

Chapter 19

Data Structures

C isn't hard:

```
void (*(*f[]))()
```

defines `f` as an array of unspecified size, of pointers to functions that return pointers to functions that return void.

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.

Key Mechanisms:

- **Array:** a mechanism to group multiple elements of a homogeneous type
- **Structure:** a mechanism to group multiple elements of a heterogeneous type

Both can be applied recursively...

Structures in C

A **struct** is a mechanism for grouping together related data items of **different types**.

- Recall that an array groups items of the same 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 these data together for each plane.

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

Memory is allocated when we declare a variable using our new data type.

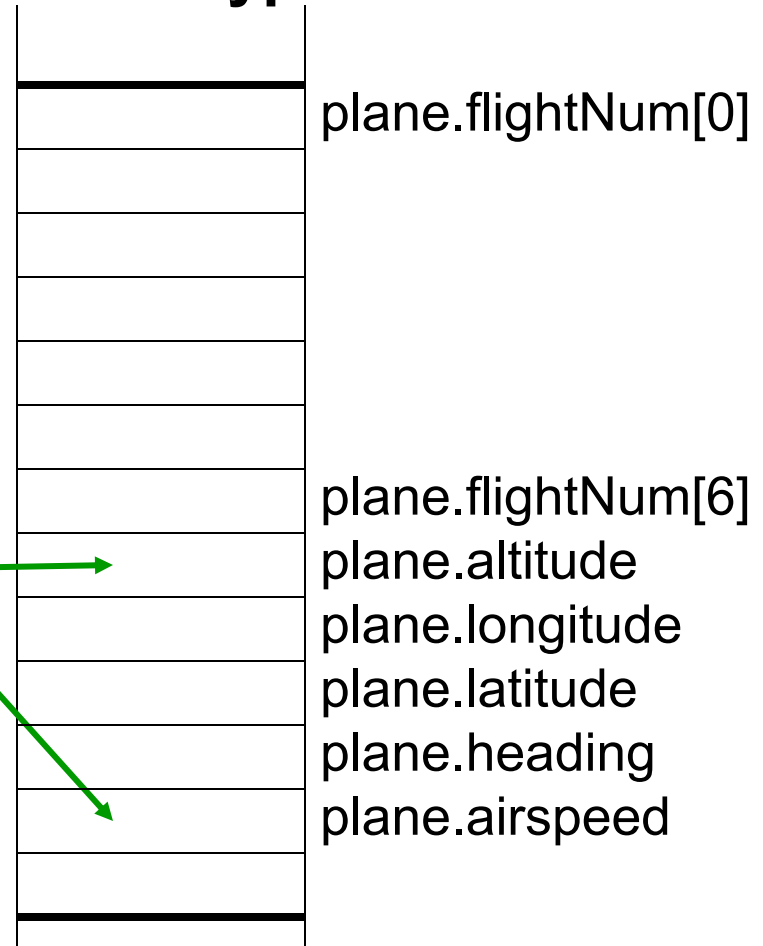
```
struct flightType plane;
```

Then, we can access individual members of this variable:

```
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.

```
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 the flightType name to declare other variables of the flightType struct type.

```
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:

