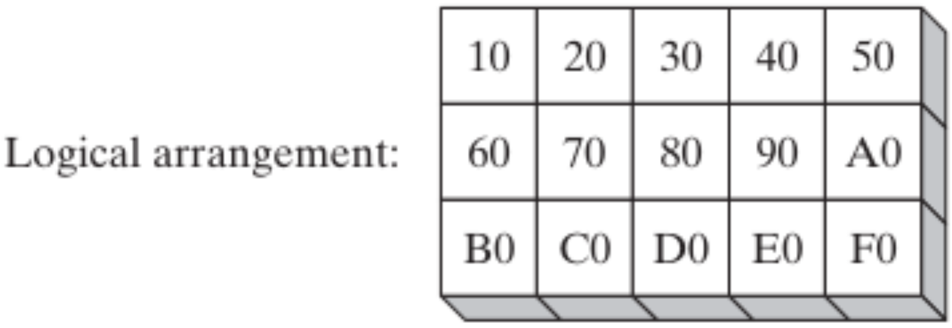
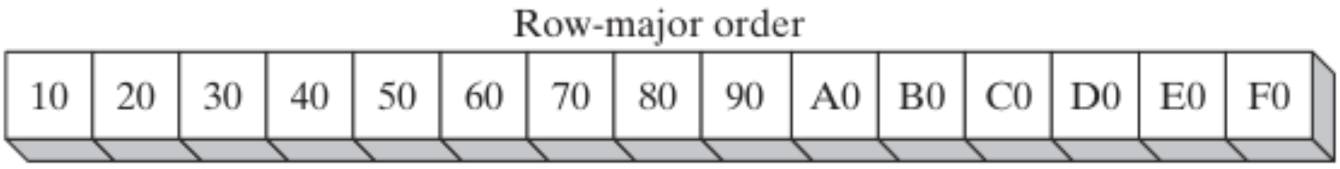
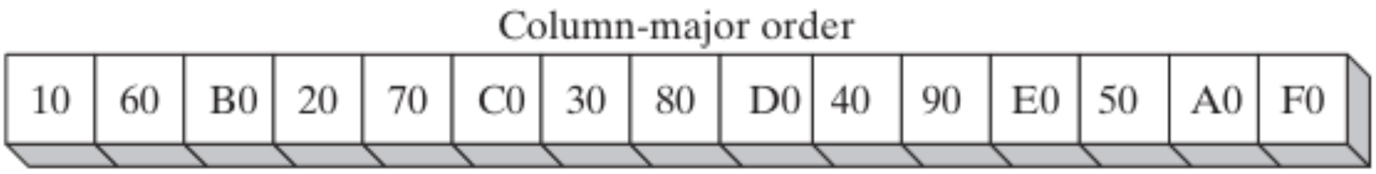
2D Arrays

Row-major order and **column-major order** are two different ways of storing a two-dimensional array in memory. The main difference is the order in which the elements of the array are stored.





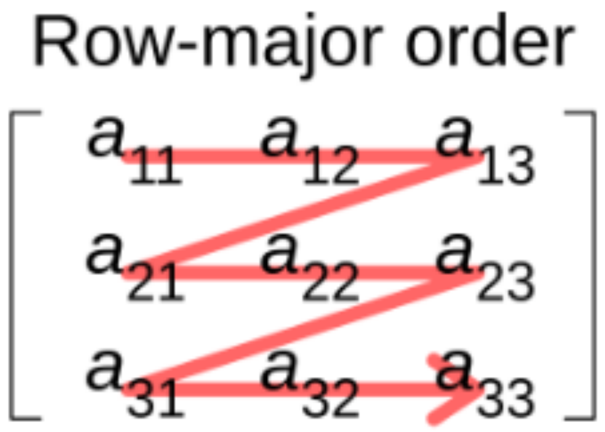


<u>Row-major order</u>

Row-major order is the most common method, and it is the method used by most high-level programming languages. In row-major order, the elements of each row are stored contiguously in memory.

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

This means that the first element of the first row is stored at the beginning of the memory block, followed by the second element of the first row, and so on. The last element of the first row is followed by the first element of the second row, and so on. This continues until the last element of the last row is stored.

