update pointer
        free(*outcomes);
        *outcomes = newOutcomes;
    }

    // Initialize the new parameter set
    (*outcomes)[*size].parameters.parameterID = strdup(parameterID);
    (*outcomes)[*size].parameters.voltage = voltage;
    (*outcomes)[*size].parameters.current = current;
    (*outcomes)[*size].parameters.speed = speed;
    (*outcomes)[*size].parameters.type = type;
    (*outcomes)[*size].parameters.weldingType = weldingType;
    (*outcomes)[*size].numOutcomes = 0; // Initialize the number of outcomes to zero

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

// Function to display the parameter sets
void displayParameterSets(TestOutcome **outcomes, int size) {
    printf("Welding Parameter Sets:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Parameter ID: %s\n", (*outcomes)[i].parameters.parameterID);
        printf("Voltage: %.2f V\n", (*outcomes)[i].parameters.voltage);
        printf("Current: %.2f A\n", (*outcomes)[i].parameters.current);
        printf("Speed: %.2f mm/s\n", (*outcomes)[i].parameters.speed);
        printf("Welding Type: ");
        switch ((*outcomes)[i].parameters.weldingType) {
            case 0:
                printf("%s\n", (*outcomes)[i].parameters.type.MIG);
                break;
            case 1:
                printf("%s\n", (*outcomes)[i].parameters.type.TIG);
                break;
            case 2:
                printf("%s\n", (*outcomes)[i].parameters.type.Arc);
                break;
        }
        printf("Test Outcomes: ");
        for (int j = 0; j < (*outcomes)[i].numOutcomes; j++) {
            printf("%.2f ", (*outcomes)[i].outcomes[j]);
        }
        printf("\n\n");
    }
}
```

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

10. Smelting Process Tracker

Description:

Create a system to track smelting processes using structures for process metadata, arrays for heat data, and unions for variable ore properties.

Specifications:

Structure: Holds process ID, ore type, and temperature.

Union: Represents variable ore properties.

Array: Heat data.

const Pointers: Protect process metadata.

Double Pointers: Allocate dynamic process records

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

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

typedef struct {
    const char *processID;
    char oreType[50];
    float temperature;
} ProcessMetadata;

typedef union {
```

```
    float carbonContent;
    float sulfurContent;
    float phosphorusContent;
} OreProperties;

typedef struct {
    ProcessMetadata metadata;
    float heatData[100];
    int numHeatData;
    OreProperties oreProperties;
    int propertyType;
} HeatRecord;

void addHeatRecord(HeatRecord **records, int *size, int *capacity, const char *processID,
const char *oreType, float temperature, float *heatData, int numHeatData, OreProperties
oreProperties, int propertyType);
void displayHeatRecords(HeatRecord **records, int size);

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

    float heatData1[] = {1000.0, 1050.0, 1100.0, 1150.0};
    float heatData2[] = {1200.0, 1250.0, 1300.0, 1350.0, 1400.0};

    OreProperties properties1;
```

```
    properties1.carbonContent = 0.5;

    OreProperties properties2;
    properties2.sulfurContent = 0.2;

    addHeatRecord(&records, &size, &capacity, "P001", "Iron Ore", 1500.0, heatData1, 4, properties1, 0);
    addHeatRecord(&records, &size, &capacity, "P002", "Copper Ore", 1400.0, heatData2, 5, properties2, 1);

    displayHeatRecords(&records, size);

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

    return 0;
}

void addHeatRecord(HeatRecord **records, int *size, int *capacity, const char *processID, const char *oreType, float temperature, float *heatData, int numHeatData, OreProperties oreProperties, int propertyType) {

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

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



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



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



(*records)[*size].metadata.processID = strdup(processID);
strcpy((*records)[*size].metadata.oreType, oreType);
(*records)[*size].metadata.temperature = temperature;



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



(*records)[*size].oreProperties = oreProperties;
(*records)[*size].propertyType = propertyType;
```

```
    (*size)++;
}


void displayHeatRecords(HeatRecord **records, int size) {
    printf("Smelting Process Heat Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Process ID: %s\n", (*records)[i].metadata.processID);
        printf("Ore Type: %s\n", (*records)[i].metadata.oreType);
        printf("Temperature: %.2f°C\n", (*records)[i].metadata.temperature);
        printf("Heat Data: ");
        for (int j = 0; j < (*records)[i].numHeatData; j++) {
            printf("%.2f ", (*records)[i].heatData[j]);
        }
        printf("\nOre Properties: ");
        switch ((*records)[i].propertyType) {
            case 0:
                printf("Carbon Content: %.2f%%", (*records)[i].oreProperties.carbonContent);
                break;
            case 1:
                printf("Sulfur Content: %.2f%%", (*records)[i].oreProperties.sulfurContent);
                break;
            case 2:
                printf("Phosphorus Content: %.2f%%",
(*records)[i].oreProperties.phosphorusContent);
                break;
        }
        printf("\n\n");
```

```
    }
}

/*******************************
11. Electroplating System Simulation
Description:
Simulate an electroplating system using structures for metal ions, arrays for plating parameters, and strings for electrolyte names.
Specifications:
Structure: Stores ion type, charge, and concentration.
Array: Plating parameters.
Strings: Electrolyte names.
const Pointers: Protect ion data.
Double Pointers: Manage dynamic plating configurations.

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

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

typedef struct {
    const char *ionType;
    int charge;
    float concentration;
} MetalIon;
```

```
typedef struct {
    float currentDensity;
    float time;
    float temperature;
} PlatingParameters;



typedef struct {
    MetalIon ion;
    PlatingParameters parameters;
    char electrolyte[50];
} PlatingConfig;



void addPlatingConfig(PlatingConfig **configs, int *size, int *capacity, const char *ionType,
int charge, float concentration, PlatingParameters parameters, const char *electrolyte);
void displayPlatingConfigs(PlatingConfig **configs, int size);


int main() {
    int size = 0, capacity = 2;
    PlatingConfig *configs = (PlatingConfig *)malloc(capacity * sizeof(PlatingConfig));



    PlatingParameters params1 = {2.0, 5.0, 25.0};
    PlatingParameters params2 = {1.5, 6.0, 30.0};
```

```
    addPlatingConfig(&configs, &size, &capacity, "Cu2+", 2, 0.1, params1, "Copper Sulfate");
    addPlatingConfig(&configs, &size, &capacity, "Zn2+", 2, 0.05, params2, "Zinc Sulfate");

    displayPlatingConfigs(&configs, size);


    for (int i = 0; i < size; i++) {
        free((void *)configs[i].ion.ionType);
    }
    free(configs);

    return 0;
}


void addPlatingConfig(PlatingConfig **configs, int *size, int *capacity, const char *ionType,
int charge, float concentration, PlatingParameters parameters, const char *electrolyte) {

    if (*size == *capacity) {
        *capacity *= 2;
        PlatingConfig *newConfigs = (PlatingConfig *)malloc(*capacity * sizeof(PlatingConfig));


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

