SET OF PROBLEMS
==============================

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>
#include <string.h>

// Union for variable parameters (Voltage or Current)
union WeldParameter {
    float voltage;
    float current;
};

// Structure to store weld type details
struct WeldType {
    int weldTypeID;
    char name[50];
    union WeldParameter param; // Voltage or Current (depending on the configuration)
    int isVoltage;  // 1 if using voltage, 0 if using current
};

// Function to print a weld configuration
void printWeldType(struct WeldType *weld) {
    if (weld->isVoltage) {
        printf("Weld Type ID: %d\n", weld->weldTypeID);
        printf("Name: %s\n", weld->name);
        printf("Voltage: %.2f V\n", weld->param.voltage);
    } else {
        printf("Weld Type ID: %d\n", weld->weldTypeID);
        printf("Name: %s\n", weld->name);
        printf("Current: %.2f A\n", weld->param.current);
    }
}

// Function to dynamically allocate memory for multiple weld types
void allocateWeldTypes(struct WeldType ***weldArray, int numTypes) {
    // Allocate memory for the array of pointers to WeldType
    *weldArray = (struct WeldType **)malloc(numTypes * sizeof(struct WeldType *));

    // Allocate memory for each WeldType
    for (int i = 0; i < numTypes; i++) {
        (*weldArray)[i] = (struct WeldType *)malloc(sizeof(struct WeldType));
    }
}
```

```
// Function to free dynamically allocated memory
void freeWeldTypes(struct WeldType ***weldArray, int numTypes) {
    for (int i = 0; i < numTypes; i++) {
        free((*weldArray)[i]);
    }
    free(*weldArray);
}

int main() {
    int numTypes = 2;
    struct WeldType **weldConfigurations;

    // Dynamically allocate memory for weld configurations
    allocateWeldTypes(&weldConfigurations, numTypes);

    // Define the first weld type configuration (using voltage)
    weldConfigurations[0]->weldTypeID = 1;
    strcpy(weldConfigurations[0]->name, "MIG");
    weldConfigurations[0]->param.voltage = 24.5f;
    weldConfigurations[0]->isVoltage = 1; // Using voltage

    // Define the second weld type configuration (using current)
    weldConfigurations[1]->weldTypeID = 2;
    strcpy(weldConfigurations[1]->name, "TIG");
    weldConfigurations[1]->param.current = 150.0f;
    weldConfigurations[1]->isVoltage = 0; // Using current

    // Print out all the configurations
    for (int i = 0; i < numTypes; i++) {
        printWeldType(weldConfigurations[i]);
        printf("\n");
    }

    // Free the dynamically allocated memory
    freeWeldTypes(&weldConfigurations, numTypes);

    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>
#include <string.h>

#define MAX_MODE_LENGTH 50
```

```
#define MAX_VOLTAGE_RANGE 1000

// Define the welding machine structure
typedef struct {
    int machineID;                // Unique ID for the machine
    char mode[MAX_MODE_LENGTH];   // Mode of the welding machine
    int speed;                    // Welding speed
    int voltageMin;               // Minimum voltage
    int voltageMax;               // Maximum voltage
} WeldingMachine;

// Function to initialize a welding machine
void initWeldingMachine(WeldingMachine *machine, int id, const char *mode, int speed, int voltageMin,
int voltageMax) {
    machine->machineID = id;
    strncpy(machine->mode, mode, MAX_MODE_LENGTH);
    machine->speed = speed;
    machine->voltageMin = voltageMin;
    machine->voltageMax = voltageMax;
}

// Function to display machine settings
void displayMachineSettings(const WeldingMachine *machine) {
    printf("Machine ID: %d\n", machine->machineID);
    printf("Mode: %s\n", machine->mode);
    printf("Speed: %d\n", machine->speed);
    printf("Voltage Range: %dV - %dV\n", machine->voltageMin, machine->voltageMax);
}

// Function to create and manage multiple machines dynamically
void manageMachines(WeldingMachine ***machines, int numMachines) {
    // Dynamically allocate memory for the array of machines
    *machines = (WeldingMachine **)malloc(numMachines * sizeof(WeldingMachine *));

    // Initialize each machine
    for (int i = 0; i < numMachines; i++) {
        (*machines)[i] = (WeldingMachine *)malloc(sizeof(WeldingMachine));
        int id = i + 1;
        const char *mode = (i % 2 == 0) ? "MIG" : "TIG";
        int speed = 100 + (i * 10);
        int voltageMin = 150 + (i * 10);
        int voltageMax = voltageMin + 50;

        initWeldingMachine((*machines)[i], id, mode, speed, voltageMin, voltageMax);
    }
}

// Function to free dynamically allocated memory
void freeMachines(WeldingMachine **machines, int numMachines) {
    for (int i = 0; i < numMachines; i++) {
        free(machines[i]);
    }
    free(machines);
}
```

```
int main() {
    WeldingMachine **machines = NULL;
    int numMachines = 5;  // Example number of machines

    // Manage the welding machines dynamically
    manageMachines(&machines, numMachines);

    // Display the settings for each machine
    for (int i = 0; i < numMachines; i++) {
        printf("\nMachine %d Settings:\n", i + 1);
        displayMachineSettings(machines[i]);
    }

    // Free dynamically allocated memory
    freeMachines(machines, numMachines);

    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>
#include <string.h>

// Define a structure to store metadata of a welding process
typedef struct {
    int processID;
    char material[50];
    char welderName[50];
} WeldingProcess;

// Define a union to store either heat input or arc length for a process
typedef union {
    double heatInput; // in Joules
    double arcLength; // in millimeters
} WeldingMetric;

// Define an array to store multiple welding processes
WeldingProcess *processArray;
int numProcesses = 0;  // Tracks the number of processes

