C++
Standard Template Library

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

February 10, 2013

Outline

Outline

vector

- Vectors are sequence containers representing arrays that can change in size.
- Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
- Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Vector

Container properties:

- Sequence: Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.
- Dynamic array Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.
- Allocator-aware The container uses an allocator object to dynamically handle its storage needs.

Vector: Constructors

- Default

```
explicit
vector (const allocator_type& alloc = allocator_type());
```

- Fill constructor

```
explicit
vector (size_type n, const value_type& val = value_type(),
        const allocator_type& alloc = allocator_type());
```

- Range constructor

```
template <class InputIterator>
vector (InputIterator first, InputIterator last,
   const allocator_type& alloc = allocator_type());
```

- Copy constructor

```
  vector (const vector& x);
```

Vector: Constructors

```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  unsigned int i;

  // empty vector of ints
  std::vector<int> first;
  // four ints with value 100
  std::vector<int> second (4,100);
  // iterating through second
  std::vector<int> third (second.begin(),second.end());
  // a copy of third
  std::vector<int> fourth (third);


  ...
  }
```

Vector: Constructors

```
// constructing vectors
#include <iostream>
#include <vector>

int main ()
{
  ...

  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';
  // The contents of fifth are: 16 2 77 29

  return 0;
}
```

Vector: Iterators

`begin`	Return iterator to beginning
`end`	Return iterator to end
`rbegin`	Return reverse iterator to reverse beginning
`rend`	Return reverse iterator to reverse end
`cbegin`	Return `const_iterator` to beginning
`cend`	Return `const_iterator` to end
`crbegin`	Return `const_reverse_iterator` to reverse beginning
`crend`	Return `const_reverse_iterator` to reverse end

Vector: Iterators

```
// vector::begin/end
#include <iostream>
#include <vector>

int main ()
{
   std::vector<int> myvector;
   for (int i=1; i<=5; i++) myvector.push_back(i);
   std::cout << "myvector contains:";
   for (std::vector<int>::iterator it = myvector.begin() ;
        it != myvector.end();
        ++it)
     std::cout << ' ' << *it;
   std::cout << '\n';

   // myvector contains: 1 2 3 4 5

   return 0;
}
```

Vector: Iterators

```
// vector::rbegin/rend
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (5);  // 5 default-constructed ints
  std::vector<int>::reverse_iterator rit = myvector.rbegin();
  int i=0;

  for (rit = myvector.rbegin();
       rit!= myvector.rend();
       ++rit)
    *rit = ++i;

  ...
}
```

Vector: Iterators

```
// vector::rbegin/rend
#include <iostream>
#include <vector>

int main ()
{
  ...

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

  //myvector contains: 5 4 3 2 1

  return 0;
}
```

Vector: Capacity

`size`	Return size
`max_size`	Return maximum size
`resize`	Change size
`capacity`	Return size of allocated storage capacity
`empty`	Test whether vector is empty
`reserve`	Request a change in capacity
`shrink_to_fit`	Shrink to fit

Vector: Capacity

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

int main ()
{
  std::vector<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';
  // 0. size: 0
  for (int i=0; i<10; i++) myints.push_back(i);
  std::cout << "1. size: " << myints.size() << '\n';
  // 1. size: 10
  myints.insert (myints.end(),10,100);
  std::cout << "2. size: " << myints.size() << '\n';
  // 2. size: 20
  myints.pop_back();
  std::cout << "3. size: " << myints.size() << '\n';
  // 3. size: 19
  return 0;
}
```

Vector: Capacity

```cpp
// comparing size, capacity and max_size
#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";
  // size: 100
  std::cout << "capacity: " << myvector.capacity() << "\n";
  // capacity: 128
  std::cout << "max_size: " << myvector.max_size() << "\n";
  // max_size: 1073741823
  return 0;
}
```

Vector: Capacity

```
// resizing vector
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  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';
  //myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
  return 0;
}
```

Vector: Capacity

```cpp
// vector::empty
#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';
  //total: 55
  return 0;
}
```

Vector: Capacity

```
#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';
    }
  }
  ...
}
```

Vector: Capacity

```
int main ()
{
  ...
 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;
}
```

Vector: Capacity

```
// 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: Capacity

```cpp
// vector::shrink_to_fit
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (100);

  std::cout << "1. capacity of myvector: "
            << myvector.capacity()
            << '\n';
  //1. capacity of myvector: 100

  ...
}
```

