HIGH-LEVEL PROGRAMMING 2

Scoped Enumerations

by Prasanna Ghali

Summary of C Enumerations

- Quite often, you might want to assign integer codes to different items in your program, e.g.,

```
int month; // Jan = 1, Feb = 2,...
month = 5; // May
```

- Someone reading your program that does not know your integer code will be confused, e.g.,

```
int team;   // Ferrari = 1,
            // McLaren = 2,...
team = 6;   // What does this mean?
```

- C *enumeration types* do this in a better way

Declaring C Enumerations

In enumeration declaration, identifiers or *enumerators* given for each possible value that enumeration type can contain

Enumeration specifies set of *integer* values of type `int`
Enumeration declarations have similar syntax to structure, only difference is use of `enum` keyword

```
enum Team {
  FERRARI, MCLAREN,
  BMW, WILLIAMS,
  RENAULT, TOYOTA
};

enum Team my_team = BMW;
```

1

2

3

`my_team` is variable of type `enum Team` and is initialized with value `BMW`

Values of Enumerators (1/3)

- By default, enumerators are assigned values 0, 1, 2, … in order

```
enum Suit {SPADE, HEART, CLUB, DIAMOND};
```

0 1 2 3

Values of Enumerators (2/3)

- Enumerators can be explicitly specified values:

```
enum Suit {
  SPADE = 4, HEART = 3,
  CLUB = 2, DIAMOND = 1
};
```

Values of Enumerators (3/3)

- Unspecified values are assigned value of previous member plus one

If 1st enumerator value is unspecified, by default, it has value of zero

```
enum Suit {
  SPADE, HEART = 8,
  CLUB = 2, DIAMOND
};
```

DIAMOND has value 3 - value of previous member SPADE plus one

Enumerations: Use Cases (1/3)

- Since enumerators are `int`s, you can use `enum` variable anywhere an `int` is legal:

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

enum Fish myfish = bass;
if (myfish == trout)
  grill_fish(myfish);
else
  bake_fish(myfish);
```

Enumerations: Use Cases (2/3)

- Since enumerators are `int`s, you can use `enum` variable anywhere an `int` is legal:

```
enum Suit {SPADE, HEART, CLUB, DIAMOND};

void fs(enum Suit);
void fi(int);

enum Suit s = CLUB;

int i = DIAMOND; // i is 3
s = SPADE;       // s is 0 (SPADE)
s++;             // s is 1 (HEART)
i += s;          // i is 4
fi(s);           // argument is 1
fs(2);           // argument is DIAMOND
```

Enumerations: Use Cases (3/3)

Enumerators are compile-time constants and therefore can be used to define array sizes.
This is preferable to preprocessor macros!!!

```c
// not preferred
#define ARRAYSIZE 10
int arr[ARRAYSIZE];
```

```c
// preferred!!!
enum {ARRAYSIZE = 15};
int arr[ARRAYSIZE];
```

Unnamed Enumerations

Unnamed enum is used when all we need is set of integer constants, rather than a type for defining integer variables

2

enum declaration need not have enumeration tag

1

```
enum {trout = 2,  bass   = 5,
      carp  = 10, salmon = 15};

int myfish = carp;
if (myfish == trout)
  grill_fish(myfish);
else
  bake_fish(myfish);
```

Summary of C Enumerations

- Enumeration can be used to give identifiers to integer codes
- Enumeration type variable is `int` and can be used in similar ways
- Type qualifier `const` and `enum` type can satisfy all symbolic constant operations for which `#define` might be used

C++ Plain Enumerations

- Similar to C enumerations with some exceptions

C++ Plain Enumerations: Differences with C Enumerations (1)

- C++ allows programmers to be explicit about size and signedness of enumerations

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

  - If `int` is too wasteful, we could instead use `char`

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

  - Default type is implementation specified

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

C++ Plain Enumerations: Differences with C Enumerations (2)

- C++ plain enumerations are more type safe
  - Implicit conversion from integer value to enumeration is not allowed
  - However, implicit conversion from enumeration to integer value is still allowed [as in C]

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

void ff(Fish);
void fi(int);

Fish f{salmon};
ff(2);              // error!!!
fi(salmon);         // ok!!!
```

C++ Plain Enumerations: Differences with C Enumerations (3)

- C++ plain enumerations are more type safe
  - Enumeration of one type doesn't convert to enumeration value of different enumeration type

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

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

C++ Plain Enumerations: Disadvantages (1/2)

- Since enumerator names are in same scope as the `enum`, name collisions can occur

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

C++ Plain Enumerations: Disadvantages (2/2)

- Having a plain enumeration value convert to `int` can lead to nasty surprises:

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

C++ Scoped Enumerations

- Considered as simple user-defined types [since C++11] to address type safety and name collision problems associated with plain enumerations:

```
enum class Color { red, green, blue };
enum struct Assets { equity, bond, future };
```

- Keyword `class` or `struct` keyword in definition means enumerators are in scope of enumeration

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

C++ Scoped Enumerations: Strongly Typed

- Scoped enumerations are strongly typed – values of scoped enumerations no longer convert implicitly to integral value!!!

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

void ffa(FireAlert);
void fi(int);

FireAlert w1 = 7; // error: no int to FirAlert conversion
int w2 = green;   // error: green not in scope
ffa(2);           // error: no int to FirAlert conversion
int w3 = FireAlert::green; // error: no FireAlert to int conversion
FireAlert w4{FireAlert::green}; // OK
fi(w4);           // error: no FireAlert to int conversion

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

C++ Enumerations: Overloading Operators (1/2)

- By default, `enum` has only assignment, initialization, and comparisons
- Since `enum` is user-defined type, we can define functions that overload operators on it

C++ Enumerations: Overloading Operators (2/2)


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

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

Weekday operator++(Weekday& w, int) { // postfix increment operator
  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
std::cout << "w3: " << (int) w3 << "\n";
```

Summary

- In general, prefer scoped enumerations because they cause fewer surprises