Mc Graw Hill Education

Because learning changes everything.®

From Bits and Gates to C and Beyond

Dynamic Data Structures in C

Chapter 19

Define a Structure

Instead, we want to capture all of the information about an airplane in a single memory object. (We are using the word "object" very informally here. It's not the same as an object in a language like C++ or Python.)

Essentially, we define a new type and tell the compiler what the components of that type are:

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

The keyword `struct` is used to tell the compiler that we are defininig a new type, and `struct flightType` is the name of that type. Each component (field) of the new type of value has a type and a name.

Fields

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

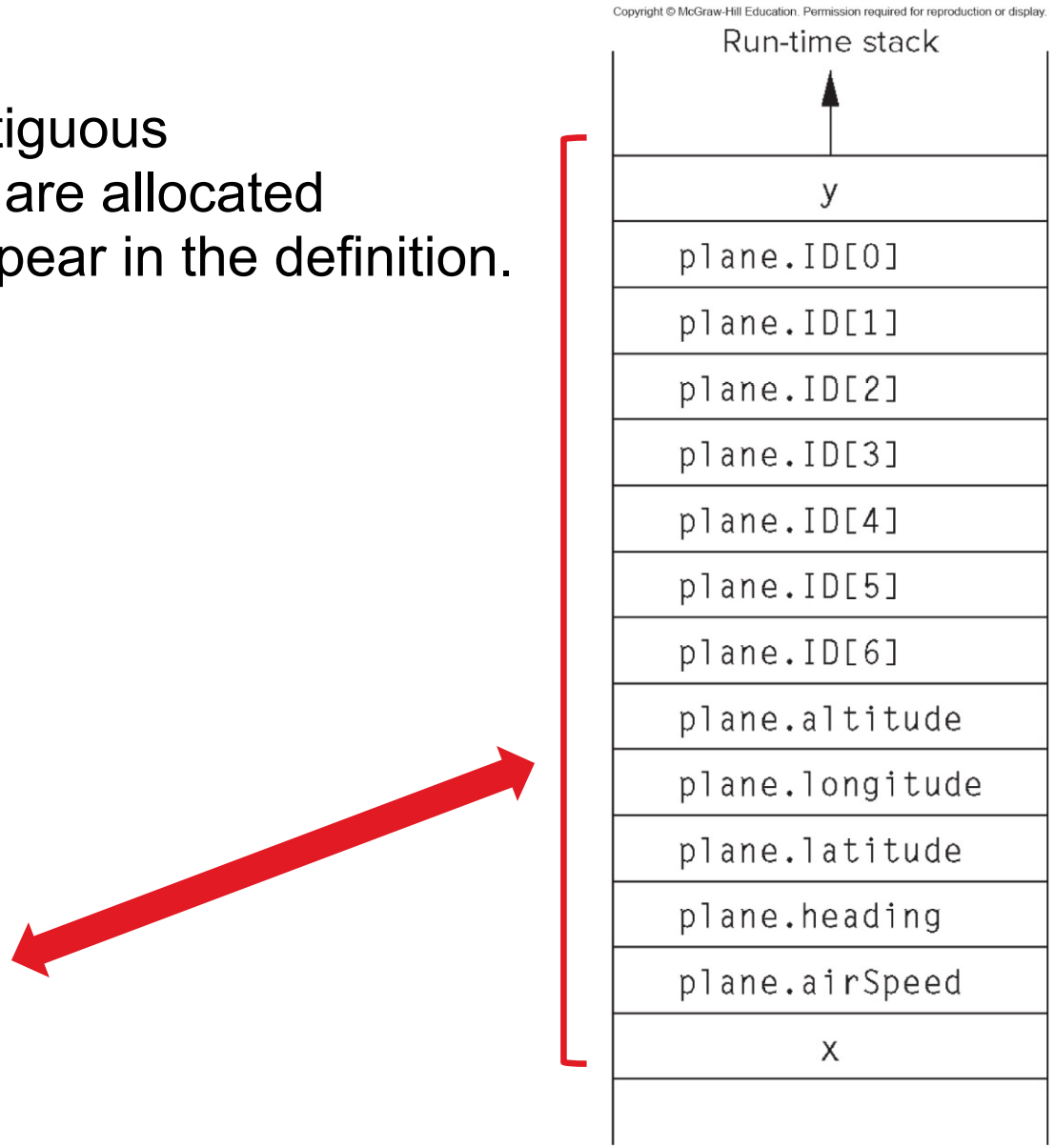
Once we have a variable, we use the dot operator (.) to access the various components (fields) of the value.

```
struct flightType plane;   // allocates a variable

plane.airspeed = 800.0;
plane.altitude = 10000;
x = plane.heading;
```

