目录

组成

- 容器（containers）
- 算法（algorithms）
- 迭代器（iterators）
- 仿函数（functors）
- 配接器（adapters）
- 空间配置器（allocator）

容器（containers）

- 序列式容器（sequence containers）：元素都是可序（ordered），但未必是有序（sorted）
- 关联式容器（associattive containers

array

array 是固定大小的顺序容器，它们保存了一个以严格的线性顺序排列的特定数量的元素。

在内部，一个数组除了它所包含的元素（甚至不是它的大小，它是一个模板参数，在编译时是固定的）以外不保存任何数据。存储大小与用语言括号语法（[]）声明的普通数组一样高效。这个类只是增加了一层成员函数和全局函数，所以数组可以作为标准容器使用。

与其他标准容器不同，数组具有固定的大小，并且不通过分配器管理其元素的分配：它们是封装固定大小数组元素的聚合类型。因此，他们不能动态地扩大或缩小。

零大小的数组是有效的，但是它们不应该被解除引用（成员的前面，后面和数据）。

与标准库中的其他容器不同，交换两个数组容器是一种线性操作，它涉及单独交换范围内的所有元素，这通常是相当低效的操作。另一方面，这允许迭代器在两个容器中的元素保持其原始容器关联。

数组容器的另一个独特特性是它们可以被当作元组对象来处理：array 头部重载 get 函数来访问数组元素，就像它是一个元组，以及专门的 tuple*size* *和* *tuple*element 类型。

```
template < class T, size_t N > class array;
```

array::begin

返回指向数组容器中第一个元素的迭代器。

```
      iterator begin() noexcept;
const_iterator begin() const noexcept;
```

```
#include <iostream>
#include <array>
int main()
{
    std::array<int, 5> myarray = {2, 16, 77,34, 50};
    std::cout << "myarray contains:";
    for(auto it = myarray.begin(); it != myarray.end(); ++i)
        std::cout << ' ' << *it;
    std::cout << '\n';
    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::end

返回指向数组容器中最后一个元素之后的理论元素的迭代器。

```
      iterator end() noexcept;
const_iterator end() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
```

```
{
    std::array<int,5> myarray = { 5, 19, 77, 34, 99 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.begin(); it != myarray.end(); ++it )
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 5 19 77 34 99
```

array::rbegin

返回指向数组容器中最后一个元素的反向迭代器。

```
      reverse_iterator rbegin () noexcept;
const_reverse_iterator rbegin () const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14} ;
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::rend

返回一个反向迭代器，指向数组中第一个元素之前的理论元素（这被认为是它的反向结束）。

```
      reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14};
    std::cout << "myarray contains";
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;
    std::cout << '\n';
    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::cbegin

返回指向数组容器中第一个元素的常量迭代器（const_iterator）；这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cbegin () const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myarray = {2, 16, 77, 34, 50};

    std::cout << "myarray contains:";

    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it;   // cannot modify *it

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::cend

返回指向数组容器中最后一个元素之后的理论元素的常量迭代器（const_iterator）。这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cend() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myarray = { 15, 720, 801, 1002, 3502 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it;   // cannot modify *it

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::crbegin

返回指向数组容器中最后一个元素的常量反向迭代器（const*reverse*iterator）

```
const_reverse_iterator crbegin（）const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin() ; rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit;   // cannot modify *rit
    std::cout << '\n';
```

```
    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::crend

返回指向数组中第一个元素之前的理论元素的常量反向迭代器（const*reverse*iterator），它被认为是其反向结束。

```
const_reverse_iterator crend() const noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin() ; rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit;   // cannot modify *rit

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::size

返回数组容器中元素的数量。

```
constexpr size_type size () noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myints;
    std::cout << "size of myints:" << myints.size() << std::endl;
    std::cout << "sizeof(myints):" << sizeof(myints) << std::endl;

    return 0;
}
```

Possible Output

```
size of myints: 5
sizeof(myints): 20
```

array::max_size

返回数组容器可容纳的最大元素数。数组对象的 max_size 与其 size 一样，始终等于用于实例化数组模板类的第二个模板参数。

```
constexpr size_type max_size() noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,10> myints;
    std::cout << "size of myints: " << myints.size() << '\n';
    std::cout << "max_size of myints: " << myints.max_size() << '\n';
```

```
    return 0;
}
```

Output

```
size of myints: 10
max_size of myints: 10
```

array::empty

返回一个布尔值，指示数组容器是否为空，即它的 size()是否为 0。

```
constexpr bool empty() noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
  std::array<int,0> first;
  std::array<int,5> second;
  std::cout << "first " << (first.empty() ? "is empty" : "is not empty") << '\n';
  std::cout << "second " << (second.empty() ? "is empty" : "is not empty") << '\n';
  return 0;
}
```

Output:

```
first is empty
second is not empt
```

array::operator[]

返回数组中第 n 个位置的元素的引用。与 array::at 相似，但 array::at 会检查数组边界并通过抛出一个 out*of*range 异常来判断 n 是否超出范围，而 array::operator[]不检查边界。

```
      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;
    unsigned int i;
    // assign some values:
    for(i=0; i<10; i++)
        myarray[i] = i;
    // print content
    std::cout << "myarray contains:";
    for(i=0; i<10; i++)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::at

返回数组中第 n 个位置的元素的引用。与 array::operator[]相似，但 array::at 会检查数组边界并通过抛出一个 out*of*range 异常来判断 n 是否超出范围，而 array::operator[]不检查边界。

```
      reference at ( size_type n );
const_reference at ( size_type n ) const;
```

```
#include <iostream>
#include <array>
int main()
{
	std::array<int, 10> myarray;
	unsigned int i;

	// assign some values:
	for (i = 0; i<10; i++)
		myarray[i] = i;

	// print content
	std::cout << "myarray contains:";
	for (i = 0; i<10; i++)
		std::cout << ' ' << myarray.at(i);
	std::cout << '\n';

	return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::front

返回对数组容器中第一个元素的引用。array::begin 返回的是迭代器，array::front 返回的是直接引用。在空容器上调用此函数会导致未定义的行为。

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

Example

```
#include <iostream>
#include <array>
int main ()
{
  std::array<int,3> myarray = {2, 16, 77};

  std::cout << "front is: " << myarray.front() << std::endl;   // 2
  std::cout << "back is: " << myarray.back() << std::endl;     // 77
  myarray.front() = 100;
  std::cout << "myarray now contains:";
  for ( int& x : myarray ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
front is: 2
back is: 77
myarray now contains: 100 16 77
```

array::back

返回对数组容器中最后一个元素的引用。array::end 返回的是迭代器，array::back 返回的是直接引用。在空容器上调用此函数会导致未定义的行为。

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

Example

```
#include <iostream>
#include <array>
```

```
int main ()
{
  std::array<int,3> myarray = {5, 19, 77};

  std::cout << "front is: " << myarray.front() << std::endl;   // 5
  std::cout << "back is: " << myarray.back() << std::endl;     // 77
  myarray.back() = 50;
  std::cout << "myarray now contains:";
  for ( int& x : myarray ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
front is: 5
back is: 77
myarray now contains: 5 19 50
```

array::data

返回指向数组对象中第一个元素的指针。

由于数组中的元素存储在连续的存储位置，所以检索到的指针可以偏移以访问数组中的任何元素。

