zip and enumerate iterators

Generated by Doxygen 1.9.8

1 zip and enumerate iterators

1.1 Python-like Zip and Enumerate Iterators

C++-implementation of Python-like zip- and enumerate-iterators which can be used in range-based for loops along with structured bindings to iterate over multiple containers at the same time. Requires C++17.

The library has been tested on the following compilers:

- g++:
  - C++17 support: g++-9 to g++-12
  - C++20 ranges compatibility: g++-10 to g++-12
- clang:
  - C++17 support: clang-9 to clang-17
  - C++20 ranges compatibility: clang-16 and clang-17

1.1.1 Properties

The `zip`-class is a container-wrapper for arbitrary iterable containers. It provides the member functions `begin()` and `end()` enabling it to be used in range-based for loops to iterate over multiple containers at the same time. The `enuerate`-function is a special case of `zip` and uses a "counting container" (similar to `std::ranges::iota`) to provide an index. Additionally, const-versions exist which do not allow the manipulation of the container elements.

1.1.2 Code Examples

The syntax is mostly similar to Python:

```
#include <vector>
#include <list>
#include "Iterators.hpp"

using namespace iterators;
std::list<std::string> strings{"a", "b", "c"};
std::vector<int> numbers{1, 2, 3};
for (auto [string, number] : zip(strings, numbers)) {
    // 'string' and 'number' are references to the container element
    string += std::to_string(number);
}

// now 'strings' contains {"a1", "b2", "c3"}
```

The for loop uses so called `ZipIterator`s which point to tuples which in turn contain references to the container elements. Therefore, no copying occurs and manipulation of the container elements is possible. Observe that the structured binding captures by value (since the values are themselves references).

If you want to prohibit manipulation, you can use `const_zip`

```
using namespace iterators;
```

```
std::list<std::string> strings{"a", "b", "c"};
std::vector<int> numbers{1, 2, 3};
for (auto [string, number] : const_zip(strings, numbers)) {
    // string += std::to_string(number);  error, string is readonly!
    std::cout « string « " " « number « std::endl;
}
```

Additionally, you can use `zip_i` to manually zip iterators or pointers:

```
using namespace iterators;
std::list<std::string> strings{"a", "b", "c"};
std::vector<int> numbers{1, 2, 3};
auto zipBegin = zip_i(strings.begin(), numbers.begin());
auto zipEnd = zip_i(strings.end(), numbers.end());
while (zipBegin != zipEnd) {
    auto [s, num] = *zipBegin;
    // ...
    ++zipBegin;
}
```

`ZipIterators` support the same operations as the least powerful underlying iterator. For example, if you zip a random access iterator (e.g. from `std::vector`) and a bidirectional iterator (e.g. from `std::list`), then the resulting `ZipIterator` will only support bidirectional iteration but no random access.

As in Python, the shortest range decides the overall range:

```
using namespace iterators;
std::list<std::string> strings{"a", "b", "c"};
std::vector<int> numbers{1, 2, 3, 4, 5, 6};
for (auto [string, number] : zip(strings, numbers)) {
    std::cout « string « " " « number « " | "
}
// prints a 1 | b 2 | c 3 |
```

The `enumerate`-function works similarly.

```
using namespace iterators;
std::list<std::string> strings{"a", "b", "c"};
for (auto [index, string] : enumerate(strings)) {
    string += std::to_string(index);
}
// now 'strings' contains {"a0", "b1", "c2"}
```

Also, an optional offset can be specified:

```
for (auto [index, string] : enumerate(strings, 4)) { // index starts from 4
    ...
}
```

And as with `zip`, a const version (`const_enumerate`) exists.

In case temporary containers are used, `zip` and `enumerate` will take ownership of the containers to guarantee well-defined memory access.

```
for (auto [index, number] : enumerate(std::array{53, 21, 17})) {
    // enumerate takes ownership of the array. The elements
    // can safely be accessed and manipulated
}
```

1.1.3 Doxygen Documentation

- HTML
- PDF

2 Namespace Documentation

2.1 iterators Namespace Reference

namespace containing zip and enumerate functions

Namespaces

- namespace impl

  namespace containing structures and helpers used to implement zip and enumerate. Normally there is no need to use any of its members directly

Functions

- template<typename ... Iterators>
  constexpr auto zip_i (Iterators ...iterators) -> impl::ZipIterator< std::tuple< Iterators... > >
- template<typename ... Iterable>
  constexpr auto zip (Iterable &&...iterable)
- template<typename ... Iterable>
  constexpr auto const_zip (Iterable &&...iterable)
- template<typename Container , typename T = std::size_t>
  constexpr auto enumerate (Container &&container, T start=T(0), T increment=T(1))
- template<typename Container , typename T = std::size_t>
  constexpr auto const_enumerate (Container &&container, T start=T(0), T increment=T(1))
- template<typename ... Iterable>
  constexpr auto zip_enumerate (Iterable &&...iterable)
- template<typename ... Iterable>
  constexpr auto const_zip_enumerate (Iterable &&...iterable)

2.1.1 Detailed Description

namespace containing zip and enumerate functions

2.1.2 Function Documentation

const_enumerate()

```
template<typename Container , typename T = std::size_t>
constexpr auto iterators::const_enumerate (
            Container && container,
            T start = T(0),
            T increment = T(1) )  [constexpr]
```

enumerate variant that does not allow manipulation of the container elements

Function that can be used in range based loops to emulate the enumerate iterator from python.

Template Parameters

Container	Container type that supports iteration
T	type of enumerate counter (default std::size_t)

Parameters

container	Source container

Parameters

start	Optional index offset (default 0)
increment	Optional index increment (default 1)

Returns

impl::ZipView that provides begin and end members to be used in range based for-loops.

const_zip()

```
template<typename ...  Iterable>
constexpr auto iterators::const_zip (
            Iterable &&...  iterable )  [constexpr]
```

Zip variant that does not allow manipulation of the container elements

Function that can be used in range based loops to emulate the zip iterator from python. As in python: if the passed containers have different lengths, the container with the least items decides the overall range

Template Parameters

Iterable	Container types that support iteration

Parameters

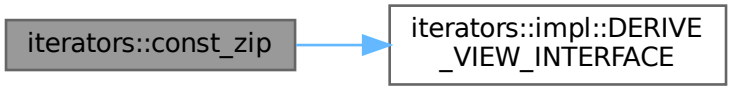
iterable	Arbitrary number of containers

Returns

impl::ZipView class that provides begin and end members to be used in range based for-loops

References iterators::impl::DERIVE_VIEW_INTERFACE().

Here is the call graph for this function:



const_zip_enumerate()

```
template<typename ...  Iterable>
constexpr auto iterators::const_zip_enumerate (
             Iterable &&...  iterable )  [constexpr]
```

zip_enumerate variant that does not allow manipulation of the container elements

combination of zip and enumerate, i.e. returns an impl::ZipView that contains an enumerator at the first position

Template Parameters

Iterable	Types of arguments

Parameters

iterable	arbitrary number of iterables followed by optionally a start and an increment

Returns

impl::ZipView with prepended enumerator

enumerate()

```
template<typename Container , typename T = std::size_t>
constexpr auto iterators::enumerate (
             Container && container,
             T start = T(0),
             T increment = T(1) )  [constexpr]
```

Function that can be used in range based loops to emulate the enumerate iterator from python.

Template Parameters

Container	Container type that supports iteration
T	type of enumerate counter (default std::size_t)

Parameters

container	Source container
start	Optional index offset (default 0)
increment	Optional index increment (default 1)

Returns

impl::ZipView that provides begin and end members to be used in range based for-loops.

zip()

```
template<typename ...  Iterable>
constexpr auto iterators::zip (
            Iterable &&...  iterable )  [constexpr]
```

Function that can be used in range based loops to emulate the zip iterator from python. As in python: if the passed containers have different lengths, the container with the least items decides the overall range

Template Parameters

Iterable	Container types that support iteration

Parameters

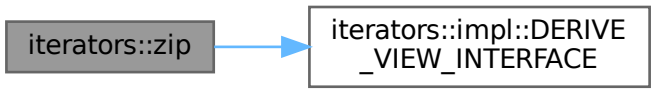
iterable	Arbitrary number of containers

Returns

impl::ZipView class that provides begin and end members to be used in range based for-loops

References iterators::impl::DERIVE_VIEW_INTERFACE().

Here is the call graph for this function:



zip_enumerate()

```
template<typename ...  Iterable>
constexpr auto iterators::zip_enumerate (
            Iterable &&...  iterable )  [constexpr]
```

combination of zip and enumerate, i.e. returns an impl::ZipView that contains an enumerator at the first position

Template Parameters

Iterable	Types of arguments

Parameters

iterable	arbitrary number of iterables followed by optionally a start and an increment

Returns

impl::ZipView with prepended enumerator

zip_i()

```
template<typename ...  Iterators>
constexpr auto iterators::zip_i (
            Iterators ...  iterators ) -> impl::ZipIterator<std::tuple<Iterators...>>   [constexpr]
```

Function that is used to create a impl::ZipIterator from an arbitrary number of iterators

Template Parameters

Iterators	type of iterators

Parameters

iterators	arbitrary number of iterators

Returns

impl::ZipIterator

Note

ZipIterators have the same iterator category as the least powerful underlying operator. This means that for example, zipping a random access iterator and a bidirectional iterator only yields a bidirectional impl::ZipIterator

2.2 iterators::impl Namespace Reference

namespace containing structures and helpers used to implement zip and enumerate. Normally there is no need to use any of its members directly

Namespaces

- namespace traits

  namespace containing type traits used in implementation of zip and enumerate

Classes

- struct CounterIterator

  Iterator of an infinite sequence of numbers. Simply increments an internal counter.
- struct CounterRange

  Represents an infinite range of numbers.
- struct RefTuple

  Proxy reference type that supports assignment and swap even on const instances.
- struct SynthesizedOperators

  CRTP-class that provides additional pointer arithmetic operators synthesized from basic operators.
- struct Unreachable

  represents the unreachable end of an infinite sequence
- class ZipIterator

  Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.

Functions

- template<std::size_t Offset, std::size_t ... Idx>
  constexpr auto **index_seq_impl** (std::index_sequence< Idx... >) noexcept
- template<std::size_t Start, std::size_t End>
  constexpr auto **index_seq** () noexcept
- template<typename Tuple , std::size_t ... Idx1, std::size_t ... Idx2>
  constexpr auto **tuple_split_impl** (Tuple &&tuple, std::index_sequence< Idx1... >, std::index_sequence< Idx2... >)
- template<std::size_t Idx, typename Tuple >
  constexpr auto **tuple_split** (Tuple &&tuple)
- template<template< typename > typename Predicate, std::size_t Idx, typename ... Ts>
  constexpr std::size_t **index_of** () noexcept
- template<typename Tuple >
  constexpr auto **swap** (Tuple &&a, Tuple &&b) -> std::enable_if_t< iterators::impl::traits::is_same_template_v< iterators::impl::RefTuple, Tuple > >
- template<typename ... Iterable>
  struct ZipView DERIVE_VIEW_INTERFACE (ZipView< Iterable... >)

  Zip-view that provides begin() and end() member functions. Use to loop over multiple ranges at the same time using ranged based for-loops.
- template<typename T >
  constexpr T sgn (T val) noexcept

2.2.1 Detailed Description

namespace containing structures and helpers used to implement zip and enumerate. Normally there is no need to use any of its members directly

2.2.2 Function Documentation

DERIVE_VIEW_INTERFACE()

```
template<typename ...  Iterable>
struct ZipView iterators::impl::DERIVE_VIEW_INTERFACE (
            ZipView< Iterable...  >  )
```

Zip-view that provides begin() and end() member functions. Use to loop over multiple ranges at the same time using ranged based for-loops.

Ranges are captured by lvalue reference, no copying occurs. Temporaries are allowed as well in which case storage is moved into the zip-view.

Template Parameters

Iterable	Underlying range types

CTor. Binds reference to ranges or takes ownership in case of rvalue references

Template Parameters

Container	range types

Parameters

containers	arbitrary number of ranges

Returns a ZipIterator to the first elements of the underlying ranges

Returns

ZipIterator created by invoking std::begin on all underlying ranges

Returns a ZipIterator to the elements following the last elements of the the underlying ranges

Returns

ZipIterator created by invoking std::end on all underlying ranges

Note

returns a ZipIterator that does not allow changing the ranges' elements

Array subscript operator (no bounds are checked)

Template Parameters

IsRandomAccess	SFINAE helper, do not specify explicitly

Parameters

index	index

Returns

zip view element at given index

Returns the smallest size of all containers. Only available if all containers know their size

Template Parameters

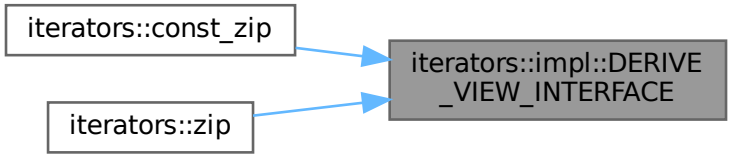
HasSize	SFINAE guard, do not specify explicitly

Returns

smallest size of all containers

Referenced by iterators::const_zip(), and iterators::zip().