Memory Allocation

A struct is allocated in a contiguous block of memory. The fields are allocated in the order in which they appear in the definition.

```
struct flightType {
   char ID[7];
   int altitude;
   int longitude;
   int latitude;
   int heading;
   double airSpeed;
};


// local variables
int x;
struct flightType plane;
int y;
```



Note: In this illustration, a double is assumed to be one word.

Using `typedef` to Name a Type

To simplify code, or to make it more expressive, we can give a new user-defined name to any type.

`typedef` *type_spec name ;*

This allows us to create a name for our struct type, so that we don't have to type `struct` each time we declare a variable.

```
typedef struct flightType Flight;


Flight plane;    // declare a variable of type Flight
                 // allocates a struct flightType
```

By convention, a user-defined type starts with an upper-case letter.

Implementing Structures

The size of a `struct` variable is determined by the size of its fields.

- LC-3: Size of a Flight is 12 words: 7 char, 4 int, 1 double.

Each field has a constant offset from the beginning of the `struct`.
Compiler knows the offset from the definition; kept in the symbol table.

To access a field:

1. Get the base address of the struct object.
2. Add the offset to the base address.
3. Load/store to read/write the field.

Generally, steps 2 and 3 can be done in one instruction (LDR/STR).

Implementation Example

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


plane.altitude = 0;
```

```
; memory layout: slide 7

AND  R0, R0, #0     ; create zero
ADD  R1, R5, #-12   ; base addr of plane
STR  R0, R1, #7     ; store to altitude
                    ; offset = 7
```

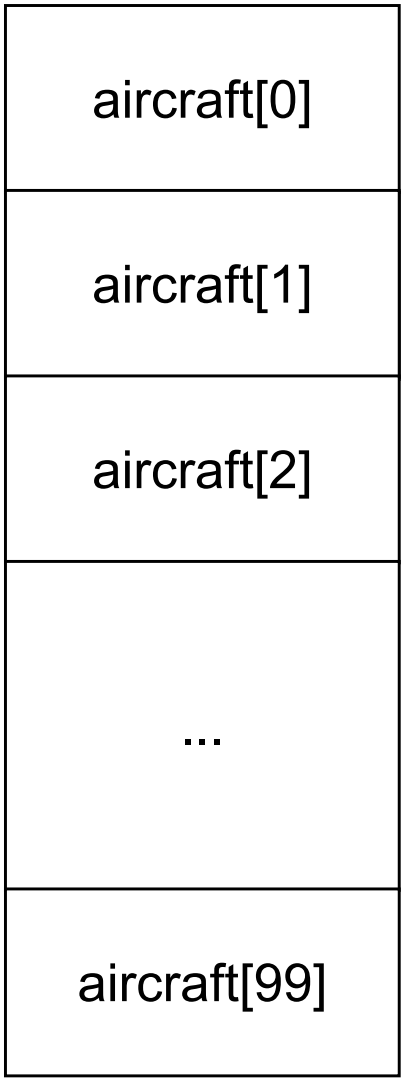
Array of Structures

Now we want a program that handles 100 airplanes.
Allocate an array:

```
Flight aircraft[100];
```

Each element of the array is a struct flightType.
Each element of the array is 12 words (LC-3).
To get to the next element of the array, add 12.

```
// calculate average air speed
sum = 0.0;
for (i = 0; i < 100; i++) {
    sum += aircraft[i].airSpeed;
}
sum = sum / 100.0;
```



Pointer to a Structure

Since a structure is a type of memory object, we can create a pointer to it.

```
Flight *aircraftPtr;
aircraftPtr = &aircraft[34];  // pt to a particular plane
```

How would we access the longitude field of that aircraft?

```
(*aircraftPtr).longitude
```

Parentheses are required, because the field operator (.) has a higher precedence than the dereference operator (*).

Since we use pointers a lot, and this notation is rather awkward and ugly, C provides a special operator (->) that dereferences a struct pointer before accessing a field.

```
aircraftPtr->longitude
```

Key point: Use dot (.) with a struct value, and arrow (->) with a struct pointer.

Passing a Structure to a Function

Structures are **passed by value**.

If we pass a structure as an argument, a copy of the structure is created. For this reason, we usually choose to pass a pointer to a structure, rather than the structure itself.

```
double AirDistance(Flight *plane1, Flight *plane2);

// in the function implementation,
// use plane1->longitude, etc.
```

Dynamic Memory Allocation

Suppose we don't know how many planes to allocate?
The user of our program will tell us, but we won't know the size of the array until the program runs.

We can specify a maximum allocate for that -- wasteful.

Instead, we use **dynamic memory allocation** to allocate exactly the amount of memory we want, exactly when we want to use it.

Two main concepts:

- The **`heap`** is where dynamic memory objects are allocated.
- We use standard functions **`malloc`** and **`free`** to allocate and deallocate memory.

The Heap

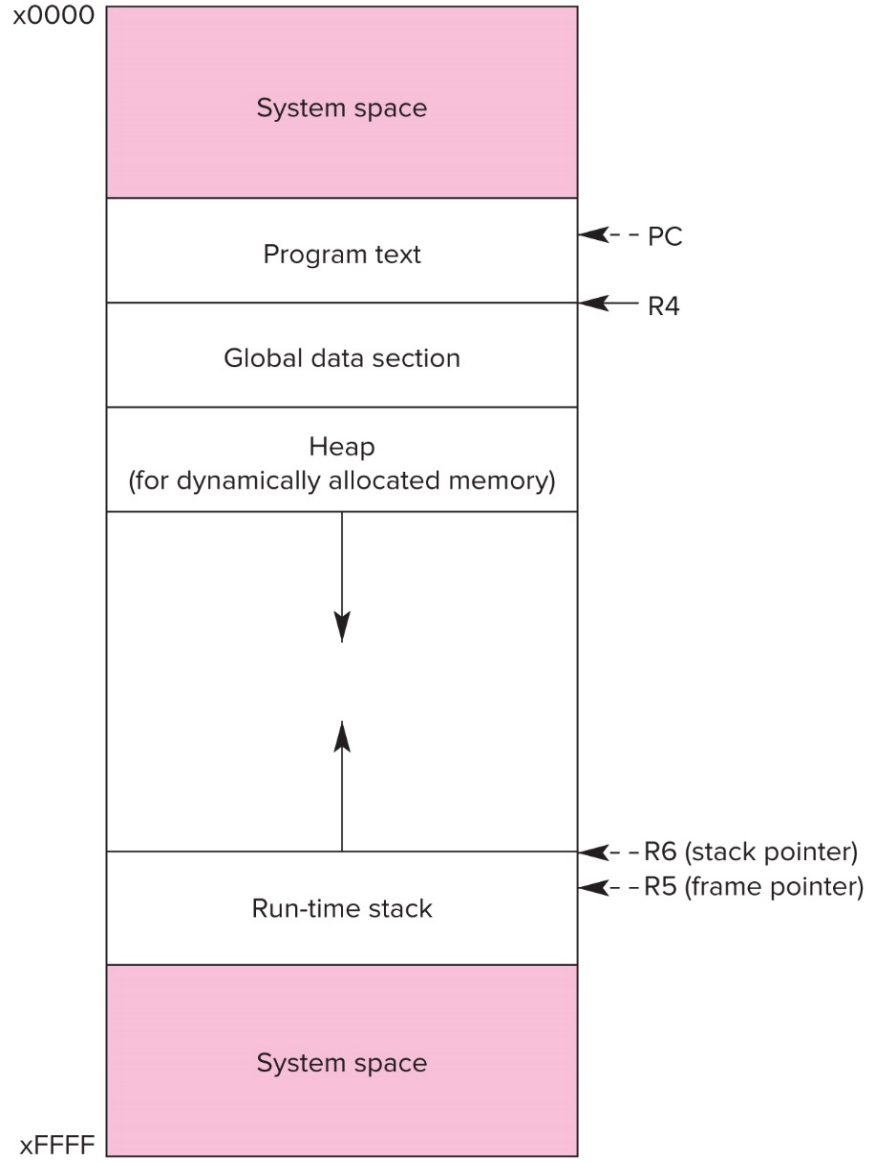
Recall the LC-3 memory map.

We have two areas to allocate variables:

1. **Global data section**:
   fixed size, determined when we compiler th
   program
2. **Runtime stack**:
   variable size, push/pop local variables and temporaries during function calls, grows downward in memory in a LIFO fashion

Now we add a third area:

The **heap** grows upward in memory.
It is not LIFO. The operating system keeps tracl of what memory is available and what is allocated. Program asks OS to allocate and deallocate memory as needed.



Memory Allocation: `malloc`

The `malloc` function allocates a block of memory and returns a pointer to the allocated block.

Argument = **number of bytes** to allocate
If the allocation cannot be performed, returns `NULL`.