```
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.

```
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 don't have to change the code in your main program file.

Generating Code for Structs

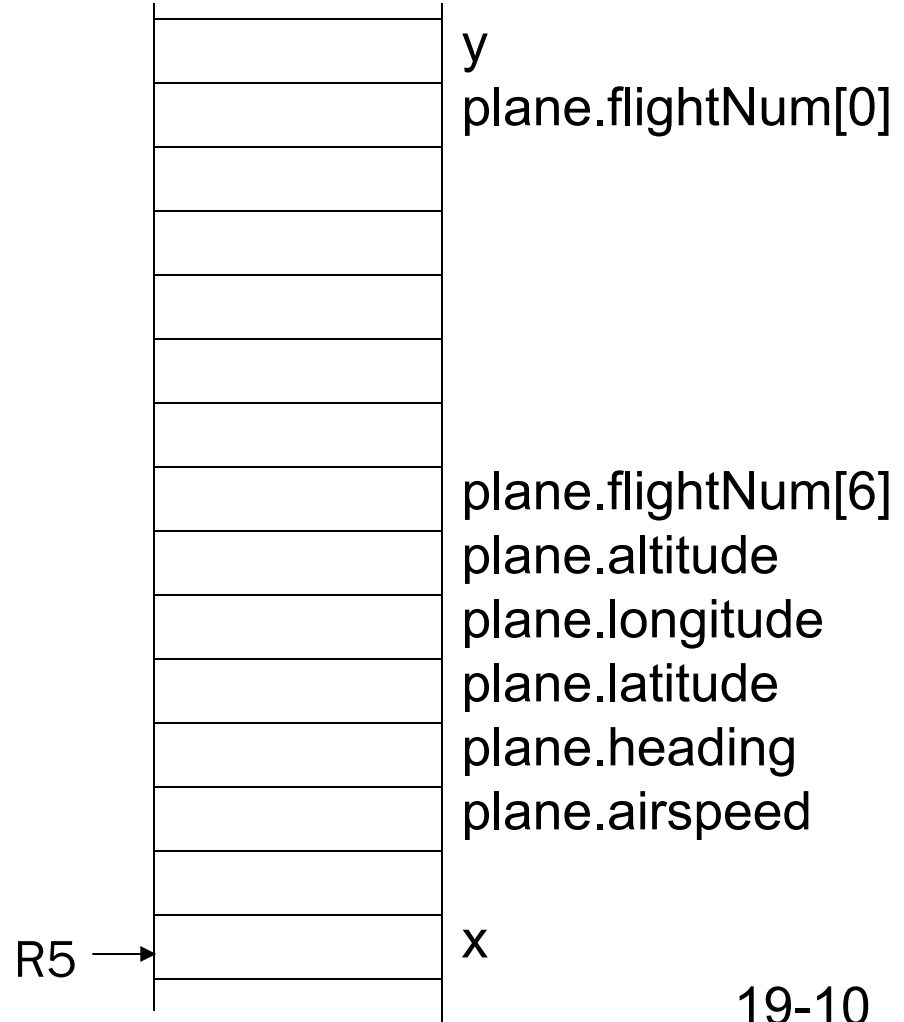
Suppose our program starts out like this:

```
int x;  
Flight plane;  
int y;
```

```
plane.altitude = 0;  
...
```

LC-3 code for this assignment:

```
AND R1, R1, #0  
ADD R0, R5, #-13 ; R0=plane  
STR R1, R0, #7 ; 8th word
```



Array of Structs

Can declare an array of structs:

```
Flight planes[100];
```

Each array element is a struct (6 members in this case).

To access a 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 dayPtr:

```
(*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.

Thus, most of the time, you'll want to pass a **pointer** to a struct.

```
int Collide(Flight *planeA, Flight *planeB)
{
    if (planeA->altitude == planeB->altitude) {
        ...
    }
    else
        return 0;
}
```

Dynamic Allocation

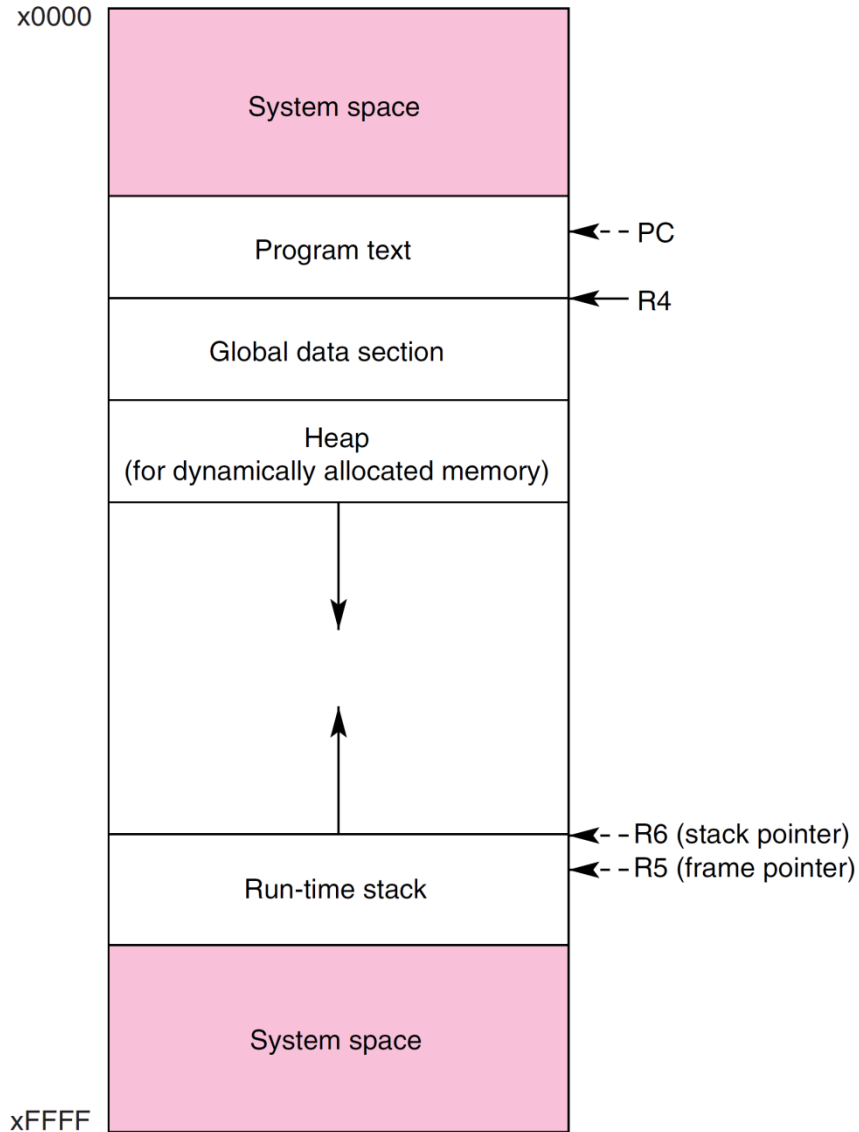
Suppose we want our weather program to handle a **variable number of planes** – as many as the user wants to enter.

