From last time: Declaring Variables in C (and Java)

Like Java, C requires variable declarations (some languages don't: awk, bash)

Declaring variables requires more from the programmer

- Extra verbiage
- Thinking ahead about how the variable will be used

But it has many benefits

- Allows compiler to check "spelling:" catch typos and type mismatches
- Allows compiler to allocate memory more efficiently
- Declaring the types of variables produces fewer surprises at runtime

Declaring Variables

Declaration statement specifies type of variable (and other attributes too)

Examples:

```
int i;
int i, j;
int i = 5;
const int i = 5; /* value of i cannot change */
static int i; /* covered later in course */
extern int i; /* covered later in course */
```

- Can have multiple declarations per statement. But can be risky …
- Unlike in Java, can use a local variable before assigning a value to it
  - But value is undefined

Declaring Variables (contd.)

- Unlike Java (and later versions of C), declaration statements in C89 must appear before any other kind of statement in any compound statement.
  - Note this **doesn't** mean that all declarations must be at the top of a *function* exclusively. Can be at start of a block

```
{
   int i;

   /* Non-declaration
      statements, even if
      they don't use j */

   int j;
}
```

Illegal in C89

```
{
   int i;
   int j;

   /* Non-declaration
   statements that use
   i and/or j. */
}
```

Legal in C89

iClicker Question

Indicate what value k is initialized to:

```
int j = 2;
int k = (1 < !!j);
```

A. !!j is undefined

B. 0

C. 1

D. Sorry, I'm having too much fun on social media

COS 217: Introduction to Programming Systems

Numbers

(in C and otherwise)

PRINCETON UNIVERSITY

The Old Ice Breaker

Q: Why do computer programmers confuse Christmas and Halloween?

A: Because Dec 25 is Oct 31

Agenda

Number Systems: Decimal, Binary, Hex, Octal + conversions among them

- Some for human beings, some for computers

Arithmetic with unsigned (non-negative) numbers

Negative numbers in a computer: representation and operations

Integers in C : representation and operations

The Decimal Number System

From Latin *decem* (“ten”)

Ten Symbols: e.g., **`0 1 2 3 4 5 6 7 8 9`**

- Different symbols in different languages
- The integers we use in C are decimal

Positional: **`2945`** $\neq$ **`2495`**

Base 10: $2945 = (2*10^3) + (9*10^2) + (4*10^1) + (5*10^0)$

What’s an example of a non-positional number system?

The Unary Number System

From late Latin ūnus (“one”)

One symbol: **1**

Effectively non-positional, since only one symbol: $\mathbf{1111_U = 1111_U}$

Base 1: $\mathbf{1111 = (1*1^3) + (1*1^2) + (1*1^1) + (1*1^0)}$

It is sum of products of powers, but easy to view as tally marks (without crossing fives)

- Integer 5 = 11111
- Integer 8 = 11111111
- Integer 25 = 1111111111111111111111111
- etc

Key Points that Apply to all Bases We Consider

All the other base systems we'll consider seriously are positional

Counting, addition, subtraction, work the same way in them as in decimal, just with different bases (number of symbols available)

Some are good for humans (base 10) and some for computers (powers-of-two bases)

There are simple methods for conversion

So now we can go over them very quickly ...

The Binary Number System

From late Latin *binarius* ("consisting of two"), from classical Latin *bis* ("twice"). Plus English suffix *-ary*

Two symbols: **0 1**

Positional: $\mathbf{1010_B \neq 1100_B}$

Base 2: $\mathbf{1010 = (1*2^3) + (0*2^2) + (1*2^1) + (0*2^0)}$

- "One zero one" in Base10 is integer value 101, and in Base 2 it is integer 5

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 – we'll see a more common name for 4 bits soon.

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
...	...

The Hexadecimal Number System

From ancient Greek ἕξ (*hex*, "six") + Latin-derived *decimal*

Sixteen symbols ("hexits"): 0 1 2 3 4 5 6 7 8 9 A B C D E F

Positional: $A13D_H \neq 3DA1_H$

Base 16: $A13D_H$: $A*16^3 + 1*16^2 + 3*16^1 + D*16^0 = 40960 + 256 + 48 + 13 =$

Computer programmers often use hexadecimal ("hex")

- In C: Hex numbers have a 0x prefix (0xA13D, etc.)

Why?

That's a zero, not a letter O

The Octal Number System

"octo" (Latin) ⇒ eight

Eight symbols: **0 1 2 3 4 5 6 7**

Positional: **1743 ≠ 7314**

Base 8: 7143_H: $7*8^3 + 1*8^2 + 4*8^1 + 3*8^0$

Computer programmers sometimes use octal (so does Mickey?)

