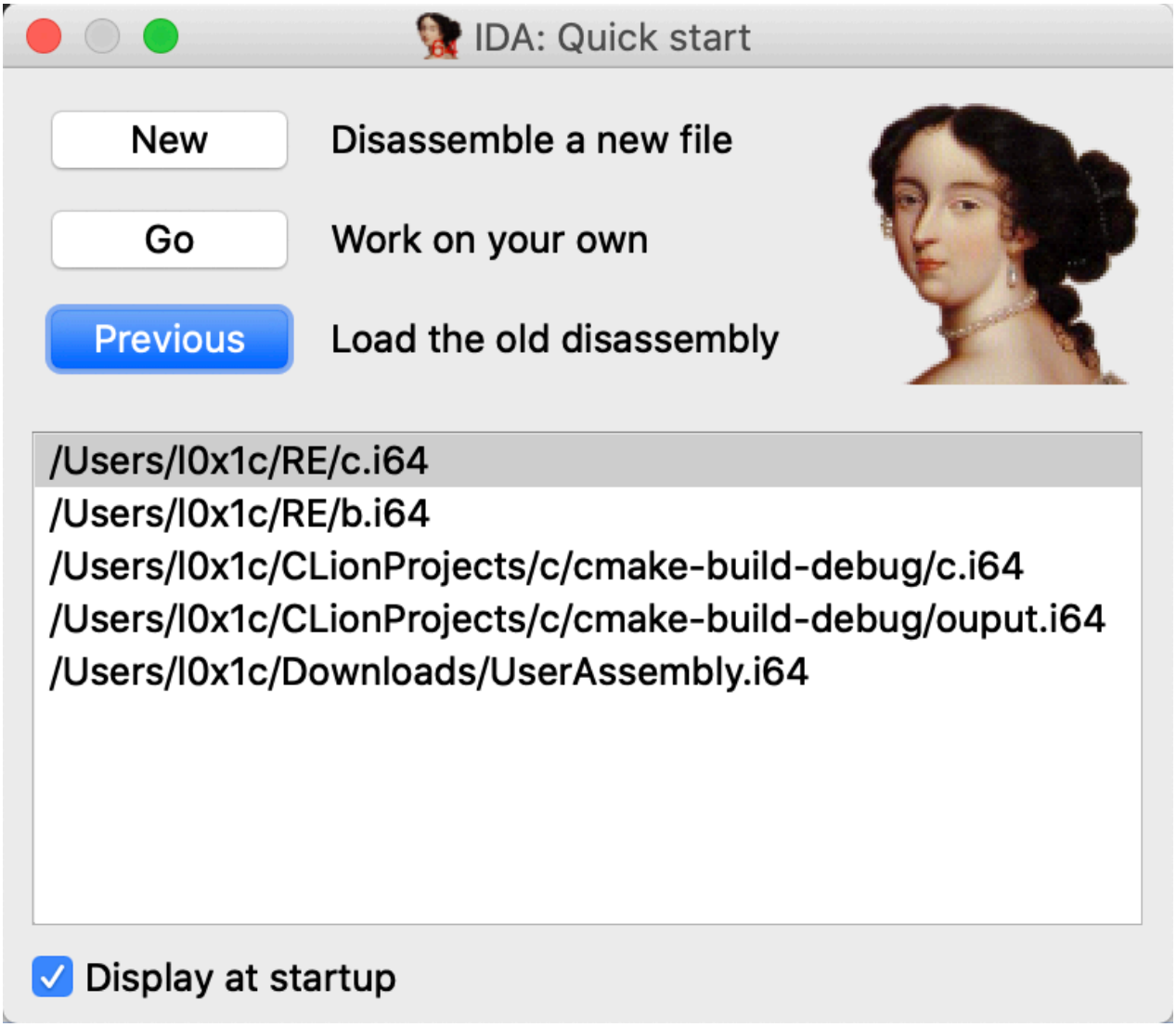
# IDA Pro

网址：https://www.youtube.com/watch?v=GEtpznMFsEs&list=PLKwUZp9HwWoDDBPvoapdbJ1rdofowT67z&index=3

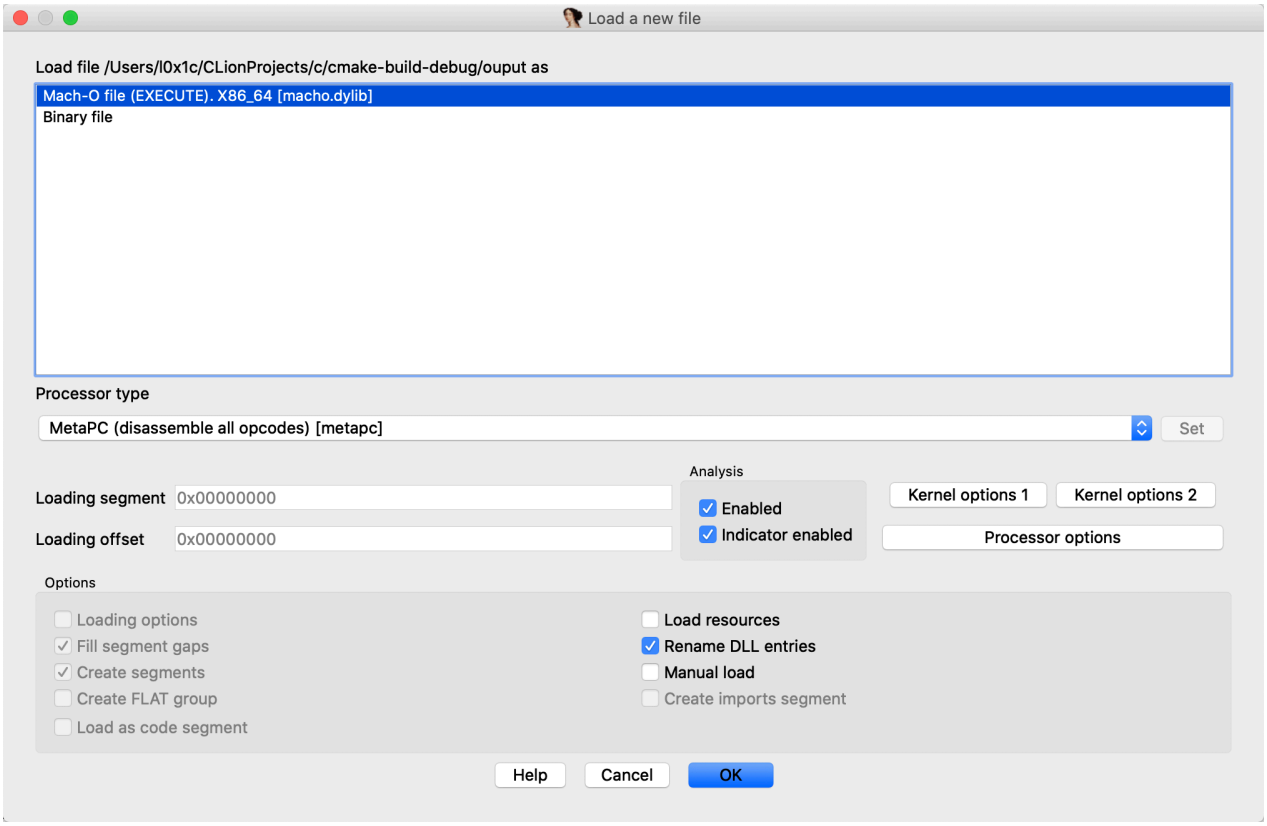
## Reverse Engineering Tutorial with IDA Pro – An Introduction

ida的一个基本操作，一般来说我们的ida在网上都会有破解的版本，同时也有正版（如果家里有矿的话），在网上down一下即可



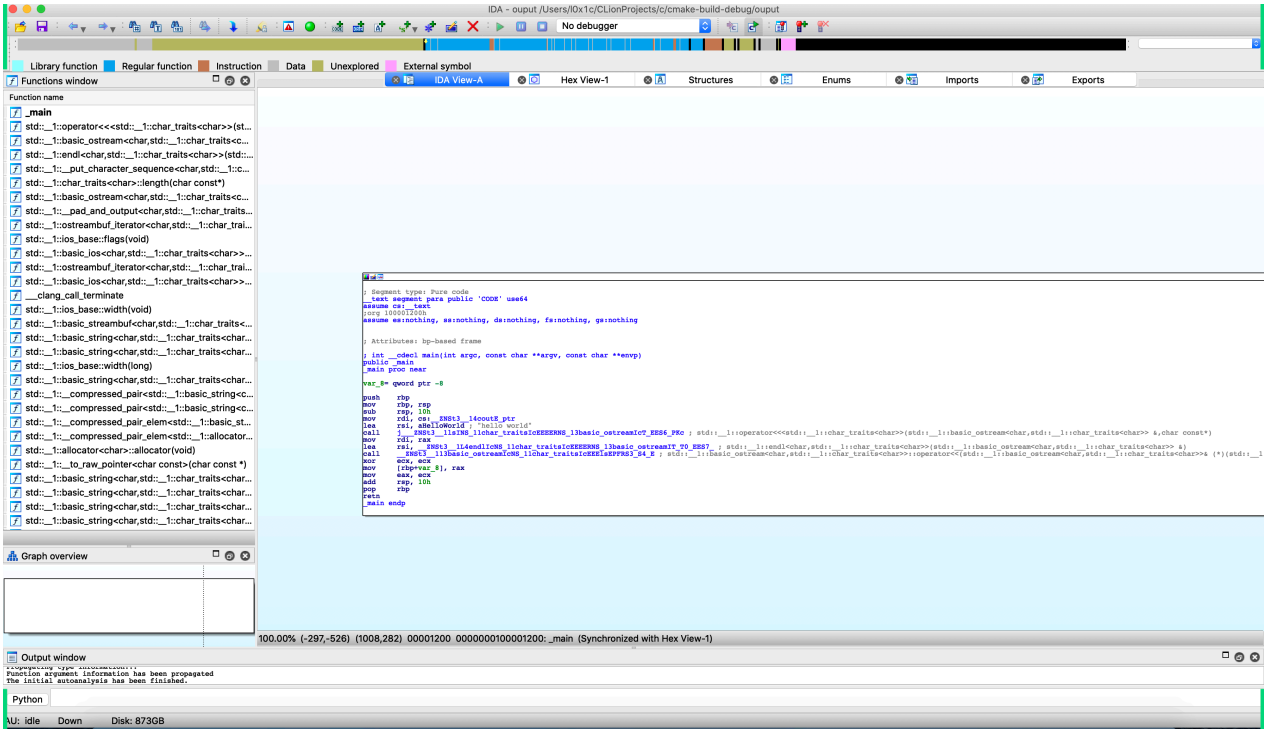

如果我们要打开一个新的文件的话，点击**new**即可，如果不喜欢看到这个欢迎的消息，直接点击我们的Display at startup取消即可，我们再启动的时候IDA会默认我们点击了**go**按钮，直接进入IDA的空白工作区，**previous**代表了我们打开最近用过的文件，IDA的注册History子项的值，最初的设定是10，也可以自己去设定

这里写了一个很简单的一个hello world我们用ida打开：

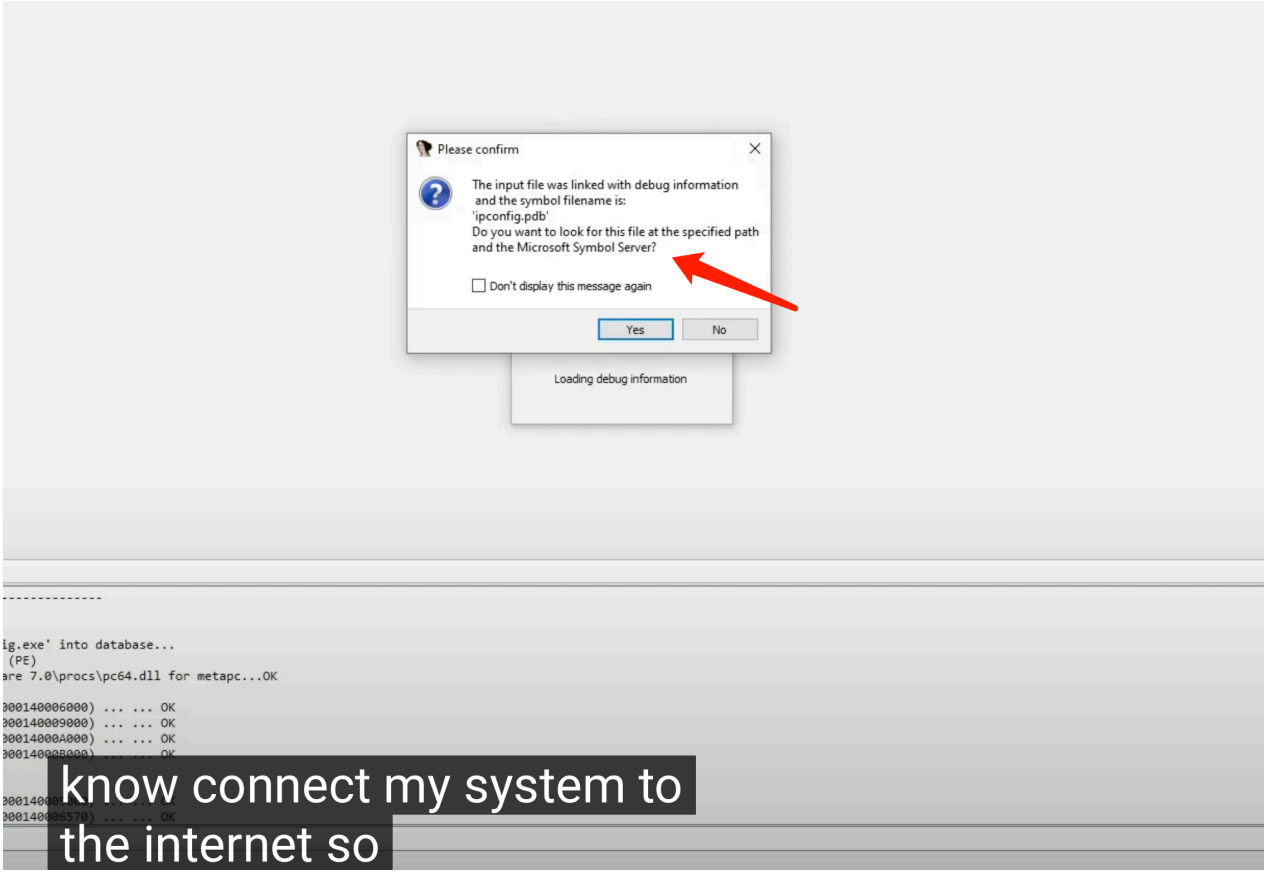



第一行应该知道是我们的加载器可以识别选定的文件，第二行是我们的Binary file，他会一直有的，这个是IDA加载无法识别文件的默认选项，提供了最低级的文件加载的方式

