

13.4 — Template non-type parameters

BY ALEX ON JUNE 19TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 18TH, 2018

In previous lessons, you've learned how to use template type parameters to create functions and classes that are type independent. However, template type parameters are not the only type of template parameters available. Template classes and functions can make use of another kind of template parameter known as a non-type parameter.

Non-type parameters

A template non-type parameter is a special type of parameter that does not substitute for a type, but is instead replaced by a value. A non-type parameter can be any of the following:

- A value that has an integral type or enumeration
- A pointer or reference to a class object
- A pointer or reference to a function
- A pointer or reference to a class member function
- `std::nullptr_t`

In the following example, we create a non-dynamic (static) array class that uses both a type parameter and an non-type parameter. The type parameter controls the data type of the static array, and the non-type parameter controls how large the static array is.

```
1  #include <iostream>
2
3  template <class T, int size> // size is the non-type parameter
4  class StaticArray
5  {
6  private:
7      // The non-type parameter controls the size of the array
8      T m_array[size];
9
10 public:
11     T* getArray();
12
13     T& operator[](int index)
14     {
15         return m_array[index];
16     }
17 };
18
19 // Showing how a function for a class with an non-type parameter is defined outside of the class
20 template <class T, int size>
21 T* StaticArray<T, size>::getArray()
22 {
23     return m_array;
24 }
25
26 int main()
27 {
28     // declare an integer array with room for 12 integers
29     StaticArray<int, 12> intArray;
30
31     // Fill it up in order, then print it backwards
32     for (int count=0; count < 12; ++count)
33         intArray[count] = count;
34
35     for (int count=11; count >= 0; --count)
36         std::cout << intArray[count] << " ";
37     std::cout << '\n';
38
39     // declare a double buffer with room for 4 doubles
40     StaticArray<double, 4> doubleArray;
41
```

```

42     for (int count=0; count < 4; ++count)
43         doubleArray[count] = 4.4 + 0.1*count;
44
45     for (int count=0; count < 4; ++count)
46         std::cout << doubleArray[count] << ' ';
47
48     return 0;
49 }

```

This code produces the following:

```

11 10 9 8 7 6 5 4 3 2 1 0
4.4 4.5 4.6 4.7

```

One noteworthy thing about the above example is that we do not have to dynamically allocate the `m_array` member variable! This is because for any given instance of the `StaticArray` class, `size` is actually constant. For example, if you instantiate a `StaticArray<int, 12>`, the compiler replaces `size` with `12`. Thus `m_array` is of type `int[12]`, which can be allocated statically.

This functionality is used by the standard library class `std::array`. When you allocate a `std::array<int, 5>`, the `int` is a type parameter, and the `5` is a non-type parameter!



[13.5 -- Function template specialization](#)



[Index](#)



[13.3 -- Template classes](#)

Share this:

