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

1. Real-Time Inventory Tracking System
Description:
Develop a system to track real-time inventory levels using structures for item details and unions for variable attributes (e.g., weight, volume). Use const pointers for immutable item codes and double pointers for managing dynamic inventory arrays.
Specifications:
Structure: Item details (ID, name, category).
Union: Attributes (weight, volume).
const Pointer: Immutable item codes.
Double Pointers: Dynamic inventory management.

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

// Define the Union for item attributes
union ItemAttributes {
    double weight;    // Weight of the item
    double volume;    // Volume of the item
};

// Define the structure for item details
struct Item {
    const char *itemCode;  // Immutable Item code
    char name[50];         // Item name
    char category[50];     // Item category
    union ItemAttributes attributes; // Union to store either weight or volume
};

// Function to create a new item and add it to the dynamic inventory
void addItem(struct Item ***inventory, int *inventorySize, const char *itemCode, const char *name, const char *category, double value, int isWeight) {
    // Dynamically allocate space for the new item
    *inventorySize += 1;
    *inventory = (struct Item **)realloc(*inventory, *inventorySize * sizeof(struct Item *));

    // Allocate memory for the new item and set its values
    (*inventory)[*inventorySize - 1] = (struct Item *)malloc(sizeof(struct Item));

    (*inventory)[*inventorySize - 1]->itemCode = itemCode; // Item code (immutable)
    strncpy((*inventory)[*inventorySize - 1]->name, name, sizeof((*inventory)[*inventorySize - 1]->name) - 1);
    strncpy((*inventory)[*inventorySize - 1]->category, category, sizeof((*inventory)[*inventorySize - 1]->category) - 1);

    // Set the attributes (weight or volume)
    if (isWeight) {
        (*inventory)[*inventorySize - 1]->attributes.weight = value;
    } else {
        (*inventory)[*inventorySize - 1]->attributes.volume = value;
    }
}
```

```
// Function to display inventory details
void displayInventory(struct Item **inventory, int inventorySize) {
    for (int i = 0; i < inventorySize; i++) {
        printf("Item Code: %s\n", inventory[i]->itemCode);
        printf("Name: %s\n", inventory[i]->name);
        printf("Category: %s\n", inventory[i]->category);
        printf("Attribute: ");
        if (inventory[i]->attributes.weight > 0) {
            printf("Weight: %.2f kg\n", inventory[i]->attributes.weight);
        } else {
            printf("Volume: %.2f liters\n", inventory[i]->attributes.volume);
        }
        printf("\n");
    }
}

// Function to free allocated memory
void freeInventory(struct Item **inventory, int inventorySize) {
    for (int i = 0; i < inventorySize; i++) {
        free(inventory[i]);
    }
    free(inventory);
}

int main() {
    struct Item **inventory = NULL;  // Double pointer for dynamic array of inventory items
    int inventorySize = 0;           // Current size of inventory

    // Adding items to the inventory
    addItem(&inventory, &inventorySize, "A001", "Laptop", "Electronics", 2.5, 1); // 1 for weight
    addItem(&inventory, &inventorySize, "B002", "Water Bottle", "Beverages", 1.5, 0); // 0 for volume
    addItem(&inventory, &inventorySize, "C003", "Desk", "Furniture", 0, 0);  // No weight, only volume

    // Display the inventory details
    displayInventory(inventory, inventorySize);

    // Free the allocated memory
    freeInventory(inventory, inventorySize);

    return 0;
}
```

2. Dynamic Route Management for Logistics
Description:
Create a system to dynamically manage shipping routes using structures for route data and unions for different modes of transport. Use const pointers for route IDs and double pointers for managing route arrays.
Specifications:
Structure: Route details (ID, start, end).
Union: Transport modes (air, sea, land).
const Pointer: Read-only route IDs.
Double Pointers: Dynamic route allocation.

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

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

// Define a structure for route details
struct Route {
    int routeID;          // Route ID
    char start[50];       // Starting location
    char end[50];         // Ending location
    union {
        char air[20];     // Air transport (e.g., flight number)
        char sea[20];     // Sea transport (e.g., ship name)
        char land[20];    // Land transport (e.g., vehicle ID)
    } transportMode;
};

// Function to dynamically allocate routes
void addRoute(struct Route ***routes, int *routeCount, int routeID, const char *start, const char *end,
const char *mode, const char *transportDetail) {
    // Reallocate memory for new route
    *routes = realloc(*routes, (*routeCount + 1) * sizeof(struct Route*));
    if (*routes == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Allocate memory for the new route
    (*routes)[*routeCount] = (struct Route*)malloc(sizeof(struct Route));
    if ((*routes)[*routeCount] == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Initialize route details
    (*routes)[*routeCount]->routeID = routeID;
    strncpy((*routes)[*routeCount]->start, start, sizeof((*routes)[*routeCount]->start) - 1);
    strncpy((*routes)[*routeCount]->end, end, sizeof((*routes)[*routeCount]->end) - 1);

    // Set transport mode
    if (strcmp(mode, "air") == 0) {
        strncpy((*routes)[*routeCount]->transportMode.air, transportDetail,
sizeof((*routes)[*routeCount]->transportMode.air) - 1);
    } else if (strcmp(mode, "sea") == 0) {
        strncpy((*routes)[*routeCount]->transportMode.sea, transportDetail,
sizeof((*routes)[*routeCount]->transportMode.sea) - 1);
    } else if (strcmp(mode, "land") == 0) {
        strncpy((*routes)[*routeCount]->transportMode.land, transportDetail,
sizeof((*routes)[*routeCount]->transportMode.land) - 1);
    }

    // Increment the route count
    (*routeCount)++;
}

// Function to print route details
void printRouteDetails(struct Route **routes, int routeCount) {
```

```
    for (int i = 0; i < routeCount; i++) {
        printf("Route ID: %d\n", routes[i]->routeID);
        printf("Start: %s\n", routes[i]->start);
        printf("End: %s\n", routes[i]->end);
        if (routes[i]->transportMode.air[0] != '\0') {
            printf("Transport Mode: Air, Flight: %s\n", routes[i]->transportMode.air);
        } else if (routes[i]->transportMode.sea[0] != '\0') {
            printf("Transport Mode: Sea, Ship: %s\n", routes[i]->transportMode.sea);
        } else if (routes[i]->transportMode.land[0] != '\0') {
            printf("Transport Mode: Land, Vehicle: %s\n", routes[i]->transportMode.land);
        }
        printf("\n");
    }
}

// Main function
int main() {
    struct Route **routes = NULL;   // Double pointer to hold the routes array
    int routeCount = 0;             // Number of routes

    // Add some routes
    addRoute(&routes, &routeCount, 1, "New York", "Los Angeles", "air", "Flight123");
    addRoute(&routes, &routeCount, 2, "London", "Paris", "sea", "Ship456");
    addRoute(&routes, &routeCount, 3, "Berlin", "Warsaw", "land", "Vehicle789");

    // Print route details
    printRouteDetails(routes, routeCount);

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

    return 0;
}
```

3. Fleet Maintenance and Monitoring
Description:
Develop a fleet management system using structures for vehicle details and unions for status (active, maintenance). Use const pointers for vehicle identifiers and double pointers to manage vehicle records.
Specifications:
Structure: Vehicle details (ID, type, status).
Union: Status (active, maintenance).
const Pointer: Vehicle IDs.
Double Pointers: Dynamic vehicle list management.

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

// Define a union to hold the vehicle status (active or maintenance)
typedef union {
    int active;       // Represents if the vehicle is active (1 or 0)
    int maintenance;  // Represents if the vehicle is under maintenance (1 or 0)
```

```
} Status;

// Define a structure for vehicle details
typedef struct {
    const int *vehicleID;  // Vehicle ID (constant pointer, cannot be modified)
    char type[20];         // Vehicle type (e.g., "Truck", "Bus", "Car")
    Status status;         // Vehicle status (active or maintenance)
} Vehicle;

// Function prototypes
void addVehicle(Vehicle **fleet, int *fleetSize, const int *id, const char *type, int status);
void displayFleet(Vehicle **fleet, int fleetSize);
void freeFleet(Vehicle **fleet, int fleetSize);

int main() {
    Vehicle **fleet = NULL;   // Double pointer to manage the fleet
    int fleetSize = 0;        // Current number of vehicles in the fleet

    // Vehicle IDs (constant integers)
    const int id1 = 101;
    const int id2 = 102;
    const int id3 = 103;

    // Adding vehicles to the fleet
    addVehicle(&fleet, &fleetSize, &id1, "Truck", 1);      // Vehicle 1: Active
    addVehicle(&fleet, &fleetSize, &id2, "Bus", 0);        // Vehicle 2: Under maintenance
    addVehicle(&fleet, &fleetSize, &id3, "Car", 1);        // Vehicle 3: Active

    // Display fleet information
    displayFleet(fleet, fleetSize);

    // Free dynamically allocated memory for fleet
    freeFleet(fleet, fleetSize);

    return 0;
}

// Function to add a vehicle to the fleet
void addVehicle(Vehicle **fleet, int *fleetSize, const int *id, const char *type, int status) {
    // Reallocate memory for the new vehicle in the fleet
    *fleet = realloc(*fleet, (*fleetSize + 1) * sizeof(Vehicle));

    // Create a new vehicle entry
    Vehicle newVehicle;
    newVehicle.vehicleID = id;
    strncpy(newVehicle.type, type, sizeof(newVehicle.type) - 1);
    newVehicle.type[sizeof(newVehicle.type) - 1] = '\0';  // Ensure null termination
    if (status == 1) {
        newVehicle.status.active = 1;  // Active status
    } else {
        newVehicle.status.maintenance = 1;  // Maintenance status
    }

    // Add the new vehicle to the fleet
    (*fleet)[*fleetSize] = newVehicle;
```

```
    (*fleetSize)++;
}

// Function to display fleet details
void displayFleet(Vehicle **fleet, int fleetSize) {
    printf("Fleet Details:\n");
    for (int i = 0; i < fleetSize; i++) {
        printf("Vehicle ID: %d\n", *(fleet[i]->vehicleID));
        printf("Type: %s\n", fleet[i]->type);
        if (fleet[i]->status.active) {
            printf("Status: Active\n");
        } else if (fleet[i]->status.maintenance) {
            printf("Status: Under Maintenance\n");
        }
        printf("\n");
    }
}

// Function to free memory allocated for the fleet
void freeFleet(Vehicle **fleet, int fleetSize) {
    free(*fleet);  // Free the allocated memory for fleet
}
```

4. Logistics Order Processing Queue
Description:
Implement an order processing system using structures for order details and unions for payment methods. Use const pointers for order IDs and double pointers for dynamic order queues.
Specifications:
Structure: Order details (ID, customer, items).
Union: Payment methods (credit card, cash).
const Pointer: Order IDs.
Double Pointers: Dynamic order queue.

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

// Define structure for order details
typedef struct {
    int orderID;
    char customer[100];
    char items[200];
} OrderDetails;

// Define union for payment methods
typedef union {
    char creditCard[20];
    double cashAmount;
} PaymentMethods;

// Define structure to represent an order with payment
typedef struct {
    const int *orderID;   // Constant pointer for Order ID
    OrderDetails details; // Order details
    PaymentMethods payment; // Payment method (credit card or cash)
```

```
} Order;

// Function to create an order
Order* createOrder(int orderID, const char* customer, const char* items, const char* paymentMethod,
void* paymentInfo) {
    Order* newOrder = (Order*)malloc(sizeof(Order)); // Allocate memory for a new order
    if (!newOrder) {
        printf("Memory allocation failed.\n");
        return NULL;
    }

    // Set the order ID as a constant pointer
    newOrder->orderID = (const int*)malloc(sizeof(int));
    *(newOrder->orderID) = orderID;

    // Set the customer and items
    strncpy(newOrder->details.customer, customer, sizeof(newOrder->details.customer) - 1);
    strncpy(newOrder->details.items, items, sizeof(newOrder->details.items) - 1);

    // Set the payment method
    if (strcmp(paymentMethod, "credit_card") == 0) {
        strncpy(newOrder->payment.creditCard, (char*)paymentInfo, sizeof(newOrder->payment.creditCard)
- 1);
    } else if (strcmp(paymentMethod, "cash") == 0) {
        newOrder->payment.cashAmount = *((double*)paymentInfo);
    } else {
        printf("Invalid payment method.\n");
        free(newOrder->orderID);
        free(newOrder);
        return NULL;
    }

    return newOrder;
}

// Function to add an order to the queue
void addOrderToQueue(Order ***orderQueue, int *queueSize, Order *newOrder) {
    *queueSize += 1;
    *orderQueue = (Order**)realloc(*orderQueue, (*queueSize) * sizeof(Order*));
    if (*orderQueue == NULL) {
        printf("Memory reallocation failed.\n");
        return;
    }

    (*orderQueue)[*queueSize - 1] = newOrder;
}

// Function to process and print orders
void processOrders(Order **orderQueue, int queueSize) {
    for (int i = 0; i < queueSize; i++) {
        printf("Processing Order ID: %d\n", *(orderQueue[i]->orderID));
        printf("Customer: %s\n", orderQueue[i]->details.customer);
        printf("Items: %s\n", orderQueue[i]->details.items);

        printf("Payment Method: ");
```

```
        if (strlen(orderQueue[i]->payment.creditCard) > 0) {
            printf("Credit Card (Last 4 digits: %s)\n", orderQueue[i]->payment.creditCard + 16);
        } else {
            printf("Cash: $%.2f\n", orderQueue[i]->payment.cashAmount);
        }
        printf("\n");
    }
}

// Function to free the memory for all orders in the queue
void freeOrderQueue(Order **orderQueue, int queueSize) {
    for (int i = 0; i < queueSize; i++) {
        free(orderQueue[i]->orderID);
        free(orderQueue[i]);
    }
    free(orderQueue);
}

int main() {
    Order **orderQueue = NULL;  // Double pointer for dynamic queue
    int queueSize = 0;

    // Example 1: Create an order with a credit card payment
    char creditCard[] = "1234567890123456";
    Order *order1 = createOrder(101, "John Doe", "Laptop, Mouse", "credit_card", creditCard);
    if (order1 != NULL) {
        addOrderToQueue(&orderQueue, &queueSize, order1);
    }

    // Example 2: Create an order with a cash payment
    double cashPayment = 250.50;
    Order *order2 = createOrder(102, "Jane Smith", "Phone, Headphones", "cash", &cashPayment);
    if (order2 != NULL) {
        addOrderToQueue(&orderQueue, &queueSize, order2);
    }

    // Process and print all orders
    processOrders(orderQueue, queueSize);

    // Free allocated memory
    freeOrderQueue(orderQueue, queueSize);

    return 0;
}
```

