**#0 – What are the three errors?**

Before opening the code, I compiled the initial program with flags for:

Debugging: -g

Optimization Level: -O1

Optimization is just a fancy way of getting the compiler to execute code quicker, anywhere between no optimizations (-O0) and the max (-O3). It allows tools like Valgrind to read errors and memory leaks better. The downside is that if you use too high an optimization level, it can obfuscate some of the error handling that Valgrind uses to check.

A screen shot of a computer program

AI-generated content may be incorrect.

It appears that, from the Warnings, mysteryAnimal was declared on line 32, but is uninitialized – causing the first error.

Next, from Valgrind’s leak check, I was unable to write to STDIN after the prompt showed. This was because the program tried to read where the pointer was looking, but failed – causing the second error.

A screenshot of a computer program

AI-generated content may be incorrect.

After correcting the pointer initialization, the program compiled without warnings, and the loop was able to execute – but it executed with a memory leak, causing the third error.

This third error was due to the creation of mysteryAnimal at the top of every while loop, only freeing the object after exiting the game. Because of this, if there happens to be more than one round played, only the latest object gets freed while the others remain referenced on the heap.

**#1 – When mysteryAnimal is declared, the next instruction prints out: “The animal is initialized to: …” It is never initialized to a variable. Why does this not crash?? What is it being initialized to??**

C++ allows us to create a pointer on the stack. Whatever memory is on the stack at the time of creation, it will point to somewhere on the heap. That memory on the heap can be whatever was left there previously, initializing the pointer to “garbage”.

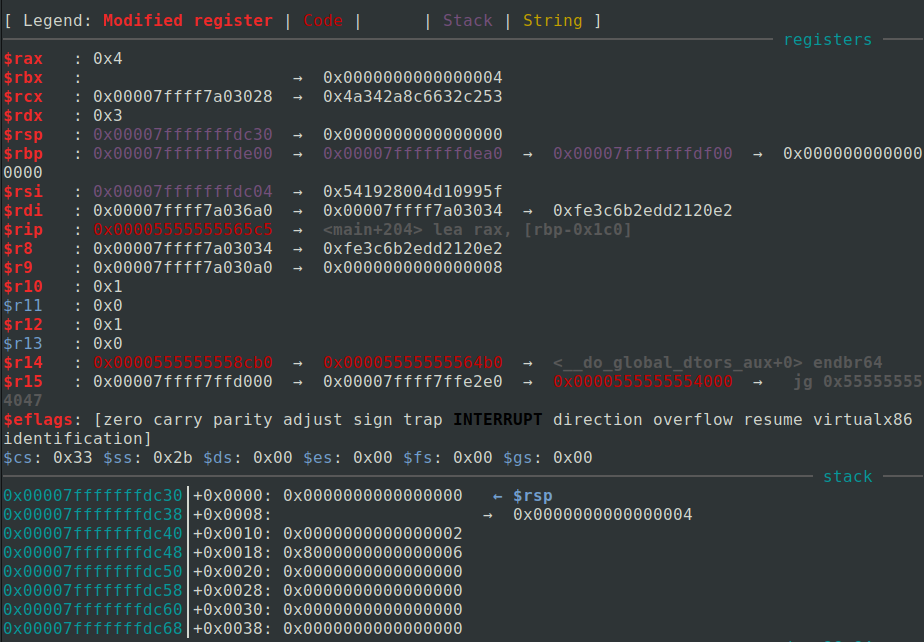
**#2 – Why is your program crashing on this line: “std::cout << "The animal should initally be nothing:…”?**

The pointer should be initialized to a value before being able to dereference it. Because this was not done, the crash results from an invalid read of memory on the heap at the moment of dereferencing.

**#3 – Run the program for 3 different guesses. On each guess, print out:**

* **- The address of the mysteryAnimal pointer**
* **- The address where the pointer is pointing**
* **- The value located at the address where the pointer is pointing.**
* **- Draw a picture diagraming each of these memory locations with their values.**
* **- Is the stack growing from the bottom (0x0000….) or the top (0xFFFFF…) of memory?**

I executed this program in GDB and entered a breakpoint after the creation of mysteryAnimal. Then, I dissected all the requested information before continuing and reaching the breakpoint again.



A screenshot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A computer screen shot of a program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

The first 2 screenshots show the stack and the requested info on starting the program. The next 6 screenshots show the same information after the input of one guess.

We can observe that the address of the mysteryAnimal pointer, the address where the pointer is pointing, and the value located at the address where the pointer is pointing do not change. Also, the stack pointer is also not changing (rsp).

RSP does not change because we remain inside the main function. All local variables are existent on the stack already, indicated by the unchanging address of mysteryAnimal (0x7ff…ffdc38).

The address the pointer is pointing to also does not change because we free that memory on the heap, and then immediately re-allocate it. Since that allocation is the same size, the memory manager under the hood takes the same block back.

**That was a long winded explanation to say that the stack grows from the top of memory**

**#4 - Relative to the stack and heap, where is the staticWelcomeMessage located? What is its address? Add it to the diagram.**



Below is the diagram for questions 3 and 4:

