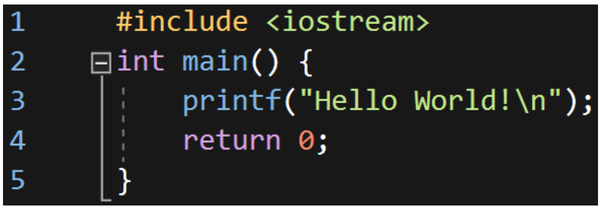
Hello World

I don't think there is a better way to start reversing than a classic "Hello World" program. There are many ways to make program print "Hello World!" in C++, I will be showing the two main ways. One with `printf()` and another with `std::cout`.

Source Code

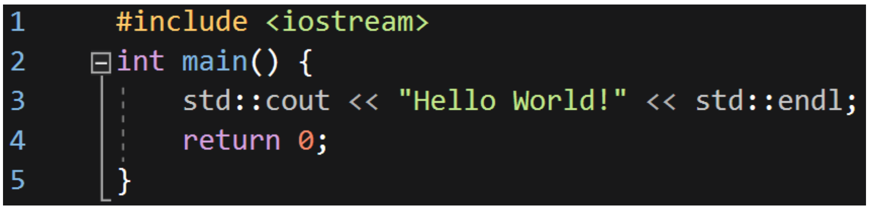
I usually won't be giving you the source code, but since we are just starting out here it is.

Using `printf()`:



```
#include <iostream>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Using `std::cout`:



```
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Reversing the `printf()` Version

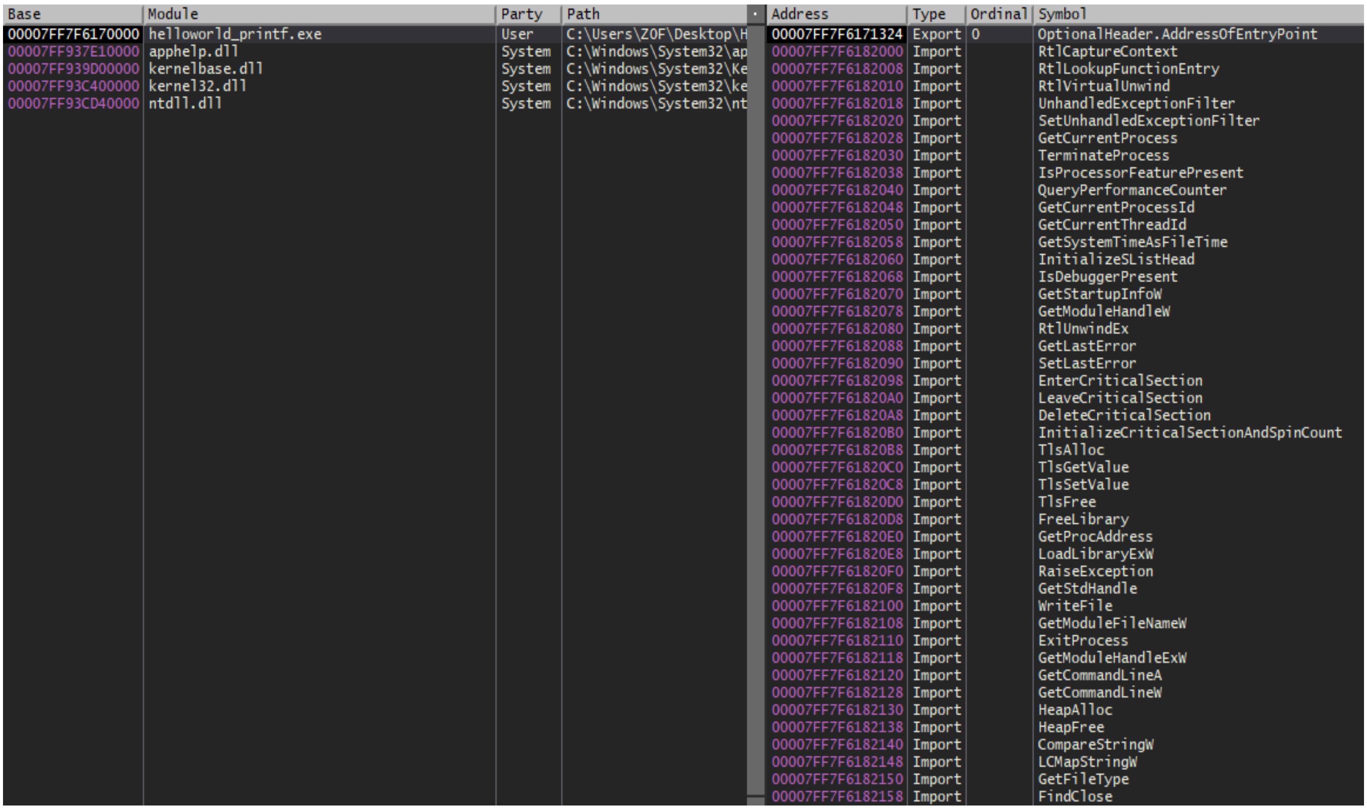
Let's view the compiled version of the programs in a disassembler. **We'll start with the `printf()` version.** x64dbg does a great job at presenting these programs so let's use that. To disassemble a program in x64dbg you can go to File > Open then select the program. You can also just drag and drop the program onto the GUI of x64dbg.

