
Programming languages — C++

Langages de programmation — C++

Adopted by INCITS (InterNational Committee for Information Technology Standards) as an American National Standard.

Date of ANSI Approval: 12/29/2003

Published by American National Standards Institute,
25 West 43rd Street, New York, New York 10036

Copyright 2003 by Information Technology Industry Council (ITI).
All rights reserved.

These materials are subject to copyright claims of International Standardization Organization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and Information Technology Industry Council (ITI). Not for resale. No part of this publication may be reproduced in any form, including an electronic retrieval system, without the prior written permission of ITI. All requests pertaining to this standard should be submitted to ITI, 1250 Eye Street NW, Washington, DC 20005.

Printed in the United States of America



PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2003

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 14882 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces the first edition (ISO/IEC 14882:1998), which has been technically revised.

23 Containers library

23.2.3.3 Class template stack

```

public:
    explicit stack(const Container& = Container());

    bool      empty() const           { return c.empty(); }
    size_type size()  const           { return c.size(); }
    value_type& top()                  { return c.back(); }
    const value_type& top() const      { return c.back(); }
    void push(const value_type& x)     { c.push_back(x); }
    void pop()                        { c.pop_back(); }
};

template <class T, class Container>
    bool operator==(const stack<T, Container>& x,
                    const stack<T, Container>& y);
template <class T, class Container>
    bool operator< (const stack<T, Container>& x,
                    const stack<T, Container>& y);
template <class T, class Container>
    bool operator!=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
template <class T, class Container>
    bool operator> (const stack<T, Container>& x,
                    const stack<T, Container>& y);
template <class T, class Container>
    bool operator>=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
template <class T, class Container>
    bool operator<=(const stack<T, Container>& x,
                    const stack<T, Container>& y);
}

```

23.2.4 Class template vector

[lib.vector]

- 1 A vector is a kind of sequence that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if `v` is a `vector<T, Allocator>` where `T` is some type other than `bool`, then it obeys the identity `&v[n] == &v[0] + n` for all `0 ≤ n < v.size()`.
- 2 A vector satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.1) and of a sequence, including most of the optional sequence requirements (23.1.1). The exceptions are the `push_front` and `pop_front` member functions, which are not provided. Descriptions are provided here only for operations on `vector` that are not described in one of these tables or for operations where there is additional semantic information.

23.2.4 Class template vector

23 Containers library

```

namespace std {
    template <class T, class Allocator = allocator<T> >
    class vector {
    public:
        // types:
        typedef typename Allocator::reference      reference;
        typedef typename Allocator::const_reference const_reference;
        typedef implementation defined          iterator;           // See 23.1
        typedef implementation defined          const_iterator;     // See 23.1
        typedef implementation defined          size_type;           // See 23.1
        typedef implementation defined          difference_type;     // See 23.1
        typedef T                                   value_type;
        typedef Allocator                          allocator_type;
        typedef typename Allocator::pointer        pointer;
        typedef typename Allocator::const_pointer  const_pointer;
        typedef std::reverse_iterator<iterator>    reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // 23.2.4.1 construct/copy/destroy:
        explicit vector(const Allocator& = Allocator());
        explicit vector(size_type n, const T& value = T(),
            const Allocator& = Allocator());
        template <class InputIterator>
            vector(InputIterator first, InputIterator last,
                const Allocator& = Allocator());
        vector(const vector<T,Allocator>& x);
        ~vector();
        vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
        template <class InputIterator>
            void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& u);
        allocator_type get_allocator() const;

        // iterators:
        iterator      begin();
        const_iterator begin() const;
        iterator      end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;
        reverse_iterator rend();
        const_reverse_iterator rend() const;

        // 23.2.4.2 capacity:
        size_type size() const;
        size_type max_size() const;
        void      resize(size_type sz, T c = T());
        size_type capacity() const;
        bool      empty() const;
        void      reserve(size_type n);

        // element access:
        reference      operator[] (size_type n);
        const_reference operator[] (size_type n) const;
        const_reference at(size_type n) const;
        reference      at(size_type n);
        reference      front();
        const_reference front() const;
        reference      back();
        const_reference back() const;
    };
}

