D:\MAL23-P7-G5-Main\Engine\Source\Utilities\CommonUtilities\include\CommonUtilities\Structures\StaticVector.hpp

```cpp
#pragma once

#include <utility>
#include <cassert>
#include <iterator>
#include <new>
#include <memory>
#include <stdexcept>
#include <limits>
#include <type_traits>

#include <CommonUtilities/Config.h>

namespace CommonUtilities
{
    template<typename T, bool Reverse>
    class StaticIterator
    {
    public:
        using iterator_concept  = std::contiguous_iterator_tag;
        using iterator_category = std::random_access_iterator_tag;
        using difference_type   = std::ptrdiff_t;
        using value_type        = std::remove_const_t<T>;
        using element_type      = T;
        using pointer           = T*;
        using reference         = T&;

        static constexpr int DIR = Reverse ? -1 : 1;

        StaticIterator() noexcept = default;
        explicit StaticIterator(pointer aPtr) noexcept : myPtr(aPtr) {}

        template<bool OtherReverse> // able to convert non-const to const, but not the other way around
        StaticIterator(const StaticIterator<value_type, OtherReverse>& aRight) noexcept : myPtr(aRight.myPtr) {};
```

```
35
36          template<bool OtherReverse>
37          StaticIterator& operator=(const StaticIterator<value_type, OtherReverse>& aRight) noexcept { myPtr = aRight.myPtr; };
38
39          reference operator*() const noexcept { return *myPtr; }
40          pointer operator->() const noexcept { return myPtr; }
41
42          StaticIterator& operator++() noexcept { myPtr += DIR; return *this; }
43          StaticIterator& operator--() noexcept { myPtr -= DIR; return *this; }
44
45          StaticIterator operator++(int) noexcept { StaticIterator temp = *this; ++(*this); return temp; }
46          StaticIterator operator--(int) noexcept { StaticIterator temp = *this; --(*this); return temp; }
47
48          StaticIterator& operator+=(difference_type aOffset) noexcept { myPtr += aOffset; return *this; }
49          StaticIterator& operator-=(difference_type aOffset) noexcept { return *this += -aOffset; }
50
51          NODISC difference_type operator-(const StaticIterator& aRight) const noexcept { return (myPtr - aRight.myPtr) * DIR; }
52
53          NODISC StaticIterator operator+(difference_type aOffset) const noexcept { return (StaticIterator(*this) += aOffset * DIR); }
54          NODISC StaticIterator operator-(difference_type aOffset) const noexcept { return (StaticIterator(*this) += -aOffset * DIR); }
55
56          reference operator[](difference_type aOffset) const noexcept { return *(*this + aOffset * DIR); }
57
58          template<class U, bool OtherReverse>
59          NODISC bool operator==(const StaticIterator<U, OtherReverse>& aRight) const noexcept { return myPtr == aRight.myPtr; }
60
61          template<class U, bool OtherReverse>
62          NODISC std::strong_ordering operator<=>(const StaticIterator<U, OtherReverse>& aRight) const noexcept
63          {
64              if constexpr (Reverse)
65              {
66                  return 0 <=> (myPtr <=> aRight.myPtr);
67              }
68              else
69              {
70                  return (myPtr <=> aRight.myPtr);
71              }
```

```
72          }
73
74          NODISC friend StaticIterator operator+(difference_type aLeft, const StaticIterator& aRight)
75          {
76              return (aRight + aLeft);
77          }
78
79      private:
80          pointer myPtr;
81
82          template<class U, bool Reverse>
83          friend class StaticIterator;
84      };
85
86      template<typename T, std::size_t Capacity>
87      class StaticVector
88      {
89      public:
90          using value_type        = T;
91          using reference         = T&;
92          using const_reference   = const T&;
93          using pointer           = T*;
94          using const_pointer     = const T*;
95          using size_type         = std::size_t;
96          using difference_type   = std::ptrdiff_t;
97
98          using iterator                  = StaticIterator<value_type, false>;
99          using const_iterator            = StaticIterator<const value_type, false>;
100         using reverse_iterator          = StaticIterator<value_type, true>;
101         using const_reverse_iterator    = StaticIterator<const value_type, true>;
102
103         static constexpr bool CopyConstructableAndCopyAssignable = std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T>;
104
105         static constexpr bool NoThrowCopyConstructableAndCopyAssignable = std::is_nothrow_constructible_v<T> &&
    std::is_nothrow_copy_assignable_v<T>;
106
107         static constexpr bool NoThrowMoveConstructor = std::is_nothrow_move_constructible_v<T> ||
```

```
            ((!std::is_nothrow_move_constructible_v<T> || !std::is_move_constructible_v<T>) && std::is_nothrow_copy_cons-
tructible_v<T>);

        static constexpr bool MoveConstructableAndMoveAssignable = std::is_move_constructible_v<T> && std::is_move_assignable_v<T>;

        static constexpr bool NoThrowMoveConstructableAndMoveAssignable = std::is_nothrow_move_constructible_v<T> &&
std::is_nothrow_move_constructible_v<T>;

        static constexpr bool NoThrowMoveAssignment = (std::is_nothrow_destructible_v<T> && NoThrowMoveConstructableAndMoveAssignable)
||
            ((!NoThrowMoveConstructableAndMoveAssignable || !MoveConstructableAndMoveAssignable) && NoThrowCopyConstruct-
ableAndCopyAssignable);

        constexpr StaticVector() = default;
        constexpr ~StaticVector() noexcept(std::is_nothrow_destructible_v<T>);

        constexpr StaticVector(std::size_t aSize) requires(std::is_default_constructible_v<T>);
        constexpr StaticVector(std::size_t aSize, const_reference aElement) requires(std::is_copy_constructible_v<T>);
