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 Kert 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

```
1  #pragma once
2
3  #include <initializer_list>
4  #include <memory>
5
6  namespace swo {
7
8  template <typename T>
9  class deque;
10
11 /**
12  * @tparam T A type that implements ==.
13  * @return True if lhs_i is equal to rhs_i for all i in [0;|lhs|).
14  */
15 template <typename T>
16 bool operator==(deque <T> &lhs, deque <T> &rhs);
17
18 /**
19  * @tparam T A type that implements ==.
20  * @return True if lhs == rhs returns False.
21  */
22 template <typename T>
23 bool operator!=(deque <T> &lhs, deque <T> &rhs);
24
25 /**
26  * @tparam T A type that implements <.
27  * @return True if lhs_i < rhs_i for all i in [0;min(|lhs|, |rhs|)).
28  */
29 template <typename T>
30 bool operator<(deque <T> &lhs, deque <T> &rhs);
31
32 /**
33  * @tparam T A type that implements < and ==.
34  * @return Returns true if lhs < rhs or lhs == rhs.
35  */
36 template <typename T>
37 bool operator<=(deque <T> &lhs, deque <T> &rhs);
38
39 /**
40  * @tparam T A type that implements < and ==.
41  * @return True if no lhs_i <= rhs_i for all i in [0;min(|lhs|, |rhs|)).
42  */
43 template <typename T>
44 bool operator>(deque <T> &lhs, deque <T> &rhs);
45
46 /**
47  * @tparam T A type that implements < and ==.
48  * @return True if lhs is not less than rhs.
49  */
50 template <typename T>
51 bool operator>=(deque <T> &lhs, deque <T> &rhs);
52
53 /**
54  * Swaps the contents of lhs and rhs
```

```
55  */
56 template <typename T>
57 void swap(deque <T> &lhs, deque <T> &rhs);
58
59 template <typename T>
60 class deque final
61 {
62
63 public: // typedefs
64     using value_type = T;
65     using reference = T &;
66     using size_type = std::size_t;
67
68 public: // nested classes
69
70     // The iterator is implemented completely in-class
71     // because it is hard to define methods of classes
72     // nested inside template classes outside the actual
73     // class definition.
74     class iterator
75     {
76         friend class deque;
77
78         /**
79         * @param it The iterator to advance.
80         * @param n The number of moves.
81         * @return A new iterator moved forward by n elements.
82         */
83         friend iterator operator+(const iterator &it, size_type n)
84         {
85             auto cpy = it;
86             return cpy += n;
87         }
88
89         /**
90         * @param it The iterator to move back.
91         * @param n The number of moves.
92         * @return A new iterator moved backward by n elements.
93         */
94         friend iterator operator-(const iterator &it, size_type n)
95         {
96             auto cpy = it;
97             return cpy -= n;
98         }
99
100        /**
101        * @return Returns the number of elements by which the iterators
102        *         lhs and rhs differ.
103        */
104        friend size_type operator-(const iterator &lhs, const iterator &rhs)
105        {
106            size_type distance = 0;
107            auto rhs_cpy = rhs;
108            while (rhs_cpy != lhs) {
109                distance += 1;
110                rhs_cpy += 1;
111            }
```

```
112             return distance;
113         }
114
115         /**
116         * @return True if the iterators refer to the same element.
117         */
118         friend bool operator==(const iterator &lhs, const iterator &rhs)
119         {
120             return lhs._current == rhs._current && lhs._first == rhs._first;
121         }
122
123         /**
124         * @return True if lhs == rhs returns false.
125         */
126         friend bool operator!=(const iterator &lhs, const iterator &rhs)
127         {
128             return !(lhs == rhs);
129         }
130
131         /**
132         * @return True if lhs denotes an element further to the begin of
133         *         the deque than rhs.
