Liite

Geneeriset algoritmit aakkosjärjestyksessä

Tässä liitteessä katsomme kutakin yksittäistä algoritmia vuorollaan. Olemme päättäneet esitellä ne aakkosjärjestyksessä (pienin poikkeuksin) niin, että niitä on helpompi hakea. Algoritmien yleinen esitysmuoto on (1) listata funktion prototyyppi, (2) sisällyttää yksi tai kaksi kappaletta algoritmin kuvausta, joissa ilmaistaan erityisesti kaikki sellaiset käyttäytymiset ja odotukset, jotka eivät ole ilmeisen selviä asioita, ja ennen kaikkea (3) sisällyttää ohjelmaesimerkki kuvaamaan, kuinka algoritmia voitaisiin käyttää.

Kaikkien algoritmien kaksi ensimmäistä argumenttia (tarpeellinen kourallinen poikkeuksia vahvistaa säännön) ovat iteraattoripari, joihin yleensä viitataan first (ensimmäinen) ja last (viimeinen) ja jotka merkitsevät säiliön tai sisäisen taulukon elementtialueen, jonka kanssa operoidaan. Elementtialueen ilmaisu (sanotaan joskus *vasemmalta mukaan ottavaksi väliksi*) kirjoitetaan tavallisesti näin

```
// luetaan: ottaa mukaan ensimmäisen ja
// jokaisen elementin viimeiseen saakka, mutta ei viimeistä
[ first, last )
```

joka ilmaisee, että alue alkaa kohdasta first ja loppuu kohtaan last, mutta last itse ei kuulu joukkoon. Kun

```
first == last
```

alueen sanotaan olevan tyhjä.

Iteraattoriparin vaatimus on, että pitää olla mahdollista saavuttaa last aloittaen kohdasta first kasvatusoperaattorin toistuvan käytön kautta. Itse kääntäjä ei voi kuitenkaan pakottaa tähän. Ellei tätä vaatimusta täytetä, on tuloksena tuntematon suorituksenaikainen käyttäytyminen — usein ohjelman lopullinen “dumppi” (ohjelman muistivedos).

Jokaisen algoritmin esittely ilmaisee minimikategorian iteraattorituesta, jonka se vaatii (katso kohdasta 12.4 lyhyt käsittely viidestä iteraattorikategoriasta). Esimerkiksi find(), joka to-

teuttaa yhden kerran vain luku -läpikäynnin säiliöön, vaatii minimissään syöttöiteraattorin (InputIterator). Sille voidaan välittää myös eteenpäin (ForwardIterator)-, kaksisuuntaisuus (Bidirectional)- tai hajakäsittely (RandomAccessIterator) -iteraattori yhtä hyvin. Tulostusiteraattorin (OutputIterator) välittäminen sille on virhe. Virheitä, jotka aiheutuvat kelpaamattoman iteraattorikategorian välittämisestä algoritmille, ei taata siepattavan käännöksen aikana, koska iteraattorikategoriat eivät ole todellisia tyyppejä, vaan tyyppiparametreja funktiomallille.

Jotkut algoritmit tukevat useita versioita, yksi hyödyntää sisäistä operaattoria, kun taas toinen hyväksyy joko funktio-olion tai osoittimen funktioon tarjoten vaihtoehtoisen toteutustavan tuosta operaattorista. Esimerkiksi unique() vertailee oletusarvoisesti kahta vierekkäistä elementtiä käyttäen taustalla olevan säiliön elementin tyypin yhtäsuuruusoperaattoria. Jos taustalla olevalla elementtityypillä ei kuitenkaan ole yhtäsuuruusoperaattoria tai jos haluamme määritellä elementin yhtäsuuruuden eri tavalla, voimme välittää joko funktio-olion tai osoittimen ja saada aikaan halutun semantiikan. Toisia algoritmeja on kuitenkin eroteltu yksilöllisesti nimettyihin “väite”ilmentymäpareihin, jotka päättyvät loppuliitteeseen _if kuten find_if(). Esimerkiksi on olemassa replace()-ilmentymä, joka käyttää sisäistä yhtäsuuruusoperaattoria ja replace_if()-ilmentymä, joka saa (väite- eli) predikaattifunktio-olion tai -osoittimen funktioon.

