Enumeration (unscoped)

- an enumeration is a distinct type whose value is restricted to a range of values
- an enum can include several explicitly named constants ("enumerators")
- the values of the constants are "integer" numbers
- it is compiler-dependent to determine an "integer" type that can represent all enumerators

```
enum Color { red, green, blue }; // an unscoped enum
Color r = red;
switch(r)
{
    case red   : std::cout << "red\n";   break;
    case green: std::cout << "green\n"; break;
    case blue  : std::cout << "blue\n";  break;
}
```

Enumeration (unscoped)

- each enumerator is associated with a value of the underlying type:

```
enum Foo { a, b, c = 10, d, e = 1, f, g = f + c };
//a = 0, b = 1, c = 10, d = 11, e = 1, f = 2, g = 12
```

- the values can be converted to their underlying type (implicitly):

```
enum color { red, yellow, green = 20, blue };
color col = red;
int n = blue; // n is 21
```

- you can specific the underlying type explicitly:

```
enum color : char { red, yellow, green = 20, blue };
```

Enumeration (scoped)

- declaring a scoped enumeration type whose underlying type is int (the keywords `class` and `struct` are exactly equivalent)

```
enum struct|class name { enumerator = constexpr ,
        enumerator = constexpr , ... }
```

- example:

```
enum class Color { red, green = 20, blue };
Color r = Color::blue;
switch(r)
{
    case Color::red   : std::cout << "red\n";   break;
    case Color::green: std::cout << "green\n"; break;
    case Color::blue  : std::cout << "blue\n";  break;
}
```

Enumeration (scoped)

- the values can also be converted to their underlying type but explicitly:

```
enum class Color { red, yellow, green = 20, blue };
Color col = Color::red;
int n = Color::blue; // NOT OK
int m = (int) Color::blue; // OK
int l = static_cast<int>(Color::blue); // OK
```

- you can also specific the underlying type:

```
enum class Color : char { red, yellow, green = 20, blue };
```

Enumeration - unscoped vs scoped

- Unscoped enum type could be misused

```
enum Color { red, yellow, blue };
enum MyColor { myblue, myyellow, myred };
Color col = red;
if (col == myred) { // Should it be true?
    ...
}
```

- `Color` shouldn't be compared with `MyColor`. You will see a compiler warning, but the expression is allowed (because implicitly converted to the underlying type).
- Use scoped enum to avoid this

```
enum class Color { red, yellow, blue };
enum class MyColor { myblue, myyellow, myred };
Color col = Color::red;
if (col == MyColor::myred) { // Compiler will give you an error here!
        ...
}
```

Enumeration

- Why don't we just use `int`?

```
int return_code = some_processing();
switch (return_code) {
    ...
    case 97: // do something for case 97
    case 98: // do something for case 98
    ...
}
```

comparing to

```
enum class ReturnCode = { ... , RECEIVED_TWICE = 97, NOT_RECEIVED = 98, ... };
ReturnCode return_code = some_processing();
switch (return_code) {
    ...
    case ReturnCode::RECEIVED_TWICE: // do something when received twice
    case ReturnCode::NOT_RECEIVED: // do something when not received
    ...
}
```

Which one do you prefer when debugging/modifying the codes?

It's your choice.