

---

# Concept reference

Copyright © 2001, 2002 Indiana University  
Copyright © 2000, 2001 University of Notre Dame du Lac  
Copyright © 2000 Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine  
Copyright © 1996-1999 Silicon Graphics Computer Systems, Inc.  
Copyright © 1994 Hewlett-Packard Company

Distributed under the Boost Software License, Version 1.0. (See accompanying file `LICENSE_1_0.txt` or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))

This product includes software developed at the University of Notre Dame and the Pervasive Technology Labs at Indiana University. For technical information contact Andrew Lumsdaine at the Pervasive Technology Labs at Indiana University. For administrative and license questions contact the Advanced Research and Technology Institute at 351 West 10th Street. Indianapolis, Indiana 46202, phone 317-278-4100, fax 317-274-5902.

Some concepts based on versions from the MTL draft manual and Boost Graph and Property Map documentation, the SGI Standard Template Library documentation and the Hewlett-Packard STL, under the following license:

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

## Concepts

- [Assignable](#)
- [InputIterator](#)
- [OutputIterator](#)
- [ForwardIterator](#)
- [BidirectionalIterator](#)
- [RandomAccessIterator](#)
- [DefaultConstructible](#)
- [CopyConstructible](#)
- [EqualityComparable](#)
- [LessThanComparable](#)
- [SignedInteger](#)

## Concept Assignable

Assignable

### Description

Assignable types must have copy constructors, `operator=` for assignment, and the `swap( )` function defined.

## Refinement of

- [CopyConstructible](#)

## Notation

$X$  A type playing the role of assignable-type in the [Assignable](#) concept.

$x, y$  Objects of type  $X$

## Valid expressions

Name	Expression	Type	Semantics
Assignment	$x = y$	$X \ \&$	Require <code>operator=</code>
Swap	<code>swap(x, y)</code>	<code>void</code>	Require <code>swap( )</code> function

## Models

- `int`

## See also

- [CopyConstructible](#)

# Concept `InputIterator`

`InputIterator`

## Description

An input iterator is an iterator that can read through a sequence of values. It is single-pass (old values of the iterator cannot be re-used), and read-only.

An input iterator represents a position in a sequence. Therefore, the iterator can point into the sequence (returning a value when dereferenced and being incrementable), or be off-the-end (and not dereferenceable or incrementable).

## Refinement of

- [Assignable](#)
- [DefaultConstructible](#)
- [EqualityComparable](#)

## Associated types

- `value_type`

```
std::iterator_traits<Iter>::value_type
```

The value type of the iterator (not necessarily what `*i` returns)

- `difference_type`

```
std::iterator_traits<Iter>::difference_type
```

The difference type of the iterator

- **category**

```
std::iterator_traits<Iter>::iterator_category
```

The category of the iterator

## Notation

**Iter** A type playing the role of iterator-type in the [InputIterator](#) concept.

**i, j** Objects of type **Iter**

**x** Object of type **value\_type**

## Type expressions

**Category tag** category must be derived from `std::input_iterator_tag`, a model of [DefaultConstructible](#), and a model of [CopyConstructible](#).

**Value type copy constructibility** **value\_type** must be a model of [CopyConstructible](#).

**Difference type properties** **difference\_type** must be a model of [SignedInteger](#).

## Valid expressions

Name	Expression	Type	Precondition	Semantics	Postcondition
Dereference	<code>*i</code>	Convertible <b>value_type</b> to	<b>i</b> is incrementable (not off-the-end)		
Preincrement	<code>++i</code>	<b>Iter</b> &	<b>i</b> is incrementable (not off-the-end)		
Postincrement	<code>i++</code>		<b>i</b> is incrementable (not off-the-end)	Equivalent to <code>(void)(++i)</code>	<b>i</b> is dereferenceable or off-the-end
Postincrement and dereference	<code>*i++</code>	Convertible <b>value_type</b> to	<b>i</b> is incrementable (not off-the-end)	Equivalent to <code>{value_type t = *i; ++i; return t;}</code>	<b>i</b> is dereferenceable or off-the-end

## Complexity

All iterator operations must take amortized constant time.

## Models

- `std::istream_iterator`

## See also

- [DefaultConstructible](#)

- [EqualityComparable](#)
- [ForwardIterator](#)
- [OutputIterator](#)

## Concept OutputIterator

OutputIterator

### Description

An output iterator is an iterator that can write a sequence of values. It is single-pass (old values of the iterator cannot be re-used), and write-only.

