More about Binary

9/6/2016

Unsigned vs. Two's Complement

8-bit example:

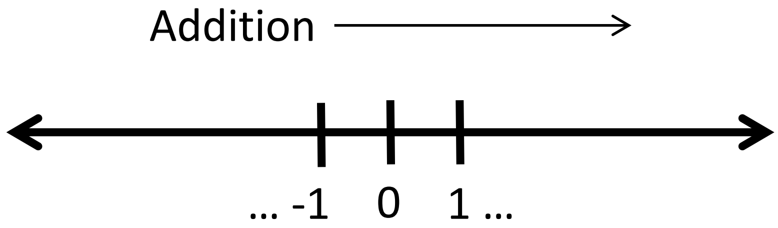
```
1   1   0   0   0   0   1   1
```

$$2^7+2^6 \qquad + \qquad 2^1+2^0 = 128+64+2+1$$
$$= 195$$

$$-2^7+2^6 \qquad + \qquad 2^1+2^0 = -128+64+2+1$$
$$= -61$$

Why does two's complement work this way?

The traditional number line



Unsigned ints on the number line

Unsigned Integers

- Suppose we had one byte
  - Can represent 2^8 (256) values
  - If unsigned (strictly non-negative): 0 – 255

252 = `11111100`

253 = `11111101`

254 = `11111110`

255 = `11111111`

What if we add one more?

Car odometer “rolls over”.

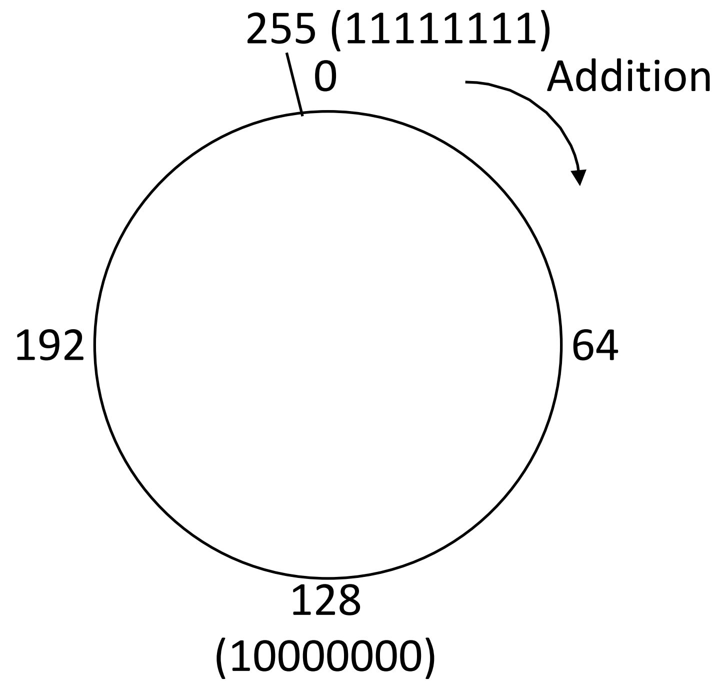
Unsigned Overflow

If we add two N-bit unsigned integers, the answer can't be more than $2^N – 1$.

$$\begin{array}{rr} & 11111010 \\ + & 00001100 \\ \hline & \cancel{1}00000110 \end{array}$$

When there should be a carry from the last digit, it is lost. This is called **overflow**, and the result of the addition is incorrect.

