Sixth Laboraory

Code Constructs

Sara Gallego Rivera

Universidad del Rosario

Escuela de Ingeniería, Ciencia y Tecnología

September 14^{th}, 2020

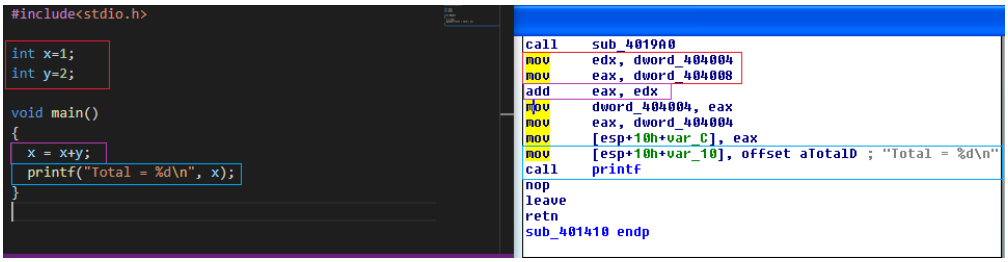
Code Construct

- Read the introduction of the section 6 "Recognizing C Code Constructs in Assembly" and explain what means a "Code Construct". What aspects may impact the way as assembly code is generated?

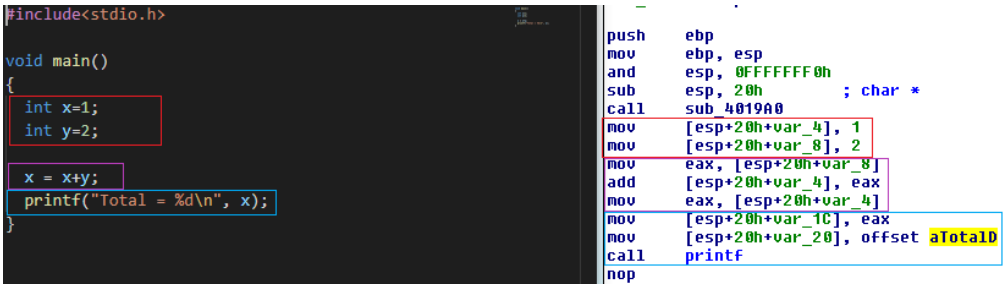
  * One of the aspects is the compiler, because compiler versions and settings can impact how a particular construct appears in disassembly. Also the programming language and the computer's architecture where the code is compiled.

- Read the section "Gobal vs Local Variables" and identify what are the differences in the compilation of a code that employs global vs one that employs local Variables.

  * Global variables can be accessed and used by any function in a program, while local variables can be accessed only by the function in which they are defined. The difference in the compilation between this two variables is that the global variables are referenced by memory addresses, and the local variables are referenced by the stack addresses.



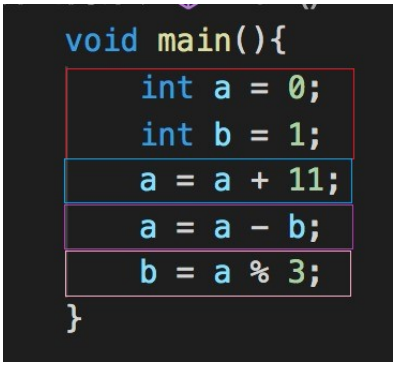
Global Variables



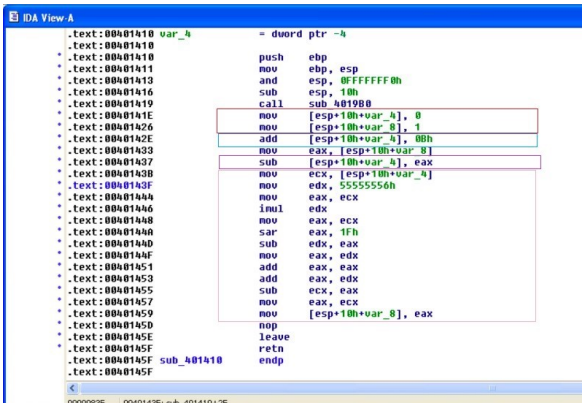
Local Variables

- Read the section "Disassembling Arithmetic Operations" and explain how the operations (addition, substraction, increment, decrement and modulo) are represented in assembly code.

