

Iterator Archetype

Author: David Abrahams, Jeremy Siek, Thomas Witt
Contact: dave@boost-consulting.com, jsiek@osl.iu.edu, witt@styleadvisor.com
Organization: [Boost Consulting](#), Indiana University [Open Systems Lab](#), [Zephyr Associates, Inc.](#)
Date: 2004-01-27
Copyright: Copyright David Abrahams, Jeremy Siek, and Thomas Witt 2004. All rights reserved

abstract: The `iterator_archetype` class constructs a minimal implementation of one of the iterator access concepts and one of the iterator traversal concepts. This is used for doing a compile-time check to see if a the type requirements of a template are really enough to cover the implementation of the template. For further information see the documentation for the `boost::concept_check` library.

Table of Contents

[Reference](#)

[iterator_archetype Synopsis](#)
[Access Category Tags](#)
[iterator_archetype Requirements](#)
[iterator_archetype Models](#)
[Traits](#)

Reference

iterator_archetype Synopsis

```
namespace iterator_archetypes
{
    // Access categories

    typedef /*implementation defined*/ readable_iterator_t;
    typedef /*implementation defined*/ writable_iterator_t;
    typedef /*implementation defined*/ readable_writable_iterator_t;
    typedef /*implementation defined*/ readable_lvalue_iterator_t;
    typedef /*implementation defined*/ writable_lvalue_iterator_t;

}

template <
    class Value
```

```

    , class AccessCategory
    , class TraversalCategory
>
class iterator_archetype
{
    typedef /* see below */ value_type;
    typedef /* see below */ reference;
    typedef /* see below */ pointer;
    typedef /* see below */ difference_type;
    typedef /* see below */ iterator_category;
};

```

Access Category Tags

The access category types provided correspond to the following standard iterator access concept combinations:

```

readable_iterator_t :=

    Readable Iterator

writable_iterator_t :=

    Writable Iterator

readable_writable_iterator_t :=

    Readable Iterator & Writable Iterator & Swappable Iterator

readable_lvalue_iterator_t :=

    Readable Iterator & Lvalue Iterator

writable_lvalue_iterator_t :=

    Readable Iterator & Writable Iterator & Swappable Iterator & Lvalue Iterator

```

iterator_archetype Requirements

The `AccessCategory` argument must be one of the predefined access category tags. The `TraversalCategory` must be one of the standard traversal tags. The `Value` type must satisfy the requirements of the iterator concept specified by `AccessCategory` and `TraversalCategory` as implied by the nested traits types.

iterator_archetype Models

`iterator_archetype` models the iterator concepts specified by the `AccessCategory` and `TraversalCategory` arguments. `iterator_archetype` does not model any other access concepts or any more derived traversal concepts.

Traits

The nested trait types are defined as follows:

```

if (AccessCategory == readable_iterator_t)

    value_type = Value
    reference  = Value
    pointer    = Value*

else if (AccessCategory == writable_iterator_t)

    value_type = void
    reference  = void
    pointer    = void

else if (AccessCategory == readable_writable_iterator_t)

    value_type = Value

    reference :=

        A type X that is convertible to Value for which the following
        expression is valid. Given an object x of type X and v of type
        Value.

        x = v

    pointer    = Value*

else if (AccessCategory == readable_lvalue_iterator_t)

    value_type = Value
    reference  = Value const&
    pointer    = Value const*

else if (AccessCategory == writable_lvalue_iterator_t)

    value_type = Value
    reference  = Value&
    pointer    = Value*

if ( TraversalCategory is convertible to forward_traversal_tag )

    difference_type := ptrdiff_t

else

    difference_type := unspecified type

iterator_category :=

    A type X satisfying the following two constraints:

    1. X is convertible to X1, and not to any more-derived
       type, where X1 is defined by:

```

```

if (reference is a reference type
    && TraversalCategory is convertible to forward_traversal_tag)
{
    if (TraversalCategory is convertible to random_access_traversal_tag)
        X1 = random_access_iterator_tag
    else if (TraversalCategory is convertible to bidirectional_traversal_tag)
        X1 = bidirectional_iterator_tag
    else
        X1 = forward_iterator_tag
}
else
{
    if (TraversalCategory is convertible to single_pass_traversal_tag
        && reference != void)
        X1 = input_iterator_tag
    else
        X1 = output_iterator_tag
}

```

2. X is convertible to TraversalCategory