When you first throw the binary into x64dbg you may notice that we aren't in the `main` function. It turns out there are many things added to our program when it's compiled. Depending on how you have x64dbg setup it may start somewhere different. It might start at the "entry point" or near some security cookie thing.

- **Entry Point** - The entry point is the true beginning of the program. There are a few things that a binary will do before it starts running the code written by the developer.
- **Security Cookies** - The security cookie is used to make sure that the return address of a function hasn't been corrupted before following it. This is done to help mitigate buffer overflows. For us, we can ignore anything to do with security cookies.

Finding `main()`

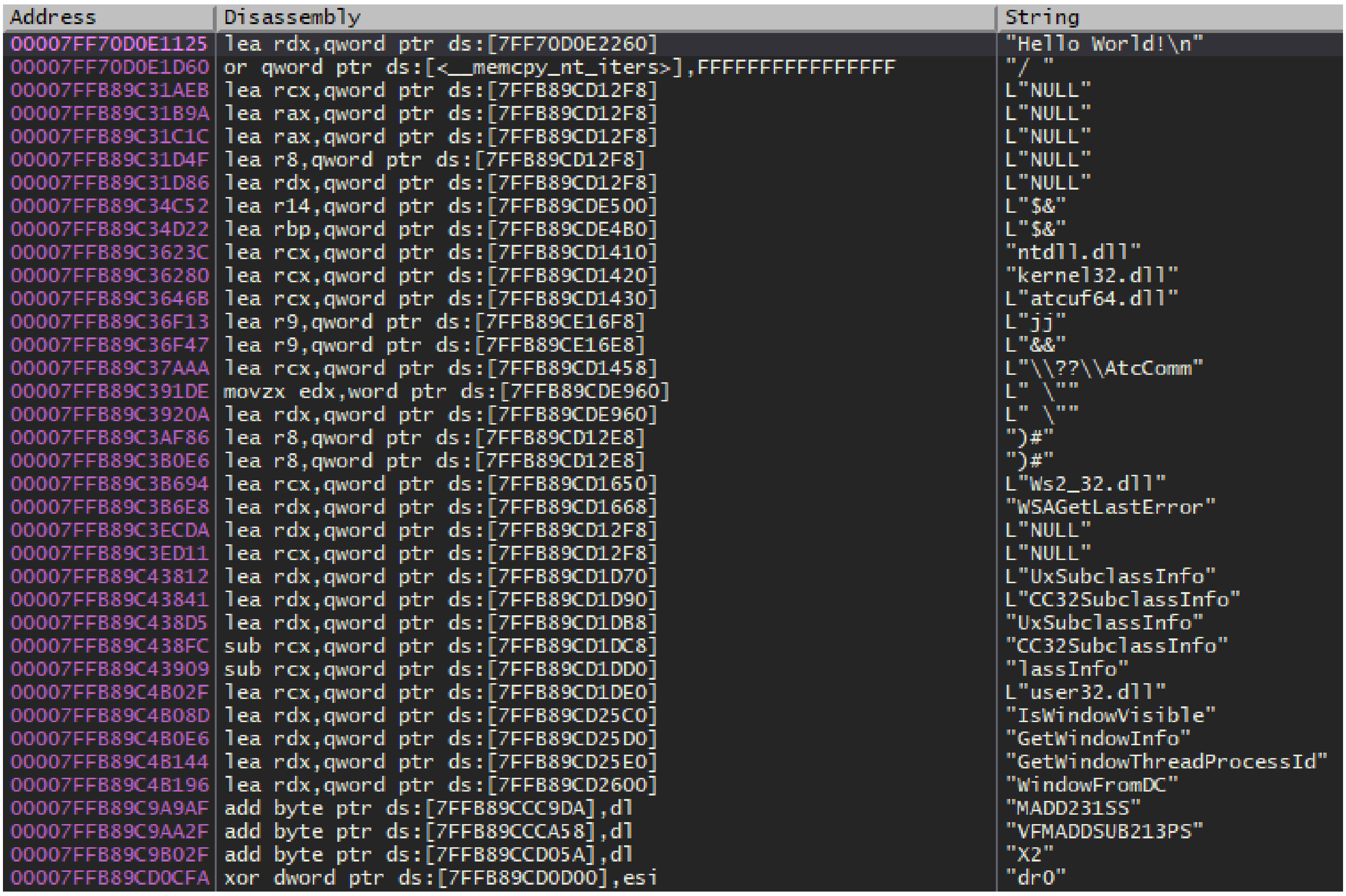
We want to find the `main()` function because that's where the code we want to reverse is at. So how do we get there? The easiest way is by going to the symbol. Go the "Symbols" tab and click on the executable.



