EL6483: Some very useful data structures commonly used in embedded systems programming (Jump tables, Circular buffers, Linked lists, Stacks and queues, Memory pools)
A jump table is essentially a list of places in the code to jump to.

- This can be thought of in C as an array of function pointers.
- In Assembly (and at the hardware level), this is simply a list of memory addresses corresponding to locations of places in the code to jump to.

Based on some conditions (e.g., sensor data, state values, etc.), an entry from the jump table is selected using the offset in the array.

Example: Consider a state machine implemented in the program that can be in one of two states state1 and state2 and under these two values, we want to call two corresponding functions update_state1() and update_state2() at some point in the code. Then, one way to call the appropriate function would be using if statements (e.g., if state == state1, etc.) or a switch statement (e.g., switch state) with case statements. Instead of if and switch statements, a jump table can be used with the function pointers stored in an array a[] = {update_state1, update_state2} and then picking the appropriate function and calling it using a[state]() where state can be any one of state1, state2, etc., with state1 and state2 defined using an enum.
Another example: consider a system with a sensor wherein depending on sensor values in a set of different ranges, we want to call different functions.

- For example, for low sensor readings, medium sensor readings, and high sensor readings, we might want to call `sensor_low()`, `sensor_mid()`, and `sensor_high()`.

- Then, we could store the sensor value ranges in one array and the function pointers in another array and based on the sensor reading, we can call the appropriate function through array lookups.

- A jump table is usually preferable to using ad-hoc if or switch statements for calling different functions under different conditions.
Using a jump table moves details into data rather than having it in code, which would be the case with ad-hoc if or switch statements, etc., i.e., the logic of the code is moved into data which can be more easily managed/changed.

Hence, if we want to change the behavior of the code, we would only need to change the data values and would not need to change code. For example, in the sensor range example above, if we wanted to add one more range between low and medium, we would simply need to update the arrays holding the sensor value ranges and the function pointers.

Also, data can be easily changed in run-time while changing behavior implementation using if or switch statements is not usually possible in run-time. Hence, using jump tables makes the code much more reconfigurable on-demand at run-time.

The interrupt vector table is also essentially a jump table. Depending on the type of the interrupt, the appropriate entry of the interrupt vector table is picked and the processor jumps to that function.
Circular Buffers

- When we have data being generated in one part of the code (e.g., in a part of the code which is reading sensor values) and being consumed (i.e., used) in another part of the code (e.g., in a part of the code which is using sensor readings to calculate actuator commands, etc.), then it would be useful to be able to think of essentially an infinite memory buffer wherein the first part of code can keep writing whenever it has data to write and the second part of code can simply wait until it sees that there is data waiting and then continue reading.
  - With an infinite memory buffer, the write part of code simply needs to keep track of a write index (i.e., where it should write to next) and does not need to know about what the read part of the code is doing. The read part of the code simply needs to keep track of a read index (i.e., where it should read from next) and wait until the write index has incremented past the read index.

- A circular buffer can be thought about as a way to mimic an infinite memory buffer by using a fixed-size buffer and considering it as connected end-to-end, i.e., when moving past the end of the fixed-size buffer, we would simply wrap around to the start of the buffer.
A circular buffer is usually implemented as a fixed-size array, e.g., `uint8_t a[256];`

The write part of the code keeps a write index, e.g., `uint32_t nw` which is initialized to 0. When it needs to write data to the buffer, it writes to `a[nw]` and then increments `nw`. If `nw` is now pointing past the end of the array, it resets `nw` to 0. Alternatively, it can simply use the modulus operator `nw % BUF_SIZE` where `BUF_SIZE` is the size of the array (e.g., 256). Then, `nw` can be incremented without having to check if it is within `BUF_SIZE`. When accessing the array, we simply use `a[nw % BUF_SIZE]`.

The read part of the code keeps a read index, e.g., `uint32_t nr` which is initialized to 0. The read part waits until `nw` is bigger than `nr` and then reads `a[nr]` and then increments `nr`. When close to the end of the array, we would need to use one of two strategies depending on how `nw` is being handled in the write part of the code.

- If `nw` is being reset to 0, then the comparison between `nr` and `nw` would need to say that new data is available even if `nw < nr` since `nw` could have wrapped around to 0 and is now close to the beginning of the array.
- Alternatively, if the modulus method of handling `nw` is used, then `nr` can also simply be incremented without wrapping around to 0 and use `a[nr % BUF_SIZE]` to read from the array; then, the comparison between `nr` and `nw` simply needs to check if `nw > nr`.