```
        free(*configs);
        *configs = newConfigs;
    }

    (*configs)[*size].ion.ionType = strdup(ionType);
    (*configs)[*size].ion.charge = charge;
    (*configs)[*size].ion.concentration = concentration;
    (*configs)[*size].parameters = parameters;
    strcpy((*configs)[*size].electrolyte, electrolyte);



    (*size)++;
}



void displayPlatingConfigs(PlatingConfig **configs, int size) {
    printf("Electroplating Configurations:\n");
    for (int i = 0; i < size; i++) {
        printf("Ion Type: %s\n", (*configs)[i].ion.ionType);
        printf("Charge: %d\n", (*configs)[i].ion.charge);
        printf("Concentration: %.2f M\n", (*configs)[i].ion.concentration);
        printf("Current Density: %.2f A/dm²\n", (*configs)[i].parameters.currentDensity);
        printf("Time: %.2f hours\n", (*configs)[i].parameters.time);
        printf("Temperature: %.2f°C\n", (*configs)[i].parameters.temperature);
        printf("Electrolyte: %s\n", (*configs)[i].electrolyte);
        printf("\n");
    }
```

}

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

12. Casting Defect Analysis

Description:

Design a system to analyze casting defects using arrays for defect data, structures for casting details, and unions for variable defect types.

Specifications:

Structure: Holds casting ID, material, and dimensions.

Union: Represents defect types (shrinkage or porosity).

Array: Defect data.

const Pointers: Protect casting data.

Double Pointers: Dynamic defect record management.

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

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

typedef struct {
    const char *castingID;
    char material[50];
    struct {
        float length;
        float width;
        float height;
    } dimensions;
```

```
} CastingDetails;



typedef union {
    struct {
        float shrinkagePercentage;
    } shrinkage;
    struct {
        float porosityLevel;
    } porosity;
} DefectType;



typedef struct {
    CastingDetails details;
    DefectType defect;
    int isShrinkage;
} DefectRecord;



void addDefectRecord(DefectRecord **records, int *size, int *capacity, const char *castingID,
const char *material, float length, float width, float height, DefectType defect, int
isShrinkage);
void displayDefectRecords(DefectRecord **records, int size);

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

```
    DefectType defect1;
    defect1.shrinkage.shrinkagePercentage = 2.5;

    DefectType defect2;
    defect2.porosity.porosityLevel = 0.3;

