Chapter 19. Boost.MultiArray

Boost.MultiArray is a library that simplifies using arrays with multiple dimensions. The most important advantage is that multidimensional arrays can be used like containers from the standard library. For example, there are member functions, such as `begin()` and `end()`, that let you access elements in multidimensional arrays through iterators. Iterators are easier to use than the pointers normally used with C arrays, especially with arrays that have many dimensions.

Example 19.1. One-dimensional array with `boost::multi_array`

```
#include <boost/multi_array.hpp>
#include <iostream>

int main()
{
  boost::multi_array<char, 1> a{boost::extents[6]};

  a[0] = 'B';
  a[1] = 'o';
  a[2] = 'o';
  a[3] = 's';
  a[4] = 't';
  a[5] = '\0';

  std::cout << a.origin() << '\n';
}
```

Boost.MultiArray provides the class `boost::multi_array` to create arrays. This is the most important class provided. It is defined in `boost/multi_array.hpp`.

`boost::multi_array` is a template expecting two parameters: The first parameter is the type of the elements to store in the array. The second parameter determines how many dimensions the array should have.

The second parameter only sets the number of dimensions, not the number of elements in each dimension. Thus, in Example 19.1, **a** is a one-dimensional array.

The number of elements in a dimension is set at runtime. Example 19.1 uses the global object **`boost::extents`** to set dimension sizes. This object is passed to the constructor of **a**.

An object of type `boost::multi_array` can be used like a normal C array. Elements are accessed by passing an index to `operator[]`. Example 19.1 stores five letters and a null character in **a** – a one-dimensional array with six elements. `origin()` returns a pointer to the first element. The example uses this pointer to write the word stored in the array – `Boost` – to standard output.

Unlike containers from the standard library, `operator[]` checks whether an index is valid. If an index is not valid, the program exits with `std::abort()`. If you don’t want the validity of indexes to be checked, define the macro `BOOST_DISABLE_ASSERTS` before you include `boost/multi_array.hpp`.

Example 19.2. Views and subarrays of a two-dimensional array

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>

int main()
{
  boost::multi_array<char, 2> a{boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::memcpy(view.origin(), "tsooB", 6);
  std::reverse(view.begin(), view.end());

  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::memcpy(subarray.origin(), "C++", 4);

  std::cout << subarray.origin() << '\n';
}
```

Example 19.2 creates a two-dimensional array. The number of elements in the first dimension is set to 2 and for the second dimension set to 6. Think of the array as a table with two rows and six columns.

The first row of the table will contain the word Boost. Since only five letters need to be stored for this word, a *view* is created which spans exactly five elements of the array.

A view, which is based on the class `boost::multi_array::array_view`, lets you access a part of an array and treat that part as though it were a separate array.

`boost::multi_array::array_view` is a template that expects the number of dimensions in the view as a template parameter. In Example 19.2 the number of dimensions for the view is 1. Because the array **a** has two dimensions, one dimension is ignored. To save the word Boost, a one-dimensional array is sufficient; more dimensions would be confusing.

As with `boost::multi_array`, the number of dimensions is passed in as a template parameter, and the size of each dimension is set at runtime. However, with `boost::multi_array::array_view` this isn't done with **`boost::extents`**. Instead it's done with **`boost::indices`**, which is another global object provided by Boost.MultiArray.

As with **`boost::extents`**, indexes must be passed to **`boost::indices`**. While only numbers may be passed to **`boost::extents`**, **`boost::indices`** accepts also ranges. These are defined using `boost::multi_array_types::index_range`.

In Example 19.2, the first parameter passed to **`boost::indices`** isn't a range, it's the number 0. When a number is passed, you cannot use `boost::multi_array_types::index_range`. In the example, the view will take the first dimension of **a** – the one with index 0.

For the second parameter, `boost::multi_array_types::index_range` is used to define a range. By passing 0 and 5 to the constructor, the first five elements of the first dimension of **a** are made available. The range starts at index 0 and ends at index 5 – excluding the element at index 5. The sixth element in the first dimension is ignored.

Thus, **`view`** is a one-dimensional array consisting of five elements – the first five elements in the first row of **a**. When **`view`** is accessed to copy a string with `std::memcpy()` and reverse the elements with `std::reverse()`, this relation doesn't matter. Once the view is created, it acts like an independent array with five elements.

When `operator[]` is called on an array of type `boost::multi_array`, the return value depends on the number of dimensions. In Example 19.1, the operator returns `char` elements because the array accessed is one dimensional.

In Example 19.2, **a** is a two-dimensional array. Thus, `operator[]` returns a subarray rather than a `char` element. Because the type of the subarray isn't public, `boost::multi_array::reference` must be used. This type isn't identical to `boost::multi_array::array_view`, even if the subarray behaves like a view. A view must be defined explicitly and can span arbitrary parts of an array, whereas a subarray is automatically returned by `operator[]` and spans all elements in every dimension.

Example 19.3. Wrapping a C array with `boost::multi_array_ref`

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>

int main()
{
  char c[12] =
  {
    't', 's', 'o', 'o', 'B', '\0',
    'C', '+', '+', '\0', '\0', '\0'
  };

  boost::multi_array_ref<char, 2> a{c, boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::reverse(view.begin(), view.end());
  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::cout << subarray.origin() << '\n';
}
```

The class `boost::multi_array_ref` wraps an existing C array. In Example 19.3, **a** provides the same interface as `boost::multi_array`, but without allocating memory. With `boost::multi_array_ref`, a C array – no matter how many dimensions it has – can be treated like a multidimensional array of type `boost::multi_array`. The C array just needs to be added as an additional parameter to the constructor.

Boost.MultiArray also provides the class `boost::const_multi_array_ref`, which treats a C array as a constant multidimensional array.