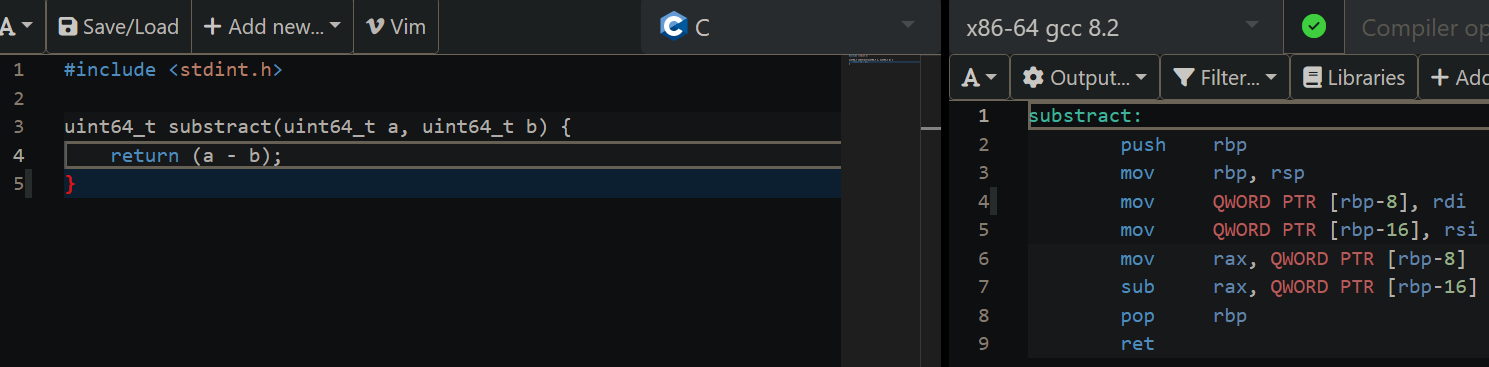
Lab Session 0x02

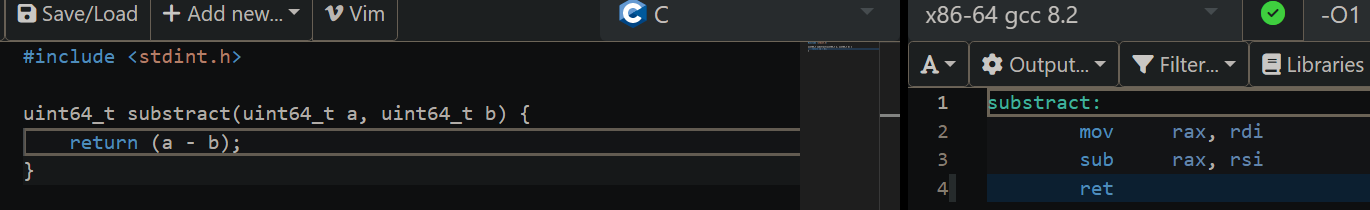
For today’s laboratory we will use [Godbolt](https://gcc.godbolt.org/) (compiler explorer).

**2. Assembly Analysis**

1. *Write a C function that subtracts two integers. Observe the calling convention (RDI/RSI) and the return value (RAX –* **return value of the function being called** [**source**](https://github.com/danbev/learning-assembly/blob/master/README.md)*).*

Regarding [registers](https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start). Regarding [optimization](https://www.rapidtables.com/code/linux/gcc/gcc-o.html)(-O option flag) – we use O1. Short assembly [guide](https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html).





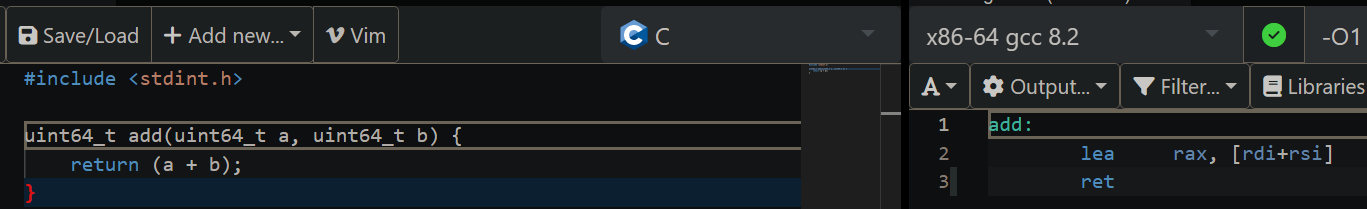
***uint64\_t*** – casts the value to an unsigned 64 bit integer (in order to get *rax, rsi, rdi*, etc...; if we use ***int*** the compiler will use *eax, esi, edi*, etc... – aka 32-bit);

***mov*** – copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory);

***sub*** – stores in the value of its second operand the result of subtracting the value of its first operand from the value of its second operand;

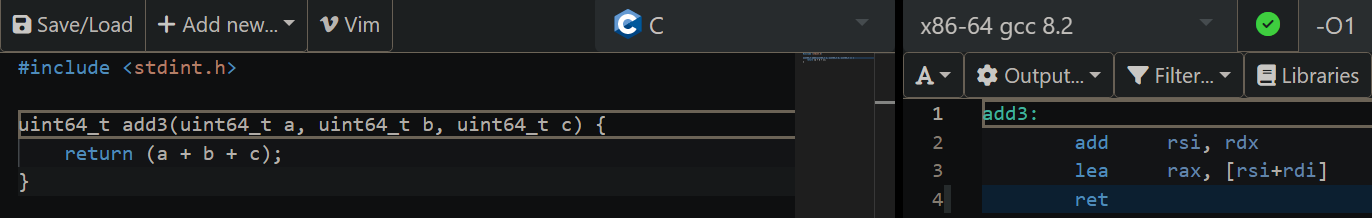
***ret*** – transfers control to the return address located on the stack (return back to where you called the function from).

1. *Write a C function that adds two integers. What assembly instruction did the compiler use?*



***lea*** – load effective address (The lea instruction places the address specified by its first operand into the register specified by its second operand. Note, the contents of the memory location are not loaded, only the effective address is computed and placed into the register). We can notice the compiler found a shorter way to do the operations.

1. *Write a C function that adds three integers. What assembly instructions do we have now?*

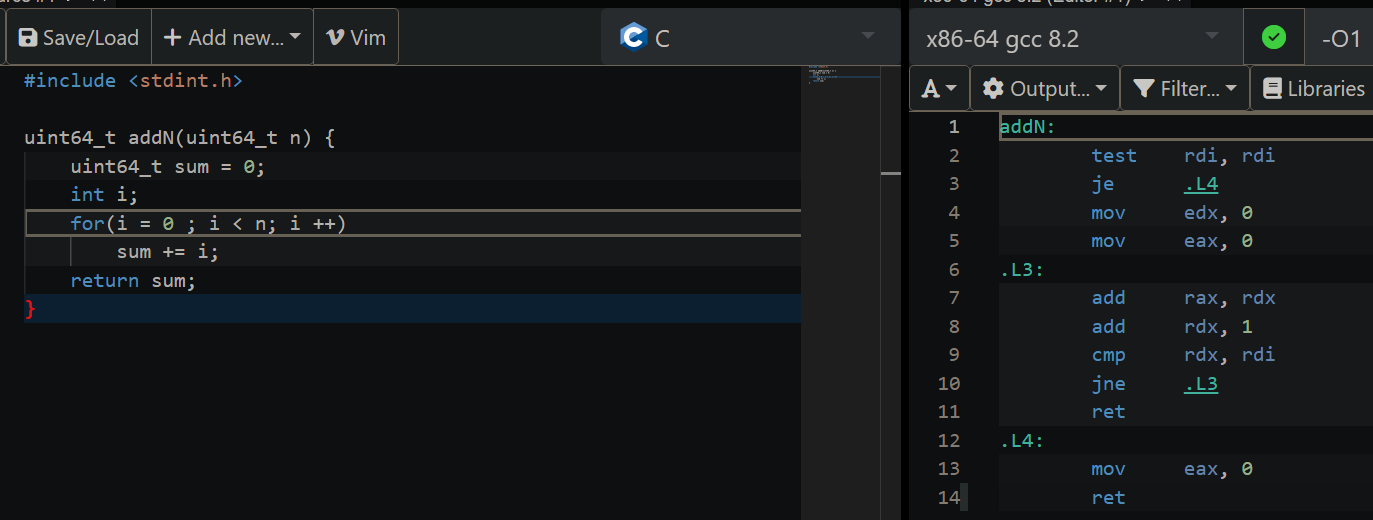


***add*** – adds together its two operands, storing the result in its first operand (we can notice the limitation of the previous optimization).

1. *Write a C function that adds the first n positive integers. Observe the loops.*

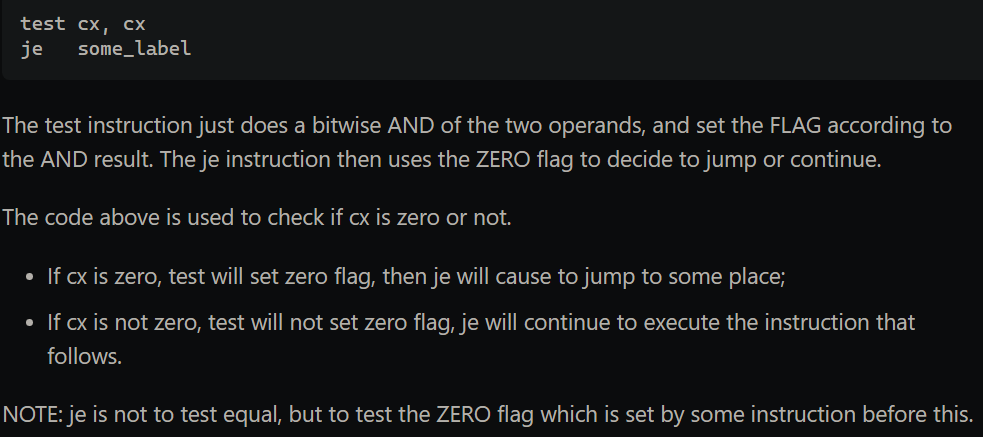
Let’s write the function in 2 different ways and observe the differences.

The „*for 1-n*” sum method:

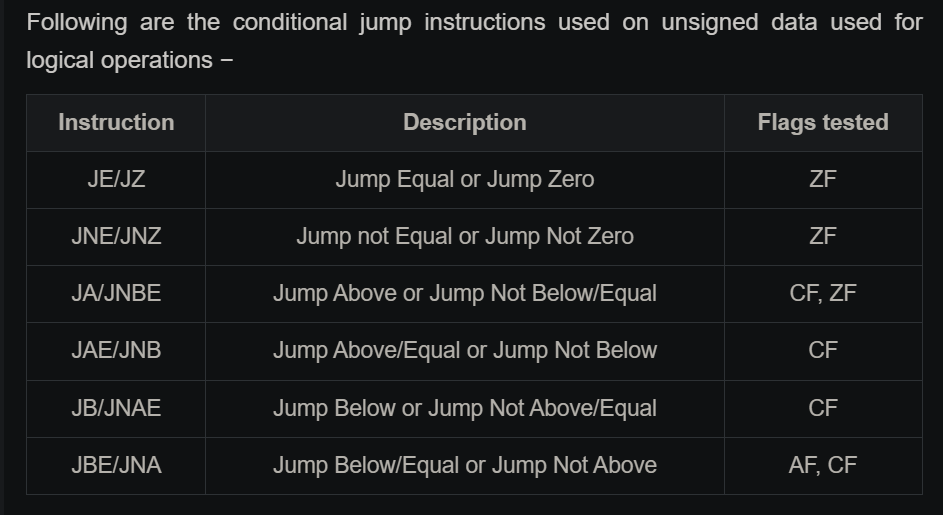


***test*** – performs an implied AND operation between corresponding bits in the two operands and sets the flags without modifying either operand;

***je*** – jump if equal ([source](https://stackoverflow.com/questions/1582960/assembly-language-je-jump-function) or [source](https://en.wikipedia.org/wiki/TEST_(x86_instruction)));

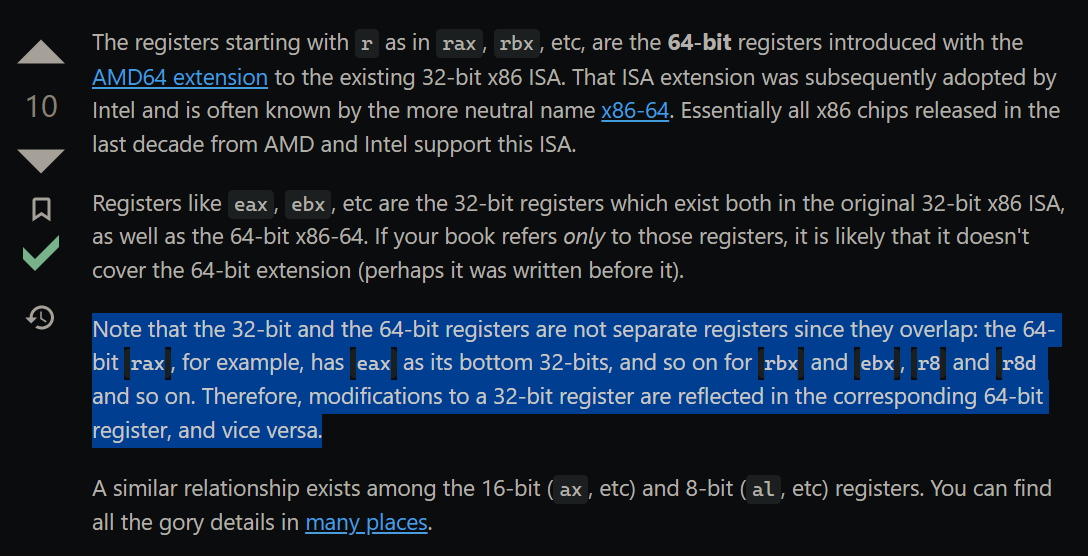


***cmp*** – compares 2 operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making. ([Source](https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm) for cmp and JNE)



