1) Define a structure to represent a 3D point in space. Write functions to calculate the distance between two points.

WTD: Design a structure to model a 3D point in space. Develop functions that calculate the Euclidean distance between any two given points using the standard distance formula. Use Distance formula D= Sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2))
(e.g: I/P P1(1.0,2.0,3.0), P2(4.0,6.0,8.0); O/P is Distance= 7.071)

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

// Define a structure to represent a 3D point
struct Point3D {
    double x;
    double y;
    double z;
};

// Function to calculate the Euclidean distance between two 3D points
double calculateDistance(struct Point3D p1, struct Point3D p2) {
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    double dz = p2.z - p1.z;

    return sqrt(dx * dx + dy * dy + dz * dz);
}

int main() {
    // Define two 3D points
    struct Point3D p1 = {1.0, 2.0, 3.0};
    struct Point3D p2 = {4.0, 6.0, 8.0};

    // Calculate the distance between the two points
    double distance = calculateDistance(p1, p2);

    // Print the result
    printf("Distance: %.3lf\n", distance);
```

```
    return 0;
}
```

2) Define a structure for a student with name, roll number, and marks in 5 subjects. Calculate the average marks for a list of students. (Structure with Arrays).

WTD: Construct a structure that encapsulates a student's details, specifically their name, roll number, and scores in five subjects. Implement a function that computes the average marks for an array of student structures

(e.g: I/P: Name: "Alex", Roll: 105, Marks: [75, 80, 88, 82, 86]; O/P: Avg Marks: 82.2)

```c
#include <stdio.h>

// Define a structure to represent a student
struct Student {
    char name[50];
    int rollNumber;
    int marks[5]; // Array to store marks in 5 subjects
};

// Function to calculate the average marks for an array of students
float calculateAverageMarks(struct Student students[], int numStudents) {
    float totalMarks = 0.0;

    // Iterate through the array of students and calculate the total marks
    for (int i = 0; i < numStudents; i++) {
        for (int j = 0; j < 5; j++) {
            totalMarks += students[i].marks[j];
        }
    }

    // Calculate the average marks
    float averageMarks = totalMarks / (numStudents * 5);
    return averageMarks;
}
```

```c
int main() {
    // Define an array of students
    struct Student students[3];

    // Initialize student data
    strcpy(students[0].name, "Alex");
    students[0].rollNumber = 105;
    students[0].marks[0] = 75;
    students[0].marks[1] = 80;
    students[0].marks[2] = 88;
    students[0].marks[3] = 82;
    students[0].marks[4] = 86;

    // Add more students with data...

    int numStudents = 1; // Update with the actual number of students

    // Calculate the average marks for the array of students
    float avgMarks = calculateAverageMarks(students, numStudents);

    // Print the average marks
    printf("Average Marks: %.2f\n", avgMarks);

    return 0;
}
```

3) Create a structure for a book with title, author, and price. Implement a function to discount the book's price by a given percentage.

WTD: Design a function that applies a specified discount percentage to the book's price, updating its value accordingly.
(e.g: I/P: Title: "Pride and Prejudice", Author: "Austen", Price: $30, Discount: 15%; O/P: New Price: $25.5)

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

```c
// Define a structure to represent a book
struct Book {
    char title[100];
    char author[100];
    float price;
};

// Function to apply a discount to the book's price
void applyDiscount(struct Book *book, float discountPercentage) {
    // Calculate the discount amount
    float discountAmount = (discountPercentage / 100) * book->price;

    // Apply the discount to the price
    book->price -= discountAmount;
}

int main() {
    // Create a book and initialize its data
    struct Book myBook;
    strcpy(myBook.title, "Pride and Prejudice");
    strcpy(myBook.author, "Austen");
    myBook.price = 30.0; // Initial price in dollars

    // Apply a 15% discount to the book's price
    float discountPercentage = 15.0;
    applyDiscount(&myBook, discountPercentage);

    // Print the new price
    printf("Title: %s\n", myBook.title);
    printf("Author: %s\n", myBook.author);
    printf("New Price: $%.2f\n", myBook.price);

    return 0;
}
```

4) Design a structure for an employee with an embedded structure for address (street, city, state, zip). (Nested Structure)

WTD: Within this structure, embed another structure specifically meant for the employee's address details, capturing street, city, state, and zip code. Ensure capabilities to extract and display this address in a coherent format.

(e.g: I/P: Name: "Bob", Address: [Street: "456 Maple Rd", City: "Brookfield", State: "WI", Zip: "53005"]; O/P: Address: 456 Maple Rd, Brookfield, WI, 53005)

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

// Define a structure for an address
struct Address {
    char street[100];
    char city[50];
    char state[20];
    char zip[10];
};

// Define a structure for an employee with an embedded Address structure
struct Employee {
    char name[100];
    struct Address address;
};

// Function to display the employee's address in a coherent format
void displayAddress(struct Employee emp) {
    printf("Address: %s, %s, %s, %s\n", emp.address.street,
emp.address.city, emp.address.state, emp.address.zip);
}

int main() {
    // Create an employee structure and initialize its data
    struct Employee employee;
    strcpy(employee.name, "Bob");
    strcpy(employee.address.street, "456 Maple Rd");
    strcpy(employee.address.city, "Brookfield");
    strcpy(employee.address.state, "WI");
```

```
    strcpy(employee.address.zip, "53005");

    // Display the employee's address
    displayAddress(employee);

    return 0;
}
```

5) Use a union to represent a 32-bit value that can be accessed as either two 16-bit values or four 8-bit values.

