

Ravesli [Ravesli](#)


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


## Урок №46. Битовые флаги и битовые маски

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 11 Сен 2020 |

 55182

[↑](#)  30

На этом уроке мы рассмотрим битовые флаги и битовые маски в языке C++.

**Примечание:** Для некоторых этот материал может показаться немного сложным. Если вы застряли или что-то не понятно — пропустите этот урок, 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. [Битовые флаги](#)
2. [Почему битовые флаги полезны?](#)
3. [Введение в std::bitset](#)
4. [Битовые маски](#)
5. [Пример с RGBA](#)
6. [Заключение](#)
7. [Тест](#)
8. [Ответы](#)

### Битовые флаги

Используя целый байт для хранения значения **логического типа данных**, вы занимаете только 1 бит, а остальные 7 из 8 — не используются. Хотя в целом это нормально, но в особых, ресурсоёмких случаях, связанных с множеством логических значений, может быть полезно «упаковать» 8 значений типа bool в 1 байт, сэкономив при этом память и увеличив, таким образом, производительность. Эти отдельные биты и называются **битовыми флагами**. Поскольку доступ к этим битам напрямую отсутствует, то для операций с ними используются **побитовые операторы**.

**Примечание:** На этом уроке мы будем использовать значения из шестнадцатеричной системы счисления.

Например:

```
1 // Определяем 8 отдельных битовых флагов (они могут представлять всё, что вы захотите)
2 // Обратите внимание, в C++11 лучше использовать "uint8_t" вместо "unsigned char"
3 const unsigned char option1 = 0x01; // шестнадцатеричный литерал для 0000 0001
4 const unsigned char option2 = 0x02; // шестнадцатеричный литерал для 0000 0010
5 const unsigned char option3 = 0x04; // шестнадцатеричный литерал для 0000 0100
6 const unsigned char option4 = 0x08; // шестнадцатеричный литерал для 0000 1000
7 const unsigned char option5 = 0x10; // шестнадцатеричный литерал для 0001 0000
8 const unsigned char option6 = 0x20; // шестнадцатеричный литерал для 0010 0000
9 const unsigned char option7 = 0x40; // шестнадцатеричный литерал для 0100 0000
10 const unsigned char option8 = 0x80; // шестнадцатеричный литерал для 1000 0000
11
12 // Байтовое значения для хранения комбинаций из 8 возможных вариантов
13 unsigned char myflags = 0; // все флаги/параметры отключены до старта
```

Чтобы узнать битовое состояние, используется побитовое И:

```
1 if (myflags & option4) ... // если option4 установлено - что-нибудь делаем
```

Чтобы включить биты, используется побитовое ИЛИ:

```
1 myflags |= option4; // включаем option4
2 myflags |= (option4 | option5); // включаем option4 и option5
```

Чтобы выключить биты, используется побитовое И с инвертированным литералом:

```
1 myflags &= ~option4; // выключаем option4
2 myflags &= ~(option4 | option5); // выключаем option4 и option5
```

Для переключения между состояниями бит, используется побитовое исключающее ИЛИ (XOR):

```
1 myflags ^= option4; // включаем или выключаем option4
2 myflags ^= (option4 | option5); // изменяем состояния option4 и option5
```

В качестве примера возьмем библиотеку 3D-графики [OpenGL](#), в которой некоторые функции принимают один или несколько битовых флагов в качестве параметров:

```
1 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // очищаем буфер цвета и глубины
```

GL\_COLOR\_BUFFER\_BIT и GL\_DEPTH\_BUFFER\_BIT определяются следующим образом (в gl2.h):

```
1 #define GL_DEPTH_BUFFER_BIT          0x00000100
2 #define GL_STENCIL_BUFFER_BIT        0x00000400
3 #define GL_COLOR_BUFFER_BIT          0x00004000
```

Вот небольшой пример:

```
1 #include <iostream>
2
```

```
3 int main()
4 {
5     // Определяем набор физических/эмоциональных состояний
6     const unsigned char isHungry = 0x01; // шестнадцатеричный литерал для 0000 0001
7     const unsigned char isSad = 0x02; // шестнадцатеричный литерал для 0000 0010
8     const unsigned char isMad = 0x04; // шестнадцатеричный литерал для 0000 0100
9     const unsigned char isHappy = 0x08; // шестнадцатеричный литерал для 0000 1000
10    const unsigned char isLaughing = 0x10; // шестнадцатеричный литерал для 0001 0000
11    const unsigned char isAsleep = 0x20; // шестнадцатеричный литерал для 0010 0000
12    const unsigned char isDead = 0x40; // шестнадцатеричный литерал для 0100 0000
13    const unsigned char isCrying = 0x80; // шестнадцатеричный литерал для 1000 0000
14
15    unsigned char me = 0; // все флаги/параметры отключены до старта
16    me |= isHappy | isLaughing; // я isHappy и isLaughing
17    me &= ~isLaughing; // Я уже не isLaughing
18
19    // Запрашиваем сразу несколько состояний (мы будем использовать static_cast<bool>)
20    std::cout << "I am happy? " << static_cast<bool>(me & isHappy) << '\n';
21    std::cout << "I am laughing? " << static_cast<bool>(me & isLaughing) << '\n';
22
23    return 0;
24 }
```

