Contents

STRING OPERATIONS

String Primitive Instructions(x86)

String primitive instructions in the x86 architecture are specialized instructions designed to efficiently process **blocks (strings) of data** in memory.
They operate on **contiguous memory locations** and are commonly used for:

- Moving data
- Comparing memory blocks
- Scanning memory
- Loading from memory into registers
- Storing register values into memory

These instructions work closely with **index registers**, **segment registers**, and the **Direction Flag (DF)** to automatically update memory addresses after each operation.

Key Characteristics (Applies to All String Instructions)

- Operate on **memory, not high-level strings**
- Automatically increment or decrement pointers based on **DF**
- Often paired with **REP prefixes** (REP, REPE, REPNE) for looping
- Use **implicit registers** (no explicit operands written in instruction)

Common Registers Used

Table 9-1 String Primitive Instructions.

Instruction	Description
MOVSB, MOVSW, MOVSD	Move string data: Copy data from memory addressed by ESI to memory addressed by EDI.
CMPSB, CMPSW, CMPSD	Compare strings: Compare the contents of two memory locations addressed by ESI and EDI.
SCASB, SCASW, SCASD	Scan string: Compare the accumulator (AL, AX, or EAX) to the contents of memory addressed by EDI.
STOSB, STOSW, STOSD	Store string data: Store the accumulator contents into memory addressed by EDI.
LODSB, LODSW, LODSD	Load accumulator from string: Load memory addressed by ESI into the accumulator.

Groups of String Primitive Instructions

1. Move String Data — MOVS

Instructions:

- MOVSB – move byte
- MOVSW – move word
- MOVSD – move doubleword

Purpose

Copies data from a source memory location to a destination memory location.

Behavior

- Source address: DS:ESI
- Destination address: ES:EDI
- After execution:
  - ESI and EDI are automatically updated
  - Incremented if DF = 0
  - Decremented if DF = 1

Typical Use

- Copying arrays or memory buffers
- Used with REP to move blocks

2. Compare Strings — CMPS

Instructions:

- CMPSB
- CMPSW
- CMPSD

Purpose

Compares two memory locations element-by-element.

Behavior

- Compares DS:ESI with ES:EDI
- Internally subtracts: (Destination) – (Source)

- Sets **flags** (ZF, SF, CF, etc.)
- Does **not** store the result

Common Usage

- Used with REPE / REPNE to find mismatches
- Often used for string comparison logic

3. Scan String — SCAS

Instructions:

- SCASB
- SCASW
- SCASD

Purpose

Searches memory for a specific value.

Behavior

- Compares accumulator (AL/AX/EAX) with: ES:EDI
- Updates flags based on comparison
- Automatically updates EDI

Typical Use

- Finding a specific byte/word/dword in memory
- Used with REPNE to scan until a match is found

4. Store String Data — STOS

Instructions:

- STOSB
- STOSW
- STOSD

Purpose

Stores the accumulator value into memory.

Behavior

- Writes:
    - AL/AX/EAX → ES:EDI
- Updates EDI automatically

Common Usage

- Initializing memory blocks
- Clearing buffers (e.g., filling with zeros using REP STOSB)

5. Load Accumulator from String — LODS

Instructions:

- LODSB
- LODSW
- LODSD

Purpose

Loads data from memory into the accumulator.

Behavior

- Loads: DS:ESI → AL/AX/EAX
- Updates ESI automatically

Typical Use

- Sequentially reading data from memory
- Used in parsing routines

Direction Flag (DF) Impact

Use CLD to clear DF (forward processing)
Use STD to set DF (backward processing)

Common Pitfalls

- Forgetting to clear/set **Direction Flag**
- Assuming operands are explicit (they are **implicit**)
- Mixing up DS and ES segments
- Using string instructions without REP when looping is required
- Forgetting that flags are modified (especially with CMPS and SCAS)

Suggested Visualizations (For Study)

- **Memory flow diagram**: DS:ESI ---> ES:EDI
- **Register update flow** showing ESI/EDI movement based on DF
- **REP loop diagram** showing ECX countdown

Short note:

- String primitive instructions process **contiguous memory**
- They rely heavily on **implicit registers**
- Pointer movement depends on **Direction Flag**
- Best used with **REP prefixes** for efficiency
- Fundamental for low-level memory manipulation in x86

Using a Repeat Prefix

A string instruction normally works on **one memory value** (or one pair of values) at a time.

You can use a **repeat prefix** to make the instruction run **many times**.

The instruction repeats **until a specific condition is met**.

The most common repeat prefix is **REP**.

REP makes the instruction repeat **while the ECX register is greater than zero**.

Each time the instruction runs, **ECX is decreased by 1**.

When **ECX becomes zero**, the instruction **stops repeating**.

Example: using **REP** can copy **10 bytes** from the string1 buffer to the string2 buffer.

```
001 cld ; clear Direction flag
002 mov esi, OFFSET string1 ; ESI points to source
003 mov edi, OFFSET string2 ; EDI points to target
004 mov ecx, 10 ; set counter to 10
005 rep movsb ; move 10 bytes
```

String instructions use the **ESI** and **EDI** registers to move through memory.

Whether these registers **increase or decrease** depends on the **Direction Flag (DF)**.

The Direction Flag can be changed **manually** using special instructions.

CLD clears the Direction Flag.

When the Direction Flag is cleared, **ESI and EDI increment** (move forward in memory).

STD sets the Direction Flag.

When the Direction Flag is set, **ESI and EDI decrement** (move backward in memory).

Table 9-2 Direction Flag Usage in String Primitive Instructions.

Value of the Direction Flag	Effect on ESI and EDI	Address Sequence
Clear	Incremented	Low-high
Set	Decremented	High-low

When the **Direction Flag is clear**, **ESI and EDI increment** after each operation.

This means the string operation moves from a **lower memory address to a higher memory address**.

When the **Direction Flag is set**, **ESI and EDI decrement** after each operation.

This means the string operation moves from a **higher memory address to a lower memory address**.

The Direction Flag must be set **correctly before** using a string instruction.

If it is not set correctly, **ESI and EDI may move in the wrong direction**.

To copy a string normally from one buffer to another, the Direction Flag should be **cleared first**.

If the Direction Flag is set, the string would be copied **in reverse order**.

The following example shows how to use the **MOVSB** instruction to copy a string from one buffer to another.

```
009 ; Copy string1 to string2
010 cld ; clear Direction flag
011 mov esi, OFFSET string1 ; ESI points to source
012 mov edi, OFFSET string2 ; EDI points to target
013 mov ecx, 10 ; set counter to 10
014 rep movsb ; move 10 bytes
```

MOVSB, MOVSW, and MOVSD

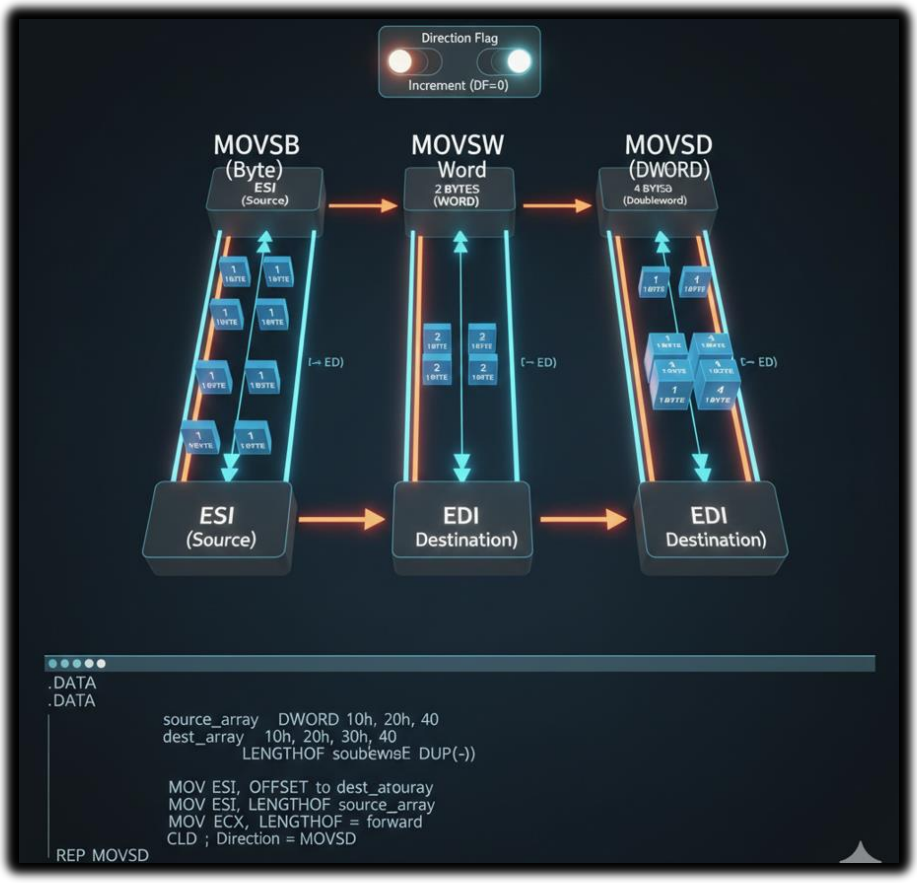
MOVSB, MOVSW, and MOVSD are used to copy data from one memory location to another.

They differ by the size of data they copy.

MOVSB copies bytes.

MOVSW copies words.

MOVSD copies doublewords.



All three instructions use ESI as the source address.

All three instructions use EDI as the destination address.

The Direction Flag controls whether ESI and EDI increment or decrement after each copy.

