When enum Just Isn't Enough: Enumeration Classes for C++

Here's a more intelligent mechanism for enumerating resources in C++.

May 01, 2003
URL:http://www.drdobbs.com/when-enum-just-isnt-enough-enumeration-c/184403955

Accessing Resources through Symbolic Constants

If you're writing even a moderately ambitious application, you will find that it needs to access many resources of various types, for example sounds and text messages. In a large application, individual resources can easily number in the hundreds. Typically, they will be referenced by a numeric value. For the time being, I'll suppose you're using `int`s to refer to resources, but I'll generalize later.

Assume that you're working on a game, which uses pre-rendered graphics, player messages, and sound effects. High-level functions allow you to write `PlaySound(1001);` and `DisplaySprite(435);`, but of course you wouldn't use magic numbers. You might declare symbolic constants instead, as in `const int explosionSnd = 1001;`. This is definitely better, but not type-safe: nothing prevents you from writing `DisplaySprite(explosionSnd);` when you meant to write `DisplaySprite(explosionSprite);`.

The traditional solution to this lack of type-safety is to group your constants inside `enum`s. Given an `enum` for sounds, `e_Sounds`, and one for sprites, `e_Sprites`, you can declare a function to take an argument of the proper type, as in `void DisplaySprite(e_Sprite);`. Once this has been done, the compiler will ensure that you cannot accidentally pass a sound constant to a sprite display function.

Enumerations Don't Always Fit the Bill

You might feel this is the best you can do, and all that needs to be done. But the `enum`-based approach is still insufficient for serious applications. The problem is that `enum` is an unintelligent structure: C++ does not allow you to programmatically extract anything significant from it. This makes it impossible to automate operations on `enum`, when this is precisely what needs to be done for many applications. Consider that huge game with its hundreds of sounds: You'd certainly like to be able to verify that every sound constant referenced in the code corresponds to a sound that is actually available in the current version of the application. The inverse task is also important: If those 25 new sounds that were just added in the new version are never even played, you'd want your automated tests to notice that.

Now it is possible to iterate over `enum`, so you might think that the first test, at least, is trivial: just write a `for` loop. Listing 1 illustrates this ideal situation.

Listing 1

```
enum Sounds {
  explosionSnd = 101,
  attackSnd = 102,
  ... // each constant increases by 1
  victorySnd = 185
}

Sounds firstSnd = explosionSnd;
Sounds lastSnd = victorySnd;

for ( int theSound = firstSnd;
theSound <= lastSnd; ++theSound )
  CheckSoundIsPresent(theSound);
```

Real-world programs do not look anything like this, however. Both initial design considerations and maintenance issues conspire to produce a situation like that in Listing 2. When you've got even one "hole" in your `enum`, you're in trouble.

Listing 2

```
enum Sounds {
  // Human sounds in [100, 200[
  explosionSnd = 101,
  attackSnd = 102,
  // ricochetSnd = 103, // removed, beta testers hated it
  deathSnd = 104,
  // Alien sounds in [200, 300[
  alienAttackSnd = 201,
  ...
  alienDeathSnd = 227,
  // Miscellaneous sounds >= 500
  victorySnd = 500
};
```

Using straightforward iteration with `int`s over an enumeration that is not continuous is legal C++; it just gives erroneous results: intermediate, "nameless" values are legal even though they are meaningless. Now, since `enum` is a user-defined type, it is possible to define an increment/decrement operator for it [1]. This makes it possible to iterate through `enum` using a `for` loop with a specific `operator++`. However, this is a false solution: It can indeed be very useful for small, continuous `enum`s — for instance, if you have an `enum` for the days of the week, it's very convenient to define an `operator++` that always returns the next day of the week — but with a large, discontinuous enumeration things go rapidly sour.

Suppose you've managed to write an `operator++` for a discontinuous `enum` — a tedious and error-prone task in the first place. Even then, maintenance issues are still a nightmare: If someone adds a sound constant to the `e_Sounds enum` in an "empty slot," your `operator++` will skip right over it without warning. If this constant is somehow passed to your `operator++`, it will fail to recognize it and at best give a wrong result, if it doesn't cause a runtime error instead.

