Поразрядные операции

Особый класс операций представляют поразрядные операции. Они выполняются над отдельными разрядами числа. В этом плане числа рассматриваются в двоичном представлении, например, 2 в двоичном представлении 10 и имеет два разряда, число 7 - 111 и имеет три разряда.

Логические операции

- **&**(логическое умножение)

  Умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

```
1   int x1 = 2; //010
2    int y1 = 5;//101
3    Console.WriteLine(x1&y1); // выведет 0
4
5   int x2 = 4; //100
6   int y2 = 5; //101
7   Console.WriteLine(x2 & y2); // выведет 4
```

  В первом случае у нас два числа 2 и 5. 2 в двоичном виде представляет число 010, а 5 - 101. Поразрядно умножим числа (0*1, 1*0, 0*1) и в итоге получим 000.

  Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же как и у числа 5, поэтому в итоге получим (1*1, 0*0, 0 *1) = 100, то есть число 4 в десятичном формате.

- **|** (логическое сложение)

  Похоже на логическое умножение, операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица. Например:

```
1   int x1 = 2; //010
2   int y1 = 5;//101
3   Console.WriteLine(x1|y1); // выведет 7 - 111
4   int x2 = 4; //100
5   int y2 = 5;//101
```

```
6   Console.WriteLine(x2 | y2); // выведет 5 - 101
```

- **^** (логическое исключающее ИЛИ)

  Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

```
1   int x = 45; // Значение, которое надо зашифровать - в двоичной форме 101101
2   int key = 102; //Пусть это будет ключ - в двоичной форме 1100110
3
4   int encrypt = x ^ key; //Результатом будет число 1001011 или 75
5   Console.WriteLine($"Зашифрованное число: {encrypt}") ;
6
7   int decrypt = encrypt ^ key; // Результатом будет исходное число 45
8   Console.WriteLine($"Расшифрованное число: {decrypt}");
```

  Здесь опять же производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Таким образом, мы получаем из 9^5 в качестве результата число 12. И чтобы расшифровать число, мы применяем ту же операцию к результату.

- **~** (логическое отрицание или инверсия)

  Еще одна поразрядная операция, которая инвертирует все разряды: если значение разряда равно 1, то оно становится равным нулю, и наоборот.

```
1   int x = 12;                 // 00001100
2   Console.WriteLine(~x);      // 11110011   или -13
```

Представление отрицательных чисел

Для записи чисел со знаком в C# применяется **дополнительный код** (two's complement), при котором старший разряд является знаковым. Если его значение равно 0, то число положительное, и его двоичное представление не отличается от представления беззнакового числа. Например, 0000 0001 в десятичной системе 1.

Если старший разряд равен 1, то мы имеем дело с отрицательным числом. Например, 1111 1111 в десятичной системе представляет -1. Соответственно, 1111 0011 представляет -13.

Чтобы получить из положительного числа отрицательное, его нужно инвертировать и прибавить единицу:

```
int x = 12;
int y = ~x;
y += 1;
Console.WriteLine(y);   // -12
```

Операции сдвига

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево.

- `x<<y` - сдвигает число x влево на y разрядов. Например, `4<<1` сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, то есть в итоге получается 1000 или число 8 в десятичном представлении.
- `x>>y` - сдвигает число x вправо на y разрядов. Например, `16>>1` сдвигает число 16 (которое в двоичном представлении 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении.

Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два. Например:

```
int a = 16; // в двоичной форме 10000
int b = 2; // в двоичной форме
int c = a << b; // Сдвиг числа 10000 влево на 2 разряда, равно 1000000 или 64 в де

Console.WriteLine($"Зашифрованное число: {c}") ;    // 64

int d = a >> b; // Сдвиг числа 10000 вправо на 2 разряда, равно 100 или 4 в десяти
Console.WriteLine($"Зашифрованное число: {d}");     // 4
```

При этом числа, которые участвую в операциях, необязательно должны быть кратны 2:

```
int a = 22; // в двоичной форме 10110
int b = 2; // в двоичной форме
int c = a << b; // Сдвиг числа 10110 влево на 2 разряда, равно 1011000 или 88 в де

```

```
5   Console.WriteLine($"Зашифрованное число: {c}") ;    // 88
6
7   int d = a >> b; // Сдвиг числа 10110 вправо на 2 разряда, равно 101 или 5 в десяти
8   Console.WriteLine($"Зашифрованное число: {d}");     // 5
```

Пример практического применения операций

Многие недооценивают поразрядные операции, не понимают, для чего они нужны. Тем не менее они могут помочь в решении ряда задач. Прежде всего они позволяют нам манипулировать данными на уровне отдельных битов. Один из примеров. У нас есть три числа, которые находятся в диапазоне от 0 до 3:

```
1   int value1 = 3;  // 0b0000_0011
2   int value2 = 2;  // 0b0000_0010
3   int value3 = 1;  // 0b0000_0001
```

