26.4 — Class template specialization

In the previous lesson 26.3 -- Function template specialization, we saw how it was possible to specialize functions in order to provide different functionality for specific data types. As it turns out, it is not only possible to specialize functions, it is also possible to specialize classes!

Consider the case where you want a class that stores 8 objects. Here's a simplified class template to do so:

```
#include <iostream>

template <typename T>
class Storage8
{
private:
    T m_array[8];

public:
    void set(int index, const T& value)
    {
        m_array[index] = value;
    }

    const T& get(int index) const
    {
        return m_array[index];
    }
};

int main()
{
    // Define a Storage8 for integers
    Storage8<int> intStorage;

    for (int count{ 0 }; count < 8; ++count)
        intStorage.set(count, count);

    for (int count{ 0 }; count < 8; ++count)
        std::cout << intStorage.get(count) << '\n';

    // Define a Storage8 for bool
    Storage8<bool> boolStorage;
    for (int count{ 0 }; count < 8; ++count)
        boolStorage.set(count, count & 3);

        std::cout << std::boolalpha;

    for (int count{ 0 }; count < 8; ++count)
    {
        std::cout << boolStorage.get(count) << '\n';
    }

    return 0;
}
```

This example prints:

```
0
1
2
3
4
5
6
7
false
true
true
true
false
true
true
true
```

While this class is completely functional, it turns out that the implementation of `Storage8<bool>` is more inefficient than it needs to be. Because all variables must have an address, and the CPU can't address anything smaller than a byte, all variables must be at least a byte in size. Consequently, a variable of type `bool` ends up using an entire byte even though technically it only needs a single bit to store its true or false value! Thus, a `bool` is 1 bit of useful information and 7 bits of wasted space. Our `Storage8<bool>` class, which contains 8 `bool`, is 1 byte worth of useful information and 7 bytes of wasted space.

As it turns out, using some basic bit logic, it's possible to compress all 8 bools into a single byte, eliminating the wasted space altogether. However, in order to do this, we'll need to revamp the class when used with type `bool`, replacing the array of 8 `bool` with a variable that is a single byte in size. While we could create an entirely new class to do so, this has one major downside: we have to give it a different name. Then the programmer has to remember that `Storage8<T>` is meant for non-bool types, whereas `Storage8Bool` (or whatever we name the new class) is meant for `bool`. That's needless complexity we'd rather avoid. Fortunately, C++ provides us a better method: class template specialization.

Class template specialization

Class template specialization allows us to specialize a template class for a particular data type (or data types, if there are multiple template parameters). In this case, we're going to use class template specialization to write a customized version of `Storage8<bool>` that will take precedence over the generic `Storage8<T>` class.

Class template specializations are treated as completely independent classes, even though they are instantiated in the same way as the templated class. This means that we can change anything and everything about our specialization class, including the way it's implemented and even the functions it makes public, just as if it were an independent class.

Just like all templates, the compiler must be able to see the full definition of a specialization to use it. Also, defining a class template specialization requires the non-specialized class to be defined first.

Here's an example of a specialized `Storage8<bool>` class:

```
#include <cstdint>

// First define our non-specialized class template
template <typename T>
class Storage8
{
private:
    T m_array[8];

public:
    void set(int index, const T& value)
    {
        m_array[index] = value;
    }

    const T& get(int index) const
    {
        return m_array[index];
    }
};

// Now define our specialized class template
template <> // the following is a template class with no templated parameters
class Storage8<bool> // we're specializing Storage8 for bool
{
// What follows is just standard class implementation details

private:
    std::uint8_t m_data{};

public:
    // Don't worry about the details of the implementation of these functions
    void set(int index, bool value)
    {
        // Figure out which bit we're setting/unsetting
        // This will put a 1 in the bit we're interested in turning on/off
        auto mask{ 1 << index };

        if (value)  // If we're setting a bit
            m_data |= mask;   // use bitwise-or to turn that bit on
        else  // if we're turning a bit off
            m_data &= ~mask;  // bitwise-and the inverse mask to turn that bit off
        }

    bool get(int index)
    {
        // Figure out which bit we're getting
        auto mask{ 1 << index };
        // bitwise-and to get the value of the bit we're interested in
        // Then implicit cast to boolean
        return (m_data & mask);
    }
```

```
};

// Same example as before
int main()
{
    // Define a Storage8 for integers (instantiates Storage8<T>, where T = int)
    Storage8<int> intStorage;

    for (int count{ 0 }; count < 8; ++count)
    {
        intStorage.set(count, count);
        }

    for (int count{ 0 }; count < 8; ++count)
    {
        std::cout << intStorage.get(count) << '\n';
    }

    // Define a Storage8 for bool  (instantiates Storage8<bool> specialization)
    Storage8<bool> boolStorage;

    for (int count{ 0 }; count < 8; ++count)
    {
        boolStorage.set(count, count & 3);
    }

        std::cout << std::boolalpha;

    for (int count{ 0 }; count < 8; ++count)
    {
        std::cout << boolStorage.get(count) << '\n';
    }

    return 0;
}
```

First, note that our specialized class template starts off with `template<>`. The template keyword tells the compiler that what follows is a template, and the empty angle braces means that there aren't any template parameters. In this case, there aren't any template parameters because we're replacing the only template parameter (`T`) with a specific type (`bool`).

