**Scoped Enumerations**

Hopefully by the time this article is published I will have finished a stripped-down version of the classic 1943 game that I have been working on over the last month. I think I have rewritten sections of the code four or five times as I learnt new tricks in C++ !

One ‘trick’ I discovered was the scoped enumeration (also known as an enumeration class) that provides additional functionality over the standard enumeration found in C++. This article briefly describes scoped enumerations in C++11 and then delves into a concept called ‘Operator Overloading’ which allows the standard functions like ++, --, <, > and == to be extended to suit your needs.

**What are Scoped Enumerations?**

Almost all programming languages provide a concept of an enumeration – a set of related constant values. In most languages, enumerations simply resolve to integer values and can be cast back and forth as needed. In C++ this casting is done implicitly and it is this capability of swapping from enumeration elements to the underlying value that can lead to problems. A simple enumeration that describes the suits in a deck of cards is shown below:

enum {

Spades,

Clubs,

Diamonds,

Hearts

} ;

int suitInPlay = Hearts;

suitInPlay++;

A couple of things to note about this code is that it is not ‘type safe’ due to the variable suitInPlay being declared as a simple integer and therefore being able to accept any valid int value rather than only those nominated in the enumeration. The increment is also problematic as it takes a valid declaration of the suitInPlay (Hearts which in turn has an underlying value of three) and then increments it to a value outside of the enumeration.

C++ 11 extends the basic enumeration type with scoped enumerations that overcomes this problem. The following code shows a scoped enumeration definition:

enum class Suits {

Spades,

Clubs,

Diamonds,

Hearts

} ;

Suits suitInPlay Suits::Clubs;

Attempting to increment the variable or convert it to a numeric will fail. The declaration does not imply an integral data type - byte, integer or long - and the developer should not make any assumptions as to the type.

The underlying data type can be specified if there is a need to do so, as shown below.

enum class Suits : **uint8\_t** {

Spades,

Clubs,

Diamonds,

Hearts

} ;

Suits suitInPlay = Clubs;

Serial.print("suitInPlay = ");

Serial.println((uint8\_t)suitInPlay);

Output:

suitInPlay = 3;

However, attempting to assign a numeric value to a variable of type Suits will fail. Likewise attempting to assign an element of the Suits enumeration to an integer variable will also fail.

**Overloading Operators**

One major benefit of using a class enumerator is the ability to override the standard operators like ++, --, <, > and == to suit your needs.

In my 1943 game, the enemy boats have gun turrets that rotate to point towards the player as they pass each other. To record the direction of the turret, I have created a scoped class called Direction as shown below. As you would expect, the enumeration lists the various directions from North to NorthEast in order. In my code, the actual values assigned to these elements are irrelevant and the default value of zero has been assigned to North, one to NorthEast and so forth up with NorthWest having a value of 7. I have an additional direction, called Unknown, which is assigned the value of 8.

enum class Direction : uint8\_t {

North,

NorthEast,

East,

SouthEast,

South,

SouthWest,

West,

NorthWest,

Unknown

};

**A simple override ..**

Values in an enumeration can be compared together using the standard equality and inequality comparisons. However, the introduction of the Unknown element in the Direction enumeration allows me to illustrate how we can override the standard operation. When comparing two directions together, the comparison should return false if either or both arguments are equal to Direction::Unknown. The following operator override demonstrates this:

inline bool operator==(const Direction lhs, const Direction rhs) {

return ((uint8\_t)lhs == (uint8\_t)Direction::Unknown

? false

: (uint8\_t)lhs == (uint8\_t)rhs);

}

Putting this operator to work reveals the following results:

Direction a = Direction::East;

Direction b = Direction::East;

if (a == b) { Serial.println("1 a is equal to b"); }

a = Direction::NorthWest;

b = Direction::North;

if (!(a == b)) { Serial.println("2 a is not equal to b"); }

a = Direction::Unknown;

b = Direction::Unknown;

if (!(a == b)) { Serial.println("3 a is not equal to b"); }

Output:

1 a is equal to b

2 a is not equal to b

3 a is not equal to b

Note that in the last test, the comparison between the two Unknown directions returned false due to the rule created in or equality comparison.

Hang on, what’s with the (!(a == b)) syntax? What about using the standard ‘not equals’ syntax? As we have overridden the equality operator we need to override the inequality operator as well. The inequality operator shown below simply calls the modified equality operator and returns the NOT of it.

inline bool operator!=(const Direction lhs, const Direction rhs) {

return ! (lhs == rhs);

}

With this new operator override in place, we can use the standard notation:

...

a = Direction::NorthWest;

b = Direction::North;

if (a != b) { Serial.println("2 a is not equal to b"); }

a = Direction::Unknown;

b = Direction::Unknown;

if (a != b) { Serial.println("3 a is not equal to b"); }

Output:

1 a is equal to b

2 a is not equal to b

3 a is not equal to b

**Getting more complex ..**

So now we can compare directions, what about comparing values?

