**23-01-2025**

**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>
* struct Composition {
* int sampleID;
* char name[50];
* union {
* float iron;
* float copper;
* float aluminum;
* } percentage;
* const char\* metalType;  // To specify the metal type
* };
* void addSample(struct Composition\*\* samples, int\* count, int id, char\* name, const char\* metalType, float percentage) {
* \*samples = realloc(\*samples, (\*count + 1) \* sizeof(struct Composition));
* struct Composition\* newSample = &(\*samples)[\*count];
* newSample->sampleID = id;
* for (int i = 0; name[i] != '\0'; i++) {
* newSample->name[i] = name[i];
* }
* newSample->metalType = metalType;
* if (metalType == "Iron") {
* newSample->percentage.iron = percentage;
* } else if (metalType == "Copper") {
* newSample->percentage.copper = percentage;
* } else if (metalType == "Aluminum") {
* newSample->percentage.aluminum = percentage;
* }
* (\*count)++;
* }
* void displaySamples(struct Composition\* samples, int count) {
* for (int i = 0; i < count; i++) {
* printf("Sample ID: %d\n", samples[i].sampleID);
* printf("Name: %s\n", samples[i].name);
* printf("Metal Type: %s\n", samples[i].metalType);
* if (samples[i].metalType == "Iron") {
* printf("Percentage Composition: %.2f%%\n", samples[i].percentage.iron);
* } else if (samples[i].metalType == "Copper") {
* printf("Percentage Composition: %.2f%%\n", samples[i].percentage.copper);
* } else if (samples[i].metalType == "Aluminum") {
* printf("Percentage Composition: %.2f%%\n", samples[i].percentage.aluminum);
* }
* printf("\n");
* }
* }
* void freeSamples(struct Composition\* samples) {
* free(samples);
* }
* int main() {
* struct Composition\* alloySamples = NULL;
* int sampleCount = 0;
* addSample(&alloySamples, &sampleCount, 101, "Alloy A", "Iron", 70.5);
* addSample(&alloySamples, &sampleCount, 102, "Alloy B", "Copper", 50.0);
* addSample(&alloySamples, &sampleCount, 103, "Alloy C", "Aluminum", 30.0);
* printf("\nAlloy Composition Analysis:\n");
* displaySamples(alloySamples, sampleCount);
* freeSamples(alloySamples);
* 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.
* #include <stdio.h>
* #include <stdlib.h>
* // Structure to hold heat treatment process details
* struct HeatTreatmentProcess {
* int processID;
* char name[50];
* float temperature;
* int duration; // in minutes
* float coolingRate;
* };
* void addProcess(struct HeatTreatmentProcess\*\* processes, int\* count, int id, char\* name, float temperature, int duration, float coolingRate) {
* \*processes = realloc(\*processes, (\*count + 1) \* sizeof(struct HeatTreatmentProcess));
* struct HeatTreatmentProcess\* newProcess = &(\*processes)[\*count];
* newProcess->processID = id;
* for (int i = 0; name[i] != '\0'; i++) {
* newProcess->name[i] = name[i];
* }
* newProcess->temperature = temperature;
* newProcess->duration = duration;
* newProcess->coolingRate = coolingRate;
* (\*count)++;
* }
* void displayProcesses(const struct HeatTreatmentProcess\* processes, int count) {
* for (int i = 0; i < count; i++) {
* printf("Process ID: %d\n", processes[i].processID);
* printf("Process Name: %s\n", processes[i].name);
* printf("Temperature: %.2f°C\n", processes[i].temperature);
* printf("Duration: %d minutes\n", processes[i].duration);
* printf("Cooling Rate: %.2f°C/min\n", processes[i].coolingRate);
* printf("\n");
* }
* }
* void freeProcesses(struct HeatTreatmentProcess\* processes) {
* free(processes);
* }
* int main() {
* struct HeatTreatmentProcess\* processes = NULL;
* int processCount = 0;
* addProcess(&processes, &processCount, 201, "Annealing", 750.0, 180, 5.0);
* addProcess(&processes, &processCount, 202, "Tempering", 500.0, 120, 2.0);
* addProcess(&processes, &processCount, 203, "Quenching", 850.0, 60, 10.0);
* printf("\nHeat Treatment Processes:\n");
* displayProcesses(processes, processCount);
* freeProcesses(processes);
* 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>
* struct TestResult {
* int testID;
* char type[30];
* float result;
* };
* union QualityMetric {
* float tensileStrength;
* float hardness;
* float elongation;
* };
* void allocateTestRecords(struct TestResult \*\*\*records, int n) {
* \*records = (struct TestResult \*\*)malloc(n \* sizeof(struct TestResult \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct TestResult \*)malloc(sizeof(struct TestResult));
* }
* }
* void freeTestRecords(struct TestResult \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of steel quality tests: ");
* scanf("%d", &n);
* struct TestResult \*\*testRecords;
* allocateTestRecords(&testRecords, n);
* union QualityMetric metric;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for test %d:\n", i + 1);
* printf("Test ID: ");
* scanf("%d", &testRecords[i]->testID);
* printf("Test Type (e.g., Tensile, Hardness, Elongation): ");
* scanf(" %[^\n]", testRecords[i]->type);
* if (testRecords[i]->type[0] == 'T' || testRecords[i]->type[0] == 't') {
* printf("Enter Tensile Strength (MPa): ");
* scanf("%f", &metric.tensileStrength);
* testRecords[i]->result = metric.tensileStrength;
* } else if (testRecords[i]->type[0] == 'H' || testRecords[i]->type[0] == 'h') {
* printf("Enter Hardness (HBW): ");
* scanf("%f", &metric.hardness);
* testRecords[i]->result = metric.hardness;
* } else if (testRecords[i]->type[0] == 'E' || testRecords[i]->type[0] == 'e') {
* printf("Enter Elongation (%%): ");
* scanf("%f", &metric.elongation);
* testRecords[i]->result = metric.elongation;
* } else {
* printf("Unknown test type. Result set to 0.\n");
* testRecords[i]->result = 0.0;
* }
* }
* printf("\nSteel Quality Test Records:\n");
* for (int i = 0; i < n; i++) {
* printf("Test ID: %d, Type: %s, Result: %.2f\n", testRecords[i]->testID, testRecords[i]->type, testRecords[i]->result);
* }
* freeTestRecords(&testRecords, n);
* 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>
* // Structure to hold material details
* struct Material {
* int materialID;
* char name[50];
* float enduranceLimit; // Stress limit in MPa
* };
* // Function to add a new material test data
* void addMaterial(struct Material\*\* materials, int\* count, int id, char\* name, float enduranceLimit) {
* \*materials = realloc(\*materials, (\*count + 1) \* sizeof(struct Material));
* struct Material\* newMaterial = &(\*materials)[\*count];
* newMaterial->materialID = id;
* for (int i = 0; name[i] != '\0'; i++) {
* newMaterial->name[i] = name[i];
* }
* newMaterial->enduranceLimit = enduranceLimit;
* (\*count)++;
* }
* // Function to display material details and stress cycle data
* void displayMaterials(const struct Material\* materials, int count, const int\* stressCycles, int cyclesCount) {
* for (int i = 0; i < count; i++) {
* printf("Material ID: %d\n", materials[i].materialID);
* printf("Material Name: %s\n", materials[i].name);
* printf("Endurance Limit: %.2f MPa\n", materials[i].enduranceLimit);
* printf("Stress Cycles:\n");
* for (int j = 0; j < cyclesCount; j++) {
* printf("Cycle %d: %d MPa\n", j + 1, stressCycles[j]);
* }
* printf("\n");
* }
* }
* // Function to free dynamically allocated memory
* void freeMaterials(struct Material\* materials) {
* free(materials);
* }
* int main() {
* struct Material\* materials = NULL;
* int materialCount = 0;
* // Example stress cycle data
* int stressCycles[] = {100, 200, 300, 400, 500};
* int cyclesCount = sizeof(stressCycles) / sizeof(stressCycles[0]);
* // Adding materials
* addMaterial(&materials, &materialCount, 101, "Steel", 250.0);
* addMaterial(&materials, &materialCount, 102, "Aluminum", 150.0);
* addMaterial(&materials, &materialCount, 103, "Titanium", 400.0);
* // Displaying materials and their stress cycles
* printf("\nMetal Fatigue Analysis:\n");
* displayMaterials(materials, materialCount, stressCycles, cyclesCount);
* // Free allocated memory
* freeMaterials(materials);
* 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>
* // Structure to store casting details
* struct Casting {
* int castingID;
* float weight;       // in kilograms
* char material[50];
* };
* // Union to represent mold properties
* union MoldProperties {
* struct {
* float length;   // in meters
* float width;    // in meters
* float height;   // in meters
* } dimensions;
* float thermalConductivity; // in W/mK
* };
* // Function to add casting records dynamically
* void addCasting(struct Casting\*\* castings, int\* count, int id, float weight, char\* material) {
* \*castings = realloc(\*castings, (\*count + 1) \* sizeof(struct Casting));
* struct Casting\* newCasting = &(\*castings)[\*count];
* newCasting->castingID = id;
* newCasting->weight = weight;
* for (int i = 0; material[i] != '\0'; i++) {
* newCasting->material[i] = material[i];
* }
* (\*count)++;
* }
* // Function to display casting and mold details
* void displayFoundry(const struct Casting\* castings, int castingCount, const int\* equipment, int equipmentCount, const union MoldProperties\* mold, int isDimensions) {
* printf("Foundry Management System:\n");
* // Display equipment data
* printf("\nEquipment Data:\n");
* for (int i = 0; i < equipmentCount; i++) {
* printf("Equipment %d: ID %d\n", i + 1, equipment[i]);
* }
* // Display casting details
* printf("\nCasting Records:\n");
* for (int i = 0; i < castingCount; i++) {
* printf("Casting ID: %d\n", castings[i].castingID);
* printf("Weight: %.2f kg\n", castings[i].weight);
* printf("Material: %s\n", castings[i].material);
* }
* // Display mold properties
* printf("\nMold Properties:\n");
* if (isDimensions) {
* printf("Dimensions: %.2f m x %.2f m x %.2f m\n", mold->dimensions.length, mold->dimensions.width, mold->dimensions.height);
* } else {
* printf("Thermal Conductivity: %.2f W/mK\n", mold->thermalConductivity);
* }
* }
* // Function to free dynamically allocated memory
* void freeCastings(struct Casting\* castings) {
* free(castings);
* }
* int main() {
* struct Casting\* castings = NULL;
* int castingCount = 0;
* // Example equipment data
* const int equipment[] = {101, 102, 103};
* int equipmentCount = sizeof(equipment) / sizeof(equipment[0]);
* // Example mold properties
* union MoldProperties mold;
* mold.dimensions.length = 2.5;
* mold.dimensions.width = 1.5;
* mold.dimensions.height = 1.0;
* // Adding casting records
* addCasting(&castings, &castingCount, 1, 500.0, "Steel");
* addCasting(&castings, &castingCount, 2, 350.0, "Aluminum");
* // Display foundry details
* displayFoundry(castings, castingCount, equipment, equipmentCount, &mold, 1);
* // Free allocated memory
* freeCastings(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>
* #include <string.h>
* // Structure for sample data
* struct Sample {
* int sampleID;
* char type[50];
* float purity; // Purity percentage
* };
* // Union for impurity type
* union ImpurityType {
* char traceElements[50];
* char oxides[50];
* };
* // Function to add sample data dynamically
* void addSample(struct Sample\*\* samples, int\* count, int id, const char\* type, float purity) {
* \*samples = realloc(\*samples, (\*count + 1) \* sizeof(struct Sample));
* (\*samples)[\*count].sampleID = id;
* strncpy((\*samples)[\*count].type, type, sizeof((\*samples)[\*count].type) - 1);
* (\*samples)[\*count].type[sizeof((\*samples)[\*count].type) - 1] = '\0';
* (\*samples)[\*count].purity = purity;
* (\*count)++;
* }
* // Function to display metal purity and impurity details
* void displaySamples(const struct Sample\* samples, int count, const float\* impurities, int impurityCount, const union ImpurityType\* impurityType, int isTraceElement) {
* printf("Metal Purity Analysis:\n");
* // Display sample details
* printf("\nSample Data:\n");
* for (int i = 0; i < count; i++) {
* printf("Sample ID: %d\n", samples[i].sampleID);
* printf("Type: %s\n", samples[i].type);
* printf("Purity: %.2f%%\n", samples[i].purity);
* }
* // Display impurity percentages
* printf("\nImpurity Percentages:\n");
* for (int i = 0; i < impurityCount; i++) {
* printf("Impurity %d: %.2f%%\n", i + 1, impurities[i]);
* }
* // Display impurity type
* printf("\nImpurity Type:\n");
* if (isTraceElement) {
* printf("Trace Elements: %s\n", impurityType->traceElements);
* } else {
* printf("Oxides: %s\n", impurityType->oxides);
* }
* }
* // Function to free dynamically allocated memory
* void freeSamples(struct Sample\* samples) {
* free(samples);
* }
* int main() {
* struct Sample\* samples = NULL;
* int sampleCount = 0;
* // Example impurity percentages
* float impurities[] = {0.5, 1.2, 0.8};
* int impurityCount = sizeof(impurities) / sizeof(impurities[0]);
* // Example impurity type
* union ImpurityType impurityType;
* strncpy(impurityType.traceElements, "Nickel, Chromium", sizeof(impurityType.traceElements) - 1);
* impurityType.traceElements[sizeof(impurityType.traceElements) - 1] = '\0';
* // Adding sample records
* addSample(&samples, &sampleCount, 1, "Iron", 98.7);
* addSample(&samples, &sampleCount, 2, "Copper", 99.2);
* // Display sample and impurity details
* displaySamples(samples, sampleCount, impurities, impurityCount, &impurityType, 1);
* // Free allocated memory
* freeSamples(samples);
* 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 for test details
* struct CorrosionTest {
* int testID;
* int duration; // Duration in hours
* char environment[50];
* };
* // Function to add a test record dynamically
* void addTest(struct CorrosionTest\*\* tests, int\* count, int id, int duration, const char\* environment) {
* \*tests = realloc(\*tests, (\*count + 1) \* sizeof(struct CorrosionTest));
