C Programming Short Notes
Part 11

Memory allocation

- Memory allocation in programming refers to the process of reserving a block of memory during the execution of a program.
- It allows programs to dynamically manage memory, enabling the creation of data structures like arrays, linked lists, and more.
- Memory allocation is crucial for managing data efficiently, especially when dealing with unknown or varying amounts of data.

Types of Memory Allocation:

1. Static Memory Allocation
2. Dynamic Memory Allocation

1. Static Memory Allocation:

- In static memory allocation, memory is allocated at compile time. The size of memory required must be known at compile time, and it remains fixed throughout the program's execution.
- Examples include global variables and local variables declared with the `static` keyword.
- Static memory allocation is simple and efficient but lacks flexibility.

2. Dynamic Memory Allocation:

- Dynamic memory allocation allows memory to be allocated and deallocated during runtime.
- It provides flexibility in managing memory, allowing the allocation of memory based on the program's needs.
- Dynamic memory allocation is commonly used when the size of data structures is unknown or needs to change dynamically.
- Memory allocated using malloc is allocated on the heap, which is a larger region of memory managed by the operating system. Heap memory needs to be explicitly allocated and deallocated using functions like malloc and free.

Types of Dynamic Memory Allocation:

1. Memory Allocation (malloc)
2. Contiguous Allocation (calloc)
3. Reallocation (realloc)
4. Free(free)

1. Memory Allocation (malloc):

- **Description:** `malloc` stands for "memory allocation". It dynamically allocates a block of memory of a specified size in bytes.

  Syntax:

  ```
  void *malloc(size_t size);
  ```

Usage:

- The `malloc` function takes one argument, which specifies the size in bytes of the memory block to be allocated.
- It returns a pointer to the beginning of the allocated memory block if successful, or `NULL` if the allocation fails.

  Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int*p,n,i;
   printf("Enter the value of n: ");
   scanf("%d",&n);
   p=(int*)malloc(n*sizeof(int));
   printf("Enter the value : ");
   for(i=0;i<n;i++)
   {
      scanf("%d",&p[i]);
   }
   for(i=0;i<n;i++)
   {
      printf("%d\t",p[i]);
   }
   free(p);
   p=NULL;
   printf("\n");
   for(i=0;i<n;i++){
      printf("%d\t",p[i]);
   }
}
```

Output:

```
Enter the value of n: 5
Enter the value : 1 2 3 4 5
1       2       3       4       5
```

2. Contiguous Allocation (calloc):

- **Description:** `calloc` stands for "contiguous allocation". It dynamically allocates memory for an array of elements and initializes all bits to zero.

  Syntax:

  ```
  void *calloc(size_t num_elements, size_t element_size);
  ```

  Usage:

  - The `calloc` function takes two arguments: the number of elements to allocate memory for (`num_elements`), and the size in bytes of each element (`element_size`).
  - It returns a pointer to the beginning of the allocated memory block if successful, or `NULL` if the allocation fails.

  Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    // Allocate memory for an array of 5 integers initialized to zero
    int *ptr = (int *)calloc(5, sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // Print the values (should be initialized to zero)
    printf("Values stored in allocated memory (calloc):\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
    // Free the allocated memory
    free(ptr);
    return 0;
}
```

Output

```
Values stored in allocated memory (calloc):
0 0 0 0 0
```

3. Reallocation (realloc):

- **Description:** `realloc` stands for "reallocation". It changes the size of the previously allocated memory block. If the new size is larger, it may move the block to a new location. If the new size is smaller, it may truncate the block.

  Syntax:

  ```
  void *realloc(void *ptr, size_t new_size);
  ```

Usage:

- The `realloc` function takes two arguments: a pointer to the previously allocated memory block (`ptr`), and the new size in bytes of the memory block (`new_size`).
- It returns a pointer to the beginning of the reallocated memory block if successful, or `NULL` if the reallocation fails. If the reallocation fails, the original memory block is left unchanged.

  Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    // Allocate memory for an array of 5 integers
    int *ptr = (int *)malloc(5 * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    // Assign values to the allocated memory
    for (int i = 0; i < 5; i++) {
        ptr[i] = i * 2;
    }
    // Print the values before reallocation
    printf("Values stored in allocated memory (before realloc):\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
    // Reallocate memory for an array of 10 integers
    ptr = (int *)realloc(ptr, 10 * sizeof(int));
    if (ptr == NULL) {
        printf("Memory reallocation failed\n");
        return 1;
    }
    // Assign new values to the reallocated memory
    for (int i = 5; i < 10; i++) {
        ptr[i] = i * 2;
```

```
    }
    // Print the values after reallocation
    printf("Values stored in allocated memory (after realloc):\n");
    for (int i = 0; i < 10; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
    // Free the reallocated memory
    free(ptr);

    return 0;
}
```

Output:

```
Values stored in allocated memory (before realloc):
0 2 4 6 8
Values stored in allocated memory (after realloc):
0 2 4 6 8 10 12 14 16 18
```

4. Free(free)

- **Description:** `free` is used to deallocate memory that was previously allocated using `malloc`, `calloc`, or `realloc`. It releases the memory back to the system for future use.
  Syntax:

  ```
  void free(void *ptr);
  ```

  Usage:
  - The `free` function takes one argument: a pointer to the previously allocated memory block (`ptr`).
  - After calling `free`, the pointer becomes invalid, and accessing it will result in undefined behavior.
    Example:

    ```
    free(ptr);
    ptr = NULL; // Optional: Set the pointer to NULL after freeing memory
                //to avoid dangling pointer
    ```

- These memory allocation functions are essential for dynamically managing memory in C programs, enabling efficient use of memory resources. Proper usage, including error handling and memory deallocation, is crucial to prevent memory leaks and undefined behavior.
- The `free` function is used to deallocate memory that was previously allocated using `malloc`, `calloc`, or `realloc`. It releases the memory back to the system for future use.
- In all the examples above, memory is freed at the end of the program using `free(ptr)`.