sgi

Iterators

Iterators

Category: iterators

Overview

Component type: overview

Summary

Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.

Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector and a doubly linked list.

The STL defines several different concepts related to iterators, several predefined iterators, and a collection of types and functions for manipulating iterators.

Description

Iterators are in fact not a single concept, but six concepts that form a hierarchy: some of them define only a very restricted set of operations, while others define additional functionality. The five concepts that are actually used by algorithms are Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator, and Random Access Iterator. A sixth concept, Trivial Iterator, is introduced only to clarify the definitions of the other iterator concepts.

The most restricted sorts of iterators are Input Iterators and Output Iterators, both of which permit "single pass" algorithms but do not necessarily support "multi-pass" algorithms. Input iterators only guarantee read access: it is possible to dereference an Input Iterator to obtain the value it points to, but not it is not necessarily possible to assign a new value through an input iterator. Similarly, Output Iterators only guarantee write access: it is possible to assign a value through an Output Iterator, but not necessarily possible to refer to that value.

Forward Iterators are a refinement of Input Iterators and Output Iterators: they support the Input Iterator and Output Iterator operations and also provide additional functionality. In particular, it is possible to use "multi-pass" algorithms with Forward Iterators. A Forward Iterator may be *constant*, in which case it is possible to access the object it points to but not to to assign a new value through it, or *mutable*, in which case it is possible to do both.

Bidirectional Iterators, like Forward Iterators, allow multi-pass algorithms. As the name suggests, they are different in that they support motion in both directions: a Bidirectional Iterator may be incremented to obtain the next element or decremented to obtain the previous element. A Forward Iterator, by contrast, is only required to support forward motion. An iterator used to traverse a singly linked list, for example, would be a Forward Iterator, while an iterator used to traverse a doubly linked list would be a Bidirectional Iterator.

Finally, Random Access Iterators allow the operations of pointer arithmetic: addition of arbitrary offsets, subscripting, subtraction of one iterator from another to find a distance, and so on.

Most algorithms are expressed not in terms of a single iterator but in terms of a *range* of iterators [1]; the notation `[first, last)` refers to all of the iterators from `first` up to, but **not including**, `last`. [2] Note that a range may be empty, *i.e.* `first` and `last` may be the same iterator. Note also that if there are `n` iterators in a range, then the notation `[first, last)` represents `n+1` positions. This is crucial: algorithms that operate on `n` things frequently require `n+1` positions. Linear search, for example (`find`) must be able to return some value to indicate that the search was unsuccessful.

Sometimes it is important to be able to infer some properties of an iterator: the type of object that is returned when it is dereferenced, for example. There are two different mechanisms to support this sort of inferrence: an older mechanism called Iterator Tags, and a newer mechanism called `iterator_traits` [3].

Concepts

- Trivial Iterator
- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

Types

- `istream_iterator`
- `ostream_iterator`

- `reverse_iterator`
- `reverse_bidirectional_iterator`
- `insert_iterator`
- `front_insert_iterator`
- `back_insert_iterator`

- `iterator_traits`

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`

Functions

- `distance_type`
- `value_type`
- `iterator_category`

- distance
- advance

- inserter
- front_inserter
- back_inserter

Notes

[1] Ranges are not a well-defined concept for Trivial Iterators, because a Trivial Iterator cannot be incremented: there is no such thing as a next element. They are also not a well-defined concept for Output Iterators, because it is impossible to compare two Output Iterators for equality. Equality is crucial to the definition of a range, because only by comparing an iterator for equality with the last element is it possible to step through a range.

[2] Sometimes the notation `[first, last)` refers to the iterators `first`, `first+1`, ..., `last-1` and sometimes it refers to the objects pointed to by those iterators: `*first`, `*(first+1)`, ..., `*(last-1)`. In most cases it will be obvious from context which of these is meant; where the distinction is important, the notation will be qualified explicitly as "range of iterators" or "range of objects".

[3] The `iterator_traits` class relies on a C++ feature known as *partial specialization*. Many of today's compilers don't implement the complete standard; in particular, many compilers do not support partial specialization. If your compiler does not support partial specialization, then you will not be able to use `iterator_traits`, and you will instead have to continue using the functions `iterator_category`, `distance_type`, and `value_type`.

See also

STL Home