* (\*tests)[\*count].testID = id;
* (\*tests)[\*count].duration = duration;
* strncpy((\*tests)[\*count].environment, environment, sizeof((\*tests)[\*count].environment) - 1);
* (\*tests)[\*count].environment[sizeof((\*tests)[\*count].environment) - 1] = '\0';
* (\*count)++;
* }
* // Function to display test details and results
* void displayTests(const struct CorrosionTest\* tests, int count, const float\* results, int resultCount, const char\* conditions) {
* printf("Corrosion Testing System:\n");
* // Display test details
* printf("\nTest Details:\n");
* for (int i = 0; i < count; i++) {
* printf("Test ID: %d\n", tests[i].testID);
* printf("Duration: %d hours\n", tests[i].duration);
* printf("Environment: %s\n", tests[i].environment);
* }
* // Display test results
* printf("\nTest Results:\n");
* for (int i = 0; i < resultCount; i++) {
* printf("Result %d: %.2f%% corrosion\n", i + 1, results[i]);
* }
* // Display test conditions
* printf("\nTest Conditions: %s\n", conditions);
* }
* // Function to free allocated memory
* void freeTests(struct CorrosionTest\* tests) {
* free(tests);
* }
* int main() {
* struct CorrosionTest\* tests = NULL;
* int testCount = 0;
* // Example test results
* float results[] = {5.2, 3.8, 6.1};
* int resultCount = sizeof(results) / sizeof(results[0]);
* // Example test conditions
* const char\* conditions = "Temperature: 25°C, Humidity: 80%";
* // Adding test records
* addTest(&tests, &testCount, 101, 48, "Salt Spray");
* addTest(&tests, &testCount, 102, 72, "High Humidity");
* // Display test details, results, and conditions
* displayTests(tests, testCount, results, resultCount, conditions);
* // Free allocated memory
* 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>
* #include <string.h>
* // Structure for welding parameters
* struct WeldingParameters {
* int paramID;
* float voltage;
* float current;
* float speed; // Welding speed in mm/s
* };
* // Union for welding types
* union WeldingType {
* char MIG[10];
* char TIG[10];
* char Arc[10];
* };
* // Function to add parameter sets dynamically
* void addParameters(struct WeldingParameters\*\* params, int\* count, int id, float voltage, float current, float speed) {
* \*params = realloc(\*params, (\*count + 1) \* sizeof(struct WeldingParameters));
* (\*params)[\*count].paramID = id;
* (\*params)[\*count].voltage = voltage;
* (\*params)[\*count].current = current;
* (\*params)[\*count].speed = speed;
* (\*count)++;
* }
* // Function to display parameters and test outcomes
* void displayOptimization(const struct WeldingParameters\* params, int count, const float\* outcomes, int outcomeCount, const union WeldingType\* type, const char\* weldingMode) {
* printf("Welding Parameter Optimization:\n");
* // Display welding type
* printf("\nWelding Type: %s\n", weldingMode);
* // Display parameter details
* printf("\nWelding Parameters:\n");
* for (int i = 0; i < count; i++) {
* printf("Parameter ID: %d\n", params[i].paramID);
* printf("Voltage: %.2f V\n", params[i].voltage);
* printf("Current: %.2f A\n", params[i].current);
* printf("Speed: %.2f mm/s\n", params[i].speed);
* }
* // Display test outcomes
* printf("\nTest Outcomes:\n");
* for (int i = 0; i < outcomeCount; i++) {
* printf("Outcome %d: %.2f quality score\n", i + 1, outcomes[i]);
* }
* }
* // Function to free allocated memory
* void freeParameters(struct WeldingParameters\* params) {
* free(params);
* }
* int main() {
* struct WeldingParameters\* params = NULL;
* int paramCount = 0;
* // Example test outcomes
* float outcomes[] = {85.5, 90.2, 88.8};
* int outcomeCount = sizeof(outcomes) / sizeof(outcomes[0]);
* // Example welding type
* union WeldingType type;
* strncpy(type.MIG, "MIG", sizeof(type.MIG) - 1);
* // Adding parameter sets
* addParameters(&params, &paramCount, 201, 24.5, 120.0, 15.0);
* addParameters(&params, &paramCount, 202, 25.0, 115.0, 12.5);
* // Display parameter details and test outcomes
* displayOptimization(params, paramCount, outcomes, outcomeCount, &type, "MIG");
* // Free allocated memory
* freeParameters(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>
* // Structure for test configurations
* struct TestConfig {
* int configID;
* char material[50];
* char measurementUnit[20];
* };
* // Function to add measurements dynamically
* void addMeasurements(float\*\* measurements, int\* count, float value) {
* \*measurements = realloc(\*measurements, (\*count + 1) \* sizeof(float));
* (\*measurements)[\*count] = value;
* (\*count)++;
* }
* // Function to display analysis details
* void displayAnalysis(const struct TestConfig\* config, const char\* surfaceType, const float\* measurements, int count) {
* printf("Metal Surface Finish Analysis:\n");
* printf("\nTest Configuration:\n");
* printf("Configuration ID: %d\n", config->configID);
* printf("Material: %s\n", config->material);
* printf("Measurement Unit: %s\n", config->measurementUnit);
* printf("\nSurface Type: %s\n", surfaceType);
* printf("\nMeasurements:\n");
* for (int i = 0; i < count; i++) {
* printf("Measurement %d: %.2f %s\n", i + 1, measurements[i], config->measurementUnit);
* }
* }
* // Function to free allocated memory
* void freeMeasurements(float\* measurements) {
* free(measurements);
* }
* int main() {
* struct TestConfig config = {101, "Stainless Steel", "micrometers"};
* const char surfaceType[] = "Polished";
* float\* measurements = NULL;
* int measurementCount = 0;
* // Adding measurements
* addMeasurements(&measurements, &measurementCount, 0.85);
* addMeasurements(&measurements, &measurementCount, 0.90);
* addMeasurements(&measurements, &measurementCount, 0.78);
* // Display analysis details
* displayAnalysis(&config, surfaceType, measurements, measurementCount);
* // Free allocated memory
* freeMeasurements(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>
* struct ProcessMetadata {
* int processID;
* char oreType[30];
* float temperature;
* };
* union OreProperties {
* float density;
* float moistureContent;
* float chemicalComposition;
* };
* void allocateProcessRecords(struct ProcessMetadata \*\*\*records, int n) {
* \*records = (struct ProcessMetadata \*\*)malloc(n \* sizeof(struct ProcessMetadata \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct ProcessMetadata \*)malloc(sizeof(struct ProcessMetadata));
* }
* }
* void freeProcessRecords(struct ProcessMetadata \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of smelting processes: ");
* scanf("%d", &n);
* struct ProcessMetadata \*\*processRecords;
* allocateProcessRecords(&processRecords, n);
* union OreProperties oreProperties;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for smelting process %d:\n", i + 1);
* printf("Process ID: ");
* scanf("%d", &processRecords[i]->processID);
* printf("Ore Type: ");
* scanf(" %[^\n]", processRecords[i]->oreType);
* printf("Temperature (in Celsius): ");
* scanf("%f", &processRecords[i]->temperature);
* printf("Select Ore Property (1: Density, 2: Moisture Content, 3: Chemical Composition): ");
* int choice;
* scanf("%d", &choice);
* if (choice == 1) {
* printf("Enter Density of Ore: ");
* scanf("%f", &oreProperties.density);
* } else if (choice == 2) {
* printf("Enter Moisture Content of Ore: ");
* scanf("%f", &oreProperties.moistureContent);
* } else if (choice == 3) {
* printf("Enter Chemical Composition of Ore: ");
* scanf("%f", &oreProperties.chemicalComposition);
* } else {
* printf("Invalid choice. No ore property assigned.\n");
* }
* }
* printf("\nSmelting Process Records:\n");
* for (int i = 0; i < n; i++) {
* printf("\nProcess ID: %d\nOre Type: %s\nTemperature: %.2f\n", processRecords[i]->processID, processRecords[i]->oreType,
* processRecords[i]->temperature);
* }
* freeProcessRecords(&processRecords, n);
* 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>
* struct MetalIon {
* char ionType[20];
* float charge;
* float concentration;
* };
* void allocatePlatingConfigurations(struct MetalIon \*\*\*configurations, int n) {
* \*configurations = (struct MetalIon \*\*)malloc(n \* sizeof(struct MetalIon \*));
* for (int i = 0; i < n; i++) {
* (\*configurations)[i] = (struct MetalIon \*)malloc(sizeof(struct MetalIon));
* }
* }
* void freePlatingConfigurations(struct MetalIon \*\*\*configurations, int n) {
* for (int i = 0; i < n; i++) {
* free((\*configurations)[i]);
* }
* free(\*configurations);
* }
* int main() {
* int n;
* printf("Enter the number of electroplating configurations: ");
* scanf("%d", &n);
* struct MetalIon \*\*platingConfigurations;
* allocatePlatingConfigurations(&platingConfigurations, n);
* char electrolyteName[50];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for electroplating configuration %d:\n", i + 1);
* printf("Ion Type (e.g., Copper, Nickel): ");
* scanf(" %[^\n]", platingConfigurations[i]->ionType);
* printf("Charge (in Coulombs): ");
* scanf("%f", &platingConfigurations[i]->charge);
* printf("Concentration (in mol/L): ");
* scanf("%f", &platingConfigurations[i]->concentration);
* printf("Electrolyte Name: ");
* scanf(" %[^\n]", electrolyteName);  // Use string to store electrolyte name
* printf("Electrolyte used: %s\n", electrolyteName);  // Display electrolyte name
* }
* printf("\nElectroplating System Configurations:\n");
* for (int i = 0; i < n; i++) {
* printf("\nPlating Configuration %d:\n", i + 1);
* printf("Ion Type: %s\n", platingConfigurations[i]->ionType);
* printf("Charge: %.2f Coulombs\n", platingConfigurations[i]->charge);
* printf("Concentration: %.2f mol/L\n", platingConfigurations[i]->concentration);
* }
* freePlatingConfigurations(&platingConfigurations, n);
* 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.
* #include <stdio.h>
* #include <stdlib.h>
* struct CastingDetails {
* int castingID;
* char material[30];
* float length;
* float width;
* float height;
* };
* union DefectType {
* float shrinkage;  // Shrinkage percentage
* float porosity;   // Porosity percentage
* };
* void allocateDefectRecords(struct CastingDetails \*\*\*records, int n) {
* \*records = (struct CastingDetails \*\*)malloc(n \* sizeof(struct CastingDetails \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct CastingDetails \*)malloc(sizeof(struct CastingDetails));
* }
* }
* void freeDefectRecords(struct CastingDetails \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of castings to analyze: ");
* scanf("%d", &n);
* struct CastingDetails \*\*castingRecords;
* allocateDefectRecords(&castingRecords, n);
* union DefectType defect;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for casting %d:\n", i + 1);
* printf("Casting ID: ");
* scanf("%d", &castingRecords[i]->castingID);
* printf("Material: ");
* scanf(" %[^\n]", castingRecords[i]->material);
* printf("Dimensions (Length, Width, Height in cm): ");
* scanf("%f %f %f", &castingRecords[i]->length, &castingRecords[i]->width, &castingRecords[i]->height);
* printf("Enter defect type (1 for Shrinkage, 2 for Porosity): ");
* int defectType;
* scanf("%d", &defectType);
* if (defectType == 1) {
* printf("Enter Shrinkage percentage: ");
* scanf("%f", &defect.shrinkage);
* printf("Shrinkage: %.2f%%\n", defect.shrinkage);
* } else if (defectType == 2) {
* printf("Enter Porosity percentage: ");
* scanf("%f", &defect.porosity);
* printf("Porosity: %.2f%%\n", defect.porosity);
* } else {
* printf("Invalid defect type.\n");
* }
* }
* printf("\nCasting Defect Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nCasting ID: %d\nMaterial: %s\nDimensions: %.2f x %.2f x %.2f cm\n", castingRecords[i]->castingID,
* castingRecords[i]->material, castingRecords[i]->length, castingRecords[i]->width, castingRecords[i]->height);
* }
* freeDefectRecords(&castingRecords, n);
* 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>
* struct SampleDetails {
* int sampleID;
* char type[30];
* float length;
* float width;
* float height;
* };
* void allocateTestRecords(float \*\*\*records, int n) {
* \*records = (float \*\*)malloc(n \* sizeof(float \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (float \*)malloc(3 \* sizeof(float));  // Allocating space for three test results
* }
* }
* void freeTestRecords(float \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of metallurgical samples: ");
* scanf("%d", &n);
* struct SampleDetails \*samples = (struct SampleDetails \*)malloc(n \* sizeof(struct SampleDetails));
* float \*\*testResults;
* allocateTestRecords(&testResults, n);
* char equipmentName[50];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for sample %d:\n", i + 1);
* printf("Sample ID: ");
* scanf("%d", &samples[i].sampleID);
* printf("Sample Type: ");
* scanf(" %[^\n]", samples[i].type);
* printf("Dimensions (Length, Width, Height in cm): ");
* scanf("%f %f %f", &samples[i].length, &samples[i].width, &samples[i].height);
* printf("Enter test results (e.g., hardness, tensile strength, etc.): ");
* scanf("%f %f %f", &testResults[i][0], &testResults[i][1], &testResults[i][2]);
* printf("Enter equipment used: ");
* scanf(" %[^\n]", equipmentName);
* printf("Equipment used: %s\n", equipmentName);
* }
* printf("\nMetallurgical Lab Sample and Test Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nSample ID: %d\nType: %s\nDimensions: %.2f x %.2f x %.2f cm\n", samples[i].sampleID, samples[i].type,
* samples[i].length, samples[i].width, samples[i].height);
* printf("Test Results: Hardness: %.2f, Tensile Strength: %.2f, Other: %.2f\n", testResults[i][0], testResults[i][1],
* testResults[i][2]);
* }
* free(samples);
* freeTestRecords(&testResults, n);
* return 0;
* }

