

前言

开始叠甲：逆向工程可能较前面几个模块较难，有一定的入门门槛，本次实验选取的内容在知识储备上理论仅需C语言知识，但逆向工程仍是一门较为辛苦需要耐心的技术，希望大家量力而行，玩的开心！！

本周实验内容（可能有些难度）：瞒天过海（30%），main不是一切（30%），梦话（25%），超级游戏大师（15%），终极加密方案（扩展题）

- 瞒天过海 会讲，main不是一切 是 瞒天过海 中一个知识点的简单变式，梦话 会提其中一部分
- 终极加密方案 较困难，算作附加题(做不出来就别卷啦)(需要一些想法和第一周花时间就能做出来的扩展题有些不同)

划重点：本周题目由于难度较大，为了大家的良好体验，题目仅占50%的分数，**另外50%的分数：**将pdf中返回值优化章节下的代码自行编译为exe文件，再使用命令行 `strip xxx.exe -o xxx.exe` 去除文件的符号表，然后再在ida中对其恢复类型（可以与未去符号的文件在ida中的显示相对比，看看哪里没有恢复到位）最后在报告中提交你的恢复结果截图以及你恢复出来的结构体在ida中的显示

实验要求：实验报告：给出详细的解题过程，本次实验不仅需包括两个部分 程序是在做什么、你是怎么获得flag的（最好附上截图和代码）**对于去除了符号的文件需要ida中伪c代码（关键部分）的恢复情况（截图）**

其他：

1. 即使没获取flag，给出二进制文件的内部逻辑也能获取大部分实验分数
2. 公平起见，助教不提供除本次实验工具之外的帮助，如果给予任何人题目上的提示将会统一告知
3. 这次解题可能有一些代码量（或许可能会是整个一级信安系列最难的）量力而行，但是跟着课程内容学其中60%~85%的分数还是简单的
4. 分数与难度成反比（提倡玩的开心，而不是卷分数）

二进制文件反编译的困难

- 控制流结构的丢失 控制流结构包括条件判断（if-else）、循环（for, while）、跳转（goto, break, continue）等。在高级编程语言中，这些结构帮助程序员以直观的方式编写逻辑。然而，当源代码被编译成二进制代码时：
 - 抽象层次降低：具有语义的高级控制流结构，转换成了更底层的跳转指令如：
`jmp,jle,jge,call,ret`等

- 优化：编译器会进行控制流图（CFG）的优化，比如消除不可达代码（dead code elimination）、循环展开（loop unrolling）等。这些优化改变了原始代码的控制流结构，可能使得一些逻辑结构在二进制中不再明显或完全不可见。
- 简化：为了减少跳转和提高效率，编译器可能会简化控制流结构，比如将多个条件判断合并，或者改变循环结构以减少循环开销。

这意味着，即使能从二进制中提取基本的控制流，这些流可能与原始代码显著不同，或者难以理解和分析。

- 数据结构的丢失 高级编程语言支持丰富的数据结构，如类、结构体、联合体、枚举等，以及相关的方法和属性。在编译过程中：
 - 内存布局：所有的数据结构都会被转换为内存中的布局。例如，类和结构体会被分解为其成员变量的内存布局，而这些信息（如数据类型的语义）通常不会直接体现在二进制代码中。
 - 类型信息的丢失：编译后的代码中，类型信息通常被丢弃，变量和数据结构会被处理为简单的内存地址或基本数据类型的操作，这使得从二进制文件中恢复出完整的数据结构非常困难。

这种类型信息的丢失导致从二进制文件恢复原始数据结构变得复杂，通常需要额外的符号信息或者手动分析来辅助。

gcc编译下文件的一些重要特征




在学习恢复数据结构之前，我们先来铺垫一些gcc编译后的文件特征（如用户代码之前有关初始化的相关代码）这样在去除符号后方便大家迅速定位重要内容

__main()

```
1 void __cdecl _main()
2 {
3     if ( !initialized )
4     {
5         initialized = 1;
6         _do_global_ctors();
7     }
8 }
```

简单观察逻辑发现：这里做的是一次初始化，且仅进行一次

__main的调用

xrefs to __main			
Direct	Type	Address	Text
 Up	p	__tmainCRTStartup+143	call __main
 Up	p	main+11	call __main
 ...	o	.pdata:00000001400070CC	RUNTIME_FUNCTION <rva __main, rva algn_140001A7F, rva

Line 1 of 3

OK Cancel Search Help

查看__main的调用,我们发现在main之前存在调用

__main的第一次调用

```
v12 = (_TCHAR *)malloc(v11);
v8[v10 / 8] = v12;
v13 = v7[v10 / 8];
v10 += 8i64;
memcpy(v12, v13, v11);
}
while ( v9 != v10 );
v14 = (_TCHAR **)((char *)v8 + v9);
}
*v14 = 0i64;
argv = v8;
__main();
v15 = (const char **)envp;
v16 = argc;
**(_QWORD **)refptr___imp___initenv = envp;
result = main(v16, (const char **)argv, v15);
mainret = result;
if ( !managedapp )
    exit(result);
if ( !has_ctor )
{
    cexit();
}
```

来到 __tmainCRTStartup 函数,这部分做的是程序的初始化,这是gcc编译的文件共用的部分, (注:同时可以记住此时main的位置这在之后是很有用的, exit 前调用的那个函数)

_do_global_ctors

进入_main的_do_global_ctors

```
void __cdecl _do_global_ctors()
{
    unsigned int v0; // ecx
    void (**v1)(void); // rbx
    __int64 *v2; // rsi
    unsigned int v3; // eax

    v0 = *refptr__CTOR_LIST__;
    if ( v0 == -1 )
    {
        v3 = 0;
        do
            v0 = v3++;
        while ( refptr__CTOR_LIST__[v3] );
    }
    if ( v0 )
    {
        v1 = (void (**)(void))&refptr__CTOR_LIST__[v0];
        v2 = &refptr__CTOR_LIST__[v0 - (unsigned __int64)(v0 - 1) - 1];
        do
            (*v1--)(v2);
        while ( v1 != (void (**)(void))v2 );
    }
    atexit(_do_global_dtors);
}
```

在编译和链接过程中，特别是在使用GNU C编译器时，会涉及到特殊的表格项，如.ctors和.dtors，这些分别对应于构造和析构函数。这些函数主要用于初始化和清理全局对象。

__CTOR_LIST__和__DTOR_LIST__是链接器用来管理这些构造和析构函数的列表。在这些列表中，每个条目都是一个函数指针，通常会包含一个特殊的指针值（如0或-1），该值表示列表的开始，并可能用来表示函数指针的个数。这些构造函数在程序开始执行前被调用，用于初始化程序所需的资源或数据；析构函数则在程序结束时被调用，用于清理资源。

refptr__CTOR_LIST就是一个指向这个表项的指针

```
.rdata:000000014000A738 00 00 00 00 00 00 00 00 align 20h
.rdata:000000014000A740 public _refptr__CTOR_LIST__
.rdata:000000014000A740 50 81 00 40 01 00 00 00 _refptr__CTOR_LIST__ dq offset __CTOR_LIST__
.rdata:000000014000A740 ; DATA XREF: __d
.rdata:000000014000A748 00 00 00 00 00 00 00 00 align 10h
.rdata:000000014000A750 public _refptr__ImageBase
.rdata:000000014000A750 00 00 00 40 01 00 00 00 _refptr__ImageBase dq 140000000h
.rdata:000000014000A750 ; DATA XREF: pre
.rdata:000000014000A750 ; _refptr__ImageBase
```

