Using enum to manage bit sets works fine in C, but things become a little more complex in C++. **Kevlin Henney** explains an alternative method of handling flags, using the bitset template

Flag waiving

Kevlin Henney is an independent software development consultant and trainer. He can be reached at **www.curbralan.com**

Take a quick look at the following code:

```
typedef enum style
{
  plain,
  bold,
  italic,
  underline = 4,
  strikethrough = 8,
  small_caps = 16
} style;
```

What language is it in? It could be C or C++, although the style is clearly C-like; C++ does not need all that typedef noise to obtain a usable type name. The following code fragments fix the language:

```
style selected = plain;
...
selected |= italic;
...
if(selected & bold)
  ...
```

It's C. To be precise, it's common but not particularly good C. The code demonstrates a weakness of the type system that encourages sloppy design. Unfortunately, given the enduring C influence on C++ coding practices, this style for flags and sets of flags is also prevalent in C++. C++ programmers without a C background can acquire these habits by osmosis from their C-speaking colleagues or through C-influenced libraries.

Flag poll

First, some quick explanation and a little bit of reformatting. As its name suggests, an enum is normally used to enumerate a set of constants. By default, the enumerators have distinct values, starting at 0 and rising by 1 for each successive enumerator. Alternatively, an enumerator can be given a specific constant integer value.

A common approach for holding a set of binary options is to treat an integer value as a collection of bits, ignoring its numeric properties. If a bit at a particular position is set, the option represented by that position is enabled. This duality is common at the systems programming level and many programmers never think to question it. C programmers and, indeed, C compilers make little distinction between enumerators, integers, flags and bit sets.

An enum is often used to list the distinct options in a bit set, but instead of acting as distinct symbols, enumerator constants are used to represent bit masks. This brings us to the first bit of laziness to tidy up in the original enum definition. As it happens, the first three values form the sequence 0, 1 and 2, which specifies no bits, the first bit and the second bit respectively. Each integer power of 2, from 0 up, represents a different bit. To make it clear that the enumerators do not form a conventional sequence, but instead represent bit masks, developers typically set the mask values explicitly. As neither C nor C++ supports binary literals, it is more common to use hexadecimal rather than either decimal or octal to define the constants:

```
enum style
{
  plain,
  bold       = 0x01,
  italic     = 0x02,
  underline  = 0x04,
  strikethrough = 0x08,
  small_caps  = 0x10
};
```

Since the aim of this column is to reconsider how we work with flags in C++, I have also taken the small liberty of dropping the C-ish typedef. Let's look at another common approach to specifying the enumerators, that makes the basic idea of sequence (bit 0, bit 1, bit 2 etc) a little more explicit:

```
enum style
{
  plain,
  bold       = 1 << 0,
  italic     = 1 << 1,
  underline  = 1 << 2,
  strikethrough = 1 << 3,
```

FACTS AT A GLANCE

- C habits still affect C++ style, such as how to work with flags.
- C++'s stronger type checking makes the C use of enums as bit sets somewhat awkward.
- The standard library's bitset template provides a simpler and more direct implementation for sets of flag options.
- Programmers can define their own types for holding bit sets based on std::bitset.
- Trait classes and policy parameters allow for flexible implementation.

```
    small_caps   = 1 << 4
};
```

This approach makes the bit-masking purpose of the enumerators a little clearer. The relationship to integers and ordinary counting numbers is less interesting than the shifted position of a set bit. But what about the code that shows how the style flags are used? Alas, the following won't compile:

```
selected |= italic;
```

This fails because enum and integer types are not interchangeable: arithmetic and bitwise operators do not apply to enums. When you use an enum where an integer is expected, you get an implicit conversion to the enum's associated integer value, in effect:

```
static_cast<int>(selected) |= static_cast<int>(italic);
```

When integers go bad

While there is not much wrong with using an enum as an integer, there is plenty wrong with using an integer as an enum. Every enumerator will map to a valid integer value but not every valid integer will map to a valid enumerator. That's why C++ banned the implicit conversion from integer to enum and why the code shown won't compile. So how do you make it compile? You can force the compiler to succumb to your wicked way with a cast as your accomplice:

```
selected = static_cast<style>(selected | italic);
```