```
      value_type* data() noexcept;
const value_type* data() const noexcept;
```

Example

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
  const char* cstr = "Test string";
  std::array<char,12> charray;
  std::memcpy (charray.data(),cstr,12);
  std::cout << charray.data() << '\n';

  return 0;
}
```

Output

```
Test string
```

array::fill

用 val 填充数组所有元素，将 val 设置为数组对象中所有元素的值。

```
void fill (const value_type& val);
```

Example

```
#include <iostream>
#include <array>

int main () {
  std::array<int,6> myarray;

  myarray.fill(5);

  std::cout << "myarray contains:";
  for ( int& x : myarray) { std::cout << ' ' << x; }
```

```
  std::cout << '\n';

  return 0;
}
```

Output

```
myarray contains: 5 5 5 5 5 5
```

array::swap

通过 x 的内容交换数组的内容，这是另一个相同类型的数组对象（包括相同的大小）。

与其他容器的交换成员函数不同，此成员函数通过在各个元素之间执行与其大小相同的单独交换操作，以线性时间运行。

```
void swap (array& x)
noexcept(noexcept(swap(declval<value_type&>(),declval<value_type&>())));
```

Example

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,5> first = {10, 20, 30, 40, 50};
  std::array<int,5> second = {11, 22, 33, 44, 55};

  first.swap (second);

  std::cout << "first:";
  for (int& x : first) std::cout << ' ' << x;
  std::cout << '\n';

  std::cout << "second:";
  for (int& x : second) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
first: 11 22 33 44 55
second: 10 20 30 40 50
```

get（array）

形如：std::get<0>(myarray)；传入一个数组容器，返回指定位置元素的引用。

```
template <size_t I，class T，size_t N> T&get（array <T，N>&arr）noexcept;
template <size_t I，class T，size_t N> T && get（array <T，N> && arr）noexcept;
template <size_t I，class T，size_t N> const T&get（const array <T，N>&arr）noexcept;
```

Example

```
#include <iostream>
#include <array>
#include <tuple>

int main ()
{
  std::array<int,3> myarray = {10, 20, 30};
  std::tuple<int,int,int> mytuple (10, 20, 30);

  std::tuple_element<0,decltype(myarray)>::type myelement;  // int myelement

  myelement = std::get<2>(myarray);
```

```
  std::get<2>(myarray) = std::get<0>(myarray);
  std::get<0>(myarray) = myelement;

  std::cout << "first element in myarray: " << std::get<0>(myarray) << "\n";
  std::cout << "first element in mytuple: " << std::get<0>(mytuple) << "\n";

  return 0;
}
```

Output

```
first element in myarray: 30
first element in mytuple: 10
```

relational operators (array)

形如：arrayA != arrayB、arrayA > arrayB；依此比较数组每个元素的大小关系。

```
（1）
template <class T, size_T N>
  bool operator == (const array <T, N>&lhs, const array <T, N>&rhs) ;
（2）
template <class T, size_T N>
  bool operator! = (const array <T, N>&lhs, const array <T, N>&rhs) ;
（3）
template <class T, size_T N>
  bool operator < (const array <T, N>&lhs, const array <T, N>&rhs) ;
（4）
template <class T, size_T N>
  bool operator <= (const array <T, N>&lhs, const array <T, N>&rhs) ;
（5）
template <class T, size_T N>
  bool operator> (const array <T, N>&lhs, const array <T, N>&rhs) ;
（6）
template <class T, size_T N>
  bool operator> = (const array <T, N>&lhs, const array <T, N>&rhs) ;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,5> a = {10, 20, 30, 40, 50};
  std::array<int,5> b = {10, 20, 30, 40, 50};
  std::array<int,5> c = {50, 40, 30, 20, 10};

  if (a==b) std::cout << "a and b are equal\n";
  if (b!=c) std::cout << "b and c are not equal\n";
  if (b<c) std::cout << "b is less than c\n";
  if (c>b) std::cout << "c is greater than b\n";
  if (a<=b) std::cout << "a is less than or equal to b\n";
  if (a>=b) std::cout << "a is greater than or equal to b\n";

  return 0;
}
```

Output

```
a and b are equal
b and c are not equal
b is less than c
```

```
c is greater than b
a is less than or equal to b
a is greater than or equal to b
```

vector

vector 是表示可以改变大小的数组的序列容器。

就像数组一样，vector 为它们的元素使用连续的存储位置，这意味着它们的元素也可以使用到其元素的常规指针上的偏移来访问，而且和数组一样高效。但是与数组不同的是，它们的大小可以动态地改变，它们的存储由容器自动处理。

在内部，vector 使用一个动态分配的数组来存储它们的元素。这个数组可能需要重新分配，以便在插入新元素时增加大小，这意味着分配一个新数组并将所有元素移动到其中。就处理时间而言，这是一个相对昂贵的任务，因此每次将元素添加到容器时矢量都不会重新分配。

相反，vector 容器可以分配一些额外的存储以适应可能的增长，并且因此容器可以具有比严格需要包含其元素（即，其大小）的存储更大的实际容量。库可以实现不同的策略的增长到内存使用和重新分配之间的平衡，但在任何情况下，再分配应仅在对数生长的间隔发生尺寸，使得在所述载体的末端各个元件的插入可以与提供分期常量时间复杂性。

因此，与数组相比，载体消耗更多的内存来交换管理存储和以有效方式动态增长的能力。

与其他动态序列容器（deques，lists 和 forward_lists ）相比，vector 非常有效地访问其元素（就像数组一样），并相对有效地从元素末尾添加或移除元素。对于涉及插入或移除除了结尾之外的位置的元素的操作，它们执行比其他位置更差的操作，并且具有比列表和 forward_lists 更不一致的迭代器和引用。

针对 vector 的各种常见操作的复杂度（效率）如下：

- 随机访问 - 常数 O(1)
- 在尾部增删元素 - 平摊（amortized）常数 O(1)}}
- 增删元素 - 至 vector 尾部的线性距离 O(n)}}

```
template < class T, class Alloc = allocator<T> > class vector;
```

vector::vector

（1）empty 容器构造函数（默认构造函数） 构造一个空的容器，没有元素。 （2）fill 构造函数 用 n 个元素构造一个容器。每个元素都是 val 的副本（如果提供）。 （3）范围（range）构造器 使用与[range，first，last]范围内的元素相同的顺序构造一个容器，其中的每个元素都是 emplace -从该范围内相应的元素构造而成。 （4）复制（copy）构造函数（并用分配器复制） 按照相同的顺序构造一个包含 x 中每个元素的副本的容器。 （5）移动（move）构造函数（和分配器移动） 构造一个获取 x 元素的容器。 如果指定了 alloc 并且与 x 的分配器不同，那么元素将被移动。否则，没有构建元素（他们的所有权直接转移）。 x 保持未指定但有效的状态。 （6）初始化列表构造函数 构造一个容器中的每个元件中的一个拷贝的 IL，以相同的顺序。

```
default (1)
explicit vector (const allocator_type& alloc = allocator_type());
fill (2)
explicit vector (size_type n);
         vector (size_type n, const value_type& val,
                 const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
  vector (InputIterator first, InputIterator last,
          const allocator_type& alloc = allocator_type());
copy (4)
vector (const vector& x);
vector (const vector& x, const allocator_type& alloc);
move (5)
vector (vector&& x);
vector (vector&& x, const allocator_type& alloc);
initializer list (6)
```