134         */
135         friend bool operator<(const iterator &lhs, const iterator &rhs)
136         {
137             return rhs - lhs > 0;
138         }
139
140         /**
141         * @return True if the iterators are euqal or lhs denotes an element
142         *         further to the begin of the deque than rhs.
143         */
144         friend bool operator<=(const iterator &lhs, const iterator &rhs)
145         {
146             return !(lhs > rhs);
147         }
148
149         /**
150         * @return  True if rhs denotes an element further to the begin of
151         *          the deque than lhs.
152         */
153         friend bool operator>(const iterator &lhs, const iterator &rhs)
154         {
155             return rhs < lhs;
156         }
157
158         /**
159         * @return True if the iterators are equal or rhs denotes an element
160         *         further to the begin of the deque than lhs.
161         */
162         friend bool operator>=(const iterator &lhs, const iterator &rhs)
163         {
164             return !(lhs < rhs);
165         }
166
167     public: // methods
168
```

```
169         /**
170         * @param n The stride by which to move the iterator forward.
171         * @return The moved iterator.
172         */
173         iterator &operator+=(size_type n)
174         {
175             if (n > 0) {
176                 while (n != 0) {
177                     n--;
178                     this->_current = this->_parent._next(this->_current);
179                 }
180             } else {
181                 while (n != 0) {
182                     n++;
183                     this->_current = this->_parent._previous(this->_current);
184                 }
185             }
186             _first = false;
187             return *this;
188         }
189
190         /**
191         * @param n The stride by which to move the iterator backward
192         * @return The moved iterator.
193         */
194         iterator &operator-=(size_type n)
195         {
196             *this += -n;
197             return *this;
198         }
199
200         /**
201         * @param i The index of the deque to refer to, relative
202         *        to the element currently being referred to by
203         *        the iterator.
204         * @return The moved iterator.
205         */
206         reference operator[](size_type i)
207         {
208             return *(*this + i);
209         }
210
211         /**
212         * @return A reference to the element being referred to
213         *         by the iterator.
214         */
215         reference operator*()
216         {
217             assert(_current);
218             return *_current;
219         }
220
221         /**
222         * Advances the iterator forward by 1.
223         * This operation exists to allow this iterator
224         * to be used with range-based for loops.
225         * @return The advanced iterator.
```

```
226         */
227         iterator &operator++()
228         {
229             return *this += 1;
230         }
231 
232         /**
233         * @see iterator#operator++()
234         */
235         iterator operator++(int)
236         {
237             iterator cpy { *this };
238             ++*this;
239             return cpy;
240         }
241 
242     protected: // methods
243         /**
244         * @param parent The container in which the elements referred to
245         *        by the iterator are contained.
246         * @param deq_it The element the iterator should refer to.
247         * @param begin Whether the iterator was created using a call
248         *        to parent#begin.
249         */
250         iterator(deque <T> &parent, T *deq_it, bool begin)
251                 : _parent { parent }, _current { deq_it }, _first { begin }
252         {}
253 
254     protected: // members
255 
256         /**
257         * The container in which the elements referred to
258         * by the iterator are contained.
259         */
260         deque <T> &_parent;
261 
262         /**
263         * The element the iterator should refer to.
264         */
265         T *_current;
266 
267         /**
268         * Whether the iterator was created using a call
269         * to parent#begin.
270         *
271         * (This is required for OTE-Iterator behaviour!)
272         */
273         bool _first;
274     };
275 
276 public: // methods
277 
278     /**
279      * Creates an empty deque.
280      */
281     deque();
282 
```

```
283     /**
284      * Creates an emtpy deque with a initial capacity.
285      * @param capacity The initial capacity.
286      */
287     explicit deque(size_type capacity);
288
289     /**
290      * Creates a deque with a default capacity and fills
291      * it up with copies of the supplied value.
292      * @param capacity The initial capacity.
293      * @param value The value to use for initialization.
294      */
295     deque(size_type capacity, const T &value);
296
297     /**
298      * Copy constructor.
