1. Alloy Composition Analysis System
Description:
Design a system to analyze alloy compositions using structures for composition details, arrays for storing multiple samples, and
unions to represent percentage compositions of different metals.
Specifications:
Structure: Stores sample ID, name, and composition details.
Union: Represents variable percentage compositions of metals.
Array: Stores multiple alloy samples.
const Pointers: Protect composition details.
Double Pointers: Manage dynamic allocation of alloy samples.

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

union MetalCompostion{
    float percentage;
};

typedef struct {
    int id;
    char name[50];
    const char *details;
    union MetalCompostion metal;

}AlloySample;

void add(AlloySample **alloy,int *count);
void display(AlloySample *alloy,int count);

int main(){
    AlloySample *alloy = NULL;
    int count =0,choice;

    while(1){
        printf("\nAlloy Composition Analysis System\n");
        printf("1. Add\n");
        printf("2.Display\n");
        printf("3.Exit\n");
```

```
        printf("Enter your choice: ");
        scanf("%d",&choice);

        switch (choice)
        {
        case 1:
            add(&alloy,&count);
            break;
        case 2:
            display(alloy,count);
            break;
        case 3:
            printf("Exiting.....\n");
            free(alloy);
            exit(0);
            break;


        default:
            printf("Invalid choice \n");

        }

    }
    return 0;

}
void add(AlloySample **alloy,int *count){
    (*count)++;
    *alloy = (AlloySample *)realloc(*alloy,(*count) * sizeof(AlloySample));
    if(*alloy == NULL){
        printf("Memory Allocation failed\n");
        exit(1);
    }
    AlloySample *new = &(*alloy)[*count-1];
    printf("Enter Sample ID: ");
    scanf("%d",&new->id);
    printf("Enter the sample NAme: ");
    scanf(" %[^\n]s",new->name);

    new->details ="Alloy is Special";
```

```
    printf("Enter the metal composition percentage: ");
    scanf("%f",&new->metal.percentage);

    printf("Sample added successfully \n");
}
void display(AlloySample *alloy,int count){
    if(count ==0){
        printf("No samples availbale \n");
    }
    printf("\n-----Alloy Sample---\n");
    for(int i=0;i<count;i++){
        printf("Sample ID: %d\n",alloy[i].id);
        printf("Sample Name: %s\n",alloy[i].name);
        printf("Details : %s\n",alloy[i].details);
        printf("Metal Composition Percentage: %.2f%%\n",alloy[i].metal.percentage);
        printf("---------------------------------\n");
    }
}
```

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

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

typedef struct
{
    int process_ID;
    const char *process_data;
    char name[50];
```

```
    float temperature;
    int duration;
    int coolingRate;
}Manager;

void add(Manager **manager,int *count);
void display(Manager *manager,int count);

int main(){
    Manager *manager = NULL;
    int count =0,choice;

    while(1){
        printf("----Heat Treatment Process Manager-------\n");
        printf("1. Add\n");
        printf("2.Display \n");
        printf("3. Exit\n");
        printf("Enter the Choice: ");
        scanf("%d",&choice);

        switch(choice){
            case 1:
                add(&manager,&count);
                break;
            case 2:
                display(manager,count);
                break;
            case 3:
                printf("Exiting.....\n");
                free(manager);
                exit(0);
                break;
            default:
                printf("Invalid Input\n");
        }
    }
    return 0;
}


void add(Manager **manager,int *count){
```

```
    (*count)++;
    *manager = (Manager *)realloc(*manager,(*count) * sizeof(Manager));
    if(*manager ==NULL){
        printf("Memory allocation failed \n");
        exit(1);
    }
    Manager *new = &(*manager)[*count-1];
    printf("ENter the id: ");
    scanf("%d",&new->process_ID);
    printf("Enter process Name: ");
    scanf(" %[^\n]s",new->name);
    printf("ENter the temperature: ");
    scanf("%f",&new->temperature);
    printf("Enter Duration:  ");
    scanf("%d",&new->duration);
    new->process_data ="Sucess";
    printf("Enter Cooling rate: ");
    scanf("%d",&new->coolingRate);

    printf("Process Added sucessfully \n");
    printf("---------------------------------------\n");

}


void display(Manager *manager,int count){
    if(count ==0){
        printf("Process is empty \n");
    }
    else{
         printf("-----------------------------------------------------\n");

        for(int i=0;i<count;i++){
            printf("Process ID: %d",manager[i].process_ID);
            printf("Process Name: %s",manager[i].name);
            printf("Process Data: %s",manager->process_data);
            printf("Process Temperature: %f C",manager[i].temperature);
            printf("Duration: %d",manager[i].duration);
            printf("Process Cooling rate: %d",manager[i].coolingRate);

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

```
        }
    }

}
```

3. Steel Quality Monitoring
Description:
Create a system to monitor steel quality using structures for test results, arrays for storing test data, and unions for variable quality metrics like tensile strength and hardness.
Specifications:
Structure: Stores test ID, type, and result.
Union: Represents tensile strength, hardness, or elongation.
Array: Test data for multiple samples.
const Pointers: Protect test IDs.
Double Pointers: Manage dynamic test records.

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

union Quality{
    float tensileStrength;
    float hardness;
    float elongation;
};

typedef struct {
    const int testID;
    char type[50];
    union Quality result;
}Test;

void add(Test **test,int *count);
void display(Test *test,int count);

int main(){
    Test *test = NULL;
    int count =0,choice;
```

```
    while(1){
        printf("\n--- Steel Quality Monitoring System ---\n");
        printf("1. Add Test\n");
        printf("2. Display Tests\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice){
            case 1:
                add(&test,&count);
                break;
            case 2:
                display(test,count);
                break;
            case 3:
                printf("Exiting.....\n");
                free(test);
                exit(0);
                break;
            default:
                printf("Invalid Choice \n");
        }
    }
    return 0;
}

void add(Test **test,int *count){
    (*count)++;
    *test = (Test *)realloc(*test,(*count)*sizeof(Test));
    if(*test ==NULL){
        printf("Memory Allocation failed \n");
        exit(1);
    }
    Test *new = &(*test)[*count -1];
    int id;
    printf("Enter ID: ");
    scanf("%d",&id);
    *(int *)&new->testID = id;
    printf("Enter test type (Tensile,Hardness, Elongation): ");
    scanf(" %[^\n]s",new->type);
```

```
    if(strcmp(new->type,"Tensile")==0){
        printf("Enter tensile strength(MPa): ");
        scanf("%f",&new->result.tensileStrength);
    }
    else if(strcmp(new->type,"Hardness")==0){
        printf("Enter the Hardness (HRC): ");
        scanf("%f",&new->result.hardness);
    }
    else if (strcmp(new->type,"Elongation")==0){
        printf("Enter Elongation : ");
        scanf("%f",&new->result.elongation);
    }
    else{
        printf("Invalid Test type! setting result to zero \n");
        new->result.tensileStrength =0;
    }

    printf("Test added successfully! \n");
}




void display(Test *test,int count){
    if(count ==0){
        printf("\nNo test available\n");
        return;
    }
    printf("\n------Test Records-------\n");
    for(int i=0;i<count;i++){
        printf("Test Id: %d\n",test[i].testID);
        printf("Test Type: %s\n",test[i].type);

        if(strcmp(test[i].type,"Tensile")==0){
            printf("Tensile Strength : %.2f MPa\n",test[i].result.tensileStrength);
        }
        else if (strcmp(test[i].type,"Hardness")==0){
            printf("Hardness: %.2f HRc\n",test[i].result.hardness);
        }
        else if(strcmp(test[i].type,"Elongation")==0){
```

```
            printf("Elongation : %.2f%%\n",test[i].result.elongation);
        }
        printf("-------------------------------------\n");

    }
}
```

4. Metal Fatigue Analysis
Description:
Develop a program to analyze metal fatigue using arrays for stress cycle data, structures for material details, and strings for material names.
Specifications:
Structure: Contains material ID, name, and endurance limit.
Array: Stress cycle data.
Strings: Material names.
const Pointers: Protect material details.
Double Pointers: Allocate dynamic material test data.

