

Ravesli [Ravesli](#)

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





Урок №59. Классы enum

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 14 Сен 2020 |

 48216

[13](#)

Хотя [перечисления](#) и считаются отдельными типами данных в языке C++, они не столь безопасны, как кажутся на первый взгляд, и в некоторых случаях позволят вам делать вещи, которые не имеют смысла. Например:

```
1 #include <iostream>
2
3 int main()
4 {
5     enum Fruits
6     {
7         LEMON, // LEMON находится внутри той же области видимости, что и Fruits
8         KIWI
9     };
10
11     enum Colors
12     {
13         PINK, // PINK находится внутри той же области видимости, что и Colors
14         GRAY
15     };
16
17     Fruits fruit = LEMON; // Fruits и LEMON доступны в одной области видимости (добавл.
```

```

18 Colors color = PINK; // Colors и PINK доступны в одной области видимости (добавлять)
19
20 if (fruit == color) // компилятор будет сравнивать эти переменные как целые числа
21     std::cout << "fruit and color are equal\n"; // и обнаружит, что они равны!
22 else
23     std::cout << "fruit and color are not equal\n";
24
25 return 0;
26 }

```

Когда C++ будет сравнивать переменные `fruit` и `color`, он неявно преобразует их в целочисленные значения и сравнит эти целые числа. Так как значениями этих двух переменных являются перечислители, которым присвоено значение `0`, то это означает, что в примере, приведенном выше, `fruit = color`. А это не совсем то, что должно быть, так как `fruit` и `color` из разных перечислений и их вообще нельзя сравнивать (фрукт и цвет!). С обычными перечислителями нет способа предотвратить подобные сравнения.

Для решения этой проблемы в C++11 добавили **классы enum** (или «*перечисления с областью видимости*»), которые добавляют перечислениям, как вы уже могли понять, локальную область видимости со всеми её правилами. Для создания такого класса нужно просто добавить **ключевое слово** `class` сразу после `enum`. Например:

```

1 #include <iostream>
2
3 int main()
4 {
5     enum class Fruits // добавление "class" к enum определяет перечисление с ограниченной
6     {
7         LEMON, // LEMON находится внутри той же области видимости, что и Fruits
8         KIWI
9     };
10
11     enum class Colors
12     {
13         PINK, // PINK находится внутри той же области видимости, что и Colors
14         GRAY
15     };
16
17     Fruits fruit = Fruits::LEMON; // примечание: LEMON напрямую не доступен, мы должны использовать Fruits::
18     Colors color = Colors::PINK; // примечание: PINK напрямую не доступен, мы должны использовать Colors::
19
20     if (fruit == color) // ошибка компиляции, поскольку компилятор не знает, как сравнивать Fruits и Colors
21         std::cout << "fruit and color are equal\n";
22     else
23         std::cout << "fruit and color are not equal\n";
24
25     return 0;
26 }

```

Стандартные перечислители находятся в той же области видимости, что и само перечисление (в [глобальной области видимости](#)), поэтому вы можете напрямую получить к ним доступ (например, PINK). Однако с добавлением класса, который ограничивает область видимости каждого перечислителя областью видимости его перечисления, для доступа к нему потребуется оператор разрешения области видимости (например, Colors::PINK). Это значительно снижает риск возникновения [конфликтов имен](#).

Поскольку перечислители являются частью класса enum, то необходимость добавлять префиксы к идентификаторам отпадает (например, можно использовать просто PINK вместо COLOR_PINK, так как Colors::COLOR_PINK уже будет лишним).

А строгие правила типов классов enum означают, что каждый класс enum считается уникальным типом. Это означает, что компилятор не сможет сравнивать перечислители из разных перечислений. Если вы попытаетесь это сделать, компилятор выдаст ошибку (как в примере, приведенном выше).

Однако учтите, что вы можете сравнивать перечислители внутри одного класса enum (так как эти перечислители принадлежат одному типу):

```
1  #include <iostream>
2
3  int main()
4  {
5      enum class Colors
6      {
7          PINK,
8          GRAY
9      };
10
11     Colors color = Colors::PINK;
12
13     if (color == Colors::PINK) // это нормально
14         std::cout << "The color is pink!\n";
15     else if (color == Colors::GRAY)
16         std::cout << "The color is gray!\n";
17
18     return 0;
19 }
```

С классами enum компилятор больше не сможет неявно конвертировать значения перечислителей в целые числа. Это хорошо! Но иногда могут быть ситуации, когда нужно будет вернуть эту особенность. В таких случаях вы можете [явно преобразовать](#) перечислитель класса enum в [тип int](#), используя оператор static_cast:

```
1  #include <iostream>
2
3  int main()
4  {
5      enum class Colors
6      {
```

```
7      PINK,  
8      GRAY  
9  };  
10  
11  Colors color = Colors::GRAY;  
12  
13  std::cout << color; // не будет работать, поскольку нет неявного преобразования в  
14  std::cout << static_cast<int>(color); // результатом будет 1  
15  
16  return 0;  
17 }
```

Если вы используете компилятор, поддерживающий C++11, то нет никакого смысла использовать обычные перечисления вместо классов enum.

