SET OF PROGRAMS
=====================
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>

#define MAX_METALS 5

// Union to represent metal composition percentages
union AlloyComposition {
    float percentages[MAX_METALS]; // Array of percentages for metals in the alloy
};

// Structure to represent an alloy sample
struct AlloySample {
    int sampleID;
    char name[50];
    union AlloyComposition composition;  // Composition details using union
};

// Function to initialize a sample
void initializeSample(struct AlloySample *sample, int sampleID, const char *name, float *compositions) {
    sample->sampleID = sampleID;
    strncpy(sample->name, name, sizeof(sample->name) - 1);

    for (int i = 0; i < MAX_METALS; i++) {
        sample->composition.percentages[i] = compositions[i];
    }
}

// Function to print sample details
void printSampleDetails(const struct AlloySample *sample) {
    printf("Sample ID: %d\n", sample->sampleID);
    printf("Alloy Name: %s\n", sample->name);
    printf("Composition:\n");

    for (int i = 0; i < MAX_METALS; i++) {
        printf("Metal %d: %.2f%%\n", i + 1, sample->composition.percentages[i]);
    }
}

// Function to allocate memory for alloy samples dynamically using double pointers
void allocateSamples(struct AlloySample ***samples, int numSamples) {
    *samples = (struct AlloySample **)malloc(numSamples * sizeof(struct AlloySample *));
```

```c
    if (*samples == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    for (int i = 0; i < numSamples; i++) {
        (*samples)[i] = (struct AlloySample *)malloc(sizeof(struct AlloySample));
        if ((*samples)[i] == NULL) {
            printf("Memory allocation failed for sample %d!\n", i);
            exit(1);
        }
    }
}

// Function to free dynamically allocated memory
void freeSamples(struct AlloySample ***samples, int numSamples) {
    for (int i = 0; i < numSamples; i++) {
        free((*samples)[i]);
    }
    free(*samples);
}

int main() {
    // Number of alloy samples
    int numSamples = 3;

    // Dynamic allocation of alloy samples
    struct AlloySample **alloySamples = NULL;
    allocateSamples(&alloySamples, numSamples);

    // Example alloy compositions (metal percentages)
    float compositions1[MAX_METALS] = {30.5, 40.0, 10.0, 15.0, 4.5};
    float compositions2[MAX_METALS] = {25.0, 50.0, 5.0, 15.0, 5.0};
    float compositions3[MAX_METALS] = {20.0, 60.0, 5.0, 10.0, 5.0};

    // Initialize alloy samples
    initializeSample(alloySamples[0], 1, "Alloy A", compositions1);
    initializeSample(alloySamples[1], 2, "Alloy B", compositions2);
    initializeSample(alloySamples[2], 3, "Alloy C", compositions3);

    // Print alloy sample details
    for (int i = 0; i < numSamples; i++) {
        printf("\nSample %d Details:\n", i + 1);
        printSampleDetails(alloySamples[i]);
    }

    // Free dynamically allocated memory
    freeSamples(&alloySamples, numSamples);

    return 0;
}
```

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.

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

// Structure to hold process details
typedef struct {
    int processID;
    double temperature;  // Temperature in degrees Celsius
    double duration;     // Duration in hours
    double coolingRate;  // Cooling rate in degrees per minute
} HeatTreatmentProcess;

// Function to create and initialize a new heat treatment process
void createProcess(HeatTreatmentProcess *process, int id, double temp, double dur, double rate) {
    process->processID = id;
    process->temperature = temp;
    process->duration = dur;
    process->coolingRate = rate;
}

// Function to print the details of a heat treatment process
void printProcessDetails(const HeatTreatmentProcess *process) {
    printf("Process ID: %d\n", process->processID);
    printf("Temperature: %.2f°C\n", process->temperature);
    printf("Duration: %.2f hours\n", process->duration);
    printf("Cooling Rate: %.2f °C/min\n", process->coolingRate);
}

// Function to allocate memory for multiple processes
void allocateProcesses(HeatTreatmentProcess ***processArray, int numProcesses) {
    *processArray = (HeatTreatmentProcess **)malloc(numProcesses * sizeof(HeatTreatmentProcess *));
    for (int i = 0; i < numProcesses; i++) {
        (*processArray)[i] = (HeatTreatmentProcess *)malloc(sizeof(HeatTreatmentProcess));
    }
}

// Function to free memory allocated for processes
void freeProcesses(HeatTreatmentProcess ***processArray, int numProcesses) {
    for (int i = 0; i < numProcesses; i++) {
        free((*processArray)[i]);
    }
    free(*processArray);
}

int main() {
    int numProcesses = 3;
    HeatTreatmentProcess **processArray;
```

```
    // Dynamically allocate memory for the processes
    allocateProcesses(&processArray, numProcesses);

    // Initialize and assign values to processes
    createProcess(processArray[0], 101, 850.0, 2.5, 0.1);
    createProcess(processArray[1], 102, 900.0, 1.0, 0.2);
    createProcess(processArray[2], 103, 1050.0, 1.5, 0.15);

    // Process names (using strings)
    const char *processNames[] = {
        "Normalizing",
        "Annealing",
        "Quenching"
    };

    // Print details of each heat treatment process
    for (int i = 0; i < numProcesses; i++) {
        printf("\nProcess Name: %s\n", processNames[i]);
        printProcessDetails(processArray[i]);
    }

    // Free allocated memory
    freeProcesses(&processArray, numProcesses);

    return 0;
}
```

3. Steel Quality Monitoring
Description:
Create a system to monitor steel quality using structures for test results, arrays for storing test data, and unions for variable quality metrics like tensile strength and hardness.
Specifications:
Structure: Stores test ID, type, and result.
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>

// Define the Union for representing different quality metrics
union QualityMetrics {
    float tensileStrength;
    float hardness;
    float elongation;
};

// Define the Structure to store each test's ID, type, and the result (union)
struct SteelTest {
    int testID;                     // Test ID
    char testType[50];              // Type of test (e.g., tensile, hardness, elongation)
    union QualityMetrics result;    // Test result (can be tensile strength, hardness, or elongation)
```

```
};

// Function to create a new test and add it to dynamic array
void addTest(struct SteelTest **tests, int *numTests, int testID, const char *type, union QualityMetrics
result) {
    // Reallocate memory for the new test
    *tests = (struct SteelTest*) realloc(*tests, (*numTests + 1) * sizeof(struct SteelTest));

    if (*tests == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    // Set the test details
    (*tests)[*numTests].testID = testID;
    strcpy((*tests)[*numTests].testType, type);
    (*tests)[*numTests].result = result;

