```
/*************************************************************************
****
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>



typedef struct {
    int ID;
    const char *itemCode;
    char name[50];
    char category[30];
} Item;
```

```
/
typedef union {
    float weight;
    float volume;
} Attributes;



void addItem(Item **inventory, Attributes **attributes, int *size, int *capacity, int ID, const char *itemCode, const char *name, const char *category, float attrValue, int isWeight);
void displayInventory(Item **inventory, Attributes **attributes, int size);


int main() {
    int size = 0, capacity = 2;
    Item *inventory = (Item *)malloc(capacity * sizeof(Item));
    Attributes *attributes = (Attributes *)malloc(capacity * sizeof(Attributes));


    addItem(&inventory, &attributes, &size, &capacity, 1, "A001", "Item1", "Category1", 12.5, 1); // weight
    addItem(&inventory, &attributes, &size, &capacity, 2, "A002", "Item2", "Category2", 3.6, 0); // volume
    addItem(&inventory, &attributes, &size, &capacity, 3, "A003", "Item3", "Category1", 8.9, 1); // weight


    displayInventory(&inventory, &attributes, size);


    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)inventory[i].itemCode);
    }
```

```
    free(inventory);
    free(attributes);

    return 0;
}

// Function to add an item to the inventory
void addItem(Item **inventory, Attributes **attributes, int *size, int *capacity, int ID, const char *itemCode, const char *name, const char *category, float attrValue, int isWeight) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Item *newInventory = (Item *)malloc(*capacity * sizeof(Item));
        Attributes *newAttributes = (Attributes *)malloc(*capacity * sizeof(Attributes));

        for (int i = 0; i < *size; i++) {
            newInventory[i] = (*inventory)[i];
            newAttributes[i] = (*attributes)[i];
        }

        free(*inventory);
        free(*attributes);
        *inventory = newInventory;
        *attributes = newAttributes;
    }
```

```
    (*inventory)[*size].ID = ID;
    (*inventory)[*size].itemCode = strdup(itemCode);
    strcpy((*inventory)[*size].name, name);
    strcpy((*inventory)[*size].category, category);


    if (isWeight) {
        (*attributes)[*size].weight = attrValue;
    } else {
        (*attributes)[*size].volume = attrValue;
    }


    (*size)++;
}

// Function to display the inventory
void displayInventory(Item **inventory, Attributes **attributes, int size) {
    printf("Inventory:\n");
    for (int i = 0; i < size; i++) {
        printf("ID: %d\n", (*inventory)[i].ID);
        printf("Item Code: %s\n", (*inventory)[i].itemCode);
        printf("Name: %s\n", (*inventory)[i].name);
        printf("Category: %s\n", (*inventory)[i].category);
        if (strlen((*inventory)[i].category) > 0) {
            printf("Weight: %.2f\n", (*attributes)[i].weight);
        } else {
```

```
            printf("Volume: %.2f\n", (*attributes)[i].volume);
        }
        printf("\n");
    }
}
```

```
/*************************************************************************
****
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>
```

```
typedef struct {
    const char *routeID;
    char start[50];
    char end[50];
} Route;


typedef union {
    char air[50];
    char sea[50];
    char land[50];
} TransportMode;


void addRoute(Route **routes, TransportMode **modes, int *size, int *capacity, const char *routeID, const char *start, const char *end, const char *mode, int transportType);
void displayRoutes(Route **routes, TransportMode **modes, int size);

int main() {
    int size = 0, capacity = 2;
    Route *routes = (Route *)malloc(capacity * sizeof(Route));
    TransportMode *modes = (TransportMode *)malloc(capacity * sizeof(TransportMode));

    addRoute(&routes, &modes, &size, &capacity, "R001", "New York", "Los Angeles", "Boeing 747", 0); // air
    addRoute(&routes, &modes, &size, &capacity, "R002", "Shanghai", "San Francisco", "Container Ship", 1); // sea
    addRoute(&routes, &modes, &size, &capacity, "R003", "Dallas", "Chicago", "Freight Train", 2); // land
```

```
    displayRoutes(&routes, &modes, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)routes[i].routeID);
    }
    free(routes);
    free(modes);

    return 0;
}

// Function to add a route to the route list
void addRoute(Route **routes, TransportMode **modes, int *size, int *capacity, const char *routeID, const char *start, const char *end, const char *mode, int transportType) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Route *newRoutes = (Route *)malloc(*capacity * sizeof(Route));
        TransportMode *newModes = (TransportMode *)malloc(*capacity * sizeof(TransportMode));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newRoutes[i] = (*routes)[i];
            newModes[i] = (*modes)[i];
        }
```

```
        // Free old arrays and update pointers
        free(*routes);
        free(*modes);
        *routes = newRoutes;
        *modes = newModes;
    }

    // Initialize the new route
    (*routes)[*size].routeID = strdup(routeID);
    strcpy((*routes)[*size].start, start);
    strcpy((*routes)[*size].end, end);

    // Set the transport mode
    switch (transportType) {
        case 0: // air
            strcpy((*modes)[*size].air, mode);
            break;
        case 1: // sea
            strcpy((*modes)[*size].sea, mode);
            break;
        case 2: // land
            strcpy((*modes)[*size].land, mode);
            break;
    }

    // Increment the size
    (*size)++;
}
```

```
// Function to display the routes
void displayRoutes(Route **routes, TransportMode **modes, int size) {
    printf("Routes:\n");
    for (int i = 0; i < size; i++) {
        printf("Route ID: %s\n", (*routes)[i].routeID);
        printf("Start: %s\n", (*routes)[i].start);
        printf("End: %s\n", (*routes)[i].end);
        if (strlen((*modes)[i].air) > 0) {
            printf("Transport Mode: Air (%s)\n", (*modes)[i].air);
        } else if (strlen((*modes)[i].sea) > 0) {
            printf("Transport Mode: Sea (%s)\n", (*modes)[i].sea);
        } else {
            printf("Transport Mode: Land (%s)\n", (*modes)[i].land);
        }
        printf("\n");
    }

}
```

```
/**********************************************************************
****
```

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>


typedef struct {
    const char *vehicleID;
    char type[30];
} Vehicle;


typedef union {
    char active[30];
```

```
    char maintenance[30];
} Status;



void addVehicle(Vehicle **fleet, Status **statuses, int *size, int *capacity, const char *vehicleID, const char *type, const char *status, int isActive);
void displayFleet(Vehicle **fleet, Status **statuses, int size);

int main() {
    int size = 0, capacity = 2;
    Vehicle *fleet = (Vehicle *)malloc(capacity * sizeof(Vehicle));
    Status *statuses = (Status *)malloc(capacity * sizeof(Status));

    addVehicle(&fleet, &statuses, &size, &capacity, "V001", "Truck", "Operational", 1); // active
    addVehicle(&fleet, &statuses, &size, &capacity, "V002", "Bus", "In Maintenance", 0); // maintenance
    addVehicle(&fleet, &statuses, &size, &capacity, "V003", "Van", "Operational", 1); // active

    displayFleet(&fleet, &statuses, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)fleet[i].vehicleID);
    }
    free(fleet);
    free(statuses);

    return 0;
```

```
}

// Function to add a vehicle to the fleet
void addVehicle(Vehicle **fleet, Status **statuses, int *size, int *capacity, const char *vehicleID, const char *type, const char *status, int isActive) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Vehicle *newFleet = (Vehicle *)malloc(*capacity * sizeof(Vehicle));
        Status *newStatuses = (Status *)malloc(*capacity * sizeof(Status));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newFleet[i] = (*fleet)[i];
            newStatuses[i] = (*statuses)[i];
        }

        // Free old arrays and update pointers
        free(*fleet);
        free(*statuses);
        *fleet = newFleet;
        *statuses = newStatuses;
    }

    // Initialize the new vehicle
    (*fleet)[*size].vehicleID = strdup(vehicleID);
    strcpy((*fleet)[*size].type, type);
```

```
    // Set the status (active or maintenance)
    if (isActive) {
        strcpy((*statuses)[*size].active, status);
    } else {
        strcpy((*statuses)[*size].maintenance, status);
    }

    // Increment the size
    (*size)++;
}

// Function to display the fleet
void displayFleet(Vehicle **fleet, Status **statuses, int size) {
    printf("Fleet:\n");
    for (int i = 0; i < size; i++) {
        printf("Vehicle ID: %s\n", (*fleet)[i].vehicleID);
        printf("Type: %s\n", (*fleet)[i].type);
        if (strlen((*statuses)[i].active) > 0) {
            printf("Status: Active (%s)\n", (*statuses)[i].active);
        } else {
            printf("Status: Maintenance (%s)\n", (*statuses)[i].maintenance);
        }
        printf("\n");
    }
}
```

```
/*************************************************************************
****
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>


typedef struct {
   const char *orderID;
   char customer[50];
   char items[100];
} Order;


typedef union {
```

```
    char creditCard[20];
    char cash[10];
} PaymentMethod;


void addOrder(Order **orders, PaymentMethod **payments, int *size, int *capacity, const char *orderID, const char *customer, const char *items, const char *payment, int isCreditCard);
void displayOrders(Order **orders, PaymentMethod **payments, int size);

int main() {
    int size = 0, capacity = 2;
    Order *orders = (Order *)malloc(capacity * sizeof(Order));
    PaymentMethod *payments = (PaymentMethod *)malloc(capacity * sizeof(PaymentMethod));

    addOrder(&orders, &payments, &size, &capacity, "O001", "John Doe", "Item1, Item2", "1234-5678-9123-4567", 1); // credit card
    addOrder(&orders, &payments, &size, &capacity, "O002", "Jane Smith", "Item3, Item4", "Cash", 0); // cash
    addOrder(&orders, &payments, &size, &capacity, "O003", "Alice Johnson", "Item5, Item6", "9876-5432-1098-7654", 1); // credit card

    displayOrders(&orders, &payments, size);


    for (int i = 0; i < size; i++) {
        free((void *)orders[i].orderID);
    }
    free(orders);
```

```
    free(payments);


    return 0;
}


// Function to add an order to the order queue
void addOrder(Order **orders, PaymentMethod **payments, int *size, int *capacity, const char *orderID, const char *customer, const char *items, const char *payment, int isCreditCard) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Order *newOrders = (Order *)malloc(*capacity * sizeof(Order));
        PaymentMethod *newPayments = (PaymentMethod *)malloc(*capacity * sizeof(PaymentMethod));



        for (int i = 0; i < *size; i++) {
            newOrders[i] = (*orders)[i];
            newPayments[i] = (*payments)[i];
        }



        free(*orders);
        free(*payments);
        *orders = newOrders;
        *payments = newPayments;
    }
```

```
    (*orders)[*size].orderID = strdup(orderID);
    strcpy((*orders)[*size].customer, customer);
    strcpy((*orders)[*size].items, items);


    if (isCreditCard) {
        strcpy((*payments)[*size].creditCard, payment);
    } else {
        strcpy((*payments)[*size].cash, payment);
    }

    (*size)++;
}

// Function to display the orders
void displayOrders(Order **orders, PaymentMethod **payments, int size) {
    printf("Orders:\n");
    for (int i = 0; i < size; i++) {
        printf("Order ID: %s\n", (*orders)[i].orderID);
        printf("Customer: %s\n", (*orders)[i].customer);
        printf("Items: %s\n", (*orders)[i].items);
        if (strlen((*payments)[i].creditCard) > 0) {
            printf("Payment Method: Credit Card (%s)\n", (*payments)[i].creditCard);
        } else {
            printf("Payment Method: Cash (%s)\n", (*payments)[i].cash);
        }
        printf("\n");
```

```
    }

}
```

```
/*************************************************************************
****
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.
*************************************************************************
****/

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


typedef struct {
    const char *trackingNumber;
    char origin[50];
```

```
    char destination[50];
} Shipment;

typedef union {
    char dispatched[50];
    char delivered[50];
} TrackingEvent;

void addShipment(Shipment **shipments, TrackingEvent **events, int *size, int *capacity, const char *trackingNumber, const char *origin, const char *destination, const char *event, int isDispatched);
void displayShipments(Shipment **shipments, TrackingEvent **events, int size);

int main() {
    int size = 0, capacity = 2;
    Shipment *shipments = (Shipment *)malloc(capacity * sizeof(Shipment));
    TrackingEvent *events = (TrackingEvent *)malloc(capacity * sizeof(TrackingEvent));

    addShipment(&shipments, &events, &size, &capacity, "TN001", "New York", "Los Angeles", "Dispatched on 2025-01-01", 1); // dispatched
    addShipment(&shipments, &events, &size, &capacity, "TN002", "Chicago", "Houston", "Delivered on 2025-01-05", 0); // delivered
    addShipment(&shipments, &events, &size, &capacity, "TN003", "San Francisco", "Seattle", "Dispatched on 2025-01-07", 1); // dispatched

    displayShipments(&shipments, &events, size);

    // Free allocated memory
```

```
    for (int i = 0; i < size; i++) {
        free((void *)shipments[i].trackingNumber);
    }
    free(shipments);
    free(events);

    return 0;
}

// Function to add a shipment to the shipment list
void addShipment(Shipment **shipments, TrackingEvent **events, int *size, int *capacity,
const char *trackingNumber, const char *origin, const char *destination, const char *event,
int isDispatched) {

    if (*size == *capacity) {
        *capacity *= 2;
        Shipment *newShipments = (Shipment *)malloc(*capacity * sizeof(Shipment));
        TrackingEvent *newEvents = (TrackingEvent *)malloc(*capacity * sizeof(TrackingEvent));

        for (int i = 0; i < *size; i++) {
            newShipments[i] = (*shipments)[i];
            newEvents[i] = (*events)[i];
        }

        free(*shipments);
        free(*events);
        *shipments = newShipments;
```

```
        *events = newEvents;
    }

    // Initialize the new shipment
    (*shipments)[*size].trackingNumber = strdup(trackingNumber);
    strcpy((*shipments)[*size].origin, origin);
    strcpy((*shipments)[*size].destination, destination);

