nasm assembly
-f = formate of assembly (ef eg = elf64)
-g = for enabling debugging
-d = specify output file
-l = create listing file (.lst) containing assembly code with addresses.

⇒ when you insert 32 bit (or lower bit) in 64 bit rax (accumulator) upper 32 bits are cleared and lower 32 bits are filled with the value.

Note → gcc -nostartfiles tells gcc not to include standard start files like crt1.o, which causing multiple definition of _start issue.

Conditional jumps & branching ⇒
Conditional jump program ⇒

⇒ extern printf ; declare the external func printf
section .data
number1 dq 42
number2 dq 41
fmt1 db "Number1 > Number2", 10, 0
; for printing string for comparision
fmt2 db "Number1 < Number2", 10, 0
section .text
global main
main:
push rbp ; save the base pointer
move rbp, rsp ; set up the base pointer
mov rax, [number1] ; load the number1 in accm

jumpifequal / jumpinstruction ⇒ je = jumpifequal
ZF

section .data
value1 db 10
value2 db 10
section .text
global _start
_start: ← here not used
mov al, [value1] ; byte because it is 8 bit
mov bl, [value2]
cmp al, bl
je eq_found ; label
notequal:
; your code for not equal

equal_found:

done:
; exit code here

jg = jump if greater

based on SF (sign flag), OF (overflow flag), ZF (zero flag)

jne = jump if not equal (if ZF = 0)

... section .data
value1 db 10
value2 db 20
section .text
global _data
_data: ← here used 16 bit
mov al, byte [value1]
mov bl, byte [value2]
cmp al, bl
jg label:

not greater:

jmp done

label:

done: ; exit code here
mov rax, 60 ← set system call no 60 for exit
xor rdi, rdi
syscall

jge = jump greater than equal : based SF and OF

```
section .data
    value db 30
    value1 db 35
section .text
    global _start
_start:
    mov al, byte [value]
    mov bl, byte [value1]
    cmp al, bl
    jge greater
```

→ jl = jump if less

→ to determine the whether the value less than other

jl instruction operations:-
1. Comparision
2. Conditional assembly
3. branching

```
section .data
    value1 db 15
    value2 db 20
section .text
    global _start ; int comp
_start:
    mov al, [value1]
    mov bl, [value2]   ← moves the value pointed by value2 in (bl) register
    cmp al, bl
    jl less ; jump to less if [al] < [bl]
notless:
notgreater:
    jmp done
greater:
done:
    mov rax, 60
    xor rdi, rdi
    syscall
```

↑ fiscal no. for exit

jumpdone ;

less: done: ~~rax~~ mov rax, 60
mov rdi, 0
syscall

The difference b/w above & greater jump instruction is that above instructions are used unsigned comparisons or signed where greater is used for signed integer comparisons.

jle = jump if less or equal = direction the flow of program based on the following operation to compare signed numbers 1. compare 2. conditional check 3. jump or continue

eg = jle label

(equals ZF=0 and CF=0 or ZF=1 and CF=0 have the same value for jle (negative or equal))

jump if above = ja ; determine whether one value is strictly greater than another.

→ compares values in registers, immediate values or values in memory locations.

→ jump if condition above yields true.

if CF = 0, ZF = 0 (carry flag) →

```
cmp cx, dx
ja above_label
```

jumps if value if cx register is strictly greater the dx reg.

↳ contains code for value not above or

jae = jump above or equal

→ jump if one value is strictly greater or equal to another value (CF = 0)

jump if below = jb : used for signed or unsigned integers comparisions.

eg: ; value1 is below value2

```
cmp cx, dx
jb label
```

jbe (jump if below or equal)

calculator program → extern printf

```
section .data
    number dq 5 ; 8 bytes quadword
    fmt db "The sum from 0 to %ld is %ld", 10, 0   (10) character
```

```
section .bss ; used to reserve memory for uninitialized vars
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    mov rcx, [number] ; loop counter
    mov rax, 0
bloop:                ; loop
    add rax, rcx
    loop bloop ; automatically decrement rcx
    mov rdi, fmt ; repeat loop until rcx = 0
    mov rsi, [number]
    mov rdx, rax      ; calculates the sum of number 0 to __
    mov rax, 0
    call printf
    mov rsp, rbp      ; to measure the time of execution of a executable
    pop rbp           use:- time ./a.out
    ret
```

→ accumulator is the last argument passed to the printf it should be cleared before passing to printf. ; indicates that there are no floating point arguments ensuring that only integer values are printed.

