What a 'code construct' means:

According to the book; a *code construct* is a code abstraction level that defines a functional property but not the details of its implementation. It is to say, a *code construct* references the class of manners in which a pseudo-code can be implemented. This way, python's or C's or Fortran's syntaxes are specific instances of *code construct*. Amongst *code construct instructions* these include, loops, conditional statement, switch statement and so on.

Aspects that may impact the way assembly code is generated:

1- *Compiler's versions and settings* may impact how a particular code construct appears in *disassembly*.
2- *Computer's architecture* may also change assembly code translation's appearance, insofar, compilers perform translation according to host's architecture. Now, considering retro-compatibility between x86, 64 or 32 bits' architectures, this may be not so bad whereas it is performing a x86 translation, but absolutely disastrous otherwise, so that disassembly may look completely different.
3- *Programming language* definitely impact on how assembly code is generated. Recall, some programming languages are compiled, while others are interpreted. This is especially notorious in memory's administration, having different programming languages to lead to completely different memory allocation.
4- *Operative system*. With each operative system come different *register keys*, nonetheless, translation can occur in different ways depending on what the OS orders to hardware.

Global vs Local Variables:

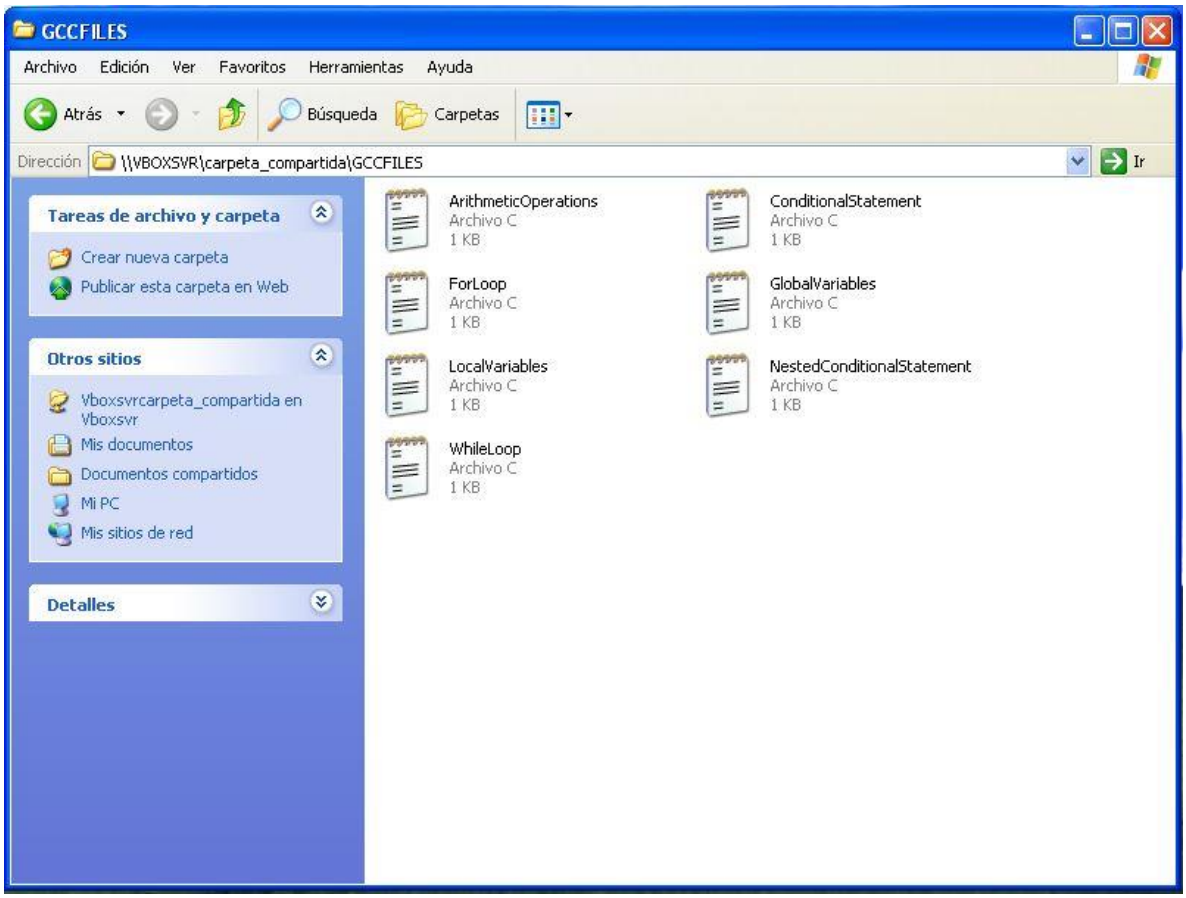
Global Variables can be accessed and used by any function in a program. Unlike, *Local Variables* can be accessed only by the function in which they are defined.