It would be fair to say this lacks both grace and convenience. Alternatively, you could overload the bitwise *or* operators to do your bidding:

```
style operator|(style lhs, style rhs)
{
   return static_cast<style>(int(lhs) | int(rhs));
}

style &operator|=(style &lhs, style rhs)
{
   return lhs = lhs | rhs;
}
```

This will allow the code to compile in its original form, but you should not forget to define the other relevant bitwise operators. One problem with this approach is that you need to define these operators anew for every enum type you want to use as a bit set. And what do you do for the definition of the bitwise *not* operator? This is used to ensure that a bit is disabled:

```
selected &= ~italic;
```

What should the value of ~italic be? bold | underline | strikethrough | small_caps or ~int(italic)? The former includes only bits that have been defined as valid for the bit set, but the latter is a simpler interpretation of the bit set concept.

A quick aside on cast style: the constructor-like form, eg int(italic) is used where the conversion is safe and would otherwise be implicit. You are constructing an int from a style. The keyword cast form, static_cast, is being used where the conversion is potentially unsafe and we are taking liberties with the type system.

Which brings us neatly to the next point: we are messing with the type system. There is an underlying reason why we are having to perform with a cast of thousands: the design is flawed. If you recall, the style type is an enumeration type. In other words, it enumerates the intended legal values of a style variable that is designed to hold one of the enumerator values at a time. However, the common practice uses a style variable to hold not an option but a set of options. While the bit representation of an enum affords such usage, it is by default a type error that is fundamentally a category error: a thing and a collection of things are different types.

This answers the question of why plain is not really a valid enumerator for style. It represents a combination of options (or, rather, their absence); it is a set rather than an individual option. The canonical C type for a bit set is the integer, signed or otherwise, but the C's lax type system allows the free and easy mixing of unrelated concepts. Here is the revised code:

```
const unsigned plain = 0;
...
unsigned selected = plain;
...
selected |= italic;
...
if(selected & bold)
   ...
```

However, what the other approaches lacked in grace, the integer bit set loses in safety and precision. There is nothing that constrains the actual use of an unsigned to be the same as its intended use.

Taking a step back from these many variations, you may realise the nagging truth: it's all a bit of a hack. Why should you be manually organising your constants according to bit masks at all? It is easy to become rooted in only one form of thinking. Sometimes a fresh look will help you break out of a rut to a more appropriate solution. Let's leave C behind.

It's time to get back to basics, ignoring all this bitwise gymnastics, and restate the core problem: you need to hold a set of options, each of which is effectively Boolean. This suggests a simple solution: hold a set of options.

```
enum style
{
   bold, italic, underline, strikethrough, small_caps
};
...
std::set<style> selected;
...
selected.insert(italic);
...
if(selected.count(bold))
   ...
```

This method is a lot simpler to work with: the style type simply enumerates the options with no hard coding of literal values. The standard set class template allows a set of options to be manipulated as a type-checked set rather than at the systems programming level. In other words, the language and library do all the work for you.

Honing for efficiency

Functionally, there are no problems with this approach, but many programmers may justifiably have concerns over its efficiency. The bitwise solution required an integer for storage, no additional dynamic memory usage, and efficient, fixed-time manipulation of the options. In contrast, an std::set is an associative node-based container that uses dynamic memory for its representation.

If you hold these space and time efficiency concerns (and have good reason to), all is not lost: you can still have abstraction and efficiency. The flat_set class template presented in the last column[1] is an improvement over std::set for this purpose. But better still is the standard bitset template:

```
std::bitset<5> selected;
...
selected.set(italic);
...
if(selected.test(bold))
   ...
```

The std::bitset class template, defined in the standard <bitset> header, is not a part of the STL, hence the seemingly non-standard member function names. But it is still standard and it still solves the problem. A more legitimate obstacle is that the size of the set must be wired in at compile time. There is also nothing that constrains you to working with the style type.

