ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 10

Bitwise Operations

Dayu Wang

- **Bitwise Operations**

  → Bitwise operations are a set of operations that work on the individual bits of binary numbers.

  - They are fundamental operations in computer systems and digital electronics, as they directly manipulate the 1s and 0s that represent data in binary form.
  - Bitwise operations are usually performed on fixed-width binary representations of integers, such as 8-bit, 16-bit, 32-bit, or 64-bit numbers.

  → In this class, we will discuss five bitwise operations:

  1) Bitwise AND
  2) Bitwise OR
  3) Bitwise XOR (exclusive OR)
  4) Bitwise LEFT SHIFT
  5) Bitwise RIGHT SHIFT

- **Bitwise AND**

  → This operation takes two binary numbers and performs a logical AND operation on each pair of corresponding bits.

  The result is a binary number with 1s only where both input bits are 1, and 0s elsewhere.

  - If the first and second bits are both 1, then the AND result is 1.
  - If either bit is 0, then the AND result is 0.

  → In C++, '&' is the bitwise AND operator.

  [Example] (6 & 5) = 4.

  → Properties of bitwise AND

  - If n is an integer, expression n & 1 extracts the rightmost bit of n.

    [Example] Given a non-negative integer n, write a function to determine whether n is odd or even.

```
1  /** Tests whether a non-negative integer is an odd number.
2      @param n: a non-negative integer to test
3      @return: {true} if n is an odd number; {false} if it is even.
4  */
5  bool is_odd(unsigned int n) { return n & 1; }
```

  - If n is an integer, statement n & 0 turns off the rightmost bit of n.

  - If n is an integer, statement n & (n – 1) turns off the rightmost on bit of n.

    [Example] Given a positive integer n, write a function to determine whether n is power of 2 in $O(1)$ time.

```
1  /** Tests whether a positive integer is power of two.
2      @param n: a positive integer to test
3      @return: {true} if n is power of two; {false} otherwise
4  */
5  bool is_power_of_two(unsigned int n) { return !(n & n - 1); }
```

- **Bitwise OR**

  → This operation takes two binary numbers and performs a logical OR operation on each pair of corresponding bits.

  The result is a binary number with 1s where either input bit is 1, and 0s elsewhere.

  - The first and second bits are both 0, then the OR result is 0.
  - If either bit is 1, then the OR result is 1.

  → In C++, '|' is the bitwise OR operator.

  [Example] (6 | 5) = 7.

→ **Properties of bitwise OR**

- If `n` is an integer, statement `n | 1` turns on the rightmost bit of `n`.

• **Bitwise XOR**

→ **This operation compares corresponding bits in the two input numbers and outputs a 1 if the bits are different, and a 0 if they are the same.**

- **If the first and second bits are the same (both 0 or both 1), then the XOR result is 0.**
- **If they are different, then the XOR result is 1.**

→ **In C++, `'^'` is the bitwise XOR operator.**

[Example] (6 ^ 4) = 2.

→ **[Important] Properties of bitwise XOR**

- **If `n` is an integer, then `(0 ^ n) == n` is always true.**
- **If `a` and `b` are integers, then `(a ^ b ^ b) == a` is always true.**

[Example] Swap two integers without using ~~temporary variables~~.

```
1  void swap_integers(int& a, int& b) {
2      a ^= b;
3      b ^= a;
4      a ^= b;
5  }
```


• **Bitwise LEFT SHIFT**

→ **This operation shifts the bits of the first operand with respect to the second operand which decides the number of places to shift.**

In C++, "<<" is the bitwise LEFT SHIFT operator.

- **The syntax for bitwise LEFT SHIFT between `x` and `y` operands is `x << y`, the value of `x` is left shifted by `y` number of bits.**
- **[Example] (5 << 2) = 20.**

→ **Properties of bitwise LEFT SHIFT**

- **If integer overflow issue is ignored, then if `n`, `k` are integers, then `(n << k) == n * 2^k` is always true.**
- **Shift is not ~~rotation~~; any bit moved outside the range will be truncated.**

[Example] `n` is an unsigned short integer (16-bit).

	n	n << 1
Decimal	**32,771**	**6**
Binary	**1000 0000 0000 0011**	**0000 0000 0000 0110**

- **Bitwise RIGHT SHIFT**

  → This operation shifts the bits of the first operand with respect to the second operand which decides the number of places to shift.

  In C++, ">>" is the bitwise RIGHT SHIFT operator.

  - The syntax for bitwise RIGHT SHIFT between x and y operands is `x >> y`, the value of x is right shifted by y number of bits.
  - [Example] (10 >> 2) = 2.

  → Properties of bitwise RIGHT SHIFT

  - If n, k are integers, then `(n >> k) == n / 2^k` is always true.
  - If n is an integer, the expression `n >> k & 1` returns the k-th bit of n.

  → Precedence of bitwise operators (from highest to lowest):

  "<<" > ">>" > '&' > '^' > '|'

  Use parentheses if different operators are mixed.

- **Advantages of bitwise operations**

  - Any bit is independent to other bits.
  - Bitwise operations can be done in parallel for all the bits.
  - Bitwise operations are very fast.

- **Single Number I**

  → (Single Number I) Given a non-empty vector of integers, every element appears twice except for one. Find that single one.

```
1  int single_number_I(const vector<int>& vec) {
2      int result = 0;
3      for (int num : vec) { result ^= num; }
4      return result;
5  }
```

  → Not required in this class:

  - (Single Number II) Given a non-empty vector of integers, every element appears three times except for one, which appears exactly once. Find that single one.
  - (Single Number III) Given an vector of at least two integers, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.