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>
* struct HardnessTest {
* int testID;
* char method[30];
* float result;
* };
* union HardnessScale {
* float rockwell;  // Rockwell hardness value
* float brinell;   // Brinell hardness value
* };
* void allocateHardnessRecords(struct HardnessTest \*\*\*records, int n) {
* \*records = (struct HardnessTest \*\*)malloc(n \* sizeof(struct HardnessTest \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct HardnessTest \*)malloc(sizeof(struct HardnessTest));
* }
* }
* void freeHardnessRecords(struct HardnessTest \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of metal hardness tests: ");
* scanf("%d", &n);
* struct HardnessTest \*\*hardnessRecords;
* allocateHardnessRecords(&hardnessRecords, n);
* union HardnessScale hardness;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for test %d:\n", i + 1);
* printf("Test ID: ");
* scanf("%d", &hardnessRecords[i]->testID);
* printf("Test Method (Rockwell / Brinell): ");
* scanf(" %[^\n]", hardnessRecords[i]->method);
* printf("Test Result (hardness value): ");
* scanf("%f", &hardnessRecords[i]->result);
* printf("Enter hardness scale (1 for Rockwell, 2 for Brinell): ");
* int scale;
* scanf("%d", &scale);
* if (scale == 1) {
* printf("Enter Rockwell hardness value: ");
* scanf("%f", &hardness.rockwell);
* printf("Rockwell Hardness: %.2f\n", hardness.rockwell);
* } else if (scale == 2) {
* printf("Enter Brinell hardness value: ");
* scanf("%f", &hardness.brinell);
* printf("Brinell Hardness: %.2f\n", hardness.brinell);
* } else {
* printf("Invalid scale selection.\n");
* }
* }
* printf("\nMetal Hardness Test Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nTest ID: %d\nMethod: %s\nResult: %.2f\n", hardnessRecords[i]->testID, hardnessRecords[i]->method,
* hardnessRecords[i]->result);
* }
* freeHardnessRecords(&hardnessRecords, n);
* 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>
* struct MaterialDetails {
* int materialID;
* char type[30];
* float density;
* };
* union PowderProperties {
* float purity;       // Purity of the powder
* float moisture;     // Moisture content in the powder
* };
* void allocatePowderData(struct MaterialDetails \*\*\*records, int n) {
* \*records = (struct MaterialDetails \*\*)malloc(n \* sizeof(struct MaterialDetails \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct MaterialDetails \*)malloc(sizeof(struct MaterialDetails));
* }
* }
* void freePowderData(struct MaterialDetails \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of powder metallurgy materials: ");
* scanf("%d", &n);
* struct MaterialDetails \*\*materialRecords;
* allocatePowderData(&materialRecords, n);
* union PowderProperties powder;
* float \*particleSizeDistribution = (float \*)malloc(n \* sizeof(float));
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for material %d:\n", i + 1);
* printf("Material ID: ");
* scanf("%d", &materialRecords[i]->materialID);
* printf("Material Type: ");
* scanf(" %[^\n]", materialRecords[i]->type);
* printf("Density (g/cm^3): ");
* scanf("%f", &materialRecords[i]->density);
* printf("Enter particle size distribution data (in microns): ");
* scanf("%f", &particleSizeDistribution[i]);
* printf("Enter powder property type (1 for Purity, 2 for Moisture): ");
* int propertyType;
* scanf("%d", &propertyType);
* if (propertyType == 1) {
* printf("Enter powder purity percentage: ");
* scanf("%f", &powder.purity);
* printf("Purity: %.2f%%\n", powder.purity);
* } else if (propertyType == 2) {
* printf("Enter powder moisture percentage: ");
* scanf("%f", &powder.moisture);
* printf("Moisture: %.2f%%\n", powder.moisture);
* } else {
* printf("Invalid property type.\n");
* }
* }
* printf("\nPowder Metallurgy Process Tracker Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMaterial ID: %d\nType: %s\nDensity: %.2f g/cm^3\n", materialRecords[i]->materialID,
* materialRecords[i]->type, materialRecords[i]->density);
* printf("Particle Size Distribution: %.2f microns\n", particleSizeDistribution[i]);
* }
* free(particleSizeDistribution);
* freePowderData(&materialRecords, n);
* 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>
* struct MaterialDetails {
* int materialID;
* char type[30];
* char recyclingMethod[30];
* };
* void allocateRecyclingData(struct MaterialDetails \*\*\*records, int n) {
* \*records = (struct MaterialDetails \*\*)malloc(n \* sizeof(struct MaterialDetails \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct MaterialDetails \*)malloc(sizeof(struct MaterialDetails));
* }
* }
* void freeRecyclingData(struct MaterialDetails \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of recycled metal materials: ");
* scanf("%d", &n);
* struct MaterialDetails \*\*materialRecords;
* allocateRecyclingData(&materialRecords, n);
* float \*impurityLevels = (float \*)malloc(n \* sizeof(float));
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for material %d:\n", i + 1);
* printf("Material ID: ");
* scanf("%d", &materialRecords[i]->materialID);
* printf("Material Type: ");
* scanf(" %[^\n]", materialRecords[i]->type);
* printf("Recycling Method: ");
* scanf(" %[^\n]", materialRecords[i]->recyclingMethod);
* printf("Impurity Level (percentage): ");
* scanf("%f", &impurityLevels[i]);
* }
* printf("\nMetal Recycling Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMaterial ID: %d\nType: %s\nRecycling Method: %s\n", materialRecords[i]->materialID,
* materialRecords[i]->type, materialRecords[i]->recyclingMethod);
* printf("Impurity Level: %.2f%%\n", impurityLevels[i]);
* }
* free(impurityLevels);
* freeRecyclingData(&materialRecords, n);
* 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.
* #include <stdio.h>
* #include <stdlib.h>
* struct MillConfig {
* int millID;
* float rollDiameter;
* float speed;
* };
* void allocateMillRecords(struct MillConfig \*\*\*records, int n) {
* \*records = (struct MillConfig \*\*)malloc(n \* sizeof(struct MillConfig \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct MillConfig \*)malloc(sizeof(struct MillConfig));
* }
* }
* void freeMillRecords(struct MillConfig \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of rolling mills to track: ");
* scanf("%d", &n);
* struct MillConfig \*\*millRecords;
* allocateMillRecords(&millRecords, n);
* float \*outputData = (float \*)malloc(n \* sizeof(float));
* char materialTypes[n][30];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for rolling mill %d:\n", i + 1);
* printf("Mill ID: ");
* scanf("%d", &millRecords[i]->millID);
* printf("Roll Diameter (in inches): ");
* scanf("%f", &millRecords[i]->rollDiameter);
* printf("Speed (in RPM): ");
* scanf("%f", &millRecords[i]->speed);
* printf("Output Data (in tons per hour): ");
* scanf("%f", &outputData[i]);
* printf("Material Type: ");
* scanf(" %[^\n]", materialTypes[i]);
* }
* printf("\nRolling Mill Performance Tracker Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMill ID: %d\nRoll Diameter: %.2f inches\nSpeed: %.2f RPM\n", millRecords[i]->millID,
* millRecords[i]->rollDiameter, millRecords[i]->speed);
* printf("Output Data: %.2f tons per hour\nMaterial Type: %s\n", outputData[i], materialTypes[i]);
* }
* free(outputData);
* freeMillRecords(&millRecords, n);
* 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.
* #include <stdio.h>
* #include <stdlib.h>
* struct MaterialProperties {
* int materialID;
* char type[30];
* float expansionCoefficient;
* };
* union Coefficients {
* float linear;      // Linear expansion coefficient
* float volumetric;  // Volumetric expansion coefficient
* };
* void allocateThermalData(struct MaterialProperties \*\*\*records, int n) {
* \*records = (struct MaterialProperties \*\*)malloc(n \* sizeof(struct MaterialProperties \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct MaterialProperties \*)malloc(sizeof(struct MaterialProperties));
* }
* }
* void freeThermalData(struct MaterialProperties \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of materials for thermal expansion analysis: ");
* scanf("%d", &n);
* struct MaterialProperties \*\*materialRecords;
* allocateThermalData(&materialRecords, n);
* float \*temperatureData = (float \*)malloc(n \* sizeof(float));
* union Coefficients coefficients;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for material %d:\n", i + 1);
* printf("Material ID: ");
* scanf("%d", &materialRecords[i]->materialID);
* printf("Material Type: ");
* scanf(" %[^\n]", materialRecords[i]->type);
* printf("Expansion Coefficient (in 1/°C): ");
* scanf("%f", &materialRecords[i]->expansionCoefficient);
* printf("Enter temperature data for analysis (in °C): ");
* scanf("%f", &temperatureData[i]);
* printf("Enter coefficient type (1 for Linear, 2 for Volumetric): ");
* int coefficientType;
* scanf("%d", &coefficientType);
* if (coefficientType == 1) {
* printf("Enter linear expansion coefficient: ");
* scanf("%f", &coefficients.linear);
* printf("Linear Coefficient: %.6f 1/°C\n", coefficients.linear);
* } else if (coefficientType == 2) {
* printf("Enter volumetric expansion coefficient: ");
* scanf("%f", &coefficients.volumetric);
* printf("Volumetric Coefficient: %.6f 1/°C\n", coefficients.volumetric);
* } else {
* printf("Invalid coefficient type.\n");
* }
* }
* printf("\nThermal Expansion Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMaterial ID: %d\nType: %s\nExpansion Coefficient: %.6f 1/°C\n", materialRecords[i]->materialID,
* materialRecords[i]->type, materialRecords[i]->expansionCoefficient);
* printf("Temperature: %.2f °C\n", temperatureData[i]);
* }
* free(temperatureData);
* freeThermalData(&materialRecords, n);
* 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>
* struct MetalDetails {
* int metalID;
* char name[30];
* float meltingPoint;
* };
* void allocateMeltingPointData(struct MetalDetails \*\*\*records, int n) {
* \*records = (struct MetalDetails \*\*)malloc(n \* sizeof(struct MetalDetails \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct MetalDetails \*)malloc(sizeof(struct MetalDetails));
* }
* }
* void freeMeltingPointData(struct MetalDetails \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of metals for melting point analysis: ");
* scanf("%d", &n);
* struct MetalDetails \*\*metalRecords;
* allocateMeltingPointData(&metalRecords, n);
* float \*temperatureData = (float \*)malloc(n \* sizeof(float));
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for metal %d:\n", i + 1);
* printf("Metal ID: ");
* scanf("%d", &metalRecords[i]->metalID);
* printf("Metal Name: ");
* scanf(" %[^\n]", metalRecords[i]->name);
* printf("Melting Point (in °C): ");
* scanf("%f", &metalRecords[i]->meltingPoint);
* printf("Enter temperature data (in °C): ");
* scanf("%f", &temperatureData[i]);
* }
* printf("\nMetal Melting Point Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMetal ID: %d\nName: %s\nMelting Point: %.2f °C\n", metalRecords[i]->metalID,
* metalRecords[i]->name, metalRecords[i]->meltingPoint);
* printf("Temperature: %.2f °C\n", temperatureData[i]);
* }
* free(temperatureData);
* freeMeltingPointData(&metalRecords, n);
* 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.
* #include <stdio.h>
* #include <stdlib.h>
* struct ProcessDetails {
* int processID;
* char oreType[30];
* float efficiency;
* };
* union ProcessParameters {
* float energy;  // Energy consumption for the process
* float duration; // Duration of the process
* };
* void allocateSmeltingData(struct ProcessDetails \*\*\*records, int n) {
* \*records = (struct ProcessDetails \*\*)malloc(n \* sizeof(struct ProcessDetails \*));
* for (int i = 0; i < n; i++) {
* (\*records)[i] = (struct ProcessDetails \*)malloc(sizeof(struct ProcessDetails));
* }
* }
* void freeSmeltingData(struct ProcessDetails \*\*\*records, int n) {
* for (int i = 0; i < n; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of smelting processes to analyze: ");
* scanf("%d", &n);
* struct ProcessDetails \*\*processRecords;
* allocateSmeltingData(&processRecords, n);
* float \*energyConsumptionData = (float \*)malloc(n \* sizeof(float));
* union ProcessParameters parameters;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for process %d:\n", i + 1);
* printf("Process ID: ");
* scanf("%d", &processRecords[i]->processID);
* printf("Ore Type: ");
* scanf(" %[^\n]", processRecords[i]->oreType);
* printf("Efficiency (as a percentage): ");
* scanf("%f", &processRecords[i]->efficiency);
* printf("Enter energy consumption for the process (in kWh): ");
* scanf("%f", &energyConsumptionData[i]);
* printf("Enter process parameter type (1 for Energy, 2 for Duration): ");
* int paramType;
* scanf("%d", &paramType);
* if (paramType == 1) {
* printf("Enter energy consumption for the process (in kWh): ");
* scanf("%f", &parameters.energy);
* printf("Energy Consumption: %.2f kWh\n", parameters.energy);
* } else if (paramType == 2) {
* printf("Enter duration for the process (in hours): ");
* scanf("%f", &parameters.duration);
* printf("Process Duration: %.2f hours\n", parameters.duration);
* } else {
* printf("Invalid parameter type.\n");
* }
* }
* printf("\nSmelting Efficiency Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nProcess ID: %d\nOre Type: %s\nEfficiency: %.2f%%\n", processRecords[i]->processID,
* processRecords[i]->oreType, processRecords[i]->efficiency);
* printf("Energy Consumption: %.2f kWh\n", energyConsumptionData[i]);
* }
* free(energyConsumptionData);
* freeSmeltingData(&processRecords, n);
* return 0;
* }

