CMSC 257 - Lab 6 - Dynamic Memory Allocation

- Create a new directory: “lab6”. All the files for this lab should be saved in this directory only.

In this lab, you will gain experience with dynamic memory allocation in C.

Objectives:

- Use malloc and calloc to allocate memory
- Use realloc to reallocate memory which was previously allocated using malloc or calloc.
- Deallocate an allocated memory using free
- Understand, detect, fix memory leaks and dangling pointers

Setup

Begin this lab with `dynamic.c file`. Copy the file dynamic.c from
/home/sonmeza/labs/lab6/dynamic.c

You should follow along with the assignment and complete the code as described.

Review

As discussed in the class we learned 3 different regions of memory to store date: the bss/data, stack and heap

1) Data

Data section is used to store various pieces of static data, such as global variables, fixed string and array constants, and local variables declared with the static keyword. Uninitialized variables are stored in the bss and initialized to 0, while statically initialized variables are stored in the data section. The size of these regions is determined at static link time and cannot be changed while the program is running.

2) Stack

The stack is the region of program memory used to store data local to a function, including nonstatic local variables, arrays, and program arguments. Each time a function is called, the program allocates space on the stack known as a stack frame to store these variables. When the function returns, the stack frame is popped off the stack, freeing the space for future use.

3) Heap

The heap is a region of memory that can be used to store large blocks of data. The program can request memory in this region using a special set of functions, described below, through a process known as dynamic allocation . Blocks allocated on the heap must be explicitly freed when they are no longer needed.

In this lab, you will use the heap and dynamic memory allocation.

Dynamic Memory Allocation in C

Up to this point in the course, your C programs have only used local and global variables (including arrays and structures) to store program information. Non-static local variables are stored within a given function's stack frame (or, in some cases, in registers). This is nice because the space used to store the variables is automatically created when the function is called, and discarded when the function returns. However, stack variables have several limitations:

● Memory used for stack variables is only valid over the lifetime of a function call. This means that pointers to local variables or arrays should not be returned from a function, because the variable referred to will be popped off the stack when the function returns, rendering the pointer invalid.

● The stack has a fixed maximum size, so storing too much data on the stack can cause a stack overflow. This can corrupt other parts of the program's memory or produce a segmentation fault. (The heap also has a maximum size, but it is much larger.)

● Prior to the C99 standard, the size of a local variable was required to be known at compile time so the compiler could know how much space to allocate on the stack. In particular, this meant that arrays had to have a fixed size. If you did not know in advance how much memory your program would require, this could present an obstacle. With C99, array size may be determined at runtime, but arrays on the stack still cannot be resized after they are created.

The solution to the above problems lies in the **malloc()** , **calloc()** , **realloc()** , and **free()** functions, defined in **<stdlib.h>** . This group of functions provides a way in which arbitrary-size blocks of memory may be requested by the program at runtime. Dynamically allocated memory is stored in the heap, not the stack, so it will persist until it is explicitly deallocated.

Malloc vs. Global Data

One small thing to note before the details of malloc are discussed is how this differs from global data. Global data is great when it is known what information is needed to be shared by the program. If there is to be a shared data structure, it should be made into a global variable. However, in cases in which the size or even existence of the data can change throughout the program, dynamically allocated memory should be used.

Requesting a New Block of Memory

Function headers for malloc and calloc:

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
Sample use of malloc and calloc:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
```

The **malloc()** and **calloc()** functions are used to request new blocks of memory on the heap. **malloc()** allocates a contiguous block of memory whose length in bytes is specified by the size parameter, and returns a pointer to the start of the block. **calloc()** takes two arguments, **nmemb** and **size**, and allocates space for an array of **nmemb** elements, each of which has size **size**. The function fills the requested memory with 0s before returning it. This is sometimes convenient, but is also slower. Both functions will return a null pointer on failure.
Often, **malloc()** and **calloc()** are used in conjunction with the **sizeof** operator, which yields the number of bytes required for a given data type. For instance, to allocate an array of n integers on the heap, you could write:

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

Note that in either case, the returned pointer is of type **void *** , so it has to be cast to the desired type.

Resizing an Existing Block

```
Function headers for realloc:

