



南开大学

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月异



汇编语言与逆向技术

第11章 C语言程序逆向分析



允公允能 日新月异

本章知识点

1. 识别函数
2. 识别变量、数组、结构体
3. 识别IF分支结构
4. 识别Switch结构
5. 识别循环结构





南开大学

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月异

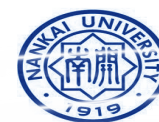
1. 识别函数



允公允能 日新月异

启动函数

- 在编写Win32应用程序时，在源码里都有一个WinMain函数。
- Windows程序的执行并不是从WinMain函数开始的，而是先执行启动函数
 - 首先执行启动函数的代码，启动函数是编译器生成的
 - 启动函数初始化进程完成后，才会执行WinMain函数



南开大学
Nankai University



允公允能 日新月异

启动函数

- C/C++程序运行时，启动函数的作用基本相同
 - 检索指向新进程的**命令行指针**
 - 检索指向新进程的**环境变量指针**
 - **全局变量**初始化
 - **内存栈**初始化





启动函数

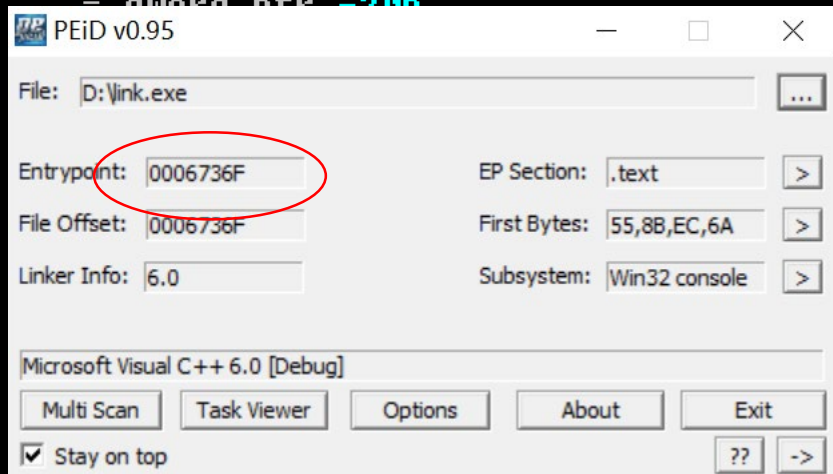
- 当所有的初始化操作完成后，启动函数就会调用应用程序的进入点函数(main和WinMain)。
- 调用WinMain函数的示例

```
GetStartupInfo (&StartupInfo);  
Int nMainRetVal = WinMain(GetModuleHandle(NULL),NULL,pszCommandLineAnsi, \  
                          (StartupInfo.dwFlags&STARTF_USESHOWWINDOW)?StartupInfo.\  
                          wShowWindow:SW_SHOWDEFAULT);
```





启动函数





允公允能 日新月异

启动函数

```
.text:00467423 50          push     eax
.text:00467424 FF 15 AC 11 40 00  call    ds:__getmainargs
.text:0046742A 68 18 90 46 00      push    offset unk_469018
.text:0046742F 68 00 90 46 00      push    offset unk_469000
.text:00467434 E8 53 00 00 00      call    _initterm
.text:00467439 FF 15 A8 11 40 00  call    ds:__p__initenv
.text:0046743F 8B 4D E0          mov     ecx, [ebp+envp]
.text:00467442 89 08          mov     [eax], ecx
.text:00467444 FF 75 E0          push    [ebp+envp]          ; envp
.text:00467447 FF 75 D4          push    [ebp+argv]          ; argv
.text:0046744A FF 75 E4          push    [ebp+argc]          ; argc
.text:0046744D E8 AE 66 FA FF      call    _main
.text:00467452 83 C4 30          add     esp, 30h
.text:00467455 89 45 DC          mov     [ebp+var_24], eax
.text:00467458 50          push     eax                ; Code
.text:00467459 FF 15 44 11 40 00  call    ds:exit
.text:0046745F          ; -----
```





允公允能 日新月异

启动函数

```
.text:0040DB00
.text:0040DB00      ; ===== S U B R O U T I N E =====
.text:0040DB00      ;
.text:0040DB00      ; Attributes: bp-based frame
.text:0040DB00
.text:0040DB00      ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0040DB00      _main      proc near      ; CODE XREF: start+DE↓p
.text:0040DB00
.text:0040DB00      var_248      = dword ptr -248h
.text:0040DB00      var_244      = dword ptr -244h
.text:0040DB00      var_240      = dword ptr -240h
.text:0040DB00      Filename    = byte ptr -23Ch
.text:0040DB00      var_13C      = dword ptr -13Ch
.text:0040DB00      var_138      = dword ptr -138h
.text:0040DB00      var_134      = dword ptr -134h
```





函数

- 程序通过**CALL**指令来调用函数，在函数执行结束后，通过**RET**指令返回调用程序继续执行

```
.text:0040DEBA 64 89 0D 00 00 00 00      mov     large fs:0, ecx
.text:0040DEC1 5F                        pop     edi
.text:0040DEC2 5E                        pop     esi
.text:0040DEC3 5B                        pop     ebx
.text:0040DEC4 8B E5                    mov     esp, ebp
.text:0040DEC6 5D                        pop     ebp
.text:0040DEC7 C3                        retn
.text:0040DEC7              _main      endp
```





允公允能 日新月异

函数

- C++ 函数定义
- `return_type function_name(parameter list) {`
 body of the function
}





函数

- 函数的参数如何传递、局部变量如何定义、函数如何返回?
- CALL指令的操作数就是所调用函数的地址或者相对地址
(MASM32的link.exe程序)

```
• .text:00467442 89 08          mov     [eax], ecx
• .text:00467444 FF 75 E0        push    [ebp+envp]      ; envp
• .text:00467447 FF 75 D4        push    [ebp+argv]      ; argv
• .text:0046744A FF 75 E4        push    [ebp+argc]      ; argc
• .text:0046744D E8 AE 66 FA FF  call    _main
• .text:00467452 83 C4 30        add     esp, 30h
```

如下图所示，指令“call _main”的二进制编码是E8 AE 66 FA FF，该指令所在的内存地址是0046744Dh，则_main函数的入口地址是 [填空1].

```

• .text:00467442 89 08          mov     [eax], ecx
• .text:00467444 FF 75 E0        push    [ebp+envp]      ; envp
• .text:00467447 FF 75 D4        push    [ebp+argv]      ; argv
• .text:0046744A FF 75 E4        push    [ebp+argc]      ; argc
• .text:0046744D E8 AE 66 FA FF  call    _main
• .text:00467452 83 C4 30        add     esp, 30h

```

```

.text:00400B00
.text:00400B00      ; ===== S U B R O U T I N E =====
.text:00400B00      |
.text:00400B00      ; Attributes: bp-based frame
.text:00400B00
.text:00400B00      ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00400B00      _main      proc near      ; CODE XREF: start+DE↓p
.text:00400B00
.text:00400B00      var_248    = dword ptr -248h
.text:00400B00      var_244    = dword ptr -244h
.text:00400B00      var_240    = dword ptr -240h
.text:00400B00      Filename   = byte ptr -23Ch
.text:00400B00      var_13C    = dword ptr -13Ch
.text:00400B00      var_138    = dword ptr -138h
.text:00400B00      var_134    = dword ptr -134h

```

正常使用填空题需3.0以上版本雨课堂

作答



如下图所示，指令“call _main”的二进制编码是E8 AE 66 FA FF，该指令所在的内存地址是0046744Dh，则_main函数的入口地址是 [填空1].

