

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExr](#)
- [Ассемблер](#)
- [Купить .PDF](#)


## Урок №45. Побитовые операторы

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 11 Сен 2020 |

 88212

[1](#)  [18](#)

**Побитовые операторы манипулируют отдельными битами в пределах переменной.**

**Примечание:** Для некоторых этот материал может показаться сложным. Если вы застряли или что-то не понятно — пропустите этот урок (и следующий), в будущем сможете вернуться и разобраться детально. Он не столь важен для прогресса в изучении языка C++, как другие уроки, и изложен здесь в большей мере для общего развития.

Оглавление:

1. [Зачем нужны побитовые операторы?](#)
2. [Побитовый сдвиг влево \(<<\) и побитовый сдвиг вправо \(>>\)](#)
3. [Что!? Разве операторы << и >> используются не для вывода и ввода данных?](#)
4. [Побитовый оператор НЕ](#)
5. [Побитовые операторы И, ИЛИ и исключающее ИЛИ \(XOR\)](#)
6. [Побитовые операторы присваивания](#)
7. [Заключение](#)
8. [Тест](#)
9. [Ответы](#)

## Зачем нужны побитовые операторы?

В далеком прошлом компьютерной памяти было очень мало и ею сильно дорожили. Это было стимулом максимально разумно использовать каждый доступный бит. Например, в [логическом типе данных bool](#) есть всего лишь два возможных значения (true и false), которые могут быть представлены одним битом, но по факту занимают целый байт памяти! А это, в свою очередь, из-за того, что переменные используют уникальные адреса памяти, а они выделяются только в байтах. Переменная bool занимает 1 бит, а другие 7 бит — тратятся впустую.

Используя побитовые операторы, можно создавать функции, которые позволят уместить 8 значений типа `bool` в переменную размером 1 байт, что значительно экономит потребление памяти. В прошлом такой трюк был очень популярен. Но сегодня, по крайней мере, в прикладном программировании, это не так.

Теперь памяти стало существенно больше и программисты обнаружили, что лучше писать код так, чтобы было проще и понятнее его поддерживать, нежели усложнять его ради незначительной экономии памяти. Поэтому спрос на использование побитовых операторов несколько уменьшился, за исключением случаев, когда необходима уж максимальная оптимизация (например, научные программы, которые используют огромное количество данных; игры, где манипуляции с битами могут быть использованы для дополнительной скорости; встроенные программы, где память по-прежнему ограничена).

**В языке C++ есть 6 побитовых операторов:**

Оператор	Символ	Пример	Операция
Побитовый сдвиг влево	<code>&lt;&lt;</code>	<code>x &lt;&lt; y</code>	Все биты в <code>x</code> сдвигаются влево на <code>y</code> бит
Побитовый сдвиг вправо	<code>&gt;&gt;</code>	<code>x &gt;&gt; y</code>	Все биты в <code>x</code> сдвигаются вправо на <code>y</code> бит
Побитовое НЕ	<code>~</code>	<code>~x</code>	Все биты в <code>x</code> меняются на противоположные
Побитовое И	<code>&amp;</code>	<code>x &amp; y</code>	Каждый бит в <code>x</code> И каждый соответствующий ему бит в <code>y</code>
Побитовое ИЛИ	<code> </code>	<code>x   y</code>	Каждый бит в <code>x</code> ИЛИ каждый соответствующий ему бит в <code>y</code>
Побитовое исключающее ИЛИ (XOR)	<code>^</code>	<code>x ^ y</code>	Каждый бит в <code>x</code> XOR с каждым соответствующим ему битом в <code>y</code>

В побитовых операциях следует использовать только целочисленные типы данных `unsigned`, так как C++ не всегда гарантирует корректную работу побитовых операторов с целочисленными типами `signed`.

**Правило:** При работе с побитовыми операторами используйте целочисленные типы данных `unsigned`.

## Побитовый сдвиг влево (`<<`) и побитовый сдвиг вправо (`>>`)

В языке C++ количество используемых бит основывается на размере типа данных (в 1 байте находятся 8 бит). Оператор побитового сдвига влево (`<<`) сдвигает биты влево. Левый операнд является выражением, в котором они сдвигаются, а правый — количество мест, на которые нужно сдвинуть. Поэтому в выражении `3 << 1` мы имеем в виду «сдвинуть биты влево в литерале 3 на одно место».

**Примечание:** В следующих примерах мы будем работать с 4-битными двоичными значениями.

Рассмотрим число 3, которое в двоичной системе равно 0011:

`3 = 0011`

`3 << 1 = 0110 = 6`

`3 << 2 = 1100 = 12`

`3 << 3 = 1000 = 8`

В последнем третьем случае один бит перемещается за пределы самого литерала! Биты, сдвинутые за пределы двоичного числа, теряются навсегда.

