[1. Битовые операции](#_Toc535526684)

[1.1. Двоичное представление числовых значений](#_Toc535526685)

[1.2. Битовая маска](#_Toc535526686)

[1.3. Битовые операторы (операции с битовыми масками)](#_Toc535526687)

[1.3.1. Побитовое И (оператор &)](#_Toc535526688)

[1.3.2. Побитовое ИЛИ (оператор |)](#_Toc535526689)

[1.3.3. Побитовое отрицание (оператор ~)](#_Toc535526690)

[1.3.4. Исключающее ИЛИ (оператор ^)](#_Toc535526691)

[1.4. Дополнительный код](#_Toc535526692)

[1.5. Операции побитового сдвига](#_Toc535526693)

[1.6. Применение поразрядных операций](#_Toc535526694)

[1.6.1. Степень числа 2](#_Toc535526695)

[1.6.2. Получение значения отдельных бит](#_Toc535526696)

[1.6.3. Установка единицы в отдельный бит](#_Toc535526697)

[1.6.4. Установка нуля в отдельный бит](#_Toc535526698)

[1.6.5. Битовый массив](#_Toc535526699)

[1.6.6. Передача параметров в функции](#_Toc535526700)

[1.7. Примеры](#_Toc535526701)

[1.7.1. Примеры для языка C](#_Toc535526702)

[1.7.2. Примеры для языка CPP](#_Toc535526703)

[1.7.2.1. Сдвиги влево](#_Toc535526704)

[1.7.2.2. Сдвиги вправо](#_Toc535526705)

[1.7.2.3. Сдвиги и перемещения](#_Toc535526706)

[1.7.2.4. Некоторые подробности](#_Toc535526707)

1. [Битовые операции](#Лаб10)

Язык Си иногда называют макроассемблером за его тягу к железу. Если не использовать оптимизацию, можно даже примерно оценить, в какие конструкции на ассемблере преобразуется код программы. Простота и минимализм языка (простоту языка не путать с простотой программирования на языке) привели к тому, что на многих платформах си остаётся единственным высокоуровневым языком программирования. Без обзора битовых операций, конечно, изучения языка было бы неполным.

Побитовые операции, как понятно из названия, позволяют оперировать непосредственно с битами. Большое количество примеров использования побитовых операций можно найти, например, в книге Генри Уоррена «Алгоритмические трюки для программистов». Здесь мы рассмотрим только сами операции и примитивные алгоритмы.

[Битовые операции](#Лаб10) не особенность языка - это скорее особенность процессоров, принципов вычислений, ассемблера. Раз уж эти операции существуют, причем именно в том виде, в каком к ним привыкли то логично предположить, что многие программы и алгоритмы рассчитывают на их наличие в инструментарии разработчика. Представьте что вам пришлось бы портировать некий криптографический алгоритм использующий эти операторы на js в котором их не было бы. Допустим вы реализуете его умножая/деля на два, но затем обнаруживаете, что при умножении старший бит у вас исчезает, а должен становится самый младшим. Кроме того ваша реализация на js была бы много медленней чем встроенная в язык реализация.

Вместо того, чтобы проводить умножение (поначалу это было десятки тактов), подключаем регистр к регистрам процессора, и выполним всё за один такт. Существовала даже программа, генерирующая быстрый код умножения на константу, например, 6 — сдвинуть, сложить и ещё раз сдвинуть. И это в своё время было быстрее, чем mul — а ведь сейчас мы без зазрения совести пишем mul eax, 6. По правилам хорошего программистского тона умножение, даже на 2 или на 4, пишется как обычное \* (>> пишут, если компилятор непроходимо туп и умножение медленнее). А вот сдвиги — именно их, а не умножение — пишут, когда работают с битовыми полями.

Есть также специальные сдвиги.

* Арифметический сдвиг вправо (левый совпадает с обычным) — для чисел со знаком.
* Сдвиг-вращение — в основном для криптографии.
* Сдвиг-вращение через регистр флагов — в основном для длинной арифметики.

Битовые операторы наиболее часто применяются при разработке драйверов устройств, например программ для модемов, дисков и принтеров, поскольку битовые операторы могут использоваться для выключения некоторых битов, например четности. (Бит четности используется для подтверждения того, что остальные биты в байте не изменялись. Он, как правило, является старшим битом в байте.)

Битовые операторы проводят операции непосредственно на битах числа, поэтому числа в примерах будут в двоичной системе счисления.

В теме рассмотрены операторы:

* | (Побитовое ИЛИ (OR)),
* & (Побитовое И (AND)),
* ^ (Исключающее ИЛИ (XOR)),
* ~ (Побитовое отрицание (NOT)),
* << (Побитовый сдвиг влево),
* >> (Побитовый сдвиг вправо).

Битовые операции изучаются в дискретной математике, а также лежат в основе цифровой техники, так как на них основана логика работы логических вентилей — базовых элементов цифровых схем. В дискретной математике, как и в цифровой технике, для описания их работы используются таблицы истинности.

О битовых операторах вам также необходимо знать:

* Некоторые побитовые операторы похожи на операторы, с которыми вы наверняка знакомы (&&, ||). Это потому, что они на самом деле в чем-то похожи. Тем не менее, путать их ни в коем случае нельзя.
* Большинство битовых операций являются операциями составного присваивания.

## Двоичное представление числовых значений

Как известно, целые числа представляются в виде последовательности цифр. Такое представление чисел называется позиционным. Весь набор цифр, которые могут использоваться в позиционном представлении числа, определяет систему счисления. В повседневной жизни используется десятичная система счисления, в которой числа представляются цифрами от 0 до 9. В программировании более популярны системы счисления с количеством цифр, равным степени двойки: восьмеричная и шестнадцатеричная.