解题思路：call 后的二进制编码 E8代表调用，剩下的则为距离下一条指令的偏移地址，注意小端字节序。因此其计算步骤：

- 1) 计算下一条指令地址：当前指令占5个字节，当前地址为0046744Dh，因此下一条指令地址为 $0046744D + 5 = 00467452h$
- 2) 计算main地址：小端序偏移 $FF\ FA\ 66\ AE + 00\ 46\ 74\ 52 = 0040DB00h$



允公允能 日新月异

栈

- 栈是一种后入先出的数据存储结构
- 函数的参数、局部变量、返回地址等被存储在栈中
- ESP (Extended Stack Pointer) 存储栈顶的内存地址
- EBP (Extended Base Pointer) 存储栈底的内存地址
- PUSH指令将数据压入栈顶
- POP指令从栈顶取出数据



南开大学
Nankai University



允公允能 日新月异

函数的开始和结束

- 开始: `push ebp` (构建栈底标志) `mov ebp, esp` (初始化栈顶和栈底在同一个位置)
- 结束: `mov esp, ebp` (清空栈) `pop ebp` (释放栈底)



函数的调用过程

- 第一步：使用push指令将参数压入栈中
- 第二步： call memory_location
 - call的返回地址压入栈中
 - EIP的值被设为memory_location
- 第三步： push ebp, mov ebp,
- 第四步： add esp ****, 在栈中分配局部变量的空间





保存返回地址

00401000	A1 10304000	MOV EAX,DWORD PTR DS:[403010]	
00401005	68 00304000	PUSH OFFSET 00403000	ASCII "Hello World!",LF,CR
0040100A	E8 09000000	CALL 00401018	
0040100F	6A 00	PUSH 0	
00401011	E8 AC000000	CALL <JMP.&kernel32.ExitProcess>	跳转至 KERNEL32.ExitProcess
00401016	CC	INT3	
00401017	CC	INT3	
00401018	55	PUSH EBP	
00401019	8BEC	MOV EBP,ESP	
0040101B	83C4 F4	ADD ESP,-0C	
0040101E	6A F5	PUSH -0B	
00401020	E8 A3000000	CALL <JMP.&kernel32.GetStdHandle>	跳转至 KERNEL32.GetStdHandle
00401025	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
00401028	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040102B	E8 20000000	CALL 00401050	
00401030	8945 F4	MOV DWORD PTR SS:[EBP-0C],EAX	
00401033	6A 00	PUSH 0	
00401035	8D45 E8	LEA EAX,[EBP-8]	

Stack [0019FF68]=0
EBP=0019FF80

地址	十六进制数据	多字节 (ANSI/OEM - 简体中文)
00403000	48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 0D 00 31	He l l o W o r l d ! □
00403010	00 00 00 32 00 00 00 33 00 00 00 0A 00 00 00 0D	

寄存器 (FPU)

EAX	32000000
ECX	00401000 hello.<ModuleEntryPoint>
EDX	00401000 hello.<ModuleEntryPoint>
EBX	002EB000
ESP	0019FF6C
EBP	0019FF80
ESI	00401000 hello.<ModuleEntryPoint>
EDI	00401000 hello.<ModuleEntryPoint>
EIP	00401018 hello.00401018
C 0	ES 002B 32Bit 0(FFFFFFFF)
P 1	CS 0023 32Bit 0(FFFFFFFF)
A 0	SS 002B 32Bit 0(FFFFFFFF)
Z 1	DS 002B 32Bit 0(FFFFFFFF)
S 0	FS 0053 32Bit 2EE000(FFF)
T 0	GS 002B 32Bit 0(FFFFFFFF)
D 0	
O 0	LastErr 00000000 ERROR_SUCCESS
EFL	00000246 (NO_NB_E_BE_NS_PE_GE_IE)

0019FF6C [0040100F] CC 返回到 hello.00401018 来自 hello.0040100F

0019FF70 [00403000] L ASCII "Hello World!",LF,CR

0019FF74 [76060419] CCI 返回到 KERNEL32.76060419





局部变量的初始化

```
; int __stdcall sub_401018(LPCVOID lpBuffer)
sub_401018      proc near                                ; CODE XREF: start+A↑p

nNumberOfBytesToWrite= dword ptr -0Ch
NumberOfBytesWritten= dword ptr -8
hFile           = dword ptr -4
lpBuffer        = dword ptr 8

push    ebp
mov     ebp, esp
add     esp, 0FFFFFFF4h
push    0FFFFFFF5h ; nStdHandle
call    GetStdHandle
```

EFI 00000207 (NO R NE BE NS PE GE G)	
0019FF5C	00000000
0019FF60	00000000
0019FF64	00000000
0019FF68	0019FF80
0019FF6C	0040100F
0019FF70	00403000
0019FF74	76060410

返回到 hello.00401018 来自 hello.0040100F
ASCII "Hello World!", LF, CR
返回到 KERNEL32.76060410





九 局部变量与参数

- 局部变量相对地址 负数（dword类型占4个字节）
- 函数参数相对地址 正数（隐含返回地址占4个字节，其他参数按类型占用相应的字节数，dword占4个字节）
- 每压入一个局部变量，ESP向负数方向(即低地址方向)增加，为局部变量腾出空间
 - `add esp 0FFFFFFF4h` → `sub esp 0Ch` 或 `add esp -0Ch`





允公允能 日新月异

栈帧 (Stack Frame)

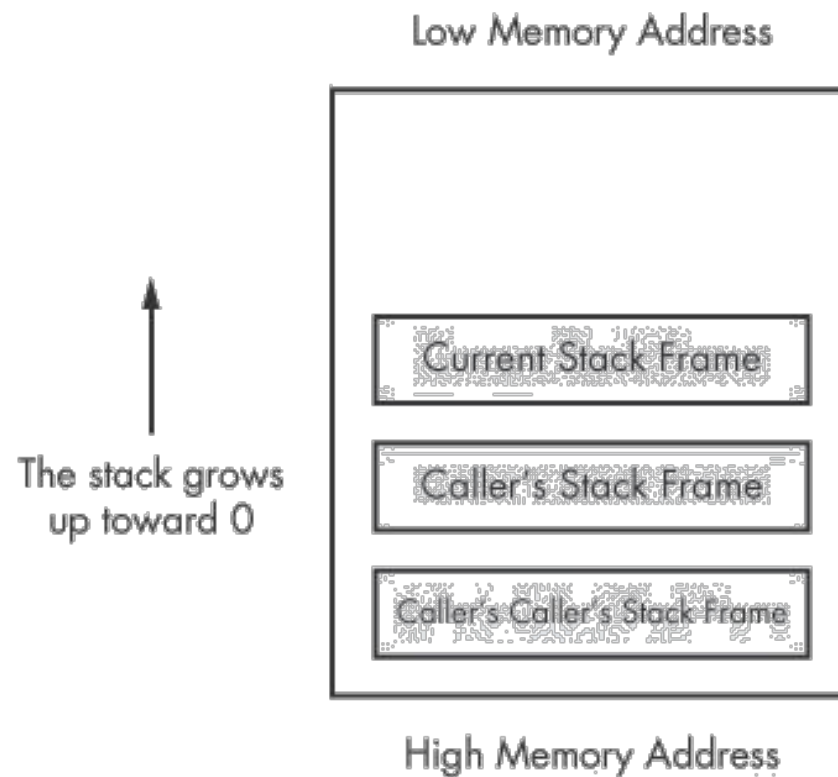
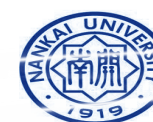


Figure 5-7. x86 stack layout



栈帧 (Stack Frame)

