Contents

[STRING OPERATIONS 2](#_Toc220507395)

[STRING PROCEDURES IN IRVINE32 17](#_Toc220507396)

[STR\_COMPARE PROCEDURE 19](#_Toc220507397)

[STR\_LENGTH PROCEDURE 22](#_Toc220507398)

[STR\_COPY PROCEDURE 23](#_Toc220507399)

[STR\_TRIM PROCEDURE 25](#_Toc220507400)

[STR\_UCASE PROCEDURE 27](#_Toc220507401)

[STRING LIBRARY DEMO PROGRAM 28](#_Toc220507402)

[2D ARRAYS 37](#_Toc220507403)

[SEARCHING AND SORTING ALGORITHMS 52](#_Toc220507404)

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:

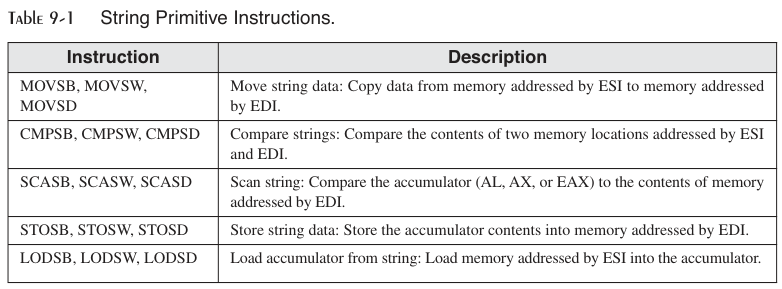
* Moving data
* Comparing memory blocks
* Scanning memory
* Loading from memory into registers
* Storing register values into memory

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



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**.

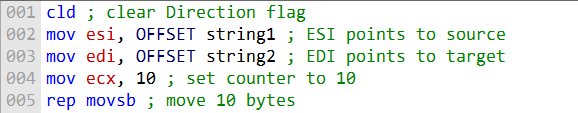
The most common repeat prefix is **REP**.

**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.



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

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

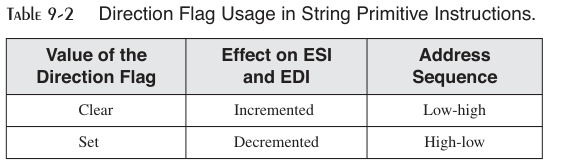
The Direction Flag can be changed **manually** using special instructions.

**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).



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**.

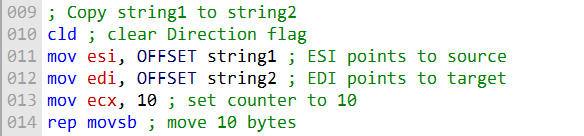
The Direction Flag must be set **correctly before** using a string instruction.

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.



MOVSB, MOVSW, and MOVSD

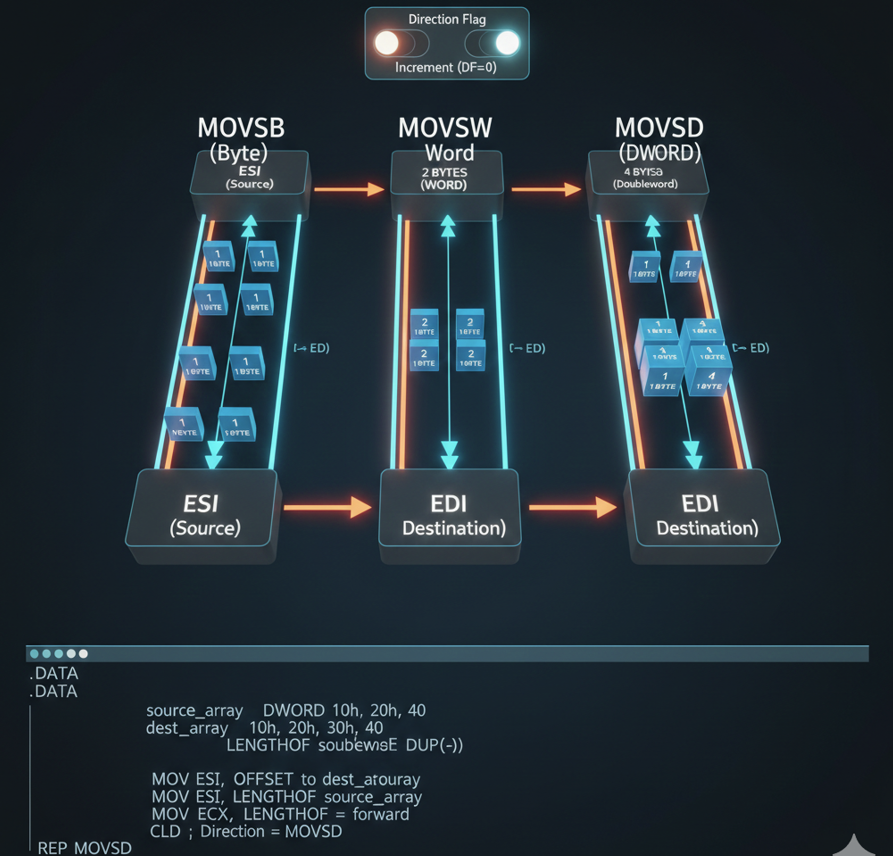
MOVSB, MOVSW, and MOVSD are used to copy data from one memory location to another.

They differ by the size of data they copy.

MOVSB copies bytes.

MOVSW copies words.

MOVSD copies doublewords.

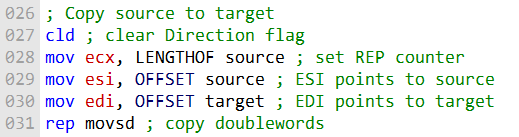


All three instructions use ESI as the source address.

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.



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.

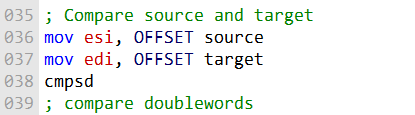
All three instructions use ESI to point to the first operand.

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:



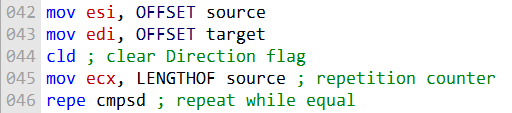
CMPSD compares two doubleword values.

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.



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.

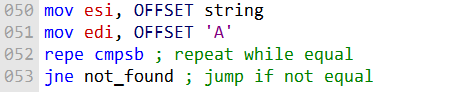
The Direction Flag controls whether ESI and EDI move forward or backward in memory.

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

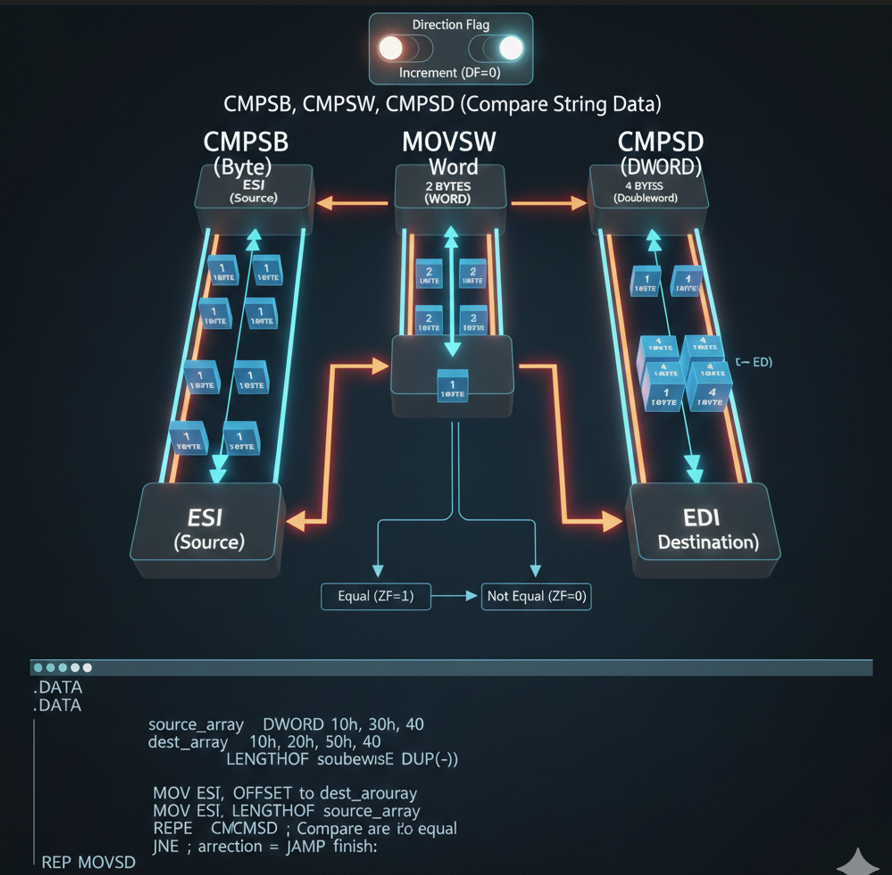
CMPSB, CMPSW, and CMPSD can be used for string-related operations.

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.



* 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

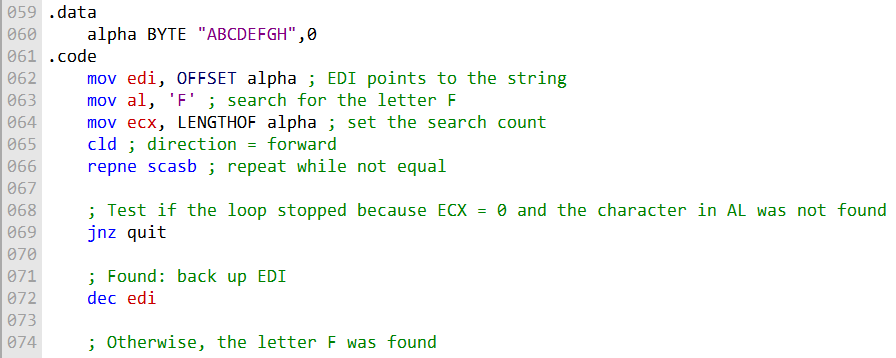
These instructions are useful for searching for a single value in a string or array.

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.



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.

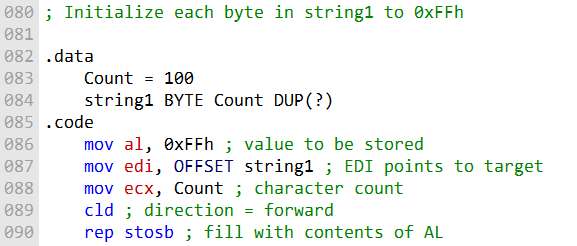
STOSW stores AX as a word.

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.



REP makes the STOSB instruction repeat while ECX > 0.

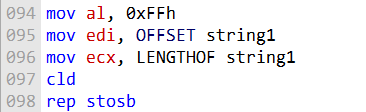
CLD clears the Direction Flag.

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.



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.



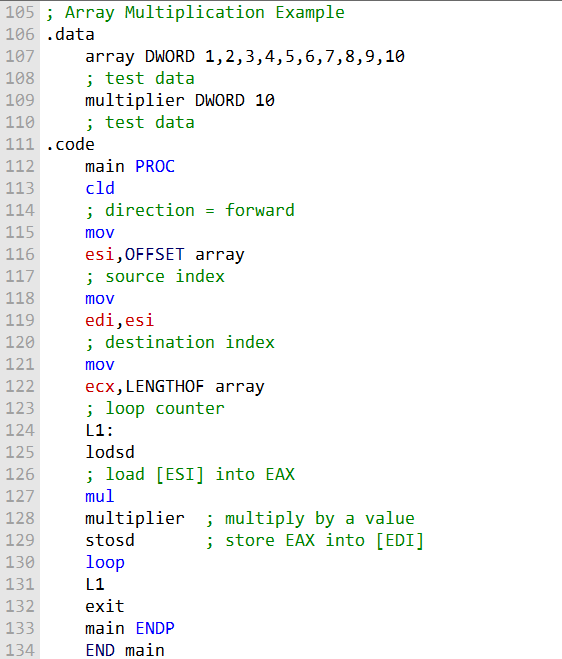
LODSB/LODSW/LODSD load a value from memory at ESI into the accumulator (AL/AX/EAX).

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.



**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**.

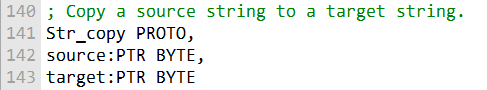
Str\_copy

**Str\_copy** copies a **source string** to a **target string**.

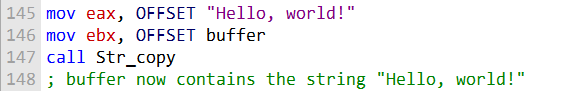
It takes **two arguments**:

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

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



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

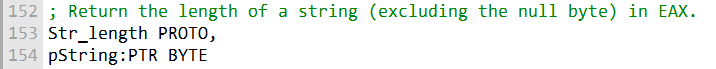


Str\_length

**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**.



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:



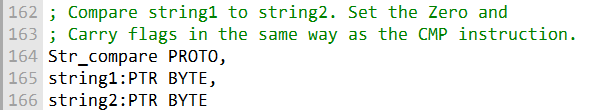
Str\_compare

**Str\_compare** compares **two strings**.

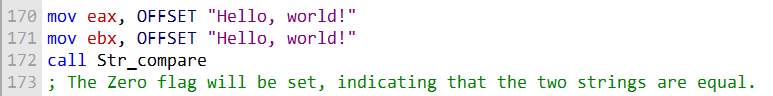
It sets the **Zero Flag (ZF)** and **Carry Flag (CF)** like the **CMP instruction**.

Takes **two arguments**:

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



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

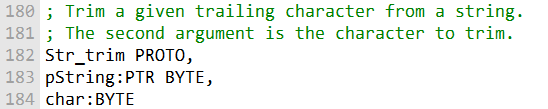


Str\_trim

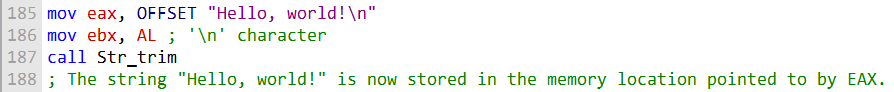
**Str\_trim** removes a **trailing character** from a string.

Takes **two arguments**:

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

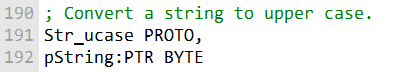


The following code shows how to use the Str\_trim procedure to trim the trailing newline character from the string "Hello, world!\n":

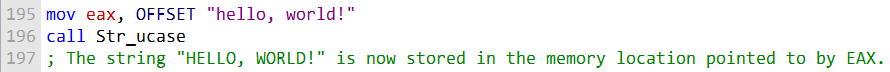


**Str\_ucase**

* **Str\_ucase** converts a string to **upper case**.
* Takes **one argument**: a **pointer to the string**.