Однако двоичная система счисления – вне конкуренции. В этой системе счисления числа записываются последовательностью из двух цифр: 0 и 1. Каждая позиция в двоичном представлении числа соответствует биту. Таким образом, с помощью бита можно задать два значения: 0 или 1. Если для представления числа используется n бит, то в этом случае существует 2n различных комбинаций, каждая из которых соответствует отдельному числу. Например, с помощью 8 бит (1 байт) можем записать 28= 256 чисел.

При представлении двоичным кодом положительных чисел можно было бы использовать стандартное математическое представление числа в двоичной системе. Однако на практике приходится иметь дело и с отрицательными числами, причем с технической точки зрения знаком минус здесь не обойтись – минус можно написать на бумаге, а реализовать его в памяти компьютера намного сложнее.

**Для определения знака числа используют старший бит в позиционной записи. Нулевой старший бит соответствует положительному числу, а единичный старший бит соответствует отрицательному числу.** При этом перевод для положительных чисел из двоичной системы счисления в десятичную осуществляется стандартными методами: если в двоичном представлении число позиционно задается как

(причем цифры bi могут принимать значения 0 или 1, а старший бит для положительных чисел bn = 0), то в десятичной системе число вычисляется как

С отрицательными числами дела обстоят несколько сложнее. Чтобы перевести отрицательное число с позиционным представлением в двоичной системе

(старший бит для отрицательного числа 1 bn = 1), необходимо проделать несложную процедуру из двух этапов.

**Во-первых**, производится побитовое инвертирование кода, т.е. каждый бит в представлении числа меняется на противоположный: 0 на 1 и 1 на 0.

**Во-вторых**, результат переводится в десятичную систему и к нему добавляется 1. Это модуль отрицательного числа.

**Чтобы получить само число, необходимо умножить на -1**. Чтобы перевести отрицательное число из десятичной системы в двоичную, проделывают обратную процедуру: от модуля отрицательного числа отнимается 1, результат переводится в бинарный код, после чего производится побитовое инвертирование. Проиллюстрируем это на примере. Рассмотрим 8-битовое бинарное положительное число 01001011, что в десятичной системе счисления соответствует числу 75

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **75** |

Определим бинарное машинное представление для отрицательного числа -75. Отнимаем от модуля числа единицу, получаем 74. Бинарное представление для этого числа 01001010

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| **0** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **74** |

После побитового инвертирования из числа 01001010 получаем 10110101. Это и есть представление числа -75.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| **0** | **1** | **0** | **0** | **1** | **0** | **1** | **0** | **74** |
| **1** | **0** | **1** | **1** | **0** | **1** | **0** | **1** | **-75** |

В том, что это так, легко убедиться: сложим числа 01001011 и 10110101.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
|  | **0** | **1** | **0** | **0** | **1** | **0** | **1** | **1** | **75** |
|  | **1** | **0** | **1** | **1** | **0** | **1** | **0** | **1** | **-75** |
| **1** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** | **0** |

Формально получаем 100 000 000, однако поскольку числа 8-битовые, лишний единичный старший бит отбрасывается, и получаем представление 00 000 000, что соответствует нулю, как и должно быть.

Теперь рассмотрим основные побитовые операции и операторы, которые используются для этого в языке программирования C++. Список побитовых операторов приведен в таблице.

Таблица. Побитовые операторы C++

|  |  |
| --- | --- |
| **Оператор** | **Назначение** |
| & | Побитовое И. Бинарный оператор. Результатом выражения a&b является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если оба сравниваемых бита равны 1. В противном случае значение бита равно 0 |
| | | Побитовое ИЛИ. Бинарный оператор. Результатом выражения a|b является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если хотя бы один из сравниваемых битов равен 1. В противном случае значение бита равно 0 |
| ^ | Побитовое исключающее ИЛИ. Бинарный оператор. Результатом выражения a^b является число, каждый бит которого в двоичном представлении равен результату сравнения соответствующих битов чисел a и b : значение бита равно 1, если один и только один из сравниваемых битов равен 1. В противном случае значение бита равно 0 |
| ~ | Побитовое отрицание (дополнение до единицы). Унарный оператор. Результатом выражения ~a является число, которое получается побитовым инвертированием числа a |
| >> | Сдвиг вправо. Бинарный оператор. В двоичном представлении числа, указанном слева от оператора, выполняется сдвиг всех битов вправо на число позиций, указанных справа от оператора. При этом старший бит знака остается неизменным, а выходящие за диапазон младшие биты теряются |
| << | Сдвиг влево. Бинарный оператор. В двоичном представлении числа, указанном слева от оператора, выполняется сдвиг всех битов влево на число позиций, указанных справа от оператора, с заполнением младших битов нулями и потерей старших битов |

Приведем некоторые примеры использования побитовых операторов. Они представлены в таблице. Даже в самых простых случаях результат может оказаться несколько неожиданным.

Таблица. Примеры использования побитовых операторов

|  |  |  |
| --- | --- | --- |
| **Выражение** | **Значение** | **Пояснение** |
| 5&3 | 1 | В двоичном представлении число 5 имеет вид 101 , а число 3 представляется как 011 . Побитовое сравнение чисел 101 и 011 с помощью оператора побитового И & дает 001 , что в десятичной системе соответствует числу 1 |
| 5|3 | 7 | Применение оператора побитового ИЛИ | для сравнения чисел 101 и 011 дает 111 , что в десятичной системе соответствует числу 7 |
| 5^3 | 6 | Применение оператора побитового исключающего ИЛИ ^ для сравнения чисел 101 и 011 дает 110 , что в десятичной системе соответствует числу 6 |
| ~5 | -6 | После применения операции побитового инвертирования ~ к числу 5 требует особых пояснений. На самом деле в 8-битовом представлении число 5 имеет вид 00000101 . В предыдущих случаях нулевые старшие разряды роли не играли, поэтому они явно не указывались. При инвертировании наличие старших нулевых битов важно. Инвертирование дает 11111010 . Это не что иное, как представление в двоичном машинном коде числа -6 . Последнее читатель может проверить самостоятельно |
| 5>>2 | 1 | После сдвига вправо на две позиции для числа 5 (двоичный код 101 ) получаем 001 . В десятичной системе это число 1 |
| 5<<2 | 20 | После сдвига влево на две позиции для числа 5 (двоичный код 101 ) получаем 10100 . В десятичной системе это число 20 |

Обращаем внимание на особенности применения операции побитового сдвига к отрицательным числам. Например, результатом выражения -6>>5 является число -1. Дело в том, что в бинарном коде 11111010 для числа -6 при сдвиге вправо на 5 позиций при условии сохранения значения старшего бита знака получаем код 11111111. Это код числа -1.