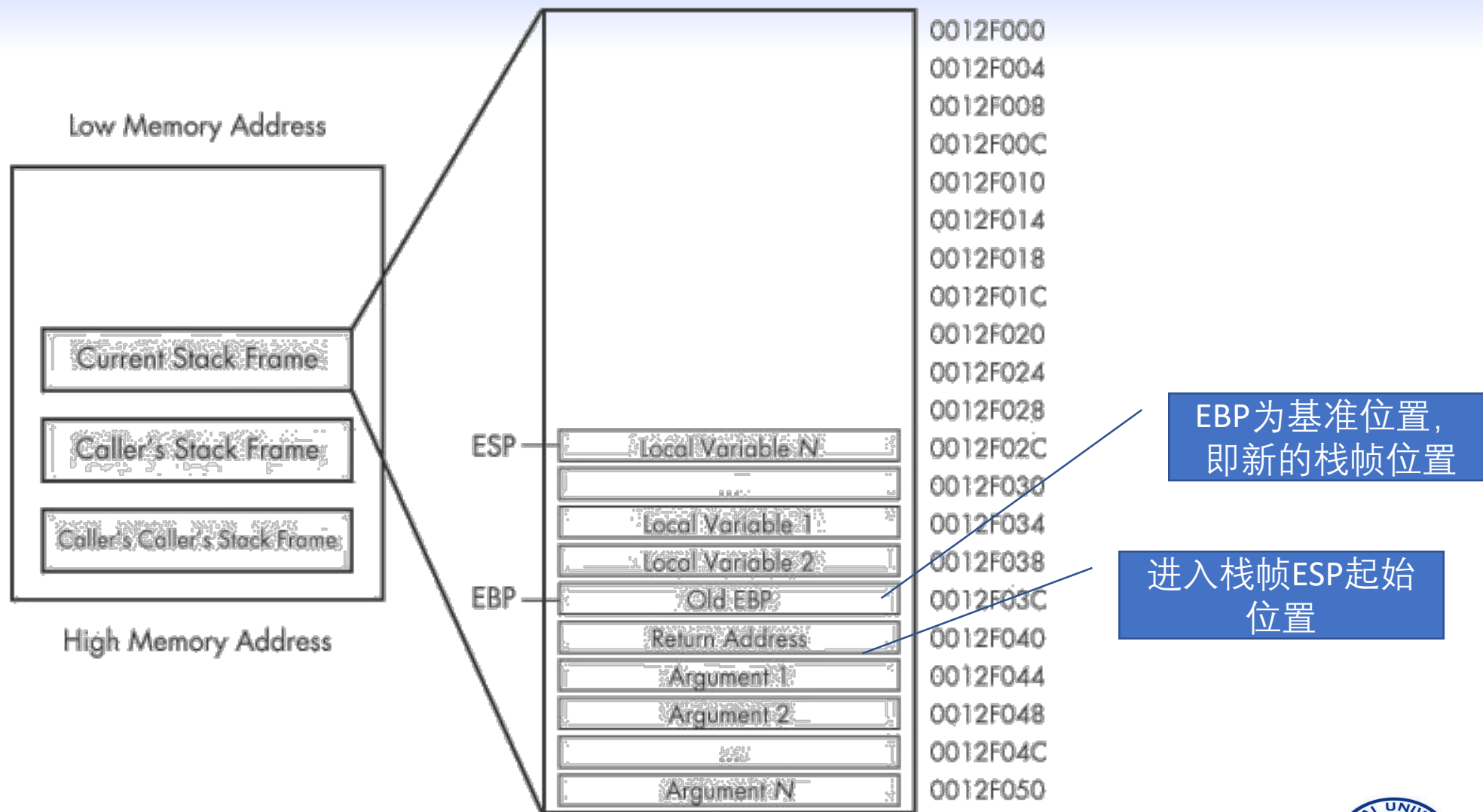


Figure 5-8. Individual stack frame

栈帧（深入理解）

- 所有元素压栈，做push操作时，ESP均向负方向增长。负方向增长的幅度与数据类型相关，如byte变量压入时，ESP减1；dword参数压栈时，ESP减4。
- 函数调用时入栈顺序：①参数入栈（栈帧前即入栈）；②返回值地址入栈（32位4个字节）；③ EBP入栈（即Push EBP操作，4个字节）；④局部变量入栈（未push，但需要腾出空间）。ESP均会发生相应变化。
- 以EBP为基准，因此参数为正，局部变量为负。因为参数在EBP之前入栈，而局部变量在EBP之后入栈。
- EBP代表栈帧地址，计算栈帧布局时，首先确定EBP地址，再通过正负关系确定参数、返回值地址、局部变量地址。



栈帧中的ESP和EBP

- 已知函数调用前栈帧的ESP地址， 经过栈初始化操作， 即

`push ebp`

`mov ebp, esp`

之后， EBP的地址更新为ESP-8（压入返回值地址、EBP地址）

- 注意参数在进入栈帧前就已经入栈， 因此栈帧阶段的ESP变化与参数无关
- 入栈时ESP变化与调用约定也无关， 调用约定规定的是调用之后回收的情况。 但要注意调用约定规定的参数入栈顺序， 一般为从右到左。



如下图所示，指令“PUSH OFFSET 00403000”执行之前，ESP的值是0019FF74。PUSH指令执行之后，ESP的值是[填空1]。

提示：32位的内存地址

00401000	A1 10304000	MOV EAX,DWORD PTR DS:[403010]
00401005	68 00304000	PUSH OFFSET 00403000
0040100A	E8 09000000	CALL 00401018
0040100F	6A 00	PUSH 0
00401011	E8 AC000000	CALL <JMP.&kernel32.ExitProcess>
00401016	CC	INT3
00401017	CC	INT3
00401018	55	PUSH EBP
00401019	8BEC	MOV EBP,ESP
0040101B	83C4 F4	ADD ESP,-0C
0040101E	6A F5	PUSH -0B
00401020	E8 A3000000	CALL <IMP.&kernel32.GetStdHandle>

Stack [0019FF70]=0
Imm=hello.00403000, ASCII "Hello World!", LF, CR

寄存器 (FPU)			
EAX	32000000		
ECX	00401000	hello.<ModuleEntryPoint>	
EDX	00401000	hello.<ModuleEntryPoint>	
EBX	0028F000		
ESP	0019FF74		
EBP	0019FF80		
0019FF74	76060419	返回到	KERNEL32.76060419
0019FF78	0028F000	0	
0019FF7C	76060400	返回到	KERNEL32.BaseThreadInitThunk
0019FF80	0019FFDC	返回到	ntdll.777F66DD
0019FF84	777F66DD	返回到	ntdll.777F66DD

正常使用填空题需3.0以上版本雨课堂

作答





允公允能 日新月异

00401000	A1 10304000	MOV EAX,DWORD PTR DS:[403010]	寄存器 (FPU)
00401005	68 00304000	PUSH OFFSET 00403000	EAX 32000000
0040100A	E8 09000000	CALL 00401018	ECX 00401000 hello.<ModuleEntryPoint>
0040100F	6A 00	PUSH 0	EDX 00401000 hello.<ModuleEntryPoint>
00401011	E8 AC000000	CALL <JMP.&kernel32.ExitProcess>	EBX 0028F000
00401016	CC	INT3	ESP 0019FF70 到 PTR ASCII "Hello World!",LF,CR
00401017	CC	INT3	EDF 0019FF80
00401018	55	PUSH EBP	ESI 00401000 hello.<ModuleEntryPoint>
00401019	8BEC	MOV EBP,ESP	EDI 00401000 hello.<ModuleEntryPoint>
0040101B	83C4 F4	ADD ESP,-0C	EIP 0040100A hello.0040100A
0040101E	6A F5	PUSH -0B	C 0 ES 002B 32Bit 0(FFFFFFFF)
00401020	E8 A3000000	CALL <IMP.&kernel32.GetStdHandle>	P 1 CS 0023 32Bit 0(FFFFFFFF)
Dest=hello.00401018			A 0 SS 002B 32Bit 0(FFFFFFFF)
地址	ASCII 数据 (IBM EBCDIC - 国际)		0019FF70 00403000 [ASCII "Hello World!",LF,CR
00403000	cA%%?		0019FF74 76060419 [返回到 KERNEL32.76060419
00403040			0019FF78 0028F000 0
00403080			0019FF7C 76060400 [KERNEL32.BaseThreadInitThunk



南开大学

Nankai University



允公允能 日新月异

函数调用过程

- 执行函数
- 恢复局部变量的栈空间
- `ret`指令从栈中读取返回地址，设置EIP
- 恢复参数占用的栈空间





函数的调用过程

3F FF75 FC PUSH DWORD PTR SS:[EBP-4]
42 E8 87000000 CALL <JMP.&kernel32.WriteFile>
47 8B45 F8 MOV EAX,DWORD PTR SS:[EBP-8]
4A C9 LEAVE
4B C2 0400 RETN 4
4E CC INT3
4E CC INT3