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



STR\_COMPARE PROCEDURE

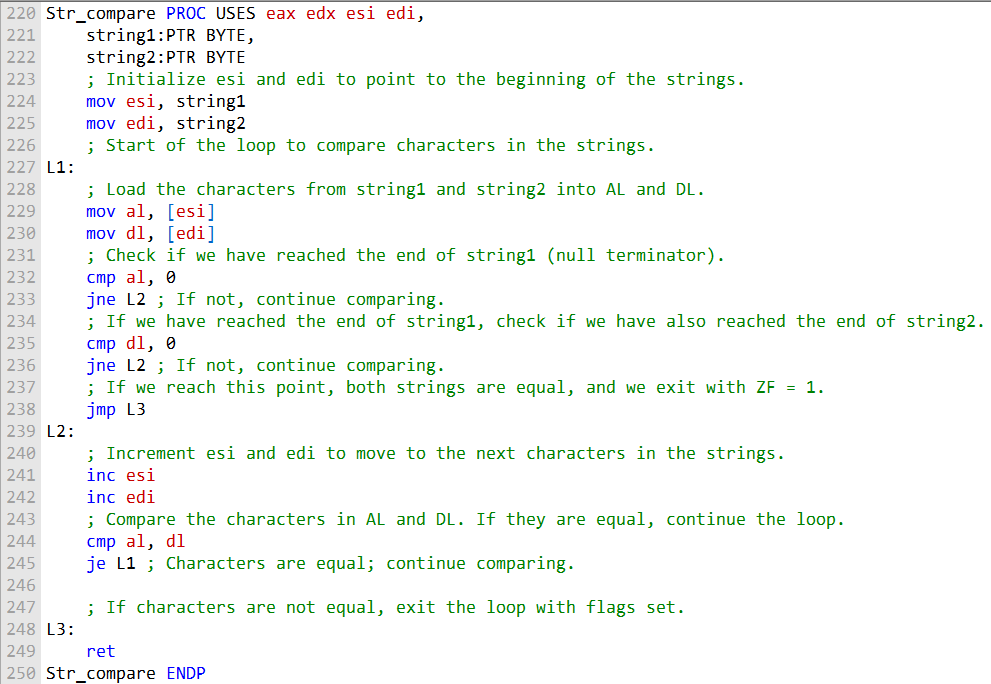
**Str\_compare** compares **two strings** byte by byte.

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:**

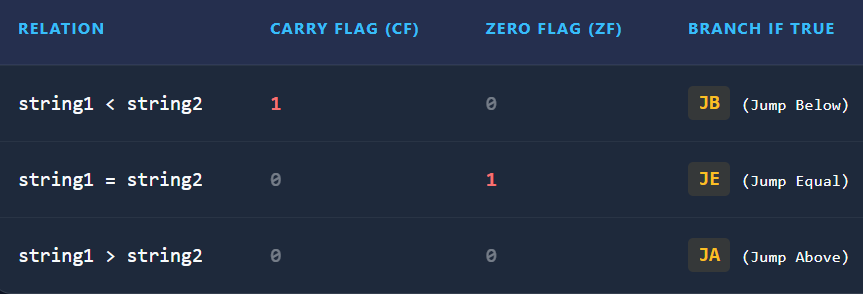
* Compare each byte of the two strings in order.
* If a byte in the two strings is **not equal**, **Carry Flag is set** and the comparison **stops**.
* If the **end of both strings** is reached without differences, **Zero Flag is set** and the comparison **stops**.



* **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:



* **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**.

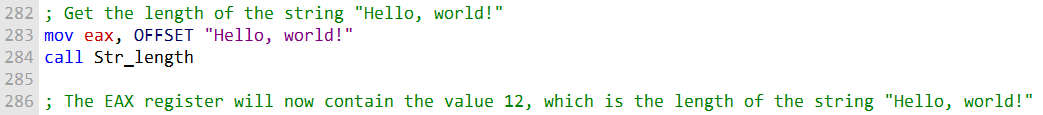
**Usage:** Pass the **address of the string** as the argument to the procedure.



* **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:



STR\_COPY PROCEDURE

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

**Arguments:**

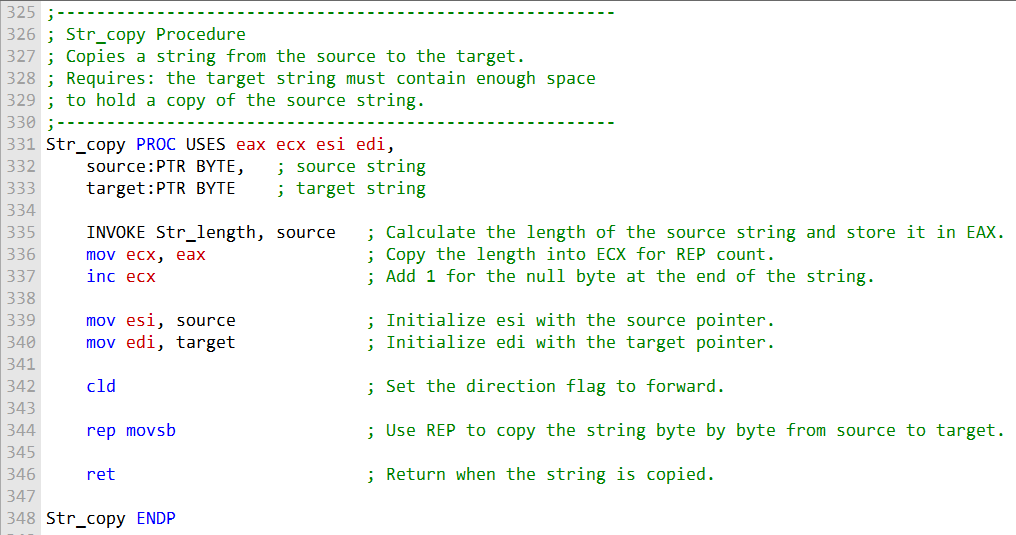
1. Pointer to the **source string**
2. Pointer to the **target string**

**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**.



**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.



STR\_TRIM PROCEDURE

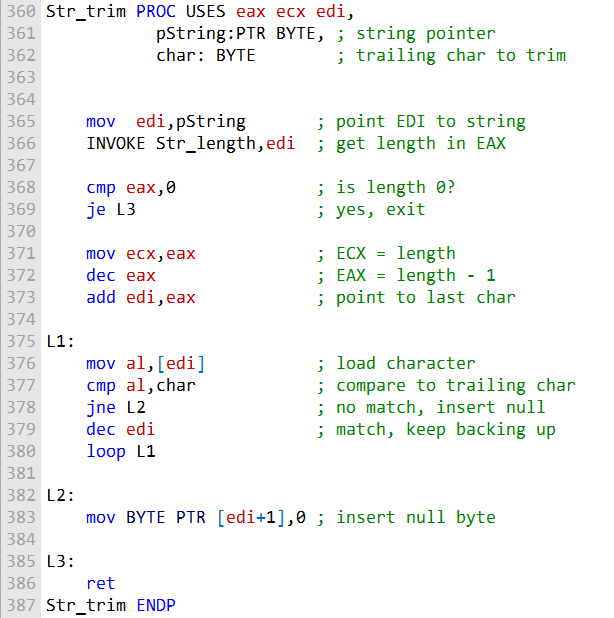
**Purpose:** Remove all occurrences of a **specified trailing character** from the **end of a null-terminated string**.

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.



**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):**

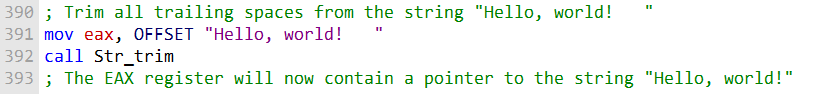
* Move byte at **[EDI]** into **AL**.
* Compare **AL** to the character to trim.
* If equal → **decrement EDI** and continue loop.
* If not equal → **exit loop**.