Both global and local variables are declared similarly in C (and many other programming languages), but they look completely different in assembly's translation.

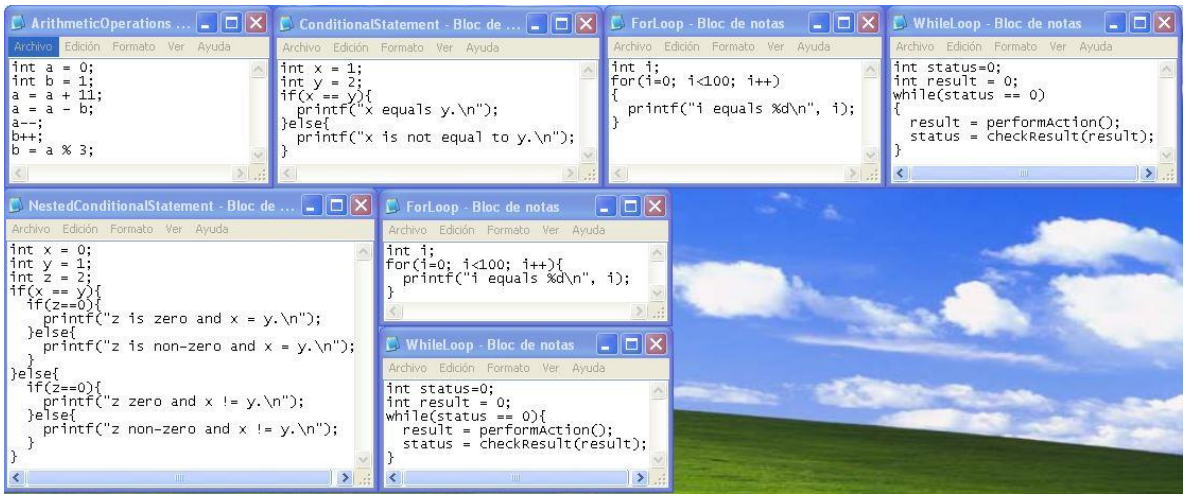
Global variables compilation's assembly	***Local variables compilation's assembly***
Global variables are referenced by memory addresses.	Local variables are referenced by the stack addresses.

Practical evidence

What we do now is write a *C code instance* for every *code construct*, such as *arithmetic operations, conditional statements, loops* and *global and local variables*.

