CSE351

hi! :)

- **Announcements:**
  - HW1 released later today

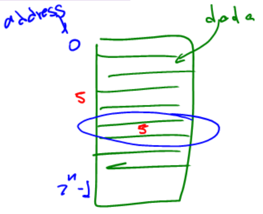
happy friday !

Today's topics

- **More on addresses/pointers**
- **Bit-level manipulations**
  - Boolean algebra
  - Boolean algebra in C

What is memory, really?

- How do we find data in memory?
- What is an address?
- What is a pointer?



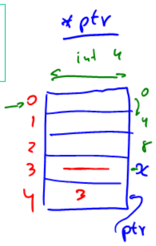
32-bit machine

Addresses and Pointers in C

& = 'address of value'
* = 'value at address'
or 'de-reference'

*(&x) is equivalent to x

- **Pointer declarations use ***
  - int * ptr; int x, y; ptr = &x;
  - Declares a variable ptr that is a pointer to a data item that is an integer
  - Declares integer values named x and y
  - Assigns ptr to point to the address where x is stored
- **We can do arithmetic on pointers**
  - ptr = ptr + 1; // really adds 4 (because an integer uses 4 bytes)
  - Changes the value of the pointer so that it now points to the next data item in memory (that may be y, may not – dangerous!)
- **To use the value pointed to by a pointer we use de-reference**
  - y = *ptr + 1; is the same as y = x + 1;
  - But, if ptr = &y then y = *ptr + 1; is the same as y = y + 1;
  - *ptr is the value stored at the location to which the pointer ptr is pointing



Arrays

*(big_array + 4)

*(big_array)

big_array [0]

index

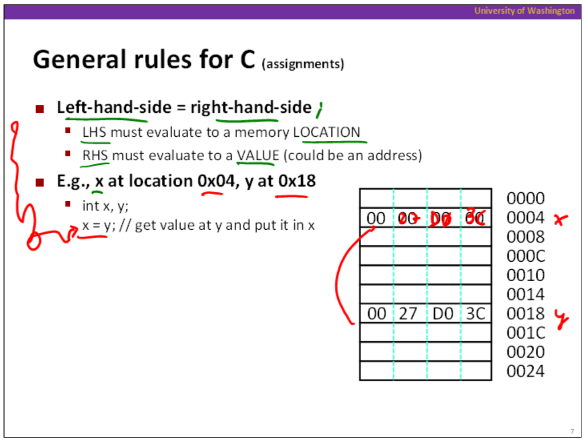
0

12

- **Arrays represent adjacent locations in memory storing the same type of data object**
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000
- **Pointers to arrays point to a certain type of object**
  - E.g., int * array_ptr;
    array_ptr = big_array; 0x00ff0000
    array_ptr = &big_array[0]; 0x00ff0000
    array_ptr = &big_array[3]; 0x00ff000c
    array_ptr = &big_array[0] + 3; 0x00ff000c
    array_ptr = big_array + 3; 0x00ff000c
    *array_ptr = *array_ptr + 1;
    array_ptr = &big_array[130];
  - In general: &big_array[i] is the same as (big_array + i)
    - *which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);*

Arrays

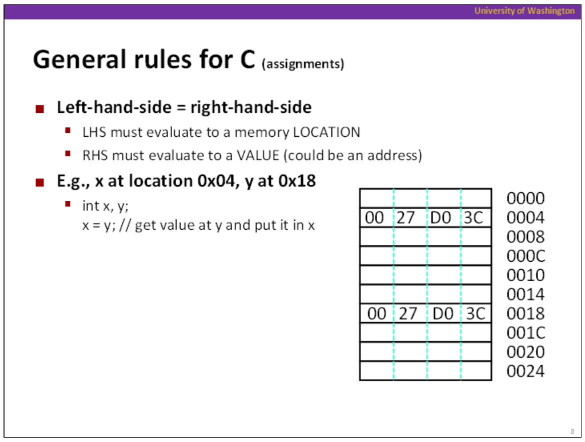
- **Arrays represent adjacent locations in memory storing the same type of data object**
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000
- **Pointers to arrays point to a certain type of object**
  - E.g., int * array_ptr;
    array_ptr = big_array; 0x00ff0000
    array_ptr = &big_array[0]; 0x00ff0000
    array_ptr = &big_array[3]; 0x00ff000c
    array_ptr = &big_array[0] + 3; 0x00ff000c *(adds 3 * size of int)*
    array_ptr = big_array + 3; 0x00ff000c *(adds 3 * size of int)*
    *array_ptr = *array_ptr + 1; 0x00ff000c *(but big_array[3] is incremented)*
    array_ptr = &big_array[130]; 0x00ff0208 *(out of bounds, C doesn't check)*
  - In general: &big_array[i] is the same as (big_array + i)
    - *which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);*