Niille algoritmeille, jotka muokkaavat operoimaansa säiliötä, on yleensä kaksi versiota: paikalleen muuttava versio, joka muuttaa säiliötä, johon sitä käytetään, ja versio, joka palauttaa kopion säiliöstä siihen tehtyine muutoksineen. On olemassa esimerkiksi sekä replace()- että replace_copy()-algoritmi. Kopiointiversio sisältää nimessään aina loppuliitteen _copy. Kopiointiversiota ei kuitenkaan ole tehty jokaiselle algoritmille, joka muuttaa siihen liittyvää säiliötä. Esimerkiksi sort()-algoritmeissa ei ole kopiointi-ilmentymää. Jos tässä tapauksessa haluamme algoritmin operoivan kopiota, pitää kopio tehdä ja välittää itse.

Jotta voisimme käyttää geneerisiä algoritmeja, pitää ottaa mukaan niihin liittyvä otsikkotiedosto:

```
#include <algorithm>
```

Jos haluamme käyttää yhtäkin neljästä numeerisesta algoritmista: adjacent_difference(), accumulate(), inner_product() ja partial_sum(), pitää ottaa mukaan otsikkotiedosto

```
#include <numeric>
```

Algoritmeja toteuttava koodi ja niihin liittyvät säiliötyypit, joilla ne operoivat tässä liitteessä, heijastavat nykyisin saatavilla olevia toteutuksia. Esimerkiksi iostream-kirjasto heijastelee esistandardin toteutusta — mukaan lukien iostream.h-otsikkotiedoston käyttö. Mallit eivät tue oletusargumentteja malliparametreille. Jos haluat ohjelmaa ajettavan nykyisessä järjestelmässäsi, voi olla tarpeen muuttaa yhtä jos toista esittelyä.

Erinomainen ja tarkempi käsittely geneerisistä algoritmeista löytyy julkaisusta [MUSSER96], vaikka se on hieman vanhentunut lopulliseen C++-vakiokirjastoon verrattuna.

accumulate()

```
template < class InputIterator, class Type >
Type accumulate(
    InputIterator first, InputIterator last,
    Type init );

template < class InputIterator, class Type,
           class BinaryOperation >
Type accumulate(
    InputIterator first, InputIterator last,
    Type init, BinaryOperation op );
```

Ensimmäinen accumulate()-versio lisää elementtien arvojen yhteissumman alueelta, joka on ilmaistu iteraattoriparilla [first,last), init:in määrittämään alkuarvoon. Kun on esimerkiksi seuraava jono {1,1,2,3,5,8}, ja alkuarvo on 0, niin tulos on 20. Toisessa versiossa yhteenlaskun sijasta välitetään binäärioperaatio (kaksioperandinen), jota käytetään elementteihin. Jos esimerkiksi välittäisimme funktio-olion times<int> accumulate():lle, on tulos 240 edellyttäen, että alkuarvo on tietysti 1 eikä 0. accumulate() on eräs numeerisista algoritmeista. Jotta voimme käyttää sitä, pitää ottaa mukaan <numeric>-otsikkotiedosto.

```
#include <numeric>
#include <list>
#include <functional>
#include <iostream.h>

/*
 * tulostus:
  accumulate()
       operating on values {1,2,3,4}
       result with default addition: 10
       result with plus<int> function object: 10
 */

int main()
{
    int ia[] = { 1, 2, 3, 4 };
    list<int,allocator> ilist( ia, ia+4 );

    int ia_result = accumulate(&ia[0], &ia[4], 0);
    int ilist_res = accumulate(
          ilist.begin(), ilist.end(), 0, plus<int>() );

    cout << "accumulate()\n\t"
        << "operating on values {1,2,3,4}\n\t"
        << "result with default addition: "
        << ia_result << "\n\t"
        << "result with plus<int> function object: "
        << ilist_res
```

```
        << endl;
}
```

adjacent_difference()

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

template < class InputIterator, class OutputIterator,
           class BinaryOperation >
OutputIterator adjacent_difference(
    InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation op );
```

Ensimmäinen adjacent_difference()-versio luo uuden jonon, jossa jokaisen uuden elementin arvo on nykyisen ja edellisen elementin erotus. Kun on esimerkiksi jono {0,1,1,2,3,5,8}, niin uuden jonon ensimmäinen elementti on yksinkertaisesti kopio alkuperäisen jonon ensimmäisestä elementistä: 0. Toinen elementti on kahden ensimmäisen elementin erotus: 1. Kolmas elementti on toisen ja kolmannen elementin erotus: 1 – 1, eli 0 jne. Uusi jono on {0,1,0,1,1,2,3}.

Toinen versio laskee vierekkäisten elementtien erotuksen käyttäen erityistä binäärioperaatiota. Kun esimerkiksi käytetään samaa jonoa, välitetään times<int>-funktio-olio. Jälleen uuden jonon ensimmäinen elementti on yksinkertaisesti kopio alkuperäisestä jonosta: 0. Toinen elementti on ensimmäisen ja toisen elementin tulo, siis 0. Kolmas elementti on toisen ja kolmannen elementin tulo: 1 * 1, eli 1 jne. Uusi jono on {0,0,1,2,6,15,40}.

Molemmissa versioissa OutputIterator osoittaa yhden yli uuden jonon viimeisen elementin. adjacent_difference() on eräs numeerisista algoritmeista. Jotta voimme käyttää kumpaakin versiota, pitää ottaa mukaan <numeric>-otsikkotiedosto.