- 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.

LC-3 Memory Map



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.

Using malloc

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

```
int airbornePlanes;  
Flight *planes;
```

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

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

If allocation fails,
malloc returns NULL.

Note: Can use array notation
or pointer notation.

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.

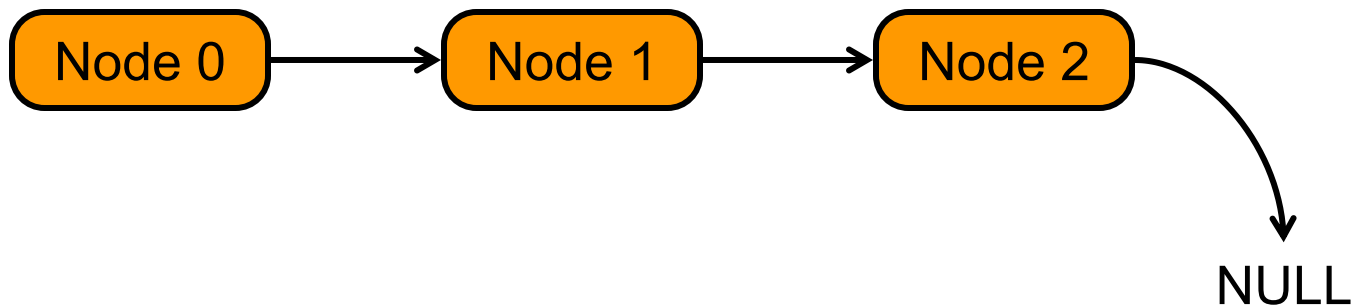
```
void free(void*);
```

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

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**.



