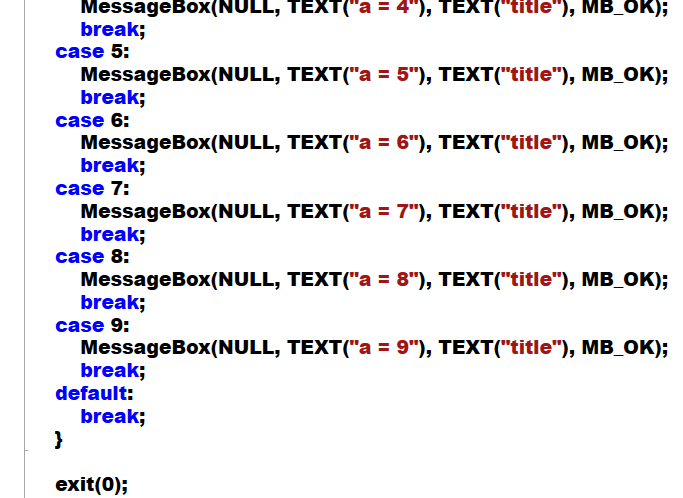
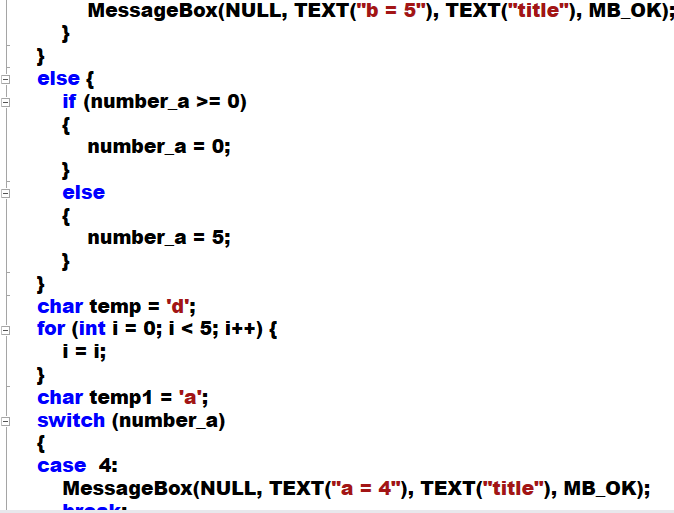
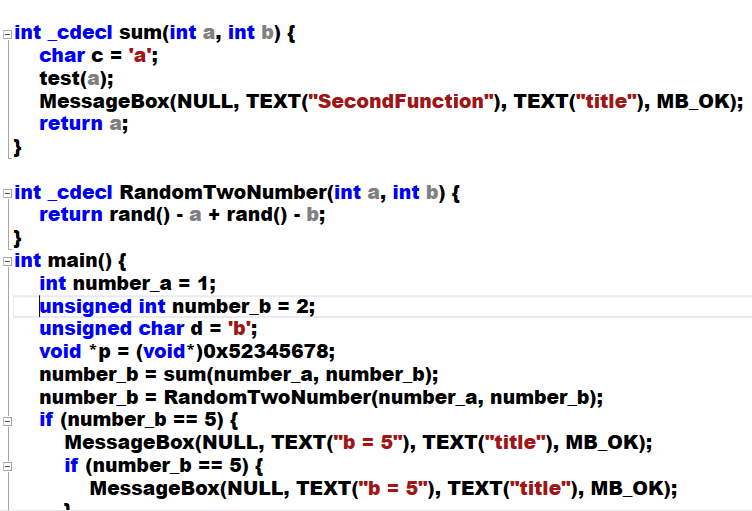
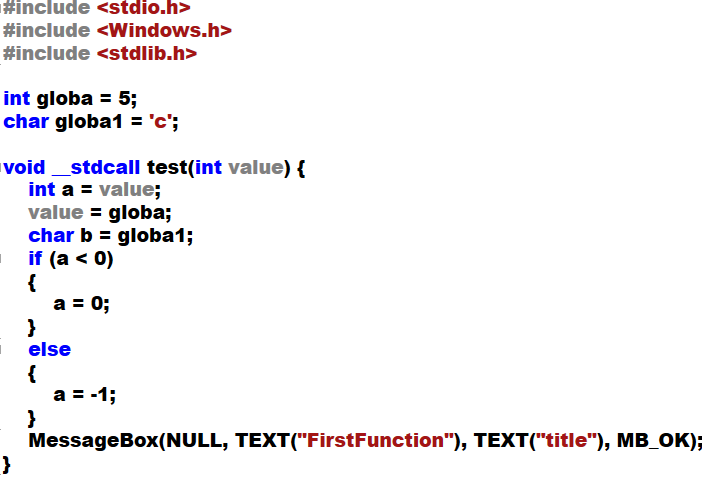
201611123021 李沅城 2016信息安全 逆向工程大作业实验报告

