CMPT 125: Introduction to Computing Science and Programming II

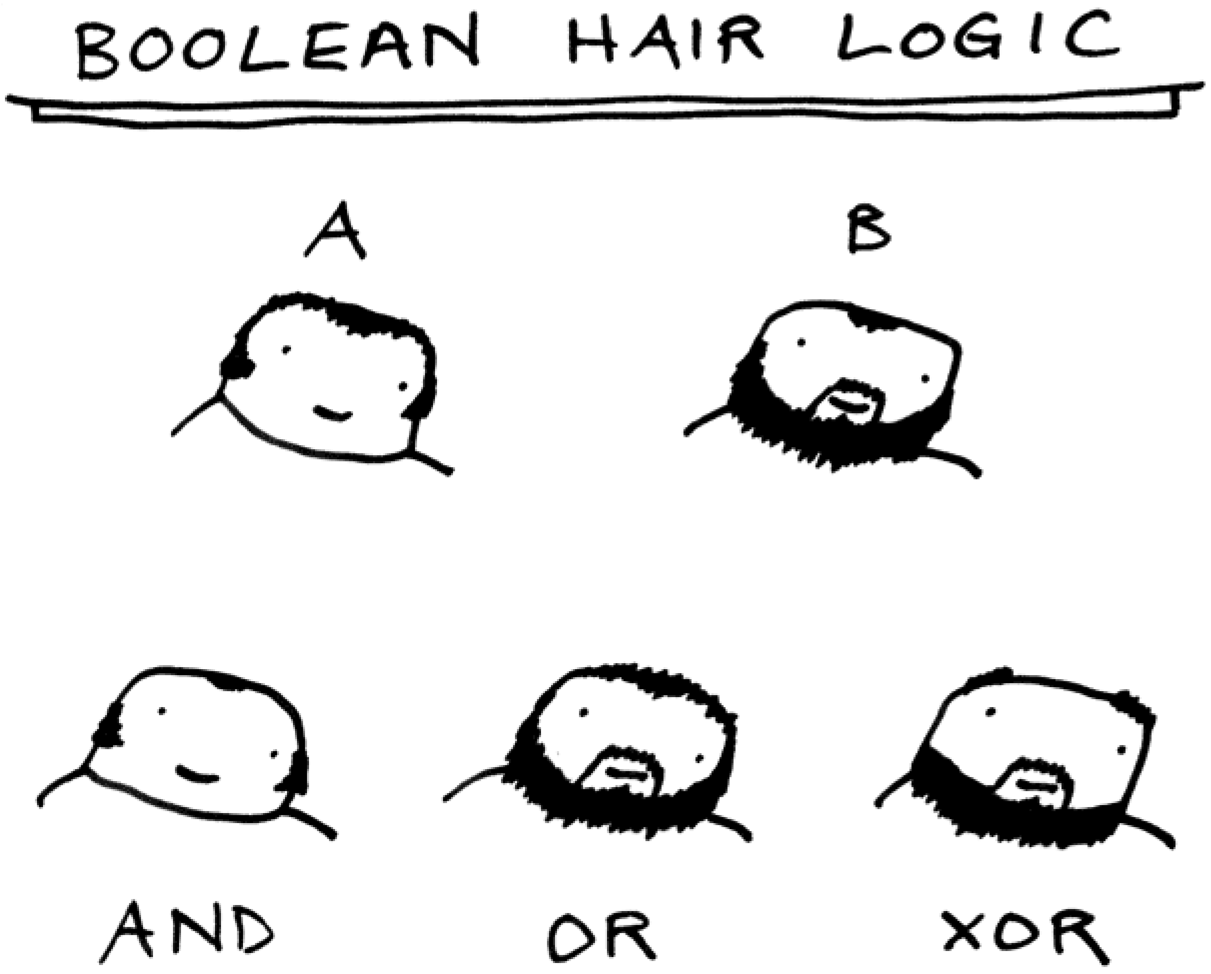
Fall 2023

Week 4: Binary encoding, recursion, algorithms

Instructor: Victor Cheung, PhD

School of Computing Science, Simon Fraser University

Joke of the day



Toothpaste For Dinner.com

Recap from Last Lecture

- Execution stack of functions – each time a function is called an execution block is created "on top" of the calling function, forming a stack
- Functions in other files – allows better organization of code and functionalities
- Preprocessor directives (Macros) – a step taken during the compilation process
- Global/Local/Static variables – affects the availability of variables to a function
- Memory allocation – allows programs to dynamically request memory for space efficiency, need to be careful about leakage

