CSc 28
Discrete Structures

Chapter 8
Number Representation

Herbert G. Mayer, CSU CSC
Status 3/10/2021

Syllabus

- Binary Numbers
- Number Conversion
- Bit Operations
- Logic Operations
- Number Conversion
- Floating Point Numbers
- Data in Memory
- Quick Excursion to C++
- Arrays in C++

Binary Representation

Binary Numbers

- Binary Digit –AKA Bit-- is the smallest unit of computation on most digital computers
- Bit has two states:

  0 represents 0 Volt [V], or ground; used for logical False, or for numeric 0

  1 represents positive Voltage [+V]; used for logical True, or for numeric 1
- Computer word consists of multiple bits, typically 32, 60, or 64 bits today
- Often words are composed of bytes, units of 8 bits that are addressable as one unit: byte-addressable
- General binary number, just like in decimal system:

$$\underset{\substack{\uparrow\\ \text{MSB}}}{b_{n-1}}b_{n-2}\ldots b_1\underset{\substack{\uparrow\\ \text{LSB}}}{b_0} = b_{n-1}\times 2^{n-1} + b_{n-2}\times 2^{n-2} + \ldots + b_1\times 2^1 + b_0\times 2^0$$

Binary Numbers Using TCR

- **Possible representations of binary numbers: *sign-magnitude (sm), one's complement (ocr)*, and *two's complement representation (tcr)***
- **Advantage of *tcr:* machine needs no subtract unit, the single adder is sufficient**
- **When subtraction is needed, just add a negative number**
- **To create a negative number: invert the positive! See inverter later**
- **Also there is no need for signed- *and* unsigned arithmetic; unsigned is sufficient**
- **C and C++ allow signed & unsigned integers. In fact, arithmetic units using *tcr* can ignore the sign bit**
- ***Tcr* just needs an adder and an inversion unit**

Binary Numbers Using TCR

- Binary numbers in *tcr* use a sign bit and a fixed number of bits for the magnitude
- For example, on an old PC you may have 32-bit integers, one for the sign, 31 for the magnitude
- Typically today your PC has 64-bit integers
- When processed in a *tcr* architecture, the most significant bit is the *sign bit*, the other 63 bits hold the actual signed value of interest
- By convention, a sign bit value of 0 stands for positive and 1 for negative numbers

Binary Numbers Using TCR

- **Invert +:** To create a negative number from a positive in *tcr*, start with the binary representation for the positive one, invert all bits, ***and add 1***
- Note that **overflow cannot happen by inversion** alone: the inverse (a negative value) of all representable, positive numbers can always be created
- But: there is **one more negative number** than positive ones in *tcr*
- TCR has **one single 0**, i.e. no negative 0 in *tcr*, as in *one's complement* and *sign-magnitude*

Binary Numbers Using TCR

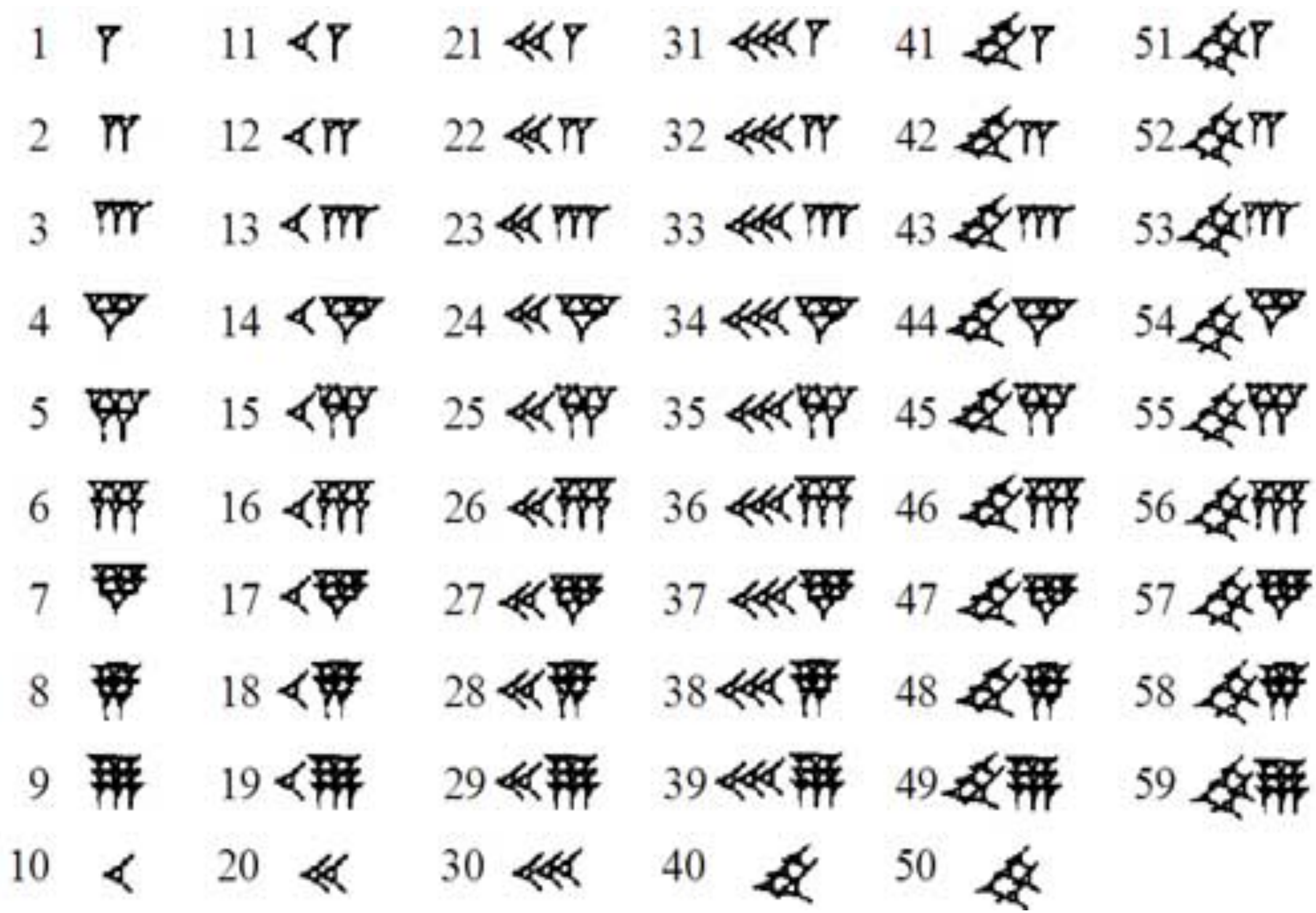
- Invert -: To invert a negative number, complement all bits of the original negative number and *add a 1*
- Ditto with inverting a negative crafting positive number: Important to add a 1 again, not to subtract it!
- However, there will be one negative value, whose positive inverse cannot be represented; it will cause overflow instead when inverted!
- That value is the smallest, negative number. For example, an 8-bit signed *tcr* integer can hold integers in the range from -128 .. 127. See the asymmetry? See the one negative value that cannot be inverted?
- On a 32-bit architecture, the range is from -2,147,483,648 to +2,147,483,647. See the slight asymmetry?