// Function to add a new welding process
void addWeldingProcess(int processID, const char *material, const char *welderName, WeldingMetric
metric, int isHeatInput) {
    // Allocate memory for a new process record
```

```
    processArray = realloc(processArray, (numProcesses + 1) * sizeof(WeldingProcess));

    // Store process metadata
    WeldingProcess *newProcess = &processArray[numProcesses];
    newProcess->processID = processID;
    strncpy(newProcess->material, material, sizeof(newProcess->material) - 1);
    strncpy(newProcess->welderName, welderName, sizeof(newProcess->welderName) - 1);

    // Handle welding metrics (heat input or arc length)
    if (isHeatInput) {
        newProcess->heatInput = metric.heatInput;
    } else {
        newProcess->arcLength = metric.arcLength;
    }

    numProcesses++;
}

// Function to display the details of all welding processes
void displayProcesses() {
    printf("\n--- Welding Process Details ---\n");
    for (int i = 0; i < numProcesses; i++) {
        printf("Process ID: %d\n", processArray[i].processID);
        printf("Material: %s\n", processArray[i].material);
        printf("Welder: %s\n", processArray[i].welderName);

        // Display the welding metric (either heat input or arc length)
        if (processArray[i].heatInput != 0.0) {
            printf("Heat Input: %.2f Joules\n", processArray[i].heatInput);
        } else {
            printf("Arc Length: %.2f mm\n", processArray[i].arcLength);
        }

        printf("\n");
    }
}

// Function to release allocated memory
void freeMemory() {
    free(processArray);
}

int main() {
    WeldingMetric metric1, metric2;

    // Add some welding processes
    metric1.heatInput = 500.0;
    addWeldingProcess(101, "Steel", "John Doe", metric1, 1);

    metric2.arcLength = 12.5;
    addWeldingProcess(102, "Aluminum", "Jane Smith", metric2, 0);

    metric1.heatInput = 350.0;
    addWeldingProcess(103, "Copper", "Alan Turner", metric1, 1);
```

```
    // Display all the welding processes
    displayProcesses();

    // Free the dynamically allocated memory for process array
    freeMemory();

    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>
#include <string.h>

// Union for different geometry parameters (width, depth, height)
union GeometryParams {
    float width;
    float depth;
    float height;
};

// Structure to store bead details including ID, material, and geometry type
struct WeldBead {
    int beadID;                 // Bead ID
    char material[50];          // Material type of the weld bead
    char geometryType[20];      // Type of geometry (flat, convex, concave, etc.)
    union GeometryParams params[3]; // Array of union to store measurements
};

// Function to print a bead's geometry details
void printBeadDetails(const struct WeldBead *bead) {
    printf("Weld Bead ID: %d\n", bead->beadID);
    printf("Material: %s\n", bead->material);
    printf("Geometry Type: %s\n", bead->geometryType);
    printf("Width: %.2f\n", bead->params[0].width);
    printf("Depth: %.2f\n", bead->params[1].depth);
    printf("Height: %.2f\n", bead->params[2].height);
}

// Function to dynamically allocate memory for multiple weld beads
void allocateBeads(struct WeldBead ***beads, int numBeads) {
    *beads = (struct WeldBead **)malloc(numBeads * sizeof(struct WeldBead *));
    if (*beads == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
```

```
    }

    // Dynamically allocate memory for each bead
    for (int i = 0; i < numBeads; i++) {
        (*beads)[i] = (struct WeldBead *)malloc(sizeof(struct WeldBead));
        if ((*beads)[i] == NULL) {
            printf("Memory allocation for bead %d failed!\n", i);
            exit(1);
        }
    }
}

// Function to free dynamically allocated memory for the beads
void freeBeads(struct WeldBead **beads, int numBeads) {
    for (int i = 0; i < numBeads; i++) {
        free(beads[i]);
    }
    free(beads);
}

int main() {
    struct WeldBead **beads;
    int numBeads = 2;

    // Dynamically allocate memory for the bead records
    allocateBeads(&beads, numBeads);

    // Initialize bead 1
    beads[0]->beadID = 101;
    strcpy(beads[0]->material, "Steel");
    strcpy(beads[0]->geometryType, "Flat");
    beads[0]->params[0].width = 10.5f;
    beads[0]->params[1].depth = 3.2f;
    beads[0]->params[2].height = 2.8f;

    // Initialize bead 2
    beads[1]->beadID = 102;
    strcpy(beads[1]->material, "Aluminum");
    strcpy(beads[1]->geometryType, "Concave");
    beads[1]->params[0].width = 8.3f;
    beads[1]->params[1].depth = 4.5f;
    beads[1]->params[2].height = 3.0f;

    // Print details of each weld bead
    for (int i = 0; i < numBeads; i++) {
        printBeadDetails(beads[i]);
        printf("\n");
    }

    // Free dynamically allocated memory
    freeBeads(beads, numBeads);

    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>
#include <string.h>

// Define the structure for a consumable
typedef struct {
    int id;              // Consumable ID
    char type[30];       // Consumable type (e.g., "Electrode", "Filler Material")
    int quantity;        // Quantity of consumables
} Consumable;

// Function to create a new consumable and add it to the inventory
void addConsumable(Consumable **inventory, int *size, int id, const char *type, int quantity) {
    // Reallocate memory for the new consumable
    *inventory = realloc(*inventory, (*size + 1) * sizeof(Consumable));
    if (*inventory == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }

    // Add new consumable to the inventory
    (*inventory)[*size].id = id;
    strncpy((*inventory)[*size].type, type, 30);  // Safely copy the consumable type
    (*inventory)[*size].quantity = quantity;