The following code shows how to use MOVSD to copy a doubleword array from one buffer to another.

```
026 ; Copy source to target
027 cld ; clear Direction flag
028 mov ecx, LENGTHOF source ; set REP counter
029 mov esi, OFFSET source ; ESI points to source
030 mov edi, OFFSET target ; EDI points to target
031 rep movsd ; copy doublewords
```

CLD clears the Direction Flag. When the Direction Flag is clear, ESI and EDI increment after each operation.

REP makes the MOVSD instruction repeat while ECX is greater than zero.

ECX usually holds the number of elements to copy.

MOVSD copies 4 bytes (one doubleword) each time it runs.

After the code finishes, ESI and EDI point 4 bytes past the end of the arrays.

This happens because the registers are incremented after the final copy.

LENGTHOF is a macro that returns the number of elements in an array.

LENGTHOF is often used to set ECX before using REP.

ESI and EDI increment or decrement because string instructions automatically move through memory.

The Direction Flag controls which direction they move in memory.

CMPSB, CMPSW, and CMPSD

CMPSB, CMPSW, and CMPSD are used to compare two memory values.

They differ by the size of data they compare.

CMPSB compares bytes.

CMPSW compares words.

CMPSD compares doublewords.

All three instructions use ESI to point to the first operand.

All three instructions use EDI to point to the second operand.

After each comparison, ESI and EDI move to the next or previous memory location.

The Direction Flag controls whether ESI and EDI increment or decrement.

Example: Comparing Doublewords:

```
035 ; Compare source and target
036 mov esi, OFFSET source
037 mov edi, OFFSET target
038 cmpsd
039 ; compare doublewords
```

CMPSD compares two doubleword values.

If the two values are equal, the Zero Flag (ZF) is set.

If the source doubleword is greater than the target doubleword, the Carry Flag (CF) is set.

If the source is not greater than the target, the Carry Flag is cleared.

To compare multiple doublewords, you can use a repeat prefix with CMPSD.

```
042 mov esi, OFFSET source
043 mov edi, OFFSET target
044 cld ; clear Direction flag
045 mov ecx, LENGTHOF source ; repetition counter
046 repe cmpsd ; repeat while equal
```

REPE (Repeat While Equal) is used with compare instructions like CMPSD.

It repeats the comparison while the values are equal.

REPE stops when ECX becomes zero or when a mismatch is found.

ECX usually contains the number of elements to compare.

LENGTHOF is a macro that returns the number of elements in an array.

LENGTHOF is often used to initialize ECX before using REPE.

The Direction Flag controls whether ESI and EDI move forward or backward in memory.

The Direction Flag must be set correctly before using string compare instructions.

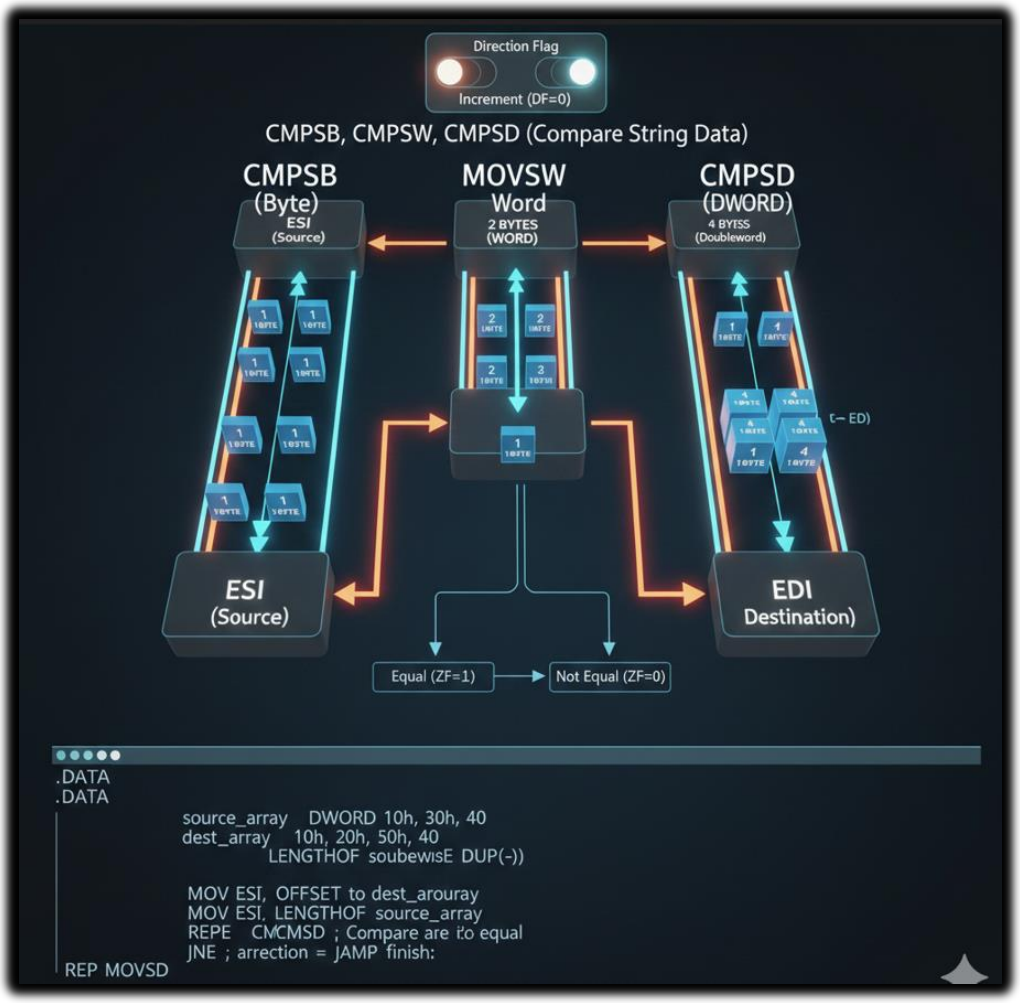
CMPSB, CMPSW, and CMPSD can be used for string-related operations.

Examples include searching for a character or comparing two strings.

The following example shows how to use CMPSB to search for the letter 'A' in a string.

```
050 mov esi, OFFSET string
051 mov edi, OFFSET 'A'
052 repe cmpsb ; repeat while equal
053 jne not_found ; jump if not equal
```

- The string is compared **one byte at a time** using **CMPSB**.
- If the letter **'A' is found**, the **Zero Flag is set**.
- When the Zero Flag is set, the **JNE instruction does not jump**.
- If the letter **'A' is not found**, the **Zero Flag is clear**.
- When the Zero Flag is clear, the **JNE instruction jumps** to the not_found label.
- **CMPSB**, **CMPSW**, and **CMPSD** are powerful instructions for **comparing memory operands**.



SCASB, SCASW, and SCASD

SCASB, SCASW, and SCASD compare a register value to memory:

SCASB: compares AL to a byte at EDI

SCASW: compares AX to a word at EDI

SCASD: compares EAX to a doubleword at EDI

These instructions are useful for searching for a single value in a string or array.

REPE (or REPZ) repeats the scan while ECX > 0 and the register matches memory.

REPNE (or REPNZ) repeats the scan while ECX > 0 and the register does not match memory.

ECX usually holds the number of elements to scan.

Example: SCASB can be used to search for the letter 'F' in the string alpha.

```
059 .data
060     alpha BYTE "ABCDEFGH",0
061 .code
062     mov edi, OFFSET alpha ; EDI points to the string
063     mov al, 'F' ; search for the letter F
064     mov ecx, LENGTHOF alpha ; set the search count
065     cld ; direction = forward
066     repne scasb ; repeat while not equal
067
068     ; Test if the loop stopped because ECX = 0 and the character in AL was not found
069     jnz quit
070
071     ; Found: back up EDI
072     dec edi
073
074     ; Otherwise, the letter F was found
```

REPNE makes the SCASB instruction repeat while the Zero Flag is clear and ECX > 0.

The Zero Flag is cleared when AL does not match the byte at EDI.

If AL matches the byte at EDI, the Zero Flag is set.

After the loop, the JNZ (jump if not zero) instruction is used to check why the loop stopped.

If the loop stopped because ECX = 0 and the character was not found, the Zero Flag is clear, and JNZ jumps to the quit label.

If the character 'F' was found, the DEC EDI instruction is used to move EDI back one position.

This ensures EDI points to the found character instead of the next memory location.

SCASB, SCASW, and SCASD are powerful for searching for a single value in a string or array.

STOSB, STOSW, and STOSD

STOSB, STOSW, and STOSD store a register value into memory at EDI.

STOSB stores AL as a byte.

STOSW stores AX as a word.

STOSD stores EAX as a doubleword.

EDI is incremented or decremented after each store based on the Direction Flag.

When combined with REP, these instructions can fill a string or array with a single value.

Example: STOSB can initialize each byte in the string1 array to 0xFF.

```
; Initialize each byte in string1 to 0xFFh

.data
    Count = 100
    string1 BYTE Count DUP(?)
.code
    mov al, 0xFFh ; value to be stored
    mov edi, OFFSET string1 ; EDI points to target
    mov ecx, Count ; character count
    cld ; direction = forward
    rep stosb ; fill with contents of AL
```

REP makes the STOSB instruction repeat while ECX > 0.

CLD clears the Direction Flag.

When the Direction Flag is clear, EDI increments after each store.

If the Direction Flag is set, EDI decrements, and the array would be filled in reverse order.

After execution, each byte in the string1 array will be set to 0xFF.

STOSB, STOSW, and STOSD are powerful for filling all elements of a string or array with a single value.

```
mov al, 0xFFh
mov edi, OFFSET string1
mov ecx, LENGTHOF string1
cld
rep stosb
```

Move the value 0xFF into the AL register.

Move the value of EDI into ECX (to set the number of bytes to fill).

Clear the Direction Flag with CLD.

Repeat the STOSB instruction while ECX > 0.

After execution, each byte in the string1 array is set to 0xFF.