5. Shipment Tracking System
Description:
Develop a shipment tracking system using structures for shipment details and unions for tracking events. Use const pointers to protect tracking numbers and double pointers to handle dynamic shipment lists.
Specifications:
Structure: Shipment details (tracking number, origin, destination).
Union: Tracking events (dispatched, delivered).
const Pointer: Tracking numbers.
Double Pointers: Dynamic shipment tracking.

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

// Structure for shipment details
typedef struct {
    const char *trackingNumber;  // const pointer to protect tracking number
    char origin[100];
    char destination[100];
} ShipmentDetails;

// Union for tracking events
typedef union {
    char dispatched[100];
    char delivered[100];
} TrackingEvents;

// Structure that combines shipment details and tracking events
typedef struct {
    ShipmentDetails details;
    TrackingEvents events;
} Shipment;

// Function to create a new shipment and assign dynamic memory for the tracking number
void createShipment(Shipment **shipmentList, int *shipmentCount, const char *trackingNumber, const
char *origin, const char *destination) {
    // Allocate memory for a new shipment
    *shipmentList = (Shipment *)realloc(*shipmentList, (*shipmentCount + 1) * sizeof(Shipment));

    // Initialize the new shipment details
    (*shipmentList)[*shipmentCount].details.trackingNumber = trackingNumber;  // const pointer for tracking
number
    strncpy((*shipmentList)[*shipmentCount].details.origin, origin,
sizeof((*shipmentList)[*shipmentCount].details.origin));
    strncpy((*shipmentList)[*shipmentCount].details.destination, destination,
sizeof((*shipmentList)[*shipmentCount].details.destination));

    (*shipmentCount)++;
}

// Function to update the tracking event of a shipment
void updateTrackingEvent(Shipment *shipment, const char *event, const char *eventDetails) {
    if (strcmp(event, "dispatched") == 0) {
        strncpy(shipment->events.dispatched, eventDetails, sizeof(shipment->events.dispatched));
    } else if (strcmp(event, "delivered") == 0) {
        strncpy(shipment->events.delivered, eventDetails, sizeof(shipment->events.delivered));
    }
}

// Function to display shipment information
void displayShipment(Shipment *shipment) {
    printf("Tracking Number: %s\n", shipment->details.trackingNumber);
    printf("Origin: %s\n", shipment->details.origin);
    printf("Destination: %s\n", shipment->details.destination);
```

```
    if (strlen(shipment->events.dispatched) > 0) {
        printf("Dispatched: %s\n", shipment->events.dispatched);
    }
    if (strlen(shipment->events.delivered) > 0) {
        printf("Delivered: %s\n", shipment->events.delivered);
    }
}

int main() {
    Shipment *shipmentList = NULL;  // Pointer to dynamically allocated array of shipments
    int shipmentCount = 0;

    // Create some shipments
    createShipment(&shipmentList, &shipmentCount, "TN123456789", "New York", "Los Angeles");
    createShipment(&shipmentList, &shipmentCount, "TN987654321", "Chicago", "San Francisco");

    // Update tracking events for the first shipment
    updateTrackingEvent(&shipmentList[0], "dispatched", "January 20, 2025");
    updateTrackingEvent(&shipmentList[0], "delivered", "January 22, 2025");

    // Update tracking events for the second shipment
    updateTrackingEvent(&shipmentList[1], "dispatched", "January 21, 2025");

    // Display shipment information
    for (int i = 0; i < shipmentCount; i++) {
        displayShipment(&shipmentList[i]);
        printf("\n");
    }

    // Free dynamically allocated memory
    free(shipmentList);

    return 0;
}
```

6. Real-Time Traffic Management for Logistics
Description:
Create a system to manage real-time traffic data for logistics using structures for traffic nodes and unions for traffic conditions. Use const pointers for node identifiers and double pointers for dynamic traffic data storage.
Specifications:
Structure: Traffic node details (ID, location).
Union: Traffic conditions (clear, congested).
const Pointer: Node IDs.
Double Pointers: Dynamic traffic data management.

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

// Define the TrafficNode structure
typedef struct {
    const int *nodeID;  // Const pointer to node ID
    char location[100]; // Location as a string
} TrafficNode;
```

```
// Define the TrafficConditions union
typedef union {
    char condition[10];  // Store traffic condition as a string
    int congestionLevel; // Alternative representation of condition (e.g., 0-100 scale)
} TrafficConditions;

// Define a structure to hold dynamic traffic data for a node
typedef struct {
    TrafficConditions *conditionData;  // Double pointer to dynamically allocated traffic conditions
    int dataCount;                     // Number of traffic conditions for this node
} TrafficData;

// Function to dynamically allocate memory for traffic data (double pointers)
void allocateTrafficData(TrafficData *trafficData, int count) {
    trafficData->dataCount = count;
    trafficData->conditionData = (TrafficConditions *)malloc(sizeof(TrafficConditions) * count);

    if (trafficData->conditionData == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
}

// Function to set the traffic condition for a specific data index
void setTrafficCondition(TrafficData *trafficData, int index, const char *condition) {
    if (index >= 0 && index < trafficData->dataCount) {
        snprintf(trafficData->conditionData[index].condition,
sizeof(trafficData->conditionData[index].condition), "%s", condition);
    }
}

// Function to display traffic information
void displayTrafficInfo(TrafficNode *node, TrafficData *trafficData) {
    printf("Traffic Node ID: %d\n", *node->nodeID);
    printf("Location: %s\n", node->location);
    printf("Traffic Conditions:\n");

    for (int i = 0; i < trafficData->dataCount; i++) {
        printf("  Condition %d: %s\n", i + 1, trafficData->conditionData[i].condition);
    }
}

int main() {
    // Example of node ID and location
    int nodeID = 101;
    TrafficNode node = {&nodeID, "Downtown Area"};

    // Create traffic data structure
    TrafficData trafficData;
    allocateTrafficData(&trafficData, 3);  // Assuming 3 different conditions for this node

    // Set some traffic conditions for the node
    setTrafficCondition(&trafficData, 0, "Clear");
    setTrafficCondition(&trafficData, 1, "Congested");
```

```
    setTrafficCondition(&trafficData, 2, "Clear");

    // Display traffic info for this node
    displayTrafficInfo(&node, &trafficData);

    // Free dynamically allocated memory for traffic data
    free(trafficData.conditionData);

    return 0;
}
```

7. Warehouse Slot Allocation System
Description:
Design a warehouse slot allocation system using structures for slot details and unions for item types. Use const pointers for slot identifiers and double pointers for dynamic slot management.
Specifications:
Structure: Slot details (ID, location, size).
Union: Item types (perishable, non-perishable).
const Pointer: Slot IDs.
Double Pointers: Dynamic slot allocation.

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

// Define the ItemType union to handle different item types (perishable/non-perishable)
union ItemType {
    char perishable[50];   // Perishable items (e.g., fruits, vegetables)
    char nonPerishable[50]; // Non-perishable items (e.g., electronics, furniture)
};

// Define the SlotDetails structure to store details of each slot
struct SlotDetails {
    const char *slotID;  // Slot ID (constant pointer to prevent modification)
    char location[100];  // Location within the warehouse
    float size;          // Size of the slot (in cubic meters)
    union ItemType item; // Item type stored in the slot (perishable or non-perishable)
};

// Function to dynamically allocate slots for the warehouse
void allocateSlots(struct SlotDetails ***slots, int numSlots) {
    *slots = (struct SlotDetails **)malloc(numSlots * sizeof(struct SlotDetails *));
    if (*slots == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    for (int i = 0; i < numSlots; i++) {
        (*slots)[i] = (struct SlotDetails *)malloc(sizeof(struct SlotDetails));
        if ((*slots)[i] == NULL) {
            printf("Memory allocation for slot %d failed\n", i);
            exit(1);
        }
    }
```

```
}

// Function to set slot details for a given slot
void setSlotDetails(struct SlotDetails *slot, const char *id, const char *location, float size, const char
*itemType, const char *itemName) {
    slot->slotID = id;
    strncpy(slot->location, location, sizeof(slot->location) - 1);
    slot->size = size;

    // Set item type and name based on the item type
    if (strcmp(itemType, "perishable") == 0) {
        strncpy(slot->item.perishable, itemName, sizeof(slot->item.perishable) - 1);
    } else if (strcmp(itemType, "non-perishable") == 0) {
        strncpy(slot->item.nonPerishable, itemName, sizeof(slot->item.nonPerishable) - 1);
    }
}

// Function to print slot details
void printSlotDetails(struct SlotDetails *slot) {
    printf("Slot ID: %s\n", slot->slotID);
    printf("Location: %s\n", slot->location);
    printf("Size: %.2f cubic meters\n", slot->size);

    printf("Item Type: ");
    if (strlen(slot->item.perishable) > 0) {
        printf("Perishable (Item: %s)\n", slot->item.perishable);
    } else {
        printf("Non-Perishable (Item: %s)\n", slot->item.nonPerishable);
    }
}

int main() {
    struct SlotDetails **slots;
    int numSlots = 3;

    // Allocate memory for slots dynamically
    allocateSlots(&slots, numSlots);

    // Set slot details
    setSlotDetails(slots[0], "A1", "Row 1, Shelf 1", 50.5, "perishable", "Bananas");
    setSlotDetails(slots[1], "B2", "Row 2, Shelf 3", 120.0, "non-perishable", "Television");
    setSlotDetails(slots[2], "C3", "Row 3, Shelf 2", 80.0, "perishable", "Apples");

    // Print slot details
    for (int i = 0; i < numSlots; i++) {
        printf("\nSlot %d Details:\n", i + 1);
        printSlotDetails(slots[i]);
    }

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

```
    return 0;
}
```

8. Package Delivery Optimization Tool
Description:
Develop a package delivery optimization tool using structures for package details and unions for delivery methods. Use const pointers for package identifiers and double pointers to manage dynamic delivery routes.
Specifications:
Structure: Package details (ID, weight, destination).
Union: Delivery methods (standard, express).
const Pointer: Package IDs.
Double Pointers: Dynamic route management.

