COMSC-110 Chapter 9

Counting On Your Fingers: Bits And Bytes

Valerie Colber, MBA, PMP, SCPM

Background on Binary and memory

Bits and Bytes

- How are numbers and text stored in computer memory?
- Why numbers and text are treated separately in programming?
- Why is there a distinction between integers and floating point numbers?
- Why are there range limits for numbers?
- Is there a precision issue with floating point variables?

What makes computers different?

- Platform : 1) desktop; 2) Server; 3) mainframe
- Operating system
- How many CPU’s
- How many instructions per second the CPU can process
- How much memory
- How much storage
- How long the instructions can be
- User interface

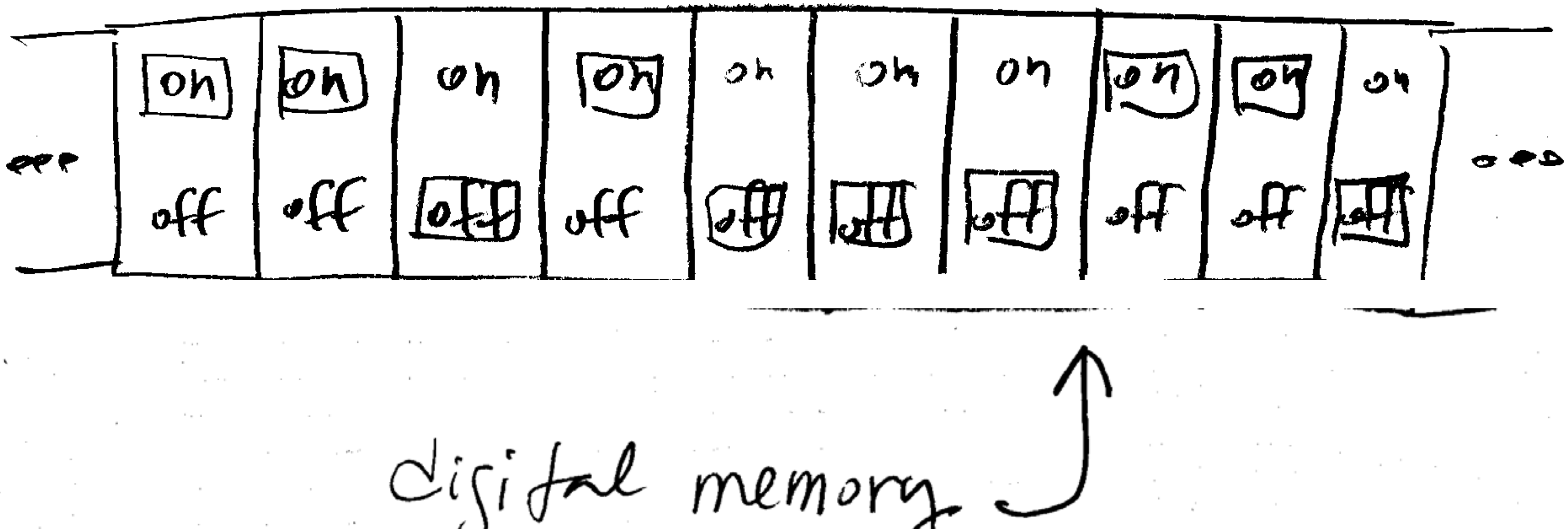
Types of Memory

- RAM (random access memory)
- ROM (read only memory)
- Cache
- Registers
- Virtual
- Volitile
- Non-volitile
- CMOS (**C**omplementary **M**etal **O**xide **S**emiconductor battery backed memory)
- Flash
- Video

Types of data

- Numbers
- Text
- Graphics
- Audio
- Video

Computer Memory: Vast Arrays Of On/Off Switches


on on on on on on on on on on
off off off off off off off off off off

digital memory

just a set of on/off switches

Computer memory

- Memory consist of millions or billions of electronic on/off switches.
- Each switch can store one of two possible values -- off or on -- zero or one.
- A switch is called a *bit*, and it is made out of several electrical components -- resistors and transistors combined in a circuit.
- By applying an electrical pulse to the circuit that comprises the bit, it can be placed into one of two possible modes, which represent "on" and "off".

Data and Program Representation

- In order to be understood by a computer, data and programs need to be represented appropriately
- Coding systems: Used to represent numeric, text-based, and multimedia data, as well as to represent programs
- Digital computers: Can only understand two states, off and on (0 and 1)
- Digital data representation: The process of representing data in digital form so it can be used by a computer

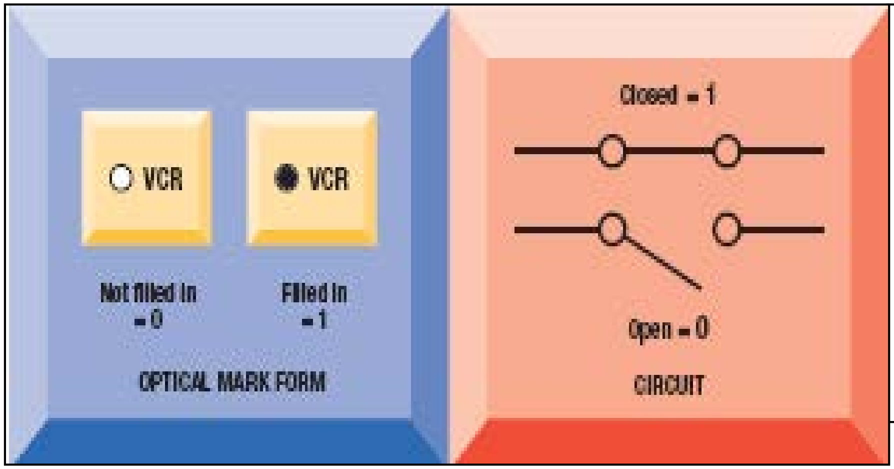
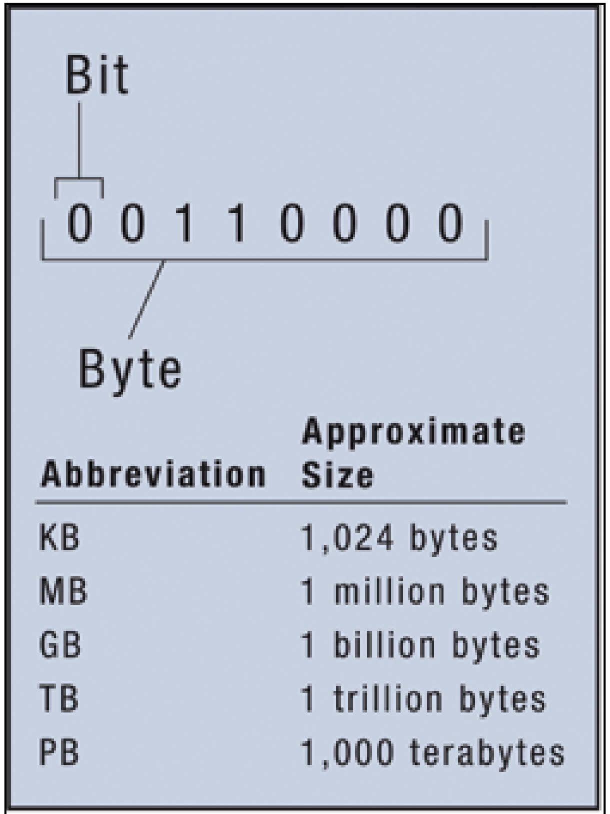


FIGURE 2-1

Ways of representing 0 and 1. Binary computers recognize only two states–off and on–usually represented by 0 and 1.

Digital Data Representation

- Bit: The smallest unit of data that a binary computer can recognize (a single 1 or 0)
- Byte = 8 bits
- Byte terminology used to express the size of documents and other files, programs, etc.
- Prefixes are often used to express larger quantities of bytes: kilobyte (KB), megabyte (MB), gigabyte (GB), etc.



Abbreviation	Approximate Size
KB	1,024 bytes
MB	1 million bytes
GB	1 billion bytes
TB	1 trillion bytes
PB	1,000 terabytes

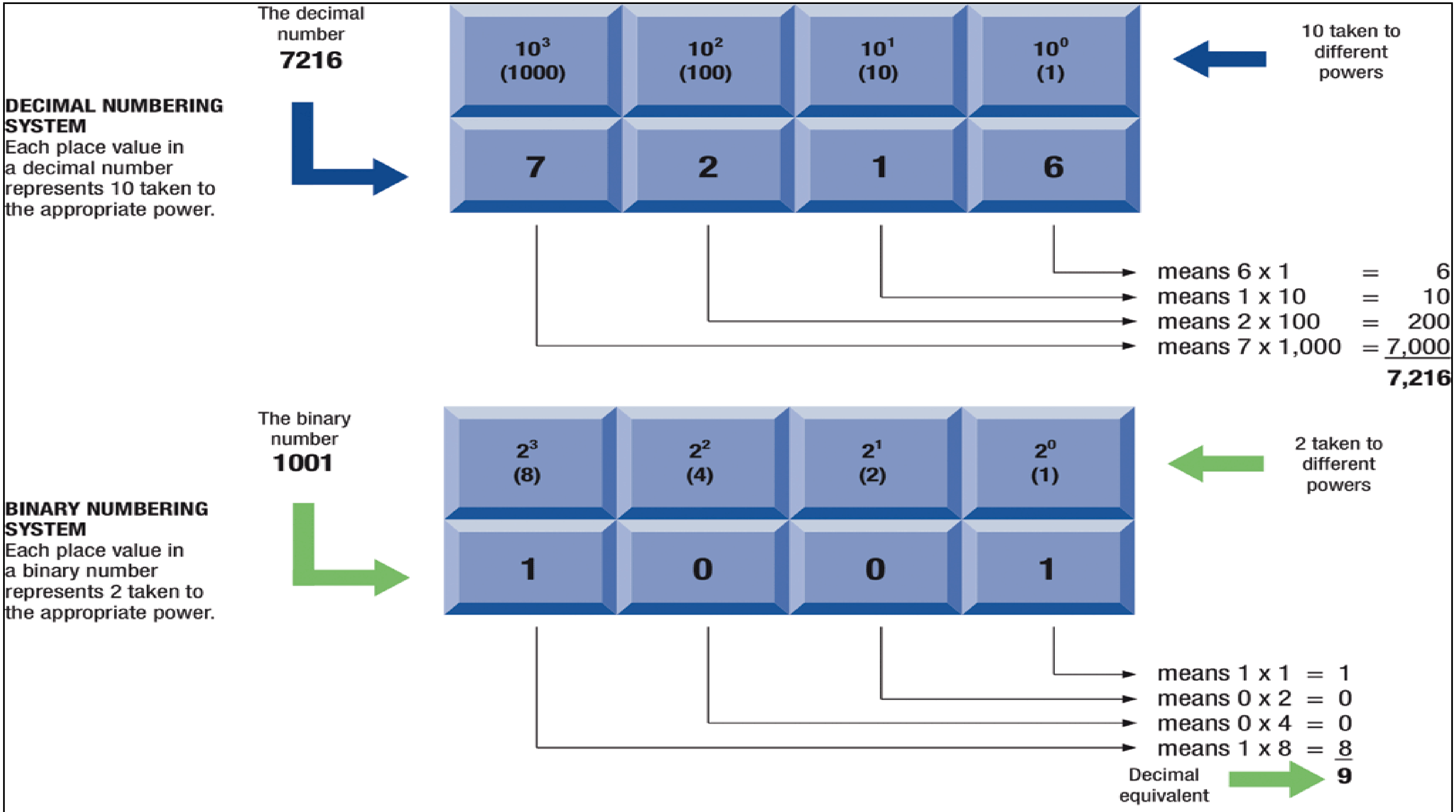
FIGURE 2-2

Bits and bytes.