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

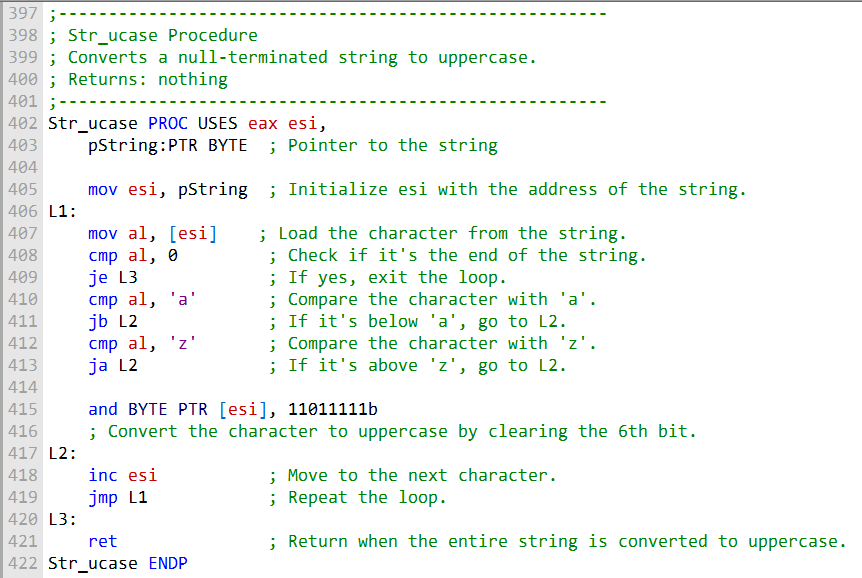
Cleanup:

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

Here is an example of how to use the Str\_trim procedure:



STR\_UCASE PROCEDURE



**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.

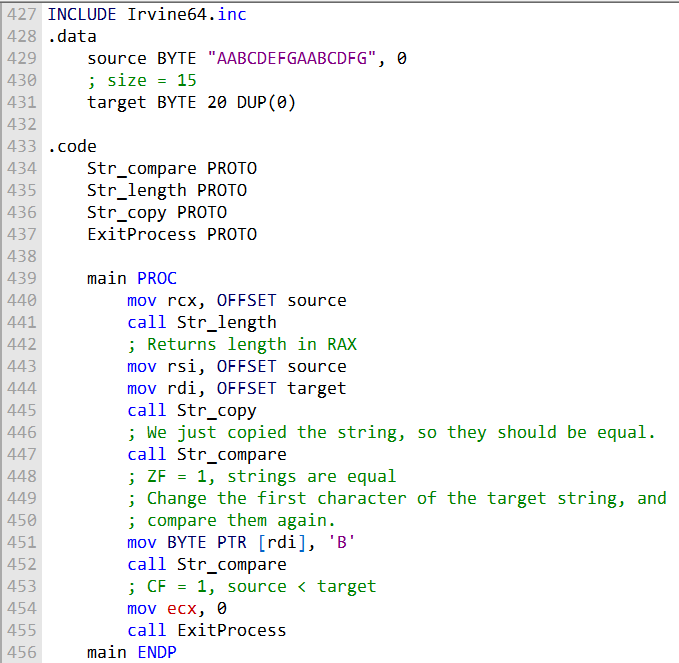
Program Output:

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**.

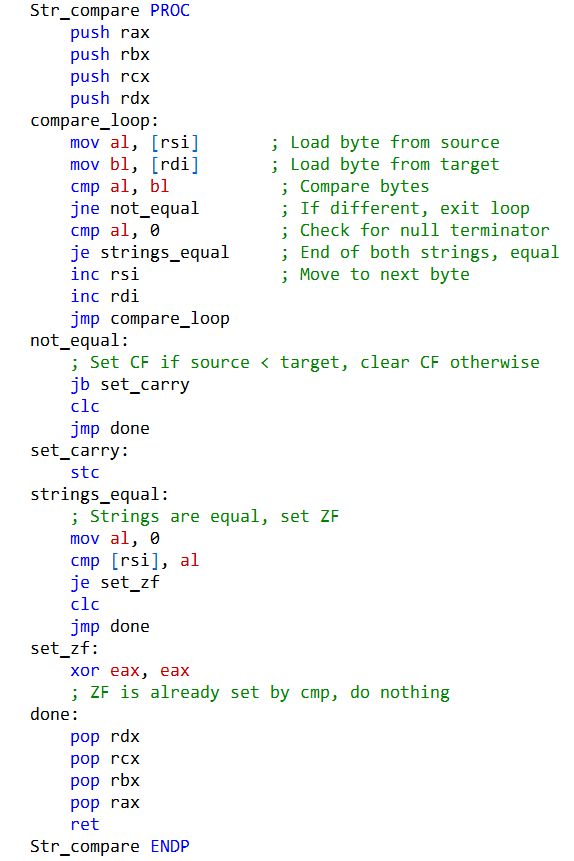
Strings using Irvine64



1️⃣. Str\_compare

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

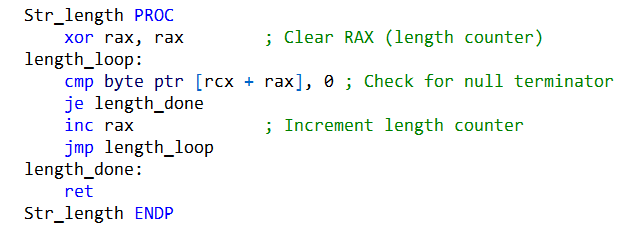
* Sets **ZF = 1** if equal
* Sets **CF = 1** if source < target



2️⃣. Str\_length

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

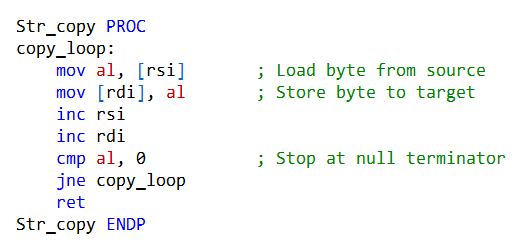
* Returns **length in RAX**



3️⃣. Str\_copy

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

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



✅ 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.



Irvine64 Library Setup

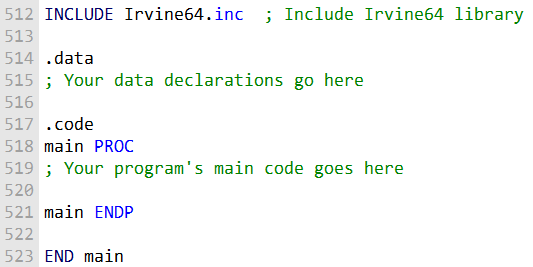
The **Irvine64 library** must be included in your assembly program.