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

// Define a structure for package details
typedef struct {
    const char* packageID;  // Package ID (constant pointer)
    double weight;          // Package weight
    char destination[100];  // Destination address
} Package;

// Define a union for delivery methods
typedef union {
    char method[10];  // Delivery method (standard, express)
} DeliveryMethod;

// Function to manage dynamic routes
void manageRoutes(Package **routes, int *routeCount, Package *newPackage) {
    // Dynamically reallocate memory to store routes
    *routes = (Package *)realloc(*routes, (*routeCount + 1) * sizeof(Package));
    if (*routes == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Add new package to routes
    (*routes)[*routeCount] = *newPackage;
    (*routeCount)++;
}

// Function to display package details
void displayPackage(const Package *pkg) {
    printf("Package ID: %s\n", pkg->packageID);
    printf("Weight: %.2f\n", pkg->weight);
    printf("Destination: %s\n", pkg->destination);
}

// Function to display all routes
void displayRoutes(Package *routes, int routeCount) {
    printf("Displaying all delivery routes:\n");
```

```
    for (int i = 0; i < routeCount; i++) {
        displayPackage(&routes[i]);
        printf("-------------------------\n");
    }
}

int main() {
    Package *routes = NULL;  // Dynamic array of packages (routes)
    int routeCount = 0;      // Counter for number of routes

    // Creating some package instances
    Package pkg1 = {"PKG001", 5.5, "123 Main St"};
    Package pkg2 = {"PKG002", 2.3, "456 Elm St"};
    Package pkg3 = {"PKG003", 7.0, "789 Pine St"};

    // Creating a union for delivery methods
    DeliveryMethod method1;
    strcpy(method1.method, "standard");

    DeliveryMethod method2;
    strcpy(method2.method, "express");

    // Adding packages to the delivery routes
    manageRoutes(&routes, &routeCount, &pkg1);
    manageRoutes(&routes, &routeCount, &pkg2);
    manageRoutes(&routes, &routeCount, &pkg3);

    // Displaying the current delivery routes
    displayRoutes(routes, routeCount);

    // Free dynamically allocated memory
    free(routes);

    return 0;
}
```

9. Logistics Data Analytics System
Description:
Create a logistics data analytics system using structures for analytics records and unions for different metrics. Use const pointers to ensure data integrity and double pointers for managing dynamic analytics data.
Specifications:
Structure: Analytics records (timestamp, metric).
Union: Metrics (speed, efficiency).
const Pointer: Analytics data.
Double Pointers: Dynamic data storage.

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

// Union for storing different types of metrics
union Metrics {
    float speed;       // Speed in km/h
    float efficiency;  // Efficiency as a percentage
```

```
};

// Structure to hold analytics records with a timestamp and a metric
struct AnalyticsRecord {
    time_t timestamp;  // Timestamp for the record
    union Metrics metric;  // The metric for the record
    char metricType;  // 'S' for speed, 'E' for efficiency
};

// Function to dynamically allocate an array of analytics records
void createAnalyticsData(struct AnalyticsRecord ***data, int numRecords) {
    // Allocate memory for the array of pointers to AnalyticsRecord
    *data = (struct AnalyticsRecord **)malloc(numRecords * sizeof(struct AnalyticsRecord *));

    for (int i = 0; i < numRecords; i++) {
        (*data)[i] = (struct AnalyticsRecord *)malloc(sizeof(struct AnalyticsRecord));
    }
}

// Function to print a specific record (using const pointer for data integrity)
void printAnalyticsRecord(const struct AnalyticsRecord *record) {
    char timeStr[20];
    strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", localtime(&(record->timestamp)));

    printf("Timestamp: %s\n", timeStr);

    if (record->metricType == 'S') {
        printf("Speed: %.2f km/h\n", record->metric.speed);
    } else if (record->metricType == 'E') {
        printf("Efficiency: %.2f%%\n", record->metric.efficiency);
    }
}

// Function to free the dynamically allocated memory
void freeAnalyticsData(struct AnalyticsRecord ***data, int numRecords) {
    for (int i = 0; i < numRecords; i++) {
        free((*data)[i]);
    }
    free(*data);
}

int main() {
    struct AnalyticsRecord **analyticsData = NULL;
    int numRecords = 5;

    // Create an array of analytics records dynamically
    createAnalyticsData(&analyticsData, numRecords);

    // Populate the data
    for (int i = 0; i < numRecords; i++) {
        analyticsData[i]->timestamp = time(NULL) - (i * 3600); // Each record is an hour apart

        if (i % 2 == 0) {
            analyticsData[i]->metricType = 'S';  // Speed
            analyticsData[i]->metric.speed = 50.5 + i;  // Example speed in km/h
```

```
        } else {
            analyticsData[i]->metricType = 'E';  // Efficiency
            analyticsData[i]->metric.efficiency = 90.0 - i;  // Example efficiency in percentage
        }
    }

    // Print the analytics records
    for (int i = 0; i < numRecords; i++) {
        printAnalyticsRecord(analyticsData[i]);
    }

    // Free the dynamically allocated memory
    freeAnalyticsData(&analyticsData, numRecords);

    return 0;
}
```

10. Transportation Schedule Management
Description:
Implement a transportation schedule management system using structures for schedule details and unions for transport types. Use const pointers for schedule IDs and double pointers for dynamic schedule lists.
Specifications:
Structure: Schedule details (ID, start time, end time).
Union: Transport types (bus, truck).
const Pointer: Schedule IDs.
Double Pointers: Dynamic schedule handling.

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

// Structure for schedule details
typedef struct {
    int id;
    char start_time[10]; // Format: HH:MM:SS
    char end_time[10];   // Format: HH:MM:SS
} Schedule;

// Union for transport types
typedef union {
    char bus[20];   // Bus type
    char truck[20]; // Truck type
} TransportType;

// Structure combining schedule and transport type
typedef struct {
    Schedule schedule;
    TransportType transport;
    int is_bus; // Flag to identify if the transport type is bus (1) or truck (0)
} TransportSchedule;

// Function to dynamically create a schedule list
void createScheduleList(TransportSchedule ***scheduleList, int *size) {
```

```
    printf("Enter the number of schedules to manage: ");
    scanf("%d", size);

    *scheduleList = (TransportSchedule **)malloc(sizeof(TransportSchedule *) * (*size));
    if (*scheduleList == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    for (int i = 0; i < *size; i++) {
        (*scheduleList)[i] = (TransportSchedule *)malloc(sizeof(TransportSchedule));
        if ((*scheduleList)[i] == NULL) {
            printf("Memory allocation failed for schedule %d!\n", i + 1);
            exit(1);
        }

        // Getting schedule details
        printf("\nEnter details for Schedule %d:\n", i + 1);
        printf("Schedule ID: ");
        scanf("%d", &((*scheduleList)[i]->schedule.id));
        printf("Start Time (HH:MM:SS): ");
        scanf("%s", (*scheduleList)[i]->schedule.start_time);
        printf("End Time (HH:MM:SS): ");
        scanf("%s", (*scheduleList)[i]->schedule.end_time);

        // Select transport type
        printf("Enter transport type (1 for Bus, 0 for Truck): ");
        scanf("%d", &((*scheduleList)[i]->is_bus));
        if ((*scheduleList)[i]->is_bus) {
            printf("Enter Bus name: ");
            scanf("%s", (*scheduleList)[i]->transport.bus);
        } else {
            printf("Enter Truck name: ");
            scanf("%s", (*scheduleList)[i]->transport.truck);
        }
    }
}

// Function to display the schedule list
void displayScheduleList(TransportSchedule **scheduleList, int size) {
    printf("\n--- Transportation Schedules ---\n");
    for (int i = 0; i < size; i++) {
        printf("\nSchedule ID: %d\n", scheduleList[i]->schedule.id);
        printf("Start Time: %s\n", scheduleList[i]->schedule.start_time);
        printf("End Time: %s\n", scheduleList[i]->schedule.end_time);

        if (scheduleList[i]->is_bus) {
            printf("Transport Type: Bus\n");
            printf("Bus Name: %s\n", scheduleList[i]->transport.bus);
        } else {
            printf("Transport Type: Truck\n");
            printf("Truck Name: %s\n", scheduleList[i]->transport.truck);
        }
    }
}
```

```
int main() {
    TransportSchedule **scheduleList = NULL;
    int size = 0;

    // Create the schedule list dynamically
    createScheduleList(&scheduleList, &size);

    // Display the schedules
    displayScheduleList(scheduleList, size);

    // Free the allocated memory
    for (int i = 0; i < size; i++) {
        free(scheduleList[i]);
    }
    free(scheduleList);

    return 0;
}
```

11. Dynamic Supply Chain Modeling
Description:
Develop a dynamic supply chain modeling tool using structures for supplier and customer details, and unions for transaction types. Use const pointers for transaction IDs and double pointers for dynamic relationship management.
Specifications:
Structure: Supplier/customer details (ID, name).
Union: Transaction types (purchase, return).
const Pointer: Transaction IDs.
Double Pointers: Dynamic supply chain modeling.

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

// Structure for supplier/customer details
typedef struct {
    int id;
    char name[100];
} Entity;

// Union for different transaction types
union Transaction {
    double purchaseAmount;
    double returnAmount;
};

// Transaction ID structure containing a transaction ID and a union for transaction details
typedef struct {
    const char *transactionID;
    union Transaction txnDetails;
    enum { PURCHASE, RETURN } txnType;
} TransactionRecord;
```

```
// Function to create a supplier or customer
Entity* createEntity(int id, const char *name) {
    Entity *newEntity = (Entity *)malloc(sizeof(Entity));
    newEntity->id = id;
    strncpy(newEntity->name, name, sizeof(newEntity->name) - 1);
    newEntity->name[sizeof(newEntity->name) - 1] = '\0';
    return newEntity;
}

// Function to create a transaction record (purchase or return)
TransactionRecord* createTransaction(const char *transactionID, enum { PURCHASE, RETURN }
txnType, double amount) {
    TransactionRecord *newTransaction = (TransactionRecord *)malloc(sizeof(TransactionRecord));
    newTransaction->transactionID = transactionID; // const pointer to transaction ID
    newTransaction->txnType = txnType;

    if (txnType == PURCHASE) {
        newTransaction->txnDetails.purchaseAmount = amount;
    } else if (txnType == RETURN) {
        newTransaction->txnDetails.returnAmount = amount;
    }

    return newTransaction;
}

// Function to print the transaction record
void printTransaction(TransactionRecord *txnRecord) {
    printf("Transaction ID: %s\n", txnRecord->transactionID);
    if (txnRecord->txnType == PURCHASE) {
        printf("Purchase Amount: %.2f\n", txnRecord->txnDetails.purchaseAmount);
    } else if (txnRecord->txnType == RETURN) {
        printf("Return Amount: %.2f\n", txnRecord->txnDetails.returnAmount);
    }
}

// Function to manage dynamic relationships between suppliers/customers and transactions
void manageSupplyChain(Entity **entities, int entityCount, TransactionRecord ***transactions, int
txnCount) {
    // Example of dynamic relationship handling
    for (int i = 0; i < entityCount; i++) {
        printf("Entity ID: %d, Name: %s\n", entities[i]->id, entities[i]->name);
        for (int j = 0; j < txnCount; j++) {
            // For simplicity, printing transaction info related to an entity
            printTransaction(transactions[i][j]);
        }
    }
}

int main() {
    // Create dynamic supply chain entities (suppliers and customers)
    Entity *supplier1 = createEntity(1, "Supplier A");
    Entity *customer1 = createEntity(2, "Customer A");

    // Create a dynamic array of entities (example: 2 entities)
    Entity *entities[] = {supplier1, customer1};
```

```
    // Create transactions for these entities (purchase and return)
    TransactionRecord *txn1 = createTransaction("TXN001", PURCHASE, 1000.0);
    TransactionRecord *txn2 = createTransaction("TXN002", RETURN, 150.0);

    // Dynamic array of transactions for each entity
    TransactionRecord **transactions = (TransactionRecord **)malloc(2 * sizeof(TransactionRecord *));
    transactions[0] = (TransactionRecord **)malloc(1 * sizeof(TransactionRecord *));
    transactions[0][0] = txn1; // Supplier has purchase transaction

    transactions[1] = (TransactionRecord **)malloc(1 * sizeof(TransactionRecord *));
    transactions[1][0] = txn2; // Customer has return transaction

    // Manage the supply chain and print details
    manageSupplyChain(entities, 2, transactions, 1);

    // Cleanup allocated memory
    free(supplier1);
    free(customer1);
    free(txn1);
    free(txn2);
    free(transactions[0]);
    free(transactions[1]);
    free(transactions);

    return 0;
}
```

12. Freight Cost Calculation System
Description:
Create a freight cost calculation system using structures for cost components and unions for different pricing models. Use const pointers for fixed cost parameters and double pointers for dynamically allocated cost records.
Specifications:
Structure: Cost components (ID, base cost).
Union: Pricing models (fixed, variable).
const Pointer: Cost parameters.
Double Pointers: Dynamic cost management.

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

#define MAX_COSTS 5 // Number of different freight cost records

// Structure to represent cost components
struct CostComponent {
    int id;          // Cost component ID
    double baseCost;  // Base cost for the component
};

// Union to represent different pricing models (Fixed and Variable)
union PricingModel {
    double fixedCost;      // Fixed pricing model
    double variableRate;   // Variable pricing model (e.g., per kg or distance)
```

```
};

// Structure that combines the cost component and pricing model
struct FreightCost {
    struct CostComponent component;  // Cost component details
    union PricingModel pricing;      // Pricing model (fixed/variable)
    int isFixed;                     // Flag to check if pricing is fixed (1) or variable (0)
};

// Function to print the details of a Freight Cost record
void printFreightCost(const struct FreightCost* freightCost) {
    printf("Cost Component ID: %d\n", freightCost->component.id);
    printf("Base Cost: %.2f\n", freightCost->component.baseCost);

    if (freightCost->isFixed) {
        printf("Pricing Model: Fixed, Cost: %.2f\n", freightCost->pricing.fixedCost);
    } else {
        printf("Pricing Model: Variable, Rate: %.2f\n", freightCost->pricing.variableRate);
    }
    printf("\n");
}

int main() {
    // Const pointer for fixed cost parameters
    const double fixedCostParameter = 100.0;

    // Dynamically allocate memory for cost records using double pointers
    struct FreightCost** costs = (struct FreightCost**) malloc(MAX_COSTS * sizeof(struct FreightCost*));
    if (costs == NULL) {
        printf("Memory allocation failed!\n");
        return -1;
    }

    // Initializing cost records dynamically
    for (int i = 0; i < MAX_COSTS; i++) {
        costs[i] = (struct FreightCost*) malloc(sizeof(struct FreightCost));
        if (costs[i] == NULL) {
            printf("Memory allocation for FreightCost failed!\n");
            return -1;
        }

        // Setting up the cost components and pricing models
        costs[i]->component.id = i + 1;
        costs[i]->component.baseCost = (i + 1) * 50.0;  // Example: base cost increases with index

        // Set fixed or variable pricing model
        if (i % 2 == 0) { // Fixed pricing for even indices
            costs[i]->pricing.fixedCost = fixedCostParameter;
            costs[i]->isFixed = 1;
        } else { // Variable pricing for odd indices
            costs[i]->pricing.variableRate = (i + 1) * 10.0;  // Example variable rate
            costs[i]->isFixed = 0;
        }
    }
```

```
    // Print all the FreightCost records
    for (int i = 0; i < MAX_COSTS; i++) {
        printFreightCost(costs[i]);
    }

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

    return 0;
}
```

13. Vehicle Load Balancing System
Description:
Design a vehicle load balancing system using structures for load details and unions for load types. Use const pointers for load identifiers and double pointers for managing dynamic load distribution.
Specifications:
Structure: Load details (ID, weight, destination).
Union: Load types (bulk, container).
const Pointer: Load IDs.
Double Pointers: Dynamic load handling.

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

// Define Union for Load Types
union LoadType {
    char bulk[50];     // Bulk load description
    char container[50]; // Container load description
};

// Define Structure for Load Details
struct LoadDetails {
    const int* loadID;          // Constant pointer for load ID (constant because it cannot be changed)
    double weight;              // Weight of the load
    char destination[100];      // Destination of the load
    union LoadType loadType;    // Load type (bulk or container)
};

// Function to dynamically allocate loads
void distributeLoad(struct LoadDetails ***loads, int numLoads) {
    *loads = (struct LoadDetails **)malloc(numLoads * sizeof(struct LoadDetails *));

    for (int i = 0; i < numLoads; i++) {
        (*loads)[i] = (struct LoadDetails *)malloc(sizeof(struct LoadDetails));
    }
}

// Function to release dynamically allocated memory
void freeLoadDistribution(struct LoadDetails ***loads, int numLoads) {
    for (int i = 0; i < numLoads; i++) {
        free((*loads)[i]);
```

```
    }
    free(*loads);
}

// Function to assign load details
void assignLoad(struct LoadDetails *load, const int *loadID, double weight, const char *destination, const
char *loadDescription, int isBulk) {
    load->loadID = loadID;         // Set constant pointer for load ID
    load->weight = weight;         // Set load weight
    strncpy(load->destination, destination, sizeof(load->destination));  // Set destination

    if (isBulk) {
        strncpy(load->loadType.bulk, loadDescription, sizeof(load->loadType.bulk));  // Set bulk load
description
    } else {
        strncpy(load->loadType.container, loadDescription, sizeof(load->loadType.container));  // Set
container load description
    }
}

// Function to display load details
void displayLoad(const struct LoadDetails *load) {
    printf("Load ID: %d\n", *(load->loadID));
    printf("Weight: %.2f\n", load->weight);
    printf("Destination: %s\n", load->destination);

    if (load->loadType.bulk[0] != '\0') {
        printf("Load Type: Bulk\n");
        printf("Bulk Description: %s\n", load->loadType.bulk);
    } else {
        printf("Load Type: Container\n");
        printf("Container Description: %s\n", load->loadType.container);
    }
    printf("\n");
}

int main() {
    int loadID1 = 101, loadID2 = 102;
    struct LoadDetails **loads;
    int numLoads = 2;

    // Dynamically allocate memory for loads
    distributeLoad(&loads, numLoads);

    // Assign details to the loads
    assignLoad(loads[0], &loadID1, 1500.0, "New York", "Steel", 1);  // Bulk load
    assignLoad(loads[1], &loadID2, 1200.0, "Los Angeles", "Electronics", 0);  // Container load

    // Display load details
    for (int i = 0; i < numLoads; i++) {
        displayLoad(loads[i]);
    }

    // Free dynamically allocated memory
    freeLoadDistribution(&loads, numLoads);
```

```
    return 0;
}
```

14. Intermodal Transport Management System
Description:
Implement an intermodal transport management system using structures for transport details and unions for transport modes. Use const pointers for transport identifiers and double pointers for dynamic transport route management.
Specifications:
Structure: Transport details (ID, origin, destination).
Union: Transport modes (rail, road).
const Pointer: Transport IDs.
Double Pointers: Dynamic transport management.

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

// Define maximum length for strings
#define MAX_LENGTH 100

// Define a structure for transport details
struct TransportDetails {
    const char *ID;        // Transport identifier (constant pointer)
    const char *origin;    // Origin of transport
    const char *destination; // Destination of transport
};

// Define a union for transport modes (rail or road)
union TransportModes {
    char rail[MAX_LENGTH]; // Rail mode of transport
    char road[MAX_LENGTH]; // Road mode of transport
};

// Structure for a transport route (includes both transport details and transport mode)
struct TransportRoute {
    struct TransportDetails details;
    union TransportModes mode;
    int isRail; // 1 for rail, 0 for road
};

// Function to create a new transport route dynamically using double pointers
void createTransportRoute(struct TransportRoute **route, const char *ID, const char *origin, const char
*destination, const char *mode) {
    // Allocate memory for a new transport route
    *route = (struct TransportRoute *)malloc(sizeof(struct TransportRoute));
    if (*route == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }

    // Assign details to the transport route
    (*route)->details.ID = ID;
    (*route)->details.origin = origin;
```

```
    (*route)->details.destination = destination;

    // Assign mode of transport
    if (strcmp(mode, "rail") == 0) {
        strcpy((*route)->mode.rail, "Rail Transport");
        (*route)->isRail = 1;
    } else if (strcmp(mode, "road") == 0) {
        strcpy((*route)->mode.road, "Road Transport");
        (*route)->isRail = 0;
    } else {
        printf("Invalid transport mode!\n");
        exit(1);
    }
}

// Function to display transport route details
void displayTransportRoute(struct TransportRoute *route) {
    printf("Transport ID: %s\n", route->details.ID);
    printf("Origin: %s\n", route->details.origin);
    printf("Destination: %s\n", route->details.destination);

    if (route->isRail) {
        printf("Mode of Transport: %s\n", route->mode.rail);
    } else {
        printf("Mode of Transport: %s\n", route->mode.road);
    }
}

// Function to free dynamically allocated memory for the transport route
void freeTransportRoute(struct TransportRoute *route) {
    free(route);
}

int main() {
    // Define pointers for dynamic transport route management
    struct TransportRoute *route1, *route2;

    // Create transport routes
    createTransportRoute(&route1, "T001", "New York", "Chicago", "rail");
    createTransportRoute(&route2, "T002", "Los Angeles", "San Francisco", "road");

    // Display transport route details
    printf("Route 1 Details:\n");
    displayTransportRoute(route1);

    printf("\nRoute 2 Details:\n");
    displayTransportRoute(route2);

    // Free dynamically allocated memory
    freeTransportRoute(route1);
    freeTransportRoute(route2);

    return 0;
}
```

15. Logistics Performance Monitoring
Description:
Develop a logistics performance monitoring system using structures for performance metrics and unions for different performance aspects. Use const pointers for metric identifiers and double pointers for managing dynamic performance records.
Specifications:
Structure: Performance metrics (ID, value).
Union: Performance aspects (time, cost).
const Pointer: Metric IDs.
Double Pointers: Dynamic performance tracking.

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

#define MAX_METRICS 10

// Union to hold different types of performance aspects (e.g., time and cost)
union PerformanceAspect {
    double time;  // Time spent for a particular task
    double cost;  // Cost incurred for the task
};

// Structure to define a performance metric
struct PerformanceMetric {
    const int *id;              // const pointer for the metric ID
    double value;               // value of the performance metric (time or cost)
    union PerformanceAspect aspect; // The performance aspect (time or cost)
};

// Function to create a new performance metric record dynamically
struct PerformanceMetric* createMetric(const int *id, double value, union PerformanceAspect aspect) {
    struct PerformanceMetric* newMetric = (struct PerformanceMetric*)malloc(sizeof(struct
PerformanceMetric));
    if (newMetric == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newMetric->id = id;          // Assign const pointer for ID
    newMetric->value = value;    // Set the value
    newMetric->aspect = aspect;  // Set the performance aspect (time or cost)
    return newMetric;
}

// Function to display the performance metric record
void displayMetric(const struct PerformanceMetric *metric) {
    printf("Metric ID: %d\n", *metric->id);
    printf("Performance Value: %.2f\n", metric->value);
    if (metric->id == NULL) {
        printf("Invalid Metric ID!\n");
    } else {
        printf("Performance Aspect (Time): %.2f\n", metric->aspect.time);
        printf("Performance Aspect (Cost): %.2f\n", metric->aspect.cost);
    }
}
```

```
int main() {
    // Sample metric IDs
    const int metricID1 = 1, metricID2 = 2;

    // Define union to store performance aspects (time and cost)
    union PerformanceAspect aspect1;
    aspect1.time = 2.5;   // Time taken for a task

    union PerformanceAspect aspect2;
    aspect2.cost = 100.75; // Cost of a task

    // Dynamic memory allocation for storing performance metrics
    struct PerformanceMetric *metrics[MAX_METRICS];
    metrics[0] = createMetric(&metricID1, 2.5, aspect1); // Metric 1 (Time)
    metrics[1] = createMetric(&metricID2, 100.75, aspect2); // Metric 2 (Cost)

    // Display the performance metrics
    displayMetric(metrics[0]);
    displayMetric(metrics[1]);

    // Free allocated memory
    free(metrics[0]);
    free(metrics[1]);

    return 0;
}
```

16. Warehouse Robotics Coordination
Description:
Create a system to coordinate warehouse robotics using structures for robot details and unions for task types. Use const pointers for robot identifiers and double pointers for managing dynamic task allocations.
Specifications:
Structure: Robot details (ID, type, status).
Union: Task types (picking, sorting).
const Pointer: Robot IDs.
Double Pointers: Dynamic task management.

