

# CSC 2400: Computer Systems

## Bits, Bytes and Data Types

Slides by Jennifer Rexford from Princeton University, slightly modified by Mirela Damian. <sup>1</sup>

## Goals for this Lecture

- Binary number system
  - Why binary?
  - Converting between decimal and binary
  - ... and octal and hexadecimal number systems
- Finite representations of binary integers
  - Unsigned and signed integers
  - Integer addition and subtraction
- Bitwise operators
  - AND, OR, NOT, and XOR
  - Shift-left and shift-right
- The C integral data types
  - char, short, int, long
  - signed and unsigned variants

2

## Why Bits (Binary Digits)?

- Computers are built using digital circuits
  - Inputs and outputs can have only two values
  - True (high voltage) or false (low voltage)
  - Represented as 1 and 0
- Can represent many kinds of information
  - Boolean (true or false)
  - Numbers (23, 79, ...)
  - Characters ('a', 'z', ...)
  - Pixels, sounds
  - Internet addresses
- Can manipulate in many ways
  - Read and write
  - Logical operations
  - Arithmetic

3

## Base 10 and Base 2

- Decimal (base 10)
  - Each digit represents a power of 10
  - $4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$
- Binary (base 2)
  - Each bit represents a power of 2
  - $10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22$

Decimal to binary conversion:

Divide repeatedly by 2 and keep remainders

$$12 / 2 = 6 \quad R = 0$$

$$6 / 2 = 3 \quad R = 0$$

$$3 / 2 = 1 \quad R = 1$$

$$1 / 2 = 0 \quad R = 1$$

$$\text{Result} = 1100$$

4

## Writing Bits is Tedious for People

- Octal (base 8)
  - Digits 0, 1, ..., 7
- Hexadecimal (base 16)
  - Digits 0, 1, ..., 9, A, B, C, D, E, F

0000 = 0	1000 = 8
0001 = 1	1001 = 9
0010 = 2	1010 = A
0011 = 3	1011 = B
0100 = 4	1100 = C
0101 = 5	1101 = D
0110 = 6	1110 = E
0111 = 7	1111 = F

Thus the 16-bit binary number

1011 0010 1010 1001

converted to hex is

B2A9

5

## Representing Colors: RGB

- Three primary colors
  - Red
  - Green
  - Blue
- Strength
  - 8-bit number for each color (e.g., two hex digits)
  - So, 24 bits to specify a color
- In HTML, on the course Web page
  - Red: `<font color="#FF0000">CSC 2400 – Computer Systems</font>`
  - Blue: `<font color="#0000FF"> ... </font>`
- Same thing in digital cameras
  - Each pixel is a mixture of red, green, and blue

6

## Finite Representation of Integers

- Fixed number of bits in memory
  - Usually 8, 16, or 32 bits
  - (1, 2, or 4 bytes)
- Unsigned integer
  - No sign bit
  - Always 0 or a positive number
  - All arithmetic is modulo  $2^n$
- Examples of unsigned integers
  - 00000001  $\rightarrow$  1
  - 00001111  $\rightarrow$  15
  - 00010000  $\rightarrow$  16
  - 00100001  $\rightarrow$  33
  - 11111111  $\rightarrow$  255

7

## Adding Two Integers

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column

	<u>Base 10</u>			<u>Base 2</u>		
	1	9	8	0	1	1
+	2	6	4	0	0	1
Sum	4	6	2	1	0	0
Carry	0	1	1	0	1	1

8

## Binary Sums and Carries

a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0

XOR  
("exclusive OR")

a	b	Carry
0	0	0
0	1	0
1	0	0
1	1	1

AND

$$\begin{array}{r}
 0100\ 0101 \longleftarrow 69 \\
 + 0110\ 0111 \longleftarrow 103 \\
 \hline
 1010\ 1100 \longleftarrow 172
 \end{array}$$

9

## Modulo Arithmetic

- Consider only numbers in a range
  - E.g., five-digit car odometer: 0, 1, ..., 99999
  - E.g., eight-bit numbers 0, 1, ..., 255
- Roll-over when you run out of space
  - E.g., car odometer goes from 99999 to 0, 1, ...
  - E.g., eight-bit number goes from 255 to 0, 1, ...
- Adding  $2^n$  doesn't change the answer
  - For eight-bit number,  $n=8$  and  $2^n=256$
  - E.g.,  $(37 + 256) \bmod 256$  is simply 37
- This can help us do subtraction...
  - Suppose you want to compute  $a - b$
  - Note that this equals  $a + (256 - 1 - b) + 1$