    (*size)++;  // Increase the inventory size
}

// Function to display the inventory
void displayInventory(Consumable *inventory, int size) {
    printf("Welding Consumable Inventory:\n");
    printf("%-10s%-20s%-10s\n", "ID", "Type", "Quantity");
    for (int i = 0; i < size; i++) {
        printf("%-10d%-20s%-10d\n", inventory[i].id, inventory[i].type, inventory[i].quantity);
    }
}

// Function to update the quantity of a consumable
void updateQuantity(Consumable *inventory, int size, int id, int quantity) {
    for (int i = 0; i < size; i++) {
        if (inventory[i].id == id) {
            inventory[i].quantity += quantity;
            printf("Updated quantity of consumable ID %d. New quantity: %d\n", id, inventory[i].quantity);
            return;
```

```
        }
    }
    printf("Consumable with ID %d not found.\n", id);
}

int main() {
    Consumable *inventory = NULL;  // Pointer to array of consumables (dynamically allocated)
    int inventorySize = 0;         // Tracks the number of consumables in the inventory

    // Add consumables to the inventory
    addConsumable(&inventory, &inventorySize, 1, "Electrode", 50);
    addConsumable(&inventory, &inventorySize, 2, "Filler Material", 100);
    addConsumable(&inventory, &inventorySize, 3, "Flux", 30);

    // Display current inventory
    displayInventory(inventory, inventorySize);

    // Update quantity of a consumable
    updateQuantity(inventory, inventorySize, 2, 20);

    // Display updated inventory
    displayInventory(inventory, inventorySize);

    // Free dynamically allocated memory for the inventory
    free(inventory);

    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>
#include <string.h>

// Define the structure for safety equipment
struct SafetyEquipment {
    int id;                // Equipment ID
    char type[30];         // Equipment type (e.g., gloves, helmet)
    int usageFrequency;    // How often it's used (number of uses)
};

// Function to display the equipment details
void displayEquipment(struct SafetyEquipment* equipment) {
    printf("Equipment ID: %d\n", equipment->id);
    printf("Equipment Type: %s\n", equipment->type);
```

```
    printf("Usage Frequency: %d\n", equipment->usageFrequency);
}

// Function to display all equipment and their availability
void displayAllEquipment(struct SafetyEquipment** equipmentArray, int numEquipments, int* availability)
{
    printf("\nSafety Equipment Details:\n");
    for (int i = 0; i < numEquipments; i++) {
        printf("\n");
        displayEquipment(equipmentArray[i]);
        printf("Availability Status: %s\n", availability[i] ? "Available" : "Not Available");
    }
}

// Function to initialize and allocate dynamic memory for equipment records
void initializeEquipment(struct SafetyEquipment*** equipmentArray, int numEquipments) {
    // Allocate memory for the equipment array using double pointer
    *equipmentArray = (struct SafetyEquipment**)malloc(numEquipments * sizeof(struct
SafetyEquipment*));

    for (int i = 0; i < numEquipments; i++) {
        // Allocate memory for each equipment record
        (*equipmentArray)[i] = (struct SafetyEquipment*)malloc(sizeof(struct SafetyEquipment));
    }
}

// Function to free dynamically allocated memory
void freeEquipment(struct SafetyEquipment** equipmentArray, int numEquipments) {
    for (int i = 0; i < numEquipments; i++) {
        free(equipmentArray[i]);
    }
    free(equipmentArray);
}

int main() {
    int numEquipments = 3;  // Number of safety equipment items
    struct SafetyEquipment** equipmentArray; // Double pointer to hold equipment data
    int availability[] = {1, 0, 1};  // 1 for available, 0 for not available

    // Initialize dynamic memory for equipment records
    initializeEquipment(&equipmentArray, numEquipments);

    // Fill in the equipment details
    equipmentArray[0]->id = 1;
    strcpy(equipmentArray[0]->type, "Welding Helmet");
    equipmentArray[0]->usageFrequency = 50;

    equipmentArray[1]->id = 2;
    strcpy(equipmentArray[1]->type, "Welding Gloves");
    equipmentArray[1]->usageFrequency = 30;

    equipmentArray[2]->id = 3;
    strcpy(equipmentArray[2]->type, "Welding Jacket");
    equipmentArray[2]->usageFrequency = 25;
```

```
    // Display the safety equipment details and availability
    displayAllEquipment(equipmentArray, numEquipments, availability);

    // Free allocated memory
    freeEquipment(equipmentArray, numEquipments);

    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>

// Enum for defect types
typedef enum {
    POROSITY,
    CRACKING,
    SPATTER
} DefectType;

// Union for different defect types
typedef union {
    float porosity_size; // Size of porosity in mm
    float crack_length;  // Length of the crack in mm
    int spatter_count;   // Number of spatter instances
} DefectDetail;

// Structure to hold information about a defect
typedef struct {
    int defect_id;        // Unique defect ID
    DefectType defect_type; // Type of the defect (e.g., porosity, cracking, spatter)
    int severity;         // Severity level (1-10)
    DefectDetail details; // Specific details for the defect type
} Defect;

// Function to display defect details
void display_defect(Defect *def) {
    printf("Defect ID: %d\n", def->defect_id);
    switch (def->defect_type) {
        case POROSITY:
            printf("Defect Type: Porosity\n");
            printf("Porosity Size: %.2f mm\n", def->details.porosity_size);
            break;
        case CRACKING:
            printf("Defect Type: Cracking\n");
```

```
            printf("Crack Length: %.2f mm\n", def->details.crack_length);
            break;
        case SPATTER:
            printf("Defect Type: Spatter\n");
            printf("Spatter Count: %d\n", def->details.spatter_count);
            break;
        default:
            printf("Unknown Defect Type\n");
    }
    printf("Severity Level: %d\n", def->severity);
}

