DATA STRUCTURES IN C

Data structures in C are essential for organizing and storing data efficiently. Here's a brief overview of some common data structures implemented in C:

- ***ARRAYS*** : Arrays are a collection of elements of the same data type stored in contiguous memory locations. They offer constant-time access to elements but have a fixed size.
- some common types of arrays along with mnemonic "cheat codes" to help remember them:
- **One-dimensional array**: A simple linear array where elements are stored in a single row.
- Cheat code: "1D array is like a single line, easy to traverse from one end to the other."
- **Two-dimensional array**: An array of arrays, forming a grid-like structure with rows and columns.
- Cheat code: "2D arrays are like tables with rows and columns, think of it as a spreadsheet."
- **Dynamic array**: An array that can dynamically resize itself as needed, typically implemented using dynamically allocated memory.
- Cheat code: "Dynamic arrays grow or shrink dynamically, just like a rubber band."
- **Sparse array**: An array where most of the elements have the same default value, so only non-default values are stored to save space.
- Cheat code: "Sparse arrays are like a sparse forest, where only a few trees stand out among the empty spaces."
- **Jagged array**: An array of arrays where each row can have a different length.
- Cheat code: "Jagged arrays have rows that are jagged, like the teeth of a saw, with different lengths."
- **Circular array**: An array where the last element is connected to the first element, forming a loop.
- Cheat code: "Circular arrays wrap around like a carousel, with no beginning or end."

INITIALIZATION AND DECLARATION OF ARRAYS (*cheat code as: IDEA*

I – Initialization :

- To initialize an array, you specify the data type, followed by the array name, and then enclose the elements within curly braces **{}**.

Ex: int numbers [5] = {1, 2, 3, 4, 5};

D – Declaration :

- To declare an array, you specify the data type, followed by the array name, and optionally, you can specify the size of the array.

Ex: int numbers [5];

E - Empty Initialization:

- If you don't initialize the array at the time of declaration, the array elements will be initialized with default values (0 for integers).

Ex: int numbers[5] = {0}; // All elements initialized to 0

A - Automatic Size Deduction:

- If you're initializing an array with values, you can omit the size of the array, and the compiler will automatically deduce the size based on the number of elements provided

Ex: int numbers [] = {1, 2, 3, 4, 5}; // Compiler automatically deduces the size as 5

As example let us consider the queue data structure, hence the **list of things should include for queue data structure are**:

1. **Data Structure Definition**: Define the structure of the queue. This typically includes:
   The data type of elements stored in the queue.
   Pointers or indices to keep track of the front and rear of the queue.
   An array or linked list to store the elements.
2. **Initialization**: Create a function to initialize the queue.
   Initialize any pointers or indices to indicate an empty queue.

Allocate memory if necessary (for dynamic queues).

3. **Enqueue Operation**: Implement a function to add an element to the rear of the queue.
   Update the rear pointer or index.
   Handle overflow conditions if the queue is full.
4. **Dequeue Operation**: Implement a function to remove an element from the front of the queue.
   Update the front pointer or index.
   Handle underflow conditions if the queue is empty.
5. **Front Operation**: Implement a function to retrieve the element at the front of the queue without removing it.
   Check if the queue is empty before accessing the front element.
6. **Empty Check**: Implement a function to check if the queue is empty.
   Typically involves checking if the front and rear pointers or indices are equal.
7. **Full Check (for fixed-size queues)**: Implement a function to check if the queue is full.
   Check if the queue has reached its maximum capacity.
8. **Memory Management**: If using dynamic memory allocation, implement functions to free memory when the queue is destroyed.
9. **Error Handling**: Include appropriate error handling for operations such as enqueueing into a full queue or de-queueing from an empty queue.

   Optional Operations:
   Peek: Implement a function to view the element at a specific position in the queue without removing it.
   Clear: Implement a function to remove all elements from the queue, leaving it empty.
   Traverse: Implement a function to iterate through all elements in the queue.

