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.

Classic C++ enum annoyances

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

Note: maintain backwards compatibility

Error prone!
### C++11 enum features

- Supports a new “underlying type” feature
  ```cpp
enum values: int {first, second};
```
- Optional: old syntax not broken
  ```cpp
  enum values {first, second};
  ```
- We can get the underlying type:
  ```cpp
typename std::underlying_type<values>::type
  ```

### enum struct

- Introduced in C++11
- Uses “proper” C++ scoping rules
  ```cpp
  enum struct values {first, second};
  a_value = values::first;
  ```
- Supports the new “underlying type” feature
  - Defaults to int
    ```cpp
    enum struct values {first, second};
    ```
  - Can also use “class”
    ```cpp
    enum class values: char {first, second};
    ```
- No implicit casting to int or other enums!
Classic C++ enum features

- Enums often as as bitfields:
  ```cpp
  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:
  ```cpp
  int i{first * third / second};
  ```
- nonsense!

enum struct

- The new enum struct syntax prevents this:
  ```cpp
  options opt{first * third / second};
  ```
- But also prevents this:
  ```cpp
  some_function(options(first | third));
  ```
- But it can be fixed.
**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:
  - |, &, ^, ~, |=, &=, ^=
- We just have to do this for the enums that we want to be bit fields.

**Operator |() as a template**

```cpp
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)
    );
}
```

What's wrong with this operator |()?
Operator Overloading Guideline

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

Our \texttt{operator|() should be defined in terms of operator|=().}

Why not define \texttt{operator|=() in terms of operator|()?}

\textbf{operator|=() as a template}

\begin{verbatim}
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;
}
\end{verbatim}

Won't compile! Why?

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

constexpr Guideline

- If it can be constexpr declare it constexpr
|operator|=() as a template

```cpp
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|()?`?

---

|operator|() as a template

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

repeat for `&=, &, ^=, ^, ~`
operator|( ) 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"

---

**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.**
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)

---

**std::enable_if**

- Possible implementation:
  ```cpp
template<bool B, class T = void>
struct enable_if {};
```
- Partially specialized for the `true` case:
  ```cpp
template<class T>
struct enable_if<true, T> { typedef T type; };
```

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

- 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.
enable_bitmask_operators

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

Why?

By default, always false. Requires opt in.

operator|() as a template

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

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

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 |=, &=, &, ^=, ^, ~
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.
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
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~;
}

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
Thanks

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