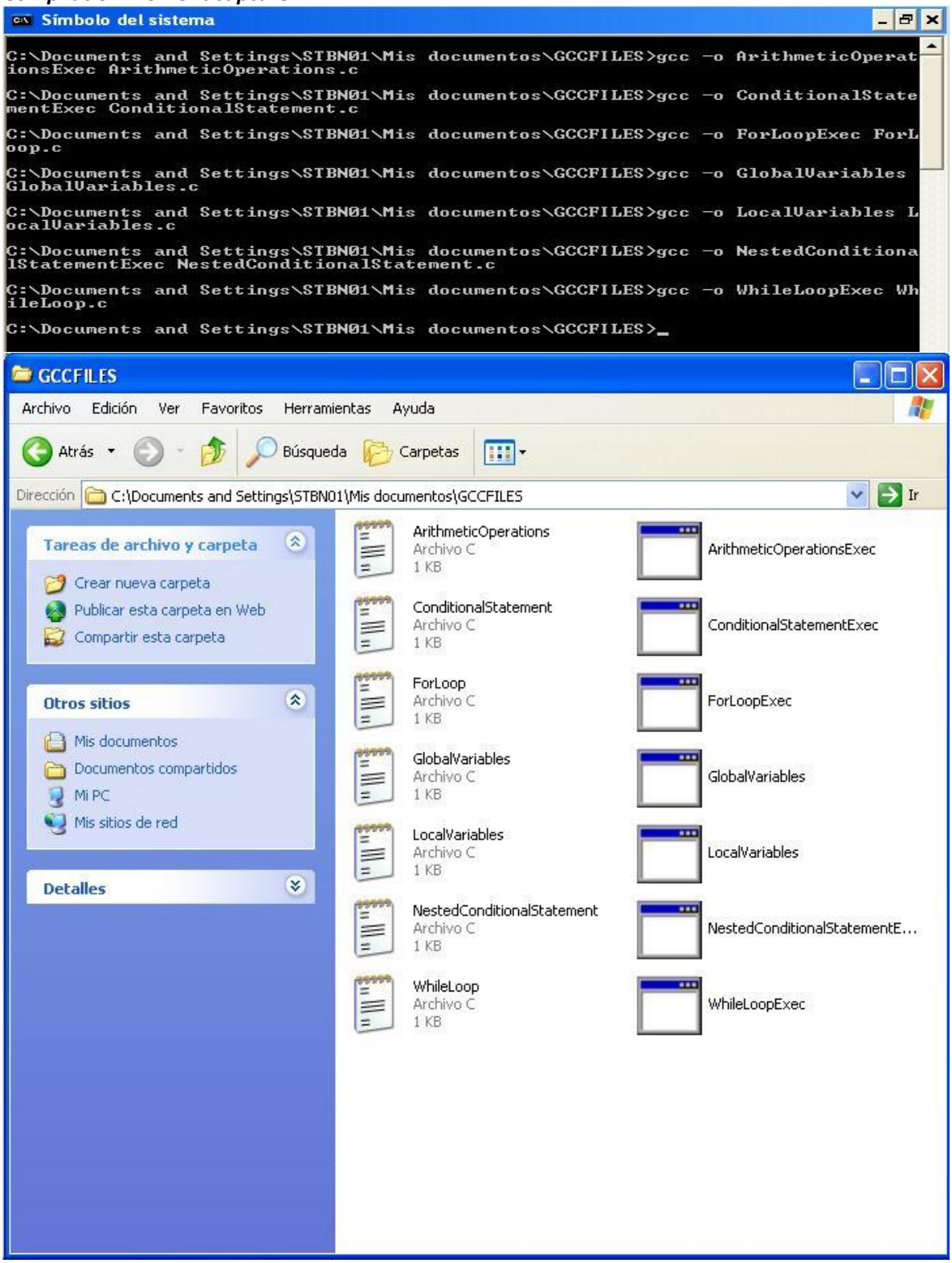


Below you can see every particular instance for every code construct.



Next, we compile each one of these with a C compiler. This way, we will get an assembly code translation which to disassembly for each program. With this in mind, let's find out what every code instruction looks like in assembly language.