Review from Last Lecture (I)

- Investigate how to use request memory for structs
  - It uses the same syntax as requesting memory for built-in variables

```c
typedef struct {
    unsigned int capacity;
    unsigned int used;
    double* data;
} doubleArrayWithSize;
```

Using a pointer allows a variable array size

```c
//declaring a normal variable
doubleArrayWithSize myArray;

//declaring a pointer to a requested memory
doubleArrayWithSize* dynamicArray = malloc(sizeof(doubleArrayWithSize));
dynamicArray->capacity = 10;
dynamicArray->used = 0;
dynamicArray->data = malloc(sizeof(double) * dynamicArray->capacity);

//...some code

free(dynamicArray->data); //free the fields first
free(dynamicArray); //free the struct last
```

Free the memory when done, careful with the order

Combining Struct Creation with Functions

- General steps: dynamically create a complete struct variable → set it up → return its address

```c
person* create_person() {
  person* p = (person*) malloc(sizeof(person));
  p->name = (char*) malloc(21*sizeof(char));
  printf("print name (up to 20 chars): ");
  scanf("%s", p->name);

  printf("print ID: ");
  scanf("%d", &(p->id));

  return p;
}
```

```c
typedef struct {
  char* name;
  int id;
} person;
```

Inside main()

```c
person* people[2];

people[0] = create_person();
people[1] = create_person();

printf("Person 1 name = %s, id = %d\n", people[0]->name, people[0]->id);
printf("Person 2 name = %s, id = %d\n", people[1]->name, people[1]->id);

for (int i=0; i<2; i++) {
    free(people[i]->name);
    free(people[i]);
}
```

Note the order: inside-outside

Review from Last Lecture (2)

size_t is essentially unsigned long long to guarantee all array elements can be indexed

- Look up other memory allocation functions: calloc, memcpy, memset

```
void* calloc( size_t num, size_t size );
```

  - calloc allocates memory for an array of num objects of size and initializes all bytes in the allocated storage to 0.

```
void* memcpy( void* dest, const void* src, std::size_t count );
```

  - memcpy copies count bytes from src to dest. Both objects are reinterpreted as arrays of unsigned char.

```
void* memset( void* dest, int ch, std::size_t count );
```

  - memset converts the value ch to unsigned char and copies it into each of the first count characters of dest.

- Useful for initializing dynamic memory, be careful with the values and count

Assignment 1

- Read the description file carefully for the questions and submission instructions
- Due on Sep 29, 11:59p, submit to CourSys (link can be found from the Canvas assignment description)
- Remember to test your code by compiling and running your programs at a CSIL machine
    - You can do it remotely or in-person
- DO NOT share your code in any platform (e.g., Piazza, Discord, Canvas, Replit …anywhere)
    - Others might use what you post, our similarity report will catch you, both you and copiers get zero for cheating
    - If you used any help (online reference, peer tutor, …etc.), state them as comment at the top of your files

Assignment 1 Tips

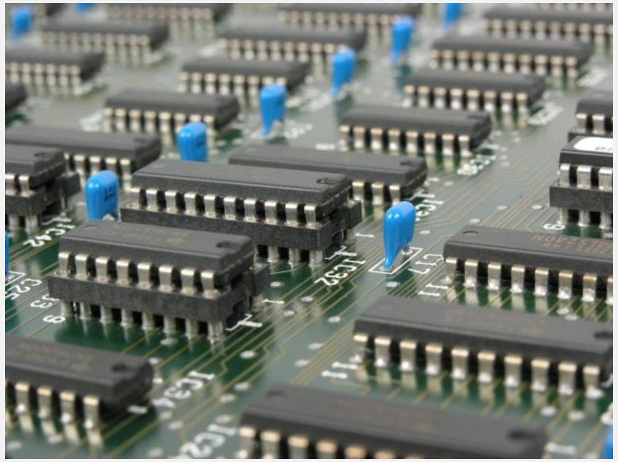
- Do 1 question at a time
  - Start by reading **a1_question1.h** and **test1.c**
  - Create **a1_question1.c** and start with the directive: #include "a1_question1.h", then implement the function there (you should put your entire answer there, as this is the only file you submit for question1)
    - Put your student information and any help used as comments at the top
  - Use the command **make test1** to compile your code to generate the program **test1**
    - Fix any errors/warnings you see, then repeat make test1 until you see the program and run it by **./test1**
  - Modify **test1.c** to have more testcases

