Memory & I/Os

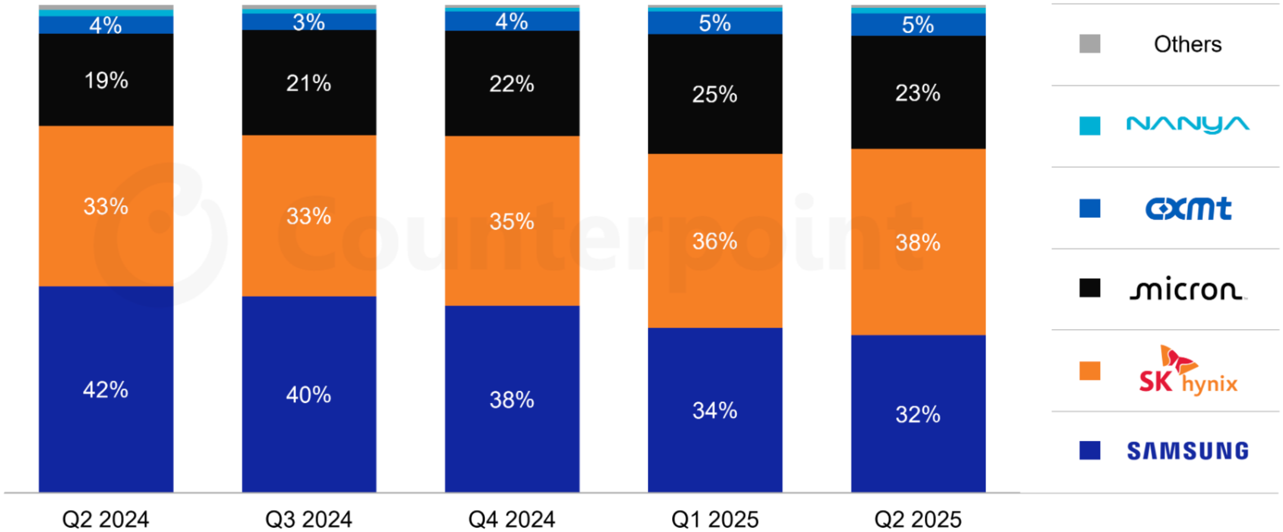
Philippe Bonnet, bonnet@di.ku.dk
HPPS 2025 – 2b

Outline

- Memory
- Pointers
- Byte ordering
- Serialization

DRAM market

Global DRAM Market Share by Revenue (Q2 2024 - Q2 2025)



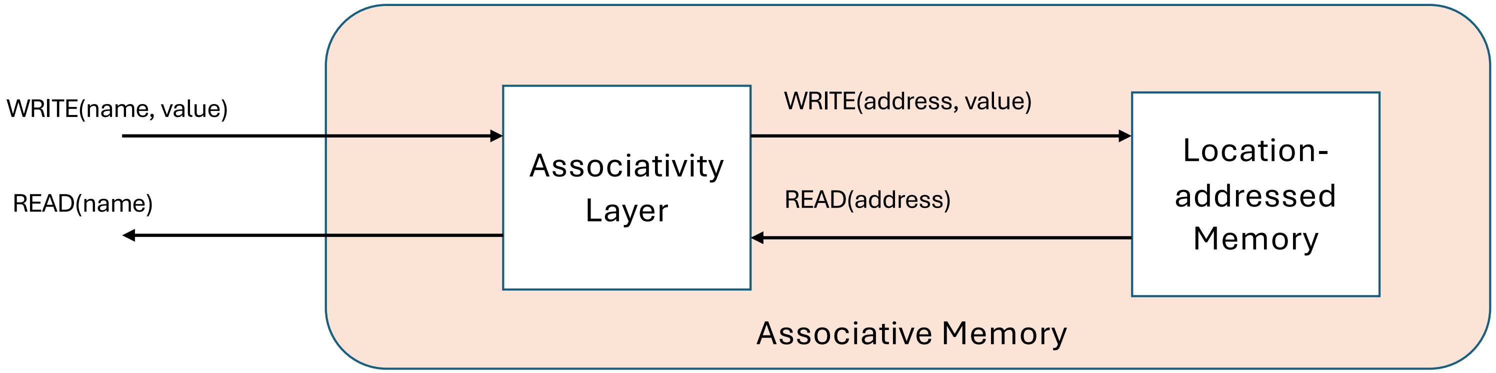
Source: Counterpoint Research Memory Tracker and Forecast, Q2 2025