The Binary Numbering System

- Numbering system: A way of representing numbers
- Decimal numbering system
  - Uses 10 symbols (0-9)
- Binary numbering system
  - Uses only two symbols (1 and 0) to represent all possible numbers
- In both systems, the position of the digits determines the power to which the base number (such as 10 or 2) is raised

Decimal & Binary Numbering Systems



The decimal number **7216**

DECIMAL NUMBERING SYSTEM
Each place value in a decimal number represents 10 taken to the appropriate power.

10^3 (1000)	10^2 (100)	10^1 (10)	10^0 (1)
7	2	1	6

10 taken to different powers

means 6 x 1 = 6
means 1 x 10 = 10
means 2 x 100 = 200
means 7 x 1,000 = 7,000
7,216

The binary number **1001**

BINARY NUMBERING SYSTEM
Each place value in a binary number represents 2 taken to the appropriate power.

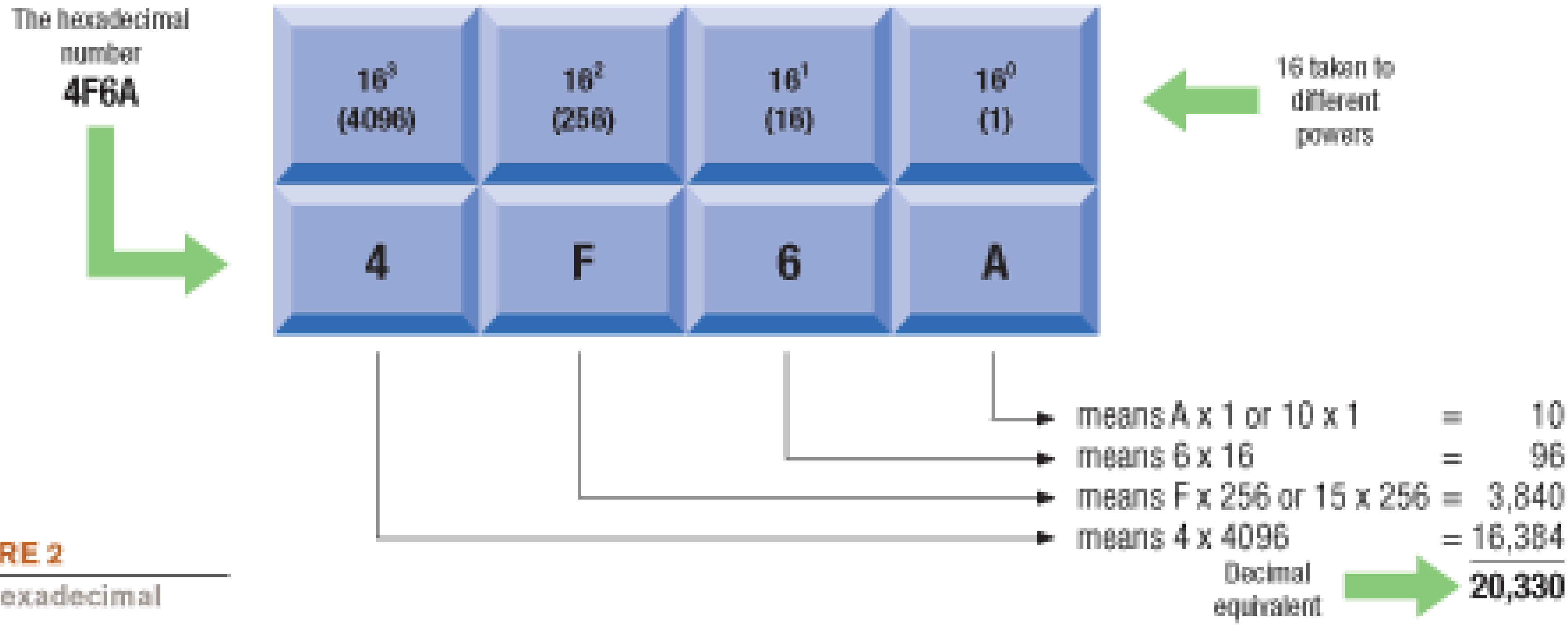
2^3 (8)	2^2 (4)	2^1 (2)	2^0 (1)
1	0	0	1

2 taken to different powers

means 1 x 1 = 1
means 0 x 2 = 0
means 0 x 4 = 0
means 1 x 8 = 8
Decimal equivalent **9**

FIGURE 2-3
Examples of using the decimal and binary numbering systems.

Hexidecimal Numbering System



means A x 1 or 10 x 1 = 10

means 6 x 16 = 96

means F x 256 or 15 x 256 = 3,840

means 4 x 4096 = 16,384

Decimal equivalent 20,330

FIGURE 2

The hexadecimal (base 16) numbering

Counting in Base 10/Decimal

- All the numbers you need to make any number in base 10 are 0-9.
- Numbers in base 10 have digits from the right to left from ones, tens, hundreds, thousands, …
- When you reach 9 and add 1, you get 10 which is zero ones in the ones column and 1 ten in the tens column. When you reach 9 in the tens column and the ones column and add 1, you get 100 which is zero ones in the ones column and zero tens in the tens column and 1 hundred in the hundreds column. And so on.

Counting in Base 10/Decimal

Column Values:

- $10^0 = 1$
- $10^1 = 10$
- 10^2 = 10 * 10 = 100
- 10^3 = 10 * 10 * 10 = 1000

Counting in Base 2/Binary

- All the numbers you need to make any number in base 2 are 0 and 1.
- Numbers in base 2 have digits from the right to left from ones, twos, fours, eights, …

Column values:

- $2^0 = 1$
- $2^1 = 2$
- 2^2 = 2 * 2 = 4
- 2^3 = 2 * 2 * 2 = 8
- Binary numbers should be grouped in 4 digits, so leading zeros may be necessary.

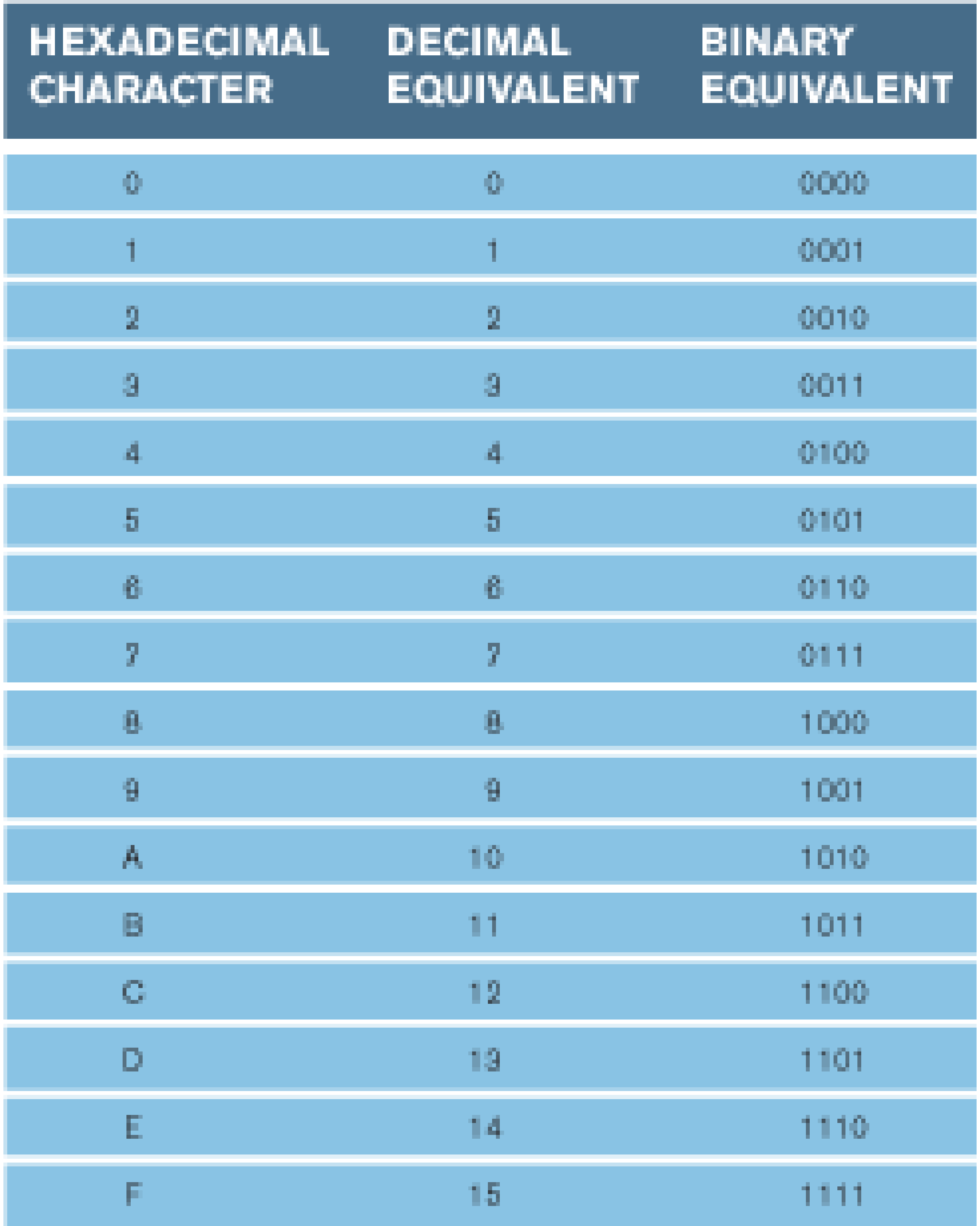
Counting in Base 2/Binary

How many 8’s, 4’s, 2’s, 1’s add up to the decimal number?

