21.11 — Overloading typecasts

In lesson 10.6 -- Explicit type conversion (casting) and static_cast, you learned that C++ allows you to convert one data type to another. The following example shows an int being converted into a double:

```
int n{ 5 };
auto d{ static_cast<double>(n) }; // int cast to a double
```

C++ already knows how to convert between the built-in data types. However, it does not know how to convert any of our user-defined classes. That's where overloading the typecast operators comes into play.

User-defined conversions allow us to convert our class into another data type. Take a look at the following class:

```
class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};
```

This class is pretty simple: it holds some number of cents as an integer, and provides access functions to get and set the number of cents. It also provides a constructor for converting an int into a Cents.

If we can convert an int into a Cents (via a constructor), then doesn't it also make sense for us to be able to convert a Cents back into an int? In some cases, this might not be true, but in this case, it does make sense.

In the following example, we have to use getCents() to convert our Cents variable back into an integer so we can print it using printInt():

```
#include <iostream>

void printInt(int value)
{
    std::cout << value;
}

int main()
{
    Cents cents{ 7 };
    printInt(cents.getCents()); // print 7

    std::cout << '\n';

    return 0;
}
```

If we have already written a lot of functions that take integers as parameters, our code will be littered with calls to getCents(), which makes it more messy than it needs to be.

To make things easier, we can provide a user-defined conversion by overloading the int typecast. This will allow us to convert our Cents class directly into an int. The following example shows how this is done:

```
class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    // Overloaded int cast
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};
```

There are three things to note:

1. To overload the function that casts our class to an int, we write a new function in our class called `operator int()`. Note that there is a space between the word operator and the type we are casting to. Such functions must be non-static members.
2. User-defined conversions do not have parameters, as there is no way to pass arguments explicitly to them. They do still have a hidden `*this` parameter, pointing to the implicit object (which is the object to be converted)

3. User-defined conversions do not declare a return type. The name of the conversion (e.g. int) is used as the return type, as it is the only return type allowed. This prevents redundancy.

Now in our example, we can call printInt() like this:

```
#include <iostream>

int main()
{
    Cents cents{ 7 };
    printInt(cents); // print 7

    std::cout << '\n';

    return 0;
}
```

The compiler will first note that function printInt takes an integer parameter. Then it will note that variable cents is not an int. Finally, it will look to see if we've provided a way to convert a Cents into an int. Since we have, it will call our operator int() function, which returns an int, and the returned int will be passed to printInt().

Such typecasts can also be invoked explicitly via `static_cast`:

```
std::cout << static_cast<int>(cents);
```

You can provide user-defined conversions to any data type you wish, including your own program-defined data types!

Here's a new class called Dollars that provides an overloaded Cents conversion:

```
class Dollars
{
private:
    int m_dollars{};
public:
    Dollars(int dollars=0)
        : m_dollars{ dollars }
    {
    }

     // Allow us to convert Dollars into Cents
     operator Cents() const { return Cents{ m_dollars * 100 }; }
};
```

This allows us to convert a Dollars object directly into a Cents object! This allows you to do something like this:

```
#include <iostream>

class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    // Overloaded int cast
    operator int() const { return m_cents; }

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};

class Dollars
{
private:
    int m_dollars{};
public:
    Dollars(int dollars=0)
        : m_dollars{ dollars }
    {
    }

    // Allow us to convert Dollars into Cents
    operator Cents() const { return Cents { m_dollars * 100 }; }
};

void printCents(Cents cents)
{
    std::cout << cents; // cents will be implicitly cast to an int here
}

int main()
{
    Dollars dollars{ 9 };
    printCents(dollars); // dollars will be implicitly cast to a Cents here

    std::cout << '\n';

    return 0;
}
```

Consequently, this program will print the value:

```
900
```

which makes sense, since 9 dollars is 900 cents!

Explicit typecasts

Just like we can make constructors explicit so that they can't be used for implicit conversions, we can also make our overloaded typecasts explicit for the same reason. Explicit typecasts can only be invoked explicitly (e.g. during non-copy initialization or by using an explicit cast like `static_cast`).

```
#include <iostream>

class Cents
{
private:
    int m_cents{};
public:
    Cents(int cents=0)
        : m_cents{ cents }
    {
    }

    explicit operator int() const { return m_cents; } // now marked as explicit

    int getCents() const { return m_cents; }
    void setCents(int cents) { m_cents = cents; }
};

class Dollars
{
private:
    int m_dollars{};
public:
    Dollars(int dollars=0)
        : m_dollars{ dollars }
    {
    }

    operator Cents() const { return Cents { m_dollars * 100 }; }
};

void printCents(Cents cents)
{
    std::cout << static_cast<int>(cents); // must use explicit cast to invoke
explicit typecast
}

int main()
{
    Dollars dollars{ 9 };
    printCents(dollars);

    std::cout << '\n';

    return 0;
}
```

Typecasts should be generally be marked as explicit. Exceptions can be made in cases where the conversion inexpensively converts to a similar user-defined type. Our `Dollars::operator Cents()` typecast was left non-explicit because there is no reason not to let a `Dollars` object be used anywhere a `Cents` is expected.

Best practice

Typecasts should be marked as explicit, except in cases where the class to be converted to is essentially synonymous.

Converting constructors vs overloaded typecasts

Overloaded typecasts and converting constructors perform similar roles: an overloaded typecast allows us to define a function that converts some program-defined type A into some other type B. A converting constructor allows us to define a function that creates some program-defined type A from some other type B. So when should you use each?

In general, a converting constructor should be preferred to an overloaded typecast, as it allows the type being constructed to own the construction.

There are a few cases where an overloaded typecast should be used instead:

- When providing a conversion to a fundamental type (since you can't define constructors for these types). Most idiomatically, these are used to provide a conversion to `bool` for cases where it makes sense to be able to use an object in a conditional statement.
- When providing a conversion to a type you can't add members to (e.g. a conversion to `std::vector`, since you can't define constructors for these types either).
- When you do not want the type being constructed to be aware of the type being converted from. This can be helpful for avoiding circular dependencies.

For an example of that last bullet, `std::string` has a constructor to create a `std::string` from a `std::string_view`. This means `<string>` must include `<string_view>`. If `std::string_view` had a constructor to create a `std::string_view` from a `std::string`, then `<string_view>` would need to include `<string>`, and this would result in a circular dependency between headers.

Instead, `std::string` has an overloaded typecast that handles conversion from `std::string` to `std::string_view` (which is fine, since it's already including `<string_view>`). `std::string_view` does not know about `std::string` at all, and thus does not need to include `<string>`. In this way, the circular dependency is avoided.