```
int numAircraft;
Flight *planes;

printf("Total number of aircraft?\n");
scanf("%d", &numAircraft);

planes = malloc(24 * numAircraft);
// allocate an array of Flighs, each takes 24 bytes on LC-3
```

Bytes? Now our code is not portable! We would have to change it for each different platform. Instead, use the compiler operator `sizeof`.

```
planes = malloc(sizeof(Flight) * numAircraft);
```

Casting the Pointer from `malloc`

The `malloc` function has no idea what it is allocating, just its size.

So how does it know what kind pointer to return? It doesn't.
It is defined to return a generic pointer type: **`void*`**

It's legal to assign a `void*` value to any kind of pointer value, but we want to "cast" the pointer to the correct type, like this:

```
planes = (Flight*) malloc(sizeof(Flight) * numAircraft);
```

Casting specifies an explicit type conversion. This allows the compiler to double-check that we're doing the right thing.

Finally, we should always check to be sure that memory was actually allocated.

```
scanf("%d", &numAircraft);
planes = (Flight*) malloc(sizeof(Flight) * numAircraft);
if (planes == NULL) {
   printf("Memory allocation error!\n");
   // could return or do something else...
}
```

Memory Deallocation: `free`

If we keep allocating memory, we will eventually run out of space.

When we are finished with memory that was allocated, we should give it back to the operating system to use for other parts of the program.

The `free` function does this.

Argument = pointer that was previously returned by `malloc`
Return value: none