WTD: Implement a union that can hold a 32-bit numerical value. This union should allow for access to the stored number as either two separate 16-bit values or as four distinct 8-bit values. (e.g: I/P: Value: 0xABCD1234; O/P: 16-bits: 0xABCD, 0x1234; 8-bits: 0xAB, 0xCD, 0x12, 0x34)

```
#include <stdio.h>
#include <stdint.h> // Include the stdint.h header for fixed-size integer types

// Define a union for a 32-bit value that can be accessed as 16-bit or 8-bit values
union Value32 {
    uint32_t full32;        // 32-bit value
    struct {
        uint16_t high16;    // Upper 16 bits
        uint16_t low16;     // Lower 16 bits
    } split16;
    struct {
        uint8_t byte1;      // Byte 1 (Most Significant Byte)
        uint8_t byte2;      // Byte 2
        uint8_t byte3;      // Byte 3
        uint8_t byte4;      // Byte 4 (Least Significant Byte)
    } split8;
};

int main() {
    union Value32 val;
    val.full32 = 0xABCD1234;
```

```
    printf("32-bit value: 0x%08X\n", val.full32);
    printf("16-bit values: 0x%04X, 0x%04X\n", val.split16.high16,
val.split16.low16);
    printf("8-bit values: 0x%02X, 0x%02X, 0x%02X, 0x%02X\n",
val.split8.byte1, val.split8.byte2, val.split8.byte3, val.split8.byte4);

    return 0;
}
```

6) Define a structure and determine its memory size. Rearrange its members to minimize memory wastage due to alignment. (Structure Memory Alignment)

WTD: Construct a structure and evaluate its memory consumption. Reorganize the structure's components in a manner that minimizes memory wastage due to alignment constraints inherent in system architecture.

(e.g: I/P: Structure: int, char, short; O/P: Memory: 11 bytes; Optimized: 10 bytes)

```
#include <stdio.h>

// Original structure
struct Original {
    int num;     // 4 bytes (assuming int is 4 bytes)
    char ch;     // 1 byte
    short sh;    // 2 bytes
};

// Optimized structure
struct Optimized {
    char ch;     // 1 byte
    short sh;    // 2 bytes
    int num;     // 4 bytes
};

int main() {
    printf("Input: Structure with members in the order - int, char,
short\n");
```

```
    printf("Output: Original Memory Size: %lu bytes; Optimized Memory Size: %lu bytes\n", sizeof(struct Original), sizeof(struct Optimized));

    return 0;
}
```

7) Implement a mini database for a library using an array of structures. Include functionalities like adding a new book and searching for a book by title. (Array of Structures)

WTD: Develop a basic book database for a library using an array of structures. This database should support operations like the addition of new books and title-based book searches.
(e.g: I/P: Add: "To Kill a Mockingbird", Search: "Mockingbird"; O/P: Found: "To Kill a Mockingbird")

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

#define MAX_BOOKS 100

// Define the book structure
struct Book {
    char title[100];
    char author[100];
    int year;
};

// Initialize an array of structures to store book information
struct Book library[MAX_BOOKS];

// Track the current number of books in the library
int numBooks = 0;

// Function to add a new book to the library
void addBook(const char title[], const char author[], int year) {
    if (numBooks < MAX_BOOKS) {
        strcpy(library[numBooks].title, title);
        strcpy(library[numBooks].author, author);
```

```c
        library[numBooks].year = year;
        numBooks++;
        printf("Book added successfully.\n");
    } else {
        printf("The library is full. Cannot add more books.\n");
    }
}

// Function to search for a book by title
void searchByTitle(const char keyword[]) {
    int found = 0;
    printf("Matching books:\n");

    for (int i = 0; i < numBooks; i++) {
        if (strstr(library[i].title, keyword) != NULL) {
            printf("Title: %s\nAuthor: %s\nYear: %d\n\n",
library[i].title, library[i].author, library[i].year);
            found = 1;
        }
    }

    if (!found) {
        printf("No matching books found.\n");
    }
}

int main() {
    // Add some sample books
    addBook("To Kill a Mockingbird", "Harper Lee", 1960);
    addBook("1984", "George Orwell", 1949);
    addBook("The Great Gatsby", "F. Scott Fitzgerald", 1925);
    addBook("The Catcher in the Rye", "J.D. Salinger", 1951);

