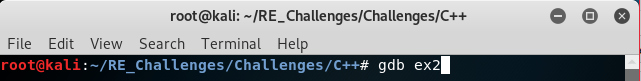
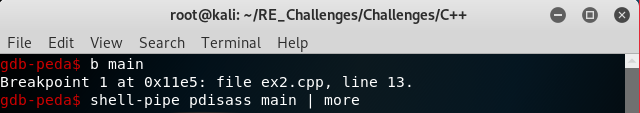
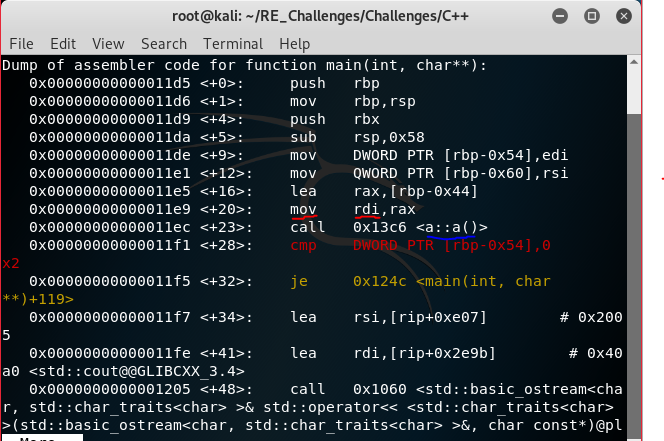
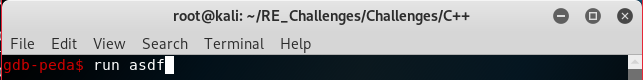
***Ex2***

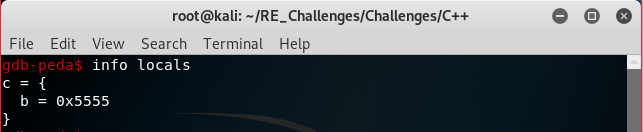
Now we get into the parts of C++ that differ the most from C outside of name mangling, object oriented programming. With object oriented programming we define abstract objects as classes and write code using instances of each defined class. Each class is instantiated into a usable object by calling a constructor method in the class definition. In simpler terms, a class describes an object and we make an object by calling a function defined in the class that constructs a usable object in the process memory. As you would imagine C++ programs can get pretty complex with the addition of Object Oriented code, not to mention C++ template programming or lambda functions. It should be stated that there is no one tool that solves all problems so you are encouraged to try these challenges with various tools but for the sake of this walk through we will use gdb with the peda extension script and shell-pipe.

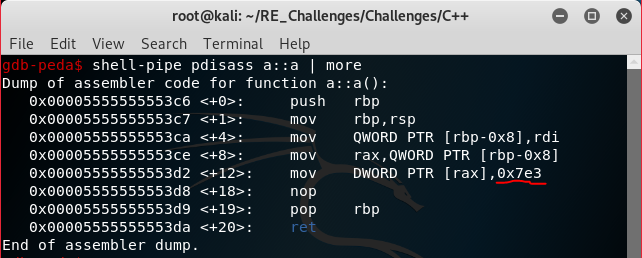
All extensions and configurations used in this explanation have been set-up correctly if you ran the set-up.sh script. However, there are many things that can be done to configure your gdb sessions and even many more extensions that can be added so reading through manuals and code repositories to see what all can be done to the tools at hand is very beneficial and encouraged.

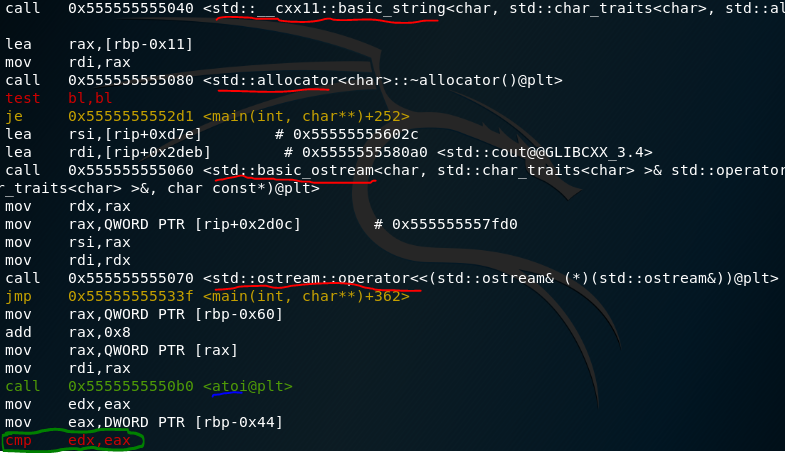
Like all challenges we want to start by examining the main function however, unlike in the C help files we will use some dynamic analysis so we want to put a “breakpoint” at the main function. This effectively causes the debuggee to hand over execution control to the debugger allowing us to inspect and change data at will. Although before we run the program and break at main we should at least look at the disassembly of main. We accomplish this with a simple “disass main”, gdb version, or “pdisass main”, peda version, but also take advantage of shell-pipe and pipe this output to the system “more” program to view the output in a scroll-able form(“less” doesn’t properly handle displaying the output).

If we start at the very beginning of the disassembly we see the prologue but soon see a call to a constructor method of a class “a”, underlined in blue. You can tell this is a constructor because the method name matches that of the class itself separated by the namespace resolution symbol “::”. How might this work in creating an object for our class? Well if you know a bit about C it will make sense rather quick. Objects can and have been “simulated” in C code through structs before with methods implemented as function pointers so looking at C++ objects in memory you will see similarities. However, do be aware that the order of data member variables and methods is compiler specific. One last thing before we dive into the constructor method you should realize that by calling a method that method has to know which object invoked it so that it updates and/or returns the data of the correct object. This bit of information about which object is which is called the “this” pointer, and is just a pointer to the base address of the specific object in memory. Due to the compiler our “this” pointer is moved into the rdi register, underlined in red.

Since we have our breakpoint on main we can go ahead and start the process, even with input, so it runs then stops at main before any code we currently care about is ran. To let the debugee continue execution as normal after this just use the “c”, continue, command and to execute a single instruction use the “s”, step, command. Once the debugee terminates you can rerun it with the same “run” command with no need to relaunch gdb.

Now that we are in main’s stack frame let’s look at the local variables of this frame. We can do this with “info locals” and by doing so we see a variables c, with further inspection you would discover the address of this local is our “this” pointer for the “a::a()” method!

Printing the disassembly of the constructor function we see that not a lot actually goes on during this call. Although one thing of interest is the hex value 0x7e3, decimal 2019, being stored in our object. Remember that the “this” pointer is our variable c in the “info locals” command but we never talked about the variable b. We are looking at an object in memory which as stated are similar to structs in memory. We can look at our object now as being a pointer to a struct with one data member who’s values is 0x7e3, 2019, after calling its constructor.

Now that we have an idea of the “a” class and one way it could be constructed into a class lets look back into the disassembly of main. At first we see a lot of allocator, string and output stream calls, underlined in red but soon see a C function “atoi”, ascii to integer, which is underlined in blue. This is immediately followed by a comparison. If we look closely we can see that our “this” pointer, originating at $rbp-0x44, is being moved into eax for the comparison. Well we know that the constructor sets the first, and only, value to 2019 which would be the value at the base address of the object’s pointer. That is a prime indicator that this might be our correct input so let’s give it a try.