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

// Structure to hold Robot details (ID, type, and status)
struct Robot {
    const char* id;     // Robot ID (constant pointer to prevent modification)
    char type[20];      // Robot type (e.g., picker, sorter)
    int status;         // 0 = inactive, 1 = active
};

// Union to represent different task types (Picking or Sorting)
union Task {
    char picking[50];  // Picking task details
    char sorting[50];  // Sorting task details
};

// Enum to define task type for better readability
```

```
enum TaskType {
    PICKING,
    SORTING
};

// Function to create and initialize a new robot
struct Robot* createRobot(const char* id, const char* type, int status) {
    struct Robot* newRobot = (struct Robot*)malloc(sizeof(struct Robot));
    newRobot->id = id;
    strcpy(newRobot->type, type);
    newRobot->status = status;
    return newRobot;
}

// Function to assign a task dynamically to robots using double pointers
void assignTask(struct Robot* robot, enum TaskType taskType, union Task** task) {
    *task = (union Task*)malloc(sizeof(union Task));  // Dynamically allocate task memory

    if (taskType == PICKING) {
        strcpy((*task)->picking, "Pick items from shelf A.");
    } else if (taskType == SORTING) {
        strcpy((*task)->sorting, "Sort items into bin B.");
    }

    printf("Task assigned to Robot ID: %s\n", robot->id);
    if (taskType == PICKING) {
        printf("Task: %s\n", (*task)->picking);
    } else if (taskType == SORTING) {
        printf("Task: %s\n", (*task)->sorting);
    }
}

// Function to print Robot details
void printRobotDetails(struct Robot* robot) {
    printf("Robot ID: %s\n", robot->id);
    printf("Type: %s\n", robot->type);
    printf("Status: %s\n", robot->status ? "Active" : "Inactive");
}

// Main function to demonstrate the system
int main() {
    // Create robots
    struct Robot* robot1 = createRobot("R001", "Picker", 1);
    struct Robot* robot2 = createRobot("R002", "Sorter", 0);

    // Print Robot details
    printRobotDetails(robot1);
    printRobotDetails(robot2);

    // Task management (dynamic allocation using double pointer)
    union Task* task1 = NULL;
    union Task* task2 = NULL;

    assignTask(robot1, PICKING, &task1);
    assignTask(robot2, SORTING, &task2);
```

```
    // Clean up dynamic memory
    free(task1);
    free(task2);
    free(robot1);
    free(robot2);

    return 0;
}
```

17. Customer Feedback Analysis System
Description:
Design a system to analyze customer feedback using structures for feedback details and unions for feedback types. Use const pointers for feedback IDs and double pointers for dynamically managing feedback data.
Specifications:
Structure: Feedback details (ID, content).
Union: Feedback types (positive, negative).
const Pointer: Feedback IDs.
Double Pointers: Dynamic feedback management.

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

#define MAX_CONTENT_LENGTH 256

// Structure to store feedback details
struct Feedback {
    const char *ID;  // Feedback ID (constant pointer)
    char content[MAX_CONTENT_LENGTH]; // Feedback content
};

// Union to categorize feedback types
union FeedbackType {
    char positive[MAX_CONTENT_LENGTH];
    char negative[MAX_CONTENT_LENGTH];
};

// Function to allocate feedback dynamically
void manageFeedback(struct Feedback **feedbackList, int *currentSize, int newSize) {
    *feedbackList = (struct Feedback *)realloc(*feedbackList, newSize * sizeof(struct Feedback));
    if (*feedbackList == NULL) {
        fprintf(stderr, "Memory allocation failed!\n");
        exit(1);
    }
    *currentSize = newSize;
}

// Function to add feedback
void addFeedback(struct Feedback **feedbackList, int *currentSize, const char *id, const char *content,
int index) {
    (*feedbackList)[index].ID = id;  // constant pointer to ID
    strncpy((*feedbackList)[index].content, content, MAX_CONTENT_LENGTH - 1);
    (*feedbackList)[index].content[MAX_CONTENT_LENGTH - 1] = '\0';  // Ensure null-termination
```

```
}

// Function to display feedback
void displayFeedback(struct Feedback *feedbackList, int size) {
    for (int i = 0; i < size; i++) {
        printf("Feedback ID: %s\n", feedbackList[i].ID);
        printf("Content: %s\n\n", feedbackList[i].content);
    }
}

int main() {
    struct Feedback *feedbackList = NULL;  // Double pointer for dynamic array of feedback
    int currentSize = 0;  // Current size of the feedback array

    // Dynamically manage feedback
    manageFeedback(&feedbackList, &currentSize, 2);

    // Add some feedback
    addFeedback(&feedbackList, &currentSize, "FB001", "Great product, very satisfied!", 0);
    addFeedback(&feedbackList, &currentSize, "FB002", "Terrible experience, will not buy again.", 1);

    // Display feedback
    displayFeedback(feedbackList, currentSize);

    // Free allocated memory
    free(feedbackList);

    return 0;
}
```

18. Real-Time Fleet Coordination
Description:
Implement a real-time fleet coordination system using structures for fleet details and unions for coordination types. Use const pointers for fleet IDs and double pointers for managing dynamic coordination data.
Specifications:
Structure: Fleet details (ID, location, status).
Union: Coordination types (dispatch, reroute).
const Pointer: Fleet IDs.
Double Pointers: Dynamic coordination.

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

// Define the structure for fleet details
typedef struct {
    const int *fleetID;  // Const pointer for fleet ID (won't be modified)
    char location[100];  // Location of the fleet
    char status[50];     // Status of the fleet
} FleetDetails;

// Define the union for coordination types
typedef union {
    char dispatch[100]; // Dispatch instructions
```

```
    char reroute[100];  // Reroute instructions
} CoordinationTypes;

// Define a structure for dynamic coordination data
typedef struct {
    CoordinationTypes *coordinationData; // Double pointer for dynamic coordination data
    int coordType;                  // Type of coordination (0 for dispatch, 1 for reroute)
} FleetCoordination;

// Function to create and initialize fleet details
FleetDetails* createFleetDetails(const int *fleetID, const char *location, const char *status) {
    FleetDetails *fleet = (FleetDetails *)malloc(sizeof(FleetDetails));
    fleet->fleetID = fleetID;  // Assign fleet ID pointer (const)
    strcpy(fleet->location, location);
    strcpy(fleet->status, status);
    return fleet;
}

// Function to initialize coordination data (dispatch or reroute)
FleetCoordination* createFleetCoordination(CoordinationTypes *coordinationData, int coordType) {
    FleetCoordination *coordination = (FleetCoordination *)malloc(sizeof(FleetCoordination));
    coordination->coordinationData = (CoordinationTypes *)malloc(sizeof(CoordinationTypes));
    *coordination->coordinationData = *coordinationData;  // Copy data into the dynamic memory
    coordination->coordType = coordType;
    return coordination;
}

// Function to display fleet details
void displayFleetDetails(FleetDetails *fleet) {
    printf("Fleet ID: %d\n", *(fleet->fleetID));
    printf("Location: %s\n", fleet->location);
    printf("Status: %s\n", fleet->status);
}

// Function to display coordination instructions
void displayCoordination(FleetCoordination *coordination) {
    if (coordination->coordType == 0) {  // Dispatch
        printf("Dispatch Instructions: %s\n", coordination->coordinationData->dispatch);
    } else if (coordination->coordType == 1) {  // Reroute
        printf("Reroute Instructions: %s\n", coordination->coordinationData->reroute);
    }
}

int main() {
    // Fleet ID (const)
    const int fleetID1 = 101;

    // Create fleet details
    FleetDetails *fleet1 = createFleetDetails(&fleetID1, "New York", "Active");

    // Coordination Data (Dispatch)
    CoordinationTypes coord1;
    strcpy(coord1.dispatch, "Dispatch to Los Angeles");

    // Create fleet coordination (dispatch type)
```

```
    FleetCoordination *coordination1 = createFleetCoordination(&coord1, 0);

    // Display fleet details and coordination data
    displayFleetDetails(fleet1);
    displayCoordination(coordination1);

    // Free allocated memory
    free(fleet1);
    free(coordination1->coordinationData);
    free(coordination1);

    return 0;
}
```

19. Logistics Security Management System
Description:
Develop a security management system for logistics using structures for security events and unions for event types. Use const pointers for event identifiers and double pointers for managing dynamic security data.
Specifications:
Structure: Security events (ID, description).
Union: Event types (breach, resolved).
const Pointer: Event IDs.
Double Pointers: Dynamic security event handling.

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

// Structure for storing security event details (ID, description)
typedef struct {
    const int eventID;          // Event identifier, marked as const (immutable)
    char description[256];      // Event description
} SecurityEvent;

// Union for event types (breach, resolved)
typedef union {
    char breachDetails[256];    // Information about the breach
    char resolutionDetails[256]; // Information about the resolution
} EventType;

// Structure for event, which includes both event details and event type
typedef struct {
    SecurityEvent event;        // Event information (ID and description)
    EventType eventType;        // Event type (breach or resolved)
    int isResolved;             // Flag to check if the event is resolved
} SecurityEventRecord;

// Function to create a new event
SecurityEventRecord* createEvent(int eventID, const char* description, const char* type, const char*
details) {
    SecurityEventRecord* newEvent = (SecurityEventRecord*)malloc(sizeof(SecurityEventRecord));
    if (newEvent == NULL) {
        printf("Memory allocation failed.\n");
        return NULL;
```

```
    }

    // Set event details
    newEvent->event.eventID = eventID;
    strncpy(newEvent->event.description, description, sizeof(newEvent->event.description) - 1);

    // Set event type and details
    if (strcmp(type, "breach") == 0) {
        strncpy(newEvent->eventType.breachDetails, details, sizeof(newEvent->eventType.breachDetails) -
1);
        newEvent->isResolved = 0;  // Breach is not resolved initially
    } else if (strcmp(type, "resolved") == 0) {
        strncpy(newEvent->eventType.resolutionDetails, details,
sizeof(newEvent->eventType.resolutionDetails) - 1);
        newEvent->isResolved = 1;  // Event is resolved
    } else {
        printf("Invalid event type.\n");
        free(newEvent);
        return NULL;
    }

    return newEvent;
}

// Function to display event information
void displayEvent(const SecurityEventRecord* event) {
    if (event == NULL) {
        printf("Event is NULL.\n");
        return;
    }

    printf("Event ID: %d\n", event->event.eventID);
    printf("Description: %s\n", event->event.description);

    if (event->isResolved) {
        printf("Event Status: Resolved\n");
        printf("Resolution Details: %s\n", event->eventType.resolutionDetails);
    } else {
        printf("Event Status: Breach\n");
        printf("Breach Details: %s\n", event->eventType.breachDetails);
    }
}

// Function to free memory for an event
void freeEvent(SecurityEventRecord* event) {
    if (event != NULL) {
        free(event);
    }
}

// Main function demonstrating the use of the system
int main() {
    // Creating some events
    SecurityEventRecord* breachEvent = createEvent(101, "Unauthorized Access Attempt", "breach",
"Detected at Warehouse A at 2 AM");
```

```
    SecurityEventRecord* resolvedEvent = createEvent(102, "Intruder Detected", "resolved", "Intruder
detained, no damage");

    // Displaying events
    printf("=== Breach Event ===\n");
    displayEvent(breachEvent);

    printf("\n=== Resolved Event ===\n");
    displayEvent(resolvedEvent);

    // Free allocated memory
    freeEvent(breachEvent);
    freeEvent(resolvedEvent);

    return 0;
}
```

20. Automated Billing System for Logistics
Description:
Create an automated billing system using structures for billing details and unions for payment methods. Use const pointers for bill IDs and double pointers for dynamically managing billing records.
Specifications:
Structure: Billing details (ID, amount, date).
Union: Payment methods (bank transfer, cash).
const Pointer: Bill IDs.
Double Pointers: Dynamic billing management.

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

#define MAX_DATE_LENGTH 11  // Format: "YYYY-MM-DD"

// Structure to hold billing details
struct BillingDetails {
    const int *billID;    // Constant pointer for bill ID
    double amount;        // Bill amount
    char date[MAX_DATE_LENGTH];  // Billing date
};

// Union for payment methods (bank transfer or cash)
union PaymentMethod {
    char bankTransfer[50]; // Bank account details
    double cashAmount;     // Cash amount
};

// Structure to hold a billing record with a payment method
struct BillingRecord {
    struct BillingDetails billingDetails;
    union PaymentMethod paymentMethod;
    int paymentType;  // 0 for bank transfer, 1 for cash
};

// Function to create a new billing record
void createBillingRecord(struct BillingRecord **billingRecords, int *recordCount, const int *billID, double
```

```
amount, const char *date, int paymentType, void *paymentDetails) {
    // Reallocate memory for new record
    *billingRecords = (struct BillingRecord *) realloc(*billingRecords, (*recordCount + 1) * sizeof(struct
BillingRecord));

    // Assign bill ID, amount, and date to the new record
    (*billingRecords)[*recordCount].billingDetails.billID = billID;
    (*billingRecords)[*recordCount].billingDetails.amount = amount;
    strncpy((*billingRecords)[*recordCount].billingDetails.date, date, MAX_DATE_LENGTH);

    // Assign payment method
    (*billingRecords)[*recordCount].paymentType = paymentType;
    if (paymentType == 0) {
        // Bank Transfer: Copy bank account details
        strncpy((*billingRecords)[*recordCount].paymentMethod.bankTransfer, (char *)paymentDetails, 50);
    } else if (paymentType == 1) {
        // Cash Payment: Assign the cash amount
        (*billingRecords)[*recordCount].paymentMethod.cashAmount = *(double *)paymentDetails;
    }