Compilation moment capture



Arithmetic operations' assembly revision.

```
IDA View-A
.text:00401419                 call    sub_4019C0
.text:0040141E                 mov     [esp+10h+var_4], 0
.text:00401426                 mov     [esp+10h+var_8], 1
.text:0040142E                 add     [esp+10h+var_4], 0Bh
.text:00401433                 mov     eax, [esp+10h+var_8]
.text:00401437                 sub     [esp+10h+var_4], eax
.text:0040143B                 sub     [esp+10h+var_4], 1
.text:00401440                 add     [esp+10h+var_8], 1
.text:00401445                 mov     ecx, [esp+10h+var_4]
.text:00401449                 mov     edx, 55555556h
.text:0040144E                 mov     eax, ecx
.text:00401450                 imul    edx
.text:00401452                 mov     eax, ecx
.text:00401454                 sar     eax, 1Fh
.text:00401457                 sub     edx, eax
.text:00401459                 mov     eax, edx
.text:0040145B                 add     eax, eax
.text:0040145D                 add     eax, edx
.text:0040145F                 sub     ecx, eax
.text:00401461                 mov     eax, ecx
.text:00401463                 mov     [esp+10h+var_8], eax
.text:00401467                 nop
.text:00401468                 leave
.text:00401469                 retn
0000082E    0040142E: sub_401410+1E
```

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4] ❶
00401017        add     eax, 0Bh
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        sub     ecx, [ebp+var_8] ❷
00401023        mov     [ebp+var_4], ecx
00401026        mov     edx, [ebp+var_4]
00401029        sub     edx, 1 ❸
0040102C        mov     [ebp+var_4], edx
0040102F        mov     eax, [ebp+var_8]
00401032        add     eax, 1 ❹
00401035        mov     [ebp+var_8], eax
00401038        mov     eax, [ebp+var_4]
0040103B        cdq
0040103C        mov     ecx, 3
```

On the left we can see book's results for disassembly of arithmetic operations. Next, we consider a comparison table which entries will correspond to differences that are highlighted in red and similarities which are highlighted in green. Keep in mind: we might consider green contiguous boxes as equivalent representations under transformations. By this, we mean literally, it is just a copy paste but changing some of the names for the variables and addresses or lines' number.

Practical laboratory	***Practical Malware Analysis book***
Second and third lines: *Immediate operands 0 and 1 were moved into memory addresses [esp+10h+var_4] and [esp+10h+var_8], respectively.*	First and second lines: *Immediate operands 0 and 1 were moved into memory addresses [ebp+var_4] and [ebp+var_8], respectively.*
Fourth line: *Bit-wise XOR addition is made right after <u>two</u> 'mov' instructions. This is because it is adding an immediate operand oBh onto [esp+10h+var_4] containing the 0 value (which might be equivalent to moving oBh value into [esp+10h+var_4]), before moving the value in [esp+10h+var_8] to a third memory address eax*	Fourth line: *Bit-wise XOR addition is made right after three <u>'mov'</u> instructions. This is because it is moving the value in [ebp+var_4] containing a 0 to a third memory address eax, before adding an immediate operand oBh onto eax containing a 0 (which might be equivalent to moving oBh value into eax), then the value stored in eax is moved into [ebp+var_4], now, given 'mov' instruction application is 'reflexive' and 'transitive', we get that the above is equivalent to moving oBh value into [ebp+var_4] before moving the value in [ebp+var_4] to a third memory address eax*

Conditional statements' assembly revision

```
IDA View-A
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 20h        ; char *
.text:00401419                 call    sub_4019B0
.text:0040141E                 mov     [esp+20h+var_4], 1
.text:00401426                 mov     [esp+20h+var_8], 2
.text:0040142E                 mov     eax, [esp+20h+var_4]
.text:00401432                 cmp     eax, [esp+20h+var_8]
.text:00401436                 jnz     short loc_401446
.text:00401438                 mov     [esp+20h+var_20], offset aXEqualsY_ ; "x equals y."
.text:0040143F                 call    puts
.text:00401444                 jmp     short loc_401452
.text:00401446 ; ---------------------------------------------------------------------------
.text:00401446
.text:00401446 loc_401446:                             ; CODE XREF: sub_401410+26↑j
.text:00401446                 mov     [esp+20h+var_20], offset aXIsNotEqualToY ; "x is not equal to y."
.text:0040144D                 call    puts
.text:00401452
.text:00401452 loc_401452:                             ; CODE XREF: sub_401410+34↑j
.text:00401452                 nop
.text:00401453                 leave
.text:00401454                 retn
.text:00401454 sub_401410      endp
00000832   00401432: sub_401410+22
```