- In C: represented with prefix 0 (e.g. 01743, etc.)
- Unix file permissions use octal: chmod 755 myfile
  - Changes permissions to _rwxr_xr_x

Converting to Decimal: Expand using Base Template

Binary to Decimal:

```
100101B = (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
```

MSB

LSB

Hex to Decimal:

$$
\begin{aligned}
25_H &= (2*16^1) + (5*16^0) \\
&= 32 + 5 \\
&= 37
\end{aligned}
$$

Similarly:

$$
\begin{aligned}
3B_H &= (3*16^1) + (11*16^0) \\
&= 48 + 11 \\
&= 59
\end{aligned}
$$

Octal to Decimal:

$$
\begin{aligned}
25_O &= (2*8^1) + (5*8^0) \\
&= 16 + 5 \\
&= 21
\end{aligned}
$$

Converting From Decimal: Subtraction Method

e.g., (Decimal) Integer to binary

- Determine largest power of 2 that's ≤ the number. Then write template from that power down:

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

- Then fill in template, subtracting that power of two and finding the next one <= the remainder, and so on.

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

Converting from Decimal: Division Method

e.g. (Decimal) Integer to binary

- Repeatedly divide by 2, write down 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 (L to R) from bottom to top: 100101_B

18 * 2 + _ _ _ _ _ 1

9 * 2^2 + _ _ _ _ 0 1

4 * 2^3 + _ _ _ 1 0 1

2 * 2^4 + _ _ 0 1 0 1

1 * 2^5 + _ 0 0 1 0 1

Similarly, (Decimal) Integer to Hex:

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

Read from bottom to top: 25_H

Conversion Among Power-of-2 Based Systems

Observations:

- For Hex, $16^1 = 2^4$, so every 1 hexit corresponds to a nybble (4 bits)
- Watch the left-padding with zeros

Binary to hexadecimal:

$0010\,0001\,0011\,1101_B$
$2\quad 1\quad 3\quad D_H$

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

Hexadecimal to binary:

$2\quad 1\quad 3\quad D_H$
$\cancel{00}10\,0001\,0011\,1101_B$

After conversion, discard leading zeros from binary number if appropriate

For octal, similar, but it's 3 bits

Agenda

Number Systems: Decimal, Binary, Hex, Octal + conversions among them

- Why did we do all this? Because program use these representations
- (Useful to memorize first 16 hexit representations in binary)

Arithmetic with Unsigned (non-negative) numbers

Negative numbers in a computer: representation and operations

Integers in C: representation and operations

First, Operations on Unsigned (Non-Negative) Integers

Unsigned much easier than signed. Non-neg numbers in natural way we think of them. Add and subtract in usual way, mod 2^k. Only, can overflow due to fixed space

Mathematics

- Non-negative integers' range is 0 to ∞

Computers

- Range limited by computer's word size
- Word size n bits $\Rightarrow$ range is 0 to $2^n - 1$ (largest unsigned n-bit binary number)
- Exceed this range, by trying to represent a larger number $\Rightarrow$ overflow

Typical computers today

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

A pretend computer for upcoming slides

- Assume n = 4, so range is 0 to $2^4 - 1$ (decimal 15)
- (All points made in those slides generalize to larger word sizes like 32 and 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: All mod 16 in this case

```
              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
All results are mod 2^4
17 mod 16 = 1
also interpreted as +ve, but weird

How would you detect overflow programmatically?

Subtracting Unsigned Integers

Subtraction

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

Start at right column
Proceed leftward
Borrow when necessary

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

Beware of overflow
All results are mod 2^4
-7 mod 16 = 9

How would you detect overflow programmatically?

Agenda

Number Systems: Decimal, Binary, Hex, Octal + conversions among them

Arithmetic with unsigned (non-negative) numbers

Negative numbers in a computer: representation and operations

Integers in C : representation and operations

Representing Negative Numbers #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

High-order bit denotes sign

`0 ⇒ positive, 1 ⇒ negative`

Remaining bits denote magnitude (unsigned)

$0101_B = 101_B = 5$

$1101_B = -101_B = -5$

Intuitive for human beings

+ easy to understand, easy to negate numbers

+ symmetric (half the bit patterns are +ve nos, half -ve)

Not good for computers

- two representations of zero
- different hardware circuits for negative numbers
  - Subtract instead of add if leading bit is 0
  - But unsigned arithmetic is different too

Not widely used for integers today

Saved by Modular Arithmetic

We want to calculate b – a

For n-bit numbers, a is the same as $(a + 2^n)$

So, b – a is the same as $(b - a) + 2^n$, which is $b + (2^n - a)$

So, if we can convert a to $(2^n - a)$, then to subtract a from b we can simply add b and $(2^n - a)$, hence using the same adder circuit

We can use the same adder circuit for addition and subtraction, signed and unsigned

$(2^n - a)$ is called the Two's Complement of a

- To negate a number, take its Two's Complement
- To subtract a number from another one, take its Two's Complement and add them

Calculating the n-bit Twos Complement of a Number

-x is 2^n – x. That's (1 followed by n 0s) – x. Drop the leading 1 to stay in n bits

