SWO3 | **Übung zu Softwareentwicklung mit klassischen Sprachen und Bibliotheken 3** | **WS 2018/19, Angabe 9**

☐ Gruppe 1 (J. Heinzelreiter)

☐ Gruppe 2 (M. Hava) Name: Niklas Vest Aufwand [h]: 10

☒ Gruppe 3 (P. Kulczycki) Übungsleiter/Tutor: ______________ Punkte: ________

Beispiel	Lösungsidee (max. 100%)	Implement. (max. 100%)	Testen (max. 100%)
1 (100 P)	75%	90%	60%

Beispiel 1: swo3::deque (`src/deque/`)

Implementieren Sie einen ADT swo3::deque (*double-ended queue*, siehe https://en.wikipedia.org/wiki/Double-ended_queue) gemäß dem im Folgenden definierten Interface. Eine swo3::deque speichert ihre Elemente in einem Ringpuffer (siehe https://en.wikipedia.org/wiki/Circular_buffer). Testen Sie ausführlich unter Zuhilfenahme von generischen Algorithmen und *range-based for loops* (siehe https://en.cppreference.com/w/cpp/language/range-for). Für eine genaue Spezifikation der einzelnen Komponenten der swo3::deque (Typen, Methoden etc.) verweisen wir auf https://en.cppreference.com/w/cpp/container/deque und https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator.

```
namespace swo3 {

/**
 * see https://en.cppreference.com/w/cpp/container/deque and
 *     https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator
 */
template <typename T> class deque final {
   using value_type = ...
   using reference  = ...
   using size_type  = ...

   class iterator final {   // implements RandomAccessIterator
      ...
   };

   deque ();
   explicit deque (size_type count);
   deque (size_type count, T const & value);

   deque (deque const & other);
   deque (deque && other);
   deque (std::initializer_list <T> init);

  ~deque ();

   deque & operator = (deque const & other);
   deque & operator = (deque && other) noexcept;
   deque & operator = (std::initializer_list <T> init);

   reference operator [] (size_type pos);

   reference at    (size_type pos);
   reference back  ();
   reference front ();
   iterator begin () noexcept;
   iterator end   () noexcept;

   bool      empty () const noexcept;
   size_type size  () const noexcept;

   void clear () noexcept;

   void push_back (T const & value);
   void push_back (T && value);
   void pop_back  ();

   void push_front (T const & value);
   void push_front (T && value);
   void pop_front  ();

   void resize (size_type count);
   void swap   (deque & other) noexcept;

   ...
};

template <typename T> bool operator == (deque const & lhs, deque const & rhs);
template <typename T> bool operator != (deque const & lhs, deque const & rhs);
template <typename T> bool operator <  (deque const & lhs, deque const & rhs);
template <typename T> bool operator <= (deque const & lhs, deque const & rhs);
template <typename T> bool operator >  (deque const & lhs, deque const & rhs);
template <typename T> bool operator >= (deque const & lhs, deque const & rhs);

}   // namespace swo3
```

Ausarbeitung 09

Niklas Vest

January 9, 2019

1 Deque

1.1 Lösungsidee

1.1.1 Ringbuffer

Die Deque speichert den Beginn des allokierten Speichers *_buff* und dessen Kapazität *_capacity*. Weiters hat die Deque zwei Felder *_begin* und *_end* die jeweils den Anfang und das Ende des Ringpuffers ”verwalten”. Wird hinten eingefügt, wird *_end* um ein Element nach rechts verschoben. Beim Einfügen am Anfang der Deque wird *_beginn* nach links verschoben. An der neuen Position des Pointers wird dann ein neues Element erstellt. Stellt die Deque beim Einfügen fest, dass die maximale Größe erreicht worden ist, vergrößert sie sich automatisch auf die doppelte Kapazität.

1.1.2 Schnittstellen

Die Schnittstellen sind im Code genauer dokumentiert und das Verhalten überschneidet sich großteils mit dem der *std::deque*. Der einzige unterschied ist, dass bei einem Aufruf von *std::deque<T>::resize* lediglich die Kapazität erhöht wird, die freien Plätze aber unbesetzt bleiben. (vgl. value-initialization bei *resize* in standard containern!)