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

typedef struct{
    int id;
    char name[50];
    int limit;
    const char *details;
}Material;

void add(Material **material,int *count);
void display(Material *material,int count);

int main(){
    Material *material = NULL;
    int count =0,choice;
```

```
    while(1){
        printf("\n---Metal Fatigue Analysis---\n");
        printf("1. add\n");
        printf("2. Display\n");
        printf("3. Exit \n");
        printf("Enter your choice: ");
        scanf("%d",&choice);

        switch (choice){
            case 1:
                add(&material,&count);
                break;
            case 2:
                display(material, count);
                break;
            case 3:
                printf("Exiting.....\n");
                free(material);
                exit(0);
                break;
            default:
                printf("Invalid Input\n");
        }
    }
    return 0;
}

void add(Material **material,int *count){
    (*count)++;
    *material = (Material *)realloc (*material,(*count) * sizeof(Material));

    if(*material == NULL){
        printf("Memory allocation falied\n");
        return;
    }
    Material *new = &(*material)[*count-1];
    printf("Enter Id: ");
    scanf("%d",&new->id);
    printf("Enter Material Name: ");
    scanf(" %[^\n]s",new->name);
```

```
    new->details = "Good Quality materials";
    printf("Enter the limit: ");
    scanf("%d",&new->limit);
    printf("Material Added sucessfully\n");
    printf("---------------------------\n");

}

void display(Material *material,int count){
    if(count ==0){
        printf("No material to disply\n");
        return;
    }
    printf("\n-- Material Analysis ----\n");
    for(int i=0;i<count;i++){
        printf("Material ID: %d\n",material[i].id);
        printf("Material Name : %s\n",material[i].name);
        printf("Details: %s\n",material[i].details);
        printf("Limit: %d\n",material[i].limit);
    }
}
```

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

union Mold{
    char dimensions[50];
```

```
    char termalConductivity[50];
};
typedef struct{
    int id;
    float weight;
    char material[50];
    const char *details;
    char type[50];
    union Mold mold;
}Stores;

void add(Stores **store,int *count);
void display(Stores *store,int count);

int main(){
    Stores *store = NULL;
    int count =0,choice;

    while(1){
        printf("\n--Foundry Management System---\n");
        printf("1. Add \n");
        printf("2. Display\n");
        printf("3. Exit\n");
        printf("Enter a Choice: ");
        scanf("%d",&choice);

        switch (choice)
        {
        case 1:
            add(&store, &count);
            break;
        case 2:
            display(store,count);
            break;
        case 3:
            printf("Exiting.....\n");
            free(store);
            exit(0);
            break;
        default:
            printf("Invalid Choice \n");
```

```
            break;
        }
    }
    return 0;
}

void add(Stores **store,int *count){
    (*count)++;
    *store = (Stores *) realloc(*store,(*count) * sizeof(Stores));
    if(*store ==NULL){
        printf("Memory allocation failed\n");
    }
    Stores *new = &(*store)[*count-1];
    printf("Enter ID: ");
    scanf("%d",&new->id);
    printf("Enter Matirial Name: ");
    scanf(" %[^\n]s",new->material);
    printf("Weight: ");
    scanf("%f",&new->weight);
    new->details ="Expensive Materials";
    printf("Type (Thermal Conductivity or Dimension): ");
    scanf(" %[^\n]s",new->type);

    if(strcmp(new->type,"Dimensions")==0){
        printf("Enter the dimensions (length X WidthXHeight): ");
        scanf(" %[^\n]s",new->mold.dimensions);
    }else if(strcmp(new->type,"Thermal Conductivity")==0){
        printf("Enter the Thermal conductivity: ");
        scanf(" %[^\n]s",new->mold.termalConductivity);
    }
    else{
        printf("Invalid");
        return;
    }
     printf("Material Added Successfully!\n");
    printf("----------------------------\n");

}
void display(Stores *store,int count){
    if (count == 0) {
        printf("No records to display.\n");
```

```
        return;
    }
    printf("\n-- Foundry Records --\n");
    for (int i = 0; i < count; i++) {
        printf("ID: %d\n", store[i].id);
        printf("Material: %s\n", store[i].material);
        printf("Weight: %.2f\n", store[i].weight);
        printf("Details: %s\n", store[i].details);
        printf("Type: %s\n", store[i].type);

        if (strcmp(store[i].type, "Dimensions") == 0) {
            printf("Dimensions: %s\n", store[i].mold.dimensions);
        } else if (strcmp(store[i].type, "Thermal Conductivity") == 0) {
            printf("Thermal Conductivity: %s\n", store[i].mold.termalConductivity);
        }
        printf("----------------------------\n");
    }
}
```

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

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

union ImpurityType {
    char traceElements[50];
    char oxides[50];
};
```

```
typedef struct {
    int sampleID;
    char type[50];
    const float *purity;
    union ImpurityType impurityType;
    float *impurityPercentages;
    int impurityCount;
} Sample;

void addSample(Sample **samples, int *count);
void displaySamples(Sample *samples, int count);

int main() {
    Sample *samples = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Metal Purity Analysis System --\n");
        printf("1. Add Sample\n");
        printf("2. Display Samples\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addSample(&samples, &count);
            break;
        case 2:
            displaySamples(samples, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(samples);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
```

```
}

void addSample(Sample **samples, int *count) {
    (*count)++;
    *samples = (Sample *)realloc(*samples, (*count) * sizeof(Sample));
    if (*samples == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    Sample *newSample = &(*samples)[*count - 1];
    printf("Enter Sample ID: ");
    scanf("%d", &newSample->sampleID);
    printf("Enter Sample Type: ");
    scanf(" %[^\n]s", newSample->type);
    printf("Enter purity: ");
    scanf("%f", &newSample->purity);
    printf("Enter impurity type (1 for trace elements, 2 for oxides): ");
    int impurityChoice;
    scanf("%d", &impurityChoice);
    if (impurityChoice == 1) {
        printf("Enter trace elements: ");
        scanf(" %[^\n]s", newSample->impurityType.traceElements);
    } else {
        printf("Enter oxides: ");
        scanf(" %[^\n]s", newSample->impurityType.oxides);
    }
    printf("Enter number of impurities: ");
    scanf("%d", &newSample->impurityCount);
    newSample->impurityPercentages = (float *)malloc(newSample->impurityCount *
sizeof(float));
    for (int i = 0; i < newSample->impurityCount; i++) {
        printf("Enter impurity percentage %d: ", i + 1);
        scanf("%f", &newSample->impurityPercentages[i]);
    }
    printf("Sample added successfully\n");
}

void displaySamples(Sample *samples, int count) {
    if (count == 0) {
        printf("No samples available\n");
        return;
```

```
    }
    printf("\n-- Sample Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Sample ID: %d\n", samples[i].sampleID);
        printf("Sample Type: %s\n", samples[i].type);
        printf("Purity: %.2f\n", *(samples[i].purity));
        if (samples[i].impurityPercentages != NULL) {
            printf("Impurity Type: ");
            if (samples[i].impurityType.traceElements[0] != '\0') {
                printf("Trace Elements: %s\n", samples[i].impurityType.traceElements);
            } else {
                printf("Oxides: %s\n", samples[i].impurityType.oxides);
            }
            printf("Impurity Percentages: ");
            for (int j = 0; j < samples[i].impurityCount; j++) {
                printf("%.2f%% ", samples[i].impurityPercentages[j]);
            }
            printf("\n");
        }
        printf("----------------------------\n");
    }
}
```

7. Corrosion Testing System
Description:
Create a program to track corrosion tests using structures for test details, arrays for test results, and strings for test conditions.
Specifications:
Structure: Holds test ID, duration, and environment.
Array: Test results.
Strings: Test conditions.
const Pointers: Protect test configurations.
Double Pointers: Dynamic allocation of test records.

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

```
typedef struct {
    int testID;
    float duration;
    char environment[50];
    const char *testConditions;
    float *testResults;
    int resultCount;
} CorrosionTest;

void addTest(CorrosionTest **tests, int *count);
void displayTests(CorrosionTest *tests, int count);

int main() {
    CorrosionTest *tests = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Corrosion Testing System --\n");
        printf("1. Add Test\n");
        printf("2. Display Tests\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addTest(&tests, &count);
            break;
        case 2:
            displayTests(tests, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(tests);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}
```

```
void addTest(CorrosionTest **tests, int *count) {
    (*count)++;
    *tests = (CorrosionTest *)realloc(*tests, (*count) * sizeof(CorrosionTest));
    if (*tests == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    CorrosionTest *newTest = &(*tests)[*count - 1];
    printf("Enter Test ID: ");
    scanf("%d", &newTest->testID);
    printf("Enter Duration: ");
    scanf("%f", &newTest->duration);
    printf("Enter Environment: ");
    scanf(" %[^\n]s", newTest->environment);
    printf("Enter Test Conditions: ");
    newTest->testConditions = "Standard Corrosion Conditions";
    printf("Enter number of results: ");
    scanf("%d", &newTest->resultCount);
    newTest->testResults = (float *)malloc(newTest->resultCount * sizeof(float));
    for (int i = 0; i < newTest->resultCount; i++) {
        printf("Enter result %d: ", i + 1);
        scanf("%f", &newTest->testResults[i]);
    }
    printf("Test added successfully\n");
}