Base	Module	Party	Path
00007FF7F6170000	helloworld_printf.exe	User	C:\Users\Z0F\Desktop\H
00007FF937E10000	apphelp.dll	System	C:\Windows\System32\ap
00007FF939D00000	kernelbase.dll	System	C:\Windows\System32\Ke
00007FF93C400000	kernel32.dll	System	C:\Windows\System32\ke
00007FF93CD40000	ntdll.dll	System	C:\Windows\System32\nt

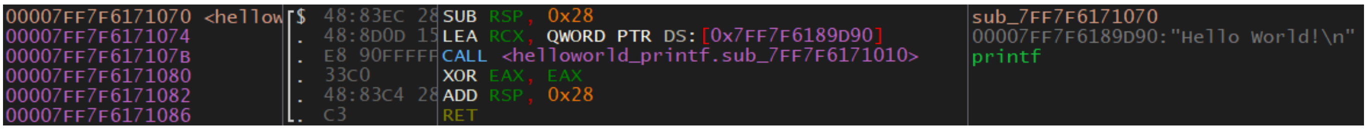
Address	Type	Ordinal	Symbol
00007FF7F6171324	Export	0	OptionalHeader.AddressOfEntryPoint
00007FF7F6182000	Import		RtlCaptureContext
00007FF7F6182008	Import		RtlLookupFunctionEntry
00007FF7F6182010	Import		RtlVirtualUnwind
00007FF7F6182018	Import		UnhandledExceptionFilter
00007FF7F6182020	Import		SetUnhandledExceptionFilter
00007FF7F6182028	Import		GetCurrentProcess
00007FF7F6182030	Import		TerminateProcess
00007FF7F6182038	Import		IsProcessorFeaturePresent
00007FF7F6182040	Import		QueryPerformanceCounter
00007FF7F6182048	Import		GetCurrentProcessId
00007FF7F6182050	Import		GetCurrentThreadId
00007FF7F6182058	Import		GetSystemTimeAsFileTime
00007FF7F6182060	Import		InitializeSListHead
00007FF7F6182068	Import		IsDebuggerPresent
00007FF7F6182070	Import		GetStartupInfoW
00007FF7F6182078	Import		GetModuleHandleW
00007FF7F6182080	Import		RtlUnwindEx
00007FF7F6182088	Import		GetLastError
00007FF7F6182090	Import		SetLastError
00007FF7F6182098	Import		EnterCriticalSection
00007FF7F61820A0	Import		LeaveCriticalSection
00007FF7F61820A8	Import		DeleteCriticalSection
00007FF7F61820B0	Import		InitializeCriticalSectionAndSpinCount
00007FF7F61820B8	Import		TlsAlloc
00007FF7F61820C0	Import		TlsGetValue
00007FF7F61820C8	Import		TlsSetValue
00007FF7F61820D0	Import		TlsFree
00007FF7F61820D8	Import		FreeLibrary
00007FF7F61820E0	Import		GetProcAddress
00007FF7F61820E8	Import		LoadLibraryExW
00007FF7F61820F0	Import		RaiseException
00007FF7F61820F8	Import		GetStdHandle
00007FF7F6182100	Import		WriteFile
00007FF7F6182108	Import		GetModuleFileNameW
00007FF7F6182110	Import		ExitProcess
00007FF7F6182118	Import		GetModuleHandleExW
00007FF7F6182120	Import		GetCommandLineA
00007FF7F6182128	Import		GetCommandLineW
00007FF7F6182130	Import		HeapAlloc
00007FF7F6182138	Import		HeapFree
00007FF7F6182140	Import		CompareStringW
00007FF7F6182148	Import		LCMapStringW
00007FF7F6182150	Import		GetFileType
00007FF7F6182158	Import		FindClose

There doesn't appear to be a `main()` function so we will have to find it some other way. One option we have is looking for the string that is going to be printed. We know that our program is going to print "Hello World!" so let's see if we can find where that string is used. We can do this in x64dbg by right-clicking in the disassembly window and going to Search for > All modules > String references. You should see something similar to what is shown in the image below.