Even if you ignore these difficulties, the second test you wanted to run remains infeasible: You still cannot tell whether a given value is a valid part of an enumeration or not. You could of course write a custom function to do this, and then you'd suffer from the very same maintenance nightmares I was just talking about. Unless you enjoy wasting hours debugging, something better is clearly indicated.

An Industrial-Strength Solution

What you need is an intelligent enumeration: one that knows its minimum and maximum and its valid ranges. Such an enumeration will allow you to iterate over it automatically, because it will furnish its own standard mechanism for doing so, which will obviate the need to write and maintain one yourself. In other words, you need a class that allows you to unify symbolic constants into a single type.

The basic idea is to make each constant of your `enum` be a class-static constant instance of a class. This class will keep a sorted list of its own instances in a class-static container; STL's `std::set` is ideal since it keeps its contents sorted. Iterating over the enumeration becomes a matter of iterating over the set. Since the set is sorted, it becomes trivial to obtain the enumeration's maximum and minimum. This design is proof against maintenance problems, since the set is built from the constants defined in the code: whether constants are added or removed, the set will always be correctly built.

There is a potential thorn in your side: it's crucially important that it be impossible to add to the enumeration during the course of the program. However, you may want to have local variables of the type of the enumeration, just like with regular `enum`s. The first requirement seems to imply you can only construct class instances before `main` is executed, and the other requires construction during the course of the program. In fact this simply forces you to distinguish between class-static instances (constructed before `main` is executed from their `int` value, immutable) and local instances (constructed dynamically by copying from a class-static instance, mutable).

I've made use of templates for implementing this design, since you'll typically use more than one such enumeration class. This template is meant to be used with the well-known idiom (which as far as I know lacks a standard name) whereby class `A` is derived from a class `B` templated on `A` itself.

Listing 3 presents a class template that implements the desired features. The class has a `private:` constructor from `int`. Only the class-static instances may be defined this way. The copy constructor (synthesized by the compiler) makes it possible to instantiate local variables that are copies of class-static instances. The class keeps a set of its static instances, which is always sorted. STL thus does all the hard work for you, and your implementation can remain simple and intuitive, or nearly so.

Listing 3

```
#include <functional>
#include <set>
template <class T>
class Enum {
private:
  // Constructors
  explicit Enum(int Value);
  // Predicate for finding the corresponding instance
  struct Enum_Predicate_Corresponds:
   public std::unary_function<const Enum<T>*, bool> {
      Enum_Predicate_Corresponds(int Value): m_value(Value) { }
      bool operator()(const Enum<T>* E)
      { return E->Get_Value() == m_value; }
   private:
      const int m_value;
  };
  // Comparison functor for the set of instances
  struct Enum_Ptr_Less:
   public std::binary_function<const Enum<T>*, const Enum<T>*, bool> {
      bool operator()(const Enum<T>* E_1, const Enum<T>* E_2)
      { return E_1->Get_Value() < E_2->Get_Value(); }
  };
public:
  // Compiler-generated copy constructor and operator= are OK.
  typedef std::set<const Enum<T>*, Enum_Ptr_Less> instances_list;
  typedef instances_list::const_iterator const_iterator;
  // Access to int value
  int Get_Value(void) const { return m_value; }
  static int Min(void) { return (*s_instances.begin())->m_value; }
  static int Max(void) { return (*s_instances.rbegin())->m_value; }
  static const Enum<T>* Corresponding_Enum(int Value)
  { const_iterator it = find_if(s_instances.begin(), s_instances.end(),
   Enum_Predicate_Corresponds(Value));
   return (it != s_instances.end()) ? *it : NULL; }
  static bool Is_Valid_Value(int Value) { return Corresponding_Enum(Value) != NULL; }
  // Number of elements
  static instances_list::size_type size(void) { return s_instances.size();
}
  // Iteration
  static const_iterator begin(void) { return s_instances.begin(); }
  static const_iterator end(void) { return s_instances.end(); }
private:
  int m_value;
  static instances_list s_instances;
};
template <class T>
inline Enum<T>::Enum(int Value):
  m_value(Value)
{
  s_instances.insert(this);
}
```

