

zip and enumerate iterators

Generated by Doxygen 1.9.1

1 Main Page	2
1.1 Python-like Zip and Enumerate Iterators	2
1.1.1 Properties	2
1.1.2 Code Examples	2
1.1.3 Doxygen Documentation	3
2 Namespace Documentation	3
2.1 iterators Namespace Reference	3
2.1.1 Detailed Description	4
2.1.2 Function Documentation	4
2.2 iterators::impl Namespace Reference	8
2.2.1 Detailed Description	9
2.2.2 Function Documentation	9
2.3 iterators::impl::traits Namespace Reference	11
2.3.1 Detailed Description	13
3 Class Documentation	13
3.1 iterators::impl::CounterIterator< T > Struct Template Reference	13
3.1.1 Detailed Description	15
3.1.2 Constructor & Destructor Documentation	15
3.1.3 Member Function Documentation	15
3.1.4 Friends And Related Function Documentation	21
3.2 iterators::impl::CounterRange< T > Struct Template Reference	21
3.2.1 Detailed Description	22
3.2.2 Constructor & Destructor Documentation	22
3.2.3 Member Function Documentation	22
3.3 iterators::impl::SynthesizedOperators< Impl > Struct Template Reference	23
3.3.1 Detailed Description	24
3.3.2 Member Function Documentation	24
3.3.3 Friends And Related Function Documentation	27
3.4 iterators::impl::Unreachable Struct Reference	28
3.4.1 Detailed Description	28
3.5 iterators::impl::ZipIterator< Iterators > Class Template Reference	29
3.5.1 Detailed Description	30
3.5.2 Member Function Documentation	32
3.5.3 Friends And Related Function Documentation	38
4 File Documentation	39
4.1 Iterators.hpp File Reference	39
4.1.1 Detailed Description	41
4.1.2 Macro Definition Documentation	42
Index	45

1 Main Page

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 `enumerate`-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 `ZipIterators` 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

- [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

2.1.2.1 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

<i>Container</i>	Container type that supports iteration
<i>T</i>	type of enumerate counter (default std::size_t)

Parameters

<i>container</i>	Source container
<i>start</i>	Optional index offset (default 0)
<i>increment</i>	Optional index increment (default 1)

Returns

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

```
2.1.2.2 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

<i>Iterable</i>	Container types that support iteration
-----------------	--

Parameters

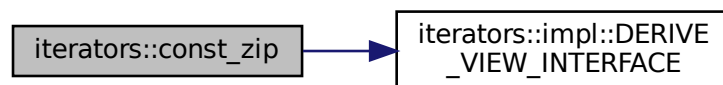
<i>iterable</i>	Arbitrary number of containers
-----------------	--------------------------------

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:



```
2.1.2.3 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

<i>Iterable</i>	Types of arguments
-----------------	--------------------

Parameters

<i>iterable</i>	arbitrary number of iterables followed by optionally a start and an increment
-----------------	---

Returns

impl::ZipView with prepended enumerator

```
2.1.2.4 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

<i>Container</i>	Container type that supports iteration
<i>T</i>	type of enumerate counter (default std::size_t)

Parameters

<i>container</i>	Source container
<i>start</i>	Optional index offset (default 0)
<i>increment</i>	Optional index increment (default 1)

Returns

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

```
2.1.2.5 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

<i>Iterable</i>	Container types that support iteration
-----------------	--

Parameters

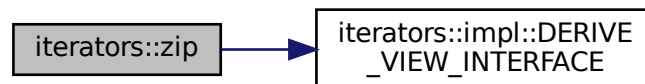
<i>iterable</i>	Arbitrary number of containers
-----------------	--------------------------------

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:



2.1.2.6 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

<i>Iterable</i>	Types of arguments
-----------------	--------------------

Parameters

<i>iterable</i>	arbitrary number of iterables followed by optionally a start and an increment
-----------------	---