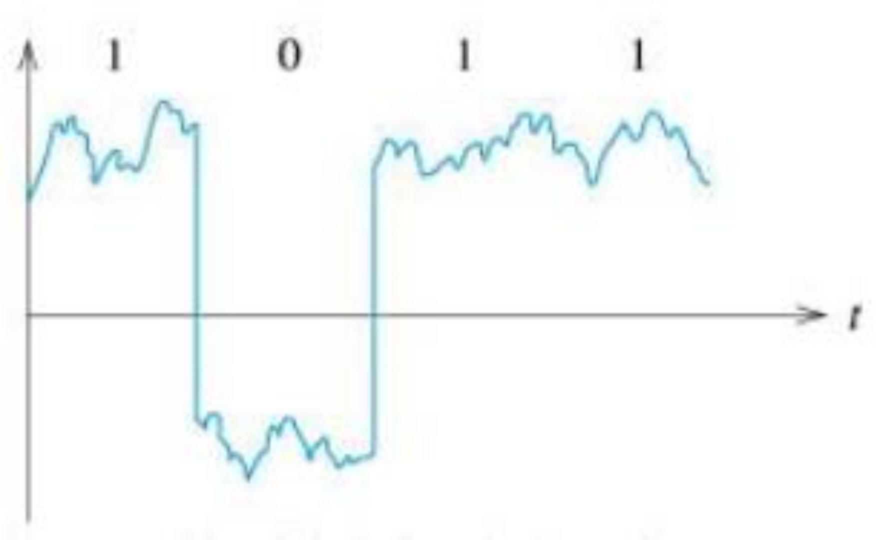
Today

- Binary encoding
  - and decoding
- Pseudo-code & algorithms
  - syntaxes & uses
- Floating-point encoding
  - and decoding

Fundamental Units in Computers

- Fundamentally, digital computers are machines that convert high and low electrical signals into 0's and 1's
  - All data are stored and transmitted in this format (just very quickly and often simultaneously)
  - Because there are only 2 possible signals, data that are represented (encoded) with this format is considered binary encoded





Data in Computers Are Just Sequences of 0's & 1's

This is the "native language" computer speaks, and they are very good at it.

This is what we call binary digits, which have 2 possible values: **0** and **1**

Compare this with decimal digits, which have 10 possible values: **0** to **9**

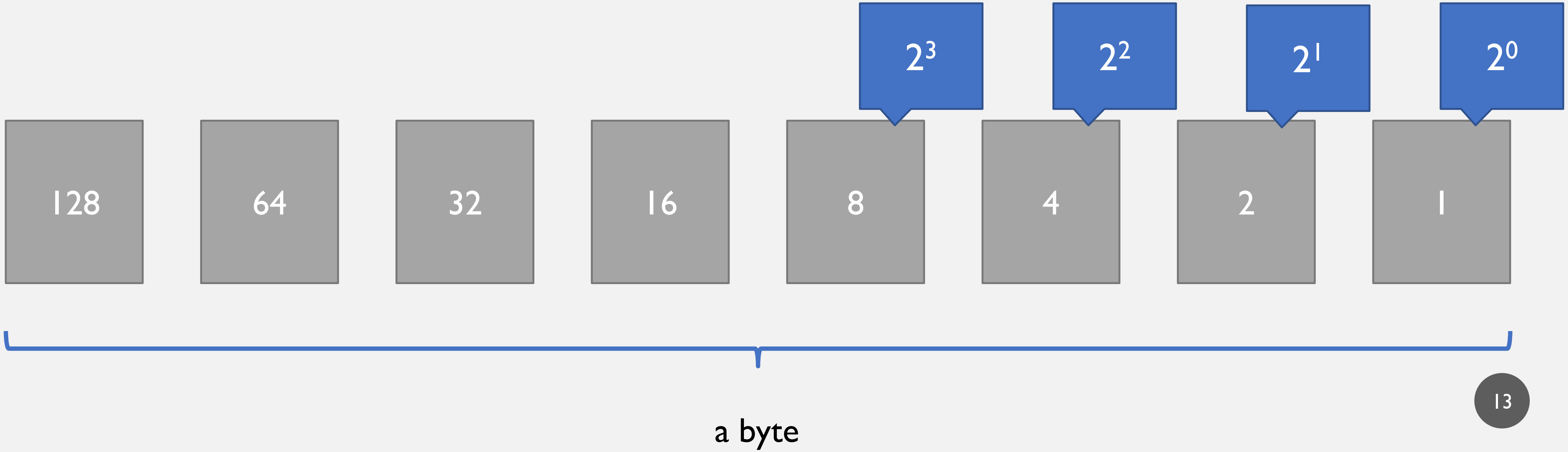
Binary Encoding (1)

