26.2 — Template non-type parameters

In previous lessons, you've learned how to use template type parameters to create functions and classes that are type independent. A template type parameter is a placeholder type that is substituted for a type passed in as an argument.

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 template parameter where the type of the parameter is predefined and is substituted for a constexpr value passed in as an argument.

A non-type parameter can be any of the following types:

- An integral type
- An enumeration type
- 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
- A floating point type (since C++20)

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

```
#include <iostream>

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

public:
    T* getArray();

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

// Showing how a function for a class with a non-type parameter is defined outside of
the class
template <typename T, int size>
T* StaticArray<T, size>::getArray()
{
    return m_array;
}

int main()
{
    // declare an integer array with room for 12 integers
    StaticArray<int, 12> intArray;

    // Fill it up in order, then print it backwards
    for (int count { 0 }; count < 12; ++count)
        intArray[count] = count;

    for (int count { 11 }; count >= 0; --count)
        std::cout << intArray[count] << ' ';
    std::cout << '\n';

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

    for (int count { 0 }; count < 4; ++count)
        doubleArray[count] = 4.4 + 0.1 * count;

    for (int count { 0 }; count < 4; ++count)
        std::cout << doubleArray[count] << ' ';

    return 0;
}
```

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 must be constexpr. 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!

Note that if you try to instantiate a template non-type parameter with a non-constexpr value, it will not work:

```
template <int size>
class Foo
{
};

int main()
{
    int x{ 4 }; // x is non-constexpr
    Foo<x> f; // error: the template non-type argument must be constexpr

    return 0;
}
```

In such a case, your compiler will issue an error.

Next lesson
26.3Function template specialization
Back to table of contents
Previous lesson
26.1Template classes