Arithmetic in Babylon

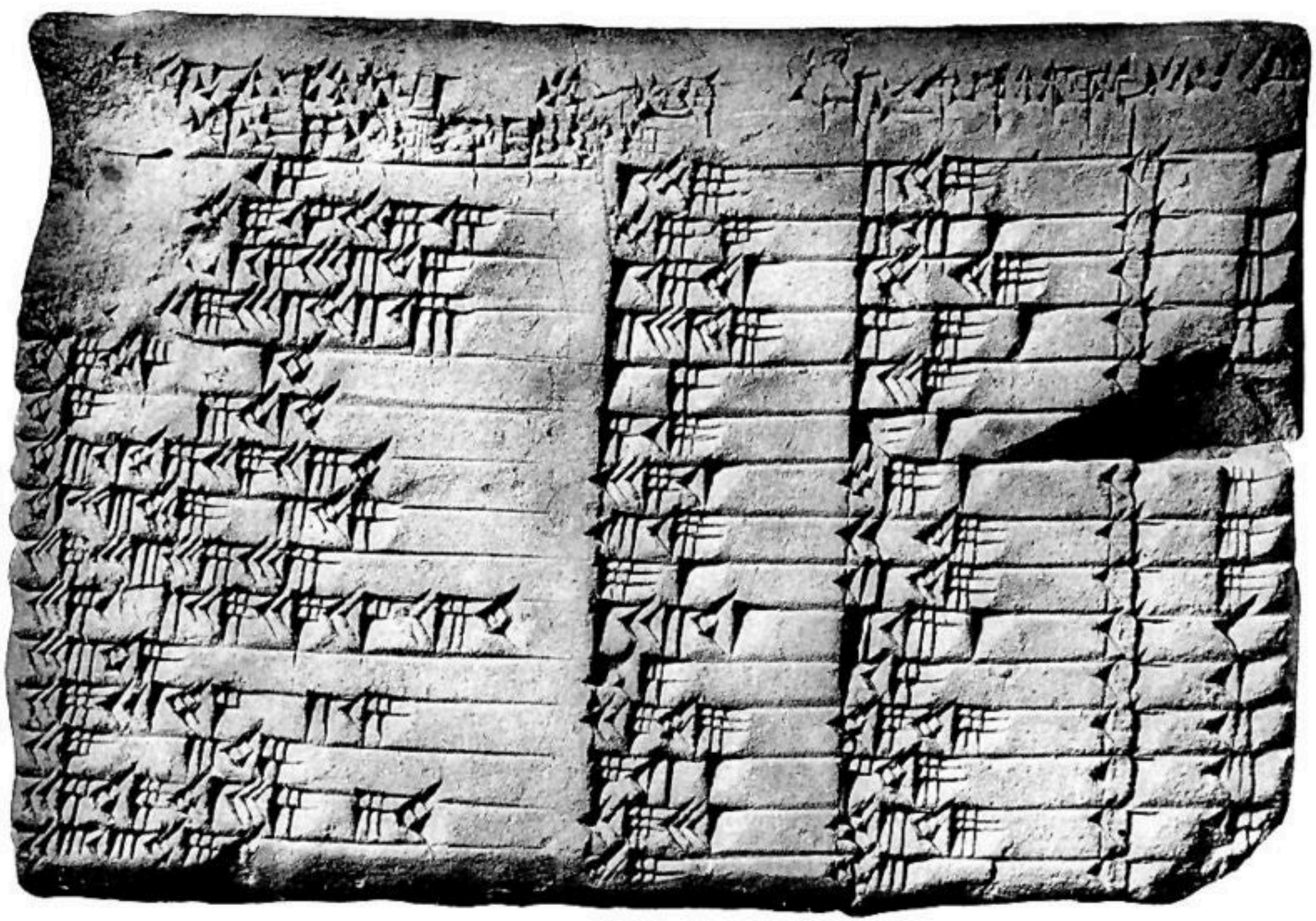
Babylonian Numbers

- Babylonian Number system used base 60
- Hence required lots of digits, close to 60 digits in all
- Otherwise arithmetic was similar to our decimal system
- Yet the notion of a zero value was not clearly worked out
- Hence the phrase “close to 60 digits”
- A 0 digit would have been handy ☺

