Universidad del Rosario

MACC
Matemáticas Aplicadas y Ciencias de la Computación

**Rosario University.**

# 6th Laboratory: IDAPro and IDC scripts

*September 14, 2020* *Andrés Felipe Florián Quitián.*

## 1 Code Constructs

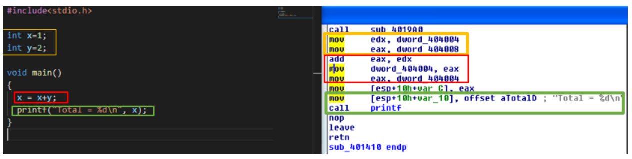
Code construct defines a functional property, but this does not give the details of its implementation, some code constructs include if statements, loops, switch. Programs can be done in blocks that combined give the structure and the functionality of the program. Nevertheless, the assembly code could change due to the compiler, and the language could be different, also the architecture can change some things of the assembly.

## 2 IDC scripts

### Global and Local Variables

Global variables could be used by any function in a program, unlike local variables can be accessed only in the function where they are defined, commonly they are defined very similarly, although in the assembly they a lot of changes. On the other hand, global variables are referenced as memory addresses and local variables by stack addresses.

With *IDAPro* we obtained the assembly of codes done in C and analyzed the differences between these, one has global variables and the other has local variables.

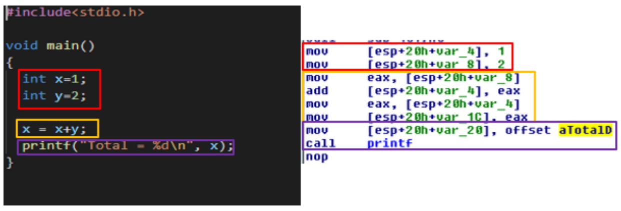


```c
#include<stdio.h>

int x=1;
int y=2;

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

```
call    sub_4019A0
mov     edx, dword_404004
mov     eax, dword_404008
add     eax, edx
mov     dword_404004, eax
mov     eax, dword_404004
mov     [esp+10h+var_C], eax
mov     [esp+10h+var_10], offset aTotalD ; "Total = %d\n"
call    printf
nop
leave
retn
sub_401410 endp
```

Figure 1: Global Variables represented in assembly

In Fig. 1 you can see the C code and its assembly, the structure of the code could be identified with the squares surrounding each part of it. The yellow square gives the global variables, we saw that the these in the assembly code are identified as a *dword* variable.



```c
#include<stdio.h>

void main()
{
  int x=1;
  int y=2;

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

```
mov     [esp+20h+var_4], 1
mov     [esp+20h+var_8], 2
mov     eax, [esp+20h+var_8]
add     [esp+20h+var_4], eax
mov     eax, [esp+20h+var_4]
mov     [esp+20h+var_1C], eax
mov     [esp+20h+var_20], offset aTotalD
call    printf
nop
```

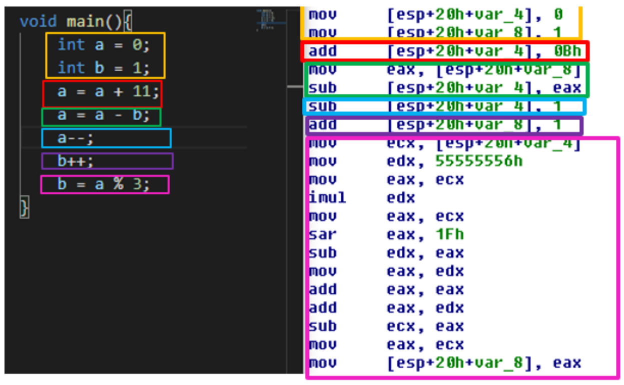
Figure 2: Local Variables represented in assembly

On the other hand, in Fig. 2 is represented a code with local variables, in the red square, you can see how are defined these variables, unlike the global variables these are identified with their names in the assembly code, due to they are variables storage in the stack, while the global variables are reference as memory addresses.

Moreover, in the images presented before, is possible to see how a value reassignment of a variables is done and how the *printf* works in the assembly code.