    // Set the tracking event (dispatched or delivered)
    if (isDispatched) {
        strcpy((*events)[*size].dispatched, event);
    } else {
        strcpy((*events)[*size].delivered, event);
    }

    // Increment the size
    (*size)++;
}

// Function to display the shipments
void displayShipments(Shipment **shipments, TrackingEvent **events, int size) {
    printf("Shipments:\n");
    for (int i = 0; i < size; i++) {
        printf("Tracking Number: %s\n", (*shipments)[i].trackingNumber);
        printf("Origin: %s\n", (*shipments)[i].origin);
        printf("Destination: %s\n", (*shipments)[i].destination);
        if (strlen((*events)[i].dispatched) > 0) {
            printf("Event: Dispatched (%s)\n", (*events)[i].dispatched);
```

```
        } else {
            printf("Event: Delivered (%s)\n", (*events)[i].delivered);
        }
        printf("\n");
    }
}

/**********************************************************************
****
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>
#include <string.h>

typedef struct {
    const char *nodeID;
```

```
    char location[50];
} TrafficNode;



typedef union {
    char clear[20];
    char congested[20];
} TrafficCondition;

// Function prototypes
void addTrafficNode(TrafficNode **nodes, TrafficCondition **conditions, int *size, int *capacity, const char *nodeID, const char *location, const char *condition, int isClear);
void displayTrafficNodes(TrafficNode **nodes, TrafficCondition **conditions, int size);

int main() {
    int size = 0, capacity = 2;
    TrafficNode *nodes = (TrafficNode *)malloc(capacity * sizeof(TrafficNode));
    TrafficCondition *conditions = (TrafficCondition *)malloc(capacity * sizeof(TrafficCondition));

    addTrafficNode(&nodes, &conditions, &size, &capacity, "N001", "Intersection A", "Clear", 1); // clear
    addTrafficNode(&nodes, &conditions, &size, &capacity, "N002", "Intersection B", "Congested", 0); // congested
    addTrafficNode(&nodes, &conditions, &size, &capacity, "N003", "Intersection C", "Clear", 1); // clear

    displayTrafficNodes(&nodes, &conditions, size);
```

```
    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)nodes[i].nodeID); // Cast to void* to free const char*
    }
    free(nodes);
    free(conditions);

    return 0;
}

// Function to add a traffic node to the list
void addTrafficNode(TrafficNode **nodes, TrafficCondition **conditions, int *size, int *capacity, const char *nodeID, const char *location, const char *condition, int isClear) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        TrafficNode *newNodes = (TrafficNode *)malloc(*capacity * sizeof(TrafficNode));
        TrafficCondition *newConditions = (TrafficCondition *)malloc(*capacity * sizeof(TrafficCondition));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newNodes[i] = (*nodes)[i];
            newConditions[i] = (*conditions)[i];
        }

        // Free old arrays and update pointers
        free(*nodes);
        free(*conditions);
```

```
        *nodes = newNodes;
        *conditions = newConditions;
    }

    // Initialize the new traffic node
    (*nodes)[*size].nodeID = strdup(nodeID);
    strcpy((*nodes)[*size].location, location);

    // Set the traffic condition (clear or congested)
    if (isClear) {
        strcpy((*conditions)[*size].clear, condition);
    } else {
        strcpy((*conditions)[*size].congested, condition);
    }

    // Increment the size
    (*size)++;
}

// Function to display the traffic nodes
void displayTrafficNodes(TrafficNode **nodes, TrafficCondition **conditions, int size) {
    printf("Traffic Nodes:\n");
    for (int i = 0; i < size; i++) {
        printf("Node ID: %s\n", (*nodes)[i].nodeID);
        printf("Location: %s\n", (*nodes)[i].location);
        if (strlen((*conditions)[i].clear) > 0) {
            printf("Condition: Clear (%s)\n", (*conditions)[i].clear);
        } else {
```

```
            printf("Condition: Congested (%s)\n", (*conditions)[i].congested);
        }
        printf("\n");
    }
}
/*************************************************************************
****
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>



typedef struct {
    const char *slotID;
```

```
    char location[50];
    int size;
} Slot;


typedef union {
    char perishable[50];
    char nonPerishable[50];
} ItemType;


void addSlot(Slot **slots, ItemType **items, int *size, int *capacity, const char *slotID, const char *location, int sizeValue, const char *itemType, int isPerishable);
void displaySlots(Slot **slots, ItemType **items, int size);

int main() {
    int size = 0, capacity = 2;
    Slot *slots = (Slot *)malloc(capacity * sizeof(Slot));
    ItemType *items = (ItemType *)malloc(capacity * sizeof(ItemType));

    addSlot(&slots, &items, &size, &capacity, "S001", "Aisle 1", 100, "Fruits", 1); // perishable
    addSlot(&slots, &items, &size, &capacity, "S002", "Aisle 2", 200, "Electronics", 0); // non-perishable
    addSlot(&slots, &items, &size, &capacity, "S003", "Aisle 3", 150, "Vegetables", 1); // perishable

    displaySlots(&slots, &items, size);

    // Free allocated memory
```

```
    for (int i = 0; i < size; i++) {
        free((void *)slots[i].slotID);
    }
    free(slots);
    free(items);

    return 0;
}

// Function to add a slot to the slot list
void addSlot(Slot **slots, ItemType **items, int *size, int *capacity, const char *slotID, const char *location, int sizeValue, const char *itemType, int isPerishable) {

    if (*size == *capacity) {
        *capacity *= 2;
        Slot *newSlots = (Slot *)malloc(*capacity * sizeof(Slot));
        ItemType *newItems = (ItemType *)malloc(*capacity * sizeof(ItemType));


        for (int i = 0; i < *size; i++) {
            newSlots[i] = (*slots)[i];
            newItems[i] = (*items)[i];
        }


        free(*slots);
        free(*items);
        *slots = newSlots;
```

```
        *items = newItems;
    }

    // Initialize the new slot
    (*slots)[*size].slotID = strdup(slotID);
    strcpy((*slots)[*size].location, location);
    (*slots)[*size].size = sizeValue;



    if (isPerishable) {
        strcpy((*items)[*size].perishable, itemType);
    } else {
        strcpy((*items)[*size].nonPerishable, itemType);
    }

    // Increment the size
    (*size)++;
}

// Function to display the slots
void displaySlots(Slot **slots, ItemType **items, int size) {
    printf("Warehouse Slots:\n");
    for (int i = 0; i < size; i++) {
        printf("Slot ID: %s\n", (*slots)[i].slotID);
        printf("Location: %s\n", (*slots)[i].location);
        printf("Size: %d\n", (*slots)[i].size);
        if (strlen((*items)[i].perishable) > 0) {
            printf("Item Type: Perishable (%s)\n", (*items)[i].perishable);
```

```
        } else {
            printf("Item Type: Non-Perishable (%s)\n", (*items)[i].nonPerishable);
        }
        printf("\n");
    }
}
```

```
/**************************************************************************
****
```

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>
```

```
typedef struct {
    const char *packageID;
    float weight;
    char destination[50];
} Package;


typedef union {
    char standard[30];
    char express[30];
} DeliveryMethod;


void addPackage(Package **packages, DeliveryMethod **methods, int *size, int *capacity,
const char *packageID, float weight, const char *destination, const char *method, int
isStandard);
void displayPackages(Package **packages, DeliveryMethod **methods, int size);


int main() {
    int size = 0, capacity = 2;
    Package *packages = (Package *)malloc(capacity * sizeof(Package));
    DeliveryMethod *methods = (DeliveryMethod *)malloc(capacity *
sizeof(DeliveryMethod));


    addPackage(&packages, &methods, &size, &capacity, "P001", 5.0, "New York", "Standard
Delivery", 1); // standard
    addPackage(&packages, &methods, &size, &capacity, "P002", 2.5, "Los Angeles", "Express
Delivery", 0); // express
    addPackage(&packages, &methods, &size, &capacity, "P003", 10.0, "Chicago", "Standard
Delivery", 1); // standard
```

```
    displayPackages(&packages, &methods, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)packages[i].packageID);
    }
    free(packages);
    free(methods);

    return 0;
}

void addPackage(Package **packages, DeliveryMethod **methods, int *size, int *capacity,
const char *packageID, float weight, const char *destination, const char *method, int
isStandard) {

    if (*size == *capacity) {
        *capacity *= 2;
        Package *newPackages = (Package *)malloc(*capacity * sizeof(Package));
        DeliveryMethod *newMethods = (DeliveryMethod *)malloc(*capacity *
sizeof(DeliveryMethod));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newPackages[i] = (*packages)[i];
            newMethods[i] = (*methods)[i];
        }
```

```
        free(*packages);
        free(*methods);
        *packages = newPackages;
        *methods = newMethods;
    }

    // Initialize the new package
    (*packages)[*size].packageID = strdup(packageID);
    (*packages)[*size].weight = weight;
    strcpy((*packages)[*size].destination, destination);

    // Set the delivery method (standard or express)
    if (isStandard) {
        strcpy((*methods)[*size].standard, method);
    } else {
        strcpy((*methods)[*size].express, method);
    }

    // Increment the size
    (*size)++;
}


void displayPackages(Package **packages, DeliveryMethod **methods, int size) {
    printf("Packages:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Package ID: %s\n", (*packages)[i].packageID);
        printf("Weight: %.2f\n", (*packages)[i].weight);
        printf("Destination: %s\n", (*packages)[i].destination);
        if (strlen((*methods)[i].standard) > 0) {
            printf("Delivery Method: Standard (%s)\n", (*methods)[i].standard);
        } else {
            printf("Delivery Method: Express (%s)\n", (*methods)[i].express);
        }
        printf("\n");
    }

}
```

/**********

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 <string.h>

typedef struct {
    const char *timestamp;
    const char *metricType;
} AnalyticsRecord;

// Define the union for metrics
typedef union {
    float speed;
    float efficiency;
} Metrics;

// Function prototypes
void addAnalyticsRecord(AnalyticsRecord **records, Metrics **metrics, int *size, int
*capacity, const char *timestamp, const char *metricType, float value, int isSpeed);
void displayAnalyticsRecords(AnalyticsRecord **records, Metrics **metrics, int size);

int main() {
    int size = 0, capacity = 2;
    AnalyticsRecord *records = (AnalyticsRecord *)malloc(capacity * sizeof(AnalyticsRecord));
    Metrics *metrics = (Metrics *)malloc(capacity * sizeof(Metrics));

    addAnalyticsRecord(&records, &metrics, &size, &capacity, "2025-01-01 10:00:00",
"Speed", 60.0, 1); // speed
```

```
    addAnalyticsRecord(&records, &metrics, &size, &capacity, "2025-01-01 10:05:00", "Efficiency", 85.0, 0); // efficiency
    addAnalyticsRecord(&records, &metrics, &size, &capacity, "2025-01-01 10:10:00", "Speed", 65.5, 1); // speed

    displayAnalyticsRecords(&records, &metrics, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].timestamp); // Cast to void* to free const char*
        free((void *)records[i].metricType); // Cast to void* to free const char*
    }
    free(records);
    free(metrics);

    return 0;
}

// Function to add an analytics record to the list
void addAnalyticsRecord(AnalyticsRecord **records, Metrics **metrics, int *size, int *capacity, const char *timestamp, const char *metricType, float value, int isSpeed) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        AnalyticsRecord *newRecords = (AnalyticsRecord *)malloc(*capacity * sizeof(AnalyticsRecord));
        Metrics *newMetrics = (Metrics *)malloc(*capacity * sizeof(Metrics));

        // Copy existing data to new arrays
```

```
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
            newMetrics[i] = (*metrics)[i];
        }

        // Free old arrays and update pointers
        free(*records);
        free(*metrics);
        *records = newRecords;
        *metrics = newMetrics;
    }

    // Initialize the new analytics record
    (*records)[*size].timestamp = strdup(timestamp);
    (*records)[*size].metricType = strdup(metricType);

    // Set the metric (speed or efficiency)
    if (isSpeed) {
        (*metrics)[*size].speed = value;
    } else {
        (*metrics)[*size].efficiency = value;
    }