Returns

impl::ZipView with prepended enumerator


```
2.1.2.7 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

<i>Iterators</i>	type of iterators
------------------	-------------------

Parameters

<i>iterators</i>	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

- [traits](#)

namespace containing type traits used in implementation of zip and enumerate

Classes

- struct [SynthesizedOperators](#)
CRTP-class that provides additional pointer arithmetic operators synthesized from basic operators.
- class [ZipIterator](#)
Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.
- struct [Unreachable](#)
represents the unreachable end of an infinite sequence
- struct [CounterIterator](#)
Iterator of an infinite sequence of numbers. Simply increments an internal counter.
- struct [CounterRange](#)
Represents an infinite range of numbers.

Functions

- `template<std::size_t Offset, std::size_t ... Ldx>`
`constexpr auto index_seq_impl (std::index_sequence< Ldx... >) noexcept`
- `template<std::size_t Start, std::size_t End>`
`constexpr auto index_seq () noexcept`
- `template<typename Tuple , std::size_t ... Ldx1, std::size_t ... Ldx2>`
`constexpr auto tuple_split_impl (Tuple &&tuple, std::index_sequence< Ldx1... >, std::index_sequence< Ldx2... >)`
- `template<std::size_t Ldx, typename Tuple >`
`constexpr auto tuple_split (Tuple &&tuple)`
- `template<template< typename > typename Predicate, std::size_t Ldx, typename ... Ts>`
`constexpr std::size_t index_of () noexcept`
- `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`
- `template<typename TZip , typename ... Iterable>`
`constexpr auto zip_enumerate_impl (TZip &&tZip, Iterable &&...iterable)`

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

2.2.2.1 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

<i>Iterable</i>	Underlying range types
-----------------	------------------------

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

Template Parameters

<i>Container</i>	range types
------------------	-------------

Parameters

<i>containers</i>	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

<i>IsRandomAccess</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>index</i>	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

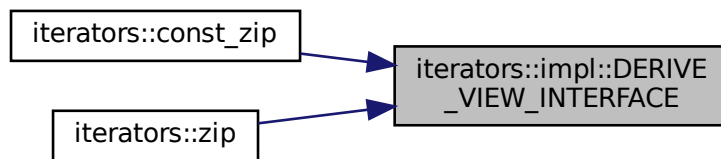
<i>HasSize</i>	SFINAE guard, do not specify explicitly
----------------	---

Returns

smallest size of all containers

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

Here is the caller graph for this function:



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

Signum function

Template Parameters

<i>T</i>	arbitrary scalar type
----------	-----------------------

Parameters

<i>val</i>	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

Classes

- struct **is_container**
- struct **is_container**< T, std::void_t< decltype(std::begin(std::declval< T >()), std::end(std::declval< T >()))> >
- struct **is_dereferencible**
- struct **is_dereferencible**< T, std::void_t< decltype(*std::declval< T >())> >
- struct **is_dereferencible**< std::tuple< Ts... >, void >
- struct **is_incrementable**
- struct **is_incrementable**< T, std::void_t< decltype(++REFERENCE(T))> >

- struct **is_incrementable**< std::tuple< Ts... >, void >
- struct **dereference**
- struct **dereference**< T, true >
- struct **values**
- struct **values**< std::tuple< Ts... > >
- struct **references**
- struct **references**< std::tuple< Ts... > >
- struct **iterator_category_value**
- struct **iterator_category_value**< T, std::void_t< typename std::iterator_traits< T >::iterator_category > >
- struct **iterator_category_from_value**
- struct **iterator_category_from_value**< 0 >
- struct **minimum_category**
- struct **minimum_category**< std::tuple< Ts... > >
- struct **is_random_accessible**
- struct **is_random_accessible**< T, std::void_t< typename std::iterator_traits< T >::iterator_category > >
- struct **is_random_accessible**< std::tuple< Ts... >, std::void_t< value_to_type_t< minimum_category_v< std::tuple< Ts... > > > > >
- struct **is_bidirectional**
- struct **is_bidirectional**< T, std::void_t< typename std::iterator_traits< T >::iterator_category > >
- struct **is_bidirectional**< std::tuple< Ts... >, std::void_t< value_to_type_t< minimum_category_v< std::tuple< Ts... > > > > >
- struct **has_size**
- struct **has_size**< T, std::void_t< decltype(std::size(std::declval< std::remove_reference_t< T >>())) > >
- struct **has_size**< std::tuple< Ts... > >

Typedefs

- template<bool Cond, typename T >
using **reference_if_t** = std::conditional_t< Cond, std::add_lvalue_reference_t< T >, T >
- template<bool Cond, typename T >
using **const_if_t** = std::conditional_t< Cond, std::add_const_t< T >, T >
- template<typename T >
using **dereference_t** = typename dereference< T, is_dereferencible_v< T > >::type
- template<typename T >
using **values_t** = typename values< T >::type
- template<typename T >
using **references_t** = typename references< T >::type

Variables

- template<typename T >
constexpr bool **is_container_v** = is_container<T>::value
- template<typename T >
constexpr bool **is_dereferencible_v** = is_dereferencible<T>::value
- template<typename T >
constexpr bool **is_incrementable_v** = is_incrementable<T>::value
- template<typename T >
constexpr std::size_t **minimum_category_v** = minimum_category<T>::value
- template<typename T >
constexpr bool **is_random_accessible_v** = is_random_accessible<T>::value
- template<typename T >
constexpr bool **is_bidirectional_v** = is_bidirectional<T>::value
- template<typename T >
constexpr bool **has_size_v** = has_size<T>::value

2.3.1 Detailed Description

namespace containing type traits used in implementation of zip and enumerate

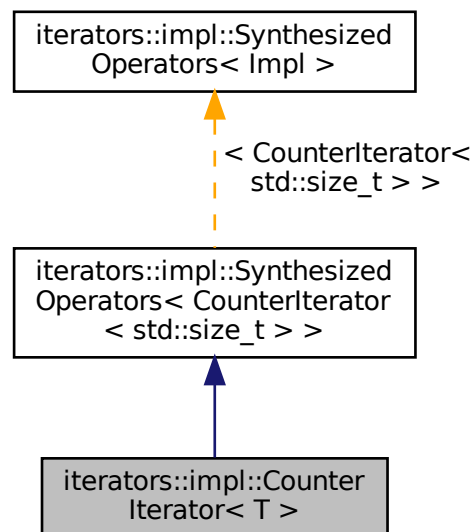
3 Class Documentation

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

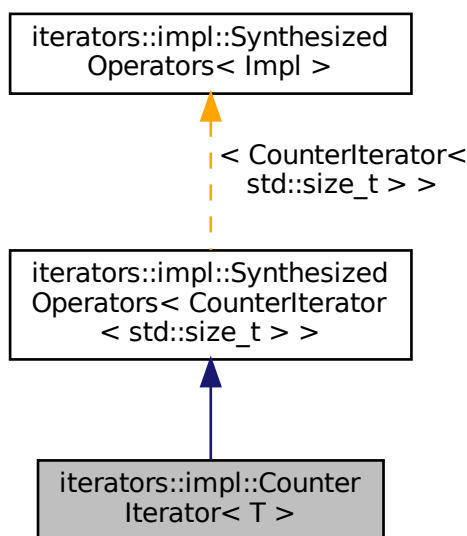
Iterator of an infinite sequence of numbers. Simply increments an internal counter.

