17.13 — Multidimensional std::array

In the prior lesson (17.12 -- Multidimensional C-style Arrays), we discussed C-style multidimensional arrays.

```
// C-style 2d array
int arr[3][4] {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 9, 10, 11, 12 }};
```

But as you're aware, we generally want to avoid C-style arrays (unless they are being used to store global data).

In this lesson, we'll take a look at how multidimensional arrays work with `std::array`.

There is no standard library multidimensional array class

Note that `std::array` is implemented as a single-dimensional array. So the first question you should ask is, "is there a standard library class for multidimensional arrays?" And the answer is… no. Too bad. Womp womp.

A two-dimensional `std::array`

The canonical way to create a two-dimensional array of `std::array` is to create a `std::array` where the template type argument is another `std::array`. That leads to something like this:

```
std::array<std::array<int, 4>, 3> arr {{  // note double braces
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 9, 10, 11, 12 }}};
```

There are a number of "interesting" things to note about this:

- When initializing a multidimensional `std::array`, we need to use double-braces (we discuss why in lesson 17.4 -- std::array of class types, and brace elision).
- The syntax is verbose and hard to read.
- Because of the way templates get nested, the array dimensions are switched. We want an array with 3 rows of 4 elements, so `arr[3][4]` is natural. `std::array<std::array<int, 4>, 3>` is backwards.

Indexing a two-dimensional `std::array` element works just like indexing a two-dimensional C-style array:

```
std::cout << arr[1][2]; // print the element in row 1, column 2
```

We can also pass a two-dimensional `std::array` to a function just like a one-dimensional `std::array`:

```
#include <array>
#include <iostream>

template <typename T, std::size_t Row, std::size_t Col>
void printArray(std::array<std::array<T, Col>, Row> &arr)
{
    for (const auto& arow: arr)   // get each array row
    {
        for (const auto& e: arow) // get each element of the row
            std::cout << e << ' ';

        std::cout << '\n';
    }
}

int main()
{
    std::array<std::array<int, 4>, 3>  arr {{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }}};

    printArray(arr);

    return 0;
}
```

Yuck. And this is for a two-dimensional `std::array`. A three-dimensional or higher `std::array` is even more verbose!

Making two-dimensional `std::array` easier using an alias templates

In lesson 10.7 -- Typedefs and type aliases, we introduced type aliases, and noted that one of the uses of type aliases is to make complex types simpler to use. However, with a normal type alias, we must explicitly specify all template arguments. e.g.

```
using Array2dint34 = std::array<std::array<int, 4>, 3>;
```

This allows us to use `Array2dint34` wherever we want a 3×4 two-dimensional std::array of `int`. But note we'd need one such alias for every combination of element type and dimensions we want to use!

This is a perfect place to use an alias template, which will lets us specify the element type, row length, and column length for a type alias as template arguments!

```
// An alias template for a two-dimensional std::array
template <typename T, std::size_t Row, std::size_t Col>
using Array2d = std::array<std::array<T, Col>, Row>;
```

We can then use `Array2d<int, 3, 4>` anywhere we want a 3×4 two-dimensional std::array of `int`. That's much better!

Here's a full example:

```
#include <array>
#include <iostream>

// An alias template for a two-dimensional std::array
template <typename T, std::size_t Row, std::size_t Col>
using Array2d = std::array<std::array<T, Col>, Row>;

// When using Array2d as a function parameter, we need to respecify the template
parameters
template <typename T, std::size_t Row, std::size_t Col>
void printArray(Array2d<T, Row, Col> &arr)
{
    for (const auto& arow: arr)   // get each array row
    {
        for (const auto& e: arow) // get each element of the row
            std::cout << e << ' ';

        std::cout << '\n';
    }
}

int main()
{
    // Define a two-dimensional array of int with 3 rows and 4 columns
    Array2d<int, 3, 4> arr {{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }}};

    printArray(arr);

    return 0;
}
```

Note how much more concise and easy to use this is!

One neat thing about our alias template is that we can define our template parameters in whatever order we like. Since a `std::array` specifies element type first and then dimension, we stick with that convention. But we have the flexibility to define either `Row` or `Col` first. Since C-style array definitions are defined row-first, we define our alias template with `Row` before `Col`.