- The process of representing a value as a sequence of 0's and 1's

- A **bit** is one value in such sequence (0 or 1), a **byte** is 8 of these values, i.e., 8 bits

Binary Encoding (2)

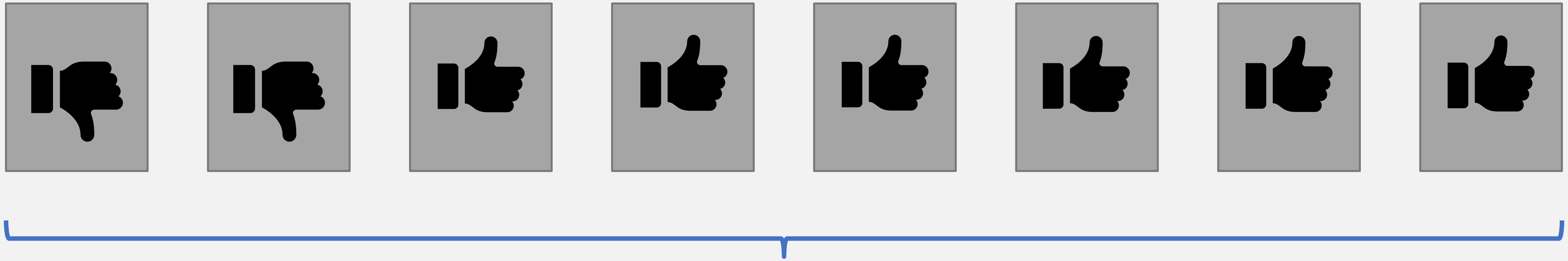
- In binary representation, each bit from right to left represents ascending **powers of 2**: 1, 2, 4, 8, 16, …etc.



Binary Encoding (3)

- When there is a 1 in that position, it means that the value it represents is there

00111111



a byte

Binary Encoding (4)

- When there is a 1 in that position, it means that the value it represents is there, and adding them up gives the value the byte is storing

00111111

32 + 16 + 8 + 4 + 2 + 1 = **63**

Not there	Not there						
128	64	32	16	8	4	2	1

a byte

Most Significant Bit (MSB)

- In our 8-bit example, the largest value represented by a bit is 128
  - We call this bit the "most significant bit" (thus, the smallest value bit is called LSB)
  - This bit can be stored in the lowest memory address, followed by the second largest significant bit, and so on… (our drawings assume memory addresses increase from left to right)
  - Some computer architectures do this in reverse, i.e., the MSB is stored in the highest memory address
- Unless otherwise specified, we assume the MSB is stored in the lowest memory address, and we write it as the leftmost bit (as shown in our previous examples)

MSB & LSB & Size

- Different computers might use different number of bits to represent the same type, e.g., 32-bit vs 64-bit
- This means the same code using int might not work in some computers because the range is different
  - If the number of bits is not enough, only the LSBs will be stored
- To make sure the size is consistent, we can use int32_t, uint32_t, int64_t, uint64_t, …etc.
  - Definitions found at inttypes.h

```
// unsigned int  with 32 bits representation
uint32_t x = 4000000000;

printf("x= %u \n", x); // %u - unsigned int

// warning! number too large for 4 bytes = 32 bits
uint32_t y = 543210987654321;
// too large [00000000,00000001,11101110,00001100,00101001,11110101,00001100,10110001]

printf("y = %llu\n", y);
// the outputs will be 703925425  [00101001,11110101,00001100,10110001]
// only the least significant 4 bytes are kept

uint64_t z = 543210987654321; // ok
printf("z = %llu\n", z);
```

```
x= 4000000000
y = 703925425
z = 543210987654321
```

