Compound Types: Enums and Structs

Unscoped enumerations

```
// Define a new unscoped enumeration named Color
enum Color
{
    // Here are the enumerators
    // These symbolic constants define all the possible values this type can hold
    // Each enumerator is separated by a comma, not a semicolon
    red,
    green,
    blue, // trailing comma optional but recommended
}; // the enum definition must end with a semicolon
```

```
int main()
{
    // Define a few variables of enumerated type Color
    Color apple { red };   // my apple is red
    Color shirt { green }; // my shirt is green
    Color cup { blue };    // my cup is blue

    Color socks { white }; // error: white is not an enumerator of Color
    Color hat { 2 };       // error: 2 is not an enumerator of Color

    return 0;
}
```

Best practice: Name your enumerated types starting with a capital letter. Name your enumerators starting with a lower case letter.

Sometimes functions will return a status code to the caller to indicate whether the function executed successfully or encountered an error. Traditionally, small negative numbers were used to represent different possible error codes. However, using magic numbers like this isn't very descriptive. A better method would be to use an enumerated type:

```
enum FileReadResult
{
    readResultSuccess,
```

```
    readResultErrorFileOpen,
    readResultErrorFileRead,
    readResultErrorFileParse,
};

FileReadResult readFileContents()
{
    if (!openFile())
        return readResultErrorFileOpen;
    if (!readFile())
        return readResultErrorFileRead;
    if (!parseFile())
        return readResultErrorFileParse;

    return readResultSuccess;
}
```

enumerations are small and inexpensive to copy, it is fine to pass (and return) them by value.

Avoiding enumerator naming collisions

```
namespace Color
{
    // The names Color, red, blue, and green are defined inside namespace Color
    enum Color
    {
        red,
        green,
        blue,
    };
}

namespace Feeling
{
    enum Feeling
    {
        happy,
        tired,
        blue, // Feeling::blue doesn't collide with Color::blue
    };
}
```

```
int main()
{
    Color::Color paint{ Color::blue };
    Feeling::Feeling me{ Feeling::blue };

    return 0;
}
```

Avoid assigning explicit values to your enumerators unless you have a compelling reason to do so.

```
enum Animal
{
    cat = -3,     // values can be negative
    dog,          // -2
    pig,          // -1
    horse = 5,
    giraffe = 5, // shares same value as horse
    chicken,      // 6
};
```

Best practise: Make the enumerator representing 0 the one that is the best default meaning for your enumeration. If no good default meaning exists, consider adding an “invalid” or “unknown” enumerator that has value 0, so that state is explicitly documented and can be explicitly handled where appropriate.

```
#include <cstdint>  // for std::int8_t
#include <iostream>

// Use an 8-bit integer as the enum underlying type
enum Color : std::int8_t
{
    black,
    red,
    blue,
};

int main()
{
    Color c{ black };
```

```
    std::cout << sizeof(c) << '\n'; // prints 1 (byte)

    return 0;
}
```

Specify the base type of an enumeration only when necessary.

Integer to unscoped enumerator conversion

```
enum Pet // no specified base
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet { 2 }; // compile error: integer value 2 won't implicitly convert to a Pet
    pet = 3;       // compile error: integer value 3 won't implicitly convert to a Pet

    return 0;
}
```

^this wont work

```
enum Pet // no specified base
{
    cat, // assigned 0
    dog, // assigned 1
    pig, // assigned 2
    whale, // assigned 3
};

int main()
{
    Pet pet { static_cast<Pet>(2) }; // convert integer 2 to a Pet
    pet = static_cast<Pet>(3);       // our pig evolved into a whale!

    return 0;
}
```

Getting the name of an enumerator

#include
#include <string_view>

```
enum Color
{
    black,
    red,
    blue,
};

constexpr std::string_view getColorName(Color color)
{
    switch (color)
    {
    case black: return "black";
    case red:   return "red";
    case blue:  return "blue";
    default:    return "???";
    }
}

int main()
{
    constexpr Color shirt{ blue };

    std::cout << "Your shirt is " << getColorName(shirt) << '\n';

    return 0;
}
```

