Numbers (in C and otherwise)

Q: Why do computer programmers confuse Christmas and Halloween?

A: Because 25 Dec == 31 Oct

The Decimal Number System

Name

- "decem" (Latin) ⇒ ten

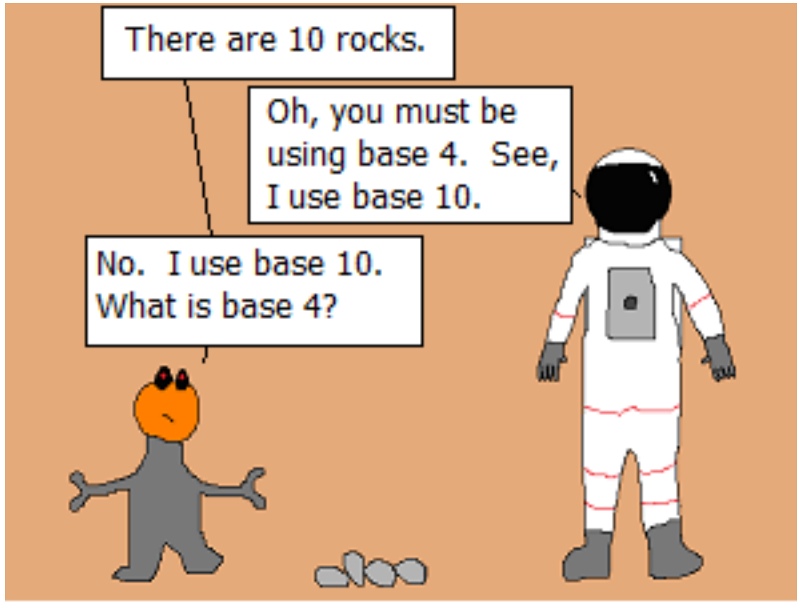
Characteristics

- For us, these symbols (Not universal ...)
  - 0 1 2 3 4 5 6 7 8 9

https://bit.ly/3ifUw1b

European (descended from the West Arabic)	0	1	2	3	4	5	6	7	8	9
Arabic-Indic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Eastern Arabic-Indic (Persian and Urdu)	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
Devanagari (Hindi)	०	१	२	३	४	५	६	७	८	९
Tamil		௧	௨	௩	௪	௫	௬	௭	௮	௯

- Positional
  - 2945 ≠ 2495
  - $2945 = (2*10^3) + (9*10^2) + (4*10^1) + (5*10^0)$



Every base is base 10.

(Most) people use the decimal number system

Why?

The Binary Number System

binary

adjective: being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal. From Late Latin *bīnārius* ("consisting of two").

Characteristics

- Two symbols: **`0 1`**
- Positional: $\mathtt{1010_B} \neq \mathtt{1100_B}$

Most (digital) computers use the binary number system

Why?

Terminology

- **Bit**: a single binary symbol ("binary digit")
- **Byte**: (typically) 8 bits
- **Nibble / Nybble**: 4 bits

Decimal-Binary Equivalence

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Decimal	Binary
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111
...	...

Decimal-Binary Conversion

Binary to decimal: expand using positional notation

$$
\begin{aligned}
100101_B &= (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0) \\
&= 32 + 0 + 0 + 4 + 0 + 1 \\
&= 37
\end{aligned}
$$

Most-significant bit (msb)

Least-significant bit (lsb)

Integer-Binary Conversion

(Decimal) Integer to binary: do the reverse

- Determine largest power of 2 that's ≤ number; write template

```
37 = (?*2⁵)+(?*2⁴)+(?*2³)+(?*2²)+(?*2¹)+(?*2⁰)
```

- Fill in template

```
 37 = (1*2⁵)+(0*2⁴)+(0*2³)+(1*2²)+(0*2¹)+(1*2⁰)
-32
  5
 -4
  1                   100101B
 -1
  0
```

Integer-Binary Conversion

Integer to binary division method

- Repeatedly divide by 2, consider remainder

```
37 / 2 = 18 R 1
18 / 2 =  9 R 0
 9 / 2 =  4 R 1
 4 / 2 =  2 R 0
 2 / 2 =  1 R 0
 1 / 2 =  0 R 1
```

Read from bottom to top: 100101_B

The Hexadecimal Number System

Name

- "hexa-" (Ancient Greek ἑξα-) ⇒ six
- "decem" (Latin) ⇒ ten

Characteristics

- Sixteen symbols
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Positional
  - A13DH ≠ 3DA1H

Computer programmers often use hexadecimal or "hex"

- In C: 0x prefix (0xA13D, etc.)