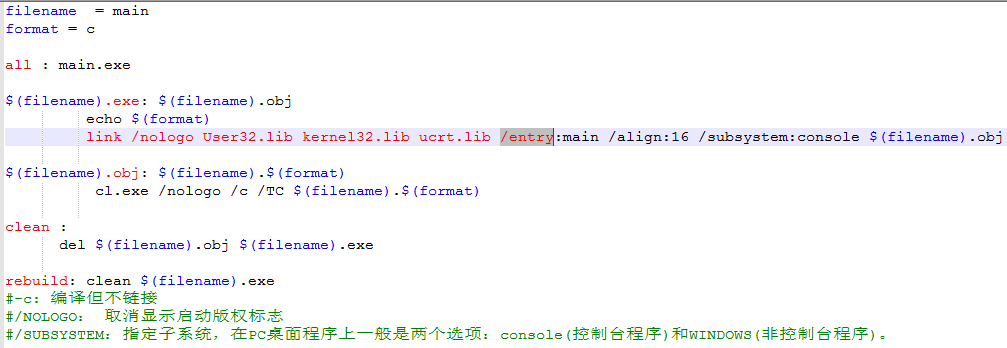
目标：两个以上的函数，子函数必须有参数，一定要有if,for,switch三种流程控制语句，有全局变量和局部变量，必须调用系统函数

1. 所要分析的代码

含有if、switch、for。有三个子函数，每个函数都有相应的参数，main调用了两个函数，一个子函数又调用了另外一个子函数，除此之外，调用了messagebox、rand、exit的系统函数。有char,int,unsigned int,unsigned char,void \* 的类型的局部变量和int、char类型的全局变量。



1. 将.c代码使用makefile转化为.exe文件。去除冗余代码段，压缩对齐长度，尽可能将.exe压缩成较小的可执行文件进行分析，最后可以压缩成2k左右的.exe文件。在优化exe的大小以便去分析时，发现有些优化参数会导致在python里面的分析出来的代码出现错误或是缺少的问题，比如在生成obj的命令中有/02时会导致代码缺少。

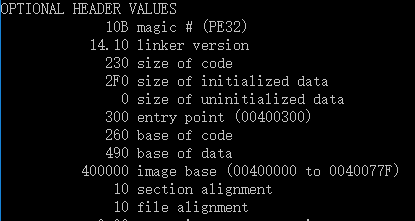


1. 利用python对.exe文件进行分析

**1、反汇编**

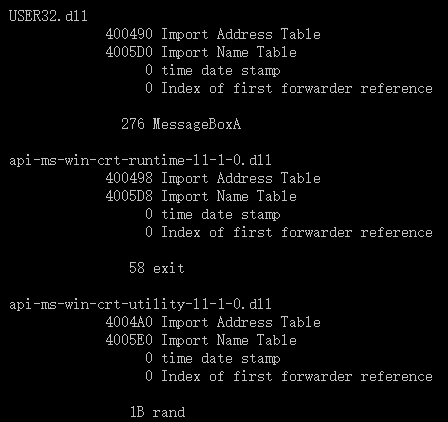
一开始我打算使用的是Pandas去进行分析，它是python里面经常用于数据分析的一个库，但在实际操作的过程中发现该库不适合用来做逆向工程，因为它需要先使用VS的dumpbin功能将.exe文件变为反汇编代码，然后存为txt或者是excel格式，且存在对齐错误的问题，而且不能分析PE文件头内的信息，方法实在是过于累赘。所以，后来改用了pefile和capstone两个python的库进行了分析。Pefile提供了解析PE文件头的功能，而capstone提供了反汇编代码的功能。这样可以在python里面实现dumpbin的功能。与Pandas相比要方便了很多。