## Arithmetic operations

In Fig. 3 you can see a C code and its assembly, it was obtained with *IDAPro*.



```c
void main(){
    int a = 0;
    int b = 1;
    a = a + 11;
    a = a - b;
    a--;
    b++;
    b = a % 3;
}
```

```
mov     [esp+20h+var_4], 0
mov     [esp+20h+var_8], 1
add     [esp+20h+var_4], 0Bh
mov     eax, [esp+20h+var_8]
sub     [esp+20h+var_4], eax
sub     [esp+20h+var_4], 1
add     [esp+20h+var_8], 1
mov     ecx, [esp+20h+var_4]
mov     edx, 55555556h
mov     eax, ecx
imul    edx
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
add     eax, eax
add     eax, edx
sub     ecx, eax
mov     eax, ecx
mov     [esp+20h+var_8], eax
```

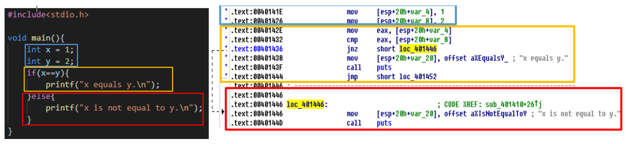
Figure 3: Arithmetic Operations in Assembly

The yellow square represents the definition of the variables $a$ and $b$, we can see that the addition is done with the command *add* it has the next structure: $ADD < dest >, < source >$ it adds source to dest, on the other hand, the *sub* command done the subtraction in this case of the variable $a = a - b$, it works pretty similar as *add*.

Moreover, for the operations $a - -$ and $b + +$, the command mentioned before preserve its structure, the only change is that in this case the dest is 1. Finally, in the pink square is shown the structure of a modulo operation.

## If statements

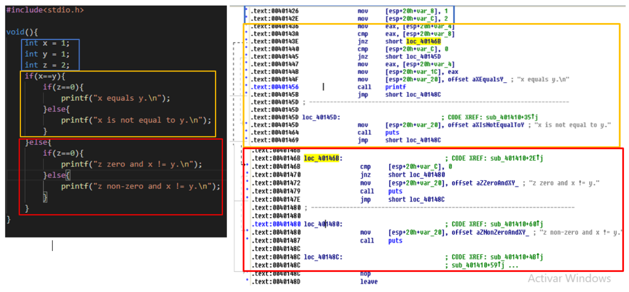
For the if statements were analyzed two codes one with an if, else statement and the other is nested if.



```
#include<stdio.h>

void main(){
    int x = 1;
    int y = 2;
    if(x==y){
        printf("x equals y.\n");
    }else{
        printf("x is not equal to y.\n");
    }
}
```

```
.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
```

(a) If statements



```
#include<stdio.h>

void(){
    int x = 1;
    int y = 1;
    int z = 2;
    if(x==y){
        if(z==0){
            printf("x equals y.\n");
        }else{
            printf("x is not equal to y.\n");
        }
    }else{
        if(z==0){
            printf("z zero and x != y.\n");
        }else{
            printf("z non-zero and x != y.\n");
        }
    }
}
```

```
.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_40146B
.text:00401440                 cmp     [esp+20h+var_C], 0
.text:00401445                 jnz     short loc_40145D
.text:00401447                 mov     eax, [esp+20h+var_4]
.text:0040144B                 mov     [esp+20h+var_1C], eax
.text:0040144F                 mov     [esp+20h+var_20], offset aXEqualsY_ ; "x equals y.\n"
.text:00401456                 call    printf
.text:0040145B                 jmp     short loc_40148C
.text:0040145D ; ---------------------------------------------------------------
.text:0040145D
.text:0040145D loc_40145D:                             ; CODE XREF: sub_401410+35↑j
.text:0040145D                 mov     [esp+20h+var_20], offset aXIsNotEqualToY ; "x is not equal to y."
.text:00401464                 call    puts
.text:00401469                 jmp     short loc_40148C
.text:0040146B
.text:0040146B loc_40146B:                             ; CODE XREF: sub_401410+2E↑j
.text:0040146B                 cmp     [esp+20h+var_C], 0
.text:00401470                 jnz     short loc_401480
.text:00401472                 mov     [esp+20h+var_20], offset aZZeroAndXY_ ; "z zero and x != y."
.text:00401479                 call    puts
.text:0040147E                 jmp     short loc_40148C
.text:00401480 ; ---------------------------------------------------------------
.text:00401480
.text:00401480 loc_401480:                             ; CODE XREF: sub_401410+60↑j
.text:00401480                 mov     [esp+20h+var_20], offset aZNonZeroAndXY_ ; "z non-zero and x != y."
.text:00401487                 call    puts
.text:0040148C
.text:0040148C loc_40148C:                             ; CODE XREF: sub_401410+4B↑j
.text:0040148C                                         ; sub_401410+59↑j ...
.text:0040148C                 nop
.text:0040148D                 leave
```

