13.5 — Introduction to overloading the I/O operators

In the prior lesson (13.4 -- Converting an enumeration to and from a string), we showed this example, where we used a function to convert an enumeration into an equivalent string:

```
#include <iostream>
#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;
}
```

Although the above example works just fine, there are two downsides:

1. We have to remember the name of the function we created to get the enumerator name.
2. Having to call such a function adds clutter to our output statement.

Ideally, it would be nice if we could somehow teach `operator<<` to output an enumeration, so we could do something like this: `std::cout << shirt` and have it do what we expect.

Introduction to operator overloading

In lesson 11.1 -- Introduction to function overloading, we introduced function overloading, which allows us to create multiple functions with the same name so long as each function has a unique function prototype. Using function overloading, we can create variations of a function that work with different data types, without having to think up a unique name for each variant.

Similarly, C++ also supports **operator overloading**, which lets us define overloads of existing operators, so that we can make those operators work with our program-defined data types.

Basic operator overloading is fairly straightforward:

- Define a function using the name of the operator as the function's name.
- Add a parameter of the appropriate type for each operand (in left-to-right order). One of these parameters must be a user-defined type (a class type or an enumerated type), otherwise the compiler will error.
- Set the return type to whatever type makes sense.
- Use a return statement to return the result of the operation.

When the compiler encounters the use of an operator in an expression and one or more of the operands is a user-defined type, the compiler will check to see if there is an overloaded operator function that it can use to resolve that call. For example, given some expression `x + y`, the compiler will use function overload resolution see if there is an `operator+(x, y)` function call that it can use to evaluate the operation. If a non-ambiguous `operator+` function can be found, it will be called, and the result of the operation returned as the return value.

Related content

We cover operator overloading in much more detail in chapter chapter 21.

For advanced readers

Operators can also be overloaded as member functions of the left-most operand. We discuss this in lesson 21.5 -- Overloading operators using member functions.

Overloading `operator<<` to print an enumerator

Before we proceed, let's quickly recap how `operator<<` works when used for output.

Consider a simple expression like `std::cout << 5`. `std::cout` has type `std::ostream` (which is a user-defined type in the standard library), and `5` is a literal of type `int`.

When this expression is evaluated, the compiler will look for an overloaded `operator<<` function that can handle arguments of type `std::ostream` and `int`. It will find such a function (also defined as part of the standard I/O library) and call it. Inside that function, `std::cout` is

used to output `x` to the console (exactly how is implementation-defined). Finally, the `operator<<` function returns its left-operand (which in this case is `std::cout`), so that subsequent calls to `operator<<` can be chained.

With the above in mind, let's implement an overload of `operator<<` to print a `Color`:

```
#include <iostream>
#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 "???";
    }
}

// Teach operator<< how to print a Color
// std::ostream is the type of std::cout, std::cerr, etc...
// The return type and parameter type are references (to prevent copies from being
made)
std::ostream& operator<<(std::ostream& out, Color color)
{
    out << getColorName(color); // print our color's name to whatever output stream
out
    return out;                 // operator<< conventionally returns its left operand

    // The above can be condensed to the following single line:
    // return out << getColorName(color)
}

int main()
{
        Color shirt{ blue };
        std::cout << "Your shirt is " << shirt << '\n'; // it works!

        return 0;
}
```

This prints:

```
Your shirt is blue
```

Let's unpack our overloaded operator function a bit. First, the name of the function is `operator<<`, since that is the name of the operator we're overloading. `operator<<` has two parameters. The left parameter is our output stream, which has type `std::ostream`. We use pass by non-const reference here because we don't want to make a copy of a `std::ostream` object when the function is called, but the `std::ostream` object needs to be modified in order to do output. The right parameter is our `Color` object. Since `operator<<` conventionally returns its left operand, the return type matches the type of the left-operand, which is `std::ostream&`.

Now let's look at the implementation. A `std::ostream` object already knows how to print a `std::string_view` using `operator<<` (this comes as part of the standard library). So `out << getColorName(color)` simply fetches our color's name as a `std::string_view` and then prints it to the output stream.

Note that our implementation uses parameter `out` instead of `std::cout` because we want to allow the caller to determine which output stream they will output to (e.g. `std::cerr << color` should output to `std::cerr`, not `std::cout`).

Returning the left operand is also easy. The left operand is parameter `out`, so we just return `out`.

Putting it all together: when we call `std::cout << shirt`, the compiler will see that we've overloaded `operator<<` to work with objects of type `Color`. Our overloaded `operator<<` function is then called with `std::cout` as the `out` parameter, and our `shirt` variable (which has value `blue`) as parameter `color`. Since `out` is a reference to `std::cout`, and `color` is a copy of enumerator `blue`, the expression `out << getColorName(color)` prints `"blue"` to the console. Finally `out` is returned back to the caller in case we want to chain additional output.

Overloading `operator>>` to input an enumerator

Similar to how we were able to teach `operator<<` to output an enumeration above, we can also teach `operator>>` how to input an enumeration:

```
#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;
}
```

There are a few differences from the output case worth noting here. First, `std::cin` has type `std::istream`, so we use `std::istream&` as the type of our left parameter and return value instead of `std::ostream&`. Second, the `pet` parameter is a non-const reference. This allows our `operator>>` to modify the value of the right operand that is passed in if our extraction results in a match.

Inside the function, we use `operator>>` to input a `std::string` (something it already knows how to do). If the value the user enters matches one of our pets, then we can assign `pet` the appropriate enumerator and return the left operand (`in`).

If the user did not enter a valid pet, then we handle that case by putting `std::cin` into “failure mode”. This is the state that `std::cin` typically goes into when an extraction fails. The caller can then check `std::cin` to see if the extraction succeeded or failed.

Related content

In lesson 17.6 -- std::array and enumerations, we show how we can use `std::array` to make our input and output operators less redundant, and avoid having to modify then when a new enumerator is added.