Counterpoint

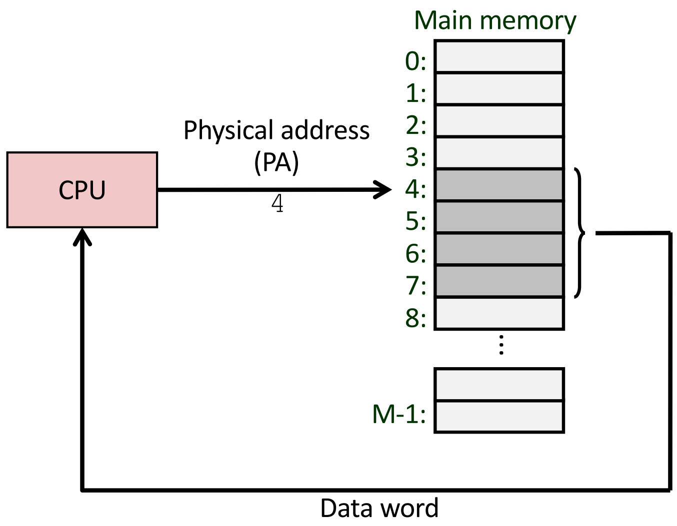
DRAM includes HBM, DDR5, DDR4

Source: Saltzer and Kaashoek

Memory Abstraction



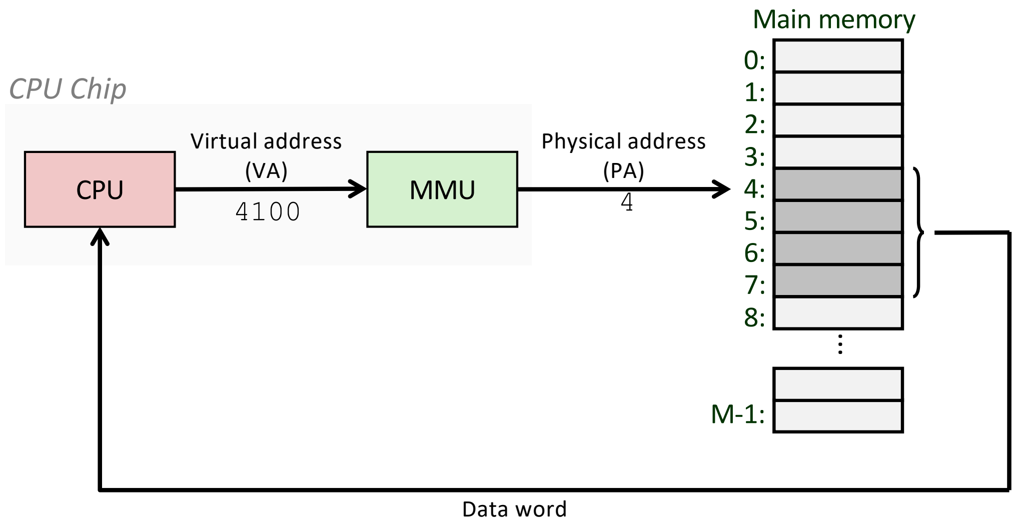
Physical addressing



Used in “simple” embedded systems, e.g., kitchen appliances

Virtual addressing

MMU: Memory Management Unit



- Used in all modern servers, desktops, and laptops
- One of the great ideas in computer science

Address space

- An address space is an ordered set of contiguous addresses (non-negative integers).
- Each word has an address . Address points to 1^{st} byte of word.
  On 64 bits machine, each word is 8B.

