Bitwise Operators

Introduction

This workshop presents a refresher course in bitwise operations, and how they are implemented in C and C++. Bitwise operations perform fast primitive actions on binary numerals at the level of their individual bits. The operators either combine two numbers using binary logic, or they shift the bits of the binary number to the left or right.

Binary Operators

Computers use binary representation, and combination, to operate. Each piece of data is represented as a set of binary values, or bits. Each bit has two possible states:

- 1 or TRUE
- 0 or FALSE

There are a number of types of binary operator which allow manipulation of data at this bitwise level. This section describes each operator.

Note that, as with the arithmetic operators, these are overloadable in C++.

AND "&"

The bitwise **AND** operator, combines two inputs by comparing their respective bit values, setting the resultant bit to TRUE if both inputs are TRUE, and setting it to FALSE in all other cases. Simply put, the output is TRUE if input1 is TRUE **AND** input2 is TRUE.

Input1	Input2	Output
0	0	0
1	0	0
0	1	0
1	1	1

In C and C++, the symbol & is used to represent a binary AND.

Consider a case where a = 15 (i.e. binary 1111) and b = 4 (i.e. binary 0100). If c = a & b, then c is equal to 4 (i.e. binary 0100). Each bit of the two inputs has been compared – the only bit which is TRUE in both inputs is the third one, so that bit is set to TRUE in the output and the other three bits are set to FALSE.

OR "|"

The bitwise **OR** operator, combines two inputs by comparing their respective bit values, setting the resultant bit to TRUE if either input is TRUE, and setting it to FALSE if both inputs are FALSE. Simply put, the output is TRUE if input1 is TRUE **OR** input2 is TRUE.

Input1	Input2	Output
0	0	0
1	0	1
0	1	1
1	1	1

In C and C++, the symbol | is used to represent a binary OR.

Consider a case where a = 15 (i.e. binary 1111) and b = 4 (i.e. binary 0100). If c = a | b, then c is equal to 15 (i.e. binary 1111). Each bit of the two inputs has been compared – every bit is TRUE in at least one input, so all bits of the output are set to TRUE.

XOR "^"

The bitwise X**OR** operator (the exclusive OR), combines two inputs by comparing their respective bit values, setting the resultant bit to TRUE if either one or the other input is TRUE, and setting it to FALSE if both inputs are FALSE or both inputs are TRUE. Simply put, the output is TRUE if only one of input1 and input2 is TRUE.

Input1	Input2	Output
0	0	0
1	0	1
0	1	1
1	1	0

In C and C++, the symbol ^ is used to represent a binary XOR.

Consider a case where a = 15 (i.e. binary 1111) and b = 4 (i.e. binary 0100). If c = a ^ b, then c is equal to 11 (i.e. binary 1011). Each bit of the two inputs has been compared – the third bit is TRUE for both inputs, so it is set to FALSE in the output.

NOT "~"

The bitwise **NOT** operator takes a single input and inverts each bit in the output. Simply put, it replaces TRUE with FALSE and vice versa

Input	Output
0	1
1	0

In C and C++, the symbol ~ is used to represent a binary NOT.

Consider a case where b = 4 (i.e. binary 0100). If c = ~ b, then c is equal to 11 (i.e. binary 1011). Each bit of the input has been inverted.

Bit Shifts "<<" and ">>"

The bits comprising a binary numeral can be shifted to the left or right. It should be evident that each shift is equivalent to multiplying or dividing the numeral by two.

In C and C++, the symbols << and >> are used to shift bits to the left and right respectively.

Consider a case where b = 4 (i.e. binary 0100). If c = b>>2, then c is equal to 1 (i.e. binary 0001). Each bit of the numeral has been shifted two spaces to the right. As bits are shifted out at one end, zeroes are shifted in at the other.

Compound Operators

As with arithmetic operators, the binary operators can be combined with the = symbol, to create compound operators.

Again these are overloadable in C++.

Operator name	Syntax	Code equivalent
Binary AND assignment	a &= b	a = a & b
Binary OR assignment	a |= b	a = a | b
Binary XOR assignment	a ^= b	a = a ^ b
Left shift assignment	a >>= b	a = a >> b
Right shift assignment	a <<= b	a = a << b

Bit-flags and Bit-masks

A common use of binary operators is in setting and checking a series of flags pertaining to the properties of the item represented by the data structure or class. We'll use superheroes to demonstrate this.