Address	Disassembly	String
00007FF70D0E1125	lea rdx,qword ptr ds:[7FF70D0E2260]	"Hello World!\n"
00007FF70D0E1D60	or qword ptr ds:[<__memcpy_nt_iters>],FFFFFFFFFFFFFFFF	"/ "
00007FFB89C31AEB	lea rcx,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C31B9A	lea rax,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C31C1C	lea rax,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C31D4F	lea r8,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C31D86	lea rdx,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C34C52	lea r14,qword ptr ds:[7FFB89CDE500]	L"$&"
00007FFB89C34D22	lea rbp,qword ptr ds:[7FFB89CDE4B0]	L"$&"
00007FFB89C3623C	lea rcx,qword ptr ds:[7FFB89CD1410]	"ntdll.dll"
00007FFB89C36280	lea rcx,qword ptr ds:[7FFB89CD1420]	"kernel32.dll"
00007FFB89C3646B	lea rcx,qword ptr ds:[7FFB89CD1430]	L"atcuf64.dll"
00007FFB89C36F13	lea r9,qword ptr ds:[7FFB89CE16F8]	L"jj"
00007FFB89C36F47	lea r9,qword ptr ds:[7FFB89CE16E8]	L"&&"
00007FFB89C37AAA	lea rcx,qword ptr ds:[7FFB89CD1458]	L"\\??\\AtcComm"
00007FFB89C391DE	movzx edx,word ptr ds:[7FFB89CDE960]	L" \""
00007FFB89C3920A	lea rdx,qword ptr ds:[7FFB89CDE960]	L" \""
00007FFB89C3AF86	lea r8,qword ptr ds:[7FFB89CD12E8]	")#"
00007FFB89C3B0E6	lea r8,qword ptr ds:[7FFB89CD12E8]	")#"
00007FFB89C3B694	lea rcx,qword ptr ds:[7FFB89CD1650]	L"Ws2_32.dll"
00007FFB89C3B6E8	lea rdx,qword ptr ds:[7FFB89CD1668]	"WSAGetLastError"
00007FFB89C3ECDA	lea rdx,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C3ED11	lea rcx,qword ptr ds:[7FFB89CD12F8]	L"NULL"
00007FFB89C43812	lea rdx,qword ptr ds:[7FFB89CD1D70]	L"UxSubclassInfo"
00007FFB89C43841	lea rdx,qword ptr ds:[7FFB89CD1D90]	L"CC32SubclassInfo"
00007FFB89C438D5	lea rdx,qword ptr ds:[7FFB89CD1DB8]	"UxSubclassInfo"
00007FFB89C438FC	sub rcx,qword ptr ds:[7FFB89CD1DC8]	"CC32SubclassInfo"
00007FFB89C43909	sub rcx,qword ptr ds:[7FFB89CD1DD0]	"lassInfo"
00007FFB89C4B02F	lea rcx,qword ptr ds:[7FFB89CD1DE0]	L"user32.dll"
00007FFB89C4B08D	lea rdx,qword ptr ds:[7FFB89CD25C0]	"IsWindowVisible"
00007FFB89C4B0E6	lea rdx,qword ptr ds:[7FFB89CD25D0]	"GetWindowInfo"
00007FFB89C4B144	lea rdx,qword ptr ds:[7FFB89CD25E0]	"GetWindowThreadProcessId"
00007FFB89C4B196	lea rdx,qword ptr ds:[7FFB89CD2600]	"WindowFromDC"
00007FFB89C9A9AF	add byte ptr ds:[7FFB89CCC9DA],dl	"MADD231SS"
00007FFB89C9AA2F	add byte ptr ds:[7FFB89CCCA58],dl	"VFMADDSUB213PS"
00007FFB89C9B02F	add byte ptr ds:[7FFB89CCD05A],dl	"X2"
00007FFB89CD0CFA	xor dword ptr ds:[7FFB89CD0D00],esi	"dr0"

As you can see there are quite a few strings. Most of these strings aren't important. Let's search for our "Hello World!" string using the search bar at the bottom. Go ahead and double click on the "Hello World!" string. If there are more than one just choose the first one. This will bring you into the function where the string is used.



```
00007FF7F6171070 <hellow[$  48:83EC 28 SUB RSP, 0x28                                     sub_7FF7F6171070
00007FF7F6171074         .  48:8D0D 15 LEA RCX, QWORD PTR DS:[0x7FF7F6189D90]            00007FF7F6189D90:"Hello World!\n"
00007FF7F617107B         .  E8 90FFFFF CALL <helloworld_printf.sub_7FF7F6171010>          printf
00007FF7F6171080         .  33C0       XOR EAX, EAX
00007FF7F6171082         .  48:83C4 28 ADD RSP, 0x28
00007FF7F6171086         L. C3         RET
```

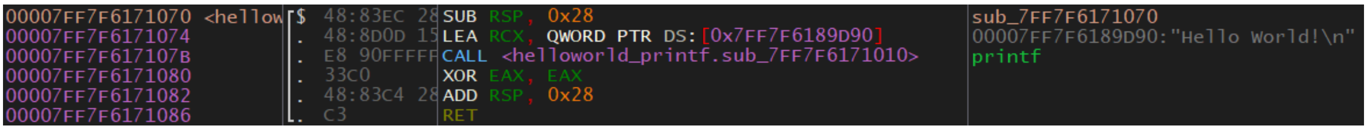
> Quick Tip: If you expand the address column you can see the function name. Functions are also marked with a dollar sign ("$") at the start of them.