LODSB, LODSW, and LODSD

LODSB, LODSW, and LODSD load data from memory at ESI into a register:

LODSB → AL (byte)

LODSW → AX (word)

LODSD → EAX (doubleword)

ESI is incremented or decremented based on the Direction Flag.

REP is rarely used with LODS because each load overwrites the previous value in the accumulator.

LODS is typically used to load a single value from memory.

Example: LODSB can load a single byte from memory into AL.

```
101 ; Load a single byte from memory into the AL register
102 lod sb
```

LODSB/LODSW/LODSD load a value from memory at ESI into the accumulator (AL/AX/EAX).

After loading, ESI is incremented or decremented based on the Direction Flag.

Example: LODSB loads a byte from memory into AL, then ESI is incremented by 1.

Array Multiplication Example:

- LODSD loads a doubleword from an array into EAX.
- Multiply the value in EAX by a constant.
- STOSD stores the result back into memory at EDI.
- Repeat for each element of the array.

```
; Array Multiplication Example
.data
    array DWORD 1,2,3,4,5,6,7,8,9,10
    ; test data
    multiplier DWORD 10
    ; test data
.code
    main PROC
    cld
    ; direction = forward
    mov
    esi,OFFSET array
    ; source index
    mov
    edi,esi
    ; destination index
    mov
    ecx,LENGTHOF array
    ; loop counter
    L1:
    lodsd
    ; load [ESI] into EAX
    mul
    multiplier  ; multiply by a value
    stosd       ; store EAX into [EDI]
    loop
    L1
    exit
    main ENDP
    END main
```

CLD clears the **Direction Flag**.

This ensures **ESI and EDI increment** after each operation.

ESI is set to the **offset of the source array**.

EDI is set to the **offset of the destination array**.

ECX is set to the **length of the array** and used as a **loop counter**.

The loop repeats **until ECX = 0**.

Each iteration of the loop:

- **LODSD** loads a doubleword from the source array (**ESI**) into **EAX**.
- **EAX** is multiplied by the value of the **multiplier variable**.
- **STOSD** stores the result into the destination array (**EDI**).

Clearing the Direction Flag ensures that **LODSD and STOSD access the next element** on each iteration.

Using **ECX as a loop counter** ensures the loop repeats **exactly as many times as the array length**.

LODSB, LODSW, and LODSD are powerful for **loading memory data into the accumulator**.

STRING PROCEDURES IN IRVINE32

The **Irvine32 library** provides procedures for working with **null-terminated strings**.

These procedures are similar to **C standard library functions**.

Str_copy

Str_copy copies a **source string** to a **target string**.

It takes **two arguments**:

1. Pointer to the **source string**
2. Pointer to the **target string**

Returns a **pointer to the target string**.

```
140 ; Copy a source string to a target string.
141 Str_copy PROTO,
142 source:PTR BYTE,
143 target:PTR BYTE
```

The following code shows how to use the **Str_copy procedure** to copy the string "Hello, world!" to the string "buffer":

```
145 mov eax, OFFSET "Hello, world!"
146 mov ebx, OFFSET buffer
147 call Str_copy
148 ; buffer now contains the string "Hello, world!"
```

Str_length

Str_length returns the **length of a string** (does **not** count the null byte).

It takes **one argument**: a **pointer to the string**.

The length is returned in the **EAX register**.

```
152 ; Return the length of a string (excluding the null byte) in EAX.
153 Str_length PROTO,
154 pString:PTR BYTE
```

The following code shows how to use the Str_length procedure to get the length of the string "Hello, world!" and store it in the EAX register:

```
157 mov eax, OFFSET "Hello, world!"
158 call Str_length
159 ; EAX now contains the value 12, which is the length of the string "Hello, world!"
```

Str_compare

Str_compare compares **two strings**.

It sets the **Zero Flag (ZF)** and **Carry Flag (CF)** like the **CMP instruction**.

Takes **two arguments**:

1. Pointer to the **first string**
2. Pointer to the **second string**

```
162 ; Compare string1 to string2. Set the Zero and
163 ; Carry flags in the same way as the CMP instruction.
164 Str_compare PROTO,
165 string1:PTR BYTE,
166 string2:PTR BYTE
```

The following code shows how to use the Str_compare procedure to compare the strings "Hello, world!" and "Hello, world!":

```
170 mov eax, OFFSET "Hello, world!"
171 mov ebx, OFFSET "Hello, world!"
172 call Str_compare
173 ; The Zero flag will be set, indicating that the two strings are equal.
```

Str_trim

Str_trim removes a **trailing character** from a string.

Takes **two arguments**:

1. Pointer to the **string**
2. The **character to trim**

```
180 ; Trim a given trailing character from a string.
181 ; The second argument is the character to trim.
182 Str_trim PROTO,
183 pString:PTR BYTE,
184 char:BYTE
```

The following code shows how to use the Str_trim procedure to trim the trailing newline character from the string "Hello, world!\n":

```
185 mov eax, OFFSET "Hello, world!\n"
186 mov ebx, AL ; '\n' character
187 call Str_trim
188 ; The string "Hello, world!" is now stored in the memory location pointed to by EAX.
```

Str_ucase

- **Str_ucase** converts a string to **upper case**.
- Takes **one argument**: a **pointer to the string**.

```
190 ; Convert a string to upper case.
191 Str_ucase PROTO,
192 pString:PTR BYTE
```

The following code shows how to use the Str_ucase procedure to convert the string "hello, world!" to upper case:

```
195 mov eax, OFFSET "hello, world!"
196 call Str_ucase
197 ; The string "HELLO, WORLD!" is now stored in the memory location pointed to by EAX.
```

STR_COMPARE PROCEDURE

Str_compare compares **two strings** byte by byte.

It sets the **Carry Flag (CF)** and **Zero Flag (ZF)** like the **CMP instruction**.

Takes **two arguments**: pointers to the **first string** and the **second string**.

Comparison process:

- Compare each byte of the two strings in order.
- If a byte in the two strings is **not equal**, **Carry Flag is set** and the comparison **stops**.
- If the **end of both strings** is reached without differences, **Zero Flag is set** and the comparison **stops**.

```
220 Str_compare PROC USES eax edx esi edi,
221     string1:PTR BYTE,
222     string2:PTR BYTE
223     ; Initialize esi and edi to point to the beginning of the strings.
224     mov esi, string1
225     mov edi, string2
226     ; Start of the loop to compare characters in the strings.
227 L1:
228     ; Load the characters from string1 and string2 into AL and DL.
229     mov al, [esi]
230     mov dl, [edi]
231     ; Check if we have reached the end of string1 (null terminator).
232     cmp al, 0
233     jne L2 ; If not, continue comparing.
234     ; If we have reached the end of string1, check if we have also reached the end of string2.
235     cmp dl, 0
236     jne L2 ; If not, continue comparing.
237     ; If we reach this point, both strings are equal, and we exit with ZF = 1.
238     jmp L3
239 L2:
240     ; Increment esi and edi to move to the next characters in the strings.
241     inc esi
242     inc edi
243     ; Compare the characters in AL and DL. If they are equal, continue the loop.
244     cmp al, dl
245     je L1 ; Characters are equal; continue comparing.
246
247     ; If characters are not equal, exit the loop with flags set.
248 L3:
249     ret
250 Str_compare ENDP
```

- **Registers saved:** Push **EAX, EDX, ESI, and EDI** onto the stack at the start because the procedure uses them.
- **Pointers loaded:** Move the pointer to the **first string** into **ESI** and the pointer to the **second string** into **EDI**.
- **Loop begins:** Repeat the following steps for each byte:
  - Compare the **bytes at ESI and EDI**.
  - If the bytes are **equal**, **increment ESI and EDI** and continue to the next iteration.
  - If the bytes are **not equal**, **set the Carry Flag (CF)** and exit the loop.
- **End-of-string check:**
  - Check if **ESI reached the null terminator** of the first string.
  - If so, check if **EDI also reached the null terminator** of the second string.
  - If both ends are reached, **set the Zero Flag (ZF)** and exit the loop.

- **Next byte processing:**
  - Increment **ESI and EDI** to point to the next byte.
  - Compare bytes at the new positions.
  - If equal, continue; if not, **set CF** and exit.
- **Procedure exit:** After the loop finishes, return to the caller.

This version organizes the logic **step by step**, making it much easier to understand how **Str_compare** works.

The following is a table of the flags affected by the Str_compare procedure:

RELATION	CARRY FLAG (CF)	ZERO FLAG (ZF)	BRANCH IF TRUE
string1 < string2	1	0	JB (Jump Below)
string1 = string2	0	1	JE (Jump Equal)
string1 > string2	0	0	JA (Jump Above)

- **Str_compare** compares **two strings character by character**.
- **ESI** points to the first string; **EDI** points to the second string.
- The procedure **loops through the strings**, comparing each character.
- The loop **stops** when:
  - A **mismatch** is found, or
  - The **end of either string** is reached.
- **Zero Flag (ZF):**
  - Set to **1** if the strings are **equal**.
  - Set to **0** if the strings are **not equal**.

- **Carry Flag (CF):**
  - Indicates the **relation between the strings**:
    - **JB (CF=1):** string1 < string2
    - **JA (CF=0, ZF=0):** string1 > string2
    - **JE (ZF=1):** string1 = string2

This complements the earlier step-by-step procedure explanation by clarifying **how ZF and CF reflect the comparison result**.

STR_LENGTH PROCEDURE

Str_length returns the **length of a string** in **EAX** (does not count the null terminator).

Takes **one argument**: a **pointer to the string**.

How it works:

- Scans the string **byte by byte** starting from the pointer.
- **EAX is incremented** for each byte until the **null terminator** (0) is reached.
- When the null terminator is found, **EAX contains the string length**.