Imagine that we have a game involving many superheroes with various different super-powers. There are sixteen possible superpowers, and each hero can have any combination of those powers. We therefore need a set of flags to indicate whether each hero has each superpower. We can store this information within a single 16 bit variable for each hero, by the use of bit flags.

We do this with a series of #defines, each set to a single binary digit (ie a power of two). For example, the first bit may be used to define whether the superhero can fly, the second bit for X-Ray vision, etc.

- #define SUPER_POWER_FLYING 1
- #define SUPER_POWER_XRAYVISION 2
- #define SUPER_POWER_STRENGTH 4
- #define SUPER_POWER_INVISIBILITY 8

In order to test whether a character can fly, we use the binary AND operator:

- If (pHero-> iPowerFlags & SUPER_POWER_FLYING)

This test will pass all heroes who have the "flying" bit set in the iPowerFlags integer, irrelevant of what other flags may be set.

To set the flag, we use

- pHero-> iPowerFlags |= SUPER_POWER_FLYING;

This statement uses the binary OR assignment to set the bit representing “flying” to TRUE without affecting any of the other flags in the integer.

To reset the flag to FALSE, we use

- pHero-> iPowerFlags &= ~SUPER_POWER_FLYING;

which uses the binary AND assignment operator to compare with the inverse of the flag for flying (ie the NOT operator turns 0001 into 1110). Again this will reset the “flying” flag to FALSE without affecting any other flags in the integer.

We can now create characters with any combinations of the super-powers, and reliably test which ones are required by specific algorithms.

This approach is widespread in games code – used to flag anything from graphical rendering properties, to AI properties, to physics properties, etc. It saves memory compared to storing a full Boolean variable (usually the same size as an integer) for each flag and, as it uses the bitwise operators, it is very fast. The definitions of the flags are often knows as “bit-masks”, as they mask out the data which is not relevant, without affecting it.

Code Sample

The simple code sample sets up a basic class “Superhero” consisting of a integer containing a set of bit-flags, and a method for testing which powers the hero has. The main loop creates a hero, sets some flags, and tests them. It then removes one of the flags and tests them again.

```
#include <iostream>
using namespace std;

#define SUPER_POWER_FLYING              1
#define SUPER_POWER_XRAYVISION          2
#define SUPER_POWER_STRENGTH            4
#define SUPER_POWER_INVISIBILITY        8

class Superhero
{
public:
       void TestPowers();
       int    iPowerFlags;
};


/*****************************************
*   Superhero::TestPowers
*
*****************************************/
void Superhero::TestPowers()
{
       //test for power flags using the binary AND
       if (iPowerFlags & SUPER_POWER_FLYING)
       {
              cout << ("Can Fly\n");
       }
       if (iPowerFlags & SUPER_POWER_XRAYVISION)
       {
```

```
                cout << ("Can See Through Walls\n");
        }
        if (iPowerFlags & SUPER_POWER_STRENGTH)
        {
                cout << ("Is Strong\n");
        }
        if (iPowerFlags & SUPER_POWER_INVISIBILITY)
        {
                cout << ("Can Disappear\n");
        }
}

/*****************************************
*   main
*
*****************************************/
int main()
{
        int i;
        Superhero Superman;
        //initialise the flags to zero
        Superman.iPowerFlags = 0;
        //Set his superpowers using the binary OR assignment
        Superman.iPowerFlags |= SUPER_POWER_FLYING;
        Superman.iPowerFlags |= SUPER_POWER_XRAYVISION;
        Superman.iPowerFlags |= SUPER_POWER_STRENGTH;

        Superman.TestPowers();
        cout << "Enter any number ";
        cin >> i;

        //Remove a power using the binary AND assignment and the binary NOT
        Superman.iPowerFlags &= ~SUPER_POWER_FLYING;

        Superman.TestPowers();
        cout << "Enter any number ";
        cin >> i;

        return 0;
}
```

Exercises

- Practice converting numbers between decimal and binary. Use a pen and paper for this, look around and write down the binary equivalent of any numbers you see.
- Use a pen and paper to perform binary AND, OR and XOR operations on two numbers.
- Extend the superhero code to include some more heroes with different powers. Add three powers which are SUPERPOWER_ROCK, SUPERPOWER_PAPER and SUPERPOWER_SCISSORS. Write the ode which decides which characters win in a game of Rock, Paper, Scissors.