19.4 — Class template specialization

ALEX SEPTEMBER 28, 2021

In the previous lesson 19.3 -- 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 an entire class!

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

```
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];
    }
};

```

Because this class is templated, it will work fine for any given type:

```cpp
#include <iostream>

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 much 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 bools, 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 bools 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 bools. 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. This works analogously to how a specialized function takes precedence over a generic template function.

Class template specialization are treated as completely independent classes, even though they are allocated 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. Here's our specialized class:

```
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:
    unsigned char m_data{};

public:
    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);
    }
};
```

First, note that we start off with `template<>`. The template keyword tells the compiler that what follows is templated, 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 single unsigned char (1 byte) instead of an array of 8 bools (8 bytes).

Now, when we declare a class of type Storage8<T>, where T is not a bool, we'll get a version stenciled from the generic templated Storage8<T> class. When we declare a class 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.

We can use the exact same example as before to show both Storage8<T> and Storage8<bool> being instantiated:

```
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;
}
```

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
```

It's worth noting that keeping the public interface between your template class and all of the specializations similar is generally a good idea, as it makes them easier to use -- however, it's not strictly necessary.

Next lesson
19.5 Partial template specialization

Back to table of contents

Previous lesson
19.3 Function template specialization

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

DP N N FOUT

Newest