Binary				Decimal
2x2x2	2x2	2	1	
2^3	2^2	2^1	2^0	
8	4	2	1	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8

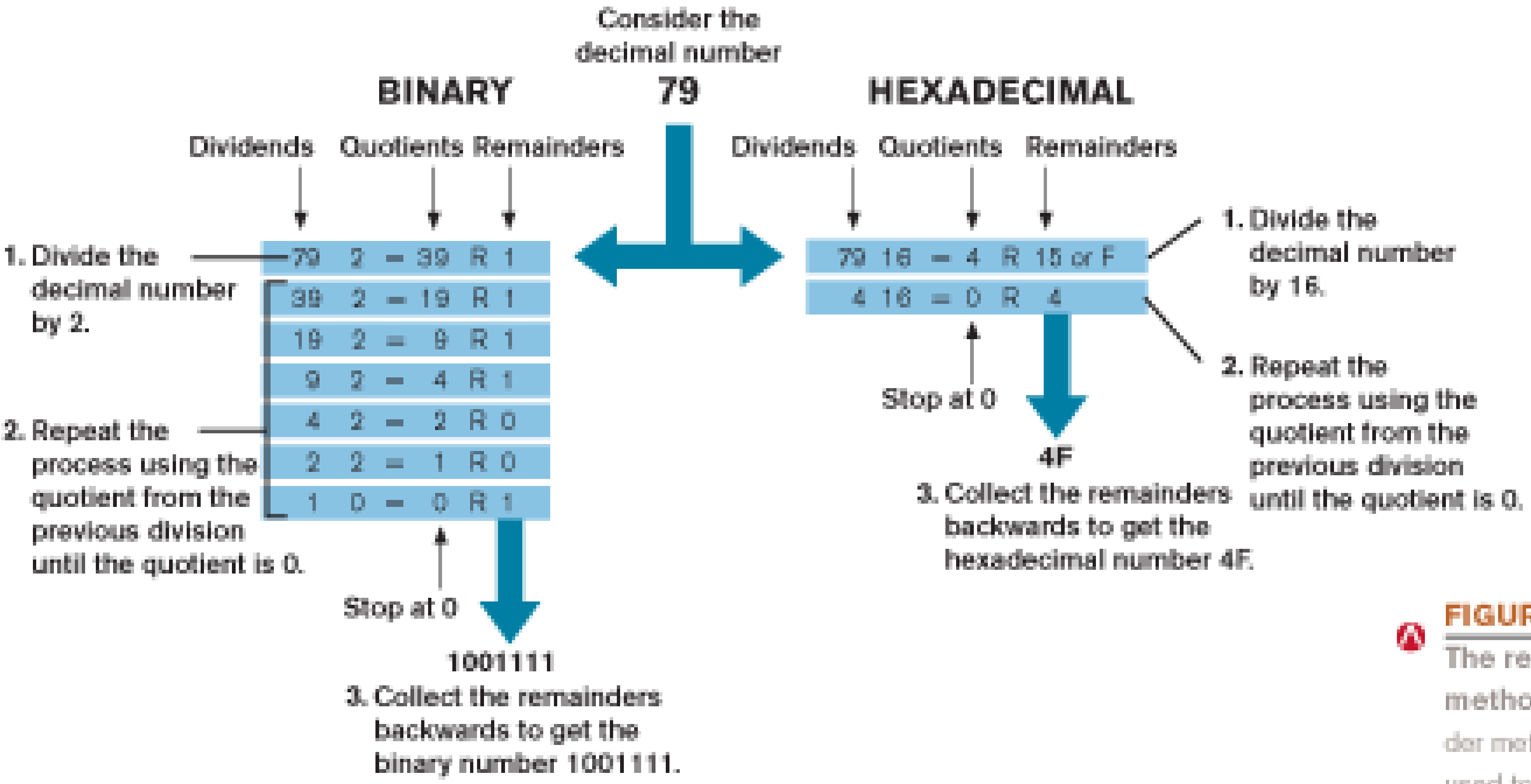
Let’s count!

In binary and hexidecimal



HEXADECIMAL CHARACTER	DECIMAL EQUIVALENT	BINARY EQUIVALENT
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

Base Conversions



Consider the decimal number 79

BINARY

Dividends Quotients Remainders

79 2 = 39 R 1
39 2 = 19 R 1
19 2 = 9 R 1
9 2 = 4 R 1
4 2 = 2 R 0
2 2 = 1 R 0
1 0 = 0 R 1

1. Divide the decimal number by 2.
2. Repeat the process using the quotient from the previous division until the quotient is 0.

Stop at 0

1001111

3. Collect the remainders backwards to get the binary number 1001111.

HEXADECIMAL

Dividends Quotients Remainders

79 16 = 4 R 15 or F
4 16 = 0 R 4

1. Divide the decimal number by 16.
2. Repeat the process using the quotient from the previous division until the quotient is 0.

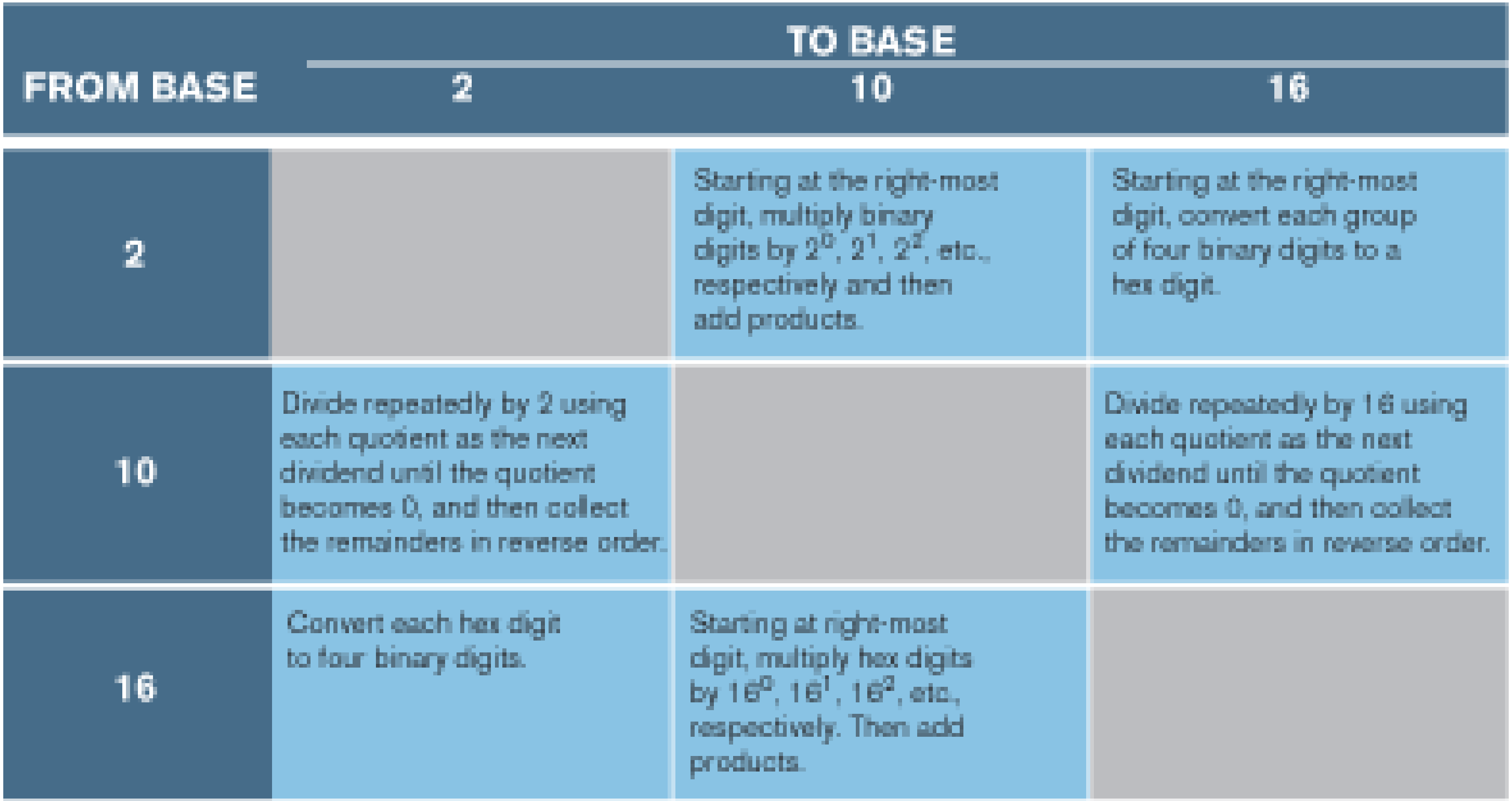
Stop at 0

4F

3. Collect the remainders backwards to get the hexadecimal number 4F.

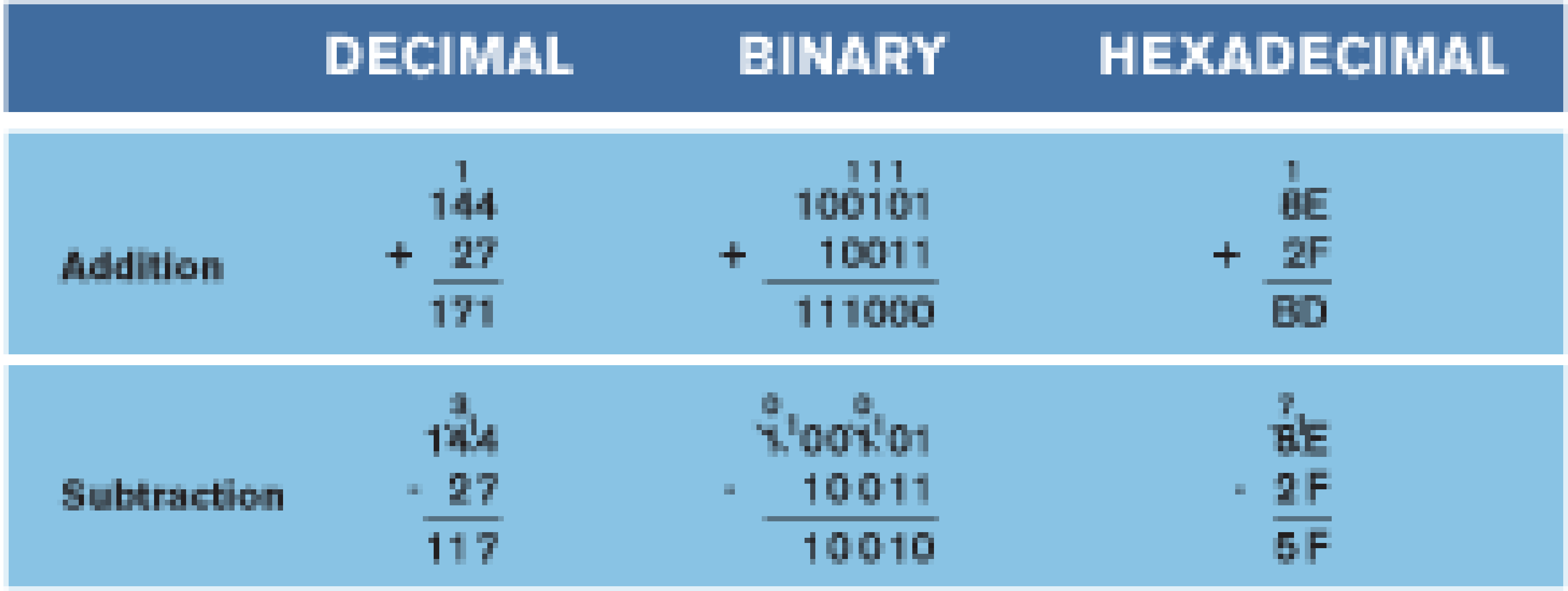
FIGURE
The rem
method
der meth
used to c

Conversions



	TO BASE		
FROM BASE	2	10	16
2		Starting at the right-most digit, multiply binary digits by 2^0, 2^1, 2^2, etc., respectively and then add products.	Starting at the right-most digit, convert each group of four binary digits to a hex digit.
10	Divide repeatedly by 2 using each quotient as the next dividend until the quotient becomes 0, and then collect the remainders in reverse order.		Divide repeatedly by 16 using each quotient as the next dividend until the quotient becomes 0, and then collect the remainders in reverse order.
16	Convert each hex digit to four binary digits.	Starting at right-most digit, multiply hex digits by 16^0, 16^1, 16^2, etc., respectively. Then add products.	