void displayTests(CorrosionTest *tests, int count) {
    if (count == 0) {
        printf("No tests available\n");
        return;
    }
    printf("\n-- Test Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Test ID: %d\n", tests[i].testID);
        printf("Duration: %.2f hours\n", tests[i].duration);
        printf("Environment: %s\n", tests[i].environment);
        printf("Test Conditions: %s\n", tests[i].testConditions);
        printf("Test Results: ");
        for (int j = 0; j < tests[i].resultCount; j++) {
            printf("%.2f ", tests[i].testResults[j]);
```

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

8. Welding Parameter Optimization
Description:
Develop a program to optimize welding parameters using structures for parameter sets, arrays for test outcomes, and unions for variable welding types.
Specifications:
Structure: Stores parameter ID, voltage, current, and speed.
Union: Represents welding types (MIG, TIG, or Arc).
Array: Test outcomes.
const Pointers: Protect parameter configurations.
Double Pointers: Manage dynamic parameter sets.

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

union WeldingType {
    char MIG[20];
    char TIG[20];
    char Arc[20];
};

typedef struct {
    int parameterID;
    float voltage;
    float current;
    float speed;
    const char *parameterConfig;
    union WeldingType weldingType;
    float *testOutcomes;
    int outcomeCount;
} WeldingParameter;

void addWeldingParameter(WeldingParameter **parameters, int *count);
```

```
void displayWeldingParameters(WeldingParameter *parameters, int count);

int main() {
    WeldingParameter *parameters = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Welding Parameter Optimization --\n");
        printf("1. Add Welding Parameter\n");
        printf("2. Display Welding Parameters\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addWeldingParameter(&parameters, &count);
            break;
        case 2:
            displayWeldingParameters(parameters, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(parameters);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addWeldingParameter(WeldingParameter **parameters, int *count) {
    (*count)++;
    *parameters = (WeldingParameter *)realloc(*parameters, (*count) *
sizeof(WeldingParameter));
    if (*parameters == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    WeldingParameter *newParam = &(*parameters)[*count - 1];
```

```
    printf("Enter Parameter ID: ");
    scanf("%d", &newParam->parameterID);
    printf("Enter Voltage: ");
    scanf("%f", &newParam->voltage);
    printf("Enter Current: ");
    scanf("%f", &newParam->current);
    printf("Enter Speed: ");
    scanf("%f", &newParam->speed);
    newParam->parameterConfig = "Optimized Welding Settings";
    printf("Enter Welding Type (MIG, TIG, Arc): ");
    char weldingType[20];
    scanf("%s", weldingType);
    if (strcmp(weldingType, "MIG") == 0) {
        strcpy(newParam->weldingType.MIG, "MIG");
    } else if (strcmp(weldingType, "TIG") == 0) {
        strcpy(newParam->weldingType.TIG, "TIG");
    } else if (strcmp(weldingType, "Arc") == 0) {
        strcpy(newParam->weldingType.Arc, "Arc");
    } else {
        printf("Invalid Welding Type\n");
        return;
    }
    printf("Enter number of test outcomes: ");
    scanf("%d", &newParam->outcomeCount);
    newParam->testOutcomes = (float *)malloc(newParam->outcomeCount *
sizeof(float));
    for (int i = 0; i < newParam->outcomeCount; i++) {
        printf("Enter test outcome %d: ", i + 1);
        scanf("%f", &newParam->testOutcomes[i]);
    }
    printf("Welding Parameter added successfully\n");
}

void displayWeldingParameters(WeldingParameter *parameters, int count) {
    if (count == 0) {
        printf("No welding parameters available\n");
        return;
    }
    printf("\n-- Welding Parameter Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Parameter ID: %d\n", parameters[i].parameterID);
```

```
        printf("Voltage: %.2fV\n", parameters[i].voltage);
        printf("Current: %.2fA\n", parameters[i].current);
        printf("Speed: %.2f m/s\n", parameters[i].speed);
        printf("Parameter Configuration: %s\n", parameters[i].parameterConfig);
        printf("Welding Type: ");
        if (strlen(parameters[i].weldingType.MIG) > 0) {
            printf("%s\n", parameters[i].weldingType.MIG);
        } else if (strlen(parameters[i].weldingType.TIG) > 0) {
            printf("%s\n", parameters[i].weldingType.TIG);
        } else {
            printf("%s\n", parameters[i].weldingType.Arc);
        }
        printf("Test Outcomes: ");
        for (int j = 0; j < parameters[i].outcomeCount; j++) {
            printf("%.2f ", parameters[i].testOutcomes[j]);
        }
        printf("\n----------------------------\n");
    }
}
```

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

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

typedef struct {
    int configID;
    char material[50];
    const char *measurementUnits;
    const char *configDetails;
```

```
} SurfaceConfig;

void addSurfaceConfig(SurfaceConfig **configs, int *count);
void displaySurfaceConfigs(SurfaceConfig *configs, int count);

int main() {
    SurfaceConfig *configs = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Metal Surface Finish Analysis --\n");
        printf("1. Add Surface Finish Configuration\n");
        printf("2. Display Surface Finish Configurations\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addSurfaceConfig(&configs, &count);
            break;
        case 2:
            displaySurfaceConfigs(configs, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(configs);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addSurfaceConfig(SurfaceConfig **configs, int *count) {
    (*count)++;
    *configs = (SurfaceConfig *)realloc(*configs, (*count) * sizeof(SurfaceConfig));
    if (*configs == NULL) {
        printf("Memory allocation failed\n");
        return;
```

```
    }
    SurfaceConfig *newConfig = &(*configs)[*count - 1];
    printf("Enter Configuration ID: ");
    scanf("%d", &newConfig->configID);
    printf("Enter Material: ");
    scanf(" %[^\n]s", newConfig->material);
    printf("Enter Measurement Units: ");
    scanf(" %[^\n]s", newConfig->measurementUnits);
    newConfig->configDetails = "Surface Finish Measurement Configuration";
    printf("Surface Finish Configuration added successfully\n");
}

void displaySurfaceConfigs(SurfaceConfig *configs, int count) {
    if (count == 0) {
        printf("No surface finish configurations available\n");
        return;
    }
    printf("\n-- Surface Finish Configuration Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Configuration ID: %d\n", configs[i].configID);
        printf("Material: %s\n", configs[i].material);
        printf("Measurement Units: %s\n", configs[i].measurementUnits);
        printf("Configuration Details: %s\n", configs[i].configDetails);
        printf("-----------------------------\n");
    }
}
```

10. Smelting Process Tracker
Description:
Create a system to track smelting processes using structures for process metadata, arrays for heat data, and unions for variable ore properties.
Specifications:
Structure: Holds process ID, ore type, and temperature.
Union: Represents variable ore properties.
Array: Heat data.
const Pointers: Protect process metadata.
Double Pointers: Allocate dynamic process records.

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

```
#include <string.h>

union OreProperties {
    char mineralType[50];
    char impurityLevel[50];
};

typedef struct {
    int processID;
    char oreType[50];
    float temperature;
    const char *processDetails;
    union OreProperties oreProperties;
} SmeltingProcess;

void addSmeltingProcess(SmeltingProcess **processes, int *count);
void displaySmeltingProcesses(SmeltingProcess *processes, int count);

int main() {
    SmeltingProcess *processes = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Smelting Process Tracker --\n");
        printf("1. Add Smelting Process\n");
        printf("2. Display Smelting Processes\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addSmeltingProcess(&processes, &count);
            break;
        case 2:
            displaySmeltingProcesses(processes, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(processes);
            exit(0);
```

```
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addSmeltingProcess(SmeltingProcess **processes, int *count) {
    (*count)++;
    *processes = (SmeltingProcess *)realloc(*processes, (*count) *
sizeof(SmeltingProcess));
    if (*processes == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    SmeltingProcess *newProcess = &(*processes)[*count - 1];
    printf("Enter Process ID: ");
    scanf("%d", &newProcess->processID);
    printf("Enter Ore Type: ");
    scanf(" %[^\n]s", newProcess->oreType);
    printf("Enter Temperature: ");
    scanf("%f", &newProcess->temperature);
    newProcess->processDetails = "Smelting Process with Variable Ore Properties";
    printf("Enter Ore Property (Mineral Type or Impurity Level): ");
    scanf(" %[^\n]s", newProcess->oreProperties.mineralType); // Can also use
impurityLevel if needed
    printf("Smelting Process Added Successfully\n");
}

void displaySmeltingProcesses(SmeltingProcess *processes, int count) {
    if (count == 0) {
        printf("No smelting processes to display\n");
        return;
    }
    printf("\n-- Smelting Process Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Process ID: %d\n", processes[i].processID);
        printf("Ore Type: %s\n", processes[i].oreType);
        printf("Temperature: %.2f\n", processes[i].temperature);
        printf("Process Details: %s\n", processes[i].processDetails);
        printf("Ore Property (Mineral Type or Impurity Level): %s\n",
```

```
processes[i].oreProperties.mineralType);
        printf("---------------------------\n");
    }
}
```

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

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

typedef struct {
    char ionType[50];
    float charge;
    float concentration;
    const char *electrolyte;
} Electroplating;

void addElectroplating(Electroplating **plating, int *count);
void displayElectroplating(Electroplating *plating, int count);

int main() {
    Electroplating *plating = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Electroplating System Simulation --\n");
        printf("1. Add Plating Configuration\n");
        printf("2. Display Plating Configurations\n");
```

```
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addElectroplating(&plating, &count);
            break;
        case 2:
            displayElectroplating(plating, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(plating);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addElectroplating(Electroplating **plating, int *count) {
    (*count)++;
    *plating = (Electroplating *)realloc(*plating, (*count) * sizeof(Electroplating));
    if (*plating == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    Electroplating *newPlating = &(*plating)[*count - 1];
    printf("Enter Ion Type: ");
    scanf(" %[^\n]s", newPlating->ionType);
    printf("Enter Charge: ");
    scanf("%f", &newPlating->charge);
    printf("Enter Concentration: ");
    scanf("%f", &newPlating->concentration);
    newPlating->electrolyte = "Acidic Electrolyte";

    printf("Plating Configuration Added Successfully\n");
}
```

```
void displayElectroplating(Electroplating *plating, int count) {
    if (count == 0) {
        printf("No plating configurations to display\n");
        return;
    }
    printf("\n-- Plating Configuration Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Ion Type: %s\n", plating[i].ionType);
        printf("Charge: %.2f\n", plating[i].charge);
        printf("Concentration: %.2f\n", plating[i].concentration);
        printf("Electrolyte: %s\n", plating[i].electrolyte);
        printf("---------------------------\n");
    }
}
```

12. Casting Defect Analysis
Description:
Design a system to analyze casting defects using arrays for defect data, structures for casting details, and unions for variable defect types.
Specifications:
Structure: Holds casting ID, material, and dimensions.
Union: Represents defect types (shrinkage or porosity).
Array: Defect data.
const Pointers: Protect casting data.
Double Pointers: Dynamic defect record management.