**2、提取PE文件头中所要用到的信息**

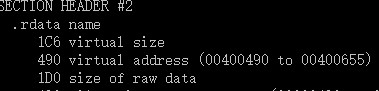


获取程序基地址image base、主函数入口entry point、导入表中的函数表、.text段数据、.data段数据。

导入函数表



获取每个段的大小和其实地址



接下来借助capstone就可以反汇编.exe的反汇编代码

**3、获取外部函数名**

在反汇编代码中寻找call指令和jmp指令，查看它们跳转的地址，如果不在代码段，则说明是外部函数，则在外部函数中寻找即可（一开始处理时已经获取了外部函数表）。Call指令一般是跳转到函数跳转表，然后跳转表再跳到外部函数存在的地方去。

Rand指令跳转

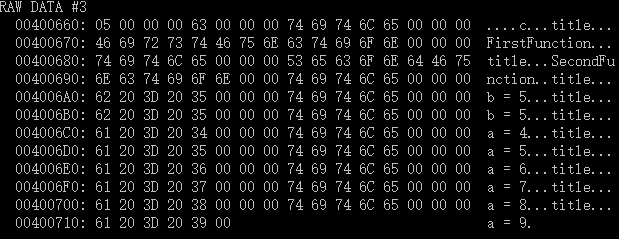


**4、获取字符串常量**

根据代码段分析，字符串常量一般出现在push的后面，一般为push地址的形式，然后就去查看该地址位于哪一节中，就可以知道它是不是字符串变量中（一般在data段中）。如果是处于data段，就挨个去获取它的字符就行



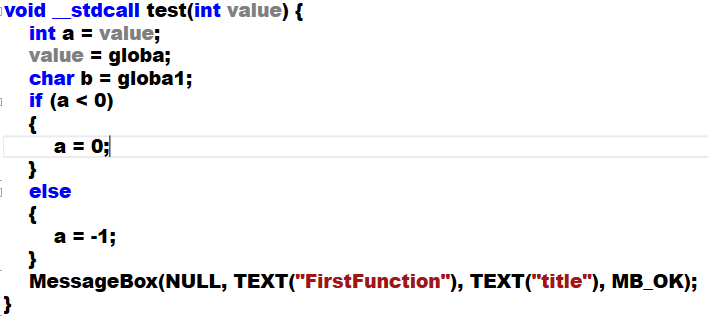
.data段

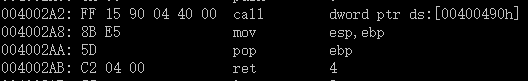


**5、获取函数之间的调用关系**

一般函数调用是call指令实现的，call指令之后跟的是函数的开头地址，而结尾是在第一个ret指令的地方。原先我以为一个函数体中出现多个return的话，那么反汇编中也会出现多个ret，但实验证明并不是这样。