→ printf →

mov rdi, fmt ÷ string passed to
mov rsi, [number] ÷ printf
mov rdx, rax ÷ third argument to printf

exabyte → (EB) unit of memory ; 1 exabyte = 1024 Peta bytes

```
program → section .data
    bNum1 db 123
    wNum dw 12345      ; words size var
array (5) size → w array times 5 dw 0
containing 0s  dNum dd 12345   ; double word
    dNum1 dq 12345
    text1 db "abc", 0
    qNum dq 3.145
    text2 db "cde", 0
section .bss
    bvar resb 1   ; reserve byte
    dvar resd 1   ; reserve 4 byte of space
    wvar resw 10
    qvar resq 3
section .text
    global main          for correct debugging
main: push rbp ; push the value of rbp on stack
    mov rbp, rsp ; setting stack frame
    lea rax, [bNum] ; loads memory address of bNum
    mov rax, bNum
    mov rax, [bNum]
    mov [bvar], rax
    lea rax, [bvar]
    lea rax, [wNum]
    mov rax, [wNum]
    lea rax, [text1]
    mov rax, text1
    mov rax, text1+1 ; load the value of second character of text
    lea rax, [text1+1]
```

```
mov rax, [text1]
mov rax, [text1+1]
mov rsp, rbp; clean
pop rbp
ret
```

lea - load effective address
used to store addresses of the registers.
data is stored in right endian, fashion.
(rightmost bytes are smaller)

al - to store 1 byte data
(register)
(accumulator)

```
;calculator.asm
extern printf
section .data
    number1 dq 128
    number2 dq 19
    neg_number dq -12
    fmt db "The numbers are %ld and %ld", 10, 0
    fmtint db "%s %ld", 10, 0
    sumi db "The sum is", 0
    difi db "The difference", 0
    inci db "Number1 increment", 0
    deci db "Number1 decremented", 0
    sali db "Number1 shift left(*4):", 0
    sari db "Number1 shift right(/4):", 0
    sarie db "Number1 shift right 2(/4) with sign expression:", 0
    divi db "The integer quotient is", 0
    remid db "The modulo is", 0

section .bss
    result1 resq 1
    modulo resq 1
```

;ld - long integers

```
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    ;display the numbers
    mov rdi, fmt
    mov rsi, [number1]
    mov rdx, [number2]
    mov rax, 0
    call printf
    ;adding
    mov rax, [number1]
    add rax, [number2]
    mov [result1], rax
    ;display the result
    mov rdi, fmtint
    ;load address of format string
    mov rsi, sumi
    mov rdx, [result1]
    mov rax, 0
    call printf
    ;subtracting
    mov rax, [number1]
    sub rax, [number2]
    mov [result1], rax
    ;display the result
    mov rdi, fmtint
    mov rsi, difi
    mov rdx, [result1]
    mov rax, 0
    call printf
    ;incrementing
    mov rax, [number1]
    inc rax ; increment rax by 1
    mov [result1], rax
    ;display the result
    mov rdi, fmtint
    mov rsi, inci
    mov rdx, [result1]
    mov rax, 0
    call printf
    ;decrementing
    dec rax;
    ;displaying the ---(...)
    ;shift left
    mov rax, [number1]
    sal rax, 2 ; shift accumulator left by 2 bits
```

rdi - 1st
rsi - 2nd
rdx - 3rd
} arguments to printf

```
    mov [result1], rax
    ;display the result
    mov rdi, fmtint
    mov rsi, sali
    mov rdx, [result1]
    mov rax, 0
    call printf
    ;shift right
    (...) sar rax, 2; shift
    ; accumulator by 2 bits
    ;display the result
    (...)
    ;shift right with sign extension
    mov rax, [neg_number]
    sar rax, 2
    mov [result1], rax
    ;display the result (...)
    ;Multiply
    mov rax, [number1]
    imul qword [number2]
    ;multiplies rax with [number2]
    ;64bit multiplication
    mov [result1], rax
    ;display the result
    mov rsi, multi
```

idiv - for signed division
imul - for signed multiplication (can take 1, 2, or 3 operands)

; division
mov rax [number1]
mov rdx, 0 ; remainder (register) for division
idiv qword [number2]
mov [result], rax ; quotient of div
mov [modulo], rdx ; remainder of div
; display the quotient

mov rsp, rbp
pop rbp
ret

rbp - stack pointer for accessing local variables

printf - rdi, rsi, rdx → arguments of printf

for unsigned - OF = set (flag)
when result is large to fit in (carry flag)
memory location CF = 1

mul (instruction) - for unsigned multiplication.

stack manipulation - registers are very fast units used for program execution.