    // Increment the number of tests
    (*numTests)++;
}

// Function to print test results
void printTestResults(struct SteelTest *tests, int numTests) {
    for (int i = 0; i < numTests; i++) {
        printf("Test ID: %d\n", tests[i].testID);
        printf("Test Type: %s\n", tests[i].testType);

        if (strcmp(tests[i].testType, "Tensile") == 0) {
            printf("Tensile Strength: %.2f MPa\n", tests[i].result.tensileStrength);
        } else if (strcmp(tests[i].testType, "Hardness") == 0) {
            printf("Hardness: %.2f HRC\n", tests[i].result.hardness);
        } else if (strcmp(tests[i].testType, "Elongation") == 0) {
            printf("Elongation: %.2f %%\n", tests[i].result.elongation);
        }
        printf("\n");
    }
}

int main() {
    struct SteelTest *tests = NULL;  // Pointer to dynamically allocated array of tests
    int numTests = 0;                // Number of tests stored

    // Example of adding some test data
    union QualityMetrics result1;
    result1.tensileStrength = 500.0;  // Tensile Strength in MPa
    addTest(&tests, &numTests, 1, "Tensile", result1);

    union QualityMetrics result2;
    result2.hardness = 60.0;  // Hardness in HRC
    addTest(&tests, &numTests, 2, "Hardness", result2);

    union QualityMetrics result3;
    result3.elongation = 15.0;  // Elongation in percentage
    addTest(&tests, &numTests, 3, "Elongation", result3);
```

```
    // Print the test results
    printTestResults(tests, numTests);

    // Free allocated memory
    free(tests);

    return 0;
}
```

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>

// Define a structure for Material
struct Material {
    int materialID;
    char name[50];
    double enduranceLimit;  // Endurance limit of the material
};

// Function to calculate the number of cycles based on stress levels and endurance limit
int analyzeFatigue(double stress[], int numCycles, struct Material *material) {
    int failedCycles = 0;

    for (int i = 0; i < numCycles; i++) {
        if (stress[i] > material->enduranceLimit) {
            failedCycles++;
        }
    }
    return failedCycles;
}

int main() {
    int numMaterials, numCycles;

    // Example: Define materials and their details
    struct Material *materials = NULL;  // Pointer to materials array

    // Input number of materials
    printf("Enter the number of materials: ");
    scanf("%d", &numMaterials);

    // Allocate memory dynamically for materials
```

```
    materials = (struct Material *)malloc(numMaterials * sizeof(struct Material));

    // Input material details
    for (int i = 0; i < numMaterials; i++) {
        printf("Enter material ID for material %d: ", i + 1);
        scanf("%d", &(materials[i].materialID));
        printf("Enter name of material %d: ", i + 1);
        scanf("%s", materials[i].name);
        printf("Enter endurance limit for material %d: ", i + 1);
        scanf("%lf", &(materials[i].enduranceLimit));
    }

    // Input number of cycles in the stress data
    printf("Enter the number of stress cycles: ");
    scanf("%d", &numCycles);

    // Dynamically allocate memory for stress cycle data
    double *stressData = (double *)malloc(numCycles * sizeof(double));

    // Input stress values for each cycle
    printf("Enter stress values for %d cycles:\n", numCycles);
    for (int i = 0; i < numCycles; i++) {
        printf("Cycle %d: ", i + 1);
        scanf("%lf", &stressData[i]);
    }

    // Perform fatigue analysis for each material
    for (int i = 0; i < numMaterials; i++) {
        printf("\nAnalyzing fatigue for material %s (ID: %d)...\n", materials[i].name, materials[i].materialID);
        int failedCycles = analyzeFatigue(stressData, numCycles, &materials[i]);
        printf("Number of cycles exceeding endurance limit: %d\n", failedCycles);
    }

    // Free dynamically allocated memory
    free(materials);
    free(stressData);

    return 0;
}
```

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>
```

```
#define MAX_CASTINGS 100
#define MAX_EQUIPMENT 10

// Structure to store casting details
typedef struct {
    int casting_id;
    float weight;
    char material[50];
} Casting;

// Union to store mold properties (either dimensions or thermal conductivity)
typedef union {
    float dimensions[3];  // e.g., length, width, height
    float thermal_conductivity;  // for material's thermal conductivity
} MoldProperties;

// Structure for equipment details
typedef struct {
    int equipment_id;
    char description[100];
} Equipment;

// Array for storing foundry equipment data
Equipment equipment_list[MAX_EQUIPMENT];

// Pointer to an array of equipment data
Equipment *equipment_ptr = equipment_list;

// Double pointer for dynamically allocated casting records
Casting **casting_records;

// Function to allocate memory for casting records dynamically
void allocate_casting_records(int num_castings) {
    casting_records = (Casting **)malloc(num_castings * sizeof(Casting *));
    for (int i = 0; i < num_castings; i++) {
        casting_records[i] = (Casting *)malloc(sizeof(Casting));
    }
}

// Function to free dynamically allocated memory for casting records
void free_casting_records(int num_castings) {
    for (int i = 0; i < num_castings; i++) {
        free(casting_records[i]);
    }
    free(casting_records);
}

// Function to add equipment to the system
void add_equipment(int id, const char *description, int index) {
    equipment_ptr[index].equipment_id = id;
    strncpy(equipment_ptr[index].description, description, 100);
}

// Function to add casting record to the system
void add_casting(int casting_id, float weight, const char *material, int index) {
```

```
    casting_records[index]->casting_id = casting_id;
    casting_records[index]->weight = weight;
    strncpy(casting_records[index]->material, material, 50);
}

// Function to print equipment details
void print_equipment(int num_equipment) {
    printf("\nEquipment Details:\n");
    for (int i = 0; i < num_equipment; i++) {
        printf("Equipment %d: %s\n", equipment_ptr[i].equipment_id, equipment_ptr[i].description);
    }
}

// Function to print casting records
void print_castings(int num_castings) {
    printf("\nCasting Details:\n");
    for (int i = 0; i < num_castings; i++) {
        printf("Casting %d: %s, %.2f kg\n", casting_records[i]->casting_id, casting_records[i]->material,
casting_records[i]->weight);
    }
}

int main() {
    // Allocate memory for 5 casting records
    allocate_casting_records(MAX_CASTINGS);

    // Add some equipment to the foundry system
    add_equipment(101, "Furnace", 0);
    add_equipment(102, "Molding Machine", 1);
    add_equipment(103, "Cooling System", 2);

    // Add some casting records
    add_casting(1, 25.5, "Steel", 0);
    add_casting(2, 30.3, "Aluminum", 1);
    add_casting(3, 18.7, "Iron", 2);

