Announcements

- Program #2
  - Due 2/27/07

- Reading
  - Chapter 11 (by Thursday)

Bit Operations

- Numbers are represented by a fixed number of bits
  - typically int = 32 bits, char = 8 bits, long = 32 or 64 bits

- C permits direct manipulation of bits within a number
  - This is powerful: can get exactly what you want
  - This can be non-portable: easy to write programs that don't work on different types of computers

- Numbers as a series of bits:

  1 0 1 1 1 0 0 0

  left most bit right most bit
Bit Shift Operator

- **C has operators to bit shift numbers**
  - Number of bits changed depends on size of variable

- **Left Shift** (number << n)
  - Move each bit of number to the left by n bit positions
  - Leftmost n bits discarded
  - Rightmost n bits gets 0

- **Right Shift** (number >> n)
  - Move each bit of number to the right by n bit positions
  - Rightmost n bits discarded
  - Leftmost n bits gets 0 (or can replicate sign bit - so 1 if negative)
    - For unsigned gets 0
    - For signed, it is implementation dependent
  - There is no >> n operator in C.

Examples of Bit Shift

```c
unsigned int a, b;
a = 0x0000 0010;
b = a >> 1; // b is now 0x0000 0008 */
b = a >> 4; // b is now 0x0000 0001 */
b = a >> 5; // b is now 0x0000 0000 */
b = a << 1; // b is now 0x0000 0020 */
b = a << 4; // b is now 0x0000 0100 */
b = a << 5; // b is now 0x0000 0200 */
```
Bitwise Operators

- **And (&&)**
  - for each bit in two numbers, "and" the bits together
- **Or (||)**
  - for each bit in two numbers, "or" the bits together
- **Xor (^)**
  - for each bit in two numbers, "xor" the bits together

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Or</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Xor</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Assignment Operator

- Assignment is an operator and the operations specified take place in a sequence:
  - `x = y + 3;`
  - `a = x = y + 3;`
  - assignment is right associative, so
    - `a = x = y + 3` is the same as `a = (x = y + 3)`
  - value of assignment operator is result of assignment
    - if truncation is applied, the truncated value is used
  - Consider: (reminder: char is only 8 bits)
    ```
    unsigned char x;
    unsigned int a, b;
    b = 0xabcd;
    a = x = b;
    at the end, a = 0x00cd;
    ```
Compound Assignment

- Shorthand to combine binary operator and assignment
  - += add right operand to the left operand and update left
    - a += 3 is equivalent to a = a + 3
  - Also: -=, *=, %=, <<=, >>=, &=, ^=, |=

- Handy for complex expressions in LHS
  - a[ i * 2 + j - f(n)] = a[ i * 2 + j - f(n)] + 3;
  - a[i * 2 + j - f(n)] += 3;
  - assumes f(n) has no side effects and returns the same value on the two sequential calls to that same function

Unary Operators

- ! logical not operator
  - if the operand is true, result is false
  - if the operand is false, result is true
  - is really an integer operand
    - produces 0 for false
    - produces 1 for true
  - b = !(a == 3);  b = (a != 3);  b = (a < 3 || a > 3);

- ~ bit-wise negation (ones complement)
  - flip each bit position
  - b = ~a;

- negation (twos complement)
  - changes sign of a number
  - a = 3;
  - b = -a;  /* b is now -3 */
Unary Operators (cont.)

- **&** Returns the address of a variable
  - int a, b;
  - a = 3;
  - b = & a; /* b now holds the location (address) of a */

- ***** Dereferences a pointer
  - int a, b;
  - a = 3;
  - b = & a; /* b now holds the location (address) of a */
  - printf("%d and %d\n", a, *b); /* 3 printed twice */

- **sizeof** - return number of bytes in variable or type
  - int a;
  - sizeof int
  - sizeof(int)
  - sizeof(a)

- **(<typeName>)** - cast to a new type
  - float a; int c,d;
  - a = (float) c / d;

Unary Operators (cont.)

- **++ increment operator**
  - can be prefix or postfix
  - ++a - add one to a and use new value as result of expression.
  - a++ - add one to a and use old value as result of expression.

- **-- decrement operator**
  - can be prefix or postfix
  - --a - subtract one from a and use new value as result of expression.
  - a-- - subtract one from a and use old value as result of expression.

Examples:

- int c=2,a=10,*b;
- b = &a;
- c = *b++;
Relational Operators

- Take two operands, result is integer
  - 0 for false, 1 for true
  - > >= < <= != ==
- != is not equal
- == is equal
- Can use variable in conditional
  - int a;
  - if (a) { … is the same as if (a != 0) { … [ note: different if (a==1)]
  - if (!a) { … is the same as if (a == 0) {

Logical Operators

- && - and
  - if both operands are non-zero result is 1
  - else result is 0
  - uses short-circuit evaluation
    - if the first operand is not true, second is not evaluated
    - if ((a++) && (--b)) { .. }
- || or
  - if either operands is non-zero result is 1
  - else result is 1
  - also uses short-circuit evaluation
    - if the first operand is true, second is not evaluated
- Caution: && is not the same as &
  - & performs a bitwise operation
- Caution: || is not the same as |
  - | performs a bitwise operation
Other Operators

- **trinary conditional operator**
  - expr1 ? expr2 : expr3
  - if expr1 is non-zero, expr2 is evaluated and is the result
  - if expr1 is zero, expr3 is evaluated and is the result
  - a = (2 > 3) ? 65 : 17;

- **Comma operator**
  - evaluates both operands, rightmost is the result
  - a = (1,3,2); vs. b = 1,3,2;
    - assignment is higher precedence than , (the comma)
    - a ends up 2 and b ends up 1

Type Conversion

- **Promotion of char and short**
  - in expressions, char and short are promoted to int
    - char a, b; int c;
    - a = b + c;
      - b is converted to int, then the sum is truncated
    - even if a, b and c are all char – b and c are converted to int

- **Arithmetic Conversions**
  - arithmetic can't be performed on operators of different types
    - converted to "higher" type
      - long double
      - double
      - float
      - unsigned long int
      - long int
      - unsigned int
      - int
Precedence

- There is a set of rules about precedence of operator
  - Most important precedence rule:
    - When in doubt, put in () to ensure correct order
  - Order (highest to lowest):
    - ()
    - Function call, subscript, postfix increment/decrement
    - rest of unary operators
    - type conversion
    - Arithmetic operators
    - Relational operators
    - Bit operators
    - Assignment operators
    - comma operator