双击进入发现他确实储存着__CTOR_LIST__的地址

```
38150                                     public __CTOR_LIST__
38150 FF FF FF FF FF FF FF FF          __CTOR_LIST__ dq 0FFFFFFFFFFFFFFFh ; DATA XREF: .rdata:_refptr__CTOR_LIST__↓o
38158 31 15 00 40 01 00 00 00          dq offset _GLOBAL__sub_I_encrypted_flag
38160 40 81 00 40 01 00 00 00          dq offset register_frame_ctor
38168 00 00 00 00 00 00 00 00          align 10h
38170                                     public __DTOR_LIST__
38170 FF FF FF FF FF FF FF FF          __DTOR_LIST__ dq 0FFFFFFFFFFFFFFFh
38178 00 00 00 00 00 00 00 00 00+qword_140008178 dq 11h dup(0) ; DATA XREF: .data:p_0↓o
38200 ?? ?? ?? ?? ?? ?? ?? ?? ??+dq 1C0h dup(?)
38200 ?? ?? ?? ?? ?? ?? ?? ?? ??+?+_text ends
38200 ?? ?? ?? ?? ?? ?? ?? ?? ??+
00000000 . Section 2 (virtual address 00000000)
```

每个条目都是一个函数指针，且表头包含一个特殊的指针值（如0或-1）

```
void __cdecl _do_global_ctors()
{
    unsigned int v0; // ecx
    void (**v1)(void); // rbx
    __int64 *v2; // rsi
    unsigned int v3; // eax

    v0 = *refptr__CTOR_LIST__;
    if ( v0 == -1 )
    {
        v3 = 0;
        do
            v0 = v3++;
        while ( refptr__CTOR_LIST__[v3] );
    }
    if ( v0 )
    {
        v1 = (void (**)(void))&refptr__CTOR_LIST__[v0];
        v2 = &refptr__CTOR_LIST__[v0 - (unsigned __int64)(v0 - 1) - 1];
        do
            (*v1--)(v2);
        while ( v1 != (void (**)(void))v2 );
    }
    atexit(_do_global_dtors);
}
```

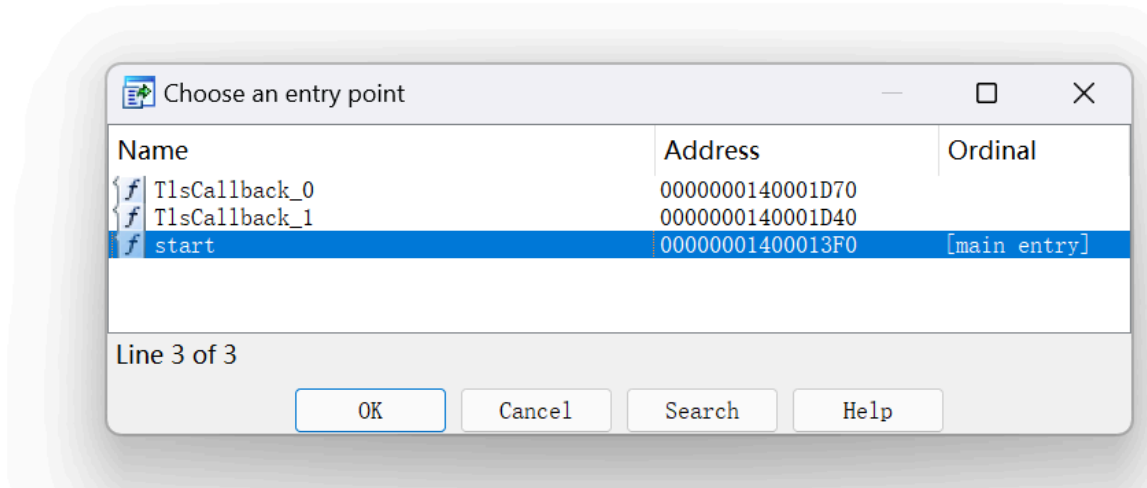
再回来看_do_global_ctors的逻辑：先获取到函数列表中最后一个函数的序号，如果不为0的话就依次调用

atexit(_do_global_dtors);：在执行完所有全局构造函数后，注册 _do_global_dtors() 函数，在程序退出时执行全局析构函数。这个函数的作用是确保在程序退出时执行全局析构函数，进行清理工作

main

我们先打开一个去除了符号表的文件

```
1 __int64 start()  
2 {  
3     unk_14000E080 = 0;  
4     return sub_140001180();  
5 }
```



首先进入的就是start(), ctrl+e 将会展示一个进入点的列表

TLSCallback: TLS 回调是 Windows 可以在程序的主入口点之前执行的函数，通常用于设置或清理特定于程序创建的每个线程的数据。它们存储在可移植可执行文件 (PE) 标头中，由 Windows 操作系统提供，以在程序的主要功能开始之前支持其他初始化任务。 TLS 回调的优点是它们在传统的程序启动之前执行，可用于各种目的，包括设置线程特定的数据或其他目的，例如反调试

“start”通常指的是程序的入口点，即操作系统加载并开始执行程序代码的地方

在pe头中我们可以看见这样的信息

▼ OptionalHeader		98h
Magic	PE64 (20Bh)	98h
MajorLinkerVersion	2	9Ah
MinorLinkerVersion	41 'y'	9Bh
SizeOfCode	30208	9Ch
SizeOfInitializedData	44544	A0h
SizeOfUninitializedData	3072	A4h
AddressOfEntryPoint	13F0h	A8h
BaseOfCode	1000h	ACH
ImageBase	140000000h	B0h
SectionAlignment	4096	B8h
FileAlignment	512	BCh
MajorOperatingSystemVersion	4	C0h
MinorOperatingSystemVersion	0	C2h
MajorImageVersion	0	C4h

- Base of Code字段提供了代码段在内存中的起始位置
- ImageBase指定了加载器应该尝试将PE文件映射到进程的虚拟地址空间中的首选地址。
- AddressOfEntryPoint是一个字段，它指定了PE文件中执行代码的起始点的相对虚拟地址（RVA）

事实上AddressOfEntryPoint+Base of Code就是ida里start的位置 总之此处是我们寻找main的起点

双击进入start()中唯一的函数则就进入先前的

```

v12 = (_TCHAR *)malloc(v11);
v8[v10 / 8] = v12;
v13 = v7[v10 / 8];
v10 += 8i64;
memcpy(v12, v13, v11);
}
while ( v9 != v10 );
v14 = (_TCHAR **)((char *)v8 + v9);
}
*v14 = 0i64;
argv = v8;
_main();
v15 = (const char **)envp;
v16 = argc;
**(_QWORD **)refptr__imp__initenv = envp;
result = main(v16, (const char **)argv, v15);
mainret = result;
if ( !managedapp )
    exit(result);
if ( !has_ctor )
{
    cexit();

```

但是符号表被去除后，我们可以根据之前提到的方法来找到main (**exit** 前调用的那个函数)

```
v0 = qword_14000E020;
v7 = (__int64)v5;
if ( v3 <= 0 )
{
    v13 = v5;
}
else
{
    v8 = v4 - 8;
    v9 = 0i64;
    do
    {
        v10 = strlen(*(const char **)(v6 + v9)) + 1;
        v11 = malloc(v10);
        *(_QWORD *)(v7 + v9) = v11;
        v12 = *(const void **)(v6 + v9);
        v9 += 8i64;
        memcpy(v11, v12, v10);
    }
    while ( v8 != v9 );
    v13 = (_QWORD *)(v7 + v8);
}
*v13 = 0i64;
qword_14000E020 = v7;
sub_140001D10();
_initenv = qword_14000E018;
result = sub_140001895((unsigned int)dword_14000E028, qword_14000E020);
dword_14000E010 = result;
if ( !dword_14000E00C )
    exit(result);
if ( !dword_14000E008 )
{
    cexit();
    return (unsigned int)dword_14000E010;
}
return result;
}
```