    // Search for books by title
    searchByTitle("Mockingbird");
    searchByTitle("1984");
    searchByTitle("Catcher");

    return 0;
}
```

8) Create a linked list of students using structures and dynamic memory allocation. (Dynamic Memory with Structures)

WTD: Implement a linked list that represents student records. For this purpose, use structures combined with dynamic memory allocation techniques. Ensure the capability to append new student records to the list.

(e.g: I/P: Add Student: "Mia", Roll: 110; O/P: Student Mia, Roll: 110 Added)

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

// Define the structure for a student
struct Student {
    char name[100];
    int roll;
    struct Student* next;
};

// Function to add a new student to the linked list
void addStudent(struct Student** head, const char name[], int roll) {
    struct Student* newStudent = (struct Student*)malloc(sizeof(struct
Student));
    if (newStudent == NULL) {
        printf("Memory allocation failed. Cannot add student.\n");
        return;
    }
    strcpy(newStudent->name, name);
    newStudent->roll = roll;
    newStudent->next = NULL;

    if (*head == NULL) {
        *head = newStudent;
    } else {
        struct Student* current = *head;
        while (current->next != NULL) {
```

```c
            current = current->next;
        }
        current->next = newStudent;
    }

    printf("Student %s, Roll: %d Added\n", name, roll);
}

// Function to print the list of students
void printStudents(struct Student* head) {
    printf("List of Students:\n");
    while (head != NULL) {
        printf("Student %s, Roll: %d\n", head->name, head->roll);
        head = head->next;
    }
}

// Function to free memory used by the linked list
void freeStudents(struct Student* head) {
    while (head != NULL) {
        struct Student* temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    struct Student* studentList = NULL; // Head of the linked list

    // Add some students
    addStudent(&studentList, "Mia", 110);
    addStudent(&studentList, "John", 111);
    addStudent(&studentList, "Alice", 112);

    // Print the list of students
    printStudents(studentList);

    // Free memory used by the linked list
    freeStudents(studentList);
```

```
    return 0;
}
```

9) Use a union to interpret a 4-byte array as an integer and a floating-point number.

WTD: Design a union that is capable of holding a 4-byte array. This union should facilitate the interpretation of this array in two ways: as an integral value and as a floating-point number.
(e.g: I/P: Bytes: [0x43, 0x48, 0x00, 0x00]; O/P: Float: 134.0)

```c
#include <stdio.h>

union Data {
    unsigned char bytes[4];
    int integer;
    float floatingPoint;
};

int main() {
    union Data data;
    data.bytes[0] = 0x00;
    data.bytes[1] = 0x00;
    data.bytes[2] = 0x48;
    data.bytes[3] = 0x43;

    printf("I/P: Bytes: [0x43, 0x48, 0x00, 0x00]\n");
    printf("O/P: Float: %.1f\n", data.floatingPoint);

    return 0;
}
```

10) Define a structure with bit fields to represent a set of configurations/settings for a device. (Bit Fields for compact Structure)

WTD: Construct a structure with bit fields to efficiently represent a device's varied configurations or settings in a compact manner.

(e.g: I/P: Config: 1001; O/P: Setting 1 & 4: ON)

```
#include <stdio.h>

// Define a structure with bit fields
struct DeviceSettings {
    unsigned int setting1 : 1; // Bit 0
    unsigned int setting2 : 1; // Bit 1
    unsigned int setting3 : 1; // Bit 2
    unsigned int setting4 : 1; // Bit 3
    // Add more settings as needed
};

int main() {
    // Create an instance of the structure
    struct DeviceSettings settings;

    // Initialize the settings
    settings.setting1 = 1; // ON
    settings.setting2 = 0; // OFF
    settings.setting3 = 0; // OFF
    settings.setting4 = 1; // ON

    // Check and print the settings
    printf("I/P: Config: %d%d%d%d\n", settings.setting4,
settings.setting3, settings.setting2, settings.setting1);
    if (settings.setting1)
        printf("O/P: Setting 1: ON\n");
    if (settings.setting2)
        printf("O/P: Setting 2: ON\n");
    if (settings.setting3)
        printf("O/P: Setting 3: ON\n");
    if (settings.setting4)
        printf("O/P: Setting 4: ON\n");
```

```
    return 0;
}
```

11) Implement a function that takes two date structures (day, month, year) and returns the difference in days. (Passing Structures to Functions)

WTD: Develop a function that accepts two date structures, each detailing the day, month, and year. This function should compute and return the difference between these dates in terms of days.
(e.g: I/P: Date1: 15/02/2020, Date2: 20/02/2020; O/P: Diff: 5 days)

```
#include <stdio.h>

// Define a Date structure
struct Date {
    int day;
    int month;
    int year;
};

// Function to calculate the difference in days between two dates
int dateDifference(struct Date date1, struct Date date2) {
    // Days in each month (non-leap year)
    int daysInMonth[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31};

    // Calculate days in Date1
    int days1 = date1.year * 365 + date1.day;
    for (int i = 1; i < date1.month; i++) {
        days1 += daysInMonth[i];
    }
    if (date1.month > 2 && ((date1.year % 4 == 0 && date1.year % 100 != 0)
|| (date1.year % 400 == 0))) {
        days1 += 1; // Leap year adjustment
    }
```

```
    // Calculate days in Date2
    int days2 = date2.year * 365 + date2.day;
    for (int i = 1; i < date2.month; i++) {
        days2 += daysInMonth[i];
    }
    if (date2.month > 2 && ((date2.year % 4 == 0 && date2.year % 100 != 0)
|| (date2.year % 400 == 0))) {
        days2 += 1; // Leap year adjustment
    }

    // Calculate the difference in days
    int difference = days2 - days1;

    return difference;
}

int main() {
    // Define two date structures
    struct Date date1 = {15, 2, 2020};
    struct Date date2 = {20, 2, 2020};

    // Calculate the difference in days
    int daysDifference = dateDifference(date1, date2);

    // Print the result
    printf("I/P: Date1: %02d/%02d/%04d, Date2: %02d/%02d/%04d\n",
date1.day, date1.month, date1.year, date2.day, date2.month, date2.year);
    printf("O/P: Diff: %d days\n", daysDifference);

    return 0;
}
```

12) Create a union that can hold an IP address as a string and as four separate byte values. Implement functions to convert between the two representations.