- Physical address space => associated to RAM

- Virtual address space => associated to each process
  (see Troels intro and more later)

Pointers

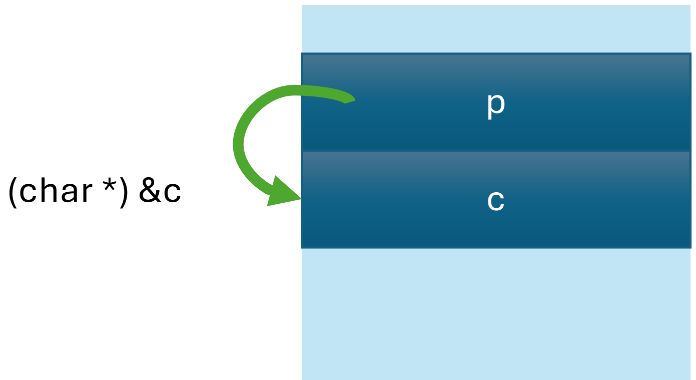
- ”A pointer is a variable that contains the address of a variable.”
  K & R
- We can access the address of any variable in a C program
  - **&x** gives us the (virtual) address of **x**
  - If x has type T, &x has type T*

Pointers

p is a char pointer.

It is a variable that contains the address of a char variable.

```
char *p;
char c;
p = &c;
```



Why pointers in C?

- To share data without copies
- To manage indirections (e.g., linked list)
- To manage data placement / locality and memory allocation

Beware initialization

```c
1  #include <stdio.h>
2
3  int main(void) {
4    int *ptr;
5    *ptr = 20;
6    printf("%d\n", *ptr);
7    return 0;
8  }
```

```
htl719@kumac ~/D/C/H/h/week2> gcc -Wall -Werror ex.c -o ex
ex.c:5:4: error: variable 'ptr' is uninitialized when used here [-Werror,-Wuninitialized]
    5 |    *ptr = 20;
      |     ^~~
ex.c:4:11: note: initialize the variable 'ptr' to silence this warning
    4 |    int *ptr;
      |            ^
      |             = NULL
1 error generated.
```

Pointers

- Pointers are valid, null or indeterminate.
- A pointer is null when assigned 0
- Null pointers evaluate to false in logical expressions
- Dereferencing indeterminate pointers leads to undefined behaviour

Always initialize pointers!

Byte ordering

Consider a 64 bit integer, i.e., 8 bytes.

How are these 8 bytes ordered in memory?

- **Big endian**
  - *Least significant byte has highest address (“comes last”).*
  - *Network: TCP*
- **Little endian**
  - *Least significant byte has highest address (“comes first”).*
  - x86, ARM, Risc-V CPUs

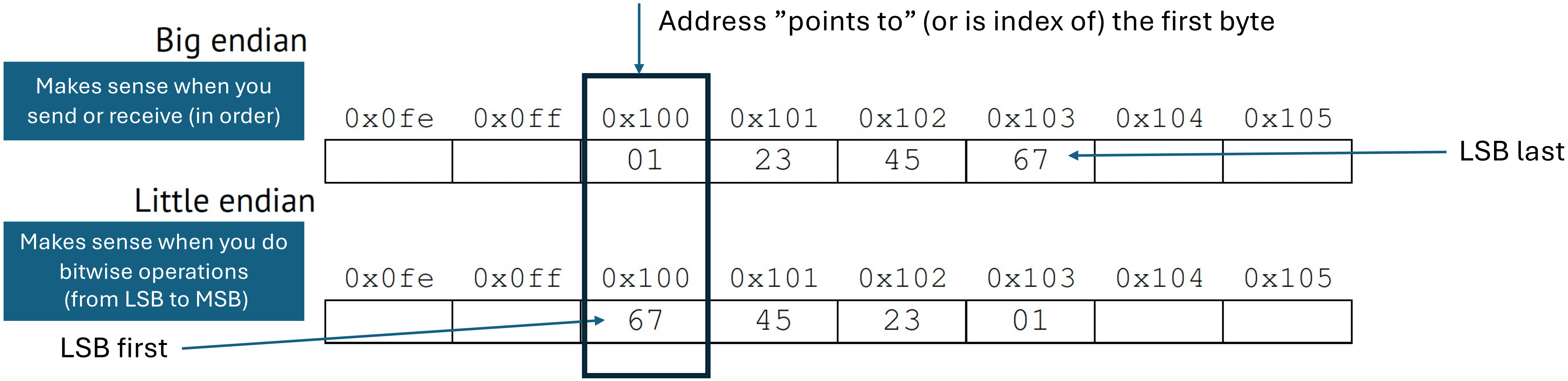
Byte re-ordering when sending/receiving on the network

Example

- Value encoded on 4 bytes (e.g., uint32_t):

$$X = 0x012345\underbrace{67}_{\text{LSB}}$$

- *Address of this value &X = 0x100*



Examining data representation

- **Code to print byte representation of data**
  - Casting pointer to `unsigned char*` allows treatment as byte array.

```
void show_bytes(unsigned char* start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}
```

`printf` directives:

- `%p`: Print pointer.
- `%x`: Print hexadecimal.