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

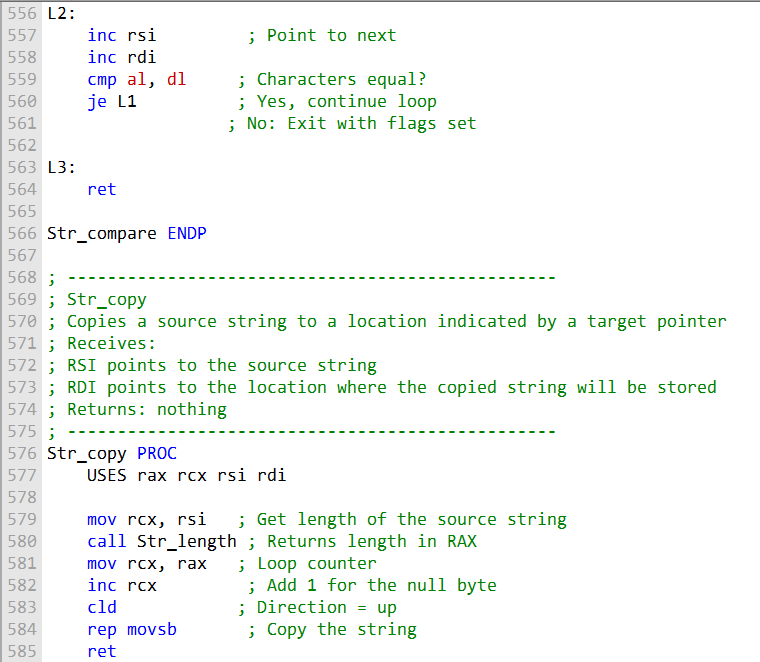
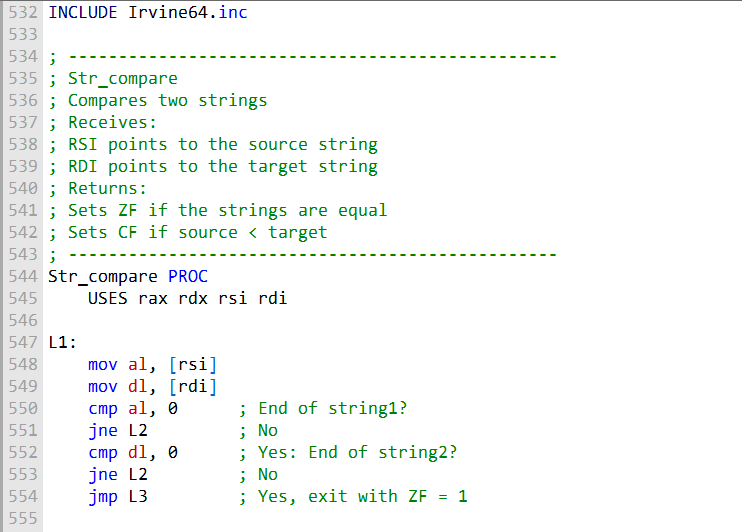
At the **beginning of your program**, include instructions to reference the Irvine64 library.

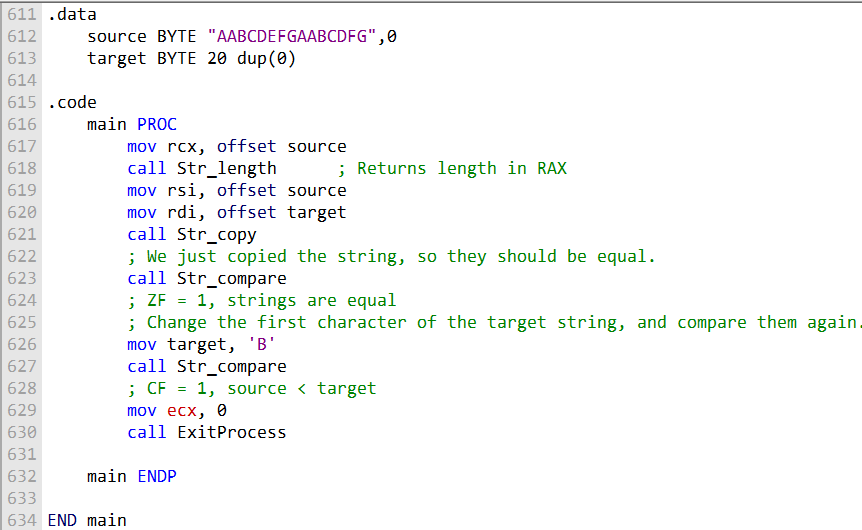
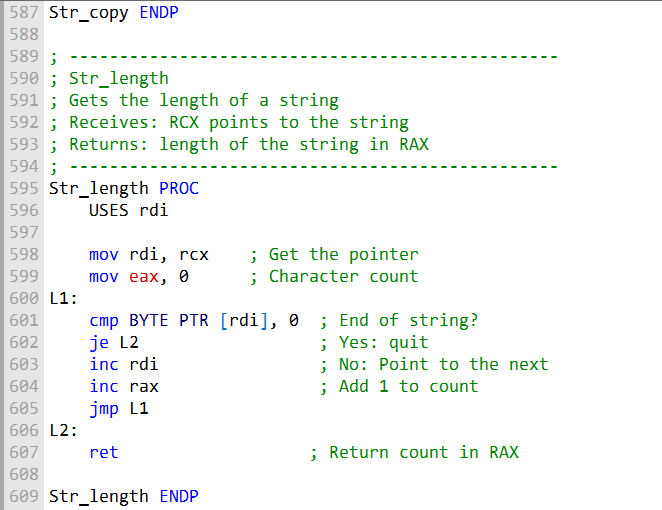
This usually involves **specifying the paths and configurations** for the library.

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



Actual Program:





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.

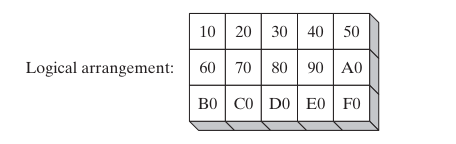
2D ARRAYS

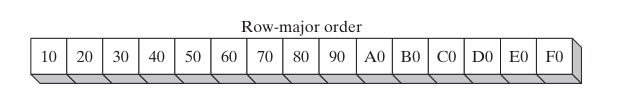
Row-major order and column-major order are just two ways computers can store a 2D array in memory.

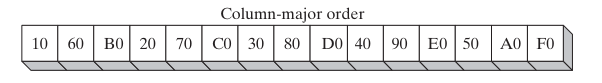
The main difference is the order in which the elements are laid out.

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

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





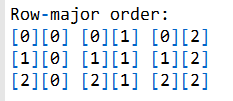


Row-major order

**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.

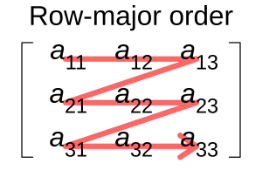


The **first element of the first row** is stored at the beginning of the memory block.

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.

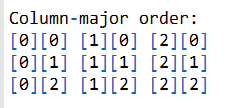


Column-major order

**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.

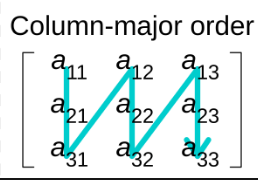


The **first element of the first column** is stored at the beginning of memory.

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

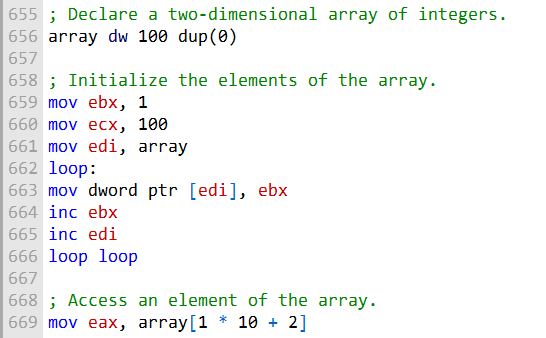
1. **Allocate memory** for the entire array.
2. The **size of the memory block** depends on:
   * The number of elements in the array
   * The **data type** of each element (e.g., byte, word, dword)
3. **Initialize the array elements** as needed.
4. To **access an element** at row i and column j, use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



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:



* 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**.