```
vector (initializer_list<value_type> il,
        const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    // constructors used in the same order as described above:
    std::vector<int> first;                // empty vector of ints
    std::vector<int> second(4, 100);       // four ints with value 100
    std::vector<int> third(second.begin(), second.end());// iterating through second
    std::vector<int> fourth(third);        // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));

    std::cout << "The contents of fifth are:";
    for(std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

vector::~vector

销毁容器对象。这将在每个包含的元素上调用 allocator_traits::destroy，并使用其分配器释放由矢量分配的所有存储容量。

```
~vector();
```

vector::operator=

将新内容分配给容器，替换其当前内容，并相应地修改其大小。

```
copy (1)
vector& operator= (const vector& x);
move (2)
vector& operator= (vector&& x);
initializer list (3)
vector& operator= (initializer_list<value_type> il);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> foo (3,0);
  std::vector<int> bar (5,0);

  bar = foo;
  foo = std::vector<int>();

  std::cout << "Size of foo: " << int(foo.size()) << '\n';
  std::cout << "Size of bar: " << int(bar.size()) << '\n';
  return 0;
}
```

Output

```
Size of foo: 0
Size of bar: 3
```

vector::begin

vector::end

vector::rbegin

vector::rend

vector::cbegin

vector::cend

vector::rcbegin

vector::rcend

vector::size

返回 vector 中元素的数量。

这是 vector 中保存的实际对象的数量，不一定等于其存储容量。

```
size_type size() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; i++) myints.push_back(i);
  std::cout << "1. size: " << myints.size() << '\n';

  myints.insert (myints.end(),10,100);
  std::cout << "2. size: " << myints.size() << '\n';

  myints.pop_back();
  std::cout << "3. size: " << myints.size() << '\n';

  return 0;
}
```

Output

```
0. size: 0
1. size: 10
2. size: 20
3. size: 19
```

vector::max_size

返回该 vector 可容纳的元素的最大数量。由于已知的系统或库实现限制，

这是容器可以达到的最大潜在大小，但容器无法保证能够达到该大小：在达到该大小之前的任何时间，仍然无法分配存储。

```
size_type max_size() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
```

```
  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  std::cout << "size: " << myvector.size() << "\n";
  std::cout << "capacity: " << myvector.capacity() << "\n";
  std::cout << "max_size: " << myvector.max_size() << "\n";
  return 0;
}
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::resize

调整容器的大小，使其包含 n 个元素。

如果 n 小于当前的容器 size，内容将被缩小到前 n 个元素，将其删除（并销毁它们）。

如果 n 大于当前容器 size，则通过在末尾插入尽可能多的元素以达到大小 n 来扩展内容。如果指定了 val，则新元素将初始化为 val 的副本，否则将进行值初始化。

如果 n 也大于当前的容器的 capacity（容量），分配的存储空间将自动重新分配。

注意这个函数通过插入或者删除元素的内容来改变容器的实际内容。

```
void resize (size_type n);
void resize (size_type n, const value_type& val);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++) myvector.push_back(i);

  myvector.resize(5);
  myvector.resize(8,100);
  myvector.resize(12);

  std::cout << "myvector contains:";
  for (int i=0;i<myvector.size();i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

vector::capacity

返回当前为 vector 分配的存储空间的大小，用元素表示。这个 capacity(容量)不一定等于 vector 的 size。它可以相等或更大，额外的空间允许适应增长，而不需要重新分配每个插入。

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

Example

```
#include <iostream>
#include <vector>
```

```
int main ()
{
  std::vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  std::cout << "size: " << (int) myvector.size() << '\n';
  std::cout << "capacity: " << (int) myvector.capacity() << '\n';
  std::cout << "max_size: " << (int) myvector.max_size() << '\n';
  return 0;
}
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::empty

返回 vector 是否为空（即，它的 size 是否为 0）

```
bool empty() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  int sum (0);

  for (int i=1;i<=10;i++) myvector.push_back(i);

  while (!myvector.empty())
  {
     sum += myvector.back();
     myvector.pop_back();
  }

  std::cout << "total: " << sum << '\n';

  return 0;
}
```

Output

```
total: 55
```

vector::reserve

请求 vector 容量至少足以包含 n 个元素。

如果 n 大于当前 vector 容量，则该函数使容器重新分配其存储容量，从而将其容量增加到 n（或更大）。

在所有其他情况下，函数调用不会导致重新分配，并且 vector 容量不受影响。

这个函数对 vector 大小没有影响，也不能改变它的元素。

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

Example

```
#include <iostream>
#include <vector>

int main ()
```

```
{
  std::vector<int>::size_type sz;

  std::vector<int> foo;
  sz = foo.capacity();
  std::cout << "making foo grow:\n";
  for (int i=0; i<100; ++i) {
    foo.push_back(i);
    if (sz!=foo.capacity()) {
      sz = foo.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }

  std::vector<int> bar;
  sz = bar.capacity();
  bar.reserve(100);   // this is the only difference with foo above
  std::cout << "making bar grow:\n";
  for (int i=0; i<100; ++i) {
    bar.push_back(i);
    if (sz!=bar.capacity()) {
      sz = bar.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }
  return 0;
}
```

Possible output

```
making foo grow:
capacity changed: 1
capacity changed: 2
capacity changed: 4
capacity changed: 8
capacity changed: 16
capacity changed: 32
capacity changed: 64
capacity changed: 128
making bar grow:
capacity changed: 100
```

vector::shrink*to*fit

要求容器减小其 capacity(容量)以适应其尺寸。

该请求是非绑定的，并且容器实现可以自由地进行优化，并且保持 capacity 大于其 size 的 vector。 这可能导致重新分配，但对矢量大小没有影响，并且不能改变其元素。

```
void shrink_to_fit();
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (100);
  std::cout << "1. capacity of myvector: " << myvector.capacity() << '\n';

  myvector.resize(10);
  std::cout << "2. capacity of myvector: " << myvector.capacity() << '\n';
```

```
  myvector.shrink_to_fit();
  std::cout << "3. capacity of myvector: " << myvector.capacity() << '\n';

  return 0;
}
```

Possible output

```
1. capacity of myvector: 100
2. capacity of myvector: 100
3. capacity of myvector: 10
```

vector::operator[]

vector::at

vector::front

vector::back

vector::data

vector::assign

将新内容分配给 vector，替换其当前内容，并相应地修改其大小。

在范围版本（1）中，新内容是从第一个和最后一个范围内的每个元素按相同顺序构造的元素。

在填充版本（2）中，新内容是 n 个元素，每个元素都被初始化为一个 val 的副本。

在初始化列表版本（3）中，新内容是以相同顺序作为初始化列表传递的值的副本。

所述内部分配器被用于（通过其性状），以分配和解除分配存储器如果重新分配发生。它也习惯于摧毁所有现有的元素，并构建新的元素。

```
range (1)
template <class InputIterator>
  void assign (InputIterator first, InputIterator last);
fill (2)
void assign (size_type n, const value_type& val);
initializer list (3)
void assign (initializer_list<value_type> il);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> first;
  std::vector<int> second;
  std::vector<int> third;

  first.assign (7,100);             // 7 ints with a value of 100

  std::vector<int>::iterator it;
  it=first.begin()+1;

  second.assign (it,first.end()-1); // the 5 central values of first

  int myints[] = {1776,7,4};
  third.assign (myints,myints+3);   // assigning from array.