```
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
```

Practical laboratory	***Practical Malware Analysis book***
Sixth and seventh lines: *Immediate operands 1 and 2 were moved into memory addresses [esp+10h+var_4] and [esp+10h+var_8], respectively.*	First and second lines: *Immediate operands 1 and 2 were moved into memory addresses [ebp+var_8] and [ebp+var_4], respectively.*
Eight line: *value 1 stored in [esp+10h+var_4] is moved into a third memory address eax.*	Third line: *value 1 stored in [ebp+var_8] is moved into a third memory address eax.*
Ninth line: bit-wise comparison (cmp) is applied over operands eax and *[esp+10h+var_8], whereas last is storing value 2.*	Fourth line: bit-wise comparison (cmp) is applied over operands eax and *[ebp+var_4], whereas last is storing value 2.*
Tenth line: jump if not zero is applied over operand short loc_401446. Which is to say, in case both numbers are different, then it jumps up to short loc_401446.	Fifth line: jump if not zero is applied over operand short loc_40102B. Which is to say, in case both numbers are different, then it jumps up to short loc_40102B.
Eleventh line: mov instruction is applied over operands [esp+20h+var_20] and off_set aXequalsY in that order, specifically. Therefore…	Sixth line: off_set aXequalsY is pushed into stack. Therefore…
Twelve and fourteenth lines: push function is called.	Seventh and eleventh lines: printf function is called.

Conditional statements' assembly revision (Nested version)

```
IDA View-A
.text:0040141E                 mov     [esp+20h+var_4], 0
.text:00401426                 mov     [esp+20h+var_8], 1
.text:0040142E                 mov     [esp+20h+var_C], 2
.text:00401436                 mov     eax, [esp+20h+var_4]
.text:0040143A                 cmp     eax, [esp+20h+var_8]
.text:0040143E                 jnz     short loc_401463
.text:00401440                 cmp     [esp+20h+var_C], 0
.text:00401445                 jnz     short loc_401455
.text:00401447                 mov     [esp+20h+var_20], offset aZIsZeroAndXY_ ; "z is zero and x = y."
.text:0040144E                 call    puts
.text:00401453                 jmp     short loc_401484
.text:00401455 ; ---------------------------------------------------------------------------
.text:00401455
.text:00401455 loc_401455:                             ; CODE XREF: sub_401410+35↑j
.text:00401455                 mov     [esp+20h+var_20], offset aZIsNonZeroAndX ; "z is non-zero and x = y."
.text:0040145C                 call    puts
.text:00401461                 jmp     short loc_401484
.text:00401463 ; ---------------------------------------------------------------------------
.text:00401463
.text:00401463 loc_401463:                             ; CODE XREF: sub_401410+2E↑j
.text:00401463                 cmp     [esp+20h+var_C], 0
.text:00401468                 jnz     short loc_401478
.text:0040146A                 mov     [esp+20h+var_20], offset aZZeroAndXY_ ; "z zero and x != y."
.text:00401471                 call    puts
.text:00401461                 jmp     short loc_401484
.text:00401463 ; ---------------------------------------------------------------------------
.text:00401463
.text:00401463 loc_401463:                             ; CODE XREF: sub_401410+2E↑j
.text:00401463                 cmp     [esp+20h+var_C], 0
.text:00401468                 jnz     short loc_401478
.text:0040146A                 mov     [esp+20h+var_20], offset aZZeroAndXY_ ; "z zero and x != y."
.text:00401471                 call    puts
.text:00401476                 jmp     short loc_401484
.text:00401478 ; ---------------------------------------------------------------------------
.text:00401478
.text:00401478 loc_401478:                             ; CODE XREF: sub_401410+58↑j
.text:00401478                 mov     [esp+20h+var_20], offset aZNonZeroAndXY_ ; "z non-zero and x != y."
.text:0040147F                 call    puts
.text:00401484
.text:00401484 loc_401484:                             ; CODE XREF: sub_401410+43↑j
.text:00401484                                         ; sub_401410+51↑j ...
.text:00401484                 nop
.text:00401485                 leave
.text:00401486                 retn
.text:00401486 sub_401410      endp
00000863   00401463: sub_401410:loc_401463
```