Here is the caller graph for this function:



sgn()

```
template<typename T >
constexpr T iterators::impl::sgn (
            T val )  [constexpr], [noexcept]
```

Signum function

Template Parameters

T	arbitrary scalar type

Parameters

val	function argument

Returns

+1 if val >= 0, -1 else

2.3 iterators::impl::traits Namespace Reference

namespace containing type traits used in implementation of zip and enumerate

2.3.1 Detailed Description

namespace containing type traits used in implementation of zip and enumerate

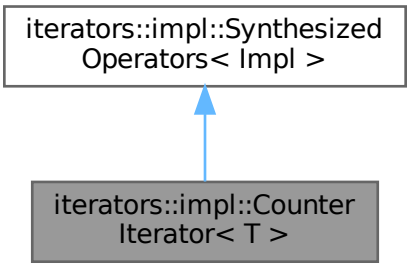
3 Class Documentation

3.1 iterators::impl::CounterIterator< T > Struct Template Reference

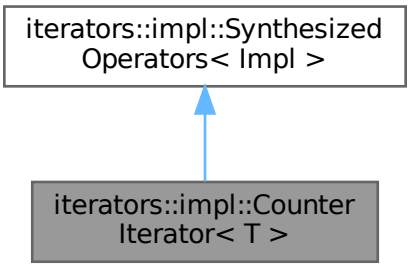
Iterator of an infinite sequence of numbers. Simply increments an internal counter.

`#include <Iterators.hpp>`

Inheritance diagram for iterators::impl::CounterIterator< T >:



Collaboration diagram for iterators::impl::CounterIterator< T >:



Public Types

- using **value_type** = T
- using **reference** = T
- using **pointer** = void
- using **iterator_category** = std::random_access_iterator_tag
- using **difference_type** = std::ptrdiff_t

Public Member Functions

- constexpr CounterIterator (T begin, T increment=T(1)) noexcept
- constexpr CounterIterator & operator++ () noexcept
- constexpr CounterIterator & operator-- () noexcept
- constexpr CounterIterator & operator+= (difference_type n) noexcept
- constexpr CounterIterator & operator-= (difference_type n) noexcept
- constexpr difference_type operator- (const CounterIterator &other) const noexcept
- constexpr bool operator== (const CounterIterator &other) const noexcept
- constexpr bool operator< (const CounterIterator &other) const noexcept
- constexpr bool operator> (const CounterIterator &other) const noexcept
- constexpr T operator* () const noexcept
- template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference_type))) > constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) const noexcept(noexcept(*(std::declval< Impl >()+n)))
- template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference_type))) > constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) noexcept(noexcept(*(std::declval< Impl >()+n)))
- template<REQUIRES_IMPL(Impl,++INSTANCE_OF_IMPL) > constexpr Impl operator++ (int) noexcept(noexcept(++std::declval< Impl >()) &&std::is_nothrow_copy_constructible_v< Impl >)
- template<REQUIRES_IMPL(Impl, --INSTANCE_OF_IMPL) > constexpr Impl operator-- (int) noexcept(noexcept(--std::declval< Impl >()) &&std::is_nothrow_copy_constructible_v< Impl >)
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL==INSTANCE_OF(T)) > constexpr bool operator!= (const T &other) const noexcept(noexcept(INSTANCE_OF_IMPL==other))
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL > INSTANCE_OF(T)) > constexpr bool operator<= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL > rhs))
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL< INSTANCE_OF(T)) > constexpr bool operator>= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL< rhs))

Friends

- constexpr bool operator== (const CounterIterator &, Unreachable) noexcept
- constexpr bool operator== (Unreachable, const CounterIterator &) noexcept
- constexpr bool **operator!=** (Unreachable, const CounterIterator &) noexcept

3.1.1 Detailed Description

template<typename T = std::size_t>
struct iterators::impl::CounterIterator< T >

Iterator of an infinite sequence of numbers. Simply increments an internal counter.

Template Parameters

Type	of the counter (most of the time this is `std::size_t`)

3.1.2 Constructor & Destructor Documentation

CounterIterator()

```
template<typename T = std::size_t>
constexpr iterators::impl::CounterIterator< T >::CounterIterator (
            T begin,
            T increment = T(1) )  [inline], [explicit], [constexpr], [noexcept]
```

CTor.

Parameters

begin	start of number sequence
increment	step size (default is 1)

Note

Depending on the template type T, increment can also be negative.

3.1.3 Member Function Documentation

operator"!=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL==INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator!= (
            const T & other ) const  [inline], [constexpr], [noexcept], [inherited]
```

Inequality comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not equal to other

operator∗()

```
template<typename T = std::size_t>
constexpr T iterators::impl::CounterIterator< T >::operator* ( ) const  [inline], [constexpr],
[noexcept]
```

Produces the counter value

Returns

value of internal counter

operator++() [1/2]

```
template<typename T = std::size_t>
constexpr CounterIterator & iterators::impl::CounterIterator< T >::operator++ ( )  [inline],
[constexpr], [noexcept]
```

Increments value by increment

Returns

reference to this

operator++() [2/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl,++INSTANCE_OF_IMPL) >
constexpr Impl iterators::impl::SynthesizedOperators< Impl >::operator++ (
            int  )  [inline], [constexpr], [noexcept], [inherited]
```

Postfix increment. Synthesized from prefix increment

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator+=()

```
template<typename T = std::size_t>
constexpr CounterIterator & iterators::impl::CounterIterator< T >::operator+= (
            difference_type n )  [inline], [constexpr], [noexcept]
```

Compound assignment increment. Increments value by n times increment

Parameters

n	number of steps

Returns

reference to this

operator-()

```
template<typename T = std::size_t>
constexpr difference_type iterators::impl::CounterIterator< T >::operator- (
            const CounterIterator< T > & other ) const  [inline], [constexpr], [noexcept]
```

Difference between two CounterIterators

Parameters

other	right hand side

Returns

integer `n` with the smallest possible absolute value such that `other + n <= *this`

Note

When other has the same increment as `*this`, then the returned value is guaranteed to fulfil `other + n == *this`. In the following example, this is not the case:

```
CounterIterator a(8, 1);
CounterIterator b(4, 3);
auto diff = a - b; // yields 1 since b + 1 <= a
```

operator--() [1/2]

```
template<typename T = std::size_t>
constexpr CounterIterator & iterators::impl::CounterIterator< T >::operator-- ( )  [inline],
[constexpr], [noexcept]
```

Decrements value by increment

Returns

reference to this

operator--() [2/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, --INSTANCE_OF_IMPL) >
constexpr Impl iterators::impl::SynthesizedOperators< Impl >::operator-- (
            int  )  [inline], [constexpr], [noexcept], [inherited]
```

Postfix decrement. Synthesized from prefix decrement

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator-=()

```
template<typename T = std::size_t>
constexpr CounterIterator & iterators::impl::CounterIterator< T >::operator-= (
            difference_type n )  [inline], [constexpr], [noexcept]
```

Compound assignment decrement. Increments value by n times increment

Parameters

n	number of steps

Returns

reference to this

operator<()

```
template<typename T = std::size_t>
constexpr bool iterators::impl::CounterIterator< T >::operator< (
            const CounterIterator< T > & other ) const  [inline], [constexpr], [noexcept]
```

Less comparison of internal counters with respect to increment of this instance

Parameters

other	right hand side

Returns

true if
`sgn(increment) **this < *other sgn(increment)`
where `sgn` is the signum function

Note

If increment is negative then both sides of the inequality are multiplied with -1. For example: let `it1 = 5` and `it2 = -2` be two CounterIterators where `it1` has negative increment. Then `it1 < it2` is true.

operator<=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL > INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator<= (
            const T & rhs ) const  [inline], [constexpr], [noexcept], [inherited]
```

Less than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not greater than other

operator==()

```
template<typename T = std::size_t>
constexpr bool iterators::impl::CounterIterator< T >::operator== (
            const CounterIterator< T > & other ) const  [inline], [constexpr], [noexcept]
```

Equality comparison.

Parameters

other	right hand side

Returns

true if counter of left and right hand side are equal

operator>()

```
template<typename T = std::size_t>
constexpr bool iterators::impl::CounterIterator< T >::operator> (
            const CounterIterator< T > & other ) const  [inline], [constexpr], [noexcept]
```

Greater comparison of internal counters with respect to increment of this instance

Parameters

other	right hand side

Returns

true if
`sgn(increment) **this > *other sgn(increment)`

where `sgn` is the signum

Note

If increment is negative then both sides of the inequality are multiplied with -1. For example: let `it1 = 5` and `it2 = -2` be two CounterIterators where `it1` has negative increment. Then `it1 > it2` is false.

operator>=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL< INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator>= (
            const T & rhs ) const  [inline], [constexpr], [noexcept], [inherited]
```

Greater than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not less than other

operator[]() [1/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference↩
_type))) >
constexpr decltype(auto) iterators::impl::SynthesizedOperators< Impl >::operator[] (
            typename Implementation::difference_type n ) const  [inline], [constexpr], [noexcept],
[inherited]
```

Array subscript operator

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

n	index

Returns

*(*this + n)

operator[]() [2/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference↩
_type))) >
constexpr decltype(auto) iterators::impl::SynthesizedOperators< Impl >::operator[] (
            typename Implementation::difference_type n )  [inline], [constexpr], [noexcept],
[inherited]
```

3.1.4 Friends And Related Symbol Documentation

operator== [1/2]

```
template<typename T = std::size_t>
constexpr bool operator== (
            const CounterIterator< T > & ,
            Unreachable  )  [friend]
```

Equality comparison with Unreachable sentinel

Returns

false

operator== [2/2]

```
template<typename T = std::size_t>
constexpr bool operator== (
            Unreachable ,
            const CounterIterator< T > &  )  [friend]
```

Equality comparison with Unreachable sentinel

Returns

false

The documentation for this struct was generated from the following file:

- Iterators.hpp

3.2 iterators::impl::CounterRange< T > Struct Template Reference

Represents an infinite range of numbers.

```
#include <Iterators.hpp>
```

Public Member Functions

- constexpr CounterRange (T start=T(0), T increment=T(1)) noexcept
- constexpr CounterIterator< T > begin () const noexcept

Static Public Member Functions

- static constexpr Unreachable end () noexcept

3.2.1 Detailed Description

template<typename T = std::size_t>
struct iterators::impl::CounterRange< T >

Represents an infinite range of numbers.

Template Parameters

T	type of number range

3.2.2 Constructor & Destructor Documentation

CounterRange()

```
template<typename T = std::size_t>
constexpr iterators::impl::CounterRange< T >::CounterRange (
            T start = T(0),
            T increment = T(1) )  [inline], [explicit], [constexpr], [noexcept]
```

CTor

Parameters

start	start of the range
increment	step size

Note

Depending on the template type T, increment can also be negative.

3.2.3 Member Function Documentation

begin()

```
template<typename T = std::size_t>
constexpr CounterIterator< T > iterators::impl::CounterRange< T >::begin ( ) const  [inline],
[constexpr], [noexcept]
```

Returns

CounterIterator representing the beginning of the sequence

end()