10

## One's and Two's Complement

- One's complement: flip every bit
  - E.g., b is 01000101 (i.e., 69 in decimal)
  - One's complement is 10111010
  - That's simply 255-69
- Subtracting from 11111111 is easy (no carry needed!)

$$\begin{array}{r}
 1111\ 1111 \\
 - 0100\ 0101 \quad \leftarrow b \\
 \hline
 1011\ 1010 \quad \leftarrow \text{one's complement}
 \end{array}$$

- Two's complement
  - Add 1 to the one's complement
  - E.g.,  $(255 - 69) + 1 \rightarrow 1011\ 1011$

11

## Putting it All Together

- Computing "a - b"
  - Same as "a + 256 - b"
  - Same as "a + (255 - b) + 1"
  - Same as "a + onesComplement(b) + 1"
  - Same as "a + twosComplement(b)"

- Example: 172 - 69

- The original number 69: 0100 0101
- One's complement of 69: 1011 1010
- Two's complement of 69: 1011 1011
- Add to the number 172: 1010 1100
- The sum comes to: 0110 0111
- Equals: 103 in decimal

$$\begin{array}{r}
 1010\ 1100 \\
 + 1011\ 1011 \\
 \hline
 1\ 0110\ 0111
 \end{array}$$

12

## Reading Two's Complement Patterns

- When a two's complement number has a highest bit 1, it indicates that the number is negative.
- To find the value, perform the same steps:
  - Unknown value: 1011 1011
  - One's complement: 0100 0100
  - Add 1 (two's complement): 0100 0101

We get a value of 69, so the original pattern must have been -69.

*Practice: Ch. 2, Exercise 2*

13

## Signed Integers

- **Sign-magnitude representation**
  - Use one bit to store the sign
    - Zero for positive number
    - One for negative number
  - Examples
    - E.g., 0010 1100 → 44
    - E.g., 1010 1100 → -44
  - Hard to do arithmetic this way, so it is rarely used
- **Complement representation**
  - One's complement
    - Flip every bit
    - E.g., 1101 0011 → -44
  - Two's complement
    - Flip every bit, then add 1
    - E.g., 1101 0100 → -44

14

## Overflow: Running Out of Room

- Adding two large integers together
  - Sum might be too large to store in the number of bits available
  - What happens?
- Unsigned integers
  - All arithmetic is “modulo” arithmetic
  - Sum would just wrap around
- Signed integers
  - Can get nonsense values
  - Example with 16-bit integers
    - Sum:  $10000+20000+30000$
    - Result:  $-5536$

15

## Question

- What does 10011001 represent? [Ex. 6, page 69]

16

## Bitwise Operators: AND and OR

- Bitwise AND (&)

&	0	1
0	0	0
1	0	1

- Bitwise OR (|)

	0	1
0	0	1
1	1	1

53 

0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

& 15 

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

---

5 

0	0	0	0	0	1	0	1
---	---	---	---	---	---	---	---

17

## Bitwise Operators: Not and XOR

- One's complement (~)

- Turns 0 to 1, and 1 to 0
- E.g., set last three bits to 0
  - $x = x \& \sim 7;$

- XOR (^)

- 0 if both bits are the same
- 1 if the two bits are different

^	0	1
0	0	1
1	1	0

*Practice: Ch. 2, Exercise 9*

18

## Bitwise Operators: Shift Left/Right

- Shift left (<<): Multiply by powers of 2
  - Shift some # of bits to the left, filling the blanks with 0

53    0 0 1 1 0 1 0 1

53<<2    1 1 0 1 0 0 0 0

- Shift right (>>): Divide by powers of 2
  - Shift some # of bits to the right. Fill in blanks with **sign bit**.

