6th Laboratory: Code Constructs

Isabella Martinez Martinez
School of Engineering, Science and Technology
Rosario University

1. A **code construct** is a code abstraction level that defines a functional property but not the details of its implementation. Notice that compiler versions and settings can impact how a particular construct appears in disassembly.
2. In assembly, the **global variables** are referenced by memory addresses, and the **local variables** are referenced by the stack addresses. Notice the assembly version of a C code with global variables:

```c
#include <stdio.h>

int x = 1;
int y = 2;

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

```asm
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
```

In this case, *dword_404004* is the variable *x,* and *dword_404008* is the variable *y.* We can see that internally the computer moves these variables to *eax* and *edx*, and it performs the sum there. Finally, it moves the result to *x*.

Now, the same code with local variables:

```c
#include <stdio.h>

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

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

```asm
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 ; "Total = %d\n"
call    printf
```

Here we can see that the computer saves the values of *x* and *y* in some stack addresses, in particular, memory address *[esp+20h+var_4]* refers to *x* and *[esp+20h+var_8]* refers to *y.* In this case it moves *y* to *eax* and then it sums *eax* to *x.*

3. Let us look at how some **basic arithmetic operations** are represented in assembly.

```c
void main(){
    int a = 0;
    int b = 1;

    a = a + 11;
    a = a - b;
    a--;
    b++;
    b = a % 3;
}
```

```asm
mov     [esp+10h+var_4], 0
mov     [esp+10h+var_8], 1
add     [esp+10h+var_4], 0Bh
mov     eax, [esp+10h+var_8]
sub     [esp+10h+var_4], eax
sub     [esp+10h+var_4], 1
add     [esp+10h+var_8], 1
mov     ecx, [esp+10h+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+10h+var_8], eax
```

Although the other operations are quite straightforward, it is difficult for the module to understand how it was compiled and how it is performing that procedure (we can see it in the purple box). The book best represents this procedure:

```asm
00401038        mov     eax, [ebp+var_4]
0040103B        cdq
0040103C        mov     ecx, 3
00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx
```

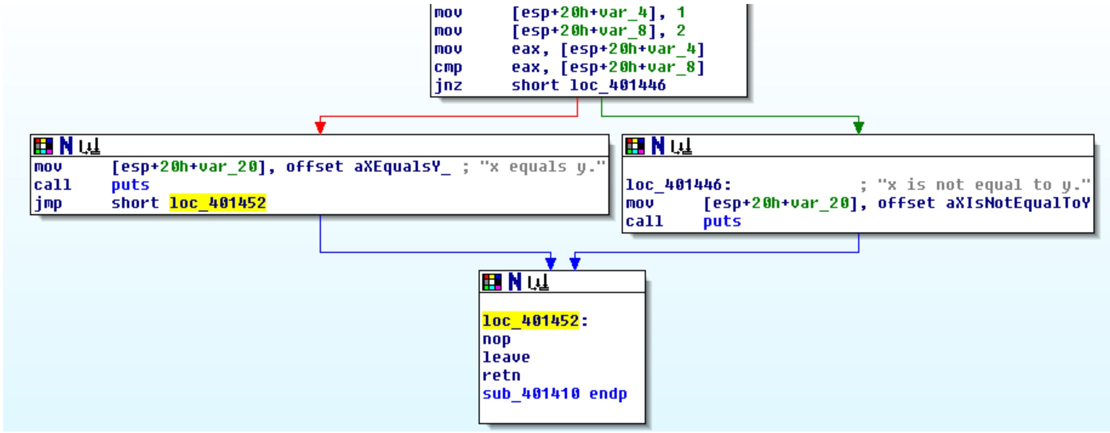
Here, *[ebp+var_4]* represents *a* and *[ebp+var_8]* represents *b.* When performing the *idiv* instruction it divides edx:eax by the operand and storing the result in *eax* and the remainder in *edx*. That is why *edx* is moved into *var_8.*

4. Notice how assembly translates an ***if*** statement in C:

```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");
    }
}
```



For this case, the visualization through the graph is clearer. On the first two lines it defines the variables *x* and *y*, then it moves *x* into *eax* to use the compare instruction *cmp*. Immediately after that, we have a *jump not zero (jnz)* which represents the *if* statement in assembly. There must be a conditional jump for an *if* statement, but not all conditional jumps correspond to *if* statements. If *x* and *y* are different then the jump occurs (the right path) and the code prints “x is not equal to y.”; otherwise, the code continues the path of execution (left path) and prints “x equals y.”.