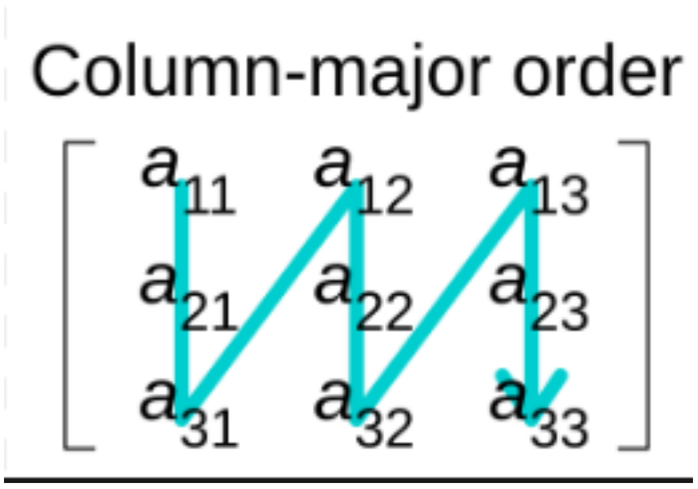

Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Column-major order

Column-major order is less common, but it is used in some applications, such as linear algebra. In column-major order, the elements of each column are stored contiguously in memory.

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

This means that the first element of the first column is stored at the beginning of the memory block, followed by the first element of the second column, and so on. The last element of the first column is followed by the second element of the second column, and so on. This continues until the last element of the last column is stored.


Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Which order to use?

The choice of which order to use depends on the application. Row-major order is generally more efficient for accessing elements of the array by row, while column-major order is more efficient for accessing elements of the array by column.

How to implement a two-dimensional array in MASM To implement a two-dimensional array in MASM, you can use either row-major order or column-major order. However, it is important to be **consistent with the ordering that you choose.**

Implementations

To implement a two-dimensional array in row-major order, you can use the following steps:

• Allocate a block of memory for the array.

• The size of the memory block depends on the size of the array and the data type of the elements of the array.

• Initialize the elements of the array. To access an element of the array, you can use the following formula:

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

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

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

```
; Declare a two-dimensional array of integers.
array dw 100 dup(0)

; Initialize the elements of the array.
mov ebx, 1
mov ecx, 100
mov edi, array
loop:
mov dword ptr [edi], ebx
inc ebx
inc edi
loop loop

; Access an element of the array.
mov eax, array[1 * 10 + 2]
```

Explanation:

- The array is declared as 100 contiguous DWORDs initialized to 0. This creates a 10x10 array since each DWORD is 2 bytes (10 * 10 = 100).
- 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 the elements.
- ECX counts iterations from 1 to 100 to fill all elements.
- Array elements are accessed using **rownumCols + col***. Here **row 1, col 2** is at offset* **10 + 2 = 12.**
- The elements are stored in row-major order - row 1, then row 2, etc sequentially in memory.

So 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 is a type of operand that allows you to access memory using the sum of two register values: the base register and the index register.

Base-index operands are very useful for accessing arrays, because they allow you to calculate the address of an element of an array using the row and column indices of the element.

To use a base-index operand, you must first load the base register with the address of the array.

You can then load the index register with the row and/or column index of the element of the array that you want to access.

Finally, you can use the base-index operand to access the element of the array.

The following example shows how to use a base-index operand to access an element of a two-dimensional array:

```
; Declare a two-dimensional array of integers.
array dw 1000h, 2000h, 3000h

; Load the base register with the address of the array.
mov ebx, OFFSET array

; Load the index register with the row and column indices of the element that we want to access.
mov esi, 1 ; row index
mov edi, 2 ; column index

; Calculate the address of the element using the base-index operand.
mov eax, [ebx + esi * 2 + edi * 4]

; Display the value of the element.
mov edx, 0 ; service number
mov ecx, 1 ; buffer offset
mov al, [eax] ; buffer byte
int 21h ; system call
```

The above code will display the value 2000h, which is the value of the element at row 1, column 2 of the array.

<u>Column-Major Order vs. Row-Major Order</u>

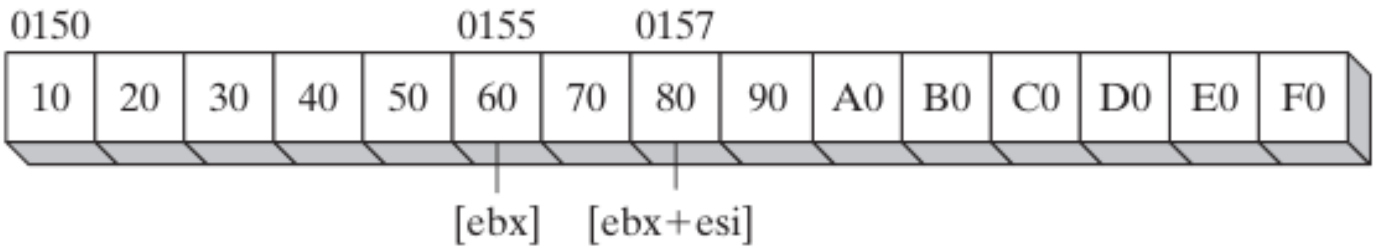
Two-dimensional arrays can be stored in memory in two different ways: column-major order and row-major order.