Close to 60 Babylonian Digits



Babylonian Numbers



Binary Numbers Cont'd

Bitwise Ops

Number of States n Bits

n	# States	*n*	# States	*n*	# States
0	none	10	1,024	20	1,048,576
1	2	11	2,048	21	2,097,152
2	4	12	4,096	22	4,194,304
3	8	13	8,192	23	8,388,608
4	16	14	16,384	24	16,777,216
5	32	15	32,768	:	:
6	64	16	65,536	30	1,073,741,824
7	128	17	13,1072	31	2,147,483,648
8	256	18	26,2144	32	4,294,967,296
9	512	19	52,4288	64	18446744073709551616

Binary Operations

A	~A
0	1
1	0

Complement

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise XOR

A	B	A + B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Addition

Convert Decimal to Binary

- **Divide n_{10} repeatedly by 2; sample here 500_{10}**
- **Store the remainder each time: will be 0 or 1**
- **Until the number n reaches 0**
- **Then list all remainder bits in the reverse order**

Decimal	Quotient	Remainder
500 / 2 =	250	0
250 / 2 =	125	0
125 / 2 =	62	1
62 / 2 =	31	0
31 / 2 =	15	1
15 / 2 =	7	1
7 / 2 =	3	1
3 / 2 =	1	1
1 / 2 =	0	1

- **Here:** $500_{10} = 111110100_2$

Exercise: Convert to Binary

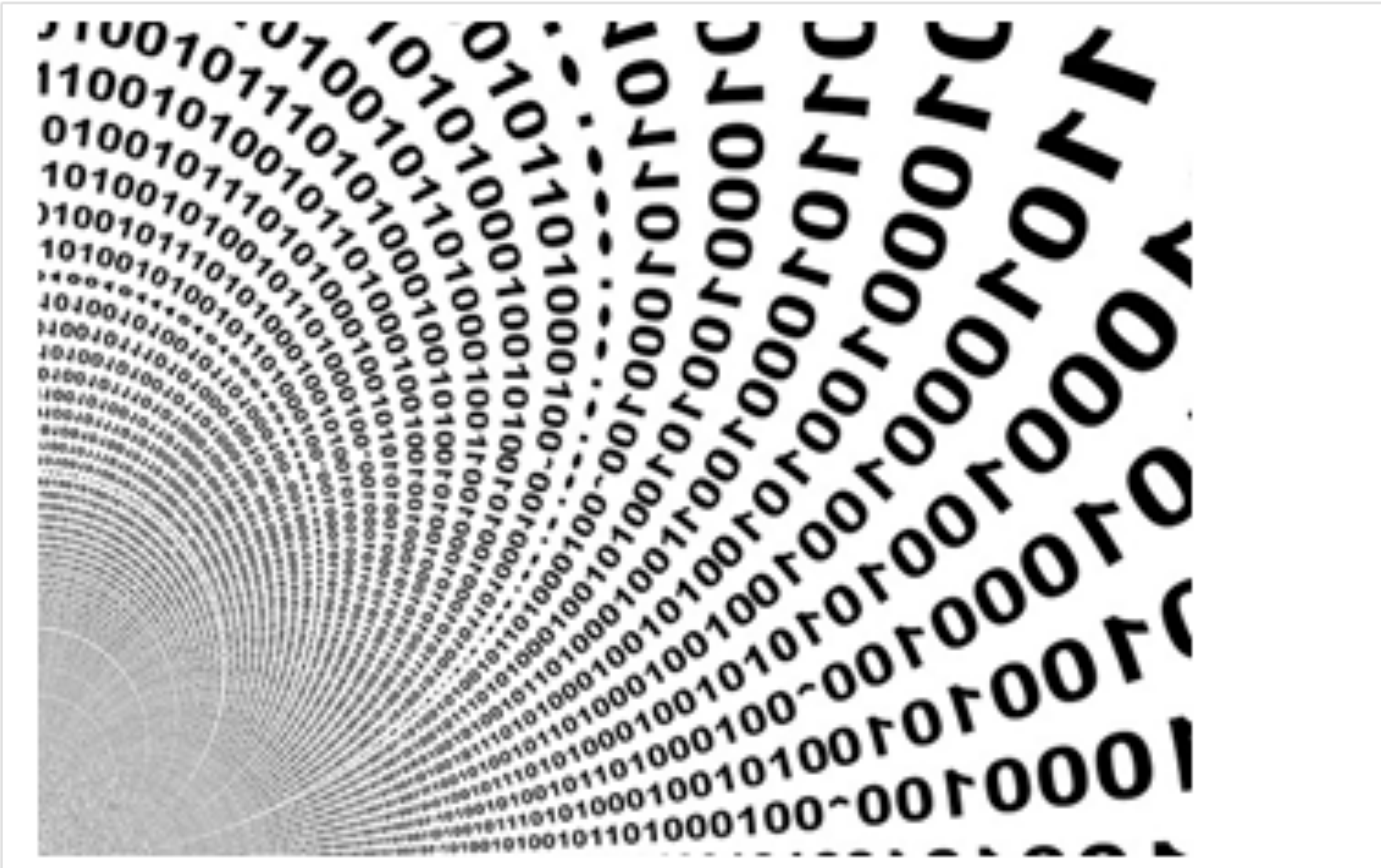
You Exercise: Conversion to Binary, Sample 678_{10}

Decimal	Quotient	Remainder
678 / 2 =	339	0
		. . .
		. . .