Calculations



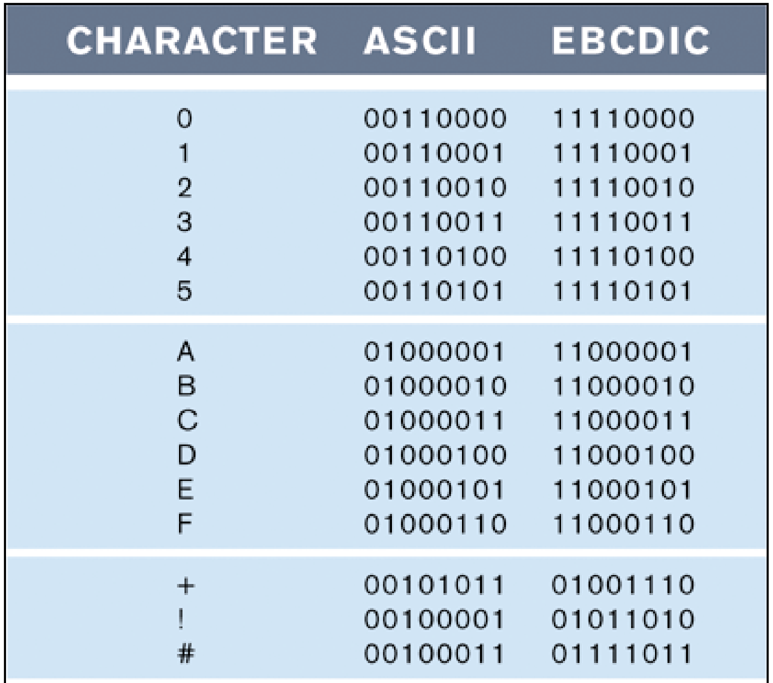
	DECIMAL	BINARY	HEXADECIMAL
Addition	1 144 + 27 171	111 100101 + 10011 111000	1 8E + 2F BD
Subtraction	3 144 - 27 117	0 0 100101 - 10011 10010	7 8E - 2F 5F

How do computers,
which only know binary,
know what keys you pressed?

Coding Systems for Text-Based Data

- ASCII and EBCDIC
  - ASCII (American Standard Code for Information Interchange): coding system traditionally used with PCs
  - EBCDIC (Extended Binary-Coded Decimal Interchange Code): developed by IBM, primarily for mainframe use
- Unicode: newer code (32 bits per character is common); universal coding standard designed to represent text-based data written in any language

Coding Systems for Text-Based Data



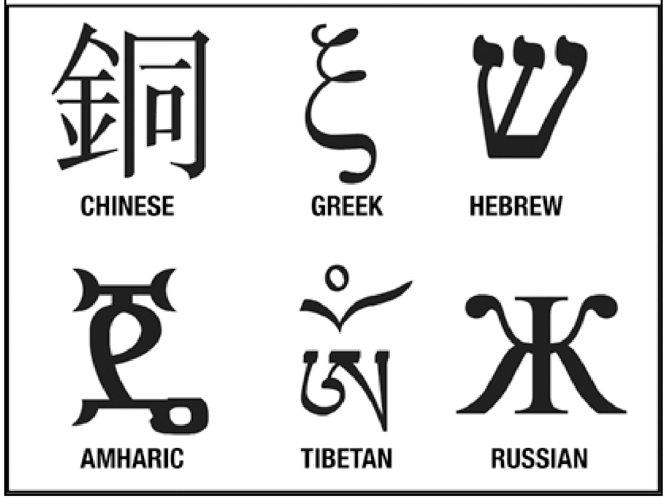
CHARACTER	ASCII	EBCDIC
0	00110000	11110000
1	00110001	11110001
2	00110010	11110010
3	00110011	11110011
4	00110100	11110100
5	00110101	11110101
A	01000001	11000001
B	01000010	11000010
C	01000011	11000011
D	01000100	11000100
E	01000101	11000101
F	01000110	11000110
+	00101011	01001110
!	00100001	01011010
#	00100011	01111011

FIGURE 2-4

Examples from the ASCII and EBCDIC codes. These common fixed-length binary codes represent all characters as unique strings of 8 bits.

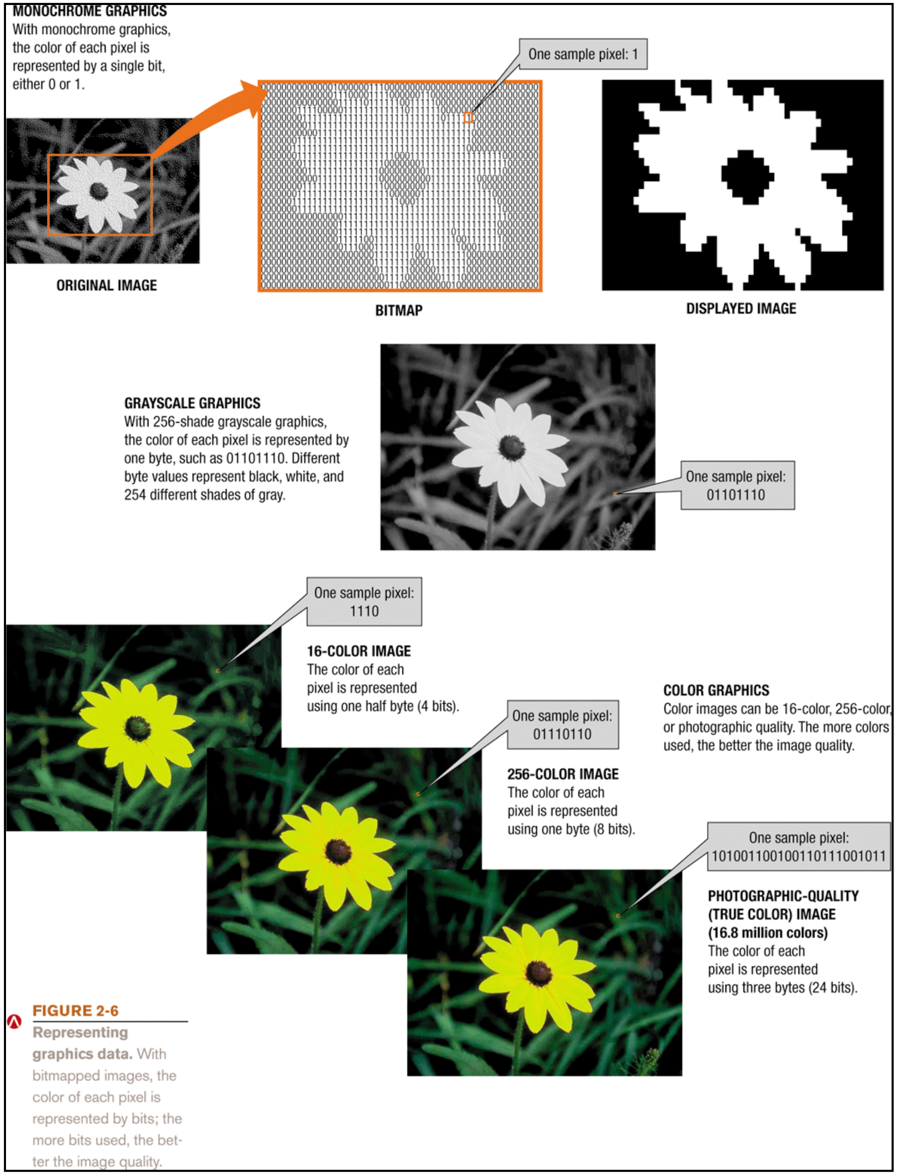
FIGURE 2-5

Unicode. Many characters, such as these, can be represented by Unicode but not by ASCII or EBCDIC.



Coding Systems for Other Types of Data

- Graphics (still images such as photos or drawings)
- Bitmapped images: A variety of bit depths are possible (4, 8, 24 bits)
- Vector-based images: Use mathematical formulas to represent images rather than a map of pixels



MONOCHROME GRAPHICS
With monochrome graphics, the color of each pixel is represented by a single bit, either 0 or 1.

GRAYSCALE GRAPHICS
With 256-shade grayscale graphics, the color of each pixel is represented by one byte, such as 01101110. Different byte values represent black, white, and 254 different shades of gray.

16-COLOR IMAGE
The color of each pixel is represented using one half byte (4 bits).

COLOR GRAPHICS
Color images can be 16-color, 256-color, or photographic quality. The more colors used, the better the image quality.

256-COLOR IMAGE
The color of each pixel is represented using one byte (8 bits).

PHOTOGRAPHIC-QUALITY (TRUE COLOR) IMAGE (16.8 million colors)
The color of each pixel is represented using three bytes (24 bits).

FIGURE 2-6
Representing graphics data. With bitmapped images, the color of each pixel is represented by bits; the more bits used, the better the image quality.

