Day 14 (Wed 02/23)

- exercise 13 review
- day 14 recap questions
- exercise 14

Announcements/reminders

- HW3: due Friday by 11pm
- HW4: soon (release tomorrow)

Exercise 13 review:

In a struct data type, "struct" is part of the name of the data type.

E.g.:

```
struct Stat {
  int num_of_goals;
  int num_of_assists;
  float pass_accuracy;
  int min_played;
  int num_of_shots;
  float shot_accuracy;
};
```

Data type is "struct Stat"

Exercise 13 review:

Common shortcut: use "typedef" to avoid the need to use the "struct" keyword when referring to the data type. E.g.:

```
typedef struct Stat {
  int num_of_goals;
  int num_of_assists;
  float pass_accuracy;
  int min_played;
  int num_of_shots;
  float shot_accuracy;
} Stat;
```

Now "Stat" can be used as the name of the data type.

The "main" function assumes that you have done this for each struct type.

Exercise 13 review (continued):

Find the player with the latest signing date:

```
index = 0;
for (int i = 1; i < TEAMSIZE; i++) {
  Player *last = &team[i];
  Player *p = &team[i];
  if ( /* p's date is later than last's date */ ) {
    index = i;
  }
}
```

Exercise 13 review (continued):

Use valgrind to analyze memory use:

```
==288== 24 bytes in 1 blocks are definitely lost in loss record 1 of 3
==288==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==288==    by 0x108910: main (main.c:10)
==288==
==288== 132 bytes in 11 blocks are definitely lost in loss record 2 of 3
==288==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==288==    by 0x108B93: create_player (soccer.c:10)
==288==    by 0x108E09: create_team (soccer.c:41)
==288==    by 0x108927: main (main.c:11)
==288==
==288== 240 bytes in 10 blocks are definitely lost in loss record 3 of 3
==288==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==288==    by 0x108C50: create_player (soccer.c:22)
==288==    by 0x108E09: create_team (soccer.c:41)
==288==    by 0x108927: main (main.c:11)
```

Exercise 13 review (continued):

Freeing memory:

```
for (int i = 0; i < TEAMSIZE; i++) {
  free(team[i].date);
  free(team[i].stat);
}
```

Observation: for each Player instance in the team array, the date and stat fields point to dynamically-allocated objects

Day 14 recap questions:

1. How do we read/write binary files in C?
2. What character represents the bitwise XOR operation? How does it differ from the OR operation?
3. What happens if you apply the bitwise operation on an integer value? (extra: what if we apply to floats)
4. What is the result of (15 >> 2) || 7?
5. What is the result of (15 >> 2) | 7?

1. Use the "rb" or "wb" when calling fopen, and use fread or fwrite to read binary data value(s)

E.g., read array of 20 int values from a file of binary data:

```
FILE *in = fopen("input.dat", "rb");
if (in == NULL) { /* handle error */ }
else {
  int *arr = malloc(sizeof(int) * 20);
  int rc = fread(arr, sizeof(int), 20, in);
  if (rc != 20) { /* handle error */ }
}
// arr now points to array of 20 elements read from input file
```

WARNING: this code is not portable across CPU architectures due to byte ordering. For multi-byte data values some CPUs store least significant byte first ("little endian"), some CPUs store most significant byte first ("big endian")

Binary data representation

"Binary" means "base 2"

Digital computers represent all numbers using base-2 rather than base 10. For example:

42 in base 10: 4×10^1 + 2×10^0

42 in base 2: 1×2^5 + 0×2^4 + 1×2^3 + 0×2^2 + 1×2^1 + 0×2^0
= 1×32 + 0×16 + 1×8 + 0×4 + 1×2 + 0×1

42 as a sequence of binary digits ("bits"): 101010

All data values are represented in binary (base-2) at the machine level.

"Bitwise" operators allow you to work directly with binary values.

2. Bitwise OR: combine two binary values by computing the result of the OR operation on each pair of binary digits.

Operator: |

Logic: 0|0=0, 0|1=1, 1|0=1, 1|1=1

Bitwise XOR: combine two binary values by computing the result of the XOR (exclusive or) operation on each pair of binary digits.

Operator: ^

Logic: 0^0=0, 0^1=1, 1^0=1, 1^1=0

Bitwise AND: combine two binary values by computing the result of the AND operation on each pair of binary digits.

Operator: &

Logic: 0&0=0, 0&1=0, 1&0=0, 1&1=1

3. The two-operand bitwise operators (|, ^, &) perform a logical operation (OR, XOR, AND) on each pair of bits in the two operands.

The operands must be values belonging to an integral (integer-like) type.

E.g., int, unsigned, long, unsigned long, char, unsigned char, etc.

Bitwise operations may NOT be performed on floating point (float or double) values.

4. || is the "logical OR" operator.

What is the result of (15 >> 2) || 7?

15 in base-2 is 1111

">>" is the right shift operator, shifting 1111 two bits to the right yields 0011, which is equal to 3

Any non-zero integer is considered TRUE, so 3 is true.

If the left operand of || is true, the entire expression is true (and the right operand is not evaluated.) All logical and relational operators yield 0 when false and 1 when true.

So, the result of (15 >> 2) || 7 is 1

5. | is the bitwise OR operator

What is the result of (15 >> 2) | 7?

15 in binary is 1111

(15 >> 2) is 0011

7 in binary is 0111

  0011
| 0111

  0111 -- bit is 1 in each position where either operand has a 1 bit

which is equal to 7