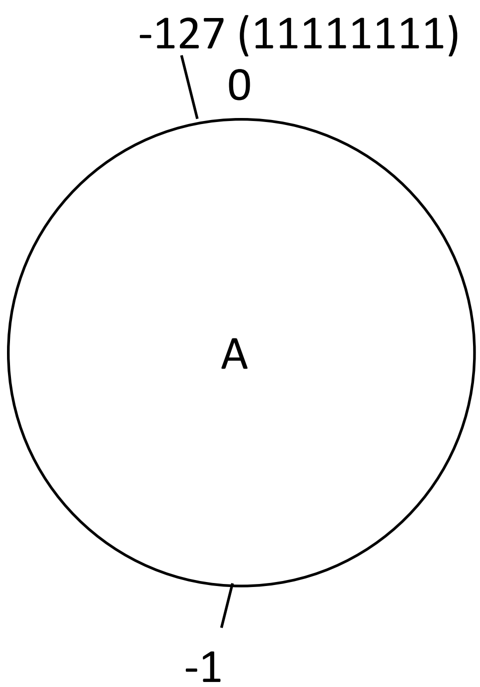
In cs31, the number line is a circle

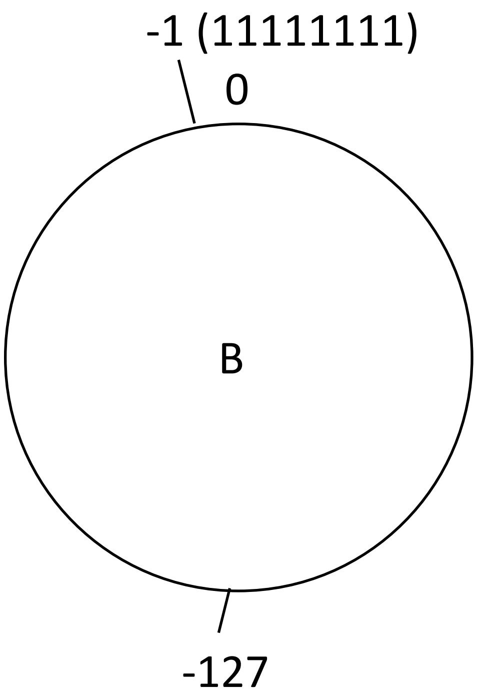


This means that all arithmetic is modular. With 8 bits, arithmetic is mod 2^8; with N bits arithmetic is mod 2^N.

255 + 4 = 259 % 256 = 3

Suppose we want to support negative values too (-127 to 127). Where should we put -1 and -127 on the circle? Why?

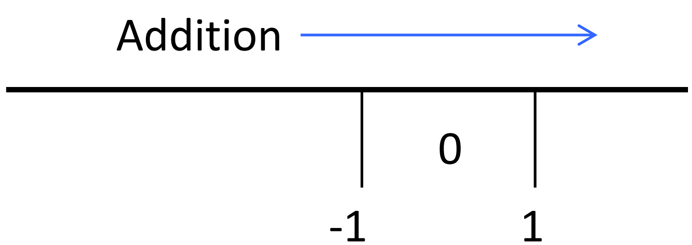




C: Put them somewhere else.

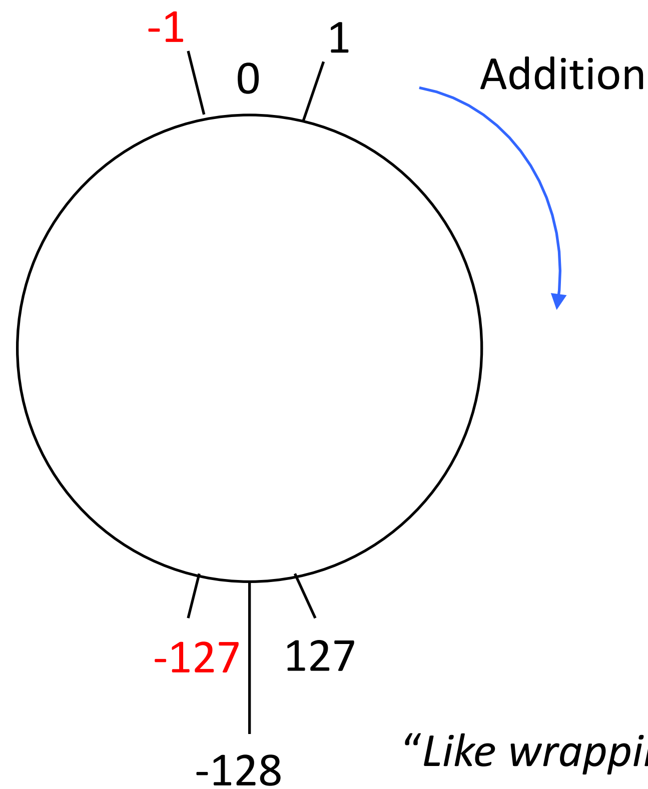
Option B is Two's Complement

- Borrows nice properties from the number line:



Only one instance of zero, with -1 and 1 on either side of it.