Особенности операций в двоичной системе таковы, что сдвиг в побитовом представлении числа на одну позицию влево означает умножение этого числа на 2. Следует только помнить, что с определенного момента при сдвиге вправо теряются старшие биты. Представим, что число задается 8 битами. Если воспользоваться командой 1<<6, получим в качестве результата значение 26 = 64. Действительно, десятичное число 1 в двоичной системе в 8-битовом представлении задается как 00000001. После сдвига влево на 6 позиций получаем 01000000, что в десятичной системе соответствует числу 64. Однако если воспользоваться командой 1<<7, получим в качестве результата -128. Объясняется это следующим обстоятельством. После сдвига влево на 7 позиций из числа 00000001 получаем число 1000000. Это отрицательное число, о чем свидетельствует старший единичный бит. Переводя это число в десятичную систему, сначала инвертируем бинарный код и получаем 01111111. Это код числа 127. Чтобы получить конечное значение, необходимо прибавить к этому результату 1 и добавить минус – в результате приходим к значению -128.

## Битовая маска

**Битовая маска** — определённые данные, которые используются для маскирования — выбора отдельных [битов](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) или [полей из нескольких битов](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BF%D0%BE%D0%BB%D0%B5) из [двоичной](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) строки или числа.

Как известно, числа в памяти компьютера представляются в двоичной системе счисления в виде последовательности битов. Один бит может иметь значение 0 или 1. Можно провести аналогию между битами и значениями типа bool: 0 обозначает false, а 1 – true. По такой аналогии число (последовательность битов) можно представить как массив значений bool. Например, тип int может обозначать массив из 32 значений bool, а long long – из 64.

Чаще всего массивы bool небольшого размера используются для обозначения некоторого подмножества объектов, выбранного из множества. Например, для обозначения элементов с индексами 1 и 4 (0-индексация), выбранных из множества из пяти элементов используется массив {false,true,false,false,true}, или {0,1,0,0,1}. Его можно представить в виде значения типа int: 100102 (индексация обычно начинается с младших битов числа, записываемых справа).

При интерпретации такого значения как обычного числа, оно будет равно 100102=1810. Но при интерпретации его как массива логических значений (битов), оно будет обозначать {0,1,0,0,1}. С точки зрения C++ эти два значения равносильны, и то, является ли значение типа int числом, или массивом bool, зависит только от контекста, в котором оно используется.

При использовании значений типа int или long long как массивов из bool, такие значения называются **битовыми масками**.

**Получение значения бит**

Битовую маску возможно использовать для получения значения бита. Для этого нужно отключить все остальные биты с помощью побитового [логического «И» (конъюнкции)](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8#AND).

Например, для получения значения пятого бита (считая слева) числа 10111011 нужно использовать маску 00001000. В результате получится:

10111011 & 00001000 = 00001000

Подобное число в языках, использующих вместо логического типа числовые типы, например в [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), будет означать истину или ложь, если этот бит принимает соответствующее значение. В языках, имеющих логические типы (например, [C++](https://ru.wikipedia.org/wiki/C%2B%2B)), необходимо произвести [приведение типа](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D1%82%D0%B8%D0%BF%D0%B0).

**Основной плюс:**

* Экономия памяти — требуется бит для хранения информации, а не байт.

**Сфера использования** в основном в интерфейсах, где приоритет отдаётся экономии памяти:

* Реализации криптографических алгоритмов
* выбор битов из [слова](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D0%BE%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%BE) состояния, идущие с [пакетом](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(%D1%81%D0%B5%D1%82%D0%B5%D0%B2%D1%8B%D0%B5_%D1%82%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D0%B8)) по интерфейсу, например, [контрольная сумма](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%83%D0%BC%D0%BC%D0%B0);
* выбор битов из [IP-адреса](https://ru.wikipedia.org/wiki/IP-%D0%B0%D0%B4%D1%80%D0%B5%D1%81) для адресации подсети 255.255.224.0
* Реализации любых других алгоритмов требующих работу с битами
* Чтобы пользователи языка не изобретали велосипеды

## Битовые операторы (операции с битовыми масками)

В противоположность большинству языков, С поддерживает все существующие битовые операторы. Поскольку С создавался, чтобы заменить ассемблер, то была необходимость поддержки всех (или по крайней мере большинства) операции, которые может выполнить ассемблер. Битовые операции — это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка С char и int. Битовые операторы не могут использоваться с float, double, long double, void и другими сложными типами. Таблица содержит имеющиеся операторы.

| **Оператор** | **Действие** |
| --- | --- |
| & | И |
| | | ИЛИ |
| ^ | Исключающее ИЛИ |
| ~ | Дополнение |
| >> | Сдвиг вправо |
| << | Сдвиг влево |

Логические операции И, ИЛИ, исключающее ИЛИ и НЕ могут быть описаны с помощью таблиц истинности

### Побитовое И (оператор &)

В побитовых (bit-wise) операциях значение бита, равное 1, рассматривается как логическая истина, а 0 как ложь. **Побитовое И (оператор &)** берёт два числа и логически умножает соответствующие биты.

