

13.6 — Class template specialization

BY ALEX ON AUGUST 16TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In the previous lesson **13.5 -- 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:

```

1  template <class T>
2  class Storage8
3  {
4  private:
5      T m_array[8];
6
7  public:
8      void set(int index, const T &value)
9      {
10         m_array[index] = value;
11     }
12
13     const T& get(int index)
14     {
15         return m_array[index];
16     }
17 };

```

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

```

1  #include <iostream>
2
3  int main()
4  {
5      // Define a Storage8 for integers
6      Storage8<int> intStorage;
7
8      for (int count=0; count<8; ++count)
9          intStorage.set(count, count);
10
11     for (int count=0; count<8; ++count)
12         std::cout << intStorage.get(count) << '\n';
13
14     // Define a Storage8 for bool
15     Storage8<bool> boolStorage;
16     for (int count=0; count<8; ++count)
17         boolStorage.set(count, count & 3);
18
19     for (int count=0; count<8; ++count)
20         std::cout << (boolStorage.get(count) ? "true" : "false") << '\n';
21
22     return 0;
23 }

```

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 specializations 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:

```

1  template <> // the following is a template class with no templated parameters
2  class Storage8<bool> // we're specializing Storage8 for bool
3  {
4  // What follows is just standard class implementation details
5  private:
6      unsigned char m_data;
7
8  public:
9      Storage8() : m_data(0)
10     {
11     }
12
13     void set(int index, bool value)
14     {
15         // Figure out which bit we're setting/unsetting
16         // This will put a 1 in the bit we're interested in turning on/off
17         unsigned char mask = 1 << index;

```

```

18
19     if (value) // If we're setting a bit
20         m_data |= mask; // Use bitwise-or to turn that bit on
21     else // if we're turning a bit off
22         m_data &= ~mask; // bitwise-and the inverse mask to turn that bit off
23 }
24
25 bool get(int index)
26 {
27     // Figure out which bit we're getting
28     unsigned char mask = 1 << index;
29     // bitwise-and to get the value of the bit we're interested in
30     // Then implicit cast to boolean
31     return (m_data & mask);
32 }
33 };

```

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 (typename `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 [0.2 -- 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:

```

1  int main()
2  {
3      // Define a Storage8 for integers (instantiates Storage8<T>, where T = int)
4      Storage8<int> intStorage;
5
6      for (int count=0; count<8; ++count)
7          intStorage.set(count, count);
8
9      for (int count=0; count<8; ++count)
10         std::cout << intStorage.get(count) << '\n';
11
12     // Define a Storage8 for bool (instantiates Storage8<bool> specialization)
13     Storage8<bool> boolStorage;
14     for (int count=0; count<8; ++count)
15         boolStorage.set(count, count & 3);
16
17     for (int count=0; count<8; ++count)
18         std::cout << (boolStorage.get(count) ? "true" : "false") << '\n';
19
20     return 0;
21 }

```

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



[13.7 -- Partial template specialization](#)



[Index](#)



[13.5 -- Function template specialization](#)

 [C++ TUTORIAL](#)  [C++](#), [CLASSES](#), [PROGRAMMING](#), [SPECIALIZATION](#), [TEMPLATES](#), [TUTORIAL](#) |  [PRINT THIS POST](#)

48 comments to 13.6 — Class template specialization



Jon
[February 9, 2019 at 11:09 pm · Reply](#)

Hello, on line 31 of the bool class specialization, can it simply be: "return (m_data & mask);"?

Do we need the "!= 0" part on there? I'm a bit confused because I thought that anything that's not 0000 0000 would be implicitly converted to "true" anyway...

Thanks, as always!

[nascardriver](#)