// Function to dynamically allocate memory for defects and initialize values
Defect* create_defect(int defect_id, DefectType defect_type, int severity, DefectDetail details) {
    Defect* new_defect = (Defect*)malloc(sizeof(Defect));
    if (new_defect != NULL) {
        new_defect->defect_id = defect_id;
        new_defect->defect_type = defect_type;
        new_defect->severity = severity;
        new_defect->details = details;
    }
    return new_defect;
}

// Function to free dynamically allocated defect memory
void free_defect(Defect* def) {
    free(def);
}

int main() {
    // Example of porosity defect
    DefectDetail porosity_details;
    porosity_details.porosity_size = 2.5; // size in mm

    // Create a defect with porosity type
    Defect* defect1 = create_defect(101, POROSITY, 8, porosity_details);
    display_defect(defect1);

    // Example of cracking defect
    DefectDetail cracking_details;
    cracking_details.crack_length = 4.0; // length in mm

    // Create a defect with cracking type
    Defect* defect2 = create_defect(102, CRACKING, 6, cracking_details);
    display_defect(defect2);

    // Example of spatter defect
    DefectDetail spatter_details;
    spatter_details.spatter_count = 15;

    // Create a defect with spatter type
    Defect* defect3 = create_defect(103, SPATTER, 7, spatter_details);
    display_defect(defect3);

    // Free dynamically allocated memory
```

```
    free_defect(defect1);
    free_defect(defect2);
    free_defect(defect3);

    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>
#include <string.h>

// Define the structure for the performance metrics
typedef struct {
    int performanceID;       // Unique identifier for the performance test
    char materialType[50];   // Material type being welded (e.g., Steel, Aluminum)
    int currentSetting;      // Current setting for the welding process (in amperes)
} PerformanceMetrics;

// Define a union to store either arc stability or penetration depth
typedef union {
    float arcStability;      // Arc stability factor (between 0 and 1)
    float penetrationDepth;  // Penetration depth in mm
} WeldingFactors;

// Define an array of output data to store performance results
#define MAX_RESULTS 100
PerformanceMetrics outputData[MAX_RESULTS];

// Function to analyze the performance of arc welding
void analyzePerformance(PerformanceMetrics *metrics, WeldingFactors *factor, int *resultCount) {
    // Simulate some analysis based on the material type and current setting
    if (strcmp(metrics->materialType, "Steel") == 0) {
        // Assuming arc stability and penetration depth based on current setting
        if (metrics->currentSetting < 100) {
            factor->arcStability = 0.8;
            factor->penetrationDepth = 2.0;
        } else {
            factor->arcStability = 0.9;
            factor->penetrationDepth = 3.5;
        }
    } else if (strcmp(metrics->materialType, "Aluminum") == 0) {
        if (metrics->currentSetting < 150) {
            factor->arcStability = 0.75;
            factor->penetrationDepth = 1.5;
        } else {
```

```
            factor->arcStability = 0.85;
            factor->penetrationDepth = 2.8;
        }
    } else {
        factor->arcStability = 0.7;
        factor->penetrationDepth = 1.0;
    }

    // Store the performance result in the output array
    outputData[*resultCount] = *metrics;
    (*resultCount)++;
}

// Function to display the performance data
void displayResults(int resultCount) {
    printf("Performance Analysis Results:\n");
    for (int i = 0; i < resultCount; i++) {
        printf("Performance ID: %d\n", outputData[i].performanceID);
        printf("Material Type: %s\n", outputData[i].materialType);
        printf("Current Setting: %d A\n", outputData[i].currentSetting);
        printf("Arc Stability: %.2f\n", outputData[i].currentSetting < 100 ? 0.8 : 0.9); // Simplified
        printf("Penetration Depth: %.2f mm\n\n", outputData[i].currentSetting < 100 ? 2.0 : 3.5); // Simplified
    }
}

int main() {
    PerformanceMetrics *configurations = NULL;
    WeldingFactors *weldingFactor = NULL;
    int resultCount = 0;

    // Allocate memory dynamically for configurations (double pointers)
    configurations = (PerformanceMetrics *)malloc(sizeof(PerformanceMetrics) * MAX_RESULTS);
    weldingFactor = (WeldingFactors *)malloc(sizeof(WeldingFactors) * MAX_RESULTS);

    if (configurations == NULL || weldingFactor == NULL) {
        printf("Memory allocation failed!\n");
        return -1;
    }

    // Sample test cases
    PerformanceMetrics test1 = {1, "Steel", 90};
    PerformanceMetrics test2 = {2, "Aluminum", 120};

    // Analyze performance for test cases
    analyzePerformance(&test1, &weldingFactor[resultCount], &resultCount);
    analyzePerformance(&test2, &weldingFactor[resultCount], &resultCount);

    // Display results
    displayResults(resultCount);

    // Free dynamically allocated memory
    free(configurations);
    free(weldingFactor);

    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>
#include <string.h>

// Define the structure to hold task details
struct Task {
    int taskID;
    int priority;
    int duration; // in minutes
    char taskName[50]; // Task name
};

// Function to compare tasks based on priority (highest priority first)
int compareTasks(const void *a, const void *b) {
    struct Task *taskA = (struct Task *)a;
    struct Task *taskB = (struct Task *)b;

    // Sort by priority first
    return taskB->priority - taskA->priority;
}

// Function to allocate and manage task records dynamically
void scheduleWelding(struct Task **tasks, int numTasks, int *timeSlots, int numSlots) {
    // Sort the tasks based on priority
    qsort(*tasks, numTasks, sizeof(struct Task), compareTasks);

    int slotIndex = 0; // To track available time slots
    printf("Welding Schedule:\n");

    for (int i = 0; i < numTasks; i++) {
        struct Task *currentTask = &(*tasks)[i];