双击进入并把他重命名为main,ida会自动为你做好大部分工作


```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     FILE *v3; // rax
4     size_t v5; // rbx
5     char Str1[336]; // [rsp+20h] [rbp-60h] BYREF
6     char Str2[336]; // [rsp+170h] [rbp-F0h] BYREF
7     char Buffer[72]; // [rsp+2C0h] [rbp+240h] BYREF
8     void *Block; // [rsp+308h] [rbp+288h] BYREF
9     int v10[128]; // [rsp+310h] [rbp+290h] BYREF
10    char Str[1016]; // [rsp+510h] [rbp+490h] BYREF
11    __int64 v12; // [rsp+908h] [rbp+888h]
12    int v13; // [rsp+914h] [rbp+894h]
13    __int64 v14; // [rsp+918h] [rbp+898h]
14    int v15; // [rsp+924h] [rbp+8A4h]
15    int j; // [rsp+928h] [rbp+8A8h]
16    int i; // [rsp+92Ch] [rbp+8ACh]
17
18    sub_140001D10(argc, argv, envp);
19    strcpy(
20        Str,
21        "70185432033430148111928453092023358633646714916212206734449928768603810714566586!
22        "24089932434142670333728435329238195539680756468901496178333876407750676411078287!
23        "26862554629837665947101571561051255313411733079815099604948586393168077529935689!
24        "36454876173316178906627962472922526749826519669350043905938359550526689497491938!
25        "40829495556820131928805627596996251147917307642228214947704622993238550441180152!
26        "66851036511713471551584450251216834888010076310365254873024293755802023666966501!
27        "94572520389872483479011324376934969346953336024558729253230965054225809355991927!
28        "62434524765130289453002513055952267707625667979138758556794493013736627694726018!
29        "424739987193251551795478339353026626990853827058193213003114182062628603");
30    sub_140001450("sleep talking: %s\n", Str);
31    memset(v10, 0, sizeof(v10));
32    v15 = strlen(Str);
33    for ( i = 0; i < v15; ++i )
34    {
35        if ( Str[i] >= 0 )
36            ++v10[Str[i]];
37    }
38    Block = 0i64;
39    for ( j = 0; j <= 127; ++j )
40    {
41        if ( v10[j] > 0 )
42        {
43            v12 = sub_1400014A4((unsigned int)(char)j);
44            sub_1400014EF(&Block, v12, (unsigned int)v10[j]);
45        }
46    }

```

数据结构信息的恢复

因为我也不知道非常学术的做法只能根据自己的的一些经验介绍一下我觉得效果还不错的思考方法

返回值优化 RVO

返回值优化（RVO：Return Value Optimization）是编译器的一种编译优化，通过该优化可以减少函数返回时产生临时对象，从而消除部分拷贝或移动操作，进而提高代码性能。

我们用以下代码来探究一下(与实验报告中相关的源码)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// 定义一个结构体, 包含整数和指针
typedef struct {
    int id;
    char name[12];
} Person;

// 定义第二个结构体, 包含指向第一个结构体的指针
typedef struct {
    Person *person;
    int role;
} Employee;

Person create_person_value(int id, char *name) {
    Person p;
    p.id = id;
    strcpy(p.name, name);
    return p;
}

Person* create_person_pointer(int id, char *name) {
    Person *p = malloc(sizeof(Person));
    p->id = id;
    strcpy(p->name, name);
    return p;
}

void display_person(Person p) {
    printf("Person ID: %d, Name: %s\n", p.id, p.name);
}

Employee* create_employee_pointer(Person *p, int role) {
    Employee *e = malloc(sizeof(Employee));
    e->person = malloc(sizeof(Person));
    *(e->person) = *p; // 复制person结构体
    e->role = role;
    return e;
}

```

```

Employee create_employee_value(Person p, int role) {
    Employee e;
    e.person = malloc(sizeof(Person));
    *(e.person) = p; // 复制person结构体
    e.role = role;
    return e;
}

void free_employee(Employee *e) {
    free(e->person);
}

void display_employee(Employee *e) {
    printf(" ID: %d, Name: %s, Role: %d\n", e->person->id, e->person->name, e->role);
}

int main() {
    Person p1 = create_person_value(1, "Alice");
    display_person(p1);

    Person *p2 = create_person_pointer(2, "Bob");
    display_person(*p2);

    Employee e1 = create_employee_value(p1, 1);
    display_employee(&e1);
    free_employee(&e1);

    Employee *e2 = create_employee_pointer(p2, 2);
    display_employee(e2);
    free_employee(e2);

    return 0;
}

```

我们先直接来看看create_employee_value函数（源代码的其他部分用于后续对结构体的恢复）

```

lea     rcx, [rbp+e1]    ; retstr
mov     rax, qword ptr [rbp+p1.id]
mov     rdx, qword ptr [rbp+p1.name+4]
mov     qword ptr [rbp+p.id], rax
mov     qword ptr [rbp+p.name+4], rdx
lea     rax, [rbp+p]
mov     r8d, 1           ; role
mov     rdx, rax         ; p
call    create_employee_value
lea     rax, [rbp+e1]

```

从汇编上来看：首先e1直接作为create_employee_value的参数被传入,p作为函数的参数按照预期是通过值传递，同时函数返回后直接对rax做填充（说明无返回）

```

p = p1;
create_employee_value(&e1, &p, 1);

```

从伪c代码上来看也很清楚，即原本应该返回的结构体在调用者函数的栈帧上留出预留空间，并将指向预留空间的指针传给被调用函数，从而减少了对结构体拷贝的性能开销

结构体恢复的思考流程

结构体的恢复可以分为这样的一个分析判断过程借用了一些编译原理的名词,试图显示的专业些

结构体语法分析：恢复出结构体的内存布局

结构体语义分析: 恢复结构体各部分结构的类型信息

将之前源代码编译的二进制文件strip

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     __int64 v3; // rdx
4     __int64 v5; // [rsp+20h] [rbp-40h] BYREF
5     __int64 v6; // [rsp+28h] [rbp-38h]
6     char v7[16]; // [rsp+30h] [rbp-30h] BYREF
7     __int64 v8; // [rsp+40h] [rbp-20h] BYREF
8     __int64 v9; // [rsp+48h] [rbp-18h]
9     __int64 v10; // [rsp+50h] [rbp-10h]
10    __int64 *v11; // [rsp+58h] [rbp-8h]
11
12    _main(argc, argv, envp);
13    sub_1400014A4(&v8, 1i64, "Alice");
14    v5 = v8;
15    v6 = v9;
16    sub_140001532(&v5);
17    v11 = (__int64 *)sub_1400014EE(2i64, "Bob");
18    v3 = v11[1];
19    v5 = *v11;
20    v6 = v3;
21    sub_140001532(&v5);
22    v5 = v8;
23    v6 = v9;
24    sub_1400015C0(v7, &v5, 1i64);
25    sub_14000163C(v7);
26    sub_14000161A(v7);
27    v10 = sub_140001562(v11, 2i64);
28    sub_14000163C(v10);
29    sub_14000161A(v10);
30    return 0;
31 }

```

我们一个个分析先从sub_1400014A4下手

首先他先接受了一个v8变量的地址（对应的是rbp-20h），另外两个类型很标准是好事（i64后缀表示ida认为他传递的是64位的，但实际可能不是要具体分析）但v8是不是__int64位这很难说，当我们试图获取关于他的更多信息时我们可以看看哪里对他进行了使用，所以我们先进去看看

```

1 __QWORD *__fastcall sub_1400014A4(__QWORD *a1, int a2, const char *a3)
2 {
3     __int64 v3; // rdx
4     __int64 v5[2]; // [rsp+20h] [rbp-10h] BYREF
5
6     LODWORD(v5[0]) = a2;
7     strcpy((char *)v5 + 4, a3);
8     v3 = v5[1];
9     *a1 = v5[0];
10    a1[1] = v3;
11    return a1;
12 }

```

- `_QWORD`对应的是64位，v8与a1相关联

我们看看与a1相关的地方：`*a1, a1[1]`，这就说明v8肯定不是简简单单的`__int64`因为我们对这个指针取值的范围已经超过了64位，现在我们先稍微对a1的结构做一个雏形出来

```

00000000
00000000 struc_1 struc ; (sizeof=0x8, mappedto_18)
00000000 field_0 dd ?
00000004 field_4 dd ?
00000008 struc_1 ends
00000008

