Boost.Array

Nicolai Josuttis

Table of Contents

Introduction

The C++ Standard Template Library STL as part of the C++ Standard Library provides a framework for processing algorithms on different kind of containers. However, ordinary arrays don't provide the interface of STL containers (although, they provide the iterator interface of STL containers).

As replacement for ordinary arrays, the STL provides class `std::vector`. However, `std::vector<>` provides the semantics of dynamic arrays. Thus, it manages data to be able to change the number of elements. This results in some overhead in case only arrays with static size are needed.

In his book, *Generic Programming and the STL*, Matthew H. Austern introduces a useful wrapper class for ordinary arrays with static size, called `block`. It is safer and has no worse performance than ordinary arrays. In *The C++ Programming Language*, 3rd edition, Bjarne Stroustrup introduces a similar class, called `c_array`, which I (Nicolai Josuttis) present slightly modified in my book *The C++ Standard Library - A Tutorial and Reference*, called `carray`. This is the essence of these approaches spiced with many feedback from boost.

After considering different names, we decided to name this class simply `array`.

Note that this class is suggested to be part of the next Technical Report, which will extend the C++ Standard (see http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1548.htm).

Update: `std::array` is (as of C++11) part of the C++ standard. The differences between `boost::array` and `std::array` are minimal. If you are using C++11, you should consider using `std::array` instead of `boost::array`.

Class `array` fulfills most but not all of the requirements of "reversible containers" (see Section 23.1, [lib.container.requirements] of the C++ Standard). The reasons array is not an reversible STL container is because:

- No constructors are provided.
- Elements may have an undetermined initial value (see the section called “Design Rationale”).
- `swap`() has no constant complexity.
- `size`() is always constant, based on the second template argument of the type.
- The container provides no allocator support.

It doesn't fulfill the requirements of a "sequence" (see Section 23.1.1, [lib.sequence.reqmts] of the C++ Standard), except that:

- `front`() and `back`() are provided.
- `operator[]` and `at`() are provided.

Reference

Header <boost/array.hpp>

```
namespace boost {
  template<typename T, std::size_t N> class array;
  template<typename T, std::size_t N> void swap(array<T, N>&, array<T, N>&);
  template<typename T, std::size_t N>
    bool operator==(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator!=(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator<(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator>(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator<=(const array<T, N>&, const array<T, N>&);
  template<typename T, std::size_t N>
    bool operator>=(const array<T, N>&, const array<T, N>&);
}
```

Class template array

boost::array — STL compliant container wrapper for arrays of constant size

Synopsis

```
// In header: <boost/array.hpp>

template<typename T, std::size_t N>
class array {
public:
  // types
  typedef T                                     value_type;
  typedef T*                                    iterator;
  typedef const T*                              const_iterator;
  typedef std::reverse_iterator<iterator>       reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
  typedef T&                                    reference;
  typedef const T&                              const_reference;
  typedef std::size_t                           size_type;
  typedef std::ptrdiff_t                        difference_type;

  // static constants
  static const size_type static_size = N;

  // construct/copy/destruct
  template<typename U> array& operator=(const array<U, N>&);

  // iterator support
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;

  // reverse iterator support
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;

  // capacity
  size_type size();
  bool empty();
  size_type max_size();

  // element access
  reference operator[](size_type);
  const_reference operator[](size_type) const;
  reference at(size_type);
  const_reference at(size_type) const;
  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;
  const T* data() const;
  T* c_array();

  // modifiers
  void swap(array<T, N>&);
  void assign(const T&);

  // public data members
  T elems[N];
};

// specialized algorithms
```

```
template<typename T, std::size_t N> void swap(array<T, N>&, array<T, N>&);

// comparisons
template<typename T, std::size_t N>
  bool operator==(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator!=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator<(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator>(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator<=(const array<T, N>&, const array<T, N>&);
template<typename T, std::size_t N>
  bool operator>=(const array<T, N>&, const array<T, N>&);
```