Why?

Binary-Hexadecimal Conversion

Observation:

- $16^1 = 2^4$, so every 1 hexit corresponds to 4 bits

Binary to hexadecimal

```
1010000100111101B
  A   1   3   DH
```

Number of bits in binary number not a multiple of 4? ⇒ pad with zeros on left

Hexadecimal to binary

```
  A   1   3   DH
1010000100111101B
```

Discard leading zeros from binary number if appropriate

Integer-Hexadecimal Conversion

Hexadecimal to (decimal) integer: expand using positional notation

```
25H = (2*16^1) + (5*16^0)
    =   32     +    5
    =   37
```

Integer to hexadecimal: use the division method

```
37 / 16 = 2 R 5
 2 / 16 = 0 R 2
```

↑ Read from bottom to top: 25_H

Are you 539_H?

Convert binary 101010 into decimal and hex

A. 21 decimal, A2 hex
B. 21 decimal, A8 hex
C. 18 decimal, 2A hex
D. 42 decimal, 2A hex

hint: convert to hex first

challenge: once you've locked in and discussed with a neighbor, figure out why this slide's title is what it is.

The Octal Number System

Name

- "octo" (Latin) ⇒ eight

Characteristics

- Eight symbols
  - 0 1 2 3 4 5 6 7
- Positional
  - $17430 \neq 73140$

Computer programmers sometimes use octal (so does Mickey!)

- In C: 0 prefix (01743, etc.)

Why?

```
[cmoretti@tars:tmp$ls -l myFile
-rw-r--r--  1 cmoretti  wheel  0 Sep  7 10:58 myFile
[cmoretti@tars:tmp$chmod 755 myFile
[cmoretti@tars:tmp$ls -l myFile
-rwxr-xr-x  1 cmoretti  wheel  0 Sep  7 10:58 myFile
```

INTEGERS

Representing Unsigned (Non-Negative) Integers

Mathematics

- Non-negative integers' range is 0 to ∞

Computers

- Range limited by computer's word size
- Word size is n bits $\Rightarrow$ range is 0 to $2^n - 1$
- Exceed range $\Rightarrow$ overflow

Typical computers today

- n = 32 or 64, so range is 0 to $2^{32} - 1$ (~4 billion) or $2^{64} - 1$ (huge … ~1.8e19)

Pretend computer

- n = 4, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4

- All points generalize to word size = n (armlab: 64)

Representing Unsigned Integers

On 4-bit pretend computer

Unsigned Integer	Rep
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Adding Unsigned Integers

Addition

```
              1
   3        0011B
+ 10      + 1010B
  --        ----
  13        1101B
```

Start at right column

Proceed leftward

Carry 1 when necessary

```
            111
   7        0111B
+ 10      + 1010B
  --        ----
   1        0001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

7 + 10 = 17

17 mod 16 = 1

Subtracting Unsigned Integers

Subtraction

```
                111
  10           1010B
-  7         - 0111B
  --           ----
   3           0011B
```

Start at right column

Proceed leftward

Borrow when necessary

```
               1
   3           0011B
- 10         - 1010B
  --           ----
   9           1001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

3 - 10 = -7

-7 mod 16 = 9

Reminder: negative numbers exist

Obsolete Attempt #1: Sign-Magnitude

Integer	Rep
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit indicates sign