Побитовое И — это что-то вроде операции, противоположной побитовому ИЛИ. Двоичный разряд результата равен 1 только тогда, когда оба соответствующих бита операндов равны 1. Другими словами, можно сказать, двоичные разряды получившегося числа — это результат умножения соответствующих битов операнда: 1х1 = 1, 1х0 = 0. Побитовому И соответствует следующая таблица истинности:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A & B** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Пример работы побитового И на выражении 38 & 53:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| **B** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| **A & B** | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

Как результат, получаем 1001002 , или 3610 .

С помощью побитового оператора И можно проверить, является ли число четным или нечетным. Для целых чисел, если младший бит равен 1, то число нечетное (основываясь на преобразовании двоичных чисел в десятичные). Зачем это нужно, если можно просто использовать %2? На моем компьютере, например, &1 выполняется на 66% быстрее. Довольно неплохое повышение производительности.

Например, если логически умножить 3 на 8, то получим 0

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

char a = 3;

char b = 8;

char c = a & b;

printf("%d", c);

cin >> a;

}



Так как в двоичном виде 3 в виде однобайтного целого представляет собой 00000011, а 8 - 00001000

Первый бит переменной c равен логическому произведению первого бита числа a и первого бита числа b. И так для каждого бита.

00000011

00001000

--------------

00000000

Соответственно, побитовое произведение чисел 31 и 17 даст 17, так как 31 это 00011111, а 17 это 00010001

00011111

00010001

--------------

00010001

Побитовое произведение чисел 35 и 15 равно 3.

00100011

00001111

--------------

00000011

Битовое И чаще всего используется для выключения битов. То есть любой бит, установленный в 0, вызывает установку соответствующего бита в другом операнде также в 0. Например, следующая функция читает символы из порта модема, используя функцию read\_modem(), и сбрасывает бит четности в 0.

char get\_char\_from\_modem(void)

{

char ch;

ch = read\_modem (); /\* получение символа из порта модема \*/

return (ch & 127);

}

### Побитовое ИЛИ (оператор |)

Аналогично работает **операция побитового ИЛИ (оператор |)**, за исключением того, что она логически суммирует соответствующие биты чисел без переноса.

Побитовое ИЛИ действует эквивалентно логическому ИЛИ, но примененному к каждой паре битов двоичного числа. Двоичный разряд результата равен 0 только тогда, когда оба соответствующих бита в равны 0. Во всех других случаях двоичный результат равен 1. То есть, если у нас есть следующая таблица истинности:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A | B** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

38 | 53 будет таким:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| **B** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| **A | B** | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |

В итоге мы получаем 1101112 , или 5510 .

Например:

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

char a = 15;

char b = 11;

char c = a | b;

printf("%d", c);

cin >> a;

}



выведет 15, так как 15 это 00001111, а 11 это 00001011

00001111

00001011

--------------

00001111

Побитовое ИЛИ для чисел 33 и 11 вернёт 43, так как 33 это 00100001, а 11 это 00001011

00100001

00001011

--------------

00101011

### Побитовое отрицание (оператор ~)

**Побитовое отрицание (оператор ~)** работает не для отдельного бита, а для всего числа целиком. Оператор инверсии меняет ложь на истину, а истину на ложь, для каждого бита.

Побитовое отрицание инвертирует все биты операнда. То есть, то что было 1 станет 0, и наоборот.

|  |  |
| --- | --- |
| **A** | **~A** |
| 0 | 1 |
| 1 | 0 |

Вот, например, операция ~52:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| **~A** | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |

Результатом будет 20310

При использовании побитового отрицания знак результата всегда будет противоположен знаку исходного числа (при работе со знаковыми числами). Почему так происходит, узнаете прямо сейчас.

Например

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

char a = 65;

char b = ~a;

printf("%d", b);

cin >> a;

}



Выведет -66, так как 65 это 01000001, а инверсия даст 10111110, что равно -66. Кстати, вот алгоритм для того, чтобы сделать число отрицательным: для нахождение дополнительного кода числа его надо инвертировать и прибавить к нему единицу.

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

char a = 107;

char b = ~a + 1;

printf("a = %d, -a = %d", a, b);

cin >> a;

}



### Исключающее ИЛИ (оператор ^)

**Исключающее ИЛИ (оператор ^)** применяет побитовую операцию XOR.

Разница между исключающим ИЛИ и побитовым ИЛИ в том, что для получения 1 только один бит в паре может быть 1:

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **A** ^ **B** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Например, выражение 138^43 будет равно

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| **B** | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| **A ^ B** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

101000012 , или 16010

Например, для чисел

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

char a = 12;

char b = 85;

char c = a ^ b;

printf("%d", c);

cin >> a;

}



будет выведено 89, так как a равно 00001100, а b равно 01010101. В итоге получим 01011001

**Иногда логические операторы && и || путают с операторами & и |. Такие ошибки могут существовать в коде достаточно долго, потому что такой код в ряде случаев будет работать. Например, для чисел 1 и 0. Но так как в си истиной является любое ненулевое значение, то побитовое умножение чисел 3 и 4 вернёт 0, хотя логическое умножение должно вернуть истину.**

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

int a = 3;

int b = 4;

printf("a & b = %d\n", a & b); //выведет 0

printf("a && b = %d\n", a && b);//выведет не 0 (конкретнее, 1)

cin >> a;

}



С помощью ^ можно поменять значения двух переменных (имеющих одинаковый тип данных) без использования временной переменной.

Также с помощью исключающего ИЛИ можно зашифровать текст. Для этого нужно лишь итерировать через все символы, и ^ их с символом-ключом. Для более сложного шифра можно использовать строку символов:

string msg = "This is a message";

char[] message = msg.toCharArray();

string key = ".\*)";

string encryptedString = new string();

for(int i = 0; i< message.length; i++){

encryptedString += message[i] ^ key.toCharArray()[i%key.length()];

}

Исключающее ИЛИ не самый надежный способ шифровки, но его можно сделать частью шифровального алгоритма.