## Почему битовые флаги полезны?

Внимательные читатели заметят, что в примерах с `myflags` мы фактически не экономим память. 8 логических значений займут 8 байт. Но пример, приведенный выше, использует 9 байт (8 для определения параметров и 1 для битового флага)! Так зачем же тогда нужны битовые флаги?

Они используются в двух случаях:

**Случай №1: Если у вас много идентичных битовых флагов.**

Вместо одной переменной `myflags`, рассмотрим случай, когда у вас есть две переменные: `myflags1` и `myflags2`, каждая из которых может хранить 8 значений. Если вы определите их как два отдельных логических набора, то вам потребуется 16 логических значений и, таким образом, 16 байт. Однако с использованием битовых флагов вам потребуется только 10 байт (8 для определения параметров и 1 для каждой переменной `myflags`). А вот если у вас будет 100 переменных `myflags`, то, используя битовые флаги, вам потребуется 108 байт вместо 800. Чем больше идентичных переменных вам нужно, тем более значительной будет экономия памяти.

Давайте рассмотрим конкретный пример. Представьте, что вы создаете игру, в которой игроку придется бороться с монстрами. Монстр, в свою очередь, может быть устойчив к определенным типам атак (выбранных случайным образом). В игре есть следующие типы атак: яд, молнии, огонь, холод, кража, кислота, паралич и слепота.

Чтобы отследить, к какому типу атаки монстр устойчив, мы можем использовать одно логическое значение на сопротивление (для одного монстра). Это 8 логических значений (сопротивлений) для одного монстра = 8 байт.

Для 100 монстров это будет 800 переменных типа `bool` и 800 байт памяти.

А вот используя битовые флаги:

```
1 const unsigned char resistsPoison    = 0x01;
2 const unsigned char resistsLightning = 0x02;
3 const unsigned char resistsFire      = 0x04;
4 const unsigned char resistsCold      = 0x08;
5 const unsigned char resistsTheft     = 0x10;
6 const unsigned char resistsAcid      = 0x20;
7 const unsigned char resistsParalysis = 0x40;
8 const unsigned char resistsBlindness = 0x80;
```

Нам нужен будет только 1 байт для хранения сопротивления каждого монстра и одноразовая плата в 8 байт для типов атак.

Таким образом, потребуется только 108 байт или примерно в 8 раз меньше памяти.

В большинстве программ, сохраненный объем памяти с использованием битовых флагов не стоит добавленной сложности. Но в программах, где есть десятки тысяч или даже миллионы похожих объектов, их использование может значительно сэкономить память. Согласитесь, знать о таком полезном трюке не помешает.

**Случай №2:** Представьте, что у вас есть функция, которая может принимать любую комбинацию из 32 различных вариантов. Одним из способов написания такой функции является использование 32 отдельных логических параметров:

```
1 void someFunction(bool option1, bool option2, bool option3, bool option4, bool option5, bool option6, bool option7, bool option8, bool option9, bool option10, bool option11, bool option12, bool option13, bool option14, bool option15, bool option16, bool option17, bool option18, bool option19, bool option20, bool option21, bool option22, bool option23, bool option24, bool option25, bool option26, bool option27, bool option28, bool option29, bool option30, bool option31, bool option32);
```

Затем, если вы захотите вызвать функцию с 10-м и 32-м параметрами, установленными как `true` — вам придется сделать следующее:

```
1 someFunction(false, false, false, false, false, false, false, false, false, true, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, true);
```

Т.е. перечислить все варианты как `false`, кроме 10 и 32 — они `true`. Читать такой код сложно, да и требуется держать в памяти порядковые номера нужных параметров (10 и 32 или 11 и 33?). Такая простыня не может быть эффективной.

А вот если определить функцию с помощью битовых флагов:

```
1 void someFunction(unsigned int options);
```

То можно выбирать и передавать только нужные параметры:

```
1 someFunction(option10 | option32);
```

Кроме того, что это читабельнее, это также эффективнее и производительнее, так как включает только 2 операции (1 побитовое ИЛИ и 1 передача параметров).

Вот почему в OpenGL используют битовые флаги вместо длинной последовательности логических значений.

Также, если у вас есть неиспользуемые битовые флаги и вам нужно позже добавить параметры, вы можете просто определить битовый флаг. Нет необходимости изменять [прототип функции](#), а это плюс к обеспечению обратной совместимости.