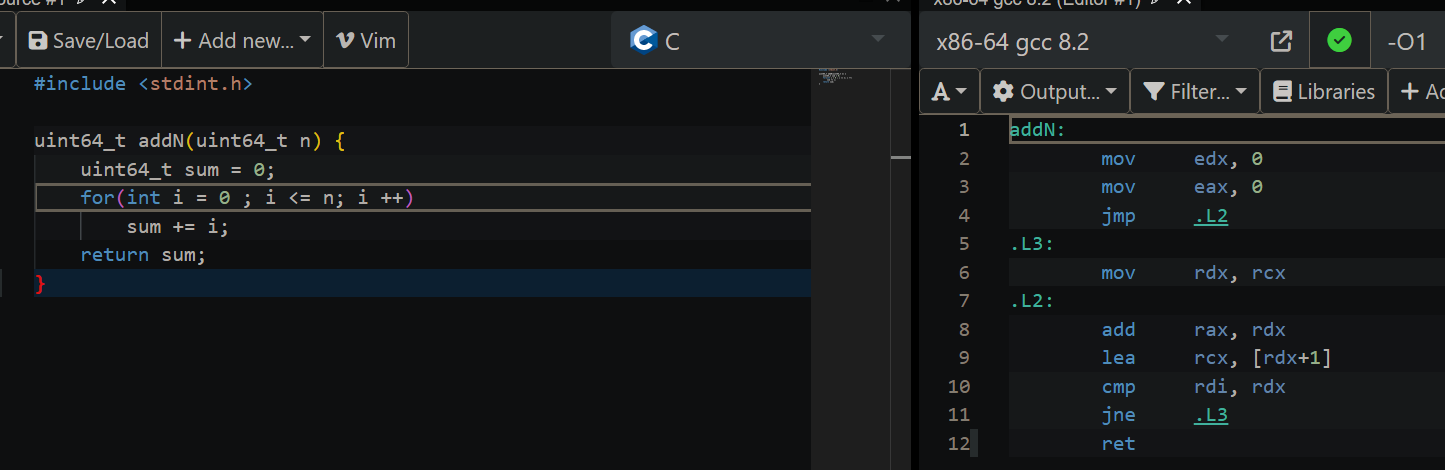
The assembly code would translate to something like this: test if ***n*** (***rdi***) equals to 0; if true, then jump to *.L4*, where ***sum*** (***eax***) gets the value 0 and then we exit the function; otherwise, ***i*** (***edx***) gets the value 0, ***sum*** (***eax***) also becomes 0, then we enter the loop *.L3*:

* add rax, rdx → sum += i
* add rdx, 1 → i += 1
* cmp rdx, rdi → compares whether or not ***i*** and ***n*** are equal
* jne .L3 → if ***i*** and ***n*** are NOT equal, jump at the beginning of the *.L3* loop, otherwise exit



([source1](https://stackoverflow.com/questions/44972293/how-is-rax-different-from-eax) & [source2](https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture))

I’ve noticed after I finished that the algorithm is wrong considering the requirement („*add the first n positive integers*”). Here is the corrected version of the algorithm:



We notice that the assembly output is different. In this case, ***i*** is ***edx/rdx*** and ***sum*** is ***eax/rax***, both being initialized with 0. Then we jump inside the for loop, *.L2*, where:

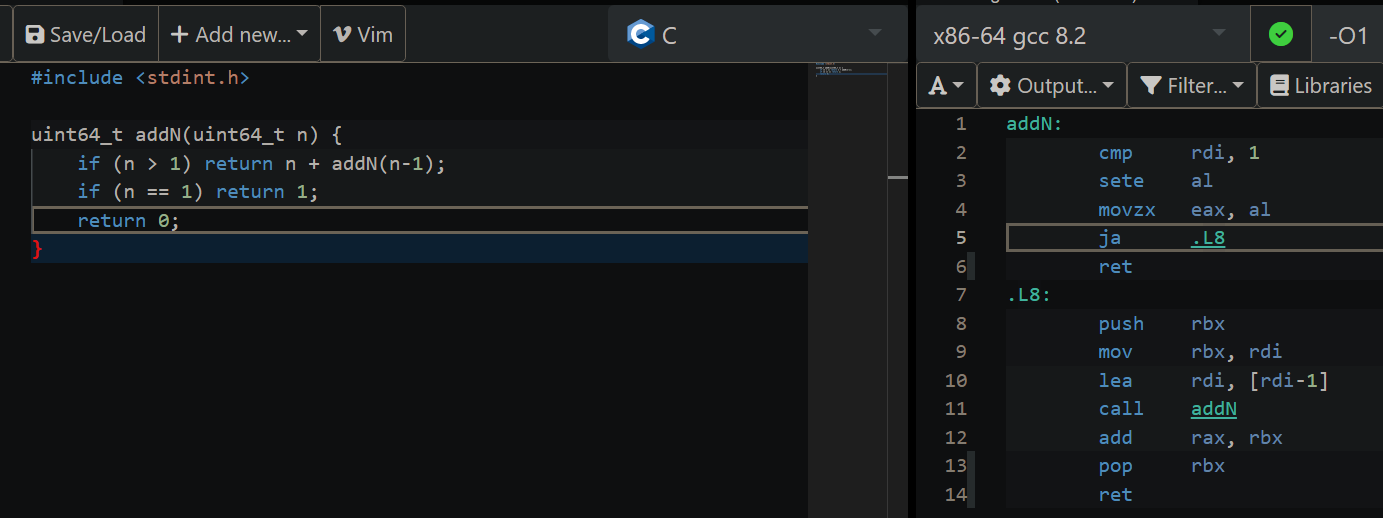
- add rax, rdx → sum += i

- lea rcx, [rdx+1] → rcx = i + 1 (the value of ***i*** is not modified)

- cmp rdi, rdx → compare ***n*** (***rdi***) and ***i***

- jne .L3 → if ***i*** is not equal to ***n***, then jump to *.L3*, where ***i*** will get the value of rcx (above rcx got the value of i+1), then we enter again *.L2* loop; otherwise exit

Divide et Impera method:



***sete*** – sets al (8-bit) to 1 if the zero flag is set or to 0 otherwise;

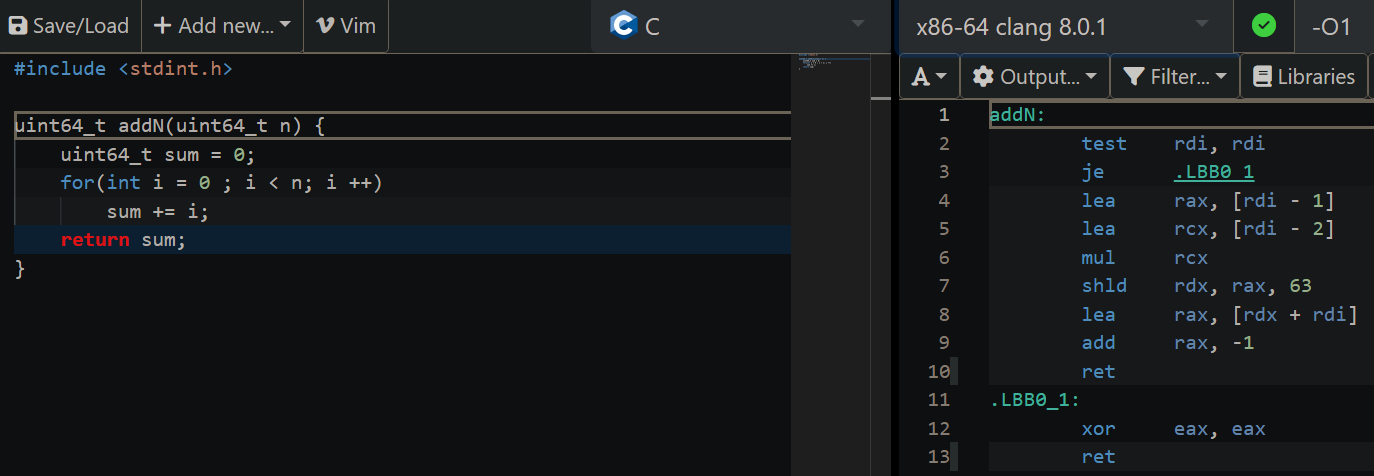
***movzx*** – reads the contents of the register or effective address as a word or byte. movzx then sign-extends the 16- or 32-bit value to the operand-size attribute of the instruction. The result is stored in the destination register by movzx;

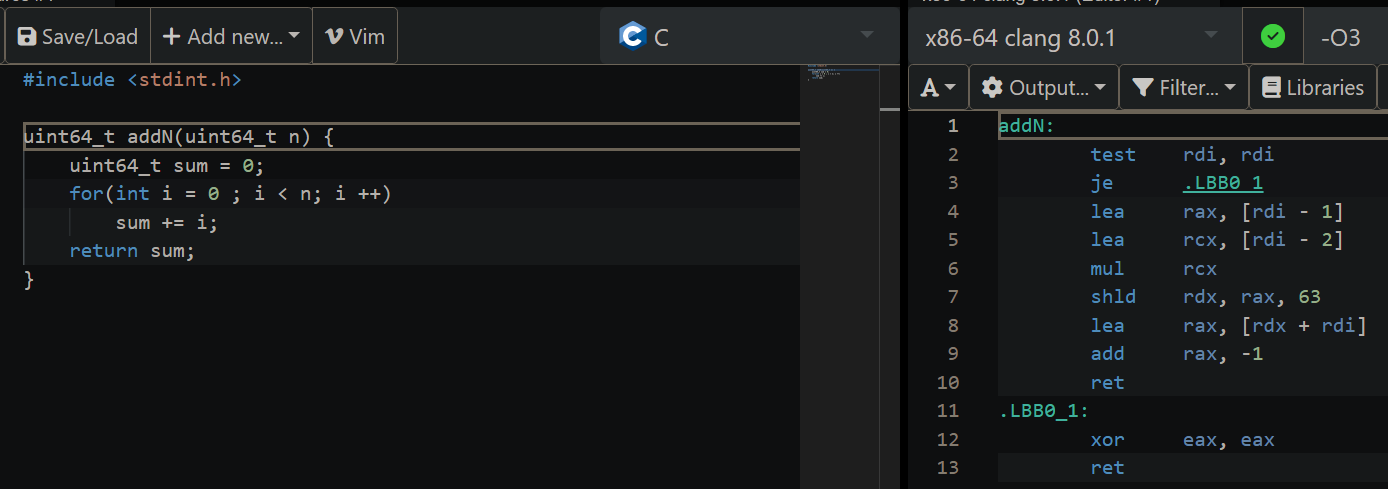
***ja*** – jump above;

***call*** – handles passing the return address for you.

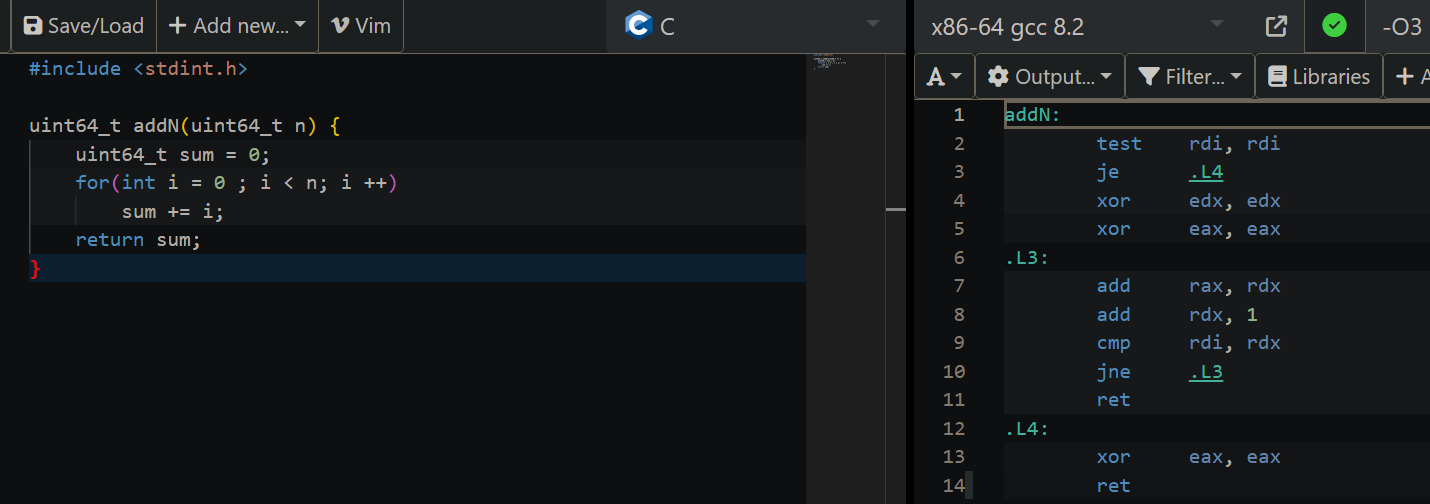
* 1. *Try also the clang compiler. Try using optimization flags O1 and O3. What happens now?*

For the first method:

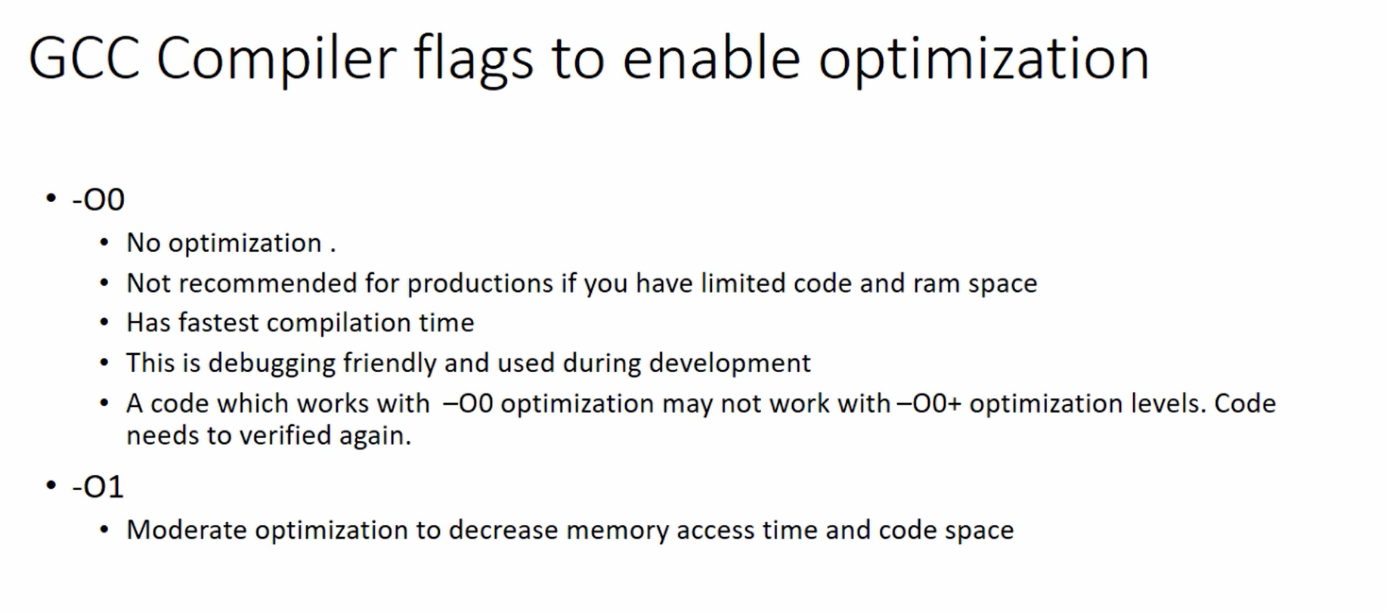


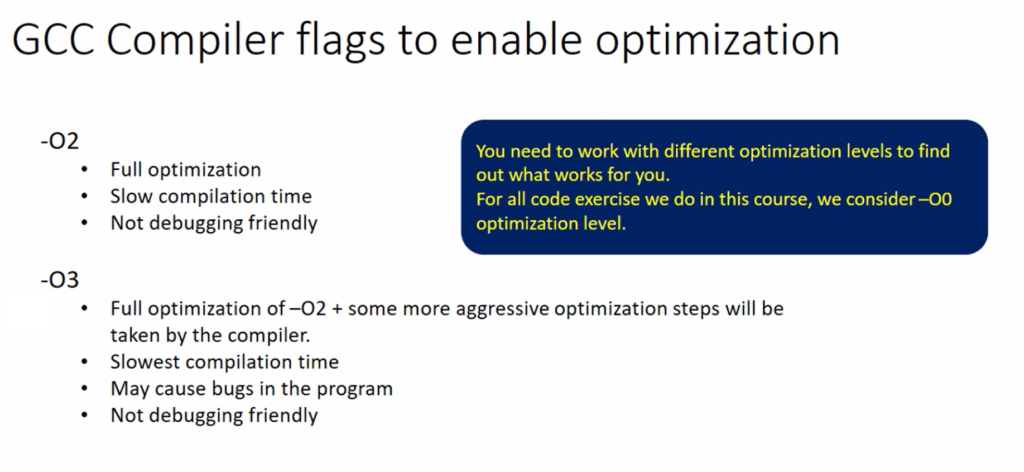


We notice no difference between the 2 outputs if we change the -O flag. We can notice a difference if we use the gcc compiler and -O3 flag:

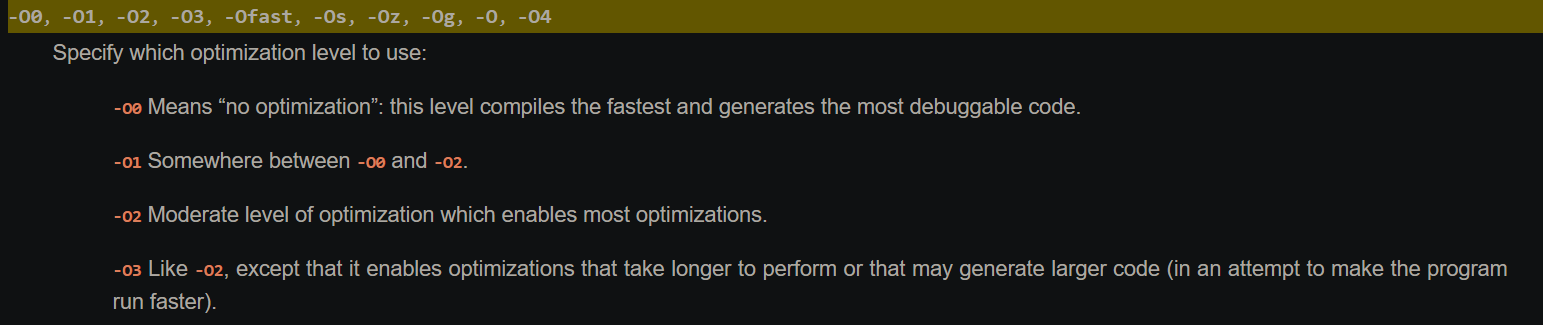


Clang is much faster and uses far less memory than GCC. ([source](https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html))





([source](https://fastbitlab.com/microcontroller-embedded-c-compiler-optimization-and-flags/) for -O flags for GCC)



([source](https://clang.llvm.org/docs/CommandGuide/clang.html#cmdoption-o0) for -O flags for Clang)

Clang is already optimized and that is probably the reason why we do not see any difference between the flags.

Let’s translate what happens in the assembly (Clang version):

addN: # @addN

test rdi, rdi

je .LBB0\_1

lea rax, [rdi - **1**]

lea rcx, [rdi - **2**]

mul rcx

shld rdx, rax, **63**

lea rax, [rdx + rdi]

**add** rax, -**1**

ret

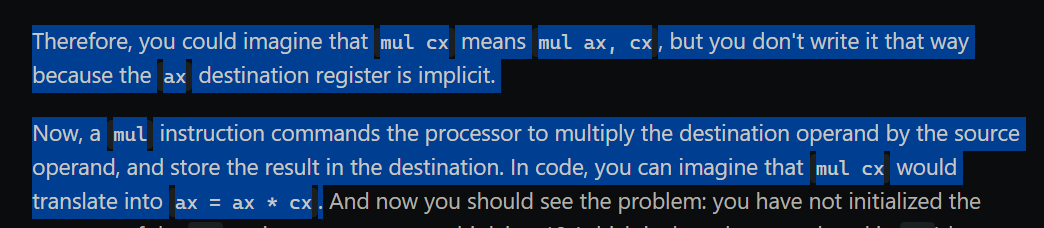
.LBB0\_1:

xor eax, eax

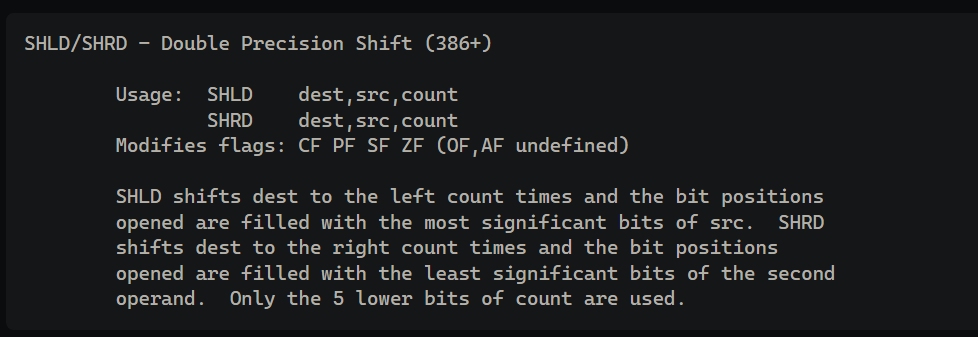
ret

If ***n*** (***rdi***) is 0, then we jump to *.LBB0\_1* where ***sum*** (***rax/eax***) gets the value 0 (because ***eax*** is xored with itself). Otherwise, ***rax*** = n – 1, ***rcx*** = n – 2.

* mul rcx → rax = rax \* rcx (rax = (n-1)(n-2)) ([source](https://stackoverflow.com/questions/40893026/mul-function-in-assembly))



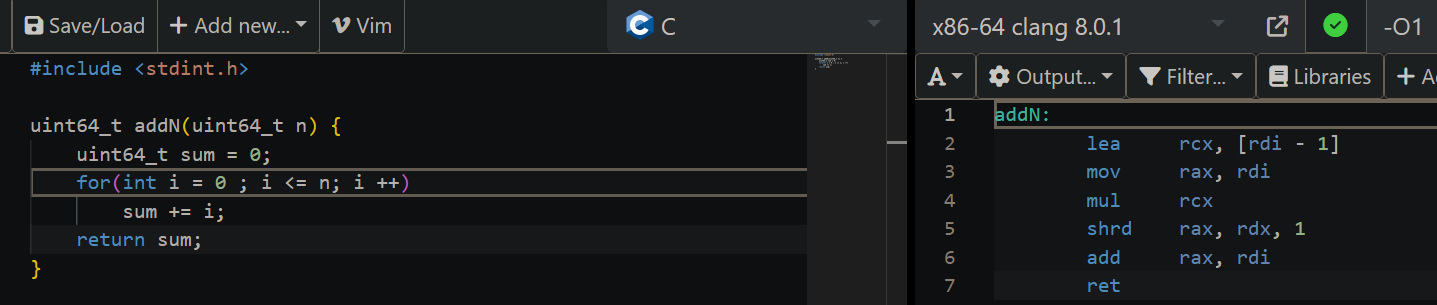
* shld rdx, rax, 63 → rdx = rax / 2 (rdx = (n-1)(n-2)/2) ([source](https://stackoverflow.com/questions/26163511/how-does-shld-and-shrd-really-really-work))



* lea rax, [rdx + rdi] → rax = rdx + rdi (rax = [(n-1)(n-2)/2] + n)
* add rax, -1 → rax = rax – 1 (rax = [(n-1)(n-2)/2] + n – 1 = (n-1)\*n/2)

Basically, the compiler recreates the formula for Gauss Sum: (n-1)\*n/2.

The corrected version of the algorithm:



The output of the assembly code is the same for -O3 flag. This code is easier to analyze:

- lea rcx, [rdi - 1] → rcx = n – 1 (rdi = n)

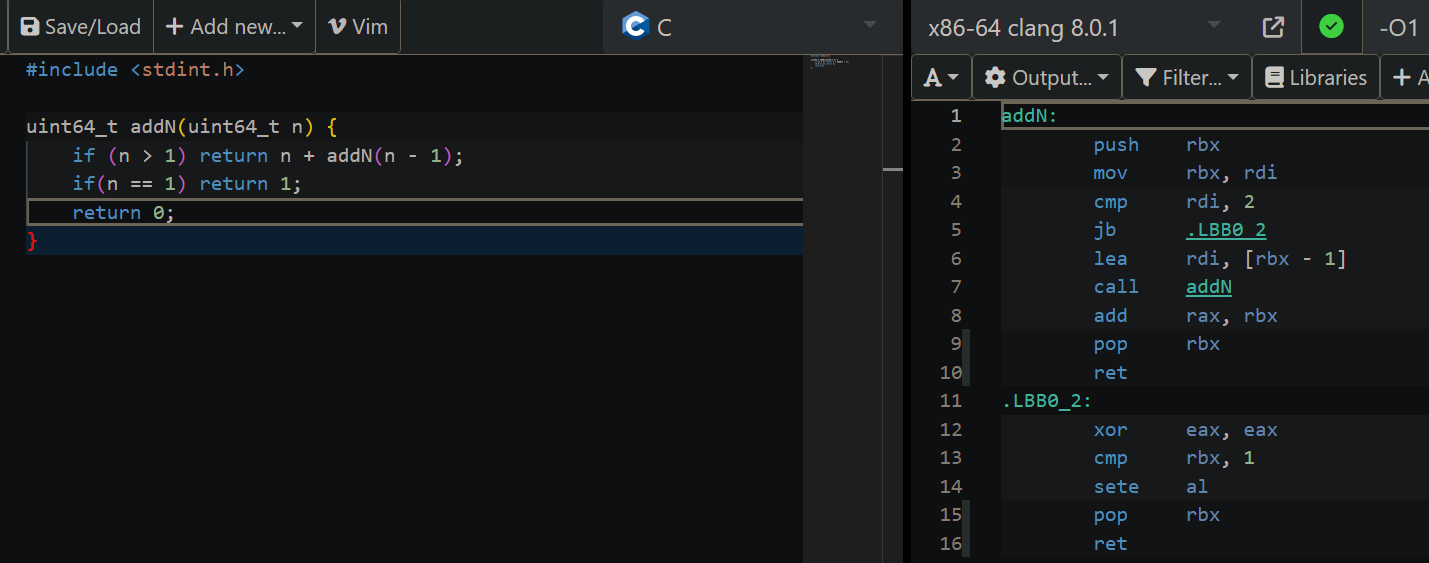
- mov rax, rdi → rax = n

- mul rcx → rax \*= rcx (rax = n(n-1))

- shrd rax, rdx, 1 → rax /= 2 (rax = n(n-1)/2)

- add rax, rdi → rax += n (rax = n(n-1)/2 + n = n(n+1)/2) → **Gauss Sum**

For the second method:

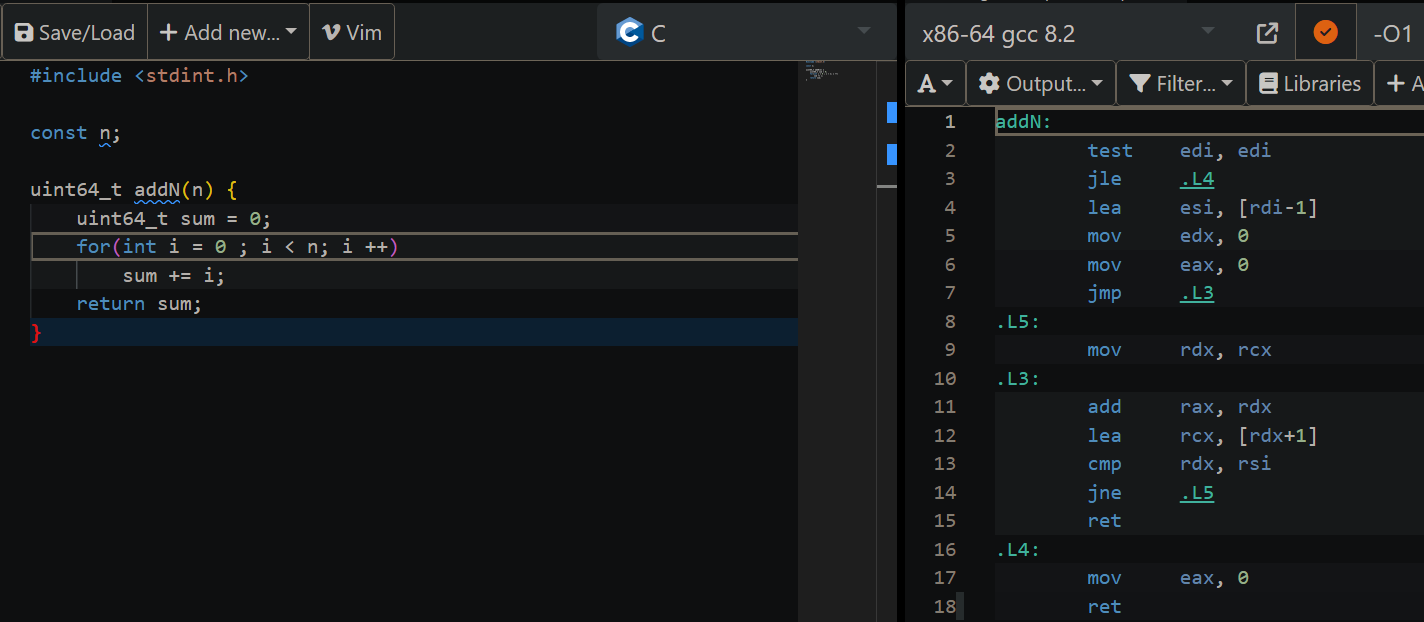


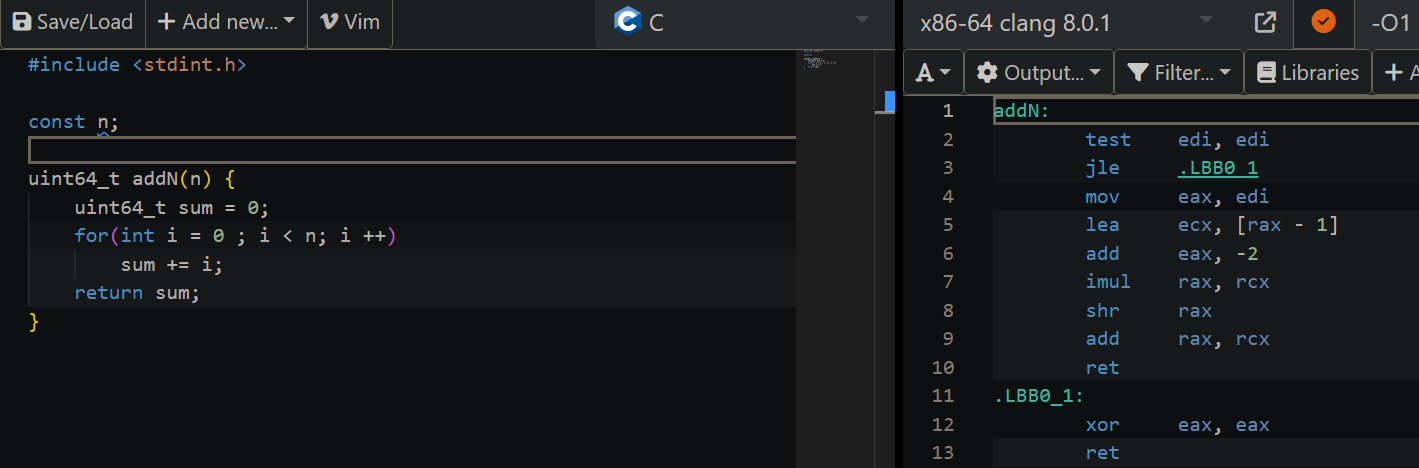
Using -O3 flag, outputs the same result. This one is pretty straight-forward as it uses the stack to remember the values of *n, n-1, n-2*, etc... until it reaches 1.