53    0 0 1 1 0 1 0 1

-75    1 0 1 1 0 1 0 1

53>>2    0 0 0 0 1 1 0 1

-75>>2    1 1 1 0 1 1 0 1

sign extension

sign extension

*Practice: Ch. 2, Exercises 10, 12, 13, 14*

19

## Bitmasks

- Used to change or query one or more bits in a variable.
- The bitmask indicates which bits are to be affected.
- Common operations:

Operation		C expression
Set N <sup>th</sup> bit of x	$x = x \text{ OR } 2^N$	$x = x   (1 \ll N);$
Clear N <sup>th</sup> bit of x	$x = x \text{ AND } (\text{NOT } 2^N)$	$x = x \& \sim(1 \ll N);$
Read N <sup>th</sup> bit of x	$?? = x \text{ AND } 2^N$	$?? = x \& (1 \ll N);$

Can extend to groups of bits

20

## Bitmask Example

```
/* This program demonstrates setting a bit, clearing a bit, and
   ** reading a bit. (pg 64) */

#include <stdio.h>

main()
{
    char a;
    int i;

    a=17;
    a=a | (1 << 3); /* set 3rd bit */
    printf("%d\n",a);

    a=a & ~(1<<4); /* clear 4th bit */
    printf("%d\n",a);

    for (i=7; i>=0; i--)
        printf("%d ",(a&(1<<i)) >> i); /* read i'th bit */
    printf("\n");
}
```

21

## Example: Counting the 1's

- How many 1 bits in a number?
  - E.g., how many 1 bits in the binary representation of 53?

0 0 1 1 0 1 0 1

- Four 1 bits
- How to count them?
  - Look at one bit at a time
  - Check if that bit is a 1
  - Increment counter
- How to look at one bit at a time?
  - Look at the last bit:  $n \& 1$
  - Check if it is a 1:  $(n \& 1) == 1$ , or simply  $(n \& 1)$

22

## Counting the Number of '1' Bits

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    unsigned n, count;

    printf("Number: ");
    if (scanf("%u", &n) != 1) {
        fprintf(stderr, "Error: Expect number.\n");
        exit(EXIT_FAILURE);
    }

    for (count=0; n; n >>= 1)
        count += (n & 1);

    printf("Number of 1 bits: %u\n", count);
    return 0;
}
```

## Practice

- Write a function `getByte` that extracts a byte from a word.

```
/*
 * getByte(x, n) - Extract byte n from word x
 * Bytes numbered from 0 (LSB) to 3 (MSB)
 * Examples: getByte(0x12345678,1) = 0x56
 * Legal operations: ~ & ^ | + << >>
 */
int getByte(int x, int n)
{

}
```

## **C Integral Data Types**

**(used to manipulate integers)**

25

## **Data Types**

- **Programming languages combine:**
  - Bits into bytes
  - Bytes into larger entities
- **Combinations of bytes have types; why?**
  - Facilitates abstraction
  - Enables compiler to do type checking
- **C has 11 primitive data types**
  - 8 integral data types (described in this lecture)
    - Four different sizes (char, short, int, and long)
    - Signed vs. unsigned
  - 3 floating-point data types (described in next lecture)

26

## C Integral Data Types

- Why char vs. short vs. int vs. long?
  - Small sizes conserve memory
  - Large sizes provide more range
- Why signed vs. unsigned?
  - Signed types allow negatives
  - Unsigned types allow larger positive numbers
  - (Dubious value: Java omits unsigned types)
- When to use unsigned?
  - When you really need that extra bit
  - When you'll do lots of bit shifting
  - When you'll never do ( $a < 0$ ) test

27

## C Integral Data Types (continued)

- Integral types:

Type	Bytes	Typically Used to Store
signed char	1	The numeric code of a character
unsigned char	1	The numeric code of a character
(signed) short	2*	A small integer
unsigned short	2*	A small non-negative integer
(signed) int	4*	An integer
unsigned int	4*	A non-negative integer
(signed) long	4*	An integer
unsigned long	4*	A non-negative integer

\* On tanner; C90 standard does not specify size

28

## The `sizeof` Operator

Category	Operators
<code>sizeof</code>	<code>sizeof(type)</code> <code>sizeof(expr)</code>

- Unique among operators: evaluated at compile-time
- Evaluates to type `size_t`; on tanner, same as `unsigned int`
- Examples

```
int i = 10;
double d = 100.0;
...
... sizeof(int) ...      /* On tanner, evaluates to 4 */
... sizeof(i) ...       /* On tanner, evaluates to 4 */
... sizeof(double)...   /* On tanner, evaluates to 8 */
... sizeof(d) ...      /* On tanner, evaluates to 8 */
... sizeof(d + 200.0) ... /* On tanner, evaluates to 8 */
```

29

## Determining Data Sizes

- To determine data sizes on your computer

