Enumerations

The material in this handout is collected from the following references:

- Section 19.3 of the text book [C++ Primer].
- Various sections of [Effective C++].
- Various sections of [More Effective C++].

Additional examples and explanations on C++ enumerations can be found [at this page] from Microsoft.

C++ provides two types of enumerations: *plain* enumerations and *scoped* enumerations.

Plain enumerations

Plain enumerations are similar to the enumerations described in C. Here's a brief review: An *enumeration* is a type that can hold a set of literal integer values specified by the user. An enumeration's possible values are named and are called *enumerators* or *enumeration constants*. For example, the following code defines enumerated type `Fish`:

```
enum Fish { trout, carp, salmon, halibut };
```

`trout`, `carp`, ... are called *enumerators* which are integer constants. Here, `trout` is equivalent to `0`, `carp` is `1`, and so on. If we wish, we can initialize these enumerators with integer constants:

```
enum Fish { trout = 10, carp = 12, salmon = carp+2, halibut = 15 };
```

> ***For plain enumerations, the enumerator names are in the same scope as the `enum` and their values implicitly convert to integers.***

Consider the following code fragment that illustrates how plain enumerator names in `enum Month` leak into the enclosing scope and how the enumerator values implicitly convert to `int`:

```
enum Month {
  jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;          // ok
Month m2 = Month::feb; // ok
int n = m2;             // ok: plain enum implicitly converts to int
Month m7 = Month(7);    // ok: explicitly convert int to Month
```

Name collisions can occur since plain enumerator names leak into the enclosing scope. The definition of enumeration type `TraficLight` is invalid because it tries to re-define names `red` and `green`.

```
enum Color { red, green, blue };
enum TrafficLight { red, yellow, green }; // error
```

Plain enumerations also have some type-safety characteristics. First, the implicit conversion from an integer value to an enumeration is not allowed:

```
enum Fish { trout, carp, salmon, halibut };
Fish f = salmon;
f = 1;     // error: int doesn't implicitly convert to Fish
```

Second, an enumeration value of one type doesn't convert to an enumeration value of a different enumeration type:

```
enum Fish { trout, carp, salmon, halibut, fish_type };
enum Color { red, green, blue };

Fish f = salmon;
Color c = blue;
f = green; // error: cannot assign f a Color enumerator
```

However, having a plain enumeration value convert to `int` can lead to nasty surprises:

```
enum Fish { trout, carp, salmon, halibut, fish_type };
enum Color { red, green, blue };

Fish f = salmon;
if (f == 2) { // oops!!! comparing fish and int
  std::cout << "salmon are blue\n";
} else {
  std::cout << "salmon are not blue\n";
}
```

The size [such as `int` vs. `short`] and signedness [such as `int` vs. `unsigned`] of pre-C++11 enumerations was implementation-dependent. It is now possible to specify the underlying type of an enumeration. The underlying type must be one of the signed or unsigned integer types - the default is `int`. We can be explicit about the underlying type:

```
enum Shape : int {Circle, Rectangle, Square};
```

If it is too wasteful of space, we could instead use a `char`:

```
enum Shape : char {Circle, Rectangle, Square};
```

A plain enumeration can be *unnamed*. For example:

```
enum { fish_size_small = 10, fish_size_large = 100 };
```

We use an unnamed enumeration when all we need is a set of integer constants, rather than a type to use for defining variables. One use in C consists of replacing macros with enumerators to define static arrays. For example, the following preprocessor macro

```
#include ARRAY_SIZE (20)
```

can be replaced with enumeration constant

```
enum { ARRAY_SIZE = 20 };
```

to define static arrays:

```
int array[ARRAY_SIZE];
```

`enum` classes

Scoped enumerations are simple user-defined types that address the type safety and name collision problems associated with plain enumerations. For example:

```
enum class Color { red, green, blue };
```

The "body" of the enumeration is simply the usual list of its enumerators. The `class` in `enum class` means that the enumerators are in the scope of the enumeration. That is, to refer to `red`, we have to say `Color::red`.

Scoped enumerations are also *strongly typed*. For example:

```
enum class TrafficLight { red, yellow, green };
enum class FireAlert { green, yellow, orange, red };

FireAlert w1 = 7;    // ERROR: no int to Warning conversion
int w2 = green;      // ERROR: green not in scope
int w3 = FireAlert::green; // ERROR: no Warning to int conversion
FireAlert w4 = FireAlert::green; // OK

void foo(TrafficLight x) {
  if (x == 9) { /* ... */ }               // ERROR: 9 is not a TrafficLight
  if (x == red) { /* ... */ }             // ERROR: no red in scope
  if (x == FireAlert::red) { /* ... */ }  // ERROR: x is not a FireAlert
  if (x == TrafficLight::red) { /* ... */ } // OK
}
```

Note that enumerators `red`, `yellow`, `green` that are present in both `enum`s don't clash because each is in the scope of its own `enum` class.

Just as with plain enumerations, we can be explicit about the underlying type:

```
enum class Color : int { red, green, blue };
```

If it is too wasteful of space, we could instead use a `char`:

```
enum class Color : char { red, green, blue };
```

By default, enumerator values are assigned values in sequence that increase from `0`. Here, the following expressions evaluate `true`:

```
enum class FireAlert { green, yellow, orange, red };

static_cast<int>(FireAlert::green) == 0
static_cast<int>(FireAlert::yellow) == 1
static_cast<int>(FireAlert::orange) == 2
static_cast<int>(FireAlert::red) == 3
```

Declaring a variable of type `FireAlert` instead of plain `int` can give both the user and compiler a hint as to the intended use. For example:

```cpp
enum class FireAlert { green, yellow, orange, red };

// compiler will generate diagnostic message that only three
// enumerators out of four are being used!!!
int foo(FireAlert key) {
  switch(key) {
    case FireAlert::green:
      return 0;
    case FireAlert::yellow:
      return 1;
    case FireAlert::orange:
      return 2;
  }
}
```

A human might or might not notice that enumerator `red` is missing. Even better, a compiler might issue a warning because only three of the four `FireAlert` enumerators are being used.

By default, an `enum` class has only assignment, initialization and comparisons (e.g., `==` and `<`) defined. However, an enumeration is a user-defined type so we can define operators for it. Here, we overload the prefix and postfix increment operators for an `enum` class:

```cpp
enum class Weekday {
  Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun
};

// prefix increment operator
Weekday& operator++(Weekday& w) {
  w = (w == Weekday::Sun) ? Weekday::Mon : Weekday(static_cast<int>(w)+1);
  return w;
}

// postfix increment operator
Weekday operator++(Weekday& w, int) {
  Weekday old{w};
  ++w;
  return old;
}

// use cases ...
Weekday w = Weekday(4); // Weekday::Thu
Weekday w2 = ++w;       // w2 is Weekday::Fri
Weekday w3 = w2++;      // w3 is Weekday::Fri
```

In general, prefer `enum` classes because they cause fewer surprises such as accidental misuse of constants.