1.1.3 Anmerkungen

Der *const* Qualifizierer wurde bei den Vergleichsoperatioren weggelassen, weil sonst sehr viele *const* Overloads sowohl für die deque, als auch deren *iterator* notwendig wären. Das würde ich idealerweise unter Verwendung von Delegation mittels *const_cast* lösen. Ich habe allerdings mein für mich gesetztes Arbeitspensum bereits erreicht und möchte weitere Details nicht mehr ausprogrammieren, weil ich eine Balance mit anderen Fächern halten muss.

Weiters ist die Klasse *iterator* komplett *in-class* implementiert, weil die Definition von Methoden auSSerhalb der Klasse relativ umständlich ist aufgrund der Generizität der ”Eltern-Klasse” *deque* und der Namensauflösung in generischen Kontexten.

1.2 Implementierung

Listing 1: deque.hpp

```cpp
#pragma once

#include <initializer_list>
#include <memory>

namespace swo {

template <typename T>
class deque;

/**
 * @tparam T A type that implements ==.
 * @return True if lhs_i is equal to rhs_i for all i in [0;|lhs|).
 */
template <typename T>
bool operator==(deque <T> &lhs, deque <T> &rhs);

/**
 * @tparam T A type that implements ==.
 * @return True if lhs == rhs returns False.
 */
template <typename T>
bool operator!=(deque <T> &lhs, deque <T> &rhs);

/**
 * @tparam T A type that implements <.
 * @return True if lhs_i < rhs_i for all i in [0;min(|lhs|, |rhs|)).
 */
template <typename T>
bool operator<(deque <T> &lhs, deque <T> &rhs);

/**
 * @tparam T A type that implements < and ==.
 * @return Returns true if lhs < rhs or lhs == rhs.
 */
template <typename T>
bool operator<=(deque <T> &lhs, deque <T> &rhs);

/**
 * @tparam T A type that implements < and ==.
 * @return True if no lhs_i <= rhs_i for all i in [0;min(|lhs|, |rhs|)).
 */
template <typename T>
bool operator>(deque <T> &lhs, deque <T> &rhs);

/**
 * @tparam T A type that implements < and ==.
 * @return True if lhs is not less than rhs.
 */
template <typename T>
bool operator>=(deque <T> &lhs, deque <T> &rhs);

/**
 * Swaps the contents of lhs and rhs
 */
template <typename T>
void swap(deque <T> &lhs, deque <T> &rhs);

template <typename T>
class deque final
{

public: // typedefs
    using value_type = T;
    using reference = T &;
    using size_type = std::size_t;

public: // nested classes

    // The iterator is implemented completely in−class
    // because it is hard to define methods of classes
    // nested inside template classes outside the actual
    // class definition .
    class iterator
    {
        friend class deque;

        /**
        * @param it The iterator to advance.
        * @param n The number of moves.
        * @return A new iterator moved forward by n elements.
        */
        friend iterator operator+(const iterator &it, size_type n)
        {
            auto cpy = it;
            return cpy += n;
        }

        /**
        * @param it The iterator to move back.
        * @param n The number of moves.
        * @return A new iterator moved backward by n elements.
        */
        friend iterator operator-(const iterator &it, size_type n)
        {
            auto cpy = it;
            return cpy -= n;
        }

        /**
        * @return Returns the number of elements by which the iterators
        *         lhs and rhs differ .
        */
        friend size_type operator-(const iterator &lhs, const iterator &rhs)
        {
            size_type distance = 0;
            auto rhs_cpy = rhs;
            while (rhs_cpy != lhs) {
                distance += 1;
                rhs_cpy += 1;
            }
            return distance;
        }

        /**
        * @return True if the iterators refer to the same element.
        */
        friend bool operator==(const iterator &lhs, const iterator &rhs)
        {
            return lhs._current == rhs._current && lhs._first == rhs._first;
        }

        /**
        * @return True if lhs == rhs returns false.
        */
        friend bool operator!=(const iterator &lhs, const iterator &rhs)
        {
            return !(lhs == rhs);
        }

        /**
        * @return True if lhs denotes an element further to the begin of
        *         the deque than rhs.
        */
        friend bool operator<(const iterator &lhs, const iterator &rhs)
        {
            return rhs - lhs > 0;
        }

        /**
        * @return True if the iterators are euqal or lhs denotes an element
        *         further to the begin of the deque than rhs.
        */
        friend bool operator<=(const iterator &lhs, const iterator &rhs)
        {
            return !(lhs > rhs);
        }

        /**
        * @return  True if rhs denotes an element further to the begin of
        *          the deque than lhs.
        */
        friend bool operator>(const iterator &lhs, const iterator &rhs)
        {
            return rhs < lhs;
        }

        /**
        * @return True if the iterators are equal or rhs denotes an element
        *         further to the begin of the deque than lhs.
        */
        friend bool operator>=(const iterator &lhs, const iterator &rhs)
        {
            return !(lhs < rhs);
        }

    public: // methods