```
#include <numeric>
#include <list>
#include <functional>
#include <iterator>
#include <iostream.h>

int main()
{
    int ia[] = { 1, 1, 2, 3, 5, 8 };

    list<int,allocator> ilist(ia, ia+6);
    list<int,allocator> ilist_result(ilist.size());


    adjacent_difference(ilist.begin(), ilist.end(),
                ilist_result.begin() );

    // generoi tulostuksen:
```

```
    // 1 0 1 1 2 3

    copy( ilist_result.begin(), ilist_result.end(),
        ostream_iterator<int>(cout," "));
    cout << endl;

    adjacent_difference( ilist.begin(), ilist.end(),
                ilist_result.begin(), times<int>() );

    // generoi tulostuksen:
    // 1 1 2 6 15 40
    copy( ilist_result.begin(), ilist_result.end(),
        ostream_iterator<int>(cout," "));
    cout << endl;
}
```

adjacent_find()

```
template< class ForwardIterator >
ForwardIterator
adjacent_find( ForwardIterator first, ForwardIterator last );

template< class ForwardIterator, class BinaryPredicate >
ForwardIterator
adjacent_find( ForwardIterator first,
            ForwardIterator last, Predicate pred );
```

adjacent_find() etsii ensimmäisen vierekkäisen tuplaparin alueelta, joka on merkitty arvoilla [first,last). Se palauttaa ForwardIterator:in parin ensimmäiseen elementtiin, jos sellainen löytyy; muussa tapauksessa se palauttaa last. Kun on esimerkiksi jono {0,1,1,2,2,4}, niin pari {1,1} tunnistetaan, jolloin palautetaan iteraattori, joka osoittaa ensimmäiseen arvoon 1.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

class TwiceOver {
public:
    bool operator() ( int val1, int val2 )
        { return val1 == val2/2 ? true : false; }
};

int main()
{
    int ia[] = { 1, 4, 4, 8 };
    vector< int, allocator > vec( ia, ia+4 );

    int *piter;
    vector< int, allocator >::iterator iter;
```

```
    // piter osoittaa ia[1]:han
    piter = adjacent_find( ia, ia+4 );
    assert( *piter == ia[ 1 ] );

    // iter osoittaa vec[2]:iin
    iter = adjacent_find( vec.begin(), vec.end(), TwiceOver() );
    assert( *iter == vec[ 2 ] );

    // päässyt tänne: kaikki ok
    cout << "ok: adjacent-find() succeeded!\n";
}
```

binary_search()

```
template< class ForwardIterator, class Type >
bool
binary_search( ForwardIterator first,
        ForwardIterator last, const Type &value );

bool
binary_search( ForwardIterator first,
            ForwardIterator last, const Type &value,
            Compare comp );
```

binary_search() etsii value-arvoa lajitellusta jonosta, joka on merkitty arvoilla [first,last). Jos arvo löytyy, palautetaan arvo tosi, muussa tapauksessa epätosi. Ensimmäinen versio olettaa, että säiliö on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria. Toisella versiolla ilmaisemme, että säiliö on lajiteltu käyttäen erityistä funktio-oliota.

```
#include <algorithm>
#include <vector>
#include <assert.h>

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};

    sort( &ia[0], &ia[12] );
    bool found_it = binary_search( &ia[0], &ia[12], 18 );
    assert( found_it == false );

    vector< int > vec( ia, ia+12 );
    sort( vec.begin(), vec.end(), greater<int>() );
    found_it = binary_search( vec.begin(), vec.end(),
                    26, greater<int>() );
    assert( found_it == true );
}
```

copy()

```
template < class InputIterator, class OutputIterator >
OutputIterator
copy( InputIterator first1, InputIterator last,
      OutputIterator first2 );
```