Description

`array` public construct/copy/destruct

1.
```
template<typename U> array& operator=(const array<U, N>& other);
```

Effects: `std::copy(rhs.begin(),rhs.end(), begin())`

`array` iterator support

1.
```
iterator begin();
const_iterator begin() const;
```

Returns: iterator for the first element
Throws: will not throw

2.
```
iterator end();
const_iterator end() const;
```

Returns: iterator for position after the last element
Throws: will not throw

`array` reverse iterator support

1.
```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

Returns: reverse iterator for the first element of reverse iteration

2.
```
reverse_iterator rend();
const_reverse_iterator rend() const;
```

Returns: reverse iterator for position after the last element in reverse iteration

`array` capacity

1.
```
size_type size();
```

Returns: `N`

2.
```
bool empty();
```

Returns: `N==0`
Throws: will not throw

3.
```
size_type max_size();
```

Returns: `N`
Throws: will not throw

`array` element access

1.
```
reference operator[](size_type i);
const_reference operator[](size_type i) const;
```

Requires: `i < N`
Returns: element with index `i`
Throws: will not throw.

2.
```
reference at(size_type i);
const_reference at(size_type i) const;
```

Returns: element with index `i`
Throws: `std::range_error` if `i >= N`

3.
```
reference front();
const_reference front() const;
```

Requires: `N > 0`
Returns: the first element
Throws: will not throw

4.
```
reference back();
const_reference back() const;
```

Requires: `N > 0`
Returns: the last element
Throws: will not throw

5.
```
const T* data() const;
```

Returns: `elems`
Throws: will not throw

6.
```
T* c_array();
```

Returns: `elems`
Throws: will not throw

`array` modifiers

1.
```
void swap(array<T, N>& other);
```

Effects: `std::swap_ranges(begin(), end(), other.begin())`
Complexity: linear in `N`

2.
```
void assign(const T& value);
```

Effects: `std::fill_n(begin(), N, value)`

`array` specialized algorithms

1.
```
template<typename T, std::size_t N> void swap(array<T, N>& x, array<T, N>& y);
```

Effects: `x.swap(y)`
Throws: will not throw.

`array` comparisons

1.
```
template<typename T, std::size_t N>
  bool operator==(const array<T, N>& x, const array<T, N>& y);
```

Returns: `std::equal(x.begin(), x.end(), y.begin())`

2.
```
template<typename T, std::size_t N>
  bool operator!=(const array<T, N>& x, const array<T, N>& y);
```

Returns: `!(x == y)`

3.
```
template<typename T, std::size_t N>
  bool operator<(const array<T, N>& x, const array<T, N>& y);
```

Returns: `std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end())`

4.
```
template<typename T, std::size_t N>
  bool operator>(const array<T, N>& x, const array<T, N>& y);
```

Returns: `y < x`

5.
```
template<typename T, std::size_t N>
  bool operator<=(const array<T, N>& x, const array<T, N>& y);
```

Returns: `!(y < x)`

6.
```
template<typename T, std::size_t N>
  bool operator>=(const array<T, N>& x, const array<T, N>& y);
```

Returns: `!(x < y)`

Design Rationale

There was an important design tradeoff regarding the constructors: We could implement array as an "aggregate" (see Section 8.5.1, [dcl.init.aggr], of the C++ Standard). This would mean:

- An array can be initialized with a brace-enclosing, comma-separated list of initializers for the elements of the container, written in increasing subscript order:

  ```
  boost::array<int,4> a = { { 1, 2, 3 } };
  ```

  Note that if there are fewer elements in the initializer list, then each remaining element gets default-initialized (thus, it has a defined value).

However, this approach has its drawbacks: **passing no initializer list means that the elements have an indetermined initial value**, because the rule says that aggregates may have:

- No user-declared constructors.
- No private or protected non-static data members.
- No base classes.
- No virtual functions.

Nevertheless, The current implementation uses this approach.

Note that for standard conforming compilers it is possible to use fewer braces (according to 8.5.1 (11) of the Standard). That is, you can initialize an array as follows:

```
boost::array<int,4> a = { 1, 2, 3 };
```

I'd appreciate any constructive feedback. **Please note: I don't have time to read all boost mails. Thus, to make sure that feedback arrives to me, please send me a copy of each mail regarding this class.**

The code is provided "as is" without expressed or implied warranty.

For more information...

To find more details about using ordinary arrays in C++ and the framework of the STL, see e.g.

The C++ Standard Library - A Tutorial and Reference
by Nicolai M. Josuttis
Addison Wesley Longman, 1999
ISBN 0-201-37926-0

Home Page of Nicolai Josuttis

Acknowledgements

Doug Gregor ported the documentation to the BoostBook format.