CSC 270 – Survey of Programming Languages

C Lecture 5 – Bitwise Operations and Operations Miscellany

Logical vs. Bitwise Operations

• Logical operations assume that the entire variable represents either true or false.
  – Combining two integer values using a logical operator produces one result, with the variable representing true or false.
• Bitwise operations assume that each bit in the variable’s value represents a separate true or false.
  – Combining two integer values using a bitwise operators produces 8 (or 16 or 32 or 64) separate bits each representing a true or a false.
Logical Values in C

• Although the 1999 standard for C includes booleans, it does not exist in older versions.
• Integers are usually used for boolean values, with nonzero values being accepted as true and zero values being accepted as false.
• Boolean operations will produce a 1 for true and a 0 for false.

&& - Logical AND

• The logical AND operator is && and produces a 1 if both operands are nonzero; otherwise, it produces a 0.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x &amp;&amp; y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
&& – An Example

```c
#include <stdio.h>
int main(void)
{
    unsigned u, v, w, x = 0x10,
             y = 0x110, z = 0x0;

    u = x && y;
    v = x && z;
    w = y && z;
    printf("u = %x\t\tv = %x\t\tw = %x\n", u, v, z);
    return(0);
}
```

Output
u = 1   v = 0   w = 0

|| - Logical OR

- The logical OR operator is `||` and produces a 1 if either operands is nonzero; otherwise, it produces a 0.

| x | y | x || y |
|---|---|------|
| 0 | 0 |  0  |
| 0 | 1 |  1  |
| 1 | 0 |  1  |
| 1 | 1 |  1  |
Logical | | – An Example

```c
#include <stdio.h>
int main(void)
{
    unsigned u, v, w, x = 0x10,
             y = 0x110, z = 0x0;

    u = x || y;
    v = x || z;
    w = y || z;
    printf("u = %x\tv = %x\tw = %x\n", u, v, z);
    return(0);
}
```

Output
- u = 1
- v = 1
- w = 1

Logical NOT

- The logical NOT operator ! Inverts the value; nonzero becomes 0 and 0 becomes 1.

<table>
<thead>
<tr>
<th>x</th>
<th>!x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logical NOT – An Example

```c
#include <stdio.h>
int main(void)
{
    unsigned x = 0x110, y;
    y = !x;
    printf("x = %x\ty = %x\n", x, y);
    x = 0x0;
    y = !x;
    printf("x = %x\ty = %x\n", x, y);
    return(0);
}
```

Output

```
x = 110  y = 0
x = 0    y = 1
```

Bitwise Operations

- Bitwise operations treat the operands as 8-bit (or 16- or 32-bit) operands. Performing a bitwise AND operation on two 8-bit integers means that 8 ANDs are performed on corresponding bits.

- Example:
  
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Bitwise AND

- A bitwise AND operation is actually 8 (or 16 or 32) AND operations.
- An example of ANDing:

\[
\begin{array}{c}
\text{cleared} & 0011011 \\
\text{unchanged} & 00001111
\end{array}
\]

- The AND instruction can be used to clear selected bits in an operand while preserving the remaining bits. This is called masking.

Bitwise AND – An Example

```c
unsigned u, v, w, x = 0xab87, 
y = 0x4633, z = 0x1111;

u = x & y;
v = x & z;
w = y & z;

printf("u = %x\tv = %x\tw = %x\n", u, v, w);
```

Output

\[ u = 203 \; \; v = 101 \; \; w = 11 \]
Bitwise OR

- A bitwise OR operation is actually 8 (or 16 or 32) OR operations.

- An example of ORing:

  unchanged 00111011

  \[\begin{array}{c}
  \text{set} \\
  00001111 \\
  00111111 \\
  \end{array}\]

- The OR instruction can be used to set selected bits in an operand while preserving the remaining bits.

