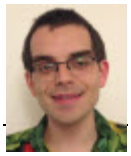
Enumerated Types - enums



By Alex Allain

Sometimes as programmers we want to express the idea that a variable will be used for a specific purpose and should only be able to have a small number of values--for instance, a variable that stores the current direction of the wind might only need to store values corresponding to north, south, east, and west. One solution to this problem might be to use an int and some #define'd values:

```
#define NORTH_WIND        0
#define SOUTH_WIND        1
#define EAST_WIND         2
#define WEST_WIND         3
#define NO_WIND           4

int wind_direction = NO_WIND;
```

The problem with this approach is that it doesn't really prevent someone from assigning a nonsensical value to wind_direction; for instance, I could set wind_direction to 453 without any complaints from my compiler. And if I looked at the type of wind_direction, i would see that it's just a plain old integer. there's just no way to know that something is wrong.

The idea behind enumerated types is to create new data types that can take on only a restricted range of values. Moreover, these values are all expressed as constants rather than magic numbers--in fact, there should be no need to know the underlying values. The names of the constants should be sufficient for the purposes of comparing values.

When you declare an enumerated type, you specify the name of the new type, and the possible values it can take on:

```
enum wind_directions_t {NO_WIND, NORTH_WIND, SOUTH_WIND, EAST_WIND, WEST_WIND};
```

Note the _t at the end of the name of the type: this stands for "type" and is a way to visually distinguish the name of the type from the name of variables. Your text editor may also have the ability to use syntax highlighting to make the new type look like other built-in types, such as int, for you.

Now we can declare a wind_directions_t variable that can only take on five values:

```
wind_directions_t wind_direction = NO_WIND;

wind_direction = 453; // doesn't work, we get a compiler error!
```

Note to C Programmers: If you're planning on using enums in C, however, you don't get this type safety. The above assignment will compile without giving you an error. By the way, if you're using enums in C, you will also need to prefix the declaration with the keyword enum: enum wind_directions_t wind_direction = NO_WIND;

You might be wondering exactly what values the constants take on--what if you wanted to compare then using < or >? You actually have a choice: if you want to set the values yourself, you may, or you can choose to use default values, which start at zero for the first constant and increase by one. In our example, NO_WIND has the value 0, and WEST_WIND has the value 4 (just like our #define'd constants).

On the other hand, we could reverse this by giving explicit values:

```
enum wind_directions_t {NO_WIND = 4, NORTH_WIND = 3, SOUTH_WIND = 2, EAST_WIND = 1, WEST_WIND = 0};
```

Why would you ever want to give explicit values to elements of an enumerated type? Isn't the whole point of constants so that you don't need to know what the values are? The answer is that if the values of the constant are never used outside of comparisons between elements of the enumeration, then there's almost no reason to define the values to be anything in particular (one exception is if you want one value to have multiple names, you'd have to set at least one value explicitly). But if you need the values for communicating with the outside world, you might need to give specific values. For example, if you decided to use an enum to store all of the possible text colors you could pass into a function to set the text colors, you'd probably need to make sure that the enum names,

such a RED or BLUE, matched up to the values corresponding to those colors.

Printing Enums

You might wonder what happens when you print out an enum: by default, you'll get the integer value of the enum. If you want to do something fancier than that, you'll have to handle it specially.

Naming Enums

One issue with enums is that the name of the enumerated type doesn't show up along with the enum. When you use the enum constant, it could really mean anything. The problem is that if you give your enums names that are too general, you can run into problems. First, it becomes hard to tell which enumeration a constant belongs to if you have several enumerated lists of values. A related problem is that sometimes you really want to use the same name. For instance, what if you had two color schemes, each of which included the color red, but for which the value of the RED constant needed to be different?

The solution to both of these problems is to include part of the name of the enum in the names of the constants. Notice that in the above example, I included "WIND" in the name of each enumerated constant. (Perhaps this wasn't entirely necessary--why not just have an enum for each ordinal direction? The answer is that it depends on whether someone else is already using the name. In this case, we avoid the problem by making the names specific enough that it's unlikely someone else will have a WEST_WIND constant.

Type Correctness

Because enums are "integer-like" types, they can safely be assigned into an integer without a cast. For instance, both of the following assignments are totally valid:

```
int my_wind = EAST_WIND;
```

or

```
wind_directions_t wind_direction = NO_WIND;

int my_wind = wind_direction;
```

As already mentioned, you can't make the reverse assignment in C++ without using a typecast. There might be times when you do need to do this, but you'd like to avoid it as best you can. For instance, if you need to convert a user's input into an enumerated type, it would be a bad idea to just typecast the variable to an int:

```
wind_directions_t wind_direction = NO_WIND;

std::cin >> static_cast<int>( wind_direction );
```

This would let the user input any value at all and, almost as bad, force the user to know the range of values that the enum could take on. A much better solution would be to shield the user from the enumeration by asking for a string and then validating the input by comparing it to the possible input strings to choose which constant to assign the enum. For instance,

```
std::cout << "Please enter NORTH, SOUTH, EAST, WEST, or NONE for our wind direction";
std::cout << std::endl;

string input_wind_dir;
cin >>

wind_directions_t wind_dir;

if ( user_wind_dir == "NORTH" )
{
        wind_dir = NORTH_WIND;
}
else if ( user_wind_dir == "SOUTH" )
{
        wind_dir = SOUTH_WIND;
}
else if ( user_wind_dir == "EAST" )
{
```

```
        wind_dir = EAST_WIND;
}
else if ( user_wind_dir == "WEST" )
{
        wind_dir = WEST_WIND;
}
else if ( user_wind_dir == "NONE" )
{
        wind_dir = NO_WIND;
}
else
{
        std::cout << "That's not a valid direction!" << std::endl;
}
```

Polymorphic Enums?

In C++, we often use polymorphism to allow old code to handle new code--for instance, as long as we subclass the interface expected by a function, we can pass in the new class and expect it to work correctly with the code that was written before the new class ever existed. Unfortunately, with enums, you can't really do this, even though there are occasional times you'd like to. (For instance, if you were managing the settings for your program and you stored all of them as enum values, then it might be nice to have an enum, settings_t, from which all of your other enums inherited so that you could store every new enum in the settings list. Note that since the list contains values of different types, you can't use templates.)

If you need this kind of behavior, you're forced to store the enums as integers and then retrieve them using typecasts to assign the particular value to the setting of interest. And you won't even get the benefit of dynamic_cast to help you ensure that the cast is safe--you'll have to rely on the fact that incorrect values cannot be stored in the list.

Summary

The Good

- Enums allow you to constrain the values a variable takes on
- Enums can be used to make your program more readable by eliminating magic numbers are specifying exact what range of values a variable expects to take on
- Enums can be used to quickly declare a range of constant values without using #define

The Gotchas

- Enum constants must be carefully named to avoid name collisions
- Enums don't work "polymorphically" except from the int type, which can be inconvenient