```
template<typename T = std::size_t>
static constexpr Unreachable iterators::impl::CounterRange< T >::end ( )  [inline], [static],
[constexpr], [noexcept]
```

Returns

Sentinel object representing the unreachable end of the sequence

The documentation for this struct was generated from the following file:

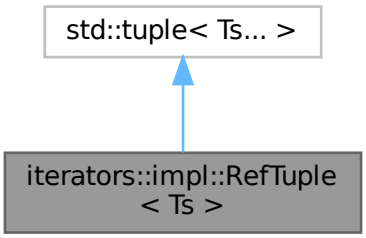
- Iterators.hpp

3.3 iterators::impl::RefTuple< Ts > Struct Template Reference

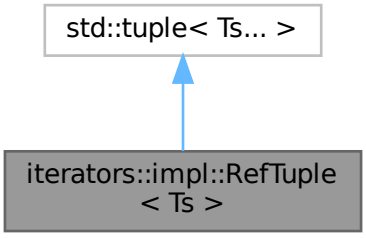
Proxy reference type that supports assignment and swap even on const instances.

`#include <Iterators.hpp>`

Inheritance diagram for iterators::impl::RefTuple< Ts >:



Collaboration diagram for iterators::impl::RefTuple< Ts >:



Public Member Functions

- template<typename Dummy = int>
  constexpr auto **operator=** (RefTuple &&other) noexcept((std::is_nothrow_move_assignable_v< std::remove_reference_t< Ts > > &&...)) -> std::enable_if_t< std::is_same_v< Dummy, int > and Assignable, RefTuple & >
- template<typename ... Us>
  decltype(auto) **operator=** (const std::tuple< Us... > &other) const
- template<typename ... Us>
  decltype(auto) **operator=** (std::tuple< Us... > &&other) const
- template<std::size_t Idx>
  constexpr decltype(auto) **get** () const &noexcept
- template<std::size_t Idx>
  constexpr decltype(auto) **get** () &noexcept
- template<std::size_t Idx>
  constexpr decltype(auto) **get** () const &&noexcept
- template<std::size_t Idx>
  constexpr decltype(auto) **get** () &&noexcept
- template<typename Tuple >
  constexpr auto **swap** (Tuple &&other) const -> std::enable_if_t< std::is_same_v< Tuple, RefTuple > and Assignable >

Static Public Attributes

- static constexpr bool **Assignable** = (not std::is_const_v<std::remove_reference_t<Ts>> && ...)

3.3.1 Detailed Description

template<typename ... Ts>
struct iterators::impl::RefTuple< Ts >

Proxy reference type that supports assignment and swap even on const instances.

Actual constness is determined by element constness

Template Parameters

Ts	

The documentation for this struct was generated from the following file:

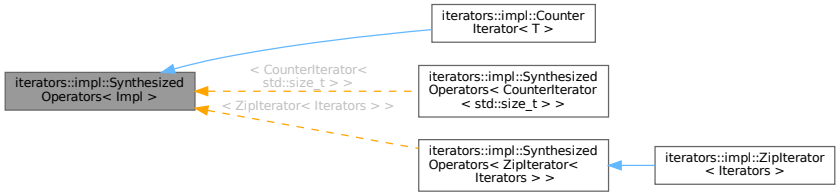
- Iterators.hpp

3.4 iterators::impl::SynthesizedOperators< Impl > Struct Template Reference

CRTP-class that provides additional pointer arithmetic operators synthesized from basic operators.

`#include <Iterators.hpp>`

Inheritance diagram for iterators::impl::SynthesizedOperators< Impl >:



Public Member Functions

- template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference_type))) >
  constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) const noexcept(noexcept(*(std::declval< Impl >()+n)))
- template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference_type))) >
  constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) noexcept(noexcept(*(std::declval< Impl >()+n)))
- template<REQUIRES_IMPL(Impl,++INSTANCE_OF_IMPL) >
  constexpr Impl operator++ (int) noexcept(noexcept(++std::declval< Impl >()) &&std::is_nothrow_copy_constructible_v< Impl >)
- template<REQUIRES_IMPL(Impl, --INSTANCE_OF_IMPL) >
  constexpr Impl operator-- (int) noexcept(noexcept(--std::declval< Impl >()) &&std::is_nothrow_copy_constructible_v< Impl >)
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL==INSTANCE_OF(T)) >
  constexpr bool operator!= (const T &other) const noexcept(noexcept(INSTANCE_OF_IMPL==other))
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL > INSTANCE_OF(T)) >
  constexpr bool operator<= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL > rhs))
- template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL< INSTANCE_OF(T)) >
  constexpr bool operator>= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL< rhs))

Friends

- template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL+=INSTANCE_OF(typename Implementation::difference_type)) >
  constexpr auto operator+ (Impl it, typename Implementation::difference_type n) noexcept(noexcept(std::declval< Impl >()+=n))
- template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL+=INSTANCE_OF(typename Implementation::difference_type)) >
  constexpr auto operator+ (typename Implementation::difference_type n, Impl it) noexcept(noexcept(std::declval< Impl >()+=n))
- template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL -=INSTANCE_OF(typename Implementation::difference_type)) >
  constexpr auto operator- (Impl it, typename Implementation::difference_type n) noexcept(noexcept(std::declval< Impl >() -=n))

3.4.1 Detailed Description

template<typename Impl>
struct iterators::impl::SynthesizedOperators< Impl >

CRTP-class that provides additional pointer arithmetic operators synthesized from basic operators.

Adds the following operators

- postfix increment and decrement (requires the respective prefix operators)
- array subscript operator[] (requires operator+ and dereference operator)
- binary arithmetic operators (requires compound assignment operators)
- inequality comparison (requires operator==)
- less than or equal comparison (requires operator>)
- greater than or equal comparison (requires operator<)

Template Parameters

Impl	Base class

3.4.2 Member Function Documentation

operator"!=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL==INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator!= (
            const T & other ) const  [inline], [constexpr], [noexcept]
```

Inequality comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not equal to other

operator++()

```
template<typename Impl >
template<REQUIRES_IMPL(Impl,++INSTANCE_OF_IMPL) >
constexpr Impl iterators::impl::SynthesizedOperators< Impl >::operator++ (
            int  )  [inline], [constexpr], [noexcept]
```

Postfix increment. Synthesized from prefix increment

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator--()

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, --INSTANCE_OF_IMPL) >
constexpr Impl iterators::impl::SynthesizedOperators< Impl >::operator-- (
            int  )  [inline], [constexpr], [noexcept]
```

Postfix decrement. Synthesized from prefix decrement

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator<=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL > INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator<= (
            const T & rhs ) const  [inline], [constexpr], [noexcept]
```

Less than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not greater than other

operator>=()

```
template<typename Impl >
template<typename T , REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL< INSTANCE_OF(T)) >
constexpr bool iterators::impl::SynthesizedOperators< Impl >::operator>= (
            const T & rhs ) const  [inline], [constexpr], [noexcept]
```

Greater than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not less than other

operator[]() [1/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference↩
_type))) >
constexpr decltype(auto) iterators::impl::SynthesizedOperators< Impl >::operator[] (
            typename Implementation::difference_type n ) const  [inline], [constexpr], [noexcept]
```

Array subscript operator

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

n	index

Returns

∗(∗this + n)

operator[]() [2/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL+INSTANCE_OF(typename Implementation::difference↩
_type))) >
```

```
constexpr decltype(auto) iterators::impl::SynthesizedOperators< Impl >::operator[] (
            typename Implementation::difference_type n )  [inline], [constexpr], [noexcept]
```

3.4.3 Friends And Related Symbol Documentation

operator+ [1/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL+=INSTANCE_OF(typename Implementation::difference↩
_type)) >
constexpr auto operator+ (
            Impl it,
            typename Implementation::difference_type n )  [friend]
```

Binary +plus operator. Synthesized from compound assignment operator+=

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

it	left hand side
n	right hand side

Returns

Instance of Impl

operator+ [2/2]

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL+=INSTANCE_OF(typename Implementation::difference↩
_type)) >
constexpr auto operator+ (
            typename Implementation::difference_type n,
            Impl it )  [friend]
```

Binary +plus operator. Synthesized from compound assignment operator+=

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

n	left hand side
it	right hand side

Returns

Instance of Impl

operator-

```
template<typename Impl >
template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL -=INSTANCE_OF(typename Implementation::difference↩
_type)) >
constexpr auto operator- (
            Impl it,
            typename Implementation::difference_type n )  [friend]
```

Binary minus operator. Synthesized from compound assignment operator-=

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

it	left hand side
n	right hand side

Returns

Instance of Impl

The documentation for this struct was generated from the following file:

- Iterators.hpp

3.5 iterators::impl::Unreachable Struct Reference

represents the unreachable end of an infinite sequence

`#include <Iterators.hpp>`

3.5.1 Detailed Description

represents the unreachable end of an infinite sequence

The documentation for this struct was generated from the following file:

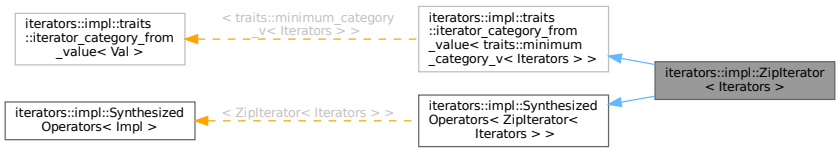
- Iterators.hpp

3.6 iterators::impl::ZipIterator< Iterators > Class Template Reference

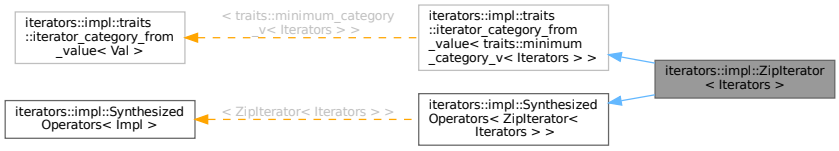
Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.

`#include <Iterators.hpp>`

Inheritance diagram for iterators::impl::ZipIterator< Iterators >:



Collaboration diagram for iterators::impl::ZipIterator< Iterators >:



Public Types

- using **value_type** = traits::values_t< Iterators >
- using **reference** = traits::references_t< Iterators >
- using **pointer** = void
- using **difference_type** = std::ptrdiff_t

Public Member Functions

- constexpr **ZipIterator** (const Iterators &iterators) noexcept(std::is_nothrow_copy_constructible_v< Iterators >)
- template<typename Its = Iterators, typename = std::enable_if_t<traits::is_incrementable_v<Its>>>
  constexpr ZipIterator & operator++ () noexcept(traits::is_nothrow_incrementible_v< Iterators >)
- template<typename Its , REQUIRES(ZipIterator::oneEqual(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
  constexpr bool operator== (const ZipIterator< Its > &other) const noexcept(noexcept(ZipIterator::one↩
  Equal(std::declval< Iterators >(), other.getIterators())))
- template<typename Its = Iterators, typename = std::enable_if_t<traits::is_dereferencible_v<Its>>>
  constexpr reference operator* () const noexcept(traits::is_nothrow_dereferencible_v< Iterators >)
- constexpr auto getIterators () const noexcept -> const Iterators &
- constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) const noexcept(noexcept(*(std↩
  ::declval< ZipIterator< Iterators > >()+n)))
- constexpr decltype(auto) operator[ ] (typename Implementation::difference_type n) noexcept(noexcept(*(std↩
  ::declval< ZipIterator< Iterators > >()+n)))
- constexpr ZipIterator< Iterators > operator++ (int) noexcept(noexcept(++std::declval< ZipIterator< Iterators > >()) &&std::is_nothrow_copy_constructible_v< ZipIterator< Iterators > >)

- constexpr ZipIterator< Iterators > operator-- (int) noexcept(noexcept(--std::declval< ZipIterator< Iterators > >()) &&std::is_nothrow_copy_constructible_v< ZipIterator< Iterators > >)
- constexpr bool operator!= (const T &other) const noexcept(noexcept(INSTANCE_OF_IMPL==other))
- constexpr bool operator<= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL > rhs))
- constexpr bool operator>= (const T &rhs) const noexcept(noexcept(INSTANCE_OF_IMPL< rhs))

bidirectional iteration

the following operators are only available if all underlying iterators support bidirectional access

- template<bool IsBidirectional = traits::is_bidirectional_v<Iterators>>
  constexpr auto operator-- () noexcept(traits::is_nothrow_decrementible_v< Iterators >) -> std::enable_if_t< IsBidirectional, ZipIterator & >

random access operators

the following operators are only available if all underlying iterators support random access

- template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
  constexpr auto operator+= (difference_type n) noexcept(traits::is_nothrow_compound_assignable_plus_v< Iterators >) -> std::enable_if_t< IsRandomAccessible, ZipIterator & >
- template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
  constexpr auto operator-= (difference_type n) noexcept(traits::is_nothrow_compound_assignable_minus_v< Iterators >) -> std::enable_if_t< IsRandomAccessible, ZipIterator & >
- template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>, REQUIRES(ZipIterator::minDifference(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
  constexpr auto operator- (const ZipIterator< Its > &other) const -> std::enable_if_t< IsRandomAccessible, difference_type >
- template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>, REQUIRES(ZipIterator::allLess(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
  constexpr auto operator< (const ZipIterator< Its > &other) const noexcept(noexcept(ZipIterator::allLess(INSTANCE_OF(Iterators), INSTANCE_OF(Its)))) -> std::enable_if_t< IsRandomAccessible, bool >
- template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>, REQUIRES(ZipIterator::allGreater(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
  constexpr auto operator> (const ZipIterator< Its > &other) const noexcept(noexcept(ZipIterator::allGreater(INSTANCE_OF(Iterators), INSTANCE_OF(Its)))) -> std::enable_if_t< IsRandomAccessible, bool >

Related Symbols

(Note that these are not member symbols.)

- template<typename ... Iterators>
  constexpr auto zip_i (Iterators ...iterators) -> impl::ZipIterator< std::tuple< Iterators... > >

3.6.1 Detailed Description

template<typename Iterators>
class iterators::impl::ZipIterator< Iterators >

Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.

ZipIterators only support the operators of the least powerful underling iterator. Zipping a random access iterator (e.g. from std::vector) and a bidirectional iterator (e.g. from std::list) results in a bidirectional iterator. All operators are SFINAE friendly.

ZipIterators return a tuple of references to the range elements. When using structured bindings, no additional reference binding is necessary.