        // Find an available time slot for the current task
        if (slotIndex + currentTask->duration <= numSlots) {
            printf("Task: %s (ID: %d), Priority: %d, Duration: %d minutes\n",
                currentTask->taskName, currentTask->taskID, currentTask->priority, currentTask->duration);
            slotIndex += currentTask->duration; // Allocate time slots
        } else {
            printf("Not enough time slots for task: %s (ID: %d)\n",
                currentTask->taskName, currentTask->taskID);
        }
    }
```

```
}

int main() {
    int numTasks = 5;
    int numSlots = 100; // Total available time slots

    // Dynamically allocate memory for task records
    struct Task *tasks = (struct Task *)malloc(numTasks * sizeof(struct Task));

    // Assigning sample task data
    tasks[0] = (struct Task){1, 3, 30, "Weld Pipe"};
    tasks[1] = (struct Task){2, 1, 20, "Weld Frame"};
    tasks[2] = (struct Task){3, 5, 40, "Weld Plate"};
    tasks[3] = (struct Task){4, 2, 10, "Weld Rod"};
    tasks[4] = (struct Task){5, 4, 50, "Weld Door"};

    // Array of available time slots
    int timeSlots[100] = {0}; // 0 means the slot is available

    // Call the scheduling function
    scheduleWelding(&tasks, numTasks, timeSlots, numSlots);

    // Free the dynamically allocated memory
    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>
#include <stdlib.h>

// Define the union for defect parameters
union DefectParams {
    double size;  // size of defect
    double location;  // location of defect
    // You could add other parameters, like type of defect, if needed
};

// Define the structure for storing inspection details
struct WeldInspection {
    int inspectionID;  // Unique identifier for the inspection
    char method[50];   // Method of inspection (e.g., visual, ultrasonic, etc.)
    union DefectParams defect;  // Defect details (could be size or location)
    double *measurementData;  // Array of measurement data (dynamic)
```

```
    int numMeasurements;  // Number of measurements in the array
};

// Function to initialize a weld inspection
void initializeInspection(struct WeldInspection *inspection, int id, const char *method, int
numMeasurements) {
    inspection->inspectionID = id;
    snprintf(inspection->method, sizeof(inspection->method), "%s", method);
    inspection->measurementData = (double *)malloc(sizeof(double) * numMeasurements);
    inspection->numMeasurements = numMeasurements;
    // Initialize measurement data with zeros or any other logic
    for (int i = 0; i < numMeasurements; i++) {
        inspection->measurementData[i] = 0.0;
    }
}

// Function to add a defect to the inspection (e.g., size of defect)
void addDefect(struct WeldInspection *inspection, double size) {
    inspection->defect.size = size;
}

// Function to print the inspection details
void printInspection(const struct WeldInspection *inspection) {
    printf("Inspection ID: %d\n", inspection->inspectionID);
    printf("Inspection Method: %s\n", inspection->method);
    printf("Number of Measurements: %d\n", inspection->numMeasurements);

    printf("Measurement Data: ");
    for (int i = 0; i < inspection->numMeasurements; i++) {
        printf("%.2f ", inspection->measurementData[i]);
    }
    printf("\n");

    // Print defect size if it's available
    printf("Defect Size: %.2f\n", inspection->defect.size);
}

// Function to free dynamically allocated memory
void freeInspection(struct WeldInspection *inspection) {
    if (inspection->measurementData) {
        free(inspection->measurementData);
    }
}

int main() {
    // Declare a pointer to a WeldInspection structure
    struct WeldInspection *inspection = (struct WeldInspection *)malloc(sizeof(struct WeldInspection));

    // Initialize the inspection
    int numMeasurements = 5;
    initializeInspection(inspection, 101, "Ultrasonic", numMeasurements);

    // Simulate measurement data
    for (int i = 0; i < numMeasurements; i++) {
        inspection->measurementData[i] = 10.0 + i * 2;  // Just an example of measurement data
```

```
    }

    // Add defect information (size of defect)
    addDefect(inspection, 1.5);  // Defect size

    // Print the inspection details
    printInspection(inspection);

    // Free allocated memory
    freeInspection(inspection);

    // Free the inspection structure itself
    free(inspection);

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

#define MAX_MOTIONS 100    // Max number of motion data points
#define MAX_ROBOT_TYPES 10 // Max number of robot types

// Define the Robot structure
typedef struct {
    int robotID;
    char *configuration; // Store configuration dynamically
    char status[20];     // Store robot status (e.g., "Idle", "Working", "Error")
} Robot;

// Store motion data in an array
int motionData[MAX_MOTIONS];

// Define robot types as strings
char *robotTypes[MAX_ROBOT_TYPES] = {"Welding_Robot_A", "Welding_Robot_B",
"Welding_Robot_C"};

// Function to assign robot configuration dynamically
void setRobotConfiguration(Robot *r, const char *config) {
    r->configuration = malloc(strlen(config) + 1); // Allocate memory for configuration
    strcpy(r->configuration, config); // Copy the configuration
}
```

```
// Function to create a new robot dynamically
void createRobot(Robot ***robots, int *count, int robotID, const char *config, const char *status) {
    // Allocate memory for new robot record
    *robots = realloc(*robots, (*count + 1) * sizeof(Robot *));
    (*robots)[*count] = malloc(sizeof(Robot)); // Allocate memory for the robot

    (*robots)[*count]->robotID = robotID;
    setRobotConfiguration((*robots)[*count], config); // Set the robot configuration
    strcpy((*robots)[*count]->status, status); // Set the robot’s status

    (*count)++; // Increment the robot count
}

// Function to set motion data
void setMotionData() {
    for (int i = 0; i < MAX_MOTIONS; i++) {
        motionData[i] = i * 10; // Example motion data (multiples of 10)
    }
}