Usage: Pass the **address of the string** as the argument to the procedure.

```
255 ;-----------------------------------------------------------
256 ; Str_length Procedure
257 ; Calculates the length of a null-terminated string.
258 ; Returns the length of the string in EAX.
259 ;-----------------------------------------------------------
260 Str_length PROC USES edi,
261     pString:PTR BYTE    ; pointer to the string
262
263     mov edi, pString    ; Initialize edi with the pointer to the string.
264     mov eax, 0          ; Initialize eax to 0, which will store the character count.
265
266 L1:
267     cmp BYTE PTR [edi], 0   ; Check if the current character is the null terminator (end of string).
268     je L2                   ; If it's the end of the string, exit the loop.
269
270     inc edi                 ; Move to the next character in the string.
271     inc eax                 ; Increment the character count by 1.
272
273     jmp L1                  ; Repeat the loop to process the next character.
274
275 L2:
276     ret                     ; Return with the length of the string in EAX.
277
278 Str_length ENDP
```

- **Registers saved:** Push **EDI** onto the stack because it will be used.
- **Pointer loaded:** Move the **string pointer** into **EDI**.
- **Length initialized:** Set **EAX = 0** to store the string length.
- **Loop:** Repeat the following for each byte:
  - Compare the **byte at [EDI]** with the **null terminator (0x00)**.
  - If it **equals 0x00**, **exit the loop**.
  - If it **does not equal 0x00**, increment both **EDI** and **EAX**.
- **Return:** After reaching the null terminator, return to the caller.
- **Result: EAX** contains the **length of the string**.

This version clearly shows **how the procedure calculates string length** step by step.

The following is an example of how to use the Str_length procedure:

```
; Get the length of the string "Hello, world!"
mov eax, OFFSET "Hello, world!"
call Str_length

; The EAX register will now contain the value 12, which is the length of the string "Hello, world!"
```

STR_COPY PROCEDURE

Purpose: Copies a **null-terminated string** from a **source** to a **target** location.

Arguments:

1. Pointer to the **source string**
2. Pointer to the **target string**

How it works:

- Calls **Str_length** to get the **length of the source string**.
- Uses **REP MOVSB** to copy the string **byte by byte** from **source** to **target**.

Registers used: Typically **ESI** points to source, **EDI** points to target, **ECX** stores string length, **AL** is not changed.

Result: The **target string** contains a copy of the **source string**.

```
;-----------------------------------------------------
; Str_copy Procedure
; Copies a string from the source to the target.
; Requires: the target string must contain enough space
; to hold a copy of the source string.
;-----------------------------------------------------
Str_copy PROC USES eax ecx esi edi,
    source:PTR BYTE,    ; source string
    target:PTR BYTE     ; target string

    INVOKE Str_length, source   ; Calculate the length of the source string and store it in EAX.
    mov ecx, eax                ; Copy the length into ECX for REP count.
    inc ecx                     ; Add 1 for the null byte at the end of the string.

    mov esi, source             ; Initialize esi with the source pointer.
    mov edi, target             ; Initialize edi with the target pointer.

    cld                         ; Set the direction flag to forward.

    rep movsb                   ; Use REP to copy the string byte by byte from source to target.

    ret                         ; Return when the string is copied.

Str_copy ENDP
```

Purpose: Copy a **null-terminated string** from a **source** to a **target**.

Arguments: Pass the **addresses** of the source and target strings.

Procedure flow:

- Call **Str_length** to calculate the **length of the source string**.
- Set up **ESI** to point to the **source string** and **EDI** to point to the **target string**.
- Use **REP MOVSB** to copy each byte from **source to target**.

Completion: Returns after the **entire string** has been copied.

Example usage: Copy "Hello, world!" from one memory location to another.

```
mov eax, OFFSET "Hello, world!"
mov ebx, OFFSET target_string
call Str_copy

; The target_string variable will now contain a copy of the string "Hello, world!"
```

STR_TRIM PROCEDURE

Purpose: Remove all occurrences of a **specified trailing character** from the **end of a null-terminated string**.

Parameters:

- pString – pointer to the **string to trim**
- char – the **ASCII character** to remove from the end

Return: None; modifies the string in-place.

- **Cases handled:**
    1. **Empty string** – do nothing
    2. **String with trailing char(s)** – remove them
    3. **String with only the trailing char** – truncate to empty string
    4. **String without trailing char** – leave unchanged
    5. **Trailing char(s) followed by other chars** – remove only the trailing char(s)
- **Logic / Steps:**
    1. Get **length of the string**.
    2. If **length = 0**, exit (empty string case).
    3. Initialize **loop counter** to string length.
    4. Point to the **last character** of the string.
    5. Loop **backwards** through the string:
        - Check if the **current character matches** the trailing character.
        - If yes, **decrement counter** and move back.
        - If no, **insert a null byte (\0)** after this character and exit loop.
    6. **Inserting the null byte truncates the string**, making any characters after it insignificant.

```
360 Str_trim PROC USES eax ecx edi,
361             pString:PTR BYTE, ; string pointer
362             char: BYTE        ; trailing char to trim
363
364
365     mov  edi,pString       ; point EDI to string
366     INVOKE Str_length,edi  ; get length in EAX
367
368     cmp eax,0              ; is length 0?
369     je L3                  ; yes, exit
370
371     mov ecx,eax            ; ECX = length
372     dec eax                ; EAX = length - 1
373     add edi,eax            ; point to last char
374
375 L1:
376     mov al,[edi]           ; load character
377     cmp al,char            ; compare to trailing char
378     jne L2                 ; no match, insert null
379     dec edi                ; match, keep backing up
380     loop L1
381
382 L2:
383     mov BYTE PTR [edi+1],0 ; insert null byte
384
385 L3:
386     ret
387 Str_trim ENDP
```

Registers saved: Push **EAX, ECX, and EDI** onto the stack.

Pointer loaded: Move the pointer to the string to be trimmed into **EDI**.

Get string length: Call **Str_length**; result is in **EAX**.

Empty string check:

- Compare **EAX** to 0.
- If 0, **string is empty** → exit procedure.

Loop setup:

- Move string length (**EAX**) to **ECX** → used as loop counter.
- Decrement **EAX** → points to the **last character**.
- Add **EAX to EDI** → EDI now points to **last character** of the string.

Loop (backwards through string):

- Move byte at **[EDI]** into **AL**.
- Compare **AL** to the character to trim.
- If equal → **decrement EDI** and continue loop.
- If not equal → **exit loop**.

Truncate string: Insert **null byte (\0) at [EDI+1]** to terminate the string.

Cleanup:

- Pop **EAX, ECX, EDI** from the stack.
- Return to caller.

Here is an example of how to use the Str_trim procedure:

```
; Trim all trailing spaces from the string "Hello, world!   "
mov eax, OFFSET "Hello, world!   "
call Str_trim
; The EAX register will now contain a pointer to the string "Hello, world!"
```

STR_UCASE PROCEDURE

```
;-----------------------------------------------------
; Str_ucase Procedure
; Converts a null-terminated string to uppercase.
; Returns: nothing
;-----------------------------------------------------
Str_ucase PROC USES eax esi,
    pString:PTR BYTE  ; Pointer to the string

    mov esi, pString  ; Initialize esi with the address of the string.
L1:
    mov al, [esi]    ; Load the character from the string.
    cmp al, 0         ; Check if it's the end of the string.
    je L3             ; If yes, exit the loop.
    cmp al, 'a'       ; Compare the character with 'a'.
    jb L2             ; If it's below 'a', go to L2.
    cmp al, 'z'       ; Compare the character with 'z'.
    ja L2             ; If it's above 'z', go to L2.

    and BYTE PTR [esi], 11011111b
    ; Convert the character to uppercase by clearing the 6th bit.
L2:
    inc esi           ; Move to the next character.
    jmp L1            ; Repeat the loop.
L3:
    ret               ; Return when the entire string is converted to uppercase.
Str_ucase ENDP
```

Purpose: Convert a null-terminated string to uppercase in-place.

Parameter: pString – pointer to the string.

Registers used: ESI points to the current character in the string; **AL** holds the current character.

Procedure flow:

1. **Setup:** Load the string pointer into **ESI**.
2. **Loop through the string:**
   - mov al, [esi] → load current character into **AL**.
   - cmp al, 0 → check for **null terminator**; if found, **exit loop**.
   - cmp al, 'a' → if **AL < 'a'**, jump to **L2** (skip conversion).
   - cmp al, 'z' → if **AL > 'z'**, jump to **L2** (skip conversion).
   - and BYTE PTR [esi], 11011111b → convert lowercase to uppercase by **clearing bit 5** of the ASCII code.
   - **L2:** label for characters not in 'a'-'z' (skip conversion).
   - inc esi → move to **next character**.
   - jmp L1 → repeat loop.
3. **End of string:** Label **L3** is reached when null terminator is detected; **exit procedure**.

Result: The original string in memory is modified; all lowercase letters are converted to uppercase. No value is returned.

STRING LIBRARY DEMO PROGRAM

Demonstrate usage of **Irvine32 string procedures**: Str_trim, Str_ucase, Str_compare, Str_length.

Read the assembly file in this folder first.

Data Section:

- string_1 = "abcde////", 0 → has trailing slashes.
- string_2 = "ABCDE", 0
- msg0–msg5 = messages to display results.