Let `z` be a ZipIterator composed from two `std::vector<int>`

```
auto [val1, val2] = *z; // val1 and val2 are references to the vector elements
val1 = 17; // this will change the respective value in the first vector
```

Template Parameters

Iterators	Underlying iterator types

3.6.2 Member Function Documentation

getIterators()

```
template<typename Iterators >
constexpr auto iterators::impl::ZipIterator< Iterators >::getIterators ( ) const -> const
Iterators&   [inline], [constexpr], [noexcept]
```

Getter for underlying iterators

Returns

Const reference to underlying iterators

operator"!=()

```
constexpr bool iterators::impl::SynthesizedOperators< ZipIterator< Iterators > >::operator!=
(
            const T & other ) const  [inline], [constexpr], [noexcept], [inherited]
```

Inequality comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not equal to other

operator*()

```
template<typename Iterators >
template<typename Its = Iterators, typename = std::enable_if_t<traits::is_dereferencible_v<↩
Its>>>
constexpr reference iterators::impl::ZipIterator< Iterators >::operator* ( ) const  [inline],
[constexpr], [noexcept]
```

Dereferences all underlying iterators and returns a tuple of the resulting range reference types

Template Parameters

Its	SFINAE guard, do not specify

Returns

tuple of references to range elements

operator++() [1/2]

```
template<typename Iterators >
template<typename Its = Iterators, typename = std::enable_if_t<traits::is_incrementable_v<↩
Its>>>
constexpr ZipIterator & iterators::impl::ZipIterator< Iterators >::operator++ ( )  [inline],
[constexpr], [noexcept]
```

Increments all underlying iterators by one

Template Parameters

Its	SFINAE guard, do not specify

Returns

reference to this

operator++() [2/2]

```
constexpr ZipIterator< Iterators > iterators::impl::SynthesizedOperators< ZipIterator< Iterators
> >::operator++ (
            int  )  [inline], [constexpr], [noexcept], [inherited]
```

Postfix increment. Synthesized from prefix increment

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator+=()

```
template<typename Iterators >
template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
```

```
constexpr auto iterators::impl::ZipIterator< Iterators >::operator+= (
            difference_type n ) -> std::enable_if_t<IsRandomAccessible, ZipIterator &>   [inline],
[constexpr], [noexcept]
```

Compound assignment increment. Increments all underlying iterators by n. Only available if all underlying iterators support at least random access

Template Parameters

IsRandomAccessible	SFINAE guard, do not specify

Parameters

n	increment

Returns

reference to this

operator-()

```
template<typename Iterators >
template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>,
REQUIRES( ZipIterator::minDifference(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
constexpr auto iterators::impl::ZipIterator< Iterators >::operator- (
            const ZipIterator< Its > & other ) const -> std::enable_if_t<IsRandomAccessible,
difference_type>   [inline], [constexpr]
```

Returns the minimum pairwise difference n between all underlying iterators of ∗this and other, such that (other + n) == ∗this Only available if all underlying iterators support at least random access

Template Parameters

Its	Iterator types of right hand side
IsRandomAccessible	SFINAE guard, do not specify

Parameters

other	right hand side

Returns

integer n such that (other + n) == ∗this

operator--() [1/2]

```
template<typename Iterators >
template<bool IsBidirectional = traits::is_bidirectional_v<Iterators>>
```

```
constexpr auto iterators::impl::ZipIterator< Iterators >::operator-- ( ) -> std::enable_if_↩
t<IsBidirectional, ZipIterator &>   [inline], [constexpr], [noexcept]
```

Decrements all underlying iterators by one. Only available if all iterators support at least bidirectional access

Template Parameters

IsBidirectional	SFINAE guard, do not specify

Returns

reference to this

operator--() [2/2]

```
constexpr ZipIterator< Iterators > iterators::impl::SynthesizedOperators< ZipIterator< Iterators
> >::operator-- (
            int  )  [inline], [constexpr], [noexcept], [inherited]
```

Postfix decrement. Synthesized from prefix decrement

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Returns

Instance of Impl

operator-=()

```
template<typename Iterators >
template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
constexpr auto iterators::impl::ZipIterator< Iterators >::operator-= (
            difference_type n ) -> std::enable_if_t<IsRandomAccessible, ZipIterator &>   [inline],
[constexpr], [noexcept]
```

Compound assignment decrement. Decrements all underlying iterators by n. Only available if all underlying iterators support at least random access

Template Parameters

IsRandomAccessible	SFINAE guard, do not specify

Parameters

n	decrement

Returns

reference to this

operator<()

```
template<typename Iterators >
template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>,
REQUIRES( ZipIterator::allLess(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
constexpr auto iterators::impl::ZipIterator< Iterators >::operator< (
            const ZipIterator< Its > & other ) const -> std::enable_if_t<IsRandomAccessible,
bool>   [inline], [constexpr], [noexcept]
```

Pairwise less comparison of underlying iterators Only available if all underlying iterators support at least random access

Template Parameters

Its	Iterator types of right hand side
IsRandomAccessible	SFINAE guard, do not specify

Parameters

other	right hand side

Returns

true if all underlying iterators compare less to the corresponding iterators from other

operator<=()

```
constexpr bool iterators::impl::SynthesizedOperators< ZipIterator< Iterators > >::operator<=
(
            const T & rhs ) const  [inline], [constexpr], [noexcept], [inherited]
```

Less than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not greater than other

operator==()

```
template<typename Iterators >
template<typename Its , REQUIRES(ZipIterator::oneEqual(INSTANCE_OF(Iterators), INSTANCE_↩
OF(Its))) >
constexpr bool iterators::impl::ZipIterator< Iterators >::operator== (
            const ZipIterator< Its > & other ) const  [inline], [constexpr], [noexcept]
```

Pairwise equality comparison of underlying iterators

Template Parameters

Its	Iterator types of right hand side

Parameters

other	right hand side

Returns

true if at least one underlying iterator compares equal to the corresponding iterator from other

operator>()

```
template<typename Iterators >
template<typename Its , bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>,
REQUIRES( ZipIterator::allGreater(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
constexpr auto iterators::impl::ZipIterator< Iterators >::operator> (
            const ZipIterator< Its > & other ) const -> std::enable_if_t<IsRandomAccessible,
bool>   [inline], [constexpr], [noexcept]
```

Pairwise grater comparison of underlying iterators Only available if all underlying iterators support at least random access

Template Parameters

Its	Iterator types of right hand side
IsRandomAccessible	SFINAE guard, do not specify

Parameters

other	right hand side

Returns

true if all underlying iterators compare greater to the corresponding iterators from other

operator>=()

```
constexpr bool iterators::impl::SynthesizedOperators< ZipIterator< Iterators > >::operator>=
(
            const T & rhs ) const  [inline], [constexpr], [noexcept], [inherited]
```

Greater than or equal comparison

Template Parameters

T	type of right hand side
Implementation	SFINAE helper, do not specify explicitly

Parameters

other	right hand side

Returns

true if this is not less than other

operator[]() [1/2]

```
constexpr decltype(auto) iterators::impl::SynthesizedOperators< ZipIterator< Iterators > >↩
::operator[] (
            typename Implementation::difference_type n ) const  [inline], [constexpr], [noexcept],
[inherited]
```

Array subscript operator

Template Parameters

Implementation	SFINAE helper, do not specify explicitly

Parameters

n	index

Returns

∗(∗this + n)

operator[]() [2/2]

```
constexpr decltype(auto) iterators::impl::SynthesizedOperators< ZipIterator< Iterators > >↩
::operator[] (
            typename Implementation::difference_type n )  [inline], [constexpr], [noexcept],
[inherited]
```

3.6.3 Friends And Related Symbol Documentation

zip_i()

```
template<typename ...  Iterators>
constexpr auto zip_i (
            Iterators ...  iterators ) -> impl::ZipIterator<std::tuple<Iterators...>>   [related]
```

Function that is used to create a impl::ZipIterator from an arbitrary number of iterators

Template Parameters

Iterators	type of iterators

Parameters

iterators	arbitrary number of iterators

Returns

impl::ZipIterator

Note

ZipIterators have the same iterator category as the least powerful underlying operator. This means that for example, zipping a random access iterator and a bidirectional iterator only yields a bidirectional impl::ZipIterator

The documentation for this class was generated from the following file:

- Iterators.hpp

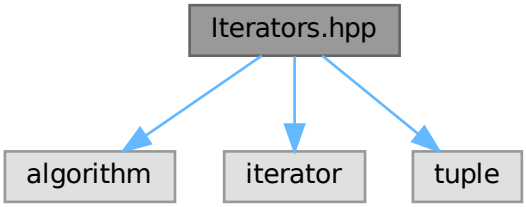
4 File Documentation

4.1 Iterators.hpp File Reference

This file contains the definitions of Python-like zip- and enumerate-functions. They can be used in range based for-loops to loop over multiple ranges at the same time, or to index a range while looping respectively.

```
#include <algorithm>
#include <iterator>
```

`#include <tuple>`
Include dependency graph for Iterators.hpp:



Classes

- struct iterators::impl::RefTuple< Ts >

  Proxy reference type that supports assignment and swap even on const instances.
- struct iterators::impl::SynthesizedOperators< Impl >

  CRTP-class that provides additional pointer arithmetic operators synthesized from basic operators.
- class iterators::impl::ZipIterator< Iterators >

  Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.
- struct iterators::impl::Unreachable

  represents the unreachable end of an infinite sequence
- struct iterators::impl::CounterIterator< T >

  Iterator of an infinite sequence of numbers. Simply increments an internal counter.
- struct iterators::impl::CounterRange< T >

  Represents an infinite range of numbers.

Namespaces

- namespace iterators

  namespace containing zip and enumerate functions
- namespace iterators::impl

  namespace containing structures and helpers used to implement zip and enumerate. Normally there is no need to use any of its members directly
- namespace iterators::impl::traits

  namespace containing type traits used in implementation of zip and enumerate

Macros

- #define **REFERENCE**(TYPE) std::declval<std::add_lvalue_reference_t<TYPE>>()
- #define ALL_NOEXCEPT(OP, NAME)
- #define **ELEMENT1** std::get<Idx>(std::forward<Tuple1>(tuple1))
- #define **ELEMENT2** std::get<Idx>(std::forward<Tuple2>(tuple2))
- #define BINARY_TUPLE_FOR_EACH(OPERATION, NAME)
- #define **BINARY_TUPLE_FOR_EACH_FOLD**(OPERATION, COMBINATOR, NAME) BINARY_TUPLE_↩ FOR_EACH(((OPERATION) COMBINATOR ...), NAME)

- #define TYPE_MAP_DEFAULT
- #define TYPE_MAP(TYPE, VALUE)
- #define TYPE_MAP_ALIAS
- #define **INSTANCE_OF**(TYPENAME) std::declval<TYPENAME>()
- #define **INSTANCE_OF_IMPL** INSTANCE_OF(Implementation)
- #define **REQUIRES_IMPL**(TYPENAME, EXPRESSION) typename Implementation = TYPENAME, typename = std::void_t<decltype(EXPRESSION)>
- #define **REQUIRES**(EXPRESSION) typename = std::void_t<decltype(EXPRESSION)>
- #define **DERIVE_VIEW_INTERFACE**(CLASS)

Functions

- template<std::size_t Offset, std::size_t ... Idx>
  constexpr auto **iterators::impl::index_seq_impl** (std::index_sequence< Idx... >) noexcept
- template<std::size_t Start, std::size_t End>
  constexpr auto **iterators::impl::index_seq** () noexcept
- template<typename Tuple , std::size_t ... Idx1, std::size_t ... Idx2>
  constexpr auto **iterators::impl::tuple_split_impl** (Tuple &&tuple, std::index_sequence< Idx1... >, std::index_sequence< Idx2... >)
- template<std::size_t Idx, typename Tuple >
  constexpr auto **iterators::impl::tuple_split** (Tuple &&tuple)
- template<template< typename > typename Predicate, std::size_t Idx, typename ... Ts>
  constexpr std::size_t **iterators::impl::index_of** () noexcept
- template<typename Tuple >
  constexpr auto **iterators::impl::swap** (Tuple &&a, Tuple &&b) -> std::enable_if_t< iterators::impl::traits::is_same_template_v< iterators::impl::RefTuple, Tuple > >
- template<typename ... Iterable>
  struct ZipView iterators::impl::DERIVE_VIEW_INTERFACE (ZipView< Iterable... >)

  Zip-view that provides begin() and end() member functions. Use to loop over multiple ranges at the same time using ranged based for-loops.
- template<typename T >
  constexpr T iterators::impl::sgn (T val) noexcept
- template<typename ... Iterators>
  constexpr auto iterators::zip_i (Iterators ...iterators) -> impl::ZipIterator< std::tuple< Iterators... > >
- template<typename ... Iterable>
  constexpr auto iterators::zip (Iterable &&...iterable)
- template<typename ... Iterable>
  constexpr auto iterators::const_zip (Iterable &&...iterable)
- template<typename Container , typename T = std::size_t>
  constexpr auto iterators::enumerate (Container &&container, T start=T(0), T increment=T(1))
- template<typename Container , typename T = std::size_t>
  constexpr auto iterators::const_enumerate (Container &&container, T start=T(0), T increment=T(1))
- template<typename ... Iterable>
  constexpr auto iterators::zip_enumerate (Iterable &&...iterable)
- template<typename ... Iterable>
  constexpr auto iterators::const_zip_enumerate (Iterable &&...iterable)

4.1.1 Detailed Description

This file contains the definitions of Python-like zip- and enumerate-functions. They can be used in range based for-loops to loop over multiple ranges at the same time, or to index a range while looping respectively.