```
                          1            11           111          1111
   00000000        00000000      00000000      00000000      00000000
 - 01001011      - 01001011    - 01001011    - 01001011    - 01001011
 ----------      ----------    ----------    ----------    ----------
                          1            01           101          0101

      11111          111111       1111111      11111111
   00000000        00000000      00000000      00000000
 - 01001011      - 01001011    - 01001011    - 01001011
 ----------      ----------    ----------    ----------
      10101          110101       0110101      10110101
```

So, how do you get 10110101 from 01001011? Simply flip the bits and add 1

$$\text{neg}(0101_B) = 1010_B + 1 = 1011_B$$
$$\text{neg}(1011_B) = 0100_B + 1 = 0101_B$$

Interpreting the n-bit Numbers in 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

Interpret leading-0 nos still as +ve, leading-1 as negative

A unifying definition: High-order bit has weight $-(2^{b-1})$

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

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

Pros and cons:

- not symmetric ("extra" negative number; -(-8) = -8)
- less intuitive for human beings

+ one representation of zero
+ same circuits add/subtract signed and unsigned int

What about One's 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

What if we didn't add 1?
To compute negative of number, just flip all its bits
Still, numbers beginning with 0 are +ve, with 1 are -ve

Definition (interpreting a ones-complement number)
High-order bit has weight $-(2^{b-1}-1)$

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

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

- Two representations for zero: may have to check for both as possible results of a calculation
- The "end round carry" problem

Adding Signed Integers in Two's Complement

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 in Two's Complement

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

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

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

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

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

Why does Two's Complement Arithmetic Work

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

$$
\begin{aligned}
&[-b] \bmod 2^4 \\
&= [2^4 - b] \bmod 2^4 \\
&= [2^4 - 1 - b + 1] \bmod 2^4 \\
&= [(2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [\mathrm{onescomp}(b) + 1] \bmod 2^4 \\
&= [\mathrm{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

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

$$
\begin{aligned}
&[a - b] \bmod 2^4 \\
&= [a + 2^4 - b] \bmod 2^4 \\
&= [a + 2^4 - 1 - b + 1] \bmod 2^4 \\
&= [a + (2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [a + \mathrm{onescomp}(b) + 1] \bmod 2^4 \\
&= [a + \mathrm{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

Agenda

Number Systems: Decimal, Binary, Hex, Octal + conversions among them

Arithmetic with unsigned (non-negative) numbers

Negative numbers in a computer: representation and operations

Integers in C : representation and operations

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 (Note: no. of bits per byte is unspecified, but it's safe to assume it's 8)
- Sizes of other integer types are not fully specified in C but are constrained:
  - `int` was intended to be "natural word size" of hardware, but isn't always
  - 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

If Sizes aren't Fixed in C, How do I Tell Size?

`sizeof` operator returns the size of a type or a variable (or an expression's result)

- 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

Default base is 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
  - There's 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

Agenda

Number Systems: Decimal, Binary, Hex, Octal + conversions among them

Arithmetic with unsigned (non-negative) numbers

Negative numbers in a computer : representation and operations

Integers in C: representation and a few operations (more on the slides)

- Plus appendix on floating point

Reading / Writing Numbers

Motivation

- Numbers come in as sequences of characters, but must be interpreted as numbers with types
- Could provide special read/write functions for each type: getchar(), putshort(), getint(), putfloat() ...
- Or parameterized functions: one function that takes a specification for the kind of data to expect

C library provides 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 (remember, no Boolean type)
    - Each evaluates to FALSE ⇒ 0, TRUE ⇒ 1
- Cast operator: (type)
- Bitwise operators: ~ & | ^ >> <<
    - C designed to be close to the hardware
    - Bitwise operators enable fast calculations/arithmetic

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? (Think about shifting digits right in decimal)

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

`5 << 1 ⇒ 10`

0101_B 1010_B

`3 << 2 ⇒ 12`

0011_B 1100_B

`3 << 3 ⇒ 8`

0011_B 1000_B ← Overflows. But Results are correct mod 2^4

What is the effect arithmetically?

Other Bitwise Operations on Unsigned Integers

Bitwise NOT (~ in C)

- Flip each bit (don't forget leading 0s)

