14.16 — Converting constructors and the explicit keyword

In lesson 10.1 -- Implicit type conversion, we introduced type conversion and the concept of implicit type conversion, where the compiler will implicitly convert a value of one type to a value of another type as needed if such a conversion exists.

This allows us to do things like this:

```
#include <iostream>

void printDouble(double d) // has a double parameter
{
    std::cout << d;
}

int main()
{
    printDouble(5); // we're supplying an int argument

    return 0;
}
```

In the above example, our `printDouble` function has a `double` parameter, but we're passing in an argument of type `int`. Because the type of the parameter and the type of the argument do not match, the compiler will see if it can implicitly convert the type of the argument to the type of the parameter. In this case, using the numeric conversion rules, int value `5` will be converted to double value `5.0` and because we're passing by value, parameter `d` will be copy initialized with this value.

User-defined conversions

Now consider the following similar example:

```
#include <iostream>

class Foo
{
private:
    int m_x{};
public:
    Foo(int x)
        : m_x{ x }
    {
    }

    int getX() const { return m_x; }
};

void printFoo(Foo f) // has a Foo parameter
{
    std::cout << f.getX();
}

int main()
{
    printFoo(5); // we're supplying an int argument

    return 0;
}
```

In this version, `printFoo` has a `Foo` parameter but we're passing in an argument of type `int`. Because these types do not match, the compiler will try to implicitly convert int value `5` into a `Foo` object so the function can be called.

Unlike the first example, where our parameter and argument types were both fundamental types (and thus can be converted using the built-in numeric promotion/conversion rules), in this case, one of our types is a program-defined type. The C++ standard doesn't have specific rules that tell the compiler how to convert values to (or from) a program-defined type.

Instead, the compiler will look to see if we have defined some function that it can use to perform such a conversion. Such a function is called a **user-defined conversion**.

Converting constructors

In the above example, the compiler will find a function that lets it convert int value `5` into a `Foo` object. That function is the `Foo(int)` constructor.

Up to this point, we've typically used constructors to explicitly construct objects:

```
Foo x { 5 }; // Explicitly convert int value 5 to a Foo
```

Think about what this does: we're providing an `int` value (`5`) and getting a `Foo` object in return.

In the context of a function call, we're trying to solve the same problem:

```
printFoo(5); // Implicitly convert int value 5 into a Foo
```

We're providing an `int` value (`5`), and we want a `Foo` object in return. The `Foo(int)` constructor was designed for exactly that!

So in this case, when `printFoo(5)` is called, parameter `f` is copy initialized using the `Foo(int)` constructor with `5` as an argument!

A constructor that can be used to perform an implicit conversion is called a **converting constructor**. By default, all constructors are converting constructors.

Only one user-defined conversion may be applied

Now consider the following example:

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

class Employee
{
private:
    std::string m_name{};

public:
    Employee(std::string_view name)
        : m_name{ name }
    {
    }

    const std::string& getName() const { return m_name; }
};

void printEmployee(Employee e) // has an Employee parameter
{
    std::cout << e.getName();
}

int main()
{
    printEmployee("Joe"); // we're supplying an string literal argument

    return 0;
}
```

In this version, we've swapped out our `Foo` class for an `Employee` class. `printEmployee` has an `Employee` parameter, and we're passing in a C-style string literal. And we have a converting constructor: `Employee(std::string_view)`.

You might be surprised to find that this version doesn't compile. The reason is simple: only one user-defined conversion may be applied to perform an implicit conversion, and this example requires two. First, our C-style string literal has to be converted to a `std::string_view` (using a `std::string_view` converting constructor), and then our `std::string_view` has to be converted into an `Employee` (using the `Employee(std::string_view)` converting constructor).

There are two ways to make this example work:

1. Use a `std::string_view` literal:

```
int main()
{
    using namespace std::literals;
    printEmployee( "Joe"sv); // now a std::string_view literal

    return 0;
}
```

This works because only one user-defined conversion is now required (from `std::string_view` to `Employee`).

2. Explicitly construct an `Employee` rather than implicitly create one:

```
int main()
{
    printEmployee(Employee{ "Joe" });

    return 0;
}
```

This also works because only one user-defined conversion is now required (from the string literal to the `std::string_view` used to initialize the `Employee` object). Passing our explicitly constructed `Employee` object to the function does not require a second conversion to take place.

This latter example brings up a useful technique: it is trivial to convert an implicit conversion into an explicit definition. We'll see more examples of this later in this lesson.

Key insight

An implicit conversion can be trivially converted into an explicit definition by using direct list initialization (or direct initialization).

When converting constructors go wrong

Consider the following program:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars{};

public:
    Dollars(int d)
        : m_dollars{ d }
    {
    }

    int getDollars() const { return m_dollars; }
};

void print(Dollars d)
{
    std::cout << "$" << d.getDollars();
}

int main()
{
    print(5);

    return 0;
}
```

When we call `print(5)`, the `Dollars(int)` converting constructor will be used to convert `5` into a `Dollars` object. Thus, this program prints:

```
$5
```

Although this may have been the caller's intent, it's hard to tell because the caller did not provide any indication that this is what they actually wanted. It's entirely possible that the caller assumed this would print `5`, and did not expect the compiler to silently and implicitly convert our `int` value to a `Dollars` object so that it could satisfy this function call.