    // Increment the size
    (*size)++;
}

// Function to display the analytics records
```

```
void displayAnalyticsRecords(AnalyticsRecord **records, Metrics **metrics, int size) {
    printf("Analytics Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Timestamp: %s\n", (*records)[i].timestamp);
        printf("Metric Type: %s\n", (*records)[i].metricType);
        if (strcmp((*records)[i].metricType, "Speed") == 0) {
            printf("Metric: Speed (%.2f)\n", (*metrics)[i].speed);
        } else {
            printf("Metric: Efficiency (%.2f)\n", (*metrics)[i].efficiency);
        }
        printf("\n");
    }
}
```

/**********

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>


typedef struct {
    const char *scheduleID;
    char startTime[20];
    char endTime[20];
} Schedule;

// Define the union for transport types
typedef union {
    char bus[30];
    char truck[30];
} TransportType;

// Function prototypes
void addSchedule(Schedule **schedules, TransportType **types, int *size, int *capacity,
const char *scheduleID, const char *startTime, const char *endTime, const char *transport,
int isBus);
void displaySchedules(Schedule **schedules, TransportType **types, int size);

int main() {
    int size = 0, capacity = 2;
    Schedule *schedules = (Schedule *)malloc(capacity * sizeof(Schedule));
```

```
    TransportType *types = (TransportType *)malloc(capacity * sizeof(TransportType));

    addSchedule(&schedules, &types, &size, &capacity, "SCH001", "08:00", "10:00", "City Bus", 1); // bus
    addSchedule(&schedules, &types, &size, &capacity, "SCH002", "12:00", "14:00", "Delivery Truck", 0); // truck
    addSchedule(&schedules, &types, &size, &capacity, "SCH003", "16:00", "18:00", "Express Bus", 1); // bus

    displaySchedules(&schedules, &types, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)schedules[i].scheduleID); // Cast to void* to free const char*
    }
    free(schedules);
    free(types);

    return 0;
}

// Function to add a schedule to the schedule list
void addSchedule(Schedule **schedules, TransportType **types, int *size, int *capacity, const char *scheduleID, const char *startTime, const char *endTime, const char *transport, int isBus) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Schedule *newSchedules = (Schedule *)malloc(*capacity * sizeof(Schedule));
```

```
    TransportType *newTypes = (TransportType *)malloc(*capacity * sizeof(TransportType));

    // Copy existing data to new arrays
    for (int i = 0; i < *size; i++) {
        newSchedules[i] = (*schedules)[i];
        newTypes[i] = (*types)[i];
    }

    // Free old arrays and update pointers
    free(*schedules);
    free(*types);
    *schedules = newSchedules;
    *types = newTypes;
}

// Initialize the new schedule
(*schedules)[*size].scheduleID = strdup(scheduleID);
strcpy((*schedules)[*size].startTime, startTime);
strcpy((*schedules)[*size].endTime, endTime);

// Set the transport type (bus or truck)
if (isBus) {
    strcpy((*types)[*size].bus, transport);
} else {
    strcpy((*types)[*size].truck, transport);
}

// Increment the size
```

```
    (*size)++;
}

// Function to display the schedules
void displaySchedules(Schedule **schedules, TransportType **types, int size) {
    printf("Transportation Schedules:\n");
    for (int i = 0; i < size; i++) {
        printf("Schedule ID: %s\n", (*schedules)[i].scheduleID);
        printf("Start Time: %s\n", (*schedules)[i].startTime);
        printf("End Time: %s\n", (*schedules)[i].endTime);
        if (strlen((*types)[i].bus) > 0) {
            printf("Transport Type: Bus (%s)\n", (*types)[i].bus);
        } else {
            printf("Transport Type: Truck (%s)\n", (*types)[i].truck);
        }
        printf("\n");
    }
}
```

/**
****

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>


typedef struct {
    const char *ID;
    char name[50];
} Entity;


typedef union {
    char purchase[50];
    char returnItem[50];
```

```
} TransactionType;



void addTransaction(Entity **entities, TransactionType **transactions, int *size, int *capacity, const char *ID, const char *name, const char *transaction, int isPurchase);

void displayTransactions(Entity **entities, TransactionType **transactions, int size);


int main() {

    int size = 0, capacity = 2;

    Entity *entities = (Entity *)malloc(capacity * sizeof(Entity));

    TransactionType *transactions = (TransactionType *)malloc(capacity * sizeof(TransactionType));


    addTransaction(&entities, &transactions, &size, &capacity, "T001", "Supplier1", "Purchase Order #12345", 1); // purchase

    addTransaction(&entities, &transactions, &size, &capacity, "T002", "Customer1", "Return Order #54321", 0); // return

    addTransaction(&entities, &transactions, &size, &capacity, "T003", "Supplier2", "Purchase Order #67890", 1); // purchase


    displayTransactions(&entities, &transactions, size);


    // Free allocated memory

    for (int i = 0; i < size; i++) {

        free((void *)entities[i].ID);

    }

    free(entities);

    free(transactions);
```

```
    return 0;
}



void addTransaction(Entity **entities, TransactionType **transactions, int *size, int *capacity, const char *ID, const char *name, const char *transaction, int isPurchase) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Entity *newEntities = (Entity *)malloc(*capacity * sizeof(Entity));
        TransactionType *newTransactions = (TransactionType *)malloc(*capacity * sizeof(TransactionType));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newEntities[i] = (*entities)[i];
            newTransactions[i] = (*transactions)[i];
        }

        // Free old arrays and update pointers
        free(*entities);
        free(*transactions);
        *entities = newEntities;
        *transactions = newTransactions;
    }

    // Initialize the new transaction
    (*entities)[*size].ID = strdup(ID);
    strcpy((*entities)[*size].name, name);
```

```
    // Set the transaction type (purchase or return)
    if (isPurchase) {
        strcpy((*transactions)[*size].purchase, transaction);
    } else {
        strcpy((*transactions)[*size].returnItem, transaction);
    }

    // Increment the size
    (*size)++;
}

// Function to display the transactions
void displayTransactions(Entity **entities, TransactionType **transactions, int size) {
    printf("Supply Chain Transactions:\n");
    for (int i = 0; i < size; i++) {
        printf("Transaction ID: %s\n", (*entities)[i].ID);
        printf("Entity Name: %s\n", (*entities)[i].name);
        if (strlen((*transactions)[i].purchase) > 0) {
            printf("Transaction Type: Purchase (%s)\n", (*transactions)[i].purchase);
        } else {
            printf("Transaction Type: Return (%s)\n", (*transactions)[i].returnItem);
        }
        printf("\n");
    }
}
```

/**
****

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>

#include <string.h>



typedef struct {

   const char *costID;

   float baseCost;

} CostComponent;



typedef union {

   float fixedCost;

   float variableCost;

} PricingModel;
```

```
void addCostRecord(CostComponent **components, PricingModel **models, int *size, int *capacity, const char *costID, float baseCost, float cost, int isFixed);
void displayCostRecords(CostComponent **components, PricingModel **models, int size);

int main() {
    int size = 0, capacity = 2;
    CostComponent *components = (CostComponent *)malloc(capacity * sizeof(CostComponent));
    PricingModel *models = (PricingModel *)malloc(capacity * sizeof(PricingModel));

    addCostRecord(&components, &models, &size, &capacity, "C001", 1000.0, 500.0, 1); // fixed cost
    addCostRecord(&components, &models, &size, &capacity, "C002", 800.0, 200.0, 0); // variable cost
    addCostRecord(&components, &models, &size, &capacity, "C003", 1500.0, 750.0, 1); // fixed cost

    displayCostRecords(&components, &models, size);

    for (int i = 0; i < size; i++) {
        free((void *)components[i].costID);
    }
    free(components);
    free(models);

    return 0;
```

```
}

void addCostRecord(CostComponent **components, PricingModel **models, int *size, int *capacity, const char *costID, float baseCost, float cost, int isFixed) {

    if (*size == *capacity) {
        *capacity *= 2;
        CostComponent *newComponents = (CostComponent *)malloc(*capacity * sizeof(CostComponent));
        PricingModel *newModels = (PricingModel *)malloc(*capacity * sizeof(PricingModel));

        for (int i = 0; i < *size; i++) {
            newComponents[i] = (*components)[i];
            newModels[i] = (*models)[i];
        }

        free(*components);
        free(*models);
        *components = newComponents;
        *models = newModels;
    }

    // Initialize the new cost record
    (*components)[*size].costID = strdup(costID);
    (*components)[*size].baseCost = baseCost;
```

```
    // Set the pricing model (fixed or variable)
    if (isFixed) {
        (*models)[*size].fixedCost = cost;
    } else {
        (*models)[*size].variableCost = cost;
    }

    // Increment the size
    (*size)++;
}

// Function to display the cost records
void displayCostRecords(CostComponent **components, PricingModel **models, int size) {
    printf("Cost Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Cost ID: %s\n", (*components)[i].costID);
        printf("Base Cost: %.2f\n", (*components)[i].baseCost);
        if ((*models)[i].fixedCost > 0) {
            printf("Pricing Model: Fixed (%.2f)\n", (*models)[i].fixedCost);
        } else {
            printf("Pricing Model: Variable (%.2f)\n", (*models)[i].variableCost);
        }
        printf("\n");
    }
}
```

/**
****

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>


typedef struct {
    const char *loadID;
    float weight;
    char destination[50];
} Load;


typedef union {
    char bulk[30];
```

```
    char container[30];
} LoadType;



void addLoad(Load **loads, LoadType **types, int *size, int *capacity, const char *loadID,
float weight, const char *destination, const char *type, int isBulk);
void displayLoads(Load **loads, LoadType **types, int size);


int main() {
    int size = 0, capacity = 2;
    Load *loads = (Load *)malloc(capacity * sizeof(Load));
    LoadType *types = (LoadType *)malloc(capacity * sizeof(LoadType));

    addLoad(&loads, &types, &size, &capacity, "L001", 5000.0, "New York", "Grain", 1); // bulk
    addLoad(&loads, &types, &size, &capacity, "L002", 3000.0, "Los Angeles", "Electronics",
0); // container
    addLoad(&loads, &types, &size, &capacity, "L003", 7000.0, "Chicago", "Coal", 1); // bulk

    displayLoads(&loads, &types, size);



    for (int i = 0; i < size; i++) {
        free((void *)loads[i].loadID);
    }
    free(loads);
    free(types);

    return 0;
}
```

```
void addLoad(Load **loads, LoadType **types, int *size, int *capacity, const char *loadID,
float weight, const char *destination, const char *type, int isBulk) {

    if (*size == *capacity) {
        *capacity *= 2;
        Load *newLoads = (Load *)malloc(*capacity * sizeof(Load));
        LoadType *newTypes = (LoadType *)malloc(*capacity * sizeof(LoadType));


        for (int i = 0; i < *size; i++) {
            newLoads[i] = (*loads)[i];
            newTypes[i] = (*types)[i];
        }


        free(*loads);
        free(*types);
        *loads = newLoads;
        *types = newTypes;
    }


    (*loads)[*size].loadID = strdup(loadID);
    (*loads)[*size].weight = weight;
    strcpy((*loads)[*size].destination, destination);
```

```
    if (isBulk) {
        strcpy((*types)[*size].bulk, type);
    } else {
        strcpy((*types)[*size].container, type);
    }



    (*size)++;
}



void displayLoads(Load **loads, LoadType **types, int size) {
    printf("Loads:\n");
    for (int i = 0; i < size; i++) {
        printf("Load ID: %s\n", (*loads)[i].loadID);
        printf("Weight: %.2f\n", (*loads)[i].weight);
        printf("Destination: %s\n", (*loads)[i].destination);
        if (strlen((*types)[i].bulk) > 0) {
            printf("Load Type: Bulk (%s)\n", (*types)[i].bulk);
        } else {
            printf("Load Type: Container (%s)\n", (*types)[i].container);
        }
        printf("\n");
    }
}
```

/**
****

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>


typedef struct {
    const char *transportID;
    char origin[50];
    char destination[50];
} Transport;


typedef union {
    char rail[30];
```

```
    char road[30];
} TransportMode;



void addTransport(Transport **transports, TransportMode **modes, int *size, int *capacity, const char *transportID, const char *origin, const char *destination, const char *mode, int isRail);
void displayTransports(Transport **transports, TransportMode **modes, int size);

int main() {
    int size = 0, capacity = 2;
    Transport *transports = (Transport *)malloc(capacity * sizeof(Transport));
    TransportMode *modes = (TransportMode *)malloc(capacity * sizeof(TransportMode));

    addTransport(&transports, &modes, &size, &capacity, "T001", "City A", "City B", "Freight Train", 1); // rail
    addTransport(&transports, &modes, &size, &capacity, "T002", "City C", "City D", "Cargo Truck", 0); // road
    addTransport(&transports, &modes, &size, &capacity, "T003", "City E", "City F", "Passenger Train", 1); // rail

    displayTransports(&transports, &modes, size);



    for (int i = 0; i < size; i++) {
        free((void *)transports[i].transportID);
    }
    free(transports);
    free(modes);
```

```
    return 0;
}


void addTransport(Transport **transports, TransportMode **modes, int *size, int *capacity,
const char *transportID, const char *origin, const char *destination, const char *mode, int
isRail) {

    if (*size == *capacity) {
        *capacity *= 2;
        Transport *newTransports = (Transport *)malloc(*capacity * sizeof(Transport));
        TransportMode *newModes = (TransportMode *)malloc(*capacity *
sizeof(TransportMode));


        for (int i = 0; i < *size; i++) {
            newTransports[i] = (*transports)[i];
            newModes[i] = (*modes)[i];
        }


        free(*transports);
        free(*modes);
        *transports = newTransports;
        *modes = newModes;
    }
```

```
    (*transports)[*size].transportID = strdup(transportID);
    strcpy((*transports)[*size].origin, origin);
    strcpy((*transports)[*size].destination, destination);


    if (isRail) {
        strcpy((*modes)[*size].rail, mode);
    } else {
        strcpy((*modes)[*size].road, mode);
    }


    (*size)++;
}


void displayTransports(Transport **transports, TransportMode **modes, int size) {
    printf("Transports:\n");
    for (int i = 0; i < size; i++) {
        printf("Transport ID: %s\n", (*transports)[i].transportID);
        printf("Origin: %s\n", (*transports)[i].origin);
        printf("Destination: %s\n", (*transports)[i].destination);
        if (strlen((*modes)[i].rail) > 0) {
            printf("Transport Mode: Rail (%s)\n", (*modes)[i].rail);
        } else {
            printf("Transport Mode: Road (%s)\n", (*modes)[i].road);
        }
        printf("\n");
```

```
    }
}
```

```
/**********************************************************************
****
```