0 ⇒ positive

1 ⇒ negative

Remaining bits indicate magnitude

$0101_B = 101_B = 5$

$1101_B = -101_B = -5$

Pros and cons

+ easy to understand, easy to negate
+ symmetric
- two representations of zero
- need different algorithms to add signed and unsigned numbers

Not used for integers today

Obsolete Attempt #2: Ones' Complement

Integer	Rep
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight $-(2^{b-1}-1)$

$$1010_B = (1*-7)+(0*4)+(1*2)+(0*1) = -5$$

$$0010_B = (0*-7)+(0*4)+(1*2)+(0*1) = 2$$

Computing negative = flipping all bits

Similar pros and cons to sign-magnitude

Two's Complement

Integer	Rep
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight $-(2^{b-1})$

$1010_B = (1*-8)+(0*4)+(1*2)+(0*1)$
$= -6$

$0010_B = (0*-8)+(0*4)+(1*2)+(0*1)$
$= 2$

Two's Complement (cont.)

Integer	Rep
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Computing negative

neg(x) = ~x + 1

neg(x) = onescomp(x) + 1

$neg(0101_B) = 1010_B + 1 = 1011_B$

$neg(1011_B) = 0100_B + 1 = 0101_B$

Pros and cons

- not symmetric
  ("extra" negative number)

+ one representation of zero

+ same algorithm adds
  signed and unsigned integers

Adding Signed Integers

pos + pos

```
             11
   3       0011B
 + 3     + 0011B
  --       ----
   6       0110B
```

pos + pos (overflow)

```
            111
   7       0111B
 + 1     + 0001B
  --       ----
  -8       1000B
```

pos + neg

```
           1111
   3       0011B
 + -1    + 1111B
  --       ----
   2       0010B
```

How would you detect overflow programmatically?

neg + neg

```
           11
  -3       1101B
 + -2    + 1110B
  --       ----
  -5       1011B
```

neg + neg (overflow)

```
          1 1
  -6       1010B
 + -5    + 1011B
  --       ----
   5       0101B
```

Subtracting Signed Integers

How would you compute 3 – 4?

```
  3         0011B
- 4       - 0100B
 --         ----
  ?         ????B
```

Subtracting Signed Integers

Perform subtraction with borrows

or

Compute two's comp and add

```
           11
  3       0011B
- 4     - 0100B
 --       ----
 -1       1111B
```

```
  3       0011B
+ -4    + 1100B
 --       ----
 -1       1111B
```

```
           11
 -5       1011B
--2     - 1110B
 --       ----
 -3       1101B
```

```
             1
 -5       1011B
+  2    + 0010B
 --       ----
 -3       1101B
```

Negating Signed Ints: Math

Question: Why does two's comp arithmetic work?

Answer: [–b] mod 2^4 = [twoscomp(b)] mod 2^4

```
[-b] mod 2^4
= [2^4 - b] mod 2^4
= [2^4 - 1 - b + 1] mod 2^4
= [(2^4 - 1 - b) + 1] mod 2^4
= [onescomp(b) + 1] mod 2^4
= [twoscomp(b)] mod 2^4
```

So: [a – b] mod 2^4 = [a + twoscomp(b)] mod 2^4

```
[a - b] mod 2^4
= [a + 2^4 - b] mod 2^4
= [a + 2^4 - 1 - b + 1] mod 2^4
= [a + (2^4 - 1 - b) + 1] mod 2^4
= [a + onescomp(b) + 1] mod 2^4
= [a + twoscomp(b)] mod 2^4
```

(AT LONG° LAST) INTEGERS IN C

@hannahbusing

° no pun intended, I swear!

Integer Data Types in C

Integer types of various sizes: {`signed`, `unsigned`} {`char`, `short`, `int`, `long`}

- Shortcuts: `signed` assumed for `short`/`int`/`long`; `unsigned` means `unsigned int`
- `char` is 1 byte
  - Number of bits per byte is unspecified (but in the 21st century, safe to assume it's 8)
  - Signedness is system dependent
- Sizes of other integer types not fully specified but constrained:
  - `int` was intended to be "natural word size" of hardware
  - 2 ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`

On `armlab`:

- Natural word size: 8 bytes ("64-bit machine")
- `char`: 1 byte
- `short`: 2 bytes
- `int`: 4 bytes (compatibility with widespread 32-bit code)
- `long`: 8 bytes

What decisions did the designers of Java make?