copy() kopioi elementtijonon, joka on merkitty arvoilla [first, last) säiliöön alkaen positiosta, joka on merkitty arvolla first2. Se palauttaa arvon first2 lisättynä yhdellä yli viimeisen lisätyn elementin. Kun on esimerkiksi jono {0,1,2,3,4,5}, niin voimme siirtää elementtejä yhdellä vasemmalle seuraavalla käynnistyksellä:

```
int ia[] = { 0, 1, 2, 3, 4, 5 };

// siirrä vasemmalle yhdellä, mikä johtaa tilanteeseen {1,2,3,4,5,5}
copy( ia+1, ia+6, ia );
```

copy() aloittaa toisesta elementistä, ia, kopioiden arvon 1 ensimmäiseen "lokeroon" ja niin edelleen, kunnes jokainen elementti on kopioitu vasemmalla puolellaan olevaan lokeroon.

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

/* generoi:
  0 1 1 3 5 8 13
  shifting array sequence left by 1:
  1 1 3 5 8 13 13
  shifting vector sequence left by 2:
  1 3 5 8 13 8 13
*/

int main()
{
    int ia[] = { 0, 1, 1, 3, 5, 8, 13 };
    vector< int, allocator > vec( ia, ia+7 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // siirrä vasemmalle yhdellä
    copy( ia+1, ia+7, ia );

    cout << "shifting array sequence left by 1:\n";
    copy( ia, ia+7, ofile ); cout << '\n';

    // siirrä vasemmalle kahdella
    copy( vec.begin()+2, vec.end(), vec.begin() );
```

```
    cout << "shifting vector sequence left by 2:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

copy_backward()

```
template < class BidirectionalIterator1,
           class BidirectionalIterator2 >
BidirectionalIterator2
copy_backward( BidirectionalIterator1 first,
               BidirectionalIterator1 last1,
               BidirectionalIterator2 last2 );
```

copy_backward() käyttäytyy samalla tavalla kuin copy() paitsi, että elementit kopioidaan päinvastaisessa järjestyksessä. Kopiointi siis alkaa kohdasta last1-1 ja etenee kohtaan first. Elementit kopioidaan myös kohdesäiliöön takaperin alkaen kohdasta last2-1 ja edeten taaksepäin kohtaan last1-first.

Olkoon esimerkiksi jono {0,1,2,3,4,5}. Voimme kopioida viimeiset kolme elementtiä (3,4,5) ensimmäiseen kolmeen elementtiin (0,1,2) asettamalla kohdan first osoitearvoon 0, last1 osoitearvoon 3 ja last2 osoitearvoon yksi yli arvon 5. Elementtiarvo 5 sijoitetaan aikaisemman arvon 2 paikalle, sitten elementti 4 aikaisemman arvon 1 paikalle ja lopuksi elementti 3 aikaisemman arvon 0 paikalle. Tulosjono on {3,4,5,3,4,5}.

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <iostream.h>

class print_elements {
public:
    void operator()( string elem ) {
        cout << elem
             << ( _line_cnt++%8 ? " " : "\n\t" );
    }
    static void reset_line_cnt() { _line_cnt = 1; }

private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* generoi:
  original list of strings:
  The light untonsured hair grained and hued like
  pale oak

  sequence after copy_backward( begin+1, end-3, end ):
```

```
  The light untonsured hair light untonsured hair grained
  and hued
*/

int main()
{
    string sa[] = {
        "The", "light", "untonsured", "hair",
        "grained", "and", "hued", "like", "pale", "oak" };

    vector< string, allocator > svec( sa, sa+10 );

    cout << "original list of strings:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n\n";

    copy_backward( svec.begin()+1, svec.end()-3, svec.end() );

    print_elements::reset_line_cnt();

    cout << "sequence after "
        << "copy_backward( begin+1, end-3, end ):\n";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n";
}
```

count()

```
template< class InputIterator, class Type >
iterator_traits<InputIterator>::distance_type
count( InputIterator first,
       InputIterator last, const Type& value );
```

count() vertaa jokaista elementtiä value-arvoon käyttäen yhtäsuuruusoperaattoria alueella, joka on merkitty arvoilla [first,last). Se palauttaa niiden säiliön elementtien lukumäärän, jotka ovat yhtäsuuria kuin value. (Huomaa, että toteutuksemme vakiokirjastosta tukee count()-algoritmin aikaisempaa määritystä.)