Addition: moves to the right



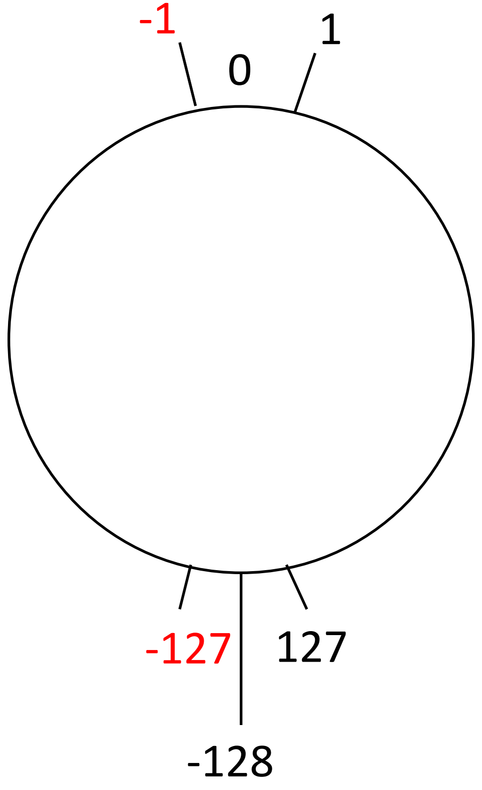
"Like wrapping number line around a circle"

Does two's complement, solve the "rolling over" (overflow) problem?

A. Yes, it's gone.

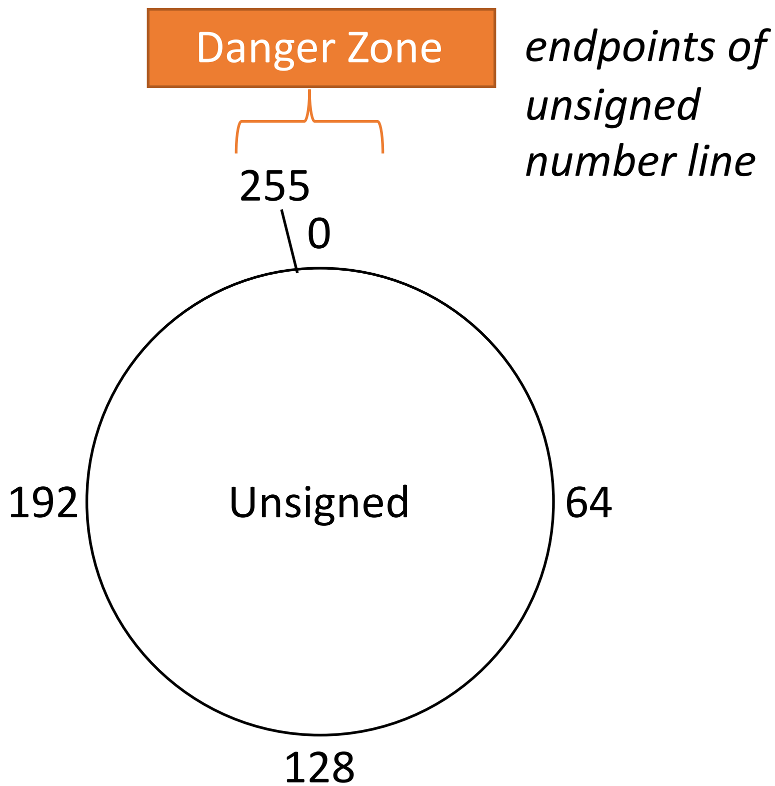
B. Nope, it's still there.