Bitwise OR – An Example

```c
unsigned u, v, w, x = 0xab87, y = 0x4633, z = 0x1111;

u = x | y;
v = x | z;
w = y | z;

printf("u = %x\tv = %x\tw = %x\n", u, v, w);
```

Output

```
u = efb7   v = bb97   w = 5733
```
Bitwise **NOT** (1s Complement)

- The bitwise NOT (better known as the *1s complement*) inverts each bit in the operand.
- Example
  ```c
  unsigned x, y = 0xab87;
  x = ~y;
  printf("x = %x\ty = %x\n", x, y);
  ```

  Output
  ```plaintext
  x = ffff5478    y = ab87
  ```

Bitwise **XOR**

- A bitwise **XOR** operation is actually 8 (or 16 or 32) **AND** operations.
- An example of **XORing**:
  ```plaintext
  \[\begin{array}{c}
  \text{unchanged} \quad 00111011 \\
  \text{inverted} \quad 00111111 \\
  \text{unchanged} \quad 00000100 \\
  \end{array}\]
  ```

- The **XOR** instruction can be used to reverse selected bits in an operand while preserving the remaining bits.
Bitwise **XOR** – An Example

```c
unsigned u, v, w, x = 0xab87,
y = 0x4633, z = 0x1111;

u = x ^ y;
v = x ^ z;
w = y ^ z;

printf("u = %x \tv = %x \tw = %x\n", u, v, w);
```

Output

\begin{verbatim}
u = edb4       v = ba96       w = 5722
\end{verbatim}

Bit Shifting

- **>>** Right shifting    **<<** Left Shifting
- \( x = 0x00ff; \)
- \( y = x << 8; /* y is 0x ff00 */ \)
- Results may vary depending on the computer – int can be different sizes on different computers.
- \( x & \sim 077 \) will turn off lowest six bits.
getbits()

/*
 * getbits() - Get n bits from position p
 */
unsigned getbits(unsigned x, int p, int n)
{
    return ((x >> (p + 1 - n)) & ~(~0 << n));
}

Assignment Operators

• An assignment operator is just another operator in C.
• We can rewrite
  \[ i = i + 2; \] as \[ i += 2; \]
or
  \[ i = i + x \ast y; \] as \[ i += x \ast y; \]
• Similarly, there are -=, *=, /=, etc.
Assignment Operators

• Caution!
  \[
i *= 2 + y;
\]
is rewritten as
  \[
i = i * (2+y);
\]
NOT
  \[
i = (i *2) + y;
\]
• This is really useful with a statement like
  \[
  yyval[yypv[p3+p4] + yypv[p1+p2]] += 2;
  \]

bitcount()

/*
 * bitcount() - Count 1s in x
 */
int     bitcount(unsigned x)
{
    int     b;
    for (b = 0;  x != 0;  x >>= 1)
        if (x & 01)
            b++;
    return(b);
}
Conditional Expressions

- Why write
  
  ```
  if (a > b)
      z = a;
  else
      a = b;
  ```
  
  when you can write
  
  ```
  z = (a > b)? a : b;
  ```

- The general form is
  
  ```
  expression1? expression2: expression3;
  ```

  when `expression1` is nonzero, `expression2` is evaluated. Otherwise `expression3` is evaluated.

- The usual rules of conversion are in effect.
  
  ```
  int   i, j, a, b;
  float x;
  ...
  i = (a > b)? j: x;
  ```
Conditional Expressions

- If this useful? YES!!
  
  \[ z = (a > b) ? a : b; /* z = \max(a, b); */ \]
  
  \[ x = (x > 0) ? x : -x; /* x = \abs(x); */ \]

  /* Print 5 values to a line */
  for (i = 0; i < MAXSIZE; i++)
    printf("%d", x[i], i % 5 == 4? \n: ' ');

Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
</tr>
<tr>
<td>[]</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ --</td>
<td>unary type</td>
</tr>
<tr>
<td>* (ptr) &amp; (address) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>/</td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; =</td>
<td>left to right</td>
</tr>
<tr>
<td>== !</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp; (bitwise AND)</td>
<td>left to right</td>
</tr>
<tr>
<td>^ (bitwise XOR)</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td>(bitwise OR)</td>
</tr>
</tbody>
</table>