- here are some cheat codes to remember the key operations and concepts related to implementing a queue data structure in C:
- **Front and Rear Pointers**:
  Cheat code: "FR"
  Remember that a queue has two pointers, front and rear.

Front points to the first element, and rear points to the last element.
New elements are added at the rear and removed from the front.

- **Enqueue Operation**:

  Cheat code: "ENQ"

  To enqueue an element, add it to the rear of the queue.

  Update the rear pointer/index to the new element.

- **Dequeue Operation:**

  Cheat code: "DEQ"

  To dequeue an element, remove it from the front of the queue.

  Update the front pointer/index to the next element.

- **Empty Queue Check:**

  Cheat code: "EMPTY"

  Check if the queue is empty by comparing the front and rear pointers/indices.

  If they are equal, the queue is empty.

- **Full Queue Check (for fixed-size queues):**

  Cheat code: "FULL"

  Check if the queue is full by comparing the number of elements with the maximum capacity.

  If the number of elements equals the maximum capacity, the queue is full.

- **Circular Queue:**

  Cheat code: "CIRC"

  In a circular queue, when the rear pointer/index reaches the end, it wraps around to the beginning.

  Utilize modular arithmetic to handle wrap-around.

- **Dynamic Memory Allocation:**

  Cheat code: "DYN"

  If implementing a dynamic queue using linked lists, remember to allocate and deallocate memory dynamically.

  Use malloc() to allocate memory for new nodes and free() to deallocate memory when nodes are removed.

- **Error Handling:**

  Cheat code: "ERR"

Implement error handling for operations like enqueueing into a full queue or de-queueing from an empty queue.
Return error codes or use exception handling mechanisms if available.

- **FIFO Principle**:
  Cheat code: "FIFO"
  Remember that queues follow the First-In-First-Out (FIFO) principle.
  Elements are removed in the same order they were added.
- **Traversal**:
  Cheat code: "TRAVERSE"
  To traverse the queue, start from the front and iterate through each element until reaching the rear.
  Use the front and rear pointers/indices for traversal.
  here's a basic implementation of a queue data structure using an array in C:

CODE :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100// Structure for queue
typedef struct {
   int arr [MAX_SIZE];
   int front, rear;
} Queue;
// Function to initialize the queue
void initialize Queue (Queue *q) {
   q->front = -1;
   q->rear = -1;
}
// Function to check if the queue is empty
int isEmpty(Queue *q) {
   return (q->front == -1 && q->rear == -1);
}

// Function to check if the queue is full
int is Full (Queue *q) {
   return (q->rear + 1) % MAX_SIZE == q->front;
```

```
}// Function to enqueue an element
void enqueue (Queue *q, int value) {
   if (is Full (q)) {
      printf ("Queue is full. Cannot enqueue.\n");
      return;
   } else if (is Empty (q)) {
      q->front = q->rear = 0;
   } else {
      q->rear = (q->rear + 1) % MAX_SIZE;
   }
   q->arr [q->rear] = value;
}

// Function to dequeue an element
int dequeue(Queue *q) {
   if (is Empty(q)) {
      printf ("Queue is empty. Cannot dequeue.\n");
      return -1;
   }
   int removed Value = q->arr[q->front];
   if (q->front == q->rear) {
      q->front = q->rear = -1;
   } else {
      q->front = (q->front + 1) % MAX_SIZE;
   }
   return remove Value;
}// Function to display the elements of the queue
void display Queue(Queue *q) {
   if (is Empty (q)) {
      printf ("Queue is empty.\n")
return;
   }
   Printf ("Queue elements: ");
   int i = q->front;
   while (i! = q->rear) {
      printf ("%d ", q->arr [i]);
```

```
        i = (i + 1) % MAX_SIZE;
    }
    Printf ("%d\n", q->arr [q->rear]);
}
int main () {
    Queue q;
    Initialize Queue(&q);
     Enqueue (&q, 10);
    Enqueue (&q, 20);
    Enqueue (&q, 30);
display Queue(&q);
printf ("Dequeued element: %d\n", dequeue(&q));
display Queue(q);
return 0;
}
```