Next, we add `<bool>` to the class name to denote that we're specializing a `bool` version of `class Storage8`.

All of the other changes are just class implementation details. You do not need to understand how the bit-logic works in order to use the class (though you can review O.2 -- Bitwise operators if you want to figure it out, but need a refresher on how bitwise operators work).

Note that this specialization class utilizes a `std::uint8_t` (1 byte unsigned int) instead of an array of 8 `bool` (8 bytes).

Now, when we instantiate an object type `Storage<T>`, where `T` is not a `bool`, we'll get a version stenciled from the generic templated `Storage8<T>` class. When we instantiate an object of type `Storage8<bool>`, we'll get the specialized version we just created. Note that we have kept the publicly exposed interface of both classes the same -- while C++ gives us free reign to add, remove, or change functions of `Storage8<bool>` as we see fit, keeping a consistent interface means the programmer can use either class in exactly the same manner.

As you might expect, this prints the same result as the previous example that used the non-specialized version of `Storage8<bool>`:

```
0
1
2
3
4
5
6
7
false
true
true
true
false
true
true
true
```

Specializing member functions

In the previous lesson, we introduced this example:

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

int main()
{
    // Define some storage units
    Storage i { 5 };
    Storage d { 6.7 };

    // Print out some values
    i.print();
    d.print();
}
```

Our desire is to specialize the `print()` function so that it prints doubles in scientific notation. Using class template specialization, we could define a specialized class for `Storage<double>`:

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

// Explicit class template specialization for Storage<double>
// Note how redundant this is
template <>
class Storage<double>
{
private:
    double m_value {};
public:
    Storage(double value)
      : m_value { value }
    {
    }

    void print();
};

// We're going to define this outside the class for reasons that will become obvious
shortly
void Storage<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}

int main()
{
    // Define some storage units
    Storage i { 5 };
    Storage d { 6.7 }; // uses explicit specialization Storage<double>

    // Print out some values
    i.print(); // calls Storage<int>::print (instantiated from Storage<T>)
    d.print(); // calls Storage<double>::print (called from explicit specialization
```

```
of Storage<double>)
}
```

However, note how much redundancy there is here. We've duplicated an entire class definition just so that we can change one member function!

Fortunately, we can do better. C++ does not require us to explicitly specialize `Storage<double>` to explicitly specialize `Storage<double>::print()`. Instead, we can let the compiler implicitly specialize `Storage<double>` from `Storage<T>`, and provide an explicit specialization of just `Storage<double>::print()`! Here's what that looks like:

```
#include <iostream>

template <typename T>
class Storage
{
private:
    T m_value {};
public:
    Storage(T value)
      : m_value { value }
    {
    }

    void print()
    {
        std::cout << m_value << '\n';
    }
};

template<>
void Storage<double>::print()
{
    std::cout << std::scientific << m_value << '\n';
}

int main()
{
    // Define some storage units
    Storage i { 5 };
    Storage d { 6.7 }; // will cause Storage<double> to be implicitly instantiated

    // Print out some values
    i.print(); // calls Storage<int>::print (instantiated from Storage<T>)
    d.print(); // calls Storage<double>::print (called from explicit specialization
of Storage<double>::print())
}
```

That's it!

Where to define class template specializations

In order to use a specialization, the compiler must be able to see the full definition of both the non-specialized class and the specialized class. If the compiler can only see the definition of the non-specialized class, it will use that instead of the specialization.

For this reason, specialized classes are often defined in header files, below the definition of the non-specialized class, so that including a single header includes both the non-specialized class and any specializations. This ensures the specialization can always be seen whenever the non-specialized class can also be seen.

If a specialization is only required in a single translation unit, it can be defined in the source file for that translation unit. Because other translation units will not be able to see the definition of the specialization, they will continue to use the non-specialized version.

Be wary of putting a specialization in its own separate header file, with the intent of including the specialization's header in any translation unit where the specialization is desired. It's a bad idea to design code that transparently changes behavior based on the presence or absence of a header file. For example, if you intend to use the specialization but forget to include the header of the specialization, you may end up using the non-specialized version instead. If you intend to use the non-specialization, you may end up using the specialization anyway if some other header includes the specialization as a transitive include.