Defining Datatypes 2

Main points:

- **At least one initializer** is required in a data definition, even if it is zero.
- Additional initializers, if any, are **separated by commas**.
- For integer data types, the initializer can be an integer literal or integer expression matching the size of the variable's type.
- To leave a variable uninitialized (assigned a random value), the **?** symbol can be used as the initializer.
- All initializers, regardless of their format, are converted to binary data by the assembler.

Example code:

```asm
01 ;AddTwo.asm - chapter 3 example:
02
03 .386
04 .model flat, stdcall
05 .stack 4096
06
07 ExitProcess PROTO, dwExitCode: DWORD
08
09 .data
10 sum DWORD 0
11
12 .code
13 main PROC
14
15     mov eax, 5
16     add eax, 6
17     mov sum, eax
18
19     INVOKE ExitProcess, 0
20
21 main ENDP
22 END main
```

This code creates a variable named sum and initializes it to the value 11. The mov instruction is

used to move the value of eax to the sum variable.

To run this code in a debugger, you can set a breakpoint on line 13 and step through the program one line at a time.

After executing line 17, hover the mouse over the sum variable to see its value. Or, you can open a Watch window to monitor the value of the sum variable.

```
01 value1 BYTE      'A'        ;character literal
02 value2 BYTE      0          ;smallest unsigned byte
03 value3 BYTE      255        ;largest unsigned byte
04 value4 SBYTE     -128       ;smallest signed byte
05 value5 SBYTE     +127       ;largest signed byte
06
07 ;Signed means it can be a negative or positive number
08 ;Unsigned means, it can only be a positive number
```

In both cases, the variable value6 will be uninitialized when the program starts running.

It is important to note that uninitialized variables can contain garbage values, so it is important to initialize them before using them.

Here is an example of how to initialize the value6 variable in both assembly language and C:

```
10 ; Question mark shows uninitialized data eg in C:
11
12 value6 BYTE ?
13 char value6;
```

DB and **BYTE** essentially refer to the same thing: an 8-bit data type.

SBYTE is used for signed 8-bit data.

```
15 ;Once the variable is initialized, it can be used safely in the program.
16 value7 BYTE 10h
17 value8 BYTE 20h
18 char value6 = 10;
19
20 value9 DB 'B'
21 value10 SB -12
```

value9 DB 'B':

- value9 is defined as a byte-sized variable (DB or BYTE directive).
- It is initialized with the character 'B'.
- Since it's a byte, it can store a single 8-bit value. In this case, it stores the ASCII value of the character 'B', which is 66 (in decimal).

value10 SB -12:

- value10 is defined as a signed byte-sized variable (SBYTE directive).
- It is initialized with the value -12.
- Being signed, it can hold both positive and negative values. In this case, it holds the value -12.

If multiple initializers(VALUES) are used in the same data definition, its label(VARIABLE), refers only to the offset of the first initializer.

This variable list is therefore acting like arrays in C.

```
list BYTE 10, 20, 30, 40
```

In the following example, assume list 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:

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

This means that the other three initializers in the list do not have their own labels.

This type of memory layout is common in many programming languages, and it is important to understand how it works in order to write efficient and correct code.

For example, if you need to access the value 20 in the list, you would need to use the offset 0001.

```
list BYTE 10, 20, 30, 40
     BYTE 50, 60, 70, 80
     BYTE 81, 82, 83, 84
```

These are arrays are right there.

The assembler will allocate contiguous memory for all of the data definitions in the .data segment, even if they do not have labels.

This means that the bytes defined in the second and third lines will be stored directly after the bytes defined in the first line.

```
list BYTE 10, 20, 30, 40
     BYTE 50, 60, 70, 80
     BYTE 81, 82, 83, 84

;This is how the above will be stored in memory
Offset  Value
0000    10
0001    20
0002    30
0003    40
0004    50
0005    60
0006    70
0007    80
0008    81
0009    82
000A    83
000B    84
```

It is important to note that the label list only refers to the offset of the first initializer in the byte sequence, which is 0000 in this case.