**1. Weld Type Configuration System**

**Description:**  
Design a system to store and manage weld type configurations using structures for weld type details, unions for variable parameters (e.g., voltage or current), and arrays for multiple configurations.

**Specifications:**

* **Structure:** Stores weld type ID, name, voltage, and current.
* **Union:** Represents either voltage or current as a variable parameter.
* **Array:** Holds multiple weld type configurations.
* **const Pointers:** Protect weld type details.
* **Double Pointers:** Manage dynamic allocation of weld configurations.
* #include <stdio.h>
* #include <stdlib.h>
* struct WeldType {
* int weldID;
* char name[30];
* float voltage;
* float current;
* };
* union VariableParameter {
* float voltage;
* float current;
* };
* void allocateWeldConfigurations(struct WeldType \*\*\*configs, int n) {
* \*configs = (struct WeldType \*\*)malloc(n \* sizeof(struct WeldType \*));
* for (int i = 0; i < n; i++) {
* (\*configs)[i] = (struct WeldType \*)malloc(sizeof(struct WeldType));
* }
* }
* void freeWeldConfigurations(struct WeldType \*\*\*configs, int n) {
* for (int i = 0; i < n; i++) {
* free((\*configs)[i]);
* }
* free(\*configs);
* }
* int main() {
* int n;
* printf("Enter the number of weld configurations: ");
* scanf("%d", &n);
* struct WeldType \*\*weldConfigs;
* allocateWeldConfigurations(&weldConfigs, n);
* union VariableParameter param;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for weld configuration %d:\n", i + 1);
* printf("Weld Type ID: ");
* scanf("%d", &weldConfigs[i]->weldID);
* printf("Weld Type Name: ");
* scanf(" %[^\n]", weldConfigs[i]->name);
* printf("Enter Voltage (in volts): ");
* scanf("%f", &weldConfigs[i]->voltage);
* printf("Enter Current (in amps): ");
* scanf("%f", &weldConfigs[i]->current);
* printf("Choose a variable parameter (1 for Voltage, 2 for Current): ");
* int choice;
* scanf("%d", &choice);
* if (choice == 1) {
* printf("Enter Voltage (in volts): ");
* scanf("%f", &param.voltage);
* printf("Variable Parameter (Voltage): %.2f volts\n", param.voltage);
* } else if (choice == 2) {
* printf("Enter Current (in amps): ");
* scanf("%f", &param.current);
* printf("Variable Parameter (Current): %.2f amps\n", param.current);
* } else {
* printf("Invalid choice.\n");
* }
* }
* printf("\nWeld Type Configurations:\n");
* for (int i = 0; i < n; i++) {
* printf("\nWeld Type ID: %d\nName: %s\nVoltage: %.2f volts\nCurrent: %.2f amps\n",
* weldConfigs[i]->weldID, weldConfigs[i]->name, weldConfigs[i]->voltage, weldConfigs[i]->current);
* }
* freeWeldConfigurations(&weldConfigs, n);
* return 0;
* }

**2. Welding Machine Settings Manager**

**Description:**  
Develop a program to manage settings for welding machines, including mode selection, input voltage range, and speed adjustments.

