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

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: Data for an aircraft in flight  
- We first need to define a new type for the compiler and define our memory needs for it.

```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 */
};
```

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.

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.flightNum[0]  // 8th word
plane.flightNum[6]
plane.altitude
plane.longitude
plane.latitude
plane.heading
plane.airspeed
```

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 */
} toChicago;
```

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

```c
struct flightType fromChicago;
```

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

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

Suppose our program starts out like this:

```c
int x;  
Flight plane;
int y;
plane.altitude = 0;
...
```

LC3 code for this assignment:

```c
AND R0, R1, #0  ; R0=plane
ADD R0, R0, #13 ; R0=plane
STR R1, R0, #7 ; 8th word
```

```c
plane.flightNum[0]  // 8th word
plane.flightNum[6]
plane.altitude
plane.longitude
plane.latitude
plane.heading
plane.airspeed
```
Array of Structs

- Can declare an array of structs:
  ```
  Flight planes[100];
  ```
- Each array element is a struct (7 words, in this case).
- To access member of a particular element:
  ```
  planes[34].altitude = 10000;
  ```
- Because the [] and . operators are at 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;
  planePtr = &planes[34];
  ```
- To access a member of the struct addressed by planePtr:
  ```
  (*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 a pointer to a struct.
  - Why?
    ```
    int Collide(Flight *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: `malloc`
  ```
  void *malloc(int numBytes);
  ```
- It returns a generic pointer (void*) to a contiguous region of memory of the requested size (in bytes).
- The bytes are allocated from a region in memory called the heap.
  - The run-time system keeps track of chunks of memory from the heap that have been allocated.
- To use malloc, we need to know how many bytes to allocate. The `sizeof` operator asks the compiler to calculate the size of a particular type.
  ```
  planes = malloc(n * sizeof(Flight));
  ```
- We also need to change the type of the return value to the proper kind of pointer — this is called “casting.”
  ```
  planes = (Flight *) malloc(n* sizeof(Flight));
  ```

Example

```c
int airbornePlanes;
Flight *planes;
printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);
planes = (Flight*) malloc(sizeof(Flight) * airbornePlanes);
/* If allocation fails, malloc returns null */
if (planes == NULL) {
  printf("Error in allocating the data array.
  souls, planes = NULL;}
  planes[0].altitude = ...
```
free()

- Once the data is no longer needed, it should be released back into the heap for later use.
- This is done using the `free` function, passing it the same address that was returned by `malloc`.
- If allocated data is not freed, the program might run out of heap memory and be unable to continue.
  - Memory leaks…

void *free(void *);

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

Dynamic Allocation Vs Static

- Example: Linked list: A collection of pointer-connected nodes with a clearly defined head and tail.
- 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

Building on Linked Lists

- The linked list is a fundamental dynamic data structure
  - Dynamic
  - Easy to add and delete nodes
  - Not necessarily the most efficient data structure for all problems
- The concepts described here will be helpful when learning about more elaborate data structures in CS 400/600.
  - Trees
  - Hash Tables
  - Directed Acyclic Graphs
  - Heaps
  - Etc…