```

23 Containers library

23.2.4 Class template vector

```

// 23.2.4.3 modifiers:
void push_back(const T& x);
void pop_back();
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
    void insert(iterator position,
                InputIterator first, InputIterator last);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap(vector<T,Allocator>&);
void clear();
};

template <class T, class Allocator>
    bool operator==(const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);
template <class T, class Allocator>
    bool operator< (const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);
template <class T, class Allocator>
    bool operator!=(const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);
template <class T, class Allocator>
    bool operator> (const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);
template <class T, class Allocator>
    bool operator>=(const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);
template <class T, class Allocator>
    bool operator<=(const vector<T,Allocator>& x,
                    const vector<T,Allocator>& y);

// specialized algorithms:
template <class T, class Allocator>
    void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
}

```

23.2.4.1 vector constructors, copy, and assignment

[lib.vector.cons]

```

vector(const Allocator& = Allocator());
explicit vector(size_type n, const T& value = T(),
                const Allocator& = Allocator());
template <class InputIterator>
    vector(InputIterator first, InputIterator last,
           const Allocator& = Allocator());
vector(const vector<T,Allocator>& x);

```

- 1 **Complexity:** The constructor template `<class InputIterator> vector(InputIterator first, InputIterator last)` makes only N calls to the copy constructor of T (where N is the distance between `first` and `last`) and no reallocations if iterators `first` and `last` are of forward, bidirectional, or random access categories. It makes order N calls to the copy constructor of T and order $\log N$ reallocations if they are just input iterators.

23.2.4.1 vector constructors, copy, and assignment

23 Containers library

```
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

2 **Effects:**

```
erase(begin(), end());
insert(begin(), first, last);
```

```
void assign(size_type n, const T& t);
```

3 **Effects:**

```
erase(begin(), end());
insert(begin(), n, t);
```

23.2.4.2 vector capacity

[lib.vector.capacity]

```
size_type capacity() const;
```

1 **Returns:** The total number of elements that the vector can hold without requiring reallocation.

```
void reserve(size_type n);
```

2 **Effects:** A directive that informs a vector of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`.3 **Complexity:** It does not change the size of the sequence and takes at most linear time in the size of the sequence.4 **Throws:** `length_error` if $n > \text{max\_size}()$.²⁴⁸⁾5 **Notes:** Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the size specified in the most recent call to `reserve()`.

```
void resize(size_type sz, T c = T());
```

6 **Effects:**

```
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size())
    erase(begin()+sz, end());
else
    ; // do nothing
```

23.2.4.3 vector modifiers

[lib.vector.modifiers]

```
iterator insert(iterator position, const T& x);
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
```

1 **Notes:** Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor or assignment operator of `T` there are no effects.

²⁴⁸⁾ `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

23 Containers library

23.2.4.3 vector modifiers

- 2 **Complexity:** If `first` and `last` are forward iterators, bidirectional iterators, or random access iterators, the complexity is linear in the number of elements in the range `[first, last)` plus the distance to the end of the vector. If they are input iterators, the complexity is proportional to the number of elements in the range `[first, last)` times the distance to the end of the vector.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

- 3 **Effects:** Invalidates all the iterators and references after the point of the erase.
- 4 **Complexity:** The destructor of `T` is called the number of times equal to the number of the elements erased, but the assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.
- 5 **Throws:** Nothing unless an exception is thrown by the copy constructor or assignment operator of `T`.

23.2.4.4 vector specialized algorithms

[lib.vector.special]

```
template <class T, class Allocator>
void swap(vector<T,Allocator>& x, vector<T,Allocator>& y);
```

- 1 **Effects:**

```
x.swap(y);
```

23.2.5 Class `vector<bool>`

[lib.vector.bool]

- 1 To optimize space allocation, a specialization of `vector` for `bool` elements is provided:

```
namespace std {
    template <class Allocator> class vector<bool, Allocator> {
    public:
        // types:
        typedef bool                                const_reference;
        typedef implementation defined            iterator;           // See 23.1
        typedef implementation defined            const_iterator;    // See 23.1
        typedef implementation defined            size_type;         // See 23.1
        typedef implementation defined            difference_type;    // See 23.1
        typedef bool                                value_type;
        typedef Allocator                            allocator_type;
        typedef implementation defined            pointer;
        typedef implementation defined            const_pointer;
        typedef std::reverse_iterator<iterator>      reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        // bit reference:
        class reference {
            friend class vector;
            reference();
        public:
            ~reference();
            operator bool() const;
            reference& operator=(const bool x);
            reference& operator=(const reference& x);
            void flip();           // flips the bit
        };
    };
}
```

```

// construct/copy/destroy:
explicit vector(const Allocator& = Allocator());
explicit vector(size_type n, const bool& value = bool(),
               const Allocator& = Allocator());
template <class InputIterator>
    vector(InputIterator first, InputIterator last,
           const Allocator& = Allocator());
vector(const vector<bool,Allocator>& x);
~vector();
vector<bool,Allocator>& operator=(const vector<bool,Allocator>& x);
template <class InputIterator>
    void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
allocator_type get_allocator() const;

// iterators:
iterator          begin();
const_iterator    begin() const;
iterator          end();
const_iterator    end() const;
reverse_iterator  rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator  rend();
const_reverse_iterator rend() const;

// capacity:
size_type size() const;
size_type max_size() const;
void      resize(size_type sz, bool c = false);
size_type capacity() const;
bool      empty() const;
void      reserve(size_type n);

// element access:
reference      operator[] (size_type n);
const_reference operator[] (size_type n) const;
const_reference at(size_type n) const;
reference      at(size_type n);
reference      front();
const_reference front() const;
reference      back();
const_reference back() const;

// modifiers:
void push_back(const bool& x);
void pop_back();
iterator insert(iterator position, const bool& x);
void      insert (iterator position, size_type n, const bool& x);
template <class InputIterator>
    void insert(iterator position,
               InputIterator first, InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap(vector<bool,Allocator>&);
static void swap(reference x, reference y);
void flip(); // flips all bits
void clear();
};

```

23 Containers library

23.2.5 Class `vector<bool>`

```

template <class Allocator>
    bool operator==(const vector<bool,Allocator>& x,
                    const vector<bool,Allocator>& y);
template <class Allocator>
    bool operator< (const vector<bool,Allocator>& x,
                   const vector<bool,Allocator>& y);
template <class Allocator>
    bool operator!=(const vector<bool,Allocator>& x,
                   const vector<bool,Allocator>& y);
template <class Allocator>
    bool operator> (const vector<bool,Allocator>& x,
                   const vector<bool,Allocator>& y);
template <class Allocator>
    bool operator>=(const vector<bool,Allocator>& x,
                   const vector<bool,Allocator>& y);
template <class Allocator>
    bool operator<=(const vector<bool,Allocator>& x,
                   const vector<bool,Allocator>& y);

// specialized algorithms:
template <class Allocator>
    void swap(vector<bool,Allocator>& x, vector<bool,Allocator>& y);
}

```

2 reference is a class that simulates the behavior of references of a single bit in `vector<bool>`.

23.3 Associative containers

[lib.associative]

1 Headers `<map>` and `<set>`:

Header `<map>` synopsis

```

namespace std {
    template <class Key, class T, class Compare = less<Key>,
              class Allocator = allocator<pair<const Key, T> > >
        class map;
    template <class Key, class T, class Compare, class Allocator>
        bool operator==(const map<Key,T,Compare,Allocator>& x,
                        const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        bool operator< (const map<Key,T,Compare,Allocator>& x,
                       const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        bool operator!=(const map<Key,T,Compare,Allocator>& x,
                       const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        bool operator> (const map<Key,T,Compare,Allocator>& x,
                       const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        bool operator>=(const map<Key,T,Compare,Allocator>& x,
                       const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        bool operator<=(const map<Key,T,Compare,Allocator>& x,
                       const map<Key,T,Compare,Allocator>& y);
    template <class Key, class T, class Compare, class Allocator>
        void swap(map<Key,T,Compare,Allocator>& x,
                  map<Key,T,Compare,Allocator>& y);
}

```