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>
#include <string.h>

typedef struct {
    const char *metricID;
    float value;
} PerformanceMetric;
```

```
typedef union {
    float time;
    float cost;
} PerformanceAspect;

void addPerformanceMetric(PerformanceMetric **metrics, PerformanceAspect **aspects,
int *size, int *capacity, const char *metricID, float value, float aspectValue, int isTime);
void displayPerformanceMetrics(PerformanceMetric **metrics, PerformanceAspect
**aspects, int size);

int main() {
    int size = 0, capacity = 2;
    PerformanceMetric *metrics = (PerformanceMetric *)malloc(capacity *
sizeof(PerformanceMetric));
    PerformanceAspect *aspects = (PerformanceAspect *)malloc(capacity *
sizeof(PerformanceAspect));

    addPerformanceMetric(&metrics, &aspects, &size, &capacity, "M001", 90.0, 120.5, 1); //
time
    addPerformanceMetric(&metrics, &aspects, &size, &capacity, "M002", 85.0, 300.0, 0); //
cost
    addPerformanceMetric(&metrics, &aspects, &size, &capacity, "M003", 95.0, 150.0, 1); //
time

    displayPerformanceMetrics(&metrics, &aspects, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
```

```
        free((void *)metrics[i].metricID);
    }
    free(metrics);
    free(aspects);

    return 0;
}


void addPerformanceMetric(PerformanceMetric **metrics, PerformanceAspect **aspects,
int *size, int *capacity, const char *metricID, float value, float aspectValue, int isTime) {

    if (*size == *capacity) {
        *capacity *= 2;
        PerformanceMetric *newMetrics = (PerformanceMetric *)malloc(*capacity *
sizeof(PerformanceMetric));
        PerformanceAspect *newAspects = (PerformanceAspect *)malloc(*capacity *
sizeof(PerformanceAspect));


        for (int i = 0; i < *size; i++) {
            newMetrics[i] = (*metrics)[i];
            newAspects[i] = (*aspects)[i];
        }


        free(*metrics);
        free(*aspects);
        *metrics = newMetrics;
```

```
        *aspects = newAspects;
    }



    (*metrics)[*size].metricID = strdup(metricID);
    (*metrics)[*size].value = value;



    if (isTime) {
        (*aspects)[*size].time = aspectValue;
    } else {
        (*aspects)[*size].cost = aspectValue;
    }



    (*size)++;
}



void displayPerformanceMetrics(PerformanceMetric **metrics, PerformanceAspect
**aspects, int size) {
    printf("Performance Metrics:\n");
    for (int i = 0; i < size; i++) {
        printf("Metric ID: %s\n", (*metrics)[i].metricID);
        printf("Value: %.2f\n", (*metrics)[i].value);
        if ((*aspects)[i].time > 0) {
            printf("Performance Aspect: Time (%.2f)\n", (*aspects)[i].time);
        } else {
```

```
            printf("Performance Aspect: Cost (%.2f)\n", (*aspects)[i].cost);
        }
        printf("\n");
    }
}
```

```
/**************************************************************************
****
```

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>
```

```
typedef struct {
    const char *robotID;
    char type[30];
    char status[30];
} Robot;


typedef union {
    char picking[30];
    char sorting[30];
} TaskType;


void addRobot(Robot **robots, TaskType **tasks, int *size, int *capacity, const char *robotID, const char *type, const char *status, const char *task, int isPicking);
void displayRobots(Robot **robots, TaskType **tasks, int size);

int main() {
    int size = 0, capacity = 2;
    Robot *robots = (Robot *)malloc(capacity * sizeof(Robot));
    TaskType *tasks = (TaskType *)malloc(capacity * sizeof(TaskType));

    addRobot(&robots, &tasks, &size, &capacity, "R001", "Type A", "Active", "Picking Orders", 1); // picking
    addRobot(&robots, &tasks, &size, &capacity, "R002", "Type B", "Idle", "Sorting Items", 0); // sorting
    addRobot(&robots, &tasks, &size, &capacity, "R003", "Type A", "Active", "Picking Orders", 1); // picking
```

```
    displayRobots(&robots, &tasks, size);



    for (int i = 0; i < size; i++) {
        free((void *)robots[i].robotID);
    }
    free(robots);
    free(tasks);

    return 0;
}



void addRobot(Robot **robots, TaskType **tasks, int *size, int *capacity, const char
*robotID, const char *type, const char *status, const char *task, int isPicking) {

    if (*size == *capacity) {
        *capacity *= 2;
        Robot *newRobots = (Robot *)malloc(*capacity * sizeof(Robot));
        TaskType *newTasks = (TaskType *)malloc(*capacity * sizeof(TaskType));



        for (int i = 0; i < *size; i++) {
            newRobots[i] = (*robots)[i];
            newTasks[i] = (*tasks)[i];
        }
```

```
        free(*robots);
        free(*tasks);
        *robots = newRobots;
        *tasks = newTasks;
    }

    (*robots)[*size].robotID = strdup(robotID);
    strcpy((*robots)[*size].type, type);
    strcpy((*robots)[*size].status, status);

    if (isPicking) {
        strcpy((*tasks)[*size].picking, task);
    } else {
        strcpy((*tasks)[*size].sorting, task);
    }

    (*size)++;
}

void displayRobots(Robot **robots, TaskType **tasks, int size) {
    printf("Robots:\n");
    for (int i = 0; i < size; i++) {
        printf("Robot ID: %s\n", (*robots)[i].robotID);
```

```
        printf("Type: %s\n", (*robots)[i].type);
        printf("Status: %s\n", (*robots)[i].status);
        if (strlen((*tasks)[i].picking) > 0) {
            printf("Task Type: Picking (%s)\n", (*tasks)[i].picking);
        } else {
            printf("Task Type: Sorting (%s)\n", (*tasks)[i].sorting);
        }
        printf("\n");
    }
}

/**************************************************************************
****
```

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>



typedef struct {
    const char *feedbackID;
    char content[200];
} Feedback;



typedef union {
    char positive[100];
    char negative[100];
} FeedbackType;

// Function prototypes
void addFeedback(Feedback **feedbacks, FeedbackType **types, int *size, int *capacity, const char *feedbackID, const char *content, const char *type, int isPositive);
void displayFeedbacks(Feedback **feedbacks, FeedbackType **types, int size);

int main() {
    int size = 0, capacity = 2;
    Feedback *feedbacks = (Feedback *)malloc(capacity * sizeof(Feedback));
    FeedbackType *types = (FeedbackType *)malloc(capacity * sizeof(FeedbackType));

    addFeedback(&feedbacks, &types, &size, &capacity, "F001", "Great service!", "Positive Experience", 1); // positive
    addFeedback(&feedbacks, &types, &size, &capacity, "F002", "Poor customer support.", "Negative Experience", 0); // negative
```

```
    addFeedback(&feedbacks, &types, &size, &capacity, "F003", "Quick delivery.", "Positive
Experience", 1); // positive

    displayFeedbacks(&feedbacks, &types, size);

    for (int i = 0; i < size; i++) {
        free((void *)feedbacks[i].feedbackID);
    }
    free(feedbacks);
    free(types);

    return 0;
}

// Function to add a feedback to the list
void addFeedback(Feedback **feedbacks, FeedbackType **types, int *size, int *capacity,
const char *feedbackID, const char *content, const char *type, int isPositive) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Feedback *newFeedbacks = (Feedback *)malloc(*capacity * sizeof(Feedback));
        FeedbackType *newTypes = (FeedbackType *)malloc(*capacity * sizeof(FeedbackType));

        for (int i = 0; i < *size; i++) {
            newFeedbacks[i] = (*feedbacks)[i];
            newTypes[i] = (*types)[i];
        }
```

```
        free(*feedbacks);
        free(*types);
        *feedbacks = newFeedbacks;
        *types = newTypes;
    }

    (*feedbacks)[*size].feedbackID = strdup(feedbackID);
    strcpy((*feedbacks)[*size].content, content);

    if (isPositive) {
        strcpy((*types)[*size].positive, type);
    } else {
        strcpy((*types)[*size].negative, type);
    }

    (*size)++;
}

void displayFeedbacks(Feedback **feedbacks, FeedbackType **types, int size) {
    printf("Customer Feedbacks:\n");
    for (int i = 0; i < size; i++) {
        printf("Feedback ID: %s\n", (*feedbacks)[i].feedbackID);
```

```
        printf("Content: %s\n", (*feedbacks)[i].content);
        if (strlen((*types)[i].positive) > 0) {
            printf("Feedback Type: Positive (%s)\n", (*types)[i].positive);
        } else {
            printf("Feedback Type: Negative (%s)\n", (*types)[i].negative);
        }
        printf("\n");
    }
}

/**********************************************************************
****
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>
```

```
typedef struct {
    const char *fleetID;
    char location[50];
    char status[30];
} Fleet;


typedef union {
    char dispatch[30];
    char reroute[30];
} CoordinationType;


void addFleet(Fleet **fleets, CoordinationType **coords, int *size, int *capacity, const char *fleetID, const char *location, const char *status, const char *coordination, int isDispatch);
void displayFleets(Fleet **fleets, CoordinationType **coords, int size);


int main() {
    int size = 0, capacity = 2;
    Fleet *fleets = (Fleet *)malloc(capacity * sizeof(Fleet));
    CoordinationType *coords = (CoordinationType *)malloc(capacity * sizeof(CoordinationType));


    addFleet(&fleets, &coords, &size, &capacity, "F001", "Location A", "Active", "Dispatched", 1); // dispatch
    addFleet(&fleets, &coords, &size, &capacity, "F002", "Location B", "Idle", "Rerouted", 0); // reroute
```

```
    addFleet(&fleets, &coords, &size, &capacity, "F003", "Location C", "Active", "Dispatched",
1); // dispatch

    displayFleets(&fleets, &coords, size);

    for (int i = 0; i < size; i++) {
        free((void *)fleets[i].fleetID);
    }
    free(fleets);
    free(coords);

    return 0;
}

void addFleet(Fleet **fleets, CoordinationType **coords, int *size, int *capacity, const char
*fleetID, const char *location, const char *status, const char *coordination, int isDispatch) {

    if (*size == *capacity) {
        *capacity *= 2;
        Fleet *newFleets = (Fleet *)malloc(*capacity * sizeof(Fleet));
        CoordinationType *newCoords = (CoordinationType *)malloc(*capacity *
sizeof(CoordinationType));

        for (int i = 0; i < *size; i++) {
            newFleets[i] = (*fleets)[i];
            newCoords[i] = (*coords)[i];
```

```
        }

        free(*fleets);
        free(*coords);
        *fleets = newFleets;
        *coords = newCoords;
    }

    (*fleets)[*size].fleetID = strdup(fleetID);
    strcpy((*fleets)[*size].location, location);
    strcpy((*fleets)[*size].status, status);

    if (isDispatch) {
        strcpy((*coords)[*size].dispatch, coordination);
    } else {
        strcpy((*coords)[*size].reroute, coordination);
    }