WTD: Implement a union capable of storing an IP address. This union should support two representations of the IP: as a singular string and as its four byte-wise components. Design functions that facilitate conversions between these representations.
(e.g: I/P: String: "10.0.0.1"; O/P: Bytes: 10, 0, 0, 1)

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

// Define a union to hold IP address
union IPAddress {
    char ipString[16]; // To store IP as a string (e.g., "10.0.0.1")
    struct {
        unsigned char byte1;
        unsigned char byte2;
        unsigned char byte3;
        unsigned char byte4;
    } ipBytes; // To store IP as four separate bytes
};

// Function to convert IP string to bytes
void stringToBytes(const char *ipString, union IPAddress *ip) {
    sscanf(ipString, "%hhu.%hhu.%hhu.%hhu",
           &ip->ipBytes.byte1, &ip->ipBytes.byte2, &ip->ipBytes.byte3,
&ip->ipBytes.byte4);
}

// Function to convert IP bytes to string
void bytesToString(const union IPAddress *ip, char *outputString) {
    snprintf(outputString, 16, "%d.%d.%d.%d",
             ip->ipBytes.byte1, ip->ipBytes.byte2, ip->ipBytes.byte3,
ip->ipBytes.byte4);
}

int main() {
    // Create an IPAddress union
```

```
    union IPAddress ip;

    // Initialize it with an IP string
    const char *inputIPString = "10.0.0.1";
    strcpy(ip.ipString, inputIPString);

    // Convert IP string to bytes
    stringToBytes(inputIPString, &ip);

    // Convert IP bytes back to string
    char outputString[16];
    bytesToString(&ip, outputString);

    // Print the results
    printf("I/P: String: \"%s\"\n", inputIPString);
    printf("O/P: Bytes: %d, %d, %d, %d\n",
            ip.ipBytes.byte1, ip.ipBytes.byte2, ip.ipBytes.byte3,
ip.ipBytes.byte4);

    return 0;
}
```

13) Create a structure representing a menu item with a name (string) and an associated function pointer to execute when the menu item is selected.(Function pointer in structures)

WTD: Define a structure that models a menu item. Each menu item should have a name (a string) and an associated function pointer. This function pointer should point to the action to be executed when the menu item is selected.
(e.g: I/P: Select: "Option2"; O/P: "Option2 Executed")

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // Add this line to include the string.h header

// Define a structure representing a menu item
struct MenuItem {
    const char *name; // Name of the menu item
```

```c
    void (*action)(); // Function pointer for the associated action
};

// Functions to be executed when menu items are selected
void option1() {
    printf("Option1 Executed\n");
}

void option2() {
    printf("Option2 Executed\n");
}

void option3() {
    printf("Option3 Executed\n");
}

int main() {
    // Create menu items and associate them with functions
    struct MenuItem menu[] = {
        {"Option1", option1},
        {"Option2", option2},
        {"Option3", option3}
    };

    // Simulate selecting a menu item (e.g., "Option2")
    const char *selectedItem = "Option2";

    // Search for the selected item and execute its associated function
    for (int i = 0; i < sizeof(menu) / sizeof(menu[0]); i++) {
        if (strcmp(selectedItem, menu[i].name) == 0) {
            menu[i].action();
            break;
        }
    }

    return 0;
}
```

14) Define a structure for a network packet, which includes an enum to represent the packet type (e.g., DATA, ACK, NACK). (Enum with structures)

WTD: Design a structure to represent a network packet. This structure should incorporate an enum to denote the packet type, which could be values like DATA, ACK, or NACK.
(e.g: I/P: Packet: ACK; O/P: This is an ACK packet)

```
#include <stdio.h>

// Enum to represent packet types
enum PacketType {
    DATA,
    ACK,
    NACK
};

// Structure for a network packet
struct NetworkPacket {
    enum PacketType type; // Enum to denote packet type
    // Other packet data members can be added here
};

int main() {
    // Create a network packet of type ACK
    struct NetworkPacket packet;
    packet.type = ACK;

    // Check the packet type and print a corresponding message
    switch (packet.type) {
        case DATA:
            printf("This is a DATA packet\n");
            break;
        case ACK:
            printf("This is an ACK packet\n");
            break;
        case NACK:
            printf("This is a NACK packet\n");
            break;
        default:
```

```
            printf("Unknown packet type\n");
            break;
    }

    return 0;
}
```

15) Design a structure to represent a command and its associated parameters. Implement a function to parse a string command into this structure. (Structure for Command Parsing)