While this example is trivial, in a larger and more complex program, it's fairly easy to be surprised by the compiler performing some implicit conversion that you did not expect, resulting in unexpected behavior at runtime.

It would be better if our `print(Dollars)` function could only be called with a `Dollars` object, not any value that can be implicitly converted to a `Dollars` (especially a fundamental type like `int`). This would reduce the possibility of inadvertent errors.

The explicit keyword

To address such issues, we can use the **explicit** keyword to tell the compiler that a constructor should not be used as a converting constructor.

Making a constructor `explicit` has two notable consequences:

- An explicit constructor cannot be used to do copy initialization or copy list initialization.
- An explicit constructor cannot be used to do implicit conversions (since this uses copy initialization or copy list initialization).

Let's update the `Dollars(int)` constructor from the prior example to be an explicit constructor:

```
#include <iostream>

class Dollars
{
private:
    int m_dollars{};

public:
    explicit Dollars(int d) // now explicit
        : m_dollars{ d }
    {
    }

    int getDollars() const { return m_dollars; }
};

void print(Dollars d)
{
    std::cout << "$" << d.getDollars();
}

int main()
{
    print(5); // compilation error because Dollars(int) is explicit

    return 0;
}
```

Because the compiler can no longer use `Dollars(int)` as a converting constructor, it can not find a way to convert `5` to a `Dollars`. Consequently, it will generate a compilation error.

Explicit constructors can be used for direct and list initialization

An explicit constructor can still be used for direct and direct list initialization:

```
// Assume Dollars(int) is explicit
int main()
{
    Dollars d1(5); // ok
    Dollars d2{5}; // ok
}
```

Now, let's go back to our prior example, where we made our `Dollars(int)` constructor explicit, and therefore the following generated a compilation error:

```
print(5); // compilation error because Dollars(int) is explicit
```

What if we actually want to call `print()` with `int` value `5` but the constructor is explicit? The workaround is simple: instead of having the compiler implicitly convert `5` into a `Dollars` that can be passed to `print()`, we can explicitly define the `Dollars` object ourselves:

```
print(Dollars{5}); // ok: create Dollars and pass to print() (no conversion required)
```

This is allowed because we can still use explicit constructors to list initialize objects. And since we've now explicitly constructed a `Dollars`, the argument type matches the parameter type, so no conversion is required!

This not only compiles and runs, it also better documents our intent, as it is explicit about the fact that we meant to call this function with a `Dollars` object.

Return by value and explicit constructors

When we return a value from a function, if that value does not match the return type of the function, an implicit conversion will occur. Just like with pass by value, such conversions cannot use explicit constructors.

The following programs shows a few variations in return values, and their results:

```
#include <iostream>

class Foo
{
public:
    explicit Foo() // note: explicit (just for sake of example)
    {
    }

    explicit Foo(int x) // note: explicit
    {
    }
};

Foo getFoo()
{
    // explicit Foo() cases
    return Foo{ };   // ok
    return { };      // error: can't implicitly convert initializer list to Foo

    // explicit Foo(int) cases
    return 5;        // error: can't implicitly convert int to Foo
    return Foo{ 5 }; // ok
    return { 5 };    // error: can't implicitly convert initializer list to Foo
}

int main()
{
    return 0;
}
```

Perhaps surprisingly, `return { 5 }` is considered a conversion.

Best practices for use of `explicit`

The modern best practice is to make any constructor that will accept a single argument `explicit` by default. This includes constructors with multiple parameters where most or all of them have default values.

This will disallow the compiler from using that constructor for implicit conversions. If an implicit conversion is required, only non-explicit constructors will be considered. If no non-explicit constructor can be found to perform the conversion, the compiler will error.

If such a conversion is actually desired in a particular case, it is trivial to convert the implicit conversion into an explicit definition using list initialization.

The following should not be made explicit:

- Copy (and move) constructors (as these do not perform conversions).

- Default constructors with no parameters (as these are only used to convert `{}` to a default object, not something we need to restrict).
- Constructors that only accept multiple arguments (as these are typically not a candidate for conversions anyway).

There are some occasions when it does make sense to make a single-argument constructor non-explicit. This can be useful when all of the following are true:

- The converted value is semantically equivalent to the argument value.
- The conversion is performant.

For example, the `std::string_view` constructor that accepts a C-style string argument is not explicit, because there is unlikely to be a case when we wouldn't be okay with a C-style string being treated as a `std::string_view` instead.

Conversely, the `std::string` constructor that takes a `std::string_view` is marked as explicit, because while a `std::string` value is semantically equivalent to a `std::string_view` value, constructing a `std::string` is not performant.

Best practice

Make any constructor that accepts a single argument `explicit` by default. If an implicit conversion between types is both semantically equivalent and performant, you can consider making the constructor non-explicit.

Do not make copy or move constructors explicit, as these do not perform conversions.