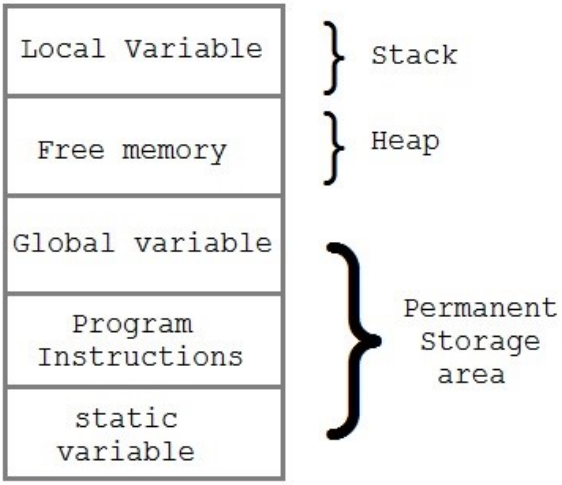
Memory Allocation

Memory Allocation Process in C



The program instructions and the global variables are stored in a region known as permanent storage area.

- ➔ The local variables are stored in another area called stack.
- ➔ The memory space between these two areas is available for dynamic allocation during execution of the program.
- ➔ This free region is called the heap.
- ➔ The size of the heap keeps changing.

Memory allocation is of two types

1. Static memory allocation
2. Dynamic memory allocation

Static Memory allocation

- Done at compile time.
- Global variables: variables declared "ahead of time," such as fixed arrays.
- Lifetime = entire runtime of program
- Advantage: efficient execution time.

Disadvantages:

- ➔ If we declare more static data space than we need, we waste space.
- ➔ If we declare less static space than we need, we are out of luck.

Dynamic memory allocation

* Done at run time.
* Data structures can grow and shrink to fit changing data requirements.
    - ➔ We can allocate (create) additional storage whenever we need them.
    - ➔ We can de-allocate (free/delete) dynamic space whenever we are done with them.
* ***Advantage:*** we can always have exactly the amount of space required - no more, no less.
* *For example,* with references to connect them, we can use dynamic data structures to create a chain of data structures called a linked list.

✓ Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running. To use this, the programmer must use either standard data types or already must have declared any derived types.

For implementing dynamic memory we have four management functions.

* **malloc** – Allocates requested number of bytes and returns a pointer to the first byte of the allocated space.
* **calloc** – Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
* **free** Frees previously allocated space.
* **realloc** – Modifies the size of previously allocated space.

malloc() function

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.
- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

➔ The malloc function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a void pointer to first byte of allocated memory. The allocated memory is not initialized. You should therefore assume that it will contain garbage and initialization is required.

The prototype is as:

void *malloc (size_t size);

In general it itself calculated for known datatypes. It is as for integer.

```
pint = (int *)malloc(sizeof (int));
```

malloc returns address of first byte in memory space, NULL for not successful.

```
int * p;
p = (int *) malloc (sizeof (int));
* p = 5;
```

calloc() function

➔ It allocates a contiguous block of memory large enough to contain an array of elements of specified size. So it requires two parameters as number of elements to be allocated and for size of each element. It returns pointer to first element of allocated array.

Its prototype is:

void *calloc (size_t element_count,size_t element_size);

For example,

An int array of 10 elements can be allocated as follows.

```
int * array = (int *) calloc (10, sizeof (int));
```

Note that this function can also malloc, written as follows.

```
int * array = (int *) malloc (sizeof (int) * 10);
```

realloc() function

➔ This is inefficient and shouid be used advisebly. Ealloc changes the size of block by deleting or extending the memory at end of block. If memory is not available it gives complete new block.

The prototype is:

void *realloc (void *ptr, size_t newSize);

➔ realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.

➔ If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

```
float *nums;
int I;
nums = (float *) calloc(5, sizeof(float));
        /* nums is an array of 5 floating point values */
        for (I = 0; I < 5; I++)
          nums[I] = 2.0 * I;
                /* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */
                nums = (float *) realloc(nums,10 * sizeof(float));
```

free() function

The name itself indicates that it clears the memory given/ allotted.

The prototype is

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

```
float *nums;
int N;
printf("Read how many numbers:");
scanf("%d",&N);
nums = (float *) calloc(N, sizeof(float));
/* use array nums */
/* when done with nums: */
free(nums);
/* would be an error to say it again - free(nums) */
```

Note:

When function problem called, space for array of size N allocated, when function ends, variable nums goes away, but the space nums points at (the array of size N) does not (allocated on the heap) - furthermore, we have no way to figure out where it is) which is the Problem called *memory leakage* for which we need to free the memory