    // Add defect records
    addDefectRecord(&records, &size, &capacity, "C001", "Steel", 10.0, 5.0, 2.0, defect1, 1); // shrinkage
    addDefectRecord(&records, &size, &capacity, "C002", "Aluminum", 12.0, 6.0, 3.0, defect2, 0); // porosity

    displayDefectRecords(&records, size);

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

    return 0;
}

// Function to add a defect record to the list
void addDefectRecord(DefectRecord **records, int *size, int *capacity, const char *castingID, const char *material, float length, float width, float height, DefectType defect, int isShrinkage) {
    // Check if more memory needs to be allocated
```

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

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

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

    // Initialize the new defect record
    (*records)[*size].details.castingID = strdup(castingID);
    strcpy((*records)[*size].details.material, material);
    (*records)[*size].details.dimensions.length = length;
    (*records)[*size].details.dimensions.width = width;
    (*records)[*size].details.dimensions.height = height;

    (*records)[*size].defect = defect;
    (*records)[*size].isShrinkage = isShrinkage;

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

```
// Function to display the defect records
void displayDefectRecords(DefectRecord **records, int size) {
    printf("Casting Defect Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Casting ID: %s\n", (*records)[i].details.castingID);
        printf("Material: %s\n", (*records)[i].details.material);
        printf("Dimensions: %.2f x %.2f x %.2f cm\n", (*records)[i].details.dimensions.length,
(*records)[i].details.dimensions.width, (*records)[i].details.dimensions.height);
        if ((*records)[i].isShrinkage) {
            printf("Defect Type: Shrinkage\n");
            printf("Shrinkage Percentage: %.2f%%\n",
(*records)[i].defect.shrinkage.shrinkagePercentage);
        } else {
            printf("Defect Type: Porosity\n");
            printf("Porosity Level: %.2f%%\n", (*records)[i].defect.porosity.porosityLevel);
        }
        printf("\n");
    }
}
```

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

13. Metallurgical Lab Automation

Description:

Automate a metallurgical lab using structures for sample details, arrays for test results, and strings for equipment names.

Specifications:

Structure: Contains sample ID, type, and dimensions.

Array: Test results.

Strings: Equipment names.

const Pointers: Protect sample details.

Double Pointers: Allocate and manage dynamic test records.

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

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


typedef struct {
    const char *sampleID;
    char type[50];
    struct {
        float length;
        float width;
        float height;
    } dimensions;
} SampleDetails;
```

```
typedef struct {
    SampleDetails sample;
    float results[100];
    int numResults;
} TestRecord;


typedef struct {
    char equipmentName[50];
} Equipment;


void addTestRecord(TestRecord **records, int *size, int *capacity, const char *sampleID,
const char *type, float length, float width, float height, float *results, int numResults);
void displayTestRecords(TestRecord **records, int size);
void addEquipment(Equipment **equipments, int *size, int *capacity, const char
*equipmentName);
void displayEquipment(Equipment **equipments, int size);

int main() {
    int recordSize = 0, recordCapacity = 2;
    TestRecord *records = (TestRecord *)malloc(recordCapacity * sizeof(TestRecord));

    int equipmentSize = 0, equipmentCapacity = 2;
    Equipment *equipments = (Equipment *)malloc(equipmentCapacity * sizeof(Equipment));
```

```
    float results1[] = {85.5, 90.0, 88.3, 87.5};
    float results2[] = {75.0, 80.3, 78.4, 79.0};
    float results3[] = {92.5, 93.0, 91.8, 90.7};


    addTestRecord(&records, &recordSize, &recordCapacity, "S001", "Steel", 10.0, 5.0, 2.0,
results1, 4);
    addTestRecord(&records, &recordSize, &recordCapacity, "S002", "Aluminum", 12.0, 6.0,
3.0, results2, 4);
    addTestRecord(&records, &recordSize, &recordCapacity, "S003", "Copper", 8.0, 4.0, 2.5,
results3, 4);


    addEquipment(&equipments, &equipmentSize, &equipmentCapacity, "Spectrometer");
    addEquipment(&equipments, &equipmentSize, &equipmentCapacity, "Hardness Tester");
    addEquipment(&equipments, &equipmentSize, &equipmentCapacity, "Tensile Tester");

    displayTestRecords(&records, recordSize);
    displayEquipment(&equipments, equipmentSize);


    for (int i = 0; i < recordSize; i++) {
        free((void *)records[i].sample.sampleID);
    }
    free(records);
    free(equipments);

    return 0;
}
```

```
// Function to add a test record to the list
void addTestRecord(TestRecord **records, int *size, int *capacity, const char *sampleID,
const char *type, float length, float width, float height, float *results, int numResults) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        TestRecord *newRecords = (TestRecord *)malloc(*capacity * sizeof(TestRecord));

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

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

    // Initialize the new test record
    (*records)[*size].sample.sampleID = strdup(sampleID);
    strcpy((*records)[*size].sample.type, type);
    (*records)[*size].sample.dimensions.length = length;
    (*records)[*size].sample.dimensions.width = width;
    (*records)[*size].sample.dimensions.height = height;

    // Copy test results
    for (int i = 0; i < numResults; i++) {
```

```
        (*records)[*size].results[i] = results[i];
    }
    (*records)[*size].numResults = numResults;

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

// Function to display the test records
void displayTestRecords(TestRecord **records, int size) {
    printf("Metallurgical Lab Test Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Sample ID: %s\n", (*records)[i].sample.sampleID);
        printf("Type: %s\n", (*records)[i].sample.type);
        printf("Dimensions: %.2f x %.2f x %.2f cm\n", (*records)[i].sample.dimensions.length,
(*records)[i].sample.dimensions.width, (*records)[i].sample.dimensions.height);
        printf("Test Results: ");
        for (int j = 0; j < (*records)[i].numResults; j++) {
            printf("%.2f ", (*records)[i].results[j]);
        }
        printf("\n\n");
    }
}

// Function to add equipment to the list
void addEquipment(Equipment **equipments, int *size, int *capacity, const char
*equipmentName) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
```

```
        *capacity *= 2;
        Equipment *newEquipments = (Equipment *)malloc(*capacity * sizeof(Equipment));

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

        // Free old array and update pointer
        free(*equipments);
        *equipments = newEquipments;
    }

    // Initialize the new equipment
    strcpy((*equipments)[*size].equipmentName, equipmentName);

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

// Function to display the equipment
void displayEquipment(Equipment **equipments, int size) {
    printf("Lab Equipment:\n");
    for (int i = 0; i < size; i++) {
        printf("Equipment Name: %s\n", (*equipments)[i].equipmentName);
    }

}
```