## Введение в `std::bitset`

Все эти биты, битовые флаги, операции-манипуляции — всё это утомительно, не правда ли? К счастью, в Стандартной библиотеке C++ есть такой объект, как `std::bitset`, который упрощает работу с битовыми флагами.

Для его использования необходимо подключить [заголовочный файл](#) `bitset`, а затем определить переменную типа `std::bitset`, указав необходимое количество бит. Она должна быть константой [типа compile time](#).

```
1 #include <bitset>
2
3 std::bitset<8> bits; // нам нужно 8 бит
```

При желании `std::bitset` можно инициализировать начальным набором значений:

```
1 #include <bitset>
2
3 std::bitset<8> bits(option1 | option2) ; // начнем с включенных option1 и option2
4 std::bitset<8> morebits(0x2) ; // начнем с битового шаблона 0000 0010
```

Обратите внимание, наше начальное значение [конвертируется в двоичную систему](#). Так как мы ввели шестнадцатеричное 2, то `std::bitset` преобразует его в двоичное 0000 0010.

В `std::bitset` есть 4 основные функции:

- ➔ функция `test()` — позволяет узнать значение бита (0 или 1).
- ➔ функция `set()` — позволяет *включить* биты (если они уже включены, то ничего не произойдет).
- ➔ функция `reset()` — позволяет *выключить* биты (если они уже выключены, то ничего не произойдет).
- ➔ функция `flip()` — позволяет изменить значения бит на противоположные (с 0 на 1 или с 1 на 0).

Каждая из этих функций принимает в качестве параметров позиции бит. Позиция крайнего правого бита (последнего) равна 0, затем порядковый номер растет с каждым последующим битом влево (1, 2, 3, 4 и т.д.). Старайтесь давать содержательные имена битовым индексам (либо путем присваивания их константным переменным, либо с помощью перечислений — о них мы поговорим на соответствующем уроке).

```
1 #include <iostream>
```

```
2 #include <bitset>
3
4 // Обратите внимание, используя std::bitset, наши options соответствуют порядковым
5 const int option_1 = 0;
6 const int option_2 = 1;
7 const int option_3 = 2;
8 const int option_4 = 3;
9 const int option_5 = 4;
10 const int option_6 = 5;
11 const int option_7 = 6;
12 const int option_8 = 7;
13
14 int main()
15 {
16     // Помните, что отсчет бит начинается не с 1, а с 0
17     std::bitset<8> bits(0x2); // нам нужно 8 бит, начнем с битового шаблона 0000 0010
18     bits.set(option_5); // включаем 4-й бит - его значение изменится на 1 (теперь мы имеем 0001 0010)
19     bits.flip(option_6); // изменяем значения 5-го бита на противоположное (теперь мы имеем 0001 1010)
20     bits.reset(option_6); // выключаем 5-й бит - его значение снова 0 (теперь мы имеем 0001 0010)
21
22     std::cout << "Bit 4 has value: " << bits.test(option_5) << '\n';
23     std::cout << "Bit 5 has value: " << bits.test(option_6) << '\n';
24     std::cout << "All the bits: " << bits << '\n';
25
26     return 0;
27 }
```

Результат выполнения программы:

```
Bit 4 has value: 1
Bit 5 has value: 0
All the bits: 00010010
```

Обратите внимание, отправляя переменную `bits` в `std::cout` — выводятся значения всех бит в `std::bitset`.

Помните, что инициализируемое значение `std::bitset` рассматривается как двоичное, в то время как функции `std::bitset` используют позиции бит!

`std::bitset` также поддерживает стандартные побитовые операторы (`|`, `&` и `^`), которые также можно использовать (они полезны при проведении операций одновременно сразу с несколькими битами).

Вместо выполнения всех побитовых операций вручную, рекомендуется использовать `std::bitset`, так как он более удобен и менее подвержен ошибкам.

## Битовые маски

Включение, выключение, переключение или запрашивание сразу нескольких бит можно осуществить в одной битовой операции. Когда мы соединяем отдельные биты вместе, в целях их модификации как группы, то это называется **битовой маской**.

Рассмотрим пример. В следующей программе мы просим пользователя ввести число. Затем, используя битовую маску, мы сохраняем только последние 4 бита, значения которых и выводим в консоль:

```

1  #include <iostream>
2
3  int main()
4  {
5      const unsigned int lowMask = 0xF; // битовая маска для хранения последних 4-х бит
6
7      std::cout << "Enter an integer: ";
8      int num;
9      std::cin >> num;
10
11     num &= lowMask; // удаляем первые биты, оставляя последние 4
12
13     std::cout << "The 4 low bits have value: " << num << '\n';
14
15     return 0;
16 }
```

Результат выполнения программы:

```

Enter an integer: 151
The 4 low bits have value: 7
```

151 в десятичной системе равно 1001 0111 в двоичной. lowMask — это 0000 1111 в 8-битной двоичной системе. 1001 0111 & 0000 1111 = 0000 0111, что равно десятичному 7.