5. Now let us look at some **nested if** statements:

```
#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 zero and x != y.\n");
        }
        else{
            printf("z non-zero and x != y.\n");
        }
    }
}
```

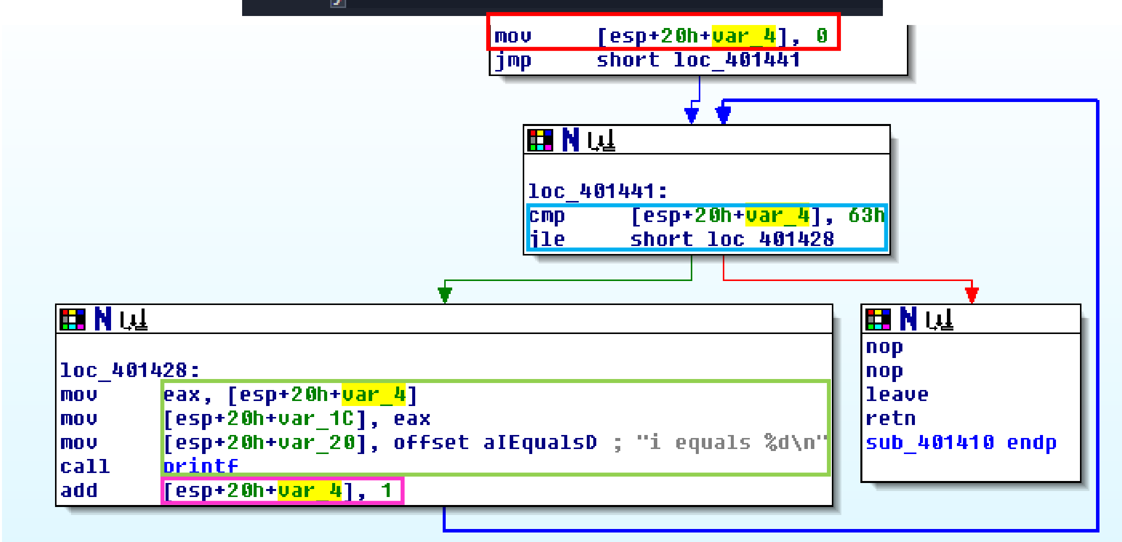
Even though the graph is considerably larger, it follows the same logic as before. On the first three lines it defines our three variables *x, y,* and *z.* Then compares *x* and *y* and if they are not equal it will jump to the right path (otherwise, it will continue the path of execution on the left path). In any case compares *z* with zero, and if *z* is not equal to zero it will jump to the right path and print its corresponding message, otherwise it will continue its path of execution and print the other message.

6. A ***for*** loop in C consists of 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);
    }
}
```



We can recognize the initialization of the counter variable *I* in red. The comparison in the blue box checks if this variable is less or equal than 99 (63 in hexadecimal) to execute the instructions in the green box. Finally, in pink we see the increment of the counter variable, and it returns to the blue box to continue with its comparison. When the counter variable is greater than 99 the program follows the right path and the loop ends.

7. To show the fact that ***while*** loops do not need an increment / decrement we will use the example from the book.