## Дополнительный код

Здесь стоит рассказать немного о способе представления отрицательных целых чисел в ЭВМ, а именно о дополнительном коде (two’s complement). Не вдаваясь в подробности, он нужен для облегчения арифметики двоичных чисел.

**Главное, что вам нужно знать о числах, записанных в дополнительном коде — это то, что старший разряд является знаковым. Если он равен 0, то число положительное и совпадает с представлением этого числа в прямом коде, а если 1 — то оно отрицательное. То есть, 10111101 — отрицательное число, а 01000011 — положительное.**

Чтобы преобразовать отрицательное число в дополнительный код, нужно инвертировать все биты числа (то есть, по сути, использовать побитовое отрицание) и добавить к результату 1.

Например, если мы имеем 109:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **A** | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| **~A** | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| **~A+1** | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

Представленным выше методом мы получаем -109 в дополнительном коде.

## Операции побитового сдвига

Операций сдвига две – битовый сдвиг влево (оператор <<) и битовый сдвиг вправо (оператор >>). Битовый сдвиг вправо сдвигает биты числа вправо, дописывая слева нули. Битовый сдвиг влево делает противоположное: сдвигает биты влево, дописывая справа нули. Вышедшие за пределы числа биты отбрасываются.

Например, сдвиг числа 5 влево на 2 позиции

00000101 << 2 == 00010100

Сдвиг числа 19 вправо на 3 позиции

00010011 >> 3 == 00000010

Независимо от архитектуры (big-endian, или little-endian, или middle-endian) числа в двоичном виде представляются слева направо, от более значащего бита к менее значащему. Побитовый сдвиг принимает два операнда – число, над которым необходимо произвести сдвиг, и число бит, на которое необходимо произвести сдвиг.

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

int a = 12;

printf("%d << 1 == %d\n", a, a << 1);

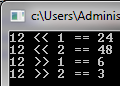
printf("%d << 2 == %d\n", a, a << 2);

printf("%d >> 1 == %d\n", a, a >> 1);

printf("%d >> 2 == %d\n", a, a >> 2);

cin >> a;

}



Так как сдвиг вправо (>>) дописывает слева нули, то для целых чисел операция равносильна целочисленному делению пополам, а сдвиг влево умножению на 2. Произвести битовый сдвиг для числа с плавающей точкой без явного приведения типа нельзя. Это вызвано тем, что для си не определено представление числа с плавающей точкой. Однако можно переместить число типа float в int, затем сдвинуть и вернуть обратно

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

float b = 10.0f;

float c = (float) (\*((unsigned int\*)&b) >> 2);

printf("%.3f >> 2 = %.3f", b, c);

cin >> b;

}



Но мы, конечно же, получим не 5.0f, а совершенно другое число.

Особенностью операторов сдвига является то, что они могут по-разному вести себя с числами со знаком и без знака, в зависимости от компилятора. Действительно, отрицательное число обычно содержит один бит знака. Когда мы будем производить сдвиг влево, он может пропасть, число станет положительным. Однако, компилятор может сделать так, что сдвиг останется знакопостоянным и будет проходить по другим правилам. То же самое и для сдвига вправо.

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

unsigned int ua = 12;

signed int sa = -11;

printf("ua = %d, ua >> 2 = %d\n", ua, ua >> 2);

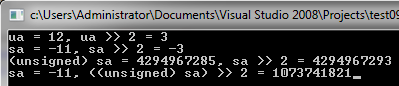
printf("sa = %d, sa >> 2 = %d\n", sa, sa >> 2);

printf("(unsigned) sa = %u, sa >> 2 = %u\n", sa, sa >> 2);

printf("sa = %d, ((unsigned) sa) >> 2 = %d", sa, ((unsigned) sa) >> 2);

cin >> ua;

}



В данном случае при первом сдвиге всё работает, как и задумано, потому что число без знака. Во втором случае компилятор VSE2013 оставляет знак. Однако если посмотреть на представление этого числа, как беззнакового, сдвиг происходит по другим правилам, с сохранением самого левого бита. В последней строчке, если привести число со знаком к числу без знака, то произойдёт обычный сдвиг, и мы получим в результате положительное число.

Побитовые операторы и операторы сдвига не изменяют значения числа, возвращая новое. Они также как и арифметические операторы, могут входить в состав сложного присваивания

#include <iostream>

using namespace std;

void main()

{

setlocale(LC\_CTYPE, "Russian");

int a = 10;

int b = 1;

cout << "a=" << a << " b=" << b << "\n";

a >>= 3;

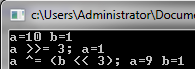
cout << "a >>= 3; a=" << a << "\n";

a ^= (b << 3);

cout << "a ^= (b << 3); a=" << a << " b=" << b << "\n";

cin >> a;

}



## Применение поразрядных операций

### Степень числа 2

Одной из возможностей применения битовых операций является получение степени числа 2. Для этого используются операции побитового сдвига влево или в право. Дело в том что степени числа 2 в двоичном представлении выглядят так:

|  |  |
| --- | --- |
| Десятичные | Двоичные |
| 1 | 0001 |
| 2 | 0010 |
| 4 | 0100 |
| 8 | 1000 |

Как можно легко заметить, единица просто сдвигается влево от правого края, каждый шаг соответствует следующей степени числа 2.

Таким образом что бы получить к примеру 2 в 4 степени, нужно просто взять единицу и применить к ней операцию битового сдвига влево на 3 позиции.

И так операция 0001 << 3 даст нам в итоге 1000 что в десятичной форме равно числу 8.

Казалось бы для чего это требуется, когда например можно просто в цикле умножить число два сколько угодно раз.

Дело в том что операции в цикле займет много тактов процессора, в то время как описанная операция выше, будет выполнена в большинстве процессоров всего за 1 такт.