**processor**代表了IDA将根据从可执行文件的头中读取到的信息，选择合适的处理器，如果ida无法正确确定与所打开的文件关联的处理器类型，在文件加载之前需要我们手动选择一种处理器的类型




这是我们进入的页面，视频里有一个地方是联网进行获取的操作




官方给了相应的解释

此命令将加载PDB文件。
如果要反汇编的程序具有配套的PDB文件，则可以使用此命令将信息从PDB文件加载到数据库中。

请注意，PDB加载器使用系统DLL IMAGEHLP.DLL加载PDB文件。如果系统DLL是来自其他操作系统版本或出于某些其他原因，则它可能会拒绝加载PDB文件。

系统在系统目录（通常是\ WINNT \ Symbols之类）和当前目录中搜索PDB文件。

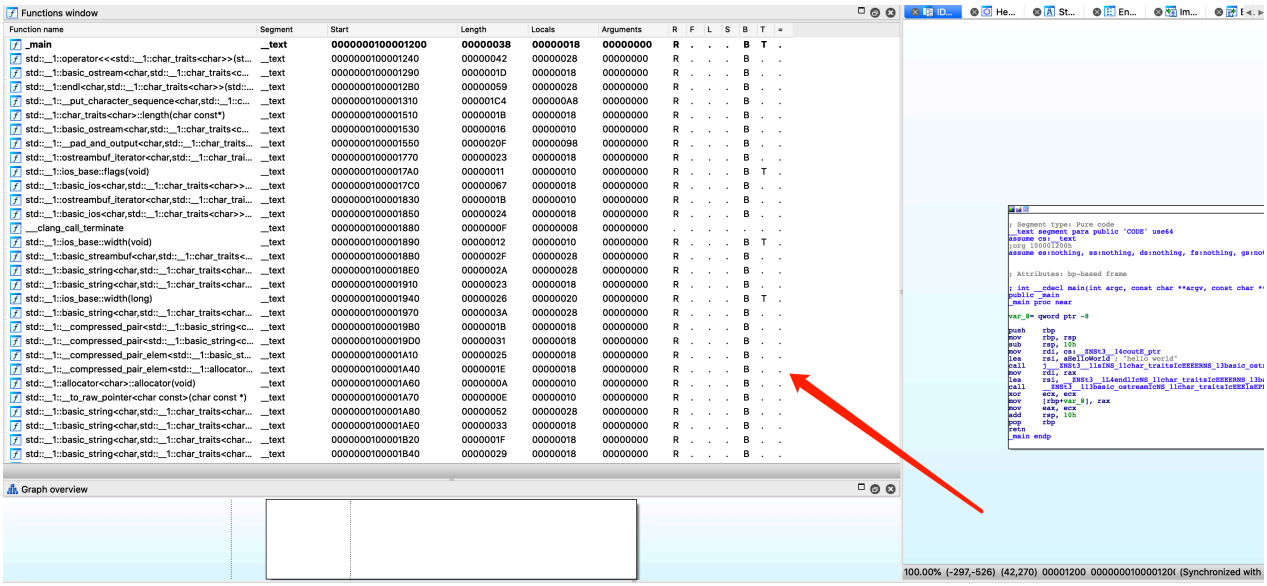
LoadPdbFile命令启动一个特殊的插件PDB.DLL，可以使用Edit-> Plugins子菜单手动运行它。此插件使用MS DIA DLL加载调试信息，因此只能加载与当前安装的IMAGEHLP.DLL兼容的PDB。

由于PDB.DLL使用仅在Windows上可用的MS DIA DLL，因此在Unix上加载PDB信息需要不同的设置。用户必须在Windows计算机上启动win32_remote.exe调试器服务器。Unix上的PDB插件可以连接到调试器服务器，并使用它来加载PDB文件。请注意，这里我们不调试任何东西，而是使用win32_remote.exe实用程序从PDB文件中提取信息并将其传输到IDA。为了确保一切正常，您必须提供以下内容：

-运行PDB.DLL或win32_remote.exe的计算机必须具有
安装的Microsoft调试工具。如有疑问，请验证
Microsoft的dia2dump.exe实用程序可以运行和加载PDB文件
在那台计算机上。

-如果IDA在Unix上运行，则必须有网络连接
在Unix计算机和运行win32_remote.exe的计算机之间。
您可以编辑pdb.cfg文件并指定远程
Windows计算机位于，因此您无需分别指定它

我们在窗口的左侧可以看到一些函数的名字以及函数的有趣的信息，列举IDA在数据库中识别的每一个函数




从左到右的功能：

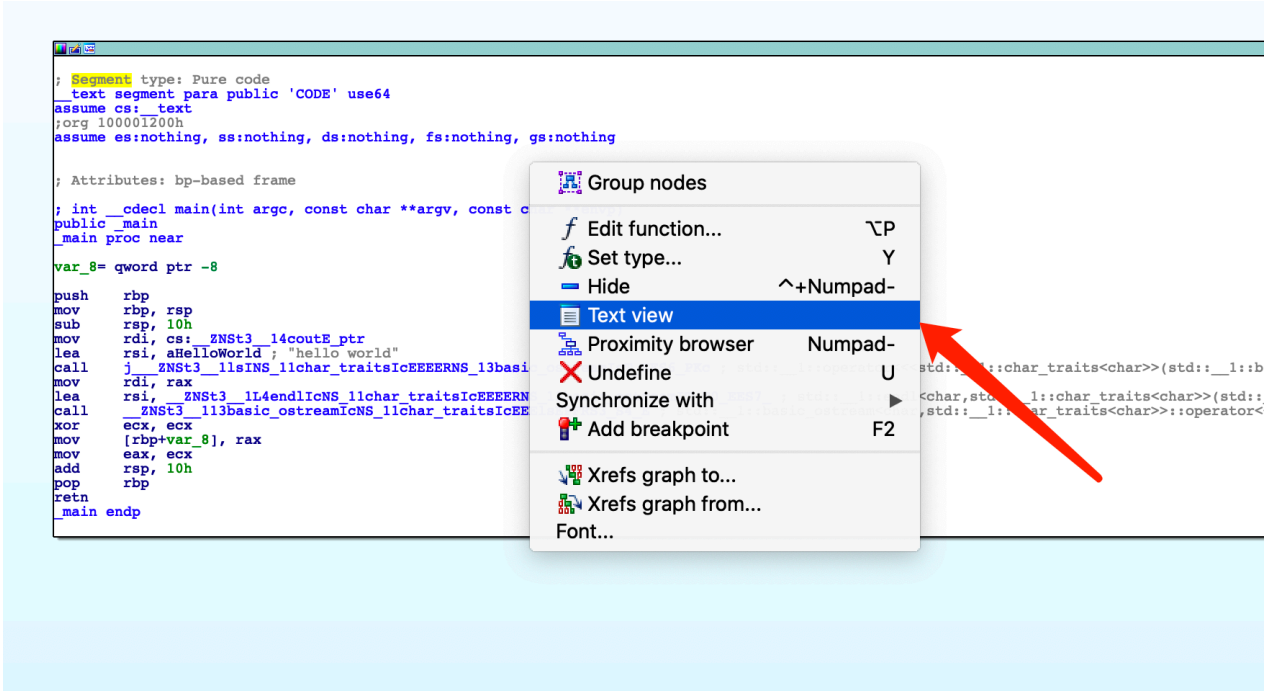
-函数名称
  -包含功能的细分
  -段内功能的偏移量
  -函数长度（以字节为单位）
  -局部变量+保存的寄存器的大小（以字节为单位）
  -传递给函数的参数的大小（以字节为单位）

该窗口的最后一列具有以下格式：

  R-函数返回给调用者
  F-远距功能
  L-库函数
  S-静态功能
  B-基于BP的帧。IDA会自动转换
    所有要存储的帧指针[BP + xxx]操作数
    变量。
  T-函数具有类型信息
  =-帧指针等于初始堆栈指针
    在这种情况下，框架指针指向框架的底部
  M-保留
  S-保留
  我-保留
  C-保留
  D-保留
  V-保留

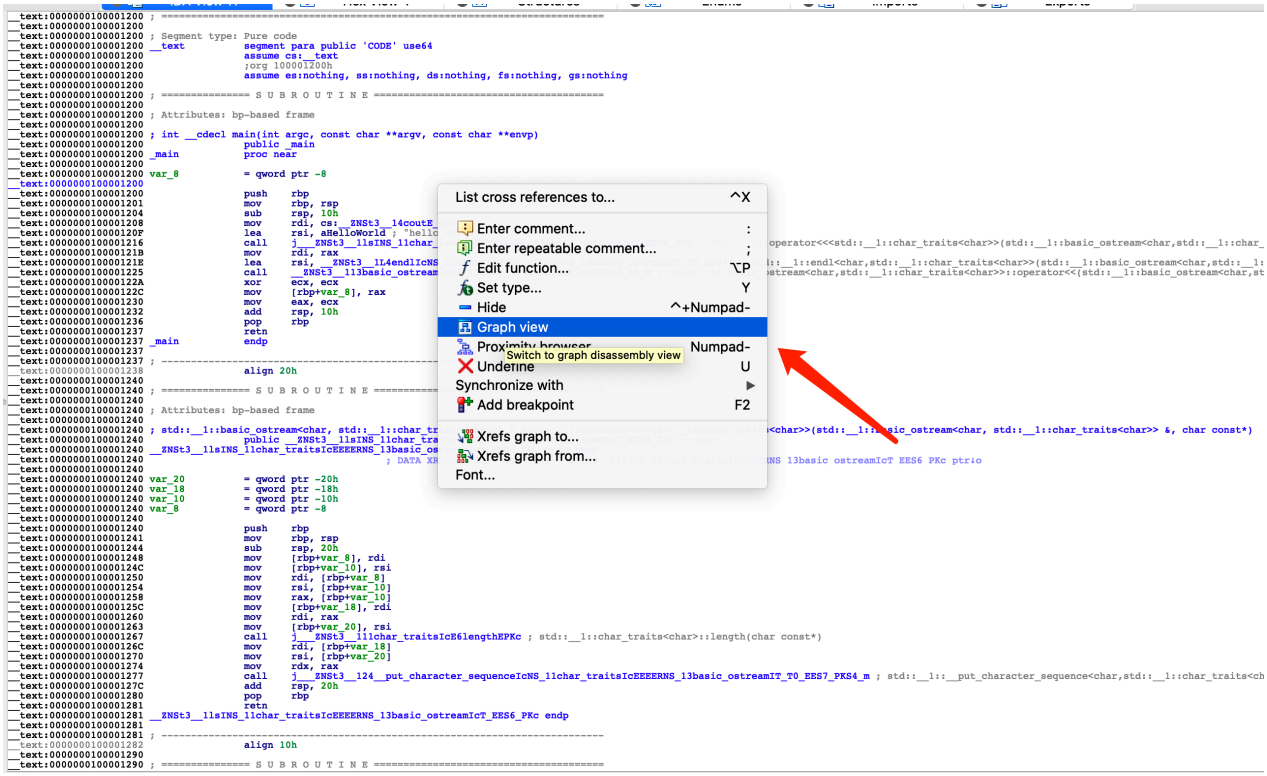
回到我们的代码的窗口

Text view：

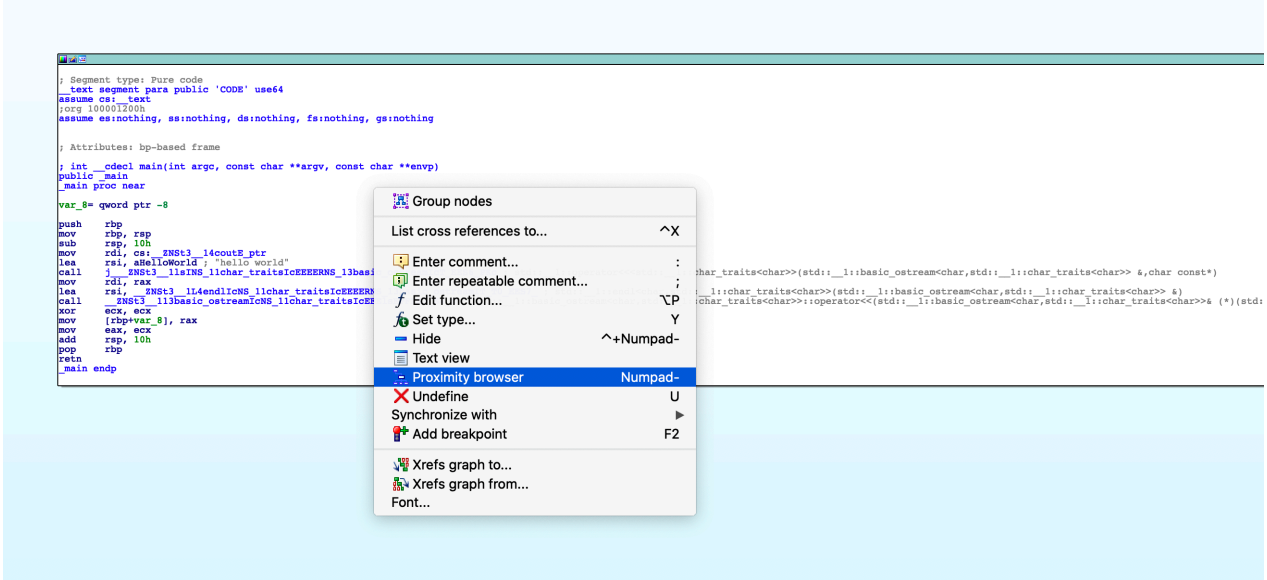

; Segment type: Pure code
__text segment para public 'CODE' use64
assume cs:__text
;org 100001200h
assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const c
public _main
_main proc near
var_8= qword ptr -8
push rbp
mov rbp, rsp
sub rsp, 10h
mov rdi, cs:__ZNSt3__14coutE_ptr
lea rsi, aHelloWorld ; "hello world"
call j___ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basi
mov rdi, rax
lea rsi, __ZNSt3__1L4endlIcNS_11char_traitsIcEEEERN
call __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEE
xor ecx, ecx
mov [rbp+var_8], rax
mov eax, ecx
add rsp, 10h
pop rbp
retn
_main endp
Group nodes
Edit function... ⌥P
Set type... Y
Hide ^+Numpad-
Text view
Proximity browser Numpad-
Undefine U
Synchronize with
Add breakpoint F2
Xrefs graph to...
Xrefs graph from...
Font...