299      */
300     deque(const deque &other);
301
302     /**
303      * Move constructor.
304      */
305     deque(deque &&other) noexcept;
306
307     /**
308      * Creates a deque from the supplied initializer list.
309      * @param il A list of elements to initialize the deque with.
310      */
311     deque(std::initializer_list <T> il);
312
313     /**
314      * Copy assignment operator.
315      */
316     deque &operator=(const deque &other);
317
318     /**
319      * Move assignment operator.
320      */
321     deque &operator=(deque &&other) noexcept;
322
323     /**
324      * Overwrites the queue withe the values contained
325      * in the supplied initializer list.
326      */
327     deque &operator=(std::initializer_list <T> il);
328
329     /**
330      * @param pos The position of the element to fetch from the deque.
331      * @return A reference to the element at the specified position.
332      */
333     reference operator[](size_type pos);
334
335     /**
336      * Does the same as {@link deque#operator[]} but does range
337      * checking in addition.
338      */
339     reference at(size_type pos);
```

```
340
341     /**
342      * @return The last element in the deque.
343      */
344     reference back();
345
346     /**
347      * @return The first element in the deque.
348      */
349     reference front();
350
351     /**
352      * @return An iterator referring to the first element in the deque.
353      */
354     iterator begin() noexcept;
355
356     /**
357      * @return An iterator referring to one past the last element (OTE)
358      *         in the deque.
359      */
360     iterator end() noexcept;
361
362     /**
363      * @return True if the deque does not contain any elements.
364      */
365     bool empty() const noexcept;
366
367     /**
368      * @return The number of elements in the deque.
369      */
370     size_type size() const noexcept;
371
372     /**
373      * Removes all elements from the deque.
374      */
375     void clear() noexcept;
376
377     /**
378      * Adds an element to the back of the deque by copying
379      * the supplied value.
380      * @param value The value to add.
381      */
382     void push_back(const T &value);
383
384     /**
385      * Adds an element to the back of the deque by moving
386      * the supplied value.
387      * @param value The value to add.
388      */
389     void push_back(T &&value);
390
391     /**
392      * Remove the last element of the deque.
393      */
394     void pop_back();
395
396     /**
```

```
397      * Adds an element to the front of the deque by copying
398      * the supplied value.
399      * @param value The value to add.
400      */
401     void push_front(const T &value);
402
403     /**
404      * Adds an element to the front of the deque by moving
405      * the supplied value.
406      * @param value The value to add.
407      */
408     void push_front(T &&value);
409
410     /**
411      * Removes the first element of the deque.
412      */
413     void pop_front();
414
415     /**
416      * Increases the deque's capacity to provide enough
417      * room to host the supplied number elements.
418      * @param count
419      */
420     void resize(size_type count);
421
422     /**
423      * Swaps this deque's contents with those of _other_
424      * @param other The deque to swap contents with.
425      */
426     void swap(deque &other) noexcept;
427
428     /**
429      * Destructor.
430      */
431     virtual ~deque();
432
433 private: // methods
434
435     /**
436      * Frees memory associated with the deque.
437      */
438     void _deallocate();
439
440     /**
441      * Makes sure that the supplied pointer is not null and
442      * throws an error including the supplied message otherwise.
443      * @param p The pointer to check
444      * @param message The message to include in the thrown error
445      *        if the pointer is null.
446      */
447     void _assert_not_null(const T *p, const std::string &message);
448
449     /**
450      * @param it A pointer to an element in the deque.
451      * @return A pointer to the next element in the deque.
452      */
453     T *_next(const T *it) const;
```

```
454
455     /**
456      * @param it A pointer to an element in the deque.
457      * @return A pointer to the previous element in the deque.
458      */
459     T *_previous(const T *it) const;
460
461 private: // constants
462     const static size_type _INITIAL_CAPACITY { 5 };
463
464 private: // members
465
466     /**
467      * The allocation method to use for the deque buffer.
