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;`
    ```c
    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
  ```c
  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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1()</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>f2()</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>f3()</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>f4()</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0x2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0x3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0x4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0x5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0x6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0x7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>0x8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>0x9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>0xA</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>0xB</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>0xC</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>0xD</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>0xE</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>0xF</td>
<td>1111</td>
</tr>
</tbody>
</table>
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:

```c
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: ^

\[
\begin{array}{ccc}
\& & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
| & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{ccc}
^ & 0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]
Bitwise operator examples

unsigned int a = 0x5555ffff, b = 0xaaaaa1111;
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

```c
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`?