Basic_Bitmasking.cpp

```cpp
/**
 *    Author: devesh95
 *
 *    Topic: Basic Bitmasking Examples
 *
 *    Description:
 *    This file provides 10 simple examples to illustrate the concept of bitmasking.
 *    Each example is accompanied by very detailed explanations (in comments) so that
 *    you can understand every basic bit-level operation and theory behind it.
 *
 *    How to compile:
 *         g++ -std=c++17 -O2 -Wall Basic_Bitmasking_Examples.cpp -o bitmask_basics
 *
 *    How to run:
 *         ./bitmask_basics
 *
 *    Topics covered:
 *       1. Basic Bitmask Operations (set, clear, toggle, check)
 *       2. Bit Shifting (left shift, right shift)
 *       3. Counting Set Bits (using a loop and built-in function)
 *       4. Checking if a Number is a Power of Two
 *       5. Isolating the Lowest Set Bit
 *       6. Clearing the Lowest Set Bit
 *       7. Enumerating All Subsets of a Set (using bitmask)
 *       8. Representing a Set using Bitmask (union, intersection)
 *       9. Inverting a Bitmask (bitwise NOT)
 *      10. Iterating Over All Set Bits in a Bitmask
 */

#include <bits/stdc++.h>
using namespace std;

#define int long long

// ---------------------------------------------------------------------
// Example 1: Basic Bitmask Operations (set, clear, toggle, check)
// ---------------------------------------------------------------------
/*
   Theory:
   - A bitmask is simply an integer where each bit represents a binary flag.
   - For example, if we want to represent a set with 8 elements, we can use an 8-bit number.
   - Each bit in the bitmask is either 0 (flag not set) or 1 (flag set).

   Basic Operations:
   1. Setting a bit:
        To set the bit at position 'pos', we use:
          mask |= (1 << pos);
        This uses a left shift to create a number with only the 'pos'-th bit as 1 and then
        applies bitwise OR (|) to set that bit in 'mask'.

   2. Clearing a bit:
```

```
        To clear the bit at position 'pos', we use:
          mask &= ~(1 << pos);
        Here, (1 << pos) creates a mask with bit 'pos' set. The '~' (NOT) operator flips all bits,
        so ~(1 << pos) has a 0 at position 'pos' and 1s elsewhere. Using bitwise AND (&) with this
        value clears that specific bit in 'mask'.

   3. Toggling a bit:
        To flip (toggle) the bit at position 'pos', use:
          mask ^= (1 << pos);
        The XOR (^) operator flips the bit if the corresponding bit in (1 << pos) is 1.

   4. Checking a bit:
        To check if the bit at position 'pos' is set, we use:
          if (mask & (1 << pos)) { ... }
        If the result is non-zero, then the bit at 'pos' is set.

   Detailed Example:
     We will start with an empty mask and then perform each operation step by step.
*/
void example_basic_operations() {
    cout << "\n----- Example 1: Basic Bitmask Operations -----\n";
    int mask = 0; // Initially, all bits are 0.
    cout << "Initial mask: " << bitset<8>(mask) << " (binary representation, 8 bits)" << "\n";

    // Setting bits:
    // Set bit at position 2 (positions are 0-indexed from the right)
    // (1 << 2) is 00000100 in binary.
    mask |= (1 << 2);
    cout << "After setting bit 2: " << bitset<8>(mask) << " -> Bit 2 is now 1" << "\n";

    // Set bit at position 5
    // (1 << 5) is 00100000 in binary.
    mask |= (1 << 5);
    cout << "After setting bit 5: " << bitset<8>(mask) << " -> Bits at positions 2 and 5 are set" << "\n";

    // Checking a bit:
    // Check if bit at position 2 is set.
    if(mask & (1 << 2))
        cout << "Bit 2 is confirmed to be set.\n";
    else
        cout << "Bit 2 is not set.\n";

    // Toggling a bit:
    // Toggle bit at position 2: if it is 1, it becomes 0; if 0, it becomes 1.
    mask ^= (1 << 2);
    cout << "After toggling bit 2: " << bitset<8>(mask) << " -> Bit 2 has been flipped.\n";

    // Clearing a bit:
    // Clear bit at position 5, i.e., set it to 0.
    mask &= ~(1 << 5);
```

```
    cout << "After clearing bit 5: " << bitset<8>(mask) << " -> Bit 5 is now 0.\n";

    // Summary:
    // We have seen how to set, clear, toggle, and check bits in a bitmask.
}

// ------------------------------------------------------------------------
// Example 2: Bit Shifting (Left Shift and Right Shift)
// ------------------------------------------------------------------------
/*
   Theory:
   - Bit shifting moves the bits of a number to the left or right.