        template<typename Iter> requires(std::forward_iterator<Iter> && std::constructible_from<T, typename Iter::value_type>)
        constexpr StaticVector(Iter aFirst, Iter aLast);
        constexpr StaticVector(std::initializer_list<T> aInitList);
        template<typename U> requires(std::constructible_from<T, U> && !std::is_same_v<T, U>)
        constexpr StaticVector(std::initializer_list<U> aInitList);

        constexpr StaticVector(const StaticVector& aOther) noexcept
            requires(std::is_trivially_constructible_v<T> && std::is_copy_constructible_v<T>);
        constexpr StaticVector(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T>)
            requires(!std::is_trivially_constructible_v<T> && std::is_copy_constructible_v<T>);

        template<std::size_t OtherCapacity> requires(std::is_copy_constructible_v<T> && (Capacity != OtherCapacity))
        constexpr StaticVector(const StaticVector<T, OtherCapacity>& aOther) noexcept(std::is_nothrow_copy_constructible_v<T> &&
(OtherCapacity < Capacity));

        constexpr StaticVector(StaticVector&& aOther) noexcept
            requires(std::is_trivially_move_constructible_v<T> && std::is_move_constructible_v<T>);
        constexpr StaticVector(StaticVector&& aOther) noexcept(NoThrowMoveConstructor)
```

```
139             requires(!std::is_trivially_move_constructible_v<T> && (std::is_move_constructible_v<T> || std::is_copy_constructibl-
    e_v<T>));
140
141         template<std::size_t OtherCapacity> requires((std::is_move_constructible_v<T> || std::is_copy_constructible_v<T>) && (Capacity
    != OtherCapacity))
142         constexpr StaticVector(StaticVector<T, OtherCapacity>&& aOther) noexcept(NoThrowMoveConstructor && (Capacity >
    OtherCapacity));
143
144         constexpr auto operator=(const StaticVector& aOther) noexcept -> StaticVector&
145             requires(std::is_trivially_copyable_v<T> && std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T>);
146         constexpr auto operator=(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T> &&
    std::is_nothrow_copy_assignable_v<T> && std::is_nothrow_destructible_v<T>) -> StaticVector&
147             requires(!std::is_trivially_copyable_v<T> && std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T>);
148
149         template<std::size_t OtherCapacity> requires(std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T> && (Capacity !=
    OtherCapacity))
150         constexpr auto operator=(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T> &&
    std::is_nothrow_copy_assignable_v<T> && std::is_nothrow_destructible_v<T> && (Capacity > OtherCapacity)) -> StaticVector&;
151
152         constexpr auto operator=(StaticVector&& aOther) noexcept -> StaticVector&
153             requires(std::is_trivially_move_assignable_v<T> && std::is_move_assignable_v<T> && std::is_trivially_move_co-
    nstructible_v<T>);
154         constexpr auto operator=(StaticVector&& aOther) noexcept(NoThrowMoveAssignment) -> StaticVector&
155             requires(!std::is_trivially_move_assignable_v<T> && (CopyConstructableAndCopyAssignable || MoveConstructableAnd-
    MoveAssignable));
156
157         template<std::size_t OtherCapacity> requires(((std::is_copy_constructible_v<T>&& std::is_copy_assignable_v<T>) ||
    (std::is_move_constructible_v<T> && std::is_move_assignable_v<T>)) && (Capacity != OtherCapacity))
158         constexpr auto operator=(StaticVector&& aOther) noexcept(NoThrowMoveAssignment && (Capacity > OtherCapacity)) ->
    StaticVector&;
159
160         template<typename U> requires(std::constructible_from<T, U> && std::is_copy_assignable_v<T> && std::is_copy_constructibl-
    e_v<T>)
161         constexpr void assign(std::initializer_list<U> aInitList);
162         constexpr void assign(std::size_t aCount, const_reference aElement);
163         template<typename Iter> requires(std::forward_iterator<Iter> && std::is_convertible_v<typename Iter::value_type, T>)
164         constexpr void assign(Iter aFirst, Iter aLast);
165
```

```
166         NODISC constexpr auto operator[](size_type aIndex) -> reference;
167         NODISC constexpr auto operator[](size_type aIndex) const -> const_reference;
168 
169         NODISC constexpr auto at(size_type aIndex) -> reference;
170         NODISC constexpr auto at(size_type aIndex) const -> const_reference;
171 
172         NODISC constexpr auto front() -> reference;
173         NODISC constexpr auto front() const -> const_reference;
174 
175         NODISC constexpr auto back() -> reference;
176         NODISC constexpr auto back() const -> const_reference;
177 
178         NODISC constexpr auto data() noexcept -> pointer;
179         NODISC constexpr auto data() const noexcept -> const_pointer;
180 
181         NODISC constexpr auto size() const noexcept -> size_type;
182         NODISC constexpr bool empty() const noexcept;
183 
184         NODISC constexpr auto capacity() const noexcept -> size_type;
185         NODISC constexpr auto free_space() const noexcept -> size_type;
186 
187         NODISC constexpr auto max_size() const noexcept -> size_type;
188 
189         constexpr void push_back(const T& aElement);
190         constexpr void push_back(T&& aElement);
191 
192         template<typename... Args> requires(std::constructible_from<T, Args...>)
193         constexpr auto emplace_back(Args&&... someArgs) -> reference;
194 
195         constexpr void pop_back();
196 
197         constexpr auto erase(const_iterator aPosition) noexcept(std::is_nothrow_move_assignable_v<T> && std::is_nothrow_destructi-
    ble_v<T>)->iterator;
198         constexpr auto erase(const_iterator aFirst, const_iterator aLast) noexcept(std::is_nothrow_move_assignable_v<T> &&
    std::is_nothrow_destructible_v<T>)->iterator;
199 
200         template<typename... Args> requires(std::constructible_from<T, Args...>)
```

```
