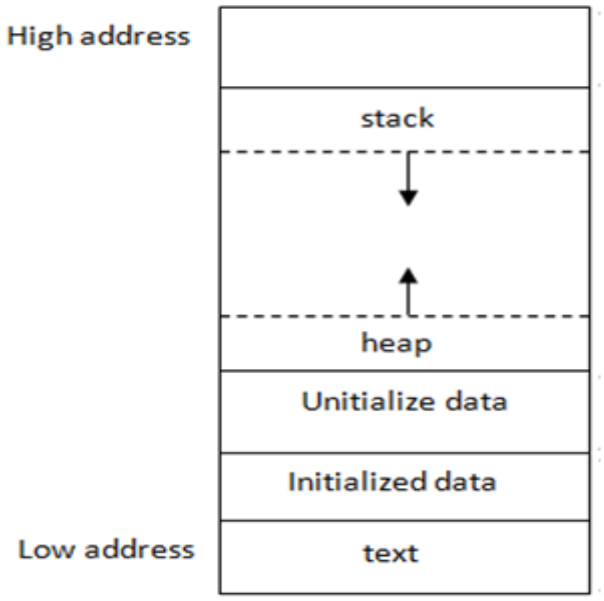
Heap Allocation

Stack Allocation: Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled. When a programs starts the main function is put into the stack and all the executable commands runs and when the main () completes its execution it removed from the stack all the local variables gets the memory inside this stack and main functions have another function they also pushed into the stack, so that after the execution of the function which functions have which function and have some extra parameters which describes about from where this function get called. Stack is something due to which recursion is possible so easily.

All the pointers get the memory from the heap;

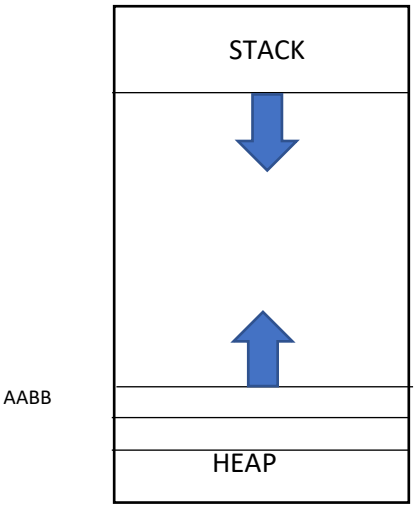
Heap Allocation: The memory is allocated during the execution of instructions written by programmers. Note that the name heap has nothing to do with the heap data structure. It is called heap because it is a pile of memory space available to programmers to allocated and de-allocate.Heap memory allocation isn't as safe as Stack memory allocation was because the data stored in this space is accessible or visible to all threads. If a programmer does not handle this memory well, a memory leak can happen in the program.The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions).The heap grows as we allocates the memory for something.



void * my_malloc(size_t size)

My_malloc create a memory block in the heap with the help of heap handle. The size of the block is same as the parameter passed to it in Bytes.

Suppose 4 is passed to it, the 4 Byte of block is created and the address is return by this function. At the prephase time we didn't know how much data block we required so we passed n*(sizeof(data_type)), which reserves the memory and later this memory can we access by this pointer which is returned by this function. Let the pointer to memory address return is AABB(hypotitical), So we can access anytime until it didn't get free.



The operating system like window have HeapAlloc(heap_handle, dwFlags, size)

Heap_handle: A handle to the heap from which the memory will be allocated. This handle is returned by the HeapAlloc() function. For the heap handle we use:

_get_heap_handle( ) return the handle of the current Heap.

dwFlags: The heap allocation options. Specifying any of these values will override the corresponding value specified when the heap was created with HeapAlloc().

Size: The number of bytes to be allocated.

If the function succeeds, the return value is a pointer to the allocated memory block.

If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.

void * my_calloc(size_t size)

My calloc used to get a memory block in the heap and assign Zero to all memory locations.

First we normally get a block memory of the size requested by the user by calling my_alloc() and after that we assign 0 to all memory location using memset.

memset(ptr,value,size) is used to fill a block of memory with a particular value.

ptr: Starting address of memory to be filled.

Value: Value to be filled.

Size: Number of bytes to be filled starting from ptr to be filled.

void * my_realloc(void *ptr, size_t size)

The function realloc is used to resize the memory block which is allocated by malloc or calloc before.