```

```
        /**
        * @param n The stride by which to move the iterator forward.
        * @return The moved iterator.
        */
        iterator &operator+=(size_type n)
        {
            if (n > 0) {
                while (n != 0) {
                    n--;
                    this->_current = this->_parent._next(this->_current);
                }
            } else {
                while (n != 0) {
                    n++;
                    this->_current = this->_parent._previous(this->_current);
                }
            }
            _first = false;
            return *this;
        }

        /**
        * @param n The stride by which to move the iterator backward
        * @return The moved iterator.
        */
        iterator &operator-=(size_type n)
        {
            *this += -n;
            return *this;
        }

        /**
        * @param i The index of the deque to refer to, relative
        *        to the element currently being referred to by
        *        the iterator.
        * @return The moved iterator.
        */
        reference operator[](size_type i)
        {
            return *(*this + i);
        }

        /**
        * @return A reference to the element being referred to
        *         by the iterator.
        */
        reference operator*()
        {
            assert(_current);
            return *_current;
        }

        /**
        * Advances the iterator forward by 1.
        * This operation exists to allow this iterator
        * to be used with range-based for loops.
        * @return The advanced iterator.
```

```cpp
        */
        iterator &operator++()
        {
            return *this += 1;
        }

        /**
        * @see iterator#operator++()
        */
        iterator operator++(int)
        {
            iterator cpy { *this };
            ++*this;
            return cpy;
        }

    protected: // methods
        /**
        * @param parent The container in which the elements referred to
        *        by the iterator are contained.
        * @param deq_it The element the iterator should refer to.
        * @param begin Whether the iterator was created using a call
        *        to parent#begin.
        */
        iterator(deque <T> &parent, T *deq_it, bool begin)
                : _parent { parent }, _current { deq_it }, _first { begin }
        {}

    protected: // members

        /**
        * The container in which the elements referred to
        * by the iterator are contained.
        */
        deque <T> &_parent;

        /**
        * The element the iterator should refer to.
        */
        T *_current;

        /**
        * Whether the iterator was created using a call
        * to parent#begin.
        *
        * (This is required for OTE-Iterator behaviour!)
        */
        bool _first;
    };

public: // methods

    /**
     * Creates an empty deque.
     */
    deque();

    /**
     * Creates an emtpy deque with a initial capacity.
     * @param capacity The initial capacity.
     */
    explicit deque(size_type capacity);

    /**
     * Creates a deque with a default capacity and fills
     * it up with copies of the supplied value.
     * @param capacity The initial capacity.
     * @param value The value to use for initialization.
     */
    deque(size_type capacity, const T &value);

    /**
     * Copy constructor.
     */
    deque(const deque &other);

    /**
     * Move constructor.
     */
    deque(deque &&other) noexcept;

    /**
     * Creates a deque from the supplied initializer list .
     * @param il A list of elements to initialize the deque with.
     */
    deque(std::initializer_list <T> il);

    /**
     * Copy assignment operator.
     */
    deque &operator=(const deque &other);

    /**
     * Move assignment operator.
     */
    deque &operator=(deque &&other) noexcept;

    /**
     * Overwrites the queue withe the values contained
     * in the supplied initializer list .
     */
    deque &operator=(std::initializer_list <T> il);

    /**
     * @param pos The position of the element to fetch from the deque.
     * @return A reference to the element at the specified position.
     */
    reference operator[](size_type pos);

    /**
     * Does the same as {@link deque#operator[]} but does range
     * checking in addition.
     */
    reference at(size_type pos);

    /**
     * @return The last element in the deque.
     */
    reference back();

    /**
     * @return The first element in the deque.
     */
    reference front();

    /**
     * @return An iterator referring to the first element in the deque.
     */
    iterator begin() noexcept;

    /**
     * @return An iterator referring to one past the last element (OTE)
     *         in the deque.
     */
    iterator end() noexcept;

    /**
     * @return True if the deque does not contain any elements.
     */