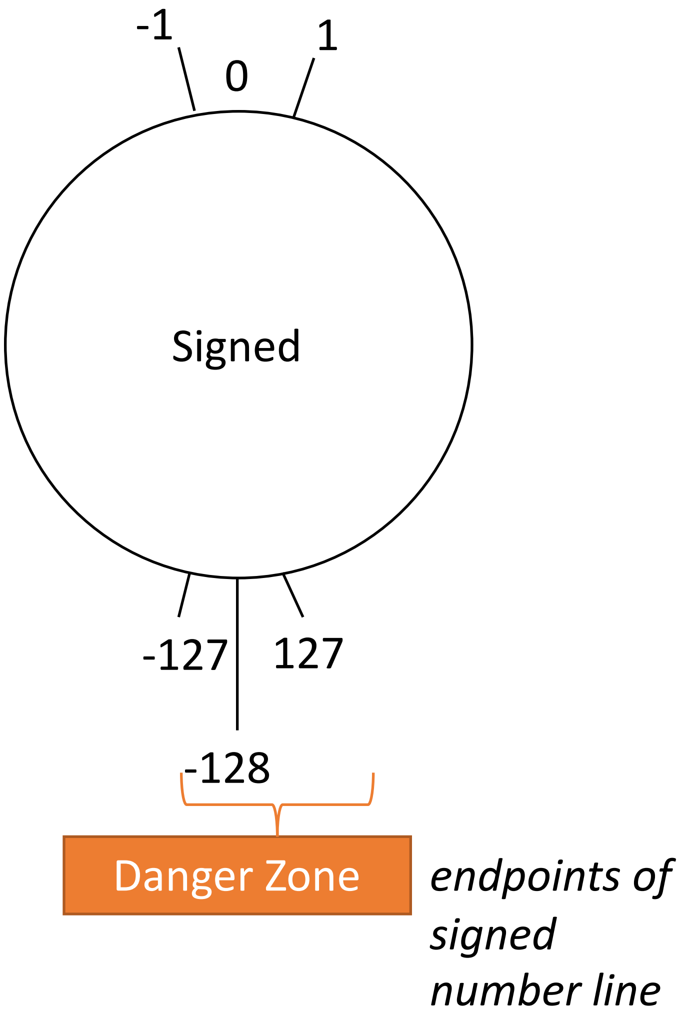
C. It's even worse now.



This is an issue we need to be aware of when adding and subtracting!

Overflow, Revisited



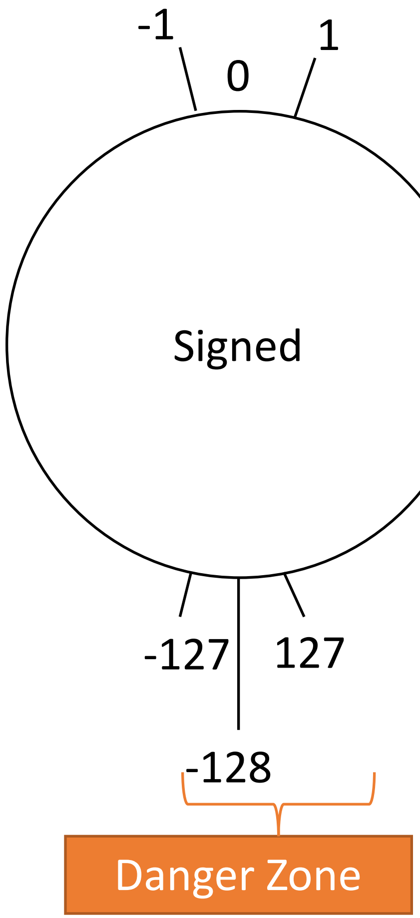


If we add a positive number and a negative number, will we have overflow? (Assume they are the same # of bits)

