

# Memory, Storing Data, and Indirection

Abyan Majid

July 7, 2023

## 1 Memory

Below is the basic structure of a memory chip.

Address	Cell
0	Data 0
1	Data 1
2	Data 2
3	Data 3
...	...
$n - 1$	Data $n - 1$

$\longleftrightarrow$   
Size (Bytes)

A memory chip is comprised of memory cells. Each memory cell:

- is capable of storing data of a specified size in bytes.
- has a unique address that points to it.

When a memory is first initialized, it has "garbage" - that is, it has random or unpredictable data. "Garbage" therefore may be remnants of data from previous programs, uninitialized variables, and many more possibilities.

### 1.1 Memory size & byte conversion

To compute the size of a memory, you multiply the number of cells with the size of the cells.

$$\text{Memory size} = \text{No. of Cells} \times \text{Cell Size}$$

All cells in a memory always have the same size. However, the size itself depends on the memory chip, hence may differ compared to other memory chips. Some memory chips have cells each sized 1 byte, some others 2 byte, 3, 4, etc.

Below lists the bit-to-byte and to higher units conversions:

- 8 Bits  $\rightarrow$  1 Byte
- 1024 Bytes  $\rightarrow$  1 Kilobyte ( $2^{10}$  bytes)
- 1024 Kilobytes  $\rightarrow$  1 Megabyte ( $2^{20}$  bytes)
- 1024 Megabytes  $\rightarrow$  1 Gigabyte ( $2^{30}$  bytes)
- 1024 Gigabyte  $\rightarrow$  1 Terabyte ( $2^{40}$  bytes)
- 1024 Terabyte  $\rightarrow$  1 Petabyte ( $2^{50}$  bytes)
- 1024 Petabyte  $\rightarrow$  1 Exabyte ( $2^{60}$  bytes)

and so on...

## 1.2 Operations with memory

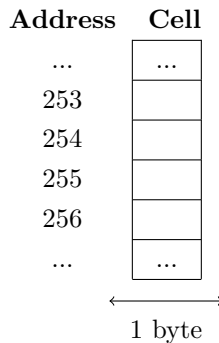
There's two operations you can do with a memory:

- Write (arg=address, data): What it means to "write" is simply to store data in a memory cell/s. When you do a "write" operation, you usually pass two arguments (1) the address of the cell, and (2) the data to be stored.
- Read (arg=address): What it means to "read" is to simply know or see the data stored in a memory cell. When you do a "read" operation, you simply pass the address of the cell which contains the data you wish to read.

## 2 Storing data in memory


### 2.1 Storing data of the same size as the cells'

Suppose we have a memory chip with cells sized 1 byte each.




And we wanna store an integer 158 to the 254th cell. If you do the math, the binary representation of the integer 158 is 0b10011110. You should notice that 0b10011110 is an 8-bit binary. Since each cell is sized 1 byte = 8 bits, 0b10011110 can be stored into one cell using a write operation (i.e. write(adr=254, data=158), NOTE that this is not actual code, just a visualization.).

Address	Cell
...	...
253	
254	0b10011110
255	
256	
...	...

  
 1 byte

Now that is pretty lengthy, and I hope you realize that even greater data will result in much longer binaries. That's why it is conventional to **represent data stored in memory cells in hexadecimal notation**. If you do the math, the hexadecimal representation of 158 is 9E.

Address	Cell
...	...
253	
254	0x9E
255	
256	
...	...

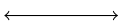
  
 1 byte

That looks much neater!

## 2.2 Storing data of higher sizes

Now what if we have a memory cell with cells sized 1 byte each, but we want to store a data that has a size higher than 1 byte?

Address	Cell
0	
1	
2	
3	
4	
...	...

  
 1 byte


Suppose that we want to store the string "Johan". In ASCII encoding, we have:

Character	Hexadecimal	Binary
J	4A	01001010
o	6F	01101111
h	68	01101000
a	61	01100001
n	6E	01101110

As you can see, in ASCII encoding, each character is exactly 1 byte long. So, even though we can't store "Johan" as a string in one cell, we can store each character in different cells such that when 5 cells are put together, it would read "Johan"!

Here's one way to do it:


Address	Cell
0	0x6F
1	0x6E
2	0x4A
3	0x68
4	0x61
...	...

  
 1 byte

## 2.3 Endians

You may notice that something's amiss. Which is that reading these cells in order, we get "onJha" instead of "Johan". Thing is, that is perfectly valid! You see, what's happening here is we have an **endian**. An endian is a ordering of multi-byte data in the memory either from the largest to the smallest, or from the smallest to the largest. In this case, our string "Johan" is stored from largest to smallest! If it were to be stored from smallest to largest, we would have the following instead:

Address	Cell
0	0x61
1	0x68
2	0x4A
3	0x6E
4	0x6F
...	...

  
 1 byte

Both orders are perfectly valid!

### 3 Indirection

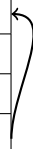
Memory indirection is when instead of storing a data to a memory cell, you store a reference that points to another cell. For example, look at the Rust code below.

```
1     fn main() {  
2         let x = 42;  
3         let y = &x;  
4     }
```

Listing 1: Example Python Code

Above is a memory indirection done in Rust, wherein we stored an integer "42" (0x2A in hexadecimal) in the memory, and stored a reference "&x" in another memory cell - such that if we read the content of  $y$ , we will get the value of  $x$ . In our usual memory diagram, the memory indirection may be depicted as follows:

Address	Cell
...	...
668	0x2A
669	
700	
701	668
...	...



In a nutshell, if you store a memory indirection to a cell  $B$ , which points to the cell  $A$ , if you read  $B$ , you will be returned the data stored in  $A$ .