An output iterator represents a position in a (possibly infinite) sequence. Therefore, the iterator can point into the sequence (returning a value when dereferenced and being incrementable), or be off-the-end (and not dereferenceable or incrementable).

### Associated types

- **value\_type**

```
std::iterator_traits<Iter>::value_type
```

The stated value type of the iterator (should be `void` for an output iterator that does not model some other iterator concept).

- **difference\_type**

```
std::iterator_traits<Iter>::difference_type
```

The difference type of the iterator

- **category**

```
std::iterator_traits<Iter>::iterator_category
```

The category of the iterator

### Notation

Iter	A type playing the role of iterator-type in the <a href="#">OutputIterator</a> concept.
ValueType	A type playing the role of value-type in the <a href="#">OutputIterator</a> concept.
i, j	Objects of type Iter
x	Object of type ValueType

### Type expressions

The type Iter must be a model of [Assignable](#).

The type ValueType must be a model of [Assignable](#).

The type Iter must be a model of [DefaultConstructible](#).

The type Iter must be a model of [EqualityComparable](#).

Category tag category must be derived from `std::output_iterator_tag`, a model of [DefaultConstructible](#), and a model of [CopyConstructible](#).

Difference type properties difference\_type must be a model of [SignedInteger](#).

## Valid expressions

Name	Expression	Type	Precondition	Semantics	Postcondition
Dereference	<code>*i</code>		<code>i</code> is incrementable (not off-the-end)		
Dereference and assign	<code>*i = x</code>		<code>i</code> is incrementable (not off-the-end)		<code>*i</code> may not be written to again until it has been incremented.
Preincrement	<code>++i</code>	Iter &	<code>i</code> is incrementable (not off-the-end)		
Postincrement	<code>i++</code>		<code>i</code> is incrementable (not off-the-end)	Equivalent to <code>(void)(++i)</code>	<code>i</code> is dereferenceable or off-the-end
Postincrement, dereference, and assign	<code>*i++ = x</code>		<code>i</code> is incrementable (not off-the-end)	Equivalent to <code>{ *i = x; ++i; }</code>	<code>i</code> is dereferenceable or off-the-end

## Complexity

All iterator operations must take amortized constant time.

## Models

- `std::ostream_iterator`, ...
- `std::insert_iterator`, ...
- `std::front_insert_iterator`, ...
- `std::back_insert_iterator`, ...

## Concept ForwardIterator

ForwardIterator

## Description

A forward iterator is an iterator that can read through a sequence of values. It is multi-pass (old values of the iterator can be re-used), and can be either mutable (data pointed to by it can be changed) or not mutable.

An iterator represents a position in a sequence. Therefore, the iterator can point into the sequence (returning a value when dereferenced and being incrementable), or be off-the-end (and not dereferenceable or incrementable).

## Refinement of

- [InputIterator](#)

- [OutputIterator](#)

## Associated types

- **value\_type**

```
std::iterator_traits<Iter>::value_type
```

The value type of the iterator

- **category**

```
std::iterator_traits<Iter>::iterator_category
```

The category of the iterator

## Notation

**Iter**     A type playing the role of iterator-type in the [ForwardIterator](#) concept.

**i, j**     Objects of type **Iter**

**x**        Object of type **value\_type**

## Type expressions

Category tag        category must be derived from `std::forward_iterator_tag`.

## Valid expressions

Name	Expression	Type	Precondition	Semantics	Postcondition
Dereference	<code>*i</code>	const-if-not-mutable <b>value_type</b> &	<b>i</b> is incrementable (not off-the-end)		
Member access	<code>i-&gt;{member-name}</code> (return type is pointer-to-object type)	const-if-not-mutable <b>value_type</b> *	<b>i</b> is incrementable (not off-the-end)		
Preincrement	<code>++i</code>	<b>Iter</b> &	<b>i</b> is incrementable (not off-the-end)		
Postincrement	<code>i++</code>	<b>Iter</b>	<b>i</b> is incrementable (not off-the-end)	Equivalent to <code>{ Iter j = i;   ++i;    return   j; }</code>	<b>i</b> is dereferenceable or off-the-end

## Complexity

All iterator operations must take amortized constant time.

## Invariants

Predecrement must return object     `&i = &(++i)`

Unique path through sequence       $i == j$  implies  $++i == ++j$

## Models

- $T^*$
- `std::hash_set<T>::iterator`

## See also

- [BidirectionalIterator](#)

# Concept BidirectionalIterator

BidirectionalIterator

## Description

A bidirectional iterator is an iterator that can read through a sequence of values. It can move in either direction through the sequence, and can be either mutable (data pointed to by it can be changed) or not mutable.

