



Using Enum Structs as Bitfields

Presentation by Jon Kalb
Based on an article in
Overload magazine by
Anthony Williams

1

What do these have in common?

- `explicit` constructors
- modern casts
 - `const_cast`
 - `static_cast`
- `explicit` conversion functions
- uniform initialization (hint: narrowing conversions)

make it harder to
convert!

Lesson Learned

- Very easy (implicit) casting often results in code where casts are not intended.
- The trend in the standard is away from implicit casting toward explicit casting.
 - Note that existing implicit casting not going away.

maintain backwards compatibility

3

Classic C++ enum annoyances

- Size of enum not specifiable
- Proper scoping rules not respected
`enum values {first, second};`
`a_value = values::first;`
- Promiscuous (implicit) casting from ints and other enums

does not compile!

error prone!

4

C++11 enum features

- Supports a new “underlying type” feature
enum values: int {first, second};

- Optional: old syntax not broken

enum values {first, second};

- We can get the underlying type:

typename std::underlying_type<values>::type

Underlying type is selected by compiler and not portable.

5

enum struct

- Introduced in C++11
- Uses “proper” C++ scoping rules
enum struct values {first, second};
a_value = values::first;

compiles

- Supports the new “underlying type” feature
 - Defaults to int

Underlying type is int.

enum struct values {first, second};

- Can also use “class”

enum class values: char {first, second};

- No implicit casting to int or other enums!

Yea!

6

Classic C++ enum features

- Enums often as bitfields:

```
enum options {first = 1, second = 2, third = 4};  
void some_function(options opt);  
some_function(options(first | third));
```

- Here we are passing `some_function()` the value 5.
- This works because implicit casting allows us to cast the enums to int and the result back.
- But it also allows this:

```
int i{first * third / second};
```

nonsense!

7

enum struct

- The new `enum struct` syntax prevents this:

```
options opt{first * third / second};
```

does not compile
:)

- But also prevents this:

```
some_function(options(first | third));
```

does not compile
:(

- But it can be fixed.

8

|std::launch

- std::launch is a scoped struct defined by the standard which supports bit manipulations
std::launch::async | std::launch::deferred
- and:
std::launch::async & std::launch::deferred
- The standard defines these operations on std::launch:
|, &, ^, ~, |=, &=, and ^=
- We just have to do this for the enums that we want to be bit fields.

easy, but tedious

9

|operator|() as a template

```
template <class E>
E operator|(E lhs, E rhs)
{
    using underlying = typename std::underlying_type<E>::type;
    return static_cast<E>(
        static_cast<underlying>(lhs)
        | static_cast<underlying>(rhs)
    );
}
```

assume appropriate
namespace: bitmask

What's wrong with this
operator|()?

10

| Operator Overloading Guideline

- Always define operators in terms of their assignment operator.
 - DRY

Our operator | () should be defined in terms of operator |=().

Why not define operator |=() in terms of operator | ()?

11

| operator |=() as a template

```
template <class E>
E& operator|=(E& lhs, E rhs)
{
    using underlying = typename std::underlying_type<E>::type;

    static_cast<E>(static_cast<underlying>&(lhs)
                  |= static_cast<underlying>(rhs));
    return lhs;
};
```

Won't compile!
Why?

non-const lvalue reference to type
'underlying' cannot bind to a value of
unrelated type

12

| operator |= () as a template

```
template <class E>
E& operator|=(E& lhs, E rhs)
{
    using underlying = typename std::underlying_type<E>::type;

    return lhs = static_cast<E>(static_cast<underlying>(lhs)
        | static_cast<underlying>(rhs));
};
```

What's missing?

13

| constexpr Guideline

- If it can be constexpr declare it constexpr

14

|operator|=() as a template

```
template <class E>
constexpr E7 operator|=(E& lhs, E rhs)
{
    using underlying = typename std::underlying_type<E>::type;

    return lhs = static_cast<E>(static_cast<underlying>(lhs)
        | static_cast<underlying>(rhs));
};
```

How do we implement
operator| ()?

15

|operator|() as a template

```
template <class E>
constexpr E operator|(E lhs, E rhs)
{
    return lhs |= rhs;
}
```

repeat for
&=, &, ^=, ^, ~

16

| () as a template

- Pros
 - Seven short templates allow us to treat any enum like a bitfield
- Cons
 - We may not want to treat *all* enums like bitfields.
 - Potential clashes with other overloads of operator | (), such as std::async
 - Too greedy!
"some string" | "some other string"

would error on
"std::underlying_type<E>::type"

17

| SFINAE to the Rescue!

- **"Substitution failure is not an error"**
- Coined by Vandervoorde and Josuttis in *C++ Templates: The Complete Guide*
- Template type deduction takes place only for function (not type) templates.

