Memory Leaks

Objectives

1 Provide an overview of how the execution of some blocks of code can cause memory leaks

Labwork

1 Download the source code from:

Main.cpp https://drive.google.com/open?id=0B_ouNNuWgNZCU1NmeHVIWWxJZFE

Color.h https://drive.google.com/open?id=0B_ouNNuWgNZCX0ZBaGVWaTdNWnc

Color.cpp https://drive.google.com/open?id=0B_ouNNuWgNZCUVVuTzQxSEpZRzA

2 You will compile and execute this code on build.tamu.edu. As we will step through the code together, please read the following steps carefully, as you will be required to change the STEP_LEVEL value as we progress and answer questions in a copy of this document.

3 Ensure STEP_LEVEL = 0; subsequently compile and execute the code. Carefully observe the output of the code, realizing that a points to a newly allocated new color object on the heap and that b points to this same location. The output should be similar (addresses will differ) to that below:

```
STEP_LEVEL == 0
Color *a = new Color;
Color *b = a;
a         0x7fb2d9c04c90         80,0,0
b         0x7fb2d9c04c90         80,0,0
```

The output in blue represents the name of an identifier, with its value in the same column in green, and the value of the object to which it points in red

4 Increment step_level to 1, review the code on lines 28-31, the compile and run the resultant program. You should observe similar output:

	STEP_LEVEL == 0 Color *a = new Color; Color *b = a; a0x7fc45b50000080,0,0 b0x7fc45b50000080,0,0 STEP_LEVEL == 1 a = nullptr a0 b0x7fc45b50000080,0,0
	Despite the fact that the point a has been set to 0 (nullptr), b can still access the memory (note the addresses above). This is important as when we put things on the heap, we must maintain the means (knowing where it is on the heap) to also remove them.
5	Increment step_level to 1, review the code on lines 36-41, the compile and run the resultant program. You should observe similar output:
	STEP_LEVEL == 0 Color *a = new Color; Color *b = a; a0x7fe88a404c9080,0,0 b0x7fe88a404c9080,0,0 STEP_LEVEL == 1 a = nullptr a0 b0x7fe88a404c9080,0,0 STEP_LEVEL == 2 delete b b0x7fe88a404c9080,0,0
	According to C++ standards, you're not suppose to be able to dereference a pointer that has been deleted. If you're running Visual Studios, you most likely encounter an exception in this step (if so, comment out the appropriate cout). If you're on Mac OS X, compiling with clang, you've probably realized that b still points to the memory location that it did in the previous step. With that said, you should be compiling on build.tamu.edu as previously requested.

	You're may be confused as to why you're still able to access the contents of the deleted color object. Well, the reason is that you're not actually deleting the object; you've essentially released the block of memory that the object previous occupied for reallocation. You can consequently still access it through the pointer... but eventually it will be reclaimed and used for something else. Thus, it is always good practice to set your pointers to the nullptr after calling delete.
6	Increment step_level to 3, review lines 43-48, compile and run the resultant program. You're output should be similar to below:

	`STEP_LEVEL == 0`
	`Color *a = new Color;`
	`Color *b = a;`
	`a0x7ff2d3c04c9080,0,0`
	`b0x7ff2d3c04c9080,0,0`
	
	`STEP_LEVEL == 1`
	`a = nullptr`
	`a0`
	`b0x7ff2d3c04c9080,0,0`
	
	`STEP_LEVEL == 2`
	`delete b`
	`b0x7ff2d3c04c9080,0,0`
	
	`STEP_LEVEL == 3`
	`b = new Color (80, 0, 0);`
	`b0x7ff2d3c04c9080,0,0`

	To reinforce the notion that memory is not deleted, but is 'freed' only to be reclaimed, note that the new Color object occupies the location of the object deleted in the previous step. On Visual Studios this won't be the case, but again, we're just trying to illustrate the lifecycle of dynamic memory here.
7	Increment to STEP_LEVEL = 4. Compile and run the resultant program. In this step we had set the pointer to the new Color object (created in the previous step) equal to the nullptr.
	What do you think happens when we set the pointer to the new color object to 0 (null ptr) or another object without first deleting it? Why?
	The address to the memory allocated for the color object has been lost (and because we don't know where that object resides anymore we can't delete it -- memory leak)

8	Increment to STEP_LEVEL = 5. Analyze the code on lines 57-70, compile and run the code.
	After the final iteration of the for-loop (once we have exited its scope), how many color objects reside within allocated memory that can no longer be accessed?
	8
9	Increment to STEP_LEVEL = 6. Analyze the code on lines 72-76 and functions called from within it. Subsequently compile and run the program.
	The name of the function called from this step_level block is ok_function. When using this function, when might a memory leak occur?
	If the pointer to the color object returned by the function is not assigned to or used to initialize a pointer to a color object
	Why is returning a new Color(R, G, B) object okay here (focus attention back to caller)?
	The pointer to the color object returned by the function is used to initialize a pointer to a color object
10	Increment to STEP_LEVEL = 7. Analyze the code in this block and functions called from within it. Subsequently compile and run the program. The output of the program should resemble that below:

	`...`
	`STEP_LEVEL == 7`
	`bad_ 0x7fff547ce370 255,0,255`
	`bad_ret 0x7fff547ce370 1417470832,32767,1902420`

	In the output above, the values in the bad_ row pertains to the address and values of local color objected created on the stack within the bad_function; the bad_ret row to the pointer values assigned to the returned address from the bad_function. In this example, the stack allocated d (color in bad_function) is destroyed when the function returns.
	Despite pointing to the same object in memory, why might the value of that object in memory differ when it is accessed within the function versus outside of it?
	When the activation record for the function is removed from the stack memory in that address space will eventually be re-used by another function's activation record
11	Finally, increment to STEP_LEVEL = 8. Analyze the code in this block and functions called from within it. Subsequently compile and run the resultant program.
	How/why does the function bad_function2 leak memory?
	The function allocates a number of objects on the heap and then calls delete after the pointer to those objects has been set to nullptr; the objects are never "deleted from the heap"

	Submission
	Submit your completed copy of this document (with each question completed) to gradescope for grading.