```

然后再去改改a1的定义

```

1 struc_1 *__fastcall sub_1400014A4(struc_1 *a1, int a2, const char *a3)
2 {
3     __int64 v3; // rdx
4     __int64 v5[2]; // [rsp+20h] [rbp-10h] BYREF
5
6     LODWORD(v5[0]) = a2;
7     strcpy((char *)v5 + 4, a3);
8     v3 = v5[1];
9     a1->field_0 = v5[0];
10    a1->field_8 = v3;
11    return a1;
12 }

```

效果一般般，我们继续试图恢复语义，与a1各部分相关联的是v5，那我们继续对v5重复上述分析过程：

- v5与a1交互
- v5与a2, a3交互

由于a2,a3是确定类型可信度更高所以选择从v5与a2,a3的交互入手 v5分为两部分被引用:
LODWORD(v5[0]),(char*)v5, 同时结合ida之前给v5的定义是使用了16字节, 结合栈图确实应该是16字节大小

```
-000000000000000013 db ? ; undefined
-000000000000000012 db ? ; undefined
-000000000000000011 db ? ; undefined
-000000000000000010 var_10 dq ?
-000000000000000008 var_8 dq ?
+000000000000000000 s db 8 dup(?)
+000000000000000008 r db 8 dup(?)
+000000000000000010 arg_0 dq ?
+000000000000000018 arg_8 dd ?
+00000000000000001C db ? ; undefined
+00000000000000001D db ? ; undefined
+00000000000000001E db ? ; undefined
+00000000000000001F db ? ; undefined
+000000000000000020 Source dq ?
+000000000000000028
+000000000000000028 ; end of stack variables
```

我们给出这样的定义

```
00000000 struc_1 struc ; (sizeof=0x10, mappedto_18)
00000000 field_0 dq ?
00000008 field_8 dq ?
00000010 struc_1 ends
00000010
✓00000000 ; -----
00000000
00000000 struc_2 struc ; (sizeof=0x10, mappedto_19)
00000000 ; XREF: sub_1400014A4/r
00000000 field_0 dd ? ; XREF: sub_1400014A4+16/w
00000000 ; sub_1400014A4+31/r
00000004 field_4 db 12 dup(?) ; XREF: sub_1400014A4+35/r
00000010 struc_2 ends
```

把v5重定义


```

1 struct_1 *__fastcall sub_1400014A4(struct_1 *a1, int a2, const char *a3)
2 {
3     __int64 v3; // rdx
4     struct_2 v5; // [rsp+20h] [rbp-10h] BYREF
5
6     v5.field_0 = a2;
7     strcpy(v5.field_4, a3);
8     v3 = *(_QWORD *)&v5.field_4[4];
9     a1->field_0 = *(_QWORD *)&v5.field_0;
10    a1->field_8 = v3;
11    return a1;
12 }

```

现在恢复v5结构体各部分类型信息

- field_0:int
- field_4:char[12]

好的我们现在已经完全完成了v5的结构恢复，现在我们重新回到a1上 这里发现似乎a1结构体各部分在代码里的使用混杂了两种类型：field_0中既有int又有char[0:4]这时我们知道前面的分析出现了一点误差，这里由于编译器做了优化用了两次总线操作来完成a1的赋值相较于拘泥于结构体的成分分布的三次总线（int,char[0:4],char[4:12]）操作减少了性能开销

最后效果图

```

1 struct_2 *__fastcall sub_1400014A4(struct_2 *a1, int a2, const char *a3)
2 {
3     __int64 v3; // rdx
4     struct_2 v5; // [rsp+20h] [rbp-10h] BYREF
5
6     v5.int = a2;
7     strcpy(v5.char_array, a3);
8     v3 = *(_QWORD *)&v5.char_array[4];
9     *(_QWORD *)&a1->int = *(_QWORD *)&v5.int;
10    *(_QWORD *)&a1->char_array[4] = v3;
11    return a1;
12 }

```

当然仅仅这个函数的信息只够我们把语义还原到这个地步，比如int代表的是id，char[12]代表的名字，需要我们进一步的分析

```

typedef struct {
    int id;
    char name[12];
} Person;

```

与源代码对比我们是正确的

在ida中重建结构体的操作

编写代码lab3.c:

```

#include <stdio.h>
#include <string.h>

typedef struct {
    char name[32];
    int name_len;
    char pass[32];
    int pass_len;
} Account;

int check(Account *a) {
    char enc[] = {19, 1, 27, 12, 28, 18, 1, 50, 12, 0, 6, 13, 3, 12, 11, 19, 13, 3, 14, 79};
    if (strcmp(a->name, "admin") != 0) return 0;
    if (a->pass_len != 20) return 0;
    for (int i = 0; i < a->pass_len; i++) {
        if (enc[i] != (a->pass[i] ^ a->name[i % a->name_len])) return 0;
    }

    return 1;
}

int main() {
    Account a;

    puts("Username:");
    scanf("%s", a.name);
    a.name_len = strlen(a.name);
    puts("Password:");
    scanf("%s", a.pass);
    a.pass_len = strlen(a.pass);

    if (check(&a)) {
        puts("Correct!");
    } else {
        puts("Wrong!");
    }

    return 0;
}

```

编译生成 lab3.exe, 拖入 IDA, F5:

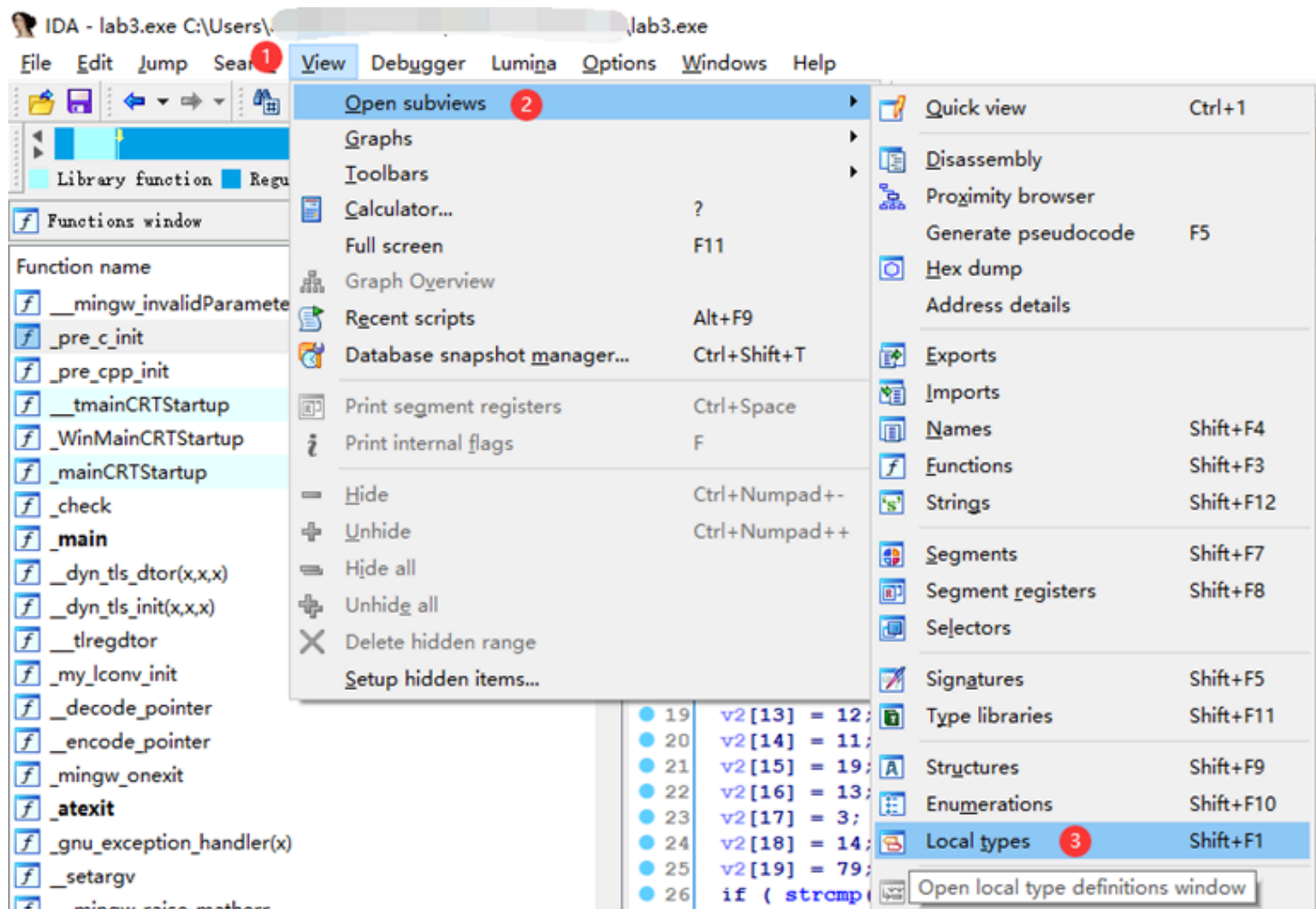
```
IDA View-A Pseudocode-A Hex View-1
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char Str[32]; // [esp+18h] [ebp-48h] BYREF
4     size_t v5; // [esp+38h] [ebp-28h]
5     char v6[32]; // [esp+3Ch] [ebp-24h] BYREF
6     size_t v7; // [esp+5Ch] [ebp-4h]
7
8     __main();
9     puts("Username:");
10    scanf("%s", Str);
11    v5 = strlen(Str);
12    puts("Password:");
13    scanf("%s", v6);
14    v7 = strlen(v6);
15    if ( check(Str) )
16        puts("Correct!");
17    else
18        puts("Wrong!");
19    return 0;
20 }
```