Coding Systems for Other Types of Data

- Audio data: Must be in digital form in order to be stored on or processed by a PC
  - Often compressed when sent over the Internet
    - MP3 files
- Video data: Displayed using a collection of frames, each frame containing a single graphical image
  - Amount of data can be substantial, but can be compressed
    - MPEG-2 files

Representing Programs: Machine Language

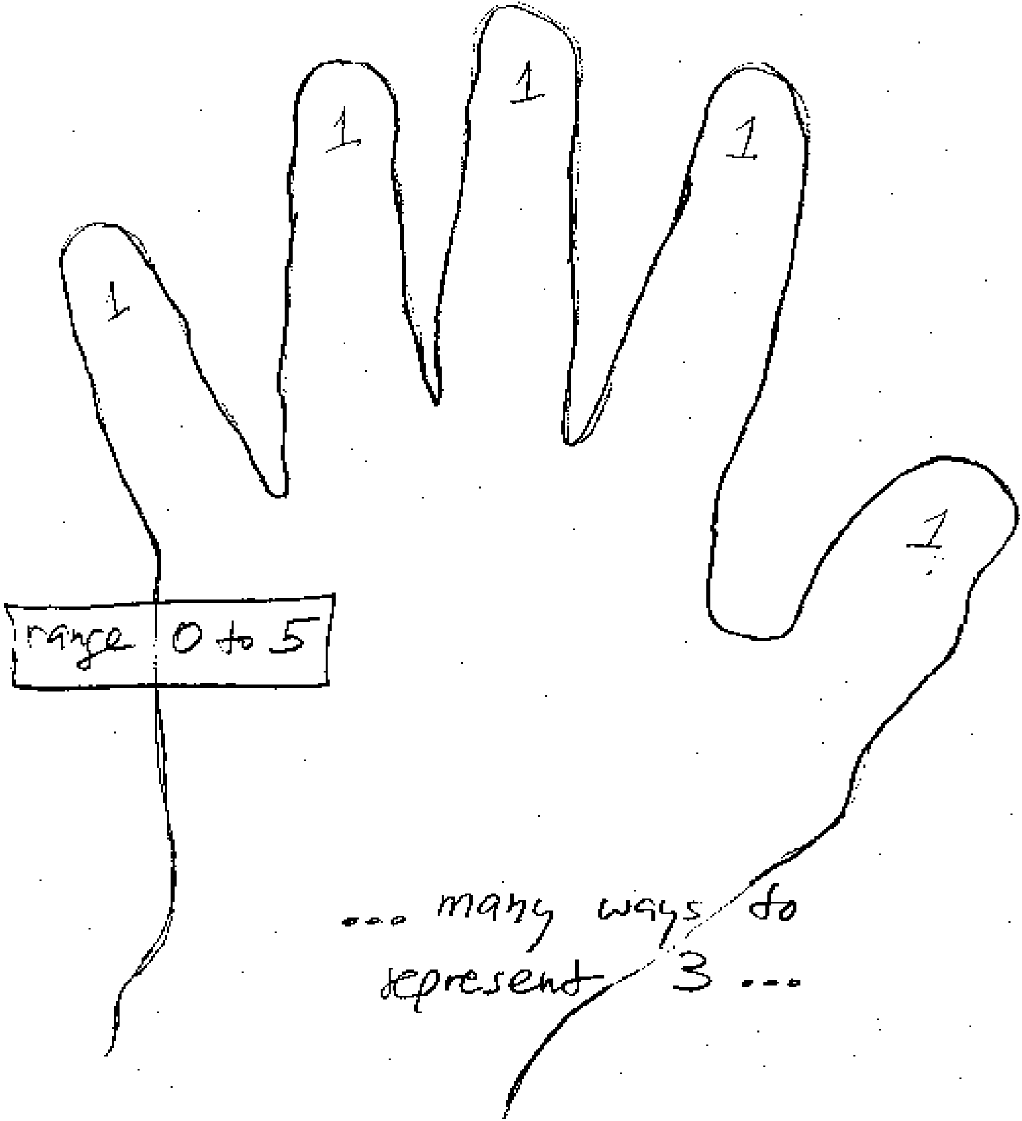
- Machine language: Binary-based language for representing computer programs the computer can execute directly
  - Early programs were written in machine language.
  - Today’s programs still need to be translated into machine language in order to be understood by the computer
- Most program are written in other programming languages
  - Language translators are used to translate the programs into machine language

Another way to count in Binary:
on your fingers!

How can a computer store a number

- Remember when you learned to count on your fingers?
- Each finger can be held up or down to represent on or off.
- By combining all ten fingers of both hands, you could group fingers in various ways to represent numbers in the range of 0 to 10.

each finger is worth 1



... many ways to represent 3 ...

Bits and counting

- Computers group memory bits together in order to get past the zero-one range limitation.
- Computers (typically) group bits in sets of eight -- a set of 8 bits is called a *byte*.

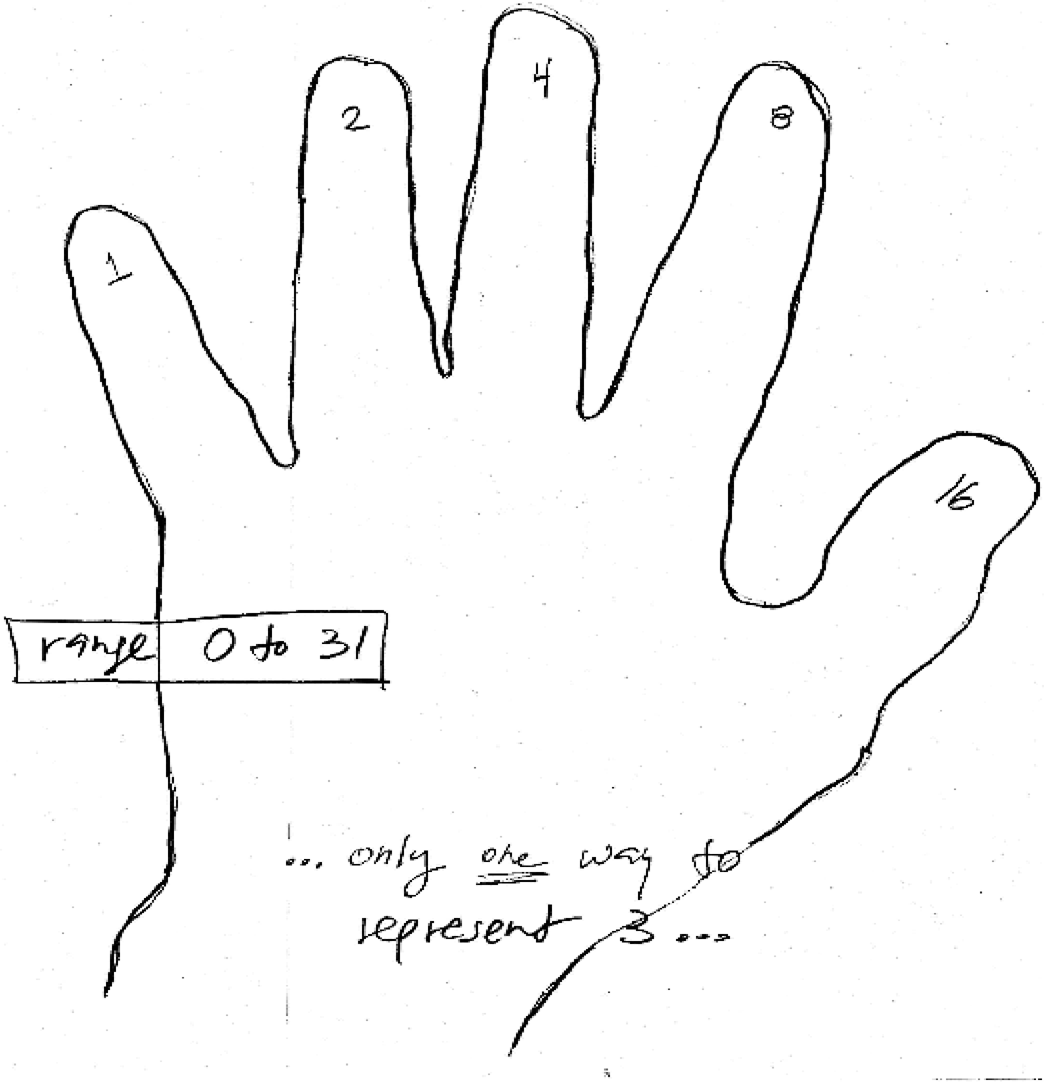
Ready set…count!

- On the fingers of one hand you learned to count from zero to five, and so you might think that a byte can be used to count from zero to eight.
- There are multiple ways to represent the number two, depending on which two fingers you hold up -- what a waste!

Learning how to count

- What if every possible combination of fingers held up or down represented a different number?
- There are 32 (that is, 2 x 2 x 2 x 2 x 2) unique combinations, ranging from all fingers down to all fingers up.
- We could actually count from 0 to 31 on the fingers of one hand, if we assigned a number to each finger configuration.
- Pinky-up could be 1, ring-finger-up could be 2, pinky-up and index-finger-up could be 9, etc.
- Extend this idea to two hands, and we can count to over 1000 -- now that's efficiency!

each finger worth 2X more than the one before



range 0 to 31

... only one way to represent 3 ...

Here’s how to do it

- Assign the value 1 to your pinky, 2 to your ring finger, 4 to your middle finger, 8 to your index finger, and 16 to your thumb.
- The key is to start with 1 and double it each time.
- For any finger held up, add its value to the total.
- For example, pinky and ring finger both up is 3 (that is, 1 + 2).
- Since some up-down finger combinations are painful or impossible, you might try holding your open hand over a flat surface, and touching the surface to represent "up" -- it's easier and faster to do.