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

14. Metal Hardness Testing System

Description:

Develop a program to track metal hardness tests using structures for test data, arrays for hardness values, and unions for variable hardness scales.

Specifications:

Structure: Stores test ID, method, and result.

Union: Represents variable hardness scales (Rockwell or Brinell).

Array: Hardness values.

const Pointers: Protect test data.

Double Pointers: Dynamic hardness record allocation

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

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

typedef union {
    float rockwellHardness;
    float brinellHardness;
} HardnessScale;

typedef struct {
```

```
    const char *testID;
    char method[50];
    HardnessScale result;
    int scaleType; // 0 for Rockwell, 1 for Brinell
} TestData;

typedef struct {
    TestData test;
    float values[100];
    int numValues;
} HardnessRecord;

void addHardnessRecord(HardnessRecord **records, int *size, int *capacity, const char *testID, const char *method, HardnessScale result, int scaleType, float *values, int numValues);
void displayHardnessRecords(HardnessRecord **records, int size);

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

    float values1[] = {70.0, 72.5, 75.0, 73.0};
    float values2[] = {250.0, 255.0, 260.0, 258.0};

    HardnessScale scale1;
```

```
    scale1.rockwellHardness = 72.5;

    HardnessScale scale2;
    scale2.brinellHardness = 255.0;

    addHardnessRecord(&records, &size, &capacity, "T001", "Rockwell", scale1, 0, values1, 4);
    addHardnessRecord(&records, &size, &capacity, "T002", "Brinell", scale2, 1, values2, 4);

    displayHardnessRecords(&records, size);

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

    return 0;
}

void addHardnessRecord(HardnessRecord **records, int *size, int *capacity, const char
*testID, const char *method, HardnessScale result, int scaleType, float *values, int
numValues) {

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

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


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


    (*records)[*size].test.testID = strdup(testID);
    strcpy((*records)[*size].test.method, method);
    (*records)[*size].test.result = result;
    (*records)[*size].test.scaleType = scaleType;

    // Copy hardness values
    for (int i = 0; i < numValues; i++) {
        (*records)[*size].values[i] = values[i];
    }
    (*records)[*size].numValues = numValues;

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

// Function to display the hardness records
void displayHardnessRecords(HardnessRecord **records, int size) {
```