跳转至 KERNEL32.WriteFile

ES 002B 32Bit 0(FFFFFFFF)
CS 0023 32Bit 0(FFFFFFFF)
SS 002B 32Bit 0(FFFFFFFF)
DS 002B 32Bit 0(FFFFFFFF)
FS 0053 32Bit 2EE000(FFF)
GS 002B 32Bit 0(FFFFFFFF)
D 0
LastErr 00000000 ERROR_SUCCESS
00000202 (NO NB NE A NS PO GE G)

十六进制数据	多字节 (ANSI/OEM - 简体中文)
00 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 0D 00 31	He l l o W o r l d !
10 00 00 00 32 00 00 00 33 00 00 00 0A 00 00 00 0D	
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D	
30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D	
40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D	

0019FF5C 0000000E
0019FF60 0000000E
0019FF64 000000A0
0019FF68 0019FF80
0019FF6C 0040100F
0019FF70 00403000

返回到 hello.00401018 来自 hello.0040100F
ASCII "Hello World!", LF, CR

0040104A C9 LEAVE
0040104B C2 0400 RETN 4
0040104E CC INT3
0040104F CC INT3
00401050 8B4424 04 MOV EAX,DWORD PTR SS:[ESP+4]

Imm=0004
栈顶 [0019FF6C]=hello.0040100F

ES 002B 32Bit 0(FFFFFFFF)
CS 0023 32Bit 0(FFFFFFFF)
SS 002B 32Bit 0(FFFFFFFF)
DS 002B 32Bit 0(FFFFFFFF)
FS 0053 32Bit 2EE000(FFF)
GS 002B 32Bit 0(FFFFFFFF)
D 0
LastErr 00000000 ERROR_SUCCESS
00000202 (NO NB NE A NS PO GE G)

地址	十六进制数据	多字节 (ANSI/OEM - 简体中文)
00403000	48 65 6C 6C 6F 20 57 6F 72 6C 64 21 0A 0D 00 31	He l l o
00403010	00 00 00 00 32 00 00 00 33 00 00 00 0A 00 00 00 0D	
00403020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0D	

0019FF6C 0040100F
0019FF70 00403000
0019FF74 76060419
0019FF78 002EB000

返回到 hello.00401018 来自 h
ASCII "Hello World!", LF, CR
返回到 KERNEL32.76060419

上一页PPT的函数调用地址

调用约定 (Calling Convention)

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near                                ; CODE XREF: start+AF↓p
var_4      = dword ptr -4
argc       = dword ptr 8
argv       = dword ptr 0Ch
envp       = dword ptr 10h

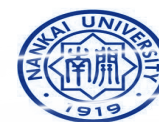
push      ebp
mov       ebp, esp
```




允公允能 日新月异

Calling Convention

- 在x86平台，函数所有参数的宽度都是32bits，函数的返回值的宽度是32bits，存储在EAX 寄存器中
- 调用约定：被调函数callee和主函数caller如何传递参数和返回值的约定
- VC 编译器支持以下两种调用约定
 - __cdecl
 - __stdcall





Calling Convention

- `__cdecl` 是 C and C++ 程序的标准函数调用

Element	Implementation
Argument-passing order	Right to left.
Stack-maintenance responsibility	Caller function pops the arguments from the stack.



平衡栈（C Declaration）

- cdecl（C Declaration）：
 - 调用者在调用函数之前将参数压入栈中，函数在执行时可以将更多的局部变量压入栈中，压栈操作ESP往负方向增长。
 - 函数返回时，只有返回地址被弹出栈，参数仍然留在栈上。调用者在函数返回后负责从栈上清除这些参数，让ESP往相反方向即正方向增长。
 - 对应到栈帧布局，cdecl约定只会回收到返回值处，剩下的参数由调用者回收（即在调用者处增加ESP）。





__cdecl

平衡栈操作

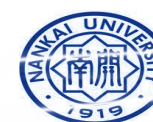
```
lea    ecx, [eax+7]
mov     dl, [eax+6]
push    ecx
push    edx
lea     eax, [esp+70h+var_64]
push    offset aS_0      ; "%*s"
push    eax              ; char *
call    _sprintf
add     esp, 10h
lea     ecx, [esp+68h+var_64]
push    0                ; int
```



Calling Convention

- `__stdcall` 是 Win32 API 函数的调用约定

Element	Implementation
Argument-passing order	Right to left.
Stack-maintenance responsibility	Callee function pops its own arguments from the stack.





平衡栈（stdcall）

- stdcall（Standard Call）：
 - 类似于 cdecl，调用者在调用函数之前将参数压入栈中，函数在执行时可以将更多的局部变量压入栈中，ESP往负方向增长。
 - 不同于 cdecl，当函数返回时，被调用函数本身负责从栈上清除参数和返回地址，ESP在函数内部往正方向增长。因此，在函数返回后，栈被恢复到调用函数之前的状态。
 - 对应到栈帧布局，stdcall会直接回收所有参数。即ESP会增长到首个参数位置。



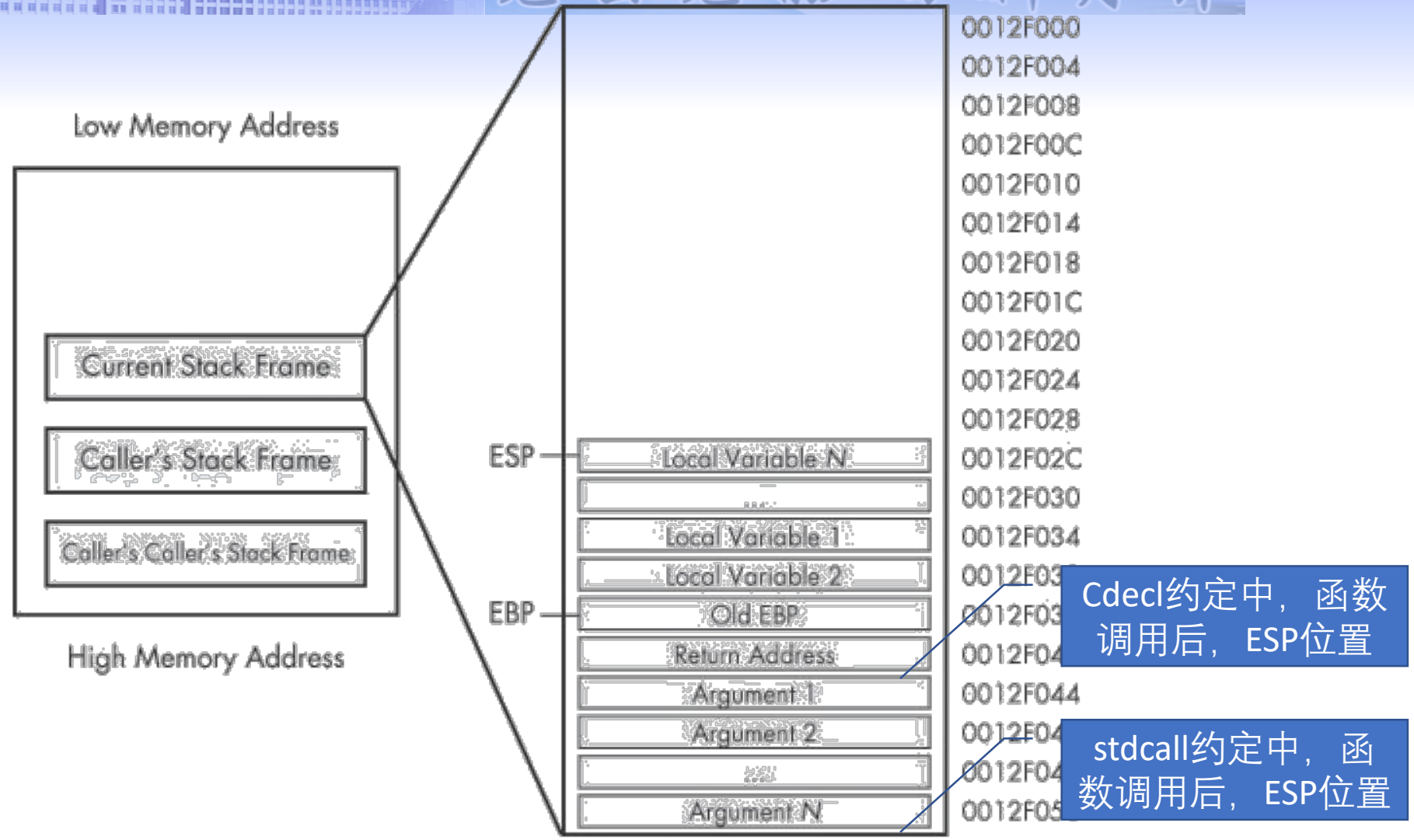


Figure 5-8. Individual stack frame



允公允能 日新月异

__stdcall

```
loc_438E61:                                ; CODE XREF: __lseek+36↑j
        push    [esp+0Ch+dwMoveMethod] ; dwMoveMethod
        push    0                          ; lpDistanceToMoveHigh
        push    [esp+14h+lDistanceToMove] ; lDistanceToMove
        push    eax                        ; hFile
        call    ds:SetFilePointer
        mov     ebx, eax
        cmp     ebx, 0FFFFFFFFh
        jnz     short loc_438E81
        call    ds:GetLastError
        jmp     short loc_438E83
```



函数URLDownloadToFileA的调用约定是?