    // Print the equipment details
    print_equipment(3);

    // Print the casting details
    print_castings(3);

    // Free dynamically allocated memory for casting records
    free_casting_records(MAX_CASTINGS);

    return 0;
}
```

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>

// Define union to represent different impurity types
union Impurity {
    float traceElements[10]; // Array for trace elements (up to 10 types of trace elements)
    float oxides[5];         // Array for oxides (up to 5 types of oxides)
};

// Define structure for metal sample data
struct MetalSample {
    int sampleID;              // Sample ID
    char metalType[50];        // Metal type (e.g., 'Aluminum', 'Steel')
    float purity;              // Purity of the sample (percentage)
    union Impurity impurity;   // Impurity type (trace elements or oxides)
};

// Function to initialize a metal sample
void initSample(struct MetalSample *sample, int id, const char *type, float purity) {
    sample->sampleID = id;
    snprintf(sample->metalType, sizeof(sample->metalType), "%s", type);
    sample->purity = purity;
}

// Function to add trace elements impurity data
void addTraceElements(struct MetalSample *sample, float *traceElements, int count) {
    for (int i = 0; i < count; i++) {
        sample->impurity.traceElements[i] = traceElements[i];
    }
}

// Function to add oxides impurity data
void addOxides(struct MetalSample *sample, float *oxides, int count) {
    for (int i = 0; i < count; i++) {
        sample->impurity.oxides[i] = oxides[i];
    }
}

// Function to print a metal sample's data
void printSampleData(const struct MetalSample *sample) {
    printf("Sample ID: %d\n", sample->sampleID);
    printf("Metal Type: %s\n", sample->metalType);
    printf("Purity: %.2f%%\n", sample->purity);

    // Print trace elements or oxides depending on which are available
    printf("Impurities:\n");
    if (sample->purity < 100.0f) {
        printf("- Trace Elements:\n");
        for (int i = 0; i < 10 && sample->impurity.traceElements[i] != 0; i++) {
            printf("  Element %d: %.2f%%\n", i + 1, sample->impurity.traceElements[i]);
        }
```

```
        printf("- Oxides:\n");
        for (int i = 0; i < 5 && sample->impurity.oxides[i] != 0; i++) {
            printf("  Oxide %d: %.2f%%\n", i + 1, sample->impurity.oxides[i]);
        }
    } else {
        printf("No impurities detected.\n");
    }
}

int main() {
    // Create an example sample
    struct MetalSample sample;
    initSample(&sample, 101, "Aluminum", 98.5f);

    // Define impurities for the sample
    float traceElements[] = {0.2f, 0.1f, 0.05f};  // 3 trace elements
    float oxides[] = {0.3f, 0.2f};                // 2 oxides

    // Add impurities to the sample
    addTraceElements(&sample, traceElements, 3);
    addOxides(&sample, oxides, 2);

    // Print the sample's data
    printSampleData(&sample);

    return 0;
}
```

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>

// Structure to hold test details
typedef struct {
    int testID;
    int duration; // in hours
    char environment[100]; // test environment (e.g., "Saltwater", "High Humidity")
} CorrosionTest;

// Function to create a corrosion test
CorrosionTest createTest(int testID, int duration, const char *environment) {
    CorrosionTest test;
    test.testID = testID;
```

```
    test.duration = duration;
    strcpy(test.environment, environment);
    return test;
}

// Function to dynamically allocate memory for an array of corrosion tests
CorrosionTest* allocateTests(int numTests) {
    return (CorrosionTest*)malloc(numTests * sizeof(CorrosionTest));
}

// Function to free dynamically allocated memory
void freeTests(CorrosionTest *tests) {
    free(tests);
}

int main() {
    int numTests, i;

    // Ask for the number of tests
    printf("Enter the number of corrosion tests: ");
    scanf("%d", &numTests);

    // Dynamic allocation for an array of corrosion tests using double pointer
    CorrosionTest *tests = allocateTests(numTests);

    // Array to store test results (e.g., corrosion rate measured in mm/year)
    double *testResults = (double*)malloc(numTests * sizeof(double));

    // Array of strings to store conditions for each test
    char **testConditions = (char**)malloc(numTests * sizeof(char*));

    // Input test details and results
    for (i = 0; i < numTests; i++) {
        printf("\nEnter details for test %d:\n", i + 1);

        // Input test details
        printf("Test ID: ");
        scanf("%d", &tests[i].testID);

        printf("Duration (in hours): ");
        scanf("%d", &tests[i].duration);

        // Input environment condition (with single word, space won't work here)
        printf("Environment: ");
        scanf("%s", tests[i].environment);  // Read single word

        // Dynamic allocation for the environment string if needed later
        testConditions[i] = (char*)malloc(100 * sizeof(char));
        strcpy(testConditions[i], tests[i].environment);  // Copy to dynamic array

        // Input test result
        printf("Enter test result (corrosion rate in mm/year): ");
        scanf("%lf", &testResults[i]);
    }
```

```
    // Display the collected test data
    printf("\nCorrosion Test Results:\n");
    for (i = 0; i < numTests; i++) {
        printf("\nTest ID: %d\n", tests[i].testID);
        printf("Duration: %d hours\n", tests[i].duration);
        printf("Environment: %s\n", tests[i].environment);
        printf("Corrosion Rate: %.2lf mm/year\n", testResults[i]);
    }

    // Free dynamically allocated memory
    free(testResults);
    for (i = 0; i < numTests; i++) {
        free(testConditions[i]);
    }
    freeTests(tests);