  * In the red square we can see that a and b are local variables because they are referenced by the stack. IDA Pro has labeled a as var_4 and initialized it to 0, and b as var_8 and initialized it to 1. On the blue square we see that a is incremented by 11. Then b is moved into eax, and on the purple square eax (b) is subtracted from a. Finally, the instructions on the pink square implement the modulo.



C code



Assembly code

```
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

00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx ❺
```

Listing 6-7: Assembly code for the arithmetic example in Listing 6-6

Assembly code presented in the book

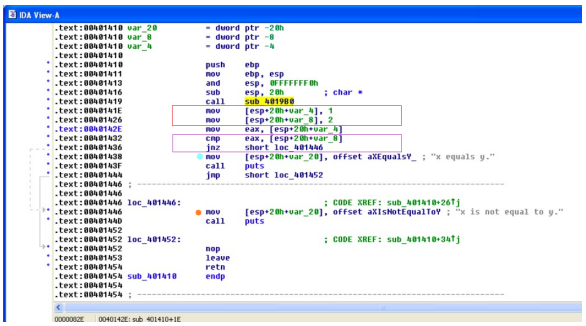
- Read the section "Recognizing if Statements" and explain how to recognize an if/else structure in assembly code.

  * In the red square we can see that x and y are local variables because they are referenced by the stack. IDA Pro has labeled x as var_4 and initialized it to 1, and y as var_8 and initialized it to 2. Then x is moved into eax. On the purple square there is a comparison (cmp) between x and y values and also there's a jump (jnz) that decides which path to take. This decision is made based on the comparison, which checks if the values of x and y are equal or not. If the values are not equal, the jump occurs (orange circle), and the code prints "x is not equal to y."; otherwise (blue circle), the code continues the path of execution and prints "x equals y."

```c
#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");
    }
}
```

C code



Assembly code

```
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
```

Listing 6-9: Assembly code for the if statement example in Listing 6-8

Assembly code presented in the book

In this example we can notice son differences between the assembly code we get and the one presented in the book. One of the differences is the use of the push function, that adds a value to the top of the stack.

- Read the section "Recognizing Nested if Statements" and explain how to recognize a "Nested IF" structure in assembly code.

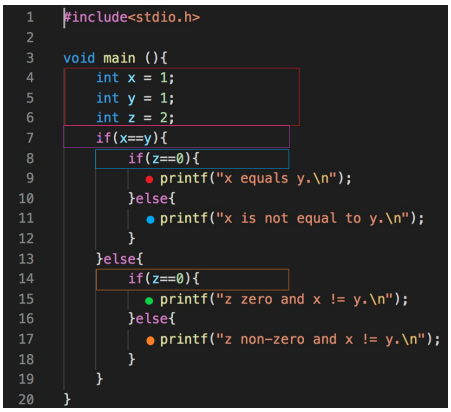
    * In the red square we can see that x, y and z are local variables because they are referenced by the stack. IDA Pro has labeled x as var_4 and initialized it to 1, y as var_8 and initialized it to 1, and z as var_C and initialized it to 2. Then x is moved into eax.

    On the purple square there is a comparison (cmp) between x and y values and also there's a jump (jnz) that decides which path to take. If the values are not equal, the jump occurs and the code jumps to the orange square; otherwise, the code continues the path of execution pass to the blue square.

    On the blue square there is a comparison (cmp) between z value and number 0 and also there's a jump (jnz). If the values are not equal, the jump occurs (blue circle) and the code prints "x is not equal to y."; otherwise (red circle), the code continues the path of execution and prints "x equals y."