Sample Binary 8-bit Numbers, tcr

Decimal Value	S	Binary Digits (Bits)						
2	0	0	0	0	0	0	1	0
16	0	0	0	1	0	0	0	0
21	0	0	0	1	0	1	0	1
31	0	0	0	1	1	1	1	1
64	0	1	0	0	0	0	0	0
100	0	1	1	0	0	1	0	0
largest positive: 127	0	1	1	1	1	1	1	1
-1	1	1	1	1	1	1	1	1
-2	1	1	1	1	1	1	1	0
-16	1	1	1	1	0	0	0	0
-21	1	1	1	0	1	0	1	1
-31	1	1	1	0	0	0	0	1
-64	1	1	0	0	0	0	0	0
-100	1	0	0	1	1	1	0	0
almost smallest ☺ -127	1	0	0	0	0	0	0	1

Binary Numbers



Two’s Complement Binary, Negative

Generate Negative Two’s Complement Binary Number: Sample -100_{10}

Decimal Value	S	Binary Digits (Bits)						
100	0	1	1	0	0	1	0	0
~100	1	0	0	1	1	0	1	1
+1								1
-100	1	0	0	1	1	1	0	0

Tcr Arithmetic, 8 bits

Arithmetic Operations on *TCR* 8-bit Binary Numbers: $99_{10}+19_{10}$

	S	Binary Digits (Bits)						
99	0	1	1	0	0	0	1	1
19	0	0	0	1	0	0	1	1
99+19 = 118	0	1	1	1	0	1	1	0

Tcr Arithmetic, Adding, Subtracting

Adding, Subtracting *TCR* Binary Numbers: $\pm\ 7_{10}$, $\pm\ 9_{10}$

9+7 = 16	S	Binary Digits						
9	0	0	0	0	1	0	0	1
7	0	0	0	0	0	1	1	1
9+7 = 16	0	0	0	1	0	0	0	0

9-7 = 9+(-7)	S	Binary Digits						
7	0	0	0	0	0	1	1	1
~7	1	1	1	1	1	0	0	0
+1								1
~7 + 1 = -7	1	1	1	1	1	0	0	1
9	0	0	0	0	1	0	0	1
9+(-7) = 2	0	0	0	0	0	0	1	0

Note that carry-bit (1) into sign = carry-bit out (1)

Tcr Arithmetic, Adding, Subtracting

7-9 = 7+(-9)	S	Binary Digits						
9	0	0	0	0	1	0	0	1
~9	1	1	1	1	0	1	1	0
+1								1
−9	1	1	1	1	0	1	1	1
7	0	0	0	0	0	1	1	1
7+(−9) = −2	1	1	1	1	1	1	1	0

There was no carry-bit into or out of sign bit

-9-7 = -9+(-7)	S	Binary Digits						
7	0	0	0	0	0	1	1	1
~7	1	1	1	1	1	0	0	0
+1								1
~7 + 1 = −7	1	1	1	1	1	0	0	1
−9	1	1	1	1	0	1	1	1
−9−7 = −16	1	1	1	1	0	0	0	0

Carry-bit (1) into sign = carry-bit out (1): No O

Adding, Subtracting *TCR* 8-bit Binary Numbers: ± 100_{10}, ± 64_{10}

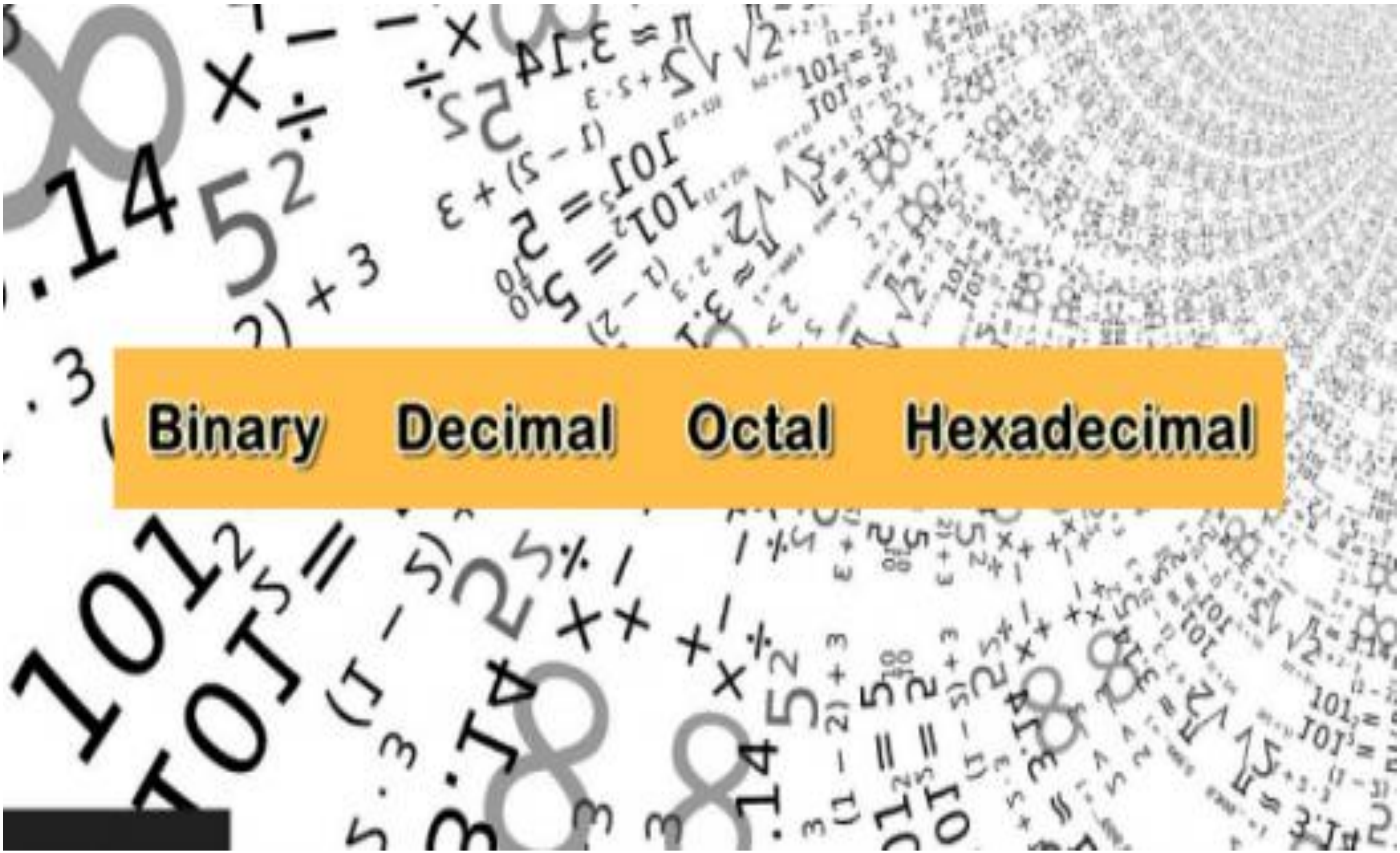
Tcr Arithmetic, 8 bits, Overflowing?