```
#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
- constexpr auto `operator[]` (typename Implementation::difference_type n) const noexcept(noexcept(*(std::declval< `CounterIterator`< std::size_t > >()+n)))
- constexpr `CounterIterator`< std::size_t > `operator++` (int) noexcept(noexcept(++std::declval< `CounterIterator`< std::size_t > >()) &&std::is_nothrow_copy_constructible_v< `CounterIterator`< std::size_t > >)
- constexpr `CounterIterator`< std::size_t > `operator--` (int) noexcept(noexcept(--std::declval< `CounterIterator`< std::size_t > >()) &&std::is_nothrow_copy_constructible_v< `CounterIterator`< std::size_t > >)
- 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))

Friends

- constexpr friend bool `operator==` (const `CounterIterator` &, `Unreachable`) noexcept
- constexpr friend bool `operator==` (`Unreachable`, const `CounterIterator` &) noexcept
- constexpr friend 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 <code>std::size_t</code>)
------	---

3.1.2 Constructor & Destructor Documentation

3.1.2.1 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

<i>begin</i>	start of number sequence
<i>increment</i>	step size (default is 1)

Note

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

3.1.3 Member Function Documentation

3.1.3.1 operator!=(=) `constexpr bool iterators::impl::SynthesizedOperators< CounterIterator<`
`std::size_t > >::operator!=` (
 const T & *other*) const `[inline], [constexpr], [noexcept], [inherited]`

Inequality comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not equal to other