There is a simple hack that allows you to avoid having to count the number of enumerators:

```
enum style
{
   bold, italic, underline, strikethrough, small_caps,
   _style_size
};
...
std::bitset<_style_size> selected;
...
```

I say this is a hack because you are using the property of enumerators to count, by default, in steps of 1 from 0. The last enumerator, _style_size, is not really part of the valid set of enumerators because it does not represent an option. However, in its favour, this technique does save you from many of the ravages of change: the addition or removal of enumerators and any consequent change to the declared size of the bitset.

Rhyme and treason

Now I think we have a better idea of what is needed: something that works like a type-safe bitset for enums. Alas, we won't find one of these in the standard so we will have to create it. Let's start with the intended usage:

```
enum_set<style> selected;
...
selected.set(italic);
...
if(selected.test(bold))
   ...
```

It seems reasonable enough to follow the std::bitset interface because the two are conceptually related. Additionally, a const subscript operator would make its use even more intuitive:

```
if(selected[bold])
   ...
```

To save on all the bit twiddling, we can use std::bitset as the representation of our enum_set. However, a quick sketch reveals a problem:

```
template<typename enum_type>
class enum_set
{
   ...
private:
   std::bitset<???> bits;
};
```

How big should the bitset be? One unsatisfactory solution would be to require the user to provide the size as well as the type, ie:

```
enum_set<style, _style_size> selected;
```

This is clumsy and smacks of redundancy. We state the type and then a property of the type, relying in this case on a little hack to keep everything in sync. It would be nice to look up the relevant type properties based on the type name, so that the type is all the user has to specify in a declaration. C++'s reflective capabilities are quite narrow, limited to runtime-type information in the form of typeid and dynamic_cast and compile-time type information in the form of sizeof. However, the traits technique[2], first used in the C++ standard library, offers a simple form of customisable compile-time type information.

It is a myth that classes exist only to act as the DNA for objects. Classes can be used to group non-instance information in the form of constants, types or static member functions, describing policies or other types. A class template can be used to describe properties that relate to its parameters, specialising as necessary. This form of compile-time lookup allows us to answer neatly the question of the bitset's size:

```
template<typename enum_type>
class enum_set
{
   ...
private:
   typedef enum_traits<enum_type> traits;
   std::bitset<traits::count> bits;
};
```

Producing a rabbit with style

However, there is no such thing as magic. If you wish to pull a rabbit out of a hat, you had better have a rabbit somewhere to hand. Here is a definition of what the traits for the style type would look like:

```
template<>
struct enum_traits<style>
{
   typedef style enum_type;
   static const bool       is_specialized = true;
   static const style      first = bold;
   static const style      last  = small_caps;
   static const int        step  = 1;
   static const std::size_t count = last - first + 1;
};
```

It is common to use struct rather than class for traits because they do not represent the type of encapsulated objects, and a private-public distinction serves no useful purpose. The trait class just shown is a full specialisation of the primary template, which is effectively just a shell with non-functional placeholder values:

```
template<typename type>
struct enum_traits
{
   typedef type enum_type;
   static const bool       is_specialized = false;
   static const style      first = type();
   static const style      last  = type();
   static const int        step  = 0;
   static const std::size_t count = 0;
};
```

The information available to an enum_traits user includes the type of the enum; whether or not the trait is valid (has been specialised); the first and last enumerator values; the step increment, if any, between

each enumerator; and the count of the enumerators. This is all good information to have but it does seem like a lot of work: a separate specialisation is required for each enum type. Fortunately, we can provide a helper class to cover most of this ground:

```
template<
   typename type,
   type last_value, type first_value = type(),
   int step_value = 1>
struct enum_traiter
{
   typedef type enum_type;
   static const bool       is_specialized = true;
   static const type       first = first_value;
   static const type       last  = last_value;
   static const int        step  = step_value;
   static const std::size_t count =
      (last - first) / step + 1;
};
```

The enum_traiter template is designed for use as a base, reducing the enum_traits specialisation for style:

```
template<>
struct enum_traits<style> :
   enum_traiter<style, small_caps>
{
};
```

This makes life a lot easier, and accommodates enum types whose enumerators do not number from 0 and whose step is not 1. However, the common case is catered for with default template parameters, remembering that the explicit default construction for integers and enums is zero initialisation.