Practice Exercise

- Convert the following sequence into decimal numbers one byte at a time

001111110000001111100100110001101110010100111111

63

128	64	32	16	8	4	2	1

A Value Can Represent Many Things

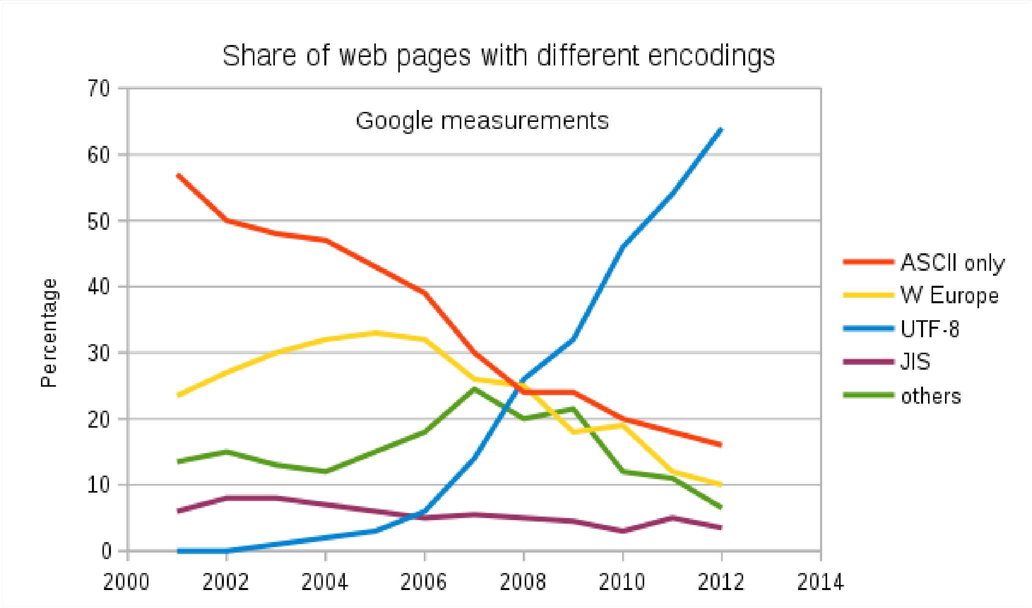
- Given a value (e.g., 63 from the previous slide), it can be used to represent many things, including:
  - the number 63 (as an age, count of something, degree, …etc. in a program)
  - the character/symbol '?' after looking up its position in the ASCII table (https://en.wikipedia.org/wiki/ASCII)
  - an identifier of a network port
  - part of another value (e.g., the G value of an RGB colour)
  - …etc.

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

00111111 → **63** →

About ASCII Code

- In the past ASCII was used in many places, but people soon find out it's not enough
  - We have many languages, characters, and thus encodings



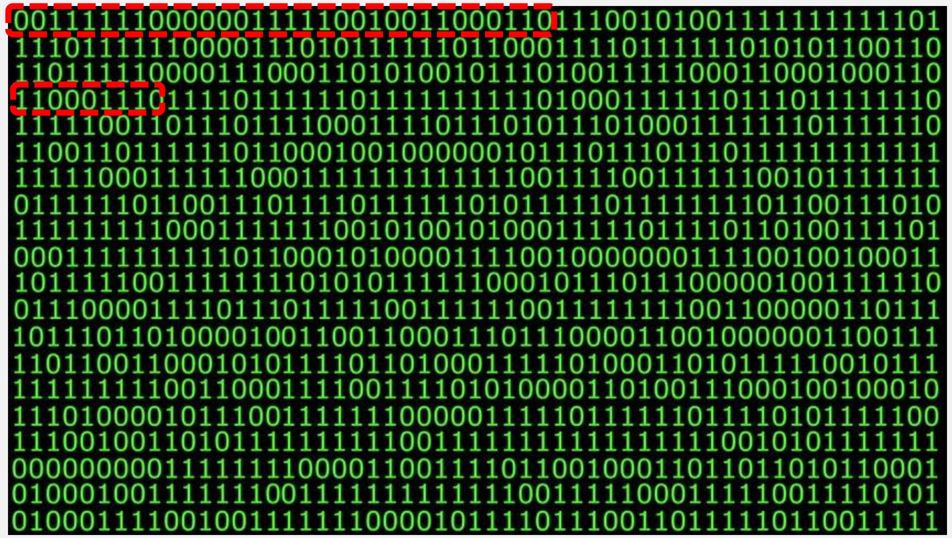
Dec	Char	Dec	Char	Dec	Char
32	SPACE	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49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