Author

tim Luchterhand

Date

10.09.21

4.1.2 Macro Definition Documentation

ALL_NOEXCEPT

```
#define ALL_NOEXCEPT(
              OP,
              NAME )
```

Value:

```
template<typename T> \
struct NAME { \
    static constexpr bool value = false; \
}; \
template<typename ...Ts> \
struct NAME <std::tuple<Ts...>> { \
    static constexpr bool value = (... && noexcept(OP)); \
};                  \
template<typename T> \
inline constexpr bool NAME##_v = NAME<T>::value;
```

BINARY_TUPLE_FOR_EACH

```
#define BINARY_TUPLE_FOR_EACH(
              OPERATION,
              NAME )
```

Value:

```
template<typename Tuple1, typename Tuple2, std::size_t ...Idx> \
static constexpr auto NAME##Impl(Tuple1 &&tuple1, Tuple2 &&tuple2, std::index_sequence<Idx...>) \
noexcept(noexcept((OPERATION))) -> decltype(OPERATION) { \
    return (OPERATION); \
} \
template<typename Tuple1, typename Tuple2> \
static constexpr auto NAME(Tuple1 &&tuple1, Tuple2 &&tuple2) \
noexcept(noexcept(NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),        \
    std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1>>{}))) \
-> decltype(NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),               \
    std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1>>{})) { \
    static_assert(std::tuple_size_v<std::remove_reference_t<Tuple1> ==
std::tuple_size_v<std::remove_reference_t<Tuple2>); \
    return NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),                \
        std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1>>{}); \
}
```

TYPE_MAP

```
#define TYPE_MAP(
              TYPE,
              VALUE )
```

Value:

```
template<>            \
struct type_to_value<TYPE> { \
    static constexpr std::size_t value = VALUE; \
};                  \
template<>            \
struct value_to_type<VALUE>{ \
    static_assert(VALUE != 0, "0 is a reserved value"); \
    using type = TYPE;\
};
```

TYPE_MAP_ALIAS

```
#define TYPE_MAP_ALIAS
```

Value:

```
        template<typename T> \
        constexpr inline std::size_t type_to_value_v = type_to_value<T>::value; \
        template<std::size_t V>                                                  \
        using value_to_type_t = typename value_to_type<V>::type;
```

TYPE_MAP_DEFAULT

```
#define TYPE_MAP_DEFAULT
```

Value:

```
        template<typename> \
        struct type_to_value {}; \
        template<std::size_t>    \
        struct value_to_type {};
```

4.2 Iterators.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file Iterators.hpp
00003  * @author tim Luchterhand
00004  * @date 10.09.21
00005  * @brief This file contains the definitions of Python-like zip- and enumerate-functions. They can be
      used in range
00006  * based for-loops to loop over multiple ranges at the same time, or to index a range while looping
      respectively.
00007  */
00008
00009 #ifndef ITERATORTOOLS_ITERATORS_HPP
00010 #define ITERATORTOOLS_ITERATORS_HPP
00011
00012 #include <algorithm>
00013 #include <iterator>
00014 #include <tuple>
00015
00016 #define REFERENCE(TYPE) std::declval<std::add_lvalue_reference_t<TYPE»()
00017
00018 #define ALL_NOEXCEPT(OP, NAME)
00019         template<typename T>
00020         struct NAME {
00021             static constexpr bool value = false;
00022         };
00023         template<typename ...Ts>
00024         struct NAME <std::tuple<Ts...» {
00025             static constexpr bool value = (... && noexcept(OP));
00026         };
00027         template<typename T>
00028         inline constexpr bool NAME##_v = NAME<T>::value;
00029
00030 #define ELEMENT1 std::get<Idx>(std::forward<Tuple1>(tuple1))
00031 #define ELEMENT2 std::get<Idx>(std::forward<Tuple2>(tuple2))
00032
00033 #define BINARY_TUPLE_FOR_EACH(OPERATION, NAME)
00034         template<typename Tuple1, typename Tuple2, std::size_t ...Idx>
00035         static constexpr auto NAME##Impl(Tuple1 &&tuple1, Tuple2 &&tuple2,
      std::index_sequence<Idx...>)
00036         noexcept(noexcept((OPERATION))) -> decltype(OPERATION) {
00037             return (OPERATION);
00038         }
00039         template<typename Tuple1, typename Tuple2>
00040         static constexpr auto NAME(Tuple1 &&tuple1, Tuple2 &&tuple2)
00041         noexcept(noexcept(NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),
00042             std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1»>{})))
00043         -> decltype(NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),
00044             std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1»>{})) {
00045             static_assert(std::tuple_size_v<std::remove_reference_t<Tuple1» ==
      std::tuple_size_v<std::remove_reference_t<Tuple2»);
00046             return NAME##Impl(std::forward<Tuple1>(tuple1), std::forward<Tuple2>(tuple2),
00047                 std::make_index_sequence<std::tuple_size_v<std::remove_reference_t<Tuple1»>{});
```

```
00048         }
00049
00050 #define BINARY_TUPLE_FOR_EACH_FOLD(OPERATION, COMBINATOR, NAME) BINARY_TUPLE_FOR_EACH( ( (OPERATION)
      COMBINATOR ...), NAME)
00051
00052 #define TYPE_MAP_DEFAULT
00053         template<typename>
00054         struct type_to_value {};
00055         template<std::size_t>
00056         struct value_to_type {};
00057
00058 #define TYPE_MAP(TYPE, VALUE)
00059         template<>
00060         struct type_to_value<TYPE> {
00061             static constexpr std::size_t value = VALUE;
00062         };
00063         template<>
00064         struct value_to_type<VALUE>{
00065             static_assert(VALUE != 0, "0 is a reserved value");
00066             using type = TYPE;
00067         };
00068
00069 #define TYPE_MAP_ALIAS
00070         template<typename T>
00071         constexpr inline std::size_t type_to_value_v = type_to_value<T>::value;
00072         template<std::size_t V>
00073         using value_to_type_t = typename value_to_type<V>::type;
00074
00075 #define INSTANCE_OF(TYPENAME) std::declval<TYPENAME>()
00076
00077 #define INSTANCE_OF_IMPL INSTANCE_OF(Implementation)
00078
00079 #define REQUIRES_IMPL(TYPENAME, EXPRESSION) typename Implementation = TYPENAME, typename =
      std::void_t<decltype(EXPRESSION)>
00080
00081 #define REQUIRES(EXPRESSION) typename = std::void_t<decltype(EXPRESSION)>
00082
00083 #ifndef __STD_RANGES_DISABLED__
00084 #ifdef __cpp_lib_ranges
00085 #include <ranges>
00086 #define DERIVE_VIEW_INTERFACE(CLASS) : std::ranges::view_interface<CLASS>
00087 #define __USE_VIEW_INTERFACE__
00088 #else
00089 #define DERIVE_VIEW_INTERFACE(CLASS)
00090 #endif
00091 #else
00092 #define DERIVE_VIEW_INTERFACE(CLASS)
00093 #endif
00094
00095 /**
00096  * @brief namespace containing zip and enumerate functions
00097  */
00098 namespace iterators {
00099
00100     /**
00101      * @brief namespace containing structures and helpers used to implement zip and enumerate.
00102      * Normally there is no need to use any of its members directly
00103      */
00104     namespace impl {
00105         /**
00106          * @brief Proxy reference type that supports assignment and swap even on const instances.
00107          * @details @copybrief
00108          * Actual constness is determined by element constness
00109          * @tparam Ts
00110          */
00111         template<typename ...Ts>
00112         struct RefTuple : public std::tuple<Ts...> {
00113             using std::tuple<Ts...>::tuple;
00114
00115             static constexpr bool Assignable = (not std::is_const_v<std::remove_reference_t<Ts» &&
      ...);
00116
00117             template<typename Dummy = int>
00118             constexpr auto operator=(RefTuple &&other)
      noexcept((std::is_nothrow_move_assignable_v<std::remove_reference_t<Ts» && ...))
00119                     -> std::enable_if_t<std::is_same_v<Dummy, int> and Assignable, RefTuple &> {
00120                 if (selfAssignGuard(other)) {
00121                     return *this;
00122                 }
00123
00124                 moveAssign(*this, other);
00125                 return *this;
00126             }
00127
00128             template<typename ...Us>
00129             decltype(auto) operator=(const std::tuple<Us...> &other) const {
00130                 if (selfAssignGuard(other)) {
```

```
00131                   return *this;
00132               }
00133
00134               copyAssign(*this, other);
00135               return *this;
00136           }
00137
00138           template<typename ...Us>
00139           decltype(auto) operator=(std::tuple<Us...> &&other) const {
00140               if (selfAssignGuard(other)) {
00141                   return *this;
00142               }
00143
00144               moveAssign(*this, other);
00145               return *this;
00146           }
00147
00148           template<std::size_t Idx>
00149           constexpr decltype(auto) get() const & noexcept {
00150               return std::get<Idx>(static_cast<std::tuple<Ts...>const &>(*this));
00151           }
00152
00153           template<std::size_t Idx>
00154           constexpr decltype(auto) get() & noexcept {
00155               return std::get<Idx>(static_cast<std::tuple<Ts...> &>(*this));
00156           }
00157
00158           template<std::size_t Idx>
00159           constexpr decltype(auto) get() const && noexcept {
00160               return std::get<Idx>(static_cast<std::tuple<Ts...> const &&>(*this));
00161           }
00162
00163           template<std::size_t Idx>
00164           constexpr decltype(auto) get() && noexcept {
00165               return std::get<Idx>(static_cast<std::tuple<Ts...> &&>(*this));
00166           }
00167
00168           template<typename Tuple>
00169           constexpr auto swap(Tuple &&other) const -> std::enable_if_t<std::is_same_v<Tuple,
      RefTuple> and Assignable> {
00170               auto tmp = std::move(*this);
00171               // move of forwarding reference because we always move, even const ref.
00172               moveAssign(*this, std::move(other));
00173               moveAssign(other, std::move(tmp));
00174           }
00175
00176
00177       private:
00178           template<typename T>
00179           constexpr bool selfAssignGuard(const T &t) const noexcept {
00180               if constexpr (std::is_same_v<T, RefTuple>) {
00181                   return &t == this;
00182               } else {
00183                   return false;
00184               }
00185           }
00186
00187           BINARY_TUPLE_FOR_EACH(((ELEMENT1 = ELEMENT2), ...), copyAssign)
00188           BINARY_TUPLE_FOR_EACH(((ELEMENT1 = std::move(ELEMENT2)), ...), moveAssign)
00189       };
00190
00191       /**
00192        * @brief namespace containing type traits used in implementation of zip and enumerate
00193        */
00194       namespace traits {
00195           template<bool Cond, typename T>
00196           using reference_if_t = std::conditional_t<Cond, std::add_lvalue_reference_t<T>, T>;
00197
00198           template<bool Cond, typename T>
00199           using const_if_t = std::conditional_t<Cond, std::add_const_t<T>, T>;
00200
00201           template<typename T, typename = std::void_t<»
00202           struct is_container : std::false_type {};
00203
00204           template<typename T>
00205           struct is_container<T, std::void_t<decltype(std::begin(std::declval<T>()),
      std::end(std::declval<T>()))»
00206                   : std::true_type {};
00207
00208           template<typename T>
00209           constexpr inline bool is_container_v = is_container<T>::value;
00210
00211           template<typename T, typename = std::void_t<»
00212           struct is_dereferencible : std::false_type {};
00213
00214           template<typename T>
00215           struct is_dereferencible<T, std::void_t<decltype(*std::declval<T>())» : std::true_type {};
```

```
00216
00217             template<typename ...Ts>
00218             struct is_dereferencible<std::tuple<Ts...>, void> {
00219                 static constexpr bool value = (is_dereferencible<Ts>::value && ...);
00220             };
00221
00222             template<typename T>
00223             constexpr inline bool is_dereferencible_v = is_dereferencible<T>::value;
00224
00225             template<typename T, typename = std::void_t<»
00226             struct is_incrementable : std::false_type {};
00227
00228             template<typename T>
00229             struct is_incrementable<T, std::void_t<decltype(++REFERENCE(T))» : std::true_type {};
00230
00231             template<typename ...Ts>
00232             struct is_incrementable<std::tuple<Ts...>, void> {
00233                 static constexpr bool value = (is_incrementable<Ts>::value && ...);
00234             };
00235
00236             template<typename T>
00237             constexpr inline bool is_incrementable_v = is_incrementable<T>::value;
00238
00239             template<typename T, bool B>
00240             struct dereference {
00241                 using type = void;
00242             };
00243
00244             template<typename T>
00245             struct dereference<T, true> {
00246                 using type = decltype(*std::declval<T>());
00247             };
00248
00249             template<typename T>
00250             using dereference_t = typename dereference<T, is_dereferencible_v<T»::type;
00251
00252             template<typename T>
00253             struct values{};
00254
00255             template<typename ...Ts>
00256             struct values<std::tuple<Ts...» {
00257                 using type =
      std::tuple<std::remove_const_t<std::remove_reference_t<dereference_t<Ts»>...>;
00258             };
00259
00260             template<typename T>
00261             using values_t = typename values<T>::type;
00262
00263             template<typename T>
00264             struct references{};
00265
00266             template<typename ...Ts>
00267             struct references<std::tuple<Ts...» {
00268                 using type = RefTuple<dereference_t<Ts>...>;
00269             };
00270
00271             template<typename T>
00272             using references_t = typename references<T>::type;
00273
00274             ALL_NOEXCEPT(++REFERENCE(Ts), is_nothrow_incrementible)
00275             ALL_NOEXCEPT(--REFERENCE(Ts), is_nothrow_decrementible)
00276             ALL_NOEXCEPT(*REFERENCE(Ts), is_nothrow_dereferencible)
00277             ALL_NOEXCEPT(REFERENCE(Ts) += 5, is_nothrow_compound_assignable_plus)
00278             ALL_NOEXCEPT(REFERENCE(Ts) -= 5, is_nothrow_compound_assignable_minus)
00279
00280             TYPE_MAP_DEFAULT
00281
00282             TYPE_MAP(std::input_iterator_tag, 1)
00283             TYPE_MAP(std::forward_iterator_tag, 2)
00284             TYPE_MAP(std::bidirectional_iterator_tag, 3)
00285             TYPE_MAP(std::random_access_iterator_tag, 4)
00286             #if __cplusplus > 201703L
00287             TYPE_MAP(std::contiguous_iterator_tag, 5)
00288             #endif
00289
00290             TYPE_MAP_ALIAS
00291
00292             template<typename T, typename = std::void_t<»
00293             struct iterator_category_value {
00294                 static constexpr std::size_t value = 0;
00295             };
00296
00297             template<typename T>
00298             struct iterator_category_value<T, std::void_t<typename
      std::iterator_traits<T>::iterator_category» {
00299                 static constexpr std::size_t value = type_to_value_v<typename
      std::iterator_traits<T>::iterator_category>;
```