**Specifications:**

* **Structure:** Contains machine ID, mode, speed, and input voltage range.
* **Array:** Stores settings for multiple machines.
* **Strings:** Represent machine modes.
* **const Pointers:** Prevent modifications to critical machine settings.
* **Double Pointers:** Allocate and manage machine setting records dynamically.
* #include <stdio.h>
* #include <stdlib.h>
* struct MachineSettings {
* int machineID;
* char mode[20];
* float speed;
* float inputVoltageRange[2];
* };
* void allocateSettings(struct MachineSettings \*\*\*settings, int count) {
* \*settings = (struct MachineSettings \*\*)malloc(count \* sizeof(struct MachineSettings \*));
* for (int i = 0; i < count; i++) {
* (\*settings)[i] = (struct MachineSettings \*)malloc(sizeof(struct MachineSettings));
* }
* }
* void freeSettings(struct MachineSettings \*\*\*settings, int count) {
* for (int i = 0; i < count; i++) {
* free((\*settings)[i]);
* }
* free(\*settings);
* }
* int main() {
* int n;
* printf("Enter the number of welding machines: ");
* scanf("%d", &n);
* struct MachineSettings \*\*settings;
* allocateSettings(&settings, n);
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for welding machine %d:\n", i + 1);
* printf("Machine ID: ");
* scanf("%d", &settings[i]->machineID);
* printf("Mode (e.g., 'MIG', 'TIG', 'Stick'): ");
* scanf(" %[^\n]", settings[i]->mode);
* printf("Speed (in mm/s): ");
* scanf("%f", &settings[i]->speed);
* printf("Input Voltage Range (Min Max): ");
* scanf("%f %f", &settings[i]->inputVoltageRange[0], &settings[i]->inputVoltageRange[1]);
* }
* printf("\nWelding Machine Settings:\n");
* for (int i = 0; i < n; i++) {
* printf("\nMachine ID: %d\nMode: %s\nSpeed: %.2f mm/s\nInput Voltage Range: %.2f to %.2f volts\n",
* settings[i]->machineID, settings[i]->mode, settings[i]->speed,
* settings[i]->inputVoltageRange[0], settings[i]->inputVoltageRange[1]);
* }
* freeSettings(&settings, n);
* return 0;
* }

**3. Welding Process Tracker**

**Description:**  
Create a system to track ongoing welding processes using structures for process metadata, unions for variable process metrics (e.g., heat input or arc length), and arrays for process data storage.

**Specifications:**

* **Structure:** Stores process ID, material, and welder name.
* **Union:** Represents either heat input or arc length.
* **Array:** Stores process data for multiple welding tasks.
* **const Pointers:** Protect metadata for ongoing processes.
* **Double Pointers:** Manage dynamic process records.
* #include <stdio.h>
* #include <stdlib.h>
* struct ProcessMetadata {
* int processID;
* char material[30];
* char welderName[30];
* };
* union ProcessMetrics {
* float heatInput;
* float arcLength;
* };
* void allocateProcesses(struct ProcessMetadata \*\*\*processes, int count) {
* \*processes = (struct ProcessMetadata \*\*)malloc(count \* sizeof(struct ProcessMetadata \*));
* for (int i = 0; i < count; i++) {
* (\*processes)[i] = (struct ProcessMetadata \*)malloc(sizeof(struct ProcessMetadata));
* }
* }
* void freeProcesses(struct ProcessMetadata \*\*\*processes, int count) {
* for (int i = 0; i < count; i++) {
* free((\*processes)[i]);
* }
* free(\*processes);
* }
* int main() {
* int n;
* printf("Enter the number of welding processes to track: ");
* scanf("%d", &n);
* struct ProcessMetadata \*\*processes;
* allocateProcesses(&processes, n);
* union ProcessMetrics metric;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for welding process %d:\n", i + 1);
* printf("Process ID: ");
* scanf("%d", &processes[i]->processID);
* printf("Material: ");
* scanf(" %[^\n]", processes[i]->material);
* printf("Welder Name: ");
* scanf(" %[^\n]", processes[i]->welderName);
* printf("Choose a metric (1 for Heat Input, 2 for Arc Length): ");
* int choice;
* scanf("%d", &choice);
* if (choice == 1) {
* printf("Enter Heat Input (in kJ/mm): ");
* scanf("%f", &metric.heatInput);
* printf("Heat Input Recorded: %.2f kJ/mm\n", metric.heatInput);
* } else if (choice == 2) {
* printf("Enter Arc Length (in mm): ");
* scanf("%f", &metric.arcLength);
* printf("Arc Length Recorded: %.2f mm\n", metric.arcLength);
* } else {
* printf("Invalid choice.\n");
* }
* }
* printf("\nWelding Process Data:\n");
* for (int i = 0; i < n; i++) {
* printf("\nProcess ID: %d\nMaterial: %s\nWelder Name: %s\n",
* processes[i]->processID, processes[i]->material, processes[i]->welderName);
* }
* freeProcesses(&processes, n);
* return 0;
* }

**4. Weld Bead Geometry Analyzer**

**Description:**  
Design a program to analyze weld bead geometry using structures for geometry details, arrays for measurements, and unions for different parameters like width, depth, and height.

**Specifications:**

* **Structure:** Contains bead ID, material, and geometry type.
* **Union:** Represents bead width, depth, or height.
* **Array:** Stores geometry measurements.
* **const Pointers:** Protect geometry data.
* **Double Pointers:** Allocate and manage bead records dynamically.
* #include <stdio.h>
* #include <stdlib.h>
* struct BeadGeometry {
* int beadID;
* char material[30];
* char geometryType[20];
* };
* union GeometryParameters {
* float width;
* float depth;
* float height;
* };
* void allocateBeadRecords(struct BeadGeometry \*\*\*records, int count) {
* \*records = (struct BeadGeometry \*\*)malloc(count \* sizeof(struct BeadGeometry \*));
* for (int i = 0; i < count; i++) {
* (\*records)[i] = (struct BeadGeometry \*)malloc(sizeof(struct BeadGeometry));
* }
* }
* void freeBeadRecords(struct BeadGeometry \*\*\*records, int count) {
* for (int i = 0; i < count; i++) {
* free((\*records)[i]);
* }
* free(\*records);
* }
* int main() {
* int n;
* printf("Enter the number of weld beads to analyze: ");
* scanf("%d", &n);
* struct BeadGeometry \*\*records;
* allocateBeadRecords(&records, n);
* union GeometryParameters parameter;
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for weld bead %d:\n", i + 1);
* printf("Bead ID: ");
* scanf("%d", &records[i]->beadID);
* printf("Material: ");
* scanf(" %[^\n]", records[i]->material);
* printf("Geometry Type: ");
* scanf(" %[^\n]", records[i]->geometryType);
* printf("Choose a parameter to enter (1 for Width, 2 for Depth, 3 for Height): ");
* int choice;
* scanf("%d", &choice);
* if (choice == 1) {
* printf("Enter Width (in mm): ");
* scanf("%f", &parameter.width);
* printf("Width Recorded: %.2f mm\n", parameter.width);
* } else if (choice == 2) {
* printf("Enter Depth (in mm): ");
* scanf("%f", &parameter.depth);
* printf("Depth Recorded: %.2f mm\n", parameter.depth);
* } else if (choice == 3) {
* printf("Enter Height (in mm): ");
* scanf("%f", &parameter.height);
* printf("Height Recorded: %.2f mm\n", parameter.height);
* } else {
* printf("Invalid choice.\n");
* }
* }
* printf("\nWeld Bead Geometry Data:\n");
* for (int i = 0; i < n; i++) {
* printf("\nBead ID: %d\nMaterial: %s\nGeometry Type: %s\n",
* records[i]->beadID, records[i]->material, records[i]->geometryType);
* }
* freeBeadRecords(&records, n);
* return 0;
* }

**5. Welding Consumable Inventory System**

**Description:**  
Develop a system to manage inventory for welding consumables, including electrodes, filler materials, and fluxes.

**Specifications:**

* **Structure:** Stores consumable ID, type, and quantity.
* **Array:** Inventory for different consumables.
* **Strings:** Represent consumable types.
* **const Pointers:** Prevent modifications to consumable details.
* **Double Pointers:** Manage inventory records dynamically.
* #include <stdio.h>
* #include <stdlib.h>
* struct Consumable {
* const int consumableID;
* char type[30];
* int quantity;
* };
* void allocateInventory(struct Consumable \*\*\*inventory, int count) {
* \*inventory = (struct Consumable \*\*)malloc(count \* sizeof(struct Consumable \*));
* for (int i = 0; i < count; i++) {
* (\*inventory)[i] = (struct Consumable \*)malloc(sizeof(struct Consumable));
* }
* }
* void freeInventory(struct Consumable \*\*\*inventory, int count) {
* for (int i = 0; i < count; i++) {
* free((\*inventory)[i]);
* }
* free(\*inventory);
* }
* int main() {
* int n;
* printf("Enter the number of consumables to track: ");
* scanf("%d", &n);
* struct Consumable \*\*inventory;
* allocateInventory(&inventory, n);
* for (int i = 0; i < n; i++) {
* int id;
* printf("\nEnter details for consumable %d:\n", i + 1);
* printf("Consumable ID: ");
* scanf("%d", &id);
* inventory[i]->consumableID = id; // ID is constant and cannot be modified later.
* printf("Type: ");
* scanf(" %[^\n]", inventory[i]->type);
* printf("Quantity: ");
* scanf("%d", &inventory[i]->quantity);
* }
* printf("\nWelding Consumable Inventory:\n");
* for (int i = 0; i < n; i++) {
* printf("\nConsumable ID: %d\nType: %s\nQuantity: %d\n",
* inventory[i]->consumableID, inventory[i]->type, inventory[i]->quantity);
* }
* freeInventory(&inventory, n);
* return 0;
* }

**6. Welding Safety Equipment Tracker**

**Description:**  
Create a program to track safety equipment for welding personnel using structures for equipment details, arrays for availability status, and strings for equipment names.

**Specifications:**

* **Structure:** Holds equipment ID, type, and usage frequency.
* **Array:** Availability status for multiple equipment items.
* **Strings:** Equipment names.
* **const Pointers:** Protect safety equipment data.
* **Double Pointers:** Allocate dynamic safety equipment records.
* #include <stdio.h>
* #include <stdlib.h>
* struct Equipment {
* const int equipmentID;
* char type[30];
* int usageFrequency;
* int isAvailable; // 1 for available, 0 for not available
* };
* void allocateEquipment(struct Equipment \*\*\*equipmentList, int count) {
* \*equipmentList = (struct Equipment \*\*)malloc(count \* sizeof(struct Equipment \*));
* for (int i = 0; i < count; i++) {
* (\*equipmentList)[i] = (struct Equipment \*)malloc(sizeof(struct Equipment));
* }
* }
* void freeEquipment(struct Equipment \*\*\*equipmentList, int count) {
* for (int i = 0; i < count; i++) {
* free((\*equipmentList)[i]);
* }
* free(\*equipmentList);
* }
* int main() {
* int n;
* printf("Enter the number of safety equipment items to track: ");
* scanf("%d", &n);
* struct Equipment \*\*equipmentList;
* allocateEquipment(&equipmentList, n);
* for (int i = 0; i < n; i++) {
* int id;
* printf("\nEnter details for equipment %d:\n", i + 1);
* printf("Equipment ID: ");
* scanf("%d", &id);
* equipmentList[i]->equipmentID = id; // ID is constant and cannot be modified later.
* printf("Type: ");
* scanf(" %[^\n]", equipmentList[i]->type);
* printf("Usage Frequency (per week): ");
* scanf("%d", &equipmentList[i]->usageFrequency);
* printf("Is Available? (1 for Yes, 0 for No): ");
* scanf("%d", &equipmentList[i]->isAvailable);
* }
* printf("\nSafety Equipment Tracker:\n");
* for (int i = 0; i < n; i++) {
* printf("\nEquipment ID: %d\nType: %s\nUsage Frequency: %d per week\nAvailability: %s\n",
* equipmentList[i]->equipmentID, equipmentList[i]->type,
* equipmentList[i]->usageFrequency,
* equipmentList[i]->isAvailable ? "Available" : "Not Available");
* }
* freeEquipment(&equipmentList, n);
* return 0;
* }

**7. Welding Defect Classification System**

**Description:**  
Design a system to classify welding defects using structures for defect data, arrays for sample analysis, and unions for defect types like porosity, cracking, or spatter.