Unicode UTF-8

- The maximum number of unique values (i.e., codes) in a byte is 255 (11111111), and ASCII actually uses only 7 bits, so there are only 128 possible values (from 0000000 to 1111111)
- To represent languages other than English, we may need a lot more than 255 types of characters
- **Unicode** uses up to **4 bytes** to represent more characters

?

?



Convert/Translate between Two Systems

- By representing a number using another set of values, we are converting it between different numerical systems
  - what we did was decoding from the binary system (00111111) to the decimal system (63)
  - we can also encode from the decimal system (63) to the binary system (00111111)
- How do we encode a number from the decimal system to the binary system?
  - Let's use 29 as an example
    - do a number of divisions, keep track of the remainders
    - when no more divisions can be done, read the remainders bottom-up: 11101
    - in other words, set the 2^i-th digit to 0 or 1 with increasing i, based on even/odd

29
14 - 1
7 - 0
3 - 1
1 - 1

Pseudo-Code

- A way to systematically describe the sequence of steps to solve a problem (usually computational)
- Similar to code, but typically uncompilable due to lack of syntax rules & variable declarations
  - high-level description of the steps (aka algorithm)
  - contains essential details needed to implement the steps
  - language independent (uses elements common to most languages, e.g., loops, if-else, assignment, comparison)
    - no syntax rules, but is consistent and readable by humans
    - no language specific elements, like type, memory allocation, …etc.

- Pseudo-code makes description of algorithms shorter and easier to understand (we'll use it sometimes)

Describing Steps Systematically

- How do we describe the sequence of steps for binary encoding with pseudo-code? Consider this:

Step 1: i = 0

Step 2: while (N > 0)

  Step 2.1: if N is even

    Step 2.1.1: set the 2^i-th digit to 0

    Step 2.1.2: set N to N/2

  Step 2.2: else

    Step 2.2.1: set the 2^i-th digit to 1

    Step 2.2.2: set N to (N-1)/2

  Step 2.3: i++

Can you convert the pseudo-code to C code?

i = 0: N = 29 is odd
set $1*2^0$ → ****1
set N = (29-1)/2 = 14

i = 1: N = 14 is even
set $0*2^1$ → ***01
set N = 14/2 = 7

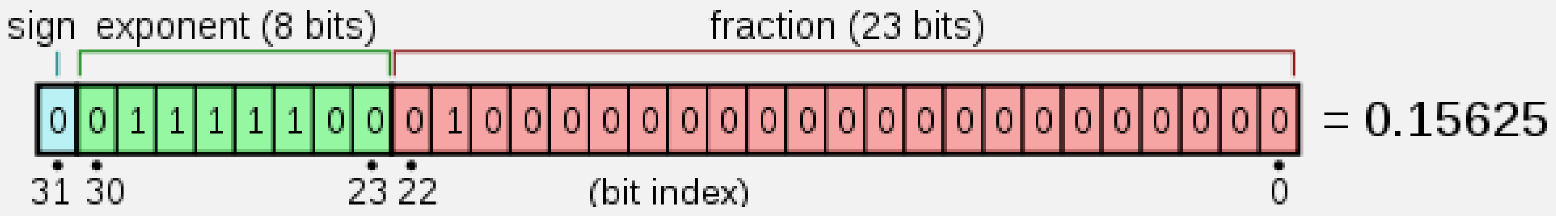
i = 2: N = 7 is odd
set $1*2^2$ → **101
set N = (7-1)/2 = 3