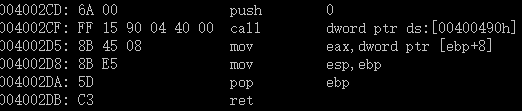
函数体中并没有return指令的时候，也会出现return



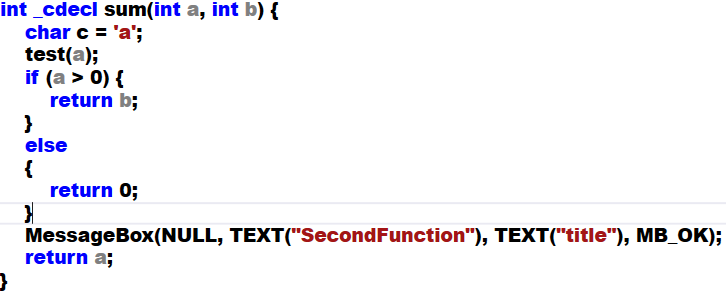


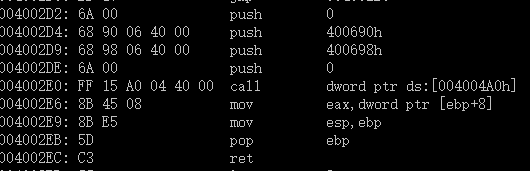
出现一个return时





出现多个return时





因此无论有return语句，在汇编语言中，总是会只有一个ret语句的。以此我们可以从程序开始判断出该程序的结尾的在哪里。然后就可以通过call指令来寻找到函数的调用关系了。

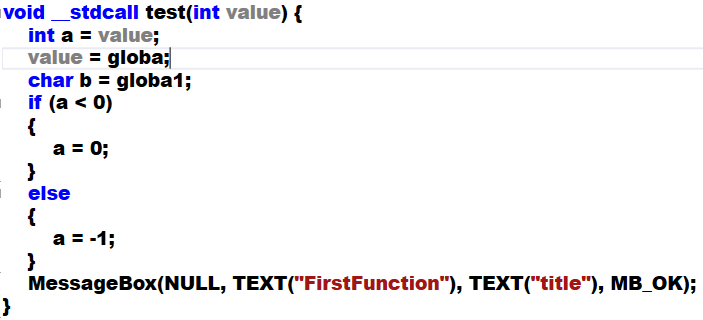
**6、函数参数**

这里需要用到主调函数和被调函数栈平衡的关系，如果根据push来分析函数参数是不可靠的方法，因为可能存在函数的参数是另外一个函数的返回值，这样就会错判函数的参数。

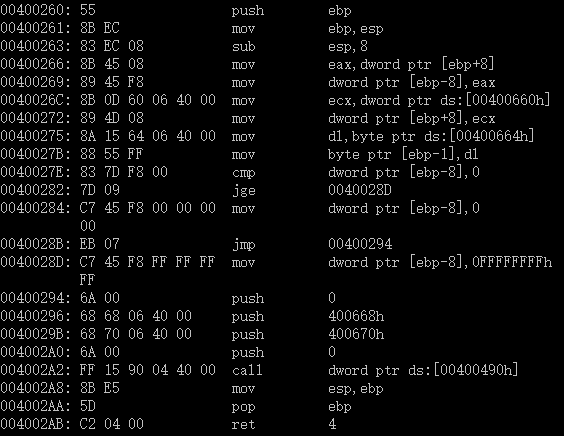
在这里，可以将栈平衡分为两类：、

1. 被调函数ret之前,被调函数做栈平衡
2. 被调函数ret之后,主调函数做栈平衡

第一种情况：以\_stdcall为关键字



此类函数会在ret之前，自己做栈平衡



第二种情况：以\_cedecl为关键字



这种函数会在主调函数中做栈平衡





采用\_\_cdecl约定时，函数参数按照从右到左的顺序入栈，并且由调用函数者把参数弹出栈以清理堆栈。因此，实现可变参数的函数只能使用该调用约定。由于每一个使用\_\_cdecl约定的函数都要包含清理堆栈的代码，所以产生的可执行文件大小会比较大。  
      采用\_\_stdcall约定时，函数参数按照从右到左的顺序入栈，被调用的函数在返回前清理传送参数的栈，函数参数个数固定。由于函数体本身知道传进来的参数个数，因此被调用的函数可以在返回前用一条ret n指令直接清理传递参数的堆栈。

一般在不做特意申明的情况下，程序默认是\_cdecl类型的，而因为\_stdcall的参数是固定的，而\_cdcel是可以在执行时才知道参数多少的。当你命名为\_stdcall时，当编译器发现你是可变参数时，会把它变成\_cdecl去执行，可以说是相当智能了。