```
00300             };
00301
00302             template<std::size_t Val>
00303             struct iterator_category_from_value {
00304                 using iterator_category = value_to_type_t<Val>;
00305             };
00306
00307             template<>
00308             struct iterator_category_from_value<0> {};
00309
00310             template<typename T>
00311             struct minimum_category {};
00312
00313             template<typename ...Ts>
00314             struct minimum_category<std::tuple<Ts...>> {
00315                 static constexpr std::size_t value =
      std::min({iterator_category_value<Ts>::value...});
00316             };
00317
00318             template<typename T>
00319             constexpr inline std::size_t minimum_category_v = minimum_category<T>::value;
00320
00321             template<typename T, typename = std::void_t<>>
00322             struct is_random_accessible {
00323                 static constexpr bool value = false;
00324             };
00325
00326             template<typename T>
00327             struct is_random_accessible<T, std::void_t<typename
      std::iterator_traits<T>::iterator_category>> {
00328                 static constexpr bool value = std::is_base_of_v<std::random_access_iterator_tag,
00329                         typename std::iterator_traits<T>::iterator_category>;
00330             };
00331
00332             template<typename ...Ts>
00333             struct is_random_accessible<std::tuple<Ts...>,
      std::void_t<value_to_type_t<minimum_category_v<std::tuple<Ts...>>>> {
00334                 static constexpr bool value = std::is_base_of_v<std::random_access_iterator_tag,
00335                         value_to_type_t<minimum_category_v<std::tuple<Ts...>>>;
00336             };
00337
00338             template<typename T>
00339             constexpr inline bool is_random_accessible_v = is_random_accessible<T>::value;
00340
00341             template<typename T, typename = std::void_t<>>
00342             struct is_bidirectional {
00343                 static constexpr bool value = false;
00344             };
00345
00346             template<typename T>
00347             struct is_bidirectional<T, std::void_t<typename
      std::iterator_traits<T>::iterator_category>> {
00348                 static constexpr bool value = std::is_base_of_v<std::bidirectional_iterator_tag,
00349                         typename std::iterator_traits<T>::iterator_category>;
00350             };
00351
00352             template<typename ...Ts>
00353             struct is_bidirectional<std::tuple<Ts...>,
      std::void_t<value_to_type_t<minimum_category_v<std::tuple<Ts...>>>> {
00354                 static constexpr bool value = std::is_base_of_v<std::bidirectional_iterator_tag,
00355                         value_to_type_t<minimum_category_v<std::tuple<Ts...>>>;
00356             };
00357
00358             template<typename T>
00359             constexpr inline bool is_bidirectional_v = is_bidirectional<T>::value;
00360
00361             template<typename T, typename = std::void_t<>>
00362             struct has_size : std::false_type {};
00363
00364             template<typename T>
00365             struct has_size<T,
      std::void_t<decltype(std::size(std::declval<std::remove_reference_t<T>>()))>>
00366                     : std::true_type {};
00367
00368             template<typename ...Ts>
00369             struct has_size<std::tuple<Ts...>> {
00370                 static constexpr bool value = (has_size<Ts>::value &&...);
00371             };
00372
00373             template<typename T>
00374             constexpr inline bool has_size_v = has_size<T>::value;
00375
00376             template<template<typename...> typename Template, typename T>
00377             struct is_same_template : std::false_type {};
00378
00379             template<template<typename...> typename Template, typename... Args>
00380             struct is_same_template<Template, Template<Args...>> : std::true_type {};
```

```
00381
00382             template<template<typename ...> typename Template, typename T>
00383             constexpr inline bool is_same_template_v = is_same_template<Template,
00384                     std::remove_const_t<std::remove_reference_t<T»>::value;
00385         }
00386
00387
00388         template<std::size_t Offset, std::size_t ...Idx>
00389         constexpr auto index_seq_impl(std::index_sequence<Idx...>) noexcept {
00390             return std::index_sequence<(Idx + Offset)...>{};
00391         }
00392
00393         template<std::size_t Start, std::size_t End>
00394         constexpr auto index_seq() noexcept {
00395             return index_seq_impl<Start>(std::make_index_sequence<End - Start>());
00396         }
00397
00398         template<typename Tuple, std::size_t ...Idx1, std::size_t ...Idx2>
00399         constexpr auto tuple_split_impl(Tuple &&tuple, std::index_sequence<Idx1...>,
      std::index_sequence<Idx2...>) {
00400             return std::make_pair(std::tuple<std::tuple_element_t<Idx1,
      Tuple>&&...>(std::get<Idx1>(std::forward<Tuple>(tuple))...),
00401                                   std::tuple<std::tuple_element_t<Idx2,
      Tuple>&&...>(std::get<Idx2>(std::forward<Tuple>(tuple))...));
00402         }
00403
00404         template<std::size_t Idx, typename Tuple>
00405         constexpr auto tuple_split(Tuple &&tuple) {
00406             return tuple_split_impl(std::forward<Tuple>(tuple),
00407                                     std::make_index_sequence<Idx>(),
00408                                     index_seq<Idx, std::tuple_size_v<Tuple»());
00409         }
00410
00411         template<template<typename> typename Predicate, std::size_t Idx, typename ...Ts>
00412         constexpr std::size_t index_of() noexcept {
00413             static_assert(Idx < sizeof...(Ts));
00414             if constexpr (Predicate<std::tuple_element_t<Idx, std::tuple<Ts...»>::value) {
00415                 return Idx;
00416             } else {
00417                 return index_of<Predicate, Idx + 1, Ts...>();
00418             }
00419         }
00420
00421         template<typename Tuple>
00422         constexpr auto swap(Tuple &&a, Tuple &&b)
00423         -> std::enable_if_t<iterators::impl::traits::is_same_template_v<iterators::impl::RefTuple,
      Tuple» {
00424             std::forward<Tuple>(a).swap(std::forward<Tuple>(b));
00425         }
00426
00427
00428         /**
00429          * @brief CRTP-class that provides additional pointer arithmetic operators synthesized from
      basic operators
00430          * @details @copybrief
00431          * Adds the following operators
00432          * - postfix increment and decrement (requires the respective prefix operators)
00433          * - array subscript operator[] (requires operator+ and dereference operator)
00434          * - binary arithmetic operators (requires compound assignment operators)
00435          * - inequality comparison (requires operator==)
00436          * - less than or equal comparison (requires operator>)
00437          * - greater than or equal comparison (requires operator<)
00438          * @tparam Impl Base class
00439          */
00440         template<typename Impl>
00441         struct SynthesizedOperators {
00442
00443             /**
00444              * Array subscript operator
00445              * @tparam Implementation SFINAE helper, do not specify explicitly
00446              * @param n index
00447              * @return *(*this + n)
00448              */
00449             template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL + INSTANCE_OF(typename
      Implementation::difference_type)))>
00450             constexpr decltype(auto) operator[](typename Implementation::difference_type n) const
00451             noexcept(noexcept(*(std::declval<Impl>() + n))) {
00452                 return *(this->getImpl() + n);
00453             }
00454
00455             /**
00456              * @copydoc SynthesizedOperators::operator[](typename Implementation::difference_type n)
00457              */
00458             template<REQUIRES_IMPL(Impl, *(INSTANCE_OF_IMPL + INSTANCE_OF(typename
      Implementation::difference_type)))>
00459             constexpr decltype(auto) operator[](typename Implementation::difference_type n)
00460             noexcept(noexcept(*(std::declval<Impl>() + n))) {
```

```
00461                 return *(this->getImpl() + n);
00462             }
00463 
00464             /**
00465              * Postfix increment. Synthesized from prefix increment
00466              * @tparam Implementation SFINAE helper, do not specify explicitly
00467              * @return Instance of Impl
00468              */
00469             template<REQUIRES_IMPL(Impl, ++INSTANCE_OF_IMPL)>
00470             constexpr Impl operator++(int)
00471             noexcept(noexcept(++std::declval<Impl>()) && std::is_nothrow_copy_constructible_v<Impl>) {
00472                 auto tmp = this->getImpl();
00473                 this->getImpl().operator++();
00474                 return tmp;
00475             }
00476 
00477             /**
00478              * Postfix decrement. Synthesized from prefix decrement
00479              * @tparam Implementation SFINAE helper, do not specify explicitly
00480              * @return Instance of Impl
00481              */
00482             template<REQUIRES_IMPL(Impl, --INSTANCE_OF_IMPL)>
00483             constexpr Impl operator--(int)
00484             noexcept(noexcept(--std::declval<Impl>()) && std::is_nothrow_copy_constructible_v<Impl>) {
00485                 auto tmp = this->getImpl();
00486                 this->getImpl().operator--();
00487                 return tmp;
00488             }
00489 
00490             /**
00491              * Binary +plus operator. Synthesized from compound assignment operator+=
00492              * @tparam Implementation SFINAE helper, do not specify explicitly
00493              * @param it left hand side
00494              * @param n right hand side
00495              * @return Instance of Impl
00496              */
00497             template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL += INSTANCE_OF(typename
     Implementation::difference_type)) >
00498             friend constexpr auto operator+(Impl it, typename Implementation::difference_type n)
00499             noexcept(noexcept(std::declval<Impl>() += n)) {
00500                 it += n;
00501                 return it;
00502             }
00503 
00504             /**
00505              * Binary +plus operator. Synthesized from compound assignment operator+=
00506              * @tparam Implementation SFINAE helper, do not specify explicitly
00507              * @param n left hand side
00508              * @param it right hand side
00509              * @return Instance of Impl
00510              */
00511             template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL += INSTANCE_OF(typename
     Implementation::difference_type))>
00512             friend constexpr auto operator+(typename Implementation::difference_type n, Impl it)
00513             noexcept(noexcept(std::declval<Impl>() += n)) {
00514                 it += n;
00515                 return it;
00516             }
00517 
00518             /**
00519              * Binary minus operator. Synthesized from compound assignment operator-=
00520              * @tparam Implementation SFINAE helper, do not specify explicitly
00521              * @param it left hand side
00522              * @param n right hand side
00523              * @return Instance of Impl
00524              */
00525             template<REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL -= INSTANCE_OF(typename
     Implementation::difference_type)) >
00526             friend constexpr auto operator-(Impl it, typename Implementation::difference_type n)
00527             noexcept(noexcept(std::declval<Impl>() -= n)) {
00528                 it -= n;
00529                 return it;
00530             }
00531 
00532             /**
00533              * Inequality comparison
00534              * @tparam T type of right hand side
00535              * @tparam Implementation SFINAE helper, do not specify explicitly
00536              * @param other right hand side
00537              * @return true if this is not equal to other
00538              */
00539             template<typename T, REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL == INSTANCE_OF(T))>
00540             constexpr bool operator!=(const T &other) const noexcept(noexcept(INSTANCE_OF_IMPL ==
     other)) {
00541                 return !(this->getImpl()== other);
00542             }
00543 
```

```
00544             /**
00545              * Less than or equal comparison
00546              * @tparam T type of right hand side
00547              * @tparam Implementation SFINAE helper, do not specify explicitly
00548              * @param other right hand side
00549              * @return true if this is not greater than other
00550              */
00551             template<typename T, REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL > INSTANCE_OF(T))>
00552             constexpr bool operator<=(const T& rhs) const noexcept(noexcept(INSTANCE_OF_IMPL > rhs)) {
00553                 return !(this->getImpl()> rhs);
00554             }
00555 
00556             /**
00557              * Greater than or equal comparison
00558              * @tparam T type of right hand side
00559              * @tparam Implementation SFINAE helper, do not specify explicitly
00560              * @param other right hand side
00561              * @return true if this is not less than other
00562              */
00563             template<typename T, REQUIRES_IMPL(Impl, INSTANCE_OF_IMPL < INSTANCE_OF(T))>
00564             constexpr bool operator>=(const T& rhs) const noexcept(noexcept(INSTANCE_OF_IMPL < rhs)) {
00565                 return !(this->getImpl() < rhs);
00566             }
00567 
00568         private:
00569             constexpr Impl &getImpl() noexcept {
00570                 return static_cast<Impl &>(*this);
00571             }
00572 
00573             constexpr const Impl &getImpl() const noexcept {
00574                 return static_cast<const Impl &>(*this);
00575             }
00576 
00577             SynthesizedOperators() = default;
00578             friend Impl;
00579         };
00580 
00581         /**
00582          * @brief Class combining multiple iterators into one. Use it to iterate over multiple ranges
      at the same time.
00583          * @details @copybrief
00584          * ZipIterators only support the operators of the least powerful underling iterator. Zipping a
      random access
00585          * iterator (e.g. from std::vector) and a bidirectional iterator (e.g. from std::list) results
      in a
00586          * bidirectional iterator. All operators are SFINAE friendly.
00587          *
00588          * ZipIterators return a tuple of references to the range elements. When using
00589          * structured bindings, no additional reference binding is necessary.
00590          *
00591          * Let ```z"` be a ZipIterator composed from two ```std::vector<int>"`
00592          * ```
00593          * auto [val1, val2] = *z; // val1 and val2 are references to the vector elements
00594          * val1 = 17; // this will change the respective value in the first vector
00595          * ```
00596          * @tparam Iterators Underlying iterator types
00597          */
00598         template<typename Iterators>
00599         class ZipIterator
00600                 : public traits::iterator_category_from_value<traits::minimum_category_v<Iterators»,
00601                   public SynthesizedOperators<ZipIterator<Iterators» {
00602 
00603         public:
00604             using value_type = traits::values_t<Iterators>;
00605             using reference = traits::references_t<Iterators>;
00606             using pointer = void;
00607             using difference_type = std::ptrdiff_t;
00608 
00609         private:
00610             BINARY_TUPLE_FOR_EACH_FOLD(ELEMENT1 == ELEMENT2, ||, oneEqual)
00611             BINARY_TUPLE_FOR_EACH_FOLD(ELEMENT1 < ELEMENT2, &&, allLess)
00612             BINARY_TUPLE_FOR_EACH_FOLD(ELEMENT1 > ELEMENT2, &&, allGreater)
00613             BINARY_TUPLE_FOR_EACH(std::min<difference_type>({ELEMENT1 - ELEMENT2 ...}), minDifference)
00614             Iterators iterators;
00615 
00616         public:
00617             using SynthesizedOperators<ZipIterator>::operator++;
00618             using SynthesizedOperators<ZipIterator>::operator--;
00619 
00620             constexpr ZipIterator() noexcept = default;
00621 
00622             explicit constexpr ZipIterator(
00623                     const Iterators &iterators)
      noexcept(std::is_nothrow_copy_constructible_v<Iterators>)
00624                     : iterators(iterators) {}
00625 
00626             /**
```

