Dynamic memory allocation

Dynamic memory allocation is the allocation of memory storage for use in a computer program during the runtime of that program. Static memory means we reserve a certain amount of memory by default inside our program to use for variables and such. Once we reserve this memory, no other program can use it, even if we are not using it at the time.

Why DMA?

If we have two programs that reserve 1000 bytes of memory each, but neither program is running, then we have 2000 bytes of memory that is being completely wasted. Suppose we only have 3000 bytes of memory, but we already have our two programs that take 1000 bytes each. Now we want to load a program that needs 1500 bytes of memory. It's impossible because we only have 3000 bytes of memory and 2000 bytes are already reserved. The answer for economic allocation of memory is dynamic memory allocation.

Static memory allocation: The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variable have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation. Memory is assigned during compilation time.

Dynamic memory allocation: It uses functions such as malloc( ) or calloc( ) to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run time.

Static memory is pre-allocated during process mapping into the main memory/ stack.

Dynamic memory is allocated on heap space of the process map. Visibility throughout the process is with the help of a pointer reference

To accomplish DMA in C the malloc function is used and the new keyword is used for C++. Both of them perform an allocation of a contiguous block of memory, malloc taking the size as parameter:

Malloc()

The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

```
ptr=(cast-type*)malloc(byte-size);
```

ptr is a pointer of type cast-type the malloc returns a pointer (of cast type) to an area of memory with size byte-size.

x=(int*)malloc(100*sizeof(int));

On successful execution of this statement a memory equivalent to 100 times the area of int bytes is reserved and the address of the first byte of memory allocated is assigned to the pointer x of type int. The malloc function will return NULL if requested memory couldnt be allocated or size argument is equal 0.

```
int* pi;
int size = 5;
pi = (int*)malloc(size * sizeof(int));
```

sizeof(int) return size of integer (4 bytes) and multiply with size which equals 5 so pi pointer now points to the first byte of 5 * 4 = 20 bytes memory block.

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL)
{
   printf("out of memory\n");
   //exit or return
}
```

Calloc()

Calloc is another memory allocation function that is normally used to request multiple blocks of storage each of the same size and then sets all bytes to zero. The general form of calloc is:

```
void *calloc(size_t num, size_t size);
```

ptr=(cast-type*) calloc(n,elem-size);

The above statement allocates contiguous space for n blocks each size of elements size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space a null pointer is returned.

Free()

Compile time storage of a variable is allocated and released by the system in accordance with its storage class. With the dynamic runtime allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited. When we no longer need the data we stored in a block of memory and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.

free(ptr); ptr is a pointer that has been created by using malloc or calloc.

Realloc()

If you need to store a series of items you read from the user, and if the only way to know how many there are is to read them until the user types some ``end'' signal, you'll have no way of knowing, as you begin reading and storing the first few, how many you'll have seen by the time you do see that ``end'' marker.

The memory allocated by using calloc or malloc might be insufficient or excess sometimes in both the situations we can change the memory size already allocated with the help of the function realloc. This process is called reallocation of memory. The general statement of reallocation of memory is :

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

ptr=realloc(ptr,newsize);

This function allocates new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the memory block. The allocated new block may be or may not be at the same region.

```
/*Example program for reallocation: eg is in C as all the 4 functns are in C rather than
C++*/
#include< stdio.h >
#include< stdlib.h >
define NULL 0
main()
{
char *buffer;
/*Allocating memory*/
if((buffer=(char *) malloc(10))==NULL)
{
printf("Malloc failed\n");
exit(1);
}
printf("Buffer of size %d created \n,_msize(buffer));
strcpy(buffer,"Bangalore");
printf("\nBuffer contains:%s\n",buffer);
/*Reallocation*/
if((buffer=(char *)realloc(buffer,15))==NULL)
{
printf("Reallocation failed\n");
exit(1);
}
printf("\nBuffer size modified.\n");
printf("\nBuffer still contains: %s\n",buffer);
strcpy(buffer,"Mysore");
printf("\nBuffer now contains:%s\n",buffer);
/*freeing memory*/
free(buffer);
}
```

Comparision between DMA in C and C++

```
int *data =new int;
```

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

This memory block can be used whenever needed during the program execution or until explicitly deallocating it, unlike the automatic memory which is available only inside the function or block of instructions where it was declared. Allowing a program to allocate dynamic storage every time it needs more until the program stops can cause it eventually to run out of available space. To prevent this behavior C++ provides the delete operator with the job of recycling a segment of memory allocated with new:

```
Delete (data);
```

The C function for memory deallocation is free() and has the same behavior as delete, it frees the space pointed by data for future use:

```
Free(data);
```

If the allocated memory is not freed when it's no longer necessary it will result in a memory leak.

Ref:

http://www.exforsys.com/tutorials/c-language/dynamic-memory-allocation-in-c.html

http://www.eskimo.com/~scs/cclass/notes/sx11.html

http://cprogramminglanguage.net/c-dynamic-memory-allocation.aspx