```
    printf("Metal Hardness Test Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Test ID: %s\n", (*records)[i].test.testID);
        printf("Method: %s\n", (*records)[i].test.method);
        if ((*records)[i].test.scaleType == 0) {
            printf("Rockwell Hardness: %.2f\n", (*records)[i].test.result.rockwellHardness);
        } else {
            printf("Brinell Hardness: %.2f\n", (*records)[i].test.result.brinellHardness);
        }
        printf("Hardness Values: ");
        for (int j = 0; j < (*records)[i].numValues; j++) {
            printf("%.2f ", (*records)[i].values[j]);
        }
        printf("\n\n");
    }
}
```

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

15. Powder Metallurgy Process Tracker

Description:

Create a program to track powder metallurgy processes using structures for material details, arrays for particle size distribution, and unions for variable powder properties.

Specifications:

Structure: Contains material ID, type, and density.

Union: Represents powder properties.

Array: Particle size distribution data.

const Pointers: Protect material configurations.

Double Pointers: Allocate and manage powder data.

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

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


typedef struct {
    const char *materialID;
    char type[50];
    float density;
} MaterialDetails;


typedef union {
    float flowRate;
```

```
    float compressibility;
} PowderProperties;

typedef struct {
    MaterialDetails material;
    PowderProperties properties;
    int isFlowRate;
    float particleSizes[100];
    int numSizes;
} PowderData;

void addPowderData(PowderData **data, int *size, int *capacity, const char *materialID,
const char *type, float density, PowderProperties properties, int isFlowRate, float
*particleSizes, int numSizes);
void displayPowderData(PowderData **data, int size);

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

    float sizes1[] = {20.0, 30.0, 40.0, 50.0};
    float sizes2[] = {10.0, 15.0, 25.0, 35.0, 45.0};

    PowderProperties properties1;
    properties1.flowRate = 15.0;
```

```
    PowderProperties properties2;
    properties2.compressibility = 500.0;



    addPowderData(&data, &size, &capacity, "M001", "Iron Powder", 7.87, properties1, 1,
sizes1, 4);
    addPowderData(&data, &size, &capacity, "M002", "Copper Powder", 8.96, properties2, 0,
sizes2, 5);


    displayPowderData(&data, size);



    for (int i = 0; i < size; i++) {
        free((void *)data[i].material.materialID);
    }
    free(data);


    return 0;
}


// Function to add powder data to the list
void addPowderData(PowderData **data, int *size, int *capacity, const char *materialID,
const char *type, float density, PowderProperties properties, int isFlowRate, float
*particleSizes, int numSizes) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        PowderData *newData = (PowderData *)malloc(*capacity * sizeof(PowderData));
```

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

        // Free old array and update pointer
        free(*data);
        *data = newData;
    }

    // Initialize the new powder data record
    (*data)[*size].material.materialID = strdup(materialID);
    strcpy((*data)[*size].material.type, type);
    (*data)[*size].material.density = density;

    (*data)[*size].properties = properties;
    (*data)[*size].isFlowRate = isFlowRate;

    // Copy particle size distribution data
    for (int i = 0; i < numSizes; i++) {
        (*data)[*size].particleSizes[i] = particleSizes[i];
    }
    (*data)[*size].numSizes = numSizes;

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

```
// Function to display the powder data records
void displayPowderData(PowderData **data, int size) {
    printf("Powder Metallurgy Process Data:\n");
    for (int i = 0; i < size; i++) {
        printf("Material ID: %s\n", (*data)[i].material.materialID);
        printf("Type: %s\n", (*data)[i].material.type);
        printf("Density: %.2f g/cm³\n", (*data)[i].material.density);
        if ((*data)[i].isFlowRate) {
            printf("Flow Rate: %.2f s/50g\n", (*data)[i].properties.flowRate);
        } else {
            printf("Compressibility: %.2f MPa\n", (*data)[i].properties.compressibility);
        }
        printf("Particle Sizes: ");
        for (int j = 0; j < (*data)[i].numSizes; j++) {
            printf("%.2f µm ", (*data)[i].particleSizes[j]);
        }
        printf("\n\n");
    }
}
```

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

16. Metal Recycling Analysis

Description:

Develop a program to analyze recycled metal data using structures for material details, arrays for impurity levels, and strings for recycling methods.

Specifications:

Structure: Holds material ID, type, and recycling method.

Array: Impurity levels.

Strings: Recycling methods.

const Pointers: Protect material details.

Double Pointers: Allocate dynamic recycling records.

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

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



typedef struct {
    const char *materialID;
    char type[50];
    char recyclingMethod[50];
} MaterialDetails;



typedef struct {
    MaterialDetails material;
```

```
    float impurityLevels[100];
    int numImpurities;
} RecyclingRecord;

void addRecyclingRecord(RecyclingRecord **records, int *size, int *capacity, const char *materialID, const char *type, const char *recyclingMethod, float *impurityLevels, int numImpurities);
void displayRecyclingRecords(RecyclingRecord **records, int size);