虽然 main 函数中并没有识别到我们自定义的结构体，但是生成的伪代码还算正常，还能看。下面看一看不正常的，双击进入 check 函数：

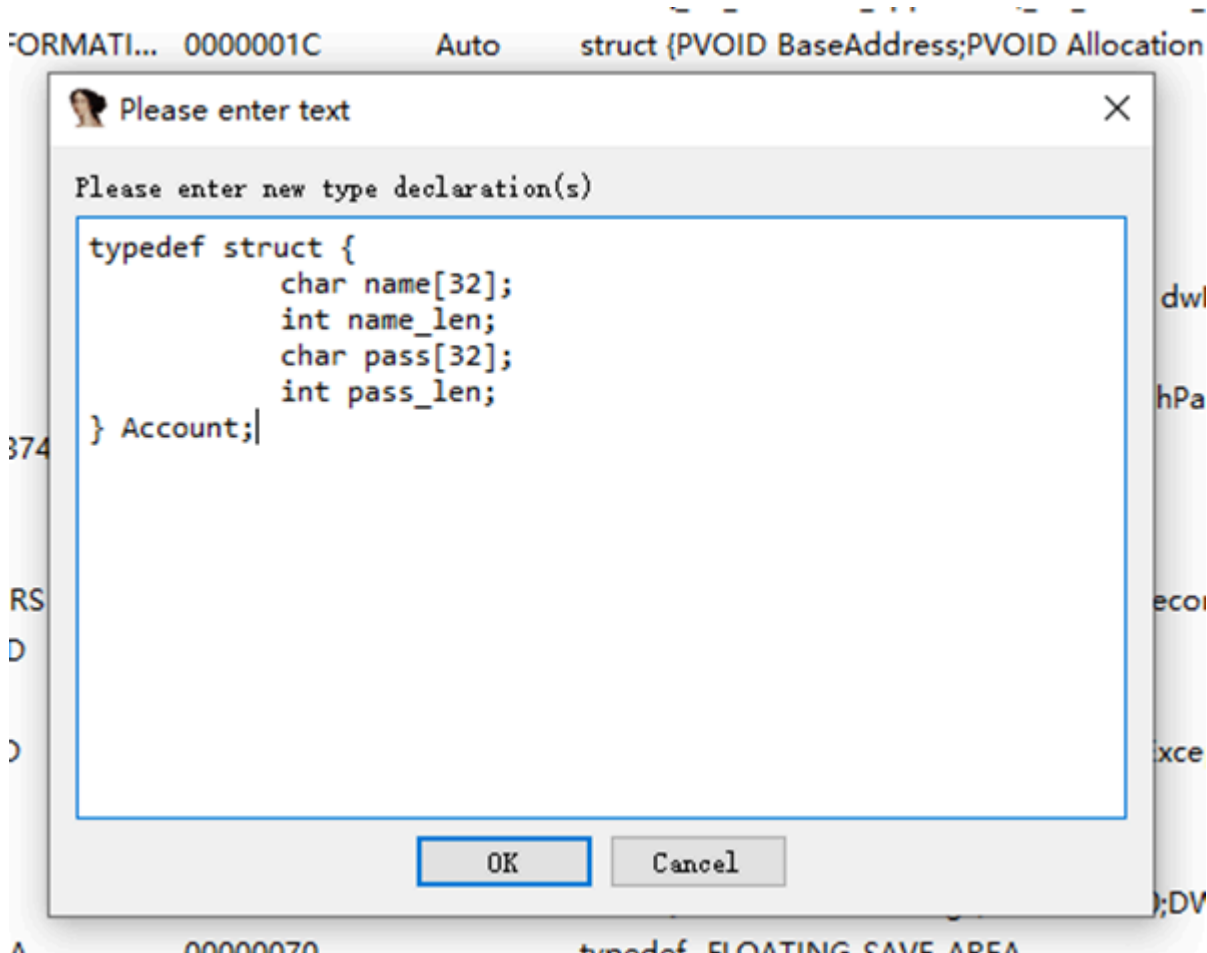
```
IDA View-A Pseudocode-A Hex View-1 Structures Enums
1 int __cdecl check(char *Str1)
2 {
3     char v2[20]; // [esp+18h] [ebp-20h]
4     int i; // [esp+2Ch] [ebp-Ch]
5
6     v2[0] = 19;
7     v2[1] = 1;
8     v2[2] = 27;
9     v2[3] = 12;
10    v2[4] = 28;
11    v2[5] = 18;
12    v2[6] = 1;
13    v2[7] = 50;
14    v2[8] = 12;
15    v2[9] = 0;
16    v2[10] = 6;
17    v2[11] = 13;
18    v2[12] = 3;
19    v2[13] = 12;
20    v2[14] = 11;
21    v2[15] = 19;
22    v2[16] = 13;
23    v2[17] = 3;
24    v2[18] = 14;
25    v2[19] = 79;
26    if ( strcmp(Str1, "admin") )
27        return 0;
28    if ( *((_DWORD *)Str1 + 17) != 20 )
29        return 0;
30    for ( i = 0; *((_DWORD *)Str1 + 17) > i; ++i )
31    {
32        if ( v2[i] != ((unsigned __int8)Str1[i + 36] ^ (unsigned __int8)Str1[i % *((_DWORD *)Str1 + 8)]) )
33            return 0;
34    }
35    return 1;
36 }
```

其中, _DWORD 相当于 unsigned int, 表示4字节数据; unsigned __int8相当于 unsigned char, 表示 1 字节数据, 这些括号将类型包含在内的就是强制类型转换, 因为我们自己定义的 Account 数据类型中既有 char 又有 int, 这两者占用大小是不同的, IDA 无法直接识别到结构体的形式, 只能给参数一种类型, 要么 char*, 要么 int*, 这就导致不是该类型的成员必定需要强制类型转换。要解决这个问题, 就需要将check函数的参数类型改为 Account*, 但是 IDA 是不知道 Account 这个结构体定义的, 需要在 IDA 中添加结构体定义, 可以使用两种方式。

