13.11 — Overloading typecasts

ALEX JULY 24, 2021

In lesson 8.5 -- 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, 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():

```
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 cast 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.
2. User-defined conversions do not take parameters, as there is no way to pass arguments to them.
3. User-defined conversions do not have a return type. C++ assumes you will be returning the correct type.

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

```
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().

We can now also explicitly cast our Cents variable to an int:

```
Cents cents{ 7 };
int c{ static_cast<int>(cents)
};
```

You can provide user-defined conversions for any data type you wish, including your own user-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:

```cpp
#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!

Next lesson
13.12 The copy constructor

Back to table of contents

Previous lesson
13.10 Overloading the parenthesis operator

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

106 COMMENTS

Newest