468      */
469     std::allocator <T> _alloc {};
470
471     /**
472      * A pointer to the first element in the buffer.
473      */
474     T *_buffer { nullptr };
475
476     /**
477      * The begin of the container. Off-The-End pointer
478      * for empty deques.
479      */
480     T *_begin { nullptr };
481
482     /**
483      * Apointer to the last element in the container.
484      */
485     T *_end { nullptr };
486
487     /**
488      * The current capacity of the deque.
489      */
490     size_type _capacity { 0 };
491
492     void _prepare_push_back();
493
494     void _prepare_push_front();
495 };
496
497 template <typename T>
498 deque <T>::deque() : deque(_INITIAL_CAPACITY)
499 {}
500
501 template <typename T>
502 deque <T>::deque(size_type capacity)
503         : _capacity { capacity }, _buffer { _alloc.allocate(capacity) }
504 {}
505
506 template <typename T>
507 deque <T>::deque(size_type capacity, const T &value)
508         : deque(capacity)
509 {
510     _begin = _buffer;
```

```
511     _end = _begin + capacity - 1;
512     // Fill unconstructed memory pointed to by _buffer/_begin
513     // with copies of _value_
514     std::uninitialized_fill(_buffer, _buffer + capacity, value);
515 }
516
517 template <typename T>
518 deque <T>::deque(const deque &other)
519 {
520     _capacity = other._capacity;
521     _buffer = _alloc.allocate(other._capacity);
522     std::copy(other._buffer, other._buffer + _capacity, _buffer);
523     // make _begin and _end point to the correct elements relative
524     // to the offsets of the other deque's _begin!
525     _begin = _buffer + (other._begin - other._buffer);
526     _end = _buffer + (other._end - other._buffer);
527 }
528
529 template <typename T>
530 deque <T>::deque(deque &&other) noexcept
531 {
532     swap(other);
533 }
534
535 template <typename T>
536 deque <T>::deque(std::initializer_list <T> il)
537         : deque(il.size())
538 {
539     _begin = _buffer;
540     _end = _begin + _capacity - 1;
541     // Copy elements from the initializer list into the unconstructed
542     // memory pointed to by _buffer/_begin
543     std::uninitialized_copy(std::cbegin(il), std::cend(il), _buffer);
544 }
545
546 template <typename T>
547 deque <T>::~deque()
548 {
549     _deallocate();
550 }
551
552 template <typename T>
553 deque <T> &deque <T>::operator=(const deque &other)
554 {
555     if (this != &other) {
556         // first destruct
557         _deallocate();
558         // then copy other deque
559         _capacity = other._capacity;
560         _buffer = _alloc.allocate(other._capacity);
561         std::copy(other._buffer, other._buffer + _capacity, _buffer);
562         _begin = _buffer + (other._begin - other._buffer);
563         _end = _buffer + (other._end - other._buffer);
564     }
565     return *this;
566 }
567
```

```
568 template <typename T>
569 deque <T> &deque <T>::operator=(deque &&other) noexcept
570 {
571     swap(other);
572     return *this;
573 }
574
575 template <typename T>
576 deque <T> &deque <T>::operator=(std::initializer_list <T> il)
577 {
578     *this = deque(std::move(il));
579     return *this;
580 }
581
582 template <typename T>
583 auto deque <T>::operator[](size_type pos) -> reference
584 {
585     return begin()[pos];
586 }
587
588 template <typename T>
589 auto deque <T>::at(size_type pos) -> reference
590 {
591     if (pos >= size() || pos < 0) {
592         throw std::invalid_argument("Index out of bounds.");
593     }
594     return (*this)[pos];
595 }
596
597 template <typename T>
598 auto deque <T>::back() -> reference
599 {
600     T *elem = _end;
601     _assert_not_null(elem, "Can not back() an empty container.");
602     return *elem;
603 }
604
605 template <typename T>
606 auto deque <T>::front() -> reference
607 {
608     T *elem = _begin;
609     _assert_not_null(elem, "Can not front() an empty container.");
610     return *elem;
611 }
612
613 template <typename T>
614 bool deque <T>::empty() const noexcept
615 {
616     return _begin == nullptr;
617 }
618
619 template <typename T>
620 auto deque <T>::size() const noexcept -> size_type
621 {
622     auto it = _begin;
623     // _end points to the last element which must
624     // also be accounted for in the total size.