```
#include <stdio.h>

int main()
{
    printf("char:      %d\n", (int)sizeof(char));
    printf("short:     %d\n", (int)sizeof(short));
    printf("int:        %d\n", (int)sizeof(int));
    printf("long:       %d\n", (int)sizeof(long));
    printf("float:      %d\n", (int)sizeof(float));
    printf("double:     %d\n", (int)sizeof(double));
    printf("long double: %d\n", (int)sizeof(long double));
    return 0;
}
```

- Output on tanner

```
char:      1
short:     2
int:       4
long:      4
float:     4
double:    8
long double: 16
```

30

## The `int` Data Type

- Description: A positive or negative integer
  - Same as `signed int`
- Size: System dependent
  - $16 \leq \text{bits in } \text{short} \leq \text{bits in } \text{int} \leq \text{bits in } \text{long}$
  - Usually 16 bits (alias 2 bytes) or 32 bits (alias 4 bytes)
  - The “natural word size” of the computer

31

## The `int` Data Type (cont.)

- Example constants (assuming 4 bytes)

Constant	Binary Representation	Note
123	00000000 00000000 00000000 01111011	decimal form
-123	11111111 11111111 11111111 10000101	negative form
2147483647	01111111 11111111 11111111 11111111	largest
-2147483648	10000000 00000000 00000000 00000000	smallest
0173	00000000 00000000 00000000 01111011	octal form
0x7B	00000000 00000000 00000000 01111011	hexadecimal form

High-order bit indicates sign

Two's complement

Leading zero means octal

Leading zero-x means hexadecimal

32

## The **unsigned int** Data Type

- Description: A positive integer
- Size: System dependent
  - Same as `int`
- Example constants (assuming 4 bytes)

Constant	Binary Representation	Note
123U	00000000 00000000 00000000 01111011	decimal form
4294967295U	11111111 11111111 11111111 11111111	largest
0U	00000000 00000000 00000000 00000000	smallest
0173U	00000000 00000000 00000000 01111011	octal form
0x7BU	00000000 00000000 00000000 01111011	hexadecimal form

Note "U" suffix

Same range as int, but shifted on number line

33

## The **long** Data Type

- Description: A positive or negative integer
  - Same as `signed long`
- Size: System dependent
  - $16 \leq \text{bits in short} \leq \text{bits in int} \leq \text{bits in long}$
  - Usually 32 bits, alias 4 bytes
- Example constants (assuming 4 bytes)

Constant	Binary Representation	Note
123L	00000000 00000000 00000000 01111011	decimal form
-123L	11111111 11111111 11111111 1000101	negative form
2147483647L	01111111 11111111 11111111 11111111	largest
-2147483648L	10000000 00000000 00000000 00000000	smallest
0173L	00000000 00000000 00000000 01111011	octal form
0x7BL	00000000 00000000 00000000 01111011	hexadecimal form

Note "L" suffix

34

## The unsigned long Data Type

- Description: A positive integer
- Size: System dependent
  - Same as `long`
- Example constants (assuming 4 bytes)

Constant	Binary Representation	Note
123UL	00000000 00000000 00000000 01111011	decimal form
4294967295UL	11111111 11111111 11111111 11111111	largest
0UL	00000000 00000000 00000000 00000000	smallest
0173UL	00000000 00000000 00000000 01111011	octal form
0x7BUL	00000000 00000000 00000000 01111011	hexadecimal form

Note "UL" suffix

35

## The short Data Type

- Description: A positive or negative integer
  - Same as `signed short`
- Size: System dependent
  - $16 \leq \text{bits in short} \leq \text{bits in int} \leq \text{bits in long}$
  - Usually 16 bits, alias 2 bytes
- Example constants (assuming 2 bytes)

Constant	Binary Representation	Note
(short)123	00000000 01111011	decimal form
(short)-123	11111111 10000101	negative form
(short)32767	01111111 11111111	largest
(short)-32768	10000000 00000000	smallest
(short)0173	00000000 01111011	octal form
(short)0x7B	00000000 01111011	hexadecimal form

No way to express constant of type short, so must use cast

36

## The unsigned short Data Type

- Description: A positive integer
- Size: System dependent
  - Same as `short`
- Example constants (assuming 4 bytes)

Constant	Binary Representation	Note
<code>(unsigned short)123U</code>	00000000 01111011	decimal form
<code>(unsigned short)65535U</code>	11111111 11111111	largest
<code>(unsigned short)0U</code>	00000000 00000000	smallest
<code>(unsigned short)0173U</code>	00000000 01111011	octal form
<code>(unsigned short)0x7BU</code>	00000000 01111011	hexadecimal form

No way to express constant of type unsigned short, so must use cast

37

## The signed char Data Type

- Description: A (small) positive or negative integer
- Size: 1 byte
- Example constants

Constant	Binary Representation	Note
<code>(signed char)123</code>	01111011	decimal form
<code>(signed char)-123</code>	10000101	negative form
<code>(signed char)127</code>	01111111	largest
<code>(signed char)-128</code>	10000000	smallest
<code>(signed char)0173</code>	01111011	octal form
<code>(signed char)0x7B</code>	01111011	hexadecimal form

No way to express constant of type signed char, so must use cast

38

## The **unsigned char** Data Type

- Description: A (small) positive integer
- Size: 1 byte
- Example constants

Constant	Binary Representation	Note
<code>(unsigned char)123</code>	01111011	decimal form
<code>(unsigned char)255</code>	11111111	largest
<code>(unsigned char)0</code>	00000000	smallest
<code>(unsigned char)0173</code>	01111011	octal form
<code>(unsigned char)0x7B</code>	01111011	hexadecimal form

No way to express constant of type unsigned char, so must use cast

39

## The **char** Data Type

- On some systems, **char** means signed char
- On other systems, **char** means unsigned char
- Obstacle to portability