Мы знаем, что значения этих чисел не будут больше 3, и нам нужно эти данные максимально сжать. Мы можем три числа сохранить в одно число. И в этом нам помогут поразрядные операции.

```
1    int value1 = 3;  // 0b0000_0011
2    int value2 = 2;  // 0b0000_0010
3    int value3 = 1;  // 0b0000_0001
4    int result = 0b0000_0000;
5    // сохраняем в result значения из value1
6    result = result | value1; // 0b0000_0011
7    // сдвигаем разряды в result на 2 разряда влево
8    result = result << 2;   // 0b0000_1100
9    // сохраняем в result значения из value2
10   result = result | value2;  // 0b0000_1110
11   // сдвигаем разряды в result на 2 разряда влево
12   result = result << 2;   // 0b0011_1000
13   // сохраняем в result значения из value3
14   result = result | value3;  // 0b0011_1001
15
16   Console.WriteLine(result);  // 57
```

Разберем этот код. Сначала определяем все сохраняемые числа value1, value2, value3. Для хранения результата определена переменная result, которая по умолчанию равна 0. Для большей наглядности ей присвоено значение в бинарном формате:

```
int result = 0b0000_0000;
```

Сохраняем первое число в result:

```
result = result | value1; // 0b0000_0011
```

Здесь мы имеем дело с логической операцией поразрядного сложения - если один из соответствующих разрядов равен 1, то результирующий разряд тоже будет равен 1. То есть фактически

```
0b0000_0000
+
0b0000_0011
=
0b0000_0011
```

Итак, первое число сохранили в result. Мы будем сохранять числа по порядку. То есть сначала в result будет идти первое число, затем второе и далее третье. Поэтому сдвигаем число result на два разряда влево (наши числа занимают в памяти не более двух разрядов):

```
result = result << 2;   // 0b0000_1100
```

То есть фактически

```
0b0000_0011 << 2 =
0b0000_1100
```

Далее повторяем логическую операцию сложения, сохраняем второе число:

```
result = result | value2;  // 0b0000_1110
```

что эквивалентно

```
0b0000_1100
+
0b0000_0010
```

```
4   =
5   0b0000_1110
```

Далее повторяем сдвиг на два разряда влево и сохраняем третье число. В итоге мы получим в двоичном представлении число `0b0011_1001`. В десятично системе это число равно 57. Но это не имеет значения, потому что нам важны конкретные биты числа. Стоит отметить, что мы сохранили в одно число три числа, и в переменной result еще есть сводобное место. Причем в реальности не важно, сколько именно битов надо сохранить. В данном случае для примера сохраняем лишь два бита.

Для восстановления данных прибегнем к обратному порядку:

```
1    result = 0b0011_1001
2    // обратное получение данных
3    int newValue3 = result & 0b000_0011;
4    // сдвигаем данные на 2 разряда вправо
5    result = result >> 2;
6    int newValue2 = result & 0b000_0011;
7    // сдвигаем данные на 2 разряда вправо
8    result = result >> 2;
9    int newValue1 = result & 0b000_0011;
10   Console.WriteLine(newValue1);    // 3
11   Console.WriteLine(newValue2);   // 2
12   Console.WriteLine(newValue3);   // 1
```

Получаем числа в порядке, обратном тому, в котором они были сохранены. Поскольку мы знаем, что каждое сохраненное число занимает лишь два разряда, то по сути нам надо получить лишь последние два бита. Для этого применяем битовую маску `0b000_0011` и операцию логического умножения, которая возвращает 1, если каждый из двух соответствующих разрядов равен 1. То есть операция

```
1   int newValue3 = result & 0b000_0011;
```

эквивалентна

```
1   0b0011_1001
2   *
```

```
0b0000_0011

=

0b0000_0001
```

Таким образом, последнее число равно 0b0000_0001 или 1 в десятичной системе

Стоит отметить, что если мы точно знаем структуру данных, то мы легко можем составить битовую маску, чтобы получить нужно число:

```
result = 0b0011_1001;
int recreatedValue1 = (result & 0b0011_0000) >> 4;
Console.WriteLine(recreatedValue1);
```

Здесь получаем первое число, которое, как мы знаем, занимает в числе биты 4 и 5. Для этого применяем умножение на битовую маску 0b0011_0000. И затем сдвигаем число на 4 разряда вправо.

```
0b0011_1001

*

0b0011_0000

=

0b0011_0000

>> 4

=

0b0000_0011
```

Аналогично, если мы точно знаем структуру, по которой сохраняются данные, то мы могли бы сохранить данные сразу в нужное место в числе result:

```
 int value1 = 3;  // 0b0000_0011
 int value2 = 2;  // 0b0000_0010
 int value3 = 1;  // 0b0000_0001
 int result = 0b0000_0000;
 // сохраняем в result значения из value1
 result = result | (value1 << 4);
 // сохраняем в result значения из value2
```

```
8    result = result | (value2 << 2);
9    // сохраняем в result значения из value3
10   result = result | value3;  // 0b0011_1001
11
12   Console.WriteLine(result);  // 57
```