



Sixth Laboratory

Code Constructs

Sara Gallego Rivera

Universidad del Rosario
Escuela de Ingeniería, Ciencia y Tecnología

September 14th, 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 "Global 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.

<pre>#include<stdio.h> int x=1; int y=2; void main() { x = x+y; printf("Total = %d\n", x); }</pre>	<pre>call sub_4019A0 mov edx, dword_404004 mov eax, dword_404008 add eax, edx rbp 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_401418 endp</pre>
--	---

Global Variables



<pre>#include<stdio.h> void main() { int x=1; int y=2; x = x+y; printf("Total = %d\n", x); }</pre>	<pre>push ebp mov ebp, esp and esp, 0FFFFFFF0h sub esp, 20h call sub_4019A0 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</pre>
--	--

Local Variables

- Read the section "Disassembling Arithmetic Operations" and explain how the operations (addition, subtraction, 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.

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

C code

```
.text:00401410 var_4 = duword ptr -4
.text:00401410
.text:00401410 push    ebp
.text:00401411 mov     ebp, esp
.text:00401413 and     esp, 0FFFFFFF0h
.text:00401416 sub     esp, 10h
.text:00401419 call    sub_401980
.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 mov     ecx, [esp+10h+var_4]
.text:0040143F mov     edx, 55555556h
.text:00401444 mov     eax, ecx
.text:00401448 imul    edx
.text:0040144B mov     eax, ecx
.text:0040144D sar     eax, 1Fh
.text:00401450 sub     edx, eax
.text:00401453 mov     eax, edx
.text:00401455 sub     ecx, eax
.text:00401457 mov     eax, ecx
.text:00401459 mov     [esp+10h+var_8], eax
.text:0040145D nop
.text:0040145F leave
.text:0040145F retn
.text:0040145F sub_401410 endp
.text:0040145F
```

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."

```

1  #include<stdio.h>
2
3  void main(){
4      int x = 1;
5      int y = 2;
6      if(x==y){
7          printf("x equals y.\n");
8      }else{
9          printf("x is not equal to y.\n");
10     }
11 }

```

C code

```

IDA View A
.text:00401010 var_20 = dword ptr -20h
.text:00401010 var_8 = dword ptr -8
.text:00401010 var_4 = dword ptr -4
.text:00401010
.text:00401010 push ebp
.text:00401010 mov ebp, esp
.text:00401010 and esp, 0FFFFFFFh ; char *
.text:00401010 sub esp, 20h
.text:00401010 call sub_401008
.text:0040101E mov [esp+20h+var_4], 1
.text:0040102E mov [esp+20h+var_8], 2
.text:00401032 cmp eax, [esp+20h+var_8]
.text:00401036 jnz short loc_401046
.text:0040103B mov [esp+20h+var_20], offset aXequalsY ; "x equals y."
.text:0040103F call puts
.text:00401046 jmp short loc_401052
;
.text:00401046 loc_401046: mov [esp+20h+var_20], offset aXisNotEqualY ; "x is not equal to y."
.text:00401046 call puts
.text:00401052 loc_401052: nop ; CODE XREF: sub_401010+347
.text:00401052 leave
.text:00401054 call ret
.text:00401054 sub_401010
.text:00401054 endp

```

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 some 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."



```

1 #include<stdio.h>
2
3 void main (){
4     int x = 1;
5     int y = 1;
6     int z = 2;
7     if(x==y){
8         if(z==0){
9             printf("x equals y.\n");
10        }else{
11            printf("x is not equal to y.\n");
12        }
13    }else{
14        if(z==0){
15            printf("z zero and x != y.\n");
16        }else{
17            printf("z non-zero and x != y.\n");
18        }
19    }
20 }

```

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]
00401021    cmp     eax, [ebp+var_4]
00401027    jnz     short loc_401047
0040102B    cmp     [ebp+var_C], 0
00401033    jnz     short loc_401038
00401038    push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040103D    call    printf
00401042    add     esp, 4
00401045    jmp     short loc_401045

00401045 loc_401045:
00401045    jmp     short loc_401069
00401047 loc_401047:
00401047    cmp     [ebp+var_C], 0
0040104B    jnz     short loc_40105C
0040104D    push    offset aZIsZeroAndXY_ ; "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    printf
00401061

```

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 some 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 variable because it is 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 recognize this type of loop by locating the four components: initialization, comparison, execution instructions, and increment/decrement.

```
1 #include<stdio.h>
2
3 void main(){
4     int i;
5     for(i=0; i<100;i++){
6         printf("i equals %d\n", i);
7     }
8 }
```

C code

```

IDA View A
[00000000]
.text:00000110 sub_000110
.text:00000110
; CODE XREF: sub_000110+8E1F
.text:00000113 uar_20h
mov dword ptr -20h, uar_20h
.text:00000116 uar_1Ch
mov dword ptr -1Ch, uar_1Ch
.text:00000118 uar_4h
mov dword ptr -4h, uar_4h
|
.text:00000118
push ebp
mov ebp, esp
.text:00000119
and esp, 0FFFFFFFh
.text:00000119
sub esp, 20h ; char *
.text:00000119
call sub_00010A0
.text:0000011C
mov [esp+20h*var_3], 0
.text:00000126
jmp short loc_0001A1
.text:00000128
short loc_0001A1
.text:00000129
; CODE XREF: sub_000110+36j
.text:00000128 loc_0001A20:
mov eax, [esp+20h*var_4]
.text:0000012C
[esp+20h*var_1C], eax
.text:0000012B
mov [esp+20h*var_10], offset a1equals10 ; "i equals 10"
.text:00000127
call printf
.text:0000012C
add [esp+20h*var_3], 1
.text:00000141
; CODE XREF: sub_000110+161f
.text:00000141 loc_0001A41:
cmp [esp+20h*var_3], 63h
jle short loc_0001A28
.text:00000143
nop
.text:00000143
nop
.text:00000143
nop
.text:00000143
leave
.text:00000143
retn
.text:0000014B sub_000110
endp

```

Assembly code

```

00401004      mov     [ebp+var_4], 0 ❶
00401008      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.

```

1  #include<stdio.h>
2
3  void main(){
4      int status = 0;
5      int result = 0;
6
7      while(status == 0){
8          result = performAction();
9          status = chechResult(result);
10     }
11 }

```

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
00401058      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.