В итоге получается увеличение производительности и не потребуется писать целый цикл.

### Получение значения отдельных бит

Зачастую требуется узнать значение отдельного бита в некоем числе, то есть 0 там или 1.

Ситуаций, в которых это может потребоваться достаточно много, особенно в низкоуровневом программировании.

Нам потребуется вот такая операция: **new = target & ( 1 < offset)** target - это число в котором мы желаем проверить некий разряд на наличие 0 или 1; offset - количество разрядов на которое требуется сдвинуть единицу от правого края влево, помните это значение будет всегда на 1 меньше реального номера интересующего вас разряда. Так что бы установить единицу в третий разряд справа вам требуется произвести сдвиг на 2 разряда а не на 3, так как единица уже стоит в 1 разряде а значит сдвиг на 1 разряд переместит ее во 2-й разряд.

В результате этой операции вам будет возвращено либо некое число либо ноль. Если данная операция вернет ноль, значит в данном разряде был ноль, если вернет любое другое отличное от нуля число, значит в данном разряде стояла единица.

**Пример:**

Предположим мы имеем двоичное число 1101 и желаем узнать значение 3 бита справа, операция бы выглядела так: 1101 & ( 0001 << 2) После выполнения операции в скобках у нас получится следующие: 1101 & 0100 которая в свою очередь вернет нам двоичное число 0100, так как операция И оставит единицы лишь там где единицы в обеих числах. Данное двоичное число соответствует десятичному числу 4, которое по сути не имеет никакого отношения к нашей задаче, узнать что находится в нужной битовой позиции. Но дело в том что нам не важно какое число получится, важно только факт будет это число нулем или нет. Следовательно если мы получаем число отличное от нуля, значит в той позиции была единица, если число равно нулю значит в той позиции был ноль. Так как выполни мы к примеру такую операцию: 1101 & 0010 на выходе мы бы получили 0000 так как не было бы ни одного разряда где в обоих числах были бы единицы.

### Установка единицы в отдельный бит

Для установки единицы в отельный разряд числа нам потребуется следующая операция:

new = target | ( 1 << offset )

Назначения полей идентичных, описанным в абзаце о получении значения отдельных бит.

Данная операция вернет нам новое значение target в котором в нужный разряд будет установлена 1.

**Пример:**

Предположим мы имеем число 1001 и желаем в 3-й разряд справа установить в единицу, операция бы выглядела так: 1001 | ( 0001 << 2)

После выполнения операции в скобках у нас получится 1001 | 0100, далее операция ИЛИ вернет нам значение 1101 что нам и требовалось.

### Установка нуля в отдельный бит

Для установки нуля в отдельный разряд числа нам потребуется такая операция:

new = target & ( ~ ( 1 << offset) )

либо:

new = target ^ ( 1 << offset )

Назначения полей идентичные, описанным в абзаце о получении значения отдельных бит. Данная операция вернет нам новое значение target в котором в нужный разряд будет установлена 0.

**Пример:**

Предположим мы имеем число 1101 и желаем в 3-м разряде установить ноль, операция бы выглядела так: 1101 & ( ~ ( 0001 << 2) )

После первого действия в скобках наша операция примет вид 1101 & ( ~0100), после чего операции инвертирования в скобках превратит наше число в 1011 и операция примет вид 1101 & 1011, затем операция И оставит единицы лишь в тех разрядах, где были единицы у обоих чисел, вернув нам число 1001.

Даже зная, как получать биты и устанавливать, для многих все равно остается вопросом для чего все это нужно. На самом деле, если бы например вы решили управлять простым текстовым LCD экраном без использования готовой библиотеки а формируя требуемые сигналы вы бы в долю секунды поняли зачем вам все эти операции. Но вот если вы работает на более высоком уровне, полезность данных операций не так очевидна. Ниже приведем пару общих примеров как такие операции можно применять и в более высокоуровневом программировании.

### Битовый массив

Зачастую в программах требуется хранить некие флаги состояний, которые как правило принимают лишь два значения, например горит светодиод или не горит.

Вариантов как это реализовать и без битовых операций полно, но раз мы говорим о них посмотрим как они могут помочь. Идея крайне простая, давайте предположим, что у нас есть 4 диода, которые мы можем включать и выключать и для каких то целей нам требуется всегда знать какие из них горят а какие нет.

Таким образом мы просто берем одну числовую переменную которая будет выполнять роль битового массива, а каждый бит будет флагом состояния конкретного светодиода.

То есть к примеру если 0 то диод выключен если 1 то диод включен.

Как установить требуемые значения и получить эти значения вы уже знаете. У этого подхода есть один не совсем очевидный плюс, то есть узнать о том горит хотя бы один диод или все выключены можно просто сравнив такую переменную состояния с нулем, так как если все диоды выключены то все биты числа будут нулями и как следствие число будет ноль.

Таким образом данная переменная у вас всегда является неким числом, которое своего рода показывает вам "срез" состояния ваших диодов, что может быть удобно в неких логических операциях.

### Передача параметров в функции

Пожалуй в высокоуровневом программировании это наиболее частое применение битовых операций.

Если вы касались работы с API различных операционных систем то могли обнаружить весьма странные на первый взгляд операции:

someFunction ( PARAM\_1 | PARAM\_2 | PARAM\_3 );

На самом деле тут передается лишь один параметр в функцию, просто битовая операция ИЛИ создает некий срез параметров, как это было со светодиодами, то есть единицы в определенных битах говорят о том требуется эта опция или не требуется.

Представьте что мы хотим написать функцию которая бы могла разом включать любое количество диодов от 1 до 4 и желаем зажечь например 1-й и 3-й диоды, вызов функции мог бы выгладить например так.

switchOnLeds ( LED\_1 | LED\_3 );

Прежде чем передать значения в функцию операция ИЛИ сделала бы например вот так 0001 | 0100 что дало бы нам 0101, именно это значение бы поступило в функцию.