A circular buffer provides a simple way to implement first-in-first-out (FIFO) queue of data between two parts of code wherein one part of code produces data and another part of code consumes the data.

Example: One part of code reading sensor values and another part of code calculating a sliding window average of the sensor values.
Consider an embedded system in which we want to maintain a list of functions we want to call periodically (e.g., *timed tasks*). One way to maintain such a list is a fixed-size array; however, the disadvantage is that the max number of elements of the array cannot be easily changed. If the array is allocated on the heap (e.g., using `malloc`), changing the size of the array requires using `realloc` that can be potentially a time-consuming operation (to find sufficient space in the heap) and could require moving the entire contents of the array to a different location.

A linked list is an alternative method. In a linked list, each element (e.g., a `struct T`) keeps a `next` pointer that points to the next element of the linked list (e.g., `struct T {int a; struct T *next;};`). By default, the `next` for an element in the linked list is initialized to be 0 (null); hence, if an element is the last in the linked list, then the `next` for it is 0. In code, to access the elements of the linked list, we only need to keep track of the pointer to the first element of the linked list as `start`. Then, to walk through all the elements of the linked list, we could define, for example, a variable `r = start`, then update as `r = r->next` until `r` is seen to be null (0).
Linked Lists

- With a linked list, there is no specific max number of elements that can be kept (except, of course, for overall memory limits of the system). Each element is allocated on the heap and the pointer to the element is assigned as \texttt{next} of the previous last element of the linked list.

- Removing an element from a linked list is also easy. If we have a linked list with elements \(a\), \(b\), and \(c\), then to remove \(b\), we would simply set \(a->\texttt{next} = c\) and de-allocate \(b\). To remove \(a\), we would set \(\texttt{start} = b\) and de-allocate \(a\). To remove \(c\), we would set \(b->\texttt{next} = 0\) and de-allocate \(c\).
Apart from a standard memory stack, the concept of stacks is used in embedded systems programming when a last-in-first-out behavior is required.

In general:

- a stack is a last-in-first-out (LIFO) data structure; last-in-first-out (LIFO) is the same as first-in-last-out (FILO)
- a queue is a first-in-first-out (FIFO) data structure

For example, when a software code initializes multiple hardware devices (or software functions) which need to be de-initialized in a LIFO order, then a stack can be used.

Example: consider an embedded microcontroller connected to a sensor device via a serial port; then, the serial port should be initialized first and then the device should be initialized (i.e., in code, we would have `init_serial_port()` and then `init_sensor()`). On power-down, for example, we would then need to first de-initialize the sensor and then de-initialize the serial port (i.e., `deinit_sensor()` and then `deinit_serial_port()`). This corresponds to LIFO behavior.
Stacks and Queues

- A stack can be implemented most simply by a fixed-size array (e.g., `stack_array[]`) and an index variable (e.g., `nw`) that keeps track of how many elements of the array we have used so far.
  - Then, to add (push) a new element to the stack, we place it into `stack_array[nw]` and increment `nw`.
  - To pop an element from the stack, we take it from `stack_array[nw-1]` and decrement `nw`.

- A queue (FIFO) can be implemented using the circular buffer data structure discussed before. A queue is used most often when one part of code generates some data and another part of code consumes (uses) that data; for example, one part of code reading sensor data and another part of code using the sensor data in a feedback loop to calculate actuator commands.
Memory Pools

- When variables are dynamically allocated (e.g., using `malloc` or `new`) and de-allocated (e.g., using `free` or `delete`), heap fragmentation can occur over time wherein there is free memory available in the system, but not in adequately sized contiguous regions (i.e., the free memory is fragmented into small non-contiguous chunks). Fixed-size memory pools are a technique to avoid this heap fragmentation.

- In a fixed-size memory pool, memory chunks of the same fixed size are allocated/deallocated, i.e., when creating a memory pool, the size (in bytes) of each item is specified and then each allocation/deallocation from the memory pool is always of this fixed size. Hence, any free memory within the memory pool is always in chunks of this fixed size, thus preventing fragmentation.

- For example, if the item type is a struct (or a class) that includes two 32-bit integers and two 32-bit floats, the size of each item is 16 bytes (this can be calculated during compile-time using the `sizeof` operator).
With a fixed-size memory pool, since all allocations and deallocations are of the same fixed size (e.g., 16 bytes), there will not be a fragmentation since any free memory will definitely be of size sufficient to be able to allocate the required number of bytes for an item from it.

A memory pool is initialized as a memory region (which is essentially equivalent to an array) of some size and then the memory pool manager allocates fixed-size chunks from it whenever requested by the application.