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

Sample use:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
```

To change the size of an existing block of memory, you use the **realloc()** function. **realloc()** resizes the given block of memory to the specified size and returns a pointer to the start of the resized block (which may or may not be the same as the old pointer). The contents of the memory block are preserved up to the minimum of the old and new sizes. If the new size is larger, the value of the newly allocated portion is indeterminate. If memory allocation fails, a null pointer is returned and the block specified by ptr is unaffected.

Deallocating a Block

```
Function declaration for free:

void free(void *ptr);
```

sample use:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

The **free()** function will deallocate the entire block beginning at **ptr**, returning it to the heap for further memory allocation. If the given pointer is null, no action will be performed. The behavior of **free()** is undefined if the pointer does not point to the start of a dynamically-allocated block that has not yet been freed. Note that you do not have to specify the size of the block to be freed, as dynamically allocated blocks have metadata that includes this information.

Things to Avoid

Dynamic memory management gives you great power. These functions give you the ability to allocate memory in a function that can be used far beyond the lifetime of the function call. They enable you to create large data structures that change size as necessary according to runtime conditions. But with great power comes great responsibility. Dynamic memory allocation requires careful discipline to avoid bugs and other problems.

1) Memory Leaks: A memory leak is an issue that occurs when all pointers to a block of dynamically-allocated memory are lost before the block of memory is freed. Memory allocated on the heap persists beyond function calls and the computer has no way of knowing when a block of memory is not needed any more.

   In Java, this is handled by garbage collection , but there are no such niceties in C. The programmer 3 must explicitly release a block of memory back into the heap using the free() command when it is done being used.

2) Misuse of free() : Passing free() an invalid pointer, such as a pointer to an already-freed block of memory or a location other than the start of a dynamically allocated block, will cause unpredictable behavior. If you're lucky, you will immediately receive an error and the program will terminate. In the worst case, you may experience data corruption, segmentation faulting, or other problems.

3) Dangling Pointers: A third type of mistake is attempting to read from or write to a block of memory that has already been freed. This can cause errors similar to those caused by returning a pointer to a local variable.

Assignment:

Task 1

In the dynamic.c file, complete the code specified by a, b and c. Note that printArray() is a pre-written function for you to use.

Compile and run code, enter 10 for the size of the array.

Get a screenshot of the output and save as **Task1**

Hint: Array elements printed should be in the range of 100 through 109. You need to use malloc for this task

Task 2

Complete d, e and f.

Run code. For the first array again enter size of 10, for the second array enter the size of 20.

Compare the starting memory addresses of initial a1 with the starting address reallocated a1.

Get a screenshot of the output and save as **Task2**

Hint: Make sure the output is 100 through 109 then 210 through 219**.** You need to use realloc for this task and realloc keeps previously initialized data.

.

Task 3

Complete g. Free the allocated memory for a1.

Complete h, i. Run code. For the first array again enter size of 10, for the second array enter the size of 20. For the third one enter the size of 30.

Get a screenshot of the output for the memory allocated with calloc and save as **Task3**

Hint: calloc initializes all elements to 0.

Compare the starting memory addresses of a1 with the starting address of a2.

Now print a1 again. Is it going to print? What will be printed out, what is wrong with that and how you can prevent this from happening? By freeing a1 you deallocated memory allocated for a1, and now program is allowed to override it. You should assign NULL to the a1 after freeing it and later on, before using a1 you need to check if it is pointing to null.

Task 4

Now intentionally remove one of the free statements (maybe free(a1))
Compile the program using -g option(As you were doing for gdb). This will enable valgrind to see line numbers.

valgrind --leak-check=full ./<executable name>
Get a screenshot of the output and save as **Task4a**
Find the memory leak (it will show the amount and line number, where the memory is allocated)

Fix the memory leak and rerun valgrind (free all dynamically allocated memories). You should observe the memory leak is fixed. You just need to free all pointers that are pointing to dynamically allocated memories. Get a screenshot of the output and save as **Task4b.** Be careful to not to free twice, which may corrupt the memory table.

Submisson

1. Submit screenshots as separate files in jpeg/png format and named like this:
    a. Task1
    b. Task2
    c. Task3
    d. Task4a
    e. Task4b