A *positional* or *place-value* notation is a numeral system in which each position is related to the next by a constant multiplier, called the *base* or *radix* of that numeral system.

The value of each digit position is the value of its digit, multiplied by a power of the base; the power is determined by the digit's position.

The value of a positional number is the total of the values of its positions.

So, in positional base-10 notation:

$$73901 = 7 \times 10^4 + 3 \times 10^3 + 9 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$$

And, in positional base-2 notation:

$$10010000010101101 = 1 \times 2^{16} + 1 \times 2^{13} + 1 \times 2^7 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$$

Why is the second example a cheat?

Do not confuse the representation with the number!

Each of the following examples is a <u>representation</u> of the same number:

255_{10}

11111111_2

FF_{16}

2010_5

377_8

3333_4

100110_3

Do not make the mistake of thinking that there is such a thing as "a base-10 number" or "a base-16 number".

There is a unique base-10 <u>representation</u> of every integer and there is a unique base-16 <u>representation</u> of every integer.

Converting from base-10 to base-2

Given a base-10 representation of an integer value, the base-2 representation can be calculated by successive divisions by 2:

```
73901     Remainder
36950        1
18475        0
 9237        1
 4618        1
 2309        0
 1154        1
  577        0
  288        1
  144        0
   72        0
   36        0
   18        0
    9        0
    4        1
    2        0
    1        0
    0        1
```

10010000010101101_2

Given a base-2 representation of an integer value, the base-10 representation can be calculated by simply expanding the positional representation:

$$10010000010101101_2 = 1 \times 2^{16} + 1 \times 2^{13} + 1 \times 2^7 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$$

$$= 65536 + 8192 + 128 + 32 + 8 + 4 + 1$$

$$= 73901$$

Are analagous… given a base-10 representation of an integer value, the base-16 representation can be calculated by successive divisions by 16:

```
73901       Remainder
 4618          13 --> D
  288          10 --> A
   18           0
    1           2
    0           1
```

$120AD_{16}$

The choice of base determines the set of numerals that will be used.

base-16 (hexadecimal or simply hex)

numerals: 0 1 . . . 9 A B C D E F

Given a base-2 representation of an integer value, the base-16 representation can be calculated by simply converting the nybbles:

```
1 0010 0000 1010 1101
1    2    0    A    D    : hex
```

The same basic "trick" works whenever the target base is a power of the source base:

```
10 010 000 010 101 101
 2   2   0   2   5   5  : octal
```

base-2	binary	0 1
base-8	octal	0 1 2 3 4 5 6 7
base-10	decimal	0 1 2 3 4 5 6 7 8 9
base-16	hex	0 1 2 3 4 5 6 7 8 9 A B C D E F

A *binary digit* or *bit* has a value of either 0 or 1; these are the values we can store in hardware devices.

A *byte* is a sequence of 8 bits.

A byte is also the fundamental unit of storage in memory.

A *nybble* is a sequence of 4 bits (half of a byte).

Consider the table at right:

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

Any storage system will have only a finite number of storage devices.

Whatever scheme we use to represent integer values, we can only allocate a finite number of storage devices to the task.

Put differently, we can only represent a (small) finite number of bits for any integer value.

This means that computations, even those involving only integers, are inherently different on a computer than in mathematics.

As an example, suppose that we decide to provide support for integer values represented by 32 bits.

There are 2^32 or precisely 4,294,967,296 different patterns of 32 bits.

So we can only represent that many different integer values.

<u>Which</u> integer values we actually represent will depend on how we <u>interpret</u> the 32 bits:

```
1 bit for sign, 31 for magnitude (abs value):      -2147483647 to +2147483647

32 bits for magnitude (no negatives):                        0 to +4294967295

2's complement representation:                     -2147483648 to +2147483647
```

data type   a collection of values together with the definitions of a number of operations that can be performed on those values

We need to provide support for a variety of data types.

For integer values, we need to provide a variety of types that allow the user to choose based upon memory considerations and range of representation.

For contemporary programming languages, we would expect:
- signed integers and unsigned integers
- 8-, 16-, 32- and (perhaps) 64-bit representations
- the common arithmetic operations (addition, subtraction, multiplication, division, etc.)
- sensible handling of issues related to limited ranges of representation
- sensible handling of computational errors resulting from abuse of operations

We store the number in base-2, using a total of n bits to represent its value.

Common values for n include 8, 16, 32 and 64, although any positive number of bits would work.

The range of represented values will extend from 0 to $2^n - 1$.

For non-negative integers, represent the value in base-2, using up to n – 1 bits, and pad to n bits with leading 0's:

```
42:    101010 --> 0010 1010
```

For negative integers, take the base-2 representation of the value (ignoring the sign) pad with 0's to n – 1 bits, invert the bits and add 1:

```
-42:   101010 --> 0010 1010
              --> 1101 0101
              --> 1101 0110
```

Weird! What's the point? Well, we've represented -42 in such a way that if we use the usual add/carry algorithm we'll find that 42 + -42 yields 0 (obviously desirable):

```
 42:   0010 1010
-42:   1101 0110
sum:   0000 0000  (ignore carry-out)
```

Here's another way to understand why this makes sense...