    Finally, on the orange square, there is a comparison (cmp) between z value and number 0 and also there's a jump (jnz). If the values are not equal, the jump occurs (orange circle) and the code prints "z non-zero and x != y."; otherwise (green circle), the code continues the path of execution and prints "z zero and x != y."

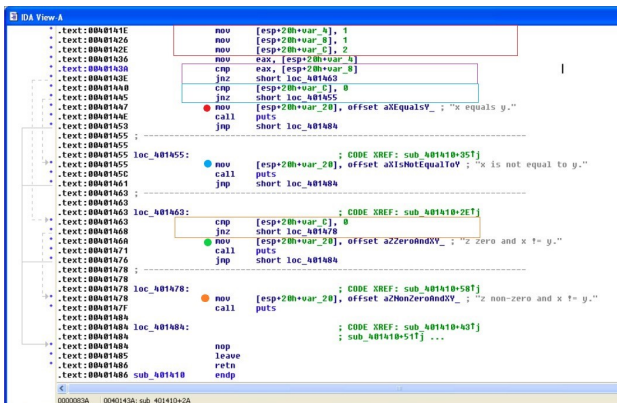
    and the code prints "x is not equal to y."; otherwise (blue circle), the code continues the path of execution and prints "x equals y."



```c
#include<stdio.h>

void main (){
    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");
        }
    }
}
```

C code



Assembly code

```
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
```

Listing 6-11: Assembly code for the nested if statement example shown in Listing 6-10

Assembly code presented in the book

In this example we can notice son differences between the assembly code we get and the one presented in the book. Some of the differences are the structure of the assembly code and the use of the push function, that adds a value to the top of the stack.

- Read the section "Recognizing Loops" and explain how to recognize a FOR structure in assembly code.

    * In the red square we can see that i is defined, it is a local variables because are referenced by the stack. Then on the pink square occurs increment of the variable, then the comparison, and finally a jump (jle) that decides the path to take based on the condition i<100 , if the jump is not taken, the printf instruction will execute, and starts the loop

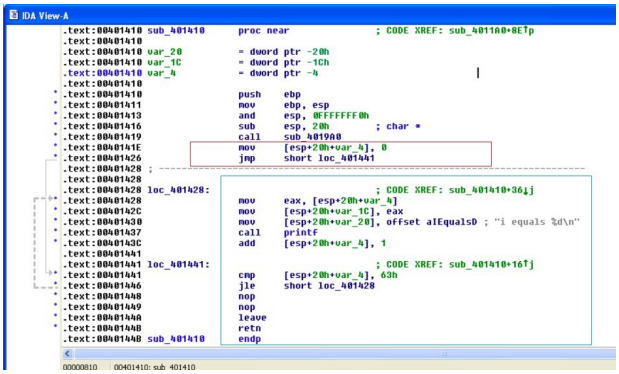
again.

We can recognized this type of loop by locating the four components: initialization, comparison, execution instructions, and increment/decrement.

```c
#include<stdio.h>

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

C code



Assembly code

```
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
```

Listing 6-13: Assembly code for the for loop example in Listing 6-12

Assembly code presented in the book

- Read the section "Recognizing Loops" and explain how to recognize a WHILE structure in assembly code.

  * The assembly code of these loop is similar to the for loop, except that it lacks an increment section. A conditional jump (jnz) occurs at the blue square and an unconditional jump at green square, but the only way for this code to stop executing repeatedly is for that conditional jump to occur.

```c
#include<stdio.h>

void main(){
    int status = 0;
    int result = 0;

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

C code

```
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
```

Listing 6-15: Assembly code for the while loop example in Listing 6-14

Assembly code presented by the book

Bibliography

- *Practical Malware Analysis, The hands-on Guide to Dissecting Malicious Software. Michael Sikorski and Andrew Honing, No starch press, 2012.*