Main Program Flow:

1. call trim_string → remove trailing slashes from string_1.
2. call upper_case → convert string_1 to uppercase.
3. call compare_strings → compare string_1 with string_2.
4. call print_length → display length of string_2.
5. exit → end program.

trim_string PROC:

- Calls Str_trim on string_1 with '/' as the character to remove.
- Displays message: "string_1 after trimming: " followed by the trimmed string.

upper_case PROC:

- Displays message: "string_1 in upper case: ".
- Calls Str_ucase to convert string_1 to uppercase.
- Displays the uppercase string.

compare_strings PROC:

- Calls Str_compare on string_1 and string_2.
- Checks flags set by Str_compare:
  - **Zero flag set** → strings are equal → display msg1.
  - **Carry flag set** → string_1 < string_2 → display msg2.
  - Otherwise → string_2 < string_1 → display msg3.

print_length PROC:

- Displays message: "Length of string_2 is ".
- Calls Str_length on string_2.
- Displays the length using WriteDec.

Program Output:

1. After trimming: string_1 after trimming: abcde
2. After uppercase: string_1 in upper case: ABCDE
3. Comparison: one of:
   - "string_1 and string_2 are equal"
   - "string_1 is less than string_2"
   - "string_2 is less than string_1"
4. Length of string_2: e.g., "Length of string_2 is 5"

Key Points:

- Demonstrates **in-place string modification** (Str_trim, Str_ucase).
- Uses **flags from Str_compare** for conditional logic.
- Combines string procedures with **output procedures** (WriteString, WriteDec).
- Serves as a **complete example** of basic string handling in **MASM with Irvine32**.

Strings using Irvine64

```
INCLUDE Irvine64.inc
.data
    source BYTE "AABCDEFGAABCDFG", 0
    ; size = 15
    target BYTE 20 DUP(0)

.code
    Str_compare PROTO
    Str_length PROTO
    Str_copy PROTO
    ExitProcess PROTO

    main PROC
        mov rcx, OFFSET source
        call Str_length
        ; Returns length in RAX
        mov rsi, OFFSET source
        mov rdi, OFFSET target
        call Str_copy
        ; We just copied the string, so they should be equal.
        call Str_compare
        ; ZF = 1, strings are equal
        ; Change the first character of the target string, and
        ; compare them again.
        mov BYTE PTR [rdi], 'B'
        call Str_compare
        ; CF = 1, source < target
        mov ecx, 0
        call ExitProcess
    main ENDP
```

1. Str_compare

Compares two null-terminated strings pointed to by RSI (source) and RDI (target).

- Sets **ZF = 1** if equal
- Sets **CF = 1** if source < target

```
Str_compare PROC
    push rax
    push rbx
    push rcx
    push rdx
compare_loop:
    mov al, [rsi]       ; Load byte from source
    mov bl, [rdi]       ; Load byte from target
    cmp al, bl           ; Compare bytes
    jne not_equal        ; If different, exit loop
    cmp al, 0            ; Check for null terminator
    je strings_equal     ; End of both strings, equal
    inc rsi              ; Move to next byte
    inc rdi
    jmp compare_loop
not_equal:
    ; Set CF if source < target, clear CF otherwise
    jb set_carry
    clc
    jmp done
set_carry:
    stc
strings_equal:
    ; Strings are equal, set ZF
    mov al, 0
    cmp [rsi], al
    je set_zf
    clc
    jmp done
set_zf:
    xor eax, eax
    ; ZF is already set by cmp, do nothing
done:
    pop rdx
    pop rcx
    pop rbx
    pop rax
    ret
Str_compare ENDP
```

2. Str_length

Returns the length of a null-terminated string pointed to by RCX.

- Returns **length in RAX**

```
Str_length PROC
    xor rax, rax        ; Clear RAX (length counter)
length_loop:
    cmp byte ptr [rcx + rax], 0 ; Check for null terminator
    je length_done
    inc rax             ; Increment length counter
    jmp length_loop
length_done:
    ret
Str_length ENDP
```

3. Str_copy

Copies a null-terminated string from RSI (source) to RDI (target).

- Uses **byte-by-byte copying**
- Returns nothing

```
Str_copy PROC
copy_loop:
    mov al, [rsi]       ; Load byte from source
    mov [rdi], al       ; Store byte to target
    inc rsi
    inc rdi
    cmp al, 0           ; Stop at null terminator
    jne copy_loop
    ret
Str_copy ENDP
```

✅ Key Notes:

- All three procedures are **safe for null-terminated ASCII strings**.
- They use standard **RSI, RDI, and RCX/RAX registers** for string operations.
- Str_compare carefully sets **ZF and CF** to match string comparison logic.

Output and Display:

The test program performs string operations but doesn't show the results. You need to add code to display them, like whether strings are equal, the string's length, and comparison results, using functions such as WriteString and WriteDec.

```
mov rsi, OFFSET msg1
call WriteString  ; Display result message
call Crlf
; Check ZF and CF flags to determine equality or comparison result
; Display results accordingly
```

Irvine64 Library Setup

The **Irvine64 library** must be included in your assembly program.

It needs to be **set up properly** in your assembly environment.

At the **beginning of your program**, include instructions to reference the Irvine64 library.

This usually involves **specifying the paths and configurations** for the library.

Example code is provided to show how to include and set it up.

```
INCLUDE Irvine64.inc  ; Include Irvine64 library

.data
; Your data declarations go here

.code
main PROC
; Your program's main code goes here

main ENDP

END main
```

Actual Program:

```
INCLUDE Irvine64.inc

; -----------------------------------------------
; Str_compare
; Compares two strings
; Receives:
; RSI points to the source string
; RDI points to the target string
; Returns:
; Sets ZF if the strings are equal
; Sets CF if source < target
; -----------------------------------------------
Str_compare PROC
    USES rax rdx rsi rdi

L1:
    mov al, [rsi]
    mov dl, [rdi]
    cmp al, 0       ; End of string1?
    jne L2          ; No
    cmp dl, 0       ; Yes: End of string2?
    jne L2          ; No
    jmp L3          ; Yes, exit with ZF = 1

L2:
    inc rsi         ; Point to next
    inc rdi
    cmp al, dl      ; Characters equal?
    je L1           ; Yes, continue loop
                    ; No: Exit with flags set

L3:
    ret

Str_compare ENDP

; -----------------------------------------------
; Str_copy
; Copies a source string to a location indicated by a target pointer
; Receives:
; RSI points to the source string
; RDI points to the location where the copied string will be stored
; Returns: nothing
; -----------------------------------------------
Str_copy PROC
    USES rax rcx rsi rdi

    mov rcx, rsi    ; Get length of the source string
    call Str_length ; Returns length in RAX
    mov rcx, rax    ; Loop counter
    inc rcx         ; Add 1 for the null byte
    cld             ; Direction = up
    rep movsb       ; Copy the string
    ret
```

```
Str_copy ENDP

; ------------------------------------------------
; Str_length
; Gets the length of a string
; Receives: RCX points to the string
; Returns: length of the string in RAX
; ------------------------------------------------
Str_length PROC
    USES rdi

    mov rdi, rcx    ; Get the pointer
    mov eax, 0      ; Character count
L1:
    cmp BYTE PTR [rdi], 0  ; End of string?
    je L2                  ; Yes: quit
    inc rdi                ; No: Point to the next
    inc rax                ; Add 1 to count
    jmp L1
L2:
    ret                   ; Return count in RAX

Str_length ENDP
```

```
.data
   source BYTE "AABCDEFGAABCDFG",0
   target BYTE 20 dup(0)

.code
   main PROC
       mov rcx, offset source
       call Str_length      ; Returns length in RAX
       mov rsi, offset source
       mov rdi, offset target
       call Str_copy
       ; We just copied the string, so they should be equal.
       call Str_compare
       ; ZF = 1, strings are equal
       ; Change the first character of the target string, and compare them again.
       mov target, 'B'
       call Str_compare
       ; CF = 1, source < target
       mov ecx, 0
       call ExitProcess

   main ENDP

END main
```

Irvine64 Library:

- INCLUDE Irvine64.inc statement includes the Irvine64 library.
- Provides access to Irvine's assembly functions and features.

USES Keyword:

- Used in Str_compare and Str_copy procedures.
- Specifies registers that are **pushed onto the stack** at the start and **popped off** at the end.
- Helps maintain **calling conventions**.

Str_compare Procedure:

- Compares two strings pointed to by RSI and RDI.
- Sets the **Zero Flag (ZF)** if the strings are equal.
- Sets the **Carry Flag (CF)** if the source is less than the target.

Str_copy Procedure:

- Copies a source string (RSI) to a target location (RDI).
- Calculates the source string length using Str_length.
- Uses rep movsb to perform the copy.

Str_length Procedure:

- Calculates the length of a **null-terminated string**.
- Receives a pointer in RCX.
- Returns the result in RAX.

.data Section:

- Contains **data declarations** for source and target strings.

.code Section:

- The **main procedure** demonstrates usage of the string procedures:
  - Copying a string
  - Comparing strings
  - Modifying a character for comparison

General Notes:

- Shows **Irvine64 register usage, stack management, and procedure calling conventions**.
- Environment must be **properly configured** to work with Irvine64.

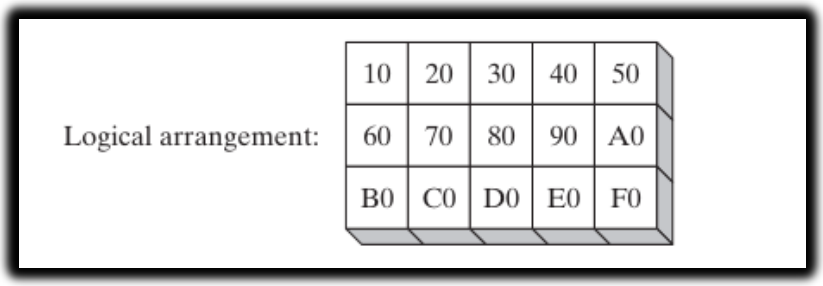
2D ARRAYS

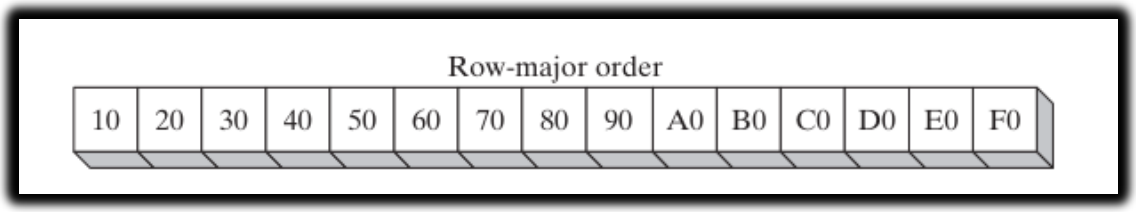
Row-major order and column-major order are just two ways computers can store a 2D array in memory.