In **column-major order,** the elements of each column are stored contiguously in memory.

In **row-major order,** the elements of each row are stored contiguously in memory.

Most high-level programming languages use row-major order to store two-dimensional arrays.

Therefore, if you are writing assembly language code that will be used with a high-level language, you should use row-major order to store your two-dimensional arrays.



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:

```
mov ebx, OFFSET array ; load the base register with the address of the array
mov esi, 1 ; load the index register with the row index
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

The calc_row_sum procedure calculates the sum of a selected row in a matrix of 8-bit integers. It receives the following inputs:

- **EBX:** The offset of the matrix in memory.
- **EAX:** The index of the row to calculate the sum for.
- **ECX:** The size of each row in the matrix, in bytes.
- The procedure returns the sum of the row in the EAX register.

The procedure works by first calculating the offset of the row in the matrix. This is done by multiplying the row index by the row size.

The procedure then adds this offset to the base address of the matrix to get the address of the first element in the row.

The procedure then iterates over the row, adding each element to the accumulator. The accumulator is initialized to 0 before the loop starts.

The loop iterates until the end of the row is reached.

After the loop finishes, the procedure returns the sum in the accumulator in the EAX register.

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
```

The BYTE PTR operand size in the MOVZX instruction is required to clarify the operand size. The MOVZX instruction converts a byte to a doubleword.

The BYTE PTR operand size tells the assembler that the operand at the address specified by the ebx + esi register is a byte.

The calc_row_sum procedure is a useful example of how to use base-index addressing to access elements of a two-dimensional array and to perform common tasks on arrays, such as calculating the sum of a row.

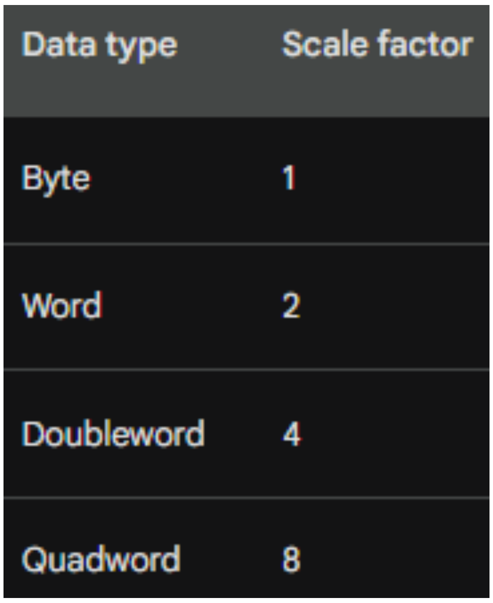
Scale Factors

A scale factor is a multiplier that is used to scale the index operand when accessing elements of an array using base-index addressing.

The scale factor is required because the size of the elements in an array can vary.

For example, if an array contains bytes, then the scale factor is 1. If an array contains words, then the scale factor is 2. If an array contains doublewords, then the scale factor is 4.

The following table shows the scale factor for different types of data:



Data type	Scale factor
Byte	1
Word	2
Doubleword	4
Quadword	8

To use a scale factor, you simply multiply the index operand by the scale factor before adding it to the base address of the array.

This will give you the address of the element in the array at the specified index.

For example, the following code accesses the element at row 1, column 2 of an array of words:

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

The TYPE tableW operand size in the MOV instruction tells the assembler that the elements in the array are words.

Therefore, the scale factor is 2. The esi * TYPE tableW operand is multiplied by 2 before it is added to the base address of the array.

This gives us the address of the element at row 1, column 2.

Scale factors can be useful for writing efficient code to access arrays.

By using scale factors, you can avoid having to keep track of the size of the elements in the array. This can make your code more readable and maintainable.

Base-Index-Displacement Operands

A base-index-displacement operand is a type of operand that allows you to access memory using the sum of the following values:

- **A displacement.**
- **A base register.**
- **An index register.**
- **An optional scale factor.**

Base-index-displacement operands are well suited for processing two-dimensional arrays. The displacement can be an array name, the base operand can hold the row offset, and the index operand can hold the column offset.