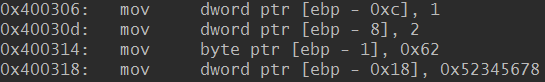
一般情况下WINDOWS API都是\_stdcall类型的，因为不同类型的编译器产生栈的方式不同，如果想要在不同的平台上运行的话，还是自己处理的栈的\_cdcel比较好。

以上就是两种函数被调用的情况，当然，这是在有参数传递的情况下才会出现的情况，如果你没有参数传递，以上的语句自然也不会出现。

而至于它后面出现的那个数字，是做了几个字节的栈平衡的关系，跟你使用的参数的大小有关，一个int的话就是4个字节。

**7、函数局部变量**

函数局部变量的形式为mov ebp的指针形式，一般为mov byte/dword ptr [bbp - xxx],值的形式，如果局部变量没有初值，编译器会先把空间划分给你，赋值时的心思也为mov byte/dword ptr [bbp - xxx],值。而出现多少种[ebp - xxx]的形式就意味着有多少个变量。



无符号和有符号的大小都是一样的，在反汇编中是无法知道类型的，只能知道大小，char是byte，Int和指针是dword，这里只需要找到特殊的指令和指令内容就可以发现局部变量了，如果后面有使用这个变量，也会调用的是[ebp - xxx]形式。

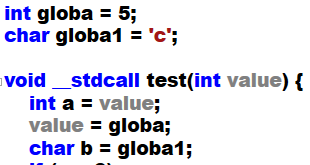
**8、全局变量**

全局变量声明后，默认是存储在.data段中的

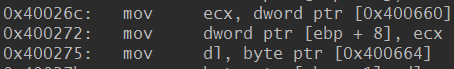
一个int一个char



当代码段使用到全局变量时



反汇编代码如下：





与局部变量不同，并不是通过寄存器寻址的形式，而是之间通过指针去调用.data段里面存储全局变量的地方，一般情况下，一个变量占四个字节的大小。

只需要找到调用全局变量的指令，然后寻址在.data段中读出数据就可以知道全局变量的大小了。

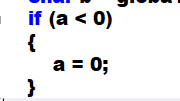
如果只是申明全局变量但是没有使用的话，全局变量的值会出现在.data段里，但是你无法知道它到底是什么，是int还是char类型？因此只有在代码段中使用了全局变量时，才能在反汇编的代码中找到全局变量。