```
push    0                ; LPBINDSTATUSCALLBACK
push    0                ; DWORD
push    offset aCEmpdownload_e ; "c:\tempdownload.exe"
mov     eax, [ebp+var_4]
mov     ecx, [eax]
push    ecx              ; LPCSTR
push    0                ; LPUNKNOWN
call    URLDownloadToFileA
mov     esp, ebp
pop     ebp
retn
endp
```

A __cdecl

B __stdcall

提交





Calling Convention

```
push 0 ; LPBINDSTATUSCALLBACK
push 0 ; DWORD
push offset aCEmpdownload_e ; "c:\tempdownload.exe"
mov eax, [ebp+var_4]
mov ecx, [eax]
push ecx ; LPCSTR
push 0 ; LPUNKNOWN
call URLDownloadToFileA
mov esp, ebp
pop ebp
ret
endp
```

- 1) 将esp退到ebp原点
- 2) 释放ebp



南开大学

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月异



识别变量、数组、结构体



允公允能 日新月异

局部变量和全局变量

- 全局变量
 - 可以任意函数访问和修改的变量
- 局部变量
 - 只能在定义该变量的函数内部，访问和修改





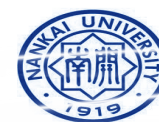
允公允能 日新月异

全局变量和局部变量

```
int x = 1;
int y = 2;

void main() {
    x = x+y;
    printf("Total = %d\n", x);
}
```

```
void main() {
    int x = 1;
    int y = 2;
    x = x+y;
    printf("Total = %d\n", x);
}
```





允公允能 日新月异

全局变量

00401003	mov	eax, dword_40CF60
00401008	add	eax, dword_40C000
0040100E	mov	dword_40CF60, eax ❶
00401013	mov	ecx, dword_40CF60
00401019	push	ecx
0040101A	push	offset aTotalD ; "total = %d\n"
0040101F	call	printf





局部变量

00401006	mov	dword ptr [ebp-4], 0
0040100D	mov	dword ptr [ebp-8], 1
00401014	mov	eax, [ebp-4]
00401017	add	eax, [ebp-8]
0040101A	mov	[ebp-4], eax
0040101D	mov	ecx, [ebp-4]
00401020	push	ecx
00401021	push	offset aTotalD ; "total = %d\n"
00401026	call	printf



mov eax, [ebp+var_4]”

[ebp+var_4] 是一个全局变量还是局部变量？

- ☒ A 局部变量
- ☐ B 全局变量

提交





允公允能 日新月异

数组

- 数组是**相同数据类型**的元素的集合，它们在内存中按顺序连续存放在一起。
- 在汇编状态下访问数组一般是通过基址加变址寻址实现的





允公允能 日新月异

数组

- `int ary[4] = {1, 2, 3, 4}`
 - 每个整数占用4个字节，数组占用了16个字节，假设数组的首地址是0x1000
 - `ary[0]` 的位置是0x1000
 - `ary[1]`的位置是0x1004
 - `ary[2]`的位置是0x1008
 - `ary[3]`的位置是0x100C
- 数组元素的地址=数组首地址+ `sizeof(元素类型)*索引值`





数组

- 数组a是局部变量，数组b是全局变量

```
int b[5] = {123,87,487,7,978};  
void main()  
{  
    int i;  
    int a[5];  
  
    for(i = 0; i<5; i++)  
    {  
        a[i] = i;  
        b[i] = i;  
    }  
}
```



数组

```

00401006      mov     [ebp+var_18], 0
0040100D      jmp     short loc_401018
0040100F loc_40100F:
0040100F      mov     eax, [ebp+var_18]
00401012      add     eax, 1
00401015      mov     [ebp+var_18], eax
00401018 loc_401018:
00401018      cmp     [ebp+var_18], 5
0040101C      jge     short loc_401037
0040101E      mov     ecx, [ebp+var_18]
00401021      mov     edx, [ebp+var_18]
00401024      mov     [ebp+ecx*4+var_14], edx
00401028      mov     eax, [ebp+var_18]
0040102B      mov     ecx, [ebp+var_18]
0040102E      mov     dword_40A000[ecx*4], eax
00401035      jmp     short loc_40100F
    
```

i=0

i++

若i>=5,则结束for

Var_14为数组首地址

Int占4个字节



允公允能 日新月异

结构体

- 在c语言中，结构体(struct)是一种数据结构，可以将不同类型的数据结构组合到一个复合的数据类型中



南开大学
Nankai University



结构体

```
struct my_structure { ❶
    int x[5];
    char y;
    double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
    int i;
    q->y = 'a';
    q->z = 15.6;
    for(i = 0; i<5; i++){
        q->x[i] = i;
    }
}

void main()
{
    gms = (struct my_structure *) malloc(
        sizeof(struct my_structure));
    test(gms);
}
```





允公允能 日新月异

test函数

```
00401000    push    ebp
00401001    mov     ebp, esp
00401003    push    ecx
00401004    mov     eax, [ebp+arg_0]
00401007    mov     byte ptr [eax+14h], 61h
0040100B    mov     ecx, [ebp+arg_0]
0040100E    fld     ds:dbl_40B120
00401014    fstp    qword ptr [ecx+18h]
00401017    mov     [ebp+var_4], 0
0040101E    jmp     short loc_401029
00401020 loc_401020:
00401020    mov     edx, [ebp+var_4]
00401023    add     edx, 1
00401026    mov     [ebp+var_4], edx
00401029 loc_401029:
00401029    cmp     [ebp+var_4], 5
0040102D    jge     short loc_40103D
0040102F    mov     eax, [ebp+var_4]
00401032    mov     ecx, [ebp+arg_0]
00401035    mov     edx, [ebp+var_4]
00401038    mov     [ecx+eax*4], edx
0040103B    jmp     short loc_401020
0040103D loc_40103D:
0040103D    mov     esp, ebp
0040103F    pop     ebp
00401040    retn
```

q->y='a'

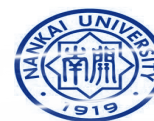
浮点数赋值, q->z=15.6

i=0

i++

i若大于等于5, 则结束for

q->x[i]=i





南開大學

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月異

3. 识别IF分支结构



识别IF分支结构