Let's suppose we have 16-bit signed integers. Now it's natural to represent 0 and 1 as:

```
     0:        0000 0000 0000 0000
     1:        0000 0000 0000 0001
```

Now, how would you represent -1? You want 1 + -1 to equal 0, so...

```
     1:        0000 0000 0000 0001
    -1:        ???? ???? ???? ????
               --------------------
     0:        0000 0000 0000 0000
```

1 + ? == 0

? must be 1

carry == 1

So, we'd want to represent -1 as:

```
              -1:    1111 1111 1111 1111
```

To negate an integer, with one exception*, just invert the bits and add 1.

```
      25985:   0110 0101 1000 0001


     -25985:   1001 1010 0111 1111


    --25985:   0110 0101 1000 0001
```

The sign of the integer is indicated by the leading bit.

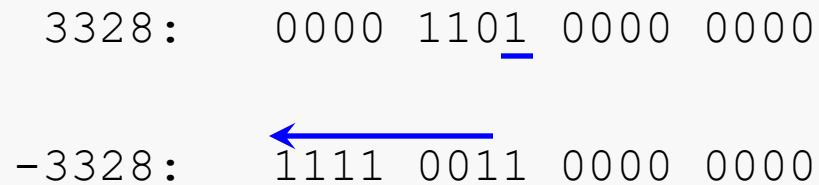There is only one representation of the value 0.

The range of representation is asymmetrical about zero:

minimum $\quad -2^{n-1}$

maximum $\quad 2^{n-1} -1$

To negate an integer, with one exception, find the right-most bit that equals 1 and then invert all of the bits to its left:

```
        3328:      0000 1101 0000 0000


       -3328:      1111 0011 0000 0000
```

Why does this work?

If the integer is non-negative, just expand the positional representation:

```
0000 1101 0000 0000   =   2^11 + 2^10 + 2^8
                      =   3328
```

If the integer is negative, take its negation (in 2's complement), expand the positional representation for that, and then take the negation of the result (in base-10).

base-10 to 2's Complement

Obvious method:

- apply the division-by-2 algorithm discussed earlier to the magnitude of the number

- if value is negative, negate the result

Alternate method:

- find the largest power of 2 that's less than the magnitude of the number

- subtract it from the magnitude of the number and set that bit-position to 1

- repeat until the magnitude equals 0

- if value is negative, negate the result

```
          0   1   2   3    4    5    6              10     11     12      set bit
          1   2   4   8   16   32   64   . . .    1024   2048   4096
3328:                                                                           11

1280:                                                                           10

 256:                                                                            8

   0!
```

"An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is too large to be represented within the available storage space [Wikipedia]".

When addition is successful on an unsigned number:

```
Carry:     0101 010
 42:       0010 1010
 42:       0010 1010
           ---------
sum:       0101 0100 (no carry-out)
```

When overflow occurs on an unsigned number:

```
 Carry: 11111 111
 255:      1111 1111
   1:      0000 0001
           ---------
 sum:      0000 0000 (carry out is one, overflow)
```

When overflow occurs on a signed number:

```
Carry:    11111 110
   42:      0010 1010
  -42:      1101 0110
  sum:      0000 0000 (ignore carry-out)
```

When overflow occurs on a signed number:

```
 Carry:    10000 000
 -128:      1000 0000
   -1:      1111 1111
  sum:      0111 1111 (positive number, overflow)
```

Be careful mixing signed and unsigned numbers, the results may surprise you:

```
int32_t x = -1;
uint32_t y = -1;

printf("print x as a signed number: %d\n", x);
printf("print x as an unsigned number: %u\n", x);


printf("print y as a signed number: %d\n", y);
printf("print y as an unsigned number: %u\n", y);
```

What does this print?

```
print x as a signed number: -1
print x as an unsigned number: 4294967295
print y as a signed number: -1
print y as an unsigned number: 4294967295
```

Key point: bits are the same for x and y, difference is how we interpret them. x == y will evaluate to true.

Be careful mixing signed and unsigned numbers, the results may surprise you:

```
int32_t x = -1;
uint32_t y = 1;

if ( x < y )
    printf("x is less than y.\n");
else
    printf("x is greater than y.\n");
```

What does this print?

```
x is greater than y.
```

Key point: when comparing signed and unsigned numbers, the signed numbers are converted to unsigned. It's best to explicitly cast to the type you want.

See more: http://stackoverflow.com/questions/5416414/signed-unsigned-comparisons

You can actually perform the 2's complement operation in your code:

```
int32_t w = 533;

// Invert (~) w and add 1
w = ~w + 1;
printf("print w: %d\n", w);
```

What does this print?

```
print w: -533.
```

What about this code?

```
int32_t z = 1 << 31;

// This should negate it right?
z = ~z + 1;

// No! most negative number, -2147483648
printf("print z after the conversion: %d\n", z);
```

The American Standard Code for Information Interchange maps a set of 128 characters into the set of integers from 0 to 127, requiring 7 bits for each numeric code:

95 of the characters are "printable" and are mapped into the codes 32 to 126:

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

The remainder are special control codes (e.g., WRU, RU, tab, line feed, etc.).