    (*size)++;
}

// Function to display the fleets
void displayFleets(Fleet **fleets, CoordinationType **coords, int size) {
    printf("Fleet Coordination:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Fleet ID: %s\n", (*fleets)[i].fleetID);
        printf("Location: %s\n", (*fleets)[i].location);
        printf("Status: %s\n", (*fleets)[i].status);
        if (strlen((*coords)[i].dispatch) > 0) {
            printf("Coordination Type: Dispatch (%s)\n", (*coords)[i].dispatch);
        } else {
            printf("Coordination Type: Reroute (%s)\n", (*coords)[i].reroute);
        }
        printf("\n");
    }
}
```

/**********

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>

typedef struct {
    const char *eventID;
    char description[200];
} SecurityEvent;

typedef union {
    char breach[50];
    char resolved[50];
} EventType;

void addSecurityEvent(SecurityEvent **events, EventType **types, int *size, int *capacity,
const char *eventID, const char *description, const char *type, int isBreach);
void displaySecurityEvents(SecurityEvent **events, EventType **types, int size);

int main() {
    int size = 0, capacity = 2;
    SecurityEvent *events = (SecurityEvent *)malloc(capacity * sizeof(SecurityEvent));
    EventType *types = (EventType *)malloc(capacity * sizeof(EventType));

    addSecurityEvent(&events, &types, &size, &capacity, "E001", "Unauthorized access
detected", "Breach", 1); // breach
```

```
    addSecurityEvent(&events, &types, &size, &capacity, "E002", "Security issue resolved", "Resolved", 0); // resolved
    addSecurityEvent(&events, &types, &size, &capacity, "E003", "Fire alarm triggered", "Breach", 1); // breach

    displaySecurityEvents(&events, &types, size);



    for (int i = 0; i < size; i++) {
        free((void *)events[i].eventID);
    }
    free(events);
    free(types);

    return 0;
}



void addSecurityEvent(SecurityEvent **events, EventType **types, int *size, int *capacity, const char *eventID, const char *description, const char *type, int isBreach) {

    if (*size == *capacity) {
        *capacity *= 2;
        SecurityEvent *newEvents = (SecurityEvent *)malloc(*capacity * sizeof(SecurityEvent));
        EventType *newTypes = (EventType *)malloc(*capacity * sizeof(EventType));



        for (int i = 0; i < *size; i++) {
            newEvents[i] = (*events)[i];
```

```
            newTypes[i] = (*types)[i];
        }

        free(*events);
        free(*types);
        *events = newEvents;
        *types = newTypes;
    }

    (*events)[*size].eventID = strdup(eventID);
    strcpy((*events)[*size].description, description);

    if (isBreach) {
        strcpy((*types)[*size].breach, type);
    } else {
        strcpy((*types)[*size].resolved, type);
    }

    (*size)++;
}

void displaySecurityEvents(SecurityEvent **events, EventType **types, int size) {
    printf("Security Events:\n");
    for (int i = 0; i < size; i++) {
```

```
        printf("Event ID: %s\n", (*events)[i].eventID);
        printf("Description: %s\n", (*events)[i].description);
        if (strlen((*types)[i].breach) > 0) {
            printf("Event Type: Breach (%s)\n", (*types)[i].breach);
        } else {
            printf("Event Type: Resolved (%s)\n", (*types)[i].resolved);
        }
        printf("\n");
    }
}

/*************************************************************************
****
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>
```

```
typedef struct {
    const char *billID;
    float amount;
    char date[20];
} BillingDetail;


typedef union {
    char bankTransfer[50];
    char cash[20];
} PaymentMethod;


void addBillingRecord(BillingDetail **details, PaymentMethod **methods, int *size, int *capacity, const char *billID, float amount, const char *date, const char *method, int isBankTransfer);
void displayBillingRecords(BillingDetail **details, PaymentMethod **methods, int size);


int main() {
    int size = 0, capacity = 2;
    BillingDetail *details = (BillingDetail *)malloc(capacity * sizeof(BillingDetail));
    PaymentMethod *methods = (PaymentMethod *)malloc(capacity * sizeof(PaymentMethod));


    addBillingRecord(&details, &methods, &size, &capacity, "B001", 1000.0, "2025-01-01", "Bank Transfer", 1); // bank transfer
```

```
    addBillingRecord(&details, &methods, &size, &capacity, "B002", 500.0, "2025-01-02", "Cash", 0); // cash
    addBillingRecord(&details, &methods, &size, &capacity, "B003", 1500.0, "2025-01-03", "Bank Transfer", 1); // bank transfer

    displayBillingRecords(&details, &methods, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)details[i].billID);
    }
    free(details);
    free(methods);

    return 0;
}

void addBillingRecord(BillingDetail **details, PaymentMethod **methods, int *size, int *capacity, const char *billID, float amount, const char *date, const char *method, int isBankTransfer) {

    if (*size == *capacity) {
        *capacity *= 2;
        BillingDetail *newDetails = (BillingDetail *)malloc(*capacity * sizeof(BillingDetail));
        PaymentMethod *newMethods = (PaymentMethod *)malloc(*capacity * sizeof(PaymentMethod));
```

```
        for (int i = 0; i < *size; i++) {
            newDetails[i] = (*details)[i];
            newMethods[i] = (*methods)[i];
        }


        free(*details);
        free(*methods);
        *details = newDetails;
        *methods = newMethods;
    }


    (*details)[*size].billID = strdup(billID);
    (*details)[*size].amount = amount;
    strcpy((*details)[*size].date, date);


    if (isBankTransfer) {
        strcpy((*methods)[*size].bankTransfer, method);
    } else {
        strcpy((*methods)[*size].cash, method);
    }


    (*size)++;
}
```

```
void displayBillingRecords(BillingDetail **details, PaymentMethod **methods, int size) {
    printf("Billing Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Bill ID: %s\n", (*details)[i].billID);
        printf("Amount: %.2f\n", (*details)[i].amount);
        printf("Date: %s\n", (*details)[i].date);
        if (strlen((*methods)[i].bankTransfer) > 0) {
            printf("Payment Method: Bank Transfer (%s)\n", (*methods)[i].bankTransfer);
        } else {
            printf("Payment Method: Cash (%s)\n", (*methods)[i].cash);
        }
        printf("\n");
    }
}
```

/**
****

21.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>
#include <string.h>


typedef struct {
    const char *start;
    const char *end;
    const char **waypoints;
    int numWaypoints;
} Route;
```

```
void addRoute(Route **routes, int *size, int *capacity, const char *start, const char *end,
const char **waypoints, int numWaypoints);
void displayRoutes(Route **routes, int size);

int main() {
    int size = 0, capacity = 2;
    Route *routes = (Route *)malloc(capacity * sizeof(Route));

    const char *waypoints1[] = {"Waypoint1", "Waypoint2", "Waypoint3"};
    const char *waypoints2[] = {"WaypointA", "WaypointB"};

    addRoute(&routes, &size, &capacity, "Port A", "Port B", waypoints1, 3);
    addRoute(&routes, &size, &capacity, "Port C", "Port D", waypoints2, 2);

    displayRoutes(&routes, size);



    for (int i = 0; i < size; i++) {
        free((void *)routes[i].start);
        free((void *)routes[i].end);
        free(routes[i].waypoints);
    }
    free(routes);

    return 0;
}
```

```
// Function to add a route to the list
void addRoute(Route **routes, int *size, int *capacity, const char *start, const char *end, const char **waypoints, int numWaypoints) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Route *newRoutes = (Route *)malloc(*capacity * sizeof(Route));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRoutes[i] = (*routes)[i];
        }

        // Free old array and update pointer
        free(*routes);
        *routes = newRoutes;
    }

    // Initialize the new route
    (*routes)[*size].start = strdup(start);
    (*routes)[*size].end = strdup(end);
    (*routes)[*size].waypoints = (const char **)malloc(numWaypoints * sizeof(const char *));
    for (int i = 0; i < numWaypoints; i++) {
        (*routes)[*size].waypoints[i] = strdup(waypoints[i]);
    }
    (*routes)[*size].numWaypoints = numWaypoints;
```

```
    // Increment the size
    (*size)++;
}

// Function to display the routes
void displayRoutes(Route **routes, int size) {
    printf("Routes:\n");
    for (int i = 0; i < size; i++) {
        printf("Start: %s\n", (*routes)[i].start);
        printf("End: %s\n", (*routes)[i].end);
        printf("Waypoints: ");
        for (int j = 0; j < (*routes)[i].numWaypoints; j++) {
            printf("%s ", (*routes)[i].waypoints[j]);
        }
        printf("\n\n");
    }
}
```

/**
****

22. 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>
#include <string.h>


typedef struct {
   const char *vesselID;
   char name[50];
   char type[30];
} Vessel;
```

```
typedef union {
    char cargoType[50];
    int passengerCount;
} Attributes;



void addVessel(Vessel **fleet, Attributes **attributes, int *size, int *capacity, const char *vesselID, const char *name, const char *type, const char *cargoType, int passengerCount, int isCargo);
void displayFleet(Vessel **fleet, Attributes **attributes, int size);


int main() {
    int size = 0, capacity = 2;
    Vessel *fleet = (Vessel *)malloc(capacity * sizeof(Vessel));
    Attributes *attributes = (Attributes *)malloc(capacity * sizeof(Attributes));


    addVessel(&fleet, &attributes, &size, &capacity, "V001", "Vessel A", "Cargo", "Electronics", 0, 1); // cargo
    addVessel(&fleet, &attributes, &size, &capacity, "V002", "Vessel B", "Passenger", "", 200, 0); // passenger
    addVessel(&fleet, &attributes, &size, &capacity, "V003", "Vessel C", "Cargo", "Furniture", 0, 1); // cargo


    displayFleet(&fleet, &attributes, size);


    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)fleet[i].vesselID);
```

```
    }
    free(fleet);
    free(attributes);

    return 0;
}

// Function to add a vessel to the fleet
void addVessel(Vessel **fleet, Attributes **attributes, int *size, int *capacity, const char *vesselID, const char *name, const char *type, const char *cargoType, int passengerCount, int isCargo) {

    if (*size == *capacity) {
        *capacity *= 2;
        Vessel *newFleet = (Vessel *)malloc(*capacity * sizeof(Vessel));
        Attributes *newAttributes = (Attributes *)malloc(*capacity * sizeof(Attributes));

        // Copy existing data to new arrays
        for (int i = 0; i < *size; i++) {
            newFleet[i] = (*fleet)[i];
            newAttributes[i] = (*attributes)[i];
        }

        // Free old arrays and update pointers
        free(*fleet);
        free(*attributes);
        *fleet = newFleet;
        *attributes = newAttributes;
    }
```

```
    // Initialize the new vessel
    (*fleet)[*size].vesselID = strdup(vesselID);
    strcpy((*fleet)[*size].name, name);
    strcpy((*fleet)[*size].type, type);

    // Set the attribute (cargo type or passenger count)
    if (isCargo) {
        strcpy((*attributes)[*size].cargoType, cargoType);
    } else {
        (*attributes)[*size].passengerCount = passengerCount;
    }

    // Increment the size
    (*size)++;
}

// Function to display the fleet
void displayFleet(Vessel **fleet, Attributes **attributes, int size) {
    printf("Fleet:\n");
    for (int i = 0; i < size; i++) {
        printf("Vessel ID: %s\n", (*fleet)[i].vesselID);
        printf("Name: %s\n", (*fleet)[i].name);
        printf("Type: %s\n", (*fleet)[i].type);
        if (strcmp((*fleet)[i].type, "Cargo") == 0) {
            printf("Cargo Type: %s\n", (*attributes)[i].cargoType);
        } else {
            printf("Passenger Count: %d\n", (*attributes)[i].passengerCount);
```

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

/**********************************************************************
****
23. 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>
#include <string.h>

typedef struct {
```

```
    const char *taskID;
    char description[100];
    char schedule[50];
} MaintenanceTask;



void addTask(MaintenanceTask **tasks, int *size, int *capacity, const char *taskID, const char *description, const char *schedule);
void displayTasks(MaintenanceTask **tasks, int size);

int main() {
    int size = 0, capacity = 2;
    MaintenanceTask *tasks = (MaintenanceTask *)malloc(capacity * sizeof(MaintenanceTask));

    addTask(&tasks, &size, &capacity, "T001", "Check engine oil levels", "2025-01-05");
    addTask(&tasks, &size, &capacity, "T002", "Inspect hull for damage", "2025-01-10");
    addTask(&tasks, &size, &capacity, "T003", "Test navigation systems", "2025-01-15");

    displayTasks(&tasks, size);



    for (int i = 0; i < size; i++) {
        free((void *)tasks[i].taskID);
    }
    free(tasks);

    return 0;
}
```

```
void addTask(MaintenanceTask **tasks, int *size, int *capacity, const char *taskID, const char *description, const char *schedule) {

    if (*size == *capacity) {
        *capacity *= 2;
        MaintenanceTask *newTasks = (MaintenanceTask *)malloc(*capacity * sizeof(MaintenanceTask));

        for (int i = 0; i < *size; i++) {
            newTasks[i] = (*tasks)[i];
        }

        free(*tasks);
        *tasks = newTasks;
    }

    (*tasks)[*size].taskID = strdup(taskID);
    strcpy((*tasks)[*size].description, description);
    strcpy((*tasks)[*size].schedule, schedule);