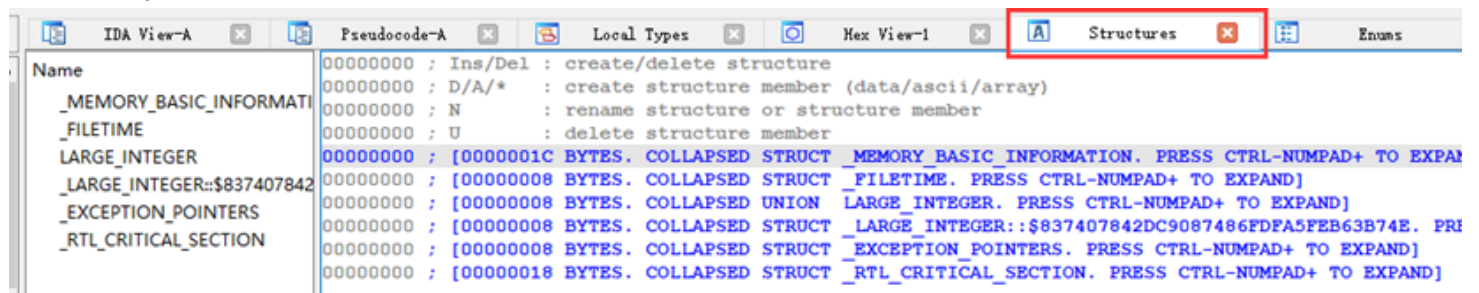
第一种方式, 比较方便的, 我们已经有了 Account 结构体的 C 语言定义, 可以直接复制粘贴。在 IDA 的菜单栏选择 View->Open subviews->Local types:



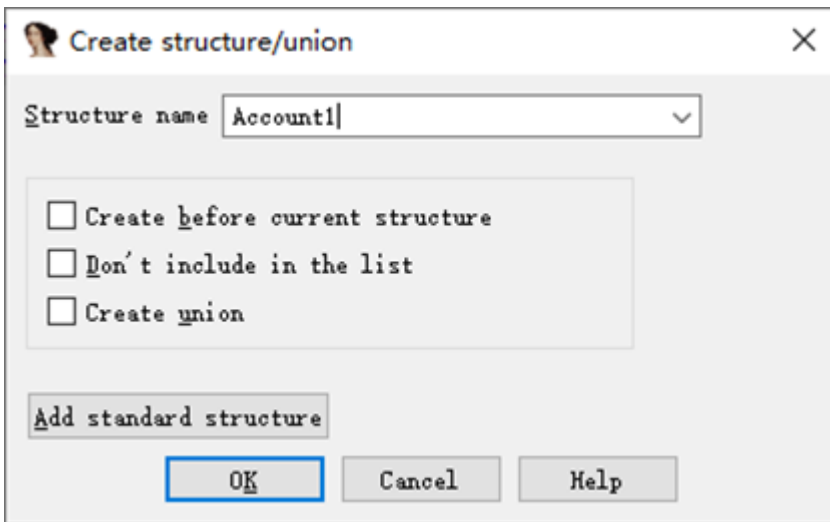
打开界面后按 insert 添加结构体定义 (键盘上没有 insert 按键可以鼠标右键->Insert...), 将我们的 Account 类型定义复制粘贴, 确定就行了:



第二种方式（由于第一种方式已经定义过了 Account 这个名字，这里要使用不同的名称，如 Account1），打开 Structures 窗口：



按 insert 或者右键->Add struct type...，在弹窗中设置结构体的名字 Account1：



确认后，界面就出现了 Account1 结构体的空定义：

```
00000000
00000000 Account1      struc ; (sizeof=0x0, mappedto_133)
00000000 Account1      ends
00000000
```

在 Account1 的最后一行按 d，添加成员：

```
00000000 Account1      struc ; (sizeof=0x1, mappedto_133)
00000000 field_0      db ?
00000001 Account1      ends
00000001
```

在 field_0 处按 y 修改类型为 char

32

，按 n 修改名称为 name，成员第一项就设置好了：

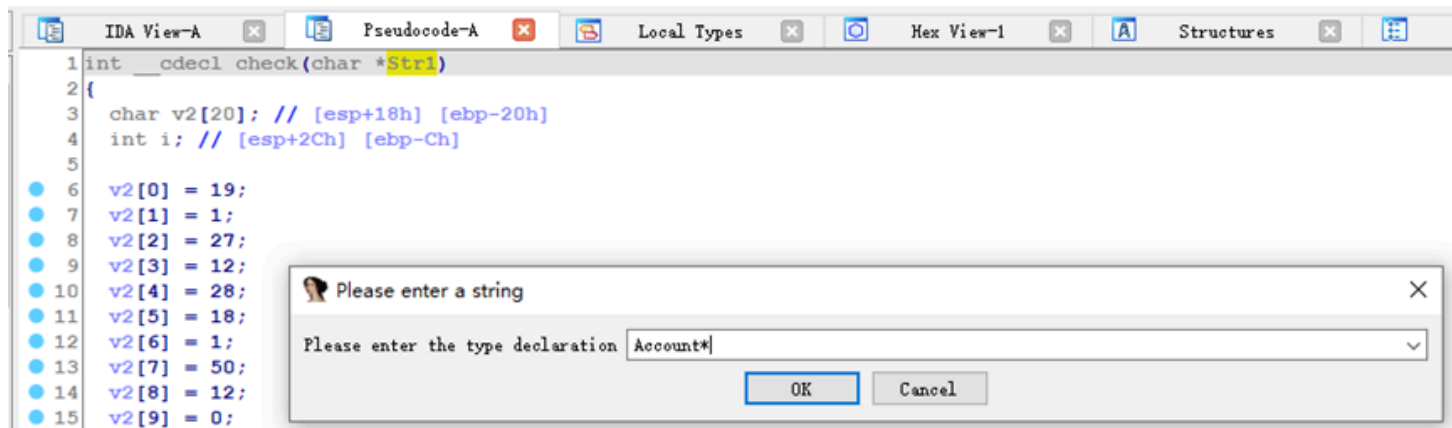
```
00000000 Account1      struc ; (sizeof=0x20, mappedto_133)
00000000 name         db 32 dup(?)
00000020 Account1      ends
00000020
```

然后依次在 Account ends 那行按 d 添加成员（注意：是 Account1 ends 那行，请不要在已经定义好的成员那一行按 d，否则会修改该成员的类型），修改为正确的类型和名称，最终结构如下：