```

```
625     // Hence for non-empty deques, _size_ starts
626     // at 1!
627     size_type size = it == nullptr ? 0 : 1;
628     while (it != _end) {
629         size += 1;
630         it = _next(it);
631     }
632     return size;
633 }
634
635 template <typename T>
636 void deque <T>::clear() noexcept
637 {
638     auto it = _begin;
639     while (it != _end) {
640         _alloc.destroy(it);
641         it = _next(it);
642     }
643     // Not dealloacting the buffer
644     // so it can be reused
645     _begin = nullptr;
646     _end = nullptr;
647 }
648
649 template <typename T>
650 void deque <T>::push_back(const T &value)
651 {
652     _prepare_push_back();
653     _alloc.construct(_end, value);
654 }
655
656 template <typename T>
657 void deque <T>::push_back(T &&value)
658 {
659     _prepare_push_back();
660     _alloc.construct(_end, std::move(value));
661 }
662
663 template <typename T>
664 void deque <T>::pop_back()
665 {
666     if (!empty()) {
667         _alloc.destroy(_end);
668         if (_begin == _end) {
669             _begin = _end = nullptr;
670         } else {
671             _end = _previous(_end);
672         }
673     }
674 }
675
676 template <typename T>
677 void deque <T>::push_front(const T &value)
678 {
679     if (empty()) {
680         push_back(value);
681     } else {
```

```
682         _prepare_push_front();
683         _alloc.construct(_begin, value);
684     }
685 }
686
687 template <typename T>
688 void deque <T>::push_front(T &&value)
689 {
690     if (empty()) {
691         push_back(std::move(value));
692     } else {
693         _prepare_push_front();
694         _alloc.construct(_begin, std::move(value));
695     }
696 }
697
698 template <typename T>
699 void deque <T>::pop_front()
700 {
701     if (!empty()) {
702         _alloc.destroy(_begin);
703
704         if (_begin == _end) {
705             _end = nullptr;
706             _begin = nullptr;
707         } else {
708             _begin = _next(_begin);
709         }
710     }
711 }
712
713 template <typename T>
714 void deque <T>::_prepare_push_back()
715 {
716     if (empty()) {
717         _begin = _buffer;
718         _end = _buffer;
719     } else {
720         auto it = _next(_end);
721         // if we bump into the begin of the ring buffer
722         if (it == _begin) {
723             // resize dis boi
724             resize(_capacity * 2);
725         }
726         _end = _next(_end);
727     }
728 }
729
730 template <typename T>
731 void deque <T>::_prepare_push_front()
732 {
733     auto it = _previous(_begin);
734     if (it == _end) {
735         resize(_capacity * 2);
736     }
737     _begin = _previous(_begin);
738 }
```

```
739
740 template <typename T>
741 void deque <T>::_deallocate()
742 {
743     if (_buffer != nullptr) {
744
745         // If the buffer still holds fully unconstructed
746         // memory, _alloc.destroy() is an invalid op!