Text representation

A text is a sequence of characters.

How to interpret binary numbers into characters?

Character sets
A character set maps a *number* to a *character*.

- **ASCII** maps characters to the range 0—127 (asciitable.com).
  - A character (char) is represented on 8 bits.
  - Some are invisible/unprintable control characters
- Unicode is a superset of ASCII that defines tens of thousands of characters for all the world's scripts.
  - UTF8, UTF16, UTF32 encodings on up to 4 bytes

ASCII (mapping) table

Control characters				Normal characters											
000	nul	016	dle	032	␣	048	0	064	@	080	P	096	`	112	p
001	soh	017	dc1	033	!	049	1	065	A	081	Q	097	a	113	q
002	stx	018	dc2	034	"	050	2	066	B	082	R	098	b	114	r
003	etx	019	dc3	035	#	051	3	067	C	083	S	099	c	115	s
004	eot	020	dc4	036	$	052	4	068	D	084	T	100	d	116	t
005	enq	021	nak	037	%	053	5	069	E	085	U	101	e	117	u
006	ack	022	syn	038	&	054	6	070	F	086	V	102	f	118	v
007	bel	023	etb	039	'	055	7	071	G	087	W	103	g	119	w
008	bs	024	can	040	(	056	8	072	H	088	X	104	h	120	x
009	tab	025	em	041	)	057	9	073	I	089	Y	105	i	121	y
010	lf	026	eof	042	*	058	:	074	J	090	Z	106	j	122	z
011	vt	027	esc	043	+	059	;	075	K	091	[	107	k	123	{
012	np	028	fs	044	,	060	<	076	L	092	␣	108	l	124	|
013	cr	029	gs	045	-	061	=	077	M	093	]	109	m	125	}
014	so	030	rs	046	.	062	>	078	N	094	^	110	n	126	~
015	si	031	us	047	/	063	?	079	O	095	_	111	o	127	del

How printf works

```
int x = 1234;
printf("x: %d\n", x);
```

The text *string* that is passed to `printf()` looks like this in memory:

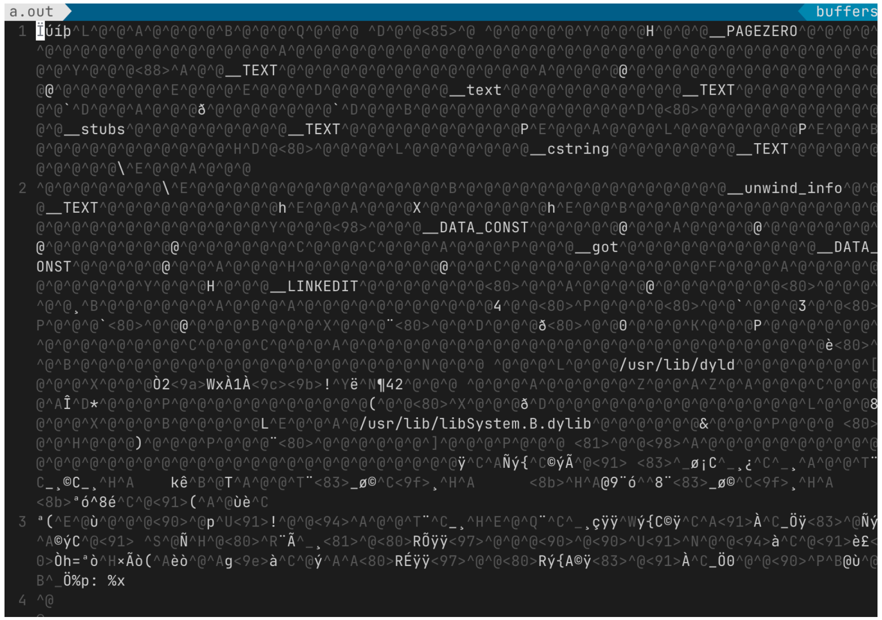
Characters	x	:		%	d	\n	\0
Bytes	120	58	32	37	100	10	0