Integer Types in Java vs. C

	Java	C

Unsigned types	`char // 16 bits`	`unsigned char`
		`unsigned short`
		`unsigned (int)`
		`unsigned long`

Signed types	`byte // 8 bits`
	`short // 16 bits`
	`int // 32 bits`
	`long // 64 bits`
		`signed char`
		`(signed) short`
		`(signed) int`
		`(signed) long`

1. Not guaranteed by C, but on **`armlab`**, **`short`** = 16 bits, **`int`** = 32 bits, **`long`** = 64 bits
2. Not guaranteed by C, but on **`armlab`**, **`char`** is unsigned

`sizeof` Operator

- Applied at compile-time
- Operand can be a data type
- Operand can be an expression, from which the compiler infers a data type

Examples, on `armlab` using `gcc217`

- `sizeof(int)` evaluates to 4
- `sizeof(i)` evaluates to 4 if `i` is a variable of type `int`
- `sizeof(1+2)` evaluates to 4

Integer Literals in C

- Decimal int: 123
- Prefixes to indicate a different base
  - Octal int: 0173 = 123
  - Hexadecimal int: 0x7B = 123
  - No prefix to indicate binary int literal
- Suffixes to indicate a different type
  - Use "L" suffix to indicate long literal
  - Use "U" suffix to indicate unsigned literal
  - No suffix to indicate char or short literals; instead, cast

char:	'{' (← really int, as seen last time), (char) 123, (char) 0173, (char) 0x7B
int:	123, 0173, 0x7B
long:	123L, 0173L, 0x7BL
short:	(short)123, (short)0173, (short)0x7B
unsigned int:	123U, 0173U, 0x7BU
unsigned long:	123UL, 0173UL, 0x7BUL
unsigned short:	(unsigned short)123, (unsigned short)0173, (unsigned short)0x7B

sizeof synthesis

Q: What is the value of the following sizeof expression on the armlab machines?

```
int i = 1;

sizeof(i + 2L)
```