The function this string is used in is `sub_7FF7F6171070`. This is probably different for you. You can see this next to the address (you might have to resize the column). The name is weird because the disassembler couldn't figure out its name so it just gave it a random name. So how do we know that this is the `main()` function? We don't. That's why reversing can be so difficult. Because we know what the code does and we have access to the source code we know that this is most likely the main function. In a realistic scenario, we wouldn't actually know for sure.

Just so we don't have to keep doing this let's change its label to `main` by right-clicking the first instruction in the function (the one with the `sub_7FF7F6171070` next to it) and choose Label > Label current address. Now we can get back to it by going to the functions tab and searching for `main`. You can get to the functions tab by going to View > Functions or by clicking the "fx" button on the toolbar above the "Memory Map" tab. The toolbar also has the debug buttons (blue arrows).

Analyzing Main

Let's figure out what `main()` is doing by looking at it in Assembly.



```
00007FF7F6171070 <hellow[$  48:83EC 28 SUB RSP, 0x28                                     sub_7FF7F6171070
00007FF7F6171074         .  48:8D0D 15 LEA RCX, QWORD PTR DS:[0x7FF7F6189D90]            00007FF7F6189D90:"Hello World!\n"
00007FF7F617107B         .  E8 90FFFFF CALL <helloworld_printf.sub_7FF7F6171010>          printf
00007FF7F6171080         .  33C0       XOR EAX, EAX
00007FF7F6171082         .  48:83C4 28 ADD RSP, 0x28
00007FF7F6171086         L. C3         RET
```

- `SUB RSP, 0x28` - On the first line we can see `RSP` is subtracted by 0x28. This is part of the function prologue. The function prologue sets up a function and you can usually ignore them.
- `LEA RCX, QWORD PTR DS:[0x7FF7F6189D90]` - `LEA` is short for **Load Effective Address** which will load an address into the destination operand. In this case, the address of our string is being put into RCX. Notice that the value is being put into RCX which is the first register used to pass parameters to functions. Sure enough, the next line is a function call. What is the `QWORD PTR` part? This is simply stating that the value is a QWORD (8 bytes) and it's a pointer. `DS` is short for data segment which can store data that doesn't fit in a register. You can ignore this and just know that it's some data from somewhere in memory. In our case, this address is the address of our string. If you want to view the string in memory you can do so by right-clicking the instruction and going to Follow in Dump > Constant: (Address).
- `CALL <helloworld_printf.sub_7FF7F6171010>` This will call the function at `sub_7FF7F6171010`. As we previously discovered, this function will receive one parameter which is the address of our string. Thanks to xAnalyzer a comment was added that this function is `printf()`. If xAnalyzer didn't figure this out then how would we know this was `printf()`? Typically I would use multiple tools and one of them would probably tell you. The next easiest way would be to debug the program and see what happens

after the call. We would see that the call has something to do with the text being printed out on the screen. Finally, we could reverse the function and figure it out that way.

Now we get to the function epilogue which is the end of the function. This portion will clean everything up before returning execution back to where it was called from.

- `XOR EAX, EAX` - If you XOR something with itself it sets it to zero. In this case, EAX is being zeroed out. I'm not sure why it's zeroing EAX instead of RAX, maybe RAX was never needed and only EAX was. I could dig into this but there really isn't any need to. We can determine from this that the function is returning 0. Based on this we can guess that the function type is an integer. If we look at our source code this is correct. Also, this further solidifies our guess that this is the `main()` function because `main()` is usually an integer type in C/C++.
- `ADD RSP, 0x28` - This is setting the stack to what is was before the function call. Notice that this is opposite to the `SUB RSP, 0x28` at the start.
- `RET` - Returns execution to the caller. This is the end of the function.

I want to touch on the fact that we passed the *address* of the string to `printf()`. In C/C++ strings are just an array of characters. The end of a string is denoted by a null character/byte (0x00). This is known as the string/null terminator. This is important because something like `printf()` will use this to know where the end of the string is so it doesn't print past it. In C/C++ a string is just a character array with the last character being a null terminator.

> 0x00 is referred to by many names. Null byte, terminator, string terminator, null terminator, and more.

I hope that wasn't too difficult to follow and understand. Finding the `main()` function can be annoying and it certainly doesn't make a good impression for people who are just starting to learn reversing, but it's what we have to do. Feel free to explore a little bit!