## Пример с RGBA

Цветные дисплейные устройства, такие как телевизоры и мониторы, состоят из миллионов пикселей, каждый из которых может отображать точку цвета. Точка цвета состоит из 3-х пучков: один красный, один зелёный и один синий (сокр. «**RGB**» от англ. «*Red, Green, Blue*»). Изменяя их интенсивность, можно воссоздать любой цвет. Количество цветов R, G и B в одном пикселе представлено 8-битным целым числом unsigned. Например, красный цвет имеет R = 255, G = 0, B = 0; фиолетовый: R = 255, G = 0, B = 255; серый: R = 127, G = 127, B = 127.

Используется еще 4-е значение, которое называется A. «**A**» от англ. «*Alfa*», которое отвечает за прозрачность. Если A = 0, то цвет полностью прозрачный. Если A = 255, то цвет непрозрачный.

В совокупности R, G, B и A составляют одно 32-битное целое число, с 8 битами для каждого компонента:

### 32-битное значение RGBA

31-24 бита	23-16 бит	15-8 бит	7-0 бит
RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
red	green	blue	alpha

Следующая программа просит пользователя ввести 32-битное шестнадцатеричное значение, а затем извлекает 8-битные цветовые значения R, G, B и A:

```

1  #include <iostream>
2
3  int main()
4  {
5      const unsigned int redBits = 0xFF000000;
6      const unsigned int greenBits = 0x00FF0000;
7      const unsigned int blueBits = 0x0000FF00;
8      const unsigned int alphaBits = 0x000000FF;
9
10     std::cout << "Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
11     unsigned int pixel;
12     std::cin >> std::hex >> pixel; // std::hex позволяет вводить шестнадцатеричные
13
14     // Используем побитовое И для изоляции красных пикселей, а затем сдвигаем значение
15     unsigned char red = (pixel & redBits) >> 24;
16     unsigned char green = (pixel & greenBits) >> 16;
17     unsigned char blue = (pixel & blueBits) >> 8;
18     unsigned char alpha = pixel & alphaBits;
19
20     std::cout << "Your color contains:\n";
21     std::cout << static_cast<int>(red) << " of 255 red\n";
22     std::cout << static_cast<int>(green) << " of 255 green\n";
23     std::cout << static_cast<int>(blue) << " of 255 blue\n";
24     std::cout << static_cast<int>(alpha) << " of 255 alpha\n";
25
26     return 0;
27 }

```

Результат выполнения программы:

```

Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): FF7F3300
Your color contains:
255 of 255 red
127 of 255 green
51 of 255 blue
0 of 255 alpha

```

В программе, приведенной выше, побитовое И используется для запроса 8-битного набора, который нас интересует, затем мы его сдвигаем вправо в диапазон от 0 до 255 для хранения и вывода.

**Примечание:** RGBA иногда может храниться как ARGB. В таком случае, главным байтом является альфа.

## Заключение

Давайте кратко повторим то, как включать, выключать, переключать и запрашивать битовые флаги.

Для запроса битового состояния используется побитовое И:

```

1  if (myflags & option4) ... // если установлен option4, то делаем что-нибудь

```



Для включения бит используется побитовое ИЛИ:

```
1 myflags |= option4; // включаем option4
2 myflags |= (option4 | option5); // включаем option4 и option5
```

Для выключения бит используется побитовое И с инвертированным литералом:

```
1 myflags &= ~option4; // выключаем option4
2 myflags &= ~(option4 | option5); // выключаем option4 и option5
```

Для переключения между битовыми состояниями используется побитовое исключающее ИЛИ (XOR):

```
1 myflags ^= option4; // включаем или выключаем option4
2 myflags ^= (option4 | option5); // изменяем на противоположные option4 и option5
```

## Тест

Есть следующий фрагмент кода:

```
1 int main()
2 {
3     unsigned char option_viewed = 0x01;
4     unsigned char option_edited = 0x02;
5     unsigned char option_favorited = 0x04;
6     unsigned char option_shared = 0x08;
7     unsigned char option_deleted = 0x80;
8
9     unsigned char myArticleFlags;
10
11     return 0;
12 }
```

**Примечание:** Статья — это myArticleFlags.

### Задание №1

Добавьте строку кода, чтобы пометить статью как уже прочитанную (option\_viewed).

### Задание №2

Добавьте строку кода, чтобы проверить, была ли статья удалена (option\_deleted).

### Задание №3

Добавьте строку кода, чтобы открепить статью от закрепленного места (option\_favorited).

