Dynamic Array Sum in C

Problem Description

An array is a container object that holds a fixed number of values of a single type. To create an array in C, you can write:

```c
int arr[n];
```

This is a static array with memory allocated at compile time.

A dynamic array can be created using the `malloc` function, allocating memory on the heap at runtime:

```c
int *arr = (int*)malloc(n * sizeof(int));
```

Here, `arr` points to the base address of the dynamically allocated integer array of size `n`. After usage, free the memory using:

```c
free(arr);
```

Task

Create a dynamic integer array of size `n`, read `n` integers from standard input, compute the sum of all elements, print the sum, and free the allocated memory.

Solution Code

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n;
    scanf("%d", &n);

    // Allocate dynamic array
    int *arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        // Allocation failed
        return 1;
    }

    // Read elements
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Calculate sum
    int sum = 0;
    for (int i = 0; i < n; i++) {
```

```
        sum += arr[i];
    }

    // Print sum
    printf("%d\n", sum);

    // Free memory
    free(arr);

    return 0;
}
```

Listing 1: Dynamic Array Sum in C

Example

Input:

```
6
16 13 7 2 1 12
```

Output:

```
51
```

Notes and Extensions

- While summing elements during input is possible, storing them in an array provides experience with dynamic memory management.
- This technique can be extended to multidimensional dynamic arrays or arrays of structs.
- Be sure to always check if `malloc` returns `NULL` before using the pointer.
- Remember to free dynamically allocated memory to avoid memory leaks.

Similar Problems

Here are some problems where you can apply the same ideas of dynamic memory allocation and array manipulation:

- **Find the Maximum and Minimum** — Dynamically allocate an array, read values, and find the maximum and minimum element.
- **Reverse an Array** — Use dynamic arrays to read input and print the elements in reverse order.
- **Count Frequency of Elements** — Dynamically allocate an array and count how many times each number appears.
- **Dynamic String Array** — Create an array of strings dynamically, read multiple strings, and print them.
- **Matrix Multiplication** — Use dynamic 2D arrays allocated on the heap to perform matrix multiplication.
- **Remove Duplicates** — Dynamically manage arrays to remove duplicate integers from the input.

- **Find Average and Median** — Compute average and median of dynamically read values.
- **Dynamic Array Resize** — Learn to resize arrays dynamically using `realloc` as the input grows.

These problems will help strengthen your understanding of memory allocation, pointers, and array handling in C.

How This Problem Can Be Made Harder

To increase the difficulty and deepen your understanding, consider these extensions:

- **Multidimensional Dynamic Arrays:** Instead of a single-dimensional array, allocate and manipulate 2D or 3D arrays dynamically. This involves allocating arrays of pointers and managing nested loops.
- **Dynamic Array Resizing:** Implement a dynamic array that grows or shrinks at runtime using `realloc`. This simulates data structures like dynamic lists or vectors.
- **Structs and Arrays:** Use dynamic arrays of structs to store complex data (e.g., students with marks, names, IDs), and perform computations on their fields.
- **Input Validation and Error Handling:** Add robust checks to ensure input correctness and handle invalid or unexpected inputs gracefully.
- **Memory Management Challenges:** Practice detecting and fixing memory leaks or invalid accesses using tools like `valgrind`.
- **Parallel Summation:** For very large arrays, explore splitting the summation task across multiple threads to improve performance.
- **Algorithms on Dynamic Arrays:** Extend the problem to sorting, searching, or applying algorithms like prefix sums or sliding windows on dynamic arrays.
- **Use of Pointers to Pointers:** Manipulate arrays of pointers (e.g., array of strings), which requires deeper pointer understanding.