201         constexpr auto emplace(const_iterator aPosition, Args&&... someArgs) -> iterator;
202
203         constexpr auto insert(const_iterator aPosition, const_reference aElement) -> iterator;
204         constexpr auto insert(const_iterator aPosition, T&& aElement) -> iterator;
205
206         template<typename Iter> requires(std::forward_iterator<Iter> && std::constructible_from<T, typename Iter::value_type> &&
    std::is_copy_assignable_v<T>)
207         constexpr auto insert(const_iterator aPosition, Iter aFirst, Iter aLast) -> iterator;
208
209         constexpr void resize(std::size_t aNewSize) requires(std::is_default_constructible_v<T>);
210         constexpr void resize(std::size_t aNewSize, const_reference aElement) requires(std::is_copy_constructible_v<T>);
211
212         constexpr void swap(StaticVector& aOther) noexcept(std::is_nothrow_swappable_v<T> && ((!std::is_move_constructible_v<T> &&
    std::is_nothrow_copy_assignable_v<T>) || std::is_nothrow_move_constructible_v<T>))
213             requires(std::is_swappable_v<T> && (std::is_copy_constructible_v<T> || std::is_move_constructible_v<T>));
214
215         constexpr void clear() noexcept(std::is_nothrow_destructible_v<T>);
216
217         NODISC constexpr auto begin() noexcept -> iterator;
218         NODISC constexpr auto end() noexcept -> iterator;
219         NODISC constexpr auto begin() const noexcept -> const_iterator;
220         NODISC constexpr auto end() const noexcept -> const_iterator;
221         NODISC constexpr auto cbegin() const noexcept -> const_iterator;
222         NODISC constexpr auto cend() const noexcept -> const_iterator;
223
224         NODISC constexpr auto rbegin() noexcept -> reverse_iterator;
225         NODISC constexpr auto rend() noexcept -> reverse_iterator;
226         NODISC constexpr auto rbegin() const noexcept -> const_reverse_iterator;
227         NODISC constexpr auto rend() const noexcept -> const_reverse_iterator;
228         NODISC constexpr auto crbegin() const noexcept -> const_reverse_iterator;
229         NODISC constexpr auto crend() const noexcept -> const_reverse_iterator;
230
231     private:
232         NODISC constexpr T* ptr_at(size_type aIndex);
233         NODISC constexpr const T* ptr_at(size_type aIndex) const;
234
235 #ifdef _DEBUG
```

```
236         union
237         {
238             T myDebugData[Capacity];
239 #endif
240             alignas(T) std::byte myData[sizeof(T) * Capacity]{};
241 #ifdef _DEBUG
242         };
243 #endif
244 
245         size_type mySize {0};
246     };
247 
248     template<typename T, std::size_t Capacity>
249     constexpr StaticVector<T, Capacity>::~StaticVector() noexcept(std::is_nothrow_destructible_v<T>)
250     {
251         if constexpr (!std::is_trivially_destructible_v<T>)
252         {
253             std::destroy(begin(), end());
254         }
255     }
256 
257     template<typename T, std::size_t Capacity>
258     constexpr StaticVector<T, Capacity>::StaticVector(std::size_t aSize) requires(std::is_default_constructible_v<T>)
259         : myData(), mySize(aSize)
260     {
261         if (mySize > Capacity)
262         {
263             mySize = 0;
264             throw std::runtime_error("Static vector does not have enough capacity for such size!");
265         }
266 
267         std::uninitialized_value_construct(begin(), end());
268     }
269 
270     template<typename T, std::size_t Capacity>
271     constexpr StaticVector<T, Capacity>::StaticVector(std::size_t aSize, const_reference aElement) requires(std::is_copy_constructibl-
    e_v<T>)
```

```
272         : myData(), mySize(aSize)
273     {
274         if (mySize > Capacity)
275         {
276             mySize = 0;
277             throw std::runtime_error("Static vector does not have enough capacity for such size!");
278         }
279 
280         std::uninitialized_fill(begin(), end(), aElement);
281     }
282 
283     template<typename T, std::size_t Capacity>
284     template<typename Iter> requires(std::forward_iterator<Iter> && std::constructible_from<T, typename Iter::value_type>)
285     constexpr StaticVector<T, Capacity>::StaticVector(Iter aFirst, Iter aLast)
286         : myData(), mySize(std::distance(aFirst, aLast))
287     {
288         if (mySize > Capacity)
289         {
290             mySize = 0;
291             throw std::runtime_error("Static vector does not have enough capacity for such size!");
292         }
293 
294         std::uninitialized_copy(aFirst, aLast, begin());
295     }
296 
297     template<typename T, std::size_t Capacity>
298     constexpr StaticVector<T, Capacity>::StaticVector(std::initializer_list<T> aInitList)
299         : StaticVector(const_iterator(aInitList.begin()), const_iterator(aInitList.end()))
300     {
301 
302     }
303 
304     template<typename T, std::size_t Capacity>
305     template<typename U> requires(std::constructible_from<T, U> && !std::is_same_v<T, U>)
306     constexpr StaticVector<T, Capacity>::StaticVector(std::initializer_list<U> aInitList)
307         : StaticVector(const_iterator(aInitList.begin()), const_iterator(aInitList.end()))
308     {
```

```
309
310     }
311
312     template<typename T, std::size_t Capacity>
313     constexpr StaticVector<T, Capacity>::StaticVector(const StaticVector& aOther) noexcept
314         requires(std::is_trivially_constructible_v<T> && std::is_copy_constructible_v<T>) = default;
315
316     template<typename T, std::size_t Capacity>
317     constexpr StaticVector<T, Capacity>::StaticVector(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T>)
318         requires(!std::is_trivially_constructible_v<T> && std::is_copy_constructible_v<T>)
319         : mySize(aOther.size())
320     {
321         std::uninitialized_copy(aOther.begin(), aOther.end(), begin());
322     }
323
324     template<typename T, std::size_t Capacity>