Stack - is data structure. Used to control flow of execution of program.
→ starts from high at memory, grows downward.
→ stack acts as temp memory preserving registers values and facilitating data transfers to functions

Program - extern printf
section .data
mystring db "ABCD", 0
mystring_len eq $ - mystring

$ - current address while execution of disassembly program

when result of operands is larger than 64 bit it stores lower 64 bit is in rax and upper 64 bit in rdx.

equ - is for defining constants.

fmt1 db "The original string: %s", 10, 0
fmt2 db "Reversed string: %s", 10, 0

section .text
global main → push rbp
main: mov rbp, rsp
mov rdi, fmt1
mov rsi, mystring
mov rax, 0
call printf
xor rax, rax
mov rbx, mystring
mov rcx, mystring_len ; counter
mov r12, 0 ; pointer (r12)
pushLoop: mov al, byte [rbx + r12]
push rax
inc r12
loop pushloop
mov rbx, mystring ; pop characters from stack
mov rcx, mystring_len
mov r12, 0 ; r12 as a pointer (char pointer)
popLoop: pop rax
mov byte [rbx+r12], al
inc r12
loop poploop
mov byte [rbx+r12], 0 ; NULL character for termination of string
mov rdi, fmt2
mov rsi, mystring
mov rax, 0
call printf
mov rsp, rbp ; restore stack pointer
pop rbp ; restor stack pointer.
ret

immediate values & constants can't be pushed

assembly doesn't have case structures etc.
But can be wedby using own functions.

function = calculate area of a circle using function.

```
→ extern printf
section. data
    radius dq 10.0
    pi  dq 3.14
    fmt db "The area of the circle is %.2f", 10,0
section .text
    global main
main: push rbp
    mov rbp, rsp
    call area ; calling area to calculate - -
    mov rdi, fmt
    movsd xmm1, [radius] ; floating point registe
                ↑ instruction to ~~load~~ floating point nums.
    mov rax, 0
    call printf
    leave ; cleans stack frame
    ret
area: ; function area
    push rbp
    mov rbp, rsp
    movsd xmm0, [radius]
    mulsd xmm0, [radius]
    mulsd xmm0, [pi]
    leave
    ref
```

localvariables:- extern printf

```
section .data
    radius dq 10.0
section .text
area:
    section .data
    .pi dq 3.14
    section .text
    push rbp
    mov rbp, rsp
    movsd xmm0, [radius]
    mulsd xmm0, [radius]
    mulsd xmm0, [pi]
    leave
    ret
circuference:
    section .data
    .pi dq 3.14
    section .text
    push rbp
    mov rbp, rsp
    movsd xmm0, [radius]
    addsd xmm0, [radius]
    mulsd xmm0, [pi]
    ~~leave~~
    ret
circle:
    section .data
    format_area db "Area: %f", 10,0
    format_circumfe db "cir is %.f", 10, 0
section .text
    push rbp
    mov rbp, rsp
```

```
Callarea
mov rdi, format_area
mov rax, 1
call printf

call circumference
mov rdi, format_circum
    -ference
mov rax, 1
call printf
leave
ret
global main
main: push rbp
    mov rbp, rsp
    Circle
    leave
    ret
```

xmm0 - is 64 bit SSE (streaming simd extension) capable of holding the floating point numbers.

.locat var.
↑ mean it is local var particular to the function only.

Stack frame = when stack pointer gets some data pushed it is decreased by 8 bytes.

SIMD instructions = single instruction multiple data instructions

→ eight byte return address is aligned before start of the program.

if rsp = 0, then its 16 byte

```
extern printf
section .data
formatText db "2 Times Pi equals: %.14f"
Pi dq 3.14159265 8979
section .text
funct3: push rbp
    movsd xmm0, [Pi]
    addsd xmm0, [Pi]
    mov rdi, formatText
    mov rax, 1
    call printf
    pop rbp
    ret
funct2: push rbp
    call funct3
    pop rbp
    ret
funct1:
    push rbp
    call funct2
    pop rbp
    ret
global main ; entry point of program is main
```

```
main:
    push rbp
    call funct1
    pop rbp
    ret
```

rsp is stored in rbp and restored from rsp on exit function.

for debugging stack alignment is crucially required.