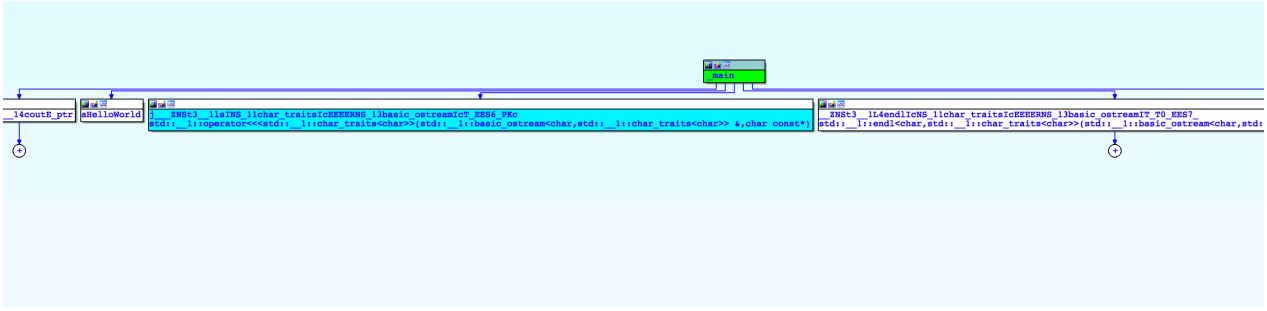

; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near
var_8 = qword ptr -8
push rbp
mov rbp, rsp
sub rsp, 10h
mov rdi, cs:__ZNSt3__14coutE_ptr
lea rsi, aHelloWorld ; "hello world"
; std::__1::basic_ostream<char, std::__1::char_traits<char>> & std::__1::operator<<<std::__1::char_traits<char>>(std::__1::basic_ostream<char, std::__1::char_traits<char>> &, char const*)
__ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_ES6_PKc proc near
; DATA XREF: la symbol ptr: __ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_ES6_PKc ptr
var_20 = qword ptr -20h
var_18 = qword ptr -18h
var_10 = qword ptr -10h
var_8 = qword ptr -8
push rbp
mov rbp, rsp
sub rsp, 20h
mov [rbp+var_8], rdi
mov [rbp+var_10], rsi
mov rdi, [rbp+var_8]
mov rsi, [rbp+var_10]
mov rax, [rbp+var_10]
mov [rbp+var_18], rdi
mov rdi, rax
mov [rbp+var_20], rsi
call j___ZNSt3__111char_traitsIcE6lengthEPKc ; std::__1::char_traits<char>::length(char const*)
mov rdi, [rbp+var_18]
mov rsi, [rbp+var_20]
mov rdx, rax
call j___ZNSt3__124__put_character_sequenceIcNS_11char_traitsIcEEEERNS_13basic_ostreamIcT_T0_EES7_PKS4_m ; std::__1::__put_character_sequence<char,std::__1::char_traits<char>>(std::__1::bas
add rsp, 20h
pop rbp
retn
__ZNSt3__1lsINS_11char_traitsIcEEEERNS_13basic_ostreamIcT_EES6_PKc endp
align 20h

Graph view:

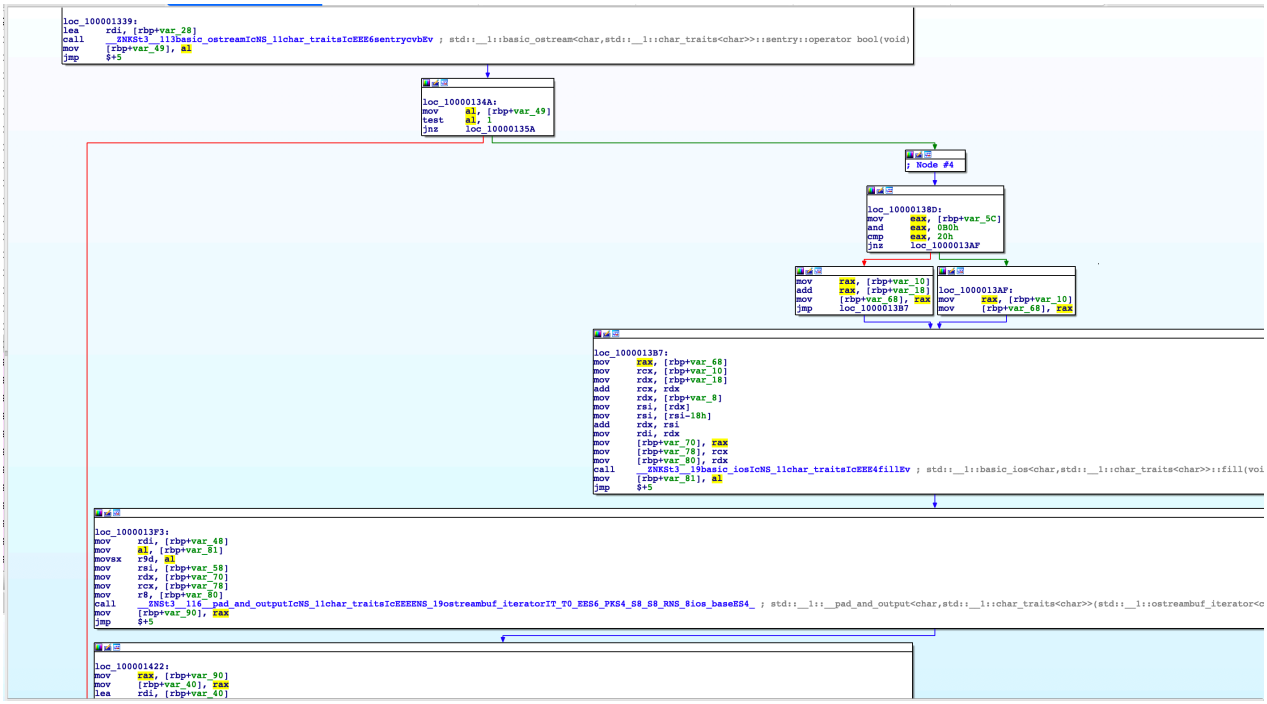



Proximity browser:



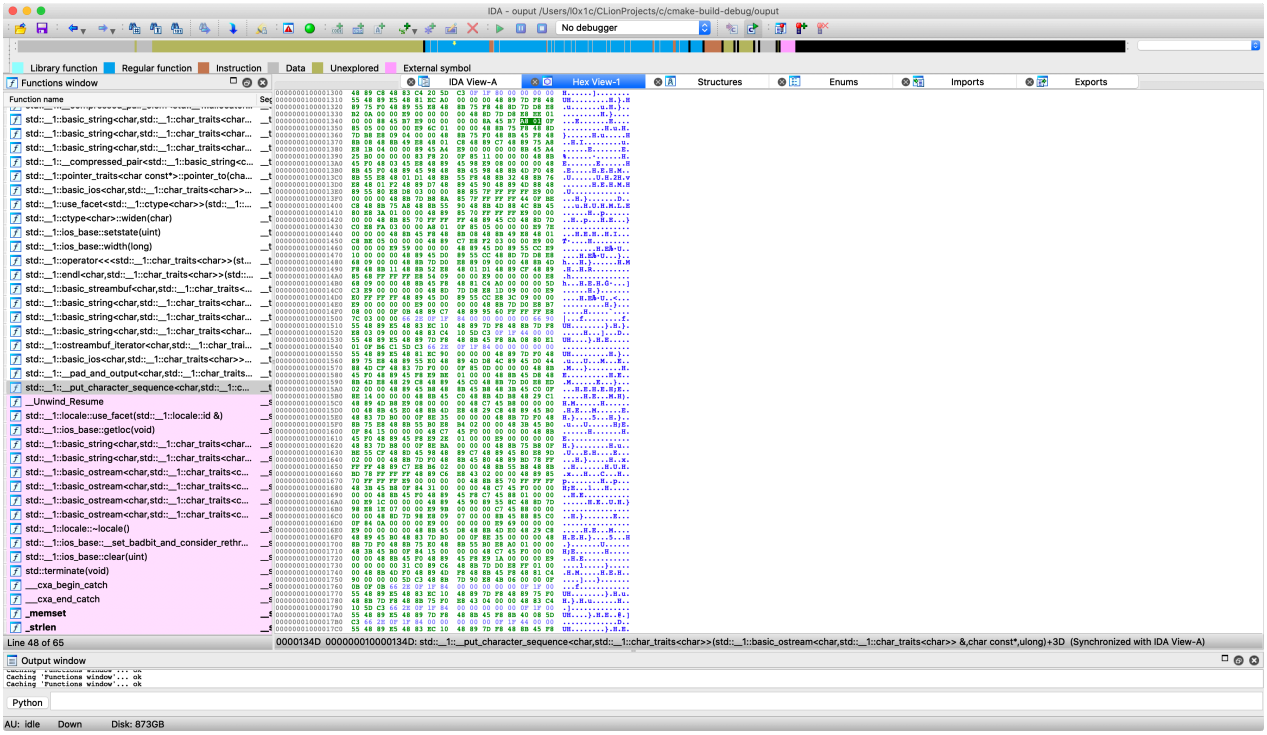


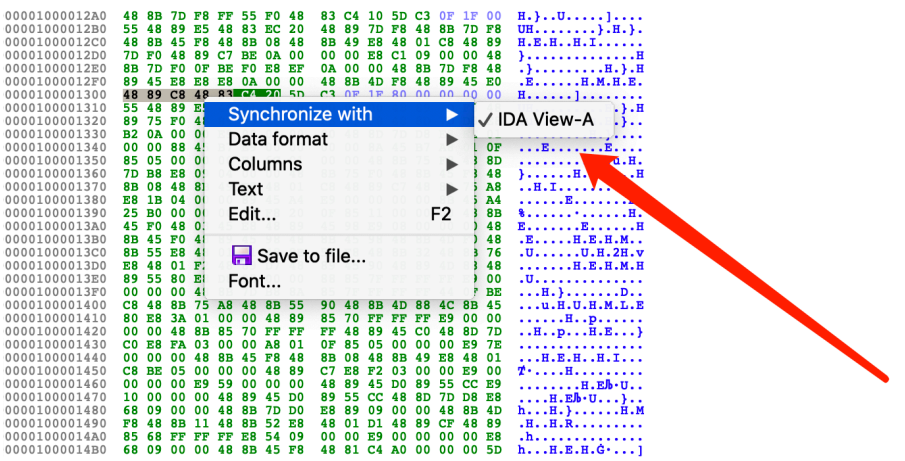

在图形模式下：




在条件跳转位置终止的基本块可能会产生两种流，YES边的检讨默认为绿色，NO边的箭头默认为红色，只有一个后继块的基本块会利用一个正常边默认为蓝色指向下一个即将执行的块

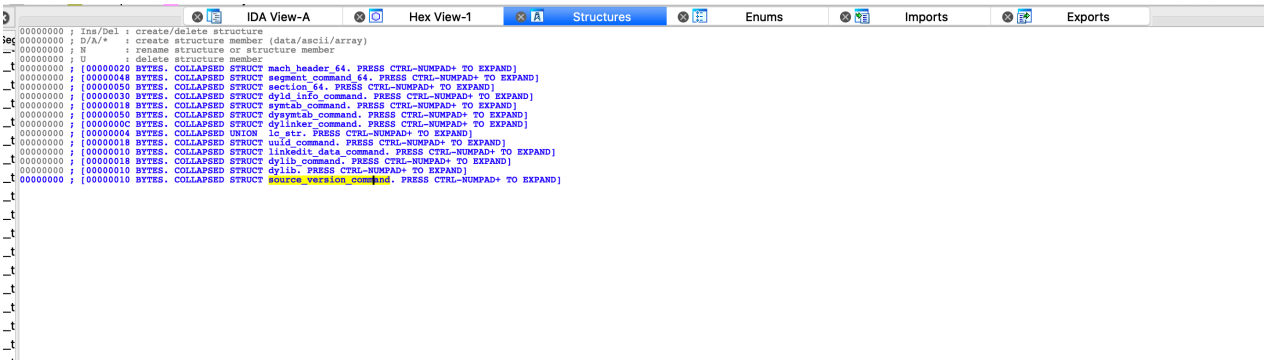
十六进制窗口：



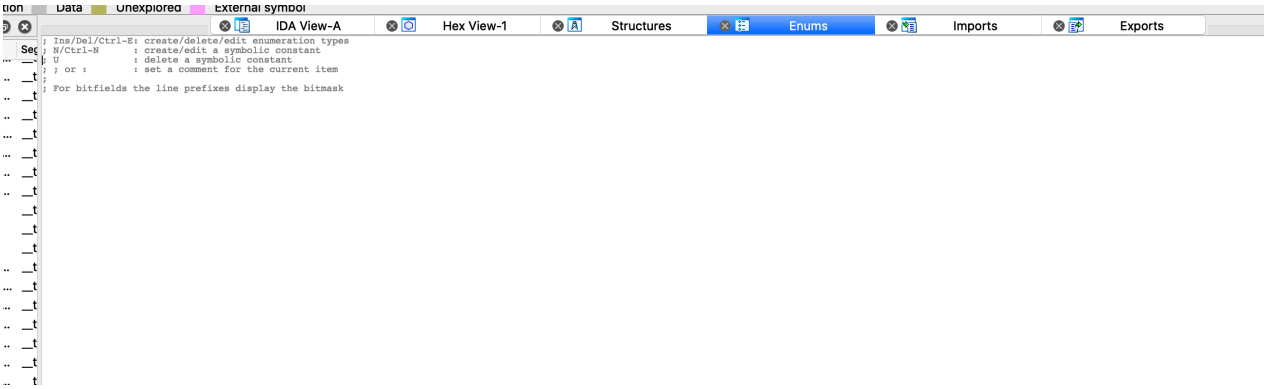

和ida view-A对应，一般来说如果我们的hex窗口中显示的全部都是问号，表示ida无法识别给定的虚拟地址范围内的值，如果程序中包含一个bss节，因为bss不占用文件的空间，但是加载会扩展这一节，以适应程序的静态存储要求