An iterator represents a position in a sequence. Therefore, the iterator can point into the sequence (returning a value when dereferenced and being incrementable), or be off-the-end (and not dereferenceable or incrementable).

## Refinement of

- [ForwardIterator](#)

## Associated types

- **value\_type**

```
std::iterator_traits<Iter>::value_type
```

The value type of the iterator

- **category**

```
std::iterator_traits<Iter>::iterator_category
```

The category of the iterator

## Notation

**Iter**      A type playing the role of iterator-type in the [BidirectionalIterator](#) concept.

**i, j**      Objects of type **Iter**

**x**      Object of type **value\_type**

## Type expressions

Category tag      category must be derived from `std::bidirectional_iterator_tag`.

## Valid expressions

Name	Expression	Type	Precondition	Semantics	Postcondition
Predecrement	--i	Iter &	i is incrementable (not off-the-end) and some dereferenceable iterator j exists such that i == ++j		
Postdecrement	i--	Iter	Same as for predecrement	Equivalent to {Iter j = i; --i; return j;}	i is dereferenceable or off-the-end

## Complexity

All iterator operations must take amortized constant time.

## Invariants

Predecrement must return object `&i = &(--i)`

Unique path through sequence `i == j` implies `--i == --j`

Increment and decrement are inverses `++i; --i;` and `--i; ++i;` must end up with the value of i unmodified, if i both of the operations in the pair are valid.

## Models

- `T *`
- `std::list<T>::iterator`

## See also

- [RandomAccessIterator](#)

## Concept RandomAccessIterator

RandomAccessIterator

## Description

A random access iterator is an iterator that can read through a sequence of values. It can move in either direction through the sequence (by any amount in constant time), and can be either mutable (data pointed to by it can be changed) or not mutable.

An iterator represents a position in a sequence. Therefore, the iterator can point into the sequence (returning a value when dereferenced and being incrementable), or be off-the-end (and not dereferenceable or incrementable).

## Refinement of

- [BidirectionalIterator](#)
- [LessThanComparable](#)



## Associated types

- **value\_type**

```
std::iterator_traits<Iter>::value_type
```

The value type of the iterator

- **category**

```
std::iterator_traits<Iter>::iterator_category
```

The category of the iterator

- **difference\_type**

```
std::iterator_traits<Iter>::difference_type
```

The difference type of the iterator (measure of the number of steps between two iterators)

## Notation

Iter	A type playing the role of iterator-type in the <a href="#">RandomAccessIterator</a> concept.
i, j	Objects of type Iter
x	Object of type value_type
n	Object of type difference_type
int_off	Object of type int

## Type expressions

Category tag      category must be derived from `std::random_access_iterator_tag`.

## Valid expressions

Name	Expression	Type	Semantics
Motion	<code>i += n</code>	Iter &	Equivalent to applying <code>i++</code> <code>n</code> times if <code>n</code> is positive, applying <code>i--</code> <code>-n</code> times if <code>n</code> is negative, and to a null operation if <code>n</code> is zero.
Motion (with integer offset)	<code>i += int_off</code>	Iter &	Equivalent to applying <code>i++</code> <code>n</code> times if <code>n</code> is positive, applying <code>i--</code> <code>-n</code> times if <code>n</code> is negative, and to a null operation if <code>n</code> is zero.
Subtractive motion	<code>i -= n</code>	Iter &	Equivalent to <code>i += (-n)</code>
Subtractive motion (with integer offset)	<code>i -= int_off</code>	Iter &	Equivalent to <code>i += (-n)</code>
Addition	<code>i + n</code>	Iter	Equivalent to <code>{ Iter j = i; j += n; return j; }</code>
Addition with integer	<code>i + int_off</code>	Iter	Equivalent to <code>{ Iter j = i; j += n; return j; }</code>
Addition (count first)	<code>n + i</code>	Iter	Equivalent to <code>i + n</code>
Addition with integer (count first)	<code>int_off + i</code>	Iter	Equivalent to <code>i + n</code>
Subtraction	<code>i - n</code>	Iter	Equivalent to <code>i + (-n)</code>
Subtraction with integer	<code>i - int_off</code>	Iter	Equivalent to <code>i + (-n)</code>
Distance	<code>i - j</code>	difference_type	The number of times <code>i</code> must be incremented (or decremented if the result is negative) to reach <code>j</code> . Not defined if <code>j</code> is not reachable from <code>i</code> .
Element access	<code>i[n]</code>	const-if-not-mutable value_type &	Equivalent to <code>*(i + n)</code>
Element access with integer index	<code>i[int_off]</code>	const-if-not-mutable value_type &	Equivalent to <code>*(i + n)</code>

## Complexity

All iterator operations must take amortized constant time.