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

union DefectType {
    char shrinkage[50];
    char porosity[50];
};

typedef struct {
    int castingID;
    char material[50];
```

```
    char dimensions[50];
    const char *details;
    union DefectType defect;
} Casting;

void addCasting(Casting **casting, int *count);
void displayCasting(Casting *casting, int count);

int main() {
    Casting *casting = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Casting Defect Analysis --\n");
        printf("1. Add Casting Defect\n");
        printf("2. Display Castings\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addCasting(&casting, &count);
            break;
        case 2:
            displayCasting(casting, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(casting);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addCasting(Casting **casting, int *count) {
    (*count)++;
    *casting = (Casting *)realloc(*casting, (*count) * sizeof(Casting));
```

```
    if (*casting == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    Casting *newCasting = &(*casting)[*count - 1];
    printf("Enter Casting ID: ");
    scanf("%d", &newCasting->castingID);
    printf("Enter Material: ");
    scanf(" %[^\n]s", newCasting->material);
    printf("Enter Dimensions: ");
    scanf(" %[^\n]s", newCasting->dimensions);
    newCasting->details = "Casting Defect Analysis";
    printf("Enter Defect Type (Shrinkage or Porosity): ");
    scanf(" %[^\n]s", newCasting->defect.shrinkage);

    printf("Casting Defect Added Successfully\n");
}

void displayCasting(Casting *casting, int count) {
    if (count == 0) {
        printf("No castings to display\n");
        return;
    }
    printf("\n-- Casting Defect Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Casting ID: %d\n", casting[i].castingID);
        printf("Material: %s\n", casting[i].material);
        printf("Dimensions: %s\n", casting[i].dimensions);
        printf("Defect: %s\n", casting[i].defect.shrinkage);
        printf("---------------------------\n");
    }
}
```

13. Metallurgical Lab Automation

Description:

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

Specifications:

Structure: Contains sample ID, type, and dimensions.

Array: Test results.
Strings: Equipment names.
const Pointers: Protect sample details.
Double Pointers: Allocate and manage dynamic test records.

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

typedef struct {
    int sampleID;
    char sampleType[50];
    char dimensions[50];
    const char *equipment;
} Sample;

void addSample(Sample **samples, int *count);
void displaySamples(Sample *samples, int count);

int main() {
    Sample *samples = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Metallurgical Lab Automation --\n");
        printf("1. Add Sample\n");
        printf("2. Display Samples\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addSample(&samples, &count);
            break;
        case 2:
            displaySamples(samples, count);
            break;
        case 3:
            printf("Exiting...\n");
```

```
            free(samples);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addSample(Sample **samples, int *count) {
    (*count)++;
    *samples = (Sample *)realloc(*samples, (*count) * sizeof(Sample));
    if (*samples == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    Sample *newSample = &(*samples)[*count - 1];
    printf("Enter Sample ID: ");
    scanf("%d", &newSample->sampleID);
    printf("Enter Sample Type: ");
    scanf(" %[^\n]s", newSample->sampleType);
    printf("Enter Dimensions: ");
    scanf(" %[^\n]s", newSample->dimensions);
    newSample->equipment = "X-ray Diffraction";

    printf("Sample Added Successfully\n");
}

void displaySamples(Sample *samples, int count) {
    if (count == 0) {
        printf("No samples to display\n");
        return;
    }
    printf("\n-- Sample Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Sample ID: %d\n", samples[i].sampleID);
        printf("Sample Type: %s\n", samples[i].sampleType);
        printf("Dimensions: %s\n", samples[i].dimensions);
        printf("Equipment Used: %s\n", samples[i].equipment);
        printf("---------------------------\n");
```

```
    }
}
```

14. Metal Hardness Testing System
Description:
Develop a program to track metal hardness tests using structures for test data, arrays for hardness values, and unions for variable hardness scales.
Specifications:
Structure: Stores test ID, method, and result.
Union: Represents variable hardness scales (Rockwell or Brinell).
Array: Hardness values.
const Pointers: Protect test data.
Double Pointers: Dynamic hardness record allocation.

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

union HardnessScale {
    char rockwell[50];
    char brinell[50];
};

typedef struct {
    int testID;
    char method[50];
    float result;
    union HardnessScale hardness;
} HardnessTest;

void addHardnessTest(HardnessTest **tests, int *count);
void displayHardnessTests(HardnessTest *tests, int count);

int main() {
    HardnessTest *tests = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n-- Metal Hardness Testing System --\n");
        printf("1. Add Hardness Test\n");
        printf("2. Display Hardness Tests\n");
```

```
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addHardnessTest(&tests, &count);
            break;
        case 2:
            displayHardnessTests(tests, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(tests);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addHardnessTest(HardnessTest **tests, int *count) {
    (*count)++;
    *tests = (HardnessTest *)realloc(*tests, (*count) * sizeof(HardnessTest));
    if (*tests == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    HardnessTest *newTest = &(*tests)[*count - 1];
    printf("Enter Test ID: ");
    scanf("%d", &newTest->testID);
    printf("Enter Method: ");
    scanf(" %[^\n]s", newTest->method);
    printf("Enter Result: ");
    scanf("%f", &newTest->result);
    printf("Enter Hardness Scale (Rockwell or Brinell): ");
    scanf(" %[^\n]s", newTest->hardness.rockwell);

    printf("Hardness Test Added Successfully\n");
```

```
}

void displayHardnessTests(HardnessTest *tests, int count) {
    if (count == 0) {
        printf("No hardness tests to display\n");
        return;
    }
    printf("\n-- Hardness Test Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Test ID: %d\n", tests[i].testID);
        printf("Method: %s\n", tests[i].method);
        printf("Result: %.2f\n", tests[i].result);
        printf("Hardness Scale: %s\n", tests[i].hardness.rockwell);
        printf("---------------------------\n");
    }
}
```

15. Powder Metallurgy Process Tracker
Description:
Create a program to track powder metallurgy processes using structures for material details, arrays for particle size distribution, and unions for variable powder properties.
Specifications:
Structure: Contains material ID, type, and density.
Union: Represents powder properties.
Array: Particle size distribution data.
const Pointers: Protect material configurations.
Double Pointers: Allocate and manage powder data.

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

union PowderProperties {
    float particleSize;
    float density;
};

typedef struct {
    int materialID;
    char materialType[50];
```

```
    float materialDensity;
    union PowderProperties powder;
} PowderMaterial;

void addPowderMaterial(PowderMaterial **materials, int *count);
void displayPowderMaterials(PowderMaterial *materials, int count);

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

    while (1) {
        printf("\n-- Powder Metallurgy Process Tracker --\n");
        printf("1. Add Powder Material\n");
        printf("2. Display Powder Materials\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addPowderMaterial(&materials, &count);
            break;
        case 2:
            displayPowderMaterials(materials, count);
            break;
        case 3:
            printf("Exiting...\n");
            free(materials);
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addPowderMaterial(PowderMaterial **materials, int *count) {
    (*count)++;
    *materials = (PowderMaterial *)realloc(*materials, (*count) *
sizeof(PowderMaterial));
```

```
    if (*materials == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    PowderMaterial *newMaterial = &(*materials)[*count - 1];
    printf("Enter Material ID: ");
    scanf("%d", &newMaterial->materialID);
    printf("Enter Material Type: ");
    scanf(" %[^\n]s", newMaterial->materialType);
    printf("Enter Material Density: ");
    scanf("%f", &newMaterial->materialDensity);
    printf("Enter Powder Particle Size: ");
    scanf("%f", &newMaterial->powder.particleSize);
    printf("Powder Material Added Successfully\n");
}

void displayPowderMaterials(PowderMaterial *materials, int count) {
    if (count == 0) {
        printf("No powder materials to display\n");
        return;
    }
    printf("\n-- Powder Material Details --\n");
    for (int i = 0; i < count; i++) {
        printf("Material ID: %d\n", materials[i].materialID);
        printf("Material Type: %s\n", materials[i].materialType);
        printf("Density: %.2f\n", materials[i].materialDensity);
        printf("Powder Particle Size: %.2f\n", materials[i].powder.particleSize);
        printf("----------------------------\n");
    }
}
```

16. Metal Recycling Analysis
Description:
Develop a program to analyze recycled metal data using structures for material details, arrays for impurity levels, and strings for recycling methods.
Specifications:
Structure: Holds material ID, type, and recycling method.
Array: Impurity levels.
Strings: Recycling methods.
const Pointers: Protect material details.

Double Pointers: Allocate dynamic recycling records.