    return 0;
}
```

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>

// Union to represent welding types
typedef union {
    int mig_welding_type;   // Placeholder for MIG specific data
    int tig_welding_type;   // Placeholder for TIG specific data
    int arc_welding_type;   // Placeholder for Arc specific data
} WeldingType;

// Structure to hold welding parameters
typedef struct {
    int param_id;    // Parameter ID
    float voltage;   // Welding voltage
    float current;   // Welding current
    float speed;     // Welding speed
    WeldingType welding_type; // Union to store welding type
} WeldingParameters;

// Structure to store test outcomes (e.g., efficiency, pass/fail)
typedef struct {
    float outcome;  // Test result outcome (e.g., rating, efficiency)
    int pass;       // Pass/Fail result (0 = Fail, 1 = Pass)
} TestOutcome;
```

```
// Function to test the welding parameters
void test_welding_params(const WeldingParameters *params, TestOutcome *outcome) {
    // Simulate some test logic (e.g., evaluate welding efficiency)
    if (params->voltage > 20 && params->current > 50) {
        outcome->pass = 1;  // Test passes
        outcome->outcome = 95.5;  // Hypothetical rating
    } else {
        outcome->pass = 0;  // Test fails
        outcome->outcome = 40.0;  // Hypothetical rating
    }
    printf("Test Result for Param ID %d: %.2f, Pass: %d\n", params->param_id, outcome->outcome,
outcome->pass);
}

// Function to create a dynamic array of welding parameters
void create_dynamic_parameters(WeldingParameters ***params, int num_params) {
    // Allocate memory for an array of pointers to WeldingParameters
    *params = (WeldingParameters **)malloc(num_params * sizeof(WeldingParameters *));

    // Dynamically allocate memory for each welding parameter set
    for (int i = 0; i < num_params; i++) {
        (*params)[i] = (WeldingParameters *)malloc(sizeof(WeldingParameters));
        // Example initialization (adjust this as necessary)
        (*params)[i]->param_id = i + 1;
        (*params)[i]->voltage = 15.0 + i;  // Example voltage values
        (*params)[i]->current = 60.0 + i;  // Example current values
        (*params)[i]->speed = 5.0 + i;     // Example speed values
    }
}

// Function to free the dynamically allocated memory
void free_dynamic_parameters(WeldingParameters ***params, int num_params) {
    for (int i = 0; i < num_params; i++) {
        free((*params)[i]);
    }
    free(*params);
}

int main() {
    int num_params = 5;  // Example number of parameter sets
    WeldingParameters **params = NULL;  // Pointer to a dynamic array of WeldingParameters

    // Create the dynamic parameters
    create_dynamic_parameters(&params, num_params);

    // Array to store test outcomes
    TestOutcome outcome;

    // Test the welding parameters
    for (int i = 0; i < num_params; i++) {
        printf("Testing parameter set %d...\n", params[i]->param_id);
        test_welding_params(params[i], &outcome);
    }

    // Free dynamically allocated memory
```

```
    free_dynamic_parameters(&params, num_params);

    return 0;
}
```

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>

#define MAX_SURFACE_TYPES 5

// Structure to hold the test configuration details
typedef struct {
    int configID;
    char material[50];
    char units[10]; // Units like "microns" or "inches"
} TestConfig;

// Function to print the configuration details
void printConfig(const TestConfig *config) {
    printf("Configuration ID: %d\n", config->configID);
    printf("Material: %s\n", config->material);
    printf("Units: %s\n", config->units);
}

// Function to calculate the average surface finish from the measurement data
double calculateAverageSurfaceFinish(double **measurements, int numMeasurements) {
    double sum = 0.0;
    for (int i = 0; i < numMeasurements; i++) {
        sum += **(measurements + i); // Dereferencing the double pointer to get the value
    }
    return sum / numMeasurements;
}

int main() {
    // Initialize the test configuration
    TestConfig config = {1001, "Steel", "microns"};

    // Pointer to configuration (const to protect modification)
    const TestConfig *configPtr = &config;

    // Measurement data array using double pointers (for dynamic memory allocation)
    int numMeasurements = 5;
    double **measurements = (double **)malloc(numMeasurements * sizeof(double *));
```

```
    // Allocating memory for each measurement
    for (int i = 0; i < numMeasurements; i++) {
        measurements[i] = (double *)malloc(sizeof(double));
    }

    // Assign some surface finish measurements
    *(measurements[0]) = 1.2;
    *(measurements[1]) = 0.8;
    *(measurements[2]) = 1.5;
    *(measurements[3]) = 1.0;
    *(measurements[4]) = 1.3;

    // Initialize an array of surface types
    char *surfaceTypes[MAX_SURFACE_TYPES] = {"Smooth", "Rough", "Textured", "Polished",
"Granular"};

    // Print the configuration details
    printConfig(configPtr);

    // Print the surface types
    printf("\nSurface Types:\n");
    for (int i = 0; i < MAX_SURFACE_TYPES; i++) {
        printf("%d. %s\n", i + 1, surfaceTypes[i]);
    }

    // Calculate and print the average surface finish
    double avgSurfaceFinish = calculateAverageSurfaceFinish(measurements, numMeasurements);
    printf("\nAverage Surface Finish: %.2f %s\n", avgSurfaceFinish, config.units);

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

    return 0;
}
```

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>

// Define a union for different ore properties
```

```
typedef union {
    float iron_density;  // Iron ore density
    float copper_density; // Copper ore density
    float gold_density;   // Gold ore density
} OreProperties;

// Define a structure for the smelting process metadata
typedef struct {
    int processID;         // Unique process ID
    char oreType[50];      // Ore type (e.g., "Iron", "Copper", "Gold")
    float temperature;     // Temperature during smelting
    OreProperties oreProps; // Ore-specific properties stored in union
} ProcessMetadata;

// Function to allocate memory for a process and initialize it
void initSmeltingProcess(ProcessMetadata **process, int processID, const char *oreType, float
temperature, OreProperties oreProps) {
    *process = (ProcessMetadata *)malloc(sizeof(ProcessMetadata));  // Allocate memory for a new
process

    if (*process != NULL) {
        (*process)->processID = processID;
        strncpy((*process)->oreType, oreType, sizeof((*process)->oreType) - 1);
        (*process)->temperature = temperature;
        (*process)->oreProps = oreProps;  // Assign ore properties from the union
    } else {
        printf("Memory allocation failed!\n");
    }
}

// Function to track the heat data for smelting
void trackHeatData(float heat[], int dataSize) {
    printf("Tracking heat data:\n");
    for (int i = 0; i < dataSize; i++) {
        printf("Heat data at index %d: %.2f\n", i, heat[i]);
    }
}

// Function to display the process metadata
void displayProcessMetadata(const ProcessMetadata *process) {
    if (process != NULL) {
        printf("Process ID: %d\n", process->processID);
        printf("Ore Type: %s\n", process->oreType);
        printf("Temperature: %.2f\n", process->temperature);

        // Display ore properties based on ore type
        if (strcmp(process->oreType, "Iron") == 0) {
            printf("Iron Ore Density: %.2f\n", process->oreProps.iron_density);
        } else if (strcmp(process->oreType, "Copper") == 0) {
            printf("Copper Ore Density: %.2f\n", process->oreProps.copper_density);
        } else if (strcmp(process->oreType, "Gold") == 0) {
            printf("Gold Ore Density: %.2f\n", process->oreProps.gold_density);
        }
    } else {
        printf("Process is NULL!\n");
```

```
    }
}

int main() {
    // Example heat data for tracking
    float heatData[] = {1500.5, 1600.7, 1700.2};
    int heatDataSize = sizeof(heatData) / sizeof(heatData[0]);

    // Track heat data
    trackHeatData(heatData, heatDataSize);

    // Create and initialize a new smelting process
    ProcessMetadata *process = NULL;
    OreProperties ironOreProps;
    ironOreProps.iron_density = 7.87f; // Example density for Iron Ore

    initSmeltingProcess(&process, 1, "Iron", 1450.0f, ironOreProps);

    // Display the process metadata
    displayProcessMetadata(process);

    // Free the dynamically allocated memory
    free(process);

    return 0;
}
```