325     template<std::size_t OtherCapacity> requires(std::is_copy_constructible_v<T> && (Capacity != OtherCapacity))
326     constexpr StaticVector<T, Capacity>::StaticVector(const StaticVector<T, OtherCapacity>& aOther)
    noexcept(std::is_nothrow_copy_constructible_v<T> && (OtherCapacity < Capacity))
327         : mySize(aOther.size())
328     {
329         if constexpr (OtherCapacity > Capacity)
330         {
331             if (aOther.size() > Capacity)
332             {
333                 mySize = 0;
334                 throw std::runtime_error("StaticVector can't store that many elements!");
335             }
336         }
337
338         std::uninitialized_copy(aOther.begin(), aOther.end(), begin());
339     }
340
341     template<typename T, std::size_t Capacity>
342     constexpr StaticVector<T, Capacity>::StaticVector(StaticVector&& aOther) noexcept
343         requires(std::is_trivially_move_constructible_v<T> && std::is_move_constructible_v<T>) = default;
344
```

```
345     template<typename T, std::size_t Capacity>
346     constexpr StaticVector<T, Capacity>::StaticVector(StaticVector&& aOther) noexcept(NoThrowMoveConstructor)
347         requires(!std::is_trivially_move_constructible_v<T> && (std::is_move_constructible_v<T> || std::is_copy_constructible_v<T>))
348         : mySize(aOther.size())
349     {
350         if constexpr ((std::is_nothrow_move_constructible_v<T> || !std::is_copy_constructible_v<T>) && std::is_move_constructibl-
    e_v<T>)
351         {
352             std::uninitialized_move(aOther.begin(), aOther.end(), begin());
353         }
354         else
355         {
356             std::uninitialized_copy(aOther.begin(), aOther.end(), begin());
357         }
358 
359         aOther.clear();
360     }
361 
362     template<typename T, std::size_t Capacity>
363     template<std::size_t OtherCapacity> requires((std::is_move_constructible_v<T> || std::is_copy_constructible_v<T>) && (Capacity !=
    OtherCapacity))
364     constexpr StaticVector<T, Capacity>::StaticVector(StaticVector<T, OtherCapacity>&& aOther) noexcept(NoThrowMoveConstructor &&
    (Capacity > OtherCapacity))
365         : mySize(aOther.size())
366     {
367         if constexpr (OtherCapacity > Capacity)
368         {
369             if (aOther.size() > Capacity)
370             {
371                 mySize = 0;
372                 throw std::runtime_error("StaticVector can't store that many elements!");
373             }
374         }
375 
376         if constexpr ((std::is_nothrow_move_constructible_v<T> || !std::is_copy_constructible_v<T>) && std::is_move_constructibl-
    e_v<T>)
377         {
```

```
            std::uninitialized_move(aOther.begin(), aOther.end(), begin());
        }
        else
        {
            std::uninitialized_copy(aOther.begin(), aOther.end(), begin());
        }

        aOther.clear();
    }

    template<typename T, std::size_t Capacity>
    constexpr auto StaticVector<T, Capacity>::operator=(const StaticVector& aOther) noexcept -> StaticVector&
        requires(std::is_trivially_copyable_v<T> && std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T>) = default;

    template<typename T, std::size_t Capacity>
    constexpr auto StaticVector<T, Capacity>::operator=(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T>
&& std::is_nothrow_copy_assignable_v<T> && std::is_nothrow_destructible_v<T>)->StaticVector&
        requires(!std::is_trivially_copyable_v<T> && std::is_copy_constructible_v<T> && std::is_copy_assignable_v<T>)
    {
        if (this == &aOther)
        {
            return *this;
        }

        if (mySize <= aOther.size())
        {
            std::copy(aOther.begin(), aOther.begin() + mySize, begin());
            std::uninitialized_copy(aOther.begin() + mySize, aOther.end(), end());
        }
        else
        {
            std::copy(aOther.begin(), aOther.end(), begin());
            if constexpr (!std::is_trivially_destructible_v<T>)
            {
                std::destroy(begin() + aOther.size(), end());
            }
        }
```

```
414
415         mySize = aOther.size();
416
417         return *this;
418     }
419
420     template<typename T, std::size_t Capacity>
421     template<std::size_t OtherCapacity> requires(std::is_copy_constructible_v<T>&& std::is_copy_assignable_v<T> && (Capacity !=
    OtherCapacity))
422     constexpr auto StaticVector<T, Capacity>::operator=(const StaticVector& aOther) noexcept(std::is_nothrow_copy_constructible_v<T>
    && std::is_nothrow_copy_assignable_v<T> && std::is_nothrow_destructible_v<T> && (Capacity > OtherCapacity)) -> StaticVector&
423     {
424         if (this == &aOther)
425         {
426             return *this;
427         }
428
429         if constexpr (OtherCapacity > Capacity)
430         {
431             if (aOther.size() > Capacity)
432             {
433                 throw std::runtime_error("StaticVector can't store that many elements!");
434             }
435         }
436
437         if constexpr (std::is_trivially_copyable_v<T>)
438         {
439             std::copy(aOther.begin(), aOther.end(), begin());
440         }
441         else
442         {
443             if (mySize <= aOther.size())
444             {
445                 std::copy(aOther.begin(), aOther.begin() + mySize, begin());
446                 std::uninitialized_copy(aOther.begin() + mySize, aOther.end(), end());
447             }
448             else
```

```
449             {
450                 std::copy(aOther.begin(), aOther.end(), begin());
451                 if constexpr (!std::is_trivially_destructible_v<T>)
452                 {
453                     std::destroy(begin() + aOther.size(), end());
454                 }
455             }
456         }
457 
458         mySize = aOther.size();
459 
460         return *this;
