26.5 — Partial template specialization

This lesson and the next are optional reading for those desiring a deeper knowledge of C++ templates. Partial template specialization is not used all that often (but can be useful in specific cases).

In lesson 26.2 -- Template non-type parameters, you learned how expression parameters could be used to parameterize template classes.

Let's take another look at the Static Array class we used in one of our previous examples:

```
template <typename T, int size> // size is the expression parameter
class StaticArray
{
private:
    // The expression parameter controls the size of the array
    T m_array[size]{};

public:
    T* getArray() { return m_array; }

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

This class takes two template parameters: one type parameter and one expression parameter.

Now, let's say we wanted to write a function to print out the whole array. Although we could implement this as a member function, we're going to do it as a non-member function instead because it will make the successive examples easier to follow.

Using templates, we might write something like this:

```
template <typename T, int size>
void print(const StaticArray<T, size>& array)
{
    for (int count{ 0 }; count < size; ++count)
        std::cout << array[count] << ' ';
}
```

This would allow us to do the following:

```
#include <iostream>

template <typename T, int size> // size is a template non-type parameter
class StaticArray
{
private:
	T m_array[size]{};

public:
	T* getArray() { return m_array; }

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

template <typename T, int size>
void print(const StaticArray<T, size>& array)
{
	for (int count{ 0 }; count < size; ++count)
		std::cout << array[count] << ' ';
}

int main()
{
	// declare an int array
	StaticArray<int, 4> int4{};
	int4[0] = 0;
	int4[1] = 1;
	int4[2] = 2;
	int4[3] = 3;

	// Print the array
	print(int4);

	return 0;
}
```

and get the following result:

```
0 1 2 3
```

Although this works, it has a design flaw. Consider the following:

```
#include <algorithm>
#include <iostream>
#include <string_view>

int main()
{
    // Declare a char array
    StaticArray<char, 14> char14{};

    // Copy some data into it
    constexpr std::string_view hello{ "Hello, world!" };
    std::copy_n(hello.begin(), hello.size(), char14.getArray());

    // Print the array
    print(char14);

    return 0;
}
```

(We covered std::strcpy in lesson 17.10 -- C-style strings if you need a refresher)

This program will compile, execute, and produce the following value (or one similar):

```
H e l l o ,   w o r l d !
```

For non-char types, it makes sense to put a space between each array element, so they don’t run together. However, with a char type, it makes more sense to print everything run together as a C-style string, which our print() function doesn’t do.

So how can we fix this?

Template specialization to the rescue?

One might first think of using template specialization. The problem with full template specialization is that all template parameters must be explicitly defined.

Consider:

```
#include <algorithm>
#include <iostream>
#include <string_view>

template <typename T, int size> // size is the expression parameter
class StaticArray
{
private:
	// The expression parameter controls the size of the array
	T m_array[size]{};

public:
	T* getArray() { return m_array; }

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

template <typename T, int size>
void print(const StaticArray<T, size>& array)
{
	for (int count{ 0 }; count < size; ++count)
		std::cout << array[count] << ' ';
}

// Override print() for fully specialized StaticArray<char, 14>
template <>
void print(const StaticArray<char, 14>& array)
{
	for (int count{ 0 }; count < 14; ++count)
		std::cout << array[count];
}

int main()
{
    // Declare a char array
    StaticArray<char, 14> char14{};

    // Copy some data into it
    constexpr std::string_view hello{ "Hello, world!" };
    std::copy_n(hello.begin(), hello.size(), char14.getArray());

    // Print the array
    print(char14);

    return 0;
}
```

As you can see, we've now provided an overloaded print function for fully specialized `StaticArray<char, 14>`. Indeed, this prints:

```
Hello, world!
```

Although this solves the issue of making sure print() can be called with a `StaticArray<char, 14>`, it brings up another problem: using full template specialization means we have to explicitly define the length of the array this function will accept! Consider the following example:

```
int main()
{
    // Declare a char array
    StaticArray<char, 12> char12{};

    // Copy some data into it
    constexpr std::string_view hello{ "Hello, mom!" };
    std::copy_n(hello.begin(), hello.size(), char12.getArray());

