

19.5 — Partial template specialization

ALEX AUGUST 22, 2021

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 [19.2 -- 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:

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

   public:
3       T* getArray() { return m_array; }
4
5       T& operator[](int index)
       {
           return m_array[index];
       }
   };
```

This class takes two template parameters, a type parameter, and an 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:

```
1 template <typename T, int size>
   void print(StaticArray<T, size>& array)
2 {
   for (int count{ 0 }; count < size;
   ++count)
3     std::cout << array[count] << ' ';
4 }
```

This would allow us to do the following:

```
1 #include <iostream>
2 #include <cstring>
3
4 template <typename T, int size> // size is the expression
   parameter
   class StaticArray
   {
5 private:
6     // The expression parameter controls the size of the array
7     T m_array[size]{};
8
9 public:
10    T* getArray() { return m_array; }
11
12    T& operator[](int index)
13    {
14        return m_array[index];
15    }
16 };
17
18 template <typename T, int size>
19 void print(StaticArray<T, size>& array)
20 {
21     for (int count{ 0 }; count < size; ++count)
22         std::cout << array[count] << ' ';
23 }
24
25 int main()
26 {
27     // declare an int array
28     StaticArray<int, 4> int4{};
29     int4[0] = 0;
30     int4[1] = 1;
31     int4[2] = 2;
32     int4[3] = 3;
33
34     // Print the array
35     print(int4);
36
37     return 0;
38 }
```

and get the following result:

```
0 1 2 3
```

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

```
1 int main()
2 {
3     // declare a char array
4     StaticArray<char, 14> char14{};
5
6     std::strcpy(char14.getArray(), "Hello,
7 world!");
8
9     // Print the array
10    print(char14);
11
12    return 0;
13 }
```

(We covered `std::strcpy` in [lesson 10.6 -- 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:

```
1  #include <iostream>
2  #include <cstring>
3
4  template <typename T, int size> // size is the expression
   parameter
   class StaticArray
5  {
6  private:
7      // The expression parameter controls the size of the array
8      T m_array[size]{};
9
10 public:
11     T* getArray() { return m_array; }
12
13     T& operator[](int index)
14     {
15         return m_array[index];
16     }
17 };
18
19 template <typename T, int size>
20 void print(StaticArray<T, size>& array)
21 {
22     for (int count{ 0 }; count < size; ++count)
23         std::cout << array[count] << ' ';
24 }
25
26 // Override print() for fully specialized StaticArray<char, 14>
27 template <>
28 void print(StaticArray<char, 14>& array)
29 {
30     for (int count{ 0 }; count < 14; ++count)
31         std::cout << array[count];
32 }
33
34 int main()
35 {
36     // declare a char array
37     StaticArray<char, 14> char14{};
38
39     std::strcpy(char14.getArray(), "Hello, world!");
40
41     // Print the array
42     print(char14);
43
44     return 0;
45 }
```

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:

```
1 int main()
2 {
3     // declare a char array
4     StaticArray<char, 12> char12{};
5
6     std::strcpy(char12.getArray(), "Hello,
7 mom!");
8
9     // Print the array
10    print(char12);
11
12    return 0;
13 }
```

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`:

```
1 // overload of print() function for partially specialized StaticArray<char, size>
2 template <int size> // size is still a templated expression parameter
3 void print(StaticArray<char, size>& array) // we're explicitly defining type char
4 here
5 {
6     for (int count{ 0 }; count < size; ++count)
7         std::cout << array[count];
8 }
```

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:

```

1  #include <iostream>
2  #include <cstring>
3
4  template <typename T, int size> // size is the expression parameter
class StaticArray
{
5  private:
6      // The expression parameter controls the size of the array
7      T m_array[size]{};
8
9  public:
10     T* getArray() { return m_array; }
11
12     T& operator[](int index)
13     {
14         return m_array[index];
15     };
16
17     template <typename T, int size>
18     void print(StaticArray<T, size>& array)
19     {
20         for (int count{ 0 }; count < size; ++count)
21             std::cout << array[count] << ' ';
22     }
23
24     // overload of print() function for partially specialized StaticArray<char,
25     // size>
26     template <int size>
27     void print(StaticArray<char, size>& array)
28     {
29         for (int count{ 0 }; count < size; ++count)
30             std::cout << array[count];
31     }
32
33     int main()
34     {
35         // Declare an char array of size 14
36         StaticArray<char, 14> char14{};
37
38         std::strcpy(char14.getArray(), "Hello, world!");
39
40         // Print the array
41         print(char14);
42
43         std::cout << ' ';
44
45         // Now declare an char array of size 12
46         StaticArray<char, 12> char12{};
47
48         std::strcpy(char12.getArray(), "Hello, mom!");
49
50         // Print the array
51         print(char12);
52
53         return 0;
54     }

```

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 function using a class parameter that's partially specialized).

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?

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

   public:
3     T* getArray() { return m_array; }
4
5     T& operator[](int index)
     {
         return m_array[index];
     }

6     void print()
     {
7         for (int i{ 0 }; i < size; ++i)
8             std::cout << m_array[i] << ' ';
9             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:

```
1  // Doesn't work
2  template <int size>
3  void StaticArray<double, size>::print()
   {
4     for (int i{ 0 }; i < size; ++i)
5         std::cout << std::scientific << m_array[i] << ' '
6         ;
       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:


```

1  #include<iostream>
2
3  template <typename T, int size> // size is the expression
  parameter
  class StaticArray
  {
4  private:
5      // The expression parameter controls the size of the array
6      T m_array[size]{};
7
8  public:
9      T* getArray() { return m_array; }
10
11     T& operator[](int index)
12     {
13         return m_array[index];
14     }
15     void print()
16     {
17         for (int i{ 0 }; i < size; ++i)
18             std::cout << m_array[i] << ' ';
19         std::cout << "\n";
20     }
21 };
22
23 template <int size> // size is the expression parameter
24 class StaticArray<double, size>
25 {
26 private:
27     // The expression parameter controls the size of the array
28     double m_array[size]{};
29
30 public:
31     double* getArray() { return m_array; }
32
33     double& operator[](int index)
34     {
35         return m_array[index];
36     }
37     void print()
38     {
39         for (int i{ 0 }; i < size; ++i)
40             std::cout << std::scientific << m_array[i] << ' ';
41         std::cout << "\n";
42     }
43 };
44
45 int main()
46 {
47     // declare an integer array with room for 6 integers
48     StaticArray<int, 6> intArray{};
49
50     // Fill it up in order, then print it
51     for (int count{ 0 }; count < 6; ++count)
52         intArray[count] = count;
53
54     intArray.print();
55
56     // declare a double buffer with room for 4 doubles
57     StaticArray<double, 4> doubleArray{};
58
59     for (int count{ 0 }; count < 4; ++count)
60         doubleArray[count] = (4.0 + 0.1 * count);
61
62     doubleArray.print();
63
64     return 0;
65 }

```

This prints:

```

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

```

While it works, this isn't a great solution, because we had 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:

```
1 | template <int size> // size is the expression parameter  
   | class StaticArray<double, size>: public StaticArray< // Then  
   | what?
```

How do we reference `StaticArray`? We can't.

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

```

1  #include<iostream>
2
3  template <typename T, int size> // size is the expression parameter
4  class StaticArray_Base
5  {
6  protected:
7      // The expression parameter controls the size of the array
8      T m_array[size]{};
9
10 public:
11     T* getArray() { return m_array; }
12
13     T& operator[](int index)
14     {
15         return m_array[index];
16     }
17
18     void print()
19     {
20         for (int i{ 0 }; i < size; ++i)
21             std::cout << m_array[i];
22         std::cout << '\n';
23     }
24
25     virtual ~StaticArray_Base() = default;
26 };
27
28 template <typename T, int size> // size is the expression parameter
29 class StaticArray: public StaticArray_Base<T, size>
30 {
31 public:
32 };
33
34 template <int size> // size is the expression parameter
35 class StaticArray<double, size>: public StaticArray_Base<double, size>
36 {
37 public:
38
39     void print()
40     {
41         for (int i{ 0 }; i < size; ++i)
42             std::cout << std::scientific << this->m_array[i] << ' ';
43         // note: The this-> prefix in the above line is needed.
44         // See https://stackoverflow.com/a/6592617 or https://isocpp.org/wiki/faq/templates#nondependent-name-lookup-members for more info on why.
45         std::cout << '\n';
46     }
47 };
48
49 int main()
50 {
51     // declare an integer array with room for 6 integers
52     StaticArray<int, 6> intArray{};
53
54     // Fill it up in order, then print it
55     for (int count{ 0 }; count < 6; ++count)
56         intArray[count] = count;
57
58     intArray.print();
59
60     // declare a double buffer with room for 4 doubles
61     StaticArray<double, 4> doubleArray{};
62
63     for (int count{ 0 }; count < 4; ++count)
64         doubleArray[count] = (4.0 + 0.1 * count);
65
66     doubleArray.print();
67
68     return 0;
69 }

```

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



Next lesson

19.6 Partial template specialization for pointers



Back to table of contents



Previous lesson

19.4 Class template specialization

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code````



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 ▼