What may not be so intuitive are the two `struct`s in the private section, `Enum_Predicate_Corresponds` and `Enum_Ptr_Less`. The first is a predicate for use with `find_if` in the `Corresponding_Enum` method. It's not strictly necessary, since I could have used an explicit loop comparing the members of the set with the sought value, but `find_if` is better style. The second `struct`, `Enum_Ptr_Less`, is absolutely necessary to keep your set sorted in the order you want. As Scott Meyer points out in [2], you have to be careful when dealing with associative containers of pointers. The containers will by default be sorted by comparing the *pointer* values, which is almost never what you want. So you have to give the set a comparison object whose `operator()`, when given two members of the set, will return whether the first one is less than the other according to the appropriate criterion. In our example, you need to compare the results of the `Get_Value` function.

Listing 4 gives an example of how the "enumeration class" may be used in a simple test driver. This code was compiled and run on Metrowerks CodeWarrior 7.

Listing 4

```
Test_Enum.h

class Test_Enum: public Enum<Test_Enum> {

private:
 explicit Test_Enum(int Value): Enum<Test_Enum>(Value) { }

public:
  static const Test_Enum enum_11;
  static const Test_Enum enum_12;
  static const Test_Enum enum_18;
  static const Test_Enum enum_20;
  static const Test_Enum enum_21;
};

Test_Enum.cpp

#include "Test_Enum.h"

Enum<Test_Enum>::instances_list Enum<Test_Enum>::s_instances;

const Test_Enum Test_Enum::enum_11(11);
const Test_Enum Test_Enum::enum_12(12);
const Test_Enum Test_Enum::enum_18(18);
const Test_Enum Test_Enum::enum_20(20);
const Test_Enum Test_Enum::enum_21(21);

Main.cpp

#include "Test_Enum.h"
int main()
{
  using std::cout;
  using std::endl;
  int Cur_Elem = 0;
  for ( Test_Enum::const_iterator i = Test_Enum::begin();
        i != Test_Enum::end(); ++i )
  {
   Cur_Elem++;
   cout << "Test_Enum element #" << Cur_Elem << " value = "
   << (*i)->Get_Value() << endl;
  }

  cout << "Total #elements = " << Test_Enum::size() << endl;
  cout << "Min enum value = " << Test_Enum::Min() << endl;
  cout << "Max enum value = " << Test_Enum::Max() << endl;

  for ( int i = Test_Enum::Min(); i <= Test_Enum::Max(); ++i )
  {
   cout << i;
   if ( Test_Enum::Is_Valid_Value(i) ) cout << " is ";
   else cout << " isn't ";
   cout << "a valid value for Test_Enum." << endl;
  }

  Test_Enum E(Test_Enum::enum_11);
  cout << "Value of E = " << E.Get_Value() << endl;

  E = Test_Enum::enum_20;
  cout << "Value of E = " << E.Get_Value() << endl;

  // Illegal code
  // bool B = E; // won't compile, illegal implicit conversion
  // E++; // won't compile, cannot treat Test_Enum as an int
  // Test_Enum X(17); // won't compile, ctor is private

  return 0;
}

Output:

Test_Enum element #1 value = 11
Test_Enum element #2 value = 12
Test_Enum element #3 value = 18
Test_Enum element #4 value = 20
Test_Enum element #5 value = 21
Total #elements = 5
Min enum value = 11
Max enum value = 21
11 is a valid value for Test_Enum.
12 is a valid value for Test_Enum.
13 isn't a valid value for Test_Enum.
14 isn't a valid value for Test_Enum.
15 isn't a valid value for Test_Enum.
16 isn't a valid value for Test_Enum.
17 isn't a valid value for Test_Enum.
18 is a valid value for Test_Enum.
19 isn't a valid value for Test_Enum.
20 is a valid value for Test_Enum.
21 is a valid value for Test_Enum.
Value of E = 11
Value of E = 20
```

Advantages and Drawbacks

Enumeration classes have the following advantages over `enum`s: You can programmatically access their minimum and maximum values, test whether a given `int` is a valid enumeration constant, and iterate over a given enumeration class. An enumeration class instance is never implicitly converted to an `int` value, although this value may be obtained explicitly. Given an `int` value, you can obtain the corresponding enumeration class constant, if it exists.