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x

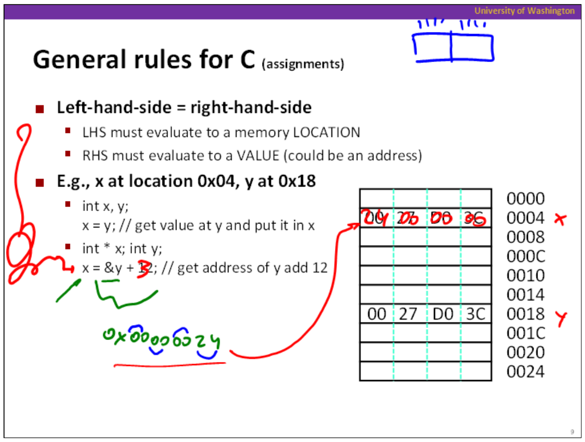
				0000
00	~~00~~ 27	~~D0~~ D0	~~3C~~ 3C	0004 x
				0008
				000C
				0010
				0014
00	27	D0	3C	0018 y
				001C
				0020
				0024



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x

				0000
00	27	D0	3C	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024

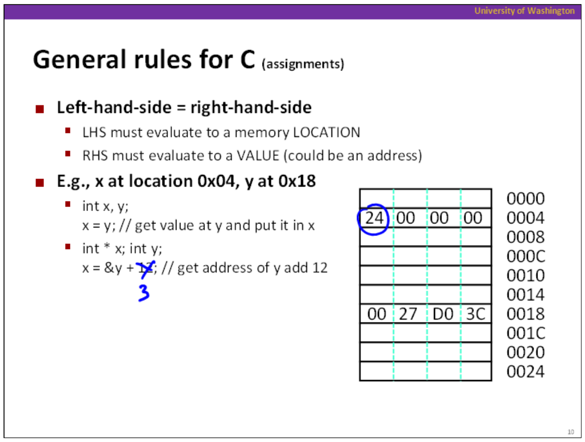


General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + ~~12~~ 3; // get address of y add 12

0x00000024

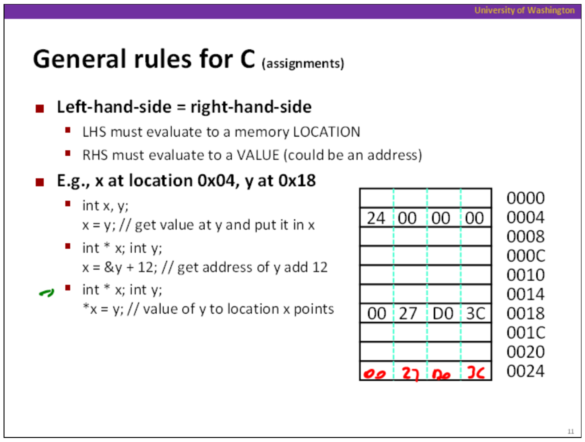
				0000
~~00~~ 24	~~27~~ 00	~~D0~~ 00	~~3C~~ 00	0004 x
				0008
				000C
				0010
				0014
00	27	D0	3C	0018 y
				001C
				0020
				0024



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + ~~12~~; // get address of y add 12
    3

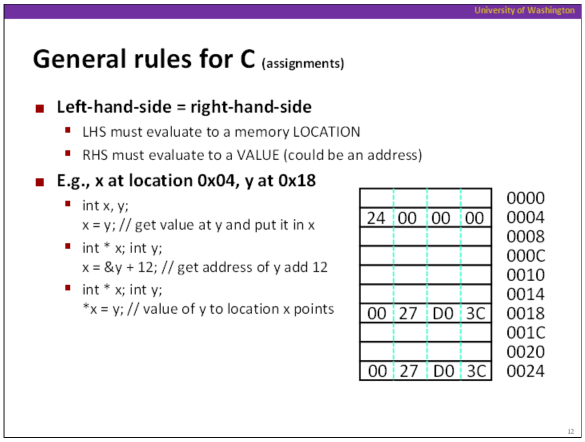
				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
				0024



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 12; // get address of y add 12
  - int * x; int y;
    *x = y; // value of y to location x points