Activar Windows

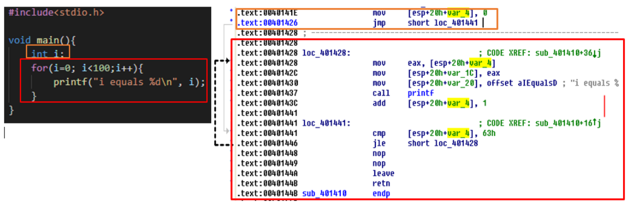
(b) Nested if Statements

In Fig. 4a you can see the if, else block, we are interested in the yellow and red squares. Yellow square give the structure of the if, in the part of the assembly code, we can see a new commands, these are *cmp* and *jnz*, *cmp* compare the value $x == y$ if it is true, it performs the code of the if, if not the command *jnz* identified a cero value, these means something false, which is why it jumps into the else section, we could see in the assembly an arrow pointong to it, specifically it jumps from *00401436* direction to *004014446*.

On the other hand, the nested if works very similar, but it has to judge two statements one for the first and the other for the next if, nested to it, this also happens to the else.

## Loops: for and while

First of all, we analyzed the for loop, in Fig. 6 is shown a C code and its assembly.

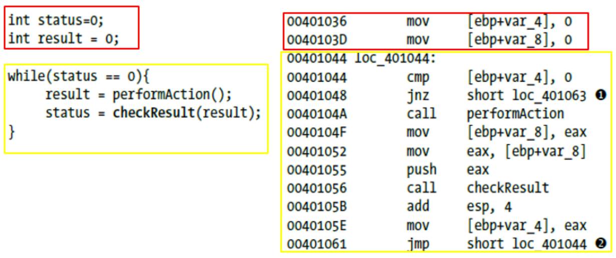


```
#include<stdio.h>

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

```
.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 %
.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
```

Figure 5: For loop

For this, the program first create the variable $i$, afterward it execute *jmp*, done this that we could identify a similar code saw before, nevertheless, it makes these instruction 100 times, due to the variable $i$ were defined in the interval $[0, 99]$, for that reason, *IDAPro* makes an arrow appointing to the section that does this, specifically, it is between *loc_401428* and *loc_401441*.



```
int status=0;
int result = 0;

while(status == 0){
      result = performAction();
      status = checkResult(result);
}
```

```
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 ❶
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 ❷
```

Figure 6: While loop (images obtained from Practical Malware Analysis, The hands on Guide to Dissecting Malicious Software)

On the other hand, the while loop works pretty similar the biggest difference is that this block works with a preposition, in the example showed before, we can see that the while is working with the value $status == 0$, which is why the assembly code uses *cmp*, to evaluate the truth value.

### Script codes

All the C codes were obtained in the book *Practical Malware Analysis, The hands on Guide to Dissecting Malicious Software.* Also, we identified that the assembly shown in the book is different from the one made by us, besides C is a compiled language, which is why the binary code must be different in all the structures.

## References

[1] Sikorski, Michael; Honig, Andrew. *Practical Malware Analysis, The hands on Guide to Dissecting Malicious Software.* No starch press, 2012.