结构体窗口：



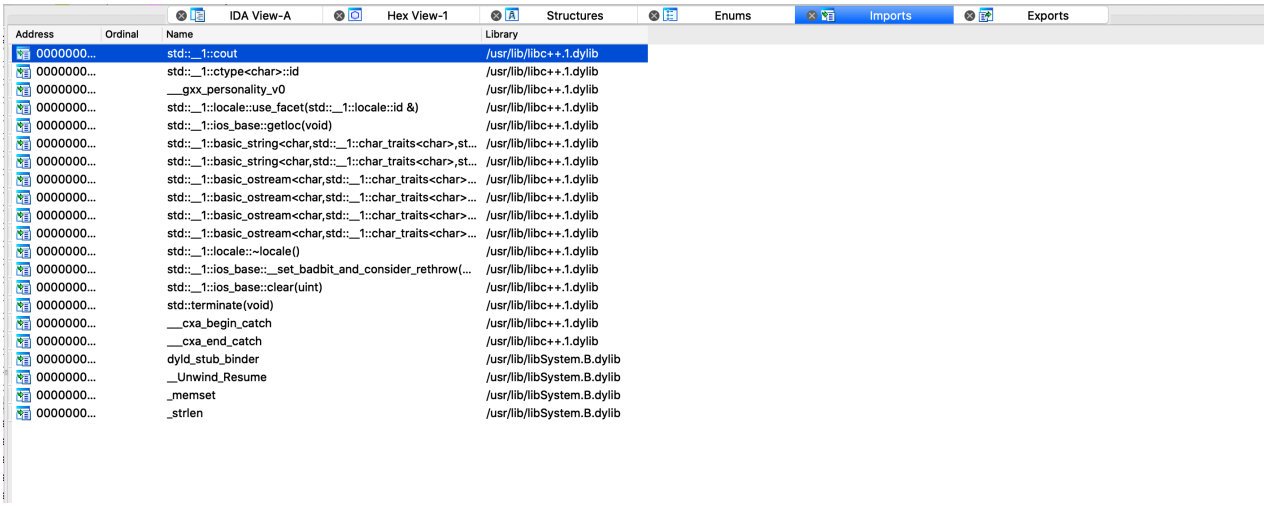

为标准数据结构布局提供现成的参考

枚举窗口：



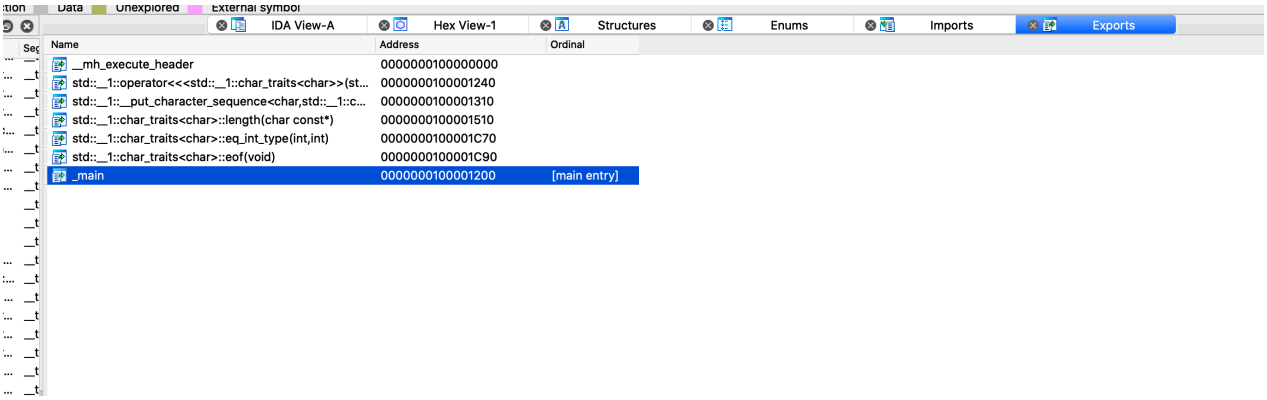

如果线程到枚举数据类型，将在枚举窗口中列出该数据类型，也可以自己枚举代替整数常量

导入窗口：



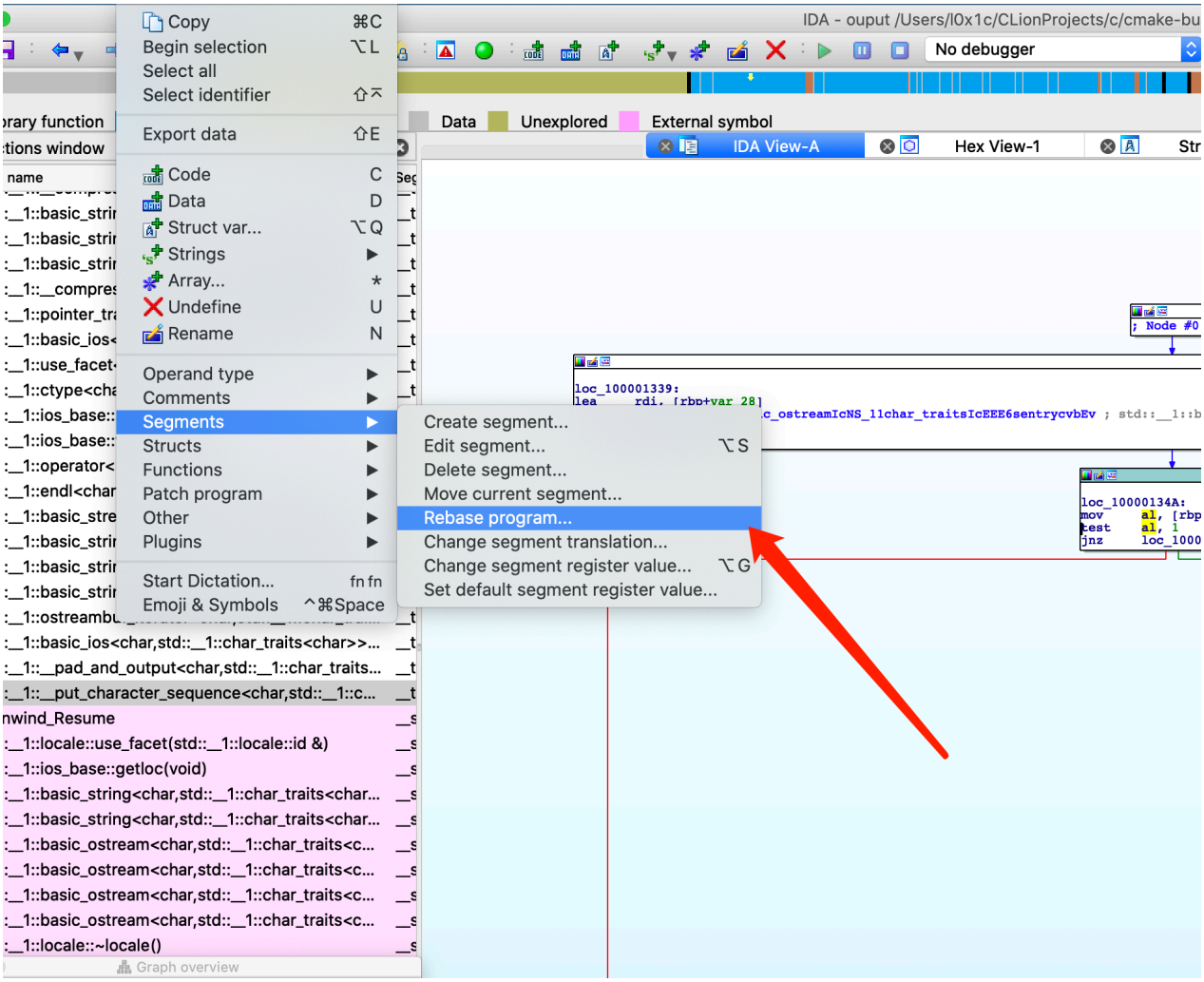

列出被分析的二进制导入的函数

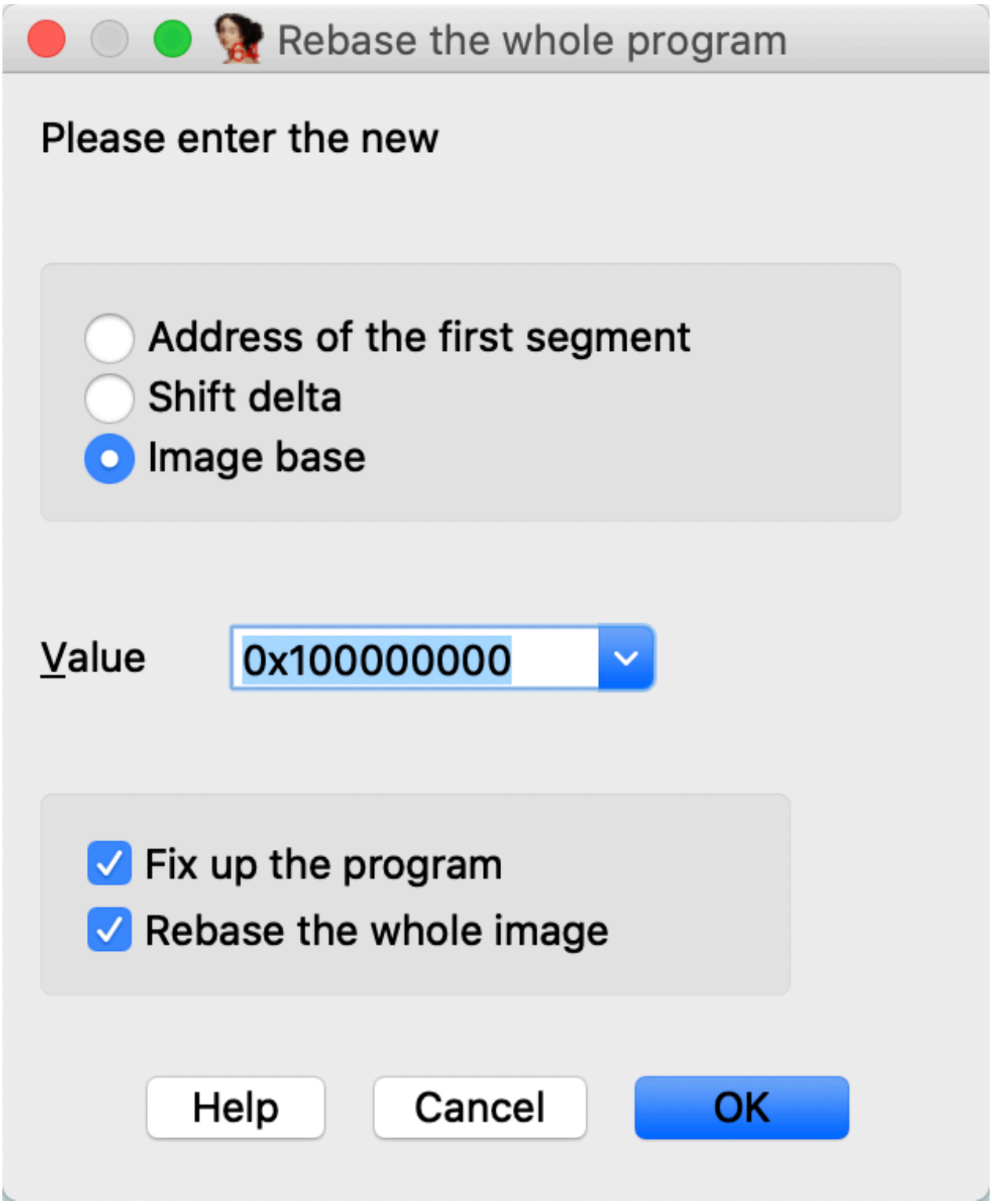
导出窗口：



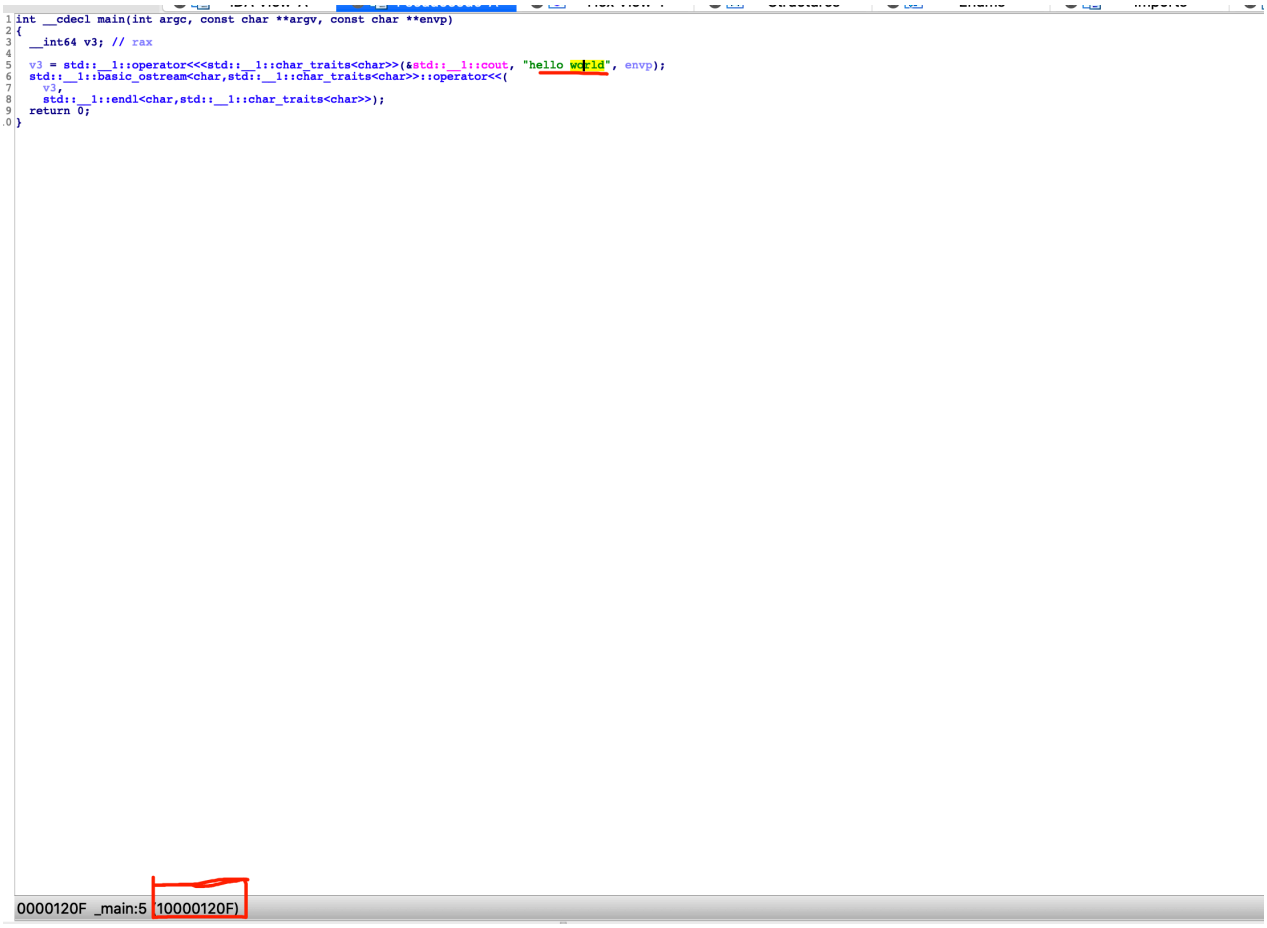

列出文件的入口点，导出窗口至少包含一个项目：程序的执行入口点

一般喜欢用od，以及xdbg进行调试的朋友，如果感觉基址感觉没有对上不太好观看的话可以修改这里


IDA - ouput /Users/l0x1c/CLionProjects/c/cmake-bu
No debugger
Data
Unexplored
External symbol
IDA View-A
Hex View-1
Copy ⌘C
Begin selection ⌥L
Select all
Select identifier ⇧⌅
Export data ⇧E
Code C
Data D
Struct var... ⌥Q
Strings ▶
Array... *
Undefine U
Rename N
Operand type ▶
Comments ▶
Segments ▶
Structs ▶
Functions ▶
Patch program ▶
Other ▶
Plugins ▶
Start Dictation... fn fn
Emoji & Symbols ^⌘Space
Create segment...
Edit segment... ⌥S
Delete segment...
Move current segment...
Rebase program...
Change segment translation...
Change segment register value... ⌥G
Set default segment register value...
Graph overview