* 1. *At the beginning of the function fix the number n to a constant value.*

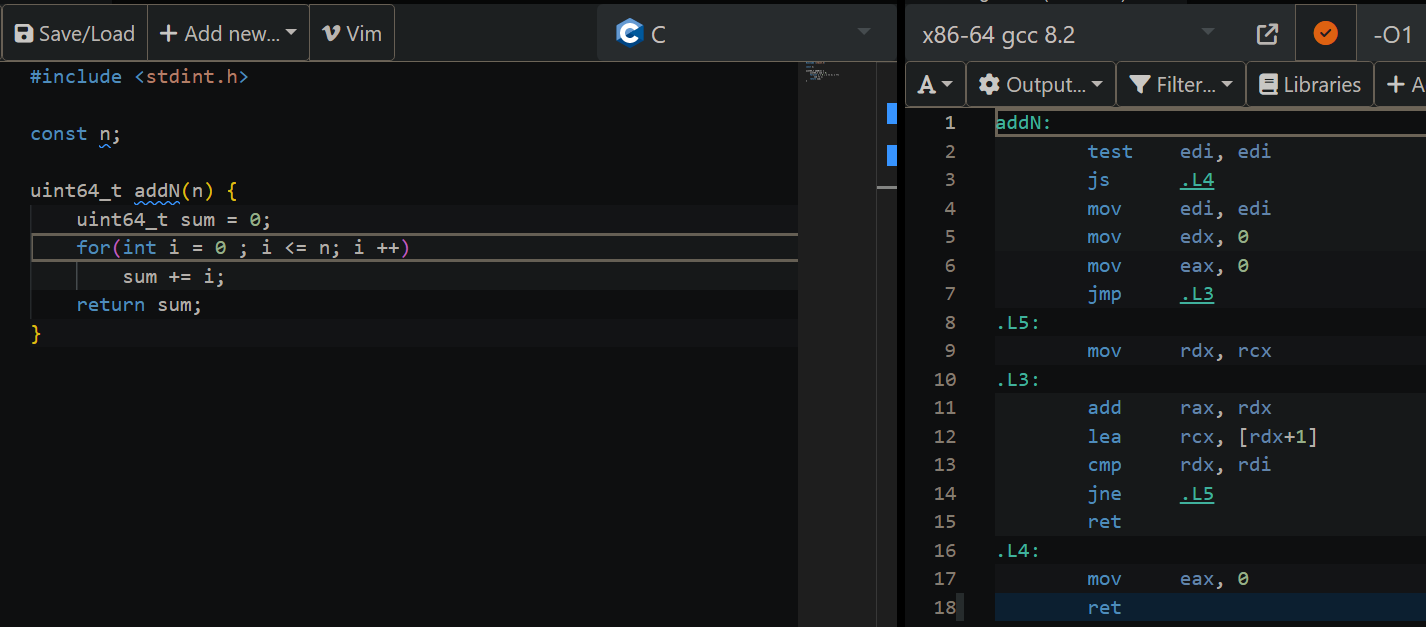
For the first method:

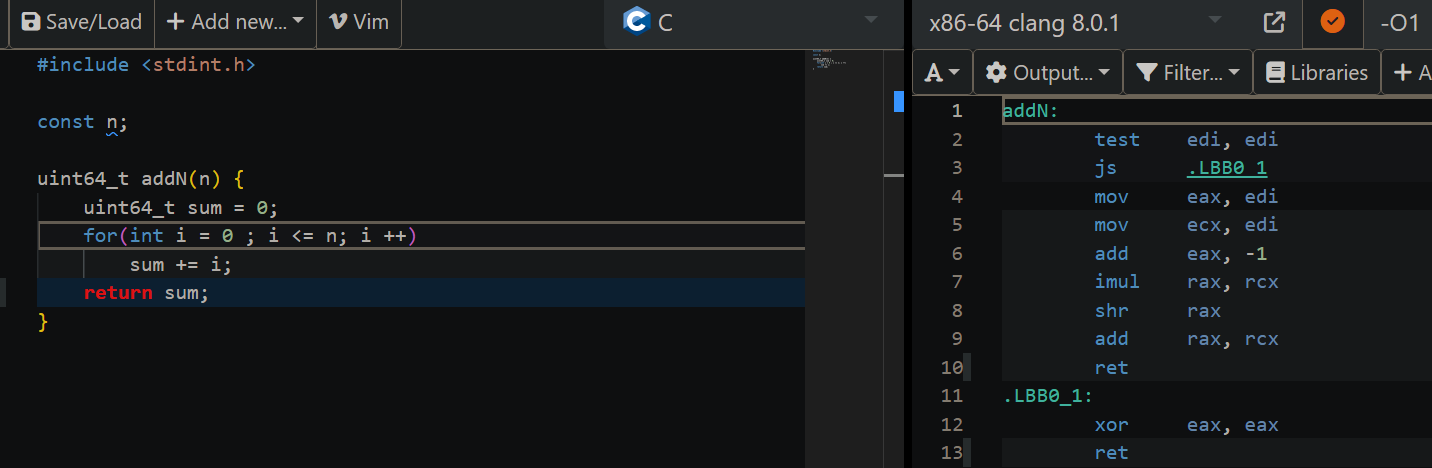
For the incorrect version:





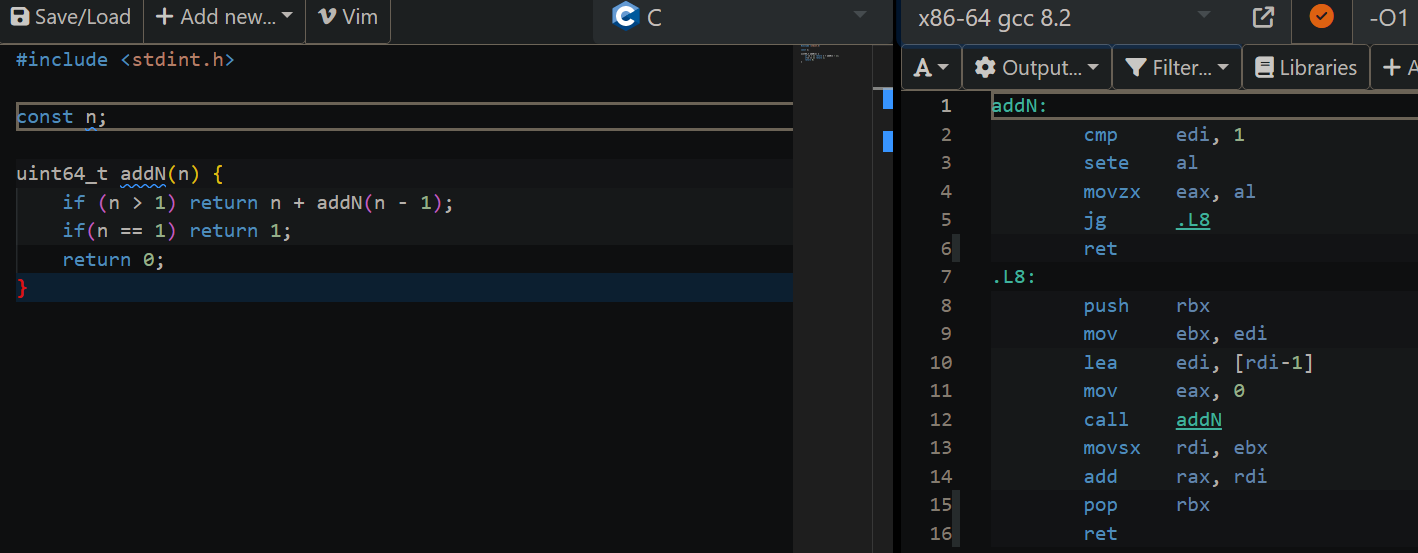
For the correct version:

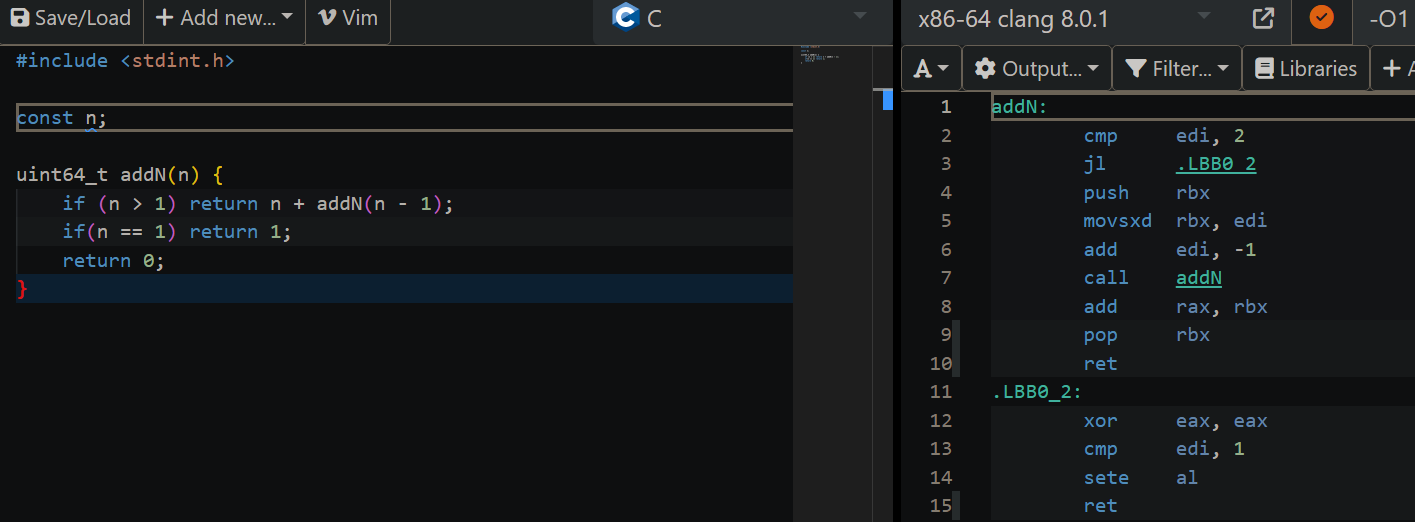




We can notice that the output for both versions are similar (for both gcc and clang).

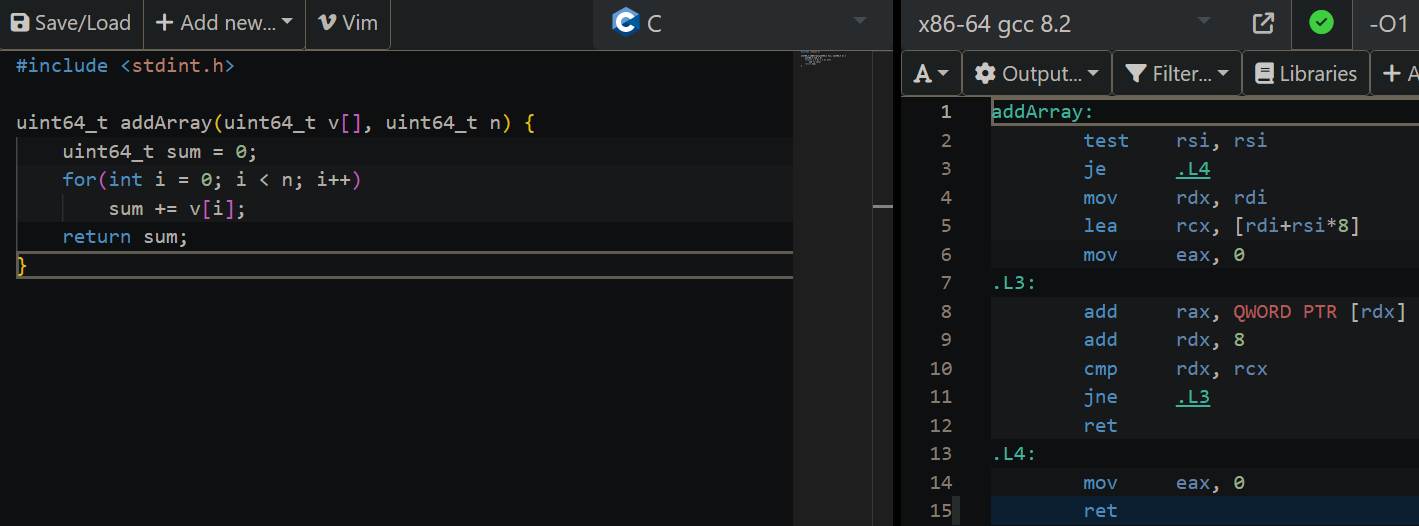
For the second method:





1. *Write a C function that adds the elements in a vector of integers. Try using flags O1 and O3.*

The only new notation is ***QWORD PTR*** which implies that the referenced date is 64-bit in size ([source](https://www.thesecuritybuddy.com/reverse-engineering/what-are-byte-ptr-word-ptr-dword-ptr-and-qword-ptr-directives-in-x86-and-x64-assembly/)). The output is easy to understand: ***n*** is ***rsi***; if ***n*** = 0, then jump to *.L4*, where ***sum*** (***eax***) will be 0; otherwise, ***rdx*** becomes v[0] (***rdi*** is ***v[0]***), ***rcx*** = ***v[n]*** and initialize sum with 0; in *.L3* is the loop where we traverse the array using ***rdx*** and we compare it with ***rcx*** in order to know when to stop.