Assume the boat is at the bottom of the screen entering from the right and leaving towards the left. If the player is stationary and located at the top-centre of the screen then to shoot towards the player the gun will need to be pointed towards the North-West initially and will swing through North and then towards the North-East as the boat leaves the screen. In my code, I was hoping to compare the current direction of the turret to the player’s position and simply add or subtract one to swing it around as required. Although the enumeration details an order to the directions where South is greater than SouthEast (in a clockwise view of the world), it does not consider that North is greater than NorthWest, ie it has no concept of the circular nature of a compass.

The following operators override the standard ‘less than’ and ‘greater than’ operators. Before calculating a return value, the operator determines whether it needs to consider the resetting of values at North by subtracting the integer values from each other. Where the comparison must cater for North, adding 8 (a full circle) to the either the right or left hand side argument (for ‘less than’ or ‘greater than’ respectively) ensures that the comparison caters for this appropriately.

inline bool operator<(const Direction lhs, const Direction rhs) {

return (abs((uint8\_t)lhs - (uint8\_t)rhs) < 4 ? (uint8\_t)lhs - (uint8\_t)rhs : (uint8\_t)lhs - (8 + (uint8\_t)rhs)) < 0;

}

inline bool operator>(const Direction lhs, const Direction rhs) {

return (abs((uint8\_t)lhs - (uint8\_t)rhs) < 4 ? (uint8\_t)lhs - (uint8\_t)rhs : (8 + (uint8\_t)lhs) - (uint8\_t)rhs) > 0;

}

Putting these two operators through their paces reveals the correct results.

Direction a = Direction::East;

Direction b = Direction::South;

if (a < b) { Serial.println("1 East is less than South"); }

if (b > a) { Serial.println("2 South is greater than East"); }

a = Direction::NorthWest;

b = Direction::North;

if (a < b) { Serial.println("3 NorthWest is less than North"); }

if (b > a) { Serial.println("4 North is greater than NorthWest"); }

a = Direction::NorthWest;

b = Direction::NorthEast;

if (a < b) { Serial.println("5 NorthWest is less than NorthEast"); }

if (b > a) { Serial.println("6 NorthEast is greater than NorthWest"); }

a = Direction::North;

b = Direction::NorthEast;

if (a < b) { Serial.println("7 North is less than NorthEast"); }

if (b > a) { Serial.println("8 NorthEast is greater than North"); }

Output:

1 East is less than South

2 South is greater than East

3 NorthWest is less than North

4 North is greater than NorthWest

5 NorthWest is less than NorthEast

6 NorthEast is greater than NorthWest

7 North is less than NorthEast

8 NorthEast is greater than North

Now that we have the ‘equality’, ‘greater than’ and ‘less than’ operators in place, we can add the following two operators to complete the comparisons. Note they perform the logical NOT of those operators previously defined.

inline bool operator<=(const Direction lhs, const Direction rhs) { return !(lhs > rhs); }

inline bool operator>=(const Direction lhs, const Direction rhs) { return !(lhs < rhs); }

**And just plain confusing ..**

Not really confusing but overriding the standard increment and decrement operators reveal an interesting compiler trick that you just have to take for granted actually works.

The operator below describes a **pre-increment** operator. If the supplied direction is NorthWest it returns North otherwise it returns the direction adjacent to the supplied on in the enumeration.

inline Direction &operator++( Direction &c ) {

c = ( c == Direction::NorthWest )

? Direction::North

: static\_cast<Direction>( static\_cast<uint8\_t>(c) + 1 );

return c;

}

A **post-increment** operator includes an anonymous second parameter of type integer. This parameter is not used in the code and is simply there to differentiate between the pre- and post- increment functions. The compiler must recognise this method signature and assume it is a post-increment function. The implementation of this operator simply calls the pre-incrementer and returns its value.

inline Direction &operator++( Direction &c, int ) {

Direction result = c;

++c;

return result;

}

The decrement functions are nearly identical to the increment operators and are shown below for completeness.

inline Direction & operator--( Direction & c ) {

c = ( c == Direction::North )

? Direction::NorthWest

: static\_cast<Direction>( static\_cast<uint8\_t>(c) - 1 );

return c;

}

inline Direction operator--( Direction & c, int ) {

Direction result = c;

++c;

return result;

}

**Conclusion**

As you can see, the scoped enumerations provide an extra layer of functionality beyond what simple enumerations offer. The ability to override the operators encapsulates the logic of the enumeration itself and saves the developer having to repeat similar logic across an application. In the examples shown in this article, the developer does not need to remember the intrinsic values underlying the directions North or East nor does he need to consider how to increase the value when it reaches the end of the enumeration. In fact, all of the code would work if you change the initial enumeration declaration to (say) North = 57 ..

I have included a simple application on GitHub that shows the enumeration in action :

<https://github.com/filmote/ScopedEnums>