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

#define MAX_IMPURITY_LEVELS 10


typedef struct {
    const int materialID;
    char materialType[50];
    char recyclingMethod[50];
} MaterialDetails;


void addMaterial(MaterialDetails **materials, int *count);
void addImpurityLevels(float ***impurityLevels, int count);
void displayMaterials(const MaterialDetails *materials, float **impurityLevels, int
count);

int main() {
    MaterialDetails *materials = NULL; // Array of material structures
    float **impurityLevels = NULL;    // Double pointer for impurity levels
    int materialCount = 0, choice;

    while (1) {
        printf("\n-- Metal Recycling Analysis --\n");
        printf("1. Add Material\n");
        printf("2. Add Impurity Levels\n");
        printf("3. Display Materials and Impurity Data\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            addMaterial(&materials, &materialCount);
            break;
        case 2:
```

```
            addImpurityLevels(&impurityLevels, materialCount);
            break;
        case 3:
            displayMaterials(materials, impurityLevels, materialCount);
            break;
        case 4:
            printf("Exiting...\n");
            free(materials);
            if (impurityLevels) {
                for (int i = 0; i < materialCount; i++) {
                    free(impurityLevels[i]);
                }
                free(impurityLevels);
            }
            return 0;
        default:
            printf("Invalid choice\n");
        }
    }
    return 0;
}

void addMaterial(MaterialDetails **materials, int *count) {
    (*count)++;
    *materials = (MaterialDetails *)realloc(*materials, (*count) * sizeof(MaterialDetails));
    if (*materials == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    MaterialDetails *newMaterial = &(*materials)[*count - 1];
    int id;
    printf("Enter Material ID: ");
    scanf("%d", &id);
    *(int *)&newMaterial->materialID = id;
    printf("Enter Material Type: ");
    scanf(" %[^\n]s", newMaterial->materialType);
    printf("Enter Recycling Method: ");
    scanf(" %[^\n]s", newMaterial->recyclingMethod);

    printf("Material Added Successfully\n");
```

```
}

void addImpurityLevels(float ***impurityLevels, int count) {
    if (count == 0) {
        printf("No materials to assign impurity levels\n");
        return;
    }

    *impurityLevels = (float **)realloc(*impurityLevels, count * sizeof(float *));
    if (*impurityLevels == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    for (int i = 0; i < count; i++) {
        (*impurityLevels)[i] = (float *)malloc(MAX_IMPURITY_LEVELS *
sizeof(float));
        if ((*impurityLevels)[i] == NULL) {
            printf("Memory allocation failed for material %d\n", i + 1);
            return;
        }

        printf("Enter impurity levels for Material %d (up to %d values, -1 to stop):\n", i + 1,
MAX_IMPURITY_LEVELS);
        for (int j = 0; j < MAX_IMPURITY_LEVELS; j++) {
            float level;
            printf("Impurity Level %d: ", j + 1);
            scanf("%f", &level);
            if (level == -1) {
                (*impurityLevels)[i][j] = -1;
                break;
            }
            (*impurityLevels)[i][j] = level;
        }
    }
    printf("Impurity Levels Added Successfully\n");
}

void displayMaterials(const MaterialDetails *materials, float **impurityLevels, int
count) {
    if (count == 0) {
```

```
        printf("No materials to display\n");
        return;
    }

    printf("\n-- Material and Impurity Data --\n");
    for (int i = 0; i < count; i++) {
        printf("Material ID: %d\n", materials[i].materialID);
        printf("Material Type: %s\n", materials[i].materialType);
        printf("Recycling Method: %s\n", materials[i].recyclingMethod);

        printf("Impurity Levels: ");
        if (impurityLevels && impurityLevels[i]) {
            for (int j = 0; j < MAX_IMPURITY_LEVELS; j++) {
                if (impurityLevels[i][j] == -1) break;
                printf("%.2f ", impurityLevels[i][j]);
            }
        }
        printf("\n---------------------------\n");
    }
}
```

17. Rolling Mill Performance Tracker
Description:
Design a system to track rolling mill performance using structures for mill configurations, arrays for output data, and strings for material types.
Specifications:
Structure: Stores mill ID, roll diameter, and speed.
Array: Output data.
Strings: Material types.
const Pointers: Protect mill configurations.
Double Pointers: Manage rolling mill records dynamically.

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

typedef struct {
    int millID;
    float rollDiameter;
    float speed;
```

```
} MillConfig;

void displayMillConfig(const MillConfig *config) {
    printf("Mill ID: %d, Roll Diameter: %.2f, Speed: %.2f\n",
           config->millID, config->rollDiameter, config->speed);
}

int main() {
    float outputData[5] = {100.5, 200.0, 150.2, 180.6, 170.9};
    char materials[3][20] = {"Steel", "Aluminum", "Copper"};
    MillConfig config = {101, 45.0, 120.0};
    const MillConfig *configPtr = &config;

    displayMillConfig(configPtr);

    MillConfig **millRecords = malloc(2 * sizeof(MillConfig *));
    for (int i = 0; i < 2; i++) {
        millRecords[i] = malloc(sizeof(MillConfig));
        millRecords[i]->millID = 100 + i;
        millRecords[i]->rollDiameter = 50.0 + i;
        millRecords[i]->speed = 110.0 + i;
    }

    for (int i = 0; i < 2; i++) {
        displayMillConfig(millRecords[i]);
        free(millRecords[i]);
    }

    free(millRecords);

    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>

typedef struct {
    int materialID;
    char type[20];
    float expansionCoefficient;
} Material;

typedef union {
    float constantValue;
    float variableCoefficient;
} ExpansionCoefficient;

void displayMaterial(const Material *mat) {
    printf("Material ID: %d, Type: %s, Expansion Coefficient: %.2f\n",
           mat->materialID, mat->type, mat->expansionCoefficient);
}

int main() {
    float temperatureData[5] = {25.0, 50.0, 75.0, 100.0, 125.0};
    Material mat = {101, "Steel", 0.000012};
    const Material *matPtr = &mat;

    displayMaterial(matPtr);

    Material **thermalRecords = malloc(3 * sizeof(Material *));
    for (int i = 0; i < 3; i++) {
        thermalRecords[i] = malloc(sizeof(Material));
        thermalRecords[i]->materialID = 200 + i;
        sprintf(thermalRecords[i]->type, "Material%d", i);
        thermalRecords[i]->expansionCoefficient = 0.00001 * (i + 1);
    }
```

```
    for (int i = 0; i < 3; i++) {
        displayMaterial(thermalRecords[i]);
        free(thermalRecords[i]);
    }

    free(thermalRecords);

    return 0;
}
```

19. Metal Melting Point Analyzer
Description:
Develop a program to analyze melting points using structures for metal details, arrays for temperature data, and strings for metal names.
Specifications:
Structure: Stores metal ID, name, and melting point.
Array: Temperature data.
Strings: Metal names.
const Pointers: Protect metal details.
Double Pointers: Allocate dynamic melting point records.

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

typedef struct {
    int metalID;
    char name[20];
    float meltingPoint;
} Metal;

void displayMetal(const Metal *metal) {
    printf("Metal ID: %d, Name: %s, Melting Point: %.2f\n",
           metal->metalID, metal->name, metal->meltingPoint);
}

int main() {
    float temperatureData[5] = {500.0, 600.0, 700.0, 800.0, 900.0};
    Metal metal = {1, "Iron", 1538.0};
    const Metal *metalPtr = &metal;
```

```
    displayMetal(metalPtr);

    Metal **metalRecords = malloc(3 * sizeof(Metal *));
    for (int i = 0; i < 3; i++) {
        metalRecords[i] = malloc(sizeof(Metal));
        metalRecords[i]->metalID = 100 + i;
        sprintf(metalRecords[i]->name, "Metal%d", i);
        metalRecords[i]->meltingPoint = 1500.0 + (i * 100);
    }

    for (int i = 0; i < 3; i++) {
        displayMetal(metalRecords[i]);
        free(metalRecords[i]);
    }

    free(metalRecords);

    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>

typedef struct {
    int processID;
    char oreType[20];
```

```
    float efficiency;
} SmeltingProcess;

typedef union {
    float energy;
    float duration;
} ProcessParameter;

void displayProcess(const SmeltingProcess *process) {
    printf("Process ID: %d, Ore Type: %s, Efficiency: %.2f\n",
           process->processID, process->oreType, process->efficiency);
}

int main() {
    float energyConsumption[5] = {150.5, 200.0, 175.3, 180.8, 165.7};
    SmeltingProcess process = {101, "Iron Ore", 90.5};
    const SmeltingProcess *processPtr = &process;

    displayProcess(processPtr);

    SmeltingProcess **processRecords = malloc(3 * sizeof(SmeltingProcess *));
    for (int i = 0; i < 3; i++) {
        processRecords[i] = malloc(sizeof(SmeltingProcess));
        processRecords[i]->processID = 300 + i;
        sprintf(processRecords[i]->oreType, "OreType%d", i);
        processRecords[i]->efficiency = 85.0 + (i * 5);
    }

    for (int i = 0; i < 3; i++) {
        displayProcess(processRecords[i]);
        free(processRecords[i]);
    }

    free(processRecords);

    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>
#include <string.h>

union Variables{
    float voltage;
    float current;
};
typedef struct{
    int id;
    char name[50];
    float voltage;
    float current;
    const char *details;
    union Variables var;
}Weld;

void add(Weld **weld,int *count);
void display(Weld *weld,int count);

int main() {
    Weld *weld = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Weld Configuration\n");
        printf("2. Display Weld Configurations\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                add(&weld, &count);
                break;
            case 2:
                display(weld, count);
                break;
            case 3:
                free(weld);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Weld **weld,int *count){
    (*count)++;
    *weld = realloc(*weld,(*count) * sizeof(Weld));

    if(*weld == NULL){
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Weld *new = &(*weld)[*count -1];

    printf("ENter Weld ID: ");
    scanf("%d",&new->id);
    printf("Enter Weld Name: ");
    scanf(" %[^\n]s",new->name);
    printf("Enter Voltage: ");
    scanf("%f",&new->voltage);
    printf("Enter Current: ");
    scanf("%f",&new->current);

    new->details ="Protected weld type details";
    printf("Sucessfully added \n");
}
void display(Weld *weld,int count){
    if(count ==0){
```