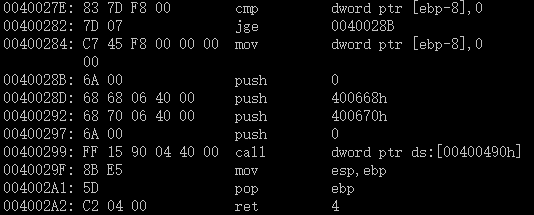
**9、if else分析**

语句结构为cmp——jcc语句，cmp后的结果存在指令寄存器里，然后jcc根据结果决定跳转不跳转。这一部分是这次实验花费时间最长的部分。

又根据if和ifelse分为两种情况

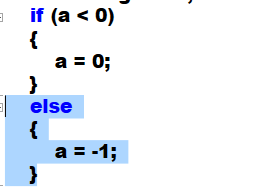
情况一：只有if的情况

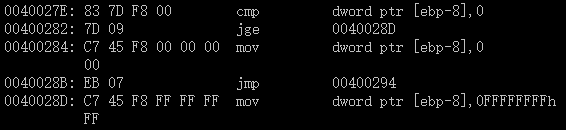




从截图的结果知道，若只有if的情况下，if语句就是cmp指令和jcc指令的合集。

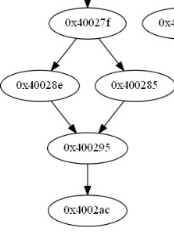
情况二：if else的情况



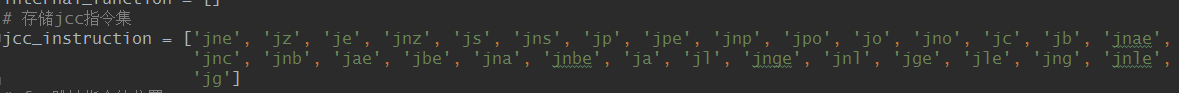


当为if else的情况时，当if语句里面的内容执行完毕后，会有一个jmp语句进行跳转。

因此，要理清内部逻辑时，需要对jmp语句、cmp和jmp搭配的语句两种语句进行区别分块，最后根据分块就可以得到如下的结果。

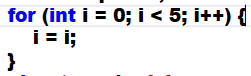


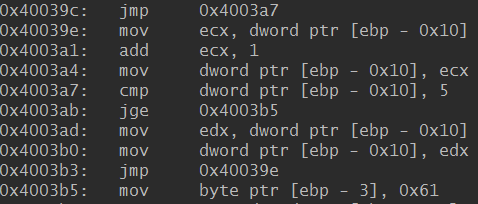
其中根据关系的不同，jcc的指令也不同，判断指令是否处于jcc指令集就可以：



**10、循环分析**

与if不同，for的结构要比较特殊一点，它是先无条件跳转，然后才进行大小判断，也就是jmp—cmp—jcc—jmp的形式来进行。

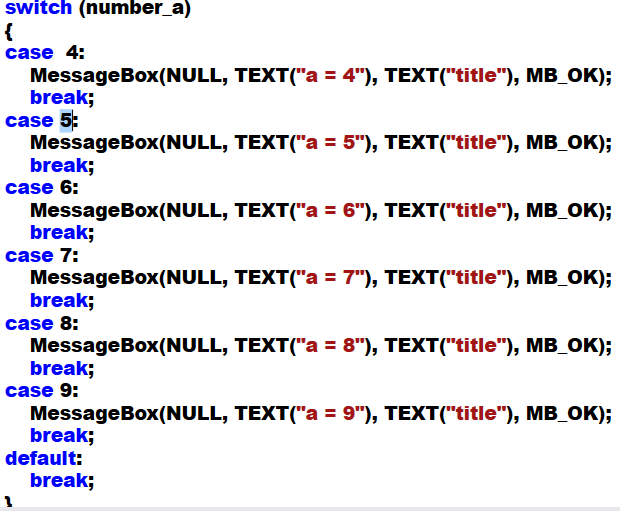


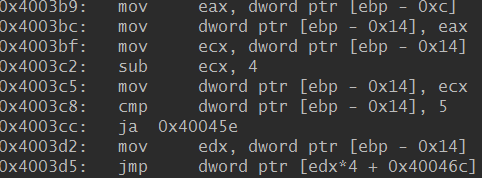


由上图可以知道，是先进行了跳转，再进行了条件判断，然后再根据结果进行跳转。因此，只要先将代码进行了分块，然后发现出现循环的情况时，就可以判定其为一个循环了。

**11、switch分析**

在C语言中实现switch结构时，会形成一个叫做跳转表一样的东西。





出于时间和内存的考虑，在原来的数字上进行减4再进行判断，如果超出了判断的范围，就直接会跳到default的代码段。如果符合判断的范围，则根据case的情况跳转到跳转表，而跳转表里会储存着case的起始代码的地址。

跳转表：



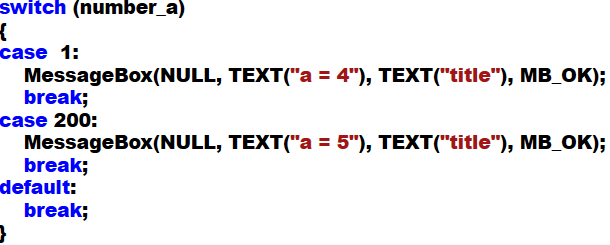
每四个字节存储一个地址，地址指向代码段中case的地址。

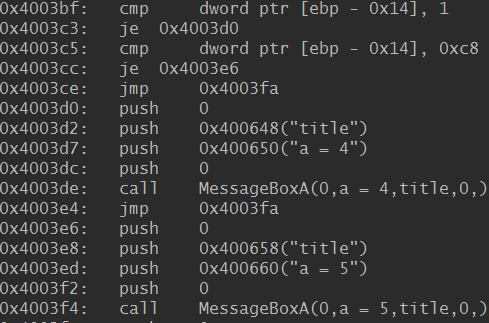
Edx的大小代表了是第几个case，而edx+基地址就代表了跳转表的位置，因为跳转表是互相挨着的，所以可以通过edx的变化来控制跳转表。