Оператор побитового сдвига вправо (>>) сдвигает биты вправо. Например:

```
12 = 1100
12 >> 1 = 0110 = 6
12 >> 2 = 0011 = 3
12 >> 3 = 0001 = 1
```

В третьем случае мы снова переместили бит за пределы литерала. Он также потерялся навсегда.

Хотя в примерах, приведенных выше, мы смещаем биты только в литералах, мы также можем смещать биты и в переменных:

```
1 unsigned int x = 4;
2 x = x << 1; // x должен стать равным 8
```

Следует помнить, что результаты операций с побитовыми сдвигами в разных компиляторах могут отличаться.

## Что!? Разве операторы << и >> используются не для вывода и ввода данных?

И для этого тоже.

Сейчас польза от использования побитовых операторов не так велика, как это было раньше. Сейчас в большинстве случаев оператор побитового сдвига влево используется для вывода данных. Например, рассмотрим следующую программу:

```
1 #include <iostream>
2
3 int main()
4 {
5     unsigned int x = 4;
6     x = x << 1; // оператор << используется для побитового сдвига влево
7     std::cout << x; // оператор << используется для вывода данных в консоль
8
9     return 0;
10 }
```

Результат выполнения программы:

8

А как компилятор понимает, когда нужно применить оператор побитового сдвига влево, а когда выводить данные? Всё очень просто. [std::cout](#) **переопределяет** значение оператора << по умолчанию на новое (вывод данных в консоль). Когда компилятор видит, что левым операндом оператора << является std::cout, то он понимает, что должен произойти вывод данных. Если левым операндом является переменная целочисленного типа данных, то компилятор понимает, что должен произойти побитовый сдвиг влево (операция по умолчанию).

## Побитовый оператор НЕ

**Побитовый оператор НЕ (~)**, пожалуй, самый простой для объяснения и понимания. Он просто меняет каждый бит на противоположный, например, с 0 на 1 или с 1 на 0. Обратите внимание, результаты побитового НЕ зависят от размера типа данных!

Предположим, что размер типа данных составляет 4 бита:

```
4 = 0100
~ 4 = 1011 (двоичное) = 11 (десятичное)
```

Предположим, что размер типа данных составляет 8 бит:

```
4 = 0000 0100
~ 4 = 1111 1011 (двоичное) = 251 (десятичное)
```

## Побитовые операторы И, ИЛИ и исключающее ИЛИ (XOR)

Побитовые операторы И (&) и ИЛИ (|) работают аналогично [логическим операторам](#) И и ИЛИ. Однако, побитовые операторы применяются к каждому биту отдельно! Например, рассмотрим выражение 5 | 6. В двоичной системе это 0101 | 0110. В любой побитовой операции операнды лучше всего размещать следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
```

А затем применять операцию к каждому столбцу с битами по отдельности. Как вы помните, логическое ИЛИ возвращает true (1), если один из двух или оба операнды истинны (1). Аналогичным образом работает и **побитовое ИЛИ**. Выражение 5 | 6 обрабатывается следующим образом:

```
0 1 0 1 // 5
0 1 1 0 // 6
-----
0 1 1 1 // 7
```

Результат:

```
0111 (двоичное) = 7 (десятичное)
```

Также можно обрабатывать и комплексные выражения ИЛИ, например, 1 | 4 | 6. Если хоть один бит в столбце равен 1, то результат целого столбца — 1. Например:

```
0 0 0 1 // 1
0 1 0 0 // 4
0 1 1 0 // 6
-----
0 1 1 1 // 7
```

Результатом 1 | 4 | 6 является десятичное 7.

**Побитовое И** работает аналогично логическому И — возвращается true, только если оба бита в столбце равны 1. Рассмотрим выражение 5 & 6:

```

0 1 0 1 // 5
0 1 1 0 // 6
-----
0 1 0 0 // 4

```

Также можно решать и комплексные выражения И, например,  $1 \& 3 \& 7$ . Только при условии, что все биты в столбце равны 1, результатом столбца будет 1.