```
int a[256];
char c;
c = (char)255;
...
... a[c] ...
/* char is unsigned => a[255] => OK */
/* char is signed => a[-1] => out of bounds */
```

40

## The **char** Data Type (cont.)

- On your system, is **char** signed or unsigned?

```
#include <stdio.h>
int main(void) {
    char c = (char)0x80;
    if (c > 0)
        printf("unsigned");
    else
        printf("signed");
    return 0;
}
```

- Output on tanner

```
signed
```

41

## The **char** Data Type (cont.)

- Q:
  - Why is type **char** called “char” (meaning “character”)?
- A:
  - Type **char** can be used for limited range arithmetic
    - As indicated previously
  - However, type **char** is used more often to store a **character code**

42

# Characters

43

## Using `char` for Characters

- Type `char` can be used for (limited range) arithmetic, but...
- Usually used to store characters – thus the name!
  - Must use a code to map 1-byte numbers to characters
  - Common code: ASCII
- Other ways to represent characters
  - Unicode (“wide” characters) -- 2 bytes

44

# The ASCII Code

## American Standard Code for Information Interchange

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Lower case: 97-122 and upper case: 65-90  
E.g., 'a' is 97 and 'A' is 65 (i.e., 32 apart)

45

# char Constants

- C has char constants (sort of) \*
- Examples

Constant	Binary Representation (assuming ASCII)	Note
'a'	01100001	letter
'0'	00110000	digit
'\0141'	01100001	octal form
'\x61'	01100001	hexadecimal form

Use **single** quotes for **char** constant  
Use **double** quotes for **string** constant

\* Technically 'a' is of type int; automatically truncated to type char when appropriate

46

## More **char** Constants

- Escape characters

Constant	Binary Representation (assuming ASCII)	Note
'\a'	00000111	alert (bell)
'\b'	00001000	backspace
'\f'	00001100	form feed
'\n'	00001010	newline
'\r'	00001101	carriage return
'\t'	00001001	horizontal tab
'\v'	00001011	vertical tab
'\\'	01011100	backslash
'\?'	00111111	question mark
'\''	00100111	single quote
'\"'	00100010	double quote
'\0'	00000000	null

Used often

47

## Reading and Writing a Character

- Subset of C I/O functions:

Task	Example Function Calls
Write a char	<pre>int status; status = putchar('a'); /* Writes to stdout */</pre>
Read a char	<pre>int c; c = getchar(); /* Reads from stdin */</pre>