747
748         if (_end != nullptr) {
749             auto buff_it = _begin;
750             while (buff_it != _end) {
751                 _alloc.destroy(buff_it);
752                 buff_it = _next(buff_it);
753             } // TODO test
754             _begin = nullptr;
755             _end = nullptr;
756         }
757
758         _alloc.deallocate(_buffer, _capacity);
759         _buffer = nullptr;
760         _capacity = 0;
761
762     }
763 }
764
765 template <typename T>
766 void deque <T>::_assert_not_null(const T *p, const std::string &message)
767 {
768     if (p == nullptr) {
769         throw std::range_error(message);
770     }
771 }
772
773 template <typename T>
774 T *deque <T>::_next(const T *it) const
775 {
776     // Add 1 to the total difference in elements and keep it
777     // in the range [0; _capacity) using modulo.
778     return _buffer + ((it - _buffer + 1) % _capacity);
779 }
780
781 template <typename T>
782 T *deque <T>::_previous(const T *it) const
783 {
784     // Subtract 1 from the total difference in elements and
785     // keep it in the range [0; _capacity) using modulo.
786     return _buffer + ((it - _buffer - 1 + _capacity) % _capacity);
787 }
788
789 template <typename T>
790 void deque <T>::resize(deque::size_type count)
791 {
792     // allocate a second buffer
793     T *new_buff = _alloc.allocate(count);
794
795     // write old values to new buffer
```

```
796     auto it = _begin;
797     auto new_buff_cpy = new_buff;
798     while (it != _end) {
799         _alloc.construct(new_buff_cpy++, *it);
800         it = _next(it);
801     }
802 
803     // delete the old buffer
804     auto s = size();
805     _deallocate();
806 
807     // reassign members
808     _buffer = new_buff;
809     _begin = _buffer;
810     _end = _begin + s - 1;
811     _capacity = count;
812 }
813 
814 template <typename T>
815 void deque <T>::swap(deque &other) noexcept
816 {
817     using std::swap;
818     swap(_alloc, other._alloc);
819     swap(_buffer, other._buffer);
820     swap(_begin, other._begin);
821     swap(_end, other._end);
822     swap(_capacity, other._capacity);
823 }
824 
825 template <typename T>
826 auto deque <T>::begin() noexcept -> deque <T>::iterator
827 {
828     return deque <T>::iterator(*this, _begin, true);
829 }
830 
831 template <typename T>
832 auto deque <T>::end() noexcept -> deque <T>::iterator
833 {
834     // OTE-Iterator
835     return iterator(*this, _next(_end), false);
836 }
837 
838 template <typename T>
839 bool operator==(deque <T> &lhs, deque <T> &rhs)
840 {
841     bool equal = false;
842     if (lhs.size() == rhs.size()) {
843         equal = true;
844         auto lhs_it = lhs.begin();
845         auto rhs_it = rhs.begin();
846         while (lhs_it != lhs.end() && equal) {
847             equal = *lhs_it++ == *rhs_it++;
848         }
849     }
850     return equal;
851 }
852 
```

```
853 template <typename T>
854 bool operator!=(deque <T> &lhs, deque <T> &rhs)
855 {
856     return !(lhs == rhs);
857 }
858
859 template <typename T>
860 bool operator<(deque <T> &lhs, deque <T> &rhs)
861 {
862     bool less = true;
863     auto lhs_it = lhs.begin();
864     auto rhs_it = rhs.begin();
865     while (lhs_it != lhs.end() && rhs_it != rhs.end() && less) {
866         less = *lhs_it++ < *rhs_it++;
867     }
868     return less;
869 }
870
871 template <typename T>
872 bool operator<=(deque <T> &lhs, deque <T> &rhs)
873 {
874     return lhs < rhs || lhs == rhs;
875 }
876
877 template <typename T>
878 bool operator>(deque <T> &lhs, deque <T> &rhs)
879 {
880     return !(lhs <= rhs);
881 }
882
883 template <typename T>