```
        printf("No weld to display\n");
        return;
    }
    for(int i=0;i<count;i++){
        printf("  ID: %d\n", weld[i].id);
        printf("  Name: %s\n", weld[i].name);
        printf("  Voltage: %.2f\n", weld[i].voltage);
        printf("  Current: %.2f\n", weld[i].current);
        printf("  Details: %s\n", weld[i].details);
    }
}
```

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>


typedef struct{
    int id;
    char mode[50];
    char voltage_range;
    float speed;
    const char *details;

}Weld;

void add(Weld **weld,int *count);
```

```
void display(Weld *weld,int count);

int main() {
    Weld *weld = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Weld Configuration\n");
        printf("2. Display Weld Configurations\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&weld, &count);
                break;
            case 2:
                display(weld, count);
                break;
            case 3:
                free(weld);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Weld **weld,int *count){
    (*count)++;
    *weld = realloc(*weld,(*count) * sizeof(Weld));

    if(*weld == NULL){
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Weld *new = &(*weld)[*count -1];

    printf("ENter Weld ID: ");
```

```
    scanf("%d",&new->id);
    printf("Enter Weld Mode: ");
    scanf(" %[^\n]s",new->mode);
    printf("Enter Voltage_range: ");
    scanf(" %[^\n]s",&new->voltage_range);
    printf("Enter Speed: ");
    scanf("%f",&new->speed);

    new->details ="Protected weld type details";
    printf("Sucessfully added \n");
}
void display(Weld *weld,int count){
    if(count ==0){
        printf("No weld to display\n");
        return;
    }
    for(int i=0;i<count;i++){
        printf("  ID: %d\n", weld[i].id);
        printf("  Mode: %s\n", weld[i].mode);
        printf("  Voltage Range: %.2f\n", weld[i].voltage_range);
        printf("  Current: %.2f\n", weld[i].speed);
        printf("  Details: %s\n", weld[i].details);
    }
}
```

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>
```

```
// Function prototypes
void weldingProcessTracker();

int main() {
    weldingProcessTracker();
    return 0;
}

void weldingProcessTracker() {
    struct WeldingProcess {
        int processID;
        char material[50];
        char welderName[50];
    };

    union ProcessMetrics {
        float heatInput;
        float arcLength;
    };

    struct WeldingProcess *processRecords;
    union ProcessMetrics metrics;
    int numProcesses, i;

    printf("Enter the number of welding processes: ");
    scanf("%d", &numProcesses);

    processRecords = (struct WeldingProcess *)malloc(numProcesses * sizeof(struct
WeldingProcess));

    for (i = 0; i < numProcesses; i++) {
        printf("Enter process ID, material, and welder name for process %d: ", i + 1);
        scanf("%d %s %s", &processRecords[i].processID, processRecords[i].material,
processRecords[i].welderName);

        printf("Enter heat input for process %d: ", i + 1);
        scanf("%f", &metrics.heatInput);
        printf("Heat Input: %.2f\n", metrics.heatInput);

        printf("Enter arc length for process %d: ", i + 1);
```

```
        scanf("%f", &metrics.arcLength);
        printf("Arc Length: %.2f\n", metrics.arcLength);
    }

    for (i = 0; i < numProcesses; i++) {
        printf("Process ID: %d, Material: %s, Welder Name: %s\n",
               processRecords[i].processID, processRecords[i].material,
processRecords[i].welderName);
    }

    free(processRecords);
}
```

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>

// Function prototypes
void weldBeadGeometryAnalyzer();
void weldingConsumableInventorySystem();

int main() {
    weldBeadGeometryAnalyzer();
    weldingConsumableInventorySystem();
    return 0;
}
```

```
void weldBeadGeometryAnalyzer() {
    struct BeadGeometry {
        int beadID;
        char material[50];
        char geometryType[50];
    };

    union GeometryParameters {
        float width;
        float depth;
        float height;
    };

    struct BeadGeometry *beadRecords;
    union GeometryParameters parameters;
    const int numBeads = 3;
    int i;

    beadRecords = (struct BeadGeometry *)malloc(numBeads * sizeof(struct
BeadGeometry));

    for (i = 0; i < numBeads; i++) {
        printf("Enter bead ID, material, and geometry type for bead %d: ", i + 1);
        scanf("%d %s %s", &beadRecords[i].beadID, beadRecords[i].material,
beadRecords[i].geometryType);

        printf("Enter width for bead %d: ", i + 1);
        scanf("%f", &parameters.width);
        printf("Width: %.2f\n", parameters.width);

        printf("Enter depth for bead %d: ", i + 1);
        scanf("%f", &parameters.depth);
        printf("Depth: %.2f\n", parameters.depth);

        printf("Enter height for bead %d: ", i + 1);
        scanf("%f", &parameters.height);
        printf("Height: %.2f\n", parameters.height);
    }

    for (i = 0; i < numBeads; i++) {
```

```
        printf("Bead ID: %d, Material: %s, Geometry Type: %s\n",
               beadRecords[i].beadID, beadRecords[i].material,
beadRecords[i].geometryType);
    }

    free(beadRecords);
}

void weldingConsumableInventorySystem() {
    struct Consumable {
        int consumableID;
        char type[50];
        int quantity;
    };

    struct Consumable **inventory;
    const int numConsumables = 3;
    int i;

    inventory = (struct Consumable **)malloc(numConsumables * sizeof(struct
Consumable *));

    for (i = 0; i < numConsumables; i++) {
        inventory[i] = (struct Consumable *)malloc(sizeof(struct Consumable));

        printf("Enter consumable ID, type, and quantity for item %d: ", i + 1);
        scanf("%d %s %d", &inventory[i]->consumableID, inventory[i]->type,
&inventory[i]->quantity);
    }

    for (i = 0; i < numConsumables; i++) {
        printf("Consumable ID: %d, Type: %s, Quantity: %d\n",
               inventory[i]->consumableID, inventory[i]->type, inventory[i]->quantity);

        free(inventory[i]);
    }

    free(inventory);
}
```

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>

// Function prototypes
void weldingConsumableInventorySystem();

int main() {
    weldingConsumableInventorySystem();
    return 0;
}

void weldingConsumableInventorySystem() {
    struct Consumable {
        int consumableID;
        char type[50];
        int quantity;
    };

    struct Consumable **inventory;
    const int numConsumables = 3;
    int i;

    inventory = (struct Consumable **)malloc(numConsumables * sizeof(struct
Consumable *));

    for (i = 0; i < numConsumables; i++) {
        inventory[i] = (struct Consumable *)malloc(sizeof(struct Consumable));
        scanf("%d %s %d", &inventory[i]->consumableID, inventory[i]->type,
&inventory[i]->quantity);
```

```
    }

    for (i = 0; i < numConsumables; i++) {
        printf("Consumable ID: %d, Type: %s, Quantity: %d\n",
               inventory[i]->consumableID, inventory[i]->type, inventory[i]->quantity);

        free(inventory[i]);
    }

    free(inventory);
}
```

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>

union Variables {
    int availability;
    int usageFrequency;
};

typedef struct {
    int id;
    char name[50];
    char type[50];
    const char *details;
    union Variables var;
} Equipment;
```

```
void add(Equipment **equipment, int *count);
void display(Equipment *equipment, int count);

int main() {
    Equipment *equipment = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Equipment\n");
        printf("2. Display Equipment\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&equipment, &count);
                break;
            case 2:
                display(equipment, count);
                break;
            case 3:
                free(equipment);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Equipment **equipment, int *count) {
    (*count)++;
    *equipment = realloc(*equipment, (*count) * sizeof(Equipment));

    if (*equipment == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Equipment *new = &(*equipment)[*count - 1];
```

```
    printf("Enter Equipment ID: ");
    scanf("%d", &new->id);
    printf("Enter Equipment Name: ");
    scanf(" %[^\n]s", new->name);
    printf("Enter Equipment Type: ");
    scanf(" %[^\n]s", new->type);
    printf("Enter Usage Frequency: ");
    scanf("%d", &new->var.usageFrequency);

    new->details = "Protected equipment details";
    printf("Successfully added.\n");
}

void display(Equipment *equipment, int count) {
    if (count == 0) {
        printf("No equipment to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", equipment[i].id);
        printf("  Name: %s\n", equipment[i].name);
        printf("  Type: %s\n", equipment[i].type);
        printf("  Usage Frequency: %d\n", equipment[i].var.usageFrequency);
        printf("  Details: %s\n", equipment[i].details);
    }
}
```

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 DefectType {
    char porosity[50];
    char cracking[50];
    char spatter[50];
};

typedef struct {
    int id;
    char type[50];
    char severity[20];
    const char *details;
    union DefectType defect;
} Defect;

void add(Defect **defect, int *count);
void display(Defect *defect, int count);

int main() {
    Defect *defect = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Defect\n");
        printf("2. Display Defects\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&defect, &count);
                break;
            case 2:
                display(defect, count);
                break;
            case 3:
                free(defect);
                printf("Exiting the program.\n");
                return 0;
```