Since the fundamental unit of data storage was quickly standardized as an 8-bit byte, the high bit was generally either set to 0 or used as a *parity-check bit*.

Some ASCII Features

The decimal digits '0' through '9' are assigned sequential codes.

Therefore, the numeric value of a digit can be obtained by subtraction: '7' – '0' = 7

The upper-case characters 'A' through 'Z' are also assigned sequential codes, as are the lower-case characters 'a' through 'z'.

This aids in sorting of character strings, but note that upper-case characters have lower-valued codes than do upper-case characters.

There are no new operations, but since ASCII codes are numeric values, it is often possible to perform arithmetic on them to achieve useful results…

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

It's easy to find ASCII tables online (including some that are clearer than this one):

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{			}	~	DEL

For good or ill, the ASCII codes are 7-bit codes, and that leads to temptation.

There exist 8-bit character encodings that extend the ASCII codes to provide for 256 different characters (e.g., ISO_8859-1:1987).

Unfortunately, none of these has achieved the status of a practical Standard in use.

```
                    _____
              . - ' &        ' - .
            /                      \
         :        O          O       ;
       (              (_             )
         :                           ;
            \            ___        /
            ` - .  _____   . - '
              :  `" " " "`
             /              \
            /        ,        \
          / | / \ / \ /\  _  \
         (_ | / \ / \ / \\_   )
           | _____ |
          ___ ) _  | _  (___
         (_____ | _____)
```
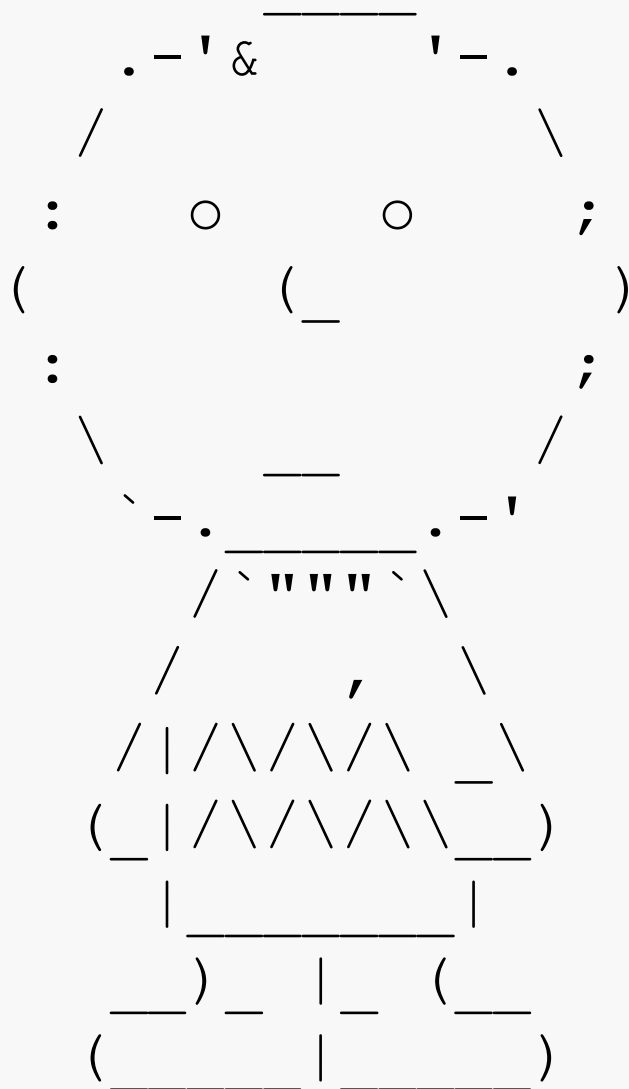
We must represent two values, TRUE and FALSE, so a single bit suffices.

We will represent TRUE by 1 and FALSE by 0.

Thus, a sequence of bits can be viewed as a sequence of logical values.

Note: this is not the view typically taken in high-level languages!

Given two Boolean logical values, there are a number of operations we can perform:

```
A   NOT A
--------
0     1
1     0
```

```
A   B   A AND B
--------------
0   0      0
0   1      0
1   0      0
1   1      1
```

```
A   B   A OR B
--------------
0   0      0
0   1      1
1   0      1
1   1      1
```

```
A   B   A NAND B
--------------
0   0      1
0   1      1
1   0      1
1   1      0
```

```
A   B   A XOR B
--------------
0   0      0
0   1      1
1   0      1
1   1      0
```

```
A   B   A NOR B
--------------
0   0      1
0   1      0
1   0      0
1   1      0
```

```
A   B   A XNOR B
--------------
0   0      1
0   1      0
1   0      0
1   1      1
```