3.1.3.2 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

3.1.3.3 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

3.1.3.4 operator++() `[2/2] constexpr CounterIterator< std::size_t > iterators::impl::SynthesizedOperators<
CounterIterator< std::size_t > >::operator++ (
int) [inline], [constexpr], [noexcept], [inherited]`

Postfix increment. Synthesized from prefix increment

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.1.3.5 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

<i>n</i>	number of steps
----------	-----------------

Returns

reference to this

3.1.3.6 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

<i>other</i>	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
```

3.1.3.7 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

3.1.3.8 operator--() [2/2] constexpr CounterIterator< std::size_t > iterators::impl::SynthesizedOperators< CounterIterator< std::size_t > >::operator-- (
int) [inline], [constexpr], [noexcept], [inherited]

Postfix decrement. Synthesized from prefix decrement

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.1.3.9 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

<i>n</i>	number of steps
----------	-----------------

Returns

reference to this

3.1.3.10 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

<i>other</i>	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.

3.1.3.11 operator<=() constexpr bool `iterators::impl::SynthesizedOperators< CounterIterator< std::size_t > >::operator<=` (
 const T & *rhs*) const [inline], [constexpr], [noexcept], [inherited]

Less than or equal comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not greater than other

3.1.3.12 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

<i>other</i>	right hand side
--------------	-----------------

Returns

true if counter of left and right hand side are equal

```

3.1.3.13 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

<i>other</i>	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.

```

3.1.3.14 operator>=() constexpr bool iterators::impl::SynthesizedOperators< CounterIterator<
std::size_t > >::operator>= (
    const T & rhs ) const [inline], [constexpr], [noexcept], [inherited]

```

Greater than or equal comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not less than other

```

3.1.3.15 operator[]() constexpr auto iterators::impl::SynthesizedOperators< CounterIterator<
std::size_t > >::operator[] (
    typename Implementation::difference_type n ) const [inline], [constexpr], [noexcept],
[inherited]

```

Array subscript operator

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>n</i>	index
----------	-------

Returns

*(*this* + *n*)

3.1.4 Friends And Related Function Documentation

3.1.4.1 operator== [1/2] `template<typename T = std::size_t>`
`constexpr friend bool operator== (`
`const CounterIterator< T > & ,`
`Unreachable) [friend]`

Equality comparison with [Unreachable](#) sentinel

Returns

false

3.1.4.2 operator== [2/2] `template<typename T = std::size_t>`
`constexpr friend 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

<i>T</i>	type of number range
----------	----------------------

3.2.2 Constructor & Destructor Documentation

3.2.2.1 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

<i>start</i>	start of the range
<i>increment</i>	step size

Note

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

3.2.3 Member Function Documentation

3.2.3.1 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

3.2.3.2 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:

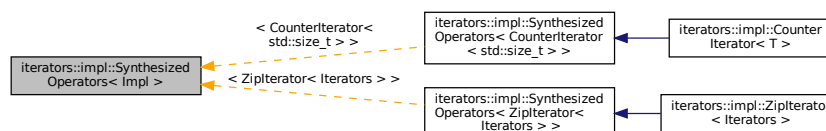
- [Iterators.hpp](#)

3.3 iterators::impl::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 auto operator[] (typename Implementation::difference_type n) const 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 friend 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 friend 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 friend auto operator- (Impl it, typename Implementation::difference_type n) noexcept(noexcept(std::declval< Impl >()-=n))`

3.3.1 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

<i>Impl</i>	Base class
-------------	------------

3.3.2 Member Function Documentation

3.3.2.1 `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

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not equal to other

3.3.2.2 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

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.3.2.3 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

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.3.2.4 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

