Bitwise Operators in C - GeeksforGeeks

Source: https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

Courses Tutorials Practice Jobs C Tutorial Interview Questions Examples Quizzes Projects Cheatsheet File Handling Multithreading Memory Layout DSA in C C++ Technical Scripter 2026 Explore C Basics C Language Introduction Identifiers in C Keywords in C Variables in C Data Types in C Operators in C Decision Making in C (if , if..else, Nested if, if-else-if) Loops in C Functions in C Arrays & Strings Arrays in C Strings in C Pointers and Structures Pointers in C Function Pointer in C Unions in C Enumeration (or enum) in C Structure Member Alignment, Padding and Data Packing Memory Management Memory Layout of C Programs Dynamic Memory Allocation in C What is Memory Leak? How can we avoid? File & Error Handling File Handling in C Read/Write Structure From/to a File in C Error Handling in C Using goto for Exception Handling in C Error Handling During File Operations in C Advanced Concepts Variadic Functions in C Signals in C language Socket Programming in C _Generics Keyword in C Multithreading in C Three 90 Challenge 90% Refund Bitwise Operators in C Last Updated : 13 Dec, 2025 In C, bitwise operators are used to perform operations directly on the binary representations of numbers. These operators work by manipulating individual bits (0s and 1s) in a number. The following 6 operators are bitwise operators (also known as bit operators as they work at the bit-level). They are used to perform bitwise operations in C. The & (bitwise AND) in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. The | (bitwise OR) in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. The ^ (bitwise XOR) in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. The << (left shift) in C takes two numbers, the left shifts the bits of the first operand, and the second operand decides the number of places to shift. The >> (right shift) in C takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift. The ~ (bitwise NOT) in C takes one number and inverts all bits of it. Let's look at the truth table of the bitwise operators. X Y X & Y X | Y X ^ Y 0 0 0 0 0 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 C #include <stdio.h> int main () { // a = 5 (00000101 in 8-bit binary) // b = 9 (00001001 in 8-bit binary) unsigned int a = 5 , b = 9 ; // The result is 00000001 printf ("a&b = %u \n " , a & b); // The result is 00001101 printf ("a|b = %u \n " , a | b); // The result is 00001100 printf ("a^b = %u \n " , a ^ b); // The result is 11111111111111111111111111111010 (assuming 32-bit unsigned int) printf ("~a = %u \n " , a = ~ a); // The result is 00010010 printf ("b<<1 = %u \n " , b << 1); // The result is 00000100 printf ("b>>1 = %u \n " , b >> 1); return 0 ; } Output a&b = 1 a|b = 13 a^b = 12 ~a = 4294967290 b<<1 = 18 b>>1 = 4 Interesting Facts About Bitwise Operators Shift Operators : Left-shift (<<) and right-shift (>>) should not be used with negative numbers. Shifting by a negative number or more than the size of the integer leads to undefined behavior. No shift occurs if the number of shifts is 0. Bitwise OR (|) : The OR of two numbers is like adding them if no carry occurs. If there is a carry, the sum is calculated as a | b + a & b. Bitwise XOR (^) : Very useful in programming problems. For example, finding the odd occurring number in a set where all other numbers occur even times can be done efficiently using XOR. C #include <stdio.h> int main (void) { int arr [] = { 12 , 12 , 14 , 90 , 14 , 14 , 14 }; int n = sizeof (arr) / sizeof (arr [0]); int res = 0 , i ; for (int i = 0 ; i < n ; i ++) res ^= arr [i]; printf ("%d" , res); return 0 ; } Output 90 Logical vs Bitwise : Bitwise operators should not replace logical operators. Logical operators (&&, ||, !) return 0 or 1, while bitwise operators return an integer value. C #include <stdio.h> int main () { int x = 2 , y = 5 ; (x & y) ? printf ("True ") : printf ("False "); (x && y) ? printf ("True ") : printf ("False "); return 0 ; } Output False True Shift and Arithmetic : Left-shift (<<) is equivalent to multiplication by 2, and right-shift (>>) is equivalent to division by 2 for positive numbers. C #include <stdio.h> int main () { int x = 19 ; printf ("x << 1 = %d \n " , x << 1); printf ("x >> 1 = %d" , x >> 1); return 0 ; } Output x << 1 = 38 x >> 1 = 9 Check Odd/Even : The AND operator (&) can quickly check if a number is odd or even. (x & 1) is non-zero if x is odd, and 0 if even. C #include <stdio.h> int main () { int x = 19 ; (x & 1) ? printf ("Odd") : printf ("Even"); return 0 ; } Output Odd Time Complexity: O(1) Auxiliary Space: O(1) Bitwise NOT (~) : Should be used carefully. Applying ~ can produce large numbers for unsigned variables or negative numbers for signed variables due to 2's complement representation. C #include <stdio.h> int main () { unsigned int x = 1 ; printf ("Signed Result %d \n " , ~ x); printf ("Unsigned Result %u" , ~ x); return 0 ; } Output Signed Result -2 Unsigned Result 4294967294 Note The output of the above program is compiler dependent . Comment Article

Tags: Article Tags: C Language Bitwise-XOR cpp-operator C-Operators