Дальше думаю уже все понятно, функция бы просмотрела все биты полученного числа и зажгла нужные диоды.

Не очевидных плюс данного метода стоит в том, что вы можете в функцию которая принимает в качестве одного аргумента простое целое число передать много разных параметров в этом аргументе.

И зажечь хоть один диод:

switchOnLeds ( LED\_1 );

Хоть все 4:

switchOnLeds( LED\_1 | LED\_2 | LED\_3 | LED\_4 );

Или еще хитрее:

switchOnLeds ( ALL\_LEDS );

Где ALL\_LEDS будет равно двоичному числу 1111.

Описанные выше примеры являются лишь одним из множества вариантов достижения цели, таким образом они не идеальны и не универсальны, именно конкретная задача диктует приемлемость того или иного подхода, в одной ситуации битовые операции будут незаменимы, в другой не дав никакого выигрыша ухудшат читаемость кода или сделают программу еще более громоздкой.

**Помните, не средства решают какую выбрать задачу а именно задача решает какие выбрать средства.**

## Примеры

### Примеры для языка C

***1. Напишем функции, которые позволяют определять и изменять определённый бит числа***

Для того, чтобы узнать, какой бит (1 или 0) стоит на позиции n, воспользуемся логическим умножением.

Пусть имеется число 9 (00001001). Нужно узнать, выставлен ли бит на позиции 3 (начиная с нуля). Для этого умножим его на число, у которого все биты равны нулю, кроме третьего:

00001001 & 00001000 = 00001000

Теперь узнаем значение бита в позиции 6

00001001 & 01000000 = 00000000

Таким образом, если мы получаем ответ, равный нулю, то на искомой позиции находится ноль, иначе единица. Чтобы получить число, состоящее из нулей с одним битом на нужной позиции, сдвинем 1 на нужное число бит влево.

#include <stdio.h>

#include <conio.h>

#include <limits.h>

int checkbit(const int value, const int position) {

int result;

if ((value & (1 << position)) == 0) {

result = 0;

} else {

result = 1;

}

return result;

}

void main() {

int a = 3;

size\_t len = sizeof(int) \* CHAR\_BIT;

size\_t i;

for (i = 0; i < len; i++) {

printf("%d", checkbit(a, i));

}

\_getch();

}



Заметьте, что в функции условие записано так

if ((value & (1 << position)) == 0)

Потому что без скобок сначала будет вычислено равенство нулю и только потом выполнено умножение.

((value & (1 << position)) == 0)

Функцию можно упростить

int checkbit(const int value, const int position) {

return ((value & (1 << position)) != 0);

}

***Функция, которая выставляет бит на n-й позиции в единицу.***

Известно, что логическое сложение любого бита с 1 будет равно 1. Так что для установки n-го бита нужно логически сложить число с таким, у которого все биты, кроме нужного, равны нулю. Как получить такое число, уже рассмотрено.

int setbit(const int value, const int position) {

return (value | (1 << position));

}

***Функция, которая устанавливает бит на n-й позиции в ноль.***

Для этого нужно, чтобы все биты числа, кроме n-го, не изменились. Умножим число на такое, у которого все биты равны единице, кроме бита под номером n. Например

0001011 & 1110111 = 0000011

Чтобы получить такую маску, сначала создадим число с нулями и одной единицей, а потом инвертируем его.

int unsetbit(const int value, const int position) {

return (value & ~(1 << position));

}

***Функция, изменющая значение n-го бита на противоположное.***

Для этого воспользуемся функцией исключающего или: применим операцию XOR к числу, которое состоит из одних нулей и одной единицы на месте нужного бита.

int switchbit(const int value, const int position) {

return (value ^ (1 << position));

}

Проверка

#include <stdio.h>

#include <conio.h>

#include <limits.h>

int checkbit(const int value, const int position) {

return ((value & (1 << position)) != 0);

}

int setbit(const int value, const int position) {

return (value | (1 << position));

}

int unsetbit(const int value, const int position) {

return (value & ~(1 << position));

}

int switchbit(const int value, const int position) {

return (value ^ (1 << position));

}

void printbits(int n) {

//CHAR\_BIT опеределён в библиотеке limits.h

//и хранит число бит в байте для данной платформы

size\_t len = sizeof(int)\* CHAR\_BIT;

size\_t i;

for (i = 0; i < len; i++) {

printf("%d", checkbit(n, i));

}

printf("\n");

}

void main() {

int a = 3;

size\_t len = sizeof(int) \* CHAR\_BIT;

size\_t i;

printbits(a);

a = setbit(a, 5);

printbits(a);

a = unsetbit(a, 5);

printbits(a);

a = switchbit(a, 11);

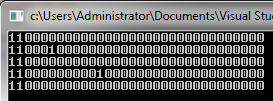
printbits(a);

a = switchbit(a, 11);

printbits(a);

\_getch();

}



### Примеры для языка CPP

Сдвиги влево

shift-expression << additive-expression

Оператор сдвига влево вызывает сдвиг битов shift-expression влево на количество позиций, определенных с помощью additive-expression. Позиции битов, освобожденные при операции сдвига, заполняются нулями. Сдвиг влево является логическим сдвигом (биты, сдвигаемые с конца отбрасываются, включая бит знака).

В следующем примере показаны операции сдвига влево с использованием чисел без знака. В этом примере показано, что происходит с битами при представлении значения как bitset.