### Задание №4

Почему следующие две строки идентичны?

```
1 myflags &= ~(option4 | option5); // выключаем option4 и option5
2 myflags &= ~option4 & ~option5; // выключаем option4 и option5
```

## Ответы

### Ответ №1

```
myArticleFlags |= option_viewed;
```

### Ответ №2

```
if (myArticleFlags & option_deleted) ...
```

### Ответ №3

```
myArticleFlags &= ~option_favorited;
```

### Ответ №4

[Законы Де Моргана](#) гласят, что если мы используем побитовое НЕ, то операторы И и ИЛИ меняются местами. Поэтому  $\sim(\text{option4} \mid \text{option5})$  становится  $\sim\text{option4} \ \& \ \sim\text{option5}$ .

Оценить статью:

★★★★★ (179 оценок, среднее: 4,86 из 5)



[← Урок №45. Побитовые операторы](#)

[Глава №3. Итоговый тест](#) →



## Комментариев: 30



1. *Хей Лонг:*  
[8 мая 2020 в 12:41](#)

Технология интересная, но мне любопытно, имеет ли она смысл в концепции ооп ? Там ведь все опции для объектов одного класса можно задать в полях этого класса. Если компилятор хорошо оптимизирует память при создании объектов, то флаги не требуются, наверное.

[Ответить](#)



1. *Илья:*  
[22 мая 2020 в 09:54](#)

На компилятор никогда надеяться нельзя. Вот представь: пишешь ты карточную игру и у тебя есть настройки количества игроков (2,3,4,5,6) и выбрать можно произвольное количество (хоть ни одного, хоть все). Можно, конечно, всюду bool пихнуть, но потом с

таким интерфейсом крайне неприятно работать и в итоге появится ещё один слой абстракции, лишь бы упростить такую работу. Битовые флаги прекрасно решат проблему.

### [Ответить](#)



2. *Сергей:*

[15 августа 2020 в 10:40](#)

Интересная мысль. Да в C++ можно сделать поля в 1 бит.

```
1 struct ABC{
2     bool a: 1;
3     bool b: 1;
4     bool c: 1;
5     bool d: 1;
6     bool e: 1;
7     bool f: 1;
8     bool g: 1;
9     bool h: 1;
10 };
```

Но как инициализировать несколько полей сразу?

С флагами это можно сделать просто с | f.

Со структурой придётся так:

```
1 ABC abc;
2 abc.c = 1;
3 abc.f = 1;
```

К сожалению C++ так не умеет (или я не прав?):

```
1 ABC( {c = 1, f = 1} )
```

### [Ответить](#)



2. *Константин:*

[2 января 2020 в 16:33](#)

С Наступившим 2020! Встал (повис:-) вопрос: значения RGBA изменяются строго бит за битом, например 0000'0000 -> 0000'0001 -> 0000'0011 -> 0000'0111 -> 0000'0011 -> и т. д... т. е. как загораются светодиоды на электронном индикаторе уровня записи? Или в произвольном порядке: 0000'0000 -> 0000'0100 -> 1000'0111 -> 0101'0110 -> и т. д.?

Код ниже превращает монитор в светофор:

файл с заголовком ToolFor.h

```
1 #pragma once
2 namespace ToolFor
3 {
4     enum ToolFor
```

```
5      {
6          A1,
7          A2,
8          A3,
9          A4,
10         A5,
11         A6,
12         A7,
13         A8,
14         B9,
15         B10,
16         B11,
17         B12,
18         B13,
19         B14,
20         B15,
21         B16,
22         G17,
23         G18,
24         G19,
25         G20,
26         G21,
27         G22,
28         G23,
29         G24,
30         R25,
31         R26,
32         R27,
33         R28,
34         R29,
35         R30,
36         R31,
37         R32,
38         MAX
39     };
40 }
```

файл `zvit.cpp` с функцией `main`:

```
1  #include <iostream>
2  #include <Windows.h>
3  #include <cstdint>
4  #include <stdint.h>
5  #include <bitset>
6  #include "ToolFor.h"
7
8  uint64_t enNum()
9  {
10     std::cout << "Hy! "; unsigned int pixel{}; std::cin >> std::hex >> pixel;
11 }
12
13 int main() {
```

```

14 SetConsoleCP(1251); SetConsoleOutputCP(1251); using std::cout; using std::endl;
15 cout << "программа монитор как светофор : включите светофор (инициализируйте)";
16 bitset<32> colors{ enNum() };
17 cout << "светофор ждет команд..." << colors << endl;
18 cout << " включаем красный свет:";
19 colors.set(ToolFor::A1);
20 colors.set(ToolFor::A2);
21 colors.set(ToolFor::A3);
22 colors.set(ToolFor::A4);
23 colors.set(ToolFor::A5);
24 colors.set(ToolFor::A6);
25 colors.set(ToolFor::A7);
26 colors.set(ToolFor::A8);
27 colors.set(ToolFor::R25);
28 colors.set(ToolFor::R26);
29 colors.set(ToolFor::R27);
30 colors.set(ToolFor::R28);
31 colors.set(ToolFor::R29);
32 colors.set(ToolFor::R30);
33 colors.set(ToolFor::R31);
34 colors.set(ToolFor::R32);
35 cout << colors << endl;
36 cout << "далее можно в том же духе, но может кто-то знает как здесь можно и";
37 cout << "побитовые ^, &, |, ~, а также цикл for(для присвоения значений) -";
38 return 0;
39 }