100+64=164?	S	Binary Digits						
100	0	1	1	0	0	1	0	0
64	0	1	0	0	0	0	0	0
128-164=-92	1	0	1	0	0	1	0	0

-100-64=-164?	S	Binary Digits						
-100	1	0	0	1	1	1	0	0
-64	1	1	0	0	0	0	0	0
-128+164 =92	0	1	0	1	1	1	0	0

Hexadecimal Numbers

Hexadecimal Numbers



Hexadecimal Numbers

- **Hexadecimal (hex) uses base 16; not 10 (decimal), not 8 (octal), not 2 (binary), just 16, no magic ☺**
- **Needs 16 different digits: 0..9 purely by convention, plus the digits a . . f; or equivalently A . . F**
- **Symbol a stands for a hex digit with value 10_{10}, while the symbol f, AKA digit f, stands for the value 15_{10}**
- **Programming tools are not picky, and allow the 6 extra digits to be lower- as well as uppercase letter**
- **Here are a few hex numbers and their equivalent decimal values:**

Hexadecimal Numbers

Decimal	Hexadecimal
0	0
1	1
9	9
10	a
11	b
15	f
16	10
33	21
127	7f
128	80
129	81
255	ff
256	100

Decimal	Hexadecimal
257	101
258	102
300	12c
16,383	3fff
16,384	4000
16,385	4001
32,767	7fff
32,768	8000
32,769	8001
65,535	ffff
65,536	1,0000
65,637	1,0001
4,294,967,295	ffff,ffff

Adding Hexadecimal Numbers

Adding Hexadecimal Numbers, $af_{16} + 65_{16}$, $10a42_{16} + 5be_{16}$

			a	f
			6	5
		1	1	4

1	0	a	4	2
		5	b	e
1	1	0	0	0

Logical Operators

Logic operations are done one bit at a time, monadic (AKA unary) or on a pair of bits, dyadic (AKA binary)

Example:

~1011 = 0100	Complement
1010 & 1100 = 1000	Bitwise AND
1010 | 1100 = 1110	Bitwise OR
1010 ^ 1100 = 0110	Bitwise XOR

Other Bases

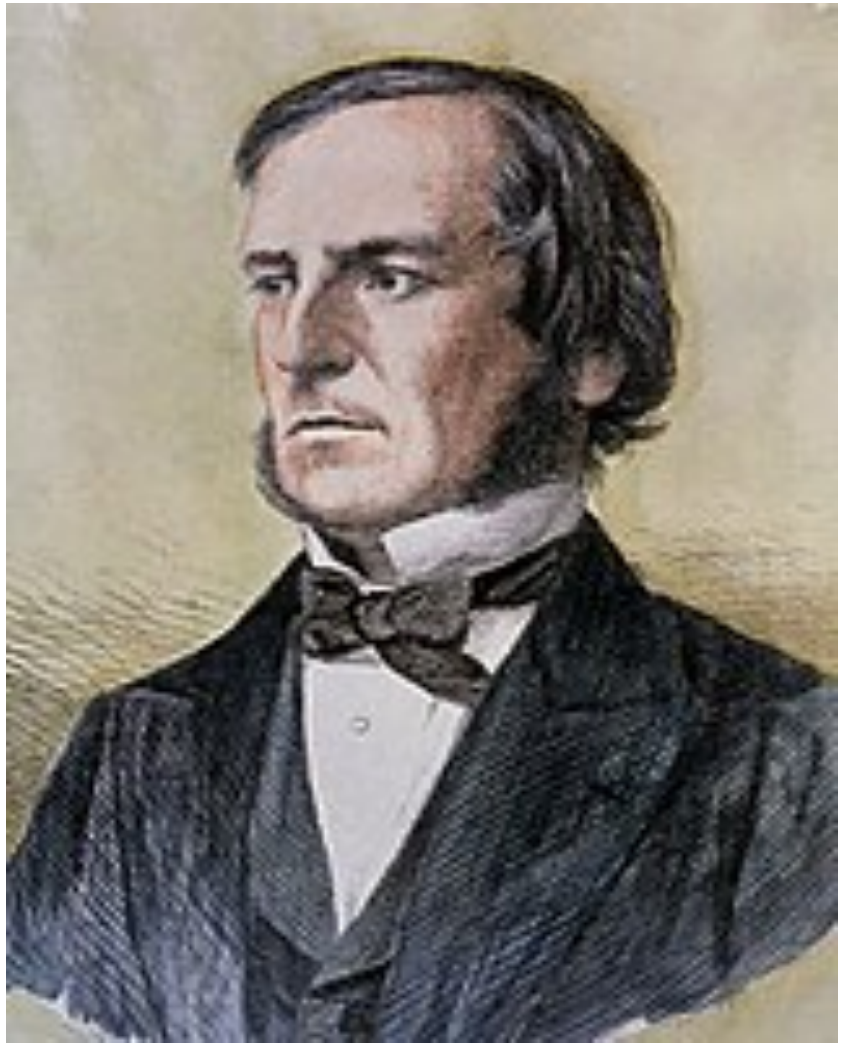
Octal base, AKA 8 → with digits 0 .. 7

Hexadecimal base, AKA 16 → with digits 0 .. 9, A, B, C, D, E, F with A or a representing 10_{10}, and F or f for 15_{10}