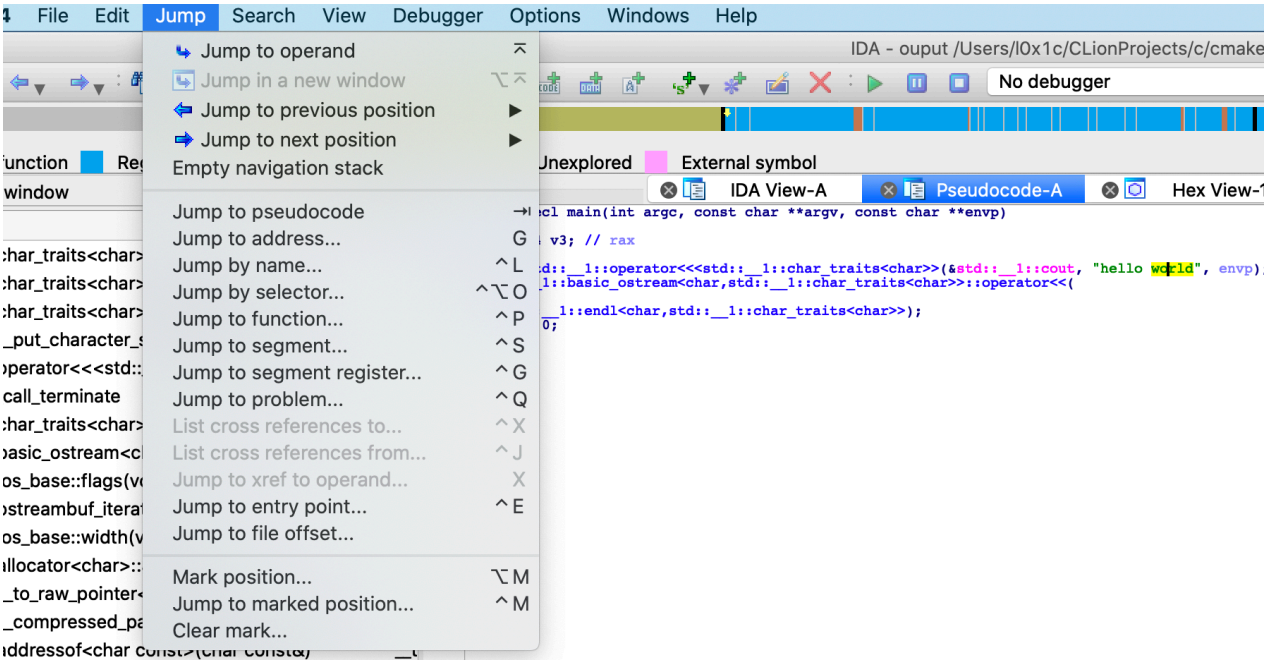



直接修改成相同的就好了，这样就可以相对应上了，一般一个小操作就是

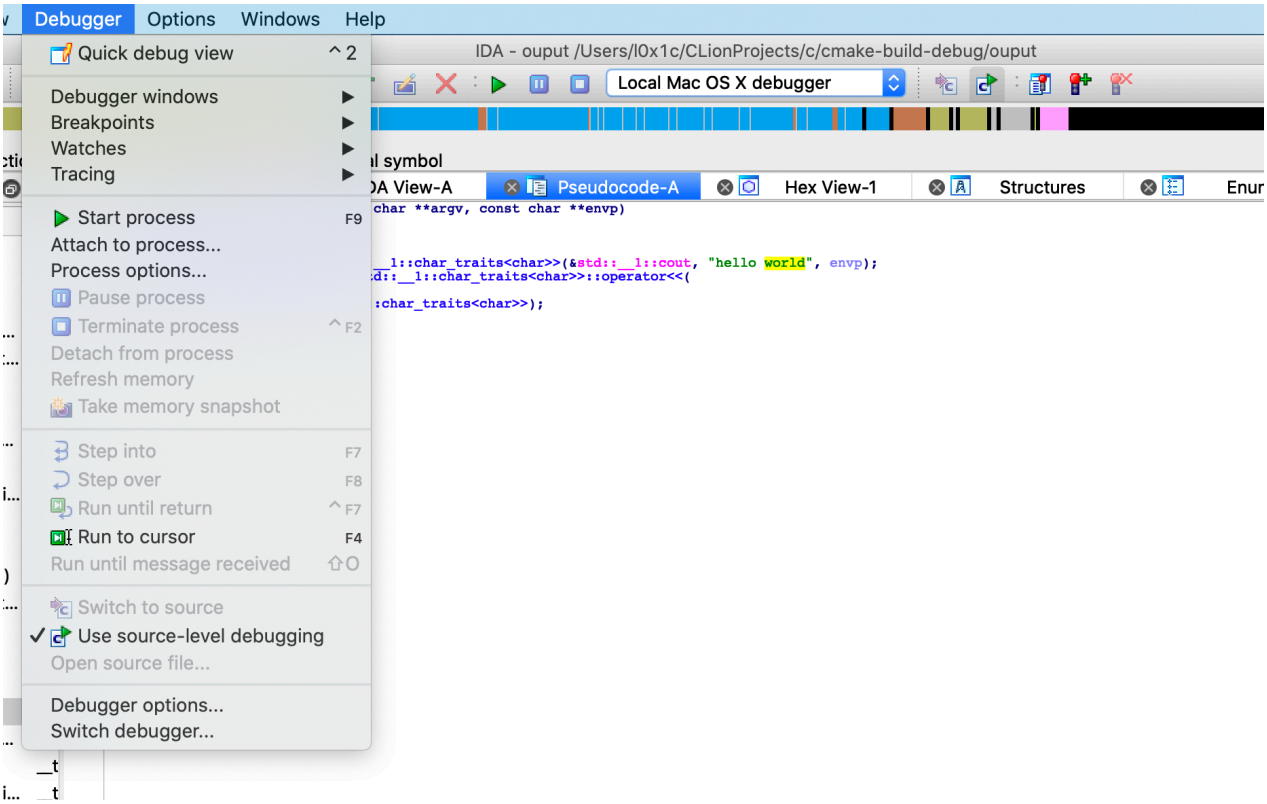



在下面的地方可以看到相应的地址位置，在od以及xdbg进行调试的时候，可以看到相应的位置

如果想跳转到各种位置，以及函数名等等的，可以直接Jump选择跳转




IDA也给我们了一个Debug的功能

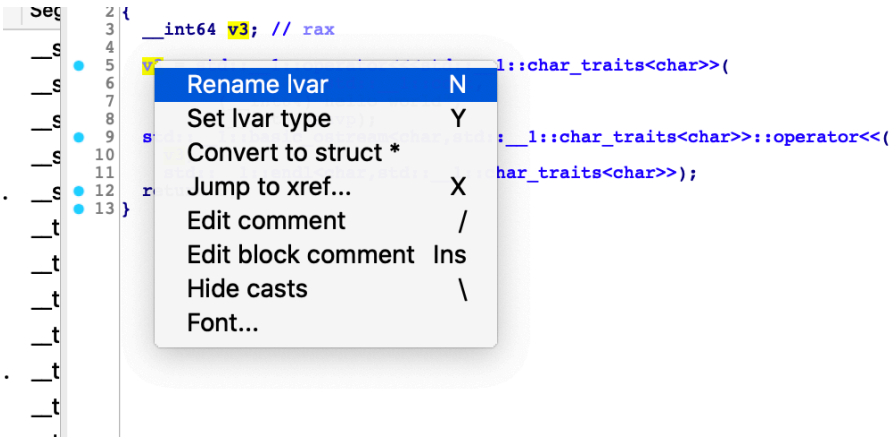


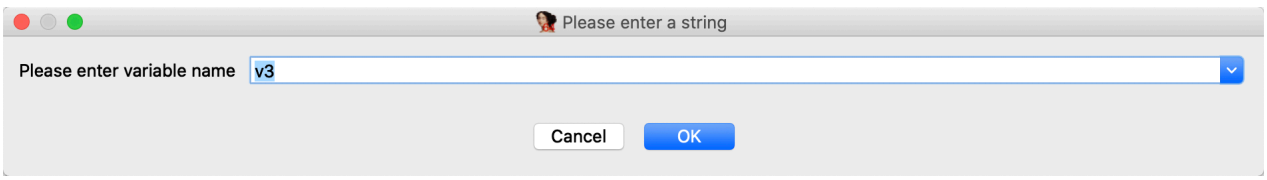

开始调试，下段点等等的操作都有

# Reverse Engineering with IDA Pro – Fixing/Rebuilding Structure/Structs (Pseudocode)

这一节主要讲述了我们的一个f5的操作以及修改成员名字以及，结构体的一些使用

修改成员名字：




假设修改成struct1

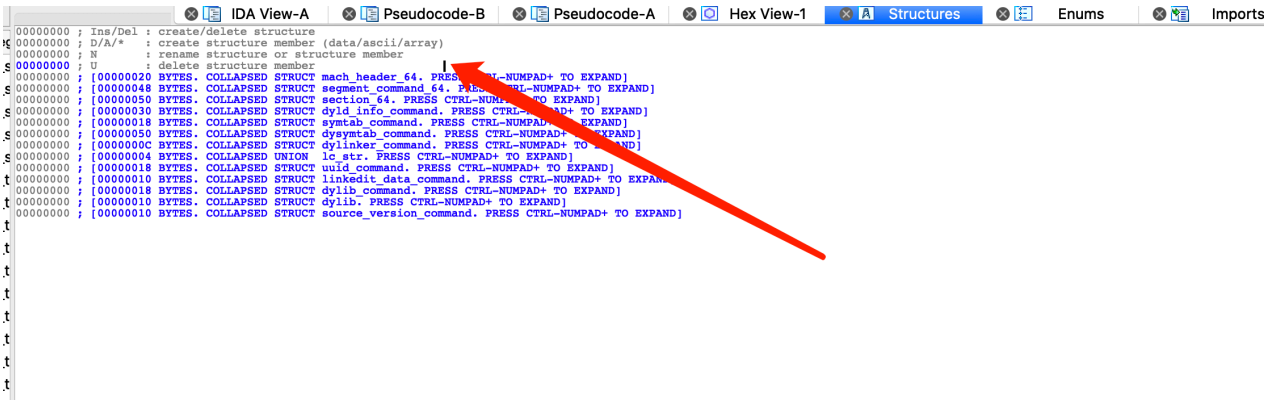
```
IDA View-A | Pseudocode-C | Pseudocode-B | Pseudocode-A

int __cdecl main(int argc, const char **argv, const char **envp)
{
  __int64 struct1; // rax

  struct1 = std::__1::operator<<<std::__1::char_traits<char>>(
              (__int64)&std::__1::cout,
              (__int64)"hello world",
              (__int64)envp);
  std::__1::basic_ostream<char,std::__1::char_traits<char>>::operator<<(
    struct1,
    std::__1::endl<char,std::__1::char_traits<char>>);
  return 0;
}
```