making own extern al function → extern := assures assembler that the function is not to be searched in source code but available as a object file where.

```
→ extern printf
  extern c_area
  extern c_circumference
  extern r_area
  extern r_circum
  global pi ; can be accessed from external function also
section .data
  pi dq 3.14159 2
  radius dq 10.0
  side1 dq 4
  side2 dq 5
  format_f db "%.5%f", 10, 0
  format_i db "%.5%d", 10, 0
  ca db "The circle area is:", 0
  cc db "The circle circum is", 0
  ra db
  rc db
section .text
global main
main: push rbp
    mov rbp, rsp
    movsd xmm0, qword[radius]
    call c_area
    mov rdi, format_f
    mov rsi, ca
    mov rax, 1
    call printf
    movsd xmm0, qword[radius]
    call c_circum
    mov rdi, format_f
    mov rsi, cc
    mov rax, 1
    call printf
    mov rdi, [side1]
    mov rsi, [side2]
    call r_area
    mov rdi, format_i
    mov rsi, ra
    mov rdx, rax
    mov rax, 0
    call printf
    mov rdi, [side1]
    mov rsi, [side2]
    call r_circum
    mov rdi, format_i
    mov rsi, rc
    mov rdx, rax
    mov rax, 0
    call printf
    mov rsp, rbp
    pop rbp
    ret
```

circle.asm

```
extern pi; globalvar pi
section .data
section .bss
section .text
global c_area
c_area: push rbp
        mov rbp, rsp
movsd xmm1, qword[pi]
mulsd xmm0, xmm0
mulsd xmm0, xmm1
mov rsp, rbp
pop rbp
ret
global c_circumference
c_cir___:
push rbp
mov rbp, rsp
movsd xmm1, qword[pi]
add xmm0, xmm0
mulsd xmm0, xmm1
mov rsp, rbp
pop rbp
ret
```

rectangle.asm

```
section .data
section .bss
section .text
global r_area
r_area:
push rbp
mov rbp, rsp
mov rax, rsi
imul rax, rdi
mov rsp, rbp
pop rbp
ret
global r_circum___
r_circu___:
push rbp
mov rbp, rsp
mov rax, rsi
add rax, rdi
add rax, rax
mov rsp, rbp
pop rbp
ret
```

makefile — # → for comment

```
myfunction: mainfunction.o circle.o rectle.o        (seperated by space)
        gcc -o myfunction.o my_.o circle.c rect__.c
myfunction.o: myfunction.asm
        nasm -f elf64 -g -F dwarf myfunction.asm
        -l myfunction.lst
```

similar for all external function assembly files

Preserving register → arguments of non floating point nature are for any function)

integer	floating point
first → rdi	→ xmm0
rsi	→ xmm1
rdx	;
rcx	;
r8	
r9	→ xmm7

before calling printf stack should be 16 byte aligned.

additional arguments are transmitted via stack.

```
→ extern printf
section .data
    first db "A",0
    second db "B",0
    Third db "C",0
    forth db "D",0
    fifth db "E",0
    ;
    ;
    ;
    ; ,0
formatstring "This string is :%s%s%s%s%s%s%s ... %s",10,0
format_string1 "PI value = %f",10,0
pi dq 3.1415
section .bss
section .text
global main
main:
    push rbp
    mov rbp, rsp
    mov rdi, format_string
    mov rsi, first
    mov rdx, second
    mov rcx, third
    mov r8, fourth
    mov r9, fifth
    ~~mov r9, sixth~~
    push tenth
    push nineth
    push eighth
    push seventh
    push sixth
    mov rax, 0
    call printf
    and rsp, 0xffff...016fs ; to align it to 16 bytes [last should be 0]
    movsd xmm0, [pi]
    mov rax, 1
    mov rdi, format_string1
    mov rax, 1
    call printf
    leave
    ret
```

According System V AMD64 ABI calling convention remaining are passed to stack and printf automatically gets them from stack.

sub rsp, 8 (to 16 byte align the stack)