```
~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. All bits ANDed with 0 are 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. OR with 1 is 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
```

^ with 1 flips value of bit

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

Aside: Using Bitwise Ops for Arithmetic

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

$x * 2^y == x << y$
- $3*4 = 3*2^2 = 3<<2 \Rightarrow 12$

Fast way to multiply by a power of 2

$x / 2^y == x >> y$
- $13/4 = 13/2^2 = 13>>2 \Rightarrow 3$

Fast way to divide unsigned by power of 2

$x \% 2^y == x \& (2^y-1)$
- $13\%4 = 13\%2^2 = 13\&(2^2-1)$
  $= 13\&3 \Rightarrow 1$

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 *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?

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?

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 using signed ints for bit-twiddling (you're not usually using the ints for their integer values when you do this, anyway)

Assignment Operator

Many high-level languages provide an assignment *statement*

C provides an assignment **operator**

- Operator takes operands as inputs and produces a result
- Performs assignment and then **evaluates to the assigned value. Why?**
  - Allows assignment to appear within larger expressions
  - Terseness of code
  - But be careful about precedence. Extra parentheses often needed. Can confuse reader.

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 Assignment Operators in C

Motivation

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

Assignment in C

- Useful: Introduce += operator to do things like d += e
  - Extend to –= *= /= ~= &= |= ^= <<= >>=
  - All evaluate to whatever was assigned. i.e., if a was 1, a+=2 is a = a + 2 so evaluates to 3
- Pre-increment and pre-decrement: ++d ––d. Evaluates to d+1 and d-1
- Post-increment and post-decrement (evaluate to *old* value): d++ d––. Evaluates to d

Sample Exam Question (Spring 2017, Exam 1)

1(b) (12 points/100) Suppose we have a 7-bit computer. Answer the following questions.

(i) (4 points) What is the largest unsigned number that can be represented in 7 bits?

In binary: In decimal:

(ii) (4 points) What is the smallest (i.e., most negative) signed number represented in 2's complement in 7 bits?

In binary: In decimal:

(iii) (2 points) Is there a number n, other than 0, for which n is equal to –n, when represented in 2's complement in 7 bits? If yes, show the number (in decimal). If no, briefly explain why not.

(iv) (2 points) When doing arithmetic addition using 2's complement representation in 7 bits, is it possible to have an overflow when the first number is positive and the second is negative? (Yes/No answer is sufficient, no need to explain.)

Sample Exam Question (Fall 2024, Exam 1)

1 (d) (1 point /32) If acc is of type int, write a statement that has the same effect as the line acc *= 10, without using multiplication and using no more than one addition operation.

(Hard!) Sample Exam Question (Fall 2020, Exam 1)

a. In the two ranges below, replace the "____" with the inclusive upper and lower bounds of decimal numbers that do not change value when moving from i-bit two's complement to (i+1)-bit two's complement (for example, when moving from four bits to represent integers to using five bits to do so). The two ranges consider two different possibilities for changing an i-bit value into an (i+1)-bit value:

If we make the change by prepending a 0 onto the front of the i-bit representation (e.g., 1001 -> 01001):

____ <= x <= ____

If we make the change by prepending a 1 onto the front of the i-bit representation (e.g., 1001 -> 11001):

____ <= x <= ____

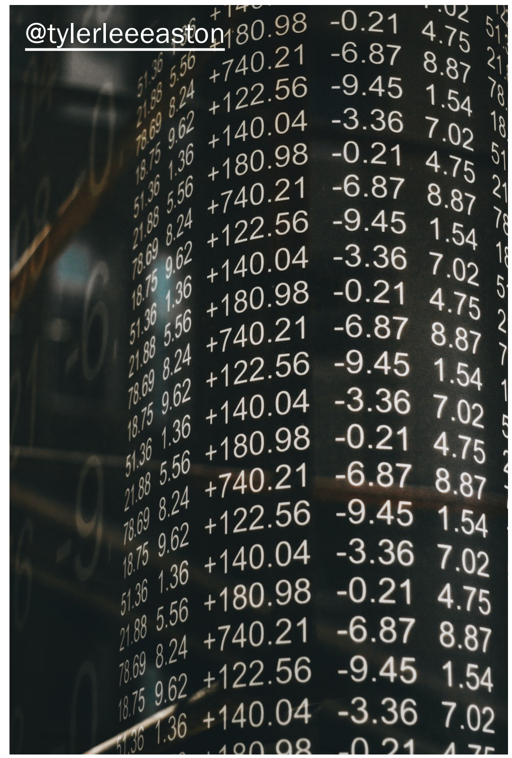
b. In the range below, replace the "____" with the inclusive upper and lower bounds of armlab C int literals for which the expression still compiles and does not change value when adding a 0 before the first character of the literal (for example, 217 -> 0217):

____ <= x <= ____

Hint 1: does a literal 09 compile?

Hint 2: the word ”expression” is intentional; note that the first character of a signed int is not necessarily a digit.

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):

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

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

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

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

- Example: 1/5 cannot be represented

```
Binary      Rational
Approx      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
...
```

Beware of round-off error

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

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.