```
#include <stdio.h>

int main(void) {
    int c;
    c = getchar();
    if (c != EOF) {
        if ((c >= 'a') && (c <= 'z'))
            c += 'A' - 'a';
        putchar(c);
    }
}
```

'a' is 97  
'A' is 65

48

## The “End-of-File Character”

- Files do not end with the “EOF character”
  - Because there is no such thing!!!
- EOF is:
  - A special **non-character** value returned by `getchar()` and related functions to indicate failure
  - #defined in `stdio.h`; typically as `-1`

49

## Using EOF

- Correct code

```
int c;  
c = getchar();  
while (c != EOF) {  
    ...  
    c = getchar();  
}
```

`getchar()` returns `int` because:

- `int` is the computer's natural word size
- `getchar()` must be able to return all valid **chars** and EOF

- Equivalent idiom

```
int c;  
while ((c = getchar()) != EOF) {  
    ...  
}
```

An expression of the form  
`x = y`  
assigns to `x`, and evaluates  
to the new value of `x`

- Incorrect code

```
char c;  
while ((c = getchar()) != EOF) {  
    ...  
}
```

What if `stdin` contains the  
`11111111` ( $\bar{y}$ ) character?

50

## Strings

- Java has a **String** class
  - `String s; // OK in Java`
- C does not have a **String** data type
  - `String s; /* Not OK in C */`
- Java and C have string constants
  - E.g. "hello"
- In C, a string is a null-terminated array of characters
  - 'a' is a **char** (01100001)
  - "a" is a string (01100001 00000000)
- More later, after discussing pointers and arrays

51

## Floating-Point Numbers

52

## C Floating-Point Data Types

- Floating-point types:

Type	Bytes	Typically Used to Store
float	4*	A low-precision/range floating-point number
double	8*	A floating-point number
long double	16*	A high-precision/range floating-point number

\* On tanner only; size is system-dependent

53

## The float Data Type

- Description:
  - A (positive or negative) floating point number
- Size: system dependent
  - bits in float <= bits in double <= bits in long double
  - Often 4 bytes; limited precision and range; infrequently used
- Example constants (assuming 4 bytes)

Constant	Note
123.456F	Typical
1.23456E2F	Typical
3.402823E38F	Largest (approx.)
-3.402823E38F	Smallest (approx.)
1.175494E-38F	Closest to 0 (approx.)

Note "F" suffix

54

## The **double** Data Type

- **Description:**
  - A (positive or negative) double-precision floating point number
- **Size: system dependent**
  - bits in **float** <= bits in **double** <= bits in **long double**
  - Often 8 bytes
- **Example constants (assuming 8 bytes)**

Constant	Note
123.456	Typical
1.23456E2	Typical
1.797693E308	Largest (approx.)
-1.79693E308	Smallest (approx.)
2.225074E-308	Closest to 0 (approx.)

Decimal point or "E" indicates floating point

55

## The **long double** Data Type

- **Description:**
  - A (positive or negative) floating point number
- **Size: system dependent**
  - bits in **float** <= bits in **double** <= bits in **long double**
  - Often 10 or 12 bytes
- **Example constants (assuming 12 bytes)**

Constant	Note
123.456L	Typical
1.23456E2L	Typical
1.189731E4932L	Largest (approx.)
-1.189731E4932L	Smallest (approx.)
3.362103E-4932L	Closest to 0 (approx.)

Note "L" suffix

56

## Data Types: C vs. Java

Java	C
boolean	(no equivalent)
byte	(no equivalent)
(no equivalent)	long double
(no equivalent)	unsigned types
char comprises 2 bytes (Unicode)	char comprises 1 byte (often ASCII)
Sizes of all types specified	char is one byte Sizes of all other types unspecified

Recall Java goal:  
Portability → specify sizes

Recall C goal:  
Create an OS → use natural word size

57

## Summary

- Computer represents everything in binary
  - Integers, floating-point numbers, characters, addresses, ...
  - Pixels, sounds, colors, etc.
- Binary arithmetic through logic operations
  - Sum (XOR) and Carry (AND)
  - Two's complement for subtraction
- Binary operations in C
  - AND, OR, NOT, XOR, shift left and shift right
  - Useful for efficient and concise code, though sometimes cryptic
- C integral data types
  - char, short, int, long (signed and unsigned)
- Characters
- Floating-point data types

58

## Required Reading

- *Textbook, Chapter 2*