A. Always

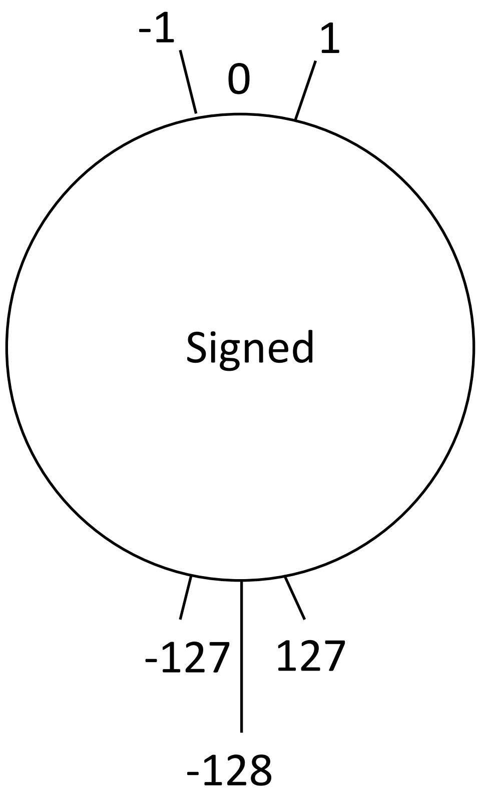
B. Sometimes

C. Never



Signed Overflow

- Overflow: IFF the sign bits of operands are the same, but the sign bit of result is different.
  - Not enough bits to store result!



Signed addition (and subtraction):

2+-1=1	2+-2=0	2+-4=-2
0010	**0**010	**0**010
+**1**111	+**1**110	+**1**100
1 0001	1 0000	1110

No chance of overflow here - signs of operands are different!

Signed Overflow

- Overflow: IFF the sign bits of operands are the same, but the sign bit of result is different.
  - Not enough bits to store result!

<u>Signed addition (and subtraction)</u>:

2+-1=1	2+-2=0	2+-4=-2	2+7=-7	-2+-7=7
0010	0010	0010	**0**010	**1**110
+1111	+1110	+1100	+**0**111	+**1**001
1 0001	1 0000	1110	**1**001	1 **0**111

Overflow here! Operand signs are the same, and they don't match output sign!

Overflow Rules

- Signed:
  - The sign bits of operands are the same, but the sign bit of result is different.
- Can we formalize unsigned overflow?
  - Need to include subtraction too, skipped it before.

Recall Subtraction Hardware

<u>Negate and add 1 to second operand:</u>

Can use the same circuit for add and subtract:

6 - 7 == 6 + ~7 + 1

```
input 1 ------------------------------->  +-------------+
input 2 --> possible bit flipper -->      | ADD CIRCUIT | ---> result
        | possible +1 input | -------->   +-------------+
```

Let’s call this +1 input: “Carry in”

How many of these unsigned operations have overflowed?

4 bit unsigned values (range 0 to 15):

```
                                         carry-in   carry-out
                                             ↓          ↓
```

Addition (carry-in = 0)

```
9 + 11  =  1001 + 1011 + 0 = 1  0100
9 +  6  =  1001 + 0110 + 0 = 0  1111
3 +  6  =  0011 + 0110 + 0 = 0  1001
```

Subtraction (carry-in = 1)

```
                    (~3)
6 -  3  =  0110 + 1100 + 1  = 1  0011
3 -  6  =  0011 + 1010 + 1  = 0  1101
                    (~6)
```

A. 1
B. 2
C. 3
D. 4
E. 5

How many of these <u>unsigned</u> operations have overflowed?

4 bit unsigned values (range 0 to 15):

```
                                      carry-in   carry-out
Addition (carry-in = 0)                   ↓          ↓
   9 + 11  =   1001 + 1011 + 0 =   1   0100 =  4
   9 +  6  =   1001 + 0110 + 0 =   0   1111 =  15
   3 +  6  =   0011 + 0110 + 0 =   0   1001 =  9
```

```
                              (~3)
Subtraction (carry-in = 1)
   6 -  3  =   0110 + 1100 + 1  = 1   0011 =  3
   3 -  6  =   0011 + 1010 + 1  = 0   1101 = 13
                              (~6)
```

A. 1

B. 2

C. 3

D. 4

E. 5

What's the pattern?

Overflow Rule Summary

- Signed overflow:
  - The sign bits of operands are the same, but the sign bit of result is different.