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not greater than other

```
3.3.2.5 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

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not less than other

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

Array subscript operator

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>n</i>	index
----------	-------

Returns

$*(\text{this} + n)$

3.3.3 Friends And Related Function Documentation

3.3.3.1 operator+ [1/2] `template<typename Impl >`

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

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

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>it</i>	left hand side
<i>n</i>	right hand side

Returns

Instance of Impl

3.3.3.2 operator+ [2/2] `template<typename Impl >`

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

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

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>n</i>	left hand side
<i>it</i>	right hand side

Returns

Instance of Impl

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

Binary minus operator. Synthesized from compound assignment operator-=

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>it</i>	left hand side
<i>n</i>	right hand side

Returns

Instance of Impl

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

- [Iterators.hpp](#)

3.4 iterators::impl::Unreachable Struct Reference

represents the unreachable end of an infinite sequence

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

3.4.1 Detailed Description

represents the unreachable end of an infinite sequence

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

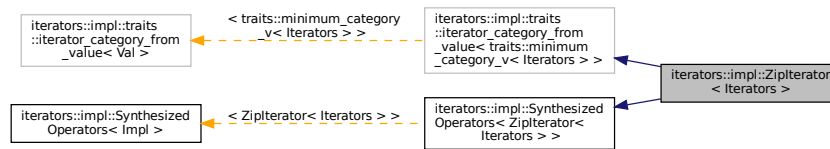
- [Iterators.hpp](#)

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

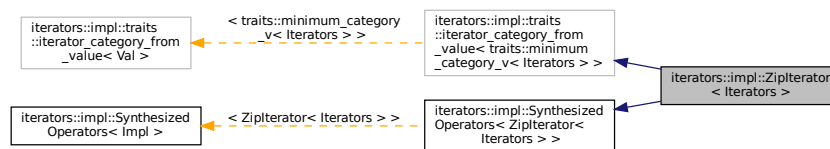
Class combining multiple iterators into one. Use it to iterate over multiple ranges at the same time.

```
#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>
constexpr **ZipIterator** (Its &&...its)
- template<typename Its = Iterators, typename = std::enable_if_t<traits::is_incrementable_v<Its>>>
constexpr **ZipIterator** & **operator++** () noexcept(traits::is_nothrow_incrementable_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::oneEqual(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 auto **operator[]** (typename Implementation::difference_type n) const 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_decrementable_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::min< Difference(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::all< Less(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
constexpr auto [operator<](#) (const [ZipIterator](#)< Its > &other) const noexcept(noexcept(ZipIterator::all< Less(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::all< Greater(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) >
constexpr auto [operator>](#) (const [ZipIterator](#)< Its > &other) const noexcept(noexcept(ZipIterator::all< Greater(INSTANCE_OF(Iterators), INSTANCE_OF(Its))) -> std::enable_if_t< IsRandomAccessible, bool >

Related Functions

(Note that these are not member functions.)

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

3.5.1 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 underlying 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

<i>Iterators</i>	Underlying iterator types
------------------	---------------------------

3.5.2 Member Function Documentation

3.5.2.1 `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

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

Inequality comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not equal to other

3.5.2.3 `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

<i>Its</i>	SFINAE guard, do not specify
------------	------------------------------

Returns

tuple of references to range elements

3.5.2.4 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

<i>Its</i>	SFINAE guard, do not specify
------------	------------------------------

Returns

reference to this

3.5.2.5 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

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.5.2.6 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

<i>IsRandomAccessible</i>	SFINAE guard, do not specify
---------------------------	------------------------------

Parameters

<i>n</i>	increment
----------	-----------

Returns

reference to this

```

3.5.2.7 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

<i>Its</i>	Iterator types of right hand side
<i>IsRandomAccessible</i>	SFINAE guard, do not specify

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

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

```

3.5.2.8 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

<i>IsBidirectional</i>	SFINAE guard, do not specify
------------------------	------------------------------

Returns

reference to this

3.5.2.9 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

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Returns

Instance of Impl

3.5.2.10 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

<i>IsRandomAccessible</i>	SFINAE guard, do not specify
---------------------------	------------------------------

Parameters

<i>n</i>	decrement
----------	-----------

Returns

reference to this

```

3.5.2.11 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

<i>Its</i>	Iterator types of right hand side
<i>IsRandomAccessible</i>	SFINAE guard, do not specify

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

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

```

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

```

Less than or equal comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not greater than other

```

3.5.2.13 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

<i>Its</i>	Iterator types of right hand side
------------	-----------------------------------

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

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

3.5.2.14 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 greater comparison of underlying iterators Only available if all underlying iterators support at least random access

Template Parameters

<i>Its</i>	Iterator types of right hand side
<i>IsRandomAccessible</i>	SFINAE guard, do not specify

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

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

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

Greater than or equal comparison

Template Parameters

<i>T</i>	type of right hand side
<i>Implementation</i>	SFINAE helper, do not specify explicitly