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

    float impurities1[] = {0.01, 0.02, 0.015, 0.017};
    float impurities2[] = {0.005, 0.008, 0.01, 0.009};
    float impurities3[] = {0.02, 0.025, 0.022, 0.023};

    addRecyclingRecord(&records, &size, &capacity, "M001", "Steel", "Electrolytic", impurities1, 4);
    addRecyclingRecord(&records, &size, &capacity, "M002", "Aluminum", "Hydrometallurgical", impurities2, 4);
    addRecyclingRecord(&records, &size, &capacity, "M003", "Copper", "Pyrometallurgical", impurities3, 4);

    displayRecyclingRecords(&records, size);
```

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

    return 0;
}

// Function to add a recycling record to the list
void addRecyclingRecord(RecyclingRecord **records, int *size, int *capacity, const char *materialID, const char *type, const char *recyclingMethod, float *impurityLevels, int numImpurities) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        RecyclingRecord *newRecords = (RecyclingRecord *)malloc(*capacity * sizeof(RecyclingRecord));

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

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

```
    // Initialize the new recycling record
    (*records)[*size].material.materialID = strdup(materialID);
    strcpy((*records)[*size].material.type, type);
    strcpy((*records)[*size].material.recyclingMethod, recyclingMethod);

    // Copy impurity levels
    for (int i = 0; i < numImpurities; i++) {
        (*records)[*size].impurityLevels[i] = impurityLevels[i];
    }
    (*records)[*size].numImpurities = numImpurities;

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

// Function to display the recycling records
void displayRecyclingRecords(RecyclingRecord **records, int size) {
    printf("Metal Recycling Analysis Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Material ID: %s\n", (*records)[i].material.materialID);
        printf("Type: %s\n", (*records)[i].material.type);
        printf("Recycling Method: %s\n", (*records)[i].material.recyclingMethod);
        printf("Impurity Levels: ");
        for (int j = 0; j < (*records)[i].numImpurities; j++) {
            printf("%.3f%% ", (*records)[i].impurityLevels[j]);
        }
        printf("\n\n");
    }
```

}

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

17. Rolling Mill Performance Tracker

Description:

Design a system to track rolling mill performance using structures for mill configurations, arrays for output data, and strings for material types.

Specifications:

Structure: Stores mill ID, roll diameter, and speed.

Array: Output data.

Strings: Material types.

const Pointers: Protect mill configurations.

Double Pointers: Manage rolling mill records dynamically.

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

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


typedef struct {
    const char *millID;
    float rollDiameter;
    float speed;
} MillConfig;
```

```
typedef struct {
    MillConfig config;
    float outputData[100];
    int numOutputs;
    char materialType[50];
} RollingMillRecord;

void addRollingMillRecord(RollingMillRecord **records, int *size, int *capacity, const char *millID, float rollDiameter, float speed, float *outputData, int numOutputs, const char *materialType);
void displayRollingMillRecords(RollingMillRecord **records, int size);

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

    float outputData1[] = {100.0, 110.0, 105.0, 108.0};
    float outputData2[] = {95.0, 98.0, 97.5, 96.0};

    addRollingMillRecord(&records, &size, &capacity, "M001", 500.0, 250.0, outputData1, 4, "Steel");
    addRollingMillRecord(&records, &size, &capacity, "M002", 600.0, 300.0, outputData2, 4, "Aluminum");

    displayRollingMillRecords(&records, size);
```

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

    return 0;
}

// Function to add a rolling mill record to the list
void addRollingMillRecord(RollingMillRecord **records, int *size, int *capacity, const char *millID, float rollDiameter, float speed, float *outputData, int numOutputs, const char *materialType) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        RollingMillRecord *newRecords = (RollingMillRecord *)malloc(*capacity * sizeof(RollingMillRecord));

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

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

```
    // Initialize the new rolling mill record
    (*records)[*size].config.millID = strdup(millID);
    (*records)[*size].config.rollDiameter = rollDiameter;
    (*records)[*size].config.speed = speed;

    // Copy output data
    for (int i = 0; i < numOutputs; i++) {
        (*records)[*size].outputData[i] = outputData[i];
    }
    (*records)[*size].numOutputs = numOutputs;
    strcpy((*records)[*size].materialType, materialType);

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

// Function to display the rolling mill records
void displayRollingMillRecords(RollingMillRecord **records, int size) {
    printf("Rolling Mill Performance Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Mill ID: %s\n", (*records)[i].config.millID);
        printf("Roll Diameter: %.2f mm\n", (*records)[i].config.rollDiameter);
        printf("Speed: %.2f m/min\n", (*records)[i].config.speed);
        printf("Material Type: %s\n", (*records)[i].materialType);
        printf("Output Data: ");
        for (int j = 0; j < (*records)[i].numOutputs; j++) {
            printf("%.2f ", (*records)[i].outputData[j]);
```