This method also scales up nicely to higher-dimensional `std::array`:

```
// An alias template for a three-dimensional std::array
template <typename T, std::size_t Row, std::size_t Col, std::size_t Depth>
using Array3d = std::array<std::array<std::array<T, Depth>, Col>, Row>;
```

Getting the dimensional lengths of a two-dimensional array

Getting the length of a two-dimensional array is slightly non-intuitive.

```
#include <array>
#include <iostream>

// An alias template for a two-dimensional std::array
template <typename T, std::size_t Row, std::size_t Col>
using Array2d = std::array<std::array<T, Col>, Row>;

int main()
{
    // Define a two-dimensional array of int with 3 rows and 4 columns
    Array2d<int, 3, 4> arr {{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }}};

    std::cout << "Rows: " << arr.size() << '\n';    // get length of first dimension
(rows)
    std::cout << "Cols: " << arr[0].size() << '\n'; // get length of second dimension
(cols)

    return 0;
}
```

In order to get the length of the first dimension, we can just call `size()` on our array. But what about the second dimension? In order to get the length of the second dimension, we need an element from that dimension. Thus, we first call `arr[0]` to get the subarray (as element 0 is guaranteed to exist), and then call `size()` on that.

To get the length of the third dimension of a 3-dimensional array, we could call `arr[0][0].size()`.

Flatting a two-dimensional array

Arrays with two or more dimensions have some challenges:

- They are more verbose to define and work with.
- It is awkward to get the length of dimensions greater than the first.
- They are increasingly hard to iterate over (requiring one more loop for each dimension).

One approach to make multidimensional arrays easier to work with is to flatten them. **Flattening** an array is a process of reducing the dimensionality of an array (often down to a single dimension).

For example, instead of creating a two-dimensional array with `Row` rows and `Col` columns, we can create a one-dimensional array with `Row * Col` elements. This gives us the same amount of storage using a single dimension.

However, because our one-dimensional array only has a single dimension, we cannot work with it as a multidimensional array. To address this, we can provide an interface that mimics a multidimensional array. This interface will accept two-dimensional coordinates, and then map them to a unique position in the one-dimensional array.

Here’s an example of that approach that works in C++11 or newer:

```
#include <array>
#include <iostream>
#include <functional>

// An alias template to allow us to define a one-dimensional std::array using two
dimensions
template <typename T, std::size_t Row, std::size_t Col>
using ArrayFlat2d = std::array<T, Row * Col>;

// A modifiable view that allows us to work with an ArrayFlat2d using two dimensions
// This is a view, so the ArrayFlat2d being viewed must stay in scope
template <typename T, std::size_t Row, std::size_t Col>
class ArrayView2d
{
private:
    // You might be tempted to make m_arr a reference to an ArrayFlat2d,
    // but this makes the view non-copy-assignable since references can't be
reseated.
    // Using std::reference_wrapper gives us reference semantics and copy
assignability.
    std::reference_wrapper<ArrayFlat2d<T, Row, Col>> m_arr {};

public:
    ArrayView2d(ArrayFlat2d<T, Row, Col> &arr)
        : m_arr { arr }
    {}

    // Get element via single subscript (using operator[])
    T& operator[](int i) { return m_arr.get()[static_cast<std::size_t>(i)]; }
    const T& operator[](int i) const { return m_arr.get()[static_cast<std::size_t>
(i)]; }

    // Get element via 2d subscript (using operator(), since operator[] doesn't
support multiple dimensions prior to C++23)
    T& operator()(int row, int col) { return m_arr.get()[static_cast<std::size_t>(row
* cols() + col)]; }
    const T& operator()(int row, int col) const { return m_arr.get()
[static_cast<std::size_t>(row * cols() + col)]; }

    // in C++23, you can uncomment these since multidimensional operator[] is
supported
//    T& operator[](int row, int col) { return m_arr.get()[static_cast<std::size_t>
(row * cols() + col)]; }
//    const T& operator[](int row, int col) const { return m_arr.get()
[static_cast<std::size_t>(row * cols() + col)]; }

    int rows() const { return static_cast<int>(Row); }
    int cols() const { return static_cast<int>(Col); }
    int length() const { return static_cast<int>(Row * Col); }
};

int main()
```

```
{
    // Define a one-dimensional std::array of int (with 3 rows and 4 columns)
    ArrayFlat2d<int, 3, 4> arr {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 10, 11, 12 };

    // Define a two-dimensional view into our one-dimensional array
    ArrayView2d<int, 3, 4> arrView { arr };

    // print array dimensions
    std::cout << "Rows: " << arrView.rows() << '\n';
    std::cout << "Cols: " << arrView.cols() << '\n';

    // print array using a single dimension
    for (int i=0; i < arrView.length(); ++i)
        std::cout << arrView[i] << ' ';

    std::cout << '\n';

    // print array using two dimensions
    for (int row=0; row < arrView.rows(); ++row)
    {
        for (int col=0; col < arrView.cols(); ++col)
            std::cout << arrView(row, col) << ' ';
        std::cout << '\n';
    }

    std::cout << '\n';

    return 0;
}
```