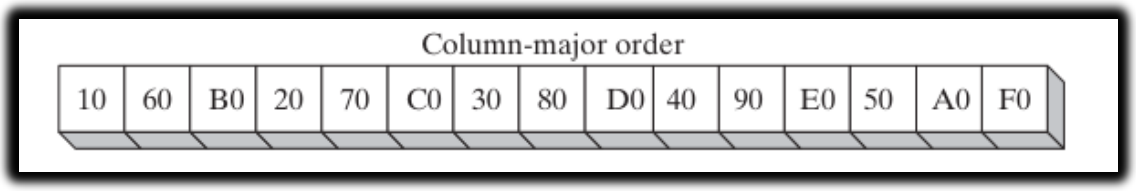
The main difference is the order in which the elements are laid out.

In **row-major order**, the computer stores one row after another.

In **column-major order**, it stores one column after another.







Row-major order

Row-major order is the most common way to store 2D arrays.

Most high-level programming languages use **row-major order**.

In this method, the elements of each **row are stored one after another** in memory.

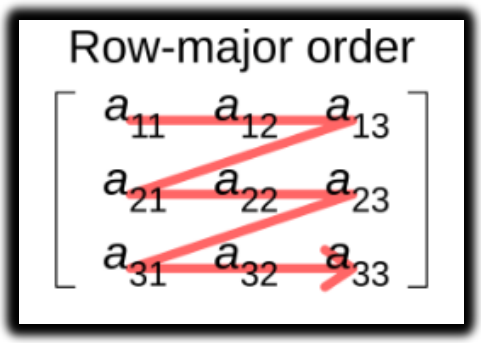
```
Row-major order:
[0][0] [0][1] [0][2]
[1][0] [1][1] [1][2]
[2][0] [2][1] [2][2]
```

The **first element of the first row** is stored at the beginning of the memory block.

Then the **remaining elements of the first row** are stored one after another.

After the last element of the first row, the **elements of the second row** are stored in order.

This process continues **row by row** until the last element of the last row is stored.

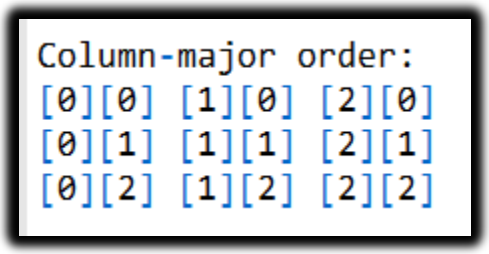


Column-major order

Column-major order is less common than row-major order.

It is used in some applications, like **linear algebra**.

In this method, the elements of each **column are stored one after another** in memory.

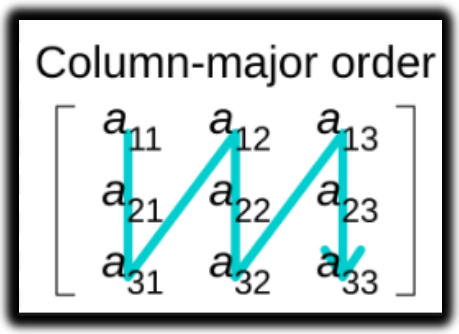


The **first element of the first column** is stored at the beginning of memory.

Then the **remaining elements of the first column** are stored one after another.

After the last element of the first column, the **elements of the second column** are stored in order.

This continues **column by column** until the last element of the last column is stored.



Which order to use?

- The choice of row-major or column-major depends on the **application**.
- **Row-major order** is generally more efficient when accessing elements **by row**.
- **Column-major order** is more efficient when accessing elements **by column**.

How to implement a two-dimensional array in MASM

- You can use **either row-major or column-major order**.
- It is important to be **consistent** with the order you choose throughout your program.

Implementing a 2D array in row-major order

1. **Allocate memory** for the entire array.
2. The **size of the memory block** depends on:
   - The number of elements in the array
   - The **data type** of each element (e.g., byte, word, dword)
3. **Initialize the array elements** as needed.
4. To **access an element** at row i and column j, use a **formula** based on row-major indexing (usually: index = i * number_of_columns + j).

```
element[i][j] = array[i * number_of_columns + j]
```

where i is the row index and j is the column index.

Explanation of 2D Array Implementation in MASM (Row-Major Order)

For example, the following code implements a two-dimensional array of integers in row-major order:

```
655 ; Declare a two-dimensional array of integers.
656 array dw 100 dup(0)
657
658 ; Initialize the elements of the array.
659 mov ebx, 1
660 mov ecx, 100
661 mov edi, array
662 loop:
663 mov dword ptr [edi], ebx
664 inc ebx
665 inc edi
666 loop loop
667
668 ; Access an element of the array.
669 mov eax, array[1 * 10 + 2]
```

- The array is **declared as 100 contiguous DWORDs** initialized to 0.
  - This forms a **10×10 array** (10 × 10 = 100).
  - *Note:* Each DWORD is actually 4 bytes, so just to be precise, 100 DWORDs = 400 bytes.
- **EBX** is used as a **counter** from 1 to 100 to store sequential values in the array.
- **EDI** points to the **start of the array** and is incremented each iteration to move through elements.
- **ECX** counts iterations from 1 to 100 to fill **all elements**.
- **Accessing elements:**
  - Use the formula: offset = row * numCols + col.
  - Example: row 1, column 2 → offset = 10 + 2 = 12.
- The elements are stored in **row-major order**:
  - Row 1 elements first, then row 2, and so on sequentially in memory.
- This shows a **typical way to declare, initialize, and access a 2D array in MASM** using **row-major layout**.

Base-Index Operands

A **base-index operand** lets you access memory using the **sum of two registers**:

Base register + **Index register**

They are **very useful for arrays**:

You can calculate the address of an array element using its **row and column indices**.

How to use a base-index operand:

Load the **base register** with the **address of the array**.

Load the **index register** with the **row and/or column index** of the element you want.

Use the **base + index** to access the desired element in memory.

This approach allows **efficient access** to elements of a **two-dimensional array**.

Example usage:

An example showing how a 2D array element is accessed with [base + index].

```
672 ; Declare a two-dimensional array of integers.
673 array dw 1000h, 2000h, 3000h
674
675 ; Load the base register with the address of the array.
676 mov ebx, OFFSET array
677
678 ; Load the index register with the row and column indices of the element that we want to access.
679 mov esi, 1 ; row index
680 mov edi, 2 ; column index
681
682 ; Calculate the address of the element using the base-index operand.
683 mov eax, [ebx + esi * 2 + edi * 4]
684
685 ; Display the value of the element.
686 mov edx, 0 ; service number
687 mov ecx, 1 ; buffer offset
688 mov al, [eax] ; buffer byte
689 int 21h ; system call
```

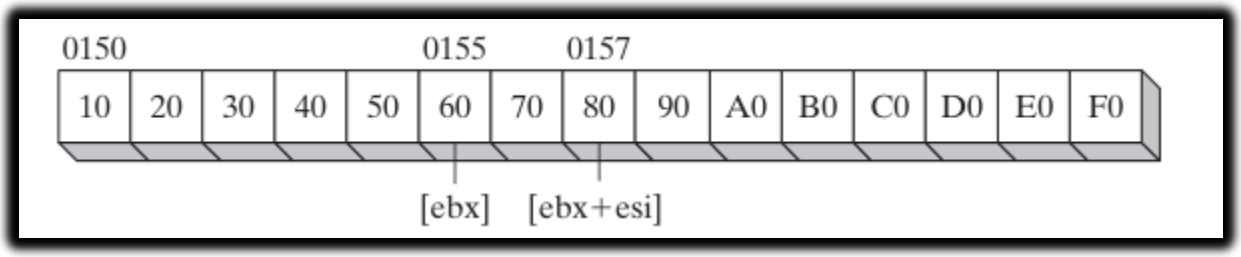
The code example **displays 2000h**, which is the value at **row 1, column 2** of the array.

Column-major vs. Row-major order:

- **Column-major order:** Elements of each **column** are stored consecutively in memory.
- **Row-major order:** Elements of each **row** are stored consecutively in memory.

Most high-level languages use **row-major order** for 2D arrays.

When writing assembly that interacts with high-level languages, it's best to **use row-major order** to stay consistent.



To access an elements of the array above, you can use the following formula:

```
element[row_index][column_index] = array[row_index * number_of_columns + column_index]
```

where row_index is the row index of the element and column_index is the column index of the element.

For example, to access the element at row 1, column 2 of the array, you would use the following code:

```
698 mov ebx, OFFSET array ; load the base register with the address of the array
699 mov esi, 1 ; load the index register with the row index
700 mov al, [ebx + esi * 2] ; access the element at row 1, column 2
```

The al register will now contain the value of the element at row 1, column 2 of the array.

Calculating a Row Sum (calc_row_sum procedure)

Purpose: Calculates the sum of a selected row in a matrix of 8-bit integers.