## Models

- `T*`
- `std::vector<T>::iterator`

- `std::vector<T>::const_iterator`
- `std::deque<T>::iterator`
- `std::deque<T>::const_iterator`

## See also

- [LessThanComparable](#)

# Concept DefaultConstructible

DefaultConstructible

## Description

DefaultConstructible objects only need to have a default constructor.

## Notation

$X$  A type playing the role of default-constructible-type in the [DefaultConstructible](#) concept.

## Valid expressions

Name	Expression	Type	Semantics
Construction	<code>X()</code>	$X$	Construct an instance of the type with default parameters.

## Models

- `int`
- `std::vector<double>`

# Concept CopyConstructible

CopyConstructible

## Description

Copy constructible types must be able to be constructed from another member of the type.

## Notation

$X$  A type playing the role of copy-constructible-type in the [CopyConstructible](#) concept.

$x, y$  Objects of type  $X$

## Valid expressions

Name	Expression	Type	Semantics
Copy construction	<code>X(x)</code>	$X$	Require copy constructor.

## Models

- `int`

## Concept EqualityComparable

EqualityComparable

### Description

Equality Comparable types must have `==` and `!=` operators.

### Notation

$X$  A type playing the role of comparable-type in the [EqualityComparable](#) concept.

$x, y$  Objects of type  $X$

### Valid expressions

Name	Expression	Type
Equality test	$x == y$	Convertible to <code>bool</code>
Inequality test	$x != y$	Convertible to <code>bool</code>

## Models

- `int`
- `std::vector<int>`

## Concept LessThanComparable

LessThanComparable

### Description

LessThanComparable types must have `<`, `>`, `<=`, and `>=` operators.

### Notation

$X$  A type playing the role of comparable-type in the [LessThanComparable](#) concept.

$x, y$  Objects of type  $X$

## Valid expressions

Name	Expression	Type	Semantics
Less than	$x < y$	Convertible to bool	Determine if one value is less than another.
Less than or equal	$x \leq y$	Convertible to bool	Determine if one value is less than or equal to another.
Greater than	$x > y$	Convertible to bool	Determine if one value is greater than another.
Greater than or equal to	$x \geq y$	Convertible to bool	Determine if one value is greater than or equal to another.

## Models

- `int`

## Concept `SignedInteger`

`SignedInteger`

### Refinement of

- [CopyConstructible](#)
- [Assignable](#)
- [DefaultConstructible](#)
- [EqualityComparable](#)
- [LessThanComparable](#)

### Notation

$T$  A type playing the role of integral-type in the [SignedInteger](#) concept.

$x, y, z$  Objects of type  $T$

$a, b$  Objects of type `int`

### Type expressions

Conversion to `int`  $T$  must be convertible to `int`.

## Valid expressions

Name	Expression	Type
Conversion from int	T(a)	T
Preincrement	++x	T &
Predecrement	--x	T &
Postincrement	x++	T
Postdecrement	x--	T
Sum	x + y	T
Sum with int	x + a	T
Sum-assignment	x += y	T &
Sum-assignment with int	x += a	T &
Difference	x - y	T
Difference with int	x - a	T
Product	x * y	T
Product with int	x * a	T
Product-assignment with int	x *= a	T &
Product with int on left	a * x	T
Quotient	x / y	T
Quotient with int	x / a	T
Right-shift	x >> y	T
Right-shift with int	x >> a	T
Right-shift-assignment with int	x >>= a	T &
Less-than comparison	x < y	Convertible to bool
Less-than comparison with int	x < a	Convertible to bool
Less-than comparison with size_t	x < boost::sample_value < std::size_t >()	Convertible to bool
Greater-than comparison	x > y	Convertible to bool
Greater-than comparison with int	x > a	Convertible to bool
Less-than-or-equal comparison	x <= y	Convertible to bool
Less-than-or-equal comparison with int	x <= a	Convertible to bool
Greater-than-or-equal comparison	x >= y	Convertible to bool

Name	Expression	Type
Greater-than-or-equal comparison with int	$x \geq a$	Convertible to bool
Greater-than-or-equal comparison with int on left	$a \geq x$	Convertible to bool
Equality comparison	$x == y$	Convertible to bool
Equality comparison with int	$x == a$	Convertible to bool