   - Left Shift (<<):
        Shifting a number left by 1 bit is equivalent to multiplying it by 2.
        Example: 5 (00000101 in binary) << 1 becomes 10 (00001010 in binary).
   - Right Shift (>>):
        Shifting right divides by 2 (ignoring remainders for integers).
        Example: 5 (00000101) >> 1 becomes 2 (00000010).

   Detailed Example:
     We start with a number and perform left and right shifts, explaining the binary representation.
*/
void example_bit_shifting() {
    cout << "\n----- Example 2: Bit Shifting -----\n";
    int num = 5; // 5 in binary is 00000101.
    cout << "Original number: " << num << " (" << bitset<8>(num) << " in binary)\n";

    // Left shift by 1: Multiply by 2.
    int leftShift = num << 1; // 00000101 becomes 00001010 (which is 10)
    cout << "After left shift by 1: " << leftShift << " (" << bitset<8>(leftShift) << " in binary)\n";

    // Right shift by 1: Divide by 2.
    int rightShift = num >> 1; // 00000101 becomes 00000010 (which is 2)
    cout << "After right shift by 1: " << rightShift << " (" << bitset<8>(rightShift) << " in binary)\n";
}

// ------------------------------------------------------------------------
// Example 3: Counting Set Bits in an Integer
// ------------------------------------------------------------------------
/*
   Theory:
   - Counting the number of 1s (set bits) in an integer is a common bitmasking task.
   - One method is to iterate over all bits, check each bit, and increment a counter.
   - Alternatively, C++ provides built-in functions like __builtin_popcountll() that count the set bits.

   Explanation using a loop (manual method):
     Initialize a counter to 0.
     While the number is non-zero, add (n & 1) to the counter.
     Right shift the number by 1 until it becomes 0.

```

```
   Pseudocode:
     count = 0;
     while(n > 0) {
         count += n & 1;
         n = n >> 1;
     }
*/
void example_count_set_bits() {
    cout << "\n----- Example 3: Counting Set Bits -----\n";
    int num;
    cout << "Enter an integer: ";
    cin >> num;

    // Approach 1: Manual counting using a loop.
    int count = 0;
    int temp = num;
    while(temp > 0) {
        count += (temp & 1); // Adds 1 if the least significant bit is 1.
        temp = temp >> 1;    // Shift right by 1 bit.
    }
    cout << "Number of set bits (manual count): " << count << "\n";

    // Approach 2: Using the built-in function.
    int builtInCount = __builtin_popcountll(num);
    cout << "Number of set bits (__builtin_popcountll): " << builtInCount << "\n";
}

// ------------------------------------------------------------------------
// Example 4: Checking if a Number is a Power of Two
// ------------------------------------------------------------------------
/*
   Theory:
   - A number is a power of two if it has exactly one bit set in its binary representation.
   - Trick: For any number n, if n is a power of two, then n & (n - 1) equals 0.
   - This works because subtracting 1 from n flips all bits after the rightmost set bit.

   Detailed Explanation:
     If n = 8 (binary 1000), then n - 1 = 7 (binary 0111).
     n & (n - 1) = 1000 & 0111 = 0000.

   Equation:
     isPowerOfTwo(n) = (n != 0) && ((n & (n - 1)) == 0)
*/
void example_power_of_two() {
    cout << "\n----- Example 4: Checking if a Number is a Power of Two -----\n";
    int n;
    cout << "Enter an integer: ";
    cin >> n;

    if(n != 0 && ((n & (n - 1)) == 0))
        cout << n << " is a power of two.\n";
    else
        cout << n << " is NOT a power of two.\n";
}
```

```cpp

// ------------------------------------------------------------------------
// Example 5: Isolating the Lowest Set Bit
// ------------------------------------------------------------------------
/*
   Theory:
   - The lowest (rightmost) set bit of an integer is the smallest power of two that divides it.
   - To isolate the lowest set bit, we can use:
         lowest = n & (-n);
   - Here, -n is the two's complement of n. Two's complement inverts the bits of n (after the rightmost set bit),
     so the AND operation leaves only the lowest set bit.

   Equation:
     lowestSetBit(n) = n & (-n)

   Detailed Example:
     Let n = 10, which in binary is 1010.
     -n (two's complement) is 0110 (if we consider a fixed number of bits; the exact binary depends on bit width).
     n & (-n) isolates 0010, which is 2, the lowest set bit.