    (*size)++;
}
```

```
void displayTasks(MaintenanceTask **tasks, int size) {
   printf("Maintenance Tasks:\n");
   for (int i = 0; i < size; i++) {
      printf("Task ID: %s\n", (*tasks)[i].taskID);
      printf("Description: %s\n", (*tasks)[i].description);
      printf("Schedule: %s\n", (*tasks)[i].schedule);
      printf("\n");
   }
}
```

/**********
****

24. 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>
#include <string.h>


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


typedef union {
    float weight;
    struct {
        float length;
        float width;
        float height;
    } dimensions;
    float temperature;
} CargoProperties;


typedef struct {
    const char *cargoID;
    CargoProperties properties;
```

```
    int isWeight; // 1 if weight property is used, 0 if dimensions property is used, 2 if temperature property is used
} Cargo;



void addCargo(Cargo **cargos, int *size, int *capacity, const char *cargoID, CargoProperties properties, int isWeight);
void displayCargos(Cargo **cargos, int size);

int main() {
    int size = 0, capacity = 2;
    Cargo *cargos = (Cargo *)malloc(capacity * sizeof(Cargo));

    // Define vessel specifications
    VesselSpecs vessel = {10000.0, 50.0, 20.0, 15.0};

    // Define some cargo properties
    CargoProperties cargo1;
    cargo1.weight = 2000.0;

    CargoProperties cargo2;
    cargo2.dimensions.length = 2.0;
    cargo2.dimensions.width = 2.0;
    cargo2.dimensions.height = 2.0;

    CargoProperties cargo3;
    cargo3.temperature = -10.0;

    // Add cargos to the list
```

```
    addCargo(&cargos, &size, &capacity, "C001", cargo1, 1); // weight
    addCargo(&cargos, &size, &capacity, "C002", cargo2, 0); // dimensions
    addCargo(&cargos, &size, &capacity, "C003", cargo3, 2); // temperature

    displayCargos(&cargos, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)cargos[i].cargoID); // Cast to void* to free const char*
    }
    free(cargos);

    return 0;
}

// Function to add a cargo to the list
void addCargo(Cargo **cargos, int *size, int *capacity, const char *cargoID, CargoProperties properties, int isWeight) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Cargo *newCargos = (Cargo *)malloc(*capacity * sizeof(Cargo));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newCargos[i] = (*cargos)[i];
        }
```

```
        // Free old array and update pointer
        free(*cargos);
        *cargos = newCargos;
    }

    // Initialize the new cargo
    (*cargos)[*size].cargoID = strdup(cargoID);
    (*cargos)[*size].properties = properties;
    (*cargos)[*size].isWeight = isWeight;

    // Increment the size
    (*size)++;
}

// Function to display the cargos
void displayCargos(Cargo **cargos, int size) {
    printf("Cargos:\n");
    for (int i = 0; i < size; i++) {
        printf("Cargo ID: %s\n", (*cargos)[i].cargoID);
        if ((*cargos)[i].isWeight == 1) {
            printf("Weight: %.2f\n", (*cargos)[i].properties.weight);
        } else if ((*cargos)[i].isWeight == 0) {
            printf("Dimensions: %.2f x %.2f x %.2f\n", (*cargos)[i].properties.dimensions.length,
(*cargos)[i].properties.dimensions.width, (*cargos)[i].properties.dimensions.height);
        } else {
            printf("Temperature Requirement: %.2f\n", (*cargos)[i].properties.temperature);
        }
        printf("\n");
```

```
    }

}

/*************************************************************************
****
25. 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>


typedef struct {
    float temperature;
    float windSpeed;
```

```
} WeatherData;



void addWeatherRecord(WeatherData **records, int *size, int *capacity, float temperature,
float windSpeed);

void displayWeatherRecords(WeatherData **records, int size);

void checkWeatherAlert(WeatherData **records, int size, const char **alertMessage);


int main() {
    int size = 0, capacity = 2;
    WeatherData *records = (WeatherData *)malloc(capacity * sizeof(WeatherData));
    const char *alertMessage = NULL;

    // Add weather records
    addWeatherRecord(&records, &size, &capacity, 25.0, 10.5);
    addWeatherRecord(&records, &size, &capacity, 30.0, 15.0);
    addWeatherRecord(&records, &size, &capacity, 35.0, 20.0);

    // Check for weather alerts
    checkWeatherAlert(&records, size, &alertMessage);

    // Display weather records
    displayWeatherRecords(&records, size);

    // Display alert message if any
    if (alertMessage != NULL) {
        printf("Alert: %s\n", alertMessage);
    }
```

```
    free(records);

    return 0;
}

// Function to add a weather record to the list
void addWeatherRecord(WeatherData **records, int *size, int *capacity, float temperature, float windSpeed) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        WeatherData *newRecords = (WeatherData *)malloc(*capacity * sizeof(WeatherData));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }

    // Initialize the new weather record
    (*records)[*size].temperature = temperature;
    (*records)[*size].windSpeed = windSpeed;
```

```
    // Increment the size
    (*size)++;
}

// Function to display the weather records
void displayWeatherRecords(WeatherData **records, int size) {
    printf("Weather Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Temperature: %.2f°C, Wind Speed: %.2f km/h\n", (*records)[i].temperature,
(*records)[i].windSpeed);
    }
    printf("\n");
}

// Function to check for weather alerts
void checkWeatherAlert(WeatherData **records, int size, const char **alertMessage) {
    for (int i = 0; i < size; i++) {
        if ((*records)[i].temperature > 30.0 && (*records)[i].windSpeed > 15.0) {
            *alertMessage = "Severe weather alert: High temperature and strong winds!";
            return;
        }
    }
    *alertMessage = NULL;
}
```

/**
****

26. 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>
#include <string.h>

typedef struct {
    const char *chartID;
    char scale[20];
    char region[50];
} ChartMetadata;
```

```
typedef union {
    float depth;
    char hazard[50];
} ChartData;


typedef struct {
    float latitude;
    float longitude;
} Coordinate;

typedef struct {
    ChartMetadata metadata;
    Coordinate *coordinates;
    int numCoordinates;
    ChartData data;
    int isDepth; // 1 if depth data is used, 0 if hazard data is used
} NauticalChart;


void addChart(NauticalChart **charts, int *size, int *capacity, const char *chartID, const char *scale, const char *region, Coordinate *coordinates, int numCoordinates, ChartData data, int isDepth);
void displayCharts(NauticalChart **charts, int size);

int main() {
    int size = 0, capacity = 2;
    NauticalChart *charts = (NauticalChart *)malloc(capacity * sizeof(NauticalChart));
```

```
    Coordinate coordinates1[] = {{34.0, -118.0}, {35.0, -119.0}};
    Coordinate coordinates2[] = {{40.0, -74.0}, {41.0, -75.0}, {42.0, -76.0}};

    ChartData data1;
    data1.depth = 30.0;

    ChartData data2;
    strcpy(data2.hazard, "Underwater rocks");

    addChart(&charts, &size, &capacity, "CH001", "1:50000", "West Coast", coordinates1, 2, data1, 1); // depth
    addChart(&charts, &size, &capacity, "CH002", "1:100000", "East Coast", coordinates2, 3, data2, 0); // hazard

    displayCharts(&charts, size);

    for (int i = 0; i < size; i++) {
        free((void *)charts[i].metadata.chartID);
        free(charts[i].coordinates);
    }
    free(charts);

    return 0;
}
```

```
// Function to add a chart to the list
void addChart(NauticalChart **charts, int *size, int *capacity, const char *chartID, const char *scale, const char *region, Coordinate *coordinates, int numCoordinates, ChartData data, int isDepth) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        NauticalChart *newCharts = (NauticalChart *)malloc(*capacity * sizeof(NauticalChart));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newCharts[i] = (*charts)[i];
        }

        // Free old array and update pointer
        free(*charts);
        *charts = newCharts;
    }

    // Initialize the new chart
    (*charts)[*size].metadata.chartID = strdup(chartID);
    strcpy((*charts)[*size].metadata.scale, scale);
    strcpy((*charts)[*size].metadata.region, region);
    (*charts)[*size].coordinates = (Coordinate *)malloc(numCoordinates * sizeof(Coordinate));
    for (int i = 0; i < numCoordinates; i++) {
        (*charts)[*size].coordinates[i] = coordinates[i];
    }
    (*charts)[*size].numCoordinates = numCoordinates;
    (*charts)[*size].data = data;
```

```
    (*charts)[*size].isDepth = isDepth;

    // Increment the size
    (*size)++;
}

// Function to display the charts
void displayCharts(NauticalChart **charts, int size) {
    printf("Nautical Charts:\n");
    for (int i = 0; i < size; i++) {
        printf("Chart ID: %s\n", (*charts)[i].metadata.chartID);
        printf("Scale: %s\n", (*charts)[i].metadata.scale);
        printf("Region: %s\n", (*charts)[i].metadata.region);
        printf("Coordinates:\n");
        for (int j = 0; j < (*charts)[i].numCoordinates; j++) {
            printf("Latitude: %.2f, Longitude: %.2f\n", (*charts)[i].coordinates[j].latitude,
(*charts)[i].coordinates[j].longitude);
        }
        if ((*charts)[i].isDepth) {
            printf("Depth: %.2f\n", (*charts)[i].data.depth);
        } else {
            printf("Hazard: %s\n", (*charts)[i].data.hazard);
        }
        printf("\n");
    }
}
```

/**
****

27. 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>

typedef struct {
    char name[50];
    const char *role;
    char schedule[50];
} Crew;
```

```
void addCrewMember(Crew **roster, int *size, int *capacity, const char *name, const char *role, const char *schedule);

void displayRoster(Crew **roster, int size);


int main() {

    int size = 0, capacity = 2;

    Crew *roster = (Crew *)malloc(capacity * sizeof(Crew));



    addCrewMember(&roster, &size, &capacity, "John Doe", "Captain", "2025-01-01 to 2025-01-31");

    addCrewMember(&roster, &size, &capacity, "Jane Smith", "First Officer", "2025-02-01 to 2025-02-28");

    addCrewMember(&roster, &size, &capacity, "Mike Johnson", "Engineer", "2025-03-01 to 2025-03-31");


    displayRoster(&roster, size);



    free(roster);


    return 0;

}



void addCrewMember(Crew **roster, int *size, int *capacity, const char *name, const char *role, const char *schedule) {


    if (*size == *capacity) {
```

```
        *capacity *= 2;
        Crew *newRoster = (Crew *)malloc(*capacity * sizeof(Crew));


        for (int i = 0; i < *size; i++) {
            newRoster[i] = (*roster)[i];
        }


        free(*roster);
        *roster = newRoster;
    }


    strcpy((*roster)[*size].name, name);
    (*roster)[*size].role = role;
    strcpy((*roster)[*size].schedule, schedule);


    (*size)++;
}


void displayRoster(Crew **roster, int size) {
    printf("Crew Roster:\n");
    for (int i = 0; i < size; i++) {
        printf("Name: %s\n", (*roster)[i].name);
        printf("Role: %s\n", (*roster)[i].role);
```

```
        printf("Schedule: %s\n", (*roster)[i].schedule);
        printf("\n");
    }
}
```

```
/**************************************************************************
****
```

28. 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>
#include <string.h>

typedef struct {
```

```
    const char *sensorID;
    char location[50];
} SensorDetails;


typedef union {
    float temperature;
    float pressure;
} SensorType;


typedef struct {
    SensorDetails details;
    SensorType type;
    int isTemperature; // 1 if temperature sensor, 0 if pressure sensor
} Sensor;


void addSensor(Sensor **sensors, int *size, int *capacity, const char *sensorID, const char *location, SensorType type, int isTemperature);
void displaySensors(Sensor **sensors, int size);

int main() {
    int size = 0, capacity = 2;
    Sensor *sensors = (Sensor *)malloc(capacity * sizeof(Sensor));


    SensorType sensor1;
```

```
    sensor1.temperature = 25.5;

    SensorType sensor2;
    sensor2.pressure = 1.2;



    addSensor(&sensors, &size, &capacity, "S001", "Location A", sensor1, 1); // temperature
    addSensor(&sensors, &size, &capacity, "S002", "Location B", sensor2, 0); // pressure

    displaySensors(&sensors, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)sensors[i].details.sensorID); // Cast to void* to free const char*
    }
    free(sensors);

    return 0;
}

// Function to add a sensor to the list
void addSensor(Sensor **sensors, int *size, int *capacity, const char *sensorID, const char
*location, SensorType type, int isTemperature) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        Sensor *newSensors = (Sensor *)malloc(*capacity * sizeof(Sensor));
```

```
        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newSensors[i] = (*sensors)[i];
        }

        // Free old array and update pointer
        free(*sensors);
        *sensors = newSensors;
    }

    // Initialize the new sensor
    (*sensors)[*size].details.sensorID = strdup(sensorID);
    strcpy((*sensors)[*size].details.location, location);
    (*sensors)[*size].type = type;
    (*sensors)[*size].isTemperature = isTemperature;

    // Increment the size
    (*size)++;
}

// Function to display the sensors
void displaySensors(Sensor **sensors, int size) {
    printf("Sensors:\n");
    for (int i = 0; i < size; i++) {
        printf("Sensor ID: %s\n", (*sensors)[i].details.sensorID);
        printf("Location: %s\n", (*sensors)[i].details.location);
        if ((*sensors)[i].isTemperature) {
            printf("Temperature: %.2f°C\n", (*sensors)[i].type.temperature);
```

```
        } else {
            printf("Pressure: %.2f bar\n", (*sensors)[i].type.pressure);
        }
        printf("\n");
    }
}

/**********************************************************************
****
29. 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>

typedef struct {
```

```
    const char *date;
    const char *author;
} LogMetadata;


typedef struct {
    LogMetadata metadata;
    char entry[200];
} LogEntry;


void addLogEntry(LogEntry **logEntries, int *size, int *capacity, const char *date, const char *author, const char *entry);
void displayLogEntries(LogEntry **logEntries, int size);

int main() {
    int size = 0, capacity = 2;
    LogEntry *logEntries = (LogEntry *)malloc(capacity * sizeof(LogEntry));


    addLogEntry(&logEntries, &size, &capacity, "2025-01-01", "Captain John", "Departed from Port A.");
    addLogEntry(&logEntries, &size, &capacity, "2025-01-02", "First Officer Jane", "Encountered rough seas.");
    addLogEntry(&logEntries, &size, &capacity, "2025-01-03", "Engineer Mike", "Routine maintenance performed.");

    displayLogEntries(&logEntries, size);
```

```
    for (int i = 0; i < size; i++) {
        free((void *)logEntries[i].metadata.date);
        free((void *)logEntries[i].metadata.author);
    }
    free(logEntries);

    return 0;
}

// Function to add a log entry to the list
void addLogEntry(LogEntry **logEntries, int *size, int *capacity, const char *date, const char *author, const char *entry) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        LogEntry *newLogEntries = (LogEntry *)malloc(*capacity * sizeof(LogEntry));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newLogEntries[i] = (*logEntries)[i];
        }

        // Free old array and update pointer
        free(*logEntries);
        *logEntries = newLogEntries;
    }

    // Initialize the new log entry
```

```
    (*logEntries)[*size].metadata.date = strdup(date);
    (*logEntries)[*size].metadata.author = strdup(author);
    strcpy((*logEntries)[*size].entry, entry);