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  std::cout << "Size of third: " << int (third.size()) << '\n';
```

```
  return 0;
}
```

Output

```
Size of first: 7
Size of second: 5
Size of third: 3
```

补充：vector::assign 与 vector::operator= 的区别：

1. vector::assign 实现源码

```
void assign(size_type __n, const _Tp& __val) { _M_fill_assign(__n, __val); }

template <class _Tp, class _Alloc>
void vector<_Tp, _Alloc>::_M_fill_assign(size_t __n, const value_type& __val)
{
  if (__n > capacity()) {
    vector<_Tp, _Alloc> __tmp(__n, __val, get_allocator());
    __tmp.swap(*this);
  }
  else if (__n > size()) {
    fill(begin(), end(), __val);
    _M_finish = uninitialized_fill_n(_M_finish, __n - size(), __val);
  }
  else
    erase(fill_n(begin(), __n, __val), end());
}
```

1. vector::operator= 实现源码

```
template <class _Tp, class _Alloc>
vector<_Tp,_Alloc>&
vector<_Tp,_Alloc>::operator=(const vector<_Tp, _Alloc>& __x)
{
  if (&__x != this) {
    const size_type __xlen = __x.size();
    if (__xlen > capacity()) {
      iterator __tmp = _M_allocate_and_copy(__xlen, __x.begin(), __x.end());
      destroy(_M_start, _M_finish);
      _M_deallocate(_M_start, _M_end_of_storage - _M_start);
      _M_start = __tmp;
      _M_end_of_storage = _M_start + __xlen;
    }
    else if (size() >= __xlen) {
      iterator __i = copy(__x.begin(), __x.end(), begin());
      destroy(__i, _M_finish);
    }
    else {
      copy(__x.begin(), __x.begin() + size(), _M_start);
      uninitialized_copy(__x.begin() + size(), __x.end(), _M_finish);
    }
    _M_finish = _M_start + __xlen;
  }
  return *this;
}
```

vector::push_back

在 vector 的最后一个元素之后添加一个新元素。val 的内容被复制（或移动）到新的元素。

这有效地将容器 size 增加了一个，如果新的矢量 size 超过了当前 vector 的 capacity，则导致所分配的存储空间自动重新分配。

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);

  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";

  return 0;
}
```

vector::pop_back

删除 vector 中的最后一个元素，有效地将容器 size 减少一个。

这破坏了被删除的元素。

```
void pop_back();
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  int sum (0);
  myvector.push_back (100);
  myvector.push_back (200);
  myvector.push_back (300);

  while (!myvector.empty())
  {
    sum+=myvector.back();
    myvector.pop_back();
  }

  std::cout << "The elements of myvector add up to " << sum << '\n';

  return 0;
}
```

Output

```
The elements of myvector add up to 600
```

vector::insert

通过在指定位置的元素之前插入新元素来扩展该 vector，通过插入元素的数量有效地增加容器大小。 这会导致分配的存储空间自动重新分配，只有在新的 vector 的 size 超过当前的 vector 的 capacity 的情况下。

由于 vector 使用数组作为其基础存储，因此除了将元素插入到 vector 末尾之后，或 vector 的 begin 之前，其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

```
single element (1)
iterator insert (const_iterator position, const value_type& val);
fill (2)
iterator insert (const_iterator position, size_type n, const value_type& val);
range (3)
template <class InputIterator>
iterator insert (const_iterator position, InputIterator first, InputIterator last);
move (4)
iterator insert (const_iterator position, value_type&& val);
initializer list (5)
iterator insert (const_iterator position, initializer_list<value_type> il);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (3,100);
  std::vector<int>::iterator it;

  it = myvector.begin();
  it = myvector.insert ( it , 200 );

  myvector.insert (it,2,300);

  // "it" no longer valid, get a new one:
  it = myvector.begin();

  std::vector<int> anothervector (2,400);
  myvector.insert (it+2,anothervector.begin(),anothervector.end());

  int myarray [] = { 501,502,503 };
  myvector.insert (myvector.begin(), myarray, myarray+3);

  std::cout << "myvector contains:";
  for (it=myvector.begin(); it<myvector.end(); it++)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 501 502 503 300 300 400 400 200 100 100 100
```

补充：insert 迭代器野指针错误：

```
int main()
{
	std::vector<int> v(5, 0);
	std::vector<int>::iterator vi;

  // 获取 vector 第一个元素的迭代器
	vi = v.begin();
```

```
    // push_back 插入元素之后可能会因为 push_back 的骚操作（创建一个新 vector 把旧 vector 的值复制到新 vector），导致 vector 迭代器 iterator 的指针变成野指针，而导致 insert 出错
    v.push_back(10);

    v.insert(vi, 2, 300);

    return 0;
}
```

改正：应该把 `vi = v.begin();` 放到 `v.push_back(10);` 后面

vector::erase

从 vector 中删除单个元素（position）或一系列元素（[first，last）），

这有效地减少了被去除的元素的数量，从而破坏了容器的大小。

由于 vector 使用一个数组作为其底层存储，所以删除除 vector 结束位置之后，或 vector 的 begin 之前的元素外，将导致容器将段被擦除后的所有元素重新定位到新的位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

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

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i=1; i<=10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);

  // erase the first 3 elements:
  myvector.erase (myvector.begin(),myvector.begin()+3);

  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); ++i)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 4 5 7 8 9 10
```

vector::swap

通过 x 的内容交换容器的内容，x 是另一个相同类型的 vector 对象。尺寸可能不同。

在调用这个成员函数之后，这个容器中的元素是那些在调用之前在 x 中的元素，而 x 的元素是在这个元素中的元素。所有迭代器，引用和指针对交换对象保持有效。

请注意，非成员函数存在具有相同名称的交换，并使用与此成员函数相似的优化来重载该算法。

```
void swap (vector& x);
```

Example

```
#include <iostream>
#include <vector>
```

```
int main ()
{
  std::vector<int> foo (3,100);   // three ints with a value of 100
  std::vector<int> bar (5,200);   // five ints with a value of 200

  foo.swap(bar);

  std::cout << "foo contains:";
  for (unsigned i=0; i<foo.size(); i++)
    std::cout << ' ' << foo[i];
  std::cout << '\n';

  std::cout << "bar contains:";
  for (unsigned i=0; i<bar.size(); i++)
    std::cout << ' ' << bar[i];
  std::cout << '\n';

  return 0;
}
```

Output

```
foo contains: 200 200 200 200 200
bar contains: 100 100 100
```

vector::clear

从 vector 中删除所有的元素（被销毁），留下 size 为 0 的容器。

不保证重新分配，并且由于调用此函数， vector 的 capacity 不保证发生变化。强制重新分配的典型替代方法是使用 swap：vector<T>().swap(x); // clear x reallocating

```
void clear() noexcept;
```

Example

```
#include <iostream>
#include <vector>

void printVector(const std::vector<int> &v)
{
	for (auto it = v.begin(); it != v.end(); ++it)
	{
		std::cout << *it << ' ';
	}
	std::cout << std::endl;
}

int main()
{
	std::vector<int> v1(5, 50);

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	v1.clear();

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	v1.push_back(11);
```

```
	v1.push_back(22);