461     }
462 
463     template<typename T, std::size_t Capacity>
464     constexpr auto StaticVector<T, Capacity>::operator=(StaticVector&& aOther) noexcept -> StaticVector&
465         requires(std::is_trivially_move_assignable_v<T> && std::is_move_assignable_v<T> && std::is_trivially_move_constructible_v<T>)
    = default;
466 
467     template<typename T, std::size_t Capacity>
468     constexpr auto StaticVector<T, Capacity>::operator=(StaticVector&& aOther) noexcept(NoThrowMoveAssignment)->StaticVector&
469         requires(!std::is_trivially_move_assignable_v<T> && (CopyConstructableAndCopyAssignable || MoveConstructableAnd-
    MoveAssignable))
470     {
471         if (this == &aOther)
472         {
473             return *this;
474         }
475 
476         if constexpr (!MoveConstructableAndMoveAssignable || !NoThrowMoveConstructableAndMoveAssignable)
477         {
478             *this = aOther;
479             aOther.clear();
480             return *this;
481         }
482         else
483         {
```

```
            if (mySize <= aOther.size())
            {
                std::copy(std::make_move_iterator(aOther.begin()), std::make_move_iterator(aOther.begin() + mySize), begin());
                std::uninitialized_move(aOther.begin() + mySize, aOther.end(), end());
            }
            else
            {
                std::copy(std::make_move_iterator(aOther.begin()), std::make_move_iterator(aOther.end()), begin());
                if constexpr (!std::is_trivially_destructible_v<T>)
                {
                    std::destroy(begin() + aOther.size(), end());
                }
            }
        }

        mySize = aOther.size();
        aOther.clear();

        return *this;
    }

    template<typename T, std::size_t Capacity>
    template<std::size_t OtherCapacity> requires(((std::is_copy_constructible_v<T>&& std::is_copy_assignable_v<T>) ||
(std::is_move_constructible_v<T> && std::is_move_assignable_v<T>)) && (Capacity != OtherCapacity))
    constexpr auto StaticVector<T, Capacity>::operator=(StaticVector&& aOther) noexcept(NoThrowMoveAssignment && (Capacity >
OtherCapacity)) -> StaticVector&
    {
        if (this == &aOther)
        {
            return *this;
        }

        if constexpr (!MoveConstructableAndMoveAssignable || !NoThrowMoveConstructableAndMoveAssignable)
        {
            *this = aOther;
            aOther.clear();
            return *this;
```

```
519         }
520         else
521         {
522             if constexpr (OtherCapacity > Capacity)
523             {
524                 if (aOther.size() > Capacity)
525                 {
526                     throw std::runtime_error("StaticVector can't store that many elements!");
527                 }
528             }
529 
530             if constexpr (std::is_trivially_move_assignable_v<T>)
531             {
532                 std::copy(std::make_move_iterator(aOther.begin()), std::make_move_iterator(aOther.end()), begin());
533             }
534             else
535             {
536                 if (mySize <= aOther.size())
537                 {
538                     std::copy(std::make_move_iterator(aOther.begin()), std::make_move_iterator(aOther.begin() + mySize), begin());
539                     std::uninitialized_move(aOther.begin() + mySize, aOther.end(), end());
540                 }
541                 else
542                 {
543                     std::copy(std::make_move_iterator(aOther.begin()), std::make_move_iterator(aOther.end()), begin());
544                     if constexpr (!std::is_trivially_destructible_v<T>)
545                     {
546                         std::destroy(begin() + aOther.size(), end());
547                     }
548                 }
549             }
550         }
551 
552         mySize = aOther.size();
553         aOther.clear();
554 
555         return *this;
```

```
556     }
557
558     template<typename T, std::size_t Capacity>
559     template<typename U> requires(std::constructible_from<T, U> && std::is_copy_assignable_v<T> && std::is_copy_constructible_v<T>)
560     constexpr void StaticVector<T, Capacity>::assign(std::initializer_list<U> aInitList)
561     {
562         assign(aInitList.begin(), aInitList.end());
563     }
564
565     template<typename T, std::size_t Capacity>
566     constexpr void StaticVector<T, Capacity>::assign(std::size_t aCount, const_reference aElement)
567     {
568         if (aCount > Capacity)
569         {
570             throw std::runtime_error("StaticVector can't store that many elements!");
571         }
572
573         if constexpr (std::is_trivially_copyable_v<T>)
574         {
575             std::fill(begin(), begin() + aCount, aElement);
576         }
577         else
578         {
579             if (mySize <= aCount)
580             {
581                 std::fill(begin(), begin() + mySize, aElement);
582                 std::uninitialized_fill(end(), end() + aCount, aElement);
583             }
584             else
585             {
586                 std::fill(begin(), begin() + aCount, aElement);
587                 if constexpr (!std::is_trivially_destructible_v<T>)
588                 {
589                     std::destroy(begin() + aCount, end());
590                 }
591             }
592         }
```

```
593
594         mySize = aCount;
595     }
596
597     template<typename T, std::size_t Capacity>
598     template<typename Iter> requires(std::forward_iterator<Iter> && std::is_convertible_v<typename Iter::value_type, T>)
599     constexpr void StaticVector<T, Capacity>::assign(Iter aFirst, Iter aLast)
600     {
601         const auto newSize = std::distance(aFirst, aLast);
602
603         if (newSize > Capacity)
604         {
605             throw std::runtime_error("StaticVector can't store that many elements!");