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>

// Structure to store ion data
struct Ion {
    char type[20];      // Type of metal ion (e.g., Copper, Gold)
    int charge;         // Charge of the ion (e.g., +2 for Copper)
    double concentration; // Concentration of the ion in molarity (mol/L)
};

// Function to simulate electroplating
void simulateElectroplating(struct Ion* ion, double voltage, double current, double time) {
    // Electroplating simulation logic (simplified)
    printf("Simulating electroplating with the following parameters:\n");
    printf("Ion Type: %s\n", ion->type);
    printf("Ion Charge: %d\n", ion->charge);
    printf("Ion Concentration: %.2f mol/L\n", ion->concentration);
```

```c
    printf("Voltage: %.2f V\n", voltage);
    printf("Current: %.2f A\n", current);
    printf("Time: %.2f hours\n", time);
    printf("Plating process in progress...\n");
    // Here, more complex electroplating formulas and calculations can be added.
}

int main() {
    // Creating an array of ion data
    struct Ion ionArray[2] = {
        {"Copper", 2, 0.1},  // Copper ion with +2 charge and concentration 0.1 mol/L
        {"Gold", 3, 0.05}    // Gold ion with +3 charge and concentration 0.05 mol/L
    };

    // Creating an array of electrolytes (strings)
    const char* electrolytes[] = {
        "Copper Sulfate",
        "Gold Chloride"
    };

    // Plating parameters (e.g., voltage, current, time)
    double platingParams[3] = {3.0, 0.5, 2.0}; // Voltage 3V, Current 0.5A, Time 2 hours

    // Using const pointers for ion data
    const struct Ion* ionPtr = &ionArray[0]; // Pointing to Copper ion
    printf("Electrolyte used: %s\n", electrolytes[0]);
    simulateElectroplating(ionPtr, platingParams[0], platingParams[1], platingParams[2]);

    // Dynamically allocating plating configurations using double pointers
    struct Ion** dynamicPlatingConfig = (struct Ion**) malloc(2 * sizeof(struct Ion*));
    dynamicPlatingConfig[0] = &ionArray[0];  // Copper ion
    dynamicPlatingConfig[1] = &ionArray[1];  // Gold ion

    // Simulate plating with dynamically allocated configuration
    printf("\nSimulating dynamic plating configurations:\n");
    for (int i = 0; i < 2; i++) {
        printf("\nUsing ion: %s\n", dynamicPlatingConfig[i]->type);
        simulateElectroplating(dynamicPlatingConfig[i], platingParams[0], platingParams[1],
platingParams[2]);
    }

    // Free the dynamically allocated memory
    free(dynamicPlatingConfig);

    return 0;
}
```

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.

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

// Structure to hold casting details
typedef struct {
    int castingID;
    char material[50];
    double length;
    double width;
    double height;
} Casting;

// Union to represent defect types (shrinkage or porosity)
typedef union {
    double shrinkageRate;  // Represents the percentage of shrinkage
    double porosityRate;   // Represents the percentage of porosity
} Defect;

// Structure to hold defect data
typedef struct {
    Casting castingDetails;  // Information about the casting
    Defect defect;           // Information about the defect
    int defectType;          // 1 for shrinkage, 2 for porosity
} DefectData;

// Function to print casting details (using const pointer)
void printCastingDetails(const Casting *casting) {
    printf("Casting ID: %d\n", casting->castingID);
    printf("Material: %s\n", casting->material);
    printf("Dimensions: %.2f x %.2f x %.2f\n",
           casting->length, casting->width, casting->height);
}

// Function to add defect data dynamically (using double pointer)
void addDefectData(DefectData **defectArray, int *defectCount, Casting casting, Defect defect, int
defectType) {
    // Reallocate memory for a new defect record
    *defectArray = realloc(*defectArray, sizeof(DefectData) * (*defectCount + 1));
    if (*defectArray == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    // Add new defect data
    (*defectArray)[*defectCount].castingDetails = casting;
    (*defectArray)[*defectCount].defect = defect;
    (*defectArray)[*defectCount].defectType = defectType;
    (*defectCount)++;
}

// Function to print defect data
```

```
void printDefectData(DefectData *defectData) {
    printCastingDetails(&defectData->castingDetails);
    if (defectData->defectType == 1) {
        printf("Defect Type: Shrinkage\n");
        printf("Shrinkage Rate: %.2f%%\n", defectData->defect.shrinkageRate);
    } else if (defectData->defectType == 2) {
        printf("Defect Type: Porosity\n");
        printf("Porosity Rate: %.2f%%\n", defectData->defect.porosityRate);
    }
}

int main() {
    // Sample castings
    Casting casting1 = {1001, "Aluminum", 10.0, 5.0, 2.5};
    Casting casting2 = {1002, "Steel", 8.0, 4.0, 3.0};

    // Allocate memory for defect data dynamically using double pointer
    DefectData *defectDataArray = NULL;
    int defectCount = 0;

    // Example of adding shrinkage defect
    Defect defect1;
    defect1.shrinkageRate = 5.5;
    addDefectData(&defectDataArray, &defectCount, casting1, defect1, 1);

    // Example of adding porosity defect
    Defect defect2;
    defect2.porosityRate = 2.3;
    addDefectData(&defectDataArray, &defectCount, casting2, defect2, 2);

    // Print defect data
    for (int i = 0; i < defectCount; i++) {
        printf("\nDefect #%d:\n", i + 1);
        printDefectData(&defectDataArray[i]);
    }

    // Free dynamically allocated memory
    free(defectDataArray);

    return 0;
}
```

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>
```

```
// Structure to hold sample details
typedef struct {
    int sampleID;
    char type[50];       // Sample type (e.g., "Steel", "Copper")
    double dimensions[3]; // Dimensions could represent length, width, height (or other properties)
} Sample;

// Structure to hold test details
typedef struct {
    int numTests;
    double *testResults; // Dynamically allocated array of test results
} TestResults;

