Chapter 16. Boost.CircularBuffer

The library Boost.CircularBuffer provides a *circular buffer*, which is a container with the following two fundamental properties:

- The capacity of the circular buffer is constant and set by you. The capacity doesn't change automatically when you call a member function such as `push_back()`. Only you can change the capacity of the circular buffer. The size of the circular buffer can not exceed the capacity you set.

- Despite constant capacity, you can call `push_back()` as often as you like to insert elements into the circular buffer. If the maximum size has been reached and the circular buffer is full, elements are overwritten.

A circular buffer makes sense when the amount of available memory is limited, and you need to prevent a container from growing arbitrarily big. Another example is continuous data flow where old data becomes irrelevant as new data becomes available. Memory is automatically reused by overwriting old data.

To use the circular buffer from Boost.CircularBuffer, include the header file `boost/circular_buffer.hpp`. This header file defines the class `boost::circular_buffer`.

Example 16.1. Using `boost::circular_buffer`

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

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  std::cout << cb.capacity() << '\n';
  std::cout << cb.size() << '\n';

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);

  std::cout << cb.size() << '\n';

  cb.push_back(3);
  cb.push_back(4);
  cb.push_back(5);

  std::cout << cb.size() << '\n';

  for (int i : cb)
    std::cout << i << '\n';
}
```

`boost::circular_buffer` is a template and must be instantiated with a type. For instance, the circular buffer **cb** in Example 16.1 stores numbers of type `int`.

The capacity of the circular buffer is specified when instantiating the class, not through a template parameter. The default constructor of `boost::circular_buffer` creates a buffer with a capacity of zero elements. Another constructor is available to set the capacity. In Example 16.1, the buffer **cb** has a capacity of three elements.

The capacity of a circular buffer can be queried by calling `capacity()`. In Example 16.1, `capacity()` will return 3.

The capacity is not equivalent to the number of stored elements. While the return value of `capacity()` is constant, `size()` returns the number of elements in the buffer, which may be different. The return value of `size()` will always be between 0 and the capacity of the circular buffer.

Example 16.1 returns 0 the first time `size()` is called since the buffer does not contain any data. After calling `push_back()` three times, the buffer contains three elements, and the second call to `size()` will return 3. Calling `push_back()` again does not cause the buffer to grow. The three new numbers overwrite the previous three. Therefore, `size()` will return 3 when called for the third time.

As a verification, the stored numbers are written to standard output at the end of Example 16.1. The output contains the numbers 3, 4, and 5 since the previously stored numbers have been overwritten.

Example 16.2. Various member functions of `boost::circular_buffer`

```cpp
#include <boost/circular_buffer.hpp>
#include <iostream>

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);
  cb.push_back(3);

  std::cout << std::boolalpha << cb.is_linearized() << '\n';

  circular_buffer::array_range ar1, ar2;

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';

  for (int i : cb)
    std::cout << i << '\n';

  cb.linearize();

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';
}
```

Example 16.2 uses the member functions `is_linearized()`, `array_one()`, `array_two()` and `linearize()`, which do not exist in other containers. These member functions clarify the internals of the circular buffer.

A circular buffer is essentially comparable to `std::vector`. Because the beginning and end are well defined, a vector can be treated as a conventional C array. That is, memory is contiguous, and the first and last elements are always at the lowest and highest memory address. However, a circular buffer does not offer such a guarantee.

Even though it may sound strange to talk about the beginning and end of a circular buffer, they do exist. Elements can be accessed via iterators, and `boost::circular_buffer` provides member functions such as `begin()` and `end()`. While you don’t need to be concerned about the position of the beginning and end when using iterators, the situation becomes a bit more complicated when accessing elements using regular pointers, unless you use `is_linearized()`, `array_one()`, `array_two()`, and `linearize()`.

The member function `is_linearized()` returns `true` if the beginning of the circular buffer is at the lowest memory address. In this case, all the elements in the buffer are stored consecutively from beginning to the end at increasing memory addresses, and elements can be accessed like a conventional C array.

If `is_linearized()` returns `false`, the beginning of the circular buffer is not at the lowest memory address, which is the case in Example 16.2. While the first three elements 0, 1, and 2 are stored in exactly this order, calling `push_back()` for the fourth time will overwrite the number 0 with the number 3. Because 3 is the last element added by a call to `push_back()`, it is now the new end of the circular buffer. The beginning is now the element with the number 1, which is stored at the next higher memory address. This means elements are no longer stored consecutively at increasing memory addresses.

If the end of the circular buffer is at a lower memory address than the beginning, the elements can be accessed via two conventional C arrays. To avoid the need to calculate the position and size of each array, `boost::circular_buffer` provides the member functions `array_one()` and `array_two()`.

Both `array_one()` and `array_two()` return a `std::pair` whose first element is a pointer to the corresponding array and whose second element is the size. `array_one()` accesses the array at the beginning of the circular buffer, and `array_two()` accesses the array at the end of the buffer.

If the circular buffer is linearized and `is_linearized()` returns `true`, `array_two()` can be called, too. However, since there is only one array in the buffer, the second array contains no elements.

To simplify matters and treat the circular buffer as a conventional C array, you can force a rearrangement of the elements by calling `linearize()`. Once complete, you can access all stored elements using `array_one()`, and you don’t need to use `array_two()`.

Boost.CircularBuffer offers an additional class called `boost::circular_buffer_space_optimized`. This class is also defined in `boost/circular_buffer.hpp`. Although this class is used in the same way as `boost::circular_buffer`, it does not reserve any memory at instantiation. Rather, memory is allocated dynamically when elements are added until the capacity is reached. Removing elements releases memory accordingly. `boost::circular_buffer_space_optimized` manages memory more efficiently and, therefore, can be a better choice in certain scenarios. For example, it may be a good choice if you need a circular buffer with a large capacity, but your program doesn’t always use the full buffer.