    // Increment the size
    (*size)++;
}

// Function to display the log entries
void displayLogEntries(LogEntry **logEntries, int size) {
    printf("Ship Log Entries:\n");
    for (int i = 0; i < size; i++) {
        printf("Date: %s\n", (*logEntries)[i].metadata.date);
        printf("Author: %s\n", (*logEntries)[i].metadata.author);
        printf("Entry: %s\n", (*logEntries)[i].entry);
        printf("\n");
    }
}
```

/**
****

30. 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>


typedef struct {
    const char *navID;
    const char **waypoints;
    int numWaypoints;
} NavigationDetails;
```

```
void addNavigation(NavigationDetails **navigations, int *size, int *capacity, const char *navID, const char **waypoints, int numWaypoints);
void displayNavigations(NavigationDetails **navigations, int size);

int main() {
    int size = 0, capacity = 2;
    NavigationDetails *navigations = (NavigationDetails *)malloc(capacity * sizeof(NavigationDetails));

    const char *waypoints1[] = {"Waypoint1", "Waypoint2", "Waypoint3"};
    const char *waypoints2[] = {"WaypointA", "WaypointB"};

    addNavigation(&navigations, &size, &capacity, "NAV001", waypoints1, 3);
    addNavigation(&navigations, &size, &capacity, "NAV002", waypoints2, 2);

    displayNavigations(&navigations, size);

    for (int i = 0; i < size; i++) {
        free((void *)navigations[i].navID);
        free(navigations[i].waypoints);
    }
    free(navigations);

    return 0;
}
```

```
// Function to add navigation details to the list
void addNavigation(NavigationDetails **navigations, int *size, int *capacity, const char *navID, const char **waypoints, int numWaypoints) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        NavigationDetails *newNavigations = (NavigationDetails *)malloc(*capacity * sizeof(NavigationDetails));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newNavigations[i] = (*navigations)[i];
        }

        // Free old array and update pointer
        free(*navigations);
        *navigations = newNavigations;
    }

    // Initialize the new navigation details
    (*navigations)[*size].navID = strdup(navID);
    (*navigations)[*size].waypoints = (const char **)malloc(numWaypoints * sizeof(const char *));
    for (int i = 0; i < numWaypoints; i++) {
        (*navigations)[*size].waypoints[i] = strdup(waypoints[i]);
    }
    (*navigations)[*size].numWaypoints = numWaypoints;

    // Increment the size
```

```
    (*size)++;
}

// Function to display the navigation details
void displayNavigations(NavigationDetails **navigations, int size) {
    printf("Navigation Waypoints:\n");
    for (int i = 0; i < size; i++) {
        printf("Navigation ID: %s\n", (*navigations)[i].navID);
        printf("Waypoints: ");
        for (int j = 0; j < (*navigations)[i].numWaypoints; j++) {
            printf("%s ", (*navigations)[i].waypoints[j]);
        }
        printf("\n\n");
    }
}
```

/**
****

31. 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>
#include <stdlib.h>
#include <string.h>


typedef struct {
    const char *speciesID;
    char speciesName[50];
    char location[100];
} AnimalData;

typedef struct {
    AnimalData data;
```

```
    char observation[200];
} ObservationRecord;

void addObservation(ObservationRecord **records, int *size, int *capacity, const char *speciesID, const char *speciesName, const char *location, const char *observation);
void displayObservations(ObservationRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    ObservationRecord *records = (ObservationRecord *)malloc(capacity * sizeof(ObservationRecord));

    addObservation(&records, &size, &capacity, "SP001", "Dolphin", "Pacific Ocean", "Sighted a pod of dolphins swimming near the shore.");
    addObservation(&records, &size, &capacity, "SP002", "Sea Turtle", "Caribbean Sea", "Observed a sea turtle laying eggs on the beach.");
    addObservation(&records, &size, &capacity, "SP003", "Shark", "Atlantic Ocean", "Spotted a great white shark near a coral reef.");

    displayObservations(&records, size);

    for (int i = 0; i < size; i++) {
        free((void *)records[i].data.speciesID);
    }
    free(records);
```

```
    return 0;
}


void addObservation(ObservationRecord **records, int *size, int *capacity, const char *speciesID, const char *speciesName, const char *location, const char *observation) {

    if (*size == *capacity) {
        *capacity *= 2;
        ObservationRecord *newRecords = (ObservationRecord *)malloc(*capacity * sizeof(ObservationRecord));


        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }


        free(*records);
        *records = newRecords;
    }

    (*records)[*size].data.speciesID = strdup(speciesID);
    strcpy((*records)[*size].data.speciesName, speciesName);
    strcpy((*records)[*size].data.location, location);
    strcpy((*records)[*size].observation, observation);


    (*size)++;
```

```
}


void displayObservations(ObservationRecord **records, int size) {
   printf("Marine Wildlife Observations:\n");
   for (int i = 0; i < size; i++) {
      printf("Species ID: %s\n", (*records)[i].data.speciesID);
      printf("Species Name: %s\n", (*records)[i].data.speciesName);
      printf("Location: %s\n", (*records)[i].data.location);
      printf("Observation: %s\n", (*records)[i].observation);
      printf("\n");
   }
}
```

/**********

32. 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>
#include <stdlib.h>
#include <string.h>


typedef struct {
    const char *beaconID;
    char type[50];
    char location[100];
} BeaconMetadata;


typedef union {
    char lightSignal[50];
    char soundSignal[50];
} BeaconType;


typedef struct {
    BeaconMetadata metadata;
    BeaconType type;
    int isLightSignal; // 1 if light signal, 0 if sound signal
} SignalData;
```

```
void addBeacon(SignalData **beacons, int *size, int *capacity, const char *beaconID, const char *type, const char *location, BeaconType signalType, int isLightSignal);

void displayBeacons(SignalData **beacons, int size);


int main() {
    int size = 0, capacity = 2;
    SignalData *beacons = (SignalData *)malloc(capacity * sizeof(SignalData));

    BeaconType beacon1;
    strcpy(beacon1.lightSignal, "Flashing light");

    BeaconType beacon2;
    strcpy(beacon2.soundSignal, "Fog horn");



    addBeacon(&beacons, &size, &capacity, "B001", "Lighthouse", "Cape Cod", beacon1, 1); // light signal
    addBeacon(&beacons, &size, &capacity, "B002", "Buoy", "San Francisco Bay", beacon2, 0); // sound signal

    displayBeacons(&beacons, size);

    for (int i = 0; i < size; i++) {
        free((void *)beacons[i].metadata.beaconID);
    }
    free(beacons);

    return 0;
}
```

```
// Function to add a beacon to the list
void addBeacon(SignalData **beacons, int *size, int *capacity, const char *beaconID, const char *type, const char *location, BeaconType signalType, int isLightSignal) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        SignalData *newBeacons = (SignalData *)malloc(*capacity * sizeof(SignalData));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newBeacons[i] = (*beacons)[i];
        }

        // Free old array and update pointer
        free(*beacons);
        *beacons = newBeacons;
    }

    // Initialize the new beacon
    (*beacons)[*size].metadata.beaconID = strdup(beaconID);
    strcpy((*beacons)[*size].metadata.type, type);
    strcpy((*beacons)[*size].metadata.location, location);
    (*beacons)[*size].type = signalType;
    (*beacons)[*size].isLightSignal = isLightSignal;

    // Increment the size
    (*size)++;
```

```
}

// Function to display the beacons
void displayBeacons(SignalData **beacons, int size) {
    printf("Coastal Navigation Beacons:\n");
    for (int i = 0; i < size; i++) {
        printf("Beacon ID: %s\n", (*beacons)[i].metadata.beaconID);
        printf("Type: %s\n", (*beacons)[i].metadata.type);
        printf("Location: %s\n", (*beacons)[i].metadata.location);
        if ((*beacons)[i].isLightSignal) {
            printf("Signal: %s\n", (*beacons)[i].type.lightSignal);
        } else {
            printf("Signal: %s\n", (*beacons)[i].type.soundSignal);
        }
        printf("\n");
    }
}
```

/**********

33. 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>
#include <stdlib.h>
#include <string.h>


typedef struct {
    const char *fuelType;
    float quantity;
} FuelData;


typedef struct {
    FuelData fuel;
    char date[20];
} ConsumptionLog;

// Function prototypes
void addLog(ConsumptionLog **logs, int *size, int *capacity, const char *fuelType, float quantity, const char *date);
void displayLogs(ConsumptionLog **logs, int size);

int main() {
    int size = 0, capacity = 2;
```

```
    ConsumptionLog *logs = (ConsumptionLog *)malloc(capacity * sizeof(ConsumptionLog));

    addLog(&logs, &size, &capacity, "Diesel", 500.0, "2025-01-01");
    addLog(&logs, &size, &capacity, "Gasoline", 300.0, "2025-01-02");
    addLog(&logs, &size, &capacity, "Diesel", 700.0, "2025-01-03");

    displayLogs(&logs, size);

    for (int i = 0; i < size; i++) {
        free((void *)logs[i].fuel.fuelType);
    }
    free(logs);

    return 0;
}

void addLog(ConsumptionLog **logs, int *size, int *capacity, const char *fuelType, float
quantity, const char *date) {

    if (*size == *capacity) {
        *capacity *= 2;
        ConsumptionLog *newLogs = (ConsumptionLog *)malloc(*capacity *
sizeof(ConsumptionLog));

        for (int i = 0; i < *size; i++) {
```

```
            newLogs[i] = (*logs)[i];
        }


        free(*logs);
        *logs = newLogs;
    }

    // Initialize the new consumption log
    (*logs)[*size].fuel.fuelType = strdup(fuelType);
    (*logs)[*size].fuel.quantity = quantity;
    strcpy((*logs)[*size].date, date);

    // Increment the size
    (*size)++;
}

// Function to display the consumption logs
void displayLogs(ConsumptionLog **logs, int size) {
    printf("Fuel Consumption Logs:\n");
    for (int i = 0; i < size; i++) {
        printf("Fuel Type: %s\n", (*logs)[i].fuel.fuelType);
        printf("Quantity: %.2f\n", (*logs)[i].fuel.quantity);
        printf("Date: %s\n", (*logs)[i].date);
        printf("\n");
    }
}
```

```
/*************************************************************************
****
34. 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>
#include <stdlib.h>
#include <string.h>


typedef struct {
    const char *responseID;
    char location[100];
    char type[50];
} ResponseDetails;


typedef struct {
    ResponseDetails details;
```

```
    char message[200];
} AlertHistory;



void addAlert(AlertHistory **alerts, int *size, int *capacity, const char *responseID, const char *location, const char *type, const char *message);
void displayAlerts(AlertHistory **alerts, int size);

int main() {
    int size = 0, capacity = 2;
    AlertHistory *alerts = (AlertHistory *)malloc(capacity * sizeof(AlertHistory));



    addAlert(&alerts, &size, &capacity, "E001", "New York Harbor", "Fire", "Fire reported on deck.");
    addAlert(&alerts, &size, &capacity, "E002", "Los Angeles Port", "Medical", "Crew member injured.");
    addAlert(&alerts, &size, &capacity, "E003", "Miami Dock", "Security", "Unauthorized access detected.");

    displayAlerts(&alerts, size);

    for (int i = 0; i < size; i++) {
        free((void *)alerts[i].details.responseID);
    }
    free(alerts);

    return 0;
}
```

```
void addAlert(AlertHistory **alerts, int *size, int *capacity, const char *responseID, const char *location, const char *type, const char *message) {

    if (*size == *capacity) {
        *capacity *= 2;
        AlertHistory *newAlerts = (AlertHistory *)malloc(*capacity * sizeof(AlertHistory));

        for (int i = 0; i < *size; i++) {
            newAlerts[i] = (*alerts)[i];
        }

        free(*alerts);
        *alerts = newAlerts;
    }

    (*alerts)[*size].details.responseID = strdup(responseID);
    strcpy((*alerts)[*size].details.location, location);
    strcpy((*alerts)[*size].details.type, type);
    strcpy((*alerts)[*size].message, message);