```
#include <algorithm>
#include <string>
#include <list>
#include <iterator>

#include <assert.h>
#include <iostream.h>
#include <fstream.h>

/*****************************************************************
* luettava teksti:
Alice Emma has long flowing red hair. Her Daddy says
```

```
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such thing,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
**********************************************************************
      * ohjelman tulostus:
      *    count(): fiery occurs 2 times
**********************************************************************
*/

int main()
{
    ifstream infile( "alice_emma" );
    assert ( infile != 0 );

    list<string,allocator> textlines;

    typedef list<string,allocator>::difference_type diff_type;
    istream_iterator< string, diff_type > instream( infile ),
                                          eos;

    copy( instream, eos, back_inserter( textlines ));
    string search_item( "fiery" );

    /**************************************************************
  * Huomaa: tämä on C++-standardin rajapinta count():iin.
  *       Kuitenkin nykyinen RogueWave-toteutus
  *       tukee aikaisempaa versiota, jossa distance_type
  *       ei ollut kehitetty ja niin count() palautti
  *       arvonsa viimeisen argumentin kautta itselleen.
  *
  * näin käynnistyksen tulisi tapahtua:
  *
  * typedef iterator_traits<InputIterator>::
  *       distance_type dis_type;
  *
  * dis_type elem_count;
  * elem_count = count( textlines.begin(), textlines.end(),
    *              search_item );
  **************************************************************/

    int elem_count = 0;
    list<string,allocator>::iterator
         ibegin = textlines.begin(),
         iend   = textlines.end();

    // count():in vanhentunut muoto
    count( ibegin, iend, search_item, elem_count );
```

```
    cout << "count(): " << search_item
         << " occurs "  << elem_count << " times\n";
}
```

count_if()

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

count_if() käyttää pred-väittämää jokaista elementtiä vastaan alueella, joka on merkitty arvoilla [first,last). Se palauttaa niiden kertojen lukumäärän, joina pred sai arvon tosi.

```
#include <algorithm>
#include <list>
#include <iostream.h>

class Even {
public:
    bool operator()( int val )
        { return val%2 ? false : true; }
};

int main()
{
    int ia[] = {0,1,1,2,3,5,8,13,21,34};
    list< int,allocator > ilist( ia, ia+10 );

/*
 * ei tueta nykyisessä toteutuksessa
 *******************************************************
 typedef
   iterator_traits<InputIterator>::distance_type
   distance_type;

   distance_type ia_count, list_count;

   // laske parilliset elementit: 4
   ia_count = count_if( &ia[0], &ia[10], Even() );
   list_count = count_if( ilist.begin(), ilist_end(),
                    bind2nd(less<int>(),10) );
 *******************************************************
 */

    int ia_count = 0;
    count_if( &ia[0], &ia[10], Even(), ia_count );

    // generoi:
```

```
    //   count_if(): on 4 elementtiä, jotka ovat parillisia

    cout << "count_if(): there are "
        << ia_count << " elements that are even.\n";

    int list_count = 0;
    count_if( ilist.begin(), ilist.end(),
            bind2nd(less<int>(),10), list_count );

    // generoi:
    // count_if(): on 7 elementtiä, jotka ovat pienempiä kuin 10.

    cout << "count_if(): there are "
        << list_count
        << " elements that are less than 10.\n";
}
```

equal()

```
template< class InputIterator1, class InputIterator2 >
bool
equal( InputIterator1 first1,
       InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
          class BinaryPredicate >
bool
equal( InputIterator1 first1, InputIterator1 last,
       InputIterator2 first2, BinaryPredicate pred );
```

equal() palauttaa arvon tosi, jos nuo kaksi jonoa ovat yhtäsuuria niiden elementtiensä lukumäärältä, jotka sisältyvät alueelle [first,last). Jos toinen jono sisältää lisäelementtejä, ei niitä oteta huomioon. Jos haluamme taata, että molemmat jonot ovat yhtäsuuria, pitää kirjoittaa

```
if ( vec1.size() == vec2.size() &&
   equal( vec1.begin(), vec1.end(), vec2.begin() );
```

tai käyttää säiliön yhtäsuuruusoperaattoria kuten vec1==vec2. Jos toinen säiliö sisältää vähemmän elementtejä kuin ensimmäinen ja algoritmin pitäisi iteroida sen loppuun, on suorituksenaikainen käyttäytyminen tuntematon. Oletusarvoisesti vertailuun käytetään taustalla olevan elementtityypin yhtäsuuruusoperaattoria; toinen versio käyttää pred-väittämää.