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:**

```
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 \geq the one we're looking for.

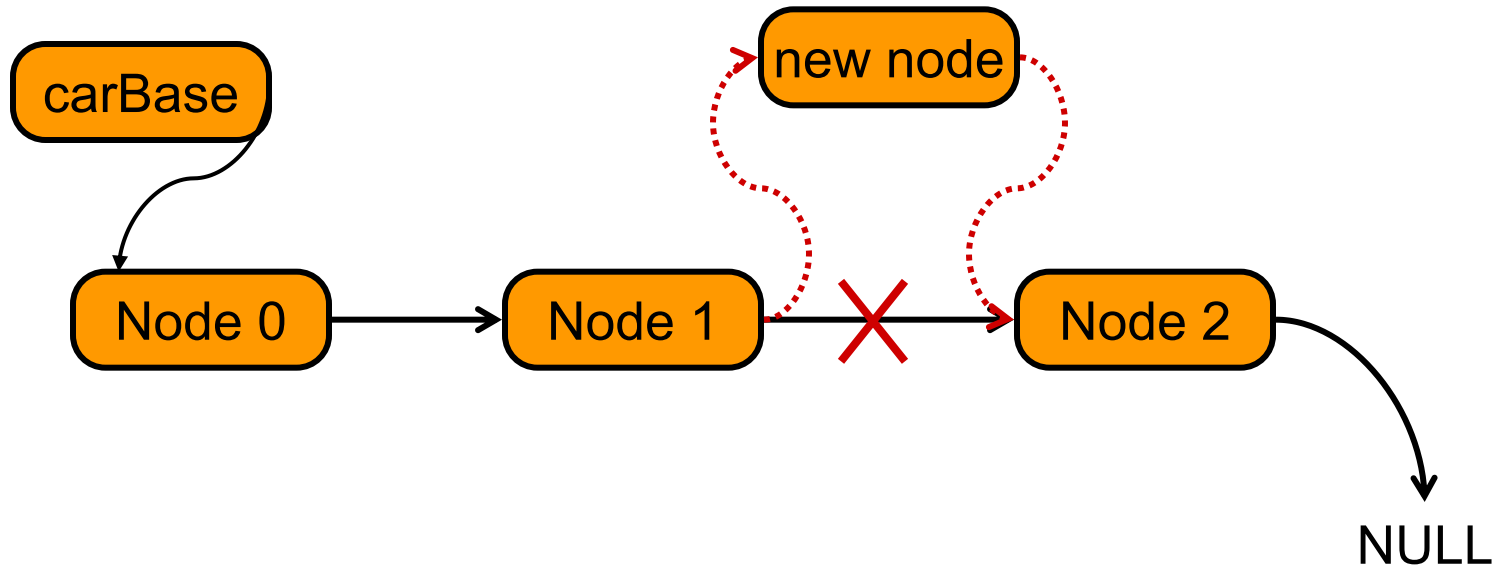
```
Car *ScanList(Car *head, int searchID)
{
    Car *previous, *current;
    previous = head;
    current = head->next;
    /* Traverse until ID  $\geq$  searchID */
    while ((current!=NULL)
           && (current->vehicleID < searchID)) {
        previous = current;
        current = current->next;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



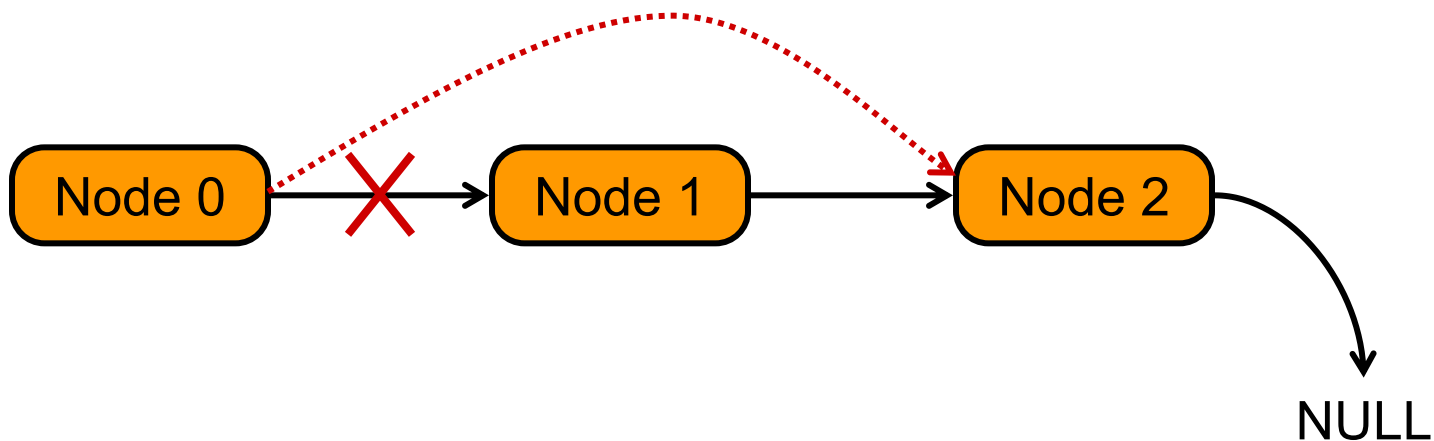
Excerpts from Code to Add a Node

```
newNode = (Car*) 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.");
    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

```
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**
- **Graphs**
- **Hash Tables**
- **...**

꼭 기억해야 할 것

- **Array:** a mechanism to group multiple elements of a homogeneous type
- **Structure:** a mechanism to group multiple elements of a heterogeneous type

Both can be applied recursively...

- **typedef:** a mechanism to define a new data type from existing ones
- **malloc, free:** library routines for dynamically allocating/deallocating memory from the heap
- pointer-based data structure
 - linked list
 - tree, graph, hash table, etc