**Specifications:**

* **Structure:** Stores defect ID, type, and severity level.
* **Union:** Represents defect types.
* **Array:** Sample analysis data.
* **const Pointers:** Protect defect classifications.
* **Double Pointers:** Manage defect data dynamically.
* #include <stdio.h>
* #include <stdlib.h>
* #include <string.h>
* // Union to represent defect types
* union DefectType {
* char porosity[30];
* char cracking[30];
* char spatter[30];
* };
* // Structure to store defect data
* struct Defect {
* const int defectID;  // Constant to protect defect classifications
* char type[30];
* char severityLevel[10];
* union DefectType defectType;
* };
* // Function to create a defect instance
* struct Defect \*createDefect(int id, const char \*type, const char \*severityLevel) {
* struct Defect \*newDefect = (struct Defect \*)malloc(sizeof(struct Defect));
* \*(int \*)&newDefect->defectID = id;  // Bypass const for initialization
* strncpy(newDefect->type, type, sizeof(newDefect->type) - 1);
* newDefect->type[sizeof(newDefect->type) - 1] = '\0';
* strncpy(newDefect->severityLevel, severityLevel, sizeof(newDefect->severityLevel) - 1);
* newDefect->severityLevel[sizeof(newDefect->severityLevel) - 1] = '\0';
* return newDefect;
* }
* int main() {
* int n;
* printf("Enter the number of welding defect samples to analyze: ");
* scanf("%d", &n);
* struct Defect \*\*defects = (struct Defect \*\*)malloc(n \* sizeof(struct Defect \*));
* for (int i = 0; i < n; i++) {
* int id;
* char type[30], severityLevel[10];
* printf("\nEnter details for defect %d:\n", i + 1);
* printf("Defect ID: ");
* scanf("%d", &id);
* printf("Defect Type (Porosity/Cracking/Spatter): ");
* scanf(" %[^\n]", type);
* printf("Severity Level (Low/Medium/High): ");
* scanf(" %[^\n]", severityLevel);
* defects[i] = createDefect(id, type, severityLevel);
* if (strcmp(type, "Porosity") == 0) {
* printf("Enter porosity details: ");
* scanf(" %[^\n]", defects[i]->defectType.porosity);
* } else if (strcmp(type, "Cracking") == 0) {
* printf("Enter cracking details: ");
* scanf(" %[^\n]", defects[i]->defectType.cracking);
* } else if (strcmp(type, "Spatter") == 0) {
* printf("Enter spatter details: ");
* scanf(" %[^\n]", defects[i]->defectType.spatter);
* } else {
* printf("Unknown defect type.\n");
* }
* }
* printf("\nWelding Defect Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nDefect ID: %d\nType: %s\nSeverity Level: %s\n",
* defects[i]->defectID, defects[i]->type, defects[i]->severityLevel);
* if (strcmp(defects[i]->type, "Porosity") == 0) {
* printf("Porosity Details: %s\n", defects[i]->defectType.porosity);
* } else if (strcmp(defects[i]->type, "Cracking") == 0) {
* printf("Cracking Details: %s\n", defects[i]->defectType.cracking);
* } else if (strcmp(defects[i]->type, "Spatter") == 0) {
* printf("Spatter Details: %s\n", defects[i]->defectType.spatter);
* }
* }
* for (int i = 0; i < n; i++) {
* free(defects[i]);
* }
* free(defects);
* return 0;
* }

**8. Arc Welding Performance Analyzer**

**Description:**  
Develop a program to analyze the performance of arc welding processes using structures for performance metrics, arrays for output data, and unions for variable factors like arc stability and penetration depth.

**Specifications:**

* **Structure:** Contains performance ID, material type, and current setting.
* **Union:** Represents arc stability or penetration depth.
* **Array:** Output data.
* **const Pointers:** Protect performance configurations.
* **Double Pointers:** Manage dynamic performance data.
* #include <stdio.h>
* #include <stdlib.h>
* // Union to represent variable factors
* union PerformanceFactors {
* float arcStability;
* float penetrationDepth;
* };
* // Structure to store performance metrics
* struct Performance {
* const int performanceID;  // Constant to protect performance configurations
* char materialType[30];
* float currentSetting;
* union PerformanceFactors factors;
* int factorType; // 1 for Arc Stability, 2 for Penetration Depth
* };
* // Function to create a performance record
* struct Performance \*createPerformance(int id, const char \*material, float current) {
* struct Performance \*newPerformance = (struct Performance \*)malloc(sizeof(struct Performance));
* \*(int \*)&newPerformance->performanceID = id;  // Bypass const for initialization
* // Copy material name without string functions
* int i = 0;
* while (material[i] != '\0' && i < sizeof(newPerformance->materialType) - 1) {
* newPerformance->materialType[i] = material[i];
* i++;
* }
* newPerformance->materialType[i] = '\0';
* newPerformance->currentSetting = current;
* newPerformance->factorType = 0; // Default unset
* return newPerformance;
* }
* int main() {
* int n;
* printf("Enter the number of performance records to analyze: ");
* scanf("%d", &n);
* struct Performance \*\*records = (struct Performance \*\*)malloc(n \* sizeof(struct Performance \*));
* for (int i = 0; i < n; i++) {
* int id;
* char material[30];
* float current;
* int factorChoice;
* printf("\nEnter details for performance record %d:\n", i + 1);
* printf("Performance ID: ");
* scanf("%d", &id);
* printf("Material Type: ");
* scanf(" %[^\n]", material);
* printf("Current Setting (in Amps): ");
* scanf("%f", &current);
* records[i] = createPerformance(id, material, current);
* printf("Enter factor type (1 for Arc Stability, 2 for Penetration Depth): ");
* scanf("%d", &factorChoice);
* if (factorChoice == 1) {
* printf("Enter arc stability (0.0 - 1.0): ");
* scanf("%f", &records[i]->factors.arcStability);
* records[i]->factorType = 1;
* } else if (factorChoice == 2) {
* printf("Enter penetration depth (in mm): ");
* scanf("%f", &records[i]->factors.penetrationDepth);
* records[i]->factorType = 2;
* } else {
* printf("Invalid choice. No factor data recorded.\n");
* }
* }
* printf("\nArc Welding Performance Analysis Results:\n");
* for (int i = 0; i < n; i++) {
* printf("\nPerformance ID: %d\nMaterial Type: %s\nCurrent Setting: %.2f Amps\n",
* records[i]->performanceID, records[i]->materialType, records[i]->currentSetting);
* if (records[i]->factorType == 1) {
* printf("Arc Stability: %.2f\n", records[i]->factors.arcStability);
* } else if (records[i]->factorType == 2) {
* printf("Penetration Depth: %.2f mm\n", records[i]->factors.penetrationDepth);
* } else {
* printf("No factor data recorded.\n");
* }
* }
* for (int i = 0; i < n; i++) {
* free(records[i]);
* }
* free(records);
* return 0;
* }

**9. Welding Schedule Optimization Tool**

**Description:**  
Create a program to optimize welding schedules using structures for task details, arrays for time slots, and strings for task names.

**Specifications:**

* **Structure:** Holds task ID, priority, and duration.
* **Array:** Time slots for scheduling.
* **Strings:** Task names.
* **const Pointers:** Protect task details.
* **Double Pointers:** Allocate and manage task records dynamically.
* #include <stdio.h>
* #include <stdlib.h>
* #define MAX\_TASK\_NAME\_LENGTH 50
* // Structure to hold task details
* struct Task {
* const int taskID;  // Constant to protect task details
* char taskName[MAX\_TASK\_NAME\_LENGTH];
* int priority;
* float duration;
* };
* // Function to create a task record
* struct Task\* createTask(int id, const char\* name, int priority, float duration) {
* struct Task \*newTask = (struct Task \*)malloc(sizeof(struct Task));
* \*(int \*)&newTask->taskID = id;  // Bypass const for initialization
* // Copy task name manually (avoiding strcpy)
* int i = 0;
* while (name[i] != '\0' && i < MAX\_TASK\_NAME\_LENGTH - 1) {
* newTask->taskName[i] = name[i];
* i++;
* }
* newTask->taskName[i] = '\0';
* newTask->priority = priority;
* newTask->duration = duration;
* return newTask;
* }
* int main() {
* int n;
* printf("Enter the number of welding tasks to schedule: ");
* scanf("%d", &n);
* struct Task \*\*tasks = (struct Task \*\*)malloc(n \* sizeof(struct Task \*));
* for (int i = 0; i < n; i++) {
* int id, priority;
* char name[MAX\_TASK\_NAME\_LENGTH];
* float duration;
* printf("\nEnter details for task %d:\n", i + 1);
* printf("Task ID: ");
* scanf("%d", &id);
* printf("Task Name: ");
* scanf(" %[^\n]", name);
* printf("Task Priority (higher number = higher priority): ");
* scanf("%d", &priority);
* printf("Task Duration (in hours): ");
* scanf("%f", &duration);
* tasks[i] = createTask(id, name, priority, duration);
* }
* // Sort tasks by priority (simple selection sort for demonstration)
* for (int i = 0; i < n - 1; i++) {
* for (int j = i + 1; j < n; j++) {
* if (tasks[i]->priority < tasks[j]->priority) {
* struct Task \*temp = tasks[i];
* tasks[i] = tasks[j];
* tasks[j] = temp;
* }
* }
* }
* // Output the optimized welding schedule
* printf("\nOptimized Welding Schedule:\n");
* for (int i = 0; i < n; i++) {
* printf("\nTask ID: %d\n", tasks[i]->taskID);
* printf("Task Name: %s\n", tasks[i]->taskName);
* printf("Priority: %d\n", tasks[i]->priority);
* printf("Duration: %.2f hours\n", tasks[i]->duration);
* }
* // Free allocated memory
* for (int i = 0; i < n; i++) {
* free(tasks[i]);
* }
* free(tasks);
* return 0;
* }

**10. Automated Weld Inspection System**

**Description:**  
Develop a system to automate the inspection of welds using structures for inspection details, arrays for measurement data, and unions for different defect parameters.

**Specifications:**

* **Structure:** Stores inspection ID, method, and results.
* **Union:** Represents defect parameters like size or location.
* **Array:** Measurement data.
* **const Pointers:** Protect inspection configurations.
* **Double Pointers:** Manage inspection records dynamically.
* #include <stdio.h>
* #define MAX\_DEFECT\_DESC\_LENGTH 50
* // Structure to store inspection details
* struct Inspection {
* int inspectionID;
* char method[MAX\_DEFECT\_DESC\_LENGTH];
* char result[MAX\_DEFECT\_DESC\_LENGTH];
* };
* // Union for defect parameters like size or location
* union Defect {
* float size;
* char location[MAX\_DEFECT\_DESC\_LENGTH];
* };
* int main() {
* int n;
* printf("Enter the number of weld inspections: ");
* scanf("%d", &n);
* struct Inspection inspections[n];
* union Defect defects[n];
* for (int i = 0; i < n; i++) {
* int id;
* char method[MAX\_DEFECT\_DESC\_LENGTH];
* char result[MAX\_DEFECT\_DESC\_LENGTH];
* float size;
* char location[MAX\_DEFECT\_DESC\_LENGTH];
* printf("\nEnter details for inspection %d:\n", i + 1);
* printf("Inspection ID: ");
* scanf("%d", &id);
* printf("Inspection Method: ");
* scanf(" %[^\n]", method);
* printf("Inspection Result: ");
* scanf(" %[^\n]", result);
* inspections[i].inspectionID = id;
* snprintf(inspections[i].method, sizeof(inspections[i].method), "%s", method);
* snprintf(inspections[i].result, sizeof(inspections[i].result), "%s", result);
* printf("Defect Size (enter 0 if none): ");
* scanf("%f", &size);
* if (size != 0) {
* defects[i].size = size;
* printf("Defect Size: %.2f\n", defects[i].size);
* } else {
* printf("Enter Defect Location (if any): ");
* scanf(" %[^\n]", location);
* snprintf(defects[i].location, sizeof(defects[i].location), "%s", location);
* printf("Defect Location: %s\n", defects[i].location);
* }
* }
* // Output the inspection results
* printf("\nInspection Summary:\n");
* for (int i = 0; i < n; i++) {
* printf("\nInspection ID: %d\n", inspections[i].inspectionID);
* printf("Method: %s\n", inspections[i].method);
* printf("Result: %s\n", inspections[i].result);
* if (defects[i].size != 0) {
* printf("Defect Size: %.2f\n", defects[i].size);
* } else {
* printf("Defect Location: %s\n", defects[i].location);
* }
* }
* return 0;
* }

**11. Welding Robot Control System**

**Description:**  
Design a control system for welding robots using structures for robot configurations, arrays for motion data, and strings for robot types.