WTD: Formulate a structure that captures a command and any associated parameters. Introduce a function capable of parsing a string-based command and populating this structure with the relevant details.
(e.g: I/P: String: "JUMP 20"; O/P: Command: JUMP, Param: 20)

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

// Structure to represent a command and its associated parameters
struct Command {
    char name[50];     // Command name (e.g., JUMP)
    int param;         // Command parameter (e.g., 20)
};

// Function to parse a string command into the Command structure
void parseCommand(const char *str, struct Command *cmd) {
    // Create a non-constant copy of the input string
    char *strCopy = strdup(str);

    // Tokenize the copied string based on spaces
    char *token = strtok(strCopy, " ");

    // First token is the command name
    if (token != NULL) {
        strcpy(cmd->name, token);
```

```
    } else {
        // Handle error if no command name is found
        strcpy(cmd->name, "UNKNOWN");
        cmd->param = 0;
        free(strCopy); // Free the copied string
        return;
    }

    // Second token is the command parameter (if available)
    token = strtok(NULL, " ");
    if (token != NULL) {
        cmd->param = atoi(token); // Convert parameter to an integer
    } else {
        cmd->param = 0; // Default parameter value if not provided
    }

    free(strCopy); // Free the copied string
}

int main() {
    const char *input = "JUMP 20";
    struct Command cmd;

    // Parse the input string into the Command structure
    parseCommand(input, &cmd);

    // Print the parsed command and parameter
    printf("Command: %s, Param: %d\n", cmd.name, cmd.param);

    return 0;
}
```

16) Create a base structure for a vehicle with attributes like weight and max_speed. Extend this to derive structures for a car (with attributes like seating capacity) and a truck (with attributes like max_load) (Structure Inheritance)

WTD: Establish a base structure to depict a vehicle, detailing attributes like its weight and maximum speed. Extend this base to derive specialized structures for different vehicle types like cars (with attributes like seating capacity) and trucks (with features like maximum load capacity). (e.g: I/P: Truck, Weight: 8000kg, Speed: 120km/h, Load: 5000kg; O/P: Can load up to 5000kg)

```
#include <stdio.h>

// Base structure for a vehicle
struct Vehicle {
    int weight;        // Weight of the vehicle in kg
    int max_speed;     // Maximum speed of the vehicle in km/h
};

// Structure for a car, derived from Vehicle
struct Car {
    struct Vehicle base;  // Embed the Vehicle structure
    int seating_capacity; // Seating capacity of the car
};

// Structure for a truck, derived from Vehicle
struct Truck {
    struct Vehicle base; // Embed the Vehicle structure
    int max_load;        // Maximum load capacity of the truck
};

// Function to print vehicle details
void printVehicleDetails(struct Vehicle *vehicle) {
    printf("Weight: %d kg\n", vehicle->weight);
    printf("Max Speed: %d km/h\n", vehicle->max_speed);
}

int main() {
    // Create a car and set its attributes
    struct Car myCar;
    myCar.base.weight = 1500;          // Car weight
```

```
    myCar.base.max_speed = 180;          // Car maximum speed
    myCar.seating_capacity = 5;          // Car seating capacity

    // Create a truck and set its attributes
    struct Truck myTruck;
    myTruck.base.weight = 8000;          // Truck weight
    myTruck.base.max_speed = 120;        // Truck maximum speed
    myTruck.max_load = 5000;             // Truck maximum load

    // Print car and truck details
    printf("Car Details:\n");
    printVehicleDetails(&myCar.base);
    printf("Seating Capacity: %d\n", myCar.seating_capacity);

    printf("\nTruck Details:\n");
    printVehicleDetails(&myTruck.base);
    printf("Maximum Load: %d kg\n", myTruck.max_load);

    return 0;
}
```

17) Define a structure for a complex number. Implement functions for addition, subtraction, multiplication, and division of two complex numbers.

WTD: Design a structure to represent a complex number. Develop a suite of functions that allow for mathematical operations on complex numbers, including addition, subtraction, multiplication, and division.
(e.g: I/P: Complex1: 4+5i, Complex2: 2+3i, SUB; O/P: Result: 2+2i).

```
#include <stdio.h>

// Structure to represent a complex number
struct Complex {
    double real;      // Real part
    double imaginary; // Imaginary part
};
```

```
// Function to add two complex numbers
struct Complex add(struct Complex a, struct Complex b) {
    struct Complex result;
    result.real = a.real + b.real;
    result.imaginary = a.imaginary + b.imaginary;
    return result;
}

// Function to subtract two complex numbers
struct Complex subtract(struct Complex a, struct Complex b) {
    struct Complex result;
    result.real = a.real - b.real;
    result.imaginary = a.imaginary - b.imaginary;
    return result;
}

// Function to multiply two complex numbers
struct Complex multiply(struct Complex a, struct Complex b) {
    struct Complex result;
    result.real = a.real * b.real - a.imaginary * b.imaginary;
    result.imaginary = a.real * b.imaginary + a.imaginary * b.real;
    return result;
}

