Static vs. Dynamic Allocation

- There are two different ways that multidimensional arrays could be implemented in C.
  - Static: When you know the size at compile time
    - A Static implementation which is more efficient in terms of space and probably more efficient in terms of time.
  - Dynamic: what if you don’t know the size at compile time?
    - More flexible in terms of run time definition but more complicated to understand and build
    - Dynamic data structures
    - Need to allocate memory at run-time – malloc
      - Once you are done using this, then release this memory – free

Next: Dynamic Memory Allocation

 Structures

- Programs are solving a ‘real world’ problem
  - Entities in the real world are real ‘objects’ that need to be represented using some data structure
    - With specific attributes
  - Objects may be a collection of basic data types
    - In C we call this a structure

 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.
  - 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 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

- To allocate memory for a struct, we declare a variable using our new data type.
- ```
  struct flightType plane;
```

- Memory is allocated, and we can access:
  - plane.flightNum[0]
  - plane.altitude
  - plane.longitude
  - plane.latitude
  - plane.heading
  - plane.airspeed

- 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 structs.
- ```
  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.
- Examples:
  - `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.
  - Pay attention…..need this in your Project 3

**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`  
  - `plane.flightNum[6]`  
  - `plane.flightNum[0]`
  - `plane.flightNum[6]`  
  - `plane.longitude`  
  - `plane.latitude`  
  - `plane.heading`  
  - `plane.airspeed`  
  - `y`  
  - `plane.flightNum[0]`  
  - `plane.altitude`  
  - `plane.longitude`  
  - `plane.latitude`  
  - `plane.airspeed`  
  - `x`

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

### Dynamic Allocation
- Size of all of our data structures have been defined statically
  - `int myarray[100]` reserves 100 locations
- What if size is only known at run-time?
  - Guess max size and allocate statically?
    - `int myarray[max_size]`
- Dynamic allocation
  - Ask for space at run-time
  - Need run-time support – call system to do this allocation
  - Provide a library call in C for users
- Where do you allocate this space – heap

### Typical Arrangement
- Stack grows towards zero
- Heap grows towards xFFFF
- Can run out of space!
Dynamic Allocation

- Suppose we want our 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 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?\n"); scanf("%d", &airbornePlanes);`
- `planes = (Flight*) malloc(sizeof(Flight) * airbornePlanes);`
- `if (planes == NULL) {`
  `printf("Error in allocating the data array.\n");`
- `}`
- `planes[0].altitude = ...`

Note: Can use array notation or pointer notation.
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`.

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

If allocated data is not freed, the program might run out of heap memory and be unable to continue. Even though it is a local variable, and the values are ‘destroyed’, the allocator assumes the memory is still in use!

---

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`.

---

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.

Read example in Chapter 19
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 = head;
    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:

```c
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

```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 are helpful when learning about more elaborate data structures:
  - Trees
  - Hash Tables
  - Directed Acyclic Graphs
  - Dynamic arrays
  - ...

Dynamic Arrays Implementation

Simple 2D Case
An Important Idea: what are Computers meant to do?

- Solve problems that are described in English (or Greek or French or Hindi or Chinese or ...) and using a box filled with electrons and magnetism to accomplish the task.

Problem Transformation - levels of abstraction

- The desired behavior: the application
  - Natural Language
  - Algorithm
  - Program
  - Machine Architecture
  - Micro-architecture
  - Logic Circuits
  - Devices

- The building blocks: electronic devices
Bits & Bytes to High Level Programs

- User application written in high level language
- Program runs on a processor

- How are high level programs implemented on processor?
  - Run-time stack, allocation of variables, translation of high level code to machine code
  - Map high level data structures to low level data structures
  - Struct to linear mapping in memory

- What else does software developer want after program is implemented correctly?

- PERFORMANCE!

Performance of Programs

- “Complexity” of algorithms
- How good/efficient is your algorithm
  - Measure using Big-Oh notation: O(N \log N)
- Next question: How well is the code executing on the machine ???????
  - Actual time to run the program
  - What are the factors that come into play
  - Where is the program and data stored
  - What are the actual machine instructions executed
- What are the technology trends and how do they play a role?

Next Topics

- Performance of programs
  - What to measure
  - Model?
  - Technology trends
- Memory organization basics
  - Memory hierarchy
  - Cache memory
  - Virtual memory