// Function prototypes
void initializeSample(Sample *s, int id, const char *type, double length, double width, double height);
void addTestResults(TestResults *test, int numTests);
void displaySample(Sample *s);
void displayTestResults(TestResults *test);
void freeTestResults(TestResults *test);

int main() {
    // Sample details
    Sample sample1;
    initializeSample(&sample1, 101, "Steel", 50.5, 30.2, 10.3);

    // Test details (dynamically allocated)
    TestResults test1;
    addTestResults(&test1, 5);  // Let's say we have 5 test results

    // Example of storing test results
    for (int i = 0; i < test1.numTests; i++) {
        test1.testResults[i] = 10.5 * (i + 1); // Just an example of test results
    }

    // Displaying the data
    displaySample(&sample1);
    displayTestResults(&test1);

    // Free dynamically allocated memory
    freeTestResults(&test1);

    return 0;
}

// Function to initialize sample details
void initializeSample(Sample *s, int id, const char *type, double length, double width, double height) {
    s->sampleID = id;
    strncpy(s->type, type, sizeof(s->type) - 1);
    s->dimensions[0] = length;
    s->dimensions[1] = width;
    s->dimensions[2] = height;
}

// Function to add test results (dynamically allocate memory)
void addTestResults(TestResults *test, int numTests) {
```

```
    test->numTests = numTests;
    test->testResults = (double *)malloc(numTests * sizeof(double));
    if (test->testResults == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
}

// Function to display sample details
void displaySample(Sample *s) {
    printf("Sample ID: %d\n", s->sampleID);
    printf("Sample Type: %s\n", s->type);
    printf("Dimensions: %.2f x %.2f x %.2f\n", s->dimensions[0], s->dimensions[1], s->dimensions[2]);
}

// Function to display test results
void displayTestResults(TestResults *test) {
    printf("Test Results:\n");
    for (int i = 0; i < test->numTests; i++) {
        printf("Test %d: %.2f\n", i + 1, test->testResults[i]);
    }
}

// Function to free dynamically allocated memory for test results
void freeTestResults(TestResults *test) {
    free(test->testResults);
}
```

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>

// Union to represent variable hardness scales
union HardnessScale {
    double rockwell;
    double brinell;
};

// Structure to store test data
struct MetalTest {
    int testID;          // Test ID
    char method[20];     // Test method (e.g., Rockwell, Brinell)
    union HardnessScale result; // Hardness result based on the scale
};
```

```c
// Function to dynamically allocate and initialize a hardness record
void allocateTestRecord(struct MetalTest **testData, int numTests) {
    *testData = (struct MetalTest *)malloc(numTests * sizeof(struct MetalTest));
    if (*testData == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
}

// Function to input hardness values
void inputHardnessValues(struct MetalTest *testData, int numTests) {
    for (int i = 0; i < numTests; i++) {
        printf("Enter test ID for test %d: ", i + 1);
        scanf("%d", &testData[i].testID);
        printf("Enter method (Rockwell/Brinell) for test %d: ", i + 1);
        scanf("%s", testData[i].method);

        if (strcmp(testData[i].method, "Rockwell") == 0) {
            printf("Enter Rockwell hardness value for test %d: ", i + 1);
            scanf("%lf", &testData[i].result.rockwell);
        } else if (strcmp(testData[i].method, "Brinell") == 0) {
            printf("Enter Brinell hardness value for test %d: ", i + 1);
            scanf("%lf", &testData[i].result.brinell);
        } else {
            printf("Invalid method entered. Please enter either Rockwell or Brinell.\n");
            i--; // Retry the current test
        }
    }
}

// Function to display test data
void displayTestData(const struct MetalTest *testData, int numTests) {
    for (int i = 0; i < numTests; i++) {
        printf("\nTest ID: %d\n", testData[i].testID);
        printf("Method: %s\n", testData[i].method);
        if (strcmp(testData[i].method, "Rockwell") == 0) {
            printf("Rockwell Hardness: %.2f\n", testData[i].result.rockwell);
        } else if (strcmp(testData[i].method, "Brinell") == 0) {
            printf("Brinell Hardness: %.2f\n", testData[i].result.brinell);
        }
    }
}

// Free allocated memory for test records
void freeTestData(struct MetalTest **testData) {
    free(*testData);
    *testData = NULL;
}

int main() {
    int numTests;

    // Asking user for number of tests to track
    printf("Enter number of metal hardness tests: ");
    scanf("%d", &numTests);
```

```
    // Double pointer to handle dynamic allocation
    struct MetalTest *testData = NULL;

    // Allocate memory dynamically for the tests
    allocateTestRecord(&testData, numTests);

    // Input hardness values for the tests
    inputHardnessValues(testData, numTests);

    // Display the collected test data
    displayTestData(testData, numTests);

    // Free dynamically allocated memory
    freeTestData(&testData);

    return 0;
}
```

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>

// Define structure for material details
struct Material {
    int materialID;
    char materialType[50];
    double density;
};

// Define union for powder properties
union PowderProperties {
    double flowability;
    double particleSize;
    double porosity;
};

// Function to create and manage powder data dynamically
void managePowderData(double ***powderData, int numMaterials, int numSizes) {
    // Allocate memory for double pointer (array of arrays)
    *powderData = (double **)malloc(numMaterials * sizeof(double *));

    // Check if memory allocation was successful
    if (*powderData == NULL) {
```

```c
        printf("Memory allocation failed.\n");
        exit(1);
    }

    for (int i = 0; i < numMaterials; i++) {
        (*powderData)[i] = (double *)malloc(numSizes * sizeof(double));

        // Check if memory allocation was successful
        if ((*powderData)[i] == NULL) {
            printf("Memory allocation failed.\n");
            exit(1);
        }
    }
}

// Function to free allocated memory for powder data
void freePowderData(double ***powderData, int numMaterials) {
    for (int i = 0; i < numMaterials; i++) {
        free((*powderData)[i]);
    }
    free(*powderData);
}

int main() {
    // Create an example material
    struct Material material1 = {101, "Iron", 7.87};

    // Initialize union for powder properties (example with flowability)
    union PowderProperties powder1;
    powder1.flowability = 1.25; // Example value

    // Create an array for particle size distribution
    double particleSizeDistribution[] = {10.5, 20.1, 30.3, 40.2, 50.6}; // Example data

    // Print material details
    printf("Material ID: %d\n", material1.materialID);
    printf("Material Type: %s\n", material1.materialType);
    printf("Material Density: %.2f g/cm^3\n", material1.density);

    // Print powder properties
    printf("Powder Flowability: %.2f\n", powder1.flowability);