A. 3

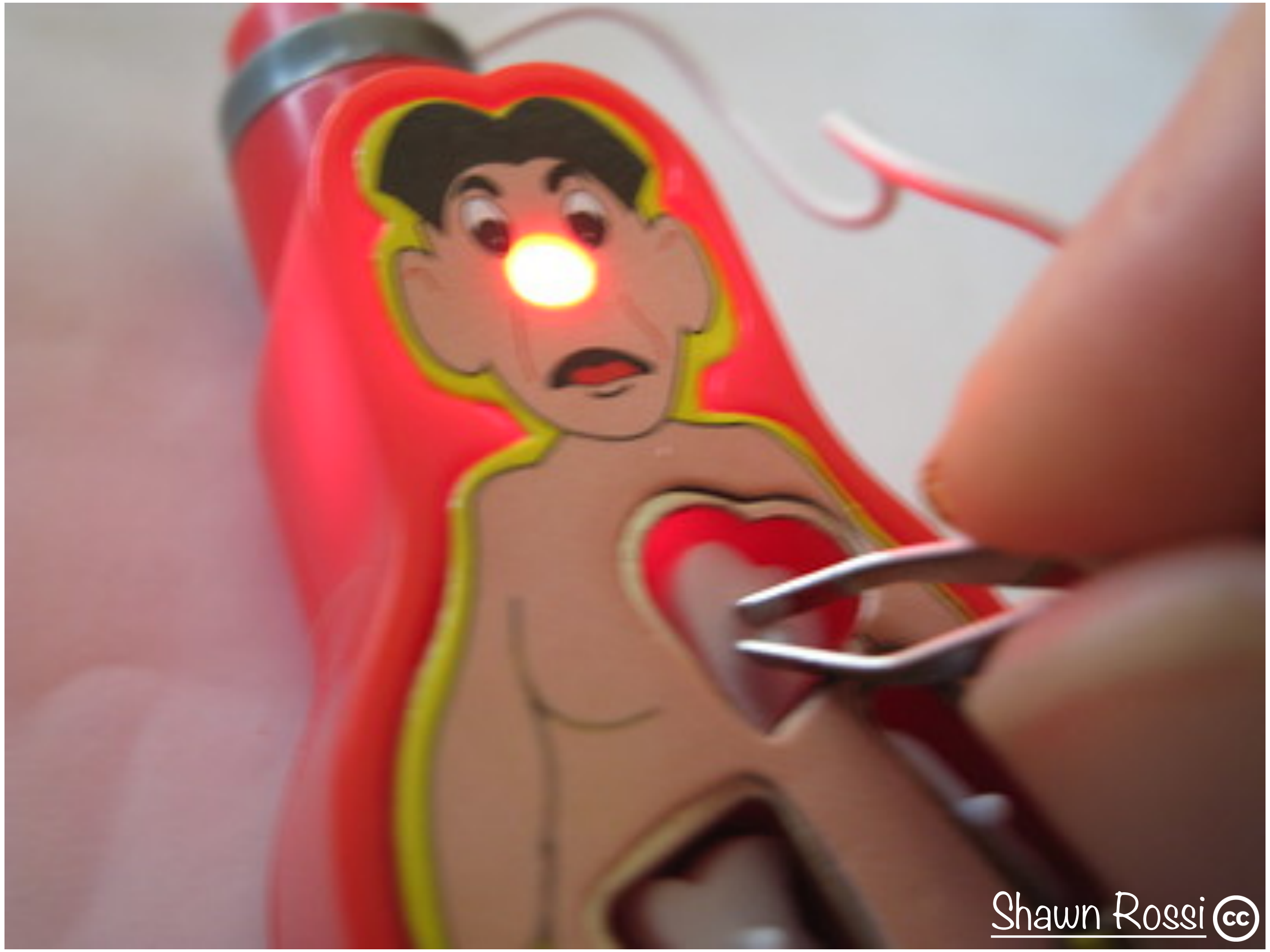
B. 4

C. 8

D. 12

E. error

OPERATIONS ON NUMBERS



Reading / Writing Numbers

Motivation

- Must convert between external form (sequence of character codes) and internal form
- Could provide getchar(), putshort(), getint(), putfloat(), etc.
- Alternative implemented in C: parameterized functions

scanf() and printf()

- Can read/write any primitive type of data
- First parameter is a format string containing conversion specs: size, base, field width
- Can read/write multiple variables with one call

See King book for details

Operators in C

- Typical arithmetic operators: + – * / %
- Typical relational operators: == != < <= > >=
  - Each evaluates to FALSE ⇒ 0, TRUE ⇒ 1
- Typical logical operators: ! && ||
  - Each interprets 0 ⇒ FALSE, non-0 ⇒ TRUE
  - Each evaluates to FALSE ⇒ 0, TRUE ⇒ 1
- Cast operator: (type)
- Bitwise operators: ~ & | ^ >> <<

Shifting Unsigned Integers

Bitwise right shift (>> in C): fill on left with zeros

`10 >> 1 ⇒ 5`

1010_B 0101_B

`10 >> 2 ⇒ 2`

1010_B 0010_B

What is the effect arithmetically?

Bitwise left shift (<< in C): fill on right with zeros

`5 << 1 ⇒ 10`

0101_B 1010_B

`3 << 2 ⇒ 12`

0011_B 1100_B

What is the effect arithmetically?