The following example shows how to use a base-index-displacement operand to access an element of a two-dimensional array of doublewords:

```
; tableD is a two-dimensional array of doublewords.
; Rowsize is the size of each row in the array, in bytes.

mov ebx, Rowsize ; load the base register with the row offset
mov esi, 2 ; load the index register with the column offset
mov eax, tableD[ebx + esi * TYPE tableD] ; access the element at row 1, column 2
```

The tableD[ebx + esi * TYPE tableD] operand specifies the address of the element at row 1, column 2 of the array.

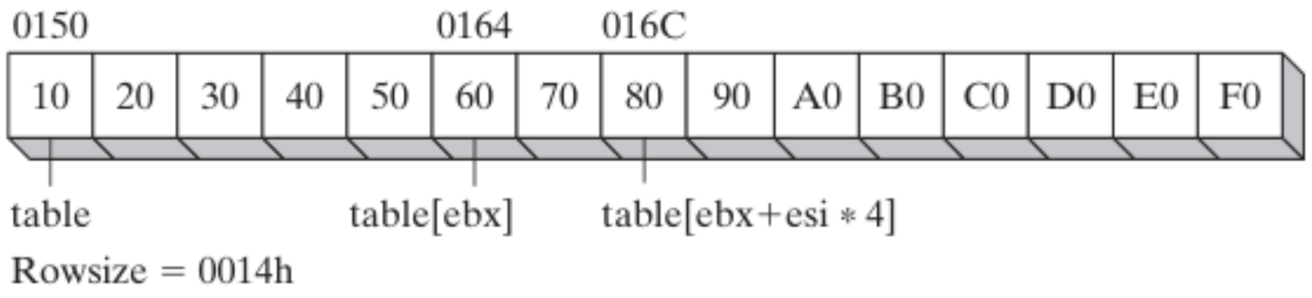
The ebx register contains the row offset, and the esi register contains the column offset.

The TYPE tableD operand size tells the assembler that the elements in the array are doublewords. Therefore, the scale factor is 4.

Base-index-displacement operands can be useful for writing efficient code to access arrays.

By using base-index-displacement operands, you can avoid having to keep track of the size of the elements in the array and the offset of each row in the array.

This can make your code more readable and maintainable.



The diagram you provided shows the positions of the EBX and ESI registers relative to the array tableD. The EBX register contains the row offset, and the ESI register contains the column offset.

The tableD array begins at offset 0150h. The EBX register contains the value 20h, which is the size of each row in the array, in bytes. Therefore, the EBX register points to the beginning of the second row in the array.

The ESI register contains the value 2, which is the column index of the element that we want to access. Therefore, the ESI register points to the third element in the second row of the array.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.

<u>Base-Index Operands in 64-Bit Mode.</u>

The program below is a short program that uses a procedure named get_tableVal to locate a value in a two-dimensional table of 64-bit integers. The program demonstrates how to use base-index-displacement operands in 64-bit mode.

```
; Two-dimensional arrays in 64-bit mode (TwoDimArrays.asm)
; Prototypes for procedures
Crlf proto
WriteInt64 proto
ExitProcess proto

.data
    table QWORD 1,2,3,4,5       ; Define a two-dimensional array with 3 rows and 5 columns
    RowSize = ($ - table)       ; Calculate the size of one row in bytes

    QWORD 6,7,8,9,10           ; Define the second row
    QWORD 11,12,13,14,15        ; Define the third row

.code
main PROC
    ; Set row and column indices
    mov rax, 1          ; Row index (zero-based)
    mov rsi, 4          ; Column index (zero-based)

    call get_tableVal   ; Call the get_tableVal procedure to retrieve the value
    call WriteInt64     ; Display the retrieved value
    call Crlf           ; Insert a line break

    mov ecx, 0
    call ExitProcess    ; End the program

main ENDP
```

The main procedure performs the following steps:

It loads the row index (1) into the RAX register.

It loads the column index (4) into the RSI register.

It calls the get_tableVal procedure to get the value at the specified row and column in the table array.

It calls the WriteInt64 procedure to display the value in the RAX register.

It calls the ExitProcess procedure to end the program.

The get_tableVal procedure performs the following steps:

It loads the row offset into the RBX register.

It multiplies the row offset by the size of a quadword to get the offset of the row in the array.

It adds the column offset to the row offset to get the offset of the element in the array.

It loads the value at the specified offset into the RAX register.

It returns from the procedure.

The get_tableVal procedure uses a base-index-displacement operand to access the element in the array.

The base operand is the RBX register, which contains the row offset.

The index operand is the RSI register, which contains the column offset.

The scale factor is omitted, because the elements in the array are quadwords, and the size of a quadword is 1.

The program you provided demonstrates how to use base-index-displacement operands in 64-bit mode.

Base-index-displacement operands are a powerful tool for accessing arrays in assembly language.