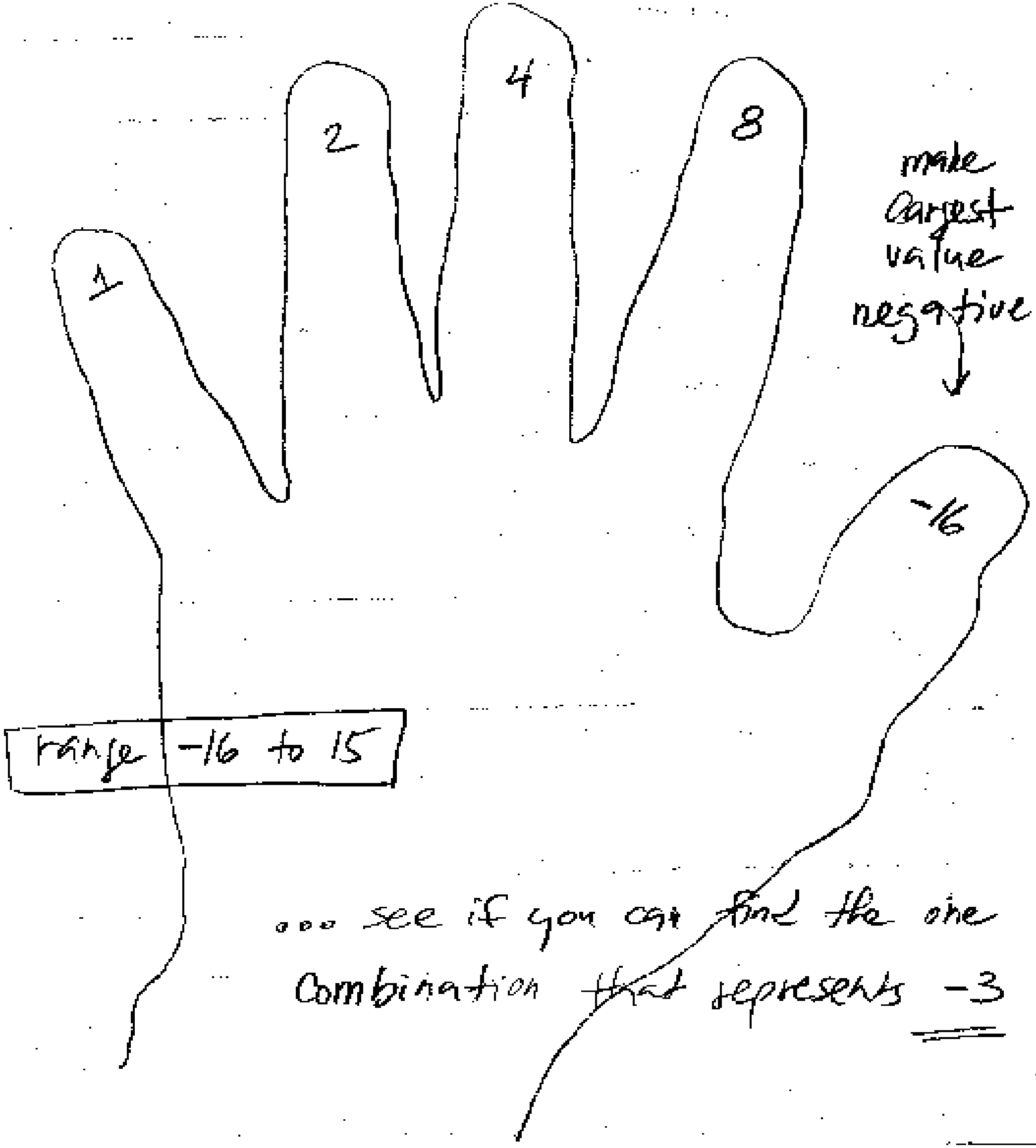
Negative numbers

- Use one finger to track whether the number is positive or negative -- a "sign bit".
- But this idea is just a little bit wasteful, because it results in two ways to represent zero!
- So a better solution is to make the largest bit represent the *negative* of its normal value.
- That is, let the thumb in a one-hand number be -16 instead of +16.
- Then all fingers except the thumb up represents 15 (that is, 1 + 2 + 4 + 8), thumb up only represents -16, thumb and pinky is -15, etc.

Negative numbers

- If we were to use two hands, the 16 thumb would go back to +16, and the thumb of the *second* hand would go negative (-512, actually).
- Integers are done this way in C++.
- The integer data type **int** uses 4 bytes for a total of 32 bits (in most of today's compilers).
- It can represent numbers in the range -2,147,483,648 to 2,147,483,647.
- The reason that C++ has additional data types for integers is that the other data types use different numbers of bytes, and therefore have different ranges!

negative numbers



make largest value negative!

range -16 to 15

... see if you can find the one combination that represents -3 !

Range limitations and data precision

Integers and range limitations

- Infinity: You can always add 1 to a number and get the next largest number, and continue doing so forever!
- Not so with computers -- depending on how many bytes they use to store a number, they eventually reach a limit.

Demonstration of round-off error

```
#include <iostream>
using namespace std;
int main()
{
  double sum = 0; // use this to accumulate the sum
  double x = 0.125; // add this ten times to get 1.25
  int i;
  for (i = 0; i < 10; i++)  sum += x;
  if (sum == 1.25)
    cout << sum << " equals 1.25\n"; // this is the answer
  else
    cout << sum << " does not equal 1.25\n";
  sum = 0; // reset for another summation
  x = 0.126; // add this ten times to get 1.26
  for (i = 0; i < 10; i++)  sum += x;
  if (sum == 1.26)
    cout << sum << " equals 1.26\n";
  else
    cout << sum << " does not equal 1.26\n"; // THIS IS THE ANSWER!
  return 0;
}
```

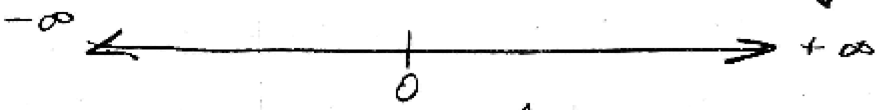
beyondInfinity.cpp

```
…
//libraries
#include <iostream>
using namespace std;
…
int main()
{
  //data
  int a = 2147483645; // 2 less than the maximum int
  int i;

  //count
  for (i = 0; i < 5; i++)
  {
    a++;
    cout << a << endl;
  } // for

} // main
```

Output

```
2147483646
2147483647
-2147483648
-2147483647
-2147483646
```

The Number Line



infinite range

a "continuum" of values, infinitely divisible

Problem: finite number of bits

↓

finite number of unique combos

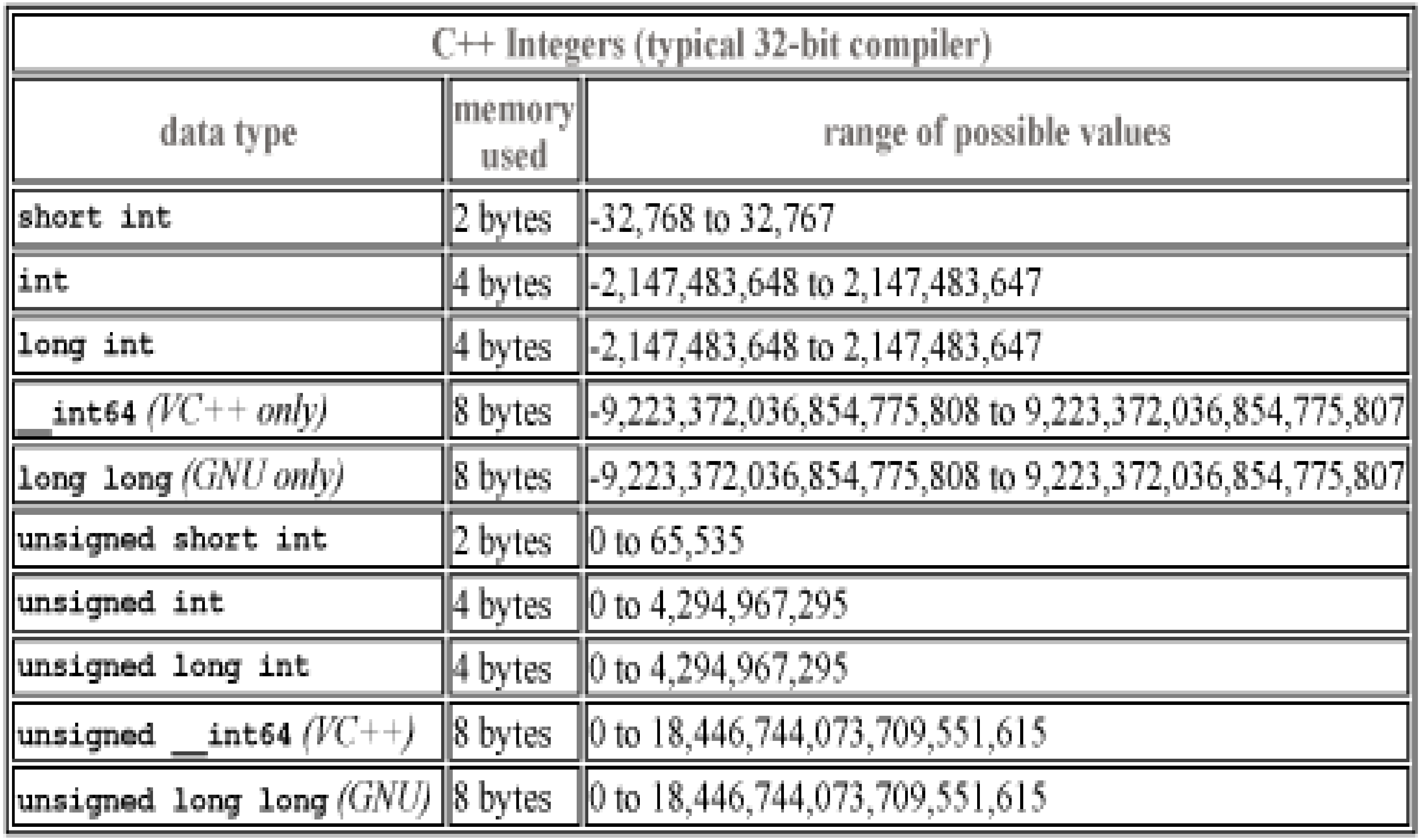
↓

CANNOT REPRESENT EVERY NUMBER!

beyondInfinity.cpp

- Integer values that would exceed the maximum limit wrap to the lower limit and continue from there.
- Similarly, values that would be below the minimum limit wrap to the upper.
- *Note that commas are not allowed in integer values in C++!*

Range Limitations For Other Integer Data Types



C++ Integers (typical 32-bit compiler)

data type	memory used	range of possible values
`short int`	2 bytes	-32,768 to 32,767
`int`	4 bytes	-2,147,483,648 to 2,147,483,647
`long int`	4 bytes	-2,147,483,648 to 2,147,483,647
`__int64` *(VC++ only)*	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`long long` *(GNU only)*	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned short int`	2 bytes	0 to 65,535
`unsigned int`	4 bytes	0 to 4,294,967,295
`unsigned long int`	4 bytes	0 to 4,294,967,295
`unsigned __int64` *(VC++)*	8 bytes	0 to 18,446,744,073,709,551,615
`unsigned long long` *(GNU)*	8 bytes	0 to 18,446,744,073,709,551,615

Range Limitations For Other Integer Data Types

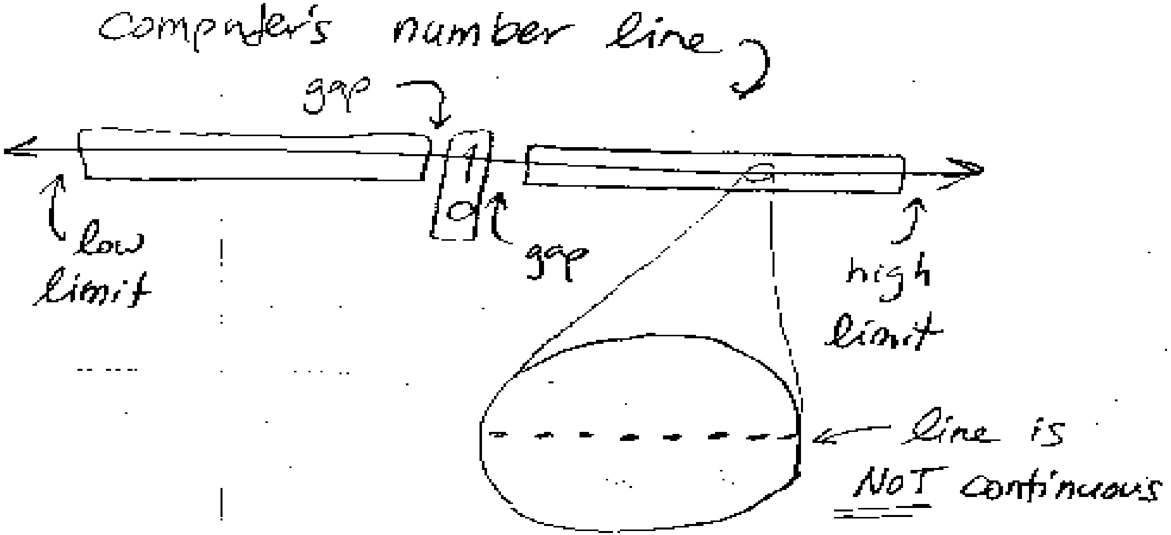
- Data type names can consist of more than one word, indicating the number of bytes used, and whether or not the negative range is allowed.
- A peculiarity of C++ is that the "long" modifier on the data type name appears to have no effect.
- That is because the number of bytes *varies* among compilers and operating systems for the same data type, and all that the language requires is that a "long int" be *at least* as big as a regular **int** .
- Here's how to find out how many bytes are used by your compiler for any specific data type:

  cout << sizeof (int) << endl; //for example.