    (*size)++;
}
```

```
void displayAlerts(AlertHistory **alerts, int size) {
   printf("Emergency Response Alerts:\n");
   for (int i = 0; i < size; i++) {
      printf("Response ID: %s\n", (*alerts)[i].details.responseID);
      printf("Location: %s\n", (*alerts)[i].details.location);
      printf("Type: %s\n", (*alerts)[i].details.type);
      printf("Message: %s\n", (*alerts)[i].message);
      printf("\n");
   }
}
```

/**********
****

35. 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>
#include<stdlib.h>
#include<string.h>

typedef struct {
    float speed;
    float capacity;
} ShipSpecs;



typedef union {
    float weatherImpact;
    float fuelConsumption;
} VariableFactors;



typedef struct {
    const char *metricID;
    float value;
    VariableFactors factor;
    int isWeatherImpact; // 1 if weather impact is used, 0 if fuel consumption is used
} PerformanceMetric;



void addPerformanceRecord(PerformanceMetric **records, int *size, int *capacity, const char *metricID, float value, VariableFactors factor, int isWeatherImpact);
void displayPerformanceRecords(PerformanceMetric **records, int size);
```

```
int main() {
    int size = 0, capacity = 2;
    PerformanceMetric *records = (PerformanceMetric *)malloc(capacity *
sizeof(PerformanceMetric));

    // Define some ship specifications
    ShipSpecs ship = {25.0, 5000.0}; // speed in knots, capacity in tons

    // Define some variable factors
    VariableFactors factor1;
    factor1.weatherImpact = 1.2;

    VariableFactors factor2;
    factor2.fuelConsumption = 300.0;

    // Add performance records to the list
    addPerformanceRecord(&records, &size, &capacity, "PM001", 100.0, factor1, 1); //
weather impact
    addPerformanceRecord(&records, &size, &capacity, "PM002", 200.0, factor2, 0); // fuel
consumption

    displayPerformanceRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].metricID); // Cast to void* to free const char*
    }
    free(records);
```

```
    return 0;
}

// Function to add a performance record to the list
void addPerformanceRecord(PerformanceMetric **records, int *size, int *capacity, const char *metricID, float value, VariableFactors factor, int isWeatherImpact) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        PerformanceMetric *newRecords = (PerformanceMetric *)malloc(*capacity * sizeof(PerformanceMetric));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
    }

    // Initialize the new performance record
    (*records)[*size].metricID = strdup(metricID);
    (*records)[*size].value = value;
    (*records)[*size].factor = factor;
    (*records)[*size].isWeatherImpact = isWeatherImpact;

    // Increment the size
```

```
    (*size)++;
}

// Function to display the performance records
void displayPerformanceRecords(PerformanceMetric **records, int size) {
    printf("Ship Performance Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Metric ID: %s\n", (*records)[i].metricID);
        printf("Value: %.2f\n", (*records)[i].value);
        if ((*records)[i].isWeatherImpact) {
            printf("Weather Impact: %.2f\n", (*records)[i].factor.weatherImpact);
        } else {
            printf("Fuel Consumption: %.2f\n", (*records)[i].factor.fuelConsumption);
        }
        printf("\n");
    }
}
```

```
/*************************************************************************
****
36. 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>


typedef struct {
    const char *portID;
    int capacity;
    char location[100];
} PortDetails;


typedef struct {
    const char *scheduleID;
```

```
    char vesselName[50];
    char dockingTime[20];
    PortDetails port;
} DockingSchedule;

void addSchedule(DockingSchedule **schedules, int *size, int *capacity, const char *scheduleID, const char *vesselName, const char *dockingTime, PortDetails port);
void displaySchedules(DockingSchedule **schedules, int size);

int main() {
    int size = 0, capacity = 2;
    DockingSchedule *schedules = (DockingSchedule *)malloc(capacity * sizeof(DockingSchedule));

    PortDetails port1 = {"P001", 5, "New York Harbor"};
    PortDetails port2 = {"P002", 10, "Los Angeles Port"};

    addSchedule(&schedules, &size, &capacity, "S001", "Vessel A", "2025-01-01 08:00", port1);
    addSchedule(&schedules, &size, &capacity, "S002", "Vessel B", "2025-01-02 09:00", port2);
    addSchedule(&schedules, &size, &capacity, "S003", "Vessel C", "2025-01-03 10:00", port1);

    displaySchedules(&schedules, size);
```

```
    for (int i = 0; i < size; i++) {
        free((void *)schedules[i].scheduleID);
    }
    free(schedules);

    return 0;
}



void addSchedule(DockingSchedule **schedules, int *size, int *capacity, const char
*scheduleID, const char *vesselName, const char *dockingTime, PortDetails port) {

    if (*size == *capacity) {
        *capacity *= 2;
        DockingSchedule *newSchedules = (DockingSchedule *)malloc(*capacity *
sizeof(DockingSchedule));


        for (int i = 0; i < *size; i++) {
            newSchedules[i] = (*schedules)[i];
        }


        free(*schedules);
        *schedules = newSchedules;
    }
```

```
    (*schedules)[*size].scheduleID = strdup(scheduleID);
    strcpy((*schedules)[*size].vesselName, vesselName);
    strcpy((*schedules)[*size].dockingTime, dockingTime);
    (*schedules)[*size].port = port;


    (*size)++;
}

void displaySchedules(DockingSchedule **schedules, int size) {
    printf("Docking Schedules:\n");
    for (int i = 0; i < size; i++) {
        printf("Schedule ID: %s\n", (*schedules)[i].scheduleID);
        printf("Vessel Name: %s\n", (*schedules)[i].vesselName);
        printf("Docking Time: %s\n", (*schedules)[i].dockingTime);
        printf("Port ID: %s\n", (*schedules)[i].port.portID);
        printf("Port Capacity: %d\n", (*schedules)[i].port.capacity);
        printf("Port Location: %s\n", (*schedules)[i].port.location);
        printf("\n");
    }
}
```

/**
****

37. 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>

typedef struct {
    float depth;
    char location[100];
    char timestamp[20];
} ExplorationData;

typedef struct {
    const char *logID;
```

```
    ExplorationData data;
} LogEntry;



void addLogEntry(LogEntry **logs, int *size, int *capacity, const char *logID, float depth,
const char *location, const char *timestamp);
void displayLogEntries(LogEntry **logs, int size);

int main() {
    int size = 0, capacity = 2;
    LogEntry *logs = (LogEntry *)malloc(capacity * sizeof(LogEntry));



    addLogEntry(&logs, &size, &capacity, "LOG001", 1000.0, "Pacific Ocean", "2025-01-01
10:00");
    addLogEntry(&logs, &size, &capacity, "LOG002", 1500.0, "Atlantic Ocean", "2025-01-02
12:00");
    addLogEntry(&logs, &size, &capacity, "LOG003", 2000.0, "Indian Ocean", "2025-01-03
14:00");

    displayLogEntries(&logs, size);



    for (int i = 0; i < size; i++) {
        free((void *)logs[i].logID);
    }
    free(logs);

    return 0;
```

```
}


void addLogEntry(LogEntry **logs, int *size, int *capacity, const char *logID, float depth,
const char *location, const char *timestamp) {

    if (*size == *capacity) {
        *capacity *= 2;
        LogEntry *newLogs = (LogEntry *)malloc(*capacity * sizeof(LogEntry));



        for (int i = 0; i < *size; i++) {
            newLogs[i] = (*logs)[i];
        }



        free(*logs);
        *logs = newLogs;
    }



    (*logs)[*size].logID = strdup(logID);
    (*logs)[*size].data.depth = depth;
    strcpy((*logs)[*size].data.location, location);
    strcpy((*logs)[*size].data.timestamp, timestamp);



    (*size)++;
```

```
}


void displayLogEntries(LogEntry **logs, int size) {
   printf("Deep-Sea Exploration Log Entries:\n");
   for (int i = 0; i < size; i++) {
      printf("Log ID: %s\n", (*logs)[i].logID);
      printf("Depth: %.2f meters\n", (*logs)[i].data.depth);
      printf("Location: %s\n", (*logs)[i].data.location);
      printf("Timestamp: %s\n", (*logs)[i].data.timestamp);
      printf("\n");
   }
}
```

/**********

38. 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>

typedef struct {
    const char *commID;
    char timestamp[20];
} CommMetadata;



typedef struct {
    CommMetadata metadata;
    char message[200];
} MessageLog;



void addMessageLog(MessageLog **logs, int *size, int *capacity, const char *commID, const char *timestamp, const char *message);
void displayMessageLogs(MessageLog **logs, int size);

int main() {
    int size = 0, capacity = 2;
    MessageLog *logs = (MessageLog *)malloc(capacity * sizeof(MessageLog));

    // Add some message logs
    addMessageLog(&logs, &size, &capacity, "C001", "2025-01-01 10:00", "Engine check completed.");
```

```
    addMessageLog(&logs, &size, &capacity, "C002", "2025-01-02 11:00", "Weather conditions clear.");
    addMessageLog(&logs, &size, &capacity, "C003", "2025-01-03 12:00", "Docking at port in 30 minutes.");

    displayMessageLogs(&logs, size);



    for (int i = 0; i < size; i++) {
        free((void *)logs[i].metadata.commID);
    }
    free(logs);

    return 0;
}



void addMessageLog(MessageLog **logs, int *size, int *capacity, const char *commID, const char *timestamp, const char *message) {

    if (*size == *capacity) {
        *capacity *= 2;
        MessageLog *newLogs = (MessageLog *)malloc(*capacity * sizeof(MessageLog));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newLogs[i] = (*logs)[i];
        }
```

```
        // Free old array and update pointer
        free(*logs);
        *logs = newLogs;
    }

    // Initialize the new message log
    (*logs)[*size].metadata.commID = strdup(commID);
    strcpy((*logs)[*size].metadata.timestamp, timestamp);
    strcpy((*logs)[*size].message, message);

    // Increment the size
    (*size)++;
}

// Function to display the message logs
void displayMessageLogs(MessageLog **logs, int size) {
    printf("Ship Communication Logs:\n");
    for (int i = 0; i < size; i++) {
        printf("Communication ID: %s\n", (*logs)[i].metadata.commID);
        printf("Timestamp: %s\n", (*logs)[i].metadata.timestamp);
        printf("Message: %s\n", (*logs)[i].message);
        printf("\n");
    }
}
```

/**
****

39. 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>

typedef struct {
    const char *vesselID;
    char name[50];
} VesselDetails;

typedef union {
    char species[50];
    float weight;
```

```
} CatchData;

typedef struct {
    VesselDetails vessel;
    CatchData catchData;
    int isSpecies; // 1 if species data is used, 0 if weight data is used
} CatchRecord;



void addCatchRecord(CatchRecord **records, int *size, int *capacity, const char *vesselID,
const char *name, CatchData catchData, int isSpecies);
void displayCatchRecords(CatchRecord **records, int size);

int main() {
    int size = 0, capacity = 2;
    CatchRecord *records = (CatchRecord *)malloc(capacity * sizeof(CatchRecord));



    CatchData catch1;
    strcpy(catch1.species, "Tuna");

    CatchData catch2;
    catch2.weight = 300.5;

    // Add catch records to the list
    addCatchRecord(&records, &size, &capacity, "V001", "Fishing Vessel A", catch1, 1); //
species
    addCatchRecord(&records, &size, &capacity, "V002", "Fishing Vessel B", catch2, 0); //
weight
```

```
    displayCatchRecords(&records, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)records[i].vessel.vesselID); // Cast to void* to free const char*
    }
    free(records);

    return 0;
}

// Function to add a catch record to the list
void addCatchRecord(CatchRecord **records, int *size, int *capacity, const char *vesselID,
const char *name, CatchData catchData, int isSpecies) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        CatchRecord *newRecords = (CatchRecord *)malloc(*capacity * sizeof(CatchRecord));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newRecords[i] = (*records)[i];
        }

        // Free old array and update pointer
        free(*records);
        *records = newRecords;
```

```
    }

    // Initialize the new catch record
    (*records)[*size].vessel.vesselID = strdup(vesselID);
    strcpy((*records)[*size].vessel.name, name);
    (*records)[*size].catchData = catchData;
    (*records)[*size].isSpecies = isSpecies;

    // Increment the size
    (*size)++;
}

// Function to display the catch records
void displayCatchRecords(CatchRecord **records, int size) {
    printf("Fishing Activity Records:\n");
    for (int i = 0; i < size; i++) {
        printf("Vessel ID: %s\n", (*records)[i].vessel.vesselID);
        printf("Vessel Name: %s\n", (*records)[i].vessel.name);
        if ((*records)[i].isSpecies) {
            printf("Catch Species: %s\n", (*records)[i].catchData.species);
        } else {
            printf("Catch Weight: %.2f kg\n", (*records)[i].catchData.weight);
        }
        printf("\n");
    }
}
```

/**
****

40. 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>

typedef struct {
    const char *location;
    float depth;
} NavigationData;

typedef union {
    float temperature;
    float pressure;
```

```
} EnvironmentalConditions;



typedef struct {
    NavigationData navigation;
    EnvironmentalConditions conditions;
    int isTemperature; // 1 if temperature, 0 if pressure
} DepthReading;



void addDepthReading(DepthReading **readings, int *size, int *capacity, const char
*location, float depth, EnvironmentalConditions conditions, int isTemperature);
void displayDepthReadings(DepthReading **readings, int size);

int main() {
    int size = 0, capacity = 2;
    DepthReading *readings = (DepthReading *)malloc(capacity * sizeof(DepthReading));

    // Define some environmental conditions
    EnvironmentalConditions condition1;
    condition1.temperature = 5.0;

    EnvironmentalConditions condition2;
    condition2.pressure = 2.5;

    // Add depth readings to the list
    addDepthReading(&readings, &size, &capacity, "North Pacific", 200.0, condition1, 1); //
temperature
```

```
    addDepthReading(&readings, &size, &capacity, "South Atlantic", 300.0, condition2, 0); // pressure

    displayDepthReadings(&readings, size);

    // Free allocated memory
    for (int i = 0; i < size; i++) {
        free((void *)readings[i].navigation.location);
    }
    free(readings);

    return 0;
}

// Function to add a depth reading to the list
void addDepthReading(DepthReading **readings, int *size, int *capacity, const char *location, float depth, EnvironmentalConditions conditions, int isTemperature) {
    // Check if more memory needs to be allocated
    if (*size == *capacity) {
        *capacity *= 2;
        DepthReading *newReadings = (DepthReading *)malloc(*capacity * sizeof(DepthReading));

        // Copy existing data to new array
        for (int i = 0; i < *size; i++) {
            newReadings[i] = (*readings)[i];
        }

        // Free old array and update pointer
```

```
        free(*readings);
        *readings = newReadings;
    }

    // Initialize the new depth reading
    (*readings)[*size].navigation.location = strdup(location);
    (*readings)[*size].navigation.depth = depth;
    (*readings)[*size].conditions = conditions;
    (*readings)[*size].isTemperature = isTemperature;

    // Increment the size
    (*size)++;
}

// Function to display the depth readings
void displayDepthReadings(DepthReading **readings, int size) {
    printf("Submarine Depth Readings:\n");
    for (int i = 0; i < size; i++) {
        printf("Location: %s\n", (*readings)[i].navigation.location);
        printf("Depth: %.2f meters\n", (*readings)[i].navigation.depth);
        if ((*readings)[i].isTemperature) {
            printf("Temperature: %.2f°C\n", (*readings)[i].conditions.temperature);
        } else {
            printf("Pressure: %.2f atm\n", (*readings)[i].conditions.pressure);
        }
        printf("\n");
    }
}
```