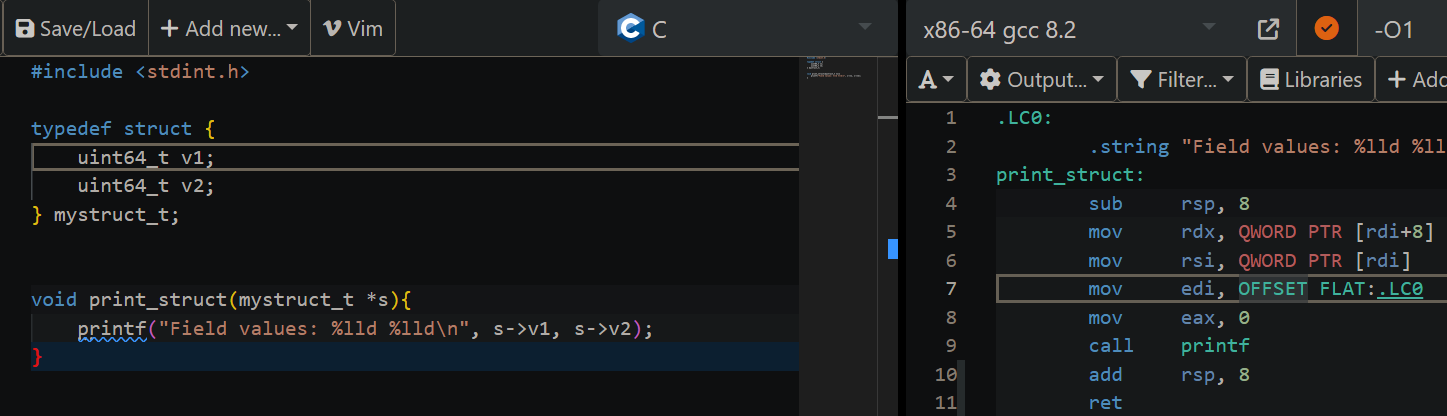


If we use -O3 flag, this is the output:

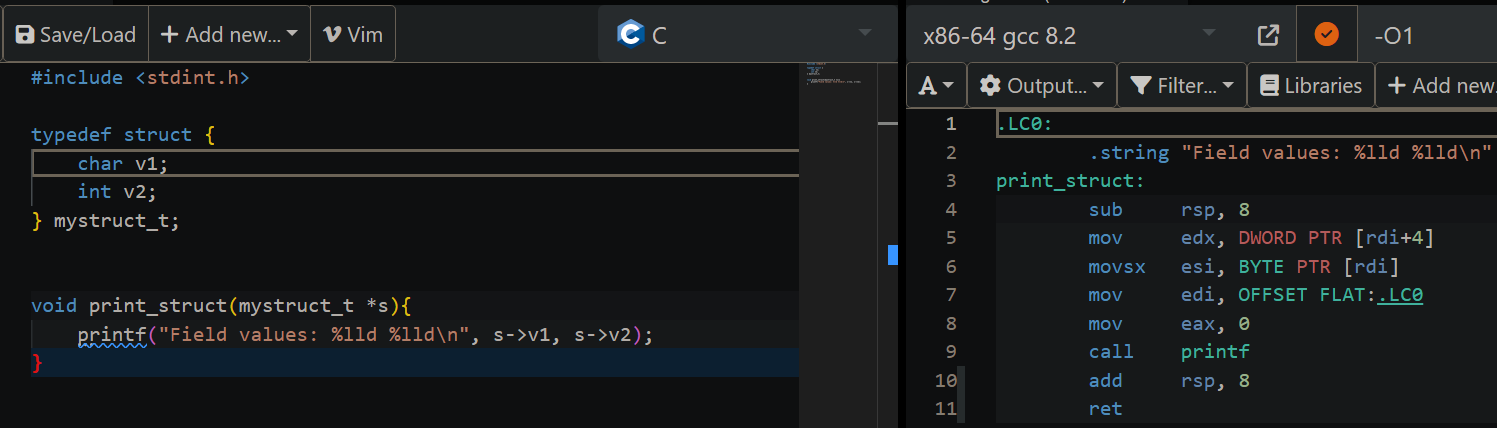
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | addArray:  test rsi, rsi  je .L7  lea rax, [rsi-**1**]  cmp rax, **2**  jbe .L8  mov rdx, rsi  mov rax, rdi  pxor xmm0, xmm0  shr rdx  sal rdx, **4**  **add** rdx, rdi  .L4:  movdqu xmm2, XMMWORD PTR [rax]  **add** rax, **16**  paddq xmm0, xmm2  cmp rax, rdx  jne .L4  movdqa xmm1, xmm0  mov rcx, rsi  psrldq xmm1, **8**  and rcx, -**2**  paddq xmm0, xmm1  mov edx, ecx  movq rax, xmm0  cmp rsi, rcx  je .L11  .L3:  **add** rax, QWORD PTR [rdi+rcx\***8**]  lea ecx, [rdx+**1**]  movsx rcx, ecx  cmp rsi, rcx  jbe .L1  **add** edx, **2**  **add** rax, QWORD PTR [rdi+rcx\***8**]  movsx rdx, edx  cmp rsi, rdx  jbe .L1  **add** rax, QWORD PTR [rdi+rdx\***8**]  ret  .L7:  xor eax, eax  .L1:  ret  .L11:  ret  .L8:  xor edx, edx  xor eax, eax  xor ecx, ecx  jmp .L3 |

Again, it testes whether n = 0, in order to exit the program with sum = 0. If n > 0, but n < 3, then it jumps to .L8, where it traverse the array (it’s not a loop, it’s one step, or two). If n >= 3, then rdx = v[n] (lines 7-12) then we have the sum (loop) but we work with bytes. It’s faster because of the operations that are being used, but it’s more complicated.

1. *Define a struct and access v1 and v2 by writing these values in the console (use printf ). Observe the pointer arithmetic (change the data types of v1 and v2) and the first string reference given to printf.*



We notice that regardless of the data type, in assembly the first value is always v2:



1. *Consider and analyze the following code that traverses a linked list:*

#include <stdint.h>

**typedef** **struct** {

**uint64\_t** v1;

**struct** **mystruct\_t** \*next;

} **mystruct\_t**;

**mystruct\_t** \***get\_last**(**mystruct\_t** \*head){

**mystruct\_t** \*cur = head;

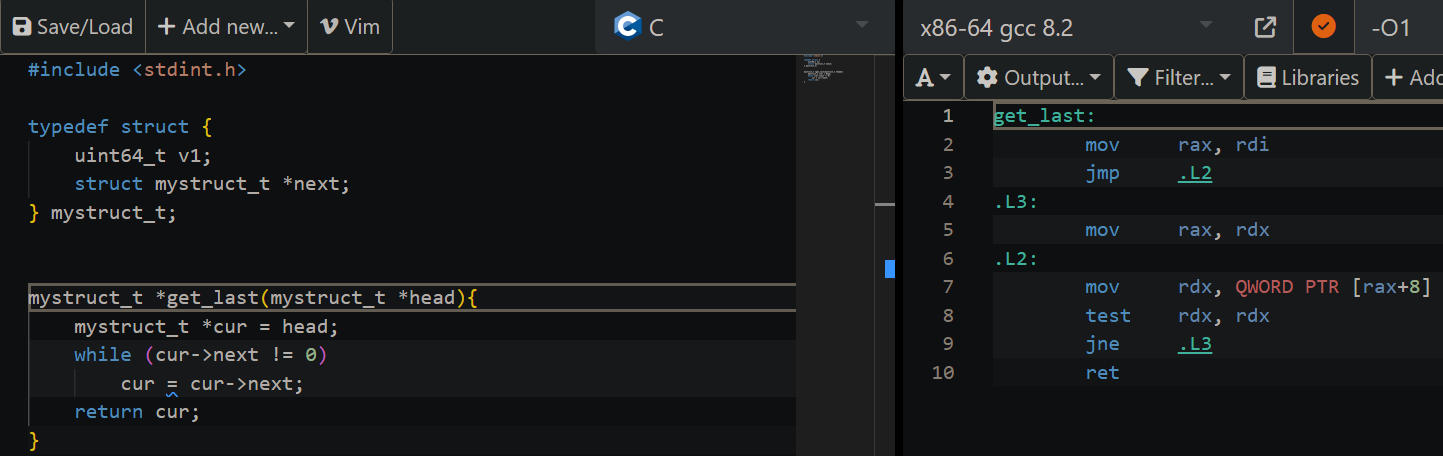
**while** (cur->next != **0**)

cur = cur->next;

**return** cur;

}

This is the code for a single linked list traversal. The assembly output is:

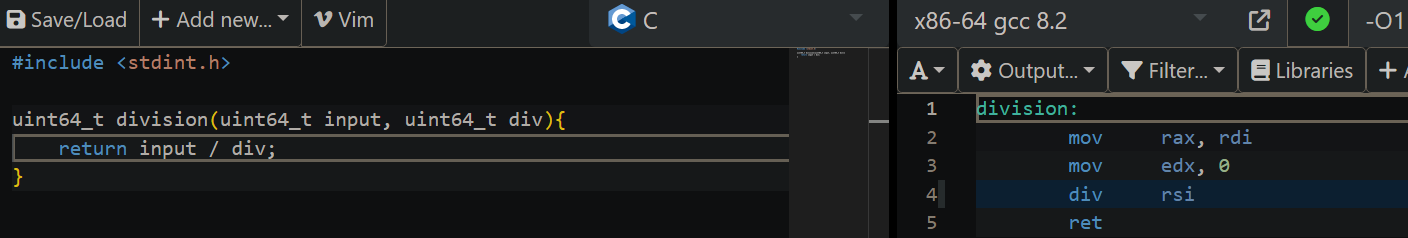


***Rax*** contains the current node, while ***rdx*** contains the following one and tests if is 0, in order to decide if we move to the next node.

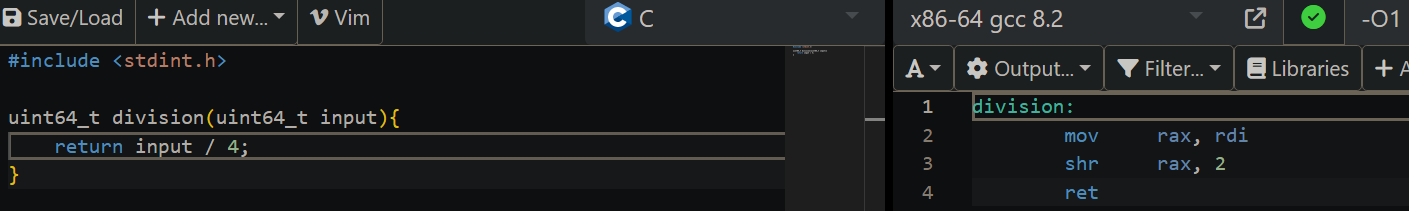
1. *Write a C function that divides an integer by constants 4, 5, 32. Do the same for multiplication by the same constants. Division is the bane of computer performance and the compiler will go to extreme lengths to avoid it.*

Division:

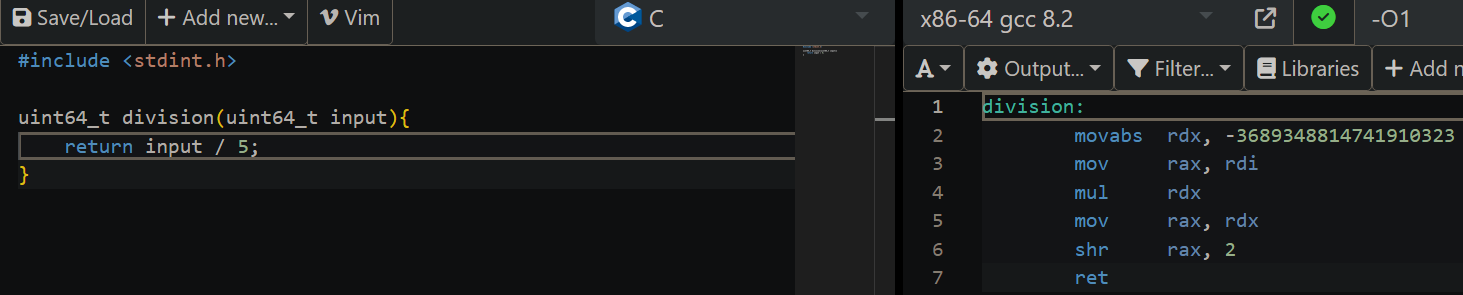
* General Function:



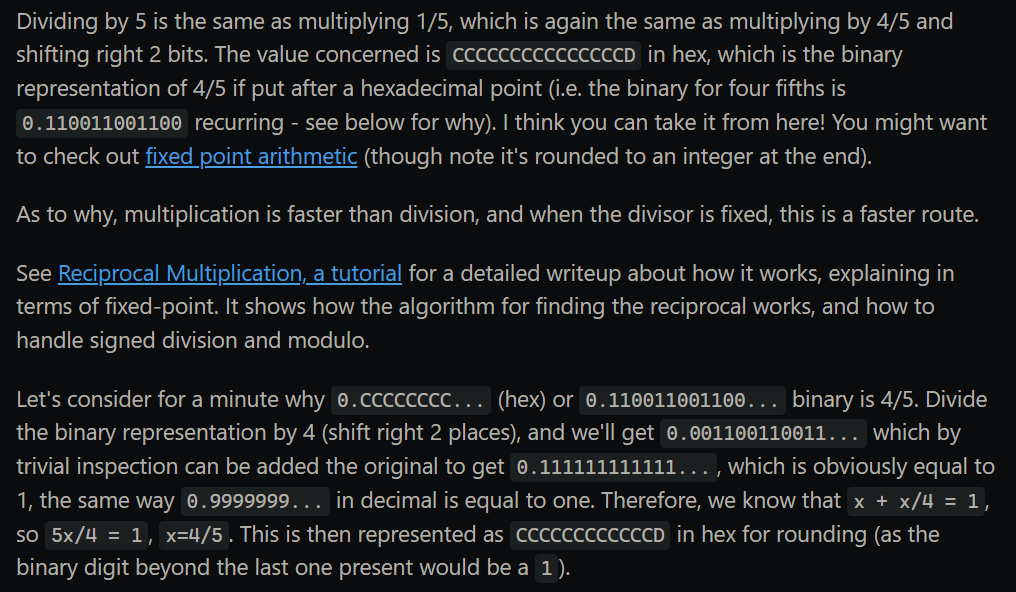
* 4:



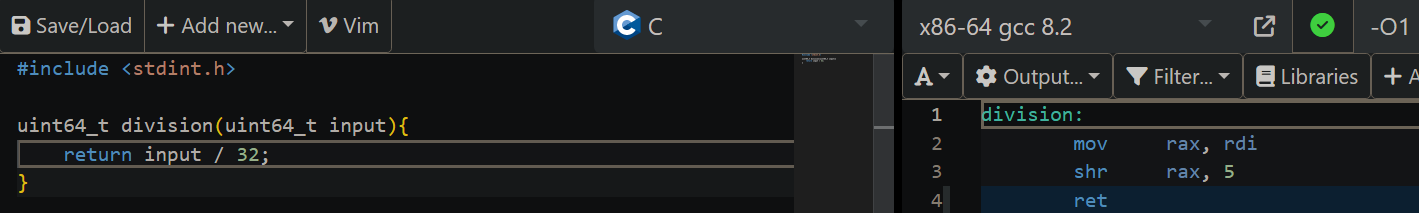
* 5:



The explanation is from [here](https://stackoverflow.com/questions/41183935/why-does-gcc-use-multiplication-by-a-strange-number-in-implementing-integer-divi).

