Quick Sort Algorithm Explanation

Step-by-Step Explanation

1. Swap Function

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

- This function swaps the values of two integers using their pointers.
- It uses a temporary variable temp to hold the value of *a while *a is assigned the value of *b, and then *b is assigned the value of temp.

2. Partition Function

```
int partition(int *arr, int low, int high) {
    int pivot = arr[high]; // Pivot is the last element
    int i = (low - 1); // Index of the smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            // If the current element is smaller than or equal to the pivot
            i++;
            swap(&arr[i], &arr[j]);
        }
```

```
    }

    // Swap the pivot element with the element at i + 1
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

- Initialization:
    - pivot is set to the last element of the current sub-array.
    - i is initialized to low - 1 (this index keeps track of the "smaller element" boundary).
- Loop:
    - Loop through elements from low to high - 1.
     - If the current element arr[j] is less than or equal to the pivot, increment i and swap arr[i] with arr[j].
- Final Swap:
    - After the loop, place the pivot element in its correct position by swapping arr[i + 1] with arr[high].
    - Return the pivot index i + 1.

3. Quick Sort Function

```
void quick_sort(int *arr, int low, int high) {
    if (low < high) {
        int pivot_index = partition(arr, low, high);
        // Recursively sort elements before and after the partition
        quick_sort(arr, low, pivot_index - 1);
        quick_sort(arr, pivot_index + 1, high);
    }
}
```

- Base Case: If low is not less than high, return (array is already sorted or has one element).
- Recursive Case:
    - Call the partition function to get the pivot index.
    - Recursively call quick_sort on the sub-arrays before and after the pivot.

4. Main Function

```
int main() {
    int arr[] = {65, 70, 75, 80, 85, 60, 55, 50, 45};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Sort the array using quick sort
    quick_sort(arr, 0, size - 1);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
```

}

- Initialization:
    - An array arr is defined with some elements.
    - size is calculated using the size of the array divided by the size of an element.
- Printing the Original Array:
    - Loop through the array and print each element.
- Sorting the Array:
    - Call quick_sort with the array, 0 as the low index, and size - 1 as the high index.
- Printing the Sorted Array:
    - Loop through the sorted array and print each element.

Summary

- The program implements the Quick Sort algorithm using a divide-and-conquer approach.
- It recursively partitions the array around a pivot element and sorts the sub-arrays.
- The swap function is used to swap elements, and the partition function places the pivot in its correct position.
- The quick_sort function orchestrates the sorting by calling itself recursively.
- Finally, the main function initializes the array, sorts it, and prints the results before and after sorting.