```

0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
-----
0 0 0 1 // 1

```

Последний оператор — **побитовое исключающее ИЛИ (^)** (сокр. «XOR» от англ. «eXclusive OR»). При обработке двух операндов, исключающее ИЛИ возвращает true (1), только если один и только один из операндов является истинным (1). Если таких нет или все операнды равны 1, то результатом будет false (0). Рассмотрим выражение  $6 \wedge 3$ :

```

0 1 1 0 // 6
0 0 1 1 // 3
-----
0 1 0 1 // 5

```

Также можно решать и комплексные выражения XOR, например,  $1 \wedge 3 \wedge 7$ . Если единиц в столбце чётное количество, то результатом будет 0, если же нечётное количество, то результат — 1. Например:

```

0 0 0 1 // 1
0 0 1 1 // 3
0 1 1 1 // 7
-----
0 1 0 1 // 5

```

## Побитовые операторы присваивания

Как и в случае с арифметическими операторами присваивания, язык C++ предоставляет побитовые операторы присваивания для облегчения внесения изменений в переменные.

Оператор	Символ	Пример	Операция
Присваивание с побитовым сдвигом влево	<<=	$x \ll= y$	Сдвигаем биты в x влево на y бит
Присваивание с побитовым сдвигом вправо	>>=	$x \gg= y$	Сдвигаем биты в x вправо на y бит
Присваивание с побитовой операцией ИЛИ	=	$x  = y$	Присваивание результата выражения $x   y$ переменной x
Присваивание с побитовой операцией И	&=	$x \&= y$	Присваивание результата выражения $x \& y$ переменной x
Присваивание с побитовой операцией исключающего ИЛИ	^=	$x \wedge= y$	Присваивание результата выражения $x \wedge y$ переменной x

Например, вместо  $x = x \ll 1$ ; мы можем написать  $x \ll= 1$ ;

## Заключение

При работе с побитовыми операторами (используя метод столбца) не забывайте о том, что:

- При вычислении побитового ИЛИ, если хоть один из битов в столбце равен 1, то результат целого столбца равен 1.
- При вычислении побитового И, если все биты в столбце равны 1, то результат целого столбца равен 1.
- При вычислении побитового исключающего ИЛИ (XOR), если единиц в столбце нечётное количество, то результат равен 1.

## Тест

### Задание №1

Какой результат  $0110 \gg 2$  в двоичной системе счисления?

### Задание №2

Какой результат  $5 \mid 12$  в десятичной системе счисления?

### Задание №3

Какой результат  $5 \& 12$  в десятичной системе счисления?

### Задание №4

Какой результат  $5 \wedge 12$  в десятичной системе счисления?

## Ответы

### Ответ №1

Результатом  $0110 \gg 2$  является двоичное число  $0001$ .

### Ответ №2

Выражение  $5 \mid 12$ :

```
0 1 0 1
1 1 0 0
-----
1 1 0 1 // 13 (десятичное)
```

### Ответ №3

Выражение  $5 \& 12$ :

```
0 1 0 1
1 1 0 0
-----
```

0 1 0 0 // 4 (десятичное)

#### Ответ №4

Выражение  $5 \wedge 12$ :

```
0 1 0 1
1 1 0 0
-----
```

1 0 0 1 // 9 (десятичное)

Оценить статью:



(328 оценок, среднее: 4,95 из 5)



[← Урок №44. Конвертация чисел из двоичной системы в десятичную и наоборот](#)

[Урок №46. Битовые флаги и битовые маски](#)



## Комментариев: 18



1. *Наталья:*

[26 сентября 2020 в 11:56](#)

Результат сдвига вправо зависит от того, какая у нас переменная — знаковая или беззнаковая. Но в разных компиляторах по-разному быть не должно, поведение операций сдвига фиксировано.

[Ответить](#)



2. *Дмитрий:*

[31 июля 2020 в 14:52](#)

Автор — ТОП! Я таки нашел что искал, хоть с спустя месяц и после того как вспомнил и разобрался XD... будь оно неладно — "исключающее или". Кстати поисковик мне тогда ответ не дал.

[Ответить](#)



3. *Борис:*

[28 апреля 2020 в 17:05](#)

Одно непонятное предложение:

"Следует помнить, что результаты операций с побитовыми сдвигами в разных компиляторах могут отличаться."

Вот что тут имелось в виду? Побитовые операции всегда выполняются строго заданным одинаковым образом. Насколько я знаю, в соседние байты процессор никогда не лезет, поэтому разряды, ушедшие за края при сдвиге, просто пропадают, не влияя на соседние байты. А возникшие новые разряды заполняются нулями. Неужели на каких-то платформах это не так? И это ли имелось в виду вообще?

[Ответить](#)



4. *Сергей:*

[16 марта 2020 в 00:49](#)

"Используя побитовые операторы, можно создавать функции, которые позволят уместить 8 значений типа bool в переменной размером 1 байт, что значительно сэкономит потребление памяти." Никогда раньше не задумывался об этом. Спасибо. Да уж минули времена когда экономили память. Но говорят, что настают времена когда уже не хватает железа с учетом разросшихся аппетитов программ.

[Ответить](#)



1. *Владислав:*

[15 апреля 2020 в 20:11](#)

Отвечу вам то же что говорю студентам своим. Если вы работаете на enterprise проекте или пишете программы на современный ПК, то занимайтесь экономией памяти за счёт очистки кучи от мусора после себя а не жмотьтесь на лишний байт. И если (тут прям ключевое слово если), а не когда вам придёт задача оптимизировать год по памяти тогда подключайте битовые поля беззнаковые типы с побитовыми операциями и вот это вот всё.

Если вы работаете с микроконтроллерами типа ардуинок или z80 (интернет вещей, автоматизация,...), то я вас поздравляю и этим придётся заниматься чуть не со старта проекта.

[Ответить](#)



5. *Артемий:*

[7 февраля 2020 в 09:44](#)

Отлично, доступно к пониманию

[Ответить](#)



6. *Мадияр:*

[5 января 2020 в 16:50](#)

Юрий , красавчик !

Все предельно ясно объяснил. Давно не понимал эту тему , на двух контекстах встретил и не смог решить.

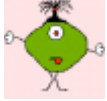


[Ответить](#)

1. Юрий:

[5 января 2020 в 17:23](#)

Спасибо) На третьем констесте, значит, должен уже решить 😊

[Ответить](#)

7. Максим:

[31 декабря 2019 в 04:37](#)

Небольшой вопрос, а как узнать какое количество памяти используется и какое количество операций совершает процессор.

Мне стало интресно что эффективнее:

```
1 int Num = 4;
2
3 std::cout << Num * 2 << std::endl;
4 // или
5 std::cout << ( Num << 1 ) << std::endl;
```

[Ответить](#)

1. Сергей:

[15 февраля 2020 в 23:58](#)

На ранних процессорах (i8086) операция умножения (например, `Num * 2`) выполнялась от 124 тактов (при частоте процессора 4,77 МГц).

А битовые операции сдвига на 1 бит (например, `Num << 1`) всего 12 тактов.

Поэтому битовые операции во много раз быстрее.

Современные процессоры благодаря архитектуре конвейеров выполняют обе операции за доли такта (т.е. выполняются одновременно), однако битовые операции значительно разгружают конвейер процессора. Именно поэтому заполнение процессора выполняется гораздо оптимальнее (это увеличивает возможности предсказания команд).

[Ответить](#)

8. Вячеслав:

[5 февраля 2019 в 20:55](#)

попробовал сделать так:

```
1 #include "pch.h"
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 { //объявляем unsigned целочисленную переменную x и присваиваем ей значение 4
7     unsigned int x = 4;
```

```

8   x = x << 1; //в этом случае это побитовый сдвиг влево на 1
9   cout << x << "\n"; //а это оператор вывода в консоль ответ 8
10  x = x >> 1; //это оператор побитового сдвига вправо
11  cout << x; //в консоль выведет 4
12
13  return 0;
14 }
```

[Ответить](#)

1. Миша))0):

[25 апреля 2019 в 15:47](#)

Вячеслав, вместо

```
1 cout << x << "\n";
```

правильнее писать

```
1 cout << x << '\n';
```

т.к. \n это один символ, а у строки в конце есть "невидимый" символ \0 который говорит, что строка закончилась и ваш вариант занимает больше оперативной памяти на ЦЕЛЫЙ байт)). А ещё правильнее писать

```
1 cout << x << endl;
```

[Ответить](#)

9. Сергей:

[24 января 2019 в 09:39](#)

Спасибо за статью.

Я бы разделил тему "Побитовые И, ИЛИ и исключающее ИЛИ (XOR)" на 3 отдельные темы. Так их будет легче найти в тексте при быстром беглом поиске.

[Ответить](#)

10. Mikhail:

[13 сентября 2018 в 20:10](#)

Действительно, в крайней степени понятный материал, спасибо большое

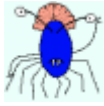
[Ответить](#)

1. Юрий:

[13 сентября 2018 в 20:45](#)

Пожалуйста 😊 Читайте.

[Ответить](#)



1. *Владимир:*

[9 февраля 2019 в 13:53](#)

Юрий скажите пожалуйста!

Если у нас есть число большое например  $(23654375 \gg 7) \& 11$

Мы первым делом производим сдвиг же, а потом сравниваем, и если совпадает хоть один, его записываем в ответ. Или неправильно я понимаю? С Уважением.

[Ответить](#)



11. *OrdinaryMind:*

[25 июня 2018 в 18:15](#)

Благодарю за статью. Необходимо было быстро освежить в памяти побитовые операции. Все написано коротко и по делу.

[Ответить](#)



1. *Юрий:*

[25 июня 2018 в 20:40](#)

Обращайтесь 😊

[Ответить](#)

## Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены \*

Имя \*

Email \*

Комментарий

☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию






☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

TELEGRAM  КАНАЛ



ПАБЛИК 

## ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «Same Game»](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020