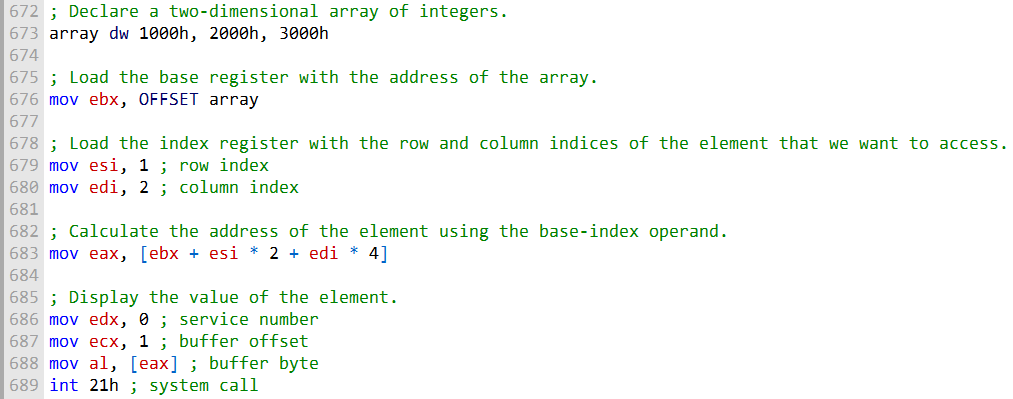
Load the **index register** with the **row and/or column index** of the element you want.

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].

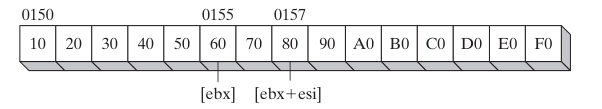
 The code example **displays 2000h**, which is the value at **row 1, column 2** of the array.

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.

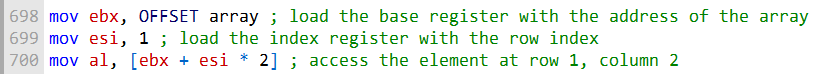


To access an elements of the array above, you can use the following formula:



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:



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)

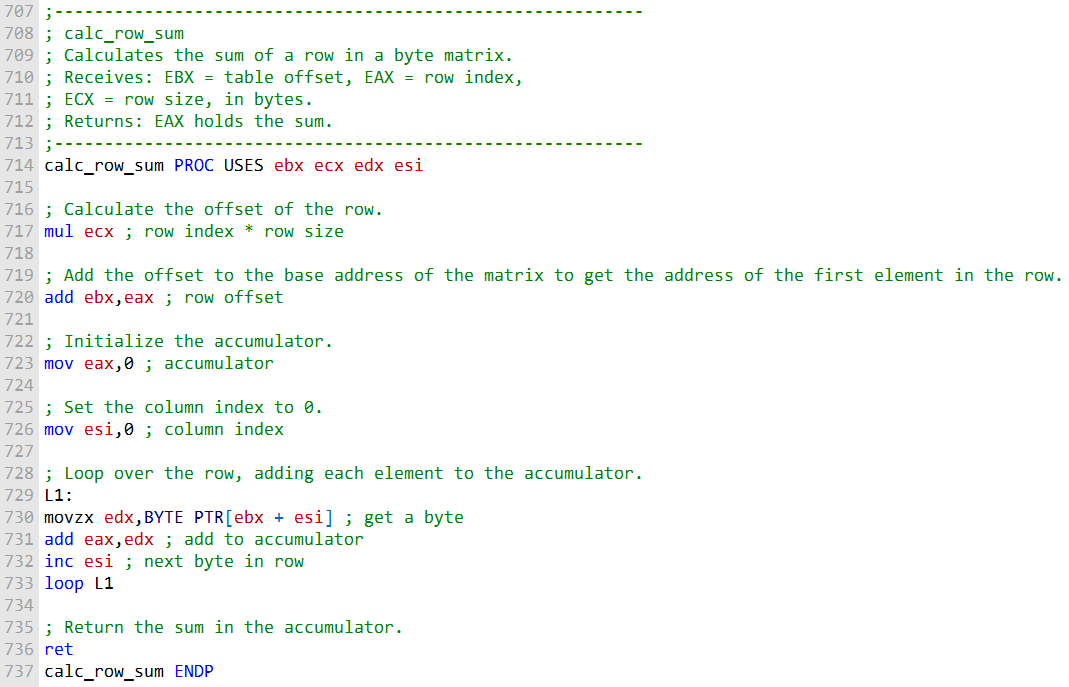
**Output:**

* 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:



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

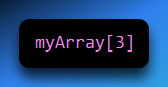
* Demonstrates **base-index addressing** for accessing elements of a **two-dimensional array**.
* Can be used to perform tasks like **calculating the sum of a row** in an array.
* Shows how to combine a **base address** and an **index register** to locate array elements in memory.

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.

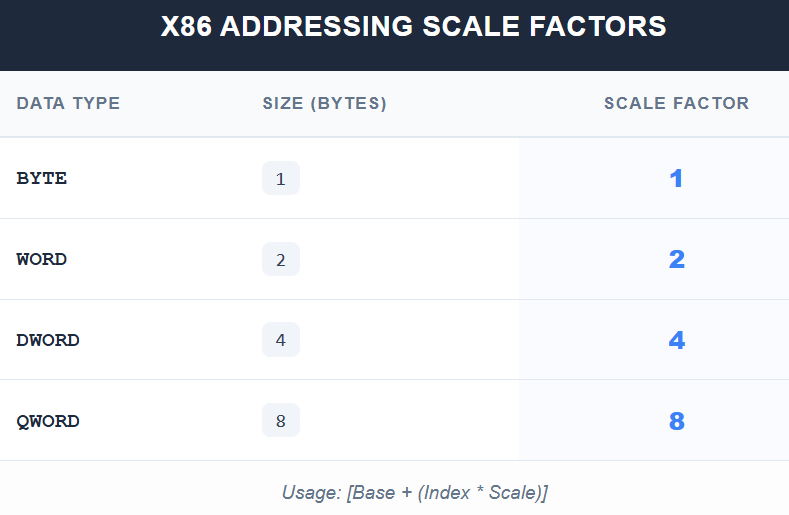
I. The Scale Factor Cheat Sheet

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).

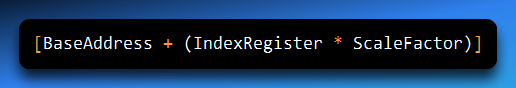
In **Assembly**, you must do that math manually. **Scale Factors** make this easier.



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

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

Syntax:



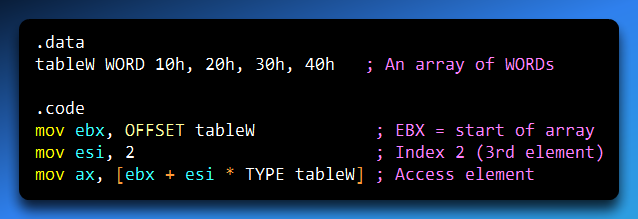
* **BaseAddress:** The start of the array (e.g., EBX)
* **IndexRegister:** The element index (e.g., ESI = 2 for the 3rd item)
* **ScaleFactor:** Multiplier based on the element size (1, 2, 4, or 8)

III. The Pro Way: The TYPE Operator **🛠️**

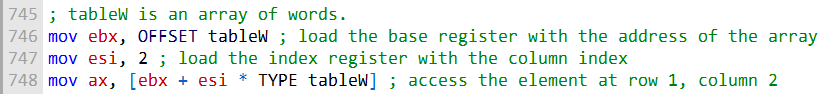
Hardcoding numbers like 4 or 8 is risky. If the array type changes (e.g., from DWORD to WORD), you must manually update every multiplication.