```
00627             * Increments all underlying iterators by one
00628             * @tparam Its SFINAE guard, do not specify
00629             * @return reference to this
00630             */
00631            template<typename Its = Iterators, typename =
      std::enable_if_t<traits::is_incrementable_v<Its>>
00632            constexpr ZipIterator &operator++()
      noexcept(traits::is_nothrow_incrementible_v<Iterators>) {
00633                std::apply([](auto &&...it) { (++it, ...); }, iterators);
00634                return *this;
00635            }
00636
00637            /**
00638             * @name bidirectional iteration
00639             * @brief the following operators are only available if all underlying iterators support
      bidirectional
00640             * access
00641             */
00642            ///@{
00643
00644            /**
00645             * Decrements all underlying iterators by one. Only available if all iterators support at
      least
00646             * bidirectional access
00647             * @tparam IsBidirectional SFINAE guard, do not specify
00648             * @return reference to this
00649             */
00650            template<bool IsBidirectional = traits::is_bidirectional_v<Iterators>>
00651            constexpr auto operator--() noexcept(traits::is_nothrow_decrementible_v<Iterators>)
00652                -> std::enable_if_t<IsBidirectional, ZipIterator &> {
00653                std::apply([](auto &&...it) { (--it, ...); }, iterators);
00654                return *this;
00655            }
00656
00657            ///@}
00658
00659            /**
00660             * @name random access operators
00661             * @brief the following operators are only available if all underlying iterators support
      random access
00662             *
00663             */
00664            ///@{
00665
00666            /**
00667             * Compound assignment increment. Increments all underlying iterators by n. Only available
      if all underlying
00668             * iterators support at least random access
00669             * @tparam IsRandomAccessible SFINAE guard, do not specify
00670             * @param n increment
00671             * @return reference to this
00672             */
00673            template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
00674            constexpr auto operator+=(difference_type n)
      noexcept(traits::is_nothrow_compound_assignable_plus_v<Iterators>)
00675                    -> std::enable_if_t<IsRandomAccessible, ZipIterator &> {
00676                std::apply([n](auto &&...it) {((it += n), ...);}, iterators);
00677                return *this;
00678            }
00679
00680            /**
00681             * Compound assignment decrement. Decrements all underlying iterators by n. Only available
      if all underlying
00682             * iterators support at least random access
00683             * @tparam IsRandomAccessible SFINAE guard, do not specify
00684             * @param n decrement
00685             * @return reference to this
00686             */
00687            template<bool IsRandomAccessible = traits::is_random_accessible_v<Iterators>>
00688            constexpr auto operator-=(difference_type n)
      noexcept(traits::is_nothrow_compound_assignable_minus_v<Iterators>)
00689                    -> std::enable_if_t<IsRandomAccessible, ZipIterator &> {
00690                std::apply([n](auto &&...it) {((it -= n), ...);}, iterators);
00691                return *this;
00692            }
00693
00694            /**
00695             * Returns the minimum pairwise difference n between all underlying iterators of *this and
      other, such that
00696             * (other + n) == *this
00697             * Only available if all underlying iterators support at least random access
00698             * @tparam Its Iterator types of right hand side
00699             * @tparam IsRandomAccessible SFINAE guard, do not specify
00700             * @param other right hand side
00701             * @return integer n such that (other + n) == *this
00702             */
00703            template<typename Its, bool IsRandomAccessible =
```

```
      traits::is_random_accessible_v<Iterators>, REQUIRES(
00704                     ZipIterator::minDifference(INSTANCE_OF(Iterators), INSTANCE_OF(Its)))>
00705             constexpr auto operator-(const ZipIterator<Its> &other) const
00706             -> std::enable_if_t<IsRandomAccessible, difference_type> {
00707                 return minDifference(iterators, other.getIterators());
00708             }
00709
00710             /**
00711              * Pairwise less comparison of underlying iterators
00712              * Only available if all underlying iterators support at least random access
00713              * @tparam Its Iterator types of right hand side
00714              * @tparam IsRandomAccessible SFINAE guard, do not specify
00715              * @param other right hand side
00716              * @return true if all underlying iterators compare less to the corresponding iterators
      from other
00717              */
00718             template<typename Its, bool IsRandomAccessible =
      traits::is_random_accessible_v<Iterators>, REQUIRES(
00719                     ZipIterator::allLess(INSTANCE_OF(Iterators), INSTANCE_OF(Its)))>
00720             constexpr auto operator<(const ZipIterator<Its> &other) const
00721             noexcept(noexcept(ZipIterator::allLess(INSTANCE_OF(Iterators), INSTANCE_OF(Its))))
00722             -> std::enable_if_t<IsRandomAccessible, bool> {
00723                 return allLess(iterators, other.getIterators());
00724             }
00725
00726             /**
00727              * Pairwise grater comparison of underlying iterators
00728              * Only available if all underlying iterators support at least random access
00729              * @tparam Its Iterator types of right hand side
00730              * @tparam IsRandomAccessible SFINAE guard, do not specify
00731              * @param other right hand side
00732              * @return true if all underlying iterators compare greater to the corresponding iterators
      from other
00733              */
00734             template<typename Its, bool IsRandomAccessible =
      traits::is_random_accessible_v<Iterators>, REQUIRES(
00735                     ZipIterator::allGreater(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
00736             constexpr auto operator>(const ZipIterator<Its> &other) const
      noexcept(noexcept(ZipIterator::allGreater(
00737                     INSTANCE_OF(Iterators), INSTANCE_OF(Its)))) ->
      std::enable_if_t<IsRandomAccessible, bool> {
00738                 return allGreater(iterators, other.getIterators());
00739             }
00740
00741             ///@}
00742
00743             /**
00744              * Pairwise equality comparison of underlying iterators
00745              * @tparam Its Iterator types of right hand side
00746              * @param other right hand side
00747              * @return true if at least one underlying iterator compares equal to the corresponding
      iterator from other
00748              */
00749             template<typename Its, REQUIRES(ZipIterator::oneEqual(INSTANCE_OF(Iterators),
      INSTANCE_OF(Its)))>
00750             constexpr bool operator==(const ZipIterator<Its> &other) const
00751             noexcept(noexcept(ZipIterator::oneEqual(std::declval<Iterators>(), other.getIterators())))
      {
00752                 return oneEqual(iterators, other.getIterators());
00753             }
00754
00755             /**
00756              * Dereferences all underlying iterators and returns a tuple of the resulting range
      reference types
00757              * @tparam Its SFINAE guard, do not specify
00758              * @return tuple of references to range elements
00759              */
00760             template<typename Its = Iterators, typename =
      std::enable_if_t<traits::is_dereferencible_v<Its>>>
00761             constexpr reference operator*() const
      noexcept(traits::is_nothrow_dereferencible_v<Iterators>) {
00762                 return std::apply([](auto &&...it) { return reference(*it...); }, iterators);
00763             }
00764
00765             /**
00766              * Getter for underlying iterators
00767              * @return Const reference to underlying iterators
00768              */
00769             constexpr auto getIterators() const noexcept -> const Iterators& {
00770                 return iterators;
00771             }
00772         };
00773
00774         /**
00775          * @brief Zip-view that provides begin() and end() member functions. Use to loop over multiple
      ranges at the
00776          * same time using ranged based for-loops.
```

```
00777          * @details @copybrief
00778          * Ranges are captured by lvalue reference, no copying occurs. Temporaries are allowed as well
    in which case
00779          * storage is moved into the zip-view.
00780          * @tparam Iterable Underlying range types
00781          */
00782         template<typename ...Iterable>
00783         struct ZipView DERIVE_VIEW_INTERFACE(ZipView<Iterable...>) {
00784         private:
00785             using ContainerTuple = std::tuple<Iterable...>;
00786             template<bool Const>
00787             using Iterators = std::tuple<decltype(std::begin(
00788                     std::declval<std::add_lvalue_reference_t<traits::const_if_t<Const,
    std::remove_reference_t<Iterable>>>()))...>;
00789             template<bool Const>
00790             using Sentinels = std::tuple<decltype(std::end(
00791                     std::declval<std::add_lvalue_reference_t<traits::const_if_t<Const,
    std::remove_reference_t<Iterable>>>()))...>;
00792             using IteratorTuple = Iterators<false>;
00793             using SentinelTuple = Sentinels<false>;
00794 
00795             template<typename Tuple, std::size_t ...Idx>
00796             constexpr auto sizeImpl(const Tuple &contTuple, std::index_sequence<Idx...>) const {
00797                 return std::min({std::size(std::get<Idx>(contTuple))...});
00798             }
00799         public:
00800             /**
00801              * CTor. Binds reference to ranges or takes ownership in case of rvalue references
00802              * @tparam Container range types
00803              * @param containers arbitrary number of ranges
00804              */
00805             template<typename ...Container>
00806             constexpr explicit ZipView(Container &&...containers) :
00807                 containers(std::forward<Container>(containers)...) {}
00808 
00809             ZipView() = default;
00810 
00811 
00812             /**
00813             * Returns a ZipIterator to the first elements of the underlying ranges
00814             * @return ZipIterator created by invoking std::begin on all underlying ranges
00815             */
00816             constexpr auto begin() {
00817                 return ZipIterator<IteratorTuple>(
00818                         std::apply([](auto &&...c) { return IteratorTuple(std::begin(c)...); },
    containers));
00819             }
00820 
00821             /**
00822             * Returns a ZipIterator to the elements following the last elements of the the underlying
    ranges
00823             * @return ZipIterator created by invoking std::end on all underlying ranges
00824             */
00825             constexpr auto end() {
00826                 return ZipIterator<SentinelTuple>(
00827                         std::apply([](auto &&...c) { return SentinelTuple(std::end(c)...); },
    containers));
00828             }
00829 
00830             /**
00831              * @copydoc ZipView::begin()
00832              * @note returns a ZipIterator that does not allow changing the ranges' elements
00833              */
00834             template<bool C = true>
00835             constexpr auto begin() const -> ZipIterator<Iterators<C>> {
00836                 return ZipIterator<Iterators<true>>(
00837                         std::apply([](auto &&...c) { return Iterators<true>(std::begin(c)...); },
    containers));
00838             }
00839 
00840             /**
00841              * @copydoc ZipView::end()
00842              */
00843             template<bool C = true>
00844             constexpr auto end() const -> ZipIterator<Sentinels<C>> {
00845                 return ZipIterator<Sentinels<true>>(
00846                         std::apply([](auto &&...c) { return Sentinels<true>(std::end(c)...); },
    containers));
00847             }
00848 
00849 #ifndef __USE_VIEW_INTERFACE__
00850             /**
00851              * Array subscript operator (no bounds are checked)
00852              * @tparam IsRandomAccess SFINAE helper, do not specify explicitly
00853              * @param index index
00854              * @return zip view element at given index
00855              */
```