可以看到已经修改成功了

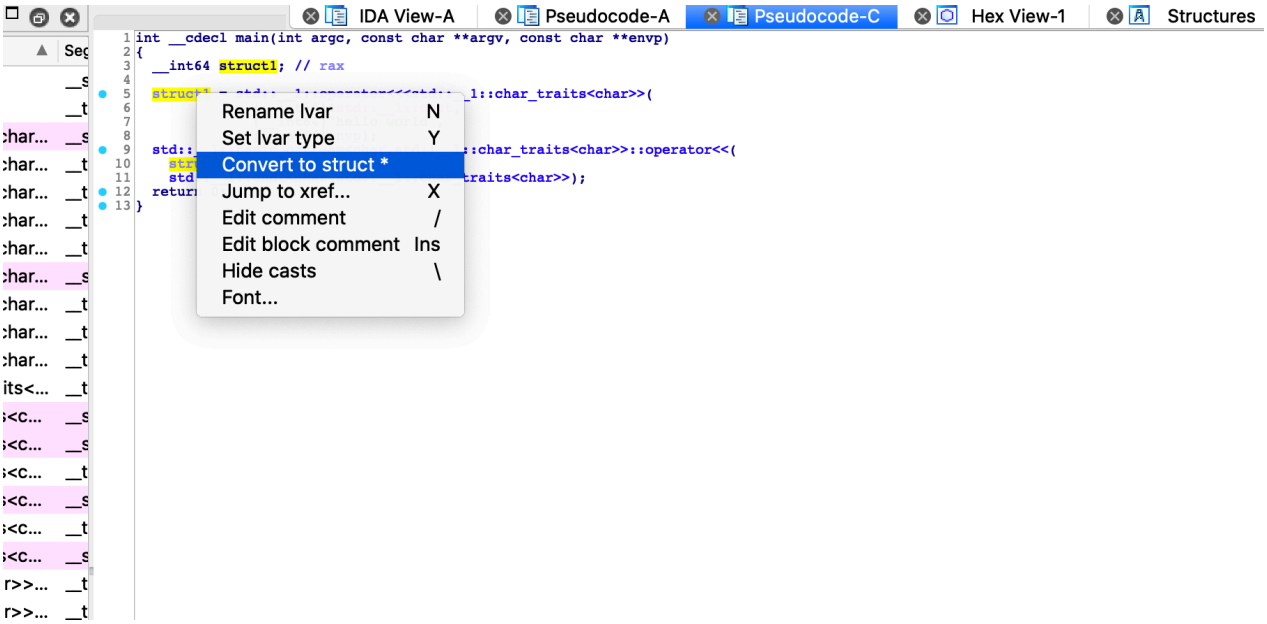
创建结构体：



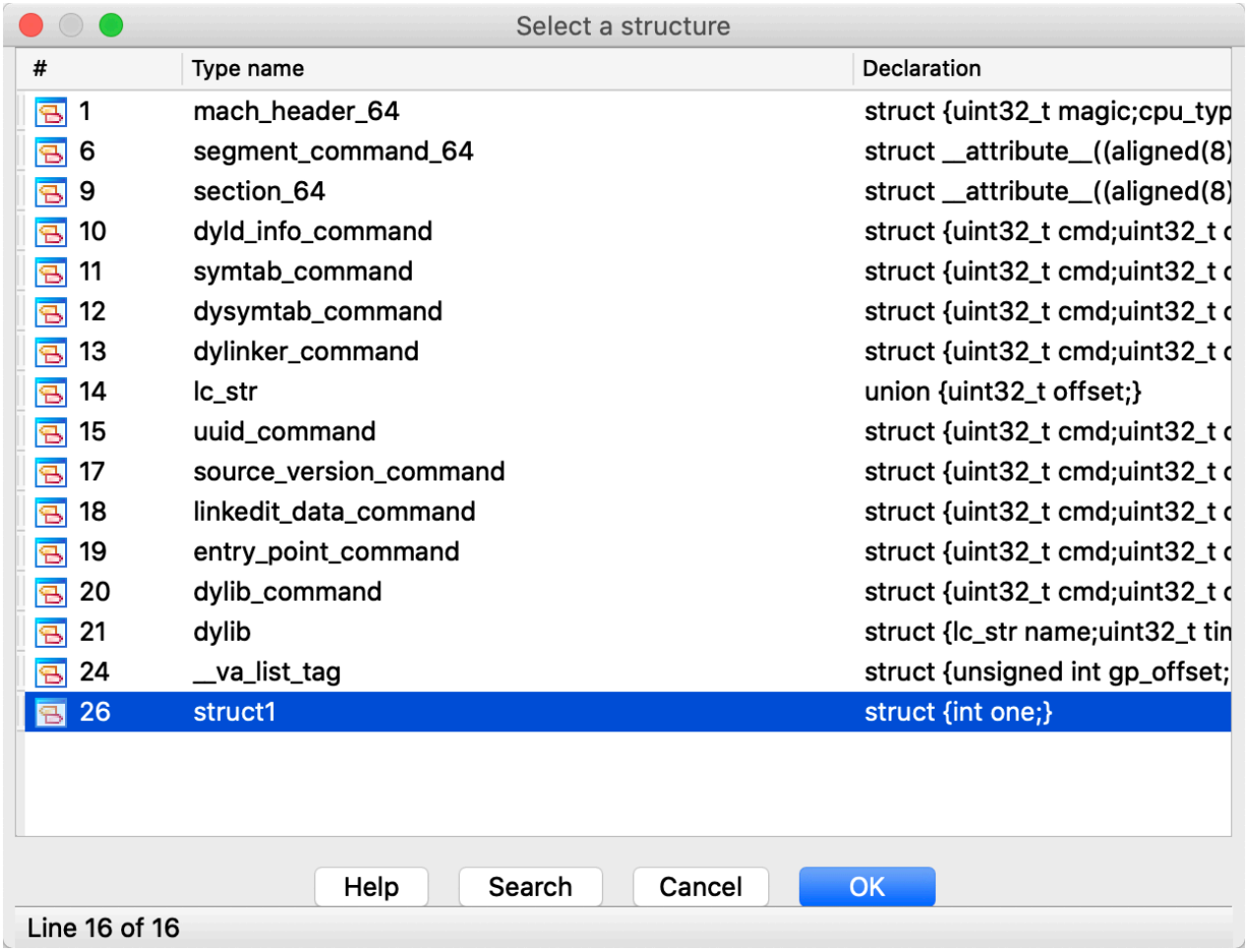

这里面可以自己去定义一些结构体，新建结构体，成员的大小，重命名成员，删除成员，删除结构体等等的操作

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; [00000020 BYTES. COLLAPSED STRUCT mach_header_64. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 struct1         struc ; (sizeof=0x4, mappedto_26)
00000000 one             dd ?
00000004 struct1         ends
00000004
00000000 ; [00000048 BYTES. COLLAPSED STRUCT segment_command_64. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000050 BYTES. COLLAPSED STRUCT section_64. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000030 BYTES. COLLAPSED STRUCT dyld_info_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000018 BYTES. COLLAPSED STRUCT symtab_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000050 BYTES. COLLAPSED STRUCT dysymtab_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [0000000C BYTES. COLLAPSED STRUCT dylinker_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000004 BYTES. COLLAPSED UNION  lc_str. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000018 BYTES. COLLAPSED STRUCT uuid_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000010 BYTES. COLLAPSED STRUCT linkedit_data_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000018 BYTES. COLLAPSED STRUCT dylib_command. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000010 BYTES. COLLAPSED STRUCT dylib. PRESS CTRL-NUMPAD+ TO EXPAND]
00000000 ; [00000010 BYTES. COLLAPSED STRUCT source_version_command. PRESS CTRL-NUMPAD+ TO EXPAND]
```

我们定义一个struct1的结构体，里面具有one的成员




我们链接一下结构体，看一下




可以看到结构体的位置被修改了，自己就是演示一下这个操作，具体的效果看一下这个视频里的我截取出来了：

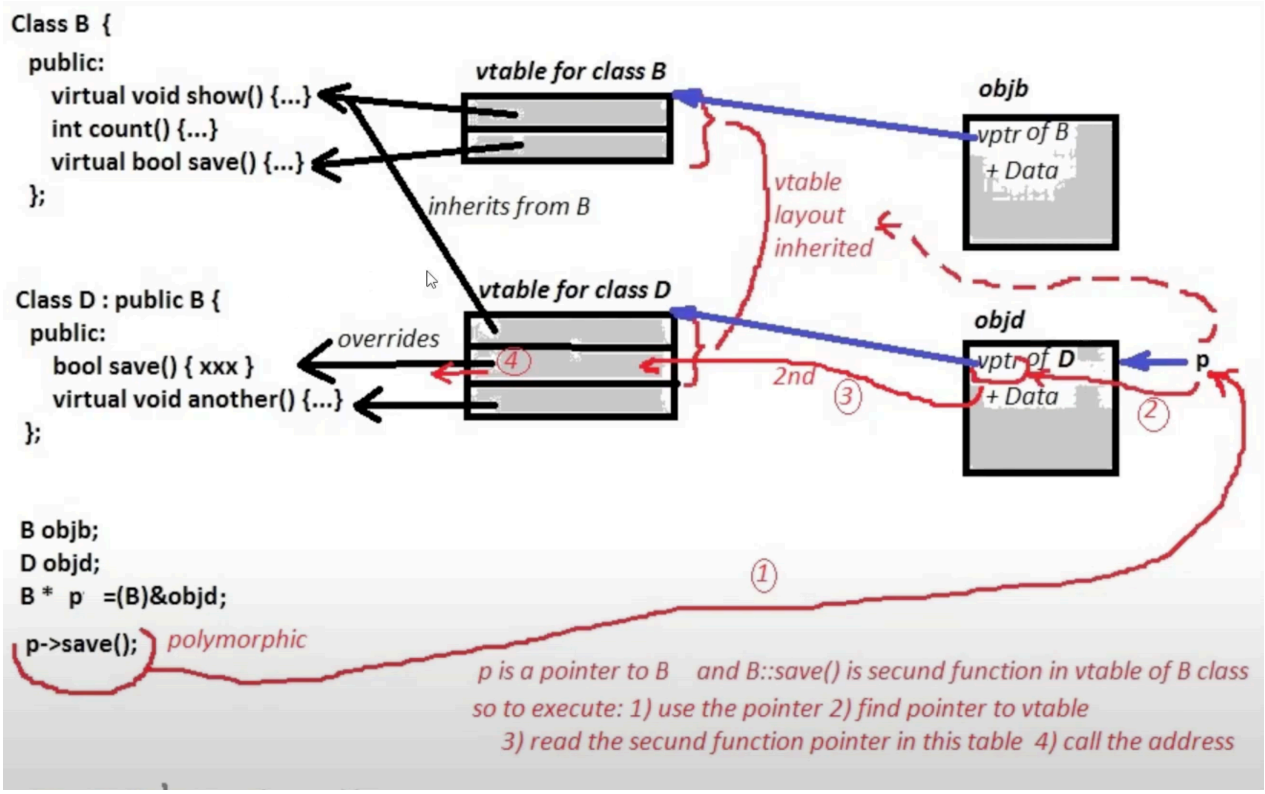
```
__int64 sub_1400010A0()
{
  mystruct1 *ptr_to_struct1; // rbx
  _QWORD *v1; // rax
  _QWORD *v2; // rdi