    // Print the array
    print(char12);

    return 0;
}
```

Calling `print()` with `char12` will call the version of `print()` that takes a `StaticArray<T, size>`, because `char12` is of type `StaticArray<char, 12>`, and our overloaded print() will only be called when passed a `StaticArray<char, 14>`.

Although we could make a copy of print() that handles `StaticArray<char, 12>`, what happens when we want to call print() with an array size of 5, or 22? We'd have to copy the function for each different array size. That's redundant.

Obviously full template specialization is too restrictive a solution here. The solution we are looking for is partial template specialization.

Partial template specialization

Partial template specialization allows us to specialize classes (but not individual functions!) where some, but not all, of the template parameters have been explicitly defined. For our challenge above, the ideal solution would be to have our overloaded print function work with StaticArray of type char, but leave the length expression parameter templated so it can vary as needed. Partial template specialization allows us to do just that!

Here's our example with an overloaded print function that takes a partially specialized StaticArray:

```
// overload of print() function for partially specialized StaticArray<char, size>
template <int size> // size is still a template non-type parameter
void print(const StaticArray<char, size>& array) // we're explicitly defining type
char here
{
	for (int count{ 0 }; count < size; ++count)
		std::cout << array[count];
}
```

As you can see here, we’ve explicitly declared that this function will only work for StaticArray of type char, but size is still a templated expression parameter, so it will work for char arrays of any size. That’s all there is to it!

Here’s a full program using this:

```
#include <algorithm>
#include <iostream>
#include <string_view>

template <typename T, int size> // size is the expression parameter
class StaticArray
{
private:
	// The expression parameter controls the size of the array
	T m_array[size]{};

public:
	T* getArray() { return m_array; }

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

template <typename T, int size>
void print(const StaticArray<T, size>& array)
{
	for (int count{ 0 }; count < size; ++count)
		std::cout << array[count] << ' ';
}

// overload of print() function for partially specialized StaticArray<char, size>
template <int size>
void print(const StaticArray<char, size>& array)
{
	for (int count{ 0 }; count < size; ++count)
		std::cout << array[count];
}

int main()
{
	// Declare an char array of size 14
	StaticArray<char, 14> char14{};

	// Copy some data into it
	constexpr std::string_view hello14{ "Hello, world!" };
	std::copy_n(hello14.begin(), hello14.size(), char14.getArray());

	// Print the array
	print(char14);

	std::cout << ' ';

	// Now declare an char array of size 12
	StaticArray<char, 12> char12{};

	// Copy some data into it
	constexpr std::string_view hello12{ "Hello, mom!" };
```

```
        std::copy_n(hello12.begin(), hello12.size(), char12.getArray());

        // Print the array
        print(char12);

        return 0;
}
```

This prints:

```
Hello, world! Hello, mom!
```

Just as we expect.

Partial template specialization can only be used with classes, not template functions (functions must be fully specialized). Our `void print(StaticArray<char, size> &array)` example works because the print function is not partially specialized (it's just an overloaded template function that happens to have a partially-specialized class parameter).

Partial template specialization for member functions

The limitation on the partial specialization of functions can lead to some challenges when dealing with member functions. For example, what if we had defined StaticArray like this?

```
template <typename T, int size>
class StaticArray
{
private:
    T m_array[size]{};

public:
    T* getArray() { return m_array; }

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

    void print() const;
};

template <typename T, int size>
void StaticArray<T, size>::print() const
{
    for (int i{ 0 }; i < size; ++i)
        std::cout << m_array[i] << ' ';
    std::cout << '\n';
}
```

print() is now a member function of class `StaticArray<T, int>`. So what happens when we want to partially specialize print(), so that it works differently? You might try this:

```
// Doesn't work, can't partially specialize functions
template <int size>
void StaticArray<double, size>::print() const
{
        for (int i{ 0 }; i < size; ++i)
                std::cout << std::scientific << m_array[i] << ' ';
        std::cout << '\n';
}
```

Unfortunately, this doesn't work, because we're trying to partially specialize a function, which is disallowed.

So how do we get around this? One obvious way is to partially specialize the entire class:

```cpp
#include <iostream>

template <typename T, int size>
class StaticArray
{
private:
	T m_array[size]{};

public:
	T* getArray() { return m_array; }

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

	void print() const;
};

template <typename T, int size>
void StaticArray<T, size>::print() const
{
	for (int i{ 0 }; i < size; ++i)
		std::cout << m_array[i] << ' ';
	std::cout << '\n';
}

// Partially specialized class
template <int size>
class StaticArray<double, size>
{
private:
	double m_array[size]{};

public:
	double* getArray() { return m_array; }

	const double& operator[](int index) const { return m_array[index]; }
	double& operator[](int index) { return m_array[index]; }

	void print() const;
};

// Member function of partially specialized class
template <int size>
void StaticArray<double, size>::print() const
{
	for (int i{ 0 }; i < size; ++i)
		std::cout << std::scientific << m_array[i] << ' ';
	std::cout << '\n';
}

int main()
{
        // declare an integer array with room for 6 integers
        StaticArray<int, 6> intArray{};

        // Fill it up in order, then print it
        for (int count{ 0 }; count < 6; ++count)
                intArray[count] = count;

        intArray.print();

        // declare a double buffer with room for 4 doubles
        StaticArray<double, 4> doubleArray{};

        for (int count{ 0 }; count < 4; ++count)
                doubleArray[count] = (4.0 + 0.1 * count);

        doubleArray.print();

        return 0;
}
```

This prints:

```
0 1 2 3 4 5
4.000000e+00 4.100000e+00 4.200000e+00 4.300000e+00
```

This works because `StaticArray<double, size>::print()` is no longer a partially specialized function -- it is a non-specialized member of partially specialized class `StaticArray<double, size>`.

However, this isn't a great solution, because we have to duplicate a lot of code from `StaticArray<T, size>` to `StaticArray<double, size>`.

If only there were some way to reuse the code in `StaticArray<T, size>` in `StaticArray<double, size>`. Sounds like a job for inheritance!

You might start off trying to write that code like this:

```
template <int size> // size is the expression parameter
class StaticArray<double, size>: public StaticArray<T, size>
```

But this doesn't work, because we've used `T` without defining it. There is no syntax that allows us to inherit in such a manner.

As an aside…

Even if we were able to define `T` as a type template parameter, when `StaticArray<double, size>` was instantiated, the compiler would need to replace the `T` in `StaticArray<T, size>` with an actual type. What actual type would it use? The only type that makes sense is `T=double`, but that would leave `StaticArray<double, size>` inheriting from itself!

Fortunately, there’s a workaround, by using a common base class:

```
#include <iostream>

template <typename T, int size>
class StaticArray_Base
{
protected:
	T m_array[size]{};

public:
	T* getArray() { return m_array; }

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

	void print() const
	{
		for (int i{ 0 }; i < size; ++i)
			std::cout << m_array[i] << ' ';
		std::cout << '\n';
	}

	// Don't forget a virtual destructor if you're going to use virtual function
resolution
};

template <typename T, int size>
class StaticArray: public StaticArray_Base<T, size>
{
};

template <int size>
class StaticArray<double, size>: public StaticArray_Base<double, size>
{
public:

	void print() const
	{
		for (int i{ 0 }; i < size; ++i)
			std::cout << std::scientific << this->m_array[i] << ' ';
// note: The this-> prefix in the above line is needed.
// See https://stackoverflow.com/a/6592617 or
https://isocpp.org/wiki/faq/templates#nondependent-name-lookup-members for more info
on why.
		std::cout << '\n';
	}
};

int main()
{
	// declare an integer array with room for 6 integers
	StaticArray<int, 6> intArray{};
```

```
        // Fill it up in order, then print it
        for (int count{ 0 }; count < 6; ++count)
                intArray[count] = count;

        intArray.print();

        // declare a double buffer with room for 4 doubles
        StaticArray<double, 4> doubleArray{};

        for (int count{ 0 }; count < 4; ++count)
                doubleArray[count] = (4.0 + 0.1 * count);

        doubleArray.print();

        return 0;
}
```

This prints the same as above, but has significantly less duplicated code.