4-bit positive integer conversion table

Dec	Bin	Oct	Hex	Dec	Bin	Oct	Hex
0	0000	0	0	**8**	1000	10	8
1	0001	1	1	**9**	1001	11	9
2	0010	2	2	**10**	1010	12	A
3	0011	3	3	**11**	1011	13	B
4	0100	4	4	**12**	1100	14	C
5	0101	5	5	**13**	1101	15	D
6	0110	6	6	**14**	1110	16	E
7	0111	7	7	**15**	1111	17	F

George Boole



British Mathematician George Boole 1815 - 1854

Base Conversion

- Converting from binary to its equivalent hex:
  1) Group binary value into 4-bit sequences (groups)
  2) Replace each group by its hex value

  Example:

  $44100_{10} = 1010\ 1100\ 0100\ 0100_2 = AC44_{16}$

- Converting from hex to its equivalent binary:
  Replace each hex digit by corresponding 4-bit value

  Example:

  $27411_{10} = 6B13_{16} = 0110101100010011_2$

Floating Point

FP Format

- **God created integers, man invented floating points ☺**
- **Floating point used to express real-valued numbers**
- **Have implied base 10, an assumed decimal point, and have integer part as well as fractional part**

Examples 1:

5 2.0 3.1415 –634.9 999.

Example 2: In scientific notation, implied base is: 10

mantissa

exponent

needs second sign?

$$-6.349 \times 10^{2}$$

sign

base

FP Format

- **Binary code used to represent floating point values, but generally just an approximation: NOT exact!**
- **IEEE 754 single-precision standard; ∃ other precisions**

$$\mathbf{s\ e_1e_2\ldots e_8\ b_1b_2\ldots b_{23}}$$

1 bit	*8 bits*	*23 bits*
Sign $0 \rightarrow +$ $1 \rightarrow -$	Interpreted as unsigned integer **e'**	Interpreted as a base 2 value defined as **m'** $= 0.b_1b_2\ldots b_{23} = b_12^{-1} + b_22^{-2} + \ldots + b_{23}2^{-23}$

if integer **e'** $\neq 0$ then FP number $= (-1)^s \times (1 + m') \times 2^{e'-127}$

Else, if **e'** $= 0$ then FP number $= (-1)^s \times m' \times 2^{-126}$

FP Format

- **Example: IEEE 754 single precision (32-bit)**

01010110010010100000000000000000

s = 0

$e' = 172_{10}$

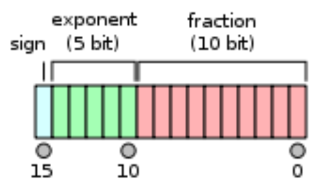
$m' = 2^{-1} + 2^{-4} + 2^{-6} = 0.578125$

$$\text{Number} = (-1)^{s} \times (1 + m') \times 2^{e'-127} = 1.578125 \times 2^{45} \approx 5.55253372027 \times 10^{13}$$

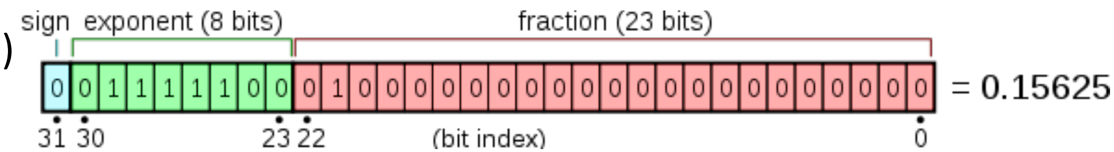
- **As more bits are available on 64-bit architecture, the more precise the mantissa and the larger the exponent range!**

Other IEEE 754 Formats

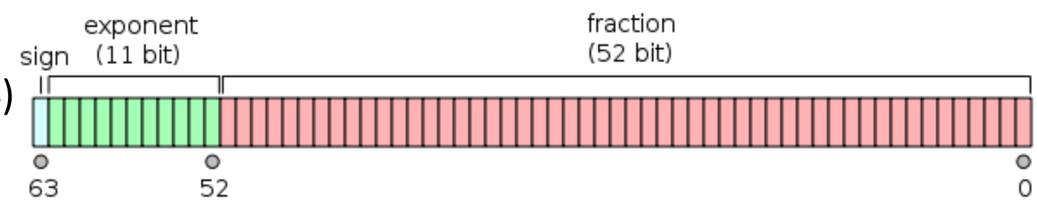
Half precision (binary16)



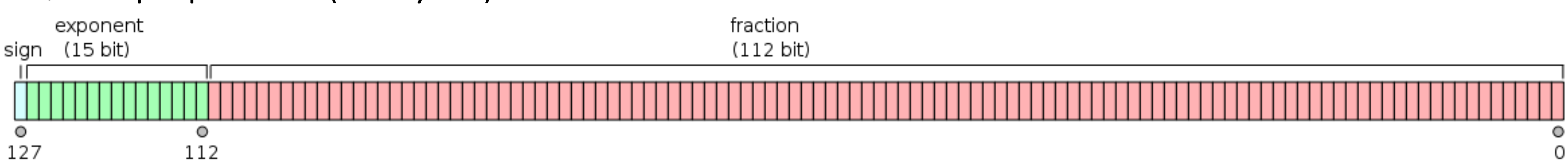
Single precision (binary32)



Double precision (binary64)



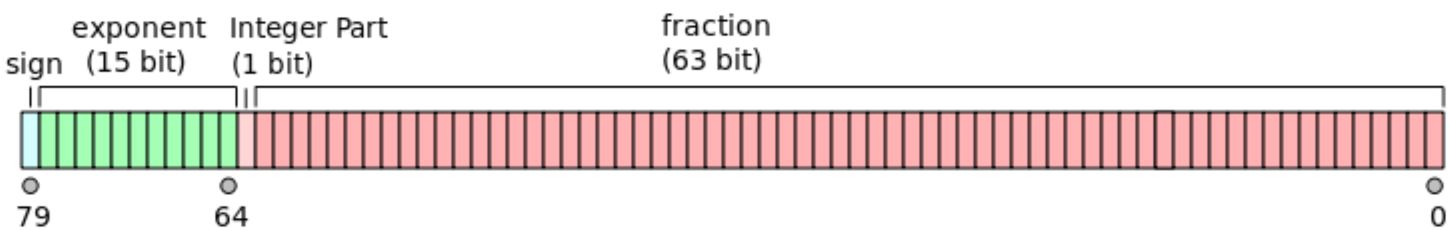
Quadruple precision (binary128)



39

Other IEEE 754 Formats

Intel x86 Extended Precision (80 bits), now common industry standard: explicit 1 in position 63 allows at times faster operation than IEEE 754!