  ptr_to_struct1 = (mystruct1 *)malloc(0x20ui64);
  v1 = malloc(0x10ui64);
  ptr_to_struct1->myint1 = 1;
  ptr_to_struct1->myint2 = 2;
  v2 = v1;
  ptr_to_struct1->myfloat1 = 0x40400000;
  sub_140001010("Val : %d \n", 1i64);
  ptr_to_struct1[1].myint1 = 1095520072;
  *(_QWORD *)&ptr_to_struct1[1].myint2 = sub_140001070;
  *(_QWORD *)&ptr_to_struct1[2].myint1 = v2;
  *v2 = sub_140001080;
  v2[1] = sub_140001090;
  sub_140001080(1234i64);
  ((void (__fastcall *)(signed __int64))v2[1])(5678i64);
  return 0i64;
}
```

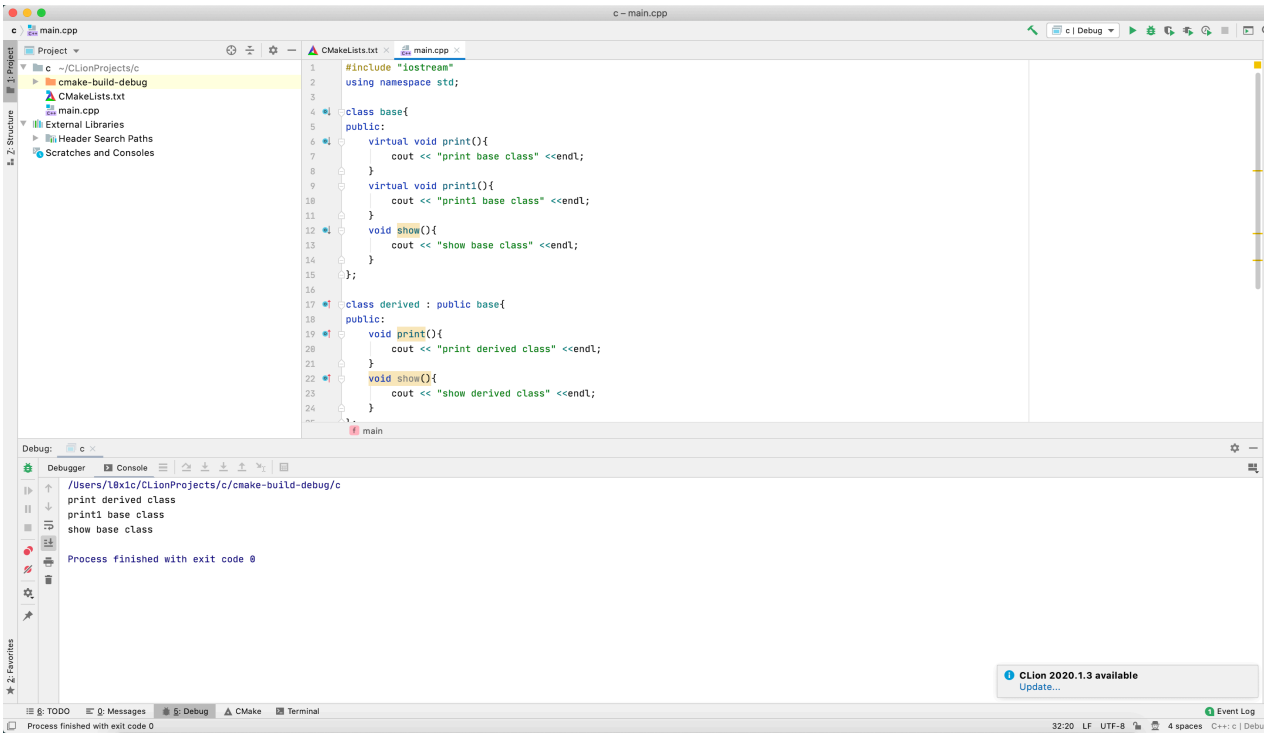
这里就是结构体修改后的结果，以前是一个数组的一个形式

# C++ Reverse Engineering with IDA Pro – Rebuilding virtual function table (vftable)

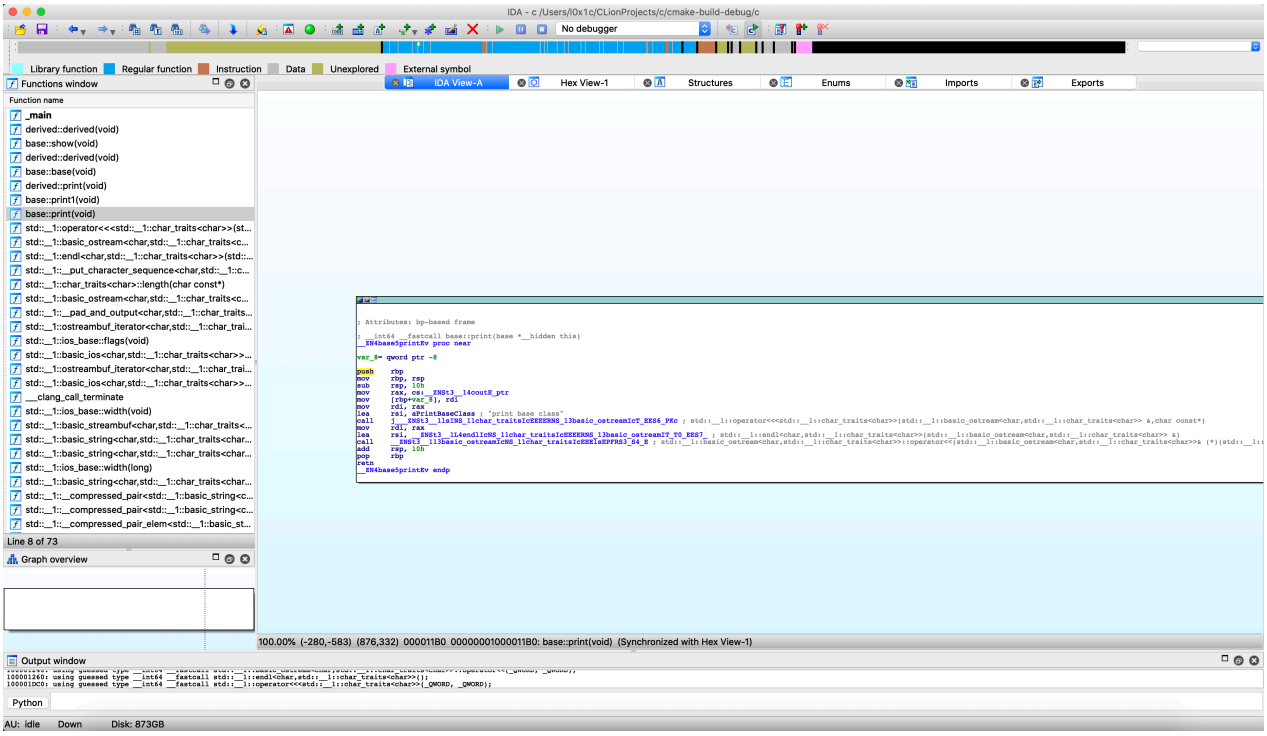
这一节主要去让我们快速去找到虚函数表的一个方法插件



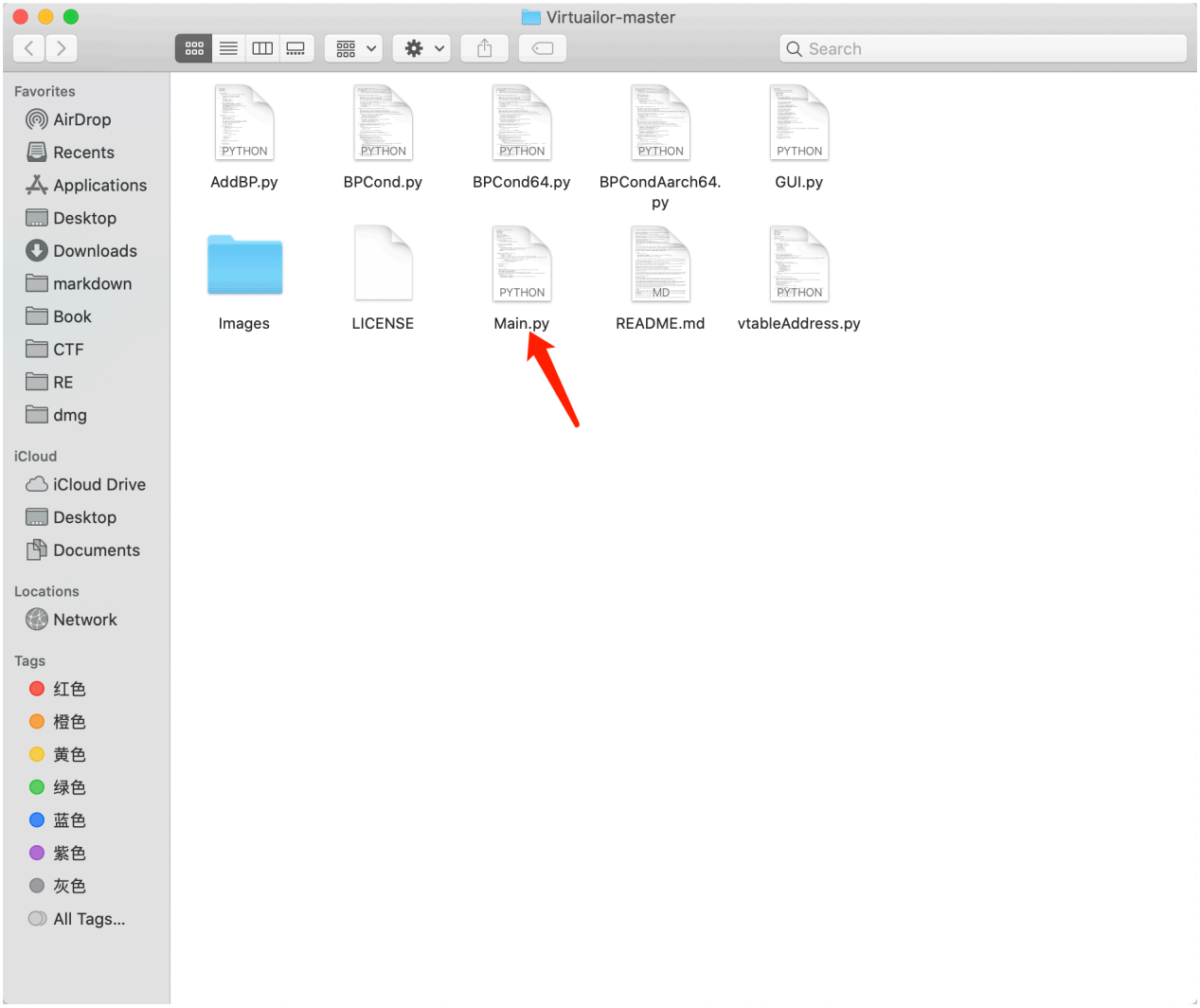

代码写一下测试一下：

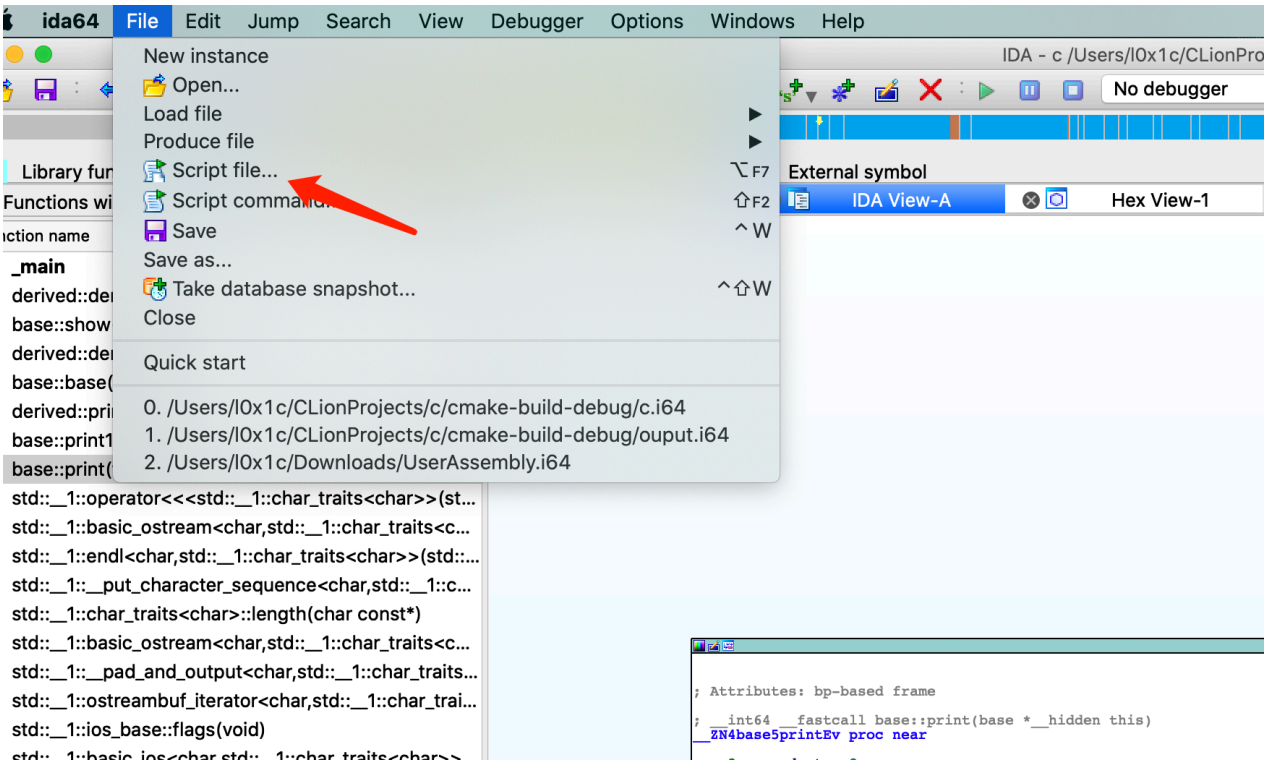



我们去ida看一下：

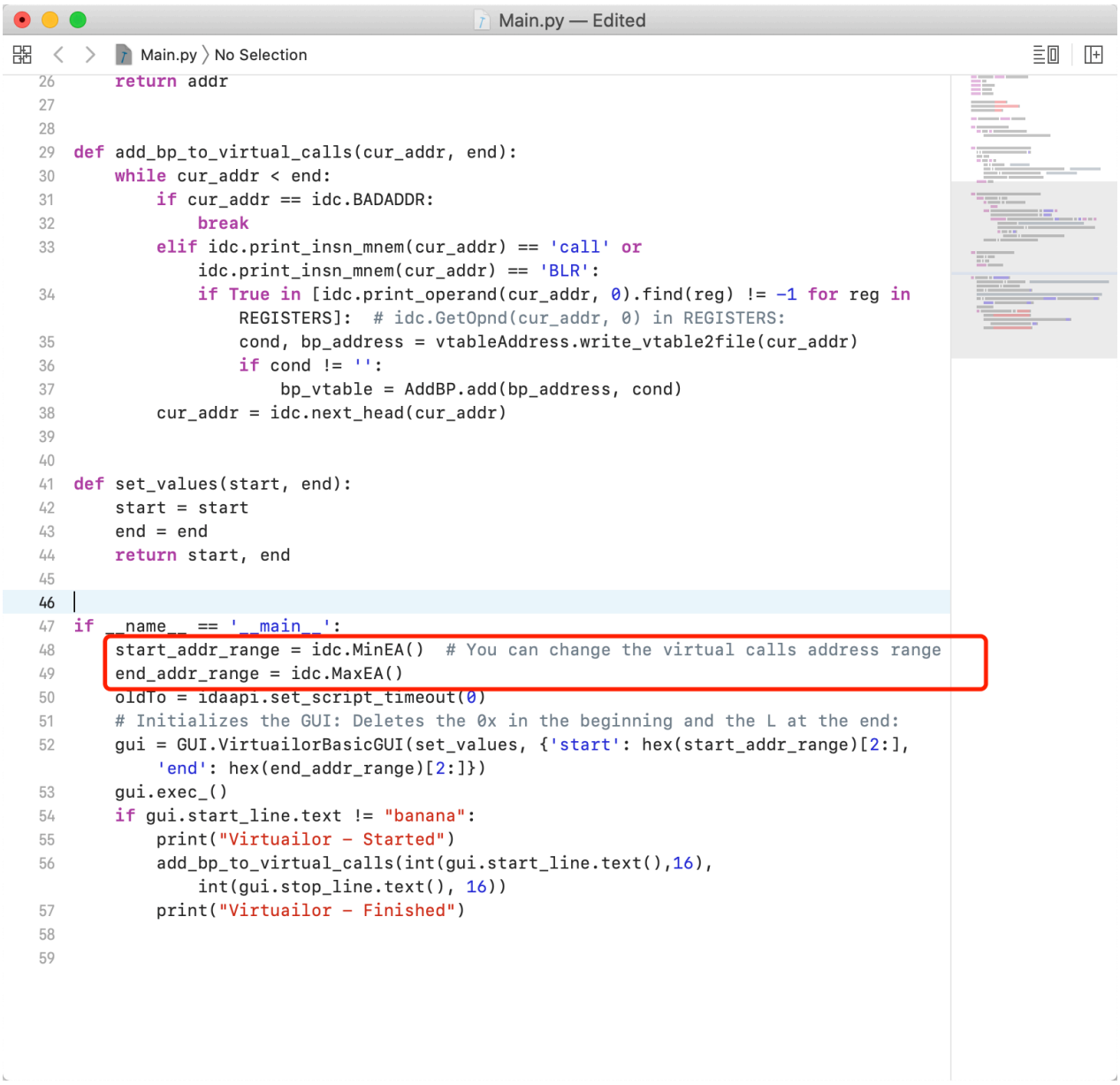



我们使用插件的话可以把虚函数表很好的在结构体里展示出来：https://github.com/0xgalz/Virtuailor


Virtuailor-master
Search
Favorites
AirDrop
Recents
Applications
Desktop
Downloads
markdown
Book
CTF
RE
dmg
iCloud
iCloud Drive
Desktop
Documents
Locations
Network
Tags
红色
橙色
黄色
绿色
蓝色
紫色
灰色
All Tags...
AddBP.py
BPCond.py
BPCond64.py
BPCondAarch64.py
GUI.py
Images
LICENSE
Main.py
README.md
vtableAddress.py

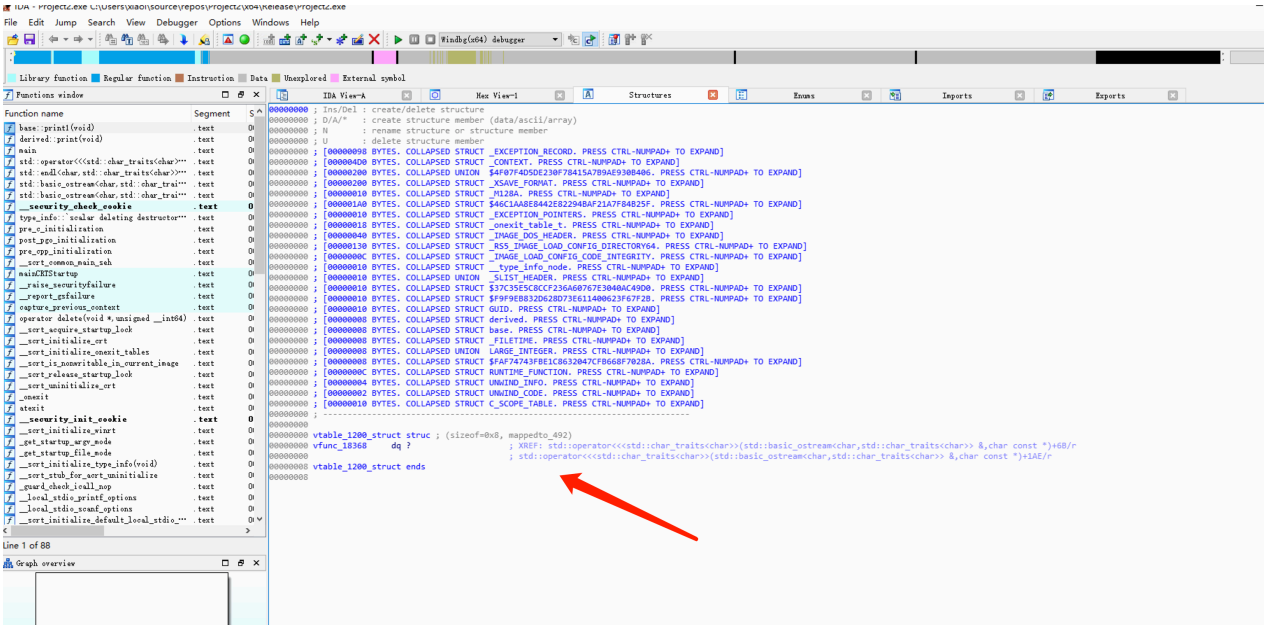

ida64
File
Edit
Jump
Search
View
Debugger
Options
Windows
Help
New instance
Open...
Load file
Produce file
Script file...
Script command...
Save
Save as...
Take database snapshot...
Close
Quick start
0. /Users/l0x1c/CLionProjects/c/cmake-build-debug/c.i64
1. /Users/l0x1c/CLionProjects/c/cmake-build-debug/ouput.i64
2. /Users/l0x1c/Downloads/UserAssembly.i64
IDA - c /Users/l0x1c/CLionPro
No debugger
External symbol
IDA View-A
Hex View-1
_main
std::__1::operator<<<std::__1::char_traits<char>>(st...