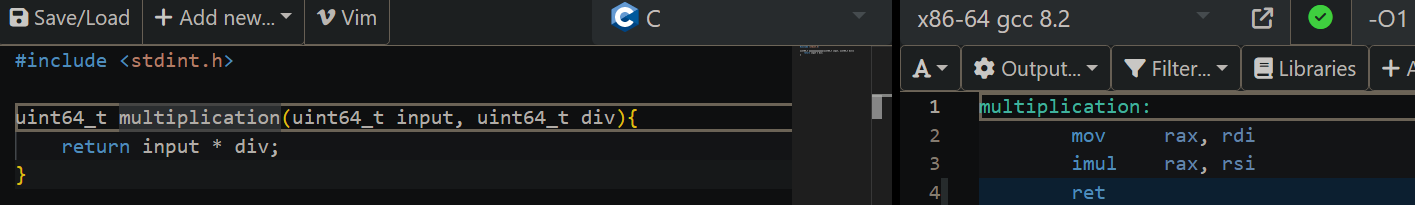


* 32:

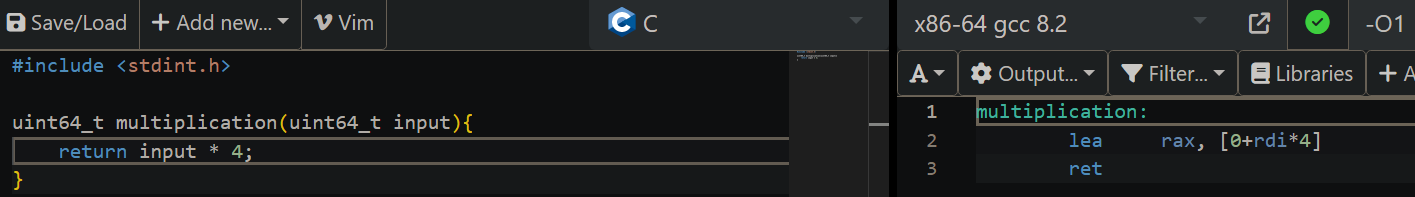


Multiplication:

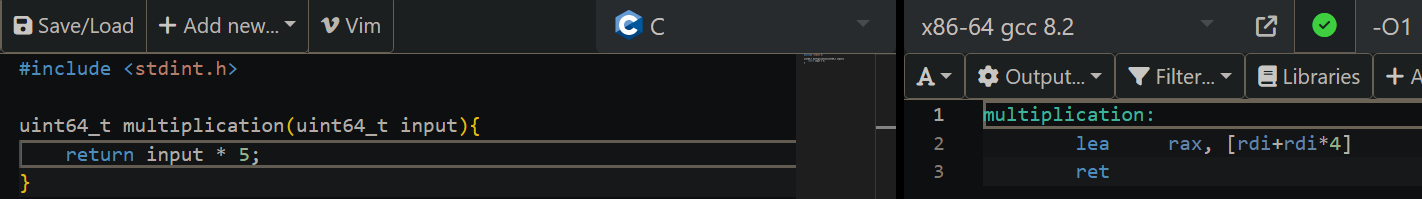
* General Function:



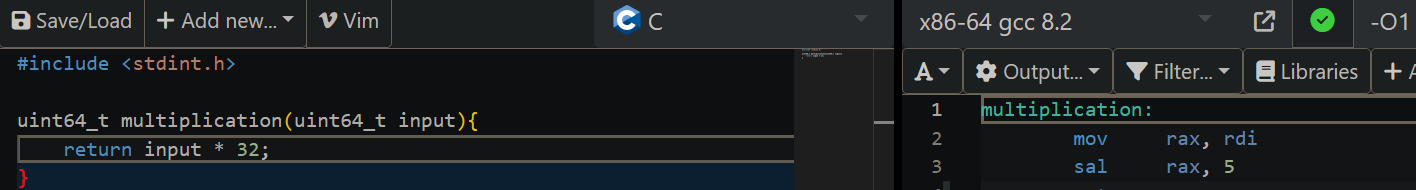
* 4:



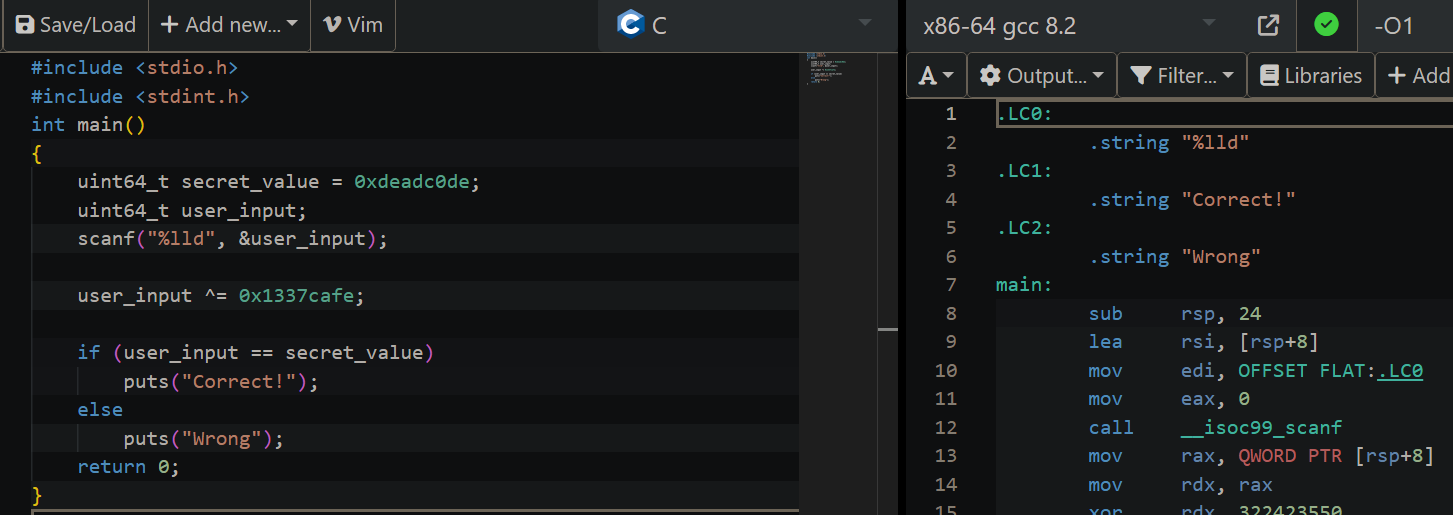
* 5:



* 32:



1. *Check out the following simple password checking code. Understand how this code works and what the corresponding assembly code is doing.*

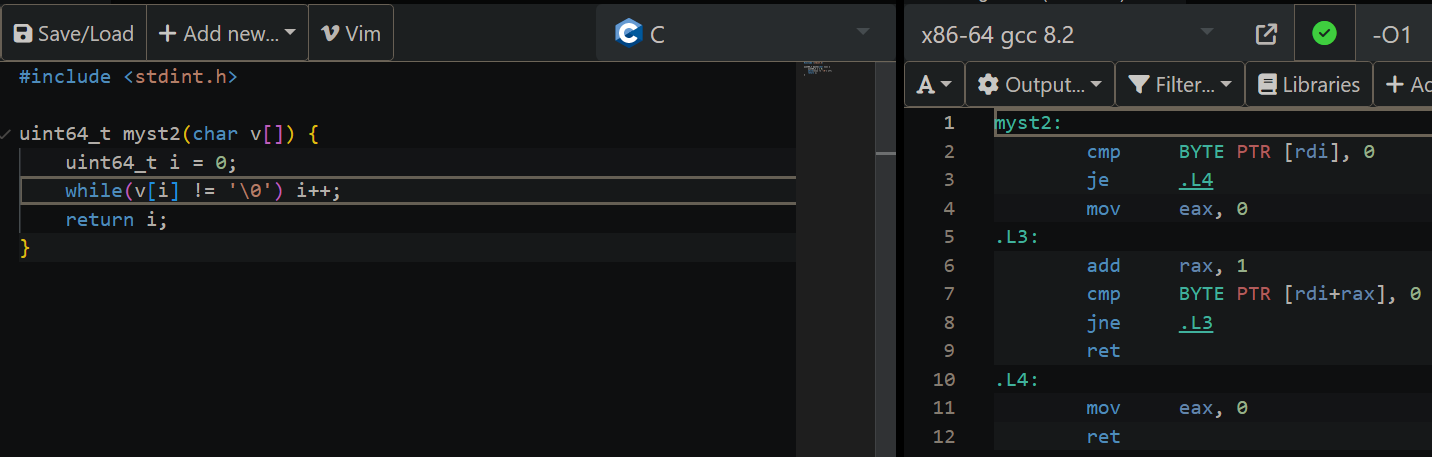


Full output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | .LC0:  .string "%lld"  .LC1:  .string "Correct!"  .LC2:  .string "Wrong"  main:  sub rsp, **24**  lea rsi, [rsp+**8**]  mov edi, OFFSET FLAT:.LC0  mov eax, **0**  call \_\_isoc99\_scanf  mov rax, QWORD PTR [rsp+**8**]  mov rdx, rax  xor rdx, **322423550**  mov QWORD PTR [rsp+**8**], rdx  mov edx, **3449424416**  cmp rax, rdx  je .L5  mov edi, OFFSET FLAT:.LC2  call puts  .L3:  mov eax, **0**  **add** rsp, **24**  ret  .L5:  mov edi, OFFSET FLAT:.LC1  call puts  jmp .L3 |

**3. Assembly to C code conversion**

3.1. Code 1:



C code:

#include <stdint.h>

**uint64\_t** **myst2**(**char** v[]) {

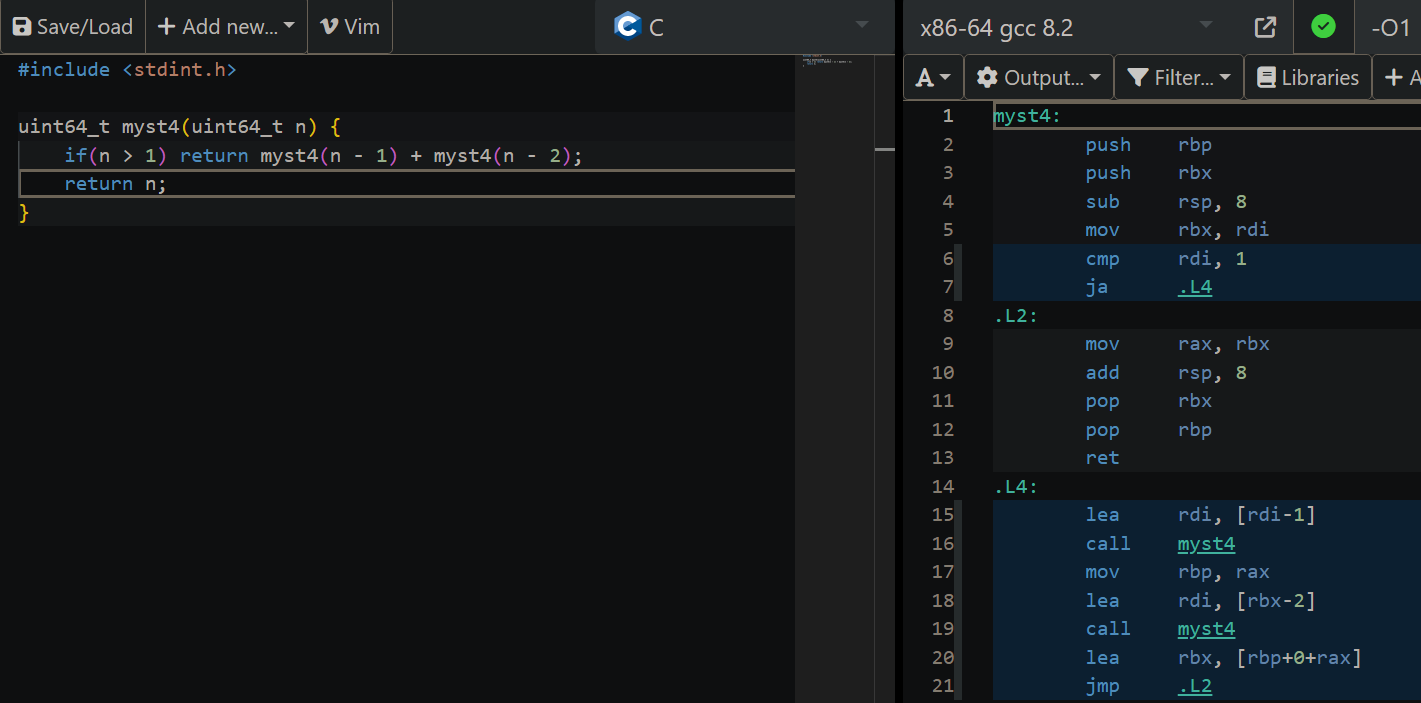
**uint64\_t** i = **0**;

**while**(v[i] != '\0') i++;

**return** i;

}

3.2. Code 2:



C code (the Fibonacci sequence):

#include <stdint.h>

**uint64\_t** **myst4**(**uint64\_t** n) {

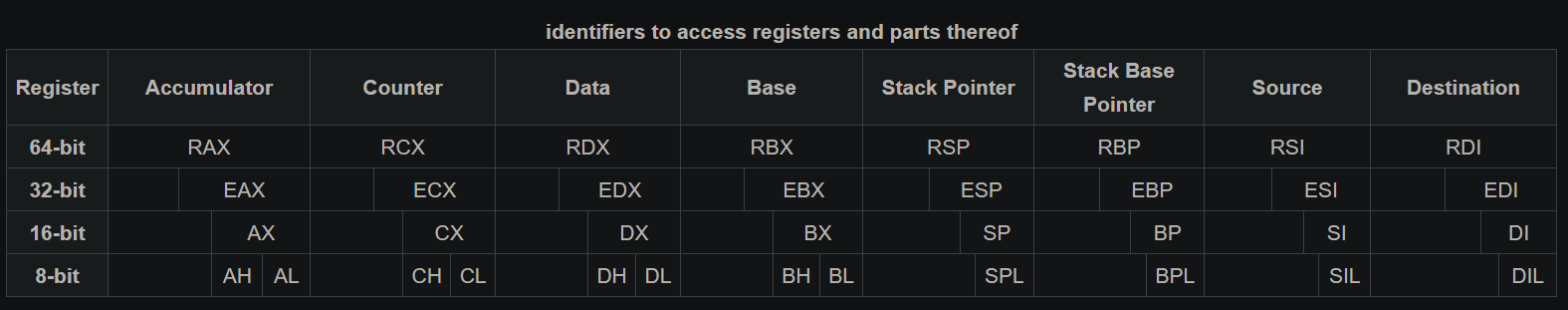
**if**(n > **1**) **return** myst4(n - **1**) + myst4(n - **2**);