```
#include <algorithm>
#include <list>
#include <iostream.h>

class equal_and_odd{
public:
  bool
  operator()( int val1, int val2 )
```

```
    {
        return ( val1 == val2 &&
             ( val1 == 0 || val1 % 2 ));
    }
};

int main()
{
    int ia[] =  { 0,1,1,2,3,5,8,13 };
    int ia2[] = { 0,1,1,2,3,5,8,13,21,34 };

    bool res;

    // tosi: molemmat ovat yhtä pitkiä kuin ia.
    // generoi: int ia[7] equal to int ia2[9]? true.

    res = equal( &ia[0], &ia[7], &ia2[0] );
    cout << "int ia[7] equal to int ia2[9]? "
        << ( res ? "true" : "false" ) << ".\n";

    list< int, allocator > ilist(  ia,  ia+7  );
    list< int, allocator > ilist2( ia2, ia2+9 );

    // generoi: list ilist equal to ilist2? true.

    res = equal( ilist.begin(), ilist.end(), ilist2.begin() );
    cout << "list ilist equal to ilist2? "
        << ( res ? "true" : "false" ) << ".\n";

    // epätosi: 0, 2, 8 are not equal and odd
    // generoi: list ilist equal_and_odd() to ilist2? false.

    res = equal( ilist.begin(), ilist.end(),
              ilist2.begin(), equal_and_odd() );

    cout << "list ilist equal_and_odd() to ilist2? "
        << ( res ? "true" : "false" ) << ".\n";

    return 0;
}
```

equal_range()

```
template< class ForwardIterator, class Type >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
        ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
pair< ForwardIterator, ForwardIterator >
equal_range( ForwardIterator first,
        ForwardIterator last, const Type &value,
        Compare comp );
```

equal_range() palauttaa iteraattoriparin. Ensimmäinen iteraattori edustaa iteraattorin arvoa, jonka lower_bound() on palauttanut; toinen iteraattori edustaa iteraattorin arvoa, jonka upper_bound() on palauttanut; katso kyseisten algoritmien kuvaukset niiden toiminnasta. Olkoon esimerkiksi seuraava jono:

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

Kun equal_range()-algoritmia kutsutaan arvolla 21, se palauttaa iteraattoriparin, jossa kumpikin osoittaa arvoa 22. Kun equal_range()-algoritmia kutsutaan arvolla 22, se palauttaa iteraattoriparin, jossa first osoittaa arvoa 22 ja second osoittaa arvoa 23. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio järjestää elementin pred-väittämän perusteella.

```
#include <algorithm>
#include <vector>
#include <utility>
#include <iostream.h>

/* generoi:
  array element sequence after sort:
  12 15 17 19 20 22 23 26 29 35 40 51

  equal_range result of search for value 23:
       *ia_iter.first: 23     *ia_iter.second: 26

  equal_range result of search for absent value 21:
       *ia_iter.first: 22     *ia_iter.second: 22

  vector element sequence after sort:
  51 40 35 29 26 23 22 20 19 17 15 12

  equal_range result of search for value 26:
       *ivec_iter.first: 26   *ivec_iter.second: 23

  equal_range result of search for absent value 21:
       *ivec_iter.first: 20   *ivec_iter.second: 20
*/
```

```
int main()
{
    int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
    vector< int, allocator > ivec( ia, ia+12 );
    ostream_iterator< int >  ofile( cout, " " );

    sort( &ia[0], &ia[12] );

    cout << "array element sequence after sort:\n";
    copy( ia, ia+12, ofile ); cout << "\n\n";

    pair< int*,int* > ia_iter;
    ia_iter = equal_range( &ia[0], &ia[12], 23 );

    cout << "equal_range result of search for value 23:\n\t"
        << "*ia_iter.first: "  << *ia_iter.first << "\t"
        << "*ia_iter.second: " << *ia_iter.second << "\n\n";

    ia_iter = equal_range( &ia[0], &ia[12], 21 );

    cout << "equal_range result of search for "
        << "absent value 21:\n\t"
        << "*ia_iter.first: "  << *ia_iter.first << "\t"
        << "*ia_iter.second: " << *ia_iter.second << "\n\n";

    sort( ivec.begin(), ivec.end(), greater<int>() );

    cout << "vector element sequence after sort:\n";
    copy( ivec.begin(), ivec.end(), ofile ); cout << "\n\n";

    typedef vector< int, allocator >::iterator iter_ivec;
    pair< iter_ivec, iter_ivec > ivec_iter;

    ivec_iter = equal_range( ivec.begin(), ivec.end(), 26,
                             greater<int>() );

    cout << "equal_range result of search for value 26:\n\t"
        << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
        << "*ivec_iter.second: " << *ivec_iter.second
        << "\n\n";

    ivec_iter = equal_range( ivec.begin(), ivec.end(), 21,
                             greater<int>() );

    cout << "equal_range result of search for "
        << "absent value 21:\n\t"
        << "*ivec_iter.first: "  << *ivec_iter.first << "\t"
        << "*ivec_iter.second: " << *ivec_iter.second
```