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	return 0;
}
```

Output

```
50 50 50 50 50
v1 size  = 5
v1 capacity  = 5

v1 size  = 0
v1 capacity  = 5
11 22
v1 size  = 2
v1 capacity  = 5
```

vector::emplace

通过在 position 位置处插入新元素 args 来扩展容器。这个新元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

分配存储空间的自动重新分配发生在新的 vector 的 size 超过当前向量容量的情况下。

由于 vector 使用数组作为其基础存储，因此除了将元素插入到 vector 末尾之后，或 vector 的 begin 之前，其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。插入一个类似的成员函数，将现有对象复制或移动到容器中。

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector = {10,20,30};

  auto it = myvector.emplace ( myvector.begin()+1, 100 );
  myvector.emplace ( it, 200 );
  myvector.emplace ( myvector.end(), 300 );

  std::cout << "myvector contains:";
  for (auto& x: myvector)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 10 200 100 20 30 300
```

vector::emplace_back

在 vector 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新元素是用 args 作为构造函数的参数来构造的。

这有效地将容器大小增加了一个，如果新的矢量大小超过了当前的 vector 容量，则导致所分配的存储空间自动重新分配。

该元素是通过调用 allocator_traits :: construct 来转换 args 来创建的。

与 push_back 相比，emplace_back 可以避免额外的复制和移动操作。

```
template <class... Args>
  void emplace_back (Args&&... args);
```

Example

```
#include <vector>
#include <string>
#include <iostream>

struct President
{
    std::string name;
    std::string country;
    int year;

    President(std::string p_name, std::string p_country, int p_year)
        : name(std::move(p_name)), country(std::move(p_country)), year(p_year)
    {
        std::cout << "I am being constructed.\n";
    }
    President(President&& other)
        : name(std::move(other.name)), country(std::move(other.country)), year(other.year)
    {
        std::cout << "I am being moved.\n";
    }
    President& operator=(const President& other) = default;
};

int main()
{
    std::vector<President> elections;
    std::cout << "emplace_back:\n";
    elections.emplace_back("Nelson Mandela", "South Africa", 1994);

    std::vector<President> reElections;
    std::cout << "\npush_back:\n";
    reElections.push_back(President("Franklin Delano Roosevelt", "the USA", 1936));

    std::cout << "\nContents:\n";
    for (President const& president: elections) {
        std::cout << president.name << " was elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
    for (President const& president: reElections) {
        std::cout << president.name << " was re-elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
}
```

Output

```
emplace_back:
I am being constructed.

push_back:
```

```
I am being constructed.
I am being moved.

Contents:
Nelson Mandela was elected president of South Africa in 1994.
Franklin Delano Roosevelt was re-elected president of the USA in 1936.
```

vector::get_allocator

返回与 vector 关联的构造器对象的副本。

```
allocator_type get_allocator() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  int * p;
  unsigned int i;

  // allocate an array with space for 5 elements using vector's allocator:
  p = myvector.get_allocator().allocate(5);

  // construct values in-place on the array:
  for (i=0; i<5; i++) myvector.get_allocator().construct(&p[i],i);

  std::cout << "The allocated array contains:";
  for (i=0; i<5; i++) std::cout << ' ' << p[i];
  std::cout << '\n';

  // destroy and deallocate:
  for (i=0; i<5; i++) myvector.get_allocator().destroy(&p[i]);
  myvector.get_allocator().deallocate(p,5);

  return 0;
}
```

Output

```
The allocated array contains: 0 1 2 3 4
```

注意：deallocate 和 destory 的关系：

deallocate 实现的源码：

```
template <class T>
inline void _deallocate(T* buffer)
{
      ::operator delete(buffer);     //为什么不用 delete [] ?  ,operator delete 区别于 delete
                                     //operator delete   是一个底层操作符
}
```

destory：

```
template <class T>
inline void _destory(T *ptr)
{
      ptr->~T();
}
```

destory 负责调用类型的析构函数，销毁相应内存上的内容（但销毁后内存地址仍保留）

deallocate 负责释放内存（此时相应内存中的值在此之前应调用 destory 销毁，将内存地址返回给系统，代表这部分地址使用引用-1）

relational operators (vector)
swap (vector)
vector
deque

deque（['dek]）（双端队列）是 double-ended queue 的一个不规则缩写。deque 是具有动态大小的序列容器，可以在两端（前端或后端）扩展或收缩。

特定的库可以以不同的方式实现 deques，通常作为某种形式的动态数组。但是在任何情况下，它们都允许通过随机访问迭代器直接访问各个元素，通过根据需要扩展和收缩容器来自动处理存储。

因此，它们提供了类似于 vector 的功能，但是在序列的开始部分也可以高效地插入和删除元素，而不仅仅是在结尾。但是，与 vector 不同，deques 并不保证将其所有元素存储在连续的存储位置：deque 通过偏移指向另一个元素的指针访问元素会导致未定义的行为。

两个 vector 和 deques 提供了一个非常相似的接口，可以用于类似的目的，但内部工作方式完全不同：虽然 vector 使用单个数组需要偶尔重新分配以增长，但是 deque 的元素可以分散在不同的块的容器，容器在内部保存必要的信息以提供对其任何元素的持续时间和统一的顺序接口（通过迭代器）的直接访问。因此，deques 在内部比 vector 更复杂一点，但是这使得他们在某些情况下更有效地增长，尤其是在重新分配变得更加昂贵的很长序列的情况下。

对于频繁插入或删除开始或结束位置以外的元素的操作，deques 表现得更差，并且与列表和转发列表相比，迭代器和引用的一致性更低。

deque 上常见操作的复杂性（效率）如下：

- 随机访问 - 常数 O(1)
- 在结尾或开头插入或移除元素 - 摊销不变 O(1)
- 插入或移除元素 - 线性 O(n)

```
template < class T, class Alloc = allocator<T> > class deque;
```

deque::deque

构造一个 deque 容器对象，根据所使用的构造函数版本初始化它的内容：

Example

```
#include <iostream>
#include <deque>

int main ()
{
  unsigned int i;

  // constructors used in the same order as described above:
  std::deque<int> first;                                // empty deque of ints
  std::deque<int> second (4,100);                       // four ints with value 100
  std::deque<int> third (second.begin(),second.end());  // iterating through second
  std::deque<int> fourth (third);                       // a copy of third

  // the iterator constructor can be used to copy arrays:
  int myints[] = {16,2,77,29};
  std::deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';
```

```
  return 0;
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

deque::push_back

在当前的最后一个元素之后，在 deque 容器的末尾添加一个新元素。val 的内容被复制（或移动）到新的元素。

这有效地增加了一个容器的大小。

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    mydeque.push_back (myint);
  } while (myint);

  std::cout << "mydeque stores " << (int) mydeque.size() << " numbers.\n";

  return 0;
}
```

deque::push_front

在 deque 容器的开始位置插入一个新的元素，位于当前的第一个元素之前。val 的内容被复制（或移动）到插入的元素。

这有效地增加了一个容器的大小。

```
void push_front (const value_type& val);
void push_front (value_type&& val);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque (2,100);      // two ints with a value of 100
  mydeque.push_front (200);
  mydeque.push_front (300);

  std::cout << "mydeque contains:";
  for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output

```
300 200 100 100
```

deque::pop_back

删除 deque 容器中的最后一个元素，有效地将容器大小减少一个。
这破坏了被删除的元素。