#include <iostream>

#include <bitset>

using namespace std;

int main() {

unsigned short short1 = 4;

bitset<16> bitset1{ short1 }; // the bitset representation of 4

cout << bitset1 << endl; // 0000000000000100

unsigned short short2 = short1 << 1; // 4 left-shifted by 1 = 8

bitset<16> bitset2{ short2 };

cout << bitset2 << endl; // 0000000000001000

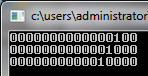
unsigned short short3 = short1 << 2; // 4 left-shifted by 2 = 16

bitset<16> bitset3{ short3 };

cout << bitset3 << endl; // 0000000000010000

cin >> short1;

}



Если выполняется сдвиг влево числа со знаком и при этом затрагивается бит знака, результат не определен. В следующем примере показано, что происходит в Visual C++, если выполняется сдвиг влево 1 бита в позицию бита знака.

#include <iostream>

#include <bitset>

using namespace std;

int main() {

short short1 = 16384;

bitset<16> bitset1{ short2 };

cout << bitset1 << endl; // 0100000000000000

short short3 = short1 << 1;

bitset<16> bitset3{ short3 }; // 16384 left-shifted by 1 = -32768

cout << bitset3 << endl; // 100000000000000

short short4 = short1 << 14;

bitset<16> bitset4{ short4 }; // 4 left-shifted by 14 = 0

cout << bitset4 << endl; // 000000000000000

cin >> short1;

}

Сдвиги вправо

shift-expression >> additive-expression

Оператор сдвига вправо вызывает сдвиг группы битов shift-expression вправо на количество позиций, определенных с помощью additive-expression. Для чисел без знака позиции битов, освобожденные при операции сдвига, заполняются нулями. Для чисел со знаком бит знака используется для заполнения освобожденных позиций битов. Другими словами, если число является положительным, используется 0, если число является отрицательным, используется 1.

**Замечание**

Результат сдвига вправо отрицательного числа со знаком зависит от реализации. Хотя Visual C++ использует бит знака для заполнения освобожденных позиций битов, нет никакой гарантии, что другие реализации также выполняют это.

В следующем примере показаны операции сдвига вправо с использованием чисел без знака.

#include <iostream>

#include <bitset>

using namespace std;

int main() {

unsigned short short11 = 1024;

bitset<16> bitset11{ short11 };

cout << bitset11 << endl; // 0000010000000000

unsigned short short12 = short11 >> 1; // 512

bitset<16> bitset12{ short12 };

cout << bitset12 << endl; // 0000001000000000

unsigned short short13 = short11 >> 10; // 1

bitset<16> bitset13{ short13 };

cout << bitset13 << endl; // 0000000000000001

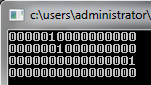
unsigned short short14 = short11 >> 11; // 0

bitset<16> bitset14{ short14 };

cout << bitset14 << endl; // 0000000000000000}

cin >> short11;

}



В следующем примере показаны операции сдвига вправо с использованием положительных чисел со знаком.

#include <iostream>

#include <bitset>

using namespace std;

int main() {

short short1 = 1024;

bitset<16> bitset1{ short1 };

cout << bitset1 << endl; // 0000010000000000

short short2 = short1 >> 1; // 512

bitset<16> bitset2{ short2 };

cout << bitset2 << endl; // 0000001000000000

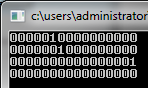
short short3 = short1 >> 11; // 0

bitset<16> bitset3{ short3 };

cout << bitset3 << endl; // 0000000000000000

cin >> short1;

}



В следующем примере показаны операции сдвига вправо с использованием отрицательных целых чисел со знаком.

#include <iostream>

#include <bitset>

using namespace std;

int main() {

short neg1 = -16;

bitset<16> bn1{ neg1 };

cout << bn1 << endl; // 1111111111110000

short neg2 = neg1 >> 1; // -8

bitset<16> bn2{ neg2 };

cout << bn2 << endl; // 1111111111111000

short neg3 = neg1 >> 2; // -4

bitset<16> bn3{ neg3 };

cout << bn3 << endl; // 1111111111111100

short neg4 = neg1 >> 4; // -1

bitset<16> bn4{ neg4 };

cout << bn4 << endl; // 1111111111111111

short neg5 = neg1 >> 5; // -1

bitset<16> bn5{ neg5 };

cout << bn5 << endl; // 1111111111111111

cin >> neg1;

}

Сдвиги и перемещения

Выражения с обеих сторон оператора сдвига должны быть целочисленными типами. Восходящие приведения целых типов выполняются в соответствии с правилами, описанным в разделе [Восходящие приведения целых типов](https://msdn.microsoft.com/ru-ru/library/fc9te331.aspx). Тип результата соответствует типу приводимого shift-expression.

В следующем примере у переменной типа char уровень повышается до int.

#include <iostream>

#include <typeinfo>

using namespace std;

int main() {

char char1 = 'a';

auto promoted1 = char1 << 1; // 194

cout << typeid(promoted1).name() << endl; // int

auto promoted2 = char1 << 10; // 99328

cout << typeid(promoted2).name() << endl; // int

cin >> char1;

}



Некоторые подробности

Результат операции сдвига не определен, если additive-expression имеет отрицательное значение или если additive-expression больше или равен количеству битов в повышаемом shift-expression. Операция сдвига не выполняется, если значение additive-expression равно 0

#include <iostream>

#include <bitset>

using namespace std;

int main() {

unsigned int int1 = 4;

bitset<32> b1{ int1 };

cout << b1 << endl; // 00000000000000000000000000000100

unsigned int int2 = int1 << -3; // C4293: '<<' : shift count negative or too big, undefined behavior

unsigned int int3 = int1 >> -3; // C4293: '>>' : shift count negative or too big, undefined behavior

unsigned int int4 = int1 << 32; // C4293: '<<' : shift count negative or too big, undefined behavior

unsigned int int5 = int1 >> 32; // C4293: '>>' : shift count negative or too big, undefined behavior

unsigned int int6 = int1 << 0;

bitset<32> b6{ int6 };

cout << b6 << endl; // 00000000000000000000000000000100 (no change)}

cin >> int1;

}