Обратите внимание, ключевое слово `class` вместе с ключевым словом `static` являются одними из самых запутанных в языке C++, поскольку результат сильно зависит от варианта применения (контекста). Хотя классы `enum` используют ключевое слово `class`, в C++ они не считаются традиционными «классами». О традиционных классах мы поговорим несколько позже.

Оценить статью:

★★★★★ (328 оценок, среднее: 4,91 из 5)



[← Урок №58. Перечисления](#)

[Урок №60. Псевдонимы типов: typedef и type alias](#) →



Комментариев: 13



1. [Анатолий:](#)
[11 июня 2020 в 03:56](#)

Между прочим в C# гораздо удобнее `enum`. Там даже можно извлекать названия полей в виде строк... А этот `enum` оч неудобен! И перевод его в разряд `class` делает его ещё более неудобным чем без... Почему вообще это придумали? И чем они вообще руководствовались? Если уж делать, то делать так чтобы было лучше, а не добавлять проблемы... Или по крайней мере так, чтобы их можно было удобно решить!!!

[Ответить](#)

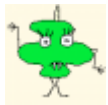


2. [Анатолий:](#)

[11 июня 2020 в 03:49](#)

Использование `enum class` скорее вызывает массу проблем, чем удобства. Потому что да, можно обращаться через четырёхточие как элемент класса. Но только тут это удобно. Однако во всех остальных местах прут неудобства со всех щелей... Приходится писать кучу никчемных операторов просто для разных операций, конъюнкции побитовой, дизъюнкции побитовой, равенства, неравенства и тд и тп... А если к несчастью вам надо извлечь данные в виде типа `unsigned`, то оказывается, это невозможно. Потому что надо добавить оператор преобразования типа в сам класс `enum`. По синтаксису это недопустимо ибо оператор приведения типа не мб внешним.... Тупик! Приходится писать явное преобразование типа... даже если вы напишете `enum EType class:unsigned {f1,f2,..}` и вам нагдо привести поля к типу `unsigned`, то вам придётся явно написать `static_cast<unsigned>(EType::f1)`. А представьте себе это в списке инициализации, там 100 элементов(это ещё не самый длинный!) и ваш код будет просто захламлён этими преобразованиями типа, которые по сути ничего не делают и нужны только для удовлетворения синтаксису ибо ваш `enum` и так этого типа ... А если вы используете старый добрый `enum` без указания, что это класс, то ни одной из вышеуказанных проблем даже не возникнет, всё делаете на автомате и никаких операторов не надо! Но с другой стороны, если это `class`, то почему туда нельзя дописать методы как в полноценный класс? Этого нет даже в последней версии C++. Возможно в какой нибудь C++25 или выше это наконец появится и `enum class` станет удобным. а пока что от него одни проблемы. не советую им пользоваться. к тому же вы же всегда к полям можете дописать префиксы и тем самым выделить их и у вас не возникнет конфликтов имён... ну т.е. всё ещё очень сыро...

[Ответить](#)



3. *Алексей:*

[15 июля 2019 в 14:49](#)

Интересно. Первый пример записал в `vim` и пробовал скомпилировать — `warning: comparison between 'enum main():Fruits' and 'enum main():Colors' [-Wenum-compare]`

[Ответить](#)



1. *Илья:*

[4 ноября 2019 в 10:27](#)

так это ж просто "warning" — не ошибка

[Ответить](#)



2. *Анатолий:*

[11 июня 2020 в 04:00](#)

Тебе надо добавить оператор сравнения... Это ещё можно сделать. Считается что они разных типов... там придётся добавить ещё много операторов если активно работать с этим `enum`. И это ещё не всё...