606         }
607
608         if constexpr (std::is_trivially_copyable_v<T>)
609         {
610             std::copy(aFirst, aLast, begin());
611         }
612         else
613         {
614             if (mySize <= newSize)
615             {
616                 std::copy(aFirst, aFirst + mySize, begin());
617                 std::uninitialized_copy(aFirst + mySize, aLast, begin() + mySize);
618             }
619             else
620             {
621                 std::copy(aFirst, aLast, begin());
622                 if constexpr (!std::is_trivially_default_constructible_v<T>)
623                 {
624                     std::destroy(begin() + newSize, end());
625                 }
626             }
627         }
628
629         mySize = newSize;
```

```
630     }
631
632     template<typename T, std::size_t Capacity>
633     constexpr auto StaticVector<T, Capacity>::operator[](size_type aIndex) -> reference
634     {
635         assert(aIndex < mySize && "Index is out of bounds!");
636         return *ptr_at(aIndex);
637     }
638     template<typename T, std::size_t Capacity>
639     constexpr auto StaticVector<T, Capacity>::operator[](size_type aIndex) const -> const_reference
640     {
641         assert(aIndex < mySize && "Index is out of bounds!");
642         return *ptr_at(aIndex);
643     }
644
645     template<typename T, std::size_t Capacity>
646     constexpr auto StaticVector<T, Capacity>::at(size_type aIndex) -> reference
647     {
648         if (aIndex >= mySize)
649         {
650             throw std::out_of_range("Index is out of bounds!");
651         }
652
653         return *ptr_at(aIndex);
654     }
655     template<typename T, std::size_t Capacity>
656     constexpr auto StaticVector<T, Capacity>::at(size_type aIndex) const -> const_reference
657     {
658         if (aIndex >= mySize)
659         {
660             throw std::out_of_range("Index is out of bounds!");
661         }
662
663         return *ptr_at(aIndex);
664     }
665
666     template<typename T, std::size_t Capacity>
```

```
667     constexpr auto StaticVector<T, Capacity>::front() -> reference
668     {
669         return *ptr_at(0);
670     }
671     template<typename T, std::size_t Capacity>
672     constexpr auto StaticVector<T, Capacity>::front() const -> const_reference
673     {
674         return *ptr_at(0);
675     }
676
677     template<typename T, std::size_t Capacity>
678     constexpr auto StaticVector<T, Capacity>::back() -> reference
679     {
680         return *ptr_at(mySize - 1);
681     }
682     template<typename T, std::size_t Capacity>
683     constexpr auto StaticVector<T, Capacity>::back() const -> const_reference
684     {
685         return *ptr_at(mySize - 1);
686     }
687
688     template<typename T, std::size_t Capacity>
689     constexpr auto StaticVector<T, Capacity>::data() noexcept -> pointer
690     {
691         return ptr_at(0);
692     }
693     template<typename T, std::size_t Capacity>
694     constexpr auto StaticVector<T, Capacity>::data() const noexcept -> const_pointer
695     {
696         return ptr_at(0);
697     }
698
699     template<typename T, std::size_t Capacity>
700     constexpr auto StaticVector<T, Capacity>::size() const noexcept -> size_type
701     {
702         return mySize;
703     }
```

```
704     template<typename T, std::size_t Capacity>
705     constexpr bool StaticVector<T, Capacity>::empty() const noexcept
706     {
707         return mySize == 0;
708     }
709 
710     template<typename T, std::size_t Capacity>
711     constexpr auto StaticVector<T, Capacity>::capacity() const noexcept -> size_type
712     {
713         return Capacity;
714     }
715     template<typename T, std::size_t Capacity>
716     constexpr auto StaticVector<T, Capacity>::free_space() const noexcept -> size_type
717     {
718         return Capacity - mySize;
719     }
720 
721     template<typename T, std::size_t Capacity>
722     constexpr auto StaticVector<T, Capacity>::max_size() const noexcept -> size_type
723     {
724         // if you see error here, add #define NOMINMAX before including <cmath>
725         return (std::numeric_limits<size_type>::max)();
726     }
727 
728     template<typename T, std::size_t Capacity>
729     constexpr void StaticVector<T, Capacity>::push_back(const T& aElement)
730     {
731         emplace_back(aElement);
732     }
733     template<typename T, std::size_t Capacity>
734     constexpr void StaticVector<T, Capacity>::push_back(T&& aElement)
735     {
736         emplace_back(std::move(aElement));
737     }
738 
739     template<typename T, std::size_t Capacity>
740     template<typename... Args> requires(std::constructible_from<T, Args...>)
```

```
741     constexpr auto StaticVector<T, Capacity>::emplace_back(Args&&... someArgs) -> reference
742     {
743         if (mySize == Capacity)
744         {
745             throw std::runtime_error("Vector has reached capacity, no more elements are allowed!");
746         }
747 
748         std::construct_at(ptr_at(mySize), std::forward<Args>(someArgs)...);
749         ++mySize;
750 
751         return *end();
752     }
753 
754     template<typename T, std::size_t Capacity>
755     constexpr void StaticVector<T, Capacity>::pop_back()
756     {
757         if (empty())
758         {
759             throw std::runtime_error("Vector is empty, nothing to pop");
760         }
761 
762         if constexpr (!std::is_trivially_destructible_v<T>)
763         {
764             std::destroy_at(ptr_at(mySize - 1));
765         }
766 
767         --mySize;
768     }
769 
770     template<typename T, std::size_t Capacity>
771     constexpr auto StaticVector<T, Capacity>::erase(const_iterator aPosition) noexcept(std::is_nothrow_move_assignable_v<T> &&
std::is_nothrow_destructible_v<T>) -> iterator
772     {