`3 << 3 ⇒ 8`

0011_B 1000_B ← Results are mod 2^4

Other Bitwise Operations on Unsigned Integers

Bitwise NOT (~ in C)

- Flip each bit

```
~10 ⇒ 5
```

1010_B 0101_B

```
~5 ⇒ 10
```

0101_B 1010_B

Bitwise AND (& in C)

- AND (1=True, 0=False) corresponding bits

```
 10          1010B
& 7        & 0111B
 --          ----
  2          0010B
```

```
 10          1010B
& 2        & 0010B
 --          ----
  2          0010B
```

Useful for “masking” bits to 0

x & 0 is 0, x & 1 is x

Other Bitwise Operations on Unsigned Ints

Bitwise OR: (| in C)

- Logical OR corresponding bits

```
  10         1010B
|  1       | 0001B
  --         ----
  11         1011B
```

Useful for "masking" bits to 1

x | 1 is 1, x | 0 is x

Bitwise exclusive OR (^ in C)

- Logical exclusive OR corresponding bits

```
  10         1010B
^ 10       ^ 1010B
  --         ----
   0         0000B
```

x ^ x sets
all bits to 0

Logical vs. Bitwise Ops

Logical AND (&&) vs. bitwise AND (&)

- `2 (TRUE) && 1 (TRUE) => 1 (TRUE)`

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
   && 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      1  00000000 00000000 00000000 00000001
```

- `2 (TRUE)  & 1 (TRUE) => 0 (FALSE)`

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
    & 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      0  00000000 00000000 00000000 00000000
```

Implication:

- Use **logical** AND to control flow of logic
- Use **bitwise** AND only when doing bit-level manipulation
- Same for OR and NOT

A Bit Complicated … challenge for the bored

How do you set bit k (where the least significant bit is bit 0) of unsigned variable u to zero (leaving everything else in u unchanged)?

A. u &= (0 << k);

B. u |= (1 << k);

C. u |= ~(1 << k);

D. u &= ~(1 << k);

E. u = ~u ^ (1 << k);

Aside: Using Bitwise Ops for Arithmetic

Can use <<, >>, and & to do some arithmetic efficiently

$x * 2^y$ == x << y

- $3*4 = 3*2^2$ = 3<<2 ⇒ 12

Fast way to multiply by a power of 2

$x / 2^y$ == x >> y

- $13/4 = 13/2^2$ = 13>>2 ⇒ 3

Fast way to divide unsigned by power of 2

$x \% 2^y$ == x & (2^y-1)

- $13\%4 = 13\%2^2 = 13\&(2^2-1)$ = 13&3 ⇒ 1

Fast way to mod by a power of 2

```
 13         1101B
& 3       & 0011B
 --         ----
  1         0001B
```

Many compilers will do these transformations automatically!

Shifting Signed Integers

Bitwise left shift (<< in C): fill on right with zeros

`3 << 1 ⇒ 6`

0011_B 0110_B

`-3 << 1 ⇒ -6`

1101_B 1010_B

`-3 << 2 ⇒ 4`

1101_B 0100_B

What is the effect arithmetically?

Results are mod 2^4

Bitwise right shift: fill on left with ???

Shifting Signed Integers (cont.)

Bitwise *arithmetic* right shift: fill on left with sign bit

```
6 >> 1 ⇒ 3
```

0110_B 0011_B

```
-6 >> 1 ⇒ -3
```

1010_B 1101_B

What is the effect arithmetically?

Bitwise *logical* right shift: fill on left with zeros

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => 5
```

1010_B 0101_B

What is the effect arithmetically???

In C, right shift (>>) could be logical (>>> in Java) or arithmetic (>> in Java)

- Not specified by standard (happens to be arithmetic on armlab)
- Best to avoid shifting signed integers

Other Operations on Signed Ints

Bitwise NOT (~ in C)

- Same as with unsigned ints

Bitwise AND (& in C)

- Same as with unsigned ints

Bitwise OR: (| in C)

- Same as with unsigned ints

Bitwise exclusive OR (^ in C)

- Same as with unsigned ints

Best to avoid with signed integers

Assignment Operator

Many high-level languages provide an assignment *statement*

C provides an assignment **operator**

- Performs assignment, and then **evaluates to the assigned value**
- Allows assignment to appear within larger expressions

Assignment Operator Examples

Examples

