Data Structures and Objects
CSIS 3700

Lab 4 — Bitwise Operations

Goal

Explore basic bitwise operations on integer values.

Preparation

Create a file and add the following two functions:

```
#include <iostream>

using namespace std;

void showBits(unsigned int n) {

  for (int i=31;i>=0;i--)
    if (n & (1 << i))
      cout << '1';
    else
      cout << '0';

  cout << endl;
}

int main(void) {
  unsigned int
    a,b,c;

  cout << "Enter␣two␣nonnegative␣integers:␣";
  cin >> a >> b;

  // your code goes here

  return 0;
}
```

The **`showBits()`** function displays an **`unsigned int`** in binary.

The and, or, exclusive-or and not operations

There are four bitwise operations available in C and C++:

Name	Symbol	Result
And	&	1 if both operand bits are 1, 0 otherwise
Or	|	0 if both operand bits are 0, 1 otherwise
Not	~	0 if the operand bit is 1, 1 otherwise
Exclusive Or	^	1 if both operand bits are different, 0 otherwise

These rules are applied to each bit in the operands.

Some examples, showing only the last four bits:

a	b	a & b	a | b	~a	a ^ b
$10 = (1010)_2$	$12 = (1100)_2$	$8 = (1000)_2$	$14 = (1110)_2$	$5 = (0101)_2$	$6 = (0110)_2$
$6 = (0110)_2$	$7 = (0111)_2$	$6 = (0110)_2$	$7 = (0111)_2$	$9 = (1001)_2$	$1 = (0001)_2$
$0 = (0000)_2$	$5 = (0101)_2$	$0 = (0000)_2$	$5 = (0101)_2$	$15 = (1111)_2$	$5 = (0101)_2$
$12 = (1100)_2$	$3 = (0011)_2$	$0 = (0000)_2$	$15 = (1111)_2$	$3 = (0011)_2$	$15 = (1111)_2$

▹*Try it out*

Add a few lines of code to your program to use the **a** and **b** values and combine them with the bitwise operators, placing each result in the **c** variable. Show the operands and each result with the **`showBits()`** function.

Using the bitwise operations — Masking

One of the most common uses for bitwise operations is for *masking*, where some bits in a value are left alone — they are *masked* — and other bits are modified.

In a masking operation, a set of bits stored in an integer is combined with a mask using one of the binary operators; the unary ~ isn't used to mask (although it can be used to create a mask value.)

The mask values to leave a bit alone (the mask value), the value to manipulate a bit and the manipulation action are described in the following table:

Operator	Mask value	Manipulate value	Action
&	1	0	Set bit to 0 (clear the bit)
|	0	1	Set bit to 1 (set the bit)
^	0	1	$b \leftarrow 1 - b$ (toggle the bit)

The following table shows an example of using a mask. Note which bits are left alone and which are modified:

a	mask	Operation	Result
$3 = (0011)_2$	$5 = (0101)_2$	a & mask	$1 = (0001)_2$
$12 = (1100)_2$	$5 = (0101)_2$	a & mask	$4 = (0100)_2$
$6 = (0110)_2$	$5 = (0101)_2$	a | mask	$7 = (0111)_2$
$9 = (1001)_2$	$5 = (0101)_2$	a | mask	$13 = (1101)_2$
$12 = (1100)_2$	$5 = (0101)_2$	a ^ mask	$9 = (1001)_2$
$3 = (0011)_2$	$5 = (0101)_2$	a ^ mask	$6 = (0110)_2$

▹*Masking notes*

- Masks are often — but not always — constants.
- Constant mask values are almost always written as hexadecimal values, since it is easier to visualize the pattern of 1s and 0s than it is when written as a decimal value.
- The result of a mask is usually examined to see if it is either a specific value or fits some pattern, such as being non-zero.

Bit shifting

In addition to the bitwise logical operators, we can also take an integer and shift its bits to the left or right by a given number of bits.

Suppose **n** $= 11 = (1011)_2$. Then, `n << 3` shifts **n** to the left three bits, yielding $88 = (1011000)_2$.

Similarly, `n >> 1` shifts **n** to the right one bit, yielding $5 = (101)_2$.

▹*Try it out*

Try shifting **a** to the left and to the right by **b** bits. Output the results using `showBits()`.

▹*Bit shifting notes*

- Shifting to the left one bit is the same as multiplying by 2. In fact, a good compiler will convert a multiply by a power of 2 into an equivalent bit shift, because shifting is a very easy instruction to create in hardware while multiplying is much more time consuming.
- Shifting to the right one bit is the same as dividing by 2, discarding any remainder. Compilers will also try to take advantage of this.
- In addition to the above, shifting is often used to align a sequence of bits to a specific mask (or vice versa) prior to a masking operation.

▹*Warning*

Shifting bits to the right has two different actions, depending on whether or not the integer is signed or unsigned. This discussion assumes the integers are unsigned.

If the shifted integer is signed, a copy of the leftmost bit is inserted for each position shifted. From a logic viewpoint, this is rarely what is wanted. However, from a mathematical viewpoint, it preserves the divide-by-two behavior when the number is negative.

Counting bits

Suppose we have some non-zero value n. Since it's not zero, there must be at least one bit in n set to 1. Suppose n has the following form:

$$n = b_{m-1}b_{m-2}\cdots b_{k+1}1\,\underbrace{0\cdots0}_{k\geq 0 \text{ bits}}$$

Then, $n-1$ must look like this:

$$n = b_{m-1}b_{m-2}\cdots b_{k+1}0\,\underbrace{1\cdots1}_{k\geq 0 \text{ bits}}$$

If we combine these with an & operator, the first $m-k-1$ bits remain the same since any bit anded with itself yields itself as the result, while the last $k+1$ bits are set to zero. Thus,

$$n \,\&\, (n-1) = b_{m-1} b_{m-2} \cdots b_{k+1} \underbrace{0 \cdots 0}_{k+1 \text{ bits}}$$

The end result is that the statement **`n = n & (n-1)`** turns off one of the 1-bits of n.

The following function uses this to count the number of 1-bits in a given integer:

```
int countBits(unsigned int n) {
  int
    count=0;

  while (n != 0) {
    count++;
    n &= n - 1;
  }

  return count;
}
```

▹*Try it out*

Add the **`countBits()`** function to your program, above **`main()`**. Use it to count the number of bits in **`a`** and output the result.

Precedence issues

You must exercise some caution with respect to precedence of the bitwise operators. Bit shifting has lower precedence than arithmetic but higher than the comparison operators. And, or and exclusive-or have lower precedence than comparisons but higher precedence than logical and. Bitwise and is higher than bitwise or which is higher than bitwise exclusive-or. Bitwise not has extremely high precedence, the same as logical not.

Further exploration for another time

Look into the **`bitset`** class.

What to turn in

Turn in the program you ended up writing.