```
void pop_back();
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque;
  int sum (0);
  mydeque.push_back (10);
  mydeque.push_back (20);
  mydeque.push_back (30);

  while (!mydeque.empty())
  {
    sum+=mydeque.back();
    mydeque.pop_back();
  }

  std::cout << "The elements of mydeque add up to " << sum << '\n';

  return 0;
}
```

Output

```
The elements of mydeque add up to 60
```

deque::pop_front

删除 deque 容器中的第一个元素，有效地减小其大小。
这破坏了被删除的元素。

```
void pop_front();
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque;

  mydeque.push_back (100);
  mydeque.push_back (200);
  mydeque.push_back (300);

  std::cout << "Popping out the elements in mydeque:";
  while (!mydeque.empty())
  {
    std::cout << ' ' << mydeque.front();
    mydeque.pop_front();
  }

  std::cout << "\nThe final size of mydeque is " << int(mydeque.size()) << '\n';
```

```
  return 0;
}
```

Output

```
Popping out the elements in mydeque: 100 200 300
The final size of mydeque is 0
```

deque::emplace_front

在 deque 的开头插入一个新的元素，就在其当前的第一个元素之前。这个新的元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。

存在一个类似的成员函数 push_front，它可以将现有对象复制或移动到容器中。

```
template <class... Args>
  void emplace_front (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque = {10,20,30};

  mydeque.emplace_front (111);
  mydeque.emplace_front (222);

  std::cout << "mydeque contains:";
  for (auto& x: mydeque)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mydeque contains: 222 111 10 20 30
```

deque::emplace_back

在 deque 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新的元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。

存在一个类似的成员函数 push_back，它可以将现有对象复制或移动到容器中

```
template <class... Args>
  void emplace_back (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque = {10,20,30};

  mydeque.emplace_back (100);
  mydeque.emplace_back (200);

  std::cout << "mydeque contains:";
```

```
  for (auto& x: mydeque)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mydeque contains: 10 20 30 100 200
```

forward_list

forward_list（单向链表）是序列容器，允许在序列中的任何地方进行恒定的时间插入和擦除操作。

forward_list（单向链表）被实现为单链表; 单链表可以将它们包含的每个元素存储在不同和不相关的存储位置中。通过关联到序列中下一个元素的链接的每个元素来保留排序。forward_list 容器和列表

之间的主要设计区别容器是第一个内部只保留一个到下一个元素的链接，而后者每个元素保留两个链接：一个指向下一个元素，一个指向前一个元素，允许在两个方向上有效的迭代，但是每个元素消耗额外的存储空间并且插入和移除元件的时间开销略高。因此，forward_list 对象比列表对象更有效率，尽管它们只能向前迭代。

与其他基本的标准序列容器（array，vector 和 deque），forward_list 通常在插入，提取和移动容器内任何位置的元素方面效果更好，因此也适用于密集使用这些元素的算法，如排序算法。

的主要缺点修饰符 Modifiers S 和列表相比这些其它序列容器 s 是说，他们缺乏可以通过位置的元素的直接访问; 例如，要访问 forward_list 中的第六个元素，必须从开始位置迭代到该位置，这需要在这些位置之间的线性时间。它们还消耗一些额外的内存来保持与每个元素相关联的链接信息（这可能是大型小元素列表的重要因素）。

该修饰符 Modifiersclass 模板的设计考虑到效率：按照设计，它与简单的手写 C 型单链表一样高效，实际上是唯一的标准容器，为了效率的考虑故意缺少尺寸成员函数：由于其性质作为一个链表，具有一个需要一定时间的大小的成员将需要它保持一个内部计数器的大小（如列表所示）。这会消耗一些额外的存储空间，并使插入和删除操作效率稍低。要获取 forward_list 对象的大小，可以使用距离算法的开始和结束，这是一个需要线性时间的操作。

forward_list::forward_list

```
default (1)
explicit forward_list (const allocator_type& alloc = allocator_type());
fill (2)
explicit forward_list (size_type n);
explicit forward_list (size_type n, const value_type& val,
                       const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
  forward_list (InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type());
copy (4)
forward_list (const forward_list& fwdlst);
forward_list (const forward_list& fwdlst, const allocator_type& alloc);
move (5)
forward_list (forward_list&& fwdlst);
forward_list (forward_list&& fwdlst, const allocator_type& alloc);
initializer list (6)
forward_list (initializer_list<value_type> il,
              const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <forward_list>
```

```
int main ()
{
  // constructors used in the same order as described above:

  std::forward_list<int> first;                          // default: empty
  std::forward_list<int> second (3,77);                  // fill: 3 seventy-sevens
  std::forward_list<int> third (second.begin(), second.end()); // range initialization
  std::forward_list<int> fourth (third);                 // copy constructor
  std::forward_list<int> fifth (std::move(fourth));  // move ctor. (fourth wasted)
  std::forward_list<int> sixth = {3, 52, 25, 90};    // initializer_list constructor

  std::cout << "first:" ; for (int& x: first)  std::cout << " " << x; std::cout << '\n';
  std::cout << "second:"; for (int& x: second) std::cout << " " << x; std::cout << '\n';
  std::cout << "third:";  for (int& x: third)  std::cout << " " << x; std::cout << '\n';
  std::cout << "fourth:"; for (int& x: fourth) std::cout << " " << x; std::cout << '\n';
  std::cout << "fifth:";  for (int& x: fifth)  std::cout << " " << x; std::cout << '\n';
  std::cout << "sixth:";  for (int& x: sixth)  std::cout << " " << x; std::cout << '\n';

  return 0;
}
```

Possible output

```
forward_list constructor examples:
first:
second: 77 77 77
third: 77 77 77
fourth:
fifth: 77 77 77
sixth: 3 52 25 90
```

forward_list::~forward_list

forward_list::before_begin

返回指向容器中第一个元素之前的位置的迭代器。

返回的迭代器不应被解除引用：它是为了用作成员函数的参数 emplace_after，insert_after，erase_after 或 splice_after，指定序列，其中执行该动作的位置的开始位置。

```
      iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
  std::forward_list<int> mylist = {20, 30, 40, 50};

  mylist.insert_after ( mylist.before_begin(), 11 );

  std::cout << "mylist contains:";
  for ( int& x: mylist ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mylist contains: 11 20 30 40 50
```

forward_list::cbefore_begin

返回指向容器中第一个元素之前的位置的 const_iterator。

一个常量性是指向常量内容的迭代器。这个迭代器可以增加和减少（除非它本身也是 const），就像 forward_list::before_begin 返回的迭代器一样，但不能用来修改它指向的内容。

返回的价值不得解除引用。

```
const_iterator cbefore_begin() const noexcept;
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
  std::forward_list<int> mylist = {77, 2, 16};

  mylist.insert_after ( mylist.cbefore_begin(), 19 );

  std::cout << "mylist contains:";
  for ( int& x: mylist ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mylist contains: 19 77 2 16
```

list

stack

queue

priority_queue

set

multiset

map

map 是关联容器，按照特定顺序存储由 key value (键值) 和 mapped value (映射值) 组合形成的元素。

在映射中，键值通常用于对元素进行排序和唯一标识，而映射的值存储与此键关联的内容。该类型的键和映射的值可能不同，并且在部件类型被分组在一起 VALUE_TYPE，这是一种对类型结合两种：