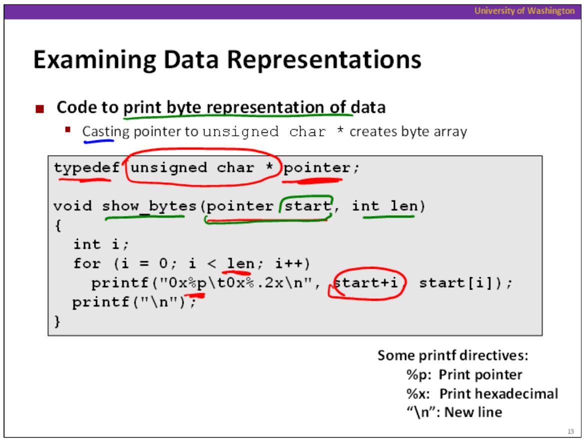
				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
00	27	D0	3C	0024



General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)
- **E.g., x at location 0x04, y at 0x18**
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 12; // get address of y add 12
  - int * x; int y;
    *x = y; // value of y to location x points

				0000
24	00	00	00	0004
				0008
				000C
				0010
				0014
00	27	D0	3C	0018
				001C
				0020
00	27	D0	3C	0024



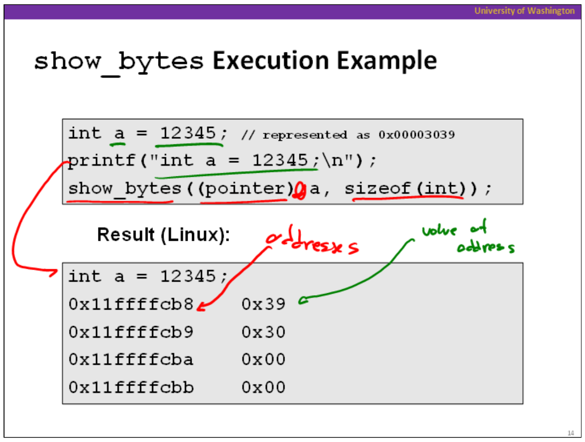
Examining Data Representations

- **Code to print byte representation of data**
  - Casting pointer to `unsigned char *` creates byte array

```
typedef unsigned char * pointer;

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

Some printf directives:
%p: Print pointer
%x: Print hexadecimal
“\n”: New line



`show_bytes` Execution Example

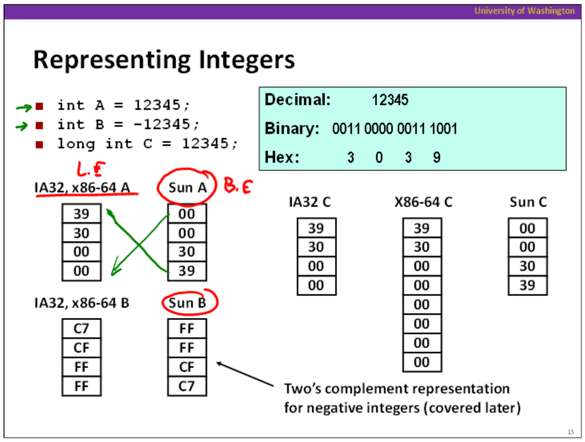
```
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

addresxs

value of address

```
int a = 12345;
0x11ffffcb8     0x39
0x11ffffcb9     0x30
0x11ffffcba     0x00
0x11ffffcbb     0x00
```



Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

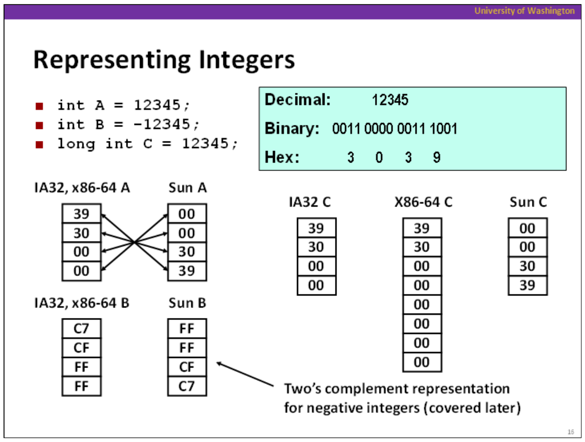
L.E B.E

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

Two’s complement representation for negative integers (covered later)



Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

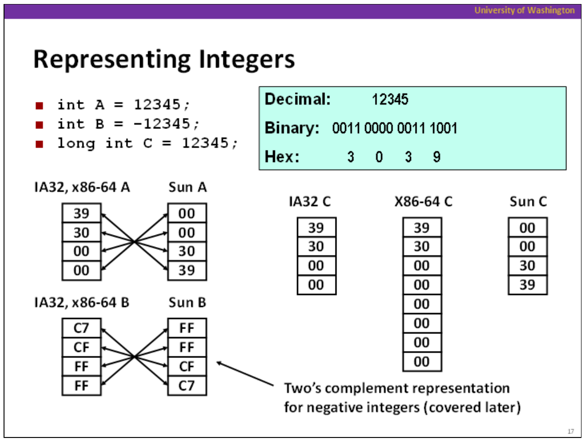
Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

Two’s complement representation for negative integers (covered later)



Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

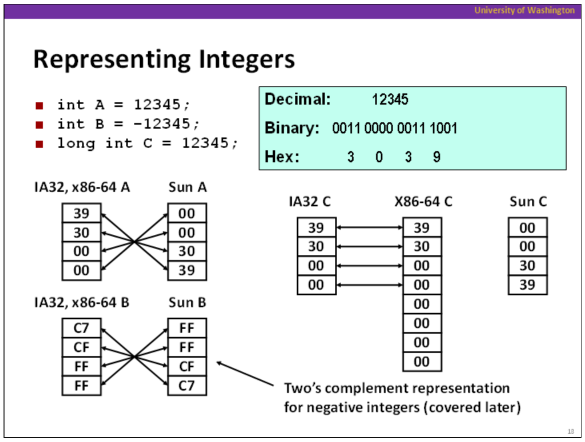
Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

Two’s complement representation for negative integers (covered later)



Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

IA32, x86-64 A	Sun A
39	00
30	00
00	30
00	39

IA32, x86-64 B	Sun B
C7	FF
CF	FF
FF	CF
FF	C7

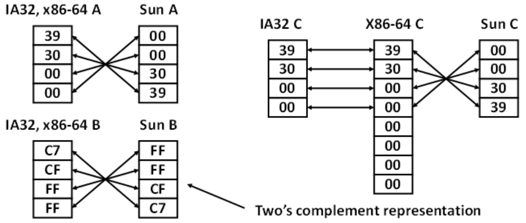
IA32 C	X86-64 C	Sun C
39	39	00
30	30	00
00	00	30
00	00	39
	00	
	00	
	00	
	00	

Two’s complement representation for negative integers (covered later)

Representing Integers

- `int A = 12345;`
- `int B = -12345;`
- `long int C = 12345;`

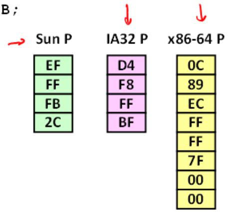
Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9



Two's complement representation
for negative integers (covered later)

Representing Pointers

- `int B = -12345;`
- `int *P = &B;`



[redacted]

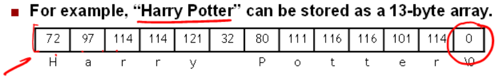
Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

- **For example, "Harry Potter" can be stored as a 13-byte array.**



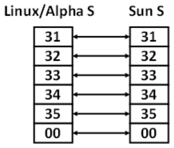
72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- **Why do we put a a 0, or null, at the end of the string?**

- **Computing string length?**

Compatibility

char S[6] = "12345";



- **Byte ordering not an issue**
- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (leading 0 bit) but can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0
  - AND: A&B = 1 when both A is 1 and B is 1
  - OR: A|B = 1 when either A is 1 or B is 1
  - XOR: A^B = 1 when either A is 1 or B is 1, but not both
  - NOT: ~A = 1 when A is 0 and vice-versa
  - DeMorgan's Law: ~(A | B) = ~A & ~B

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

~	
0	1
1	0

General Boolean Algebras

- **Operate on bit vectors**
  - Operations applied bitwise

```
                                  1
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
-> 01000001     01111001      00111100      10101010
```

- **All of the properties of Boolean algebra apply**

```
  01010101
^ 01010101
  00000000
```

- **How does this relate to set operations?**

Representing & Manipulating Sets

- **Representation**
  - Width *w* bit vector represents subsets of {0, …, *w*–1}
  - $a_j = 1$ if $j \in A$

01101001	{ 0, 3, 5, 6 }
76543210	
01010101	{ 0, 2, 4, 6 }
76543210	

- **Operations**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ^, ~ are available in C**
  - Apply to any “integral” data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (char data type)**
  - `~0x41 --> 0xBE`
    `~01000001₂ --> 10111110₂`
  - `~0x00 --> 0xFF`
    `~00000000₂ --> 11111111₂`
  - `0x69 & 0x55 --> 0x41`
    `01101001₂ & 01010101₂ --> 01000001₂`
  - `0x69 | 0x55 --> 0x7D`
    `01101001₂ | 01010101₂ --> 01111101₂`

Contrast: Logic Operations in C

- **Contrast to logical operators**
  - `&&, ||, !`
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**   differen   ~0x41
  - `!0x41 --> 0x00`
  - `!0x00 --> 0x01`
  - `!!0x41 --> 0x01`
    0
  - -> `0x69 && 0x55 --> 0x01`
  - `0x69 || 0x55 --> 0x01`
  - `p && *p++`   (avoids null pointer access, null pointer = 0x00000000
)