```
            << "\n\n";
}
```

fill()

```
template< class ForwardIterator, class Type >
void
fill( ForwardIterator first,
      ForwardIterator last, const Type& value );
```

fill() sijoittaa kopion value-arvosta jokaiseen elementtiin alueella, joka on merkitty arvoilla [first, last).

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/* generoi:
  original array element sequence:
  0 1 1 2 3 5 8

  array after fill(ia+1,ia+6):
  0 9 9 9 9 9 8

  original list element sequence:
  c eiffel java ada perl

  list after fill(++ibegin,--iend):
  c c++ c++ c++ perl
*/

int main()
{
    const int value = 9;
    int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
    ostream_iterator< int > ofile( cout, " " );

    cout << "original array element sequence:\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    fill( ia+1, ia+6, value );

    cout << "array after fill(ia+1,ia+6):\n";
    copy( ia, ia+7, ofile ); cout << "\n\n";

    string the_lang( "c++" );
    string langs[5] = { "c", "eiffel", "java", "ada", "perl" };

    list< string, allocator > il( langs, langs+5 );
```

```
    ostream_iterator< string > sofile( cout, " " );

    cout << "original list element sequence:\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";

    typedef list<string,allocator>::iterator iterator;

    iterator ibegin = il.begin(), iend = il.end();
    fill( ++ibegin, --iend, the_lang );

    cout << "list after fill(++ibegin,--iend):\n";
    copy( il.begin(), il.end(), sofile ); cout << "\n\n";
}
```

fill_n()

```
template< class ForwardIterator, class Size, class Type >
void
fill_n( ForwardIterator first,
        Size n, const Type& value );
```

fill_n() sijoittaa kopion value-arvosta count elementtiin alueella [first, first+count).

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

class print_elements {
public:
    void operator()( string elem ) {
        cout << elem
             << ( _line_cnt++%8 ? " " : "\n\t" );
    }
    static void reset_line_cnt() { _line_cnt = 1; }

private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;

/* generoi:
original element sequence of array container:
0 1 1 2 3 5 8

array after fill_n( ia+2, 3, 9 ):
0 1 9 9 9 5 8

original sequence of strings:
     Stephen closed his eyes to hear his boots
```

```
        crush crackling wrack and shells

sequence after fill_n() applied:
        Stephen closed his xxxxx xxxxx xxxxx xxxxx xxxxx
        xxxxx crackling wrack and shells
*/

int main()
{
    int value = 9; int count = 3;
    int ia[]  = { 0, 1, 1, 2, 3, 5, 8 };
    ostream_iterator< int > iofile( cout, " " );

    cout << "original element sequence of array container:\n";
    copy( ia, ia+7, iofile ); cout << "\n\n";

    fill_n( ia+2, count, value );

    cout << "array after fill_n( ia+2, 3, 9 ):\n";
    copy( ia, ia+7, iofile ); cout << "\n\n";

    string replacement( "xxxxx" );
    string sa[] = { "Stephen", "closed", "his", "eyes", "to",
                    "hear", "his", "boots", "crush", "crackling",
                    "wrack", "and", "shells" };

    vector< string, allocator > svec( sa, sa+13 );

    cout << "original sequence of strings:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n\n";

    fill_n( svec.begin()+3, count*2, replacement );

    print_elements::reset_line_cnt();

    cout << "sequence after fill_n() applied:\n\t";
    for_each( svec.begin(), svec.end(), print_elements() );
    cout << "\n";
}
```

find()

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

Elementtien, jotka sijaitsevat alueella, joka on merkitty arvoilla [first,last), yhtäsuuruutta vertaillaan value-arvoon taustalla olevan tyypin yhtäsuuruusoperaattoria käyttäen. Jos vastaavuus

löytyy, etsintä päättyy. find() palauttaa InputIterator:in elementtiin. Ellei vastaavuutta löydy, palautetaan last.