// Function to divide two complex numbers
struct Complex divide(struct Complex a, struct Complex b) {
    struct Complex result;
    double denominator = b.real * b.real + b.imaginary * b.imaginary;
    result.real = (a.real * b.real + a.imaginary * b.imaginary) /
denominator;
    result.imaginary = (a.imaginary * b.real - a.real * b.imaginary) /
denominator;
    return result;
}

// Function to print a complex number
void printComplex(struct Complex num) {
    if (num.imaginary >= 0) {
        printf("%.2lf + %.2lfi\n", num.real, num.imaginary);
    } else {
```

```
        printf("%.2lf - %.2lfi\n", num.real, -num.imaginary);
    }
}

int main() {
    struct Complex complex1 = {4.0, 5.0};
    struct Complex complex2 = {2.0, 3.0};

    struct Complex sum = add(complex1, complex2);
    printf("Sum: ");
    printComplex(sum);

    struct Complex difference = subtract(complex1, complex2);
    printf("Difference: ");
    printComplex(difference);

    struct Complex product = multiply(complex1, complex2);
    printf("Product: ");
    printComplex(product);

    struct Complex quotient = divide(complex1, complex2);
    printf("Quotient: ");
    printComplex(quotient);

    return 0;
}
```

18) Use a union to convert endianness (big-endian to little-endian and vice versa) of an integer.

WTD: Implement a union that can hold an integer value. Use this union to facilitate endianness conversion, toggling between big-endian and little-endian representations of the integer.
(e.g: I/P: Int: 0xAABBCCDD; O/P: Converted: 0xDDCCBBAA)

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

```c
union EndianConverter {
    unsigned int intValue;
    unsigned char byteArray[4];
};

int main() {
    union EndianConverter converter;

    // Input integer in big-endian format (0xAABBCCDD)
    converter.byteArray[0] = 0xAA;
    converter.byteArray[1] = 0xBB;
    converter.byteArray[2] = 0xCC;
    converter.byteArray[3] = 0xDD;

    // Convert to little-endian format
    unsigned int littleEndianValue =
        (converter.byteArray[3] << 24) |
        (converter.byteArray[2] << 16) |
        (converter.byteArray[1] << 8) |
        (converter.byteArray[0]);

    printf("I/P: Int: 0xAABBCCDD; O/P: Converted: 0x%08X\n",
littleEndianValue);

    return 0;
}
```

19) Define a structure for representing time (hours, minutes, seconds). Implement functions to add time durations and convert this duration to seconds.

WTD: Define a structure that models time, detailing hours, minutes, and seconds. Introduce functions that can sum two time durations and also convert a given duration into its equivalent representation in seconds.
(e.g: I/P: Time1: 1h 20m, Time2: 0h 50m; O/P: Sum: 2h 10m, Seconds: 7800s)

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

```c
// Define a structure for time
struct Time {
    int hours;
    int minutes;
    int seconds;
};

// Function to add two time durations
struct Time addTime(struct Time t1, struct Time t2) {
    struct Time sum;
    sum.hours = t1.hours + t2.hours;
    sum.minutes = t1.minutes + t2.minutes;
    sum.seconds = t1.seconds + t2.seconds;

    // Adjust for overflow
    if (sum.seconds >= 60) {
        sum.minutes += sum.seconds / 60;
        sum.seconds %= 60;
    }
    if (sum.minutes >= 60) {
        sum.hours += sum.minutes / 60;
        sum.minutes %= 60;
    }

    return sum;
}

// Function to convert time duration to seconds
int timeToSeconds(struct Time t) {
    return t.hours * 3600 + t.minutes * 60 + t.seconds;
}

int main() {
    // Initialize two time durations
    struct Time time1 = {1, 20, 0};
    struct Time time2 = {0, 50, 0};

    // Add the time durations
    struct Time sum = addTime(time1, time2);
```

```
    // Convert the sum to seconds
    int seconds = timeToSeconds(sum);

    printf("I/P: Time1: %dh %dm, Time2: %dh %dm; O/P: Sum: %dh %dm,
Seconds: %ds\n",
           time1.hours, time1.minutes, time2.hours, time2.minutes,
sum.hours, sum.minutes, seconds);

    return 0;
}
```

20) Design a union that can represent an error code as both an integer and a descriptive string.

WTD: Design a union capable of representing an error code in two formats: as a numerical integer and as a descriptive string detailing the error's nature.
(e.g: I/P: Code: 500; O/P: Description: "Internal Server Error")

```
#include <stdio.h>

// Define a structure for error information
struct ErrorInfo {
    int code;
    const char *description;
};

int main() {
    // Initialize an error structure
    struct ErrorInfo error;
    error.code = 500;
    error.description = "Internal Server Error";

    // Access the error information
    int code = error.code;
    const char *description = error.description;

    printf("I/P: Code: %d; O/P: Description: \"%s\"\n", code,
description);
```

```
    return 0;
}
```

21) Create a union to hold an IP address both as a string (like "192.168.1.1") and as four separate byte values.

WTD: Create a union that can house an IP address. This union should support dual representations: as a singular cohesive string and as its constituent byte values.
(e.g: I/P: IP String: "10.0.2.15"; O/P: Octets: [10, 0, 2, 15])

```
#include <stdio.h>
#include <stdint.h>
#include <string.h> // Include the string.h header for strcpy