773         const auto eraseIter = begin() + std::distance(cbegin(), aPosition);
774 
775         if (eraseIter == end())
776         {
```

```
777             pop_back();
778         }
779         else
780         {
781             for (auto it = eraseIter; it != (end() - 1); ++it)
782             {
783                 *it = std::move(*(it + 1));
784             }
785 
786             std::destroy_at(std::to_address(end() - 1));
787 
788             --mySize;
789         }
790 
791         return eraseIter;
792     }
793 
794     template<typename T, std::size_t Capacity>
795     constexpr auto StaticVector<T, Capacity>::erase(const_iterator aFirst, const_iterator aLast) noexcept(std::is_nothrow_move_assi-
    gnable_v<T> && std::is_nothrow_destructible_v<T>) -> iterator
796     {
797         const auto eraseIterFirst = begin() + std::distance(cbegin(), aFirst);
798         const auto eraseIterLast = begin() + std::distance(cbegin(), aLast);
799 
800         auto it1 = eraseIterFirst;
801         auto it2 = eraseIterLast;
802 
803         while (it2 != end())
804         {
805             *it1 = std::move(*it2);
806 
807             ++it1;
808             ++it2;
809         }
810 
811         std::destroy(it1, end());
812 
```

```
813         mySize = std::distance(begin(), it1);
814 
815         return eraseIterFirst;
816     }
817 
818     template<typename T, std::size_t Capacity>
819     template<typename... Args> requires(std::constructible_from<T, Args...>)
820     constexpr auto StaticVector<T, Capacity>::emplace(const_iterator aPosition, Args&&... someArgs) -> iterator
821     {
822         if (mySize == Capacity)
823         {
824             throw std::runtime_error("Vector has reached capacity, no more elements are allowed!");
825         }
826 
827         const auto insertIter = begin() + std::distance(cbegin(), aPosition);
828 
829         if (insertIter == end())
830         {
831             emplace_back(std::forward<Args>(someArgs)...);
832         }
833         else
834         {
835             std::construct_at(std::to_address(end()), std::move(*(end() - 1)));
836 
837             std::move_backward(insertIter, end() - 1, end());
838 
839             *insertIter = T{ std::forward<Args>(someArgs)... };
840 
841             ++mySize;
842         }
843 
844         return insertIter;
845     }
846 
847     template<typename T, std::size_t Capacity>
848     constexpr auto StaticVector<T, Capacity>::insert(const_iterator aPosition, const_reference aElement) -> iterator
849     {
```

```
850         return emplace(aPosition, aElement);
851     }
852     template<typename T, std::size_t Capacity>
853     constexpr auto StaticVector<T, Capacity>::insert(const_iterator aPosition, T&& aElement) -> iterator
854     {
855         return emplace(aPosition, std::move(aElement));
856     }
857 
858     template<typename T, std::size_t Capacity>
859     template<typename Iter> requires(std::forward_iterator<Iter> && std::constructible_from<T, typename Iter::value_type> &&
    std::is_copy_assignable_v<T>)
860     constexpr auto StaticVector<T, Capacity>::insert(const_iterator aPosition, Iter aFirst, Iter aLast) -> iterator
861     {
862         const auto count = std::distance(aFirst, aLast);
863         const auto newSize = mySize + count;
864 
865         if (newSize > Capacity)
866         {
867             throw std::runtime_error("StaticVector can't store that many elements!");
868         }
869 
870         const auto insertIter = begin() + std::distance(cbegin(), aPosition);
871 
872         if (insertIter == end())
873         {
874             std::uninitialized_copy(aFirst, aLast, end());
875         }
876         else
877         {
878             std::uninitialized_move(end() - count, end(), end());
879 
880             std::move_backward(insertIter, end() - count, end());
881 
882             std::copy(aFirst, aLast, insertIter);
883         }
884 
885         mySize = newSize;
```

```
886
887         return insertIter;
888     }
889
890     template<typename T, std::size_t Capacity>
891     constexpr void StaticVector<T, Capacity>::resize(std::size_t aNewSize)
892         requires(std::is_default_constructible_v<T>)
893     {
894         if (aNewSize > Capacity)
895         {
896             throw std::runtime_error("StaticVector can't store that many elements!");
897         }
898
899         if (aNewSize > mySize)
900         {
901             std::uninitialized_value_construct(end(), begin() + aNewSize);
902         }
903         else
904         {
905             std::destroy(begin() + aNewSize, end());
906         }
907
908         mySize = aNewSize;
909     }
910     template<typename T, std::size_t Capacity>
911     constexpr void StaticVector<T, Capacity>::resize(std::size_t aNewSize, const_reference aElement)
912         requires(std::is_copy_constructible_v<T>)
913     {
914         if (aNewSize > Capacity)
915         {
916             throw std::runtime_error("StaticVector can't store that many elements!");
917         }
918
919         if (aNewSize > mySize)
920         {
921             std::uninitialized_fill(end(), begin() + aNewSize, aElement);
922         }
```

```
923         else
924         {
925             std::destroy(begin() + aNewSize, end());
926         }
927 
928         mySize = aNewSize;
929     }
930 
931     template<typename T, std::size_t Capacity>
932     constexpr void StaticVector<T, Capacity>::swap(StaticVector& aOther) noexcept(std::is_nothrow_swappable_v<T> &&
((!std::is_move_constructible_v<T> && std::is_nothrow_copy_assignable_v<T>) || std::is_nothrow_move_constructible_v<T>))
933         requires(std::is_swappable_v<T> && (std::is_copy_constructible_v<T> || std::is_move_constructible_v<T>))
934     {
935         if (this == &aOther)
936         {
937             return;
938         }
939 