```
00856             template<bool IsRandomAccess = traits::is_random_accessible_v<IteratorTuple>,
00857                 typename = std::enable_if_t<IsRandomAccess>>
00858             constexpr auto operator[](std::size_t index) {
00859                 return begin()[index];
00860             }
00861 
00862             /**
00863              * @copydoc ZipView::operator[](std::size_t index)
00864              */
00865             template<bool C = true, bool IsRandomAccess = traits::is_random_accessible_v<Iterators<C>>,
00866                 typename = std::enable_if_t<IsRandomAccess>>
00867             constexpr auto operator[](std::size_t index) const {
00868                 return begin()[index];
00869             }
00870 
00871 
00872             /**
00873              * Returns the smallest size of all containers. Only available if all containers know
     their size
00874              * @tparam HasSize SFINAE guard, do not specify explicitly
00875              * @return smallest size of all containers
00876              */
00877             template<bool HasSize = traits::has_size_v<ContainerTuple>>
00878             constexpr auto size() const -> std::enable_if_t<HasSize, std::size_t> {
00879                 return sizeImpl(containers,
     std::make_index_sequence<std::tuple_size_v<ContainerTuple>>());
00880             }
00881 
00882 #endif
00883 
00884         private:
00885             ContainerTuple containers;
00886         };
00887 
00888         /**
00889          * @brief represents the unreachable end of an infinite sequence
00890          */
00891         struct Unreachable {};
00892 
00893         /**
00894          * Signum function
00895          * @tparam T arbitrary scalar type
00896          * @param val function argument
00897          * @return +1 if val >= 0, -1 else
00898          */
00899         template<typename T>
00900         constexpr T sgn(T val) noexcept {
00901             return val < 0 ? T(-1) : T(1);
00902         }
00903 
00904         /**
00905          * @brief Iterator of an infinite sequence of numbers. Simply increments an internal counter
00906          * @tparam Type of the counter (most of the time this is ```std::size_t"')
00907          */
00908         template<typename T = std::size_t>
00909         struct CounterIterator : public SynthesizedOperators<CounterIterator<T>> {
00910             using value_type = T;
00911             using reference = T;
00912             using pointer = void;
00913             using iterator_category = std::random_access_iterator_tag;
00914             using difference_type = std::ptrdiff_t;
00915             static_assert(std::is_integral_v<T> && !std::is_floating_point_v<T>);
00916 
00917             using SynthesizedOperators<CounterIterator<T>>::operator++;
00918             using SynthesizedOperators<CounterIterator<T>>::operator--;
00919 
00920             /**
00921              * CTor.
00922              * @param begin start of number sequence
00923              * @param increment step size (default is 1)
00924              * @note Depending on the template type T, increment can also be negative.
00925              */
00926             explicit constexpr CounterIterator(T begin, T increment = T(1)) noexcept:
00927                     counter(begin), increment(increment) {}
00928 
00929             constexpr CounterIterator() noexcept: CounterIterator(T(0)) {}
00930 
00931             /**
00932              * Increments value by increment
00933              * @return reference to this
00934              */
00935             constexpr CounterIterator &operator++() noexcept {
00936                 counter += increment;
00937                 return *this;
00938             }
00939 
00940             /**
```

```
00941             * Decrements value by increment
00942             * @return reference to this
00943             */
00944            constexpr CounterIterator &operator--() noexcept {
00945                counter -= increment;
00946                return *this;
00947            }
00948
00949            /**
00950             * Compound assignment increment. Increments value by n times increment
00951             * @param n number of steps
00952             * @return reference to this
00953             */
00954            constexpr CounterIterator &operator+=(difference_type n) noexcept {
00955                counter += n * increment;
00956                return *this;
00957            }
00958
00959            /**
00960             * Compound assignment decrement. Increments value by n times increment
00961             * @param n number of steps
00962             * @return reference to this
00963             */
00964            constexpr CounterIterator &operator-=(difference_type n) noexcept {
00965                counter -= n * increment;
00966                return *this;
00967            }
00968
00969            /**
00970             * Difference between two CounterIterators
00971             * @param other right hand side
00972             * @return integer ```n"' with the smallest possible absolute value such that ```other + n
      <= *this"'
00973             * @note When other has the same increment as ```*this"', then the returned value is
      guaranteed to
00974             * fulfil ```other + n == *this"'. In the following example, this is not the case:
00975             * ```
00976             * CounterIterator a(8, 1);
00977             * CounterIterator b(4, 3);
00978             * auto diff = a - b; // yields 1 since b + 1 <= a
00979             * ```
00980             */
00981            constexpr difference_type operator-(const CounterIterator &other) const noexcept {
00982                return static_cast<difference_type>((counter - other.counter) / other.increment);
00983            }
00984
00985            /**
00986             * Equality comparison.
00987             * @param other right hand side
00988             * @return true if counter of left and right hand side are equal
00989             */
00990            constexpr bool operator==(const CounterIterator &other) const noexcept {
00991                return counter == other.counter;
00992            }
00993
00994            /**
00995             * Equality comparison with Unreachable sentinel
00996             * @return false
00997             */
00998            friend constexpr bool operator==(const CounterIterator &, Unreachable) noexcept {
00999                return false;
01000            }
01001
01002            /**
01003             * @copydoc operator==(const CounterIterator &, Unreachable)
01004             */
01005            friend constexpr bool operator==(Unreachable, const CounterIterator &) noexcept {
01006                return false;
01007            }
01008
01009            friend constexpr bool operator!=(Unreachable, const CounterIterator &) noexcept {
01010                return true;
01011            }
01012
01013            /**
01014             * Less comparison of internal counters with respect to increment of this instance
01015             * @param other right hand side
01016             * @return true if
01017             * ```
01018             * sgn(increment) **this < *other sgn(increment)
01019             * ```
01020             * where ```sgn"' is the signum
01021             * function
01022             * @note If increment is negative then both sides of the inequality are multiplied with
      -1.
01023             * For example: let ```it1 = 5"' and ```it2 = -2"' be two CounterIterators where ```it1"'
      has negative
```

```
01024             * increment. Then ```it1 < it2"' is true.
01025             */
01026            constexpr bool operator<(const CounterIterator &other) const noexcept {
01027                return sgn(increment) *  counter < sgn(increment) * other.counter;
01028            }
01029
01030            /**
01031             * Greater comparison of internal counters with respect to increment of this instance
01032             * @param other right hand side
01033             * @return true if
01034             * ```
01035             * sgn(increment) **this > *other sgn(increment)
01036             * ```
01037             * where ```sgn"' is the signum
01038             * @note If increment is negative then both sides of the inequality are multiplied with
     -1.
01039             * For example: let ```it1 = 5"' and ```it2 = -2"' be two CounterIterators where ```it1"'
     has negative
01040             * increment. Then ```it1 > it2"' is false.
01041             */
01042            constexpr bool operator>(const CounterIterator &other) const noexcept {
01043                return sgn(increment) * counter > sgn(increment) * other.counter;
01044            }
01045
01046            /**
01047             * Produces the counter value
01048             * @return value of internal counter
01049             */
01050            constexpr T operator*() const noexcept {
01051                return counter;
01052            }
01053
01054        private:
01055            T counter;
01056            T increment;
01057        };
01058
01059        /**
01060         * @brief Represents an infinite range of numbers
01061         * @tparam T type of number range
01062         */
01063        template<typename T = std::size_t>
01064        struct CounterRange {
01065            /**
01066             * CTor
01067             * @param start start of the range
01068             * @param increment step size
01069             * @note Depending on the template type T, increment can also be negative.
01070             */
01071            explicit constexpr CounterRange(T start = T(0), T increment = T(1)) noexcept:
01072                start(start), increment(increment) {}
01073
01074            /**
01075             * @return CounterIterator representing the beginning of the sequence
01076             */
01077            [[nodiscard]] constexpr CounterIterator<T> begin() const noexcept {
01078                return CounterIterator<T>(start, increment);
01079            }
01080
01081            /**
01082             * @return Sentinel object representing the unreachable end of the sequence
01083             */
01084            [[nodiscard]] static constexpr Unreachable end() noexcept {
01085                return Unreachable{};
01086            }
01087
01088        private:
01089            T start;
01090            T increment;
01091        };
01092    }
01093
01094    /**
01095     * Function that is used to create a impl::ZipIterator from an arbitrary number of iterators
01096     * @tparam Iterators type of iterators
01097     * @param iterators arbitrary number of iterators
01098     * @return impl::ZipIterator
01099     * @note ZipIterators have the same iterator category as the least powerful underlying operator.
     This means that
01100     * for example, zipping a random access iterator and a bidirectional iterator only yields a
     bidirectional
01101     * impl::ZipIterator
01102     * @relatesalso impl::ZipIterator
01103     */
01104    template<typename ...Iterators>
01105    constexpr auto zip_i(Iterators ...iterators) -> impl::ZipIterator<std::tuple<Iterators...>> {
01106        using IteratorTuple = std::tuple<Iterators...>;
```

```
01107         return impl::ZipIterator<IteratorTuple>(IteratorTuple(std::move(iterators)...));
01108     }
01109
01110     /**
01111      * Function that can be used in range based loops to emulate the zip iterator from python.
01112      * As in python: if the passed containers have different lengths, the container with the least
      items decides
01113      * the overall range
01114      * @tparam Iterable Container types that support iteration
01115      * @param iterable Arbitrary number of containers
01116      * @return impl::ZipView class that provides begin and end members to be used in range based
      for-loops
01117      * @relatesalso impl::ZipView
01118      */
01119     template<typename ...Iterable>
01120     constexpr auto zip(Iterable &&...iterable) {
01121         return impl::ZipView<Iterable...>(std::forward<Iterable>(iterable)...);
01122     }
01123
01124     /**
01125      * Zip variant that does not allow manipulation of the container elements
01126      *
01127      * @copydoc zip
01128      */
01129     template<typename ...Iterable>
01130     constexpr auto const_zip(Iterable &&...iterable) {
01131         return impl::ZipView<impl::traits::reference_if_t<std::is_lvalue_reference_v<Iterable>,
01132
      std::add_const_t<std::remove_reference_t<Iterable»>...>(std::forward<Iterable>(iterable)...);
01133     }
01134
01135     namespace impl{
01136         template<typename TZip, typename ...Iterable>
01137         constexpr auto zip_enumerate_impl(TZip &&tZip, Iterable &&...iterable) {
01138             if constexpr (sizeof...(Iterable) == 0 ||
      !(std::is_integral_v<std::remove_reference_t<Iterable» || ...)) {
01139                 return std::forward<TZip>(tZip)(impl::CounterRange(0ul, 1ul),
      std::forward<Iterable>(iterable)...);
01140             } else {
01141                 constexpr auto CtrRangeArgsIdx = impl::index_of<std::is_integral, 0,
01142
      std::remove_reference_t<Iterable>...>();
01143                 static_assert(CtrRangeArgsIdx >= sizeof...(Iterable) - 2);
01144                 auto [its, enumArgs] = impl::tuple_split<CtrRangeArgsIdx>(
01145                         std::forward_as_tuple(std::forward<Iterable>(iterable)...));
01146                 return std::apply(std::forward<TZip>(tZip), std::tuple_cat(std::make_tuple(
01147                         std::make_from_tuple<impl::CounterRange<
01148                                 std::remove_reference_t<std::tuple_element_t<0, decltype(enumArgs)»»(
01149                                 enumArgs)), std::move(its)));
01150             }
01151         }
01152     }
01153
01154     /**
01155      * Function that can be used in range based loops to emulate the enumerate iterator from python.
01156      * @tparam Container Container type that supports iteration
01157      * @tparam T type of enumerate counter (default std::size_t)
01158      * @param container Source container
01159      * @param start Optional index offset (default 0)
01160      * @param increment Optional index increment (default 1)
01161      * @return impl::ZipView that provides begin and end members to be used in range based for-loops.
01162      * @relatesalso impl::ZipView
01163      */
01164     template<typename Container, typename T = std::size_t>
01165     constexpr auto enumerate(Container &&container, T start = T(0), T increment = T(1)) {
01166         return zip(impl::CounterRange(start, increment), std::forward<Container>(container));
01167     }
01168
01169     /**
01170      * enumerate variant that does not allow manipulation of the container elements
01171      *
01172      * @copydoc enumerate
01173      */
01174     template<typename Container, typename T = std::size_t>
01175     constexpr auto const_enumerate(Container &&container, T start = T(0), T increment = T(1)) {
01176         return const_zip(impl::CounterRange(start, increment), std::forward<Container>(container));
01177     }
01178
01179     /**
01180      * combination of zip and enumerate, i.e. returns an impl::ZipView that contains an enumerator at
      the first position
01181      * @tparam Iterable Types of arguments
01182      * @param iterable arbitrary number of iterables followed by optionally a start and an increment
01183      * @return impl::ZipView with prepended enumerator
01184      */
01185     template<typename ...Iterable>
01186     constexpr auto zip_enumerate(Iterable &&...iterable) {
```

```
01187         return impl::zip_enumerate_impl([](auto &&... args) { return
      zip(std::forward<decltype(args)>(args)...); },
01188                                         std::forward<Iterable>(iterable)...);
01189     }
01190
01191     /**
01192      * zip_enumerate variant that does not allow manipulation of the container elements
01193      *
01194      * @copydoc zip_enumerate
01195      */
01196     template<typename ...Iterable>
01197     constexpr auto const_zip_enumerate(Iterable &&...iterable) {
01198         return impl::zip_enumerate_impl([](auto &&... args) { return
      const_zip(std::forward<decltype(args)>(args)...); },
01199                                         std::forward<Iterable>(iterable)...);
01200     }
01201
01202 }
01203
01204 namespace std {
01205
01206     template<typename ...Ts>
01207     struct tuple_size<iterators::impl::RefTuple < Ts...» : integral_constant<std::size_t,
      sizeof...(Ts)> {};
01208
01209     template<std::size_t Idx, typename ...Ts>
01210     struct tuple_element<Idx, iterators::impl::RefTuple<Ts...» {
01211         using type = std::tuple_element_t<Idx, std::tuple<Ts...»;
01212     };
01213 }
01214
01215 #endif //ITERATORTOOLS_ITERATORS_HPP
```

Index