```
i = 0;
   /* Side effect: assign 0 to i.
      Evaluate to 0. */

j = i = 0; /* Assignment op has R to L associativity */
   /* Side effect: assign 0 to i.
      Evaluate to 0.
      Side effect: assign 0 to j.
      Evaluate to 0. */

while ((i = getchar()) != EOF) …
   /* Read a character or EOF value.
      Side effect: assign that value to i.
      Evaluate to that value.
      Compare that value to EOF.
      Evaluate to 0 (FALSE) or 1 (TRUE). */
```

Special-Purpose Assignment in C

Motivation

- The construct a = b + c is flexible
- The construct i = i + c is somewhat common
- The construct i = i + 1 is very common

Assignment in C

- Introduce += operator to do things like i += c
- Extend to –= *= /= ~= &= |= ^= <<= >>=
- All evaluate to whatever was assigned
- Pre-increment and pre-decrement: ++i ––i
- Post-increment and post-decrement (evaluate to *old* value): i++ i––

Plusplus Playfulness

Q: What are i and j set to in the following code?

```
i = 5;
j = i++;
j += ++i;
```

A. 5, 7

B. 7, 5

C. 7, 11

D. 7, 12

E. 7, 13

Incremental Iffiness

Q: What does the following code print?

```
int i = 1;
switch (i++) {
   case 1: printf("%d", ++i);
   case 2: printf("%d", i++);
}
```

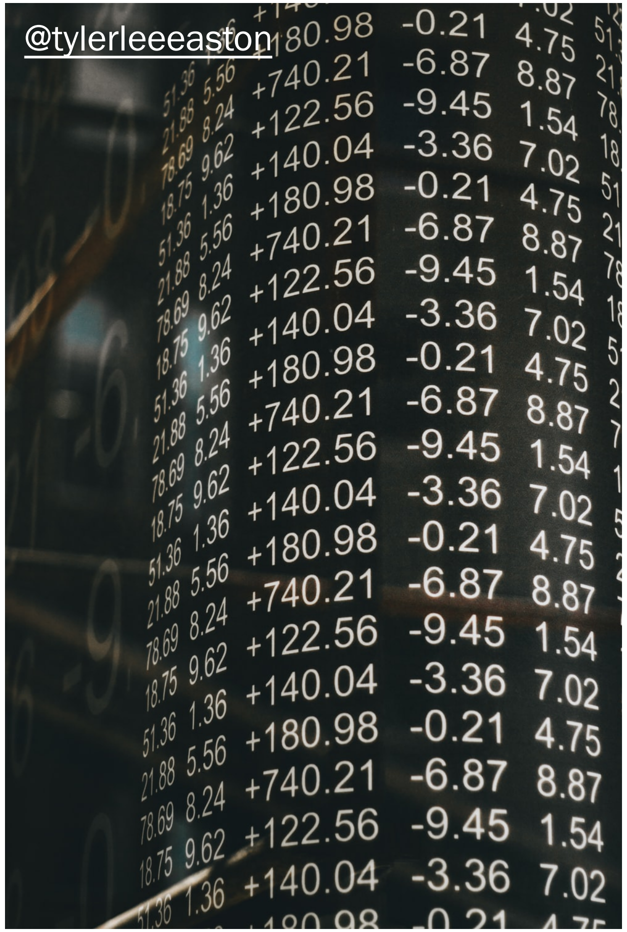
A. 1

B. 2

C. 3

D. 22

E. 33



APPENDIX: FLOATING POINT

Rational Numbers

Mathematics

- A rational number is one that can be expressed as the ratio of two integers
- Unbounded range and precision

Computer science

- Finite range and precision
- Approximate using floating point number

Floating Point Numbers

Like scientific notation: e.g., c is

$$2.99792458 \times 10^{8} \text{ m/s}$$

This has the form

$$(\text{multiplier}) \times (\text{base})^{(\text{power})}$$

In the computer,

- Multiplier is called mantissa
- Base is almost always 2
- Power is called exponent

Floating-Point Data Types

C specifies:

- Three floating-point data types:
  float, double, and long double
- Sizes unspecified, but constrained:
- sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

On ArmLab (and on pretty much any 21st-century computer using the IEEE standard)

- float: 4 bytes
- double: 8 bytes

On ArmLab (but varying across architectures)

- long double: 16 bytes

Floating-Point Literals

How to write a floating-point number?

- Either fixed-point or "scientific" notation
- Any literal that contains decimal point or "E" is floating-point
- The default floating-point type is double
- Append "F" to indicate float
- Append "L" to indicate long double

Examples

- double: 123.456, 1E-2, -1.23456E4
- float: 123.456F, 1E-2F, -1.23456E4F
- long double: 123.456L, 1E-2L, -1.23456E4L

IEEE Floating Point Representation

Common finite representation: IEEE floating point