**Specifications:**

* **Structure:** Holds robot ID, configuration, and status.
* **Array:** Motion data for robotic operations.
* **Strings:** Robot types.
* **const Pointers:** Protect robot configurations.
* **Double Pointers:** Allocate and manage robot records dynamically.
* #include <stdio.h>
* #include <string.h>
* #define MAX\_ROBOTS 5
* #define MAX\_STRING\_LENGTH 20
* // Structure to store robot configurations and status
* struct Robot {
* int robotID;
* char configuration[MAX\_STRING\_LENGTH];
* char status[MAX\_STRING\_LENGTH];
* };
* // Function to display robot details
* void displayRobotDetails(struct Robot\* robot) {
* printf("\nRobot ID: %d\n", robot->robotID);
* printf("Configuration: %s\n", robot->configuration);
* printf("Status: %s\n", robot->status);
* }
* int main() {
* int n;
* // Input the number of robots
* printf("Enter the number of robots: ");
* scanf("%d", &n);
* // Protect robot configurations with const pointers
* const struct Robot \*const robotPtr[n];
* // Declare an array to store robot configurations
* struct Robot robots[MAX\_ROBOTS];
* // Dynamically allocate robot records using double pointers
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for robot %d:\n", i + 1);
* // Get robot details
* printf("Enter Robot ID: ");
* scanf("%d", &robots[i].robotID);
* printf("Enter Configuration (e.g., arm type, range): ");
* scanf(" %[^\n]", robots[i].configuration);  // Allow spaces in the configuration
* printf("Enter Status (e.g., Active, Idle): ");
* scanf(" %[^\n]", robots[i].status);  // Allow spaces in the status
* // Assign the address of robots[i] to robotPtr
* robotPtr[i] = &robots[i];
* }
* // Display robot details
* for (int i = 0; i < n; i++) {
* displayRobotDetails((struct Robot \*)robotPtr[i]);
* }
* return 0;
* }

**12. Weld Quality Data Logger**

**Description:**  
Create a data logger for weld quality metrics using structures for weld details, arrays for quality data, and unions for different quality parameters.

**Specifications:**

* **Structure:** Stores weld ID, material, and quality score.
* **Union:** Represents different quality parameters.
* **Array:** Quality data for multiple welds.
* **const Pointers:** Protect weld details.
* **Double Pointers:** Manage dynamic quality data.
* #include <stdio.h>
* #include <string.h>
* #define MAX\_WELDS 5
* #define MAX\_STRING\_LENGTH 20
* // Union to represent different quality parameters (e.g., tensile strength, hardness)
* union QualityParams {
* float tensileStrength;
* float hardness;
* float elongation;
* };
* // Structure to store weld details
* struct Weld {
* int weldID;
* char material[MAX\_STRING\_LENGTH];
* float qualityScore;
* union QualityParams qualityParam;  // Union for different quality parameters
* };
* // Function to display weld details
* void displayWeldDetails(const struct Weld\* weld) {
* printf("\nWeld ID: %d\n", weld->weldID);
* printf("Material: %s\n", weld->material);
* printf("Quality Score: %.2f\n", weld->qualityScore);
* printf("Tensile Strength: %.2f\n", weld->qualityParam.tensileStrength);  // Example parameter
* }
* int main() {
* int n;
* // Input the number of welds
* printf("Enter the number of welds: ");
* scanf("%d", &n);
* // Declare an array to store weld details
* struct Weld welds[MAX\_WELDS];
* // Declare a pointer array to store pointers to welds
* struct Weld \*weldPtr[MAX\_WELDS];
* // Dynamically allocate and store weld data using double pointers
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for weld %d:\n", i + 1);
* // Get weld details
* printf("Enter Weld ID: ");
* scanf("%d", &welds[i].weldID);
* printf("Enter Material Type (e.g., Steel, Aluminum): ");
* scanf(" %[^\n]", welds[i].material);  // Allow spaces in the material type
* printf("Enter Quality Score: ");
* scanf("%f", &welds[i].qualityScore);
* // Get the specific quality parameter (for example, tensile strength)
* printf("Enter Tensile Strength: ");
* scanf("%f", &welds[i].qualityParam.tensileStrength);  // Store in the union
* // Assign the address of welds[i] to weldPtr
* weldPtr[i] = &welds[i];
* }
* // Display weld details
* for (int i = 0; i < n; i++) {
* displayWeldDetails(weldPtr[i]);
* }
* return 0;
* }

**13. Thermal Input Analysis Tool**

**Description:**  
Develop a program to analyze thermal input in welding using structures for thermal details, arrays for time-temperature data, and unions for heat input variables.

**Specifications:**

* **Structure:** Holds thermal input ID, current, and voltage.
* **Union:** Represents heat input or time-temperature correlation.
* **Array:** Time-temperature data.
* **const Pointers:** Protect thermal input data.
* **Double Pointers:** Manage thermal data dynamically.
* #include <stdio.h>
* #define MAX\_DATA 5
* // Union to represent heat input or time-temperature correlation
* union HeatInput {
* float heatInput;          // Heat input
* float timeTemperature[2]; // Time-temperature correlation (time, temperature)
* };
* // Structure to hold thermal input details
* struct ThermalInput {
* int thermalID;
* float current;
* float voltage;
* union HeatInput heatInputData;
* };
* // Function to display thermal input details
* void displayThermalInput(const struct ThermalInput\* thermalInput) {
* printf("\nThermal Input ID: %d\n", thermalInput->thermalID);
* printf("Current: %.2f A\n", thermalInput->current);
* printf("Voltage: %.2f V\n", thermalInput->voltage);
* printf("Heat Input: %.2f J\n", thermalInput->heatInputData.heatInput);  // Example heat input
* printf("Time-Temperature Data: Time = %.2f, Temperature = %.2f\n",
* thermalInput->heatInputData.timeTemperature[0], thermalInput->heatInputData.timeTemperature[1]); // Example time-temp
* }
* int main() {
* int n;
* // Input number of thermal input data
* printf("Enter the number of thermal input records: ");
* scanf("%d", &n);
* // Declare array to store thermal input data
* struct ThermalInput thermalInputs[MAX\_DATA];
* // Declare a pointer array to store pointers to thermal input structures
* const struct ThermalInput\* thermalPtr[MAX\_DATA];
* // Input thermal data and populate the array
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for thermal input %d:\n", i + 1);
* // Get thermal input details
* printf("Enter Thermal Input ID: ");
* scanf("%d", &thermalInputs[i].thermalID);
* printf("Enter Current (A): ");
* scanf("%f", &thermalInputs[i].current);
* printf("Enter Voltage (V): ");
* scanf("%f", &thermalInputs[i].voltage);
* // Get heat input or time-temperature data
* printf("Enter Heat Input (in joules): ");
* scanf("%f", &thermalInputs[i].heatInputData.heatInput);  // Store in the union
* printf("Enter Time (s): ");
* scanf("%f", &thermalInputs[i].heatInputData.timeTemperature[0]);  // Store time in the union
* printf("Enter Temperature (°C): ");
* scanf("%f", &thermalInputs[i].heatInputData.timeTemperature[1]);  // Store temperature in the union
* // Assign the address of thermalInputs[i] to thermalPtr
* thermalPtr[i] = &thermalInputs[i];
* }
* // Display thermal input details
* for (int i = 0; i < n; i++) {
* displayThermalInput(thermalPtr[i]);
* }
* return 0;
* }

**14. Welding Procedure Specification Manager**

**Description:**  
Create a program to manage welding procedure specifications using structures for procedure details, arrays for parameters, and strings for procedure names.

**Specifications:**

* **Structure:** Contains procedure ID, material, and joint type.
* **Array:** Welding parameters.
* **Strings:** Procedure names.
* **const Pointers:** Protect procedure details.
* **Double Pointers:** Allocate dynamic procedure records.
* #include <stdio.h>
* #include <string.h>
* #define MAX\_PROCEDURES 5
* #define MAX\_NAME\_LENGTH 50
* // Structure to hold welding procedure details
* struct WeldingProcedure {
* int procedureID;
* char material[MAX\_NAME\_LENGTH];
* char jointType[MAX\_NAME\_LENGTH];
* float weldingParameters[3]; // Assume three welding parameters for simplicity (e.g., voltage, speed, and current)
* };
* // Function to display welding procedure details
* void displayWeldingProcedure(const struct WeldingProcedure\* procedure) {
* printf("\nWelding Procedure ID: %d\n", procedure->procedureID);
* printf("Material: %s\n", procedure->material);
* printf("Joint Type: %s\n", procedure->jointType);
* printf("Welding Parameters (Voltage, Speed, Current): %.2f, %.2f, %.2f\n",
* procedure->weldingParameters[0], procedure->weldingParameters[1], procedure->weldingParameters[2]);
* }
* int main() {
* int n;
* // Input number of welding procedures
* printf("Enter the number of welding procedures: ");
* scanf("%d", &n);
* // Declare an array to store welding procedure data
* struct WeldingProcedure procedures[MAX\_PROCEDURES];
* // Declare a pointer array to store pointers to welding procedure structures
* const struct WeldingProcedure\* procedurePtr[MAX\_PROCEDURES];
* // Input welding procedure data and populate the array
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for welding procedure %d:\n", i + 1);
* // Get welding procedure details
* printf("Enter Procedure ID: ");
* scanf("%d", &procedures[i].procedureID);
* printf("Enter Material: ");
* scanf("%s", procedures[i].material);  // Get material name
* printf("Enter Joint Type: ");
* scanf("%s", procedures[i].jointType);  // Get joint type
* // Get welding parameters (e.g., voltage, speed, current)
* printf("Enter Welding Parameters (Voltage, Speed, Current): ");
* scanf("%f %f %f", &procedures[i].weldingParameters[0],
* &procedures[i].weldingParameters[1],
* &procedures[i].weldingParameters[2]);
* // Assign the address of procedures[i] to procedurePtr
* procedurePtr[i] = &procedures[i];
* }
* // Display welding procedure details
* for (int i = 0; i < n; i++) {
* displayWeldingProcedure(procedurePtr[i]);
* }
* return 0;
* }

**15. Joint Design Data Tracker**

**Description:**  
Design a tracker for joint designs in welding using structures for joint details, arrays for dimensions, and unions for variable joint parameters.

