Data Representation

Number Systems

(from an altitude of 10,000 ft)

Roman number system (Roman numerals):

- uses the symbols I, V, X, L, C, D, M…
- non-positional (I means ‘1’ and X means ‘10’ wherever they occur in a number)

Decimal number system (base 10):

- uses the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 (no unique symbol for ten!)
- a positional number system (position in the number determines value)
  - in the number 100, ‘1’ has the value of “one hundred”; in 1000, ‘1’ represents “one thousand”

Binary number system (base 2):

- uses only the symbols 0 and 1
- also positional
  - in 100_2, ‘1’ has the value of “four”, while in 10000_2, ‘1’ signifies “sixteen”

Binary

(from an altitude of 5,000 ft)

Binary is simply another way of representing numbers.

$1_2 = 1_{10}$	$1_{10} = 1_2$
$10_2 = 2_{10}$	$10_{10} = 1010_2$
$100_2 = 4_{10}$	$100_{10} = 1100100_2$
$1000_2 = 8_{10}$	$1000_{10} = 1111101000_2$
$10000_2 = 16_{10}$	
$100000_2 = 32_{10}$	
$1000000_2 = 64_{10}$	

Why is this relevant?

- because binary uses only 2 symbols, it can be implemented by dual-state systems like circuits (computers!)

Some random information:

- the word “bit” is short for binary digit, and a “byte” is 8 bits. Randomness FTW!

Data Representation

Computers think in terms of 1's and 0's (binary!)

- on / off
- high / low
- open / closed

Humans… well, I don't know HOW we think, but we do it differently!

- with words and language, with our senses, spatially...

So, how can a computer represent data that's meaningful to us?

- 0110100100100000011000010111010001100101001000000110011101101100011101010110010
  1001000000110000101110011001000000110000100100000011000110110100001101001011011
  0001100100 (true story)
- with magic?
- luckily, this problem has been largely solved for us (yay for other people!).

Data Representation

Imagine you're a computer (but not one that's 18+ years old--yikes!)

How would you represent (using binary):

- an `int`
  - numeric value… easy
- a `double`
  - numeric value… not as easy as you might think
- a `bool`
  - two-state type using a number system with 2 symbols? Easy!
  - can't address an individual bit, though, so somewhat wasteful...
- a `char`
  - hmmm… I'll show you on some of the next slides! =)
- a `string`
  - strings are 0 or more `char`acters grouped together
  - we'll learn about arrays and objects later

Data Representation

Example values for *32-bit* architecture:

Type	Bytes (Bits)	Range of Values
`bool`	1 (1)	0 - 1
`char`	1 (8)	-128 - 127
`int`	4 (32)	-2,147,483,648 - 2,147,483,647
`unsigned int`	4 (32)	0 - 4,294,967,647
`double`	16 (128)	-2.2251×10^{308} - 2.2251×10^{308}

In a *64-bit* architecture, an `int` can range from:

–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

(-2^{63} to $2^{63}-1$ or roughly -10^{19} to 10^{19})

Take-Away message:

All data types are stored the same way:
a bunch of ones and zeros.

Type-casting

All data types are stored on your computer the same way...

- this means we should be able to switch between types (e.g., from a double to an int)
- and, sure enough, we can!

General syntax (really easy):

```
data_type(value) // converts 'value' to data_type
```

Example:

type-casting to int

```
double number = 3.12345;
cout << "Behold! An integer: " << int(number) << endl;

// cast 65 to a char (why we can do this will become apparent soon)
cout << "An uppercase letter: " << char(65) << endl;
```

The char data type

A char variable is internally stored by C++ *as an integer*

- output streams (such as `cout`) and input streams (like `cin`) implicitly know how to convert between the number and the symbol it represents, and vice versa
- This number represents the "ASCII code" of the character

This has some interesting effects in C++:

- we can assign a char variable a numeric value:

```
char symbol = 65;  // symbol now holds the value for 'A'
char symbol = 'A'; // <-- just like if we had done this
```

- we can assign an int variable a character value:

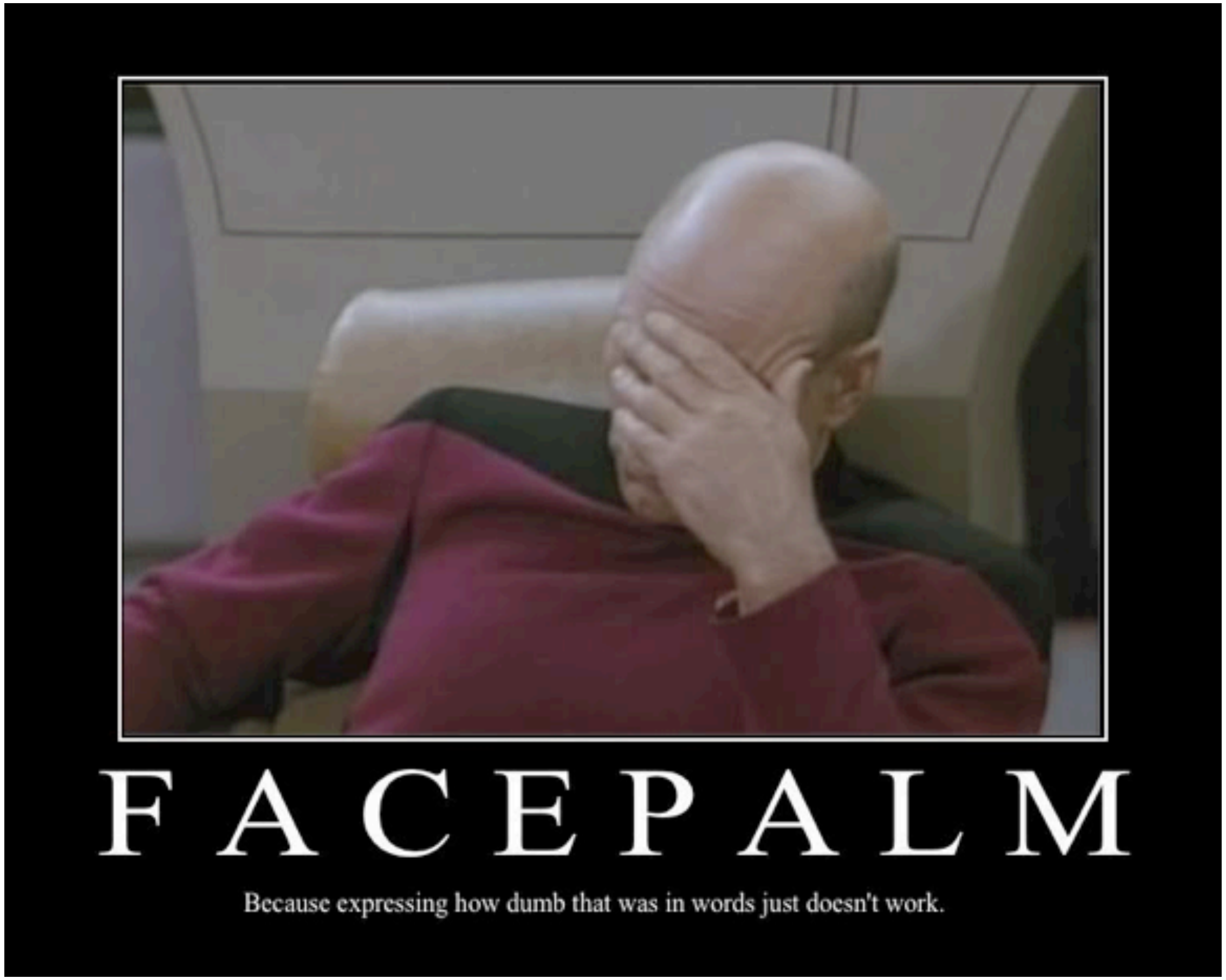
```
int number = 'A';  // number now holds the value 65
int number = 65;   // <-- just like if we had done this
```

The char data type

Strange but true (so be careful):

```
'7' != 7;    // the character '7' does *NOT* equal the number 7

'7' == 55;   // '7' actually equals the number 55 (its ASCII code)
```



ASCII Table

American Standard Code for Information Interchange

000		(nul)	016	►	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	☺	(soh)	017	◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	☻	(stx)	018	↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥	(etx)	019	‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦	(eot)	020	¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	♣	(enq)	021	§	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠	(ack)	022	▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	•	(bel)	023	↨	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	◘	(bs)	024	↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009		(tab)	025	↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010		(lf)	026		(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂	(vt)	027	←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	♀	(np)	028	∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013		(cr)	029	↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♫	(so)	030	▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	☼	(si)	031	▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

A few notes:

- Observe that digits (48-57), uppercase letters (65-90), and lowercase letters (97-122) occur in contiguous blocks.
- Codes 0-31 are considered "non-printing" characters (eg: `ACK`, `EOF`, `NUL`, `BEL`).
- You certainly don't need to memorize the table; just learn its properties.

The char data type

So, think of a char as an integer type that cin and cout treat specially!

We can do math with chars, just like we would any other int type:

```
char c1 = ‘A’;      // c1 is ‘A’ (65)
char c2 = ‘A’ + 1;  // c2 is ‘B’ (65 + 1 = 66)
char c3 = ‘C’ + 32; // c3 is ‘c’ (67 + 32 = 99)
char c4 = ‘d’ - 32; // c4 is ‘D’ (100 - 32 = 68)
char c5 = ‘5’ + 4;  // c5 is ‘9’ (53 + 4 = 57)
```

You can prove it to yourself (if you don’t believe me):

```
// outputs: A B c D 9
cout << c1 << “ ” << c2 << “ ” << c3 << “ ” << c4 << “ ” << c5 << endl;
```

The char data type

So, think of a char as an integer type that cin and cout treat specially!

Be aware:

- the ASCII codes for digits are not adjacent to the ASCII codes for upper case letters:

```
char wtf_1 = ‘5’ + 5;  // wtf_1 is ‘:’ (53 + 5 = 58)
```

- the codes for uppercase letters are not adjacent to those for lowercase letters:

```
char wtf_2 = ‘Z’ + 1;  // wtf_2 is ‘[’ (90 + 1 = 91)
```

Just so you know:

- “`wtf`” = “`Win The Fight`” (I promise)

Uninitialized Values

Remember:

- variables with primitive types (`bool`, `int`, `char`, `double`) that are not assigned an initial value <u>will still have a value</u>
- we just don't know what that value is (could be `8.8e-311` or `0` or anything)
- do NOT depend on an uninitialized variable having a specific value (this is a logic error!)
- uninitialized values are called *garbage values*

The exception:

- the string data type is *not* a primitive type, so it does not follow this rule.
- the string type is a *user-defined* data type, predefined by other people for our use.
- its initial value is an *empty string* (“”, a string that is 0 characters long)--you CAN depend on this behavior in your programs.