Bias

$$b = \frac{2^n}{2} - 1$$

Or, equivalently:

$$b = (2^{n-1}) - 1$$

Where:

- b = the bias
- n = the number of bits in the exponent

More simply, the biases are shown in the table below:

Type	Bits	Bias
Half	5	15
Single	8	127
Double	11	1023
Extended	15	16383
Quad	15	16383

4

Data in Memory

Data in Memory

- Data are typically stored in memory. CPU needs to load these data to manipulate values
- If they are not in memory, data must be moved from secondary storage into memory; Slowly!
- Many computers organize their memory in units of bytes: are 8-Bit units, each unit being addressable
- Structured data generally processed in units of integers, or floating point, or decimal, or extended-precision FP versions
- Processor has specific operations for any type, i.e. float ops, integer ops, byte move ops, and more
- Bytes are arranged in a word sequentially, in either big endian or little endian order!

Data in Memory

- On a 64-bit, byte-addressable architecture it is convenient to view memory as a linear sequence of bytes, the first at address 0, the last at $2^{64} - 1$
- On such a machine, words are contiguous groups of 8 bytes, whose first address is evenly divisible by 8, i.e. the rightmost 3 address bits are by definition 0
- This is referred to as an aligned address; more precisely an 8-byte-aligned address
- Similarly on 32-bit word architecture: first byte address is evenly divisible by 4
- Most computers still can load/store words at any address, even unaligned; which causes multiple bus transactions
- Unaligned load/store OK, but way slower than aligned!

Data in Memory

C++ Data Type	Size [bits]	Min Value	Max Value
`char, signed char`	8	-128	127
`unsigned char`	8	0	255
`short int, signed short int`	16	-32768	+32767
`unsigned short int`	16	0	65535
`int, signed int [single]`	32	-2147483648	+2147483647
`unsigned int [single precision]`	32	0	4294967295
`float`	32	approx -10^{38}	approx 10^{38}
`double`	64	approx -10^{308}	approx 10^{308}

Formal language definition: long int no shorter than int!

Data in Memory: ASCII Characters

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	‘
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