```
typedef pair<const Key, T> value_type;
```

在内部，映射中的元素总是按照由其内部比较对象（比较类型）指示的特定的严格弱排序标准按键排序。映射容器通常比 unordered_map 容器慢，以通过它们的键来访问各个元素，但是它们允许基于它们的顺序对子集进行直接迭代。 在该映射值地图可以直接通过使用其相应的键来访问括号运算符（（操作符[] ）。 映射通常如实施

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

map::map

构造一个映射容器对象，根据所使用的构造器版本初始化其内容：

（1）空容器构造函数（默认构造函数）

构造一个空的容器，没有元素。

（2）范围构造函数

构造具有一样多的元素的范围内的容器[第一，最后一个），其中每个元件布设构造的从在该范围内它的相应的元件。

（3）复制构造函数（并用分配器复制）

使用 x 中的每个元素的副本构造一个容器。

（4）移动构造函数（并与分配器一起移动）

构造一个获取 x 元素的容器。 如果指定了 alloc 并且与 x 的分配器不同，那么元素将被移动。否则，没有构建元素（他们的所有权直接转移）。 x 保持未指定但有效的状态。

（5）初始化列表构造函数

用 il 中的每个元素的副本构造一个容器。

```
empty (1)
explicit map (const key_compare& comp = key_compare(),
              const allocator_type& alloc = allocator_type());
explicit map (const allocator_type& alloc);
range (2)
template <class InputIterator>
  map (InputIterator first, InputIterator last,
       const key_compare& comp = key_compare(),
       const allocator_type& = allocator_type());
copy (3)
map (const map& x);
map (const map& x, const allocator_type& alloc);
move (4)
map (map&& x);
map (map&& x, const allocator_type& alloc);
initializer list (5)
map (initializer_list<value_type> il,
     const key_compare& comp = key_compare(),
     const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  std::map<char,int> second (first.begin(),first.end());

  std::map<char,int> third (second);

  std::map<char,int,classcomp> fourth;                 // class as Compare
```

```
  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare

  return 0;
}
```

map::begin

返回引用 map 容器中第一个元素的迭代器。
由于 map 容器始终保持其元素的顺序，所以开始指向遵循容器排序标准的元素。
如果容器是空的，则返回的迭代器值不应被解除引用。

```
      iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['b'] = 100;
  mymap['a'] = 200;
  mymap['c'] = 300;

  // show content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 200
b => 100
c => 300
```

map::key_comp

返回容器用于比较键的比较对象的副本。

```
key_compare key_comp() const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  std::map<char,int>::key_compare mycomp = mymap.key_comp();

  mymap['a']=100;
  mymap['b']=200;
  mymap['c']=300;

  std::cout << "mymap contains:\n";

  char highest = mymap.rbegin()->first;     // key value of last element
```

```
  std::map<char,int>::iterator it = mymap.begin();
  do {
    std::cout << it->first << " => " << it->second << '\n';
  } while ( mycomp((*it++).first, highest) );

  std::cout << '\n';

  return 0;
}
```

Output

```
mymap contains:
a => 100
b => 200
c => 300
```

map::value_comp

返回可用于比较两个元素的比较对象，以获取第一个元素的键是否在第二个元素之前。

```
value_compare value_comp() const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['x']=1001;
  mymap['y']=2002;
  mymap['z']=3003;

  std::cout << "mymap contains:\n";

  std::pair<char,int> highest = *mymap.rbegin();          // last element

  std::map<char,int>::iterator it = mymap.begin();
  do {
    std::cout << it->first << " => " << it->second << '\n';
  } while ( mymap.value_comp()(*it++, highest) );

  return 0;
}
```

Output

```
mymap contains:
x => 1001
y => 2002
z => 3003
```

map::find

在容器中搜索具有等于 k 的键的元素，如果找到则返回一个迭代器，否则返回 map::end 的迭代器。

如果容器的比较对象自反地返回假（即，不管元素作为参数传递的顺序），则两个 key 被认为是等同的。

另一个成员函数 map::count 可以用来检查一个特定的键是否存在。

```
      iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>
```

```
int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;

  mymap['a']=50;
  mymap['b']=100;
  mymap['c']=150;
  mymap['d']=200;

  it = mymap.find('b');
  if (it != mymap.end())
    mymap.erase (it);

  // print content:
  std::cout << "elements in mymap:" << '\n';
  std::cout << "a => " << mymap.find('a')->second << '\n';
  std::cout << "c => " << mymap.find('c')->second << '\n';
  std::cout << "d => " << mymap.find('d')->second << '\n';

  return 0;
}
```

Output

```
elements in mymap:
a => 50
c => 150
d => 200
```

map::count

在容器中搜索具有等于 k 的键的元素，并返回匹配的数量。

由于地图容器中的所有元素都是唯一的，因此该函数只能返回 1（如果找到该元素）或返回零（否则）。

如果容器的比较对象自反地返回错误（即，不管按键作为参数传递的顺序），则两个键被认为是等同的。

```
size_type count (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  char c;

  mymap ['a']=101;
  mymap ['c']=202;
  mymap ['f']=303;

  for (c='a'; c<'h'; c++)
  {
    std::cout << c;
    if (mymap.count(c)>0)
      std::cout << " is an element of mymap.\n";
    else
      std::cout << " is not an element of mymap.\n";
  }
```

```
  return 0;
}
```

Output

```
a is an element of mymap.
b is not an element of mymap.
c is an element of mymap.
d is not an element of mymap.
e is not an element of mymap.
f is an element of mymap.
g is not an element of mymap.
```

map::lower_bound

将迭代器返回到下限

返回指向容器中第一个元素的迭代器，该元素的键不会在 k 之前出现（即，它是等价的或者在其后）。

该函数使用其内部比较对象（key_comp）来确定这一点，将迭代器返回到 key_comp（element_key，k）将返回 false 的第一个元素。

如果 map 类用默认的比较类型（less）实例化，则函数返回一个迭代器到第一个元素，其键不小于 k。

一个类似的成员函数 upper_bound 具有相同的行为 lower_bound，除非映射包含一个 key 值等于 k 的元素：在这种情况下，lower_bound 返回指向该元素的迭代器，而 upper_bound 返回指向下一个元素的迭代器。

```
      iterator lower_bound (const key_type& k);
const_iterator lower_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;

  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)

  mymap.erase(itlow,itup);        // erases [itlow,itup)

  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 20
e => 100
```

map::upper_bound

将迭代器返回到上限

返回一个指向容器中第一个元素的迭代器，它的关键字被认为是在 k 之后。

该函数使用其内部比较对象（key_comp）来确定这一点，将迭代器返回到 key_comp（k，element_key）将返回 true 的第一个元素。

如果 map 类用默认的比较类型（less）实例化，则函数返回一个迭代器到第一个元素，其键大于 k。

类似的成员函数 lower_bound 具有与 upper_bound 相同的行为，除了 map 包含一个元素，其键值等于 k：在这种情况下，lower_bound 返回指向该元素的迭代器，而 upper_bound 返回指向下一个元素的迭代器。

```
      iterator upper_bound (const key_type& k);
const_iterator upper_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;

  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)

  mymap.erase(itlow,itup);        // erases [itlow,itup)

  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 20
e => 100
```

map::equal_range

获取相同元素的范围

返回包含容器中所有具有与 k 等价的键的元素的范围边界。 由于地图容器中的元素具有唯一键，所以返回的范围最多只包含一个元素。

如果没有找到匹配，则返回的范围具有零的长度，与两个迭代器指向具有考虑去后一个密钥对所述第一元件 k 根据容器的内部比较对象（key_comp）。如果容器的比较对象返回 false，则两个键被认为是等价的。

