Data Structures

A data structure is a particular organization of data in memory.
- We want to group related items together.
- We want to organize these data bundles in a way that is convenient to program and efficient to execute.

An array is one kind of data structure.
In this chapter, we look at two more:
- struct – directly supported by C
- linked list – built from struct and dynamic allocation

Structures in C

- A struct is a mechanism for grouping together related data items of different types.
  - Recall that an array groups items of a single type.
  - Example: We want to represent an airborne aircraft:

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

- We can use a struct to group data fields for each plane in a single named entity.

Defining a Struct

- We first need to define a new type for the compiler and tell it what our struct looks like.

```
struct flightType {
   char flightNum[7]; /* max 6 characters */
   int altitude; /* in meters */
   int longitude; /* in tenths of degrees */
   int latitude; /* in tenths of degrees */
   int heading; /* in tenths of degrees */
   double airSpeed; /* in km/hr */
};
```

- This tells the compiler how big our struct is and how the different data items (“members”) are laid out in memory.
- But it does not allocate any memory.
Declaring and Using a Struct

To allocate memory for a struct, we declare a variable using our new data type.

```c
struct flightType plane;
```

Memory is allocated, and we can access individual members of this variable:

```c
plane.airSpeed = 800.0;
plane.altitude = 10000;
```

A struct's members are laid out in the order specified by the definition.

Defining and Declaring at Once

You can both define and declare a struct at the same time.

```c
struct flightType
{
    char flightNum[7]; /* max 6 characters */
    int altitude; /* in meters */
    int longitude; /* in tenths of degrees */
    int latitude; /* in tenths of degrees */
    int heading; /* in tenths of degrees */
    double airSpeed; /* in km/hr */
} maverick;
```

And you can use `flightType` to declare other structs.

```c
struct flightType iceMan;
```

typedef

C provides a way to define a data type by giving a new name to a predefined type.

**Syntax:**

```
typedef <type> <name>;
```

**Examples:**

```c
typedef int Color;
typedef struct flightType Flight;
typedef struct ab_type {  
    int a;
    double b;
} ABGroup;
```

Using typedef

This gives us a way to make code more readable by giving application-specific names to types.

```c
Color pixels[500];
Flight plane1, plane2;
```

**Typical practice**

Put typedef's into a header file, and use type names in main program. If the definition of Color/Flight changes, you might not need to change the code in your main program file.
Array of Structs

- Can declare an array of structs:
  
  ```
  Flight planes[100];
  ```

  - Each array element is a struct (of size `sizeof(Flight)`).
  - To access member of a particular element:
    ```
    planes[34].altitude = 10000;
    ```

  - Because [] and . operators have the same precedence, and both associate left-to-right, this is the same as:
    ```
    (planes[34]).altitude = 10000;
    ```

Pointer to Struct

- We can declare and create a pointer to a struct:
  ```
  Flight *planePtr;
  ```

  - To access a member of the struct addressed by pointer:
    ```
    (*planePtr).altitude = 10000;
    ```

    - Because the . operator has higher precedence than *, this is **NOT** the same as:
      ```
      *planePtr.altitude = 10000;
      ```

- C provides special syntax for accessing a struct member through a pointer:
  ```
  planePtr->altitude = 10000;
  ```

Passing Structs as Arguments

- Unlike an array, a struct is always **passed by value** into a function.
  - This means the struct members are copied to the function’s activation record, and changes inside the function are not reflected in the calling routine’s copy.

  - Most of the time, you’ll want to pass **pointer** to a struct.
    ```
    int Collide(Fligh t *planeA, Flight *planeB) {
        if (planeA->altitude == planeB->altitude) {
            ...
        } else {
            return 0;
        }
    }
    ```

Dynamic Allocation

- Suppose we want our weather program to handle a **variable number of planes** – as many as the user wants to enter.
  - We can’t allocate an array, because we don’t know the maximum number of planes that might be required.
  - Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few planes’ worth of data is needed.

  **Solution:**
  Allocate storage for data dynamically, as needed.
malloc

- The Standard C Library provides a function for allocating memory at run-time: \texttt{malloc}.
  
  \texttt{void *malloc(size_t numBytes)};

- It returns a generic pointer (\texttt{void*}) to a contiguous region of memory of the requested size (in bytes).

- The bytes are allocated from a region in memory called the \textit{heap}.
  - The run-time system keeps track of chunks of memory from the heap that have been allocated.