// Define a union to hold an IP address
union IPAddress {
    char ipString[16]; // Assuming IPv4 address as a string (e.g.,
"192.168.1.1")
    struct {
        uint8_t octet1;
        uint8_t octet2;
        uint8_t octet3;
        uint8_t octet4;
    } octets; // IPv4 address as four separate bytes
};

int main() {
    // Initialize the union with an IP address
    union IPAddress ip;
    strcpy(ip.ipString, "10.0.2.15");

    // Access the IP address as a string
    printf("I/P: IP String: \"%s\"; ", ip.ipString);

    // Access the IP address as four separate bytes
    printf("O/P: Octets: [%d, %d, %d, %d]\n",
```

```
            ip.octets.octet1, ip.octets.octet2, ip.octets.octet3,
ip.octets.octet4);

    return 0;
}
```

22) Create a structure to simulate file attributes like name, type (using enum), size, and creation date (use the Date structure from problem 2).

WTD: Implement a structure that simulates file attributes. This structure should capture details like the file's name, its type (represented using an enum), its size in bytes, and its creation date (leveraging a previously defined Date structure).
(e.g: I/P: Name: "document.txt", Type: Document, Size: 1024 bytes, Creation Date: 2023-08-29; O/P: File: "document.txt", Type: Document, Size: 1024 bytes, Created on: 2023-08-29)

```
#include <stdio.h>
#include <string.h> // Include the string.h header for strcpy

// Define an enum to represent file types
enum FileType {
    Document,
    Image,
    Audio,
    Video,
    Other
};

// Define a structure for date (as used in Problem 2)
struct Date {
    int year;
    int month;
    int day;
};

// Define a structure to simulate file attributes
struct FileAttributes {
    char name[256]; // Assuming a maximum name length of 255 characters
```

```
    enum FileType type;
    size_t size; // Using size_t for file size
    struct Date creationDate; // Using the previously defined Date
structure
};

int main() {
    // Create a FileAttributes structure
    struct FileAttributes file;

    // Fill in the file attributes
    strcpy(file.name, "document.txt");
    file.type = Document;
    file.size = 1024;
    file.creationDate.year = 2023;
    file.creationDate.month = 8;
    file.creationDate.day = 29;

    // Print the file attributes
    printf("I/P: Name: \"%s\", Type: %d, Size: %lu bytes, Creation Date:
%d-%02d-%02d\n",
           file.name, file.type, file.size, file.creationDate.year,
           file.creationDate.month, file.creationDate.day);

    // Optionally, you can map the enum values to type names
    const char* fileTypeNames[] = {
        "Document", "Image", "Audio", "Video", "Other"
    };

    // Access the file type using the enum and print it
    printf("O/P: File: \"%s\", Type: %s, Size: %lu bytes, Created on:
%d-%02d-%02d\n",
           file.name, fileTypeNames[file.type], file.size,
           file.creationDate.year, file.creationDate.month,
file.creationDate.day);

    return 0;
}
```

23) Create a structure to represent serialized data packets with fields like header, payload, and checksum. Implement functions to serialize and deserialize data.

WTD: Construct a structure that represents serialized data packets, detailing elements like header, payload, and checksum. Develop accompanying functions that enable the serialization of this structured data into a string and its subsequent deserialization back into the structure. (e.g: I/P: Header: 0x1234, Payload: [0xAA, 0xBB, 0xCC], Checksum: 0xABCD; O/P: Serialized: [0x12, 0x34, 0xAA, 0xBB, 0xCC, 0xAB, 0xCD])

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

// Define a structure to represent the data packet
struct DataPacket {
    uint16_t header;
    uint8_t payload[3];
    uint16_t checksum;
};

// Function to serialize a DataPacket into a byte array
void serializeDataPacket(const struct DataPacket *packet, uint8_t
*serialized) {
    serialized[0] = (uint8_t)(packet->header >> 8);  // Extract the high
byte of header
    serialized[1] = (uint8_t)(packet->header);       // Extract the low
byte of header
    serialized[2] = packet->payload[0];
    serialized[3] = packet->payload[1];
    serialized[4] = packet->payload[2];
    serialized[5] = (uint8_t)(packet->checksum >> 8);  // Extract the high
byte of checksum
    serialized[6] = (uint8_t)(packet->checksum);       // Extract the low
byte of checksum
}

// Function to deserialize a byte array into a DataPacket
void deserializeDataPacket(const uint8_t *serialized, struct DataPacket
*packet) {
```

```
    packet->header = (uint16_t)(((uint16_t)serialized[0] << 8) |
serialized[1]);
    packet->payload[0] = serialized[2];
    packet->payload[1] = serialized[3];
    packet->payload[2] = serialized[4];
    packet->checksum = (uint16_t)(((uint16_t)serialized[5] << 8) |
serialized[6]);
}

int main() {
    struct DataPacket originalPacket = {
        .header = 0x1234,
        .payload = {0xAA, 0xBB, 0xCC},
        .checksum = 0xABCD
    };

    uint8_t serializedData[7];
    serializeDataPacket(&originalPacket, serializedData);

    printf("Serialized: [0x%02X, 0x%02X, 0x%02X, 0x%02X, 0x%02X, 0x%02X,
0x%02X]\n",
           serializedData[0], serializedData[1], serializedData[2],
           serializedData[3], serializedData[4], serializedData[5],
           serializedData[6]);

    struct DataPacket deserializedPacket;
    deserializeDataPacket(serializedData, &deserializedPacket);

    printf("Deserialized: Header: 0x%04X, Payload: [0x%02X, 0x%02X,
0x%02X], Checksum: 0x%04X\n",
           deserializedPacket.header, deserializedPacket.payload[0],
           deserializedPacket.payload[1], deserializedPacket.payload[2],
           deserializedPacket.checksum);

    return 0;
}
```