[Ответить](#)

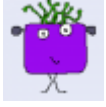


4. *WiseWanderer:*
[27 июня 2019 в 01:29](#)

Забыли написать, что можно тип указывать:

```
1 | enum class EnumName : unsigned long long { CONST_1, CONST_2 };
```

[Ответить](#)



5. *Владимир:*
[29 ноября 2018 в 12:36](#)

Возвращаясь к уроку 46, можно вместо такой записи

```
1 | const unsigned char option1 = 0x01; // шестнадцатеричный литерал для 0000 0001
2 | const unsigned char option2 = 0x02; // шестнадцатеричный литерал для 0000 0010
3 | const unsigned char option3 = 0x04; // шестнадцатеричный литерал для 0000 0100
4 | const unsigned char option4 = 0x08; // шестнадцатеричный литерал для 0000 1000
5 | const unsigned char option5 = 0x10; // шестнадцатеричный литерал для 0001 0000
6 | const unsigned char option6 = 0x20; // шестнадцатеричный литерал для 0010 0000
7 | const unsigned char option7 = 0x40; // шестнадцатеричный литерал для 0100 0000
8 | const unsigned char option8 = 0x80; // шестнадцатеричный литерал для 1000 0000
9 |
10 | unsigned char myflags = 0;
```

Использовать такую:

```
1 | enum class Options{
2 |     OPTION_1 = 0x01, // шестнадцатеричный литерал для 0000 0001
3 |     OPTION_2 = 0x02, // шестнадцатеричный литерал для 0000 0010
4 |     OPTION_3 = 0x04, // шестнадцатеричный литерал для 0000 0100
5 |     OPTION_4 = 0x08, // шестнадцатеричный литерал для 0000 1000
6 |     OPTION_5 = 0x10, // шестнадцатеричный литерал для 0001 0000
7 |     OPTION_6 = 0x30, // шестнадцатеричный литерал для 0010 0000
8 |     OPTION_7 = 0x40, // шестнадцатеричный литерал для 0100 0000
9 |     OPTION_8 = 0x80 // шестнадцатеричный литерал для 1000 0000
10 | };
11 |
12 | unsigned char myFlags{ 0 };
```

Как плюсы, используется 4 байта для всех флагов (сюда можно затолкать и 32 флага) вместо 8 байтов + повышение читабельности кода. Единственное, что нехорошо — постоянно придётся делать явное преобразование типов:

```
1 | myFlags |= (static_cast<int>(Options::OPTION_1) | static_cast<int>(Options::OPTION_2));
```

Но если убрать ключевое слово `class`, то можно работать с флагами без преобразования.

[Ответить](#)



1. Константин:

[17 февраля 2019 в 01:33](#)

Владимир, а скажи мне: в чём разница между

```
1 | const unsigned char option6 = 0x20; // шестнадцатеричный литерал для 0010 0000
```

и

```
1 | OPTION_6 = 0x30, // шестнадцатеричный литерал для 0010 0000
```

?

Если это ОЧЕпятка, то как у тебя этот код вообще работал? Если это просто теория, то набросай какую задачу можно целиком (не фрагмент) этими флагами решить. Спасибо.

[Ответить](#)



1. Владимир:

[18 февраля 2019 в 21:46](#)

Как я указал в комментарии, разница в том, что `enum` выделит 4 байта для хранения этих флагов, в то время как каждое использование `const unsigned char` будет создавать переменную размером в 1 байт. Если создать 32 флага, то через `const unsigned char` это всё займет 32 байта, в то время, как через `enum` — все те же 4. Так же повышает и читаемость кода, т.к. флаги группируются в одно перечисление, к тому же получают своё имя, по которому сгруппированы. Ещё раз повторюсь, читать комментарий надо было внимательнее, я там всё это указал.

Код прекрасно работает, если не знаком с битовыми масками, то тебе в 46-ой урок, читай, вникай. Там и задача как раз есть, её и реши, только вместо этого:

```
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 | }
```

Используй это:

```

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

```

[Ответить](#)



1. *Алексей:*

[10 мая 2019 в 22:02](#)

Идея, наверное, неплохая... Вот только вопрос в производительности этого кода (и читаемости тоже)... Мне что-то внутри подсказывает что в больших проектах постоянное использование `static_cast` будет неплохо тормозить работу проекта... Ну а без `class` не очень безопасно... Впрочем, если я не прав, то поправьте



2. *Илья:*

[4 ноября 2019 в 10:31](#)

Можно делать, но слово `class` надо убрать.
Для примера — библиотека SFML:

```
1 sf::Color::Red
```

`Color` — перечисление
`Red` — перечислитель со значением `0xFF000000`;
так что такая идея повсеместно используется.



6. *Алибек:*

[15 мая 2018 в 12:23](#)

Что-то я не вижу преимуществ `enum` перед `enum class`

[Ответить](#)



1. *Юрий:*

[16 мая 2018 в 20:55](#)

Классы enum добавили позднее и это уже усовершенствованные enum. Если ваш компилятор поддерживает C++11, то смысла использовать enum в рабочих проектах, вместо классов enum — нет.

[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 «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -

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