    bool empty() const noexcept;

    /**
     * @return The number of elements in the deque.
     */
    size_type size() const noexcept;

    /**
     * Removes all elements from the deque.
     */
    void clear() noexcept;

    /**
     * Adds an element to the back of the deque by copying
     * the supplied value.
     * @param value The value to add.
     */
    void push_back(const T &value);

    /**
     * Adds an element to the back of the deque by moving
     * the supplied value.
     * @param value The value to add.
     */
    void push_back(T &&value);

    /**
     * Remove the last element of the deque.
     */
    void pop_back();

    /**
```

```
     * Adds an element to the front of the deque by copying
     * the supplied value.
     * @param value The value to add.
     */
    void push_front(const T &value);

    /**
     * Adds an element to the front of the deque by moving
     * the supplied value.
     * @param value The value to add.
     */
    void push_front(T &&value);

    /**
     * Removes the first element of the deque.
     */
    void pop_front();

    /**
     * Increases the deque's capacity to provide enough
     * room to host the supplied number elements.
     * @param count
     */
    void resize(size_type count);

    /**
     * Swaps this deque's contents with those of _other_
     * @param other The deque to swap contents with.
     */
    void swap(deque &other) noexcept;

    /**
     * Destructor.
     */
    virtual ~deque();

private: // methods

    /**
     * Frees memory associated with the deque.
     */
    void _deallocate();

    /**
     * Makes sure that the supplied pointer is not null and
     * throws an error including the supplied message otherwise.
     * @param p The pointer to check
     * @param message The message to include in the thrown error
     *         if the pointer is null.
     */
    void _assert_not_null(const T *p, const std::string &message);

    /**
     * @param it A pointer to an element in the deque.
     * @return A pointer to the next element in the deque.
     */
    T *_next(const T *it) const;
```

```cpp

    /**
     * @param it A pointer to an element in the deque.
     * @return A pointer to the previous element in the deque.
     */
    T *_previous(const T *it) const;

private: // constants
    const static size_type _INITIAL_CAPACITY { 5 };

private: // members

    /**
     * The allocation method to use for the deque buffer.
     */
    std::allocator <T> _alloc {};

    /**
     * A pointer to the first element in the buffer.
     */
    T *_buffer { nullptr };

    /**
     * The begin of the container. Off-The-End pointer
     * for empty deques.
     */
    T *_begin { nullptr };

    /**
     * Apointer to the last element in the container.
     */
    T *_end { nullptr };

    /**
     * The current capacity of the deque.
     */
    size_type _capacity { 0 };

    void _prepare_push_back();

    void _prepare_push_front();
};

template <typename T>
deque <T>::deque() : deque(_INITIAL_CAPACITY)
{}

template <typename T>
deque <T>::deque(size_type capacity)
        : _capacity { capacity }, _buffer { _alloc.allocate(capacity) }
{}

template <typename T>
deque <T>::deque(size_type capacity, const T &value)
        : deque(capacity)
{
    _begin = _buffer;
    _end = _begin + capacity - 1;
    // Fill unconstructed memory pointed to by _buffer/_begin
    // with copies of _value_
    std::uninitialized_fill(_buffer, _buffer + capacity, value);
}

template <typename T>
deque <T>::deque(const deque &other)
{
    _capacity = other._capacity;
    _buffer = _alloc.allocate(other._capacity);
    std::copy(other._buffer, other._buffer + _capacity, _buffer);
    // make _begin and _end point to the correct elements relative
    // to the offsets of the other deque's _begin!
    _begin = _buffer + (other._begin - other._buffer);
    _end = _buffer + (other._end - other._buffer);
}

template <typename T>
deque <T>::deque(deque &&other) noexcept
{
    swap(other);
}

template <typename T>
deque <T>::deque(std::initializer_list <T> il)
        : deque(il.size())
{
    _begin = _buffer;
    _end = _begin + _capacity - 1;
    // Copy elements from the initializer list into the unconstructed
    // memory pointed to by _buffer/_begin
    std::uninitialized_copy(std::cbegin(il), std::cend(il), _buffer);
}

template <typename T>
deque <T>::~deque()
{
    _deallocate();
}

template <typename T>
deque <T> &deque <T>::operator=(const deque &other)
{
    if (this != &other) {
        // first destruct
        _deallocate();
        // then copy other deque
        _capacity = other._capacity;
        _buffer = _alloc.allocate(other._capacity);
        std::copy(other._buffer, other._buffer + _capacity, _buffer);
        _begin = _buffer + (other._begin - other._buffer);
        _end = _buffer + (other._end - other._buffer);
    }
    return *this;
}