Inputs:

- EBX → Base offset/address of the matrix in memory
- EAX → Row index to calculate the sum for
- ECX → Size of each row (in bytes)

Output:

- Returns the **sum of the row** in the EAX register

How it works:

- Calculate the **row offset**: row_index × row_size.
- Add the offset to the **base address** to get the address of the **first element of the row**.
- Initialize an **accumulator** to 0.
- Iterate through the row, **adding each element** to the accumulator.
- Stop when the end of the row is reached.
- Return the **sum** in EAX.

Here is a more detailed explanation of the code:

```
;-----------------------------------------------------------
; calc_row_sum
; Calculates the sum of a row in a byte matrix.
; Receives: EBX = table offset, EAX = row index,
; ECX = row size, in bytes.
; Returns: EAX holds the sum.
;-----------------------------------------------------------
calc_row_sum PROC USES ebx ecx edx esi

; Calculate the offset of the row.
mul ecx ; row index * row size

; Add the offset to the base address of the matrix to get the address of the first element in the row.
add ebx,eax ; row offset

; Initialize the accumulator.
mov eax,0 ; accumulator

; Set the column index to 0.
mov esi,0 ; column index

; Loop over the row, adding each element to the accumulator.
L1:
movzx edx,BYTE PTR[ebx + esi] ; get a byte
add eax,edx ; add to accumulator
inc esi ; next byte in row
loop L1

; Return the sum in the accumulator.
ret
calc_row_sum ENDP
```

BYTE PTR in MOVZX

- The BYTE PTR operand size is required in the MOVZX instruction to specify that the source operand is a **byte**.
- MOVZX converts a byte to a **doubleword** (zero-extends the value).
- Example: MOVZX EAX, BYTE PTR [EBX + ESI]
    - Tells the assembler that [EBX + ESI] is a byte in memory.

calc_row_sum procedure

- Demonstrates **base-index addressing** for accessing elements of a **two-dimensional array**.
- Can be used to perform tasks like **calculating the sum of a row** in an array.
- Shows how to combine a **base address** and an **index register** to locate array elements in memory.

Assembly Scale Factors: Navigating Arrays

This section bridges the gap between **how humans think** (index 0, 1, 2…) and **how memory works** (byte 0, 4, 8…). It's the *"Smart Calculator"* of assembly addressing.

I. The Scale Factor Cheat Sheet

When you work with an array in **C++** or **Java**, you just write:

The compiler automatically calculates the memory offset. For example, if it's an array of integers (4 bytes each), index 3 points **12 bytes into memory** (3 × 4).

In **Assembly**, you must do that math manually. **Scale Factors** make this easier.

X86 ADDRESSING SCALE FACTORS

DATA TYPE	SIZE (BYTES)	SCALE FACTOR
BYTE	1	1
WORD	2	2
DWORD	4	4
QWORD	8	8

*Usage: [Base + (Index * Scale)]*

II. The Formula: Base + Index × Scale + Displacement

The most powerful way to access memory in x86 is **Scaled Indexed Addressing**.

Syntax:

```
[BaseAddress + (IndexRegister * ScaleFactor)]
```

- **BaseAddress:** The start of the array (e.g., EBX)
- **IndexRegister:** The element index (e.g., ESI = 2 for the 3rd item)
- **ScaleFactor:** Multiplier based on the element size (1, 2, 4, or 8)

III. The Pro Way: The TYPE Operator 🛠️

Hardcoding numbers like 4 or 8 is risky. If the array type changes (e.g., from DWORD to WORD), you must manually update every multiplication.

Solution: Use the TYPE operator in MASM (Kip Irvine).

Example: Accessing a Word Array

```
.data
tableW WORD 10h, 20h, 30h, 40h    ; An array of WORDs

.code
mov ebx, OFFSET tableW            ; EBX = start of array
mov esi, 2                        ; Index 2 (3rd element)
mov ax, [ebx + esi * TYPE tableW] ; Access element
```

Or

```
745 ; tableW is an array of words.
746 mov ebx, OFFSET tableW ; load the base register with the address of the array
747 mov esi, 2 ; load the index register with the column index
748 mov ax, [ebx + esi * TYPE tableW] ; access the element at row 1, column 2
```

Explanation:

- TYPE tableW evaluates to 2 (size of WORD)
- Effective address calculation:

```
Address + (2 * 2) = Address + 4 bytes
```

- Moves 30h into AX

✅ Using TYPE makes your code **future-proof**.

IV. Why Scale Factors Matter for 2D Arrays

For 2D arrays (rows × columns), scale factors are essential:

```
Address + (RowSize * RowIndex) + (ColumnIndex * ScaleFactor)
```

- Keep ESI or EDI as simple counters (0, 1, 2, 3)
- Hardware handles the actual memory offset

V. Scale Factors in String Operations

- Instructions like MOVSB or STOSD automatically increment pointers by 1, 2, or 4.
- **Scaled Indexing** is useful when manually processing strings or tables of pointers.

Example: Table of string addresses (DWORDs)

- Each address is 4 bytes
- To access the 5th string:

```
[EBX + ESI * 4]
```

Big Ideas to Remember

- **Scale Factor:** Converts a human-friendly index into a memory offset
- **Permitted Scales:** Only 1, 2, 4, and 8 are allowed by CPU hardware
- **TYPE Operator:** Always use it for maintainable code
- **Logic:**

```
Value = [Base + (Index * TYPE)]
```

Base-Index-Displacement Operands

Base-Index-Displacement Operand

- This operand allows you to access memory using the **sum of several components**:
  - **Displacement** (a constant or label, e.g., array name or offset)
  - **Base register** (usually holds the starting address or row offset)
  - **Index register** (usually holds the element or column offset)
  - **Optional scale factor** (multiplies the index register to account for element size)
- Well suited for **two-dimensional arrays**:
  - **Displacement** → base address of the array
  - **Base register** → row offset
  - **Index register** → column offset
- Example: Accessing an element of a **2D array of doublewords**:

```
mov EAX, [ARRAY + EDI*4 + EBX]
```

- ARRAY → displacement (array start)
- EBX → base (row offset)
- EDI*4 → index*scale (column offset, 4 bytes per doubleword)

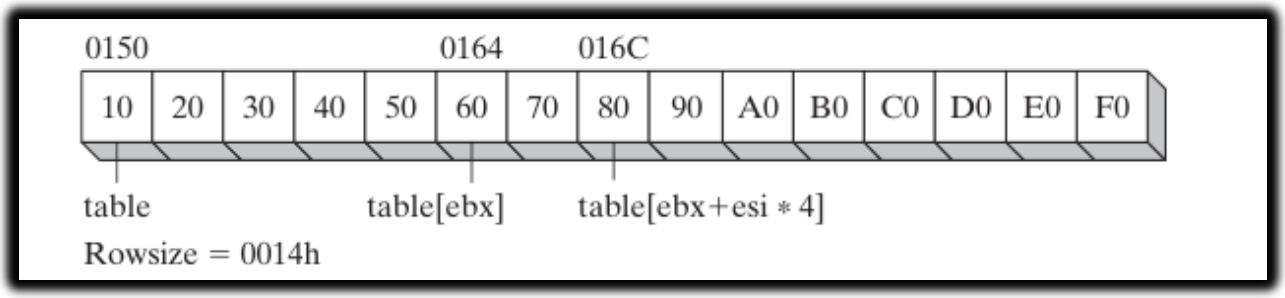
```
759 ; tableD is a two-dimensional array of doublewords.
760 ; Rowsize is the size of each row in the array, in bytes.
761
762 mov ebx, Rowsize ; load the base register with the row offset
763 mov esi, 2 ; load the index register with the column offset
764 mov eax, tableD[ebx + esi * TYPE tableD] ; access the element at row 1, column 2
```

- tableD[ebx + esi * TYPE tableD] points to the element at **row 1, column 2** of the array.
- **EBX** holds the **row offset**.
- **ESI** holds the **column offset**.
- TYPE tableD tells the assembler the elements are **doublewords (4 bytes)**, so the **scale factor is 4**.

Base-index-displacement operands help you write **efficient array access code**.

Using them means you **don't have to manually track element sizes or row offsets**.

This approach makes your code **easier to read and maintain**.



- The diagram shows **EBX** and **ESI** relative to the tableD array.
- **EBX** contains the **row offset**, and **ESI** contains the **column offset**.
- The tableD array starts at **offset 0150h**.
- **EBX = 20h** → this is the **size of one row in bytes**, so EBX points to the **start of the second row**.
- **ESI = 2** → this is the **column index**, so ESI points to the **third element in the second row**.

Base-index-displacement operands are a **powerful way to access arrays efficiently** in assembly language.

