6th Laboratory

Juan E Murcia

September 2020

1 Code construct

1. 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?

   A code construct is an abstraction in which it's defined a functional property, but not its implementation. In the case of C code, code construct can help recognize variables, loops, if statements and so. There are several factors that can change how the assembly code is generated, on of them is the compiler used, two different C compilers can generate different assembly codes that reproduce the same instruction, the OS is also a key factor on how that assembly code is generated, and the most important, the architecture where the code is being compiled.

2. 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. Ref: Section "Registers" Pag 104, Section "The Stack" Pag 110, Section "Stack Layout" Pag 111. Section "C Main Method and Offsets" Pag 116.

   The following codes were the used to analyze the changes between local and global variables:

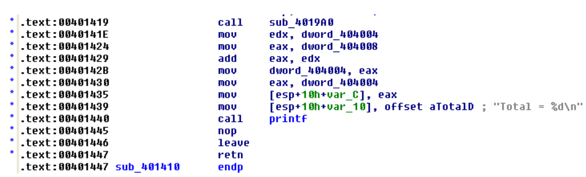
Listing 1: Global variables

```c
#include<stdio.h>

int x = 1;
int y = 2;

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

Listing 2: Local variables

```c
#include<stdio.h>

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

And the resulting assembly output was:



```
.text:00401419                 call    sub_4019A0
.text:0040141E                 mov     edx, dword_404004
.text:00401424                 mov     eax, dword_404008
.text:00401429                 add     eax, edx
.text:0040142B                 mov     dword_404004, eax
.text:00401430                 mov     eax, dword_404004
.text:00401435                 mov     [esp+10h+var_C], eax
.text:00401439                 mov     [esp+10h+var_10], offset aTotalD ; "Total = %d\n"
.text:00401440                 call    printf
.text:00401445                 nop
.text:00401446                 leave
.text:00401447                 retn
.text:00401447 sub_401410      endp
```

Figure 1: Assembly for global variables

```
.text:0040141E                 mov     [esp+20h+var_4], 1
.text:00401426                 mov     [esp+20h+var_8], 2
.text:0040142E                 mov     eax, [esp+20h+var_8]
.text:00401432                 add     [esp+20h+var_4], eax
.text:00401436                 mov     eax, [esp+20h+var_4]
.text:0040143A                 mov     [esp+20h+var_1C], eax
.text:0040143E                 mov     [esp+20h+var_20], offset aTotalD ; "Total = %d\n"
.text:00401445                 call    printf
.text:0040144A                 nop
.text:0040144B                 leave
.text:0040144C                 retn
.text:0040144C sub_401410      endp
```

Figure 2: Assembly for local variables

The clear difference is that global variables are stored in the data section of the memory and are called with memory variables like dword_404004 for x and dword_404008 for y, but when we see the local case, both values of the variables are put into the stack and then summed using their stack position. This makes sense because global variables are accessible for the entire code (storing those values at the data section) meanwhile the local case variables are only accessible during the function execution (storing them in the stack).

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

   The following code was used to test the arithmetic operations:

Listing 3: Arithmetic operations

```
#include<stdio.h>

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

And the generated assembly code was:

```
.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
.text:00401469 sub_401410      endp
```

Figure 3: Assembly for arithmetic operations

This case we find that simple arithmetic instructions are performed in a easy way, the assignation is performed with a mov to the stack because we used local variables, addition and substraction were performed with add and sub opcodes as expected, increments and decrements are an add or a sub, but the messy part comes with the modulo, where the compiler used several instruction to performn it, in which we can find sar (shifft arithmetic right) and imul a signed multiply, plenty of mov, add and sub.

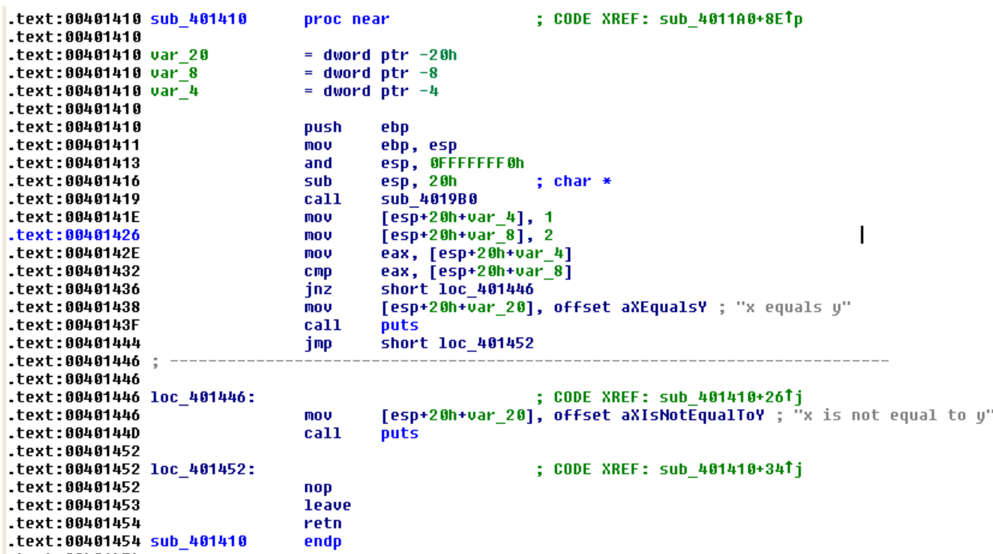
4. Read the section "Recognizing if Statements" and explain to your classmates how to recognize an if/else structure in assembly code.

   The following code was used to see the assembly of an if statement:

Listing 4: If statement