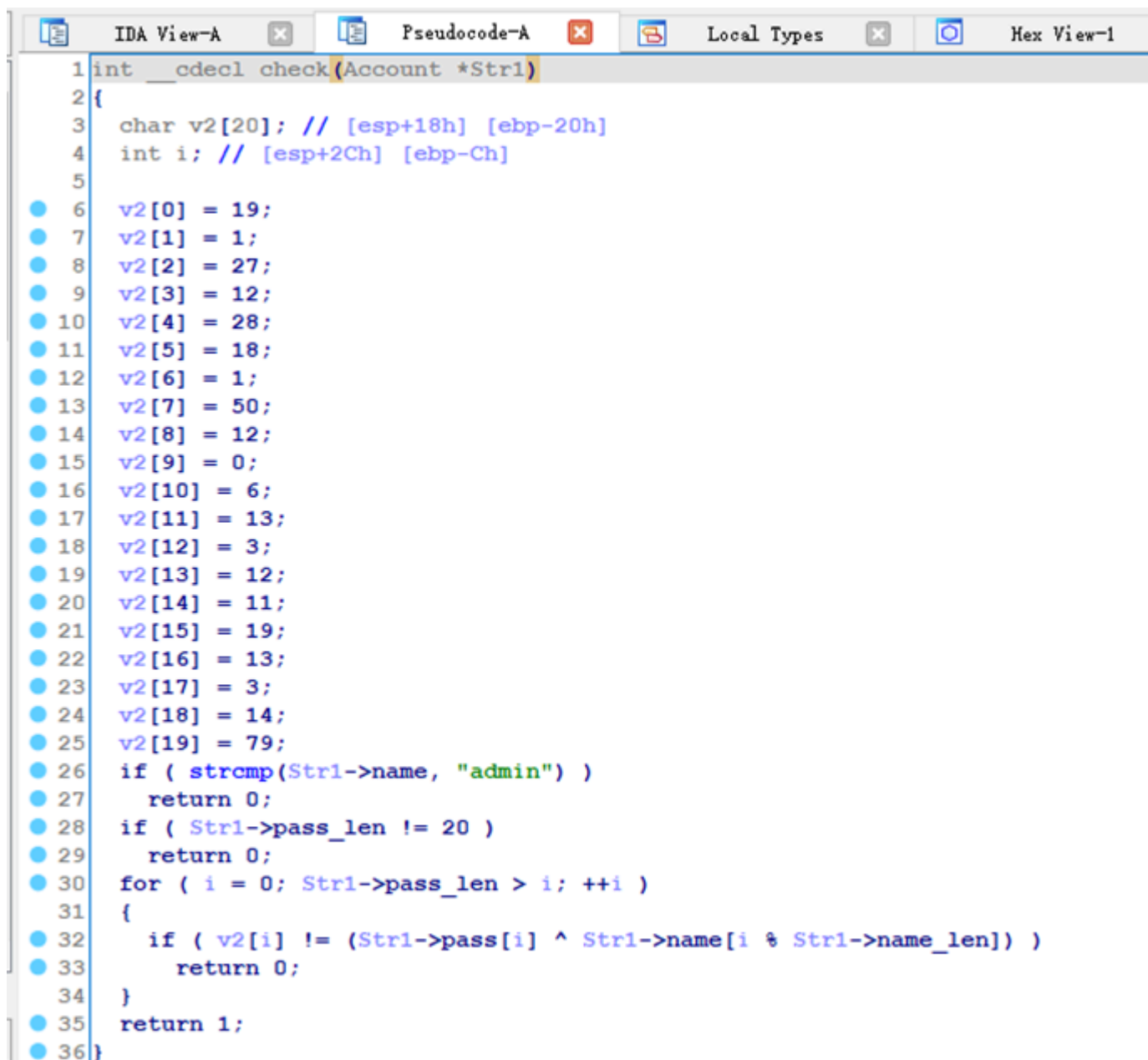
```
00000000 Account1      struc ; (sizeof=0x48, mappedto_133)
00000000 name         db 32 dup(?)
00000020 name_len      dd ?
00000024 pass         db 32 dup(?)
00000044 pass_len      dd ?
00000048 Account1      ends
00000048
```

这样结构体就定义好了（检查一下 Account1 ends 这一行左边的数字，是不是 48，其含义是 Account1 结构体占用的空间是 0x48 字节）。

两种方式均可创建结构体定义。然后回到伪代码窗口，按y将参数类型改为 Account*（也可以使用 Account1*，下面仅以 Account 说明相关的操作）：



修改后的结果：



强制类型转换没了，生成的伪代码与源代码基本一致。再按 ESC 返回到 main，IDA 自动识别了 Account 结构体类型，结果如下（如果没有自动修改，手动按 y 修改即可）：