std::__1::basic_ostream<char,std::__1::char_traits<c...
std::__1::endl<char,std::__1::char_traits<char>>(std::...
std::__1::__put_character_sequence<char,std::__1::c...
std::__1::char_traits<char>::length(char const*)
std::__1::basic_ostream<char,std::__1::char_traits<c...
std::__1::__pad_and_output<char,std::__1::char_traits...
std::__1::ostreambuf_iterator<char,std::__1::char_trai...
std::__1::ios_base::flags(void)
; Attributes: bp-based frame
; __int64 __fastcall base::print(base *__hidden this)
__ZN4base5printEv proc near


这个main.py我们需要修改一下



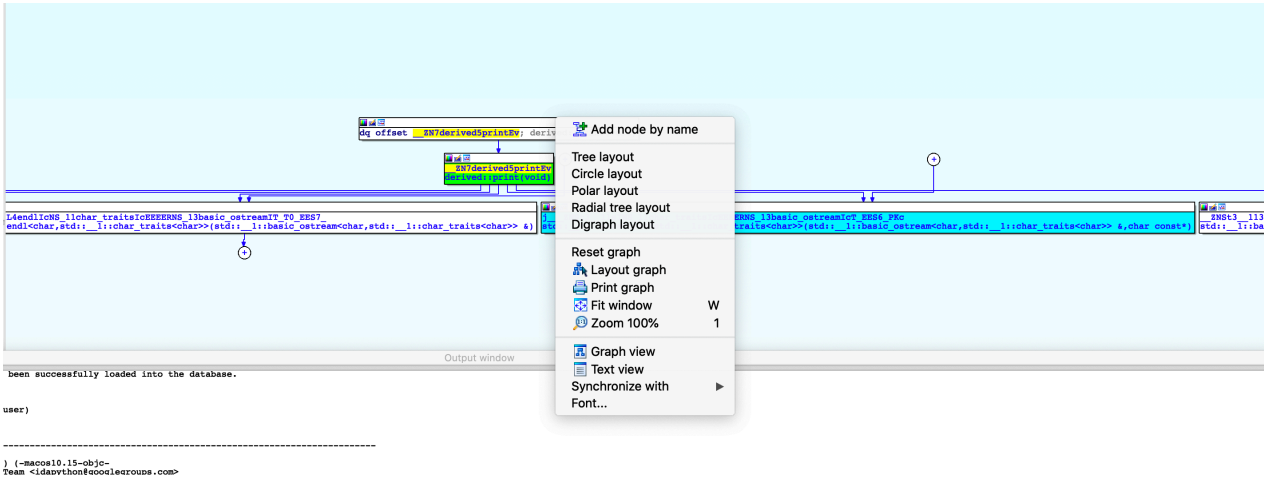

我们运行脚本后可看到成功了


我们local windbg运行后可以直接看到虚函数表：

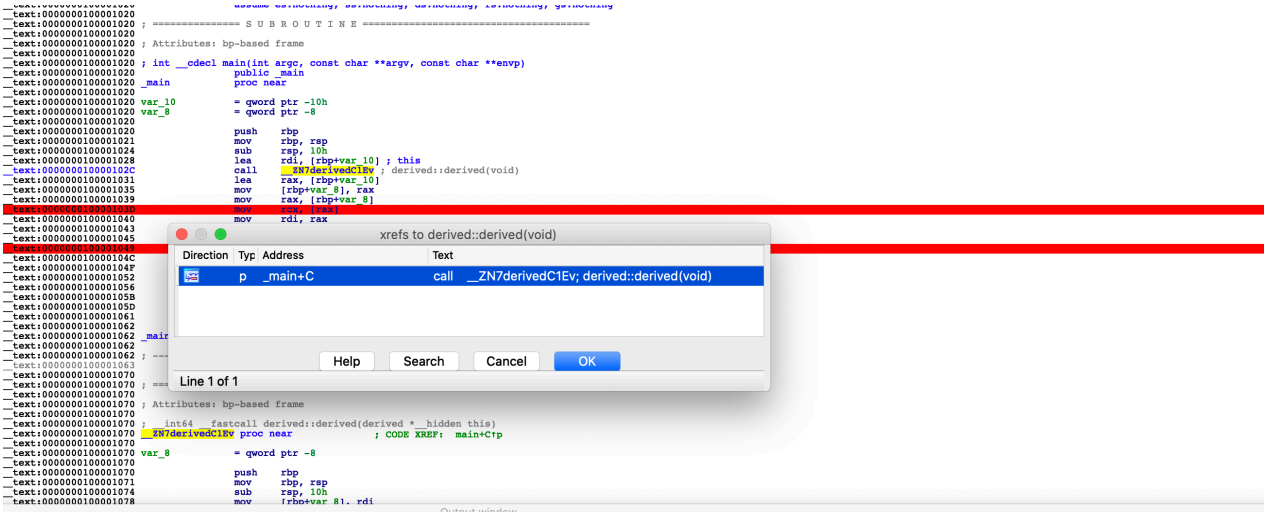


# Reverse Engineering Tutorial with IDA Pro – Function Cross Reference & Proximity Browser

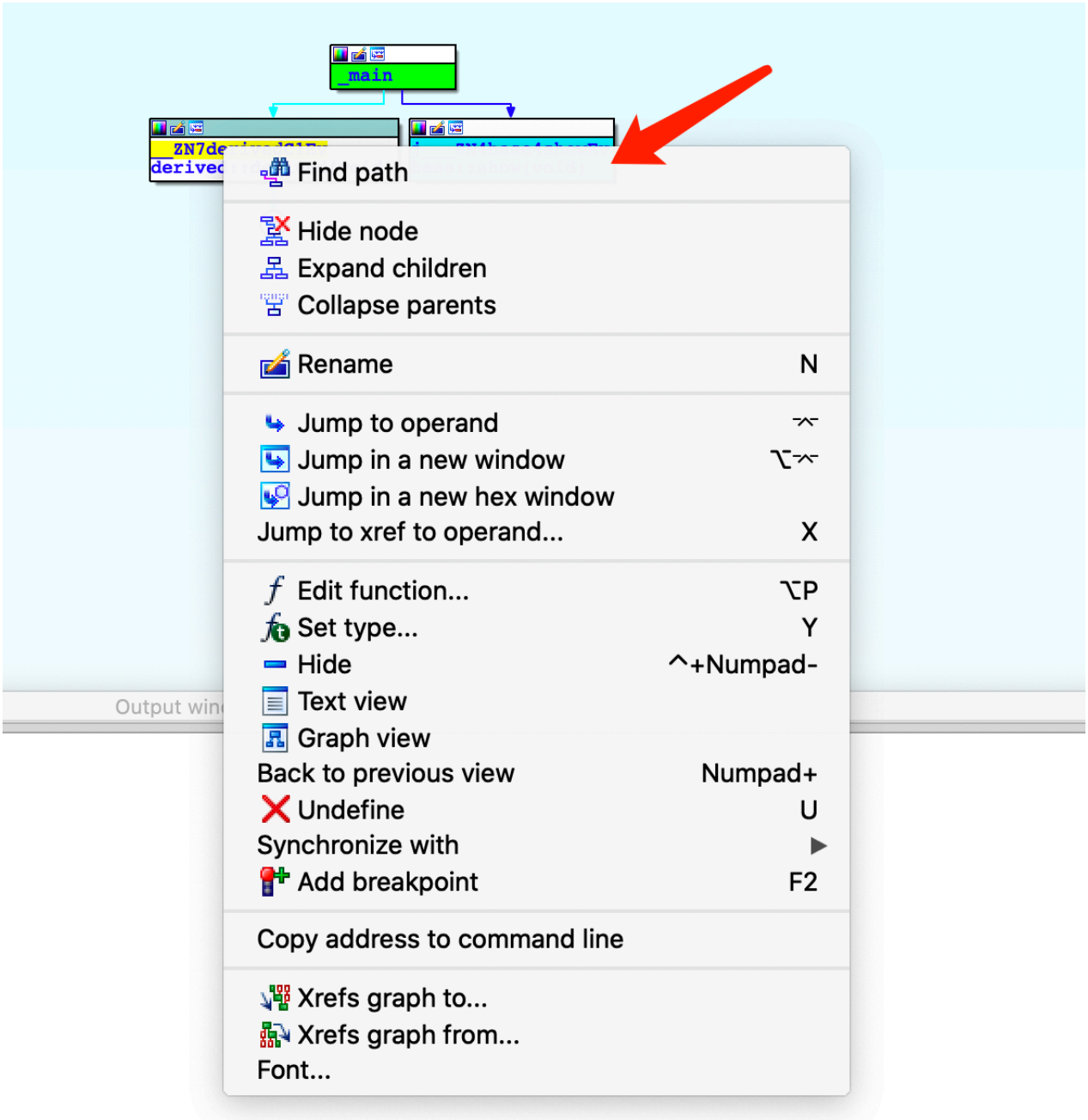
主要介绍了交叉引用，流程图的几种，演示一下：

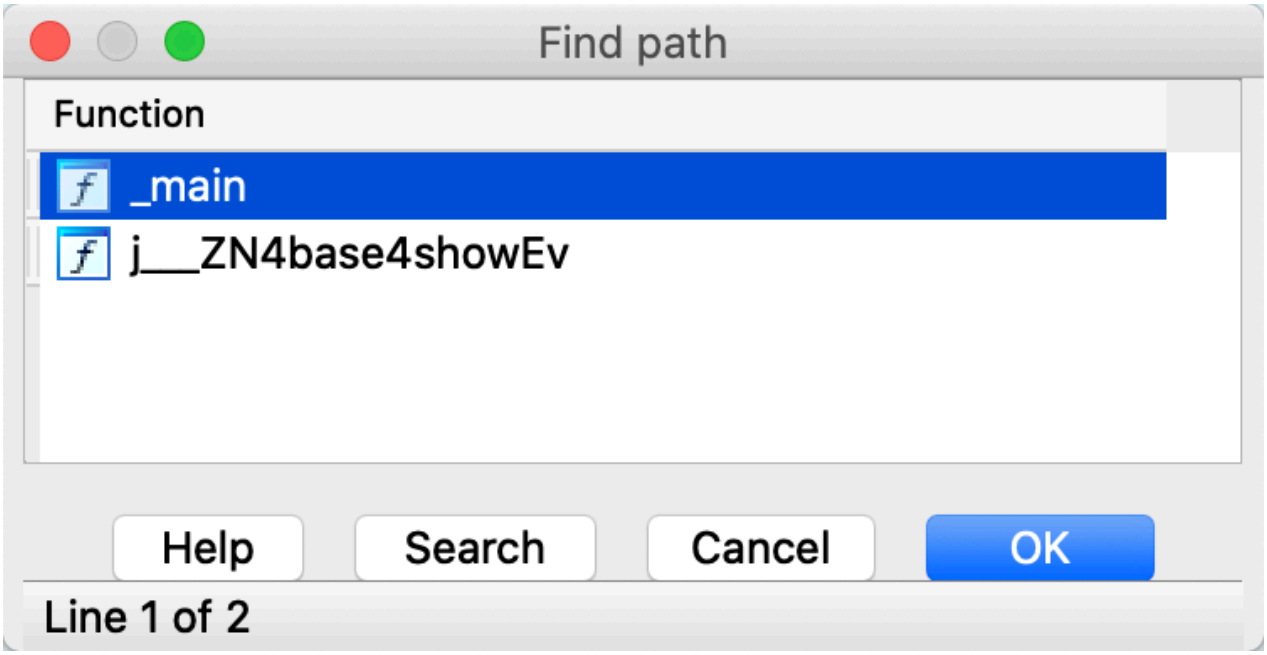



可以根据 Tree layout那五个类似的去改变流程图的一个观看方式，如果要查看一个函数是谁调用的（x/ctrl+x）：




也可以用find path去寻找：






# Reverse Engineering IDA Pro – How to do Binary Diffing - Patch Analysis

介绍了bindiff这个工具，写两个代码测试一下

```
#include "stdio.h"

int main(){
    for(int i=0;i<10;i++){
        if (i == 5){
            printf("is there! \n");
            return 0;
        }
        printf("Hello there ! \n");
    }
    return 0;
}
```

```
#include "stdio.h"

int main(){
    for(int i=0;i<10;i++){
        printf("Hello there ! \n");
    }
    return 0;
}
```

找到了相关的文章：https://www.cnblogs.com/lsdb/p/10543411.html 自行看一下，这个软件不知道为什么出问题了，什么版本都不好使打不开 就很奇怪，问了点大师傅说可能是书出了问题，让我换用diaphora

# Reverse Engineering IDA Pro- Code Coverage Measurement with Dynamo Rio & Lighthouse

这节介绍了两个插件，一个是Dynamo Rio 一个是 Lighthouse代码覆盖率浏览器

https://github.com/DynamoRIO/dynamorio/wiki

https://github.com/gaasedelen/lighthouse

感觉github上的介绍比视频里的好很多，（但是我没看懂这两个插件有什么好处和帮助）