What about fractional numbers?

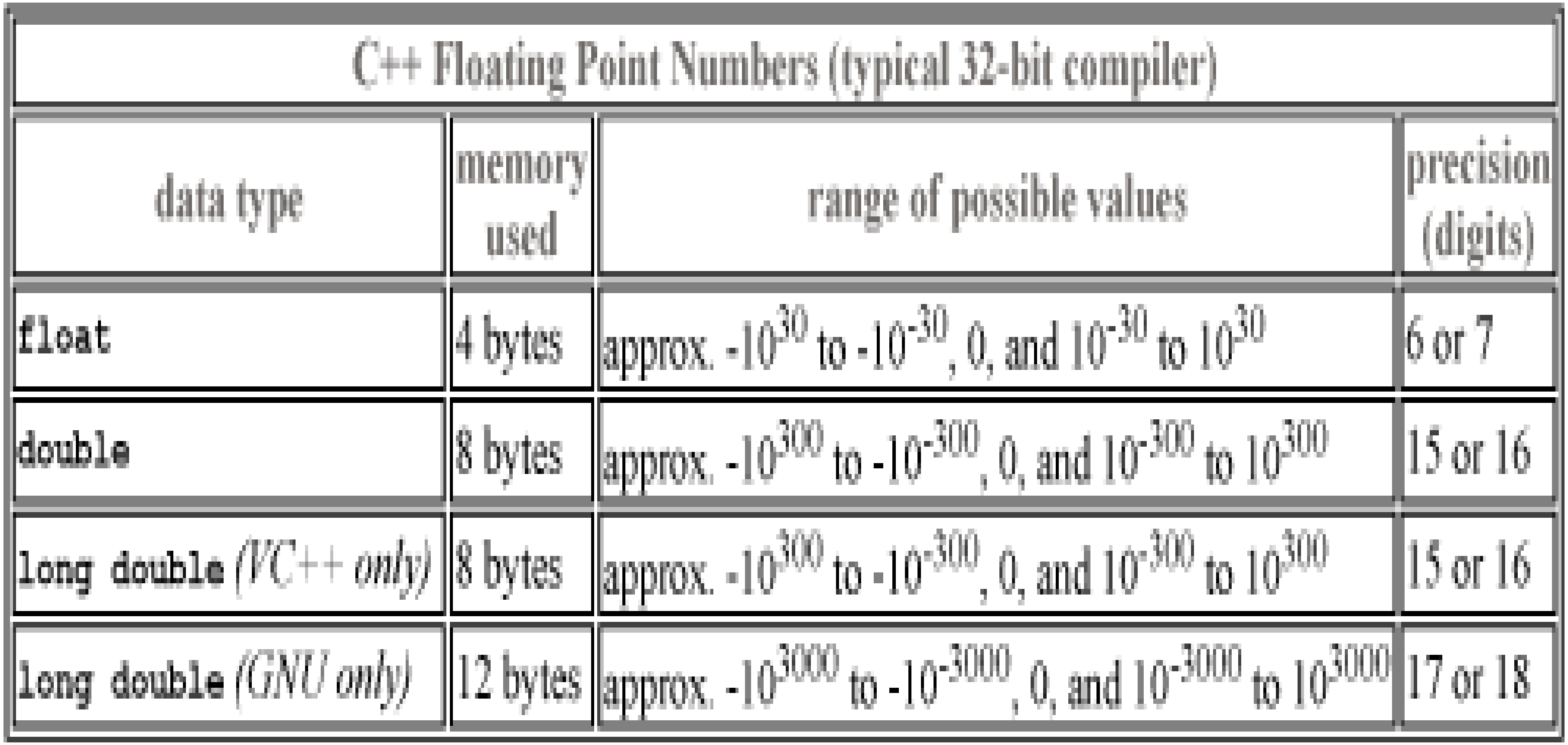
- Modify the bit-counting approach so that instead of representing whole numbers, we represent fractional values instead.
- That is, the pinky's value is ½, ring finger is 1, index finger is 2, etc. -- or we could start with ¼ or less for the pinky.
- This is basically what C++ does, and for this there needs to be a second set of data types -- ones whose first bit is a fraction instead of 1.
- Actually, the value of the first bit varies, depending on the size of the value being stored -- for example, why stop at ¼ if the number stored is only 10?

Floating Point Number Representation

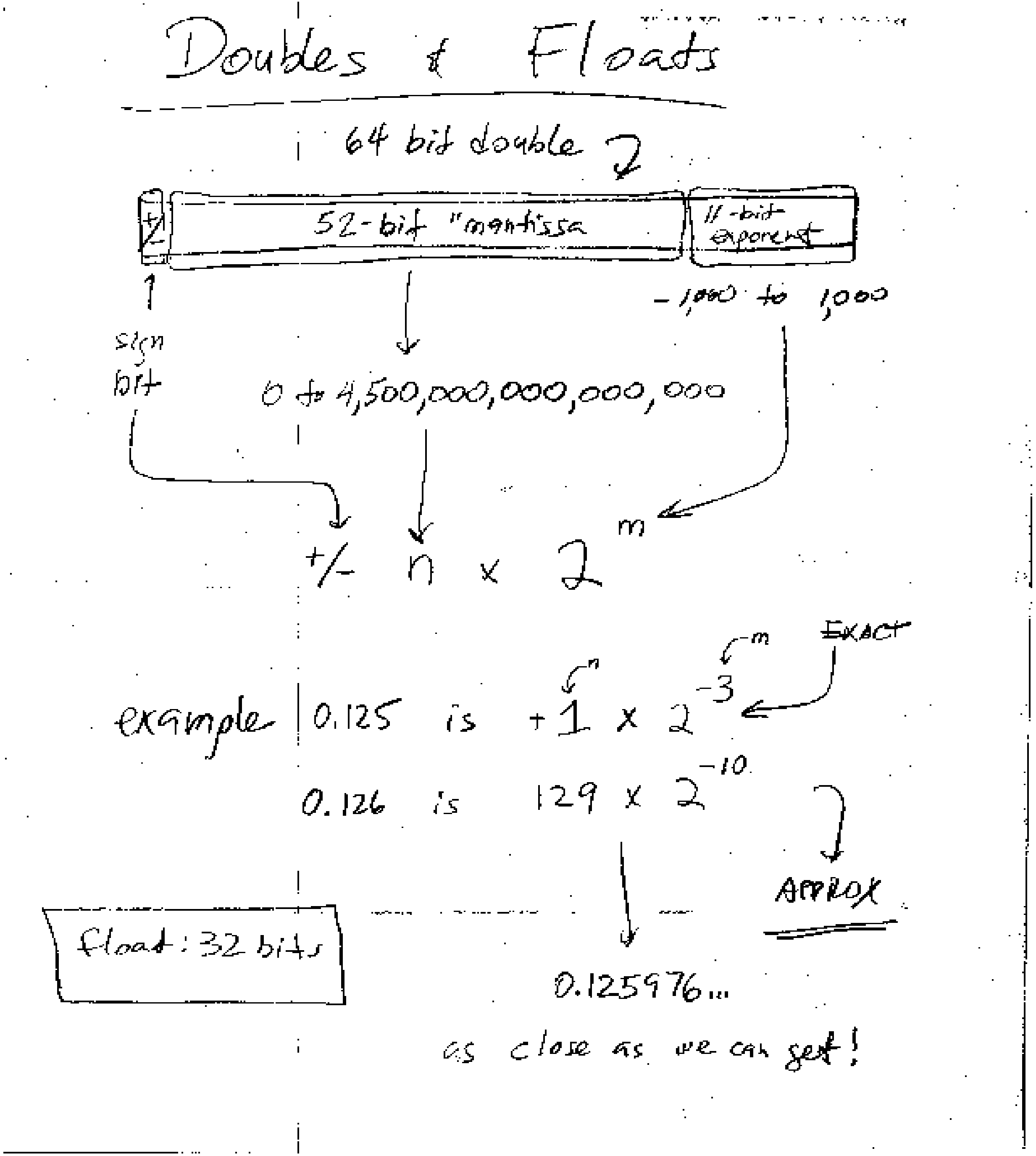


1024

Floating point numbers



C++ Floating Point Numbers (typical 32-bit compiler)			
data type	memory used	range of possible values	precision (digits)
`float`	4 bytes	approx. -10^{30} to -10^{-30}, 0, and 10^{-30} to 10^{30}	6 or 7
`double`	8 bytes	approx. -10^{300} to -10^{-300}, 0, and 10^{-300} to 10^{300}	15 or 16
`long double` *(VC++ only)*	8 bytes	approx. -10^{300} to -10^{-300}, 0, and 10^{-300} to 10^{300}	15 or 16
`long double` *(GNU only)*	12 bytes	approx. -10^{3000} to -10^{-3000}, 0, and 10^{-3000} to 10^{3000}	17 or 18



Doubles & Floats

64 bit double

+/-	52-bit "mantissa	11-bit exponent
1 sign bit	0 to 4,500,000,000,000,000	-1,000 to 1,000

$$\pm\ n \times 2^{m}$$

example | 0.125 is $+\overset{n}{1} \times 2^{\overset{m}{-3}}$ EXACT

0.126 is 129×2^{-10} APPROX

0.125976...

as close as we can get!

float: 32 bits

Precision

- Precision is an issue for floating point numbers, but not for integers.
- That is why we need separate data types for both -- so we can choose the precision of integers at the expense of fractional capabilities, or the fractional capabilities of floating points at the expense of precision.
- It turns out that setprecision(20) simply directs that the output be displayed with the number of decimal digits specified.
- If the number is stored with less precision than is asked for in the output, the computer "makes up" the digits that it does not know exactly!
- It's better than generating a error and terminating the program at that point.

