CMSC 313
Introduction to Computer Systems
Lecture 4
Introduction to C, cont. &
Arrays and Strings

Alan Sussman
als@cs.umd.edu

Administrivia
• Project 1 questions – see TAs in office hours or email Prof. Sussman
  – public tests available in public Grace directory, and submit server open
• First quiz Wednesday in discussion
• Read Chapter 5 of Reek
  – parts of Chapters 8 and 9 covered today and Wednesday

OPERATORS

Bitwise operators
• Operate on the bits of operand(s)
• Bitwise AND: &
• Bitwise OR: |
• Bitwise NOT (unary): ~
• Bitwise XOR: ^
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 0
  – check bit: AND 1
  – set bit: OR 1
  – flip bit: XOR 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?

```c
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: \( \text{expr1} \ ? \ \text{expr2} \ : \ \text{expr3} \)
• If \( \text{expr1} \) is nonzero, evaluates to \( \text{expr2} \); otherwise, evaluates to \( \text{expr3} \)
• 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?
  
  ```c
  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)
Implicit type conversion

- Arithmetic operators require their operands to be of the same type to perform the operation
- `int` is actually “smallest” type used
- There is a hierarchy of types (Reek, § 5.4.2)
  - floating-point numbers over integers
  - wide over small
  - unsigned over signed
- Operation result is of the new type
- What to do? 2000000000 * 3

Mixed-type assignments

- RHS is converted to LHS type before storage
- Can mean either promotion or truncation

```c
char a, b = 'b', c = 'c';
float f = 2.25, g = 4.9999;
unsigned int i, j;
unsigned char ch;
a = b + c; /* a = 98 + 99 = 197? */
i = f; j = g; /* i: 2; j: 4 */
i = ch = 0xabcd; /* i: 0xcd; ch: 0xcd */
```

Arrays in C

- Much like Java (or many other languages)
  - All elements are the same type
  - Elements indexed by the subscript operator `[ ]`
- Sizes must be known at compile time (constant expressions only) and are static
- Can’t assign to arrays (can initialize, though)
- Can use `==` and `!=`, but meaning isn’t what you might think (wait until pointers)
- Syntax for creating arrays is slightly different
  - C: `int a[5];` creates array of 5 `ints`
  - Java: `int[] a = new int[5];` creates array of 5 `ints`
Array initialization

- Supply a list of values in braces, separated by commas:
  ```c
  int a[5] = {1, 1, 2, 3, 5};
  ```
- Occurs when array is first created
  - When this occurs depends on the array's storage class
  - Also means you can't initialize after declaration
  - And you can't initialize with variable expressions
- Zeroes pad the array when initializer is short
- Use of an initializer allows size to be omitted
  - Can't omit size otherwise when declaring local variables
  (parameters are an exception, though)
- Initializers with excess elements cause errors

Array initialization examples

- ```c
  int a[3] = {1, 4, 7};
  int b[5] = {2, 8};
  int c[] = {3, 9, 5, 2 + 6};
  int d[1000] = {0};
  ```

These are illegal (assuming i is an int):

- ```c
  int w[i];
  int x[];
  int y[4] = {2 * i, 3 + 2};
  int z[5]; /* this alone is OK */
  z = {1, 1, 2, 3, 5};
  ```

Parameters in C

- You can use parameters as variables, but why is it safe?
- In C, variables are passed by value – a copy is passed

  ```c
  int abs_value(int x) {
    if (x < 0)
      x = -x;
    return x;
  }
  ```

  ```c
  int main() {
    int n = -17, a;
    a = abs_value(n);
    printf("%d %d\n", a, n);
    return 0;
  }
  ```

Array parameters

- Array parameters act as if they were passed by reference (as we'll discuss later)
- If a function modifies elements of an array parameter, the array passed in is modified

  ```c
  void function(int a[]);
  ```

  ```c
  int array[10];
  function(array);
  ```

- Sizes for array parameters are ignored – only types matter
  - So "void function(int a[12397]);" is equivalent to the above prototype
Array parameters, cont.

- You generally have to pass array size along with the array.
- Functions only know about the elements of an array—the size of an array parameter isn’t known.

```c
void multiply_array(int factor, int arr[], int ct) {
    int i;
    for (i = 0; i < ct; i++)
        arr[i] *= factor;
}

int main() {
    int a[] = {1, 2, 3};
    multiply_array(5, a, 3);
    printf("[\%d, \%d, \%d]\n", a[0], a[1], a[2]);
    return 0;
}
```

---

Use of symbolic constants

- `#define` preprocessor directive
- `#define name value`
- All occurrences of `name` in the source file are replaced by `value`
- Used to define constants for things such as array sizes and other values to improve program maintainability

```c
#define ARR_SIZE 3
int main() {
    int i, a[ARR_SIZE] = {1, 2, 3};
    multiply_array(5, a, ARR_SIZE);
    printf("[\%d, \%d, \%d]\n", a[0], a[1], a[2]);
    return 0;
}
```