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  2  3  4  5  6  7 | int x1 = 2; //010  int y1 = 5;//101  Console.WriteLine(x1&y1); // выведет 0    int x2 = 4; //100  int y2 = 5; //101  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  2  3  4  5  6 | int x1 = 2; //010  int y1 = 5;//101  Console.WriteLine(x1|y1); // выведет 7 - 111  int x2 = 4; //100  int y2 = 5;//101  Console.WriteLine(x2 | y2); // выведет 5 - 101 |

* **^** (логическое исключающее ИЛИ)
* Также эту операцию называют XOR, нередко ее применяют для простого шифрования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int x = 45; // Значение, которое надо зашифровать - в двоичной форме 101101  int key = 102; //Пусть это будет ключ - в двоичной форме 1100110    int encrypt = x ^ key; //Результатом будет число 1001011 или 75  Console.WriteLine($"Зашифрованное число: {encrypt}") ;    int decrypt = encrypt ^ key; // Результатом будет исходное число 45  Console.WriteLine($"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  2 | int x = 12; // 00001100  Console.WriteLine(~x); // 11110011 или -13 |

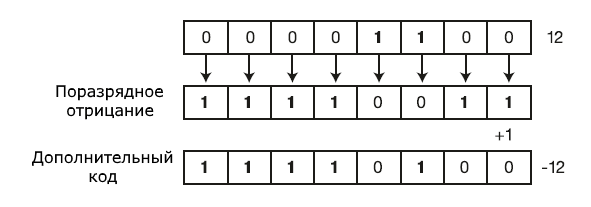
### Представление отрицательных чисел

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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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 десятичной системе    Console.WriteLine($"Зашифрованное число: {c}") ; // 64    int d = a >> b; // Сдвиг числа 10000 вправо на 2 разряда, равно 100 или 4 в десятичной системе  Console.WriteLine($"Зашифрованное число: {d}"); // 4 |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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 десятичной системе    Console.WriteLine($"Зашифрованное число: {c}") ; // 88    int d = a >> b; // Сдвиг числа 10110 вправо на 2 разряда, равно 101 или 5 в десятичной системе  Console.WriteLine($"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  2  3 | int value1 = 3; // 0b0000\_0011  int value2 = 2; // 0b0000\_0010  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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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; // 0b0000\_0011  // сдвигаем разряды в result на 2 разряда влево  result = result << 2; // 0b0000\_1100  // сохраняем в result значения из value2  result = result | value2; // 0b0000\_1110  // сдвигаем разряды в result на 2 разряда влево  result = result << 2; // 0b0011\_1000  // сохраняем в result значения из value3  result = result | value3; // 0b0011\_1001    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:

|  |  |
| --- | --- |
| 1 | int result = 0b0000\_0000; |

Сохраняем первое число в result:

|  |  |
| --- | --- |
| 1 | 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

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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):

|  |  |
| --- | --- |
| 1 | result = result << 2; // 0b0000\_1100 |

То есть фактически

|  |  |
| --- | --- |
| 1  2 | 0b0000\_0011 << 2 =  0b0000\_1100 |

Далее повторяем логическую операцию сложения, сохраняем второе число:

|  |  |
| --- | --- |
| 1 | result = result | value2; // 0b0000\_1110 |

что эквивалентно

|  |  |
| --- | --- |
| 1  2  3  4  5 | 0b0000\_1100  +  0b0000\_0010  =  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  2  3  4  5  6  7  8  9  10  11  12 | result = 0b0011\_1001  // обратное получение данных  int newValue3 = result & 0b000\_0011;  // сдвигаем данные на 2 разряда вправо  result = result >> 2;  int newValue2 = result & 0b000\_0011;  // сдвигаем данные на 2 разряда вправо  result = result >> 2;  int newValue1 = result & 0b000\_0011;  Console.WriteLine(newValue1); // 3  Console.WriteLine(newValue2); // 2  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  2  3  4  5 | 0b0011\_1001  \*  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:

|  |  |
| --- | --- |
| 1  2  3 | 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | 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  result = result | (value2 << 2);  // сохраняем в result значения из value3  result = result | value3; // 0b0011\_1001    Console.WriteLine(result); // 57 |