以上是正常的switch case情况。

而switch因为情况的不同也会出现不同的变化：

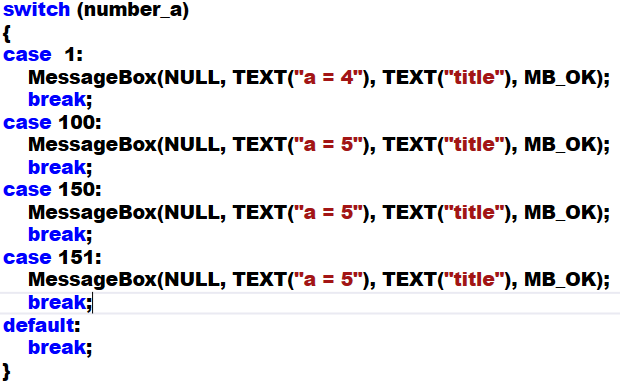
1. case的情况过少时

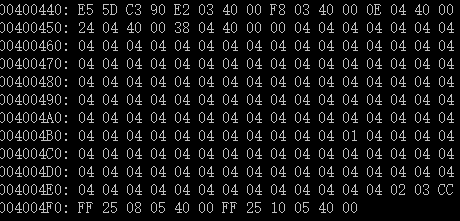




在这种情况下，编译器会把switch变成类似if else的情况，直接通过cmp-jcc来进行跳转，尽可能让运行时不必浪费太多的时间。

1. case情况下差距过大的情况

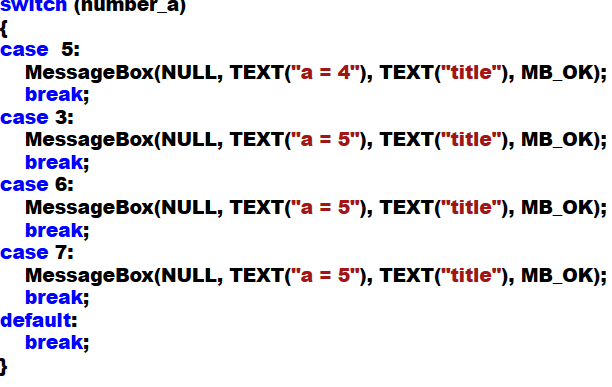




可以看出当case相差过大时，编译器为了节省空间和时间，又单独给了一个表去存case的地址（也就是上面一堆04中会出现的几个不同的数字），通过先找到case的序号，然后才去找序号所对应的真正的case的地址，这样做的好处就是尽可能的节省了空间，按照原本的规则，若是最大的case为150，则编译器也需要150\*4的空间去存地址，但如果case差距过大，如上图的情况，它的空间就会很浪费了，而现在(150-4)\*1+4\*4的空间比原来会节省很多，因此编译器产生了优化。

这里还有另外一种情况是case相差过大并且只有一个特别大或者特别小的时候，编译器会按照cmp的形式去处理这个特殊的case，而其他case依旧是以跳转表的形式去处理。

1. case并没有按顺序排列





当case是并没有按照顺序依次排列的时候，编译器在编译时，会把它按照顺排列的方式去进行排列，也就是说在C语言中没有按照顺序排列的时候，在编译时会重新给它按顺序排列好。

1. **case有差距，但差距过小**

当case的差距过小的时候，并不会按照二次跳转的形式去编译，此时，编译器会把跳转表中间缺少的case给补全，而地址就是default的地址，也就是缺少这个case的情况时，程序会执行default里面的命令。

**12、绘图**

到此为止，本次python分析出来的代码段内容已经全部没有了，之后在python中打印出来的相关结果中，直接粘贴复制到Graphviz中，稍加美化就可以画出这次作业所需要的图片了。