Vector: Capacity

```
int main ()
{
  ...
  myvector.resize(10);
  std::cout << "2. capacity of myvector: "
            << myvector.capacity()
            << '\n';
  //2. capacity of myvector: 100

  myvector.shrink_to_fit();
  std::cout << "3. capacity of myvector: "
            << myvector.capacity()
            << '\n';
  //3. capacity of myvector: 10

  return 0;
}
```

Vector: Element access

`operator[]`	Access element
`at`	Access element
`front`	Access first element
`back`	Access last element
`data`	Access data

Vector: Element access

```
// vector::operator[]
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (10); // 10 zero-initialized elements

  std::vector<int>::size_type sz = myvector.size();

  // assign some values:
  for (unsigned i=0; i<sz; i++)
   myvector[i]=i;

  ...
}
```

Vector: Element access

```
int main ()
{
  ...
  // reverse vector using operator[]:
  for (unsigned i=0; i<sz/2; i++)
  {
    int temp;
    temp = myvector[sz-1-i];
    myvector[sz-1-i]=myvector[i];
    myvector[i]=temp;
  }
  std::cout << "myvector contains:";
  for (unsigned i=0; i<sz; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  //myvector contains: 9 8 7 6 5 4 3 2 1 0
  return 0;
}
```

Vector: Element access

```
// vector::at
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (10);   // 10 zero-initialized ints
  // assign some values:
  for (unsigned i=0; i<myvector.size(); i++)
    myvector.at(i)=i;
  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    std::cout << ' ' << myvector.at(i);
  std::cout << '\n';
  //myvector contains: 0 1 2 3 4 5 6 7 8 9
  return 0;
}
```

Vector: Element access

```
// vector::front
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  myvector.push_back(78);
  myvector.push_back(16);
  // now front equals 78, and back 16
  myvector.front() -= myvector.back();
  std::cout << "myvector.front() is now "
            << myvector.front()
            << '\n';
  //myvector.front() is now 62
  return 0;
}
```

Vector: Element access

```
// vector::back
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  myvector.push_back(10);
  while (myvector.back() != 0)
  {
    myvector.push_back ( myvector.back() -1 );
  }
  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size() ; i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  //myvector contains: 10 9 8 7 6 5 4 3 2 1 0
  return 0;
}
```

Vector: Element access

```
// vector::data
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector (5);
  int* p = myvector.data();
  *p = 10;
  ++p;
  *p = 20;
  p[2] = 100;
  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); ++i)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  //myvector contains: 10 20 0 100 0
  return 0;
}
```

Vector: Modifiers

`assign`	Assign vector content
`push_back`	Add element at the end
`pop_back`	Delete last element
`insert`	Insert elements
`erase`	Erase elements
`swap`	Swap content
`clear`	Clear content
`emplace`	Construct and insert element
`emplace_back`	Construct and insert element at the end

Vector: Element access

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

Vector: Element access

```
int main ()
{
  ...

  std::cout << "Size of first: "
            << int (first.size())
            << '\n';
  //Size of first: 7
  std::cout << "Size of second: "
            << int (second.size())
            << '\n';
  //Size of second: 5
  std::cout << "Size of third: "
            << int (third.size())
            << '\n';
  //Size of third: 3
  return 0;
}
```

Vector: Element access

```cpp
// vector::push_back
#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: Element access

```
// vector::pop_back
#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 << "Elements of myvector add up to " << sum << '\n';
  //Elements of myvector add up to 600
  return 0;
}
```

Vector: Element access

```
// inserting into a vector
#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());
  ...
}
```

Vector: Element access

```
// inserting into a vector
#include <iostream>
#include <vector>

int main ()
{
  ...
  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';
  // myvector contains: 501 502 503 300 300 400 400 200 100 100 100
  return 0;
}
```

Vector: Element access

```
// erasing from vector
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  for (int i=1; i<=10; i++)
   myvector.push_back(i);
  myvector.erase (myvector.begin()+5); // erase the 6th element
  // 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';
  //myvector contains: 4 5 7 8 9 10
  return 0;
}
```

Vector: Element access

```cpp
// swap vectors
#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';
  // foo contains: 200 200 200 200 200

  ...
}
```

Vector: Element access

```
// swap vectors

int main ()
{
...

  std::cout << "bar contains:";
  for (unsigned i=0; i<bar.size(); i++)
    std::cout << ' ' << bar[i];
  std::cout << '\n';
  // bar contains: 100 100 100
  return 0;
}
```

Vector: Element access

