**Oriette Ramos**

**Lesson Plan 21**

**ISTA-220**

1. Give five examples (using valid C# code) of the five bitwise operators listed in the text.
   * **The not operator is the tilde and changes all the 1s to 0s and vice versa in the 32 bits that represent the int: int x = ~150; declares an integer type variable x and assigns it the value -151.**
   * **The left shift operator is two consecutive less than operators (<<) that takes an integer argument and shits the bits that many places to the left. The leftmost bits are discarded and zeros are added to the right: int x = 150 << 3; declares an integer type variable x and assigns it the value 1200.**
   * **The or operator is a single pipe (the vertical line) and performs a bitwise or operation, returning a value containing a 1 in each position in which either of the operands has a 1: int x = 150 | 15; declares an integer type variable x and assigns it the value 159.**
   * **The and operator is an ampersand that performs a bitwise and operation. And is similar to the bitwise or operator, but it returns a value containing a 1 in each position where both of the operands have a 1: int x = 150 & 15; declares an integer type variable x and assigns it the value 6.**
   * **The xor operator is a carat and performs a bitwise exclusive or operation, returning a 1 in each bit where there is a 1 in one operand or the other but not both: int x = 150 ^ 15; declares an integer type variable x and assigns it the value 153.**
2. Does C# implement the right-shift (>>) operator? If so, give an example of its operation using valid C# code.  
   **The right shift operator is two consecutive greater than operators (>>) that takes an integer argument and shits the bits that many places to the right. The rightmost bits are discarded and zeros are added to the left: int x = 150 >> 3; declares an integer type variable x and assigns it the value 18.**
3. Explain in detail this code: bits & (1 << index);.  
   **First I'll start with the part with the highest precedence; (1 << index) will return a value whos binary equivalent is a 1 that has index zeros to the right. 1 is binary 00000001 so (1 << 3) is 00001000 which happens to be decimal 8. The and operator is comparing the binary value of the variable bits to the binary value of (1 << index) and looking for bits in the same location. So if bits contained the value 40 (00101000) and index contained the value of 3 then 00101000 and 00001000 both have 1s in the 4th position and the statement returns 00001000 which is decimal 8.**
4. Explain in detail this code: bits |= (1 << index);.  
   **The code in parenthesis works exactly as explained above. The compound assignment operator with the bitwise or compares the value in the binary value of the variable bits to the binary value of (1 << index) and returns a value that has 1s in all the binary positions in either number. So if bits contained the value 8 (00001000) and index contained the value of 5, then 00001000 or 00100000 has 1s in the 4th and 6th positions and is 00101000 which is decimal 40. The because it's a compound assignment operator, bits is assigned the value 40.**
5. Explain in detail this code: bits &= (1 << index);.  
   **This code is nearly identical to the code described in question 3 with the only difference being the compound assignment operator. So if bits was 40, and index was 3, bits would be assigned the value of 8.**
6. How does C# interpret this: bool peek = bits[n];?  
   **This statement is converted to a call to the get accessor for bits, and the index argument is set to the value of n. In essence, you are declaring a bool type variable peek and assigning it the value returned from the get accessor based on the value of n; so if n was 3, it would pass 3 to the get accessor.**
7. How does C# interpret this: bits[n] = true;?  
   **This statement is converted to a call to the set accessor for bits where the index is the value of n and sets the value to true (1).**
8. How does C# interpret this: bits[n] ^= true;?  
   **This statement is converted to a call to both the get and set accessor for bits where the index is the value of n. It basically inverts the bit value at the nth position in bits.**
9. Assume that users were assigned read, write, and execute permissions according to this scheme: read = 1, write = 2, execute = 4. How would you interpret the following user permissions:  
   (a) permission = 0  
   **No permissions.**  
   (b) permission = 1  
   **Read.**  
   (c) permission = 2  
   **Write.**  
   (d) permission = 3  
   **Read and Write.**  
   (e) permission = 4  
   **Execute.**  
   (f) permission = 5  
   **Read and Execute.**  
   (g) permission = 6  
   **Write and Execute.** (h) permission = 7  
   **Read, Write, and Execute.**
10. Answer the previous question by converting the decimal permissions into binary permissions. What does this tell you about using this scheme of permissions?  
    **This scheme of permissions was written with integer indexing as an array of Boolean values.**  
    (a) permission = 0  
    **No permissions/000.**  
    (b) permission = 1  
    **Read/001.**  
    (c) permission = 2  
    **Write/010.**  
    (d) permission = 3  
    **Read and Write/011.**  
    (e) permission = 4  
    **Execute/100.**  
    (f) permission = 5  
    **Read and Execute/101.**  
    (g) permission = 6  
    **Write and Execute/110.**  
    (h) permission = 7  
    **Read, Write, and Execute/111.**