```

```
template <typename T>
deque <T> &deque <T>::operator=(deque &&other) noexcept
{
    swap(other);
    return *this;
}

template <typename T>
deque <T> &deque <T>::operator=(std::initializer_list <T> il)
{
    *this = deque(std::move(il));
    return *this;
}

template <typename T>
auto deque <T>::operator[](size_type pos) -> reference
{
    return begin()[pos];
}

template <typename T>
auto deque <T>::at(size_type pos) -> reference
{
    if (pos >= size() || pos < 0) {
        throw std::invalid_argument("Index out of bounds.");
    }
    return (*this)[pos];
}

template <typename T>
auto deque <T>::back() -> reference
{
    T *elem = _end;
    _assert_not_null(elem, "Can not back() an empty container.");
    return *elem;
}

template <typename T>
auto deque <T>::front() -> reference
{
    T *elem = _begin;
    _assert_not_null(elem, "Can not front() an empty container.");
    return *elem;
}

template <typename T>
bool deque <T>::empty() const noexcept
{
    return _begin == nullptr;
}

template <typename T>
auto deque <T>::size() const noexcept -> size_type
{
    auto it = _begin;
    // _end points to the last element which must
    // also be accounted for in the total size.
    // Hence for non-empty deques, _size_ starts
    // at 1!
    size_type size = it == nullptr ? 0 : 1;
    while (it != _end) {
        size += 1;
        it = _next(it);
    }
    return size;
}

template <typename T>
void deque <T>::clear() noexcept
{
    auto it = _begin;
    while (it != _end) {
        _alloc.destroy(it);
        it = _next(it);
    }
    // Not dealloacting the buffer
    // so it can be reused
    _begin = nullptr;
    _end = nullptr;
}

template <typename T>
void deque <T>::push_back(const T &value)
{
    _prepare_push_back();
    _alloc.construct(_end, value);
}

template <typename T>
void deque <T>::push_back(T &&value)
{
    _prepare_push_back();
    _alloc.construct(_end, std::move(value));
}

template <typename T>
void deque <T>::pop_back()
{
    if (!empty()) {
        _alloc.destroy(_end);
        if (_begin == _end) {
            _begin = _end = nullptr;
        } else {
            _end = _previous(_end);
        }
    }
}

template <typename T>
void deque <T>::push_front(const T &value)
{
    if (empty()) {
        push_back(value);
    } else {
        _prepare_push_front();
        _alloc.construct(_begin, value);
    }
}

template <typename T>
void deque <T>::push_front(T &&value)
{
    if (empty()) {
        push_back(std::move(value));
    } else {
        _prepare_push_front();
        _alloc.construct(_begin, std::move(value));
    }
}

template <typename T>
void deque <T>::pop_front()
{
    if (!empty()) {
        _alloc.destroy(_begin);

        if (_begin == _end) {
            _end = nullptr;
            _begin = nullptr;
        } else {
            _begin = _next(_begin);
        }
    }
}

template <typename T>
void deque <T>::_prepare_push_back()
{
    if (empty()) {
        _begin = _buffer;
        _end = _buffer;
    } else {
        auto it = _next(_end);
        // if we bump into the begin of the ring buffer
        if (it == _begin) {
            // resize dis boi
            resize(_capacity * 2);
        }
        _end = _next(_end);
    }
}

template <typename T>
void deque <T>::_prepare_push_front()
{
    auto it = _previous(_begin);
    if (it == _end) {
        resize(_capacity * 2);
    }
    _begin = _previous(_begin);
}
```

```

template <typename T>
