CS 288 2019S Section 102
Dynamic Memory Management

Dynamic memory management refers to the manual allocation and deallocation of memory using a group of functions in the C standard library.

The C programming language manages memory statically, automatically, or dynamically. For static and automatic objects, the size of the allocation must be compile-time constant. If the required size is not known until run-time, then using fixed-size objects is inadequate.

Automatic data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory management in which memory is explicitly allocated on an area of memory structured for this purpose and deallocated once it is no longer needed. The area of memory from which application programs may dynamically allocate memory is called the heap.

Overview of Functions

In C, the functions `malloc()`, `calloc()`, and `realloc()` are used to allocate memory on the heap. The program accesses this block of memory via a pointer that these functions return. When the memory is no longer needed, the pointer is passed to the function `free()` which deallocates the memory so that it can be used for other purposes. The C dynamic memory allocation and deallocation functions are declared in `<stdlib.h>` header file.

Function	Description
`malloc`	allocates a block of memory of the specified number of bytes
`calloc`	allocates a block of memory based on the specified number of objects, and the size in bytes of a single object
`realloc`	expands or contracts the specified block of memory when possible; allocates a new block otherwise
`free`	releases the specified block of memory back to the system

The allocation functions return a void pointer (`void *`), which indicates that it is a pointer to a region of unknown data type. This is the case because the functions allocate memory based on byte count but not on type. If the allocation functions fail to allocate the requested block of memory, a null pointer is returned.

Allocating Clean Memory

The functions `malloc()` and `realloc()` do not initialize the memory allocated, while `calloc()` guarantees that all bytes of the allocated memory block have been initialized to `0`. Uninitialized memory contains remnants of previously used and discarded data.

Usage Example

Creating an array of ten integers with automatic scope is straightforward in C:

```
int array[10];
```

However, the size of the array is fixed at compile time. If one wishes to allocate a similar array dynamically, the following code can be used:

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

This computes the number of bytes that ten integers occupy in memory, then requests that many bytes from `malloc()` and assigns the result to a pointer named `array`.

Because `malloc()` might not be able to service the request, it might return a null pointer and it is good programming practice to check for this:

```
int *array = malloc(10 * sizeof(int));
if (array == NULL)
    exit(1);
```

When the program no longer needs the dynamic array, it must eventually call `free()` to return the memory it occupies to the system:

```
free(array);
```

After allocation with `malloc()`, elements of the array are uninitialized variables. The function `calloc()`, on the other hand, returns an allocation that has already been cleared:

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

With `realloc()` we can resize the amount of memory a pointer points to. For example, if we have a pointer acting as an array of size `n` and we want to change it to an array of size `m`, we can use `realloc()`.

```
int *array = malloc(n * sizeof(int));
/* ... */
array = realloc(array, m * sizeof(int));
```

Note that `realloc()` must be assumed to have changed the base address of the block (i.e. it has failed to extend the size of the original block, and has allocated a new larger block elsewhere and copied the old contents into it). Therefore, any pointers to addresses within the original block must be assumed to be no longer valid.

Common Errors

Improper use of dynamic memory allocation is a prolific source of bugs. The most common errors are:

- **Unavailable Memory:** Memory allocation is not guaranteed to succeed. Using the returned value, without checking if the allocation is successful, results in a segmentation fault.
- **Memory Leaks:** A memory leak occurs when allocated memory is no longer referenced. Since its address is no longer known, deallocating the memory is no longer possible.
- **Dangling Pointers:** A dangling pointer is a pointer that points to a deallocated memory region. Using a pointer that points to a now freed memory region results in an unpredictable behavior.
- **Wild Pointers:** Wild pointers arise when a pointer is used prior to initialization to some known valid address.