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

```
/*******************************
18. Thermal Expansion Analysis
Description:
Create a program to analyze thermal expansion using arrays for temperature data, structures for material properties, and unions for variable coefficients.
Specifications:
Structure: Contains material ID, type, and expansion coefficient.
Union: Represents variable coefficients.
Array: Temperature data.
const Pointers: Protect material properties.
Double Pointers: Dynamic thermal expansion record allocation.

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

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

typedef union {
    float linearCoefficient;
    float volumetricCoefficient;
```

```
} ExpansionCoefficient;



typedef struct {
    const char *materialID;
    char type[50];
    ExpansionCoefficient coefficient;
    int coefficientType;
} MaterialProperties;



typedef struct {
    MaterialProperties material;
    float temperatureData[100];
    int numTemperatures;
} ThermalExpansionRecord;



void addThermalExpansionRecord(ThermalExpansionRecord **records, int *size, int
*capacity, const char *materialID, const char *type, ExpansionCoefficient coefficient, int
coefficientType, float *temperatureData, int numTemperatures);
void displayThermalExpansionRecords(ThermalExpansionRecord **records, int size);

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

```
    float temperatureData1[] = {20.0, 40.0, 60.0, 80.0, 100.0};
    float temperatureData2[] = {25.0, 50.0, 75.0, 100.0, 125.0};

    // Define some expansion coefficients
    ExpansionCoefficient coefficient1;
    coefficient1.linearCoefficient = 12.0e-6; // per °C

    ExpansionCoefficient coefficient2;
    coefficient2.volumetricCoefficient = 35.0e-6; // per °C

    // Add thermal expansion records
    addThermalExpansionRecord(&records, &size, &capacity, "M001", "Steel", coefficient1, 0, temperatureData1, 5);
    addThermalExpansionRecord(&records, &size, &capacity, "M002", "Aluminum", coefficient2, 1, temperatureData2, 5);

    displayThermalExpansionRecords(&records, size);

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

    return 0;
}

// Function to add a thermal expansion record to the list
```

```
void addThermalExpansionRecord(ThermalExpansionRecord **records, int *size, int *capacity, const char *materialID, const char *type, ExpansionCoefficient coefficient, int coefficientType, float *temperatureData, int numTemperatures) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        ThermalExpansionRecord *newRecords = (ThermalExpansionRecord *)malloc(*capacity * sizeof(ThermalExpansionRecord));

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

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

    // Initialize the new thermal expansion record
    (*records)[*size].material.materialID = strdup(materialID);
    strcpy((*records)[*size].material.type, type);
    (*records)[*size].material.coefficient = coefficient;
    (*records)[*size].material.coefficientType = coefficientType;

    // Copy temperature data
    for (int i = 0; i < numTemperatures; i++) {
        (*records)[*size].temperatureData[i] = temperatureData[i];
    }
```

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

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

// Function to display the thermal expansion records
void displayThermalExpansionRecords(ThermalExpansionRecord **records, int size) {
    printf("Thermal Expansion Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Material ID: %s\n", (*records)[i].material.materialID);
        printf("Type: %s\n", (*records)[i].material.type);
        if ((*records)[i].material.coefficientType == 0) {
            printf("Linear Expansion Coefficient: %.2e per °C\n",
(*records)[i].material.coefficient.linearCoefficient);
        } else {
            printf("Volumetric Expansion Coefficient: %.2e per °C\n",
(*records)[i].material.coefficient.volumetricCoefficient);
        }
        printf("Temperature Data: ");
        for (int j = 0; j < (*records)[i].numTemperatures; j++) {
            printf("%.2f °C ", (*records)[i].temperatureData[j]);
        }
        printf("\n\n");
    }
}
```

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

19. Metal Melting Point Analyzer

Description:

Develop a program to analyze melting points using structures for metal details, arrays for temperature data, and strings for metal names.

Specifications:

Structure: Stores metal ID, name, and melting point.

Array: Temperature data.

Strings: Metal names.

const Pointers: Protect metal details.

Double Pointers: Allocate dynamic melting point records.

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

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

typedef struct {
    const char *metalID;
    char name[50];
    float meltingPoint;
} MetalDetails;

typedef struct {
```

```
    MetalDetails metal;
    float temperatureData[100];
    int numTemperatures;
} MeltingPointRecord;


void addMeltingPointRecord(MeltingPointRecord **records, int *size, int *capacity, const char *metalID, const char *name, float meltingPoint, float *temperatureData, int numTemperatures);
void displayMeltingPointRecords(MeltingPointRecord **records, int size);

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

    float temperatureData1[] = {20.0, 50.0, 100.0, 150.0, 200.0};
    float temperatureData2[] = {25.0, 75.0, 125.0, 175.0, 225.0};

    addMeltingPointRecord(&records, &size, &capacity, "M001", "Iron", 1538.0, temperatureData1, 5);
    addMeltingPointRecord(&records, &size, &capacity, "M002", "Gold", 1064.0, temperatureData2, 5);

    displayMeltingPointRecords(&records, size);
```

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

    return 0;
}

void addMeltingPointRecord(MeltingPointRecord **records, int *size, int *capacity, const char *metalID, const char *name, float meltingPoint, float *temperatureData, int numTemperatures) {

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

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

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