Enumeration classes do have some drawbacks: for one, they are more verbose to define (but not to use) than `enum`s. For another, they cannot be used directly in a switch statement, although you can of course switch using the `Get_Value` function. Even though these are not major handicaps, the fact remains that using enumeration classes is less straightforward than using ordinary `enum`s.

Keep in mind, however, that enumeration classes are not meant to blindly replace all `enum`s in existing code, but rather to replace those that correspond to collections of related constants rather than symbolic representations of unified abstract concepts. In other words, if you're using an `enum` to group together the numbers of all the error messages your application uses, you should consider an enumeration class instead; but a days-of-the-week enum should remain a plain `enum`.

Generalization: Why Limit Yourself To ints?

What if you refer to resources using something more sophisticated than a dumb `int`? What use is an enumeration class when your sound-playing code reads `PlaySound("Phaser1");`? Well, in fact, an enumeration class is just what you need. The basic ideas I've presented here apply just as well when the underlying enumerated type is something other than `int` — as long as it can be ordered (i.e., it has or can have an `operator<`). The benefits can be even greater, since an enumeration class always allows iteration over the underlying type even when this is impossible in the general case. In practice, this means an enumeration class enables you to iterate over every sound-resource string in your application, from "Arrakis" to "Zabulon Computation."

A generalized enumeration class is templated on its underlying type. Listing 5 provides such a class.

Listing 5

```
#include <functional>
#include <set>
```

```
template <class TValue, class T>
class Tmpl_Enum {

private:

 // Constructors
 explicit Tmpl_Enum(const TValue& Value);

 // Predicate for finding the corresponding instance
 struct Enum_Predicate_Corresponds:
    public std::unary_function<const Tmpl_Enum<TValue, T>*, bool> {

      Enum_Predicate_Corresponds(const TValue& Value): m_value(Value) {
}
      bool operator()(const Tmpl_Enum<TValue, T>* E)
      { return E->Get_Value() == m_value; }

    private:
      const TValue& m_value;
 };

 // Comparison functor for the set of instances
 struct Enum_Ptr_Less:
    public std::binary_function<const Tmpl_Enum<TValue, T>*, const Tmpl_Enum<TValue, T>*, bool> {
      bool operator()(const Tmpl_Enum<TValue, T>* E_1, const Tmpl_Enum<TValue, T>* E_2)
      { return E_1->Get_Value() < E_2->Get_Value(); }
 };

public:
 // Compiler-generated copy constructor and operator= are OK.

 typedef std::set<const Tmpl_Enum<TValue, T>*, Enum_Ptr_Less> instances_list;
 typedef instances_list::const_iterator const_iterator;

 // Access to TValue value
 const TValue& Get_Value(void) const { return m_value; }
 static const TValue& Min(void) { return (*s_instances.begin())->m_value; }
 static const TValue& Max(void) { return (*s_instances.rbegin())->m_value; }
 static const Tmpl_Enum<TValue, T>* Corresponding_Enum(const TValue& Value)
 { const_iterator it = find_if(s_instances.begin(), s_instances.end(), Enum_Predicate_Corresponds(Value));
    return (it != s_instances.end()) ? *it : NULL; }
 static bool Is_Valid_Value(const TValue& Value) { return Corresponding_Enum(Value) != NULL; }

 // Number of elements
 static instances_list::size_type size(void) { return s_instances.size(); }

 // Iteration
 static const_iterator begin(void) { return s_instances.begin(); }
 static const_iterator end(void) { return s_instances.end(); }

private:
 TValue m_value;

 static instances_list s_instances;
};


template <class TValue, class T>
inline Tmpl_Enum<TValue, T>::Tmpl_Enum(const TValue& Value):
 m_value(Value)
{
 s_instances.insert(this);
}
```

Listing 6 gives a sample testing program together with its output. The generalization is fairly straightforward: You must add another template parameter for the underlying type, which I've called `TValue`. To avoid copy inefficiencies, `TValue` is always passed by reference to `const` and likewise the predicate stores a `const TValue&`.