```

### Ответить



1. Константин:

7 января 2020 в 21:19

Уррра! Допетрил сам — код в файле zvit.cpp должен быть такой:

```

1  uint_least32_t en0x()
2  {
3      std::cout << "Hy! "; uint_least64_t rValue{}; std::cin >> std::hex >>
4      return rValue;
5  }
6  int main()
7  {
8      using namespace ToolFor; SetConsoleCP(1251); SetConsoleOutputCP(1251);
9
10     cout << "программа управления цветами пикселя (инициализируйте его значение)";
11     bitset<32> pix{ en0x() };
12     cout << "начальное состояние..." << pix << endl;
13     cout << "сперва выбираем степень непрозрачности от " << static_cast<uint_least32_t>(A1) << " до " << static_cast<uint_least32_t>(A255) << " - ";
14     uint_least32_t proz{ en0x() };
15     cout << "прозрачность задана...." << proz << endl;
16
17     for (uint_least32_t poz{ static_cast<uint_least32_t>(A1) }; poz < static_cast<uint_least32_t>(A255); ++poz)

```

```

18     pix.set(poz);
19     cout << pix << endl;
20     cout << "теперь задаём цвета и силу их свечения - ШЕСТИНАДЦАТЕРИЧНЫМ ЗНАЧЕНИЯМИ" << endl;
21     cout << "синий от " << static_cast<uint_least32_t>(B1) + 1 << " до " << static_cast<uint_least32_t>(B1) + 15 << endl;
22     cout << "зелёный от " << static_cast<uint_least32_t>(G1) + 1 << " до " << static_cast<uint_least32_t>(G1) + 15 << endl;
23     cout << "красный от " << static_cast<uint_least32_t>(R1) + 1 << " до " << static_cast<uint_least32_t>(R1) + 15 << endl;
24     uint_least64_t blue{ en0x() }, green{ en0x() }, red{ en0x() };
25
26     for (uint32_t poz{ static_cast<uint32_t>(B1) }; poz < blue; ++poz)
27         pix.set(poz);
28     for (uint32_t poz{ static_cast<uint32_t>(G1) }; poz < green; ++poz)
29         pix.set(poz);
30     for (uint32_t poz{ static_cast<uint32_t>(R1) }; poz < red; ++poz)
31         pix.set(poz);
32     cout << pix;
33     return 0;
34 }

```

[Ответить](#)



3. *Александр:*  
[19 августа 2019 в 21:50](#)

Что-то не понимаю в чём преимущества битового флага. Мы же ещё 8 переменных типа char определяем?

[Ответить](#)



1. *Александр:*  
[4 февраля 2020 в 07:02](#)

Выше было написано в чём преимущество, оно возникает в том случае, когда переменных приблизительно 100 и более, в таком случае нам понадобится 100 флагов и 8 характеристик этих флагов (включение и исключение), без использования флагов было бы 800 переменных

[Ответить](#)



4. *Кекс:*  
[19 августа 2019 в 15:20](#)

Можно ли вместо включения бита сразу пользоваться переключателем ^=?

[Ответить](#)



5. *Анастасия:*  
[31 июля 2019 в 15:45](#)

Эта тема сначала показалась жутко сложной, но как только пошла практика, меня осенило. Иногда даже удивляешься, как люди умеют использовать имеющиеся ресурсы с такой

гениальной экономией.