Output Precision vs. Floating Point Precision

So this loop will execute 10 iterations.

```
double x;
for (x = 0; x != 2.5; x += 0.25)  cout << x << endl;
```

Whereas this loop won't end even though it looks like it should.

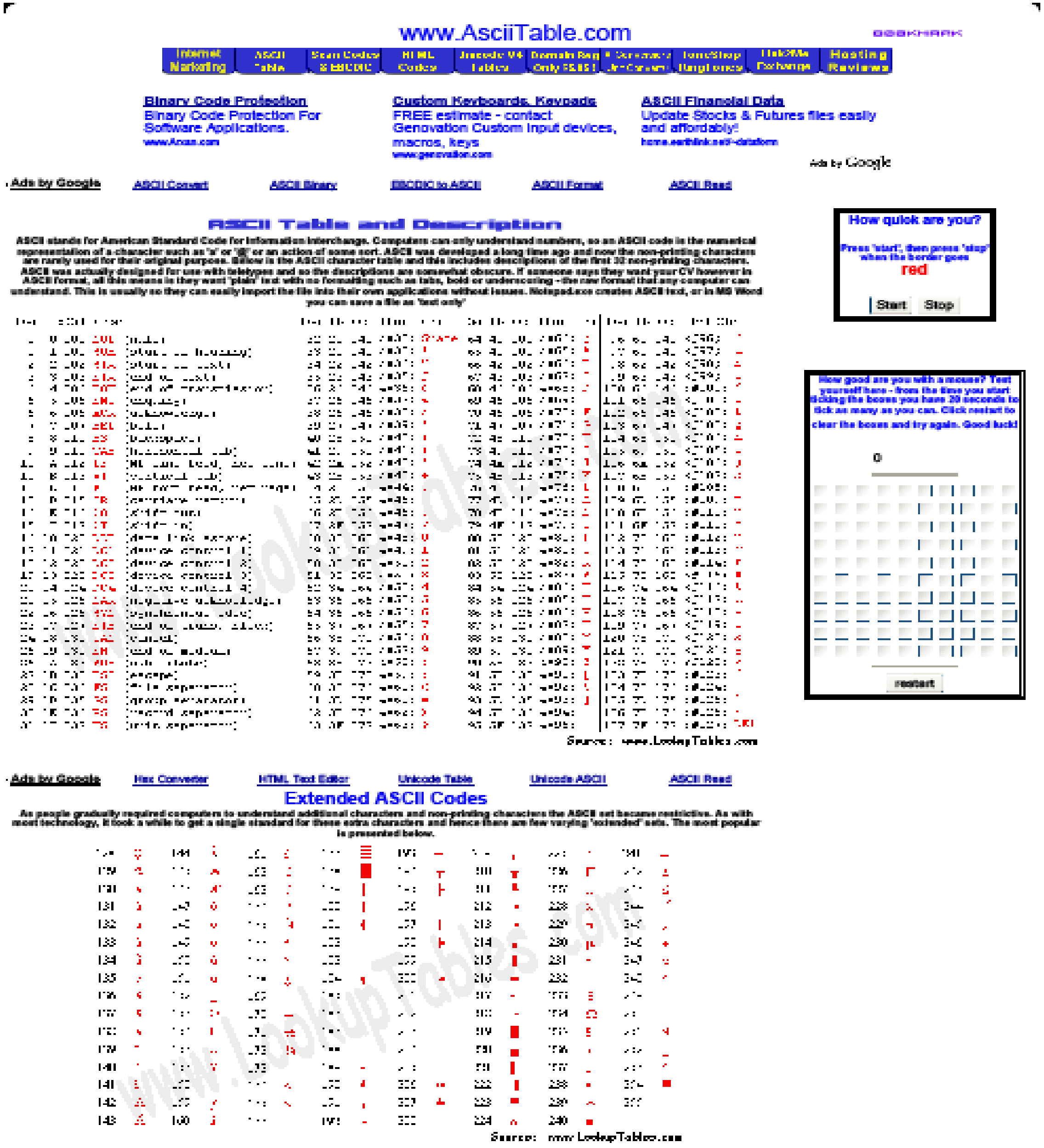
```
double x;
for (x = 0; x != 1; x += 0.1) cout << x << endl;
```

Representing characters

- Numbers are used to stand for characters, and the data type distinguishes between the two.
- The number 65 when stored in an **int** is 65, but when stored in the single-character data type **char** , it's **'A‘.**
- The data type **char** takes 1 byte of memory, which gives it enough bits 256 unique combinations.
- That's enough for the entire English alphabet, both uppercase and lowercase, plus punctuation symbols and digits, and a lot more.
- The standard for determining which numbers stand for which characters is called the ASCII (American Standard Code for Information Interchange) standard.

Representing characters

- Text stored in "string" variables is actually a series of **char** variables.
- The number of bytes needed to store a "string" variable depends on the number of characters of text stored in it.
- There is one **char** per character, including spaces (code 32), plus one special character to mark the end: a *null* (code 0).
- So even empty text has one character.



www.AsciiTable.com

ASCII Table and Description

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

How quick are you?
Press 'start', then press 'stop' when the border goes red
Start
Stop
How good are you with a mouse? Test yourself here - from the time you start ticking the boxes you have 20 seconds to tick as many as you can. Click restart to clear the boxes and try again. Good luck!
0
restart

Extended ASCII Codes

As people gradually required computers to understand additional characters and non-printing characters the ASCII set became restrictive. As with most technology, it took a while to get a single standard for these extra characters and hence there are few varying 'extended' sets. The most popular is presented below.

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	ÿ	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
141	ì	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Source: www.LookupTables.com

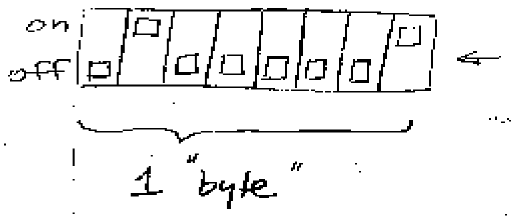
What happens if you add or subtract from a **char** variable?

- That is, for **char c = 'A';** , what does the expression **c++** do?
- It advances the character to the next one in the ASCII sequence, **'B'** (code 66)!
- So the following code block prints the alphabet:

```
char c = 'A';
for (c = 'A'; c <= 'Z'; c++) cout << c << ' ';
cout << endl;
```

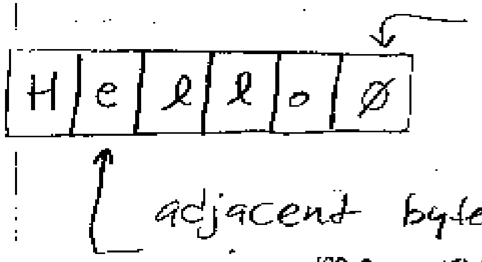
Storing Text

char 'A'



on/off bits that represent uppercase A

string



"null" – all bits off

adjacent bytes in memory

New data type

Data type Bool

- There is another data type that is separate from the numerics and text.
- It is the only thing that a bit can represent by itself: two exact opposites.
- Ironically, since bits are grouped as bytes in computer memory, it still takes a whole byte's worth of bits to do this.
- After all, to use just one finger, you still have to involve a hand!
- Booleans are used as "flags".
- The data type name is **bool** , and the two possible values are **true** and **false** .
- No, there are not any "yes", "no", "left", or "right" values -- you just have to decide which of a pair of opposites is to be represented by **true** and which is **false** .

See example:
keepingScoreBool.cpp

Literal Values
Truncation
data types
typecasting

Literal values

- Values written directly into code are called *literal values*.
- Depending on how they are written, they have a C++ data type associated with them.
- For example, whole numbers are of data type **int** (like **0** , **100** , or **-1**).
- Decimal numbers are of data type **double** (like **0.0** , **3.14159** , or **-40.0**).

Literal values

- Literals are usually decimal values, but whole number literals can also be octal (base 8) or hexadecimal (base 16).
- To write a number as base 8 octal, pre-pend a zero (like **011** , which is the same as decimal 9, or **077** which is the same as decimal 63 -- 7 eights plus 7 ones).
- Note that 08 and 09 are undefined and will cause compiler errors, because the digits 8 and 9 do not exist in base 8.
- To write a number as base 16 hexadecimal, prepend 0x (like **0xFFFF** or **0xABCD1234**).
- Note that A through F are digits added to the decimal set of 0 through 9 in order to get 16 digits for the base 16 numbering system.

Literal values

- Literal **float** values are written by appending uppercase or lowercase F (like **3.14159f** or **100.0F**).
- Unsigned data type values are written by appending uppercase or lowercase U. Longs are written by appending uppercase or lowercase L.

Literal values

- Literal **char** values are written as single characters, with single-quotes (or apostrophes) before and after (like **'A'**).
- Literal **string** values are written as any number of characters, with quote marks before and after (like **"Hello"**).
- Literal **bool** values are **true** and **false** .

Literal constants

ints: 0, 100, -1

as octal: 00, 07, 012345

as hexidecimals: 0xABCD1234, 0xF, 0xC0C0

doubles: 0.0, 3.14159265, -40.0

floats: 0F, 3.14F, -40.0F

longs: 0L, 100L, -1L

chars: 'A', '\n', '4'

bools: true, false

Using data types in C++

- C++ has a variety of data types, and for each variable in your program you have to decide which data type to use.
- Also, the data type for the result of a math operation depends on the data types of the values in the operation.
- Here's the problem, if you have 3 **int** values (for example, **a** , **b** , and **c**) and you want the average, the expression **(a + b + c) / 3** will truncate (that is, lose) the fractional part of the answer, because that is what integer division does.

Typecasting

- You can write code that *forces* type casting -- this is a way to convert a value of one type into a related type, such as an **int** to a **double** .

Typecasting in expressions

```
int a = 100;
int b = 17;
int c = -29;
double average1 = (a + b + c) / 3; // LOSES THE
FRACTIONAL RESULT
double average2 = double(a + b + c) / 3; // that's
better!
double average3 = (a + b + c) / 3.0; // this works,
too
```

Typecasting of parameter values

- Function parameter lists specify the data type for values used in the parentheses of a call.
- So, what if you have a variable of a certain data type, and you want to call a function using that variable, but the data type specified for the parameter list is not the same?
- For example, you have an **int** value and you want to take its square root using the C++ library function **sqrt** , whose parameter list specifies **double** .
- Use type casting to convert a value of one type into a related type, such as an **int** to a **double**

Typecasting of parameter values

```
int x = 100;
double y1 = sqrt(x); // WRONG
double y2 = sqrt(double(x)); // that's better!
```

This *looks like* a function call inside a function call (which is okay to do in C++).

The inner "call“ returns a **double** version of **x** , which is valid for the square root call.

Type casting to truncate the fractional part of a floating point number

```
double x = 456.999;
int y = int(x); // value is 456
```

...or...

```
int y = (int)x; // another way to write it
```

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns
- Understanding Computers Today and Tomorrow by Deborah Morley and Charles Parker