- More precisely: ISO/IEEE 754 standard

Using 32 bits (type **`float`** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbbbb

Using 64 bits (type **`double`** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

When was floating-point invented?

mantissa (noun): decimal part of a logarithm, 1865, ← Answer: long before computers!
from Latin mantisa "a worthless addition, makeweight"

COMMON LOGARITHMS $\log_{10} x$

x	0	1	2	3	4	5	6	7	8	9	Δ_m +	1	2	3
50	·6990	6998	7007	7016	7024	7033	7042	7050	7059	7067	9	1	2	3
51	·7076	7084	7093	7101	7110	7118	7126	7135	7143	7152	8	1	2	2
52	·7160	7168	7177	7185	7193	7202	7210	7218	7226	7235	8	1	2	2
53	·7243	7251	7259	7267	7275	7284	7292	7300	7308	7316	8	1	2	2
54	·7324	7332	7340	7348	7356	7364	7372	7380	7388	7396	8	1	2	2
55	·7404	7412	7419	7427	7435	7443	7451	7459	7466	7474	8	1	2	2

Floating Point Example

11000001110110110000000000000000

32-bit representation

Sign (1 bit):

- 1 ⇒ negative

Exponent (8 bits):

- $10000011_B = 131$
- 131 – 127 = 4

Mantissa (23 bits):

- $1.10110110000000000000000_B$
- $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5})+(1*2^{-6})+(1*2^{-7}) +(0*2^{-\dots})= 1.7109375$

Number:

- $-1.7109375 * 2^4 = -27.375$

Floating Point Consequences

"Machine epsilon": smallest positive number you can add to 1.0 and get something other than 1.0

For float: $\varepsilon \approx$ 10–7

- No such number as 1.000000001
- Rule of thumb: "almost 7 digits of precision"

For double: $\varepsilon \approx 2 \times$ 10–16

- Rule of thumb: "not quite 16 digits of precision"

These are all relative numbers

Floating Point Consequences, cont

Just as decimal number system can represent only some rational numbers with finite digit count…

- Example: 1/3 cannot be represented

Binary number system can represent only some rational numbers with finite digit count

- Example: 1/5 cannot be represented

Beware of round-off error

- Error resulting from inexact representation
- Can accumulate
- Be careful when comparing two floating-point numbers for equality

Decimal Approx	Rational Value
.3	3/10
.33	33/100
.333	333/1000
...	

Binary Approx	Rational Value
0.0	0/2
0.01	1/4
0.010	2/8
0.0011	3/16
0.00110	6/32
0.001101	13/64
0.0011010	26/128
0.00110011	51/256
...	

Floating away ...

What does the following code print?

```
double sum = 0.0;
double i;
for (i = 0.0; i != 10.0; i++)
   sum += 0.1;
if (sum == 1.0)
   printf("All good!\n");
else
   printf("Yikes!\n");
```

A. All good!

B. Yikes!

C. (Infinite loop)

D. (Compilation error)

B: Yikes!

... loop terminates, because we can represent 10.0 exactly by adding 1.0 at a time.

... but sum isn't 1.0 because we can't represent 1.0 exactly by adding 0.1 at a time.