```
// clearing vectors
#include <iostream>
#include <vector>

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

  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  // myvector contains: 100 200 300

  ...
}
```

Vector: Element access

```
// clearing vectors


int main ()
{
  ...

  myvector.clear();
  myvector.push_back (1101);
  myvector.push_back (2202);

  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';
  // myvector contains: 1101 2202
  return 0;
}
```

Vector: Element access

```cpp
// vector::emplace
#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';
  // myvector contains: 10 200 100 20 30 300
  return 0;
}
```

Vector: Element access

```cpp
// vector::emplace_from
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector = {10,20,30};
  myvector.emplace_back (100);
  myvector.emplace_back (200);
  std::cout << "myvector contains:";
  for (auto& x: myvector)
    std::cout << ' ' << x;
  std::cout << '\n';
  // myvector contains: 10 20 30 100 200
  return 0;
}
```

Outline

Copy

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first,
                     InputIterator last,
                     OutputIterator result);
```

- Copies the elements in the range `[first,last)` into the range beginning at `result`.
- The function returns an iterator to the end of the destination range (which points to the element following the last element copied)
- The ranges shall not overlap in such a way that result points to an element in the range `[first,last)`. For such cases, see `copy_backward`.

Copy

```
// copy algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::copy
#include <vector>        // std::vector

int main () {
  int myints[]={10,20,30,40,50,60,70};
  std::vector<int> myvector (7);
  std::copy ( myints, myints+7, myvector.begin() );
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin();
       it!=myvector.end();
       ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // myvector contains: 10 20 30 40 50 60 70
  return 0;
}
```

For each

```
template <class InputIterator, class Function>
Function for_each (InputIterator first,
                   InputIterator last,
                   Function fn);
```

- Applies function `fn` to each of the elements in the range `[first,last)`.

For each

```
// for_each example
#include <iostream>      // std::cout
#include <algorithm>     // std::for_each
#include <vector>        // std::vector

void myfunction (int i) {  // function:
  std::cout << ' ' << i;
}

struct myclass {             // function object type:
  void operator() (int i) {std::cout << ' ' << i;}
} myobject;
```

For each

```
// for_each example
int main () {
  std::vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);

  std::cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myfunction);
  std::cout << '\n';
  // myvector contains: 10 20 30

  ...
}
```

For each

```
// for_each example
int main () {
  ...


  // or:
  std::cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myobject);
  std::cout << '\n';
  // myvector contains: 10 20 30
  return 0;
}
```

Find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first,
                    InputIterator last,
                    const T& val);
```

- Returns an iterator to the first element in the range `[first,last)` that compares equal to `val`. If no such element is found, the function returns `last`.
- The function uses `operator==` to compare the individual elements to `val`.

Find

```
// find example
#include <iostream>      // std::cout
#include <algorithm>     // std::find
#include <vector>        // std::vector

int main () {
  int myints[] = { 10, 20, 30 ,40 };
  int * p;

  // pointer to array element:
  p = std::find (myints,myints+4,30);
  ++p;
  std::cout << "The element following 30 is "
            << *p
            << '\n';
  // The element following 30 is 40

  ...
}
```

Find

```
// find example

int main () {
  ...

  std::vector<int> myvector (myints,myints+4);
  std::vector<int>::iterator it;

  // iterator to vector element:
  it = find (myvector.begin(), myvector.end(), 30);
  ++it;
  std::cout << "The element following 30 is "
            << *it
            << '\n';
  // The element following 30 is 40
  return 0;
}
```

Find if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first,
                       InputIterator last,
                       UnaryPredicate pred);
```

- Returns an iterator to the first element in the range `[first,last)` for which `pred` returns true. If no such element is found, the function returns `last`.

Find if

```
// find_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::find_if
#include <vector>        // std::vector

bool IsOdd (int i) {
  return ((i%2)==1);
}
```

Find if

```
int main () {
  std::vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(25);
  myvector.push_back(40);
  myvector.push_back(55);
  std::vector<int>::iterator it = std::find_if (myvector.begin(),
                                                myvector.end(),
                                                IsOdd);
  std::cout << "The first odd value is "
            << *it
            << '\n';
  // The first odd value is 25
  return 0;
}
```