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

Example: Accessing a Word Array

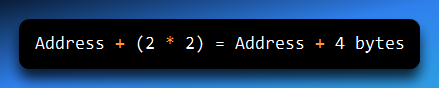


Or



**Explanation:**

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

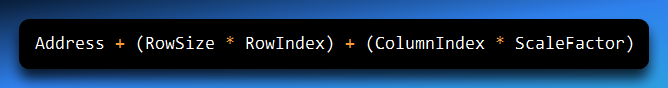


* Moves 30h into AX

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

IV. Why Scale Factors Matter for 2D Arrays

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



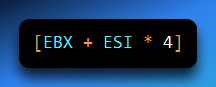
* Keep ESI or EDI as simple counters (0, 1, 2, 3)
* Hardware handles the actual memory offset

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.

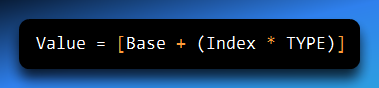
**Example:** Table of string addresses (DWORDs)

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



Big Ideas to Remember

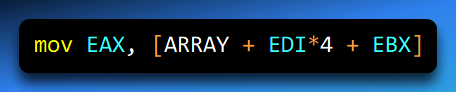
* **Scale Factor:** Converts a human-friendly index into a memory offset
* **Permitted Scales:** Only 1, 2, 4, and 8 are allowed by CPU hardware
* **TYPE Operator:** Always use it for maintainable code
* **Logic:**



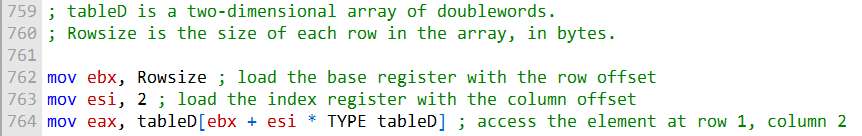
Base-Index-Displacement Operands

**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**:



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

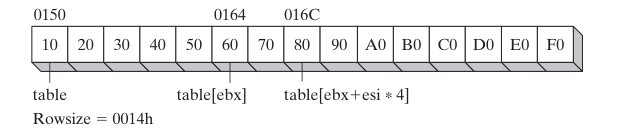


* tableD[ebx + esi \* TYPE tableD] points to the element at **row 1, column 2** of the array.
* **EBX** holds the **row offset**.
* **ESI** holds the **column offset**.
* TYPE tableD tells the assembler the elements are **doublewords (4 bytes)**, so the **scale factor is 4**.

**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**.

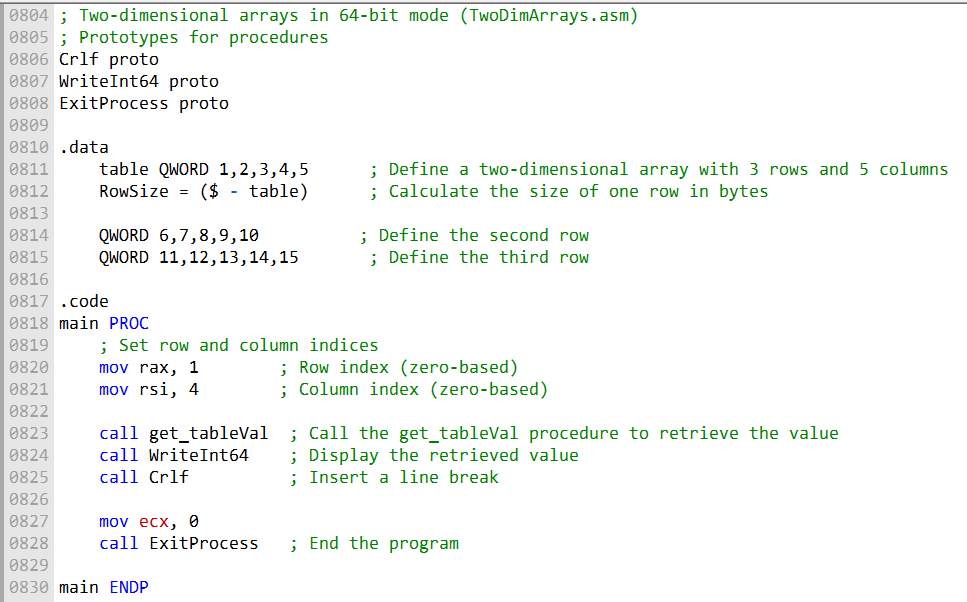


* The diagram shows **EBX** and **ESI** relative to the tableD array.
* **EBX** contains the **row offset**, and **ESI** contains the **column offset**.
* The tableD array starts at **offset 0150h**.
* **EBX = 20h** → this is the **size of one row in bytes**, so EBX points to the **start of the second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



Main Procedure Steps

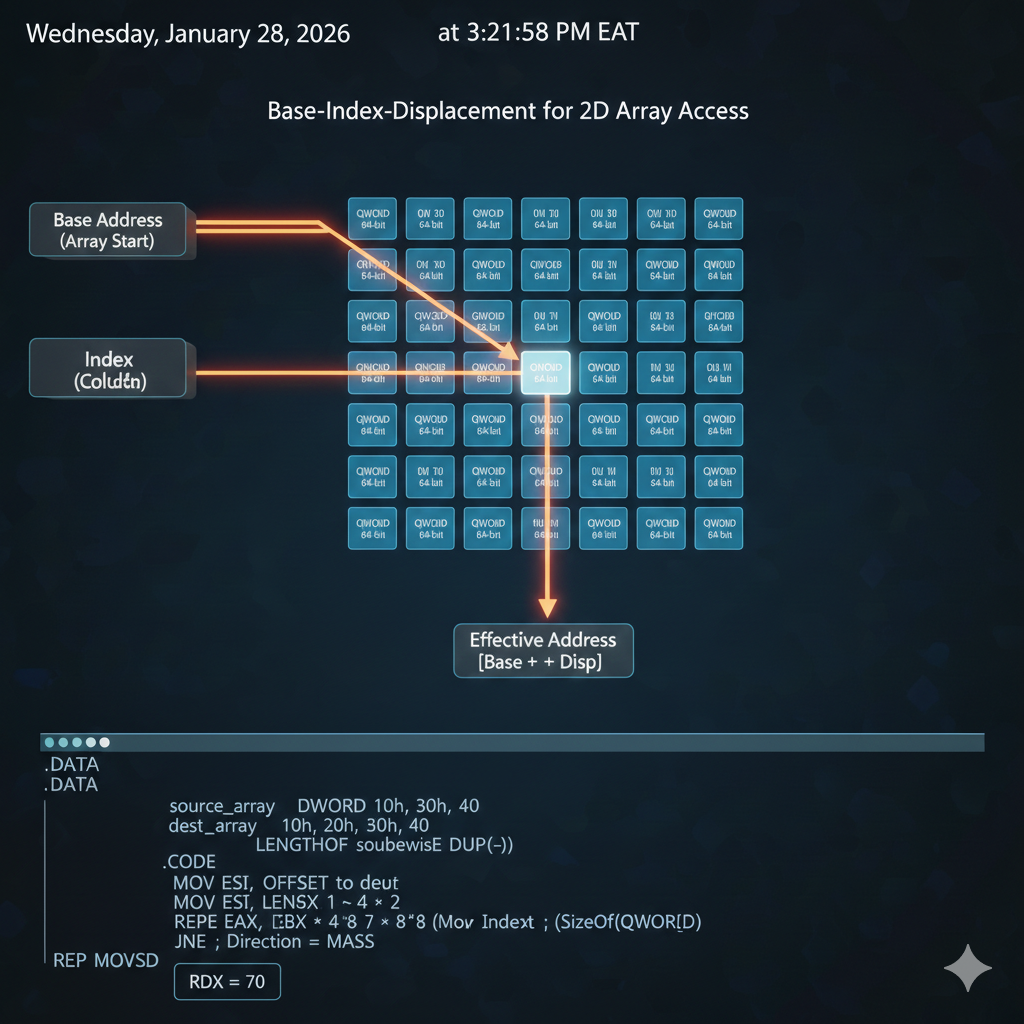
* Loads the **row index (1)** into the **RAX** register.
* Loads the **column index (4)** into the **RSI** register.
* Calls the **get\_tableVal** procedure to get the value at that row and column.
* Calls **WriteInt64** to display the value in **RAX**.
* Calls **ExitProcess** to end the program.