```
    (*records)[*size].metal.metalID = strdup(metalID);
    strcpy((*records)[*size].metal.name, name);
    (*records)[*size].metal.meltingPoint = meltingPoint;


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


    (*size)++;
}


void displayMeltingPointRecords(MeltingPointRecord **records, int size) {
    printf("Metal Melting Point Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Metal ID: %s\n", (*records)[i].metal.metalID);
        printf("Name: %s\n", (*records)[i].metal.name);
        printf("Melting Point: %.2f °C\n", (*records)[i].metal.meltingPoint);
        printf("Temperature Data: ");
        for (int j = 0; j < (*records)[i].numTemperatures; j++) {
            printf("%.2f °C ", (*records)[i].temperatureData[j]);
        }
        printf("\n\n");
    }
}
```

```
/*******************************
20. Smelting Efficiency Analyzer
Description:
Design a system to analyze smelting efficiency using structures for
process details, arrays for energy consumption data, and unions for
variable process parameters.
Specifications:
Structure: Contains process ID, ore type, and efficiency.
Union: Represents process parameters (energy or duration).
Array: Energy consumption data.
const Pointers: Protect process configurations.
Double Pointers: Manage smelting efficiency records dynamically.

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

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



typedef struct {
    const char *processID;
    char oreType[50];
    float efficiency;
} ProcessDetails;



typedef union {
```

```
    float energyConsumption;
    float duration;
} ProcessParameters;


typedef struct {
    ProcessDetails details;
    ProcessParameters parameters;
    int isEnergy;
    float energyData[100];
    int numEnergyData;
} SmeltingEfficiencyRecord;

// Function prototypes
void addSmeltingEfficiencyRecord(SmeltingEfficiencyRecord **records, int *size, int *capacity, const char *processID, const char *oreType, float efficiency, ProcessParameters parameters, int isEnergy, float *energyData, int numEnergyData);
void displaySmeltingEfficiencyRecords(SmeltingEfficiencyRecord **records, int size);

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

    // Define some energy consumption data
    float energyData1[] = {1200.0, 1300.0, 1250.0, 1275.0};
    float energyData2[] = {1500.0, 1600.0, 1550.0, 1575.0};

    // Define some process parameters
```

```
    ProcessParameters parameters1;
    parameters1.energyConsumption = 1275.0;

    ProcessParameters parameters2;
    parameters2.duration = 5.0;

    // Add smelting efficiency records
    addSmeltingEfficiencyRecord(&records, &size, &capacity, "P001", "Iron Ore", 85.0,
parameters1, 1, energyData1, 4);
    addSmeltingEfficiencyRecord(&records, &size, &capacity, "P002", "Copper Ore", 90.0,
parameters2, 0, energyData2, 4);

    displaySmeltingEfficiencyRecords(&records, size);

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

    return 0;
}

// Function to add a smelting efficiency record to the list
void addSmeltingEfficiencyRecord(SmeltingEfficiencyRecord **records, int *size, int
*capacity, const char *processID, const char *oreType, float efficiency, ProcessParameters
parameters, int isEnergy, float *energyData, int numEnergyData) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
```

```
        *capacity *= 2;
        SmeltingEfficiencyRecord *newRecords = (SmeltingEfficiencyRecord *)malloc(*capacity
* sizeof(SmeltingEfficiencyRecord));

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

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

    // Initialize the new smelting efficiency record
    (*records)[*size].details.processID = strdup(processID);
    strcpy((*records)[*size].details.oreType, oreType);
    (*records)[*size].details.efficiency = efficiency;

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

    // Copy energy consumption data
    for (int i = 0; i < numEnergyData; i++) {
        (*records)[*size].energyData[i] = energyData[i];
    }
    (*records)[*size].numEnergyData = numEnergyData;
```

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

// Function to display the smelting efficiency records
void displaySmeltingEfficiencyRecords(SmeltingEfficiencyRecord **records, int size) {
    printf("Smelting Efficiency Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Process ID: %s\n", (*records)[i].details.processID);
        printf("Ore Type: %s\n", (*records)[i].details.oreType);
        printf("Efficiency: %.2f%%\n", (*records)[i].details.efficiency);
        if ((*records)[i].isEnergy) {
            printf("Energy Consumption: %.2f kWh\n",
(*records)[i].parameters.energyConsumption);
        } else {
            printf("Duration: %.2f hours\n", (*records)[i].parameters.duration);
        }
        printf("Energy Consumption Data: ");
        for (int j = 0; j < (*records)[i].numEnergyData; j++) {
            printf("%.2f kWh ", (*records)[i].energyData[j]);
        }
        printf("\n\n");
    }
}
```