```
00401006        mov     [ebp+var_8], 0
0040100D        mov     [ebp+var_4], 1
00401014        mov     [ebp+var_C], 2
0040101B        mov     eax, [ebp+var_8]
0040101E        cmp     eax, [ebp+var_4]
00401021        jnz     short loc_401047 ❶
00401023        cmp     [ebp+var_C], 0
00401027        jnz     short loc_401038 ❷
00401029        push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E        call    printf
00401033        add     esp, 4
00401036        jmp     short loc_401045
00401038 loc_401038:
00401038        push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D        call    printf
00401042        add     esp, 4
00401045 loc_401045:
00401045        jmp     short loc_401069
00401047 loc_401047:
00401047        cmp     [ebp+var_C], 0
0040104B        jnz     short loc_40105C ❸
0040104D        push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052        call    printf
00401057        add     esp, 4
0040105A        jmp     short loc_401069
0040105C loc_40105C:
0040105C        push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061        call    printf00401061
```

Now, every nested conditional statement is a conditional statement, meaning, conditional statement's analysis made before is applicable to this particular case. And so, we omit writing down its analysis, insofar, it would be nothing but redundant and irrelevant in this particular discussion. Furthermore, these disessamblations are extensive enough to describe than in detail. However, we still compare both images and find out some particular either not so meaningful differences between both.

For loop statement's assembly revision

```
IDA View-A
.text:00401410                 push    ebp
.text:00401411                 mov     ebp, esp
.text:00401413                 and     esp, 0FFFFFFF0h
.text:00401416                 sub     esp, 20h        ; char *
.text:00401419                 call    sub_4019A0
.text:0040141E                 mov     [esp+20h+var_4], 0
.text:00401426                 jmp     short loc_401441
.text:00401428 ; ---------------------------------------------------------------------------
.text:00401428
.text:00401428 loc_401428:                             ; CODE XREF: sub_401410+36↓j
.text:00401428                 mov     eax, [esp+20h+var_4]
.text:0040142C                 mov     [esp+20h+var_1C], eax
.text:00401430                 mov     [esp+20h+var_20], offset aIEqualsD ; "i equals %d\n"
.text:00401437                 call    printf
.text:0040143C                 add     [esp+20h+var_4], 1
.text:00401441
.text:00401441 loc_401441:                             ; CODE XREF: sub_401410+16↑j
.text:00401441                 cmp     [esp+20h+var_4], 63h
.text:00401446                 jle     short loc_401428
.text:00401448                 nop
.text:00401449                 nop
.text:0040144A                 leave
.text:0040144B                 retn
.text:0040144B sub_401410      endp
0000083C    0040143C: sub_401410+2C
```