*/
void example_lowest_set_bit() {
    cout << "\n----- Example 5: Isolating the Lowest Set Bit -----\n";
    int n;
    cout << "Enter an integer: ";
    cin >> n;
    int lowest = n & (-n);
    cout << "Lowest set bit of " << n << " is " << lowest << " (binary: " << bitset<8>(lowest) << ")\n";
}

// ------------------------------------------------------------------------
// Example 6: Clearing the Lowest Set Bit
// ------------------------------------------------------------------------
/*
   Theory:
   - To remove (or clear) the lowest set bit of an integer, we use:
         n = n & (n - 1);
   - This works because n - 1 flips all bits up to and including the lowest set bit.
   - When you AND n with n - 1, the lowest set bit is turned off.

   Equation:
     cleared = n & (n - 1)

   Detailed Explanation:
     For example, let n = 12 (binary 1100).
     n - 1 = 11 (binary 1011).
     n & (n - 1) = 1100 & 1011 = 1000 (binary), which equals 8.
*/
void example_clear_lowest_set_bit() {
    cout << "\n----- Example 6: Clearing the Lowest Set Bit -----\n";
    int n;
```

```
    cout << "Enter an integer: ";
    cin >> n;
    int cleared = n & (n - 1);
    cout << "After clearing the lowest set bit, the value is " << cleared << " (binary: " << bitset<8>(cleared) << ")\n";
}

// ------------------------------------------------------------------------
// Example 7: Enumerating All Subsets of a Set
// ------------------------------------------------------------------------
/*
   Theory:
   - A set with n elements has 2^n possible subsets.
   - Each subset can be represented by a bitmask of length n.
   - If the i-th bit in the mask is 1, then the i-th element is included in the subset.

   Explanation:
     For each mask from 0 to 2^n - 1:
       - The bitmask itself (in binary) represents a subset.
       - We can iterate through each bit to decide which elements are present.

   Equation/Process:
     For each mask, for each i in [0, n-1]:
         if (mask & (1 << i)) is true, then include element i in the subset.
*/
void example_enumerate_subsets() {
    cout << "\n----- Example 7: Enumerating All Subsets -----\n";
    int n;
    cout << "Enter n (number of elements): ";
    cin >> n;
    cout << "All subsets (each represented as a bitmask):\n";
    int total = 1 << n;
    for (int mask = 0; mask < total; mask++) {
        // Display the bitmask in binary (show only n bits)
        string binaryMask = bitset<8>(mask).to_string().substr(8 - n);
        cout << "Mask " << binaryMask << " represents subset: { ";
        for (int i = 0; i < n; i++) {
            if(mask & (1 << i))
                cout << i << " ";
        }
        cout << "}\n";
    }
}

// ------------------------------------------------------------------------
// Example 8: Representing a Set using Bitmask (Union and Intersection)
// ------------------------------------------------------------------------
/*
   Theory:
   - We can represent a set of elements (e.g., numbers 0 to n-1) as a bitmask.
   - Union of two sets (A and B) is performed using bitwise OR (|).
   - Intersection of two sets is performed using bitwise AND (&).

   Detailed Example:
```

```
     Let set A = {0, 2} and set B = {1, 2} for n = 3 elements.
       Representations:
         A is 101 in binary (bits at positions 0 and 2 are 1).
         B is 110 in binary (bits at positions 1 and 2 are 1).
       Then:
         Union = 101 | 110 = 111 (binary), representing {0, 1, 2}.
         Intersection = 101 & 110 = 100 (binary), representing {2}.
*/
void example_set_operations() {
    cout << "\n----- Example 8: Set Operations using Bitmask -----\n";
    // For n = 3, define two sets:
    int maskA = (1 << 0) | (1 << 2);  // Represents set A = {0, 2} (binary 101)
    int maskB = (1 << 1) | (1 << 2);  // Represents set B = {1, 2} (binary 110)
    cout << "Set A (bitmask): " << bitset<3>(maskA) << " -> represents {0,2}\n";
    cout << "Set B (bitmask): " << bitset<3>(maskB) << " -> represents {1,2}\n";

    int unionMask = maskA | maskB;      // Union: bitwise OR
    int intersectMask = maskA & maskB;    // Intersection: bitwise AND

    cout << "Union (A U B): " << bitset<3>(unionMask) << " -> represents {0,1,2}\n";
    cout << "Intersection (A ∩ B): " << bitset<3>(intersectMask) << " -> represents {2}\n";
}

// ---------------------------------------------------------------------
// Example 9: Inverting a Bitmask (Bitwise NOT)
// ---------------------------------------------------------------------
/*
   Theory:
   - The bitwise NOT operator (~) inverts every bit in an integer (0 becomes 1 and 1 becomes
0).
   - However, when dealing with fixed-length bitmasks, you need to ensure that only the
desired number
     of bits is considered.

