Multiple Initializers (or Arrays)

If multiple initializers are used in the same data definition (making a list of data), its label refers only to the offset of the first initializer. Other members can be accessed by incrementing the offset of first member. In the following example, assume **data** is located at offset 0000. If so, the value 10 is at offset 0000, 20 is at offset 0001, 30 is at offset 0002, and 40 is at offset 0003. It is just like array of bytes, all bytes are consecutive and accessible through a common mechanism.

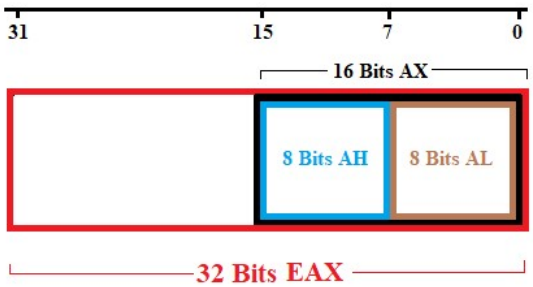
```
data BYTE 10,20,30,40
```

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

In Assembly language, to access integer 40 stored in 4th byte of the list, we will increment its label "**data**" by 3. i.e.

```
data+0 refers to offset 0000
data+1 refers to offset 0001
data+2 refers to offset 0002
data+3 refers to offset 0003
```

Recall Lab 1, first 8 bits of EAX register are called AL register. Byte variable can only be stored in AL register,



because both have same size. To print a byte (a number) from the list "**data**", offset of the respective byte will be moved to AL register. Before doing this, we will remove garbage value from EAX Register by storing 0 in it. This is done because the incoming byte will only overwrite first 8 bits of EAX register. Rest of the

24 bits will still have garbage value. After byte has been moved to AL, **WriteInt** will print the value of EAX register as whole.

Task 1: Declare an integer array, initialize it and display it.

Strings

Strings works just like arrays we discussed above. A string is an array of characters and each character is represented by a byte. Each character is represented by ASCII code (ranging from 0 to 255) which occupy a single byte. The only difference is that in array, integer values are separated by commas. While in string, characters are not separated by commas but encapsulated in quotations.

We already knew that by moving the first offset of string in EDX register and calling **WriteString** results into printing of string on console. But how can we access each letter of array? By doing the same as we did earlier for integers. But instead of characters, their respective ASCII codes will print on console. Jut replace `data BYTE 10,20,30,40` with `str BYTE "abcd"`. Console will print ASCII codes of a,b,c and d.

WriteChar: The WriteChar procedure writes a single character to the console window. Pass the character (or its ASCII code) in AL. Sample call:

```
mov al,'A'
call WriteChar       ; displays: "A"
```

Task 2: Declare a string variable of size 5, display the ASCII Code for the data stored on 1st two indexes and the data itself for the next two indexes.

DUP Operator

The *DUP operator* allocates storage for multiple data items, using a constant expression as a counter. It is particularly useful when allocating space for a string or array, and can be used with initialized or uninitialized data.

Syntax:

name BYTE 20 DUP(0)

An array with size of 20 bytes, initialized with 1 byte of 0 , 20 times.

Bytes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

name BYTE 4 DUP(1,2,3,4)

An array with size of 16, initialized with 4 bytes of (1,2,3,4), 4 times.

Bytes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

name BYTE 2 DUP("STACK")

An array with size of 10, initialized with 5 bytes of ("STACK"). 2 times. (As one character is one byte)

Bytes	1	2	3	4	5	6	7	8	9	10
Value	S	T	A	C	K	S	T	A	C	K

name BYTE 4 DUP(1,2)

An array with size of 8, initialized with 2 bytes of (1,2). 4 times.

Bytes	1	2	3	4	5	6	7	8
Value	1	2	1	2	1	2	1	2

In DUP allocation, length of the array or string is pre-defined. We can access these elements in the same way we accessed elements earlier in previous two examples.

SizeOf

This procedure returns the size of an array or list.

Syntax:

```
sizeof data       ;Considering "data" as variable
```

Task 3: Output the size of integer array, string array and declaration with DUP.