    // Print particle size distribution
    printf("Particle Size Distribution:\n");
    for (int i = 0; i < sizeof(particleSizeDistribution) / sizeof(particleSizeDistribution[0]); i++) {
        printf("%.2f ", particleSizeDistribution[i]);
    }
    printf("\n");

    // Dynamically allocate memory for powder data
    double **powderData;
    int numMaterials = 1;
    int numSizes = 5;
    managePowderData(&powderData, numMaterials, numSizes);
```

```
    // Example: Populate the powder data
    for (int i = 0; i < numMaterials; i++) {
        for (int j = 0; j < numSizes; j++) {
            powderData[i][j] = particleSizeDistribution[j] * (i + 1); // Just an example calculation
        }
    }

    // Print powder data
    printf("\nPowder Data (Particle Size Adjustments):\n");
    for (int i = 0; i < numMaterials; i++) {
        for (int j = 0; j < numSizes; j++) {
            printf("%.2f ", powderData[i][j]);
        }
        printf("\n");
    }

    // Free dynamically allocated memory
    freePowderData(&powderData, numMaterials);

    return 0;
}
```

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>

// Define structure for material details
struct Material {
    int materialID;
    char materialType[50];
    double density;
};

// Define union for powder properties
union PowderProperties {
    double flowability;
    double particleSize;
    double porosity;
};

// Function to create and manage powder data dynamically
void managePowderData(double ***powderData, int numMaterials, int numSizes) {
    // Allocate memory for double pointer (array of arrays)
    *powderData = (double **)malloc(numMaterials * sizeof(double *));

    // Check if memory allocation was successful
```

```
    if (*powderData == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    for (int i = 0; i < numMaterials; i++) {
        (*powderData)[i] = (double *)malloc(numSizes * sizeof(double));

        // Check if memory allocation was successful
        if ((*powderData)[i] == NULL) {
            printf("Memory allocation failed.\n");
            exit(1);
        }
    }
}

// Function to free allocated memory for powder data
void freePowderData(double ***powderData, int numMaterials) {
    for (int i = 0; i < numMaterials; i++) {
        free((*powderData)[i]);
    }
    free(*powderData);
}

int main() {
    // Create an example material
    struct Material material1 = {101, "Iron", 7.87};

    // Initialize union for powder properties (example with flowability)
    union PowderProperties powder1;
    powder1.flowability = 1.25; // Example value

    // Create an array for particle size distribution
    double particleSizeDistribution[] = {10.5, 20.1, 30.3, 40.2, 50.6}; // Example data

    // Print material details
    printf("Material ID: %d\n", material1.materialID);
    printf("Material Type: %s\n", material1.materialType);
    printf("Material Density: %.2f g/cm^3\n", material1.density);

    // Print powder properties
    printf("Powder Flowability: %.2f\n", powder1.flowability);

    // Print particle size distribution
    printf("Particle Size Distribution:\n");
    for (int i = 0; i < sizeof(particleSizeDistribution) / sizeof(particleSizeDistribution[0]); i++) {
        printf("%.2f ", particleSizeDistribution[i]);
    }
    printf("\n");

    // Dynamically allocate memory for powder data
    double **powderData;
    int numMaterials = 1;
    int numSizes = 5;
    managePowderData(&powderData, numMaterials, numSizes);
```

```c
    // Example: Populate the powder data
    for (int i = 0; i < numMaterials; i++) {
        for (int j = 0; j < numSizes; j++) {
            powderData[i][j] = particleSizeDistribution[j] * (i + 1); // Just an example calculation
        }
    }

    // Print powder data
    printf("\nPowder Data (Particle Size Adjustments):\n");
    for (int i = 0; i < numMaterials; i++) {
        for (int j = 0; j < numSizes; j++) {
            printf("%.2f ", powderData[i][j]);
        }
        printf("\n");
    }

    // Free dynamically allocated memory
    freePowderData(&powderData, numMaterials);

    return 0;
}
```

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.

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

// Structure to store mill configuration
struct RollingMill {
    int millID;
    float rollDiameter;
    float speed;
};

// Array of strings to store material types
#define MAX_MATERIALS 5
char* materialTypes[MAX_MATERIALS] = {"Steel", "Aluminum", "Copper", "Brass", "Titanium"};

// Function to dynamically allocate memory for mill records
void trackMillPerformance(struct RollingMill** mills, int* totalMills) {
    // Prompt user to enter number of mills to track
    printf("Enter the number of mills to track: ");
    scanf("%d", totalMills);
```

```c
    // Dynamically allocate memory for the mills
    *mills = (struct RollingMill*)malloc(*totalMills * sizeof(struct RollingMill));

    // Take input for each mill
    for (int i = 0; i < *totalMills; i++) {
        printf("\nEnter details for Mill %d:\n", i + 1);
        (*mills)[i].millID = i + 1;  // Auto-increment mill ID for simplicity
        printf("Enter roll diameter (in mm): ");
        scanf("%f", &(*mills)[i].rollDiameter);
        printf("Enter speed (in m/s): ");
        scanf("%f", &(*mills)[i].speed);
    }
}

// Function to display mill performance
void displayMillPerformance(struct RollingMill* mills, int totalMills) {
    printf("\nRolling Mill Performance Tracker:\n");
    printf("--------------------------------------------------\n");

    for (int i = 0; i < totalMills; i++) {
        printf("Mill ID: %d\n", mills[i].millID);
        printf("Roll Diameter: %.2f mm\n", mills[i].rollDiameter);
        printf("Speed: %.2f m/s\n", mills[i].speed);
        printf("Material Type: %s\n", materialTypes[i % MAX_MATERIALS]); // Assign material type
cyclically
        printf("--------------------------------------------------\n");
    }
}

int main() {
    struct RollingMill* mills = NULL;
    int totalMills = 0;

    // Track mill performance
    trackMillPerformance(&mills, &totalMills);

    // Display mill performance
    displayMillPerformance(mills, totalMills);