940         auto leftIt = begin();
941         auto rightIt = aOther.begin();
942 
943         for (; leftIt != end() && rightIt != aOther.end(); ++leftIt, ++rightIt)
944         {
945             std::swap(*leftIt, *rightIt);
946         }
947 
948         if (leftIt != end())
949         {
950             if constexpr (std::is_move_constructible_v<T>)
951             {
952                 std::uninitialized_move(leftIt, end(), aOther.end());
953             }
954             else
955             {
956                 std::uninitialized_copy(leftIt, end(), aOther.end());
957             }
958 
```

```
959             if constexpr (!std::is_trivially_destructible_v<T>)
960             {
961                 std::destroy(leftIt, end());
962             }
963         }
964 
965         if (rightIt != aOther.end())
966         {
967             if constexpr (std::is_move_constructible_v<T>)
968             {
969                 std::uninitialized_move(rightIt, aOther.end(), end());
970             }
971             else
972             {
973                 std::uninitialized_copy(rightIt, aOther.end(), end());
974             }
975 
976             if constexpr (!std::is_trivially_destructible_v<T>)
977             {
978                 std::destroy(rightIt, aOther.end());
979             }
980         }
981 
982         std::swap(mySize, aOther.mySize);
983     }
984 
985     template<typename T, std::size_t Capacity>
986     constexpr void StaticVector<T, Capacity>::clear() noexcept(std::is_nothrow_destructible_v<T>)
987     {
988         if constexpr (std::is_trivially_destructible_v<T>)
989         {
990             mySize = 0;
991         }
992         else
993         {
994             if constexpr (std::is_nothrow_destructible_v<T>)
995             {
```

```
996                 std::destroy(begin(), end());
997                 mySize = 0;
998             }
999             else
1000            {
1001                while (!empty())
1002                {
1003                    pop_back();
1004                }
1005            }
1006        }
1007    }
1008
1009    template<typename T, std::size_t Capacity>
1010    constexpr auto StaticVector<T, Capacity>::begin() noexcept -> iterator
1011    {
1012        return iterator(ptr_at(0));
1013    }
1014    template<typename T, std::size_t Capacity>
1015    constexpr auto StaticVector<T, Capacity>::end() noexcept -> iterator
1016    {
1017        return iterator(ptr_at(mySize));
1018    }
1019
1020    template<typename T, std::size_t Capacity>
1021    constexpr auto StaticVector<T, Capacity>::begin() const noexcept -> const_iterator
1022    {
1023        return const_iterator(ptr_at(0));
1024    }
1025    template<typename T, std::size_t Capacity>
1026    constexpr auto StaticVector<T, Capacity>::end() const noexcept -> const_iterator
1027    {
1028        return const_iterator(ptr_at(mySize));
1029    }
1030
1031    template<typename T, std::size_t Capacity>
1032    constexpr auto StaticVector<T, Capacity>::cbegin() const noexcept -> const_iterator
```

```
1033    {
1034        return begin();
1035    }
1036    template<typename T, std::size_t Capacity>
1037    constexpr auto StaticVector<T, Capacity>::cend() const noexcept -> const_iterator
1038    {
1039        return end();
1040    }
1041
1042    template<typename T, std::size_t Capacity>
1043    constexpr auto StaticVector<T, Capacity>::rbegin() noexcept -> reverse_iterator
1044    {
1045        return reverse_iterator(ptr_at(mySize - 1));
1046    }
1047    template<typename T, std::size_t Capacity>
1048    constexpr auto StaticVector<T, Capacity>::rend() noexcept -> reverse_iterator
1049    {
1050        return reverse_iterator(ptr_at(-1));
1051    }
1052
1053    template<typename T, std::size_t Capacity>
1054    constexpr auto StaticVector<T, Capacity>::rbegin() const noexcept -> const_reverse_iterator
1055    {
1056        return const_reverse_iterator(ptr_at(mySize - 1));
1057    }
1058    template<typename T, std::size_t Capacity>
1059    constexpr auto StaticVector<T, Capacity>::rend() const noexcept -> const_reverse_iterator
1060    {
1061        return const_reverse_iterator(ptr_at(-1));
1062    }
1063
1064    template<typename T, std::size_t Capacity>
1065    constexpr auto StaticVector<T, Capacity>::crbegin() const noexcept -> const_reverse_iterator
1066    {
1067        return rbegin();
1068    }
1069    template<typename T, std::size_t Capacity>
```

```
1070      constexpr auto StaticVector<T, Capacity>::crend() const noexcept -> const_reverse_iterator
1071      {
1072          return rend();
1073      }
1074
1075      template<typename T, std::size_t Capacity>
1076      constexpr T* StaticVector<T, Capacity>::ptr_at(size_type aIndex)
1077      {
1078          return std::launder(reinterpret_cast<T*>(&myData[0] + sizeof(T) * aIndex));
1079      }
1080
1081      template<typename T, std::size_t Capacity>
1082      constexpr const T* StaticVector<T, Capacity>::ptr_at(size_type aIndex) const
1083      {
1084          return std::launder(reinterpret_cast<const T*>(&myData[0] + sizeof(T) * aIndex));
1085      }
1086
1087      // GLOBAL FUNCTIONS
1088
1089      template<typename T, std::size_t Capacity> requires(std::is_swappable_v<T> && (std::is_copy_constructible_v<T> ||
      std::is_move_constructible_v<T>))
1090      constexpr void swap(StaticVector<T, Capacity>& aLeft, StaticVector<T, Capacity>& aRight)
1091      {
1092          aLeft.swap(aRight);
1093      }
1094
1095      template<typename T, std::size_t LeftCapacity, std::size_t RightCapacity> requires(std::equality_comparable<T>)
1096      constexpr bool operator==(const StaticVector<T, LeftCapacity>& aLeft, const StaticVector<T, RightCapacity>& aRight) noexcept
1097      {
1098          return std::equal(aLeft.begin(), aLeft.end(), aRight.begin(), aRight.end());
1099      }
1100  }
```