Program to separate the string 64 bits into 8 byte with 8 bit each:- extern printb
extern printf

```
section .data
    msg n1 db "Number1",10,0
    msg n2 db "Number2",10,0
    msg1 db "XOR",10,0
    msg2 db "OR",10,0
    msg3 db "AND",10,0
    msg4 db "NOT number1",10,0
    msg5 db "SHL 2 lower
    lowerbytes of number1",10,0
    msg6 db "SHR2 lowerbyte
    of number1",10,0
    msg7 db "SAL 2 lower
    byte of number2",10,0
    msg8 db "SAR2 ——
    msg9 db "ROL 2 lowerbyte
    of number1",10,0
    msg10 db —— number2",10,0
    msg11 db "ROR 2 lower
    byte of number1",10,0
    msg12 ——— number2",10,0
    number1 dq -72
    number2 dq 1064
section .bss
section .text
    global main
main:
    push rbp
    mov rbp, rsp
    mov rsi, msgn1
    call printmsg
    mov rdi, [number1]
    call printb
    mov rsi, msgn2
    call printmsg
    mov rdi, [number2]
    call printb
    ;to print binary
    ; representation
    ; of number
    mov rsi, msg1
    call printmsg
    mov rax, [number1]
    xor rax, [number2]
    mov rdi, rax
    call printb
    mov rsi, msg2
    call printmsg
    mov rax, [number1]
    or rax, [number2]
    mov rdi,
    call printb
    mov rsi, msg3
    call printmsg
    mov rax, [number1]
    and rax, [number2] ; bitwise and.
    mov rdi, rax
    call printb
; function to print string
```

```
...'not' operation is same
    mov rsi, msg5
    call printmsg
    mov rdx, [number1]
    shl al, 2 ; shift lowerbyte 'al' of rax by 2 position
    mov rdi, rax
    call printb
    shr — same
    rol al, 2
    rol - rotate left lower byte al of rax by 2 position
    sal - arithmatic shift left
    sar - ——— right
    ror: — — — right — —
    leave
    ret
printmsg: section .data
    fmtstr db "%s",0
section .text
    mov rdi, fmtstr
    mov rax, 0
    call printf
    ret
printb:- have same argument register hierarchy as printf.
```

printb.c :- ⑧

```
#include <stdio.h>
Void printb (long long n)
{ for(c=63
    long long s, c;
    for(c=63; c>=0; c--)
    { s = n >> c;
    if((c+1)%8 == 0)
        printf(" "), //print space for
                    //every 8 bit grp
    if(s & 1)
        printf("1");
    else
        printf("0");
    printf("\n");
}
```

1000

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
                              72  1 0
```

makefile:

```
bit1: bit1.o printb.o
    gcc -o bit1
bit.o printb.o -no-pie
    nasm printb.c
printb.o : printb.c
    gcc -c -o printb.o
        printb.c
```

Assembly

macros → functions reduces the performance of the program. Macros can be used in place of that. Macros are set of instruction. Whenever macro is called NASM replaces macro with it definition. Processed by Macro processor in Preprocessor.

macro → extern printf

single line macro: %define double_it (r) sal r, 1

multi line macro: %macro printf 2 ← mean two args

```
%macro printf 2
section .data
%%arg1 db %1,0
%%fmtint db "%s %ld",0
section .text
mov rdi %%fmtint
mov rsi, %%arg1
mov rdx, [%2]   <- second parameter value
mov rax, 0
call printf
%endmacro
```

; %1 for reference of first arg in macro

```
section .data
    number dq 15
section .bss
section .text
global main
main: — —
    printf "The number is", number
    mov rax, [number]
    double_it(rax)
    mov [number], rax
    printf "The number is", number
    leave
    ret
```

%% :- for variables in macro, with %% ensures creation of fresh variable using the unique number generated for each invocation of macro. % can also be used but on first call macro will work but if called again error occurs

consolIO:
```
section .data
    message db "Hello, World", 10, 0
    msgLen1 dq $ - message - 1
    message2 db "Input the text: ", 0
    msgLen2 dq
    message3 db "your input: ", 0
    msgLen3 dq
    input_len equ 10

section .bss
    input resb input_len + 1   ; reserve memory for input buffer
    terminator                 ; to calculate null → string terminator
                               ; input_length

section .text
    global main
main: push rbp
    mov rbp, rsp
    mov rsi, message1
    mov rdx, msgLen1
    call prints

    mov rsi, message2
    mov rdx, msgLen2
    Call prints
    mov rsi, input ; loads address of input buffer
    mov rdx, input_length
    call reads
    mov rsi, message3
    mov rdx, msgLen3
    call prints

    mov rsi, input
    mov rdx, input_length
    call prints

leav
Ret
prints:
    mov rax, 1
    mov rdi, 1 ; set rdi for standard output
    syscall
    leave
    ret
reads:
    mov rax, 0
    mov rdi, 0 ; for standard input
    syscall
    leave
    ret
```

This program contains buffer overflow.

Inputting without buffer overflow

```
section .data
    message1 db "Hello world", 10, 0
    message2 db "Input only (a-z): ", 0
    message3 db "your input: ", 0
    input_len equ 10
    newline dq 0xa ; newline char in hex = a
section .bss
    input resb inputLen + 1
section .text
    global main
main:
    push rbp ; save previous value of rbp on stack
    mov rdi, message1
    call prints
    mov rdi, message2
```

```
mov rdi, input
mov rsi, inputLen
call reads
leave
ret
```

'rsi' is the first argument to syscall unlike prints which have rdi