Reversing the `std::cout` Version

This is where things get interesting. `std::cout` is part of C++, whereas `printf()` is part of C. This means that the compiler has full control over `std::cout` but not as much control over `printf()`. Think about how you would make a program that just prints text to the screen *once* as efficient as possible. Think about this, what could you do instead of calling a function just once? The answer is to not call any function, just put the text to be printed inside the print function or put the print function with the text to be printed. Sounds really fun for us reverse engineers doesn't it?!

Here is an example:

```
int Add(int num1, int num2){
    return num1+num2;
}
int main(){
    int total = Add(2, 3);
}
```

How could we make this more efficient? There are two ways.
The example below is more efficient because there are no parameters to be passed.

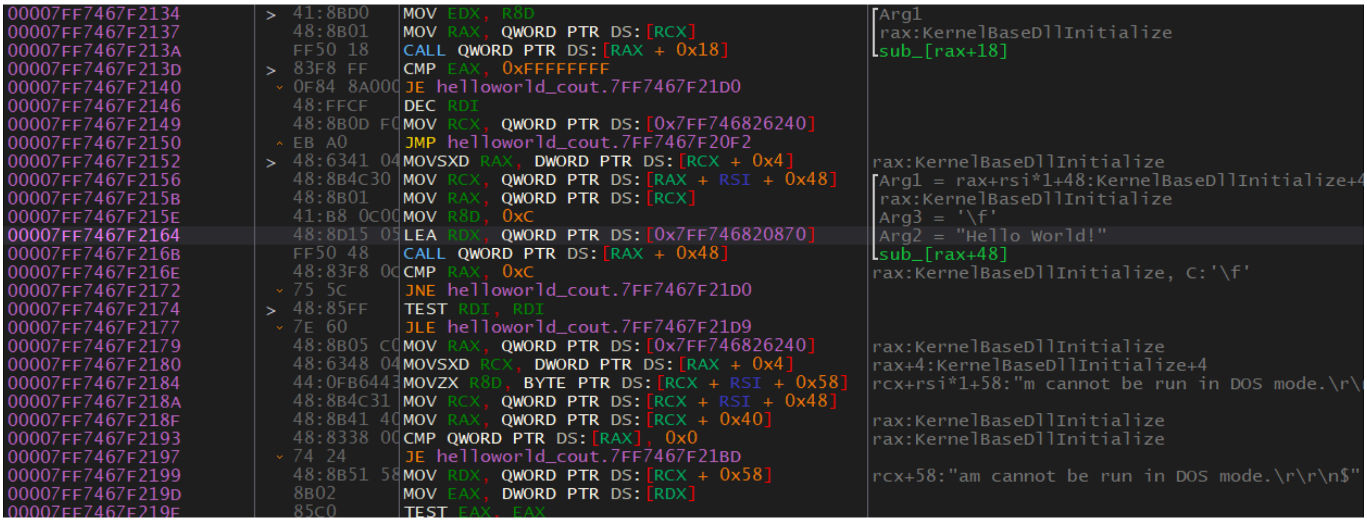
```
int Add(){
    return 2+3;
}
int main(){
    int total = Add();
}
```

The example below is more efficient because there is no function call at all. Instead, the code for the `Add()` function has essentially been copied and pasted into `main()`. It may not seem like it because I have such a small example but that's exactly what has happened in the example. This is called (depending on who you ask) **function inlining, inline expansion, or just in inlining.** I'll just call it inlining.

```
int main(){
    int total = 2 + 3;
}
```