```
int x = 1;
int y = 2;

if(x == y){
    printf("x equals y.\n");
}else{
    printf("x is not equal to y.\n");
}
```

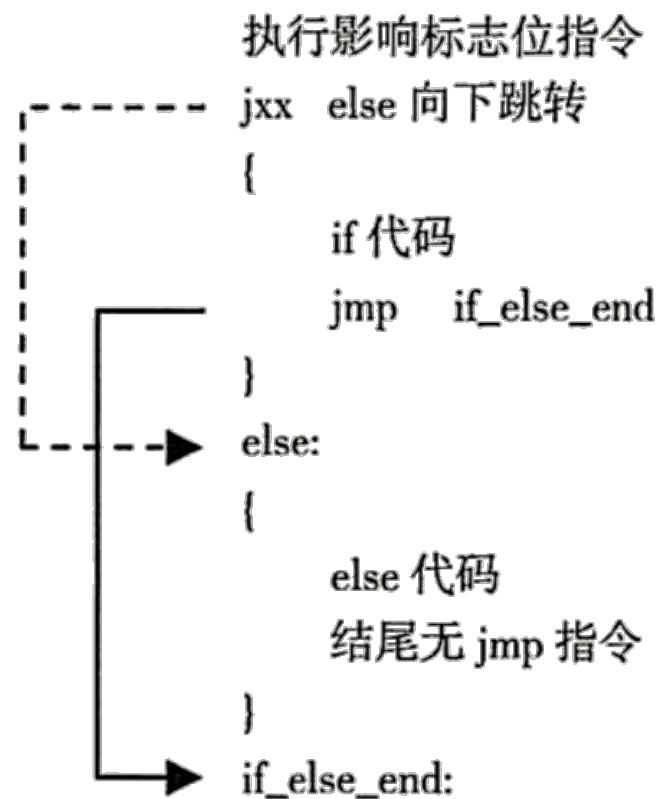
```
00401006      mov     [ebp+var_8], 1
0040100D      mov     [ebp+var_4], 2
00401014      mov     eax, [ebp+var_8]
00401017      cmp     eax, [ebp+var_4] ❶
0040101A      jnz     short loc_40102B ❷
0040101C      push    offset aXEqualsY_ ; "x equals y.\n"
00401021      call    printf
00401026      add     esp, 4
00401029      jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B      push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030      call    printf
```



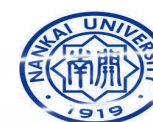


识别IF分支结构

- IF语句的识别特征，jxx
的跳转和一个无条件
jmp指令



识别IF分支结构





南開大學

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月異



4. 识别Switch结构



允公允能 日新月异

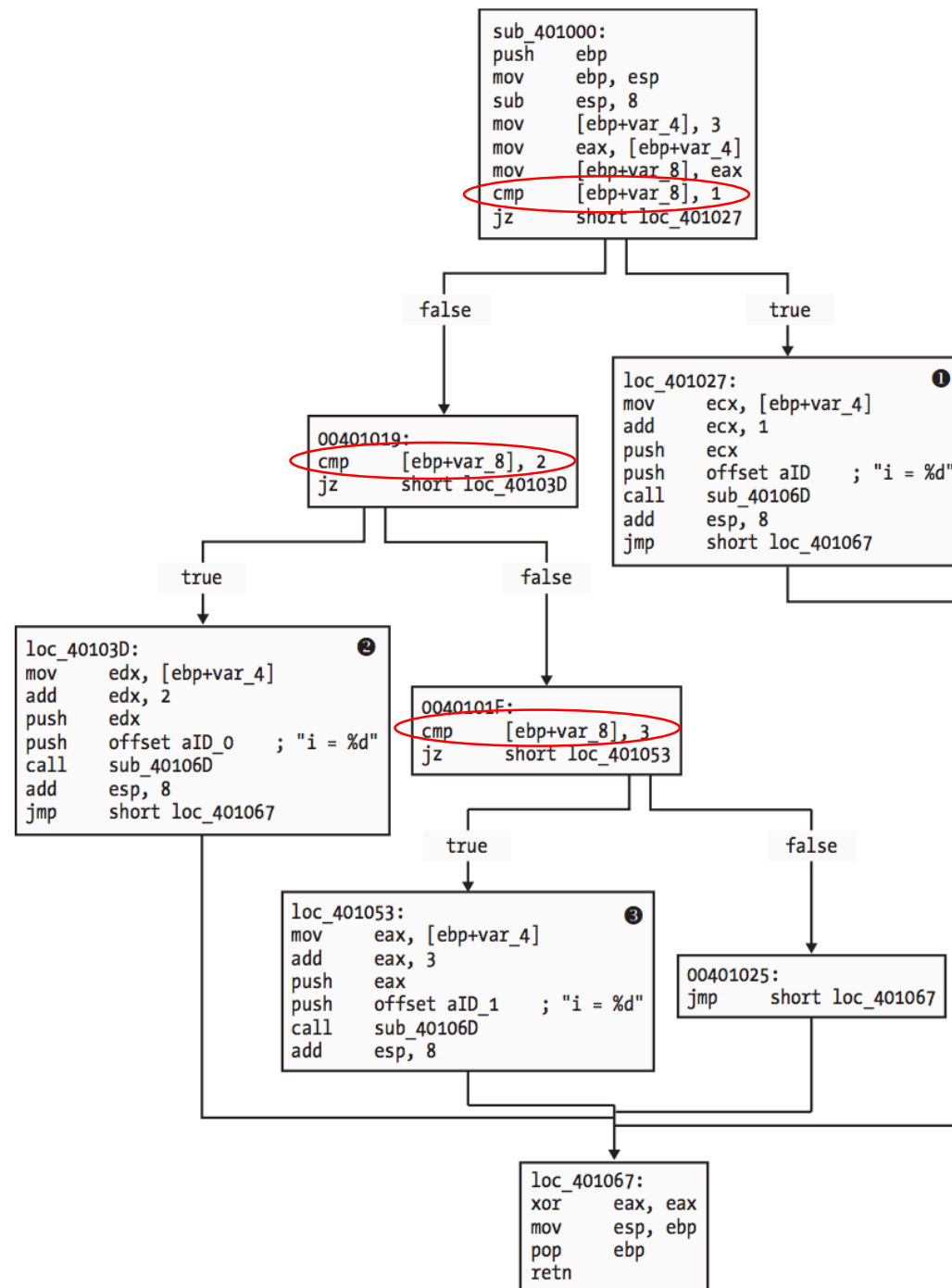
识别Switch结构

- Switch结构用来实现基于字符或者整数的决策。
- Switch结构通常以两种方式被编译
 - 使用IF方式
 - 使用跳转表

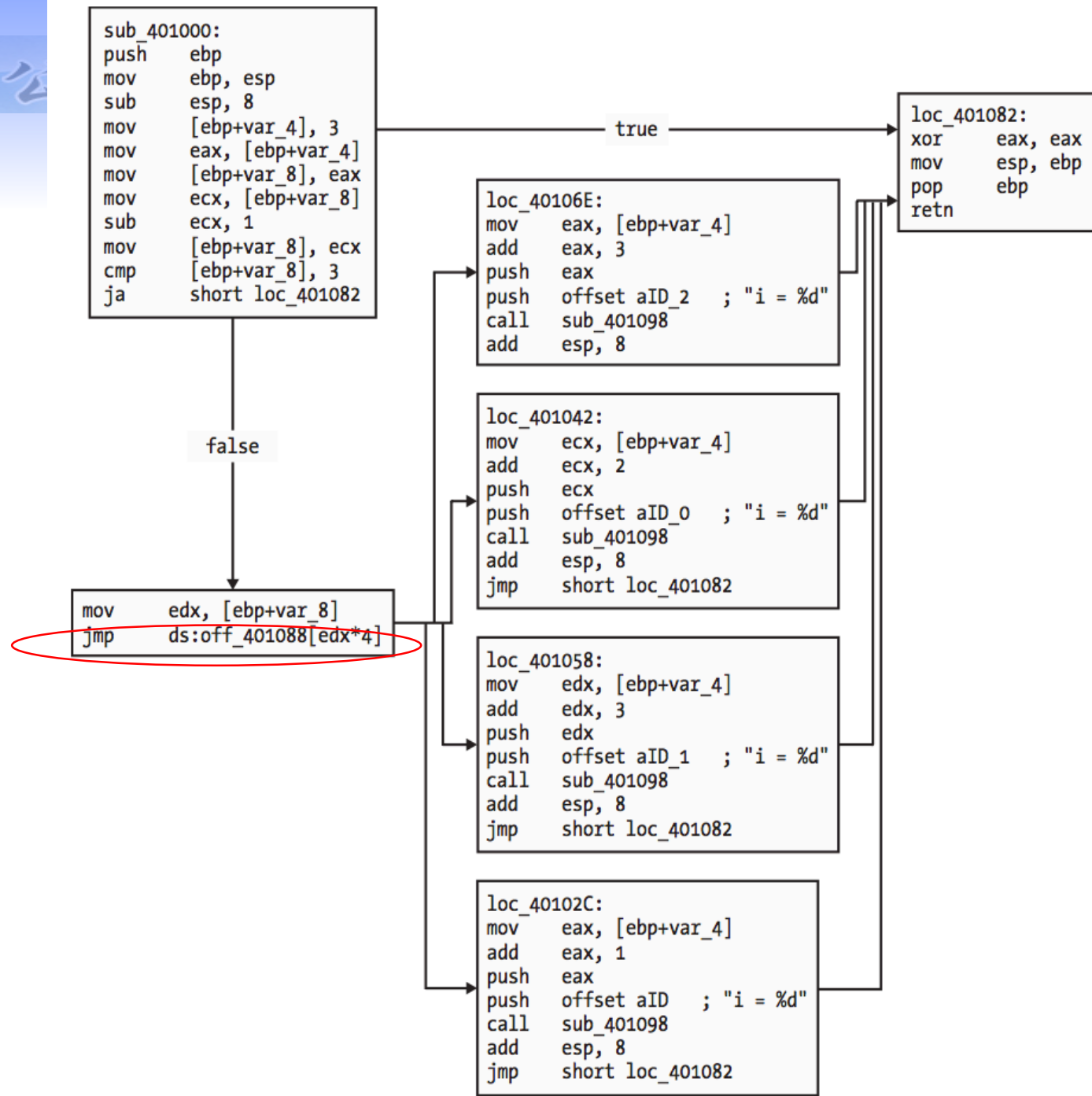


南开大学
Nankai University

Switch-IF方式



Switch-跳转表方式





南开大学

NANKAI UNIVERSITY, P.R. CHINA 1919

允公允能 日新月异

5. 识别循环



允公允能 日新月异

识别循环

- FOR循环是一个C/C++编程使用的基本循环机制。
- FOR循环有4个组件：
 - 初始化
 - 比较
 - 指令执行体
 - 递增或递减

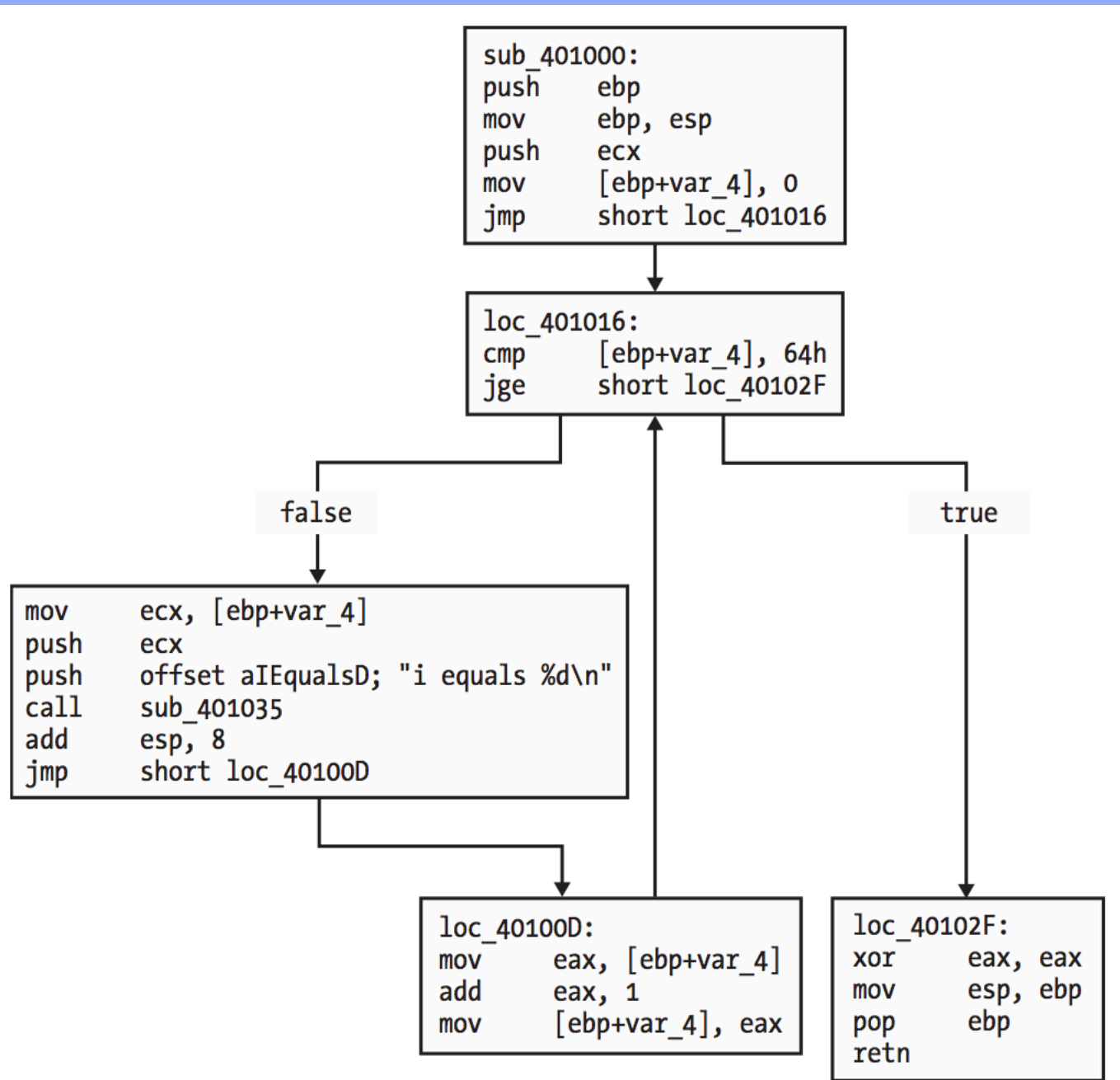


识别循环