Find end

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end (ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2);
template <class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate>
ForwardIterator1 find_end (ForwardIterator1 first1,
                           ForwardIterator1 last1,
                           ForwardIterator2 first2,
                           ForwardIterator2 last2,
                           BinaryPredicate pred);
```

Find end

- Searches the range `[first1,last1)` for the last occurrence of the sequence defined by `[first2,last2)`, and returns an iterator to its first element, or `last1` if no occurrences are found.
- The elements in both ranges are compared sequentially using `operator==` (or `pred`, in version (2)): A subsequence of `[first1,last1)` is considered a match only when this is true for all the elements of `[first2,last2)`.
- This function returns the last of such occurrences. For an algorithm that returns the first instead, see `search`.

Find end

```
// find_end example
#include <iostream>      // std::cout
#include <algorithm>     // std::find_end
#include <vector>        // std::vector

bool myfunction (int i, int j) {
  return (i==j);
}
```

Find end

```
int main () {
  int myints[] = {1,2,3,4,5,1,2,3,4,5};
  std::vector<int> haystack (myints,myints+10);
  int needle1[] = {1,2,3};

  // using default comparison:
  std::vector<int>::iterator it;
  it = std::find_end (haystack.begin(), haystack.end(),
                      needle1, needle1+3);
  if (it!=haystack.end())
    std::cout << "needle1 last found at position "
              << (it-haystack.begin())
              << '\n';
  // needle1 found at position 5

  ...
  }
```

Find end

```
int main () {
  ...

  int needle2[] = {4,5,1};

  // using predicate comparison:
  it = std::find_end (haystack.begin(), haystack.end(),
                      needle2, needle2+3, myfunction);
  if (it!=haystack.end())
    std::cout << "needle2 last found at position "
              << (it-haystack.begin())
              << '\n';
  // needle2 found at position 3
  return 0;
  }
```

Count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
  count (InputIterator first, InputIterator last, const T& val);
```

- Returns the number of elements in the range `[first,last)` that compare equal to `val`.
- The function uses `operator==` to compare the individual elements to `val`.

Count

```
// count algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::count
#include <vector>        // std::vector

int main () {
  // counting elements in array:
  int myints[] = {10,20,30,30,20,10,10,20};   // 8 elements
  int mycount = std::count (myints, myints+8, 10);
  std::cout << "10 appears "
            << mycount
            << " times.\n";
  // 10 appears 3 times.

  ...
}
```

Count

```
// count algorithm example
int main () {
  ...

  // counting elements in container:
  std::vector<int> myvector (myints, myints+8);
  mycount = std::count (myvector.begin(), myvector.end(), 20);
  std::cout << "20 appears "
            << mycount
            << " times.\n";
  // 20 appears 3 times.
  return 0;
}
```

Count if

```
template <class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if (InputIterator first, InputIterator last,
              UnaryPredicate pred);
```

- Returns the number of elements in the range `[first,last)` for which `pred` is true.
- The function uses `operator==` to compare the individual elements to `val`.

Count if

```
// count_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::count_if
#include <vector>        // std::vector

bool IsOdd (int i) {
  return ((i%2)==1);
}
```

Count if

```
int main () {
  std::vector<int> myvector;
  for (int i=1; i<10; i++)
    myvector.push_back(i); // myvector: 1 2 3 4 5 6 7 8 9

  int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
  std::cout << "myvector contains "
            << mycount
            << " odd values.\n";
  // myvector contains 5 odd values.
  return 0;
}
```

Replace if

```
template <class ForwardIterator, class UnaryPredicate, class T>
void replace_if (ForwardIterator first, ForwardIterator last,
                 UnaryPredicate pred, const T& new_value );
```

- Assigns `new_value` to all the elements in the range `[first,last)` for which `pred` returns true.

Replace if

```
// replace_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_if
#include <vector>        // std::vector

bool IsOdd (int i) {
  return ((i%2)==1);
}
```

Replace if

```
int main () {
  std::vector<int> myvector;
  for (int i=1; i<10; i++)
    myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
  std::replace_if (myvector.begin(), myvector.end(), IsOdd, 0);
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin();
       it!=myvector.end();
       ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // myvector contains: 0 2 0 4 0 6 0 8 0
  return 0;
}
```

Replace copy

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator
    replace_copy (InputIterator first, InputIterator last,
                  OutputIterator result,
                  const T& old_value, const T& new_value);
```

- Copies the elements in the range `[first,last)` to the range beginning at `result`, replacing the appearances of `old_value` by `new_value`.
- The function uses `operator==` to compare the individual elements to `old_value`.

Replace copy

