S.4.5a — Enum classes

BY ALEX ON APRIL 23RD, 2015 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

Although enumerated types are distinct types in C++, they are not type safe, and in some cases will allow you to do things that don't make sense. Consider the following case:

```
#include <iostream>

int main()
{
    enum Color
    {
        RED, // RED is placed in the same scope as Color
        BLUE
    };

    enum Fruit
    {
        BANANA, // BANANA is placed in the same scope as Fruit
        APPLE
    };

    Color color = RED; // Color and RED can be accessed in the same scope (no prefix needed)
    Fruit fruit = BANANA; // Fruit and BANANA can be accessed in the same scope (no prefix nee

    if (color == fruit) // The compiler will compare a and b as integers
        std::cout << "color and fruit are equal\n"; // and find they are equal!
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

When C++ compares color and fruit, it implicitly converts color and fruit to integers, and compares the integers. Since color and fruit have both been set to enumerators that evaluate to value 0, this means that in the above example, color will equal fruit. This is definitely not as desired since color and fruit are from different enumerations and are not intended to be comparable! With standard enumerators, there's no way to prevent comparing enumerators from different enumerations.

C++11 defines a new concept, the **enum class** (also called a **scoped enumeration**), which makes enumerations both strongly typed and strongly scoped. To make an enum class, we use the keyword **class** after the enum keyword. Here's an example:

```
#include <iostream>

int main()
{
    enum class Color // "enum class" defines this as a scoped enumeration instead of a standar
    {
        RED, // RED is inside the scope of Color
        BLUE
    };

    enum class Fruit
    {
        BANANA, // BANANA is inside the scope of Fruit
        APPLE
```

```
    };

    Color color = Color::RED; // note: RED is not directly accessible any more, we have to use
    Fruit fruit = Fruit::BANANA; // note: BANANA is not directly accessible any more, we have

    if (color == fruit) // compile error here, as the compiler doesn't know how to compare dif
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

With normal enumerations, enumerators are placed in the same scope as the enumeration itself, so you can typically access enumerators directly (e.g. RED). However, with enum classes, the strong scoping rules mean that all enumerators are considered part of the enumeration, so you have to use a scope qualifier to access the enumerator (e.g. Color::RED). This helps keep name pollution and the potential for name conflicts down.

Because the enumerators are part of the enum class, there's no need to prefix the enumerator names (e.g. it's okay to use RED instead of COLOR_RED, since Color::COLOR_RED is redundant).

The strong typing rules means that each enum class is considered a unique type. This means that the compiler will *not* implicitly compare enumerators from different enumerations. If you try to do so, the compiler will throw an error, as shown in the example above.

However, note that you can still compare enumerators from within the same enum class (since they are of the same type):

```
#include <iostream>

int main()
{
    enum class Color
    {
        RED,
        BLUE
    };

    Color color = Color::RED;

    if (color == Color::RED) // this is okay
        std::cout << "The color is red!\n";
    else if (color == Color::BLUE)
        std::cout << "The color is blue!\n";

    return 0;
}
```

With enum classes, the compiler will no longer implicitly convert enumerator values to integers. This is mostly a good thing. However, there are occasionally cases where it is useful to be able to do so. In these cases, you can explicitly convert an enum class enumerator to an integer by using a static_cast to int:

```
#include <iostream>

int main()
{
    enum class Color
    {
        RED,
        BLUE
    };
```

```
10
11      Color color = Color::BLUE;
12
13      std::cout << color; // won't work, because there's no implicit conversion to int
14      std::cout << static_cast<int>(color); // will print 1
15
16      return 0;
17  }
```

If you're using a C++11 compiler, there's little reason to use normal enumerated types instead of enum classes.

Note that the class keyword, along with the static keyword, is one of the most overloaded keywords in the C++ language, and can have different meanings depending on context. Although enum classes use the class keyword, they aren't considered "classes" in the traditional C++ sense. We'll cover actual classes later.

Also, just in case you ever run into it, "enum struct" is equivalent to "enum class". But this usage is not recommended and is not commonly used.

 **S.4.7 -- Structs**

 **Index**

 **S.4.5 -- Enumerated types**

C++ TUTORIAL | PRINT THIS POST

107 comments to S.4.5a — Enum classes

« Older Comments 1 2

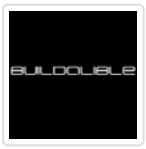
BuIlDaLiBlE

November 16, 2019 at 5:50 am · Reply

Absolutely not important but in the first code block there is an additional new line after the include directive, however in the other code blocks there isn't.

Alex
November 16, 2019 at 5:59 pm · Reply

Thanks for pointing out the inconsistency.



BuIlDaLiBlE
November 17, 2019 at 1:37 am · Reply

You still forgot the last block ;)

Alex
November 17, 2019 at 9:36 am · Reply

Weird, it looks fine to me. Maybe a caching issue?

BuIlDaLiBlE
November 17, 2019 at 9:40 am · Reply

I checked later and it was indeed fixed. Seems ok now. Can this be the cache though? AFAIK HTML always loads fresh and it's the external files that are cached.

nascardriver
November 17, 2019 at 9:44 am · Reply

I fixed the last block, not a caching issue.