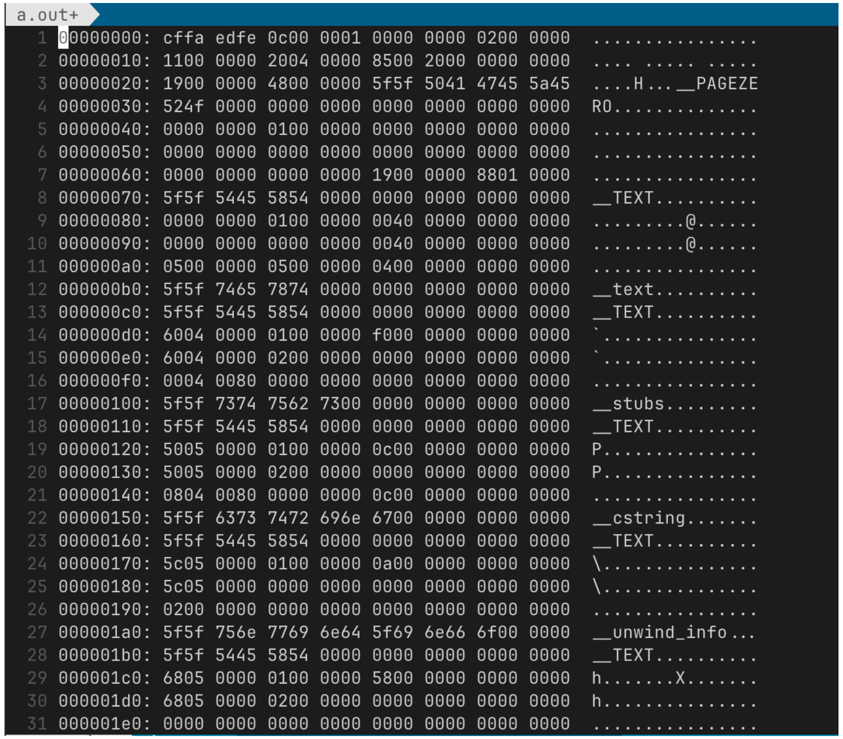
`printf()` rewrites format specifiers (`%d`) to the textual representation of their corresponding value argument:

Characters	x	:		1	2	3	4	\n	\0
Bytes	120	58	32	49	50	51	52	10	0

These bytes (except the `0`) are then written to *standard output* (typically the terminal) which interprets them as characters and eventually draws pixels on the screen.

How about files?





Binary file: sequence of bytes

Text file: sequence of characters (e.g., ASCII)

Writing bytes

```
#include <stdio.h>

int main() {
  // Open for writing ("w")
  FILE *f = fopen("output", "w");

  char c = 42;

  fwrite(&c, sizeof(char), 1, f);

  fclose(f);
}
```

- Produces a file `output`.
- File contains the byte 42, corresponding to the ASCII character `*`.
- **`char` is just an 8-bit integer type!**
  - No special “character” meaning.
  - Most Unicode characters will not fit in a single `char` (e.g. ’æ’ needs 16 bits in UTF-8).
  - Name is unfortunate/historical.
  - Signedness is *implementation-defined* for historical reasons.

Converting an non-negative integer to ASCII

```
FILE *f = fopen("output", "w");
int x = 1337;                   // Number to write;
char s[10];                     // Output buffer.
int i = 10;                     // Index of last character written.
while (1) {
  int d = x % 10;               // Pick out last decimal digit.
  x = x / 10;                   // Remove last digit.
  i = i - 1;                    // Index of next character.
  s[i] = '0' + d;               // Save ASCII character for digit.
  if (x == 0) { break; }        // Stop if all digits written.
}
fwrite(&s[i], sizeof(char), 10-i, f); // Write ASCII bytes.
fclose(f);                            // Close output file.
```

