CMSC 216
Introduction to Computer Systems
Lecture 4
Introduction to C, cont.
Larry Herman & Alan Sussman

Notes
• Project 1 questions – see TAs or instructors in office hours
  – public tests available in public Grace directory, and submit server open, today
• Read Chapter 5 of Reek

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:

```
1 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?

Compound assignment

• C supports several compound assignment operators that can save you time and typing
  – include +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
  – does NOT include !=

• Can reduce possibility of errors:
  a[i * j + k / 2] = a[i * i + k / 2] + 10;
a[i * j + k / 2] += 10;

• But be careful:
  a[f(b) % n] = a[f(b) % n] + 1;
  vs.
  a[f(b) % n] += 1;

Increment/decrement operators

• These work just like they do in Java
• Remember the difference between ++i and i ++?
• What does this function output?
  void foo() {
    int i = 10, j = 5;
    printf("%d\n", --i);
    printf("%d\n", j++);
    printf("%d\n", i++ + j);
  }
**The sizeof operator**

- Unary operator, evaluates to the number of bytes necessary to hold its operand
- Operand can be an expression or a type name
- Does NOT evaluate the expression
- Examples:
  
  ```c
  int i = 5;
  printf("%d\n", sizeof(i));
  printf("%d\n", sizeof(unsigned char));
  printf("%d\n", sizeof(i++));
  printf("%d\n", i);
  ```

**Other unary operators**

- *(typename)* is a unary operator
  - Works just as in Java
- There is also a − operator which performs arithmetic negation
  - so code like "a *= -1;" is really just wasteful
- We'll discuss the unary & and * operators soon, when we discuss pointers

**Boolean operators**

- Relational operators: <, >, <=, >=
- Equality operators: ==, !=
- Logical operators: &&, ||, !
- Function just as you'd expect from working in Java, except that they evaluate to 1 (if true) or 0 (if false), so this example actually makes sense:
  
  ```c
  int i;
  i = (! 3) == (4 < 2);
  i = (! 2) || (5 && i);
  ```
  - Remember that the logical operators do short-circuit, affecting whether or not parts of expressions get evaluated

**Conditional operator**

- The only ternary operator
- Syntax: `expr1 ? expr2 : expr3`
- If expr1 is nonzero, evaluates to expr2; otherwise, evaluates to expr3
- Don't abuse this; use it only when it helps reduce code duplication:
  
  ```c
  if (a > 5)
      b[2 * c + f(d / 5)] = 3;
  else
      b[2 * c + f(d / 5)] = -20;
  ```
Comma operator

• Yes, the comma is an operator
• Evaluates left operand, then right operand
• Value of expression with comma is value of last operand
• Has lowest precedence of all operators
• So what gets stored in \( i \) after each statement? What does each statement evaluate to?
  \[ i = 1, 2, 3, 4; \]
  \[ i = (1, 2, 3, 4); \]

Precedence and associativity

• Different operators can fall on different precedence levels
• Ties among levels are settled by associativity rule for that level
• Some operators impose restrictions on evaluation order, but aside from that, compiler can optimize
• Full table in *Pointers on C*, pgs. 114-115

Lvalues and Rvalues

• An rvalue is anything that can appear on the right side of an assignment statement
  – virtually any expression
• An lvalue is anything that can appear on the left side of an assignment statement
  – values that represent a place to store a value
• The right and left sides of an assignment statement are treated differently
  – right hand side is a value, left hand side is a location to store a value (an address)