884 bool operator>=(deque <T> &lhs, deque <T> &rhs)
885 {
886     return !(lhs < rhs);
887 }
888
889 template <typename T>
890 void swap(deque <T> &lhs, deque <T> &rhs)
891 {
892     lhs.swap(rhs);
893 }
894
895 } // namespace swo
```

Listing 2: main.cpp

```
 1 #include <iostream>
 2 #include <cassert>
 3 #include "deque.hpp"
 4
 5
 6 using swo::deque;
 7
 8 int main()
 9 {
10     /*
11      * Constructors
12      * (Checked with debugger for simplcitiy)
```

```
13      */
14
15      // construct default
16      deque <int> di1;
17      // construct from capacity
18      deque <int> di2(100);
19      // construct from capacity and value
20      deque <int> di3(5, -1);
21      // construct from initializer list
22      deque <int> di4 { 1, 2, 3, 4, 5 };
23
24      // copy from lvalue reference
25      deque <int> di5(di4);
26      // copy from rvalue reference
27      deque <int> di6(std::move(di4));
28
29      /*
30       * Operators
31       */
32
33      // =
34      // copy assignment
35      di4 = di3;
36      // move assignment
37      di5 = std::move(di6);
38      // initializer list assignment
39      di6 = { -2, -1, 0, 1, 2 };
40
41      // []
42      assert(di6[0] == -2);
43      assert(di6[4] == 2);
44      assert(di4[2] == -1);
45
46      /*
47       * Methods
48       */
49      // at
50      assert(di6.at(0) == -2);
51      assert(di4.at(3) == -1);
52      bool threw = false;
53      try {
54          di4.at(100);
55      } catch (const std::exception &exc) {
56          threw = true;
57      }
58      assert(threw);
59
60      // front
61      assert(di6.front() == -2);
62      assert(di5.front() == 1);
63
64      // back
65      assert(di6.back() == 2);
66      assert(di5.back() == 5);
67
68      // empty
69      assert(di1.empty());
```

```
70      assert(di2.empty());
71      assert(!di3.empty());
72      assert(!di6.empty());
73
74      // size
75      assert(di1.size() == 0);
76      assert(di2.size() == 0);
77      assert(di3.size() == 5);
78      assert(deque <int>({ 1, 2, 3 }).size() == 3); // all other deques had size 5
        :/
79
80      /*
81       * Void bois
82       */
83      // clear
84      deque <char> ds1 = { 'h', 'e', 'l', 'l', 'o' };
85      ds1.clear();
86      assert(ds1.empty());
87
88      // push_back
89      ds1.push_back('h');
90      ds1.push_back('i');
91      assert(ds1.size() == 2);
92
93      // pop_back
94      ds1.pop_back();
95      assert(ds1.size() == 1);
96      assert(ds1.back() == 'h');
97
98      // push_front
99      ds1.push_front('a');
100     assert(ds1.size() == 2);
101     assert(ds1.front() == 'a');
102
103     // pop_front
104     ds1.pop_front();
105     assert(ds1.front() == 'h');
106
107     // swap
108     swap(di3, di6);
109
110     // comparisons
111     deque <int> cmp1 { 1, 2, 3, 4, 5 };
112     deque <int> cmp2 { 2, 3, 4, 5, 6 };
113     assert(cmp1 < cmp2);
114     assert(cmp1 == cmp1);
115     assert(cmp1 != cmp2);
116     assert(cmp1 >= cmp1);
117     assert(cmp1 >= cmp1);
118
119     /*
120      * Iterators
121      * (Ony rudimentary tests)
122      */
123     auto it = di6.begin();
124     while (it != di6.end()) {
125         std::cout << *it << ' ' << std::flush;
```

```
126         it += 1;
127     }
128     std::cout << std::endl;
129 
130     auto beg = di3.begin();
131     auto mid = di3.begin() + (di3.size() / 2);
132     assert(mid - beg == di3.size() / 2);
133     assert(beg - di3.begin() == 0);
134 
135     for (auto el : di3) {
136         std::cout << el << ' ' << std::flush;
137     }
138     std::cout << std::endl;
139 
140     return 0;
141 }
```

1.3 Tests

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