```
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Defect **defect, int *count) {
    (*count)++;
    *defect = realloc(*defect, (*count) * sizeof(Defect));

    if (*defect == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Defect *new = &(*defect)[*count - 1];

    printf("Enter Defect ID: ");
    scanf("%d", &new->id);
    printf("Enter Defect Type: ");
    scanf(" %[^\n]s", new->type);
    printf("Enter Severity Level: ");
    scanf(" %[^\n]s", new->severity);

    new->details = "Protected defect classification";
    printf("Successfully added.\n");
}

void display(Defect *defect, int count) {
    if (count == 0) {
        printf("No defects to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", defect[i].id);
        printf("  Type: %s\n", defect[i].type);
        printf("  Severity: %s\n", defect[i].severity);
        printf("  Details: %s\n", defect[i].details);
    }
}
```

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>

union PerformanceFactors {
    float arcStability;
    float penetrationDepth;
};

typedef struct {
    int id;
    char materialType[50];
    float currentSetting;
    const char *details;
    union PerformanceFactors factor;
} Performance;

void add(Performance **performance, int *count);
void display(Performance *performance, int count);

int main() {
    Performance *performance = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Performance Data\n");
        printf("2. Display Performance Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
```

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

        switch (choice) {
            case 1:
                add(&performance, &count);
                break;
            case 2:
                display(performance, count);
                break;
            case 3:
                free(performance);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Performance **performance, int *count) {
    (*count)++;
    *performance = realloc(*performance, (*count) * sizeof(Performance));

    if (*performance == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Performance *new = &(*performance)[*count - 1];

    printf("Enter Performance ID: ");
    scanf("%d", &new->id);
    printf("Enter Material Type: ");
    scanf(" %[^\n]s", new->materialType);
    printf("Enter Current Setting: ");
    scanf("%f", &new->currentSetting);

    new->details = "Protected performance configuration";
    printf("Successfully added.\n");
}

void display(Performance *performance, int count) {
```

```
    if (count == 0) {
        printf("No performance data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", performance[i].id);
        printf("  Material Type: %s\n", performance[i].materialType);
        printf("  Current Setting: %.2f\n", performance[i].currentSetting);
        printf("  Details: %s\n", performance[i].details);
    }
}
```

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>

typedef struct {
    int id;
    char name[50];
    int priority;
    float duration;
    const char *details;
} Task;

void add(Task **tasks, int *count);
void display(Task *tasks, int count);

int main() {
    Task *tasks = NULL;
```

```
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Task\n");
        printf("2. Display Tasks\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&tasks, &count);
                break;
            case 2:
                display(tasks, count);
                break;
            case 3:
                free(tasks);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Task **tasks, int *count) {
    (*count)++;
    *tasks = realloc(*tasks, (*count) * sizeof(Task));

    if (*tasks == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Task *new = &(*tasks)[*count - 1];

    printf("Enter Task ID: ");
    scanf("%d", &new->id);
    printf("Enter Task Name: ");
    scanf(" %[^\n]s", new->name);
    printf("Enter Task Priority: ");
```

```
    scanf("%d", &new->priority);
    printf("Enter Task Duration (in hours): ");
    scanf("%f", &new->duration);

    new->details = "Protected task details";
    printf("Task successfully added.\n");
}

void display(Task *tasks, int count) {
    if (count == 0) {
        printf("No tasks to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", tasks[i].id);
        printf("  Name: %s\n", tasks[i].name);
        printf("  Priority: %d\n", tasks[i].priority);
        printf("  Duration: %.2f hours\n", tasks[i].duration);
        printf("  Details: %s\n", tasks[i].details);
    }
}
```

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>
#include <string.h>

union DefectParams {
    float size;
    char location[50];
};
```

```
typedef struct {
    int id;
    char method[50];
    char result[100];
    const char *details;
    union DefectParams defect;
} Inspection;

void add(Inspection **inspections, int *count);
void display(Inspection *inspections, int count);

int main() {
    Inspection *inspections = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Inspection\n");
        printf("2. Display Inspections\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&inspections, &count);
                break;
            case 2:
                display(inspections, count);
                break;
            case 3:
                free(inspections);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Inspection **inspections, int *count) {
```

```
    (*count)++;
    *inspections = realloc(*inspections, (*count) * sizeof(Inspection));

    if (*inspections == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Inspection *new = &(*inspections)[*count - 1];

    printf("Enter Inspection ID: ");
    scanf("%d", &new->id);
    printf("Enter Inspection Method: ");
    scanf(" %[^\n]s", new->method);
    printf("Enter Inspection Result: ");
    scanf(" %[^\n]s", new->result);

    new->details = "Protected inspection details";
    printf("Inspection successfully added.\n");
}

void display(Inspection *inspections, int count) {
    if (count == 0) {
        printf("No inspections to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", inspections[i].id);
        printf("  Method: %s\n", inspections[i].method);
        printf("  Result: %s\n", inspections[i].result);
        printf("  Details: %s\n", inspections[i].details);
    }
}
```

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>

typedef struct {
    int id;
    char configuration[50];
    char status[50];
    const char *details;
} Robot;

void add(Robot **robots, int *count);
void display(Robot *robots, int count);

int main() {
    Robot *robots = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Robot Configuration\n");
        printf("2. Display Robot Configurations\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&robots, &count);
                break;
            case 2:
                display(robots, count);
                break;
            case 3:
                free(robots);
                printf("Exiting the program.\n");
                return 0;
            default:
```

```
            printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Robot **robots, int *count) {
    (*count)++;
    *robots = realloc(*robots, (*count) * sizeof(Robot));

    if (*robots == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Robot *new = &(*robots)[*count - 1];

    printf("Enter Robot ID: ");
    scanf("%d", &new->id);
    printf("Enter Robot Configuration: ");
    scanf(" %[^\n]s", new->configuration);
    printf("Enter Robot Status: ");
    scanf(" %[^\n]s", new->status);

    new->details = "Protected robot configuration details";
    printf("Robot successfully added.\n");
}

void display(Robot *robots, int count) {
    if (count == 0) {
        printf("No robots to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", robots[i].id);
        printf("  Configuration: %s\n", robots[i].configuration);
        printf("  Status: %s\n", robots[i].status);
        printf("  Details: %s\n", robots[i].details);
    }
}
```

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>
#include <string.h>

union QualityParams {
    float hardness;
    float tensileStrength;
};

typedef struct {
    int id;
    char material[50];
    float qualityScore;
    const char *details;
    union QualityParams quality;
} WeldQuality;

void add(WeldQuality **weldQualities, int *count);
void display(WeldQuality *weldQualities, int count);

int main() {
    WeldQuality *weldQualities = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Weld Quality Data\n");
        printf("2. Display Weld Quality Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                add(&weldQualities, &count);
                break;
            case 2:
                display(weldQualities, count);
                break;
            case 3:
                free(weldQualities);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(WeldQuality **weldQualities, int *count) {
    (*count)++;
    *weldQualities = realloc(*weldQualities, (*count) * sizeof(WeldQuality));

    if (*weldQualities == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    WeldQuality *new = &(*weldQualities)[*count - 1];

    printf("Enter Weld ID: ");
    scanf("%d", &new->id);
    printf("Enter Material: ");
    scanf(" %[^\n]s", new->material);
    printf("Enter Quality Score: ");
    scanf("%f", &new->qualityScore);

    new->details = "Protected weld quality details";
    printf("Weld quality data successfully added.\n");
}

void display(WeldQuality *weldQualities, int count) {
    if (count == 0) {
        printf("No weld quality data to display.\n");
```

```
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", weldQualities[i].id);
        printf("  Material: %s\n", weldQualities[i].material);
        printf("  Quality Score: %.2f\n", weldQualities[i].qualityScore);
        printf("  Details: %s\n", weldQualities[i].details);
    }
}
```

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>
#include <string.h>

union HeatInputParams {
    float heatInput;
    float timeTemperature[2];  // Example: time[0] and temperature[1]
};

typedef struct {
    int id;
    float current;
    float voltage;
    const char *details;
    union HeatInputParams heatInput;
} ThermalInput;

void add(ThermalInput **thermalInputs, int *count);
void display(ThermalInput *thermalInputs, int count);
```

```
int main() {
    ThermalInput *thermalInputs = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Thermal Input Data\n");
        printf("2. Display Thermal Input Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&thermalInputs, &count);
                break;
            case 2:
                display(thermalInputs, count);
                break;
            case 3:
                free(thermalInputs);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(ThermalInput **thermalInputs, int *count) {
    (*count)++;
    *thermalInputs = realloc(*thermalInputs, (*count) * sizeof(ThermalInput));

    if (*thermalInputs == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    ThermalInput *new = &(*thermalInputs)[*count - 1];

    printf("Enter Thermal Input ID: ");
    scanf("%d", &new->id);
```

```
    printf("Enter Current: ");
    scanf("%f", &new->current);
    printf("Enter Voltage: ");
    scanf("%f", &new->voltage);

    new->details = "Protected thermal input data";
    printf("Thermal input data successfully added.\n");
}

void display(ThermalInput *thermalInputs, int count) {
    if (count == 0) {
        printf("No thermal input data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", thermalInputs[i].id);
        printf("  Current: %.2f\n", thermalInputs[i].current);
        printf("  Voltage: %.2f\n", thermalInputs[i].voltage);
        printf("  Details: %s\n", thermalInputs[i].details);
    }
}
```

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>

typedef struct {
    int id;
    char material[50];
    char jointType[50];
```

```
    const char *details;
} WeldingProcedure;

void add(WeldingProcedure **procedures, int *count);
void display(WeldingProcedure *procedures, int count);

int main() {
    WeldingProcedure *procedures = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Welding Procedure\n");
        printf("2. Display Welding Procedures\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&procedures, &count);
                break;
            case 2:
                display(procedures, count);
                break;
            case 3:
                free(procedures);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(WeldingProcedure **procedures, int *count) {
    (*count)++;
    *procedures = realloc(*procedures, (*count) * sizeof(WeldingProcedure));