This prints:

```
Rows: 3
Cols: 4
1 2 3 4 5 6 7 8 9 10 11 12
1 2 3 4
5 6 7 8
9 10 11 12
```

Because `operator[]` can only accept a single subscript prior to C++23, there are two alternate approaches:

- Use `operator()` instead, which can accept multiple subscripts. This lets you use `[]` for single index indexing and `()` for multiple-dimension indexing. We've opted for this approach above.
- Have `operator[]` return a subview that also overloads `operator[]` so that you can chain `operator[]`. This is more complex and doesn't scale well to higher dimensions.

In C++23, `operator[]` was extended to accept multiple subscripts, so you can overload it to handle both single and multiple subscripts (instead of using `operator()` for multiple subscripts).

Related content

We cover `std::reference_wrapper` in lesson 17.5 -- Arrays of references via std::reference_wrapper.

std::mdspan C++23

Introduced in C++23, `std::mdspan` is a modifiable view that provides a multidimensional array interface for a contiguous sequence of elements. By modifiable view, we mean that a `std::mdspan` is not just a read-only view (like `std::string_view`) -- if the underlying sequence of elements is non-const, those elements can be modified.

The following example prints the same output as the prior example, but uses `std::mdspan` instead of our own custom view:

```
#include <array>
#include <iostream>
#include <mdspan>

// An alias template to allow us to define a one-dimensional std::array using two
dimensions
template <typename T, std::size_t Row, std::size_t Col>
using ArrayFlat2d = std::array<T, Row * Col>;

int main()
{
    // Define a one-dimensional std::array of int (with 3 rows and 4 columns)
    ArrayFlat2d<int, 3, 4> arr {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 10, 11, 12 };

    // Define a two-dimensional span into our one-dimensional array
    // We must pass std::mdspan a pointer to the sequence of elements
    // which we can do via the data() member function of std::array or std::vector
    std::mdspan mdView { arr.data(), 3, 4 };

    // print array dimensions
    // std::mdspan calls these extents
    std::size_t rows { mdView.extents().extent(0) };
    std::size_t cols { mdView.extents().extent(1) };
    std::cout << "Rows: " << rows << '\n';
    std::cout << "Cols: " << cols << '\n';

    // print array in 1d
    // The data_handle() member gives us a pointer to the sequence of elements
    // which we can then index
    for (std::size_t i=0; i < mdView.size(); ++i)
        std::cout << mdView.data_handle()[i] << ' ';
    std::cout << '\n';

    // print array in 2d
    // We use multidimensional [] to access elements
    for (std::size_t row=0; row < rows; ++row)
    {
        for (std::size_t col=0; col < cols; ++col)
            std::cout << mdView[row, col] << ' ';
        std::cout << '\n';
    }
    std::cout << '\n';

    return 0;
}
```

This should be fairly straightforward, but there are a few things worth noting:

- `std::mdspan` let us define a view with as many dimensions as we want.

- The first parameter to the constructor of `std::mdspan` should be a pointer to the array data. This can be a decayed C-style array, or we can use the `data()` member function of `std::array` or `std::vector` to get this data.
- To index a `std::mdspan` in one-dimension, we must fetch the pointer to the array data, which we can do using the `data_handle()` member function. We can then subscript that.
- In C++23, `operator[]` accepts multiple indices, so we use `[row, col]` as our index instead of `[row][col]`.

C++26 will include `std::mdarray`, which essentially combines `std::array` and `std::mdspan` into an owning multidimensional array!