    // Increase the record count
    (*recordCount)++;
}

// Function to display a billing record
void displayBillingRecord(struct BillingRecord record) {
    printf("Bill ID: %d\n", *(record.billingDetails.billID));
    printf("Amount: %.2f\n", record.billingDetails.amount);
    printf("Date: %s\n", record.billingDetails.date);

    if (record.paymentType == 0) {
        printf("Payment Method: Bank Transfer\n");
        printf("Bank Details: %s\n", record.paymentMethod.bankTransfer);
    } else if (record.paymentType == 1) {
        printf("Payment Method: Cash\n");
        printf("Cash Amount: %.2f\n", record.paymentMethod.cashAmount);
    }
    printf("\n");
}

int main() {
    struct BillingRecord *billingRecords = NULL;  // Pointer to an array of billing records
    int recordCount = 0;  // Number of records in the array

    // Sample bill ID and details
    int billID1 = 101;
    double cashAmount1 = 500.75;
    char date1[] = "2025-01-22";
    createBillingRecord(&billingRecords, &recordCount, &billID1, cashAmount1, date1, 1, &cashAmount1);
 // Cash payment

    int billID2 = 102;
    double bankTransferAmount2 = 1000.50;
    char bankDetails2[] = "Bank ABC, Account 12345";
    createBillingRecord(&billingRecords, &recordCount, &billID2, bankTransferAmount2, date1, 0,
```

```
bankDetails2);  // Bank transfer

    // Display the billing records
    for (int i = 0; i < recordCount; i++) {
        displayBillingRecord(billingRecords[i]);
    }

    // Free allocated memory
    free(billingRecords);
    return 0;
}
```

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

Vessel Navigation System
Description:
Design a navigation system that tracks a vessel’s current position and routes using structures and arrays. Use const pointers for immutable route coordinates and strings for location names. Double pointers handle dynamic route allocation.
Specifications:
Structure: Route details (start, end, waypoints).
Array: Stores multiple waypoints.
Strings: Names of locations.
const Pointers: Route coordinates.
Double Pointers: Dynamic allocation of routes.

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

#define MAX_WAYPOINTS 10

typedef struct {
    const double *start_latitude;
    const double *start_longitude;
    const double *end_latitude;
    const double *end_longitude;
    const char *start_location;
    const char *end_location;
    const char *waypoints[MAX_WAYPOINTS];
} Route;

void print_route(Route *route) {
    printf("Start Location: %s (%.2f, %.2f)\n", route->start_location, *route->start_latitude,
*route->start_longitude);
    printf("End Location: %s (%.2f, %.2f)\n", route->end_location, *route->end_latitude,
*route->end_longitude);
    printf("Waypoints:\n");
    for (int i = 0; i < MAX_WAYPOINTS && route->waypoints[i] != NULL; i++) {
        printf(" - %s\n", route->waypoints[i]);
    }
}

int main() {
    double start_lat = 34.0522, start_lon = -118.2437;
```

```
    double end_lat = 36.1699, end_lon = -115.1398;

    const double *start_lat_ptr = &start_lat;
    const double *start_lon_ptr = &start_lon;
    const double *end_lat_ptr = &end_lat;
    const double *end_lon_ptr = &end_lon;

    const char *start_loc = "Los Angeles";
    const char *end_loc = "Las Vegas";

    const char *waypoints[] = {"Waypoint 1", "Waypoint 2", "Waypoint 3", NULL};

    Route *route = (Route *)malloc(sizeof(Route));
    route->start_latitude = start_lat_ptr;
    route->start_longitude = start_lon_ptr;
    route->end_latitude = end_lat_ptr;
    route->end_longitude = end_lon_ptr;
    route->start_location = start_loc;
    route->end_location = end_loc;

    for (int i = 0; i < MAX_WAYPOINTS; i++) {
        route->waypoints[i] = waypoints[i];
    }

    print_route(route);

    free(route);
    return 0;
}
```

2. Fleet Management Software
Description:
Develop a system to manage multiple vessels in a fleet, using arrays for storing fleet data and structures for vessel details. Unions represent variable attributes like cargo type or passenger count.
Specifications:
Structure: Vessel details (name, ID, type).
Union: Cargo type or passenger count.
Array: Fleet data.
const Pointers: Immutable vessel IDs.
Double Pointers: Manage dynamic fleet records.

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

#define MAX_VESSELS 5

typedef union {
    int passenger_count;
    char cargo_type[50];
} VesselAttributes;

typedef struct {
    char name[50];
    int id;
    char type[20];
```

```
    VesselAttributes attributes;
} Vessel;

void print_vessel_details(Vessel *vessel) {
    printf("Vessel Name: %s\n", vessel->name);
    printf("Vessel ID: %d\n", vessel->id);
    printf("Vessel Type: %s\n", vessel->type);

    if (vessel->type[0] == 'C') { // Cargo vessel
        printf("Cargo Type: %s\n", vessel->attributes.cargo_type);
    } else { // Passenger vessel
        printf("Passenger Count: %d\n", vessel->attributes.passenger_count);
    }
}

int main() {
    Vessel *fleet = (Vessel *)malloc(MAX_VESSELS * sizeof(Vessel));

    for (int i = 0; i < MAX_VESSELS; i++) {
        snprintf(fleet[i].name, sizeof(fleet[i].name), "Vessel_%d", i);
        fleet[i].id = i + 100;
        snprintf(fleet[i].type, sizeof(fleet[i].type), (i % 2 == 0) ? "Cargo" : "Passenger");

        if (i % 2 == 0) { // Cargo vessel
            snprintf(fleet[i].attributes.cargo_type, sizeof(fleet[i].attributes.cargo_type), "General Cargo");
        } else { // Passenger vessel
            fleet[i].attributes.passenger_count = 100 + i;
        }

        print_vessel_details(&fleet[i]);
    }

    free(fleet);
    return 0;
}
```

3. Ship Maintenance Scheduler
Description:
Create a scheduler for ship maintenance tasks. Use structures to define tasks and arrays for schedules. Utilize double pointers for managing dynamic task lists.
Specifications:
Structure: Maintenance task (ID, description, schedule).
Array: Maintenance schedules.
const Pointers: Read-only task IDs.
Double Pointers: Dynamic task lists.

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

#define MAX_TASKS 5

typedef struct {
    int task_id;
    char description[100];
```

```
    char schedule[50];
} MaintenanceTask;

void print_maintenance_task(MaintenanceTask *task) {
    printf("Task ID: %d\n", task->task_id);
    printf("Description: %s\n", task->description);
    printf("Schedule: %s\n", task->schedule);
}

int main() {
    MaintenanceTask **tasks = (MaintenanceTask **)malloc(MAX_TASKS * sizeof(MaintenanceTask *));

    for (int i = 0; i < MAX_TASKS; i++) {
        tasks[i] = (MaintenanceTask *)malloc(sizeof(MaintenanceTask));
        tasks[i]->task_id = i + 1;
        snprintf(tasks[i]->description, sizeof(tasks[i]->description), "Task %d description", i + 1);
        snprintf(tasks[i]->schedule, sizeof(tasks[i]->schedule), "2025-01-%02d", i + 1);

        print_maintenance_task(tasks[i]);
    }

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

    return 0;
}
```

4. Cargo Loading Optimization
Description:
Design a system to optimize cargo loading using arrays for storing cargo weights and structures for vessel specifications. Unions represent variable cargo properties like dimensions or temperature requirements.
Specifications:
Structure: Vessel specifications (capacity, dimensions).
Union: Cargo properties (weight, dimensions).
Array: Cargo data.
const Pointers: Protect cargo data.
Double Pointers: Dynamic cargo list allocation.

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

#define MAX_CARGO 5

typedef struct {
    double capacity;
    double length;
    double width;
    double height;
} VesselSpecs;

typedef union {
```

```
    double weight;
    struct {
        double length;
        double width;
        double height;
    } dimensions;
} CargoProperties;

typedef struct {
    VesselSpecs specs;
    CargoProperties cargo;
} CargoItem;

void print_cargo_details(CargoItem *cargo_item) {
    printf("Vessel Capacity: %.2f tons\n", cargo_item->specs.capacity);
    printf("Cargo Weight: %.2f tons\n", cargo_item->cargo.weight);
}

int main() {
    CargoItem **cargo_list = (CargoItem **)malloc(MAX_CARGO * sizeof(CargoItem *));

    for (int i = 0; i < MAX_CARGO; i++) {
        cargo_list[i] = (CargoItem *)malloc(sizeof(CargoItem));
        cargo_list[i]->specs.capacity = 500.0;
        cargo_list[i]->cargo.weight = 100.0 + i * 10;

        print_cargo_details(cargo_list[i]);
    }

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

    return 0;
}
```

5. Real-Time Weather Alert System
Description:
Develop a weather alert system for ships using strings for alert messages, structures for weather data, and arrays for historical records.
Specifications:
Structure: Weather data (temperature, wind speed).
Array: Historical records.
Strings: Alert messages.
const Pointers: Protect alert details.
Double Pointers: Dynamic weather record management.

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

#define MAX_RECORDS 5
```

```
typedef struct {
    double temperature;
    double wind_speed;
} WeatherData;

void print_weather_alert(WeatherData *data, const char *alert_message) {
    printf("Weather Alert: %s\n", alert_message);
    printf("Temperature: %.2f°C\n", data->temperature);
    printf("Wind Speed: %.2f km/h\n", data->wind_speed);
}

int main() {
    WeatherData **weather_records = (WeatherData **)malloc(MAX_RECORDS * sizeof(WeatherData
*));
    const char *alert_message = "Severe Weather Alert!";

    for (int i = 0; i < MAX_RECORDS; i++) {
        weather_records[i] = (WeatherData *)malloc(sizeof(WeatherData));
        weather_records[i]->temperature = 20.0 + i;
        weather_records[i]->wind_speed = 15.0 + i * 2;

        print_weather_alert(weather_records[i], alert_message);
    }

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

    return 0;
}
```

6. Nautical Chart Management
Description:
Implement a nautical chart management system using arrays for coordinates and structures for chart metadata. Use unions for depth or hazard data.
Specifications:
Structure: Chart metadata (ID, scale, region).
Union: Depth or hazard data.
Array: Coordinate points.
const Pointers: Immutable chart IDs.
Double Pointers: Manage dynamic charts.

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

#define MAX_COORDINATES 100

// Structure to hold chart metadata
typedef struct {
    int id;
    double scale;
    char region[50];
```

```
} ChartMetadata;

// Union to hold either depth or hazard information
typedef union {
    double depth;
    char hazard[50];
} DepthOrHazard;

// Structure to represent a coordinate point
typedef struct {
    double latitude;
    double longitude;
} Coordinate;

// Chart structure
typedef struct {
    ChartMetadata metadata;
    Coordinate coordinates[MAX_COORDINATES];
    DepthOrHazard data;
    int numCoordinates;
} NauticalChart;

// Constants for immutable chart IDs
const int chartId = 101;

// Double pointers for dynamic memory allocation of charts
NauticalChart **charts;

void createChart(int id, double scale, const char* region, NauticalChart* chart) {
    chart->metadata.id = id;
    chart->metadata.scale = scale;
    snprintf(chart->metadata.region, sizeof(chart->metadata.region), "%s", region);
}

void displayChart(NauticalChart* chart) {
    printf("Chart ID: %d, Region: %s, Scale: %.2f\n", chart->metadata.id, chart->metadata.region,
chart->metadata.scale);
}

int main() {
    // Example chart creation
    NauticalChart chart1;
    createChart(1, 1.5, "Pacific Ocean", &chart1);
    displayChart(&chart1);

    return 0;
}
```

7. Crew Roster Management
Description:
Develop a system to manage ship crew rosters using strings for names, arrays for schedules, and structures for roles.
Specifications:
Structure: Crew details (name, role, schedule).
Array: Roster.

Strings: Crew names.
const Pointers: Protect role definitions.
Double Pointers: Dynamic roster allocation.

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

#define MAX_CREW 100

// Structure to hold crew details
typedef struct {
    char name[50];
    char role[50];
    char schedule[50];
} CrewMember;

// Array to store the crew roster
CrewMember roster[MAX_CREW];

// Constants to protect role definitions
const char *const roles[] = {"Captain", "Engineer", "Cook", "Navigator"};

void addCrewMember(int index, const char* name, const char* role, const char* schedule) {
    snprintf(roster[index].name, sizeof(roster[index].name), "%s", name);
    snprintf(roster[index].role, sizeof(roster[index].role), "%s", role);
    snprintf(roster[index].schedule, sizeof(roster[index].schedule), "%s", schedule);
}

void displayCrewRoster() {
    for (int i = 0; i < MAX_CREW; i++) {
        if (roster[i].name[0] != '\0') {
            printf("Name: %s, Role: %s, Schedule: %s\n", roster[i].name, roster[i].role, roster[i].schedule);
        }
    }
}

int main() {
    // Example crew member addition
    addCrewMember(0, "John Doe", "Captain", "Mon-Fri 8-5");
    addCrewMember(1, "Jane Smith", "Engineer", "Mon-Fri 9-6");

    displayCrewRoster();

    return 0;
}
```

8. Underwater Sensor Monitoring
Description:
Create a system for underwater sensor monitoring using arrays for readings, structures for sensor details, and unions for variable sensor types.
Specifications:
Structure: Sensor details (ID, location).
Union: Sensor types (temperature, pressure).

Array: Sensor readings.
const Pointers: Protect sensor IDs.
Double Pointers: Dynamic sensor lists.

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

// Structure to hold sensor details
typedef struct {
    int id;
    char location[50];
} Sensor;

// Union to store different sensor types
typedef union {
    double temperature;
    double pressure;
} SensorType;

// Array to store sensor readings
double sensorReadings[100];

// Constants to protect sensor IDs
const int sensorId = 202;

// Double pointers for dynamic sensor list
Sensor **sensors;

void addSensor(int id, const char* location, Sensor* sensor) {
    sensor->id = id;
    snprintf(sensor->location, sizeof(sensor->location), "%s", location);
}

void displaySensorData(Sensor* sensor, SensorType* type) {
    printf("Sensor ID: %d, Location: %s, Temperature: %.2f\n", sensor->id, sensor->location,
type->temperature);
}

int main() {
    // Example sensor creation and monitoring
    Sensor sensor1;
    SensorType type1;
    addSensor(1, "Deep Sea", &sensor1);
    type1.temperature = 15.5;
    displaySensorData(&sensor1, &type1);

    return 0;
}
```

9. Ship Log Management
Description:
Design a ship log system using strings for log entries, arrays for daily records, and structures for log metadata.
Specifications:
Structure: Log metadata (date, author).

Array: Daily log records.
Strings: Log entries.
const Pointers: Immutable metadata.
Double Pointers: Manage dynamic log entries.

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

#define MAX_LOGS 100

// Structure to hold log metadata
typedef struct {
    char date[11];
    char author[50];
} LogMetadata;

// Structure to hold individual log entries
typedef struct {
    LogMetadata metadata;
    char entry[200];
} ShipLog;

// Array to store daily log records
ShipLog logs[MAX_LOGS];

// Constants for immutable metadata
const char *const logDate = "2025-01-22";

void addLogEntry(int index, const char* date, const char* author, const char* entry) {
    snprintf(logs[index].metadata.date, sizeof(logs[index].metadata.date), "%s", date);
    snprintf(logs[index].metadata.author, sizeof(logs[index].metadata.author), "%s", author);
    snprintf(logs[index].entry, sizeof(logs[index].entry), "%s", entry);
}

void displayLog() {
    for (int i = 0; i < MAX_LOGS; i++) {
        if (logs[i].entry[0] != '\0') {
            printf("Date: %s, Author: %s, Log Entry: %s\n", logs[i].metadata.date, logs[i].metadata.author,
logs[i].entry);
        }
    }
}

int main() {
    // Example log entry
    addLogEntry(0, "2025-01-22", "Captain", "Departed port, heading towards new coordinates.");
    displayLog();

    return 0;
}
```

10. Navigation Waypoint Manager
Description:
Develop a waypoint management tool using arrays for storing waypoints, strings for waypoint names, and

structures for navigation details.
Specifications:
Structure: Navigation details (ID, waypoints).
Array: Waypoint data.
Strings: Names of waypoints.
const Pointers: Protect waypoint IDs.
Double Pointers: Dynamic waypoint storage.

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

#define MAX_WAYPOINTS 100

// Structure to hold navigation details
typedef struct {
    int id;
    char waypoints[50][100]; // Array of waypoint names
} NavigationDetails;

// Array to store waypoints
NavigationDetails waypoints[MAX_WAYPOINTS];

// Constants for immutable waypoint IDs
const int waypointId = 301;

void addWaypoint(int index, const char* name) {
    snprintf(waypoints[index].waypoints[index], sizeof(waypoints[index].waypoints[index]), "%s", name);
}

void displayWaypoints() {
    for (int i = 0; i < MAX_WAYPOINTS; i++) {
        if (waypoints[i].waypoints[i][0] != '\0') {
            printf("Waypoint: %s\n", waypoints[i].waypoints[i]);
        }
    }
}

int main() {
    // Example waypoint addition
    addWaypoint(0, "Point A");
    addWaypoint(1, "Point B");

    displayWaypoints();

    return 0;
}
```

11. Marine Wildlife Tracking
Description:
Create a system for tracking marine wildlife using structures for animal data and arrays for observation records.
Specifications:
Structure: Animal data (species, ID, location).
Array: Observation records.

Strings: Species names.
const Pointers: Protect species IDs.
Double Pointers: Manage dynamic tracking data.

```
#include <stdio.h>

typedef struct {
    char species[50];  // Species name
    int id;            // Unique ID for each animal
    double location[2];  // Latitude and Longitude
} AnimalData;

typedef struct {
    AnimalData *animal;  // Pointer to the tracked animal
    char date[20];       // Observation date
    char observer[50];   // Name of observer
} ObservationRecord;

int main() {
    AnimalData animal1 = {"Whale", 12345, {34.0522, -118.2437}};
    ObservationRecord records[100];

    // Example of setting up a record
    records[0].animal = &animal1;
    snprintf(records[0].date, sizeof(records[0].date), "2025-01-22");
    snprintf(records[0].observer, sizeof(records[0].observer), "John Doe");

    printf("Observed %s with ID %d at location (%f, %f) on %s by %s\n",
           records[0].animal->species, records[0].animal->id,
           records[0].animal->location[0], records[0].animal->location[1],
           records[0].date, records[0].observer);

    return 0;
}
```

12. Coastal Navigation Beacon Management
Description:
Design a system to manage coastal navigation beacons using structures for beacon metadata, arrays for signals, and unions for variable beacon types.
Specifications:
Structure: Beacon metadata (ID, type, location).
Union: Variable beacon types.
Array: Signal data.
const Pointers: Immutable beacon IDs.
Double Pointers: Dynamic beacon data management.

```
#include <stdio.h>

typedef union {
    int light_color;  // Color of the light for a beacon
    float signal_strength;  // Signal strength
    char sound_type[30];  // Type of sound for beacon
} BeaconType;

typedef struct {
```

```
    int id;
    char type[50];       // Type of beacon (e.g., lighthouse, buoy)
    double location[2];  // Location of the beacon (latitude, longitude)
    BeaconType beacon_data;  // Data for different beacon types
} BeaconMetadata;

int main() {
    BeaconMetadata beacon1 = {101, "Lighthouse", {33.8895, -118.2208}, .beacon_data.light_color = 5};

    printf("Beacon ID: %d, Type: %s, Location: (%f, %f), Light Color: %d\n",
           beacon1.id, beacon1.type, beacon1.location[0], beacon1.location[1],
beacon1.beacon_data.light_color);

    return 0;
}
```

13. Fuel Usage Tracking
Description:
Develop a fuel usage tracking system for ships using structures for fuel data and arrays for consumption logs.
Specifications:
Structure: Fuel data (type, quantity).
Array: Consumption logs.
Strings: Fuel types.
const Pointers: Protect fuel data.
Double Pointers: Dynamic fuel log allocation.

```
#include <stdio.h>

typedef struct {
    char fuel_type[30];  // Type of fuel
    double quantity;     // Quantity of fuel consumed
} FuelData;

typedef struct {
    FuelData *fuel;  // Pointer to fuel data
    char timestamp[20];  // Timestamp of consumption
} ConsumptionLog;

int main() {
    FuelData fuel1 = {"Diesel", 200.5};
    ConsumptionLog logs[100];

    // Example of a consumption log
    logs[0].fuel = &fuel1;
    snprintf(logs[0].timestamp, sizeof(logs[0].timestamp), "2025-01-22 14:00");

    printf("Fuel type: %s, Quantity: %.2f, Timestamp: %s\n",
           logs[0].fuel->fuel_type, logs[0].fuel->quantity, logs[0].timestamp);

    return 0;
}
```

14. Emergency Response System
Description:

Create an emergency response system using strings for messages, structures for response details, and arrays for alert history.
Specifications:
Structure: Response details (ID, location, type).
Array: Alert history.
Strings: Alert messages.
const Pointers: Protect emergency IDs.
Double Pointers: Dynamic alert allocation.

```
#include <stdio.h>

typedef struct {
    int id;  // Unique response ID
    double location[2];  // Latitude and Longitude of the emergency location
    char type[50];  // Type of emergency (e.g., fire, rescue)
} ResponseDetails;

typedef struct {
    char message[100];  // Alert message
    char timestamp[20]; // Timestamp of the alert
} AlertHistory;

int main() {
    ResponseDetails response1 = {1001, {40.7128, -74.0060}, "Fire"};
    AlertHistory alerts[100];

    // Example of an alert history record
    snprintf(alerts[0].message, sizeof(alerts[0].message), "Fire reported at location 40.7128, -74.0060.");
    snprintf(alerts[0].timestamp, sizeof(alerts[0].timestamp), "2025-01-22 15:00");

    printf("Emergency ID: %d, Type: %s, Location: (%f, %f), Message: %s\n",
           response1.id, response1.type, response1.location[0], response1.location[1], alerts[0].message);

    return 0;
}
```

15. Ship Performance Analysis
Description:
Design a system for ship performance analysis using arrays for performance metrics, structures for ship specifications, and unions for variable factors like weather impact.
Specifications:
Structure: Ship specifications (speed, capacity).
Union: Variable factors.
Array: Performance metrics.
const Pointers: Protect metric definitions.
Double Pointers: Dynamic performance records.

```
#include <stdio.h>

typedef struct {
    double speed;    // Ship speed in knots
    int capacity;    // Capacity in passengers or cargo weight
} ShipSpecifications;

typedef union {
    float wind_speed;  // Wind speed in knots
```

```
    float sea_condition; // Sea condition factor (1 to 5)
} PerformanceFactors;

typedef struct {
    ShipSpecifications specs;
    PerformanceFactors factors;
    double performance_metric;  // Performance metric based on conditions
} PerformanceMetrics;

int main() {
    ShipSpecifications ship1 = {25.5, 1000};  // Ship's speed in knots and capacity
    PerformanceMetrics metrics = {ship1, .factors.wind_speed = 15.0, 20.5};  // Wind speed is 15 knots

    printf("Ship Speed: %.2f knots, Capacity: %d, Wind Speed: %.2f knots, Performance Metric: %.2f\n",
           metrics.specs.speed, metrics.specs.capacity, metrics.factors.wind_speed,
metrics.performance_metric);

    return 0;
}
```

16. Port Docking Scheduler
Description:
Develop a scheduler for port docking using arrays for schedules, structures for port details, and strings for vessel names.
Specifications:
Structure: Port details (ID, capacity, location).
Array: Docking schedules.
Strings: Vessel names.
const Pointers: Protect schedule IDs.
Double Pointers: Manage dynamic schedules.

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

#define MAX_VESSEL_NAME_LENGTH 100

// Structure to store port details
typedef struct {
    int portID;
    int capacity;
    char location[100];
} Port;

// Structure to store docking schedule details
typedef struct {
    int scheduleID;
    int portID;
    char vesselName[MAX_VESSEL_NAME_LENGTH];
    char dockingTime[20];  // e.g., "2025-01-22 14:00"
} DockingSchedule;

// Function to protect schedule ID using a constant pointer
void printDockingSchedule(const DockingSchedule* schedule) {
    printf("Schedule ID: %d\n", schedule->scheduleID);
```

```
    printf("Port ID: %d\n", schedule->portID);
    printf("Vessel Name: %s\n", schedule->vesselName);
    printf("Docking Time: %s\n", schedule->dockingTime);
}

// Function to dynamically allocate a new docking schedule
DockingSchedule* createDockingSchedule(int scheduleID, int portID, const char* vesselName, const
char* dockingTime) {
    DockingSchedule* newSchedule = (DockingSchedule*)malloc(sizeof(DockingSchedule));
    if (newSchedule != NULL) {
        newSchedule->scheduleID = scheduleID;
        newSchedule->portID = portID;
        strncpy(newSchedule->vesselName, vesselName, MAX_VESSEL_NAME_LENGTH);
        strncpy(newSchedule->dockingTime, dockingTime, 20);
    }
    return newSchedule;
}

// Function to free dynamically allocated docking schedule
void freeDockingSchedule(DockingSchedule* schedule) {
    free(schedule);
}

// Function to add a docking schedule to a dynamic list
void addDockingSchedule(DockingSchedule*** scheduleList, int* scheduleCount, DockingSchedule*
newSchedule) {
    // Reallocate memory for the new docking schedule
    *scheduleList = (DockingSchedule**)realloc(*scheduleList, sizeof(DockingSchedule*) *
(*scheduleCount + 1));
    if (*scheduleList != NULL) {
        (*scheduleList)[*scheduleCount] = newSchedule;
        (*scheduleCount)++;
    }
}

// Function to print all docking schedules
void printAllSchedules(DockingSchedule** scheduleList, int scheduleCount) {
    for (int i = 0; i < scheduleCount; i++) {
        printDockingSchedule(scheduleList[i]);
        printf("\n");
    }
}

int main() {
    // Create some sample ports
    Port ports[] = {
        {1, 5, "New York"},
        {2, 3, "Los Angeles"},
        {3, 4, "San Francisco"}
    };

    int scheduleCount = 0;
    DockingSchedule** dockingSchedules = NULL;  // Double pointer to manage dynamic docking
schedule list
```

```
    // Adding docking schedules dynamically
    addDockingSchedule(&dockingSchedules, &scheduleCount, createDockingSchedule(1, 1, "Vessel A",
"2025-01-22 12:00"));
    addDockingSchedule(&dockingSchedules, &scheduleCount, createDockingSchedule(2, 2, "Vessel B",
"2025-01-22 13:00"));
    addDockingSchedule(&dockingSchedules, &scheduleCount, createDockingSchedule(3, 3, "Vessel C",
"2025-01-22 14:00"));

    // Print all docking schedules
    printf("Docking Schedules:\n");
    printAllSchedules(dockingSchedules, scheduleCount);

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

    return 0;
}
```

