**BIT MANIPULATION**

1. **INTRODUCTION**

**Bitwise Operators** are special operators that operate directly on the binary representation of numbers. There are 6 primary Bitwise operators in C++. Most of the Bitwise operators are binary in nature, i.e., they operate on two operands. The only exception is Bitwise NOT operator which is unary in nature, i.e., it operates on a single operand. These operators are as follows –

* **Bitwise AND (&)**
* **Bitwise OR (|)**
* **Bitwise XOR (^)**
* **Bitwise NOT (~)**
* **Left-shift (<<)**
* **Right-shift (>>)**

The Bitwise Algorithms are used to perform operations at bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are sometimes used to improve the efficiency of a program. The time & space complexity of Bitwise operators is O(1).

A bitwise summary of the first three operators is shown in the table below –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **A & B** | **A | B** | **A ^ B** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Bitwise AND:** It performs a logical AND operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

**Bitwise OR:** It performs a logical OR operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

**Bitwise XOR:** It performs a logical XOR operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

The other three operators depend highly on how the numbers are represented in their binary form in the system. Their behavior is undefined in many cases like negative numbers, floating-point numbers, etc. The Left-shift and Right-shift operators cannot be used on negative numbers.

**Bitwise NOT Operator:** It toggles all the bits of the binary representation of a number, i.e., all 1-bits are changed to 0-bits and vice-versa. It works perfectly for unsigned integers as there is no sign issue.

**Left-shift Operator:** Let the number **A** be represented as a **n**-bit number, then **A << x** (where **x** represents the number of left shifts to be performed) shifts the **(n - x)** trailing bits to the left and adds **x** 0 bits at the end.

**Right-shift Operator:** Let the number **A** be represented as a **n**-bit number, then **A >> x** (where **x** represents the number of right shifts to be performed) shifts the **(n - x)** leading bits to the right and appends **x** 0 bits at the start ignoring the last **x** bits.

Let the number A = 45 (101101) and B = 89 (1011001), then –  
A & B = 9 (1001)  
A | B = 125 (1111101)  
A ^ B = 116 (1110100)  
A << 3 = 360 (101101000)  
B >> 2 = 22 (10110)  
~A = 4294967251 (11111111111111111111111111010010)

The Left-shift and Right-shift operators follow a very simple rule where,   
A << x = A \* 2xA >> x = ⌊ A / 2x ⌋

1. **Placeholder**