Listing 6

```
Test_Enum_String.h

#include <string>
#include "Tmpl_Enum.h"

class Test_Enum_String:
  public Tmpl_Enum<std::string, Test_Enum_String> {

private:

  explicit Test_Enum_String(const std::string& Value):
    Tmpl_Enum<std::string, Test_Enum_String>(Value) { }

public:

  static const Test_Enum_String enum_Alpha;
  static const Test_Enum_String enum_Beta;
  static const Test_Enum_String enum_Delta;
  static const Test_Enum_String enum_Epsilon;
  static const Test_Enum_String enum_Omega;
};

Test_Enum_String.cpp

#include "Test_Enum_String.h"
Tmpl_Enum<std::string, Test_Enum_String>::instances_list
Tmpl_Enum<std::string, Test_Enum_String>::s_instances;

const Test_Enum_String Test_Enum_String::enum_Alpha("Alpha");
const Test_Enum_String Test_Enum_String::enum_Beta("Beta");
const Test_Enum_String Test_Enum_String::enum_Delta("Delta");
const Test_Enum_String Test_Enum_String::enum_Epsilon("Epsilon");
const Test_Enum_String Test_Enum_String::enum_Omega("Omega");

Main.cpp

#include "Test_Enum_String.h"
void Validate_Str_Elem(const std::string& theString)
{
  using std::cout;
  using std::endl;
  cout << theString;
  if ( Test_Enum_String::Is_Valid_Value(theString) )
    cout << " is ";
  else
    cout << " isn't ";
  cout << "a valid value for Test_Enum_String." << endl;
}

int main()
{
  using std::cout;
  using std::endl;
  int Cur_Elem = 0;
  for ( Test_Enum_String::const_iterator
        i = Test_Enum_String::begin();
        i != Test_Enum_String::end(); ++i )
  {
    Cur_Elem++;
    cout << "Test_Enum_String element #" << Cur_Elem << " value = " << (*i)->Get_Value() << endl;
  }

  cout << "Total #elements = " << Test_Enum_String::size() << endl;

  cout << "Min enum value = " << Test_Enum_String::Min() << endl;
  cout << "Max enum value = " << Test_Enum_String::Max() << endl;

  Validate_Str_Elem("Alpha");
  Validate_Str_Elem("Gamma");
  Validate_Str_Elem("Beta");
  Validate_Str_Elem("BetA");

  Test_Enum_String E(Test_Enum_String::enum_Alpha);
  cout << "Value of E = " << E.Get_Value() << endl;

  E = Test_Enum_String::enum_Omega;
  cout << "Value of E = " << E.Get_Value() << endl;
  }
  return 0;
}

Output:

Test_Enum_String element #1 value = Alpha
Test_Enum_String element #2 value = Beta
Test_Enum_String element #3 value = Delta
Test_Enum_String element #4 value = Epsilon
Test_Enum_String element #5 value = Omega
Total #elements = 5
Min enum value = Alpha

Max enum value = Omega
Alpha is a valid value for Test_Enum_String.
Gamma isn't a valid value for Test_Enum_String.
```

```
Beta is a valid value for Test_Enum_String.
BetA isn't a valid value for Test_Enum_String.
Value of E = Alpha
Value of E = Omega
```

If the underlying type does not have an `operator<` (or if this operator doesn't do what you want), you can still use an enumeration class, but you will have to generalize still further, with a third template parameter, this one a comparison object to pass to the `Enum_Ptr_Less` object. The details are left as an exercise for the reader.

Conclusion

Enumeration classes enable the encapsulation of collections of related constants. They allow programmatic access to the minimum and maximum values of the collection as well as straightforward iteration and membership tests. In their simpler form, they can be viewed as an improved form of `enum`. By templating them on their underlying enumerated type, they can be applied to collections of constants of any ordered type.

Bibliography

[1] Bjarne Stroustrup. *The C++ Programming Language*, 3d Edition, (Addison-Wesley, 2000), p. 265.

[2] Scott Meyers. *Effective STL* (Addison-Wesley, 2001), Item 20, p. 88.

Download the Code

<meynard.zip>

Yves Meynard has a Ph.D. in Computer Science (Operational Research) from the Université de Montréal. He holds the post of Software Analyst at SNC Lavalin Energy Control Systems. His areas of interest include optimization heuristics and code refactoring. He may be reached at ymeynard@globetrotter.net.