Parameters

<i>other</i>	right hand side
--------------	-----------------

Returns

true if this is not less than other

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

Array subscript operator

Template Parameters

<i>Implementation</i>	SFINAE helper, do not specify explicitly
-----------------------	--

Parameters

<i>n</i>	index
----------	-------

Returns

*(**this* + *n*)

3.5.3 Friends And Related Function Documentation

3.5.3.1 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

<i>Iterators</i>	type of iterators
------------------	-------------------

Parameters

<i>iterators</i>	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

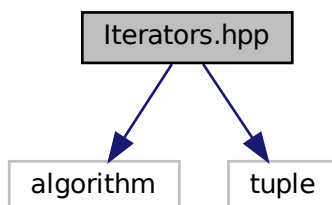
4.1 Iterators.hpp File Reference

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 <tuple>
```

Include dependency graph for Iterators.hpp:



Classes

- 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

- [iterators](#)

namespace containing zip and enumerate functions

- [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

- [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>(tuple1)`
- `#define ELEMENT2 std::get<Idx>(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, type-
name = std::void_t<decltype(EXPRESSION)>`
- `#define REQUIRES(EXPRESSION) typename = std::void_t<decltype(EXPRESSION)>`
- `#define DERIVE_VIEW_INTERFACE(CLASS)`

Typedefs

- `template<bool Cond, typename T >
using iterators::impl::traits::reference_if_t = std::conditional_t< Cond, std::add_lvalue_reference_t< T >,
T >`
- `template<bool Cond, typename T >
using iterators::impl::traits::const_if_t = std::conditional_t< Cond, std::add_const_t< T >, T >`
- `template<typename T >
using iterators::impl::traits::dereference_t = typename dereference< T, is_dereferencible_v< T > >::type`
- `template<typename T >
using iterators::impl::traits::values_t = typename values< T >::type`
- `template<typename T >
using iterators::impl::traits::references_t = typename references< T >::type`

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 Ldx, typename Tuple >`
`constexpr auto iterators::impl::tuple_split (Tuple &&tuple)`
- `template<template< typename > typename Predicate, std::size_t Ldx, typename ... Ts>`
`constexpr std::size_t iterators::impl::index_of () noexcept`
- `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 TZip, typename ... Iterable>`
`constexpr auto iterators::impl::zip_enumerate_impl (TZip &&tZip, 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)`

Variables

- `template<typename T >`
`constexpr bool iterators::impl::traits::is_container_v = is_container<T>::value`
- `template<typename T >`
`constexpr bool iterators::impl::traits::is_dereferencible_v = is_dereferencible<T>::value`
- `template<typename T >`
`constexpr bool iterators::impl::traits::is_incrementable_v = is_incrementable<T>::value`
- `template<typename T >`
`constexpr std::size_t iterators::impl::traits::minimum_category_v = minimum_category<T>::value`
- `template<typename T >`
`constexpr bool iterators::impl::traits::is_random_accessible_v = is_random_accessible<T>::value`
- `template<typename T >`
`constexpr bool iterators::impl::traits::is_bidirectional_v = is_bidirectional<T>::value`
- `template<typename T >`
`constexpr bool iterators::impl::traits::has_size_v = has_size<T>::value`

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

4.1.2.1 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;
```

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

4.1.2.3 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; \
};
```

4.1.2.4 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;
```

4.1.2.5 TYPE_MAP_DEFAULT #define TYPE_MAP_DEFAULT

Value:

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


Index

ALL_NOEXCEPT
 Iterators.hpp, 42

begin
 iterators::impl::CounterRange< T >, 22

BINARY_TUPLE_FOR_EACH
 Iterators.hpp, 42

const_enumerate
 iterators, 4

const_zip
 iterators, 5

const_zip_enumerate
 iterators, 5

CounterIterator
 iterators::impl::CounterIterator< T >, 15

CounterRange
 iterators::impl::CounterRange< T >, 22

DERIVE_VIEW_INTERFACE
 iterators::impl, 9

end
 iterators::impl::CounterRange< T >, 23

enumerate
 iterators, 6

getIterators
 iterators::impl::ZipIterator< Iterators >, 32

iterators, 3
 const_enumerate, 4
 const_zip, 5
 const_zip_enumerate, 5
 enumerate, 6
 zip, 6
 zip_enumerate, 7
 zip_i, 7