```
// replace_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_copy
#include <vector>        // std::vector

int main () {
  int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
  std::vector<int> myvector (8);
  std::replace_copy (myints, myints+8, myvector.begin(), 20, 99);
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin();
       it!=myvector.end();
       ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // myvector contains: 10 99 30 30 99 10 10 99
  return 0;
}
```

Replace copy if

```
template <class InputIterator, class OutputIterator,
          class UnaryPredicate, class T>
  OutputIterator
    replace_copy_if (InputIterator first, InputIterator last,
                     OutputIterator result, UnaryPredicate pred,
                     const T& new_value);
```

- Copies the elements in the range `[first,last)` to the range beginning at result, replacing those for which `pred` returns true by `new_value`.

Replace copy if

```
// replace_copy_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_copy_if
#include <vector>        // std::vector

bool IsOdd (int i) {
return ((i%2)==1);
}
```

Replace copy if

```
int main () {
  std::vector<int> foo,bar;
  for (int i=1; i<10; i++)
    foo.push_back(i); // 1 2 3 4 5 6 7 8 9
  bar.resize(foo.size()); // allocate space
  std::replace_copy_if (foo.begin(), foo.end(),
                        bar.begin(), IsOdd, 0);
// 0 2 0 4 0 6 0 8 0
  std::cout << "bar contains:";
  for (std::vector<int>::iterator it=bar.begin();
       it!=bar.end();
       ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // second contains: 0 2 0 4 0 6 0 8 0
  return 0;
}
```

Sort

```
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort (RandomAccessIterator first, RandomAccessIterator last,
           Compare comp);
```

- Sorts the elements in the range `[first,last)` into ascending order.
- The elements are compared using `operator<` for the first version, and `comp` for the second.
- Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

Sort

```
// sort algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector

bool myfunction (int i,int j) {
  return (i<j);
}

struct myclass {
  bool operator() (int i,int j) {
    return (i<j);
  }
} myobject;
```

Sort

```
int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);
  // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);
  //(12 32 45 71)26 80 53 33

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction);
  // 12 32 45 71(26 33 53 80)

  ...
}
```

Sort

```
int main () {
  ...

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);
  //(12 26 32 33 45 53 71 80)

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin();
       it!=myvector.end();
       ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // myvector contains: 12 26 32 33 45 53 71 80
  return 0;
}
```

Merge

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                      InputIterator2 first2, InputIterator2 last2,
                      OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                      InputIterator2 first2, InputIterator2 last2,
                      OutputIterator result, Compare comp);
```

- Combines the elements in the sorted ranges `[first1,last1)` and `[first2,last2)`, into a new range beginning at `result` with all its elements sorted.
- The elements are compared using `operator<` for the first version, and `comp` for the second. The elements in both ranges shall already be ordered according to this same criterion (`operator<` or `comp`).

Merge

```cpp
// merge algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::merge, std::sort
#include <vector>       // std::vector

int main () {
  int first[] = {5,10,15,20,25};
  int second[] = {50,40,30,20,10};
  std::vector<int> v(10);
  std::sort (first, first+5);
  std::sort (second, second+5);
  std::merge (first, first+5, second, second+5, v.begin());
  std::cout << "The resulting vector contains:";
  for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';
  // The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
  return 0;
}
```

Next permutation

```
template <class BidirectionalIterator>
bool next_permutation (BidirectionalIterator first,
                       BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool next_permutation (BidirectionalIterator first,
                       BidirectionalIterator last,
                       Compare comp);
```

- Rearranges the elements in the range `[first,last)` into the next lexicographically greater permutation.
- The comparisons of individual elements are performed using either `operator<` for the first version, or `comp` for the second.

Next permutation

```
// next_permutation example
#include <iostream>      // std::cout
#include <algorithm>     // std::next_permutation, std::sort

int main () {
  int myints[] = {1,2,3};
  std::sort (myints,myints+3);
  std::cout << "The 3! possible permutations with 3 elements:\n";
  do {
    std::cout << myints[0]
              << ' '
              << myints[1]
              << ' '
              << myints[2]
              << '\n';
  } while ( std::next_permutation(myints,myints+3) );


 ...
}
```

Next permutation

```
// next_permutation example
int main () {
  ...

  std::cout << "After loop: "
            << myints[0]
            << ' '
            << myints[1]
            << ' '
            << myints[2]
            << '\n';
  return 0;
}
```

Next permutation

```
// The 3! possible permutations with 3 elements:
// 1 2 3
// 1 3 2
// 2 1 3
// 2 3 1
// 3 1 2
// 3 2 1
// After loop: 1 2 3
```