Alex
November 17, 2019 at 9:52 am · Reply

:)

hellmet
October 30, 2019 at 2:13 am · Reply

In 8.15 (specifically, this comment https://www.learncpp.com/cpp-tutorial/8-15-nested-types-in-classes/#comment-337755),
I see

```
1 enum Name : Type {...}
```

I'm not sure how that works, but perhaps worth mentioning here.

nascardriver
October 30, 2019 at 6:24 am · Reply

Thanks for pointing it out! I added the base declaration to lesson S.4.5
https://www.learncpp.com/cpp-tutorial/45-enumerated-types/#base

hellmet
October 30, 2019 at 9:13 am · Reply

My pleasure!

MisterNotMister
August 31, 2019 at 1:55 am · Reply

Hi, what happens if you are using a struct or class
and you define an enum inside it like as such.

```
My gibberish output code here

//struct.h

struct Foo{
  enum myEnum{
  BLUE = 0,
  RED,
  };
};

//random.cpp

#include "struct.h

int APPLE = Foo::myEnum::RED;
```

(dumb example, but is this also a right way of using enums?)
EDIT: Why aren't code tags working for me???

nascardriver
August 31, 2019 at 11:42 pm · Reply

That's perfectly fine. The only problem is that nested types can't be forward declared, which can quickly lead to circular dependencies. If the nested type is used by name outside of the class, don't declare it inside the class. If the nested type is only used internally or by returns, you can declare it inside of the class.

> Why aren't code tags working for me?
They stop working after editing a comment. They start working again when you refresh the page.

YB
August 22, 2019 at 1:48 pm · Reply

In C# there is ToString() for the enum values to get the name of the value in string format.

Is there any plan to add the same functionality for C++ enum class?
Are there any suggestions on how to achieve the functionality with what is currently available?

nascardriver
August 23, 2019 at 2:02 am · Reply

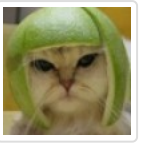
> Is there any plan to add the same functionality for C++ enum class?
No

> Are there any suggestions on how to achieve the functionality with what is currently available?
You'll have to wait until arrays or containers have been covered. After chapter 6, you can use an array or `std::map` to map enumerators to strings.

EDIT: You could use if-statements to check for all enumerators and return the corresponding name.

Q
June 2, 2019 at 8:39 pm · Reply

Enum classes seem like an utterly pointless addition. Instead of bending over backwards to disallow nonsensical comparisons, why not just trust programmers not to be idiots?



Alex
June 11, 2019 at 2:31 pm · Reply

http://www.stroustrup.com/C++11FAQ.html#enum has some justifications for the addition.

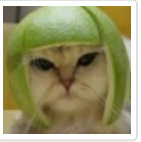
Pharap
April 27, 2019 at 7:53 am · Reply

I've recently been reading the isocpp style guidelines and they advise against using ALL_CAPS for enumerators.

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps

Would the author(s) of the article consider possibly adopting some of the recommendations of this guide?
In particular, not using all-caps for enumerators.

This article is quite popular, so if it sets a good example then it will be responsible for future C++ programmers following good guidelines, which I think can only be a good thing.

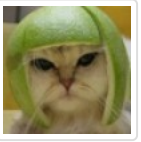


Alex
April 30, 2019 at 5:06 pm · Reply

Thanks for noting this. I'll flag this for reconsideration when this lesson gets updated.

Puya
April 1, 2019 at 10:32 am · Reply

Hi, thanks as always for your helpful tutorials. Is this a typo "(e.g. it's okay to use RED instead of COLOR_RED, since Color::COLOR_RED is redundant)"? Should it be "(e.g. it's okay to use RED instead of Color::RED, since Color::RED is redundant)"?



Alex
April 3, 2019 at 12:29 pm · Reply

No, it's correct as written. With an enum class, You can't use RED, you have to use Color::RED.

B_H@cker
December 27, 2018 at 6:51 am · Reply

В чем отличие "emun" от "class enum"?

nascardriver
December 27, 2018 at 7:10 am · Reply

That's what the entire lesson is about. Give it another read.

Alaa Mahran
December 1, 2018 at 11:39 am · Reply

Can you please elaborate more about this line: "If you're using a C++11 compiler, there's little reason to use normal enumerated types instead of enum classes.", I missed your point here! What is the relation between C++11 and enum classes and why it would be preferable in older versions but not in C++11?

nascardriver
December 2, 2018 at 1:49 am · Reply

enum class was added to C++ in C++11, they aren't available before that. If you're using a compiler that doesn't support C++11 (you should get a new one), you can't use enum class.

Yaroslav
October 12, 2018 at 10:37 am · Reply

Hello. may be i missed it somewhere.

```
1 enum class myenum {zero, one};
2 myenum two = myenum(666);
3 std::cout << int(myenum::zero); // all is ok, i need to use myenum:: and casting it to int i
4 std::cout << int(two) // why it is `two` and not `myenum::two`, why it is telling me that `t
```

nascardriver
October 13, 2018 at 6:08 am · Reply

@myenum doesn't have an enumerator named "two". "two" is a variable.

C.E
October 11, 2018 at 9:42 am · Reply

Alex, when you mention strong typing rules, what do you exactly mean, from my research these "rules" govern variable assignment and implicit type conversion. Reprhasing the question would be how these specefic rules function if I am using non-class based enumerators. In addition you speak of the keyword class being overloaded(in what sense)

Alex

October 11, 2018 at 1:01 pm · Reply

"strong typing" in the context of an enum class means the enumerators live inside the scope of the type (rather than in the global scope) and thus must be prefixed by the type name to be accessible.

The keyword class is more often used for creating user defined structs that have behaviors attached. Essentially the entirety of chapter 8 is about this.

« Older Comments 1 2