void deque <T>::_deallocate()
{
    if (_buffer != nullptr) {

        // If the buffer still holds fully unconstructed
        // memory, _alloc.destroy() is an invalid op!

        if (_end != nullptr) {
            auto buff_it = _begin;
            while (buff_it != _end) {
                _alloc.destroy(buff_it);
                buff_it = _next(buff_it);
            } // TODO test
            _begin = nullptr;
            _end = nullptr;
        }

        _alloc.deallocate(_buffer, _capacity);
        _buffer = nullptr;
        _capacity = 0;

    }
}

template <typename T>
void deque <T>::_assert_not_null(const T *p, const std::string &message)
{
    if (p == nullptr) {
        throw std::range_error(message);
    }
}

template <typename T>
T *deque <T>::_next(const T *it) const
{
    // Add 1 to the total difference in elements and keep it
    // in the range [0;_capacity) using modulo.
    return _buffer + ((it - _buffer + 1) % _capacity);
}

template <typename T>
T *deque <T>::_previous(const T *it) const
{
    // Subtract 1 from the total difference in elements and
    // keep it in the range [0;_capacity) using modulo.
    return _buffer + ((it - _buffer - 1 + _capacity) % _capacity);
}

template <typename T>
void deque <T>::resize(deque::size_type count)
{
    // allocate a second buffer
    T *new_buff = _alloc.allocate(count);

    // write old values to new buffer
    auto it = _begin;
    auto new_buff_cpy = new_buff;
    while (it != _end) {
        _alloc.construct(new_buff_cpy++, *it);
        it = _next(it);
    }

    // delete the old buffer
    auto s = size();
    _deallocate();

    // reassign members
    _buffer = new_buff;
    _begin = _buffer;
    _end = _begin + s - 1;
    _capacity = count;
}

template <typename T>
void deque <T>::swap(deque &other) noexcept
{
    using std::swap;
    swap(_alloc, other._alloc);
    swap(_buffer, other._buffer);
    swap(_begin, other._begin);
    swap(_end, other._end);
    swap(_capacity, other._capacity);
}

template <typename T>
auto deque <T>::begin() noexcept -> deque <T>::iterator
{
    return deque <T>::iterator(*this, _begin, true);
}

template <typename T>
auto deque <T>::end() noexcept -> deque <T>::iterator
{
    // OTE-Iterator
    return iterator(*this, _next(_end), false);
}

template <typename T>
bool operator==(deque <T> &lhs, deque <T> &rhs)
{
    bool equal = false;
    if (lhs.size() == rhs.size()) {
        equal = true;
        auto lhs_it = lhs.begin();
        auto rhs_it = rhs.begin();
        while (lhs_it != lhs.end() && equal) {
            equal = *lhs_it++ == *rhs_it++;
        }
    }
    return equal;
}
```

```
template <typename T>
bool operator!=(deque <T> &lhs, deque <T> &rhs)
{
    return !(lhs == rhs);
}

template <typename T>
bool operator<(deque <T> &lhs, deque <T> &rhs)
{
    bool less = true;
    auto lhs_it = lhs.begin();
    auto rhs_it = rhs.begin();
    while (lhs_it != lhs.end() && rhs_it != rhs.end() && less) {
        less = *lhs_it++ < *rhs_it++;
    }
    return less;
}

template <typename T>
bool operator<=(deque <T> &lhs, deque <T> &rhs)
{
    return lhs < rhs || lhs == rhs;
}

template <typename T>
bool operator>(deque <T> &lhs, deque <T> &rhs)
{
    return !(lhs <= rhs);
}

template <typename T>
bool operator>=(deque <T> &lhs, deque <T> &rhs)
{
    return !(lhs < rhs);
}

template <typename T>
void swap(deque <T> &lhs, deque <T> &rhs)
{
    lhs.swap(rhs);
}

} // namespace swo
```