   Explanation:
     If you have a 4-bit bitmask and you want to invert it, you use:
         inverted = ~mask & ((1 << numBits) - 1)
     This masks out only the lowest numBits bits after inverting.

   Detailed Example:
     Let mask = 0101 (for 4 bits). Then ~mask gives a number with many bits set,
     so we use & ((1 << 4) - 1) = & (16 - 1) = & 15 (which is 1111 in binary).
     Thus, inverted = ~0101 & 1111 = 1010.
*/
void example_invert_bitmask() {
    cout << "\n----- Example 9: Inverting a Bitmask -----\n";
    int numBits = 4;
    int mask = 0b0101; // 4-bit representation: 0101
    cout << "Original mask (4 bits): " << bitset<4>(mask) << "\n";
    int inverted = ~mask & ((1 << numBits) - 1); // Invert only 4 bits.
    cout << "Inverted mask: " << bitset<4>(inverted) << " -> Inversion of 0101 is 1010\n";
}

// ---------------------------------------------------------------------
```

```
// Example 10: Iterating Over All Set Bits in a Bitmask
// ------------------------------------------------------------------------
/*
   Theory:
   - Often, you want to perform some operation on each element represented by a set bit in a bitmask.
   - Two common methods:
     Method 1: Check each bit from 0 to n-1.
         for (int i = 0; i < n; i++) {
             if (mask & (1 << i)) {
                 // Process element i.
             }
         }
     Method 2: Repeatedly isolate and remove the lowest set bit.
         while (mask) {
             int lowbit = mask & -mask;   // Isolate lowest set bit.
             int pos = __builtin_ctzll(mask); // Count trailing zeros to get the position.
             // Process element at position pos.
             mask &= (mask - 1); // Remove the lowest set bit.
         }

   Explanation:
     Both methods allow you to iterate only over those positions where the bit is 1.
*/
void example_iterate_set_bits() {
    cout << "\n----- Example 10: Iterating Over Set Bits -----\n";
    int n;
    cout << "Enter n (number of bits in your bitmask): ";
    cin >> n;
    int mask;
    cout << "Enter the bitmask as an integer (should be between 0 and " << ((1 << n) - 1) << "): ";
    cin >> mask;

    // Method 1: Check each bit position.
    cout << "Method 1: Checking each bit position:\n";
    for (int i = 0; i < n; i++) {
        if(mask & (1 << i))
            cout << "Bit " << i << " is set.\n";
    }

    // Method 2: Isolate and remove the lowest set bit repeatedly.
    cout << "Method 2: Isolating the lowest set bit repeatedly:\n";
    int temp = mask;
    while(temp) {
        int lowbit = temp & (-temp);
        // __builtin_ctzll returns the number of trailing zeros, which is the position of the lowest set bit.
        int pos = __builtin_ctzll(temp);
        cout << "Bit " << pos << " is set.\n";
        temp &= (temp - 1); // Clear the lowest set bit.
    }
}

```

```cpp
// ------------------------------------------------------------------------
// Main Menu to run the examples
// ------------------------------------------------------------------------
int32_t main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    while (true) {
        cout << "\n============================================================\n";
        cout << "             Basic Bitmasking Examples - Menu\n";
        cout << "============================================================\n";
        cout << " 1.  Basic Bitmask Operations\n";
        cout << " 2.  Bit Shifting (Left and Right Shift)\n";
        cout << " 3.  Counting Set Bits in an Integer\n";
        cout << " 4.  Check if a Number is a Power of Two\n";
        cout << " 5.  Isolate the Lowest Set Bit\n";
        cout << " 6.  Clear the Lowest Set Bit\n";
        cout << " 7.  Enumerate All Subsets of a Set\n";
        cout << " 8.  Set Operations (Union and Intersection)\n";
        cout << " 9.  Invert a Bitmask (Bitwise NOT)\n";
        cout << "10.  Iterate Over All Set Bits\n";
        cout << "11.  Exit\n";
        cout << "Enter your choice: ";

        int choice;
        cin >> choice;
        if(choice == 11)
            break;

        switch(choice) {
            case 1:  example_basic_operations(); break;
            case 2:  example_bit_shifting(); break;
            case 3:  example_count_set_bits(); break;
            case 4:  example_power_of_two(); break;
            case 5:  example_lowest_set_bit(); break;
            case 6:  example_clear_lowest_set_bit(); break;
            case 7:  example_enumerate_subsets(); break;
            case 8:  example_set_operations(); break;
            case 9:  example_invert_bitmask(); break;
            case 10: example_iterate_set_bits(); break;
            default: cout << "Invalid choice. Please try again.\n";
        }
    }

    return 0;
}
```