Dynamically Allocated Memory in C

All throughout the C course (COP 3223), all examples of variable declarations were statically allocated memory. The word "static" means "not changing" while the word "dynamic" means "changeable," roughly speaking.

In regards to memory, what this means is as follows:

(1) static – the memory requirements are known at compile-time. Namely, after a program compiles, we can perfectly predict how much memory will be needed and when for statically allocated variables. The input the program may receive on different executions of the code will NOT affect how much memory is allocated. One serious consequence of this is that any statically allocated variable can only have its memory reserved while the function within which it was declared is running. For example, if you declare an int x in function A, if function A has completed, no memory is reserved to store x anymore.

(2) dynamic – the memory requirements are NOT known (for sure) at compile-time. It may be the case that on different executions of the program, different amounts of memory are allocated; thus, the input may affect memory allocation.

If you want to allocate memory in one function, and have that memory available after the function is completed, you HAVE to allocate memory dynamically in that function!!!

Secondly, since dynamically allocated memory isn't "freed" automatically at the end of the function within which it's declared, this shifts the responsibility of freeing the memory to the user. This can be done with the free function.

malloc, calloc functions

Here are the formal descriptions of the two functions we will typically use to allocate memory dynamically:

```
// Allocates unused space for an object    //
whose size in bytes is specified by size //
and whose value is unspecified, and     //
returns a pointer to the beginning of the //
memory allocated. If the memory can't be //
found, NULL is returned.

void *malloc(size_t size);


// Allocates an array of size nelem with
// each element of size elsize, and returns
// a pointer to the beginning of the memory
// allocated. The space shall be
initialized // to all bits 0. If the memory
can't be
// found, NULL is returned.

void *calloc(size_t nelem, size_t elsize);
```

Although these specifications seem confusing, they basically say that you need to tell the function how many bytes to allocate (how you specify this to the two functions is different) and then, if the function successfully finds this memory, a pointer to the beginning of the block of memory is returned. If unsuccessful, NULL is returned.

Dynamically Allocated Arrays

Sometimes you won't know how big an array you will need for a program until run-time. In these cases, you can dynamically allocated space for an array using a pointer. Consider the following program that reads from a file of numbers. We will assume that the first integer in the file stores how many integers are in the rest of the file.

The program on the following page only reads in all the values into the dynamically allocated array and then print these values out in reverse order.

Note that actual parameter passed to the malloc function. We must specify the total number of bytes we need for the array. This number is the product of the number of array elements and the size (in bytes) of each array element.

It should be fairly easy to see how we can change the code below to utilize calloc instead of malloc. In this particular example, since there is no need to initialize the whole block of memory to 0, there's no obvious advantage to using calloc. But, when you want to initialize all the memory locations to 0, it makes sense to use calloc, since this function takes care of that task.