Dynamic Memory Allocation

1. **Malloc:**
   - Used to allocate space in memory during the execution of the program. I.e. Malloc allocates size bytes in the Dynamic Data segment.
   - Does not initialize the memory allocated during execution. It also carries garbage value.
   - Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
   - Because the function doesn't know what type of data we're planning on storing, it can't return a pointer to an ordinary type, like int or char. Instead, it returns a value of type void *. A void * value is a generic pointer, which is just a memory address.
   - Syntax: **void *malloc(int num);**
   - This function allocates an array of num bytes and leave them uninitialized.
   - The size parameter malloc expects is in bytes, which is also the size of one char.
   - For anything besides char, use sizeof to obtain the size of one element.
   - E.g.
     int *a = malloc (4 * sizeof (int));
   - Malloc is more efficient than calloc because it doesn't have to clear the memory block that it allocates.
2. **Calloc:**
   - Calloc is also like malloc, but calloc initializes the allocated memory to zero while malloc doesn't.
   - Furthermore, malloc allocates a single block of memory. Whereas, calloc allocates multiple blocks of memory.
   - Syntax: **void *calloc(int num, int size);**
   - Calloc also clears the memory that it allocates.
3. **Realloc:**
   - Modifies the allocated memory size by malloc and calloc to new size.
   - If enough space doesn't exist in memory of current block to extend, a new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.
   - Syntax: **void *realloc(void *address, int newsize);**
   - This function re-allocates memory extending it upto newsize.
   - **Note:** Address must point to a memory block obtained by a previous call of malloc, calloc or realloc.
4. **Free:**
   - Malloc and other memory allocation functions obtain memory blocks from a storage pool known as a **heap**. If we call these functions too often, or if we ask them for large block of memory, it can exhaust the heap which will cause the functions to return a null pointer.
   - Memory allocated via malloc is not released automatically. Other non-malloced memory is freed automatically when it goes out of scope.

- If we keep calling malloc without releasing any of the memory, memory will run out.
- Furthermore, sometimes, there may be a **memory leak**. A block of memory that is no longer accessible to a program is said to be **garbage**. A program that leaves behind garbage has a memory leak. Unlike some languages, C does not have a garbage collector. To release unneeded memory, we can use the free function.
- I.e. Frees the allocated memory by malloc, calloc and realloc and returns the memory to the system.
- This function releases a block of memory block specified by address.
- Syntax: **void free(*address);**
- E.g. Consider the code below.

  a = malloc(...);

  b = malloc(...);

  a = b; ← Here, a's old memory block is no longer accessible, but it has not been removed. This is an example of a memory leak.

  However, if we use the free function, we can reuse that block of memory later on.

  a = malloc(...);

  b = malloc(...);

  free(a);

  a = b;
- **Note:** The argument to free() must be a pointer that was previously returned by a memory allocation function. The argument may also be a null pointer, in which case free() has no effect. However, passing a pointer to any other object, like an array or a variable, to free() will cause an undefined behaviour.
- Be careful of dangling pointers when using free().
- A pointer pointing to a memory location that has been deleted or freed is called **dangling pointer**.
- These types of problems can result when memory is accessed after it has been freed.
- When you use free(), you deallocated the memory block that a pointer points to, but you didn't change the pointer. If we try to access or modify a deallocated memory block, then we will get errors.
- **Note:** When a block of memory is freed, all the pointers that pointed to it are now dangling pointers.
- The pointer does not point to a valid object. This is sometimes referred to as a **premature free**.

- The use of dangling pointers can result in a number of different types of problems, including:
    - Unpredictable behavior if the memory is accessed.
    - Segmentation faults when the memory is no longer accessible.
    - Potential security risks.

5. **Linked Lists:**
    - A **linked list** consists of a chain of structures, called **nodes**, with each node containing a pointer to the next node in the chain.
      Note: The last node in the linked list contains a null pointer.
    - A linked list is more flexible than an array because we can easily insert or delete nodes.
      I.e. Items can be added or removed from the middle of the list.
      Furthermore, there is no need to define an initial size.
    - However, accessing a specific node in the linked list will take longer if the node is near the end.
      I.e. There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
      Furthemore, dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
    - **Note:** **(*node).value** can be written as **node->value**. The -> operator functions is a combination of the * and . operators.