Вот кстати практика. И конечно нужно было всё собрать в кучу 😊

```
1  #include <iostream>
2
3  int main()
4  {
5      unsigned char option_viewed = 0x01;
6      unsigned char option_edited = 0x02;
7      unsigned char option_favorited = 0x04;
8      unsigned char option_shared = 0x08;
9      unsigned char option_deleted = 0x80;
10
11     unsigned char myArticleFlags = 0x00;
12
13     myArticleFlags |= option_viewed;
14     if (myArticleFlags & option_viewed)
15     {
16         std::cout << "This article was viewed: ";
17         std::cout << static_cast<int>(myArticleFlags & option_viewed) << '\n';
18     }
19
20     myArticleFlags |= option_deleted;
21     if (myArticleFlags & option_deleted)
22     {
23         std::cout << "This article was deleted: ";
24         std::cout << static_cast<int>(myArticleFlags & option_deleted) << '\n';
25     }
26
27     myArticleFlags |= option_favorited;
28     if (myArticleFlags & option_favorited)
29     {
30         std::cout << "This article is marked as a favorite: ";
31         std::cout << static_cast<int>(myArticleFlags & option_favorited) << '\n';
32     }
33
34     std::cout << "Next, remove from the favorites.\n";
35
36     myArticleFlags &= ~option_favorited;
37     if (myArticleFlags & option_favorited)
38         std::cout << "This article is marked as a favorite: ";
39     else
40         std::cout << "This article is not marked as a favorite.\n";
41
42     return 0;
43
44 }
```

[Ответить](#)



6. Анастасия:

[30 мая 2019 в 19:31](#)

Сначала мозг ушёл в бунт после строчки

```
1 | const unsigned char option5 = 0x10; //шестнадцатеричный литерал для 0001 0000
```

но потом всё стало интуитивно понятно. Кроме момента, когда просим пользователя ввести значения пикселя в 16-ричной системе исчисления. Я тут одна не понимаю, причём тут 16-ричная система исчисления и как пользователь (например, я) может осознанно что-то в ней вводить, предварительно не прикинув всё в десятичной или хотя бы двоичной системе?

[Ответить](#)



1. Анастасия:

[31 мая 2019 в 22:46](#)

Перечитала 36 урок и поняла, почему  $0x10 = 0001\ 0000$  и так далее. Суффикс 0x означает, что запись дальше — в 16-ричной системе.

[Ответить](#)



1. Денис:

[23 ноября 2019 в 14:39](#)

а я так и не понял, почему  $0x10 = 0001\ 0000$ , расскажите плз)

[Ответить](#)



1. Артемий:

[7 февраля 2020 в 15:55](#)

36 урок



7. Вячеслав:

[9 февраля 2019 в 00:51](#)

Попробовал все объединить и вот что вышло :

```
1 | #include "pch.h"
2 | #include <iostream>
3 | using namespace std;
4 | int main()
5 | { //определяем набор опций (флаги) состояний статьи
6 |     const uint8_t option_viewed = 0x01; //просмотрена
7 |     const uint8_t option_edited = 0x02; //отредактирована
8 |     const uint8_t option_favorited = 0x04; //избранная
9 |     const uint8_t option_shared = 0x08; //доступна
```



```

10  const uint8_t option_deleted = 0x80; //удалена
11
12  uint8_t myArticleFlags; //статья
13
14  myArticleFlags |= option_viewed; //выделяем статью как просмотренную
15  cout << "My articles flags is viewed:" << '\n';
16
17  if (myArticleFlags & option_deleted); //проверяем удалина ли статья
18  cout << "My articles flags is not deleted:" << '\n';
19
20
21  myArticleFlags &= ~option_favorited; //открепляем статью
22  cout << "My articles flags is checkout as a favorite:";
23
24  return 0;
25
26 }

```

### Ответить



1. Анастасия:

[30 мая 2019 в 19:54](#)

Здравствуйте. У меня вопросы и замечания к Вашему коду (не просто так ведь Вы его на всеобщее обозрение выложили?):

1) зачем

```
1 #include "pch.h"
```

?

2)

```

1 if (myArticleFlags & option_deleted); //проверяем удалина ли статья
2   cout << "My articles flags is not deleted:" << '\n';

```

насколько я понимаю, после условия if если хотим, чтобы действие выполнялось только при этом условии, точки с запятой быть не должно, иначе это два разных "стэйтмента". И второе. Вы, видимо, предполагаете, что если есть нужная единица для option\_deleted, то всё выражение не будет 0, то есть будет истинным. Это вроде и понятно, но дойти до этой логической цепочки можно не сразу. Я сделала так:

```
1 if ((myArticleFlags & option_deleted) >= 16) std::cout << "Article was
```

16 — это 0001 0000, у него есть одна из двух нужных единиц для deleted. Может, можно и лучше сделать. На мой взгляд тут есть сложный момент. Т.к. удалённая статья в двоичной системе должна иметь вид \*1\*1 \*\*\*\*

3) "удалИна", хоть и закомментировано и к коду не относится, но всё равно режет глаза.

### Ответить



1. Анастасия:

[31 мая 2019 в 22:52](#)

Я была не права, переводя 0x80 в двоичную систему, как из десятичной с результатом 0101 0000. На самом деле это запись в шестнадцатеричной системе, а значит 1000 0000 и действительно можно написать

```
1 | if (myArticleFlags & option_deleted) std::cout << "Article was deleted"
```

так как условие после if будет true только если myArticleFlags будет иметь вид 1\*\*\*  
\*\*\*\*

[Ответить](#)



1. *Victor:*

[23 июля 2019 в 00:27](#)

Здравствуйте! =) Как у вас успехи с C++ ? Вы его все ещё изучаете или уже — всё? =)

Если ещё да — можно задать несколько вопросов??