```
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
```

Notice that the *jmp* instruction at **2** causes the entire previous segment to repeat indefinitely until the conditional jump at **1** can be carried out, which is where the while condition is broken.

8. In assembly it is recognized that a **function** is being executed by the *call* statement. The three most common calling conventions are:

Convention	Parameters	Who cleans up the stack?
cdecl	Pushed onto the stack from right to left.	The caller

stdcall	Pushed onto the stack from right to left.	The callee
fastcall	The first few are passed in registers, and additional arguments are loaded from right to left	The callee

For example, notice our C code:

```c
#include <stdio.h>

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

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

    printf("the function returned the number &d\n", adder(x,y));
}
```

In assembly, the function *adder* is compiled as:

```
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
```

And the *main* function, where the code executes becomes:

```
mov     [esp+20h+var_4], 1
mov     [esp+20h+var_8], 2
mov     eax, [esp+20h+var_8]
mov     [esp+20h+var_1C], eax
mov     eax, [esp+20h+var_4]
mov     [esp+20h+var_20], eax
call    sub_401410
mov     [esp+20h+var_1C], eax
mov     [esp+20h+var_20], offset aTheFunctionRet ; "the function returned
call    printf
```

Notice how the two variables *x* and *y* are moved onto the stack from right to left, and then the function *adder* (*sub_401410)* is called.

9. **Switch** statements are compiled in two common ways: using the if style or using jump tables. For example, notice this switch with three cases:

```c
#include <stdio.h>

void main(){
    int i = 2;

    switch(i){
        case 1:
            printf("i = %d", i+1);
            break;
        case 2:
            printf("i = %d", i+2);
            break;
        case 3:
            printf("i = %d", i+3);
            break;
        default:
            break;
    }
}
```

```
cmp     [esp+20h+var_4], 3
jz      short loc_401476
cmp     [esp+20h+var_4], 3
jg      short loc_40148F
cmp     [esp+20h+var_4], 1
jz      short loc_401444
cmp     [esp+20h+var_4], 2
jz      short loc_40145D
jmp     short loc_40148F
```

```
loc_401444:                             ; CODE XREF: sub_401410+29↑j
                mov     eax, [esp+20h+var_4]
                add     eax, 1
                mov     [esp+20h+var_1C], eax
                mov     [esp+20h+var_20], offset aID ; "i = %d"
                call    printf
                jmp     short loc_401490
; ---------------------------------------------------------------------------

loc_40145D:                             ; CODE XREF: sub_401410+30↑j
                mov     eax, [esp+20h+var_4]
                add     eax, 2
                mov     [esp+20h+var_1C], eax
                mov     [esp+20h+var_20], offset aID ; "i = %d"
                call    printf
                jmp     short loc_401490
; ---------------------------------------------------------------------------

loc_401476:                             ; CODE XREF: sub_401410+1B↑j
                mov     eax, [esp+20h+var_4]
                add     eax, 3
                mov     [esp+20h+var_1C], eax
                mov     [esp+20h+var_20], offset aID ; "i = %d"
                call    printf
                jmp     short loc_401490
```

From this disassembly it is very difficult to know whether the original code was a *switch* statement or a sequence of *if* statements, since both can contain a bunch of *cmp* and conditional jumps.

Now, with 5 cases, assembly optimizes code by using a jumps table (*off_40504C*) instead of 5 conditionals. This table stores each of the switch instances.

```c
#include <stdio.h>

void main(){
    int i = 2;

    switch (i){
        case 1:
            printf("i = %d", i + 1);
            break;
        case 2:
            printf("i = %d", i + 2);
            break;
        case 3:
            printf("i = %d", i + 3);
            break;
        case 4:
            printf("i = %d", i + 4);
            break;
        case 5:
            printf("i = %d", i + 5);
            break;
        default:
            break;
    }
}
```

```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h          ; char *
call    sub_401A20
mov     [esp+20h+var_4], 2
cmp     [esp+20h+var_4], 5
ja      loc_4014BE
mov     eax, [esp+20h+var_4]
shl     eax, 2
add     eax, offset off_40504C
mov     eax, [eax]
jmp     eax
```

```
off_40504C      dd offset loc_4014BE
                dd offset loc_401441
                dd offset loc_40145A
                dd offset loc_401473
                dd offset loc_40148C
                dd offset loc_4014A5
```

Referencias

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* San Francisco: No Starch Press, Inc.