Reversing Time

Open HelloWorld_Cout.exe in x64dbg and let's get started. First, we want to find the main function. Let's use the same technique we used previously which was searching for the string.



```
00007FF7467F2134  > 41:8BD0     MOV EDX, R8D                                  Arg1
00007FF7467F2137    48:8B01     MOV RAX, QWORD PTR DS:[RCX]                   rax:KernelBaseDllInitialize
00007FF7467F213A    FF50 18     CALL QWORD PTR DS:[RAX + 0x18]                sub_[rax+18]
00007FF7467F213D  > 83F8 FF     CMP EAX, 0xFFFFFFFF
00007FF7467F2140  ⌄ 0F84 8A000  JE helloworld_cout.7FF7467F21D0
00007FF7467F2146    48:FFCF     DEC RDI
00007FF7467F2149    48:8B0D F0  MOV RCX, QWORD PTR DS:[0x7FF746826240]
00007FF7467F2150  ^ EB A0       JMP helloworld_cout.7FF7467F20F2
00007FF7467F2152  > 48:6341 04  MOVSXD RAX, DWORD PTR DS:[RCX + 0x4]          rax:KernelBaseDllInitialize
00007FF7467F2156    48:8B4C30   MOV RCX, QWORD PTR DS:[RAX + RSI + 0x48]      Arg1 = rax+rsi*1+48:KernelBaseDllInitialize+4
00007FF7467F215B    48:8B01     MOV RAX, QWORD PTR DS:[RCX]                   rax:KernelBaseDllInitialize
00007FF7467F215E    41:B8 0C00  MOV R8D, 0xC                                  Arg3 = '\f'
00007FF7467F2164    48:8D15 05  LEA RDX, QWORD PTR DS:[0x7FF746820870]        Arg2 = "Hello World!"
00007FF7467F216B    FF50 48     CALL QWORD PTR DS:[RAX + 0x48]                sub_[rax+48]
00007FF7467F216E    48:83F8 0C  CMP RAX, 0xC                                  rax:KernelBaseDllInitialize, C:'\f'
00007FF7467F2172  ⌄ 75 5C       JNE helloworld_cout.7FF7467F21D0
00007FF7467F2174  > 48:85FF     TEST RDI, RDI
00007FF7467F2177  ⌄ 7E 60       JLE helloworld_cout.7FF7467F21D9
00007FF7467F2179    48:8B05 C0  MOV RAX, QWORD PTR DS:[0x7FF746826240]        rax:KernelBaseDllInitialize
00007FF7467F2180    48:6348 04  MOVSXD RCX, DWORD PTR DS:[RAX + 0x4]          rax+4:KernelBaseDllInitialize+4
00007FF7467F2184    44:0FB6443  MOVZX R8D, BYTE PTR DS:[RCX + RSI + 0x58]     rcx+rsi*1+58:"m cannot be run in DOS mode.\r\
00007FF7467F218A    48:8B4C31   MOV RCX, QWORD PTR DS:[RCX + RSI + 0x48]
00007FF7467F218F    48:8B41 40  MOV RAX, QWORD PTR DS:[RCX + 0x40]            rax:KernelBaseDllInitialize
00007FF7467F2193    48:8338 00  CMP QWORD PTR DS:[RAX], 0x0                   rax:KernelBaseDllInitialize
00007FF7467F2197  ⌄ 74 24       JE helloworld_cout.7FF7467F21BD
00007FF7467F2199    48:8B51 58  MOV RDX, QWORD PTR DS:[RCX + 0x58]            rcx+58:"am cannot be run in DOS mode.\r\r\n$"
00007FF7467F219D    8B02        MOV EAX, DWORD PTR DS:[RDX]
00007FF7467F219F    85C0        TEST EAX, EAX
```

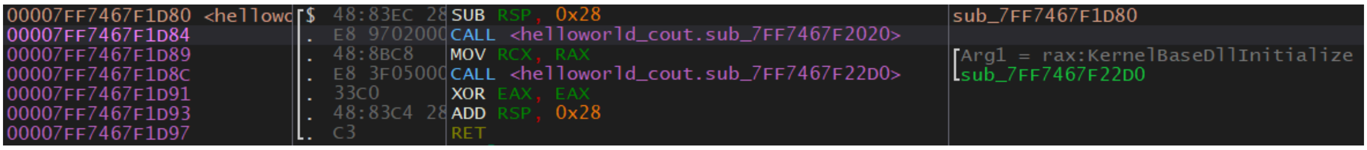
What is our string doing there? By looking at the code we can guess that the next function to be called is going to result in the string being printed. Also, note that the first parameter passed to the function is something to do with "KernelBaseDLLInitialize". This is an indicator that it's a function of a library, not a function implemented in HelloWorld_Cout.exe. If we are right, this could help us figure out what's going on.

Let's set a breakpoint on the LEA instruction that is dealing with our string. You can set a breakpoint by pressing F2 or by right-clicking and going to Breakpoint > Toggle. Then run the program until you hit that breakpoint. We can then step over the call by clicking the corresponding blue arrow at the top or by pressing F8. When we jump over this call we can see that the string appears in the console. It's extremely likely at this point that we are correct that this is `std::cout` or something to do with it. But how do we know that the function that was just called doesn't do something else? You can investigate the function that printed the text into the terminal. The function is accessed via an offset from RAX. This offset is 0x48. Because this function isn't be called with an address, we will need to know what RAX is and calculate the offset if we want to find the function. So we will have to run the function again until we reach that point. When execution has gotten to that line we can now see what RAX is. Now we go to RAX + 0x48 either by doing our own math or by pressing Enter and letting x64dbg figure it out then follow the call for us. From here you can reverse the function.

Another method to figure out what's going on is to write your own program that uses `std::cout` and see if things match. You could use the graph view to see the layout of the function in the program you're reversing and your own program to see if they match. You could, of course, look at the individual Assembly instructions but I think this is a little more tedious. This is a common technique that more experienced reverses use. Eventually, you will see patterns like this and be able to figure out pretty quickly what's going on.