// Function to display robot information
void displayRobotInfo(Robot **robots, int robotCount) {
    for (int i = 0; i < robotCount; i++) {
        printf("Robot ID: %d\n", robots[i]->robotID);
        printf("Configuration: %s\n", robots[i]->configuration);
        printf("Status: %s\n", robots[i]->status);
        printf("Motion Data: ");
        for (int j = 0; j < MAX_MOTIONS; j++) {
            printf("%d ", motionData[j]);
        }
        printf("\n\n");
    }
}

// Function to free allocated memory for robots
void freeMemory(Robot **robots, int robotCount) {
    for (int i = 0; i < robotCount; i++) {
        free(robots[i]->configuration); // Free configuration memory
        free(robots[i]); // Free robot object memory
    }
    free(robots); // Free the array of robot pointers
}

int main() {
    Robot **robots = NULL; // Double pointer for dynamically managing robots
    int robotCount = 0;    // Initialize robot count to 0

    // Create robots dynamically
    createRobot(&robots, &robotCount, 1, "Config_A", "Idle");
    createRobot(&robots, &robotCount, 2, "Config_B", "Working");

    // Set motion data (this could represent a robot’s motion instructions)
    setMotionData();

    // Display robot information
```

```
    displayRobotInfo(robots, robotCount);

    // Free dynamically allocated memory
    freeMemory(robots, robotCount);

    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 <stdlib.h>

// Union to represent different quality parameters
union QualityParameters {
    float tensile_strength;  // Tensile strength in MPa
    float hardness;          // Hardness value
    float porosity;          // Porosity level (percentage)
};

// Structure to store weld details
struct Weld {
    int weld_id;                // Weld ID
    char material[20];          // Material used
    float quality_score;        // Quality score of the weld
    union QualityParameters quality;  // Quality parameter (tensile_strength, hardness, or porosity)
};

// Function to log weld details
void logWeldData(struct Weld* weld) {
    printf("Weld ID: %d\n", weld->weld_id);
    printf("Material: %s\n", weld->material);
    printf("Quality Score: %.2f\n", weld->quality_score);

    printf("Quality Parameter:\n");
    // Assuming we’re logging tensile strength for the example
    printf("Tensile Strength: %.2f MPa\n", weld->quality.tensile_strength);
}

// Main function
int main() {
    // Array of welds (static array for simplicity)
    struct Weld welds[3];

    // Initialize weld 1
    welds[0].weld_id = 101;
```

```
    snprintf(welds[0].material, sizeof(welds[0].material), "Steel");
    welds[0].quality_score = 85.5;
    welds[0].quality.tensile_strength = 520.0; // Example parameter for weld 1

    // Initialize weld 2
    welds[1].weld_id = 102;
    snprintf(welds[1].material, sizeof(welds[1].material), "Aluminum");
    welds[1].quality_score = 92.0;
    welds[1].quality.tensile_strength = 460.0; // Example parameter for weld 2

    // Initialize weld 3
    welds[2].weld_id = 103;
    snprintf(welds[2].material, sizeof(welds[2].material), "Copper");
    welds[2].quality_score = 78.4;
    welds[2].quality.tensile_strength = 490.0; // Example parameter for weld 3

    // Print the data for each weld
    for (int i = 0; i < 3; i++) {
        logWeldData(&welds[i]);
        printf("\n");
    }

    // Using constant pointer to protect weld details
    const struct Weld* const_weld = &welds[0];
    printf("Constant Pointer - Weld ID: %d, Material: %s\n", const_weld->weld_id, const_weld->material);

    // Dynamic memory allocation for double pointer (e.g., for dynamic quality data)
    struct Weld** dynamic_welds = malloc(3 * sizeof(struct Weld*));
    for (int i = 0; i < 3; i++) {
        dynamic_welds[i] = &welds[i];
    }

    // Printing dynamic weld data using double pointer
    for (int i = 0; i < 3; i++) {
        printf("Dynamic Pointer - Weld ID: %d, Material: %s\n", dynamic_welds[i]->weld_id,
dynamic_welds[i]->material);
    }

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

    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>
#include <stdlib.h>

#define MAX_TIME 10

// Structure to hold thermal input details
typedef struct {
    int thermalInputID;
    float current; // in Amperes
    float voltage; // in Volts
} ThermalInput;

// Union to represent heat input or time-temperature correlation
typedef union {
    float heatInput; // Heat input in Joules
    struct {
        float time[MAX_TIME];      // Time data array
        float temperature[MAX_TIME]; // Corresponding temperature data array
    } timeTemperature;
} HeatOrTimeTemp;

// Function to calculate the heat input
float calculateHeatInput(float current, float voltage) {
    return current * voltage; // Heat Input = Voltage * Current (simplified model)
}

// Function to dynamically allocate time-temperature data
void storeTimeTemperatureData(double ***data, int size) {
    *data = (double **)malloc(size * sizeof(double *));
    for (int i = 0; i < size; i++) {
        (*data)[i] = (double *)malloc(2 * sizeof(double)); // 2 elements per entry (time, temperature)
    }
}

// Function to display thermal input details
void displayThermalInput(ThermalInput *input) {
    printf("Thermal Input ID: %d\n", input->thermalInputID);
    printf("Current: %.2f A\n", input->current);
    printf("Voltage: %.2f V\n", input->voltage);
}

// Function to display heat or time-temperature data
void displayHeatOrTimeTempData(HeatOrTimeTemp *data, int useTimeTemp) {
    if (useTimeTemp) {
        printf("Time-Temperature Data:\n");
        for (int i = 0; i < MAX_TIME; i++) {
            printf("Time: %.2f s, Temperature: %.2f °C\n", data->timeTemperature.time[i],
data->timeTemperature.temperature[i]);
        }
    } else {
        printf("Heat Input: %.2f Joules\n", data->heatInput);
    }
}

int main() {
```

```
    // Declare and initialize thermal input
    ThermalInput thermalInput = {1, 150.0, 24.0}; // Example: ID=1, Current=150A, Voltage=24V
    HeatOrTimeTemp heatData;

    // Calculate the heat input
    heatData.heatInput = calculateHeatInput(thermalInput.current, thermalInput.voltage);

    // Time-temperature data initialization (example)
    for (int i = 0; i < MAX_TIME; i++) {
        heatData.timeTemperature.time[i] = i * 1.0; // time in seconds
        heatData.timeTemperature.temperature[i] = 100.0 + i * 5.0; // temperature increases by 5°C every
second
    }

    // Display thermal input details
    displayThermalInput(&thermalInput);

    // Display heat input data
    displayHeatOrTimeTempData(&heatData, 0);  // 0 indicates displaying heat input

    // Display time-temperature data
    displayHeatOrTimeTempData(&heatData, 1);  // 1 indicates displaying time-temperature data

    // Dynamically manage time-temperature data (double pointers)
    double **dynamicData;
    storeTimeTemperatureData(&dynamicData, MAX_TIME);

    // Example: Assign dynamic data
    for (int i = 0; i < MAX_TIME; i++) {
        dynamicData[i][0] = i * 1.0;        // time in seconds
        dynamicData[i][1] = 100.0 + i * 5.0; // temperature increases by 5°C
    }

    // Display dynamic time-temperature data
    printf("\nDynamically Allocated Time-Temperature Data:\n");
    for (int i = 0; i < MAX_TIME; i++) {
        printf("Time: %.2f s, Temperature: %.2f °C\n", dynamicData[i][0], dynamicData[i][1]);
    }

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

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

// Define structure for welding procedure specification
typedef struct {
    int procedureID;              // Procedure ID
    char material[50];            // Material type (e.g., Steel, Aluminum)
    char jointType[50];           // Joint type (e.g., Butt, Fillet)
    double weldingParameters[5];  // Array to store parameters like voltage, current, speed, etc.
    char procedureName[100];      // Name of the procedure
} WeldingProcedure;

// Function to display welding procedure details
void displayProcedure(WeldingProcedure *procedure) {
    printf("Procedure ID: %d\n", procedure->procedureID);
    printf("Material: %s\n", procedure->material);
    printf("Joint Type: %s\n", procedure->jointType);
    printf("Procedure Name: %s\n", procedure->procedureName);
    printf("Welding Parameters: ");
    for (int i = 0; i < 5; i++) {
        printf("%.2f ", procedure->weldingParameters[i]);
    }
    printf("\n\n");
}

// Function to create and store a welding procedure
void createProcedure(WeldingProcedure **procedures, int *numProcedures, int id, const char *material,
const char *jointType, const double params[], const char *name) {
    // Allocate memory for a new welding procedure
    *procedures = (WeldingProcedure*) realloc(*procedures, (*numProcedures + 1) *
sizeof(WeldingProcedure));

    // Fill the new procedure details
    WeldingProcedure *newProcedure = &(*procedures)[*numProcedures];
    newProcedure->procedureID = id;
    strncpy(newProcedure->material, material, sizeof(newProcedure->material) - 1);
    strncpy(newProcedure->jointType, jointType, sizeof(newProcedure->jointType) - 1);
    strncpy(newProcedure->procedureName, name, sizeof(newProcedure->procedureName) - 1);

    for (int i = 0; i < 5; i++) {
        newProcedure->weldingParameters[i] = params[i];
    }

    // Increment the procedure count
    (*numProcedures)++;
}

int main() {
    WeldingProcedure *procedures = NULL; // Double pointer for dynamic array of welding procedures
    int numProcedures = 0;               // To track the number of procedures
```

```
    // Create and store some welding procedures
    double params1[] = {10.5, 12.0, 0.75, 100.0, 20.0};
    createProcedure(&procedures, &numProcedures, 1, "Steel", "Butt", params1, "Steel Butt Weld");

    double params2[] = {15.0, 14.0, 1.00, 120.0, 25.0};
    createProcedure(&procedures, &numProcedures, 2, "Aluminum", "Fillet", params2, "Aluminum Fillet
Weld");

    // Display all procedures
    for (int i = 0; i < numProcedures; i++) {
        displayProcedure(&procedures[i]);
    }

    // Free dynamically allocated memory
    free(procedures);

    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>
#include <stdlib.h>

#define MAX_JOINTS 5

// Union for variable joint parameters
typedef union {
    double thickness;  // Thickness for a Butt Joint
    double radius;     // Radius for a Fillet Joint
} JointParams;

// Structure to store joint details
typedef struct {
    int jointID;
    char jointType[20];  // Type of joint (e.g., Butt, Fillet)
    double angle;        // Angle for the joint
    JointParams params;  // Parameters (thickness or radius)
} Joint;

// Function to display joint details
void displayJoint(Joint *joint) {
    printf("Joint ID: %d\n", joint->jointID);
    printf("Joint Type: %s\n", joint->jointType);
    printf("Angle: %.2f\n", joint->angle);
```

```
    if (joint->jointType[0] == 'B') {
        printf("Thickness: %.2f\n", joint->params.thickness);  // Butt joint uses thickness
    } else if (joint->jointType[0] == 'F') {
        printf("Radius: %.2f\n", joint->params.radius);      // Fillet joint uses radius
    }
}

// Function to create and manage joint records dynamically using double pointers
void createJoint(Joint **joints, int jointID, const char *type, double angle, double param) {
    // Dynamically allocate memory for a new joint
    *joints = (Joint *)realloc(*joints, sizeof(Joint));
    (*joints)->jointID = jointID;
    snprintf((*joints)->jointType, sizeof((*joints)->jointType), "%s", type);
    (*joints)->angle = angle;

    if (type[0] == 'B') {
        (*joints)->params.thickness = param;  // Set thickness for Butt Joint
    } else if (type[0] == 'F') {
        (*joints)->params.radius = param;     // Set radius for Fillet Joint
    }
}