```
pair<const_iterator,const_iterator> equal_range (const key_type& k) const;
pair<iterator,iterator>             equal_range (const key_type& k);
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['a']=10;
```

```
  mymap['b']=20;
  mymap['c']=30;

  std::pair<std::map<char,int>::iterator,std::map<char,int>::iterator> ret;
  ret = mymap.equal_range('b');

  std::cout << "lower bound points to: ";
  std::cout << ret.first->first << " => " << ret.first->second << '\n';

  std::cout << "upper bound points to: ";
  std::cout << ret.second->first << " => " << ret.second->second << '\n';

  return 0;
}
```

Output

```
lower bound points to: 'b' => 20
upper bound points to: 'c' => 30
```

multimap

无序容器（Unordered Container）：unordered_set、unordered_multiset、unordered_map、unordered_multimap

包括：

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

都是以哈希表实现的。

unordered_set、unodered_multiset 结构：

unordered_map、unodered_multimap 结构：

unordered_set

unordered_multiset

unordered_map

unordered_multimap

tuple

元组是一个能够容纳元素集合的对象。每个元素可以是不同的类型。

```
template <class... Types> class tuple;
```

Example

```
#include <iostream>     // std::cout
#include <tuple>        // std::tuple, std::get, std::tie, std::ignore

int main ()
{
  std::tuple<int,char> foo (10,'x');
  auto bar = std::make_tuple ("test", 3.1, 14, 'y');

  std::get<2>(bar) = 100;                                   // access element
```

```
  int myint; char mychar;

  std::tie (myint, mychar) = foo;                              // unpack elements
  std::tie (std::ignore, std::ignore, myint, mychar) = bar;  // unpack (with ignore)

  mychar = std::get<3>(bar);

  std::get<0>(foo) = std::get<2>(bar);
  std::get<1>(foo) = mychar;

  std::cout << "foo contains: ";
  std::cout << std::get<0>(foo) << ' ';
  std::cout << std::get<1>(foo) << '\n';

  return 0;
}
```

Output

```
foo contains: 100 y
```

tuple::tuple

构建一个 tuple（元组）对象。

这涉及单独构建其元素，初始化取决于调用的构造函数形式：

（1）默认的构造函数

构建一个 元组对象的元素值初始化。

（2）复制/移动构造函数

该对象使用 tpl 的内容进行初始化 元组目的。tpl 的相应元素被传递给每个元素的构造函数。

（3）隐式转换构造函数

同上。tpl 中的 所有类型都可以隐含地转换为构造中它们各自元素的类型元组 目的。

（4）初始化构造函数 用 elems 中的相应元素初始化每个元素。elems 的相应元素被传递给每个元素的构造函数。

（5）对转换构造函数

该对象有两个对应于 pr.first 和的元素 pr.second。PR 中的所有类型都应该隐含地转换为其中各自元素的类型元组 目的。

（6）分配器版本

和上面的版本一样，除了每个元素都是使用 allocator alloc 构造的。

```
default (1)
constexpr tuple();
copy / move (2)
tuple (const tuple& tpl) = default;
tuple (tuple&& tpl) = default;
implicit conversion (3)
template <class... UTypes>
  tuple (const tuple<UTypes...>& tpl);
template <class... UTypes>
  tuple (tuple<UTypes...>&& tpl);
initialization (4)
explicit tuple (const Types&... elems);
template <class... UTypes>
  explicit tuple (UTypes&&... elems);
conversion from pair (5)
template <class U1, class U2>
  tuple (const pair<U1,U2>& pr);
```

```
template <class U1, class U2>
  tuple (pair<U1,U2>&& pr);
allocator (6)
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple&& tpl);
template<class Alloc,class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple<UTypes...>& tpl);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple<UTypes...>&& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const Types&... elems);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, UTypes&&... elems);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, const pair<U1,U2>& pr);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, pair<U1,U2>&& pr);
```

Example

```
#include <iostream>     // std::cout
#include <utility>      // std::make_pair
#include <tuple>        // std::tuple, std::make_tuple, std::get

int main ()
{
  std::tuple<int,char> first;                             // default
  std::tuple<int,char> second (first);                    // copy
  std::tuple<int,char> third (std::make_tuple(20,'b'));   // move
  std::tuple<long,char> fourth (third);                   // implicit conversion
  std::tuple<int,char> fifth (10,'a');                    // initialization
  std::tuple<int,char> sixth (std::make_pair(30,'c'));    // from pair / move

  std::cout << "sixth contains: " << std::get<0>(sixth);
  std::cout << " and " << std::get<1>(sixth) << '\n';

  return 0;
}
```

Output

```
sixth contains: 30 and c
```

pair

这个类把一对值（values）结合在一起，这些值可能是不同的类型（T1 和 T2）。每个值可以被公有的成员变量 first、second 访问。

pair 是 tuple（元组）的一个特例。

pair 的实现是一个结构体，主要的两个成员变量是 first second 因为是使用 struct 不是 class，所以可以直接使用 pair 的成员变量。

应用：

- 可以将两个类型数据组合成一个如 map<key,value>
- 当某个函数需要两个返回值时

```
template <class T1, class T2> struct pair;
```

pair::pair

构建一个 pair 对象。

这涉及到单独构建它的两个组件对象，初始化依赖于调用的构造器形式：

（1）默认的构造函数

构建一个 对对象的元素值初始化。

（2）复制/移动构造函数（和隐式转换）

该对象被初始化为 pr 的内容 对目的。pr 的相应成员被传递给每个成员的构造函数。

（3）初始化构造函数

会员 第一是由一个和成员构建的第二与 b。

（4）分段构造

构造成员 first 和 second 到位，传递元素 first_args 作为参数的构造函数 first，和元素 second_args 到的构造函数 second 。

```
default (1)
constexpr pair();
copy / move (2)
template<class U, class V> pair (const pair<U,V>& pr);
template<class U, class V> pair (pair<U,V>&& pr);
pair (const pair& pr) = default;
pair (pair&& pr) = default;
initialization (3)
pair (const first_type& a, const second_type& b);
template<class U, class V> pair (U&& a, V&& b);
piecewise (4)
template <class... Args1, class... Args2>
  pair (piecewise_construct_t pwc, tuple<Args1...> first_args,
                                   tuple<Args2...> second_args);
```

Example

```
#include <utility>      // std::pair, std::make_pair
#include <string>       // std::string
#include <iostream>     // std::cout

int main () {
  std::pair <std::string,double> product1;                     // default constructor
  std::pair <std::string,double> product2 ("tomatoes",2.30);   // value init
  std::pair <std::string,double> product3 (product2);          // copy constructor
  product1 = std::make_pair(std::string("lightbulbs"),0.99);   // using make_pair (move)
  product2.first = "shoes";                  // the type of first is string
  product2.second = 39.90;                   // the type of second is double

  std::cout << "The price of " << product1.first << " is $" << product1.second << '\n';
  std::cout << "The price of " << product2.first << " is $" << product2.second << '\n';
  std::cout << "The price of " << product3.first << " is $" << product3.second << '\n';
  return 0;
}
```

Output

```
The price of lightbulbs is $0.99
The price of shoes is $39.9
The price of tomatoes is $2.3
```