Lecture 12
Static and Dynamic Memory Allocations. Functions malloc() and free( ). Dynamic Memory Allocation for 1-D Arrays

There are 2 kinds of memory allocation in C/C++ languages: a) **static memory allocation** and b) **dynamic memory allocation.**

Static memory allocation refers to the process of allocating memory for the named variables at compile time before the corresponding program is executed. Static memory allocation is performed by compiler conform variable declaration statements for global and local variables. Global variables, constants (specified by keyword **const)** and static memory class local variables (specified by keyword **static)** having lifetime of a whole program are allocated in **fixed memory**, but automatic memory class local variables (specified by keyword **auto** or by default) having lifetime of a function call are allocated on **the stack** (is a part of computer memory where data is added and removed in a Last-In-First-Out (LIFO) manner). Memory allocated statically can not be reallocated or freed (deallocated) by program itself (by programmer).

Dynamic memory allocation is the allocation memory for the unnamed dynamic variables at runtime, during program execution. In C language dynamic memory allocation is performed by program itself (by programmer) using special functions of standard library header file **stdlib.h.** Dynamic memory is allocated from a large part of memory area called **the heap** (also called the **free store**). Dynamic memory is allocated, accessed, managed, reallocated and freed at runtime by using pointer variables just allocated statically before at compile time.

Functions malloc( ) and free( ).

The function **malloc( )** is the basic function used to allocate memory on the heap (on the free store) in C language. Its prototype is:

void*   malloc(int size);

where **void*** is the type of function which represents void pointer type of returning value and **size** is the parameter variable name which represents the size of the block of memory needed in bytes.

If the allocation is performed successfully, function **malloc( )** returns the beginning address (of void pointer type) of the block of memory allocated. Note that because **malloc( )** function returns a void pointer it is recommended to cast the type of returning value of function **malloc( )** to a needed pointer type especially for compatibility of C language programs with C++ language programs.

If the allocation is not performed successfully function **malloc( )** returns the NULL pointer value to indicate that no memory was allocated and usually in this case the program execution is terminated.

Memory allocated by **malloc( )** function will continue to exist until the program terminates or the memory explicitly deallocated by the programmer (that is, the block is said to be ‘freed’). This is achieved by use of the function **free( ).** Its prototype is:

void free (void* p);

where **p** is the void pointer parameter for the beginning address of the block of memory just allocated before by **malloc( )** function or other functions used for dynamic memory allocation.

After **free( )** function call, the memory pointed to by **p** is not more available for the program and what is why, it is strictly recommended to assign NULL pointer value to the pointer **p** immediately after **free( )** function call.

Static and Dynamic memory allocations for 1-D array.

The standard method for **static memory allocation of 1-D array**, for example, of one hundred integers, is the following declaration:

```
                int A[100];
```

This statement allocates the block of the memory of two hundred bytes which can be used for storing one hundred or less integers. If number of elements of actual array is more less than the declared number of elements, the using of static memory allocation for array is not very good and efficient choice because some part of the memory statically allocated is never used by program. What is way, in programming, for arrays is recommended and is often used dynamic memory allocation.

To allocate dynamically a similar 1-D array having number of elements introduced from keyboard the following code is recommended:

```
int* A, n;
printf("Enter number of elements of array: ");
scanf("%d", &n);
A = (int*) malloc(n* sizeof(int) ); // or  A = (int*) malloc( n*sizeof(*A) );
if (A = = NULL) // or shorter  if ( !A)
 {
  puts( "\n Memory was not allocated"); // or using here return statement
 }
```

After this code it is possible to use allocated memory having access to it by means of pointer **A** which can be used as name of dynamic array.

Related functions for dynamic memory allocation.

Function **malloc( )** returns a block of memory that is allocated for the programmer to use, but is not initialized. The memory is usually initialized by different ways – by using the **memset( )** function, or by one or more assignment statements, or by introducing data from keyboard or file.
An alternative way is to use the **calloc( )** function, which allocates memory dynamically and then initializes it. Its prototype is:

```
                void*  calloc( int n,  int sizel );
```

which allocates a region of memory large enough to hold **n** elements of **sizel** bytes each. The allocated region is initialized to zero.
It is often useful to be able to reallocate a block of memory. This can be done using the function **realloc( )** which returns a pointer to a memory region of the specified **size**, containing the same data as the old region pointed to by pointer **p** (truncated if the new size is less than the old size). If function **realloc( )** is unable to allocate the memory region in place, it allocates new storage, copies the required data, frees the old region, assigns the NULL to old pointer and returns the address of new memory region. If this allocation fails, **realloc( )** function maintains the original pointer **p** unchanged, and returns the NULL pointer value. The additional part of reallocated memory is not initialized. The function prototype is:

```
                void* realloc( void* p,  int size );
```

Lecturer: Mihail Kulev, associate professor