C++ Highlights

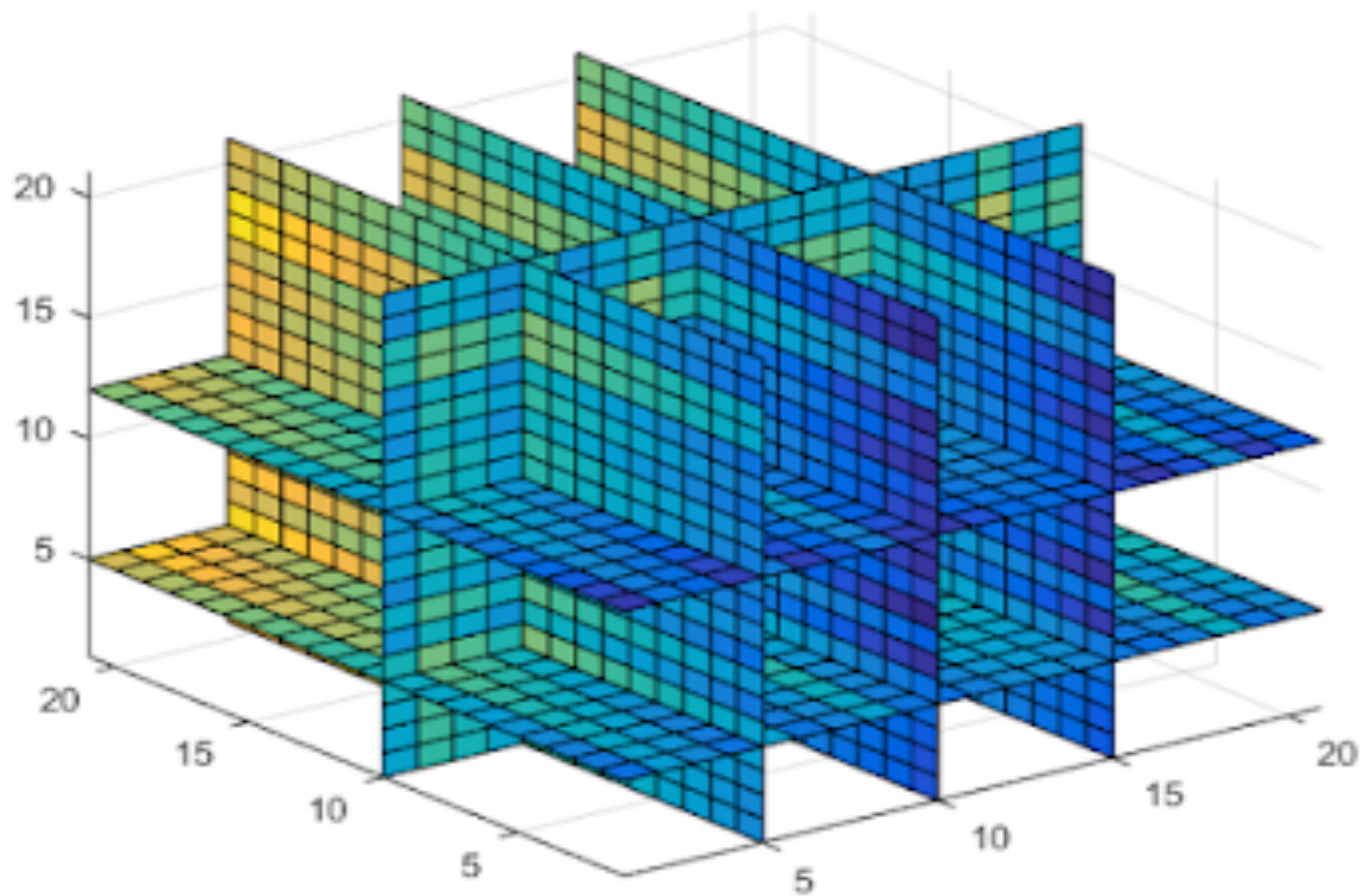
Quick Excursion to C++

- You have likely learned how to write complete Java, or C programs; some of you have learned C++
- You know scalar objects, of type bool, char, int, etc.
- There are 2 common aggregate types, aside from (scalar) pointer type: structures and arrays
- Other languages also have sets, sequences, and further high-level data structures
- Array is an aggregate of elements all of the same type, individually addressable by an index
- Struct is an aggregate of named elements of different types
- Arrays in C++ cannot be assigned as a whole; only element-wise in loops; hard limitation!

Quick Excursion to C++

- Array is a collection of objects of the same type, under one single name; only index differentiates elements
- Arrays require loops to be manipulated; note loops can be replaced via recursive algorithms
- Index is an integer expression
- Low bound of an array in Java and C++ is 0
- Hence the high-bound int value of an array is 1 less than the number of elements declared; there is the 0 element
- Bounds violations in C++ are usually errors; but are possible, are undetected in C, and can be disastrous!
- Structure elements are referred to by their field names

Arrays in C++



Arrays in C++

```
// declaration of a scalar int, named i
// if global: statuic. If local: automatic in C++
int i;          // uninitialized scalar int

// initialized scalar int, named j
int j = 109;    // initialized scalar int

// declaration of an int array k[] with 2 elements
int k[ 2 ];     // brackets make k[] an array of 2

// declaration of 2-dim int array of 5*3 = 15 ints
int mat1[ 5 ][ 5 ];

// bad habit to use “magic numbers” ☹ use symbol ☺
#define MAX 5
int mat2[ MAX ][ MAX ]; // 25 elements
```

Java and C++ Arrays in Memory

- Single-dimensional arrays conventionally stored in memory using increasing, sequential memory addresses
- Thus, the "next" element with next higher index is stored at the next higher, available address
- Generally, no holes are left in memory within arrays
- For multi-dimensional arrays there are several options:
  - One is named column-major order, as done for Fortran; not further discussed here
  - A more common one is row-major order, implemented in most programming languages, including Java and C++
- In row-major order, indices of multi-dimensional arrays vary most quickly from right to left for next element, or next higher dimension

Array Element Assignments

```
// good habit to use symbolic constant
#define SIZE 1000
int mat3[ SIZE ][ SIZE ];  // 1,000,000 elements

// access array element by “indexing”; shown later
#define N 999
int vector[ N ];
. . .
vector[  0  ] = 109; // first element at index 0
vector[  5  ] = 111; // index >=0 and < N. Why 5?
vector[ N-1 ] = -99; // last element
vector[  N  ] =   0; // error, no such element!
vector[  22 ] =   0; // again: poor programming
```

Array Element References

```
// good habit to use symbolic constants!
#define V_LENGTH 1000    // explain . . .
int i = 12;              // explain: why 12?
. . .
int vector[ V_LENGTH ];  // 1000 elements
vector[ i ] = 2014;      // element i assigned
. . .
cout << “vector[“ << i << “] = “ << vector[ i ]
     << endl;
. . .
vector[ i ] = vector[ i-1 ] + vector[ i+1 ]; // i?
```

More Arrays

```
// declarations with initialization
// Assume static space, i.e. outside function body!
#define MAX 8
int prime1[ MAX ] = { 1, 2, 3, 5, 7, 11, 13, 17 };
int prime2[ MAX ] = { 1, 2, 3, 5, 7, 11 }; // less OK

// Initial C++ static objects: 0; dynamic: garbage!
// declaration without explicit size: must infer size!
int prime3[ ] = { 1, 2, 3, 5, 7, 11 };
// how many elements, i.e. size?
// lowest index =
// highest index is =
cout << prime1[ 6 ] << endl;  // output?
cout << prime2[ 6 ] << endl;  // output?
cout << prime3[ 6 ] << endl;  // output? careful!
```

Key Learnings: Arrays

- All elements of an array have the same type, named the element type
- Distinguishable by index
- Index expression must yield an integer value
- Index of low bound in C++ and Java is 0
- High bound (or index of the last element) is size-1
- Bounds violations not checked at run-time in C++
- Array assignments as a whole not allowed in C++

Summary

- Computers use binary numbers internally
- Conventional integer numbers have limited numeric capacity, hence overflow can happen, both for float and integer numbers
- Other bases: 8, 10, 16; also C++
- Two's Complement is common integer numeric representation; One's Complement also used n some old supercomputers
- Many computer architectures are byte-oriented, with 8 bits per byte; each holding 1 ASCII character
- Chinese symbols encoded with 2 bytes
- Some past architectures had different storage units than byte; e.g. 60-bit words

References

1. Byte definition: https://en.wikipedia.org/wiki/Byte
2. Two's Complement: https://en.wikipedia.org/wiki/Two%27s_complement
3. Wiki gateway to large number of binary arithmetic operations: https://en.wikipedia.org/wiki/Category:Binary_arithmetic
4. George Boole Info: https://en.wikipedia.org/wiki/George_Boole