i = 3: N = 3 is odd
set $1*2^3$ → *1101
set N = (3-1)/2 = 1

i = 4: N = 1 is odd
set $1*2^4$ → 11101
set N = (1-1)/2 = 0

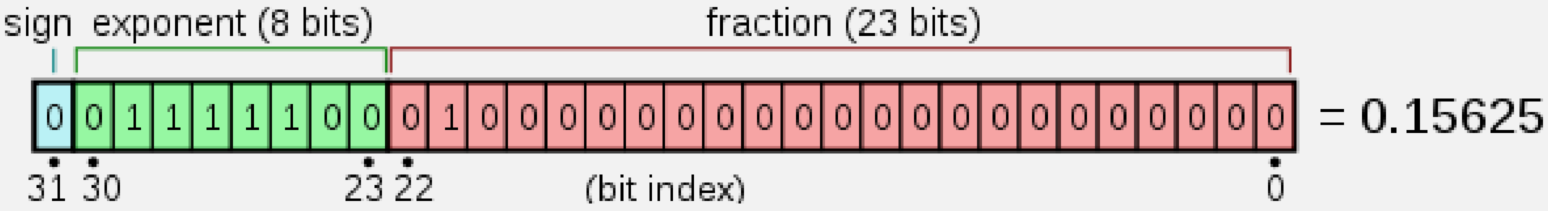
What about Non-Integers?

- With our previous encoding technique, we can only represent integers, how about 0.5? or 1/3?
- We use a total of 32 bits differently to represent numbers with decimal places (in scientific notation)



- 1 bit for sign (0 means positive, 1 means negative)
- 23 bits for the significand (significant digits of the number)
  - $1.b_{22}b_{21}\ldots b_0$ (b_{22} represents the digit of ½, b_{21} represents the digit of ¼ …)
- 8 bits for the exponent – ranges from -127 to 128

Scientific Notation of Numbers with Decimal Places



$$value = (-1)^{b_{31}} \times (1 + fraction) \times 2^{exponent-127}$$

- value = $(-1)^0 \times (1 + \frac{1}{4}) \times 2^{124-127} = 1.25 \times 2^{-3} = 0.15625$

- Range of the representation: $1.b_{22}b_{21}\ldots b_0 \times 2^{exponent-127}$
  - smallest number happens when all fraction & exponent bits are 0: $1 \times 2^{-127} = 5.9 \times 10^{-39}$
  - largest number happens when all fraction & exponent bits are 1: $(2-2^{-23}) \times 2^{128} = 6.8 \times 10^{38}$

Floating Point Encoding

- Example: -3.625
- Step 1: split the number at the decimal point, take note of the sign
- Step 2: apply the binary encoding for integers on the integral part (3 → 11)
- Step 3: apply another version of binary encoding for fraction part (0.625 → 0.101):
  - 0.625 * 2 = 1.25 → 1
  - 0.25 * 2 = 0.5 → 0
  - 0.5 * 2 = 1 (stop when there is no more fraction)
  - Get the binary representation of the decimal part by reading from top to bottom: 0.<u>101</u> (it means 0.625 = 1/2+1/8)
- Step 4: combine the integral part and fraction part (11.101) and normalize it: 1.1101×2^1
- Step 5: apply the binary encoding for the exponent part (exponent – 127 = 1 → exponent = 128)

-3.625 →

1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Another Example of Floating Point Encoding

- Value: 4.1
- Step 1: split the number at the decimal point, take note of the sign
- Step 2: apply the binary encoding for integers on the integral part (4 → 100)
- Step 3: apply another version of binary encoding for fraction part (0.1 → 0.0 0011 0011 0011 0011…)
  - Sometimes you get recurring fractions!
- Step 4: combine the integral part and fraction part (100.0 0011 0011 0011 0011 …) and normalize it: 1.00 0 0011 0011 0011 0011 … x 2^2
- Step 5: apply the binary encoding for the exponent part (exponent – 127 = 2 → exponent = 129)
- What is the answer?

Today's Review

- Binary encoding
  - Using only 1's & 0's to represent values
  - conversion between decimal & binary representations (there are others)
- Pseudo-code & algorithms
  - High-level description of an algorithm, language independent, show important steps in a consistent way
- Floating-point encoding (32-bit)
  - Using only 1's & 0's to represent fractional values
  - sign (1-bit), exponent (8-bit), fraction (23-bit)

Homework!

- For practice, perform binary encoding for 1.625, 42, 42.6875, -9.9
- Investigate how double is different from float in C
- Download code files W04-01_Example01.c, W04-01_Example02.c, W04-01_Example03.c from Canvas and run them, take a look at the output and explain them
- Read this for steps & proofs for fraction conversions: https://indepth.dev/posts/1019/the-simple-math-behind-decimal-binary-conversion-algorithms