```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")
        }
}
```

And we obtained the next assembly code: We identify that x variable



```
.text:00401410 sub_401410      proc near               ; CODE XREF: sub_4011A0+8E↑p
.text:00401410
.text:00401410 var_20          = dword ptr -20h
.text:00401410 var_8           = dword ptr -8
.text:00401410 var_4           = dword ptr -4
.text:00401410
.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
```

Figure 4: Assembly if statements

is being stored in $[esp + 20h + var\_4]$ and y in $[esp + 20h + var\_8]$, then x is moved to eax so we can compare it with y, if ZF is not 0, or in other words, x is not equal to y, the code jumps to loc_401452, if they are equal we call the function puts with the argument "x equals y" and then it jumps to the end of the subroutine, in loc_401452 we find that the function puts is being called with argument "x is not equal to y" and continue to the end of the subroutine.

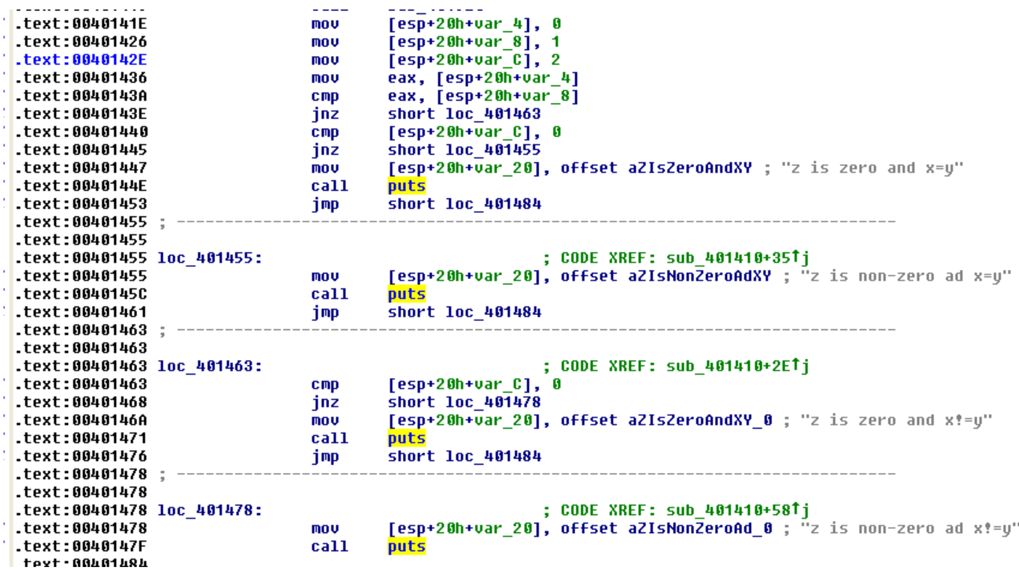
5. Read the section "Recognizing Nested if Statements" and explain to your classmates how to recognize a "Nested IF" structure in assembly code.

   The following code was used to see the assembly of nested ifs:

Listing 5: Nested if

```
#include<stdio.h>

void main() {
        int x = 0;
        int y = 1;
        int z = 2;
        if (x==y) {
                if (z==0) {
                        printf("z is zero and x=y\n");
                }
                else {
                        printf("z is non-zero and x=y\n");
                }
        }
        else {
                if (z==0) {
                        printf("z is zero and x!=y\n");
                }
                else {
                        printf("z is non-zero and x!=y\n");
                }
        }
}
```

   And we obtained the next assembly code:



```
.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 aZIsNonZeroAdXY ; "z is non-zero ad 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 aZIsZeroAndXY_0 ; "z is 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 aZIsNonZeroAd_0 ; "z is non-zero ad x!=y"
.text:0040147F                 call    puts
.text:00401484
```

Figure 5: Assembly nested if

Its obvious that the behavior is the same as the in the if case, just that in this particular case we also have a jnz instruction after the in a loc, that represents the nested ifs

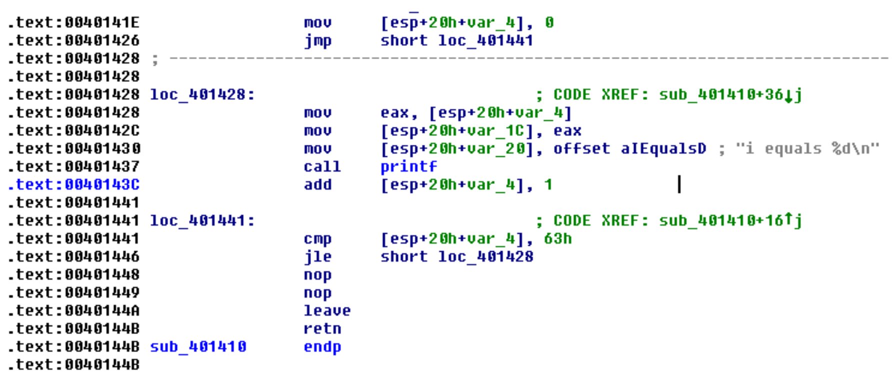
6. Read the section "Recognizing Loops" and explain to your classmates how to recognize a FOR structure in assembly code.

   The following code was used to see the assembly of nested ifs:

Listing 6: For

```
#include<stdio.h>

void main() {
        int i;

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

And we obtained the next assembly code:



```
.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
.text:0040144B
```

Figure 6: Assembly for

This case is interesting, because we clearly see that the variable i was not initialized, but the program stored 0 at its stack position, then it jumps to the loc that have the for, this loc compares i and 0x63 that is 99 decimal, and jumps if i is less or equals 99, inside the loc that the for jumps into, we see that it calls printf with the parameters "i equals%d" and i, then it increment by 1 i, and the code will continue its execution back to the for comparison, repeating the instructions until i es greater than 99.

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

   The following code was used to see the assembly of nested ifs:

Listing 7: While

```
#include<stdio.h>

void main() {
        int status=10;
        int i=0;

        while (status>10 \&\& i<0) {
                printf("i equals %d and status %d\n",i,status);
                i++;
                status-=2;
        }
}
```

And we obtained the next assembly code: What is interesting here, is

```
.text:0040141E                 mov     [esp+20h+var_4], 0Ah
.text:00401426                 mov     [esp+20h+var_8], 0
.text:0040142E                 jmp     short loc_401456
.text:00401430 ; ---------------------------------------------------------------------------
.text:00401430
.text:00401430 loc_401430:                             ; CODE XREF: sub_401410+52↓j
.text:00401430                 mov     eax, [esp+20h+var_4]
.text:00401434                 mov     [esp+20h+var_18], eax
.text:00401438                 mov     eax, [esp+20h+var_8]
.text:0040143C                 mov     [esp+20h+var_1C], eax
.text:00401440                 mov     [esp+20h+var_20], offset aIEqualsDAndSta ; "i equals %d and status %d\n"
.text:00401447                 call    printf
.text:0040144C                 add     [esp+20h+var_8], 1
.text:00401451                 sub     [esp+20h+var_4], 2
.text:00401456
.text:00401456 loc_401456:                             ; CODE XREF: sub_401410+1E↑j
.text:00401456                 cmp     [esp+20h+var_4], 0
.text:0040145B                 jle     short loc_401464
.text:0040145D                 cmp     [esp+20h+var_8], 4
.text:00401462                 jle     short loc_401430
.text:00401464
.text:00401464 loc_401464:                             ; CODE XREF: sub_401410+4B↑j
.text:00401464                 nop
.text:00401465                 leave
.text:00401466                 retn
.text:00401466 sub_401410      endp
```

Figure 7: Assembly while

that it has the same structure of the for, putting the instructions just before testing the exit clause, and the way it manages the logical ands making two comparisons.

8. Read the section "Understanding Function Call Conventions" and explain to your classmates how to recognize a "function call" in assembly code.

