CMSC 216
Introduction to Computer Systems
Lecture 3
Introduction to C, cont.

Administrivia

- Read Chapters 4 and 5 of Reek

Chapter 4, Reek

STATEMENTS

Assignment statements

- Any expression can appear as a statement
- An assignment is just an expression (typically used as an expression statement)
  - The **=** is an *operator* in C (and right associative)
  - legal expression statements, assuming **`int x, y;`**

```
x = 3;
y + 3;
x == y;
```

- An expression statement is useful when the expression has a *side effect*
- An assignment returns a value - whatever value was assigned to the variable on the left hand side of the assignment

```
y = x = 123;
y = 5 + (x = 3);
```

C control statements

- These are very similar to Java, with one important difference: C has no boolean type
  - scalar expressions used instead; 0 is false, everything else is true
- C has **`if`**/**`else`**, **`while`**, **`for`**, **`do`**-**`while`**, and **`switch`** statements, just as Java does
  - but can't declare variables in **`for`** loop header
- **`break`**: immediately end loop
- **`continue`**: skip remainder of loop body, return to beginning of loop
  - in case of **`for`** loop, perform increment
- don't abuse **`break`**/**`continue`**, and rarely use **`continue`** if at all

Perfectly valid boolean examples

```
int c = ???;
if (c)
  f1();
if (c != 0)
  f2();
if (c == 2)
  f3();
if (c = 2)
  f4();
```

Function	-1	0	1	2
`f1()`	Y	N	Y	Y
`f2()`	Y	N	Y	Y
`f3()`	N	N	N	Y
`f4()`	Y	Y	Y	Y

The table shows which functions will be executed if certain values are assigned to **`c`** during its initialization. **`f4()`** always executes! Why?

Chapter 5, Reek

OPERATORS

Basic arithmetic operators

- Most of the operators you know from Java are in C
  - **`+`** adds, **`-`** subtracts, **`/`** divides, **`%`** performs remainder (modulus), **`*`** multiplies
- C also performs integer division (rather than floating point division) when both operands are integers
  - So, **`3.0 / 2.0 == 1.5`** (also **`== 3 / 2.0`**)
  - But, **`3 / 2 == 1`**

Numeric base conversion

- Computer only works in binary (base 2)
- People generally prefer decimal (base 10), but sometimes hexadecimal (base 16) is also useful
- Converting between these representations is important for us - hex is easily translated to binary, but decimal to/from either hex or binary can be a bit challenging
- It pays to memorize a few powers of 2

Simple conversion table

Decimal	Hex	Binary
0	`0x0`	`0000`
1	`0x1`	`0001`
2	`0x2`	`0010`
3	`0x3`	`0011`
4	`0x4`	`0100`
5	`0x5`	`0101`
6	`0x6`	`0110`
7	`0x7`	`0111`

Decimal	Hex	Binary
8	`0x8`	`1000`
9	`0x9`	`1001`
10	`0xA`	`1010`
11	`0xB`	`1011`
12	`0xC`	`1100`
13	`0xD`	`1101`
14	`0xE`	`1110`
15	`0xF`	`1111`

Working with other bases

- `printf()` has format specifiers to allow printing of values in hex or octal (not binary, though)
  - `%x` / `%X` : hexadecimal (`a-f`/`A-F`)
  - `%o` : octal
- "`%08x`" often used for printing `unsigned int`s as zero-padded, 8-digit hex numbers
- There also exist similar mechanisms for reading in hexadecimal and octal representations of numbers using `scanf()`
- But remember, all values are stored in binary, regardless of what base the numeric literal is!

Bit operations

- Numbers are represented using a fixed number of bits
  - typically a **`char`** is 8 bits, an **`int`** is 32 bits, a **`long`** is 32 or 64 bits
- C permits direct manipulation of the bits within a number
  - this is powerful and allows you to do exactly what you want
  - these can be nonportable: it's easy to write programs that don't work the same on different platforms
  - usually unsigned integers are used for bitwise operations
- An **`unsigned char`** as a series of bits:

0	**1**	**0**	**1**	**0**	**1**	**1**	**0**

leftmost (or high-order) bit

rightmost (or low-order) bit

Shifting operators

- The **<<** and **>>** operators shift a value a given number of bits to the left or right, respectively
- Should only be used with unsigned integer as left operand
- Right operand must be between 0 and (# of bits of left operand) - 1
- Zero bits replace the vacated bits
- Examples:

```
unsigned int a = 0x55555555;  /* 0101 ... */
printf("a << 2: %08x\n", a << 2);
printf("a >> 3: %08x\n", a >> 3);
printf("a: %08x\n", a);
```

Bitwise operators

- We can use the logical operations of AND, OR, NOT, and XOR on the bits of numbers, using bitwise operators
- Bitwise AND: **`&`**
- Bitwise OR: **`|`**
- Bitwise NOT (unary): **`~`**
- Bitwise XOR: **`^`**

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Bitwise operator examples

```
unsigned int a = 0x5555ffff, b = 0xaaaa1111;
unsigned int ones = 0;
ones = ~ones;
printf("a AND b: %08x\n",            a & b);
printf("a AND 0: %08x\n",            a & 0);
printf("a AND ones: %08x\n",         a & ones);
printf("a OR b: %08x\n",             a | b);
printf("a OR 0: %08x\n",             a | 0);
printf("a OR ones: %08x\n",          a | ones);
printf("a XOR b: %08x\n",            a ^ b);
printf("a XOR 0: %08x\n",            a ^ 0);
printf("a XOR ones: %08x\n",         a ^ ones);
printf("Complement of a: %08x\n", ~a);
```

Bitmasking

- Using the bitwise operators with specific bit patterns, or masks, we can access specific bits in an integer value
  - clear bit: AND with 0
  - check bit: AND with 1
  - set bit: OR with 1
  - flip bit: XOR with 1

Bitmasking in action

- Goal: to make bits 2-4 (from left) have bit pattern 110

```
unsigned char foo = 0xab; /* 0xab: 1010 1011 */
foo &= 0x8f;  /* 0x8f: 1000 1111 - clear 2-4 */
foo |= 0x60;  /* 0x60: 0110 0000 - set 2-4 */
```

- How would we set the second-to-least significant byte in an **`int`** to the value in **`foo`**?