    // Free dynamically allocated memory
    free(mills);

    return 0;
}
```

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.

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

// Structure for material properties
typedef struct {
    int materialID;            // Material ID
    char type[50];             // Material type (e.g., Steel, Aluminum, etc.)
    double expansionCoefficient; // Thermal expansion coefficient
} MaterialProperties;

// Union for variable coefficients (assuming temperature-dependent expansion coefficient)
typedef union {
    double constantCoefficient;  // Constant thermal expansion coefficient
    double temperatureCoefficient; // A coefficient that varies with temperature
} ExpansionCoefficient;

// Array for temperature data (e.g., temperature readings)
#define NUM_TEMPERATURES 5
double temperatures[NUM_TEMPERATURES] = {100, 200, 300, 400, 500}; // Example temperature data

// Function to calculate thermal expansion
double calculateThermalExpansion(double initialLength, MaterialProperties material, double temperature)
{
    // For simplicity, assuming linear thermal expansion
    double deltaTemperature = temperature - 20.0; // Assuming initial temperature is 20°C
    return initialLength * material.expansionCoefficient * deltaTemperature;
}

// Function to print the thermal expansion results
void printThermalExpansion(MaterialProperties *material, double **expansionResults, int numMaterials,
int numTemperatures) {
    printf("Thermal Expansion Results:\n");
    for (int i = 0; i < numMaterials; i++) {
        printf("Material: %s (ID: %d)\n", material[i].type, material[i].materialID);
        for (int j = 0; j < numTemperatures; j++) {
            printf("Temperature: %.2f°C -> Expansion: %.6f\n", temperatures[j], expansionResults[i][j]);
        }
        printf("\n");
    }
}

int main() {
    // Example materials
    MaterialProperties materials[] = {
        {1, "Steel", 0.000012},    // Material 1: Steel with expansion coefficient
        {2, "Aluminum", 0.000022}  // Material 2: Aluminum with expansion coefficient
    };

    int numMaterials = sizeof(materials) / sizeof(materials[0]);

    // Dynamically allocate memory for thermal expansion results
    double **expansionResults = (double **)malloc(numMaterials * sizeof(double *));
    for (int i = 0; i < numMaterials; i++) {
        expansionResults[i] = (double *)malloc(NUM_TEMPERATURES * sizeof(double));
```

```
    }

    // Initial length for thermal expansion calculation
    double initialLength = 1.0; // meters (just an example)

    // Calculate the thermal expansion for each material at each temperature
    for (int i = 0; i < numMaterials; i++) {
        for (int j = 0; j < NUM_TEMPERATURES; j++) {
            expansionResults[i][j] = calculateThermalExpansion(initialLength, materials[i], temperatures[j]);
        }
    }

    // Print the thermal expansion results
    printThermalExpansion(materials, expansionResults, numMaterials, NUM_TEMPERATURES);

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

    return 0;
}
```

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>

// Define structure for Metal
typedef struct {
    int metalID;
    char name[30];       // Metal name (string)
    float meltingPoint;  // Melting point in Celsius
} Metal;

// Function to print the metal details
void printMetalDetails(const Metal *metal) {
    printf("Metal ID: %d\n", metal->metalID);
    printf("Metal Name: %s\n", metal->name);
    printf("Melting Point: %.2f°C\n", metal->meltingPoint);
}

// Function to allocate dynamic memory for melting point records
void allocateMeltingPoints(Metal **metalPtr, int numMetals) {
```

```c
    *metalPtr = (Metal *)malloc(numMetals * sizeof(Metal));
    if (*metalPtr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
}

// Function to input metal details
void inputMetalDetails(Metal *metal) {
    printf("Enter Metal ID: ");
    scanf("%d", &metal->metalID);
    getchar();  // to clear the newline character left by scanf

    printf("Enter Metal Name: ");
    fgets(metal->name, sizeof(metal->name), stdin);
    metal->name[strcspn(metal->name, "\n")] = 0;  // Remove the trailing newline character

    printf("Enter Melting Point (°C): ");
    scanf("%f", &metal->meltingPoint);
}

// Main function to interact with the user and store the details
int main() {
    int numMetals, i;
    Metal *metals = NULL; // Pointer to dynamically allocated array of Metal structs

    // Ask user for the number of metals to be analyzed
    printf("Enter number of metals: ");
    scanf("%d", &numMetals);

    // Allocate memory for the array of metals
    allocateMeltingPoints(&metals, numMetals);

    // Input metal details
    for (i = 0; i < numMetals; i++) {
        printf("\nEnter details for Metal %d:\n", i + 1);
        inputMetalDetails(&metals[i]);
    }

    // Print out the metal details
    printf("\nMetal Melting Point Details:\n");
    for (i = 0; i < numMetals; i++) {
        printf("\nMetal %d Details:\n", i + 1);
        printMetalDetails(&metals[i]);
    }

    // Free dynamically allocated memory
    free(metals);

    return 0;
}
```

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.

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

// Union to store either energy or duration
union ProcessParams {
    float energy;  // Energy consumption
    int duration;  // Duration of the process
};

// Structure to store process details
struct SmeltingProcess {
    int processID;
    char oreType[50];
    float efficiency;  // Efficiency percentage
    union ProcessParams params;  // Process parameters (energy or duration)
};

// Function to calculate and display efficiency
void displaySmeltingEfficiency(struct SmeltingProcess *process) {
    printf("Process ID: %d\n", process->processID);
    printf("Ore Type: %s\n", process->oreType);
    printf("Efficiency: %.2f%%\n", process->efficiency);

    // Depending on the process type, display either energy or duration
    printf("Parameter - ");
    if (process->efficiency > 75.0) {
        printf("Energy consumption: %.2f units\n", process->params.energy);
    } else {
        printf("Duration: %d minutes\n", process->params.duration);
    }
}

// Function to manage dynamic records using double pointers
void manageSmeltingProcesses(struct SmeltingProcess **processRecords, int numProcesses) {
    for (int i = 0; i < numProcesses; i++) {
        displaySmeltingEfficiency(processRecords[i]);
    }
}

int main() {
    // Sample energy consumption data (for simplicity, just two values for illustration)
    float energyData[] = {500.0, 300.0};
    int durationData[] = {90, 120};

    // Dynamically allocating memory for an array of SmeltingProcess structures
    int numProcesses = 2;
```

```c
    struct SmeltingProcess **processRecords = malloc(numProcesses * sizeof(struct SmeltingProcess*));

    // Initialize the processes
    for (int i = 0; i < numProcesses; i++) {
        processRecords[i] = malloc(sizeof(struct SmeltingProcess));
        processRecords[i]->processID = i + 1;

        if (i == 0) {
            snprintf(processRecords[i]->oreType, sizeof(processRecords[i]->oreType), "Iron");
            processRecords[i]->efficiency = 80.0;  // 80% efficiency
            processRecords[i]->params.energy = energyData[i];  // Energy for process 1
        } else {
            snprintf(processRecords[i]->oreType, sizeof(processRecords[i]->oreType), "Copper");
            processRecords[i]->efficiency = 60.0;  // 60% efficiency
            processRecords[i]->params.duration = durationData[i];  // Duration for process 2
        }
    }

    // Display efficiency data for each smelting process
    manageSmeltingProcesses(processRecords, numProcesses);

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

    return 0;
}
```