Iterators.hpp, 39
 ALL_NOEXCEPT, 42
 BINARY_TUPLE_FOR_EACH, 42
 TYPE_MAP, 42
 TYPE_MAP_ALIAS, 42
 TYPE_MAP_DEFAULT, 43

iterators::impl, 8
 DERIVE_VIEW_INTERFACE, 9
 sgn, 11

iterators::impl::CounterIterator< T >, 13
 CounterIterator, 15
 operator!=, 15
 operator<, 18
 operator<=, 19
 operator>, 19
 operator>=, 20
 operator*, 16
 operator++, 16
 operator+=, 17
 operator-, 17
 operator--, 17, 18
 operator==, 18
 operator==, 19, 21
 operator[], 20

iterators::impl::CounterRange< T >, 21
 begin, 22
 CounterRange, 22
 end, 23

iterators::impl::SynthesizedOperators< Impl >, 23
 operator!=, 24
 operator<=, 25
 operator>=, 26
 operator+, 27
 operator++, 25
 operator-, 28
 operator--, 25
 operator[], 26

iterators::impl::traits, 11

iterators::impl::Unreachable, 28

iterators::impl::ZipIterator< Iterators >, 29
 getIterators, 32
 operator!=, 32
 operator<, 35
 operator<=, 36
 operator>, 37
 operator>=, 37
 operator*, 32
 operator++, 33
 operator+=, 33
 operator-, 34
 operator--, 34, 35
 operator==, 35
 operator==, 36
 operator[], 38
 zip_i, 38

operator!=
 iterators::impl::CounterIterator< T >, 15
 iterators::impl::SynthesizedOperators< Impl >, 24
 iterators::impl::ZipIterator< Iterators >, 32

operator<
 iterators::impl::CounterIterator< T >, 18
 iterators::impl::ZipIterator< Iterators >, 35

operator<=
 iterators::impl::CounterIterator< T >, 19
 iterators::impl::SynthesizedOperators< Impl >, 25
 iterators::impl::ZipIterator< Iterators >, 36

operator>
 iterators::impl::CounterIterator< T >, 19
 iterators::impl::ZipIterator< Iterators >, 37

operator>=
 iterators::impl::CounterIterator< T >, 20
 iterators::impl::SynthesizedOperators< Impl >, 26
 iterators::impl::ZipIterator< Iterators >, 37

operator*
 iterators::impl::CounterIterator< T >, [16](#)
 iterators::impl::ZipIterator< Iterators >, [32](#)
operator+
 iterators::impl::SynthesizedOperators< Impl >, [27](#)
operator++
 iterators::impl::CounterIterator< T >, [16](#)
 iterators::impl::SynthesizedOperators< Impl >, [25](#)
 iterators::impl::ZipIterator< Iterators >, [33](#)
operator+=
 iterators::impl::CounterIterator< T >, [17](#)
 iterators::impl::ZipIterator< Iterators >, [33](#)
operator-
 iterators::impl::CounterIterator< T >, [17](#)
 iterators::impl::SynthesizedOperators< Impl >, [28](#)
 iterators::impl::ZipIterator< Iterators >, [34](#)
operator--
 iterators::impl::CounterIterator< T >, [17](#), [18](#)
 iterators::impl::SynthesizedOperators< Impl >, [25](#)
 iterators::impl::ZipIterator< Iterators >, [34](#), [35](#)
operator-=
 iterators::impl::CounterIterator< T >, [18](#)
 iterators::impl::ZipIterator< Iterators >, [35](#)
operator==
 iterators::impl::CounterIterator< T >, [19](#), [21](#)
 iterators::impl::ZipIterator< Iterators >, [36](#)
operator[]
 iterators::impl::CounterIterator< T >, [20](#)
 iterators::impl::SynthesizedOperators< Impl >, [26](#)
 iterators::impl::ZipIterator< Iterators >, [38](#)

sgn
 iterators::impl, [11](#)

TYPE_MAP
 Iterators.hpp, [42](#)
TYPE_MAP_ALIAS
 Iterators.hpp, [42](#)
TYPE_MAP_DEFAULT
 Iterators.hpp, [43](#)

zip
 iterators, [6](#)
zip_enumerate
 iterators, [7](#)
zip_i
 iterators, [7](#)
 iterators::impl::ZipIterator< Iterators >, [38](#)