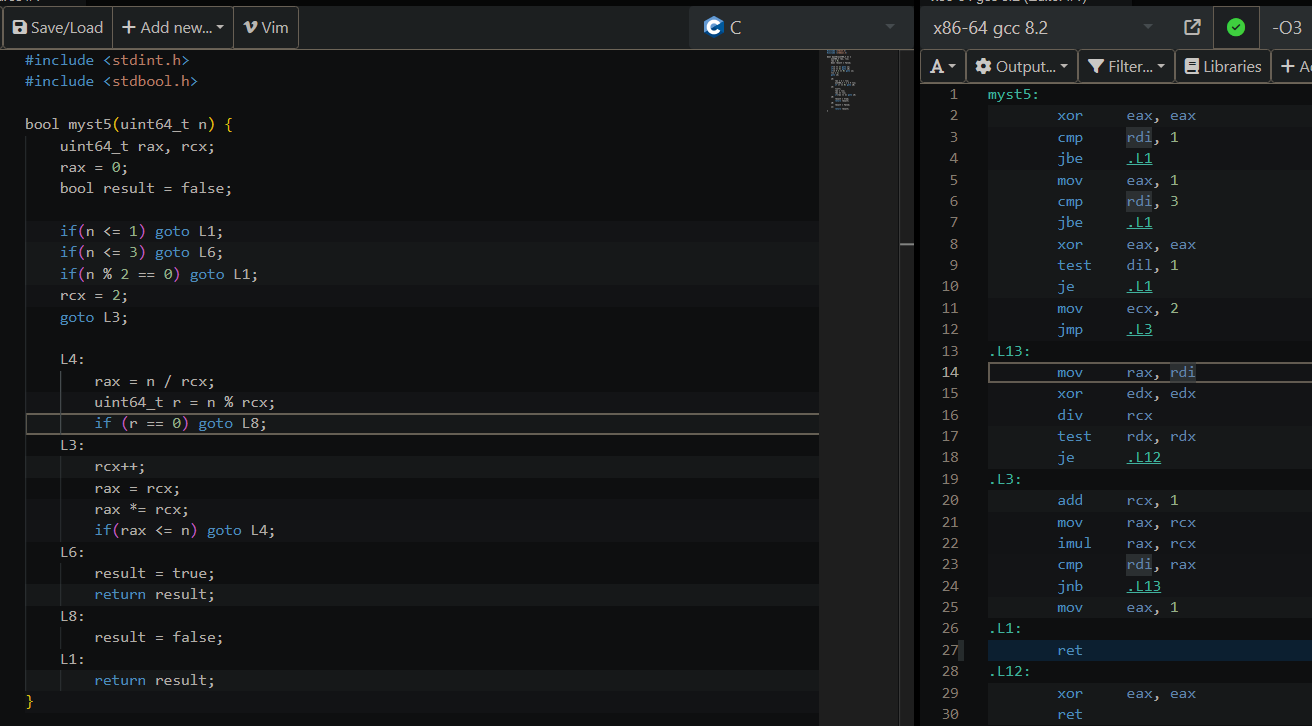
**return** n;

}

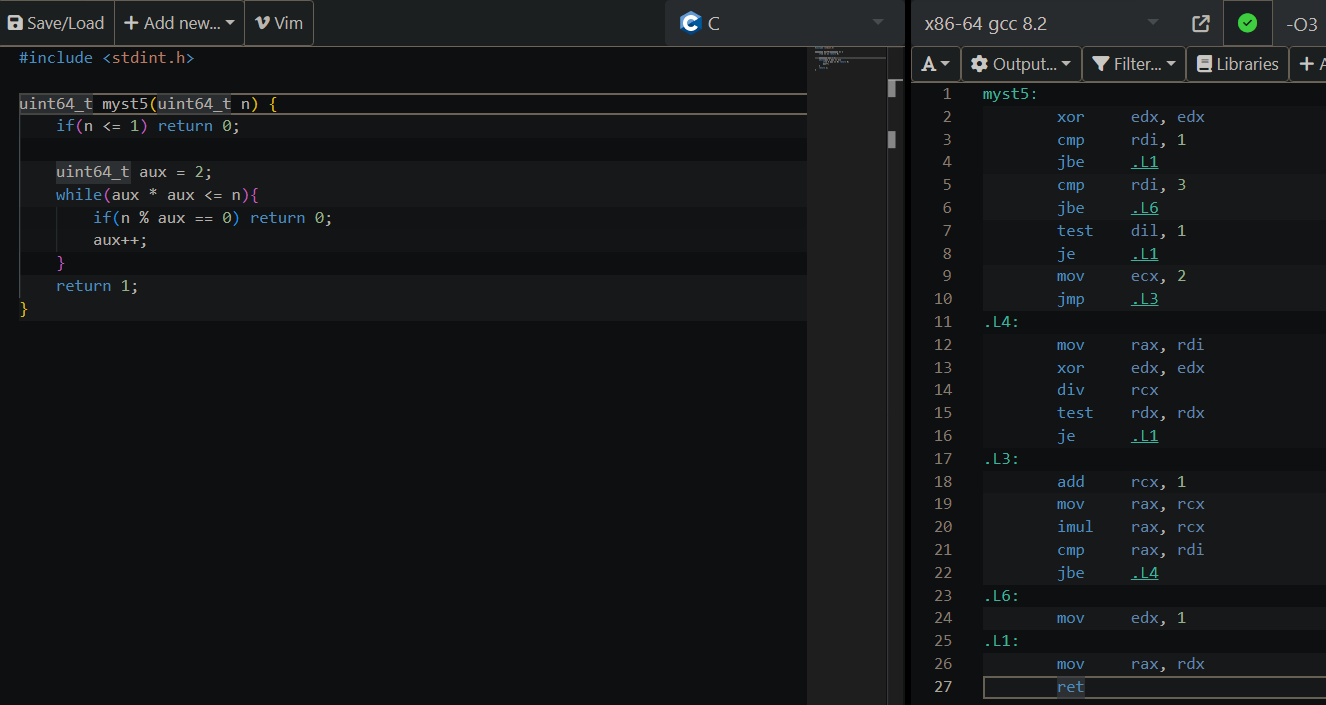
3.3. Code 3:

Just to remember:





I did a „translation” of the assembly code and we can notice we obtain almost the same output (the logic is the same). If we refine the code, we realize the code just checks whether or not a number is prime, so we can write:



C code (prime number):

#include <stdint.h>

**uint64\_t** **myst5**(**uint64\_t** n) {

**if**(n <= **1**) **return** **0**;

**uint64\_t** aux = **2**;

**while**(aux \* aux <= n){

**if**(n % aux == **0**) **return** **0**;

aux++;

}

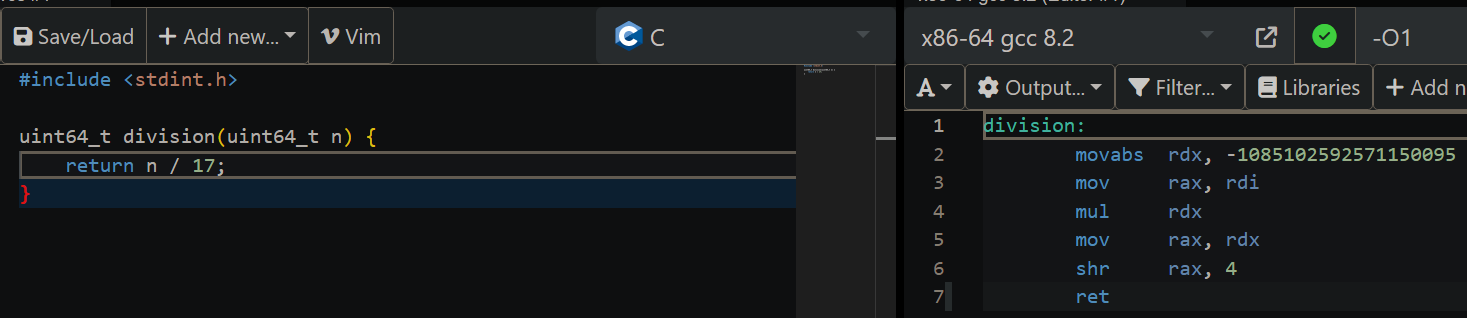
**return** **1**;

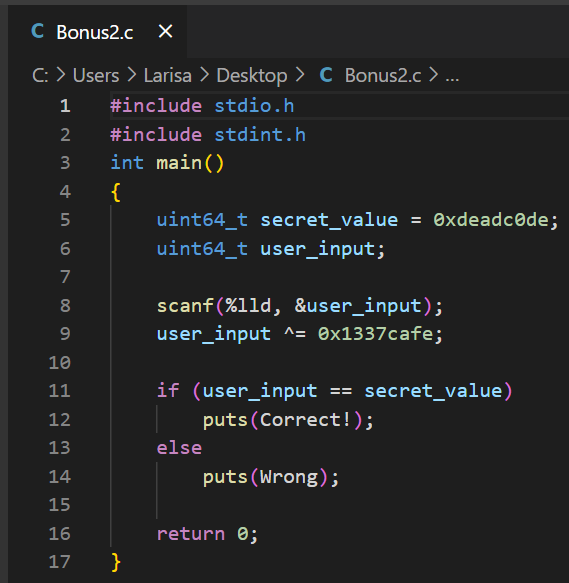
}

**4. Bonus 1**

At a first glance, we can notice that this assembly is very similar with the function that divides a number by 5. So, following the same logic, we know we have a number ***n*** that will be divided by another one ***x***.

And the assembly has this instruction: *shr rax, 4*, right before giving the final answer, so:



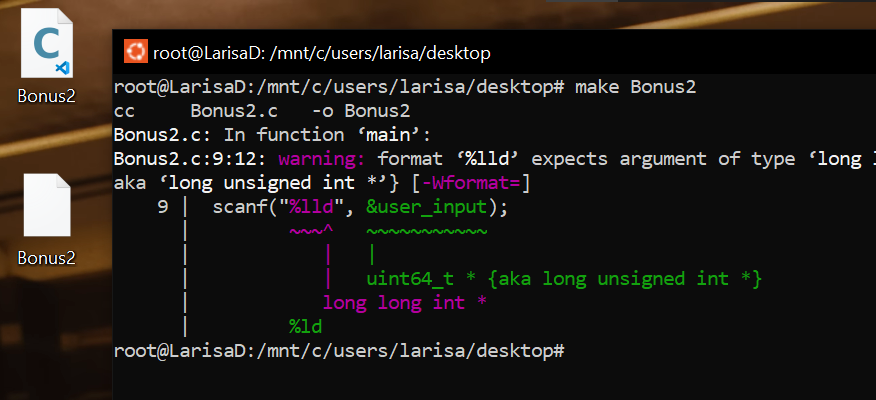
****So the function just takes a number and returns the result of the operation (number / 17).

**5. Bonus 2**

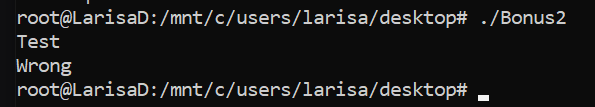
*Take the last piece of code presented in Section 2, write the C program on your computer and compile it with gcc. Edit the binary file (not the source code!) to make it print Correct! when the wrong secret value is given and vice-versa.*

First step (writing the code: „*Bonus2.c*”):

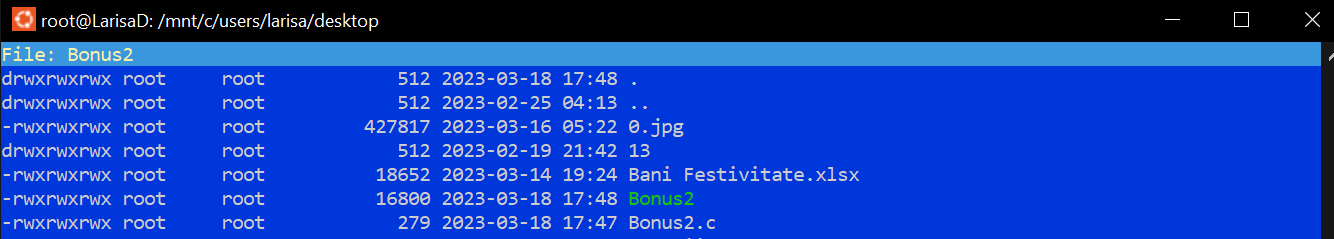
I will use Linux Subsystem for Windows to compile:



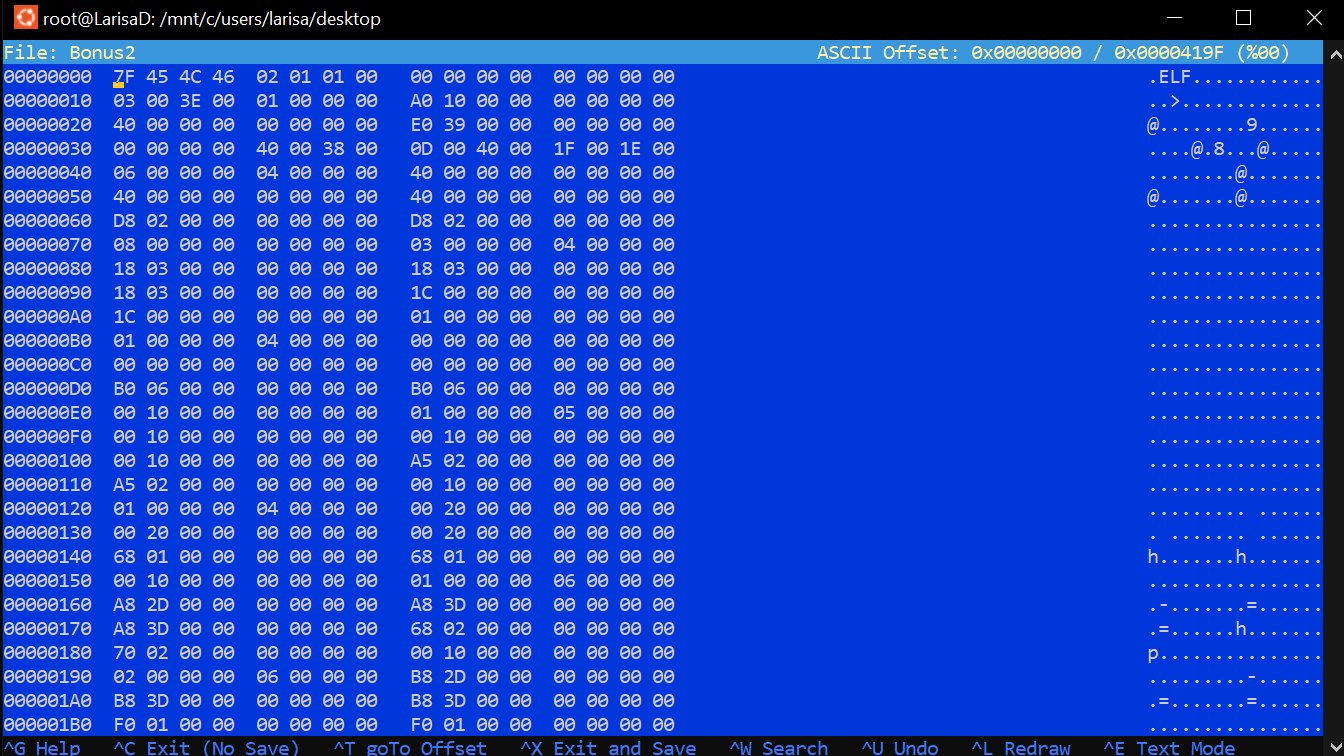
Test:



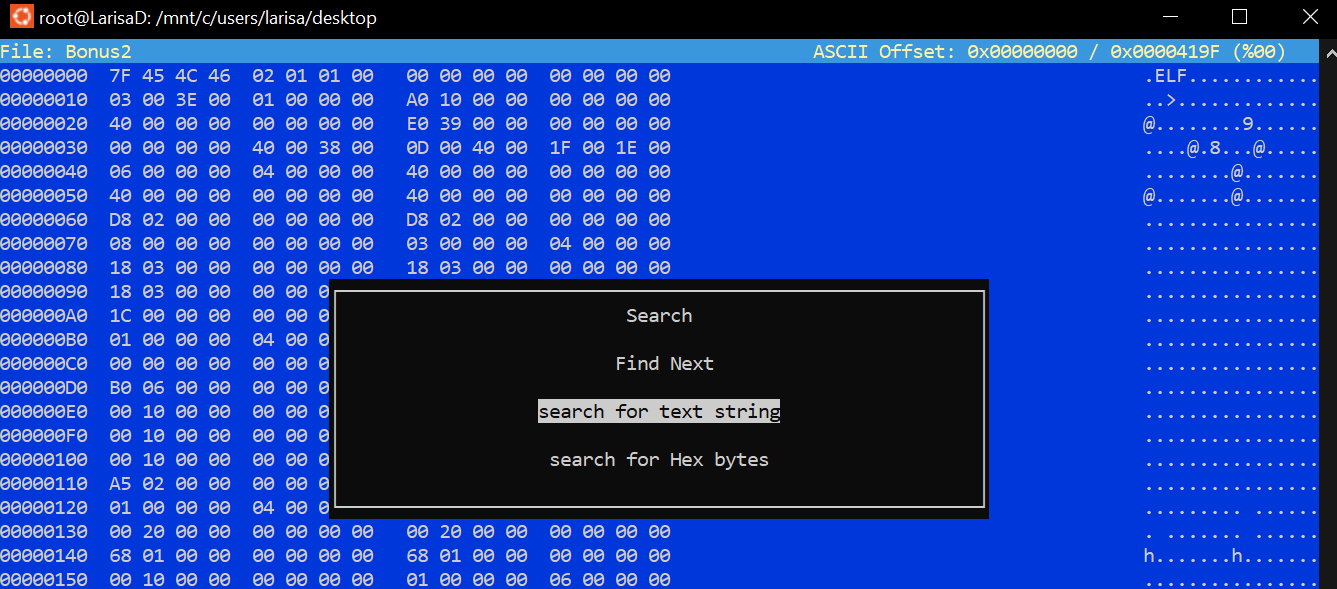
Command: ***hexeditor*** and select *Bonus2.out*



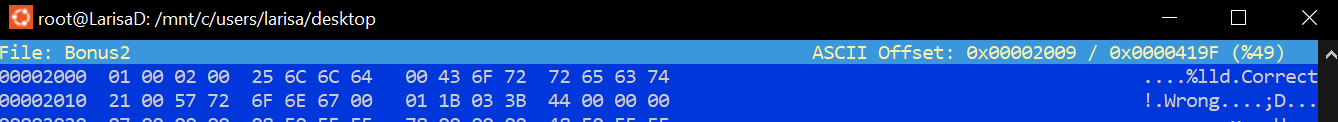
And we get:



*CTRL+W*:



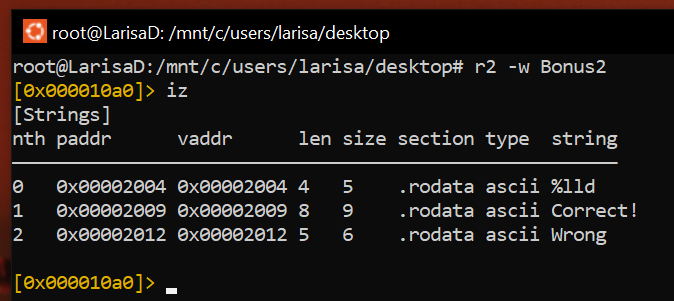
And we search: *Correct*



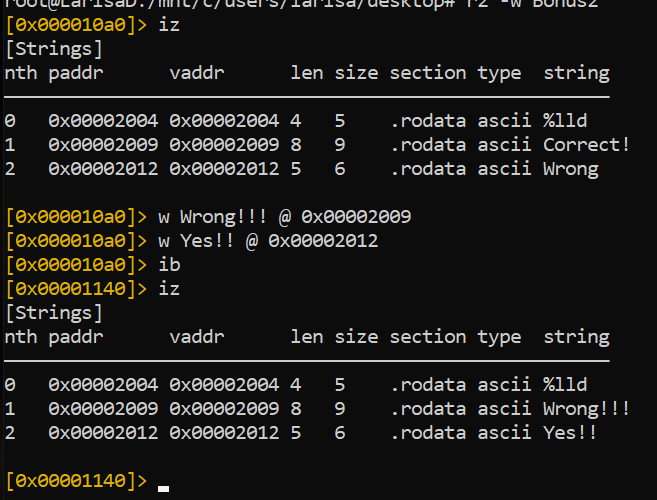
We can see that:

* 43 6F 72 72 65 63 74 21 → Correct! (len = 8)
* 57 72 6F 6E 67 → Wrong (len = 5)

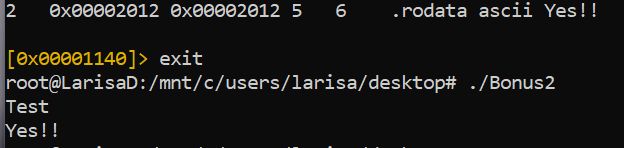
If we attempt to just swap the strings, we will get weird values. That is because the program will expect the length of the original strings, but, as you can see the 2 words have different lengths ([source1](https://reverseengineering.stackexchange.com/questions/17415/how-to-change-a-string-in-a-arm-32bit-lsb-executable-binary-using-radare2), [source2](https://reverseengineering.stackexchange.com/questions/26887/cant-modify-string-in-radare2-rodata-section)): ***r2 -w Bonus2***, ***iz*** (**radare2**)



We can use **radare2** to swap the 2 strings, using the commands *w Wrong!!! @ 0x00002009*, *w Yes!! @ 0x00002012*:

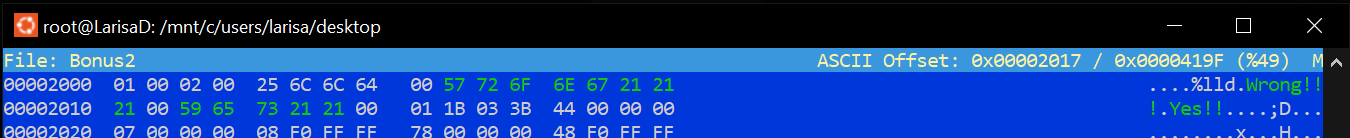


If we perform the same test as above, we get:

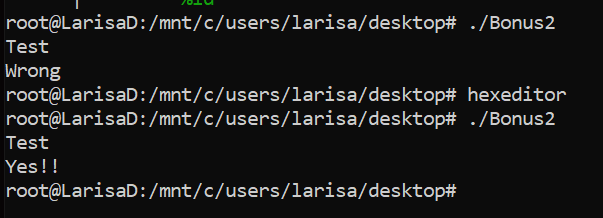


But, the exercise asks to edit the binary file so let’s do that:

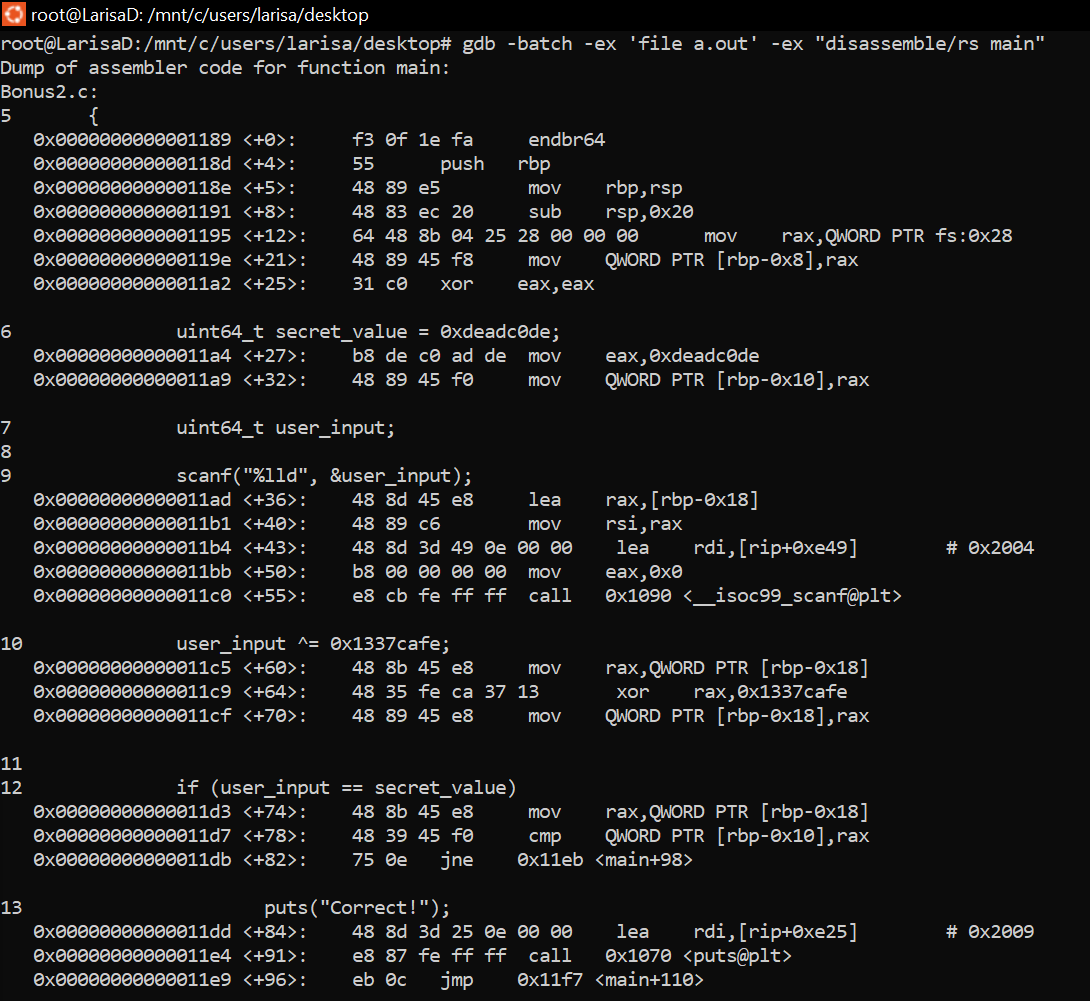
* 57 72 6F 6E 67 21 21 21 → Wrong!!! (len = 5 + 3 = 8, just like the original)
* 59 65 73 21 21 → Yes!! (len = 5, just like the original)



And now, if we perform the same test as above, we get:

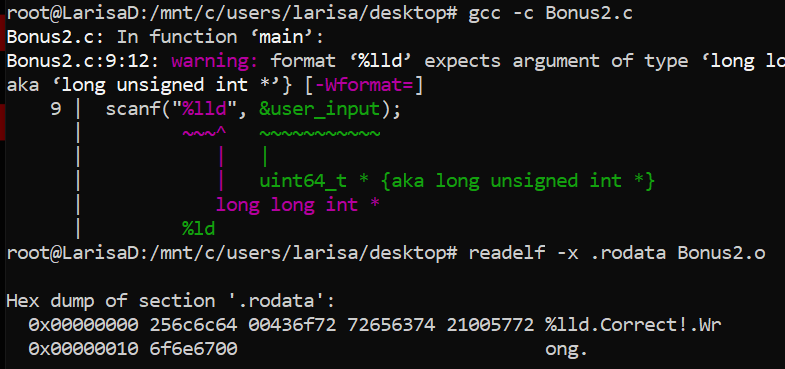


Ok, now, let’s attempt to really swap the 2 strings. We might think that the 2 values are stored in a registry in assembly that we can modify, but that is not the case. Commands: ***gcc -std=c99 -O0 -g Bonus2.c***, ***gdb -batch -ex 'file a.out' -ex "disassemble/rs main"*** ([source](https://stackoverflow.com/questions/6441721/can-i-give-objdump-an-address-and-have-it-disassemble-the-containing-function) or [tutorial](https://www.youtube.com/watch?v=JcnCp4IWZvQ))

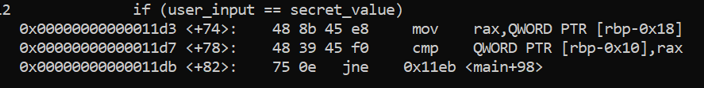


The full output is in *output.txt*.

Hard coded strings are stored in rodata, as we saw in a photo from above or here: ***readelf -x .rodata Bonus2.o*** ([source1](https://stackoverflow.com/questions/40027227/gcc-generate-o-file-instead-of-executable), [source2](https://stackoverflow.com/questions/1685483/how-can-i-examine-contents-of-a-data-section-of-an-elf-file-on-linux))



But, what we can do, instead of trying to swap the 2 strings, we can change the ***JNE*** command to ***JE*** command (at line 12, if user\_input is not equal to secret\_value, will jump to „Wrong”).



***JNE*** has the hex code ***0x750E*** and ***JE*** has ***0x740E***. So, using hexeditor, we search for hex bytes and change the value ([source](https://rderik.com/blog/using-radare2-to-patch-a-binary/)):