2. *Анастасия:*

[28 июля 2019 в 17:10](#)

ответ Виктору: да, ещё изучаю, сейчас на 120-м уроке. Странно, что Вы хотите задать вопрос именно мне, но если так, то задавайте.

[Ответить](#)



1. *Victor:*

[10 сентября 2019 в 10:58](#)

Ничего странного. Элементарный "недорандом"... ))

Вопрос тоже "странный". Не хотите ли в компании изучать ++?

Прочитал тут одну книжечку об обучении, там весьма убедительно рекомендуют делать это в паре, или не большой группе, по огромному ряду причин.



2. *Анастасия:*

[21 сентября 2019 в 14:19](#)

ответ Виктору: почему бы и нет? Я прерывалась на три недели, сейчас на 160 уроке. Давайте свяжемся как-то иначе и обсудим, что мы можем предпринять для повышения эффективности обучения. Моя почта [amolchkova@mail.ru](mailto:amolchkova@mail.ru) и скайп anastasiya.molchkova



3. *Victor:*

[23 сентября 2019 в 22:32](#)

Написал и туда, и туда, на всякий случай ещё и сюда пишу. Вот email: [cosintup@gmail.com](mailto:cosintup@gmail.com)



8. Евгений:

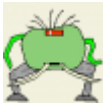
[26 октября 2018 в 04:01](#)

А не проще использовать битовые поля?

```
1 struct BYTE
2 {
3     char bit_0 : 1
4     char bit_1 : 1
5     char bit_2 : 1
6     char bit_3 : 1
7     char bit_4 : 1
8     char bit_5 : 1
9     char bit_6 : 1
10    char bit_7 : 1
11 }
```

Так вроде ещё удобней, вообще не надо никаких битовых операций)

[Ответить](#)



1. Виталий:

[16 сентября 2019 в 03:46](#)

битовые поля это хорошо, но побитовые операции несут за собой много интересных преимуществ  
читать нужно внимательнее

[Ответить](#)

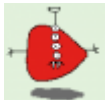


9. korvell:

[9 июня 2018 в 14:20](#)

Сложная тема, понял не до конца. 😞

[Ответить](#)



10. Илья:

[30 января 2018 в 21:19](#)

да ладно, в хваленном c++ нельзя отформатировать литерал??? нули в начале не увеличивают память, они нужны как заполнитель байта.

может быть я непонятно объясняю:

x = 0x00abcdef//hex

в двоичном коде:

вывод по умолчанию (грубо 3 байта)

1010 1011 1100 1101 1110 1111

однако резерв 4 байта, куда делись нули в начале?

0 0 a b c d e f

0000 0000 1010 1011 1100 1101 1110 1111

понятно что они ничего не значат, они нужны для правильного расчёта функций

[Ответить](#)

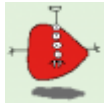


1. *Юрий:*

[30 января 2018 в 21:27](#)

Виноват. Неправильно понял ваш вопрос. Нужно заполнить вручную те 4 пустые бита с помощью побитовых операторов.

[Ответить](#)



1. *Илья:*

[30 января 2018 в 21:33](#)

в таком случае нужно еще проверять каждое значение функции, а это дополнительная операция, мне хочется найти решение чтобы значение переменной при всех раскладах было бы в формате 32 бита  
вывод в hex или bin неважно, нули на своем месте

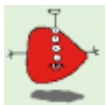
[Ответить](#)



1. *Юрий:*

[30 января 2018 в 21:55](#)

Здесь уже помочь не смогу, чтобы значение переменной при всех раскладах было бы в формате 32 бита — значит нужно всё равно заполнить полностью эти 32 бита вручную — моё видение проблемы. Стандартное решение (и есть ли оно) мне неизвестно, попробуйте спросить на [Stackoverflow](#).



11. *Илья:*

[30 января 2018 в 16:53](#)

добрый день

как указать в c++ чтобы переменная unsigned long int была в любом случае 32 бита, даже если она полностью состоит из 0.

например в hex формате число 0x0bca6351 зайдёт в переменную как bca6351

[Ответить](#)



1. *Denis:*

[28 марта 2018 в 19:54](#)

А почему вы взяли , что так вообще можно. Коль уж каждой переменной типа(int) выделяется своя память , то она или выделяется или нет , так как откуда компьютер знает , не запишете ли вы в него число побольше 0 , например даже 21902. Он динамически в

работе программы в статические переменные просто бы не выделял новой памяти. P.S мой ответ нет , нельзя.

[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