```
#ifndef _circular_buffer_hpp
#define _circular_buffer_hpp

#include <array>
#include <iterator>
#include <algorithm>
#include <iostream>
#include <cmath>
#include <stdexcept>

template<typename T, int cap, typename Container = std::array<T, cap>>
class CircularBuffer {
  public :
    using value_type = T;

    CircularBuffer() : _head(0), _tail(0), _size(0), _current(0), _capacity(cap) {}

    T& head() {
      return c.at(_head);
    }

    T& tail() {
      return c.at(_tail);
    }

    T const& head() const {
      return c.at(_head);
    }

    T const& tail() const {
      return c.at(_tail);
    }

    void push_back(T val) noexcept {
      if (_current >= _capacity) {
        _current = 0;
      }

      c.at(_current++) = val;

      _tail = _current - 1;

      if (_size++ >= _capacity) {
        _size = _capacity;
        _head++;
        if (_head >= _capacity) {
          _head = 0;
        }
      }
    }

    void place_back(T val) {
      if (full()) {
        throw std::overflow_error("place_back(): full buffer");
      }

      push_back(val);
    }

    void pop() {
      if (_size <= 0) {
        throw std::underflow_error("pop(): empty buffer");
      }

      _head++;
      if (_head >= _capacity) {
        _head = 0;
      }
      _size--;
    }

    std::size_t size() const noexcept {
      return _size;
    }

    std::size_t capacity() noexcept {
      return _capacity;
    }

    bool empty() const noexcept {
      return (size() <= 0);
    }

    bool full() const noexcept {
      return (int)size() >= _capacity;
    }

    T& operator[](std::size_t index);
    T const& operator[](std::size_t index) const;

    /**
     * Create a circular buffer iterator
     */
    template <typename Buffer, typename Iterator>
    class CircularBufferIterator {
      public :
        using iterator_category = std::forward_iterator_tag;
        using value_type = typename Buffer::value_type;
        using difference_type = std::ptrdiff_t;
        using pointer = typename Buffer::value_type*;
        using reference = typename Buffer::value_type&;

        CircularBufferIterator() : _done(true) {}

        CircularBufferIterator(const Buffer& buf, Iterator begin) :
          _buf(buf), _begin(begin), _cursor(begin), _done(false) {}

        CircularBufferIterator(const Buffer& buf, Iterator begin, bool done) :
          _buf(buf), _begin(begin), _cursor(begin), _done(done) {}

        reference operator*() const {
          return *_cursor;
        }

        pointer operator->() const {
          return _cursor;
        }

        CircularBufferIterator& operator++() {
          ++_cursor;
          if (_cursor == _buf.end()) {
            _cursor = (Iterator)_buf.begin();
          }

          _done = _cursor == _begin;

          return *this;
        }

        CircularBufferIterator operator++(int) {
          iterator tmp(*this);
          ++_cursor;
          if (_cursor == _buf.end()) {
            _cursor = (Iterator)_buf.begin();
          }

          _done = _cursor == _begin;
          return tmp;
        }

        bool operator==(const CircularBufferIterator& it) const {
          if (_done && it._done) {
            return true;
          }
          else if (!_done && !it._done) {
            return (this->_cursor == it._cursor);
          }

          return false;
        }

        bool operator!=(const CircularBufferIterator& it) const {
          return !(*this == it);
        }

      private :
        const Buffer& _buf;
        const Iterator _begin;
        Iterator _cursor;
        bool _done;
    };

    typedef CircularBufferIterator<Container, typename Container::iterator> iterator;

    iterator begin() {
      unsigned int offset = _head % _capacity;
      return CircularBuffer::iterator(c, c.begin() + offset);
    }

    iterator end() {
      unsigned int offset = _tail + 1 % _capacity;
      return CircularBuffer::iterator(c, c.begin() + offset, full());
    }

    friend std::ostream& operator<<(std::ostream& os, const CircularBuffer& buf) {
      return (os << "head: " << buf._head << ", tail: " << buf._tail << ", current: "
          << buf._current << ", capacity: " << buf._capacity << ", size: " << buf.size
());
    }

  private :
    Container c;
    int _head;
    int _tail;
    int _size;
    int _current;
    int _capacity;
};

#endif
```