Bit Patterns in

C!

```c
#include <stdio.h>

int main()
{
    unsigned int a, b;

    a = 1;
    b = 2;

    printf("a-b: %d...", a - b);

    if (a - b > 0) {
        printf("a > b\n");
    } else if (a - b == 0) {
        printf("a == b\n");
    } else {
        printf("a < b\n");
    }
}
```

Representing Bit Patterns as Literals in C

- To use a bit pattern in C, you can prefix it with **`0b`** or **`0B`**
  - Example: **`0b11100110`**

```c
#include <stdio.h>

int main()
{
    unsigned char b = 0b11100110;
    printf("%d\n", b);
}
```

```
learncli$ gcc -Wall -Wextra -std=c11 \
>         unsigned-bit-pattern.c
learncli$ ./a.out
230
```

A Pair of Hex Digits Encodes a Byte!

Hex is very often how bit pattern data is represented for humans.

Where have you seen examples of it?

- HTML/CSS RGB color codes #FFFFFF
- git commit IDs are hexadecimals

To use hexadecimal in C, prefix the hex vector with 0x or 0X.

- The hex digits are case-insensitive, uppercase preferred

```c
unsigned char h = 0xE6;
printf("%d\n", h);
```

```
learncli$ gcc -Wall -Wextra -g \
>         unsigned-hex-pattern.c
learncli$ ./a.out
230
```

C's Integer Types and Sizes in Memory

Type	Type in stdint.h	Bytes	Signed	Min	Max
`char`	`uint8_t`	1	No	0	255
`signed char`	`int8_t`	1	Yes	-128	127
`unsigned short`	`uint16_t`	2	No	0	65,535
`short`	`int16_t`	2	Yes	-32,768	32,767
`unsigned int`	`uint32_t`	4	No	0	4,294,967,295
`int`	`int32_t`	4	Yes	-2,147,483,648	2,147,483,647
`unsigned long long`	`uint64_t`	8	No	0	18,446,744,073,709,551,615
`long long`	`int64_t`	8	Yes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

C99 Standard added **stdint.h** that which defined the fixed-width types.

C's built-in types can be system dependent (eg, `long` is different on 32-bit vs 64-bit)

Modern best practice to use these types for *portability* purposes.

Data in Memory

```
uint32_t a, b;
a = 1;
b = 2;
```

a	00000000	00000000	00000000	00000001
b	00000000	00000000	00000000	00000010

- One of the compiler's jobs is to handle the bookkeeping of variables names and their addresses in memory.
  - The address of a variable refers to its lowest byte.
  - Consider how tedious it would be to only work in terms of addresses!
- To the left, **a** has address **0** and **b** has address **4**
  - We are illustrating a *little-endian* machine (like your laptop) meaning the least significant bytes are held in lower addresses.

	Address	Contents_2	Contents_{16}	Contents_{10}
	F	00000000	00	0
	E	00000000	00	0
	D	00000000	00	0
	C	00000000	00	0
	B	00000000	00	0
	A	00000000	00	0
	9	00000000	00	0
	8	00000000	00	0
	7	00000000	00	0
	6	00000000	00	0
	5	00000000	00	0
b	4	00000010	02	2
	3	00000000	00	0
	2	00000000	00	0
	1	00000000	00	0
a	0	00000001	01	1

Resolving the Mystery of the Opening Question

```c
#include <stdio.h>

int main()
{
    unsigned int a, b;

    a = 1;
    b = 2;

    printf("a-b: %d...", a - b);

    if (a - b > 0) {
        printf("a > b\n");
    } else if (a - b == 0) {
        printf("a == b\n");
    } else {
        printf("a < b\n");
    }
}
```

a	00000000	00000000	00000000	00000001
b	00000000	00000000	00000000	00000010

Subtractions are recast as addition by negating the right operand.

-b	11111111	11111111	11111111	11111110
a	00000000	00000000	00000000	00000001
a + -b	11111111	11111111	11111111	11111111

`printf`'s %d format specifier interprets (a + -b) as a *signed (two's complement!) integer.* So the output is -1.

In a relational comparison (greater than, less than, etc) the left operand's type is chosen. In this case the left operand is an unsigned integer and 1111...1111 which is greater than 0000...0000.

The exact same bit pattern is being interpreted two different ways in the same program! A nasty bug.

1) Convert 0b11111100100 to hexadecimal

If binary digits *are not* a multiple of 4, add 0s at the front to "pad" until you have a multiple of 4. Then replace groups of 4 with its corresponding hex digit.

2) Convert 0x07B2 to binary

Substitute each digit with its binary representation.

Binary_2	Hexadecimal_{16}	Decimal_{10}
0000	0	00
0001	1	01
0010	2	02
0011	3	03
0100	4	04
0101	5	05
0110	6	06
0111	7	07
1000	8	08
1001	9	09
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15