```
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
```

Practical laboratory	***Practical Malware Analysis book***
Sixth line: immediate operand *value 0 is moved into memory address [esp+20h+var_4].*	First line: immediate operand *value 0 is moved into memory address [ebp+var_4].*
Seventh line: *Jumps up to process short loc_401441.*	Second line: *Jumps up to process short loc_401016.*
Seventeenth line: Perform bit-wise comparison with that stored at *[esp+20h+var_4] and whatever immediate operand value 63h represents. We may suspect 63h is actually storing the bounds' limit condition for the loop.*	Eight line: Perform bit-wise comparison with that stored at *[ebp+var_4] and whatever immediate operand value 64h represents. We may suspect 64h is actually storing the bounds' limit condition for the loop.*
Eighteenth line: Jumps up to short loc_401428 if previous comparison throws 'not greater than' between evaluating input *[esp+20h+var_4] and fixed condition 63h.*	Ninth line: Jumps up to short loc_40102F if previous comparison throws 'not greater than' between evaluating input *[ebp+var_4] and fixed condition 64h.*
Fifteenth line: performs addition over operand *[esp+20h+var_4]* and immediate operand 1. We may suspect this is counter progression.	Sixth line: performs addition over operand *[ebp+var_4]* and immediate operand 1. We may suspect this is counter progression.

While loop statement's assembly revision

```
IDA View-A
.text:00401428                 and     esp, 0FFFFFFF0h
.text:0040142B                 sub     esp, 20h
.text:0040142E                 call    sub_4019C0
.text:00401433                 mov     [esp+20h+var_4], 0
.text:0040143B                 mov     [esp+20h+var_8], 0
.text:00401443                 jmp     short loc_40145E
.text:00401445 ; ---------------------------------------------------------------------------
.text:00401445
.text:00401445 loc_401445:                             ; CODE XREF: sub_401425+3E↓j
.text:00401445                 call    sub_401410
.text:0040144A                 mov     [esp+20h+var_8], eax
.text:0040144E                 mov     eax, [esp+20h+var_8]
.text:00401452                 mov     [esp+20h+var_20], eax
.text:00401455                 call    sub_40141A
.text:0040145A                 mov     [esp+20h+var_4], eax
.text:0040145E
.text:0040145E loc_40145E:                             ; CODE XREF: sub_401425+1E↑j
.text:0040145E                 cmp     [esp+20h+var_4], 0
.text:00401463                 jz      short loc_401445
.text:00401465                 nop
.text:00401466                 nop
.text:00401467                 leave
.text:00401468                 retn
.text:00401468 sub_401425      endp
00000843   00401443: sub_401425+1E
```

```
00401036          mov     [ebp+var_4], 0
0040103D          mov     [ebp+var_8], 0
00401044 loc_401044:
00401044          cmp     [ebp+var_4], 0
00401048          jnz     short loc_401063
0040104A          call    performAction
0040104F          mov     [ebp+var_8], eax
00401052          mov     eax, [ebp+var_8]
00401055          push    eax
00401056          call    checkResult
0040105B          add     esp, 4
0040105E          mov     [ebp+var_4], eax
00401061          jmp     short loc_401044
```

Practical laboratory	***Practical Malware Analysis book***
Fourth line: *mov operation is applied over [esp+20h+var_4] and immediate operand 0 in that specific order. Meaning, value 0 is being moved into [esp+20h+var_4]. Something identical occurs with [esp+20h+var_8] at the fifth line.*	Fourth line: *mov operation is applied over [esp+var_4] and immediate operand 0 in that specific order. Meaning, value 0 is being moved into [ebp+var_4]. Something identical occurs with [ebp+var_8] at the second line.*
Sixth line: *program makes a jump up to process short loc_40145E.*	Third line: *Program accesses process short loc_401044.*
Eighteenth line: *a bit-wise comparison is applied over operands [esp+20h+var_4] and 0.*	Fifth line: *a bit-wise comparison is applied over operands [esp+var_4] and 0.*
Nineteenth line: *makes a jump if zero on operand short loc_40145E, so that 'jumps back' to iterate insofar it will perform this action again, until jump if not zero becomes true.*	Nineteenth line: *makes a jump if zero on operand short loc_401044, so that 'jumps back' to iterate insofar it will perform this action again, until jump if not zero becomes true.*