Day 15 (Fri 02/25)

- exercise 14 review
- day 15 recap questions
- exercise 15
  - note: Part 3 is optional

Announcements/reminders:

- HW3: due *this evening* by 11pm
- HW4: due Friday 3/4 by 11pm
  - written assignment, no late submissions
- midterm project: will introduce in class on Monday

Exercise 14 review

Converting from string of '0' and '1' digits to binary integer:

```
int str_to_int(char msg[], int len) {
  int result = 0;

  for (int i = 0; i < len; i++) {
    int index = len - i - 1;
    char c = msg[index];
    if (c == '1') {
      result |= (1 << i);
    }
  }

  return result;
}
```

Exercise 14 review (continued)

Converting from binary integer to string of '0' and '1' digits:

```
void int_to_str(int num_encrypted, char msg_encrypted[], int len) {
  char bits[32];
  int num_bits = 0;

  for (int i = 0; i < len; i++) { // generate bits in reverse order
    if ((num_encrypted & 1) == 1)
          { bits[num_bits] = '1'; }
      else { bits[num_bits] = '0'; }
    num_bits++;
    num_encrypted >>= 1;
  }

  // ...copy digits to msg_encrypted in reverse order...
}
```

Exercise 14 review (continued)

```
  // Perform the encryption

  for (int i = 1; i < n; i++) {
    num_encrypted ^= (num_encrypted << 1);
  }
```

Day 15 recap questions

1. What is two’s complement representation?
2. How does representation of integers and floating-point values differ in C?
3. What is type narrowing?
4. What is type casting?
5. What is type casting?
6. What is the output of the code segment below?

1. Two's complement is how *signed* integers are represented on all modern computer architectures.

Idea: most significant bit makes a *negative* contribution to the value of the integer.

Consider the bit string 10000101

As an 8 bit *unsigned* value: 128 + 4 + 1 = 133

As an 8 bit signed two's complement value: -128 + 4 + 1 = -123

Why two's complement is used: arithmetic (addition, subtraction, etc.) works exactly the same way for both unsigned and signed values.

Negating a two's complement value: invert all bits and add 1.

Why?

A bit string where every bit is 1 has the value -1

a + ~a = -1 (e.g., 10010110 + 01101001 = 11111111)

so, a + ~a = -1

rearranging: -a = ~a + 1

2. How does representation of integers and floating-point values differ in C?

Integer representation: either *unsigned* or *signed* (two's complement on any modern CPU)

Floating-point representation: IEEE 754

IEEE 754 is essentially base-2 scientific notation

(Normalized) floating point values are represented as

  +/- 1.x * 2^y

  x is the fraction (represented in base 2)

  y is the exponent (represented in base 2, can be positive or negative)

Arithmetic on floating point values may involve rounding, results should generally be considered to be approximate.

Also: some numbers can't be represented exactly. For example, 0.1 has no exact representation (becomes a "repeating decimal" in the fraction.)

3. Type narrowing: converting a "larger" type to a "smaller" type, e.g., double to int.

May lose information.

For example:

```
float f_val = 3.5;
int i_val = f_val;          // narrowing conversion, i is 3
```

4. Type promotion: converting a "smaller" type to a "larger" type, e.g., int to double.

Will *generally* not lose information, although some promotions (e.g., int to float) may lose information in some cases.

5. What is type casting?

Type casting is an *explicit* conversion from one type to another.

Can be used to eliminate warnings in some cases:

```
size_t len = strlen(str);
for (int i = 0; i < (int)len; i++) {        // cast prevents compiler warning
  char c = str[i];
  // ...
}
```

An explicit type cast is a good idea when your program does a narrowing conversion. (Lets the programmer know the conversion is intentional.)

6. What is the output of the code segment below?

```
int n = 32065;
float x = 24.79;
printf("int n = %d but (char) n = %c\n", n, (char) n);
printf("float x = %f but (long) x = %ld\n", x, (long) x);


// Note: in base 16, 32065 = 7D41
// 41 in base 16 is 4×16 + 1 = 65
// ASCII 65 is 'A'
```