```
message3 — mov rdi, input
           call prints
           mov rdi, newLine
           call prints

prints: push rbp
        mov rbp, rsp
        push r12 ; saved callee onto stack
        xor rdx, rdx
        mov r12, rdi
.lengthloop:
        cmp byte [r12], 0     ; compare byte at the address in r12 with 0 = '\0' (NULL) of string
        je .lengthfound
        inc rdx
        inc r12
        jmp .lengthloop
.lengthfound:
        cmp rdx, 0 ; if length of string is 0
        je .done
        mov rsi, rdi ; to make rdi, 1
        mov rdx, 1
        mov rdi, 1
        syscall
.done:
        pop r12
        leave
        ret

reads:  section .data
        section .bss
          .inputchar resb 1
        section .text
        push rbp
        push r12
        push r13   } save values of r's
        push r14
          mov r12, rdi
          mov r13, rsi
          mov r14, r14 ← character counter
.readc:
        mov rax, 0
        mov rdi, 0
        lea rsi, [.inputchar]
        mov rdx, 1
        syscall
        mov al, [.inputchar]
        cmp al, byte [newLine] ; if input is new line
        je .done
        cmp al, 97 ; compare with ascii = 'a'
        jl .readc ; ignore if it is less than 97
        cmp al, 122
        jg .readc ; greater than 122
        inc r14
        cmp r14, r13
        ja .readc ; if counter > max length (ignore)
        mov byte [r12], al  }
        inc r12             } store the character in buffer
        jmp .readc          }
.done:
        inc r12
        mov byte [r12], 0 ; null character
        pop r14
        p— 3
           2
        leave
        ret
```

file operation, windows and linux file i/p are completely different

for linux: NR_READ } special
NR_WRITE } syscall symbols

flags of file operations in fcntl.h

```
section .data
  CREAT equ 1
  OVERWRIT equ 1
  APPEND equ 1
  O_WRITE equ 1 . flag related to write file
  READ equ 1 ; flag for read file operation
  O_READ equ 1
  DELETE equ 1
```

```
NR_READ equ 0
NR_WRITE equ 1
NR_OPEN equ 2
- close equ 3
- LSEEK equ 8
- CREATE equ 85
- UNLINK equ 87
; file flags
                    80's
O_CREATE equ 0000100q ; to create a file if
O_APPEND equ 00-2000q ; it doesn't exists
; ACCESS MODE
O_RDONLY equ 000000q
O_WRITE_ONLY equ 000001q
O_READ_WRITE equ 000002q
S_USER_READ equ 00400q
S_USER_WRITE equ 00,200q
newline eq 0xa
bufferLength equ 64; sized buffer
fileName db 'hacked.txt', 0

FD dq 0 ; file descriptor
myText1 "I Hello everyone", NL, 0 ; file text to write to file
    Len $-myText1-1
    2
    Len
```

conditional block
if CREATE flag is non-zero

```
;error message for file I/O
Errorr_create db "- - - -", 10,0
    -close
    -write
    -open
    -read
    -append
    -Delete
    -Print
    -Position
success_close db "Succesfully
ok_create db "file created and opened
    ;OK", newline, 0
ok-close
ok-write
ok-open      R/W", n-
ok-Append
  -Delete
  -Read
  -Position
section .bss
  buffer resb bufferLength
section .text
  global main
main:
;IF CREATE
  mov rdi, fileName
  Call createFile
  mov qword[FD], rax
  ;write to file
mov rdi, qword[FD]
mov rsi, message1
  rdx, ____ Len1
  call writeFile
  ;closefile
  mov rdi, qword[FD]
  call closeFile
;ENDIF; if block ends
;IF OVERWRITE
  ;open file
  mov rdi, filename
  call openFile
  mov qword[FD], rax
  ;overwrite
  mov rdi, qword[FD]
  mov rsi, msg2
  mov rdx, msg2len
  call writeFile
  mov rdi, qword[FD]
  call closeFile
;ENDIF
;IF APPEN
  mov rdi, filename
  Call appendFile
  mov rd qword[FD], rax
mov rdi, qword[FD]
  mov rsi, msg3
  mov rdx, [msg3len]qword
  call writeFile
```