   The following code was used to see the assembly of nested ifs:

Listing 8: Function

```c
#include<stdio.h>

int sumar(int a, int b) {
        return a+b;
}

void main() {
        int a = 1;
        int b = 5;
        int sum = sumar(a,b);
        printf("La suma dio %d\n",sum);
```

```
}
```

And we obtained the next assembly code:

```
.text:0040142B                 mov     [esp+20h+var_4], 1
.text:00401433                 mov     [esp+20h+var_8], 5
.text:0040143B                 mov     eax, [esp+20h+var_8]
.text:0040143F                 mov     [esp+20h+var_1C], eax
.text:00401443                 mov     eax, [esp+20h+var_4]
.text:00401447                 mov     [esp+20h+var_20], eax
.text:0040144A                 call    sub_401410
.text:0040144F                 mov     [esp+20h+var_C], eax
.text:00401453                 mov     eax, [esp+20h+var_C]
.text:00401457                 mov     [esp+20h+var_1C], eax
.text:0040145B                 mov     [esp+20h+var_20], offset aLaSumaDioD ; "La suma dio %d\n"
.text:00401462                 call    printf
.text:00401467                 nop
.text:00401468                 leave
.text:00401469                 retn
.text:00401469 sub 40141D      endp
            sub_401410      proc near               ; CODE XREF: sub_40141D+2D↓p

            arg_0           = dword ptr  8
            arg_4           = dword ptr  0Ch

                            push    ebp
                            mov     ebp, esp
                            mov     edx, [ebp+arg_0]
                            mov     eax, [ebp+arg_4]
                            add     eax, edx
                            pop     ebp
                            retn
            sub_401410      endp
```

Figure 8: Assembly function

As expected, the compiler created a subroutine where the instructions of the function were stored, and in the moment we called the function in the code, the subroutine was called with the two arguments.

9. Read the section "Analyzing switch Statements" and explain to your classmates how to recognize a switch structure in assembly code.

   The following code was used to see the assembly of nested ifs:

Listing 9: Switch

```
#include<stdio.h>

void main() {
        int i = 0;
        switch(i):
                case 1:
                        printf("i = %d", i+1);
                        break;
```

```
        case 1:
                printf("i = %d", i+1);
                break;
        case 1:
                printf("i = %d", i+1);
                break;
        default:
                break;
}
```

And we obtained the next assembly code:

```
.text:0040141E                 mov     [esp+20h+var_4], 2
.text:00401426                 cmp     [esp+20h+var_4], 3
.text:0040142B                 jz      short loc_401476
.text:0040142D                 cmp     [esp+20h+var_4], 3
.text:00401432                 jg      short loc_40148F
.text:00401434                 cmp     [esp+20h+var_4], 1
.text:00401439                 jz      short loc_401444
.text:0040143B                 cmp     [esp+20h+var_4], 2
.text:00401440                 jz      short loc_40145D
.text:00401442                 jmp     short loc_40148F
.text:00401444 ; ---------------------------------------------------------------------------
.text:00401444
.text:00401444 loc_401444:                             ; CODE XREF: sub_401410+29↑j
.text:00401444                 mov     eax, [esp+20h+var_4]
.text:00401448                 add     eax, 1
.text:0040144B                 mov     [esp+20h+var_1C], eax
.text:0040144F                 mov     [esp+20h+var_20], offset aID ; "i = %d\n"
.text:00401456                 call    printf
.text:0040145B                 jmp     short loc_401490
.text:0040145D ; ---------------------------------------------------------------------------
.text:0040145D
.text:0040145D loc_40145D:                             ; CODE XREF: sub_401410+30↑j
.text:0040145D                 mov     eax, [esp+20h+var_4]
.text:00401461                 add     eax, 2
.text:00401464                 mov     [esp+20h+var_1C], eax
.text:00401468                 mov     [esp+20h+var_20], offset aID ; "i = %d\n"
.text:0040146F                 call    printf
.text:00401474                 jmp     short loc_401490
.text:00401476 ; -
.text:00401476
.text:00401476 loc_401476:                             ; CODE XREF: sub_401410+1B↑j
.text:00401476                 mov     eax, [esp+20h+var_4]
.text:0040147A                 add     eax, 3
.text:0040147D                 mov     [esp+20h+var_1C], eax
.text:00401481                 mov     [esp+20h+var_20], offset aID ; "i = %d\n"
.text:00401488                 call    printf
.text:0040148D                 jmp     short loc_401490
.text:0040148F ; ---------------------------------------------------------------------------
.text:0040148F
.text:0040148F loc_40148F:                             ; CODE XREF: sub_401410+22↑j
.text:0040148F                                         ; sub_401410+32↑j
.text:0040148F                 nop
.text:00401490
.text:00401490 loc_401490:                             ; CODE XREF: sub_401410+4B↑j
.text:00401490                                         ; sub_401410+64↑j ...
.text:00401490                 nop
.text:00401491                 leave
.text:00401492                 retn
.text:00401492 sub_401410      endp
.text:00401492
```

Figure 9: Assembly switch

It is interesting that it start making comparisons between i that is in the stack and the values 1, 2 and 3, if it's equal it jumps to here it is called the function printf, is it is greater than 3 or don't satisfy any comparison, it jumps to the nd of the subroutine to end execution.