Getting an enumeration from a string

```
#include <iostream>
#include <optional> // for std::optional
#include <string>
#include <string_view>

enum Pet
{
    cat,   // 0
    dog,   // 1
```

```
    pig,   // 2
    whale, // 3
};

constexpr std::string_view getPetName(Pet pet)
{
    switch (pet)
    {
    case cat:   return "cat";
    case dog:   return "dog";
    case pig:   return "pig";
    case whale: return "whale";
    default:    return "???";
    }
}

constexpr std::optional<Pet> getPetFromString(std::string_view sv)
{
    if (sv == "cat")   return cat;
    if (sv == "dog")   return dog;
    if (sv == "pig")   return pig;
    if (sv == "whale") return whale;

    return {};
}

int main()
{
    std::cout << "Enter a pet: cat, dog, pig, or whale: ";
    std::string s{};
    std::cin >> s;

    std::optional<Pet> pet { getPetFromString(s) };

    if (!pet)
        std::cout << "You entered an invalid pet\n";
    else
        std::cout << "You entered: " << getPetName(*pet) << '\n';

    return 0;
}
```

Overloading operator>> to input an enumerator

```
#include <iostream>
#include <limits>
#include <optional>
#include <string>
#include <string_view>

enum Pet
{
    cat,   // 0
    dog,   // 1
    pig,   // 2
    whale, // 3
};

constexpr std::string_view getPetName(Pet pet)
{
    switch (pet)
    {
    case cat:   return "cat";
    case dog:   return "dog";
    case pig:   return "pig";
    case whale: return "whale";
    default:    return "???";
    }
}

constexpr std::optional<Pet> getPetFromString(std::string_view sv)
{
    if (sv == "cat")   return cat;
    if (sv == "dog")   return dog;
    if (sv == "pig")   return pig;
    if (sv == "whale") return whale;

    return {};
}

// pet is an in/out parameter
std::istream& operator>>(std::istream& in, Pet& pet)
{
    std::string s{};
    in >> s; // get input string from user

    std::optional<Pet> match { getPetFromString(s) };
```

```
    if (match) // if we found a match
    {
        pet = *match; // set Pet to the matching enumerator
        return in;
    }

    // We didn't find a match, so input must have been invalid
    // so we will set input stream to fail state
    in.setstate(std::ios_base::failbit);

    // On an extraction failure, operator>> zero-initializes fundamental
types
    // Uncomment the following line to make this operator do the same thing
    // pet = {};

    return in;
}

int main()
{
    std::cout << "Enter a pet: cat, dog, pig, or whale: ";
    Pet pet{};
    std::cin >> pet;

    if (std::cin) // if we found a match
        std::cout << "You chose: " << getPetName(pet) << '\n';
    else
    {
        std::cin.clear(); // reset the input stream to good
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        std::cout << "Your pet was not valid\n";
    }

    return 0;
}
```

Scoped enumerations (enum classes)

That solution is the scoped enumeration (often called an enum class in C++ for reasons that will become obvious shortly).

Scoped enumerations work similarly to unscoped enumerations (13.2 -- Unscoped enumerations), but have two primary differences: They **won't implicitly convert to integers**, and the enumerators are only

placed into the scope region of the enumeration (not into the scope region where the enumeration is defined).

```
#include <iostream>
int main()
{
    enum class Color // "enum class" defines this as a scoped enumeration rather than an unscoped enumeration
    {
        red, // red is considered part of Color's scope region
        blue,
    };

    enum class Fruit
    {
        banana, // banana is considered part of Fruit's scope region
        apple,
    };

    Color color { Color::red }; // note: red is not directly accessible, we have to use Color::red
    Fruit fruit { Fruit::banana }; // note: banana is not directly accessible, we have to use Fruit::banana

    if (color == fruit) // compile error: the compiler doesn't know how to compare different types Color and Fruit
        std::cout << "color and fruit are equal\n";
    else
        std::cout << "color and fruit are not equal\n";

    return 0;
}
```

This program produces a compile error on line 19, since the scoped enumeration won't convert to any type that can be compared with another type.

What is an aggregate?

In general programming, an **aggregate data type** (also called an aggregate) is any type that can contain multiple data members. Some types of aggregates allow members to have different types (e.g. structs), while others require that all members must be of a single type (e.g. arrays).

To simplify a bit, an aggregate in C++ is either a C-style array, or a class type (struct, class, or union) that has:

- No user-declared constructors
- No private or protected non-static data members
- No virtual functions
  The popular type std::array is also an aggregate.