```
int i;  
  
for(i=0; i<100; i++)  
{  
    printf("i equals %d\n", i);  
}
```

```
00401004    mov     [ebp+var_4], 0 ❶  
0040100B    jmp     short loc_401016 ❷  
0040100D loc_40100D:  
0040100D    mov     eax, [ebp+var_4] ❸  
00401010    add     eax, 1  
00401013    mov     [ebp+var_4], eax ❹  
00401016 loc_401016:  
00401016    cmp     [ebp+var_4], 64h ❺  
0040101A    jge     short loc_40102F ❻  
0040101C    mov     ecx, [ebp+var_4]  
0040101F    push    ecx  
00401020    push    offset aID ; "i equals %d\n"  
00401025    call    printf  
0040102A    add     esp, 8  
0040102D    jmp     short loc_40100D ❼
```







While循环

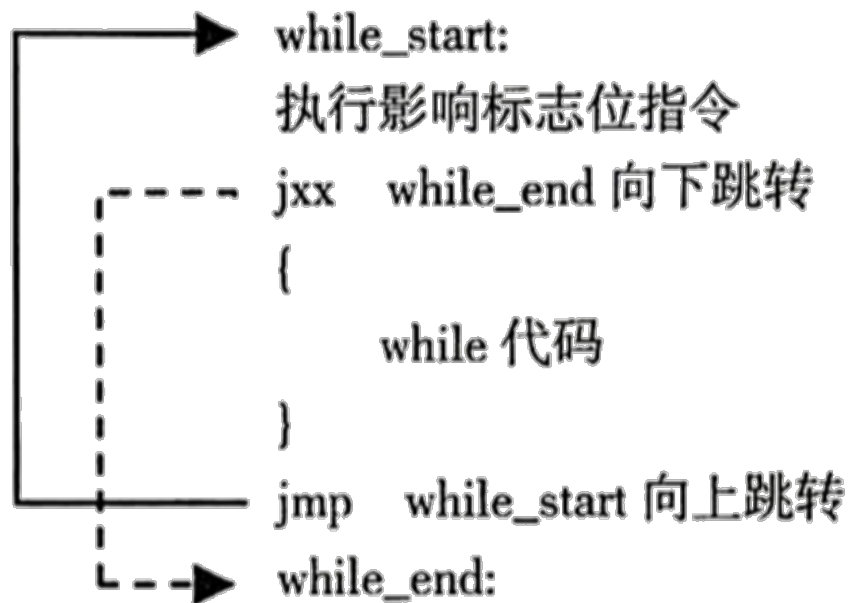
int status=0;	00401036	mov	[ebp+var_4], 0
int result = 0;	0040103D	mov	[ebp+var_8], 0
	00401044	loc_401044:	
while(status == 0){	00401044	cmp	[ebp+var_4], 0
result = performAction();	00401048	jnz	short loc_401063 ❶
status = checkResult (result);	0040104A	call	performAction
}	0040104F	mov	[ebp+var_8], eax
	00401052	mov	eax, [ebp+var_8]
	00401055	push	eax 返回值默认存在EAX中
	00401056	call	checkResult
	0040105B	add	esp, 4
	0040105E	mov	[ebp+var_4], eax
	00401061	jmp	short loc_401044 ❷





允公允能 日新月异

While循环的识别特征





Do循环

兼容64位编译器，未用push