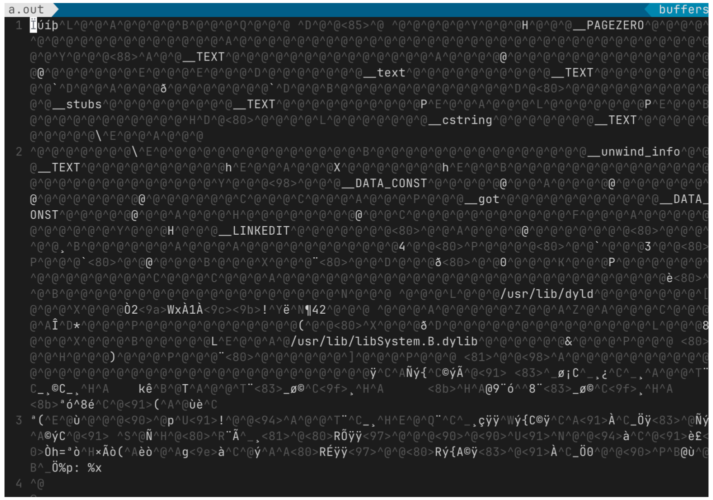
Reading all bytes in a file

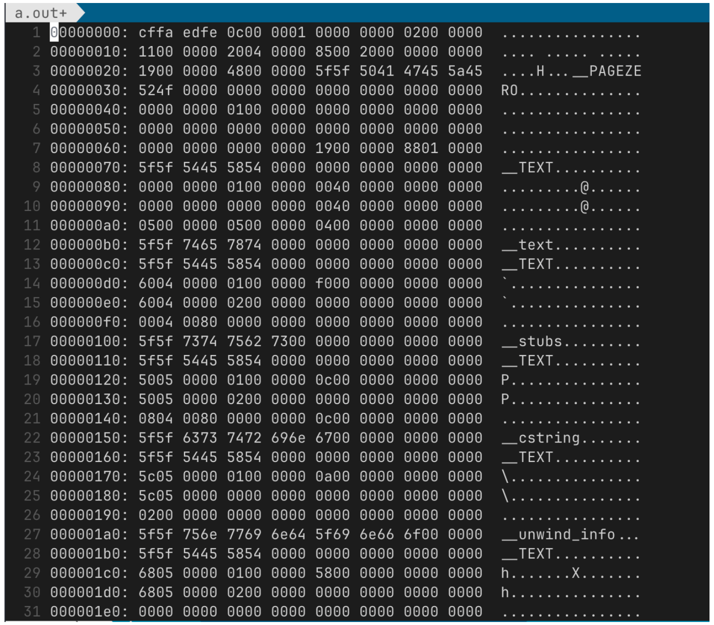
```
#include <stdio.h>
#include <assert.h>

int main(int argc, char* argv[]) {
  FILE *f = fopen(argv[1], "r");
  unsigned char c;
  while (fread(&c, sizeof(char), 1, f) == 1) {
    printf("%3d␣", (int)c);
    if (c > 31 && c < 127) {
      fwrite(&c, sizeof(char), 1, stdout);
    }
    printf("\n");
  }
}
```

Reading all bytes in a file

```
$ gcc -o fread-bytes -Wall -Wextra -pedantic fread-bytes.c
$ ./fread-bytes fread-bytes.c
 35 #
105 i
110 n
 99 c
108 l
117 u
100 d
101 e
 32
 60 <
...
```

How about files?





Compactness of storage

- A 32-bit integer takes up to 12 bytes to store as base-10 ASCII digits
- 4 bytes as raw data
- **Raw data takes up less space and is much faster to read.**
- But we need special programs to decode the data to human-readable form.

Take-aways

- Memory is abstracted as an array of bytes.
- The unit of read and write to memory is a word (8B on 64 bits machines)
  - Little endian: LSB first (on all modern processors); big endian on network.
- Pointers are variables that contain the addresses of variables, i.e., indexes in the memory array.
  - As variable, a pointer can be stored in memory (we can thus have pointers to pointers …)
- Text is just another interpretation of binary data (through character maps, e.g., ASCII or Unicode)
  - Interpreted when printing to terminal or when writing to/reading from a file (binary files are more compact than text files)