If substituting the template parameters into the function declaration fails to produce a valid declaration then the template is removed from the function overload set *without causing a compilation error.*

18

Constrained Template

- A function template that is designed to be usable only for certain types (and SFINAE for other types) is called a ***constrained template***.
- There are a number of ways of creating this, but the `std::enable_if` type function is both easy to use and understand.
 - As of C++11 (via Boost)

19

`std::enable_if`

- Possible implementation:

```
template<bool B, class T = void>
struct enable_if {};
```

What is the type of
“T” in the *false* case?

- Partially specialized for the *true* case:

```
template<class T>
struct enable_if<true, T> { typedef T type; };
```

T is not defined in
the false case.

So, the substitution
fails (which is *not* an
error),

and the template is
removed from the
overload set.

20

|enable_bitmask_operators

```
template <class E>  
constexpr bool enable_bitmask_operators(E) { return false; }
```

This is a template
function not a
template class.
Why?

By default, always
false.
Requires opt in.

21

|operator|() as a template

```
template <class E>  
constexpr E operator|(E lhs, E rhs)  
{  
    return lhs |= rhs;  
}
```

22

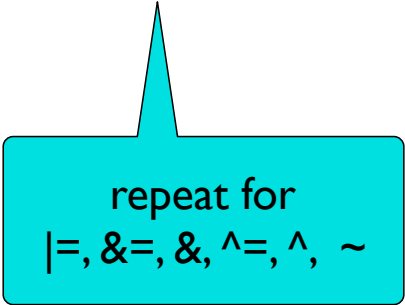
| operator | () as a template

```
template <class E>
constexpr
E
operator|(E lhs, E rhs)
{
    return lhs |= rhs;
}
```

23

| operator | () as a template

```
template <class E>
constexpr
typename std::enable_if<enable_bitmask_operators(E{})>, E>::type
operator|(E lhs, E rhs)
{
    return lhs |= rhs;
}
```

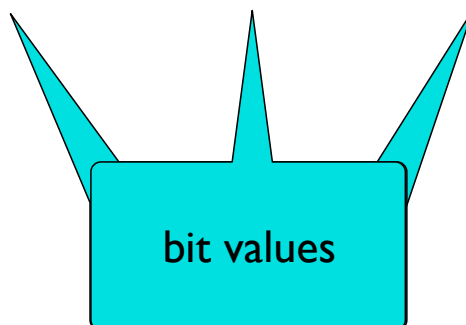


repeat for
|=, &=, &, ^=, ^, ~

24

| defining our bitfield enum struct

```
namespace user {  
    enum struct my_bitmask {first = 1, second = 2, third = 4};  
    ~~~  
}
```



25

| defining our bitfield enum struct

```
namespace user {  
    enum struct my_bitmask {first = 1, second = 2, third = 4};  
    constexpr bool enable_bitmask_operators(my_bitmask) {return true;}  
    ~~~  
}
```

This is an overload,
not a specialization.

This could have
been implemented
as a class template.

But the specialization
would have to be in the
original namespace.

26

using our bitfield enum struct

```
#include "user.hpp"
#include "bitmask.hpp"
#include "iostream"

int main()
{
    auto a(user::my_bitmask::first);
    auto b(user::my_bitmask::second);

    std::cout << "a | b: " << int(a | b) << "\n";
}
```

Won't compile!
Why?

invalid operands to binary
expression ('user::my_bitmask' and
'user::my_bitmask')

defining our bitfield enum struct

```
namespace user {
    enum struct my_bitmask {first = 1, second = 2, third = 4};
    constexpr bool enable_bitmask_operators(my_bitmask) {return true;}
    using bitmask::operator|;
    ~~~
}
```

repeat for
|, &=, &, ^=, ^, ~

pull the operator
into scope

defining our bitfield enum struct

```
namespace user {  
    enum struct my_bitmask {first = 1, second = 2, third = 4};  
    constexpr bool enable_bitmask_operators(my_bitmask) {return true;}  
    using bitmask::operator|;  
    using bitmask::operator|=;  
    using bitmask::operator&;  
    using bitmask::operator&;  
    using bitmask::operator^;  
    using bitmask::operator^=;  
    using bitmask::operator~;  
}
```

29

using our bitfield enum struct

```
int main()  
{  
    auto constexpr a(user::my_bitmask::first);  
    auto constexpr b(user::my_bitmask::second);  
  
    std::cout << "a | b: " << int(a | b) << "\n";  
    auto c(a);  
    std::cout << "c |= b: " << int(c | b) << "\n";  
    int k[static_cast<int>(a | b)];  
}
```

output:
a | b: 3
c |= b: 3

constexpr

30

| Thanks

- Anthony Williams - original article
- Louis Dionne - pulling operators into scope
- Jay Miller - using function overload rather than template specialization