The Operating system provide the handle of the

Ptr – The pointer which is pointing the previously allocated memory block by malloc or calloc.s

Size – The new size of memory block.

void my_free (void* ptr)

It is used to free the allocated memory inside the heap. For this purpose OS have :

HeapFree(Heap_handle, dwFlags, ptr);

Heap_handle: A handle to the heap from which the memory will be allocated. This handle is returned by the HeapAlloc() function. For the heap handle we use:
_get_heap_handle( ) return the handle of the current Heap.
dwFlags: The heap allocation options. Specifying any of these values will override the corresponding value specified when the heap was created with HeapAlloc().
Ptr : Pointer to the current allocated memory.

size_t free_space_in_my_heap (void *ptr)

It is used to tell about the space used by our malloc, There is no fixed size a malloc can achieve it can get memory until its didn't get clash with Stack memory, because they grows in opposite direction. For this purpose we use
HeapSize(Heap_handle, dwFlags, ptr);
Heap_handle: A handle to the heap from which the memory will be allocated. This handle is returned by the HeapAlloc() function. For the heap handle we use:
_get_heap_handle( ) return the handle of the current Heap.
dwFlags: The heap allocation options. Specifying any of these values will override the corresponding value specified when the heap was created with HeapAlloc().
Ptr : Pointer to the current allocated memory.

Test Case:

```
int main()
{

	int* ptr;	// This pointer will hold the base address of the block created

	int n, i;
	n = 5; // Get the number of elements for the array
	printf("Enter number of elements: %d\n", n);

	ptr = (int*)my_malloc(n * sizeof(int));  // Dynamically allocate memory using
	my_malloc()


	// Check if the memory has been successfully allocated by my_malloc or not
	if (ptr == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
	else {

		// Memory has been successfully allocated
		printf("Memory successfully allocated using my_malloc.\n");

		// Get the elements of the array
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}

		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}
	}

	return 0;
}
```

Case2

```
int main()
{

	// This pointer will hold the
	// base address of the block created
	int *ptr, *ptr1;
	int n, i;

	// Get the number of elements for the array
	n = 5;
	printf("Enter number of elements: %d\n", n);

	// Dynamically allocate memory using my_malloc()
	ptr = (int*)my_malloc(n * sizeof(int));

	// Dynamically allocate memory using calloc()
	ptr1 = (int*)my_calloc(n, sizeof(int));

	// Check if the memory has been successfully
	// allocated by malloc or not
	if (ptr == NULL || ptr1 == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
	else {

		// Memory has been successfully allocated
		printf("Memory successfully allocated using malloc.\n");

		// Free the memory
		my_free(ptr);
		printf("Malloc Memory successfully freed.\n");

		// Memory has been successfully allocated
		printf("\nMemory successfully allocated using calloc.\n");

		// Free the memory
		my_free(ptr1);
		printf("Calloc Memory successfully freed.\n");
	}
}
```

Case3

```
int main()
{

	// This pointer will hold the
	// base address of the block created
	int* ptr;
	int n, i;

	// Get the number of elements for the array
	n = 5;
	printf("Enter number of elements: %d\n", n);

	// Dynamically allocate memory using my_calloc()
	ptr = (int*)my_calloc(n, sizeof(int));

	// Check if the memory has been successfully
	// allocated by malloc or not
	if (ptr == NULL) {
		printf("Memory not allocated.\n");
		exit(0);
	}
	else {

		// Memory has been successfully allocated
		printf("Memory successfully allocated using my_calloc.\n");

		// Get the elements of the array
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}

		// Print the elements of the array
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}

		// Get the new size for the array
		n = 10;
		printf("\n\nEnter the new size of the array: %d\n", n);

		// Dynamically re-allocate memory using realloc()
		ptr = my_realloc(ptr, n * sizeof(int));

		// Memory has been successfully allocated
```

```
            printf("Memory successfully re-allocated using my_realloc.\n");

            // Get the new elements of the array
            for (i = 5; i < n; ++i) {
                ptr[i] = i + 1;
            }

            // Print the elements of the array
            printf("The elements of the array are: ");
            for (i = 0; i < n; ++i) {
                printf("%d, ", ptr[i]);
            }

            my_free(ptr);
        }

        return 0;
}
```