This means that the other two byte sequences do not have their own labels.

Why did we use a byte to deal with integers, yet they are dwords? an integer is 4 bytes?

There are a few reasons why you might want to use a byte to deal with integers in assembly, even though they are typically 4 bytes in size.

• Efficiency: Using bytes can be more efficient in terms of memory usage and execution speed. For example, if you are working with an array of integers, you can store the array in half the amount of memory if you use bytes instead of dwords. Additionally, some assembly language instructions are specifically designed to work with bytes, which can make your code more efficient.

• Compatibility: Some older CPUs and embedded systems only support bytes. If you need to write code that is compatible with these platforms, you will need to use bytes to deal with integers.

• Special cases: There are some special cases where you may need to use bytes to deal with integers, even on modern CPUs. For example, if you are working with a register file that only has 8-bit registers, you will need to use bytes to store and manipulate integers.

Here are some examples of when you might want to use a byte to deal with integers in assembly:

• When you are working with an array of integers and you need to save memory.
• When you are writing code for an older CPU or embedded system that only supports bytes.
• When you are working with a register file that only has 8-bit registers.
• When you are working with a file format that specifies that integers should be stored as bytes.

It is important to note that using bytes to deal with integers can also have some drawbacks.

For example, if you are not careful, you can accidentally overflow a byte variable, which can lead to unexpected results.

Additionally, some assembly language instructions are not designed to work with bytes, so you will need to be careful when using them.

Overall, whether or not to use bytes to deal with integers in assembly depends on your specific needs. If you are unsure, it is always best to consult the documentation for the CPU or platform

that you are targeting.

List BYTE 10 there we store 10 as a number using a byte? how does 10 which is supposed to be 4 bytes fit inside there?

You cannot store a 4-byte integer in a 1-byte variable. The list BYTE 10 statement simply stores the value 10 in the first byte of the list variable. The other three bytes of the list variable will be uninitialized.

If you try to access the other three bytes of the list variable, you will get unexpected results. For example, if you try to print the value of list[1], you may get a garbage value or even a program crash.

If you need to store a 4-byte integer, you should use a DWORD (double word) variable. DWORD variables are 4 bytes in size and can store any integer value from -2,147,483,648 to 2,147,483,647.

Here is an example of how to store a 4-byte integer in a DWORD variable:

```
my_integer DWORD 10

;To access it ...

mov eax, my_integer
```

Yes, you can store an integer in assembly in a byte, or a dword, etc. The size of the variable that you use to store the integer will depend on the range of values that you need to store.

For example, if you need to store a small integer, such as a counter or a flag, you can use a byte variable. Byte variables are 1 byte in size and can store any integer value from 0 to 255.

If you need to store a larger integer, such as the number of bytes in a file or the size of a memory allocation, you can use a dword variable. Dword variables are 4 bytes in size and can store any integer value from -2,147,483,648 to 2,147,483,647.

You can also use larger variables, such as qword (8 bytes) and word (16 bytes), to store even larger integers.

I get it fully, a byte can hold 256 characters, so it can store values up to that point. and that is if its unsigned. If signed, it can store values of integers from -128 to 127, If its a DWORD, the range increases, and so on.

Chatgpt:

You've got it exactly right! Your understanding is spot on.

• A byte can indeed store values from 0 to 255 when it's unsigned.
• If it's signed, a byte can store values from -128 to 127.
• A DWORD, being 4 bytes, can store a much wider range of values, from -2,147,483,648 to 2,147,483,647, when signed.
• The size of the variable determines the range of values it can hold, with larger variables capable of representing larger ranges of integers.

So, your realization is accurate: the size of the variable determines the range of values it can store. Bytes are great for small numbers, while DWORDs, qwords, etc., are used for larger integer values. Keep up the good work with your understanding of assembly language!

```
;Within a single data definition, its initializers can use different radixes.
;Character and string literals can be freely mixed. eg

list1 BYTE 10, 32, 41H, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h
```

****