**Specifications:**

* **Structure:** Stores joint ID, type, and angle.
* **Union:** Represents joint parameters.
* **Array:** Dimensions for multiple joints.
* **const Pointers:** Protect joint data.
* **Double Pointers:** Manage joint records dynamically.
* #include <stdio.h>
* #define MAX\_JOINTS 5
* // Union to represent variable joint parameters (e.g., angle or width)
* union JointParameters {
* float angle;
* float width;
* };
* // Structure to store joint details (ID, type, and associated parameters)
* struct JointDesign {
* int jointID;
* char jointType[50];
* union JointParameters parameters;
* float dimensions[3];
* };
* // Function to display joint design details
* void displayJointDesign(const struct JointDesign\* joint) {
* printf("\nJoint Design ID: %d\n", joint->jointID);
* printf("Joint Type: %s\n", joint->jointType);
* // Check the type of parameter and display it
* if (joint->jointType[0] == 'A') {
* printf("Angle: %.2f degrees\n", joint->parameters.angle);
* } else {
* printf("Width: %.2f units\n", joint->parameters.width);
* }
* printf("Dimensions (Length, Width, Thickness): %.2f, %.2f, %.2f\n",
* joint->dimensions[0], joint->dimensions[1], joint->dimensions[2]);
* }
* int main() {
* int n;
* // Input number of joints
* printf("Enter the number of joint designs: ");
* scanf("%d", &n);
* // Declare an array to store joint designs
* struct JointDesign joints[MAX\_JOINTS];
* // Declare a pointer array to store pointers to joint designs
* const struct JointDesign\* jointPtr[MAX\_JOINTS];
* // Input joint design details and populate the array
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for joint design %d:\n", i + 1);
* // Get joint details
* printf("Enter Joint ID: ");
* scanf("%d", &joints[i].jointID);
* printf("Enter Joint Type (e.g., Angle Joint, Butt Joint): ");
* scanf("%s", joints[i].jointType);
* // Set the union parameter based on joint type
* if (joints[i].jointType[0] == 'A') {
* printf("Enter Angle for Joint (degrees): ");
* scanf("%f", &joints[i].parameters.angle);
* } else {
* printf("Enter Width for Joint (units): ");
* scanf("%f", &joints[i].parameters.width);
* }
* // Get dimensions (length, width, thickness)
* printf("Enter Dimensions (Length, Width, Thickness): ");
* scanf("%f %f %f", &joints[i].dimensions[0],
* &joints[i].dimensions[1],
* &joints[i].dimensions[2]);
* // Assign the address of joints[i] to jointPtr
* jointPtr[i] = &joints[i];
* }
* // Display joint design details
* for (int i = 0; i < n; i++) {
* displayJointDesign(jointPtr[i]);
* }
* return 0;
* }

**16. Filler Metal Selector Tool**

**Description:**  
Develop a program to select filler metals using structures for metal properties, arrays for test results, and strings for metal names.

**Specifications:**

* **Structure:** Holds filler metal ID, composition, and diameter.
* **Array:** Test results for filler metals.
* **Strings:** Filler metal names.
* **const Pointers:** Protect filler metal data.
* **Double Pointers:** Allocate and manage filler metal records.
* #include <stdio.h>
* #include <string.h>
* #define MAX\_METALS 5
* struct FillerMetal {
* int metalID;
* char name[50];
* char composition[100];
* float diameter;
* };
* void displayFillerMetal(const struct FillerMetal\* metal) {
* printf("\nFiller Metal ID: %d\n", metal->metalID);
* printf("Metal Name: %s\n", metal->name);
* printf("Composition: %s\n", metal->composition);
* printf("Diameter: %.2f mm\n", metal->diameter);
* }
* int main() {
* int n;
* printf("Enter the number of filler metals: ");
* scanf("%d", &n);
* struct FillerMetal metals[MAX\_METALS];
* const struct FillerMetal\* metalPtr[MAX\_METALS];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for filler metal %d:\n", i + 1);
* printf("Enter Metal ID: ");
* scanf("%d", &metals[i].metalID);
* printf("Enter Metal Name: ");
* scanf(" %[^\n]%\*c", metals[i].name);
* printf("Enter Metal Composition: ");
* scanf(" %[^\n]%\*c", metals[i].composition);
* printf("Enter Metal Diameter (in mm): ");
* scanf("%f", &metals[i].diameter);
* metalPtr[i] = &metals[i];
* }
* printf("\n--- Filler Metal Details ---\n");
* for (int i = 0; i < n; i++) {
* displayFillerMetal(metalPtr[i]);
* }
* return 0;
* }

**17. Welding Power Source Configuration**

**Description:**  
Create a system to configure welding power sources using structures for source details, arrays for power settings, and strings for source types.

**Specifications:**

* **Structure:** Contains source ID, type, and capacity.
* **Array:** Power settings for multiple sources.
* **Strings:** Source types.
* **const Pointers:** Protect power source configurations.
* **Double Pointers:** Allocate and manage source records.
* #include <stdio.h>
* #define MAX\_SOURCES 5
* struct PowerSource {
* int sourceID;
* char type[50];
* float capacity;
* };
* void displayPowerSource(const struct PowerSource\* source) {
* printf("\nPower Source ID: %d\n", source->sourceID);
* printf("Type: %s\n", source->type);
* printf("Capacity: %.2f kVA\n", source->capacity);
* }
* int main() {
* int n;
* printf("Enter the number of power sources: ");
* scanf("%d", &n);
* struct PowerSource sources[MAX\_SOURCES];
* const struct PowerSource\* sourcePtr[MAX\_SOURCES];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for power source %d:\n", i + 1);
* printf("Enter Source ID: ");
* scanf("%d", &sources[i].sourceID);
* printf("Enter Source Type: ");
* scanf(" %[^\n]%\*c", sources[i].type);
* printf("Enter Source Capacity (in kVA): ");
* scanf("%f", &sources[i].capacity);
* sourcePtr[i] = &sources[i];
* }
* printf("\n--- Power Source Details ---\n");
* for (int i = 0; i < n; i++) {
* displayPowerSource(sourcePtr[i]);
* }
* return 0;
* }

**18. Welding Skill Assessment System**

**Description:**  
Develop a program to assess the skills of welders using structures for skill data, arrays for test results, and strings for skill levels.

**Specifications:**

* **Structure:** Holds welder ID, name, and skill score.
* **Array:** Test results for skill assessment.
* **Strings:** Skill levels.
* **const Pointers:** Protect skill assessment data.
* **Double Pointers:** Manage skill records dynamically.
* #include <stdio.h>
* #define MAX\_WELDERS 5
* struct Welder {
* int welderID;
* char name[50];
* float skillScore;
* };
* void displayWelderSkill(const struct Welder\* welder) {
* printf("\nWelder ID: %d\n", welder->welderID);
* printf("Name: %s\n", welder->name);
* printf("Skill Score: %.2f\n", welder->skillScore);
* }
* int main() {
* int n;
* printf("Enter the number of welders: ");
* scanf("%d", &n);
* struct Welder welders[MAX\_WELDERS];
* const struct Welder\* welderPtr[MAX\_WELDERS];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for welder %d:\n", i + 1);
* printf("Enter Welder ID: ");
* scanf("%d", &welders[i].welderID);
* printf("Enter Welder Name: ");
* scanf(" %[^\n]%\*c", welders[i].name);
* printf("Enter Skill Score (0-100): ");
* scanf("%f", &welders[i].skillScore);
* welderPtr[i] = &welders[i];
* }
* printf("\n--- Welder Skill Details ---\n");
* for (int i = 0; i < n; i++) {
* displayWelderSkill(welderPtr[i]);
* }
* return 0;
* }

**19. Welding Arc Stability Analyzer**

**Description:**  
Design a program to analyze welding arc stability using structures for stability data, arrays for voltage readings, and unions for different stability metrics.

**Specifications:**

* **Structure:** Contains stability ID, voltage, and current.
* **Union:** Represents stability metrics like arc length or consistency.
* **Array:** Voltage readings.
* **const Pointers:** Protect stability data.
* **Double Pointers:** Allocate and manage stability records dynamically.
* #include <stdio.h>
* #define MAX\_STABILITY\_RECORDS 5
* struct Stability {
* int stabilityID;
* float voltage;
* float current;
* };
* union StabilityMetrics {
* float arcLength;
* float consistency;
* };
* void displayStabilityData(const struct Stability\* stability, const union StabilityMetrics\* metrics) {
* printf("\nStability ID: %d\n", stability->stabilityID);
* printf("Voltage: %.2f V\n", stability->voltage);
* printf("Current: %.2f A\n", stability->current);
* printf("Arc Length: %.2f mm\n", metrics->arcLength);
* printf("Consistency: %.2f %%\n", metrics->consistency);
* }
* int main() {
* int n;
* printf("Enter the number of stability records: ");
* scanf("%d", &n);
* struct Stability stabilities[MAX\_STABILITY\_RECORDS];
* union StabilityMetrics metrics[MAX\_STABILITY\_RECORDS];
* const struct Stability\* stabilityPtr[MAX\_STABILITY\_RECORDS];
* const union StabilityMetrics\* metricsPtr[MAX\_STABILITY\_RECORDS];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for stability record %d:\n", i + 1);
* printf("Enter Stability ID: ");
* scanf("%d", &stabilities[i].stabilityID);
* printf("Enter Voltage (in Volts): ");
* scanf("%f", &stabilities[i].voltage);
* printf("Enter Current (in Amps): ");
* scanf("%f", &stabilities[i].current);
* printf("Enter Arc Length (in mm): ");
* scanf("%f", &metrics[i].arcLength);
* printf("Enter Consistency (in %%): ");
* scanf("%f", &metrics[i].consistency);
* stabilityPtr[i] = &stabilities[i];
* metricsPtr[i] = &metrics[i];
* }
* printf("\n--- Welding Arc Stability Records ---\n");
* for (int i = 0; i < n; i++) {
* displayStabilityData(stabilityPtr[i], metricsPtr[i]);
* }
* return 0;
* }

**20. Welding Training Simulation System**

**Description:**  
Create a simulation system for welding training using structures for training details, arrays for progress data, and strings for training modules.

**Specifications:**

* **Structure:** Stores training ID, module name, and trainee progress.
* **Array:** Progress data for multiple trainees.
* **Strings:** Training module names.
* **const Pointers:** Protect training details.
* **Double Pointers:** Manage training records dynamically.
* #include <stdio.h>
* #define MAX\_TRAINEES 5
* #define MAX\_MODULES 3
* struct Training {
* int trainingID;
* char moduleName[50];
* float progress;
* };
* void displayTrainingDetails(const struct Training\* training) {
* printf("\nTraining ID: %d\n", training->trainingID);
* printf("Module Name: %s\n", training->moduleName);
* printf("Trainee Progress: %.2f%%\n", training->progress);
* }
* int main() {
* int n;
* printf("Enter the number of trainees: ");
* scanf("%d", &n);
* struct Training trainingRecords[MAX\_TRAINEES];
* const struct Training\* trainingPtr[MAX\_TRAINEES];
* for (int i = 0; i < n; i++) {
* printf("\nEnter details for trainee %d:\n", i + 1);
* printf("Enter Training ID: ");
* scanf("%d", &trainingRecords[i].trainingID);
* printf("Enter Module Name: ");
* scanf(" %[^\n]", trainingRecords[i].moduleName);
* printf("Enter Progress (in %%): ");
* scanf("%f", &trainingRecords[i].progress);
* trainingPtr[i] = &trainingRecords[i];
* }
* printf("\n--- Welding Training Simulation Records ---\n");
* for (int i = 0; i < n; i++) {
* displayTrainingDetails(trainingPtr[i]);
* }
* return 0;
* }