24) Implement a structure that can store an array of data. However, the data type can vary - it could be an array of integers, floats, or characters. Use a union to manage this variability.

WTD: Design a structure with the ability to store an array of data. To accommodate different data types for the array (like integers, floats, or characters), use a union. Ensure mechanisms to correctly identify and retrieve the stored data type.
(e.g: I/P: DataType: Integer, Data: [1, 2, 3, 4, 5]; O/P: Array of Integers: [1, 2, 3, 4, 5])

```
#include <stdio.h>

// Enumeration to represent the data type
enum DataType {
    INTEGER,
    FLOAT,
    CHARACTER
};

// Define a union to hold different types of data
union DataUnion {
    int intArray[5];
    float floatArray[5];
    char charArray[5];
};

// Define the main structure to store data with a data type identifier
struct DataContainer {
    enum DataType dataType;
    union DataUnion data;
};

// Function to print the array based on data type
void printArray(struct DataContainer container) {
    switch (container.dataType) {
        case INTEGER:
            printf("Array of Integers: [");
```

```
            for (int i = 0; i < 5; i++) {
                printf("%d", container.data.intArray[i]);
                if (i < 4) printf(", ");
            }
            printf("]\n");
            break;
        case FLOAT:
            printf("Array of Floats: [");
            for (int i = 0; i < 5; i++) {
                printf("%.2f", container.data.floatArray[i]);
                if (i < 4) printf(", ");
            }
            printf("]\n");
            break;
        case CHARACTER:
            printf("Array of Characters: [");
            for (int i = 0; i < 5; i++) {
                printf("'%c'", container.data.charArray[i]);
                if (i < 4) printf(", ");
            }
            printf("]\n");
            break;
        default:
            printf("Invalid data type\n");
    }
}

int main() {
    struct DataContainer dataContainer;

    // Store an array of integers
    dataContainer.dataType = INTEGER;
    for (int i = 0; i < 5; i++) {
        dataContainer.data.intArray[i] = i + 1;
    }
    printArray(dataContainer);

    // Store an array of floats
    dataContainer.dataType = FLOAT;
    for (int i = 0; i < 5; i++) {
```

```
        dataContainer.data.floatArray[i] = (i + 1) * 1.5;
    }
    printArray(dataContainer);

    // Store an array of characters
    dataContainer.dataType = CHARACTER;
    for (int i = 0; i < 5; i++) {
        dataContainer.data.charArray[i] = 'A' + i;
    }
    printArray(dataContainer);

    return 0;
}
```

25) Create a structure that can represent a color in both RGB and CMYK formats. Use a union to switch between the RGB representation and the CMYK representation.

WTD: Formulate a structure capable of representing color information. This structure should support dual representations: in RGB format and in CMYK format. Using a union, ensure seamless switching between these two representations.
(e.g: I/P: Color Type: RGB, RGB: [255, 0, 128]; O/P: RGB Color: [255, 0, 128])

```
#include <stdio.h>

// Enumeration to represent the color type
enum ColorType {
    RGB,
    CMYK
};

// Define a union to hold RGB and CMYK color representations
union ColorData {
    struct {
        int red;
        int green;
        int blue;
    } rgb;
```

```c
    struct {
        float cyan;
        float magenta;
        float yellow;
        float key;
    } cmyk;
};

// Define the main structure to represent a color
struct Color {
    enum ColorType type;
    union ColorData data;
};

// Function to print the color based on its type
void printColor(struct Color color) {
    switch (color.type) {
        case RGB:
            printf("RGB Color: [%d, %d, %d]\n", color.data.rgb.red,
color.data.rgb.green, color.data.rgb.blue);
            break;
        case CMYK:
            printf("CMYK Color: [%.2f, %.2f, %.2f, %.2f]\n",
color.data.cmyk.cyan, color.data.cmyk.magenta, color.data.cmyk.yellow,
color.data.cmyk.key);
            break;
        default:
            printf("Invalid color type\n");
    }
}

int main() {
    struct Color color;

    // Store an RGB color
    color.type = RGB;
    color.data.rgb.red = 255;
    color.data.rgb.green = 0;
    color.data.rgb.blue = 128;
```

```c
    printColor(color);

    // Store a CMYK color
    color.type = CMYK;
    color.data.cmyk.cyan = 0.2;
    color.data.cmyk.magenta = 0.8;
    color.data.cmyk.yellow = 0.0;
    color.data.cmyk.key = 0.0;
    printColor(color);

    return 0;
}
```

—--Happy Learning—---