Getting To Main

So was the string put into `std::cout` or was `std::cout` put into `main()`? Figuring this out is pretty easy. First, go back to the instruction that referenced the string. We want to find out what called this function. Go to the first instruction in this function, this is also the functions address/location. Right-click it and select Find references to > Selected Address(es). We can see that this function is called once, go to where it is called form. This is where it was called from:



```
00007FF7467F1D80 <hellowc[$  48:83EC 28 SUB RSP, 0x28                                  sub_7FF7467F1D80
00007FF7467F1D84         .   E8 9702000 CALL <helloworld_cout.sub_7FF7467F2020>
00007FF7467F1D89         .   48:8BC8    MOV RCX, RAX                                   [Arg1 = rax:KernelBaseDllInitialize
00007FF7467F1D8C         .   E8 3F05000 CALL <helloworld_cout.sub_7FF7467F22D0>        [sub_7FF7467F22D0
00007FF7467F1D91         .   33C0       XOR EAX, EAX
00007FF7467F1D93         .   48:83C4 28 ADD RSP, 0x28
00007FF7467F1D97        L.   C3         RET
```

This appears to be `main()` because of how small it is, but there are no guarantees. This is the function that causes our string to be printed so it's very likely that this is `main()` or something related to it.

So we now know that in this case, the compiler decided to put our string into `std::cout`.

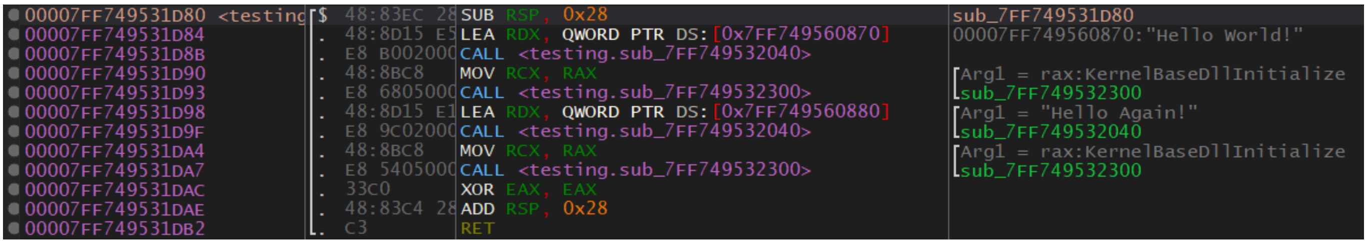
What's with the second call? I encourage you to find out what this call is on your own. Here is a quick hint: the two functions called are very similar.

▶ Answer (click to reveal):

The second call is `std::endl`. You will see that the function prints "\n" which is the newline character.

`std::cout` Used Twice

As I said, inlining is almost never used when a function is called more than once. So what does it look like if `std::cout` is used twice? Well, it look much more like the `printf()` version.



```
00007FF749531D80 <testing[$  48:83EC 28 SUB RSP, 0x28                                  sub_7FF749531D80
00007FF749531D84         .   48:8D15 E5 LEA RDX, QWORD PTR DS:[0x7FF749560870]         00007FF749560870:"Hello World!"
00007FF749531D8B         .   E8 B002000 CALL <testing.sub_7FF749532040>
00007FF749531D90         .   48:8BC8    MOV RCX, RAX                                   [Arg1 = rax:KernelBaseDllInitialize
00007FF749531D93         .   E8 6805000 CALL <testing.sub_7FF749532300>                [sub_7FF749532300
00007FF749531D98         .   48:8D15 E1 LEA RDX, QWORD PTR DS:[0x7FF749560880]         [Arg1 = "Hello Again!"
00007FF749531D9F         .   E8 9C02000 CALL <testing.sub_7FF749532040>                [sub_7FF749532040
00007FF749531DA4         .   48:8BC8    MOV RCX, RAX                                   [Arg1 = rax:KernelBaseDllInitialize
00007FF749531DA7         .   E8 5405000 CALL <testing.sub_7FF749532300>                [sub_7FF749532300
00007FF749531DAC         .   33C0       XOR EAX, EAX
00007FF749531DAE         .   48:83C4 28 ADD RSP, 0x28
00007FF749531DB2        L.   C3         RET
```

As you can see there is one call for `std::cout` and one call for `std::endl` for each one of the two strings being printed, totaling in four function calls.

More About Finding `main()`

There are a few techniques that you can use to find `main()`. In the future, I will talk about these methods. One thing I did want to mention is that `main()` should be called in the entry function. If you're looking for main, it's best to start from the bottom of the entry function and work your way up. That's it for now, I will add more about this topic in the future.

Here is my quick blog post on finding `main()`: https://www.z0fsec.tk/2019/06/research-finding-main-function.html

Final Notes

The `printf()` version should be fairly easy to understand. The `std::cout` is much more confusing. Hopefully, you get the general idea of what's going on though. From this point, I would highly recommend that you write your own programs and see what's going on. Visual Studio allows you to view the Assembled version of your program along with the source code which can be great for learning. Don't worry if this bit was confusing, the `std::cout` thing is really weird and counter-intuitive.