```
;closefile
%ENDIF

%IFO_WRITE
;openandoverwriteat
;anoffset
    mov rdi, filename
    call openFile
    mov qword[FD], rax
    ;positionoffset
    mov rdi, qword[FD]
    mov rsi, qword[msglen2]
    mov rdx, 0 ;indicates
              ;seekingfrom
              ;beginning
    call positionFile
    ;writetooffset
    ;-----
    ;closefile
    %ENDIF
    %IFREAD
    ;openfile
    ;read
    mov rdi qword[FD]
    mov rsi, buffer
    mov rdx, bufferlen
    call readfile
    ;closefile
    %ENDIF
%IF O_READ
    ;offsetread
    ;openfile
    ;positionatoffset
    mov rdi, qword[FD]
    mov rsi, qword[msglen2]
    mov rdx, 0
    call positionFile
    ;read fromfile
    mov rdi, qword[FD]
    mov rsi, buffer
    mov rdx, 0 ; bufferlength
    mov rdi, rax
    call readfile
    mov rdi, rax
    call printstring
    ;closefile
    %ENDIF
%IF DELETE
    mov rdi, filename
    call deleteFile
    %ENDIF
    leave
    ret
global readfile
readFile:
    mov rax, NR_READ
    syscall
    cmp rax, 0
    jl readerror
    mov byte[rsi+rax], 0
    mov rax, rsi
    mov rsi, OK_read ;pushnow
    call printstring ;error
readerror:
    mov rsi, error_read
```

```
call printstring
ret

global deletefile

global appendfile
appendfile:
    mov rax, NR_OPEN
    mov rsi, O_READ|O_WRITE|O_APPEND ;modeofopening
    call syscall
    cmp rax, 0
    jl appenderror
    mov rdi, OKerror
    push rax
    call printstring
    pop rax

(|) = Pipe operator to combine
    file opening flags.
global openFile
openFile:
    ;-----
    mov rsi, O_READ_WRITE ;open in read write mode
global writefile
writeFile:
    mov rax, NR_WRITE
    syscall
    cmp rax, 0

global positionfile:
positionfile:
    mov rax, NR_LSEEK
```

```
Syscalls are always stored in rax.
    syscall

    ;closfilefunction

    ;creatfilefunction
global createfile
createfile:
    mov ra: ---
    mov rsi, S_USER_READ
      | S_USER_WRITE; these
    ;are file permission mean only
    ;user can read write init, this is not
    ;an access mode