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)



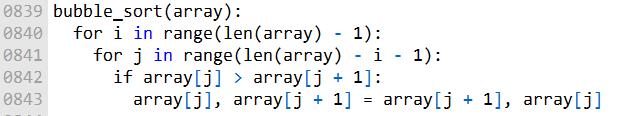
Key Takeaways

* 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**.

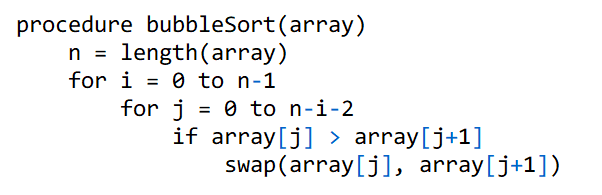
SEARCHING AND SORTING ALGORITHMS

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**.



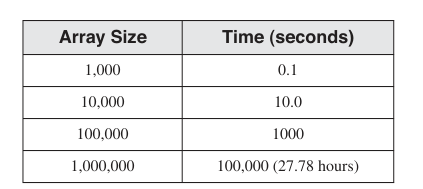
Or



* 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(n2).**

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:



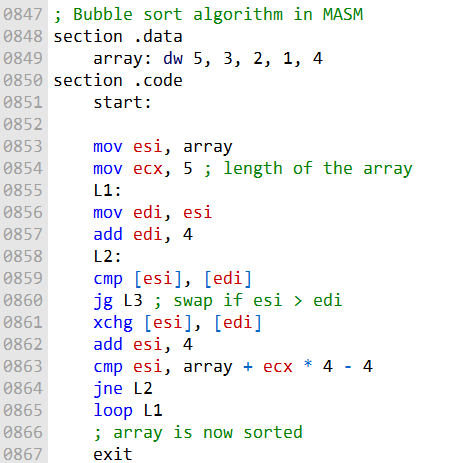
As you can see, the sort time increases quadratically with the array size.

**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**



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.

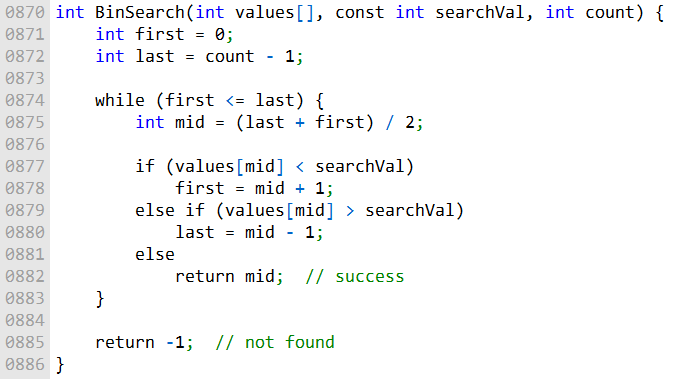
**Loop Control:**

* 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:**

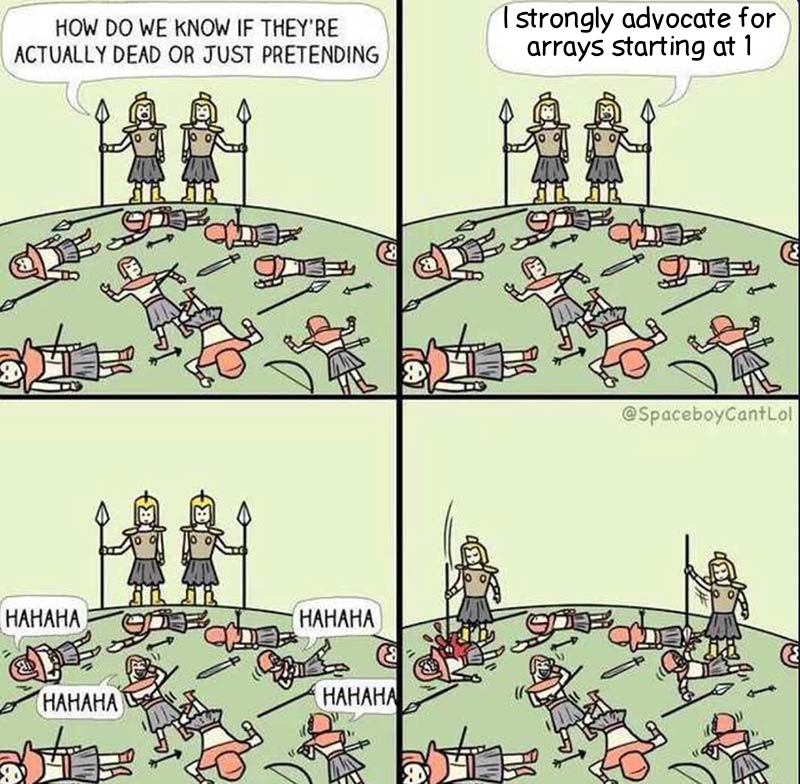
* After all iterations, the **array is sorted in ascending order**.

C++ version:

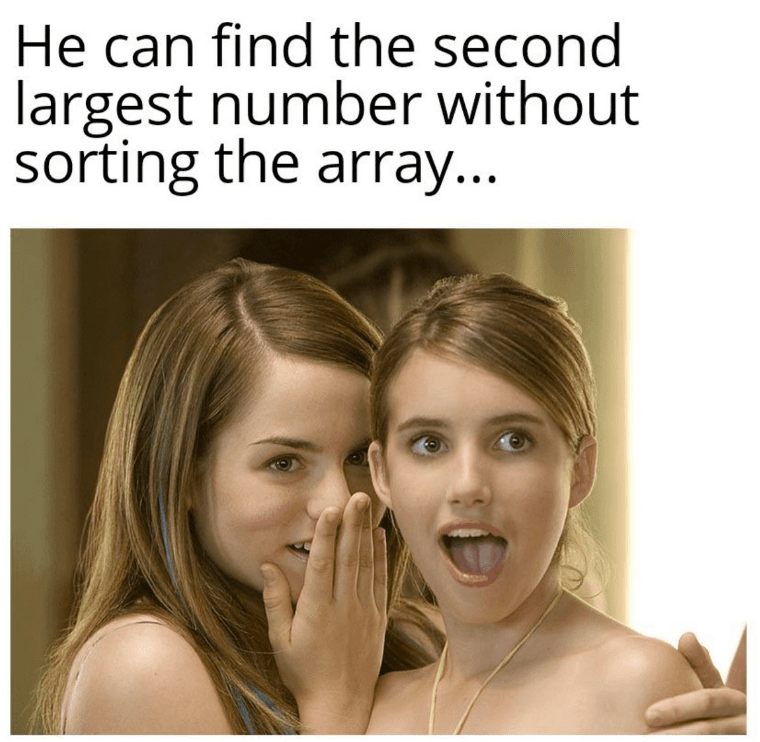


Final program: Read the file **BinarySortTest.asm**











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.**