Wrapping and forwarding

The implementation for enum_set becomes a fairly simple matter of wrapping and forwarding to a std::bitset:

```
template<typename enum_type>
class enum_set
{
public:
   enum_set()
   {
   }
   enum_set(enum_type setting)
   {
      set(setting);
   }
   enum_set &operator&=(const enum_set &rhs)
   {
      bits &= rhs.bits;
      return *this;
   }
   enum_set &operator|=(const enum_set &rhs)
   {
      bits |= rhs.bits;
      return *this;
   }
   enum_set &operator^=(const enum_set &rhs)
   {
      bits ^= rhs.bits;
      return *this;
   }
   std::size_t count() const
   {
      return bits.count();
   }
   std::size_t size() const
   {
      return bits.size();
   }
   bool operator[](enum_type testing) const
   {
      return bits.test(to_bit(testing));
   }
   enum_set &set()
   {
      bits.set();
      return *this;
   }
   enum_set &set(enum_type setting, bool value = true)
   {
      bits.set(to_bit(setting), value);
      return *this;
   }
   enum_set &reset()
   {
      bits.reset();
      return *this;
   }
   enum_set &reset(enum_type resetting)
   {
      bits.reset(to_bit(resetting));
      return *this;
   }
   enum_set &flip()
   {
      bits.flip();
      return *this;
   }
   enum_set &flip(enum_type flipping)
   {
      bits.flip(to_bit(flipping));
      return *this;
   }
   enum_set operator~() const
   {
      return enum_set(*this).flip();
   }
   bool any() const
   {
      return bits.any();
   }
   bool none() const
   {
      return bits.none();
   }
   ...
private:
   typedef enum_traits<enum_type> traits;
   static std::size_t to_bit(enum_type value)
   {
      return (value - traits::first) / traits::step;
   }
   std::bitset<traits::count> bits;
};
```

```
...
```

There is one final tweak that, for no extra hassle, makes the code a little more generic. A common use of trait classes in the standard library is as policy parameters. The default is to use the default trait for a type, but the user could provide an alternative. The use of policies is a long-standing C++ technique that has matured over the last decade[3,4,5,6,7].

The only difference to the enum_set template would be to remove the traits typedef and add a defaulted parameter:

```
template<
   typename enum_type,
   typename traits = enum_traits<enum_type> >
class enum_set
{
   ...
};
```

This facility can be used to define sets on a subset of options:

```
struct simple_style : enum_traiter<style, underline>
{
};
...
enum_set<style, simple_style> selected;
...
```

This declaration of selected allows it to hold only bold, italic and underline options.

Looking to the standard library can be a good way of finding either a solution or the inspiration for one. There are a lot of programming tasks that are repetitive and tedious. Because of their repetitive nature they become part of the background hum of programming, often forgotten. Taking the opportunity to call into question certain C bit bashing tactics leads to the identification of new abstractions which, once implemented, reduce the density and quantity of code needed to express what are in essence high-level ideas. There is no need to jump through hoops when using the abstractions, although their implementation does call on some relatively advanced idioms.

How far could you take the use of enum_traits? It is possible to extend it to accommodate existing enum types that are defined in terms of a bit-shifted progression rather than an arithmetic one. Based on traits, you can also define iteration for enum types. For the moment, these are left as exercises for the reader. ■

References

1. Kevlin Henney, "Bound and Checked", *Application Development Advisor*, January 2002, available from www.appdevadvisor.co.uk
2. Nathan Myers, "Traits: A new and useful template technique", *C++ Report*, June 1995, available from www.cantrip.org
3. Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001
4. Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns*, Addison-Wesley, 1995
6. Kevlin Henney, "Making an Exception", *Application Development Advisor*, May 2001, available from www.appdevadvisor.co.uk
7. Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994