global printString
printstring:
    mov r12, rdi ;moving charcode
    mov rdx, 0   ;string counting chars
strloop:
    cmp byte[r12], 0
    je strdone
    inc rdx
    inc r12
    jmp strloop
strdone:
    cmp rdx, 0
    je exit
```

mov rsi, rdi
mov rax, 1 ; for correct operation (printing)
mov rdi, 1 ; 1 = file descriptor of stdout
syscall

exit
ret

C++ :- conditional statements

Types in C++ :- Builtin | User defined | Derived

- Builtin: → integer, → char, → Boolean, → float, → Double, → void, → wide character
- User defined: → struct, → enum, → union
- Derived: · Array, · function, · Pointer

Prefix
binary :- 0b
octal :- 0 (zero)
hex :- 0x
dec :- default

Eg :-
```
int main() {
  unsigned short a = 0b ---;
  printf("%hu\n", a);
  int b = 0123;
  printf("%d ---
  unsigned long long d = 0x FF ---;   (64 bit)
  printf("%llu\n", d);
  return 0;
}
```

Floating point numbers :- Typo precision :- the amount of memory floating point number takes to store the floating point number.
Eg :- float :- 4 byte of precision

long double c = 0.3L ; L = for long double
float a = 0.1F;
double planck_const = 6.6260e-34;

---

format specifier:
%lf, %le or %lg :- double with decimal digit
Long double with dec. digits :- %Lf, %Le, %Lg

generally we use %g or %Lg

%le :- scientific notation
%lf :- decimal representation
%lg :- same scientific notation for defined (same as %e)

→ format specifier for float with decimal :- %e %f %g

Char :- type – char – ASCII ; signed char (similar to char but guaranteed to be signed)
(u) (2 byte charset) char16_t :- UTF-16
(U) (4 ——) char32_t :- UTF-32
unsigned char :-
(L) wchar_t :- UNICODE (wide character)

char16_t 'Wu'

Narrow chars :- char, signed char, unsigned char
Wide chars :- char16_t, char32_t, wchar_t

Escape char :-
\a :- alert
\0 :- Null char
'\u9291', '\U0041'

Cut value of character
Eg :-
```
char x = 'M';
wchar_t y = L'Z';
printf("Window binary start with %c%lc", x, y);
```

enum types :-
```
enum cars {clubs, diamonds, hears, spades};
enum suits : char {myClubs, myDiamonds, myHears, mySpades};
cars card1 = clubs;
cars card2 = cars::diamonds;
```
(use when enum class cars)

boolean:- bool b=true;
bool c=false;

7>7 ⇒ 0 (false)

voidtype:-
void print message (void){
printf("No i/p no o/p");}

Default value in c++:- integer ⇒ 0
char ⇒ \0
bool ⇒ false
pointer ⇒ null

switch:- switch(condition){
case(expression1): {
} break;
case(expression 2): {
} break;
default: {
}}

Eg: with enum class
enum class myCarsModels{BMW,
ferrari};
int main(){ myCarsModels myCar=
myCarsModels::BMW;
switch(myCars){
case(myCarsModels::BMW):{
printf("I have a BMW");} break;
default: {cout<<"I have no car";}}

for loop:-
goto:- for(int i=0; i<10; i++)
{if(i==5)
goto end;
cout<<i<<endl;}
end:
cout<<"end"<<endl;

To include all the headers in c++ :- #include ...

Eg:- pointer ⇒ struct task{
task * pNext;
string description;};
int main(){task item;
item.description="dosomething";
task* pTask = new Task;
//object

removing memory leak in linked list:-

⇒ bool removeHead()
{if(nullptr==pHead) delete pTask;
return false; pHead, pTask->description=:-
task* pTask = pHead; delete pTask;
pHead = pHead->next;
return (pHead!=nullptr);}

Patterns ⇒
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

for(int i=1; i<=size; i++){
for(int j=1; size; j++){
printf("%d", j);
}
printf("\n");}

startTriangle ⇒
*
**
***

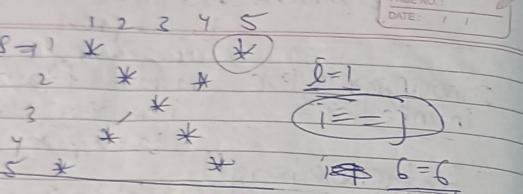
startTriangleInverted ⇒
***
**
*
for(int i=1; i<=size; i++){
for(j=size; j>=1; j--)
{printf("*");

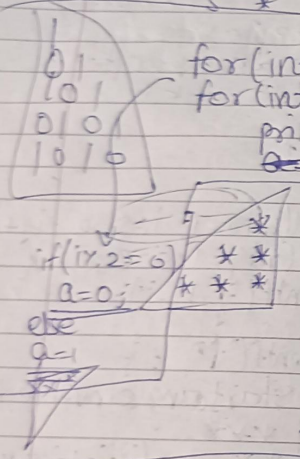
numberTriangle
1
1 2
1 2 3

oddNumberTriangle:-
for(int i=0; i<=size; i++){
for(int j=1; j<=size; j++){

Alphabet pattern ⇒
ABCD
ABCD
ABCD
ABCD

starcase →






```
for(int i=0; i<size; i++){
for(int j=0; j<size; j++){
printf("%d", a);
if(j%2==0) a=0;
```



```
for(int i=1; i<=size; i++){
for(int j=size-i; j>=1; j--)
{printf(" ");
}
printf("\n");}
for(int j=1; i<=size; i++)
{ for(int j=size; j<=i; j++)
```

Constexpr = function will be constexp if function
1. is single line
2. have literal args

⇒ auto funcname(- -) -> int{}

⇒ noexcept function = { return integer } indicates that the function not throws any exception.

void funct(- - -) noexcept

or

noexcept(true/false)

```
void log_message(const string &msg, bool clear_screen
=false){
if(clear_screen) clear_the_screen();
cout<<msg<<endl;}
```

initializer_list ⇒ int sum(initializer_list<int>value)
{int sum=0; value.size() = returns no. of items in the list.
for(int i: value){ sum+=i;}
int main(){ cout<< sum({5,-4,-3,-2,-1,0}) <<endl;
return 0;} ↑ initializer list (container)

Variable argument lists ⇒ we cstdarg

```
int sum(int first, ...){ //must have atleast
                         one parameter so
int sum=0,               that preprocessor
va_list args;            can work.
va_start(args, first);   // initializes args with
int i=first;             first.
while(i!=-1){            // traverses stack of variable args
sum+=i;
i=va_arg(args, int); //ensure every parameter of
                     int type
}
va_end(args); //restores the stack that manipulated
return sum;}  //by va_start
```

macro

→ When a function is called the compiler creates a stack frame for function pushes the items into the stack. while returning pops the stack frame and gets the called functions address.
This creates problem of memory overhead.
This problem is overcome by inline functions.

→ Shared libraries in windows are in .dll (dynamic linked libraries)

→ To link a .cpp source with (c) language.
→ extern 'c'

dynamic memory allocation to arrays:-

→ int *ptr = new int[5]{1, 2, 3, 4, 5};