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     Account Str; // [esp+18h] [ebp-48h] BYREF
4
5     __main();
6     puts("Username:");
7     scanf("%s", &Str);
8     Str.name_len = strlen(Str.name);
9     puts("Password:");
10    scanf("%s", Str.pass);
11    Str.pass_len = strlen(Str.pass);
12    if ( check(&Str) )
13        puts("Correct!");
14    else
15        puts("Wrong!");
16    return 0;
17 }
```

这样程序的逻辑就都恢复出来了

一些其他说明

题目在命令行显示乱码时

可以使用如下命令（只对当前命令行生效）：

- 设置当前代码页为 UTF-8：chcp 65001
- 设置当前代码页为 GBK：chcp 936

ida中文编码不显示

对于中文编码，ida不能自动识别此时可以

选中字符符号->Alt+A->Currently->右键->ins(电脑上的快捷键)->输入gbk->再选择即可



String literal at 2047



Currently: (no string literal)



Encodings



Encoding name

UTF-8

UTF-16LE

UTF-32LE

OK

Ca

Manage defaults

OK

Insert...

Ins

Copy

Insert C

Copy all

Ctrl+Shift+Ins

Quick filter

Ctrl+F

Modify filters...

Ctrl+Shift+F

Hide column

Columns...

Font...

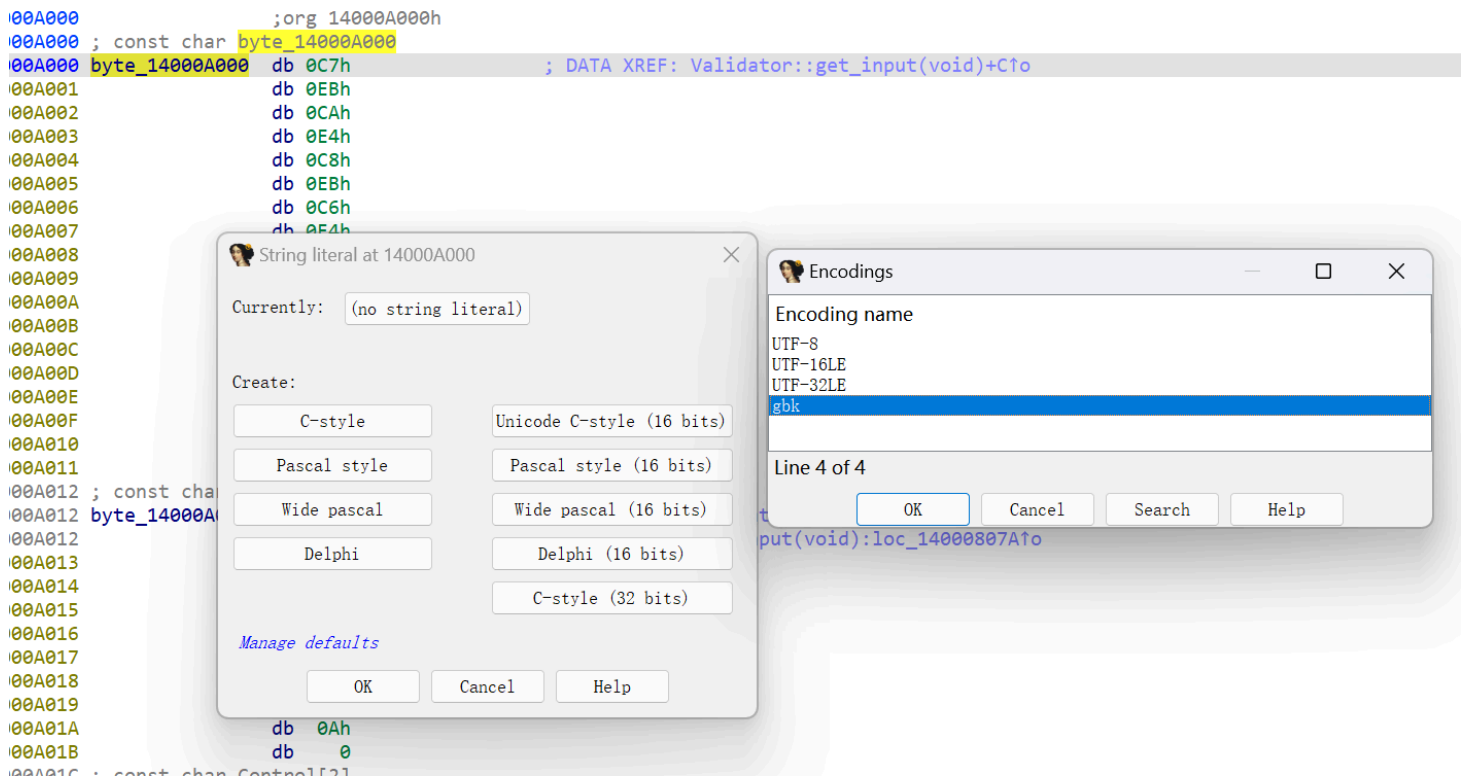
; co
aFla

:get_
_input()

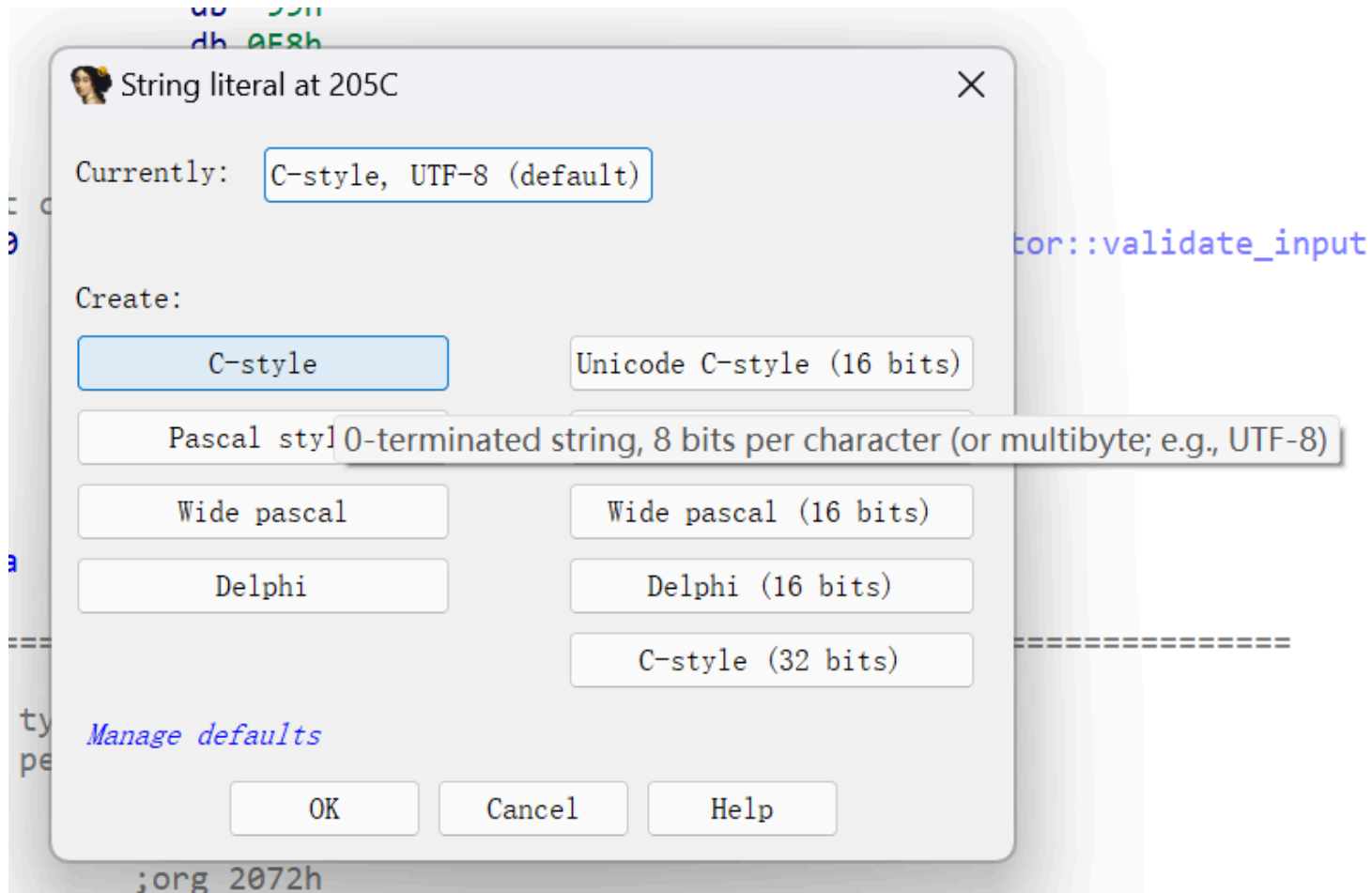
db 0AFh
db 0AFh
db 0

; const char aFlag_0[4]
aFlag_0 db 'flag'
" "

, DATA_ARE1, validator::vali

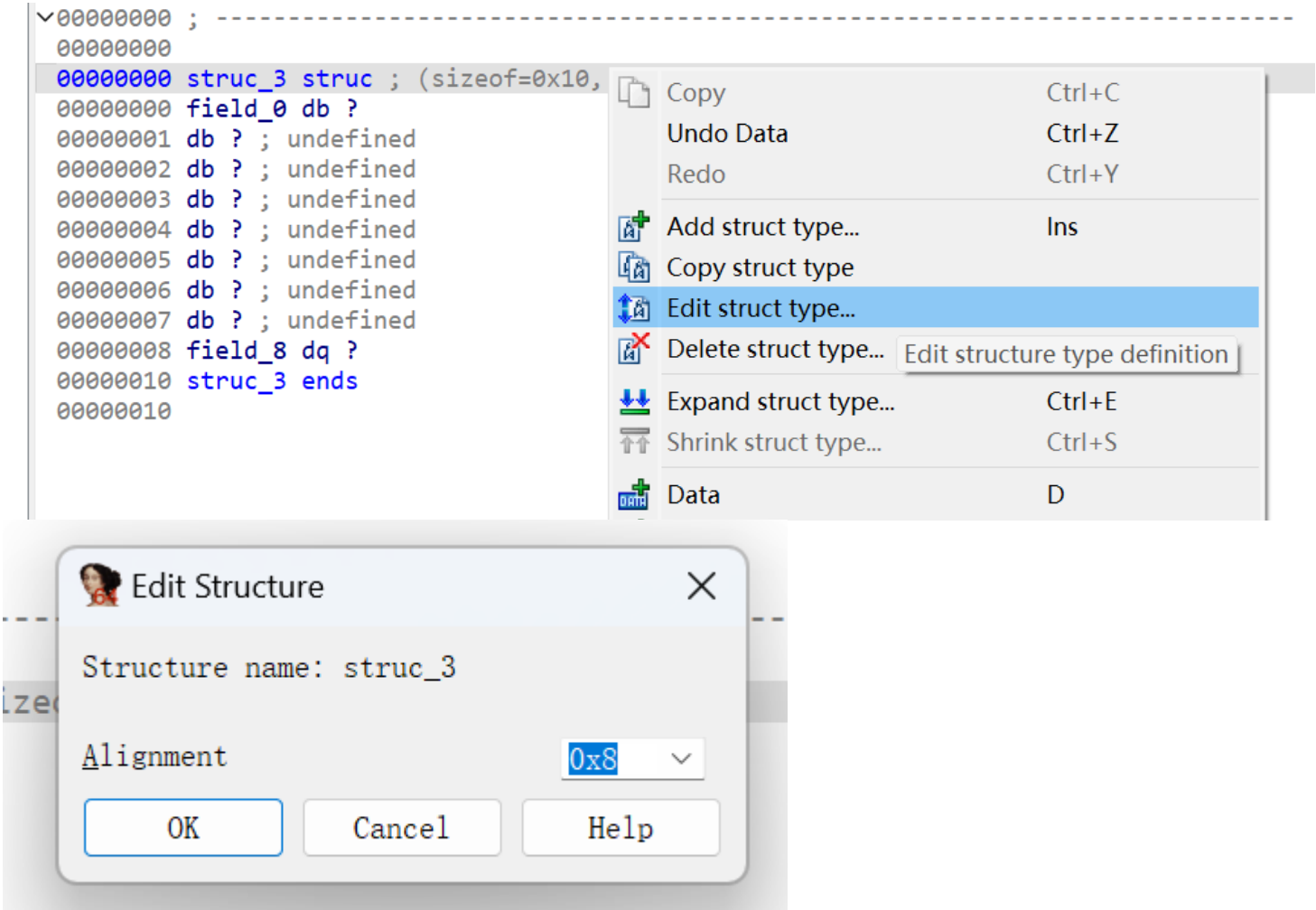


ida8.3中更方便的方式是点击C-style，他会自动恢复，如果不能请还是按照上述操作



结构体对齐

有些时候结构体为了访问效率可能会浪费一些内存用于内存对齐，在ida中设置内存对齐的方式如下



此次讲义颜浩翔编写，参考杨锦东学长编写的逆向入门