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

```
    }
    WeldingProcedure *new = &(*procedures)[*count - 1];

    printf("Enter Procedure ID: ");
    scanf("%d", &new->id);
    printf("Enter Material: ");
    scanf(" %[^\n]s", new->material);
    printf("Enter Joint Type: ");
    scanf(" %[^\n]s", new->jointType);

    new->details = "Protected welding procedure details";
    printf("Welding procedure successfully added.\n");
}

void display(WeldingProcedure *procedures, int count) {
    if (count == 0) {
        printf("No welding procedures to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", procedures[i].id);
        printf("  Material: %s\n", procedures[i].material);
        printf("  Joint Type: %s\n", procedures[i].jointType);
        printf("  Details: %s\n", procedures[i].details);
    }
}
```

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>
```

```
#include <string.h>

union JointParams {
    float angle;
    float radius;
};

typedef struct {
    int id;
    char type[50];
    const char *details;
    union JointParams joint;
} JointDesign;

void add(JointDesign **jointDesigns, int *count);
void display(JointDesign *jointDesigns, int count);

int main() {
    JointDesign *jointDesigns = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Joint Design Data\n");
        printf("2. Display Joint Designs\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&jointDesigns, &count);
                break;
            case 2:
                display(jointDesigns, count);
                break;
            case 3:
                free(jointDesigns);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
```

```
        }
    }
}

void add(JointDesign **jointDesigns, int *count) {
    (*count)++;
    *jointDesigns = realloc(*jointDesigns, (*count) * sizeof(JointDesign));

    if (*jointDesigns == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    JointDesign *new = &(*jointDesigns)[*count - 1];

    printf("Enter Joint ID: ");
    scanf("%d", &new->id);
    printf("Enter Joint Type: ");
    scanf(" %[^\n]s", new->type);

    new->details = "Protected joint design details";
    printf("Joint design data successfully added.\n");
}

void display(JointDesign *jointDesigns, int count) {
    if (count == 0) {
        printf("No joint design data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", jointDesigns[i].id);
        printf("  Type: %s\n", jointDesigns[i].type);
        printf("  Details: %s\n", jointDesigns[i].details);
    }
}
```

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>

typedef struct {
    int id;
    char composition[50];
    float diameter;
    const char *details;
} FillerMetal;

void add(FillerMetal **fillerMetals, int *count);
void display(FillerMetal *fillerMetals, int count);

int main() {
    FillerMetal *fillerMetals = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Filler Metal Data\n");
        printf("2. Display Filler Metal Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&fillerMetals, &count);
                break;
            case 2:
                display(fillerMetals, count);
                break;
            case 3:
                free(fillerMetals);
                printf("Exiting the program.\n");
                return 0;
```

```
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(FillerMetal **fillerMetals, int *count) {
    (*count)++;
    *fillerMetals = realloc(*fillerMetals, (*count) * sizeof(FillerMetal));

    if (*fillerMetals == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    FillerMetal *new = &(*fillerMetals)[*count - 1];

    printf("Enter Filler Metal ID: ");
    scanf("%d", &new->id);
    printf("Enter Composition: ");
    scanf(" %[^\n]s", new->composition);
    printf("Enter Diameter: ");
    scanf("%f", &new->diameter);

    new->details = "Protected filler metal details";
    printf("Filler metal data successfully added.\n");
}

void display(FillerMetal *fillerMetals, int count) {
    if (count == 0) {
        printf("No filler metal data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", fillerMetals[i].id);
        printf("  Composition: %s\n", fillerMetals[i].composition);
        printf("  Diameter: %.2f\n", fillerMetals[i].diameter);
        printf("  Details: %s\n", fillerMetals[i].details);
    }
}
```

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>

typedef struct {
    int id;
    char type[50];
    float capacity;
    const char *details;
} PowerSource;

void add(PowerSource **sources, int *count);
void display(PowerSource *sources, int count);

int main() {
    PowerSource *sources = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Power Source\n");
        printf("2. Display Power Sources\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&sources, &count);
                break;
```

```
            case 2:
                display(sources, count);
                break;
            case 3:
                free(sources);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(PowerSource **sources, int *count) {
    (*count)++;
    *sources = realloc(*sources, (*count) * sizeof(PowerSource));

    if (*sources == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    PowerSource *new = &(*sources)[*count - 1];

    printf("Enter Power Source ID: ");
    scanf("%d", &new->id);
    printf("Enter Source Type: ");
    scanf(" %[^\n]s", new->type);
    printf("Enter Capacity: ");
    scanf("%f", &new->capacity);

    new->details = "Protected power source details";
    printf("Power source successfully added.\n");
}

void display(PowerSource *sources, int count) {
    if (count == 0) {
        printf("No power sources to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", sources[i].id);
```

```
        printf("  Type: %s\n", sources[i].type);
        printf("  Capacity: %.2f\n", sources[i].capacity);
        printf("  Details: %s\n", sources[i].details);
    }
}
```

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

typedef struct {
    int id;
    char name[50];
    float skillScore;
    const char *details;
} Welder;

void add(Welder **welders, int *count);
void display(Welder *welders, int count);

int main() {
    Welder *welders = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Welder Data\n");
        printf("2. Display Welder Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                add(&welders, &count);
                break;
            case 2:
                display(welders, count);
                break;
            case 3:
                free(welders);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(Welder **welders, int *count) {
    (*count)++;
    *welders = realloc(*welders, (*count) * sizeof(Welder));

    if (*welders == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    Welder *new = &(*welders)[*count - 1];

    printf("Enter Welder ID: ");
    scanf("%d", &new->id);
    printf("Enter Name: ");
    scanf(" %[^\n]s", new->name);
    printf("Enter Skill Score: ");
    scanf("%f", &new->skillScore);

    new->details = "Protected welder skill assessment details";
    printf("Welder data successfully added.\n");
}

void display(Welder *welders, int count) {
    if (count == 0) {
```

```
        printf("No welder data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", welders[i].id);
        printf("  Name: %s\n", welders[i].name);
        printf("  Skill Score: %.2f\n", welders[i].skillScore);
        printf("  Details: %s\n", welders[i].details);
    }
}
```

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

union ArcStabilityMetrics {
    float arcLength;
    float consistency;
};

typedef struct {
    int id;
    float voltage;
    float current;
    union ArcStabilityMetrics metrics;
    const char *details;
} ArcStability;

void add(ArcStability **arcStabilityData, int *count);
void display(ArcStability *arcStabilityData, int count);
```

```
int main() {
    ArcStability *arcStabilityData = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Arc Stability Data\n");
        printf("2. Display Arc Stability Data\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&arcStabilityData, &count);
                break;
            case 2:
                display(arcStabilityData, count);
                break;
            case 3:
                free(arcStabilityData);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(ArcStability **arcStabilityData, int *count) {
    (*count)++;
    *arcStabilityData = realloc(*arcStabilityData, (*count) * sizeof(ArcStability));

    if (*arcStabilityData == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    ArcStability *new = &(*arcStabilityData)[*count - 1];

    printf("Enter Arc Stability ID: ");
    scanf("%d", &new->id);
```

```
    printf("Enter Voltage: ");
    scanf("%f", &new->voltage);
    printf("Enter Current: ");
    scanf("%f", &new->current);

    new->details = "Protected arc stability metrics details";
    printf("Arc stability data successfully added.\n");
}

void display(ArcStability *arcStabilityData, int count) {
    if (count == 0) {
        printf("No arc stability data to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", arcStabilityData[i].id);
        printf("  Voltage: %.2f\n", arcStabilityData[i].voltage);
        printf("  Current: %.2f\n", arcStabilityData[i].current);
        printf("  Details: %s\n", arcStabilityData[i].details);
    }
}
```

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

typedef struct {
    int id;
    char moduleName[50];
    float progress;
```

```
    const char *details;
} TrainingModule;

void add(TrainingModule **modules, int *count);
void display(TrainingModule *modules, int count);

int main() {
    TrainingModule *modules = NULL;
    int count = 0, choice;

    while (1) {
        printf("\n1. Add Training Module\n");
        printf("2. Display Training Modules\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add(&modules, &count);
                break;
            case 2:
                display(modules, count);
                break;
            case 3:
                free(modules);
                printf("Exiting the program.\n");
                return 0;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }
}

void add(TrainingModule **modules, int *count) {
    (*count)++;
    *modules = realloc(*modules, (*count) * sizeof(TrainingModule));

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

```
    }
    TrainingModule *new = &(*modules)[*count - 1];

    printf("Enter Module ID: ");
    scanf("%d", &new->id);
    printf("Enter Module Name: ");
    scanf(" %[^\n]s", new->moduleName);
    printf("Enter Progress: ");
    scanf("%f", &new->progress);

    new->details = "Protected training details";
    printf("Training module successfully added.\n");
}

void display(TrainingModule *modules, int count) {
    if (count == 0) {
        printf("No training modules to display.\n");
        return;
    }
    for (int i = 0; i < count; i++) {
        printf("  ID: %d\n", modules[i].id);
        printf("  Module Name: %s\n", modules[i].moduleName);
        printf("  Progress: %.2f\n", modules[i].progress);
        printf("  Details: %s\n", modules[i].details);
    }
}
```