// Function to protect joint data with const pointers
void protectJointData(const Joint *joint) {
    printf("\nProtected Joint Data:\n");
    printf("Joint ID: %d\n", joint->jointID);
    printf("Joint Type: %s\n", joint->jointType);
    printf("Angle: %.2f\n", joint->angle);

    if (joint->jointType[0] == 'B') {
        printf("Thickness: %.2f\n", joint->params.thickness);
    } else if (joint->jointType[0] == 'F') {
        printf("Radius: %.2f\n", joint->params.radius);
    }
}

int main() {
    // Array of joints
    Joint *jointsArray = NULL;

    // Create joints with dynamic memory allocation using double pointers
    createJoint(&jointsArray, 1, "Butt", 45.0, 0.25);  // Butt joint with thickness of 0.25
    createJoint(&jointsArray, 2, "Fillet", 90.0, 0.15); // Fillet joint with radius of 0.15

    // Display joint details
    printf("\nJoint Details:\n");
    displayJoint(&jointsArray[0]);  // Display first joint
    displayJoint(&jointsArray[1]);  // Display second joint

    // Protect joint data with const pointer
    protectJointData(&jointsArray[0]);
    protectJointData(&jointsArray[1]);

    // Free the dynamically allocated memory
```

```
    free(jointsArray);

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

// Define a structure to hold filler metal properties
typedef struct {
    int fillerMetalID;
    char composition[50]; // Composition of the metal (e.g., alloy type)
    double diameter;      // Diameter of the filler metal (in mm)
} FillerMetal;

// Function to display details of a filler metal
void displayFillerMetal(FillerMetal *filler) {
    printf("Filler Metal ID: %d\n", filler->fillerMetalID);
    printf("Composition: %s\n", filler->composition);
    printf("Diameter: %.2f mm\n", filler->diameter);
}

// Function to create a new filler metal record dynamically
FillerMetal* createFillerMetal(int id, const char *composition, double diameter) {
    FillerMetal *filler = (FillerMetal*)malloc(sizeof(FillerMetal));
    if (filler != NULL) {
        filler->fillerMetalID = id;
        strcpy(filler->composition, composition);
        filler->diameter = diameter;
    }
    return filler;
}

// Function to release the dynamically allocated memory
void freeFillerMetal(FillerMetal *filler) {
    free(filler);
}

// Main function to test the Filler Metal Selector Tool
int main() {
    // Define the number of filler metals
    int numFillerMetals = 3;
```

```
    // Double pointer to hold the dynamic array of filler metals
    FillerMetal **fillerMetalArray = (FillerMetal**)malloc(numFillerMetals * sizeof(FillerMetal*));

    if (fillerMetalArray == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Create filler metal records and store them in the array
    fillerMetalArray[0] = createFillerMetal(101, "Stainless Steel", 1.5);
    fillerMetalArray[1] = createFillerMetal(102, "Mild Steel", 1.0);
    fillerMetalArray[2] = createFillerMetal(103, "Aluminum", 0.8);

    // Display the details of each filler metal
    printf("Filler Metal Details:\n");
    for (int i = 0; i < numFillerMetals; i++) {
        displayFillerMetal(fillerMetalArray[i]);
    }

    // Free the dynamically allocated memory for filler metals
    for (int i = 0; i < numFillerMetals; i++) {
        freeFillerMetal(fillerMetalArray[i]);
    }
    // Free the memory for the array of pointers
    free(fillerMetalArray);

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

// Define a structure for welding power source details
typedef struct {
    int sourceID;       // Unique ID for each source
    char sourceType[50];  // Type of source, e.g., "MIG", "TIG", "Stick"
    float capacity;     // Capacity of the welding power source (e.g., 200 amps)
} WeldingSource;

// Function to configure and initialize a welding source
void configureSource(WeldingSource *source, int id, const char *type, float cap) {
    source->sourceID = id;
```

```
    strncpy(source->sourceType, type, sizeof(source->sourceType) - 1);
    source->capacity = cap;
}

// Function to print details of a welding source
void printSourceDetails(const WeldingSource *source) {
    printf("Source ID: %d\n", source->sourceID);
    printf("Source Type: %s\n", source->sourceType);
    printf("Source Capacity: %.2f Amps\n", source->capacity);
}

// Function to allocate memory for welding sources using double pointers
void allocateSources(WeldingSource ***sources, int count) {
    *sources = (WeldingSource **)malloc(count * sizeof(WeldingSource *));
    if (*sources == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    for (int i = 0; i < count; i++) {
        (*sources)[i] = (WeldingSource *)malloc(sizeof(WeldingSource));
        if ((*sources)[i] == NULL) {
            printf("Memory allocation for source %d failed\n", i);
            exit(1);
        }
    }
}

// Function to free allocated memory for welding sources
void freeSources(WeldingSource **sources, int count) {
    for (int i = 0; i < count; i++) {
        free(sources[i]);
    }
    free(sources);
}

int main() {
    WeldingSource **sources; // Double pointer to hold multiple source configurations
    int sourceCount = 3;  // Number of welding power sources

    // Allocate memory for the sources
    allocateSources(&sources, sourceCount);

    // Configure the welding sources
    configureSource(sources[0], 1, "MIG", 250.0f);
    configureSource(sources[1], 2, "TIG", 150.0f);
    configureSource(sources[2], 3, "Stick", 300.0f);

    // Print the details of each source
    for (int i = 0; i < sourceCount; i++) {
        printSourceDetails(sources[i]);
        printf("\n");
    }

    // Free allocated memory
    freeSources(sources, sourceCount);
```

```
    return 0;
}
```