Using malloc

- To use \texttt{malloc}, we need to know how many bytes to allocate. The \texttt{sizeof} operator asks the compiler to calculate the size of a particular type.
  
  \begin{verbatim}
  planes = malloc(n * sizeof(Flight));
  \end{verbatim}

- We may (but don't have to, because \texttt{void *} is special) change the type of the return value to the proper kind of pointer – this is called \textit{casting}.
  
  \begin{verbatim}
  planes = (Flight*) malloc(n* sizeof(Flight));
  \end{verbatim}

Example

```c
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?\n");
scanf("%d", &airbornePlanes);

planes = malloc(sizeof(Flight)*airbornePlanes);
if (planes == NULL) {
    printf("Error in allocating the data array.\n");
    ...
}
planes[0].altitude = ...
```

- If allocation fails, \texttt{malloc} returns \texttt{NULL}.

- Note: Can use array notation or pointer notation.

free and calloc

- Once the data is no longer needed, it should be released back into the heap for later use.
  - This is done using the \texttt{free} function, passing it the same address that was returned by \texttt{malloc}.
    
    \begin{verbatim}
    void free(void*);
    \end{verbatim}

- If allocated data is not freed, the program might run out of heap memory and be unable to continue.

- Sometimes we prefer to initialize allocated memory to zeros. \texttt{calloc} function does this:
    
    \begin{verbatim}
    void *calloc(size_t count, size_t size);
    \end{verbatim}
The Linked List Data Structure

- A **linked list** is an ordered collection of **nodes**, each of which contains some data, connected using **pointers**.
  - Each node points to the next node in the list.
  - The first node in the list is called the **head**.
  - The last node in the list is called the **tail**.

Node 0 — Node 1 — Node 2 — NULL

Linked List vs. Array

- A linked list can only be accessed **sequentially**.
- To find the 5th element, for instance, you must start from the head and follow the links through four other nodes.

**Advantages of linked list:**
- Dynamic size
- Easy to add additional nodes as needed
- Easy to add or remove nodes from the middle of the list (just add or redirect links)

**Advantage of array:**
- Can easily and quickly access arbitrary elements

Example: Car Lot

- Create an inventory database for a used car lot.
- Support the following actions:
  - **Search**: the database for a particular vehicle.
  - **Add**: a new car to the database.
  - **Delete**: a car from the database.
- The database must remain sorted by vehicle ID.
- Since we don’t know how many cars might be on the lot at one time, we choose a linked list representation.

Car data structure

- Each car has the following characteristics: vehicle ID, make, model, year, mileage, cost.
- Because it's a linked list, we also need a pointer to the next node in the list:

```c
typedef struct carType Car;
struct carType {
  int vehicleID;
  char make[20];
  char model[20];
  int year;
  int mileage;
  double cost;
  Car *next; /* ptr to next car in list */
};
```
Scanning the List

- Searching, adding, and deleting all require us to find a particular node in the list. We scan the list until we find a node whose ID is >= the one we’re looking for.

```c
Car * ScanList(Car *head, int searchID)
{
    Car *previous, *current;
    previous = head;
    current = head->next;
    /* Traverse until ID >= searchID */
    while ((current != NULL) && (current->vehicleID < searchID)) {
        previous = current;
        current = current->next;
    }
    return previous;
}
```

Adding a Node

- Create a new node with the proper info.
- Find the node (if any) with a greater vehicleID.
- “Splice” the new node into the list:

```
new node

Node 0   Node 1   Node 2
NULL
```

Excerpts from Code to Add a Node

```c
newNode = malloc(sizeof(Car));
    /* initialize node with new car info */
    ...
    prevNode = ScanList(head, newNode->vehicleID);
    nextNode = prevNode->next;
    if ((nextNode == NULL) || (nextNode->vehicleID != newNode->vehicleID)) {
        prevNode->next = newNode;
        newNode->next = nextNode;
    } else {
        printf("Car already exists in database.\n");
        free(newNode);
    }
```

Deleting a Node

- Find the node that points to the desired node.
- Redirect that node’s pointer to the next node (or NULL).
- Free the deleted node’s memory.
Excerpts from Code to Delete a Node

```c
printf("Enter vehicle ID of car to delete:\n");
scanf("%d", vehicleID);

prevNode = ScanList(head, vehicleID);
delNode = prevNode->next;

if ((delNode != NULL) && (delNode->vehicleID == vehicleID))
    prevNode->next = delNode->next;
    free(delNode);
else {
    printf("Vehicle not found in database.\n");
}
```

Building on Linked Lists

The linked list is a fundamental data structure.
- **Dynamic**
- Easy to add and delete nodes

The concepts described here will be helpful when learning about more elaborate data structures:
- **Trees**
- **Hash Tables**
- **Directed Acyclic Graphs**
- ...