- Unsigned: overflow
  - The carry-in bit is different from the carry-out.

C_{in}	C_{out}	C_{in} XOR C_{out}
0	0	0
0	**1**	**1**
1	**0**	**1**
1	1	0

So far, all arithmetic on values that were the same size. What if they're different?

Suppose we have a signed 8-bit value, 00010110 (22), and we want to add it to a signed 4-bit value, 1011 (-5). How should we represent the four-bit value?

A. 1101 (don’t change it)
B. 00001101 (pad the beginning with 0’s)
C. 11111011 (pad the beginning with 1’s)
D. Represent it some other way.

Sign Extension

- When combining signed values of different sizes, expand the smaller to equivalent larger size:

```
char y=2, x=-13;
short z = 10;
```

```
   z = z + y;                         z = z + x;

0000000000001010              0000000000000101
+       00000010              +       11110011
0000000000000010              1111111111110011
```

Fill in high-order bits with sign-bit value to get same numeric value in larger number of bytes.

Let’s verify that this works

4-bit signed value, sign extend to 8-bits, is it the same value?

`0111    ---->  0000 0111`   obviously still 7

`1010    ---->  1111 1010`   is this still -6?

-128 + 64 + 32 + 16 + 8 + 0 + 2 + 0 = -6   yes!

Operations on Bits

- For these, doesn’t matter how the bits are interpreted (signed vs. unsigned)

- Bit-wise operators (AND, OR, NOT, XOR)

- Bit shifting

Bit-wise Operators

- bit operands, bit result (interpret as you please)

& (AND) | (OR) ~(NOT) ^(XOR)

A	B	A & B	A | B	~A	A ^ B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

```
  01010101        01101010        10101010      ~10101111
| 00100001      & 10111011      ^ 01101001      ---------
----------      ----------      ----------       01010000
  01110101        00101010        11000011
```

More Operations on Bits

- Bit-shift operators: << left shift, >> right shift

```
01010101 << 2  is 01010100
                  2 high-order bits shifted out
                  2 low-order bits filled with 0
01101010 << 4  is 10100000
01010101 >> 2  is 00010101
01101010 >> 4  is 00000110

10101100 >> 2  is 00101011 (logical shift)
               or 11101011 (arithmetic shift)
```

Arithmetic right shift: fills high-order bits w/sign bit

C automatically decides which to use based on type:

signed: arithmetic, unsigned: logical

Floating Point Representation

1 bit for sign
8 bits for exponent
23 bits for precision

sign | exponent | fraction |

I don't expect you to memorize this

$$\text{value} = (-1)^{\text{sign}} * 1.\text{fraction} * 2^{(\text{exponent}-127)}$$

let's just plug in some values and try it out

```
0x40ac49ba: 0 10000001   01011000100100110111010
      sign = 0 exp = 129   fraction = 2902458
```

$$= 1*1.2902458*2^2 = 5.16098$$

Think of scientific notation: $1.933\text{e-}4 = 1.933 * 10^{-4}$

Character Representation

- Represented as one-byte integers using ASCII.
- ASCII maps the range 0-127 to letters, punctuation, etc.

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

asciicharstable.com

Characters and strings in C

```
char c = ‘J’;
char s[6] = “hello”;
s[0] = c;
printf(“%s\n”, s);
```

Will print: `Jello`

- Character literals are surrounded by single quotes.
- String literals are surrounded by double quotes.
- Strings are stored as arrays of characters.

Discussion question: how can we tell where a string ends?

A. Mark the end of the string with a special character.

B. Associate a length value with the string, and use that to store its current length.

C. A string is always the full length of the array it's contained within (e.g., `char name[20]` must be of length 20).

D. All of these could work (which is best?).

E. Some other mechanism (such as?).

What will this snippet print?

```
char c = 'J';
char s[6] = "hello";
s[5] = c;
printf("%s\n", s);
```

A. `Jello`

B. `hellJ`

C. `helloJ`

D. Something else, that we can determine.

E. Something else, but we can't tell what.