Base-Index Operands in 64-Bit Mode

The program demonstrates how to use **base-index-displacement operands** to access a **2D array of 64-bit integers**.

```
0804 ; Two-dimensional arrays in 64-bit mode (TwoDimArrays.asm)
0805 ; Prototypes for procedures
0806 Crlf proto
0807 WriteInt64 proto
0808 ExitProcess proto
0809
0810 .data
0811     table QWORD 1,2,3,4,5      ; Define a two-dimensional array with 3 rows and 5 columns
0812     RowSize = ($ - table)      ; Calculate the size of one row in bytes
0813
0814     QWORD 6,7,8,9,10          ; Define the second row
0815     QWORD 11,12,13,14,15       ; Define the third row
0816
0817 .code
0818 main PROC
0819     ; Set row and column indices
0820     mov rax, 1         ; Row index (zero-based)
0821     mov rsi, 4         ; Column index (zero-based)
0822
0823     call get_tableVal  ; Call the get_tableVal procedure to retrieve the value
0824     call WriteInt64    ; Display the retrieved value
0825     call Crlf          ; Insert a line break
0826
0827     mov ecx, 0
0828     call ExitProcess   ; End the program
0829
0830 main ENDP
```

Main Procedure Steps

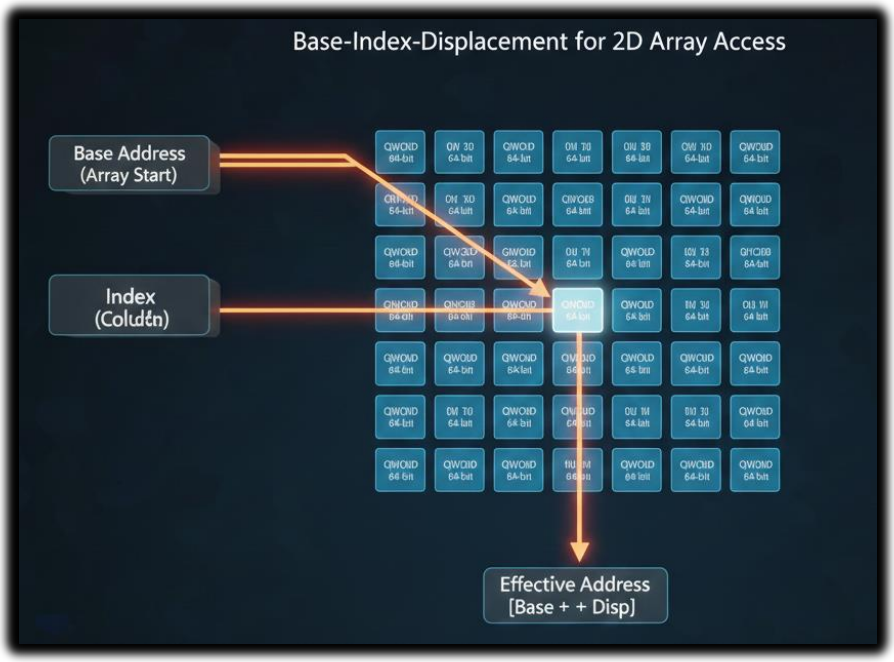
- Loads the **row index (1)** into the **RAX** register.
- Loads the **column index (4)** into the **RSI** register.
- Calls the **get_tableVal** procedure to get the value at that row and column.
- Calls **WriteInt64** to display the value in **RAX**.
- Calls **ExitProcess** to end the program.

get_tableVal Procedure Steps

- Loads the **row offset** into **RBX**.
- Multiplies the row offset by the **size of a quadword** (8 bytes) to get the **row's byte offset**.
- Adds the **column offset** to the row offset to get the **element's byte offset**.
- Loads the **value at that offset** into **RAX**.
- Returns from the procedure.

Base-Index-Displacement Usage

- **Base operand:** RBX (row offset)
- **Index operand:** RSI (column offset)
- **Scale factor:** omitted (quadwords = 8 bytes, but in 64-bit mode the hardware treats this as 1)



Key Takeaways

- The program shows how **base-index-displacement operands** work in **64-bit assembly**.
- Using base-index-displacement operands makes **array access simple, efficient, and readable**.

SEARCHING AND SORTING ALGORITHMS

Bubble Sort – Key Points

- **Bubble Sort** is a simple sorting algorithm for arrays.
- It repeatedly **compares adjacent elements** and **swaps them if they are out of order**.
- The process starts at the **beginning of the array** and works toward the end.
- If the **first element > second element**, swap them.
- Then move to the next pair and repeat the comparison.
- The algorithm **iterates through the array multiple times** until no swaps are needed.
- After each full pass, the **largest unsorted element "bubbles" to its correct position**.

```
bubble_sort(array):
  for i in range(len(array) - 1):
    for j in range(len(array) - i - 1):
      if array[j] > array[j + 1]:
        array[j], array[j + 1] = array[j + 1], array[j]
```

Or

```
procedure bubbleSort(array)
    n = length(array)
    for i = 0 to n-1
        for j = 0 to n-i-2
            if array[j] > array[j+1]
                swap(array[j], array[j+1])
```

- Outer loop ensures we make enough passes to sort the entire array.
- Inner loop compares adjacent elements.
- Each pass moves the next largest element to its correct position.
- It is not very efficient for large arrays, because it has to compare every pair of elements in the array for each iteration.
- For an array of size n, the bubble sort algorithm has a **time complexity of $O(n^2)$.**

Analysis of bubble sort performance.

The following table shows the sort times for various array sizes, assuming that 1000 array elements can be sorted in 0.1 second:

Array Size	Time (seconds)
1,000	0.1
10,000	10.0
100,000	1000
1,000,000	100,000 (27.78 hours)

As you can see, the sort time increases quadratically with the array size.

Sort times grow quadratically: if 1,000 elements take 0.1 seconds, doubling the array size roughly **quadruples the time**.

This means that the bubble sort algorithm is not very efficient for large arrays.

Recommendation: For large arrays, use **more efficient algorithms** like:

- **Quicksort**
- **Merge Sort**

```
0847 ; Bubble sort algorithm in MASM
0848 section .data
0849     array: dw 5, 3, 2, 1, 4
0850 section .code
0851     start:
0852
0853     mov esi, array
0854     mov ecx, 5 ; length of the array
0855     L1:
0856     mov edi, esi
0857     add edi, 4
0858     L2:
0859     cmp [esi], [edi]
0860     jg L3 ; swap if esi > edi
0861     xchg [esi], [edi]
0862     add esi, 4
0863     cmp esi, array + ecx * 4 - 4
0864     jne L2
0865     loop L1
0866     ; array is now sorted
0867     exit
```

How the Code Works

Array Initialization: The array is defined in the **data segment** with the values to be sorted.

Register Setup:

- ESI points to the **start of the array**.
- ECX holds the **length of the array** (e.g., 5 elements).

Outer Loop (L1):

- Iterates through the array multiple times.
- Corresponds to the **outer loop counter (cx1)**.

Inner Loop (L2):

- Compares **adjacent elements** and performs **swaps** if needed.
- Corresponds to the **inner loop counter (cx2)**.

Comparison and Swap:

- CMP checks if the **current element ([ESI])** is greater than the **next element ([EDI])**.
- If true, XCHG swaps the two elements.

Loop Control:

- The inner loop runs until ESI reaches the **end of the array** (array + ECX * 4 - 4).
- The outer loop uses LOOP to **decrement the counter** and repeats until all passes are done.

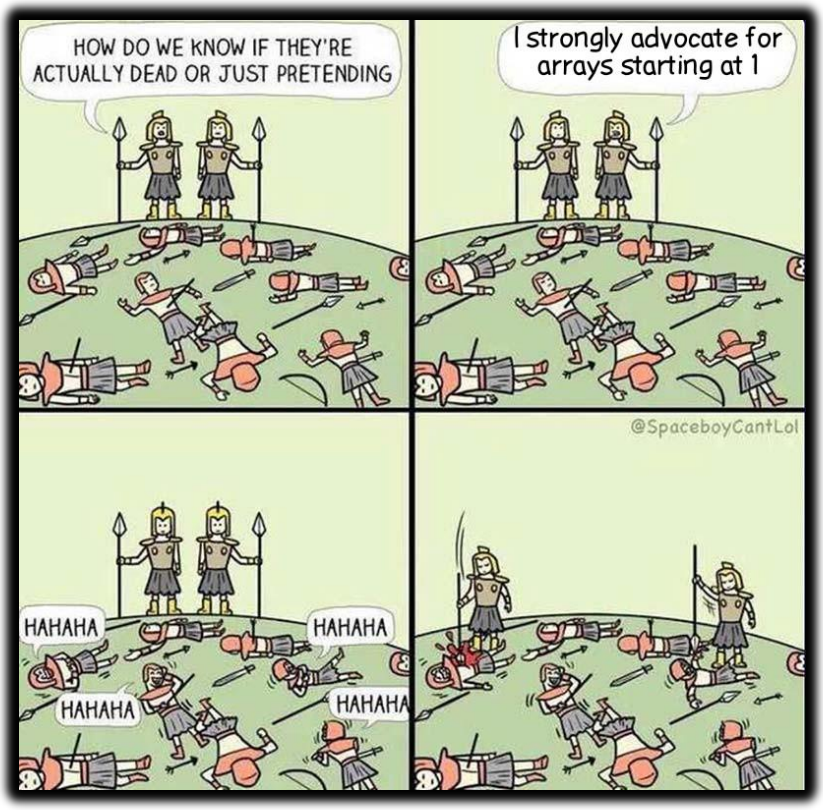
Result:

- After all iterations, the **array is sorted in ascending order**.

C++ version:

```
0870 int BinSearch(int values[], const int searchVal, int count) {
0871     int first = 0;
0872     int last = count - 1;
0873
0874     while (first <= last) {
0875         int mid = (last + first) / 2;
0876
0877         if (values[mid] < searchVal)
0878             first = mid + 1;
0879         else if (values[mid] > searchVal)
0880             last = mid - 1;
0881         else
0882             return mid;  // success
0883     }
0884
0885     return -1;  // not found
0886 }
```

Final program: Read the file **BinarySortTest.asm**


HOW DO WE KNOW IF THEY'RE ACTUALLY DEAD OR JUST PRETENDING
I strongly advocate for arrays starting at 1
@SpaceboyCantLol
HAHAHA
HAHAHA
HAHAHA
HAHAHA


LET'S PUT YOU OUTSIDE, LIL' GUY.
ALL LIFE IS PRECIOUS.
Arrays start at 1
OWLTURD.COM

Do arrays start at 0 or 1?

In most programming languages (C, C++, Java, and Assembly arrays in MASM), arrays start at index 0.

Example: myArray[0] is the **first element**.

In some languages like **Fortran or MATLAB**, you can start arrays at 1—but in **assembly**, you always use **0-based indexing** unless you explicitly add an offset.

So, in assembly, the first element is always at the **base address + 0 × element size**.

✅ **Short answer:** **Arrays start at 0 in ASM.**