17. Deep-Sea Exploration Data Logger
Description:
Create a data logger for deep-sea exploration using structures for exploration data and arrays for logs.
Specifications:
Structure: Exploration data (depth, location, timestamp).
Array: Logs.
const Pointers: Protect data entries.
Double Pointers: Dynamic log storage.

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

// Define the structure for exploration data
struct ExplorationData {
    float depth;        // Depth in meters
    char location[50];  // Location (e.g., coordinates)
    time_t timestamp;    // Timestamp of the exploration event
};

// Function to create a timestamp
void createTimestamp(time_t *timestamp) {
    *timestamp = time(NULL);
}

// Function to log exploration data
void logExplorationData(struct ExplorationData **logs, int *logCount, float depth, const char *location) {
    // Reallocate memory for the logs (dynamically resizing the array)
    *logs = realloc(*logs, (*logCount + 1) * sizeof(struct ExplorationData));
    if (*logs == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        exit(1);
    }
```

```
    // Populate the new log entry
    struct ExplorationData newLog;
    newLog.depth = depth;
    strncpy(newLog.location, location, sizeof(newLog.location) - 1);
    newLog.location[sizeof(newLog.location) - 1] = '\0';  // Ensure null termination
    createTimestamp(&newLog.timestamp);

    // Store the new log entry
    (*logs)[*logCount] = newLog;
    (*logCount)++;
}

// Function to display all exploration logs
void displayLogs(struct ExplorationData *logs, int logCount) {
    for (int i = 0; i < logCount; i++) {
        printf("Log %d:\n", i + 1);
        printf("Depth: %.2f meters\n", logs[i].depth);
        printf("Location: %s\n", logs[i].location);
        printf("Timestamp: %s", ctime(&logs[i].timestamp));  // ctime returns a string of the time
        printf("\n");
    }
}

int main() {
    struct ExplorationData *logs = NULL;  // Pointer to the array of logs
    int logCount = 0;  // Counter to track the number of logs

    // Example of logging exploration data
    logExplorationData(&logs, &logCount, 1000.5, "39.7684 N, 86.1580 W");
    logExplorationData(&logs, &logCount, 1200.2, "40.7128 N, 74.0060 W");

    // Display the logs
    displayLogs(logs, logCount);

    // Free dynamically allocated memory
    free(logs);

    return 0;
}
```

18. Ship Communication System
Description:
Develop a ship communication system using strings for messages, structures for communication metadata, and arrays for message logs.
Specifications:
Structure: Communication metadata (ID, timestamp).
Array: Message logs.
Strings: Communication messages.
const Pointers: Protect communication IDs.
Double Pointers: Dynamic message storage.

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

```c
#include <time.h>

#define MAX_MESSAGES 10  // Maximum number of messages to log

// Structure to store communication metadata
typedef struct {
    int comm_id;         // Communication ID
    time_t timestamp;    // Timestamp of the communication
} CommMetaData;

// Function to get the current timestamp
time_t get_current_timestamp() {
    return time(NULL);
}

// Function to create a new message
char* create_message(const char* message_content) {
    char* message = (char*)malloc(strlen(message_content) + 1);
    if (message != NULL) {
        strcpy(message, message_content);
    }
    return message;
}

// Function to print the communication log
void print_communication_log(char** message_log, CommMetaData* metadata_log, int count) {
    for (int i = 0; i < count; i++) {
        printf("Communication ID: %d\n", metadata_log[i].comm_id);
        printf("Timestamp: %s", ctime(&metadata_log[i].timestamp)); // Format timestamp
        printf("Message: %s\n\n", message_log[i]);
    }
}

// Function to free the dynamically allocated memory
void free_messages(char** message_log, int count) {
    for (int i = 0; i < count; i++) {
        free(message_log[i]);
    }
    free(message_log);
    free(metadata_log);
}

int main() {
    CommMetaData* metadata_log = (CommMetaData*)malloc(sizeof(CommMetaData) *
MAX_MESSAGES); // Log of communication metadata
    char** message_log = (char**)malloc(sizeof(char*) * MAX_MESSAGES); // Log of messages
    int current_count = 0;

    // Protecting communication IDs using const pointers
    const int* comm_id_ptr;

    // Simulating communication messages
    for (int i = 0; i < MAX_MESSAGES; i++) {
        CommMetaData comm_data;
        comm_data.comm_id = i + 1;
```

```
        comm_data.timestamp = get_current_timestamp();

        comm_id_ptr = &comm_data.comm_id; // Protect the comm_id pointer

        // Create a message (dynamic memory allocation)
        char* message = create_message("Ship communication message.");
        if (message == NULL) {
            fprintf(stderr, "Error allocating memory for message.\n");
            break;
        }

        // Store metadata and message in logs
        metadata_log[current_count] = comm_data;
        message_log[current_count] = message;
        current_count++;
    }

    // Print the communication log
    print_communication_log(message_log, metadata_log, current_count);

    // Free allocated memory
    free_messages(message_log, current_count);

    return 0;
}
```

19. Fishing Activity Tracker
Description:
Design a system to track fishing activities using arrays for catch records, structures for vessel details, and unions for variable catch data like species or weight.
Specifications:
Structure: Vessel details (ID, name).
Union: Catch data (species, weight).
Array: Catch records.
const Pointers: Protect vessel IDs.
Double Pointers: Dynamic catch management.

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

// Structure to store vessel details (ID, name)
typedef struct {
    int vesselID;
    char vesselName[50];
} Vessel;

// Union to store variable catch data (species or weight)
typedef union {
    char species[50];
    float weight;
} CatchData;

// Structure to store each catch record (including vessel details and catch data)
typedef struct {
```

```
    Vessel vessel;
    CatchData catchData;
    int isSpecies; // 1 if catchData is species, 0 if it is weight
} CatchRecord;

// Function to print vessel details
void printVessel(Vessel* vessel) {
    printf("Vessel ID: %d\n", vessel->vesselID);
    printf("Vessel Name: %s\n", vessel->vesselName);
}

// Function to print catch details
void printCatchRecord(CatchRecord* catchRecord) {
    printf("Vessel ID: %d\n", catchRecord->vessel.vesselID);
    printf("Vessel Name: %s\n", catchRecord->vessel.vesselName);

    if (catchRecord->isSpecies) {
        printf("Catch Species: %s\n", catchRecord->catchData.species);
    } else {
        printf("Catch Weight: %.2f kg\n", catchRecord->catchData.weight);
    }
}

// Function to dynamically add a catch record
void addCatchRecord(CatchRecord*** records, int* numRecords, Vessel vessel, CatchData catchData, int
isSpecies) {
    // Reallocate memory for the new catch record
    *records = realloc(*records, (*numRecords + 1) * sizeof(CatchRecord));

    // Create a new catch record
    CatchRecord newCatch;
    newCatch.vessel = vessel;
    newCatch.isSpecies = isSpecies;

    if (isSpecies) {
        strcpy(newCatch.catchData.species, catchData.species);
    } else {
        newCatch.catchData.weight = catchData.weight;
    }

    // Add the new catch record to the array
    (*records)[*numRecords] = newCatch;
    (*numRecords)++;
}

int main() {
    // Create sample vessels
    Vessel vessel1 = {1, "Seawolf"};
    Vessel vessel2 = {2, "Ocean Explorer"};

    // Create some catch data
    CatchData catch1;
    strcpy(catch1.species, "Salmon");

    CatchData catch2;
```

```
    catch2.weight = 30.5f;

    // Dynamically manage catch records
    CatchRecord* catchRecords = NULL;
    int numRecords = 0;

    // Add some catch records
    addCatchRecord(&catchRecords, &numRecords, vessel1, catch1, 1);  // Species
    addCatchRecord(&catchRecords, &numRecords, vessel2, catch2, 0);  // Weight

    // Print the catch records
    for (int i = 0; i < numRecords; i++) {
        printCatchRecord(&catchRecords[i]);
    }

    // Free allocated memory
    free(catchRecords);
    return 0;
}
```

20. Submarine Navigation System
Description:
Create a submarine navigation system using structures for navigation data, unions for environmental conditions, and arrays for depth readings.
Specifications:
Structure: Navigation data (location, depth).
Union: Environmental conditions (temperature, pressure).
Array: Depth readings.
const Pointers: Immutable navigation data.
Double Pointers: Manage dynamic depth logs.

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

// Define a structure for navigation data
struct NavigationData {
    float latitude;   // Latitude of the submarine
    float longitude;  // Longitude of the submarine
    float depth;      // Current depth of the submarine
};

// Define a union for environmental conditions
union EnvironmentalConditions {
    float temperature; // Temperature in Celsius
    float pressure;    // Pressure in Pascals
};

// Define a structure to hold depth readings over time
struct DepthLog {
    int num_readings; // Number of depth readings
    float *depth_readings; // Pointer to dynamically allocated array of depth readings
};

// Function to print navigation data
```

```
void print_navigation_data(const struct NavigationData *nav_data) {
    printf("Navigation Data:\n");
    printf("Latitude: %.2f, Longitude: %.2f, Depth: %.2f meters\n",
           nav_data->latitude, nav_data->longitude, nav_data->depth);
}

// Function to print environmental conditions (either temperature or pressure)
void print_environmental_conditions(const union EnvironmentalConditions *env_cond, int is_temperature)
{
    if (is_temperature) {
        printf("Temperature: %.2f°C\n", env_cond->temperature);
    } else {
        printf("Pressure: %.2f Pascals\n", env_cond->pressure);
    }
}

// Function to log and print depth readings
void log_depth_readings(struct DepthLog *depth_log, float *new_depths, int num_readings) {
    depth_log->num_readings = num_readings;
    depth_log->depth_readings = (float *)malloc(num_readings * sizeof(float)); // Dynamically allocate
memory

    if (depth_log->depth_readings != NULL) {
        memcpy(depth_log->depth_readings, new_depths, num_readings * sizeof(float)); // Copy new
depths
    } else {
        printf("Memory allocation failed for depth readings\n");
    }
}

void print_depth_readings(const struct DepthLog *depth_log) {
    printf("Depth Readings:\n");
    for (int i = 0; i < depth_log->num_readings; i++) {
        printf("Reading %d: %.2f meters\n", i + 1, depth_log->depth_readings[i]);
    }
}

int main() {
    // Example usage of the navigation system

    // 1. Create and set navigation data
    const struct NavigationData nav_data = {40.7128, -74.0060, 250.0}; // Example coordinates and depth
    print_navigation_data(&nav_data);

    // 2. Create environmental conditions (Temperature)
    union EnvironmentalConditions env_cond;
    env_cond.temperature = 10.5; // Temperature in Celsius
    print_environmental_conditions(&env_cond, 1); // 1 means it's temperature

    // 3. Log and print depth readings
    float depths[] = {250.0, 255.0, 260.0, 265.0}; // Example depth readings
    struct DepthLog depth_log;
    log_depth_readings(&depth_log, depths, 4); // Log 4 depth readings
    print_depth_readings(&depth_log);
```

```
    // 4. Free dynamically allocated memory for depth readings
    free(depth_log.depth_readings);

    return 0;
}
```