```
#include "stdafx.h"
```

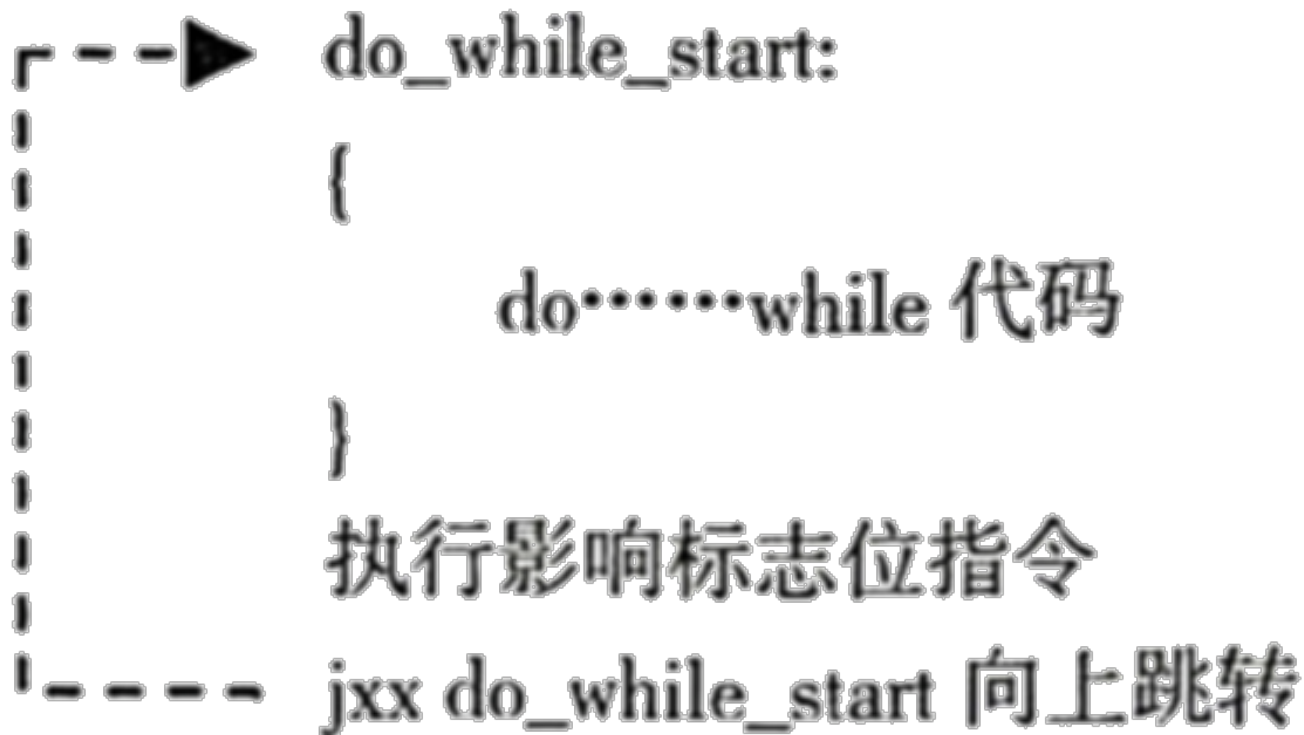
```
int _tmain(int argc, _TCHAR* argv[]) {  
    int nCount = 0;  
    do  
    {  
        printf("%d\r\n", nCount);  
        nCount++;  
    } while (nCount < argc);  
  
    return 0;  
}
```

```
mov     edx, [rsp+20h]      ;参数 2: edx=nCount  
lea     rcx, asc_14000678C ;参数 1: "%d\r\n"  
call    cs:printf          ;调用 printf 函数  
mov     eax, [rsp+20h]  
inc     eax  
mov     [rsp+20h], eax      ;nCount=nCount+1  
mov     eax, [rsp+40h]      ;eax=argc  
cmp     [rsp+20h], eax  
jl      short loc_140001039 ;if (nCount<argc), 跳转到 do 循环开始
```





Do循环的识别特征





允公允能 日新月异

本章知识点

1. 识别函数
2. 识别变量、数组、结构体
3. 识别IF分支结构
4. 识别Switch结构
5. 识别循环结构



南开大学
Nankai University

```
public start
start proc near
push    offset Format    ; "Please enter a challenge: "
call    ds:printf
add     esp, 4
push    offset Str
push    offset aS        ; "%S"
call    ds:scanf
add     esp, 8
push    offset Str       ; Str
call    ds:strlen
add     esp, 4
cmp     eax, 6
jb      loc_40110D
```

```
push    offset aPleaseEnterThe ; "Please enter the solution: "
call    ds:printf
add     esp, 4
push    offset dword_4030AD
push    offset dword_4030A9
push    offset dword_4030A5
push    offset dword_4030A1
push    offset aUUUU        ; "%u-%u-%u-%u"
call    ds:scanf
add     esp, 14h
cmp     eax, 4
jb      loc_40111D
```





```
push    offset a0000      ; "%u-%u-%u-%u"
call    ds:scanf
add     esp, 14h
cmp     eax, 4
jb      loc_40111D
```

```
movzx   eax, byte_4030B2
movzx   ecx, byte_4030B4
add     eax, ecx
movzx   ecx, byte_4030B5
add     eax, ecx
cmp     eax, dword_4030A1
jnz     loc_40111D
```

```
mov     eax, dword_4030A5
add     eax, 18h
not     eax
cmp     eax, 0BADF000Dh
jnz     short loc_40111D
```

```
mov     eax, dword_4030A9
mov     ecx, 0C48h
cdq
div     ecx
mov     esi, eax
movzx   eax, Str
movzx   ecx, byte_4030B3
mul     ecx
cmp     eax, esi
jnz     short loc_40111D
```



```
push    offset Str      ; Str
call    ds:strlen
add     esp, 4
mov     ecx, eax
sub     eax, eax
xor     edx, edx
mov     edi, dword_4030AD
xor     edi, 31337h
```

```
loc_4010EA:
cmp     edx, ecx
jnb     short loc_4010FA
```

```
movzx   ebx, Str[edx]
add     eax, ebx
inc     edx
jmp     short loc_4010EA
```

```
loc_4010FA:
sub     edi, 7Bh ; '{'
cmp     eax, edi
jnz     short loc_40111D
```

```
mov     dword_4030D1, 1
jmp     short loc_40111D
```

```
loc_40111D:
cmp     dword_4030D1, 0
```

```
loc_4010FA:  
sub     edi, 7Bh ; '{'  
cmp     eax, edi  
jnz     short loc_40111D
```

```
mov     dword_4030D1, 1  
jmp     short loc_40111D
```

```
loc_40111D:  
cmp     dword_4030D1, 0  
jz      short loc_401136
```

```
push    offset aCongratulation ; "Congratulations, you made it!\n\r"  
call    ds:printf  
add     esp, 4  
jmp     short locret_401144
```

```
loc_401136:  
push    offset aWrong ; "Wrong :(\n\r"  
call    ds:printf  
add     esp, 4
```

```
locret_401144:  
retn  
start endp
```