```
free(planes);
```

It's important to use exactly the same pointer value that was returned. The OS remembers the size that was allocated for that pointer, so it can correctly return the right amount of memory to the heap.

Best practice: Always free memory when it is no longer being used.

Memory Allocation Bugs

Using `malloc` and `free` correctly can be tricky, and is a notorious source of bugs in complex programs.

Example: The variable that holds the pointer gets changed, and the value is lost. This is known as a memory leak, because there is no way to ever deallocate the memory without having the pointer.

Example: The memory object is allocated inside a function and passed out for the rest of the program to use. Who "owns" this pointer? How do you know when it should be freed?

Example: We have a variable that points to an allocated object. We have called free, but the variable still contains the pointer. This is known as a "dangling pointer." We can still dereference the pointer -- at best, it now points to garbage data and the program crashes; at worst, we get garbage data that looks legitimate and we use it, creating a very hard-to-find error.

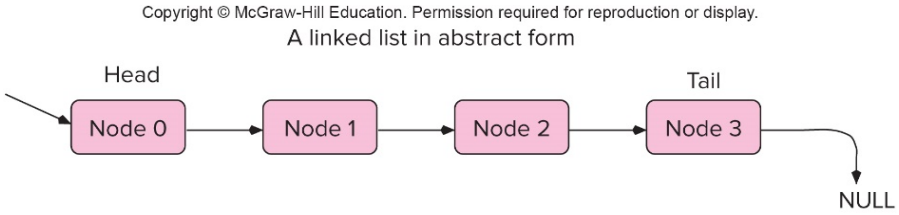
Linked List

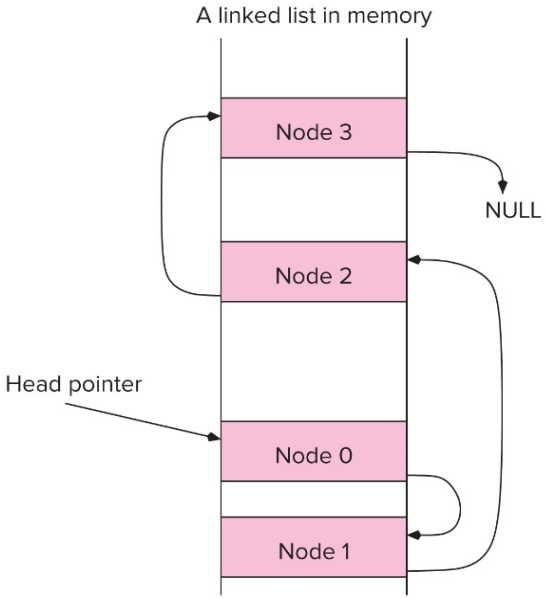
Instead of pre-allocating a number of items (e.g., array), we allocate an item whenever we need it (using `malloc`).
An item is typically called a **node** of the list.

The items are allocated in arbitrary places in memory (on the heap). Each node has a **pointer** that tells where to find the **next node** in the list.

A **head** pointer points to the beginning of the list.

A null pointer means that we're at the end of the list: there is no next node.





C Implementation of a Linked List

A linked list node must have (a) data and (b) a pointer to the next node.

```
typedef struct flightType Flight;

struct flightType {
   char ID[6];
   int altitude;
   int longitude;
   int latitude;
   int heading;
   double airSpeed;
   Flight *next;
};


Flight *list = NULL;   // list is represented by head pointer
                       // initially, the pointer is NULL
                       // because the list is empty
```

Create a List Node

```
Flight * CreateFlight(char *ID, int longitude, int latitude,
                      int altitude, double airSpeed) {
   Flight *newFlight;

   // allocate a new node
   newFlight = (Flight*)malloc(sizeof(Flight));

   // initialize node data
   strcpy(newFlight->ID, ID);
   newFlight->longitude = longitude;
   newFlight->latitude = latitude;
   newFlight->altitude = altitude;
   newFlight->airSpeed = airSpeed;

   // initialize pointer
   newFlight->next = NULL;

   return newFlight;
}
```

Traversing a Linked List

Suppose we have a linked list created already. (We'll see how in the next slides.) We often want to visit each node in the list, either to do something to each node or to look for a particular node.

```
void PrintAirspace(Flight *list) {
   // list points to the head of a linked list of Flights
   int count = 1;
   printf("Aircraft in Airspace----------------------\n");
   while (list != NULL) { // while not at the end
      printf("Aircraft:  %d\n", count);
      printf("Longitude: %d\n", list->longitude);
      printf("Latitude:  %d\n", list->latitude);
      // etc. etc.
      count = count + 1;
      list = list->next;  // move to next node
   }
   printf("\n\n");
}
```

Because learning changes everything.®

www.mheducation.com