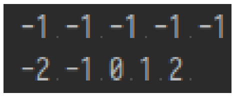
Listing 2: main.cpp

```
#include <iostream>
#include <cassert>
#include "deque.hpp"


using swo::deque;

int main()
{
    /*
     * Constructors
     * (Checked with debugger for simplcitiy)
     */

    // construct default
    deque <int> di1;
    // construct from capacity
    deque <int> di2(100);
    // construct from capacity and value
    deque <int> di3(5, -1);
    // construct from initializer list
    deque <int> di4 { 1, 2, 3, 4, 5 };

    // copy from lvalue reference
    deque <int> di5(di4);
    // copy from rvalue reference
    deque <int> di6(std::move(di4));

    /*
     * Operators
     */

    // =
    // copy assignment
    di4 = di3;
    // move assignment
    di5 = std::move(di6);
    // initializer list assignment
    di6 = { -2, -1, 0, 1, 2 };

    // []
    assert(di6[0] == -2);
    assert(di6[4] == 2);
    assert(di4[2] == -1);

    /*
     * Methods
     */
    // at
    assert(di6.at(0) == -2);
    assert(di4.at(3) == -1);
    bool threw = false;
    try {
        di4.at(100);
    } catch (const std::exception &exc) {
        threw = true;
    }
    assert(threw);

    // front
    assert(di6.front() == -2);
    assert(di5.front() == 1);

    // back
    assert(di6.back() == 2);
    assert(di5.back() == 5);

    // empty
    assert(di1.empty());
    assert(di2.empty());
    assert(!di3.empty());
    assert(!di6.empty());

    // size
    assert(di1.size() == 0);
    assert(di2.size() == 0);
    assert(di3.size() == 5);
    assert(deque <int>({ 1, 2, 3 }).size() == 3); // all other deques had size 5 :/

    /*
     * Void bois
     */
    // clear
    deque <char> ds1 = { 'h', 'e', 'l', 'l', 'o' };
    ds1.clear();
    assert(ds1.empty());

    // push_back
    ds1.push_back('h');
    ds1.push_back('i');
    assert(ds1.size() == 2);

    // pop_back
    ds1.pop_back();
    assert(ds1.size() == 1);
    assert(ds1.back() == 'h');

    // push_front
    ds1.push_front('a');
    assert(ds1.size() == 2);
    assert(ds1.front() == 'a');

    // pop_front
    ds1.pop_front();
    assert(ds1.front() == 'h');

    // swap
    swap(di3, di6);

    // comparisons
    deque <int> cmp1 { 1, 2, 3, 4, 5 };
    deque <int> cmp2 { 2, 3, 4, 5, 6 };
    assert(cmp1 < cmp2);
    assert(cmp1 == cmp1);
    assert(cmp1 != cmp2);
    assert(cmp1 >= cmp1);
    assert(cmp1 >= cmp1);

    /*
     * Iterators
     * (Ony rudimentary tests)
     */
    auto it = di6.begin();
    while (it != di6.end()) {
        std::cout << *it << ' ' << std::flush;
```

```cpp
        it += 1;
    }
    std::cout << std::endl;

    auto beg = di3.begin();
    auto mid = di3.begin() + (di3.size() / 2);
    assert(mid - beg == di3.size() / 2);
    assert(beg - di3.begin() == 0);

    for (auto el : di3) {
        std::cout << el << ' ' << std::flush;
    }
    std::cout << std::endl;

    return 0;
}
```

1.3 Tests

Alle Assertions in *main.cpp* waren erfolgreich und in der folgenden Abbildung sehen sie die (korrekte) Ausgabe der Print-Statements:



```
-1 -1 -1 -1 -1
-2 -1 0 1 2
```