```
#include <algorithm>
#include <iostream.h>
#include <list>
#include <string>

int main()
{
    int array[ 17 ] = { 7,3,3,7,6,5,8,7,2,1,3,8,7,3,8,4,3 };

    int elem = array[ 9 ];
    int *found_it;

    found_it = find( &array[0], &array[17], elem );

    // generoi: find the first occurrence of 1 found!
    cout << "find the first occurrence of "
        << elem << "\t"
        << ( found_it ? "found!\n" : "not found!\n" );

    string beethoven[] = {
         "Sonata31", "Sonata32", "Quartet14", "Quartet15",
         "Archduke", "Symphony7" };

    string s_elem( beethoven[ 1 ] );

    list< string, allocator > slist( beethoven, beethoven+6 );
    list< string, allocator >::iterator iter;

    iter = find( slist.begin(), slist.end(), s_elem );

    // generoi: find the first occurrence of Sonata32  found!

    cout << "find the first occurrence of "
        << s_elem << "\t"
        << ( found_it ? "found!\n" : "not found!\n" );
}
```

find_if()

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

Elementtejä, jotka sijaitsevat merkityllä alueella [first,last), tutkitaan vuorollaan ja jokaiseen käytetään pred-väitettä. Jos pred saa arvokseen tosi, etsintä päättyy. find_if() palauttaa Input-Iterator:in elementtiin. Ellei vastaavuutta löydy, palautetaan last.

```
#include <algorithm>
#include <list>
#include <set>
#include <string>
#include <iostream.h>

// tarjoaa vaihtoehtoisen yhtäsuuruusoperaattorin
// palauttaa arvon tosi, jos merkkijono löytyy
// jäsenolion friendset:istä
class OurFriends {
public:
   bool operator()( const string& str ) {
         return ( friendset.count( str ));
   }

   static void
   FriendSet( const string *fs, int count ) {
         copy( fs, fs+count,
               inserter( friendset, friendset.end() ));
   }

private:
   static set< string, less<string>, allocator > friendset;
};

set< string, less<string>, allocator > OurFriends::friendset;

int main()
{
   string Pooh_friends[] = { "Piglet", "Tigger", "Eyeore"  };
   string more_friends[] = { "Quasimodo", "Chip", "Piglet" };
   list<string,allocator> lf( more_friends, more_friends+3 );

   // täytä Pooh-ystävien luettelo
   OurFriends::FriendSet( Pooh_friends, 3 );

   list<string,allocator>::iterator our_mutual_friend;
   our_mutual_friend =
               find_if( lf.begin(), lf.end(), OurFriends());

   // generoi:
   //   Ah, imagine our friend Piglet is also a friend of Pooh.
   if ( our_mutual_friend != lf.end() )
       cout << "Ah, imagine our friend "
            << *our_mutual_friend
```

```
            << " is also a friend of Pooh.\n";

    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 );
```

Ensimmäisen jonon alueelta, joka on merkitty iteraattoriparilla [first1,last1), etsitään toisen jonon viimeistä esiintymää, joka merkitty iteraattoriparilla [first2,last2). Olkoon esimerkiksi ensimmäinen jono Mississippi ja toinen ss, niin find_end() palauttaa ForwardIterator:in toisen ss-kirjainparin ensimmäiseen s-kirjaimeen. Ellei toista jonoa löydy ensimmäisestä jonosta, palautetaan last1. Ensimmäisessä versiossa käytetään taustalla olevan tyypin yhtäsuuruusoperaattoria. Toisessa versiossa käytetään binääripredikaattioperaatiota, jonka käyttäjä välittää.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
#include <assert.h>

int main()
{
    int array[ 17 ]  = { 7,3,3,7,6,5,8,7,2,1,3,7,6,3,8,4,3 };
    int subarray[ 3 ] = { 3, 7, 6 };

    int *found_it;

    // etsi jonon 3,7,6 viimeinen esiintymä
    // taulukosta ja palauta ensimmäisen elementin osoite...

    found_it = find_end( &array[0], &array[17],
                &subarray[0], &subarray[3] );

    assert( found_it == &array[10] );

    vector< int, allocator > ivec( array, array+17 );
    vector< int, allocator > subvec( subarray, subarray+3 );

    vector< int, allocator >::iterator found_it2;
```

```
    found_it2 = find_end( ivec.begin(), ivec.end(),
                          subvec.begin(), subvec.end(),
                          equal_to<int>() );

    assert( found_it2 == ivec.begin()+10 );

    cout << "ok: find_end correctly returned beginning of "
         << "last matching sequence: 3,7,6!\n";
}
```

find_first_of()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator1
find_first_of( ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2 );

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

Jonon alue, joka on merkitty arvoilla [first2,last2), sisältää elementtikokoelman, josta etsitään jonoa, jonka alue on merkitty arvoilla [first1,last1). Ajatellaan, että haluamme löytää ensimmäisen vokaalin merkkijonosta synesthesia. Jotta voisimme sen tehdä, määrittelemme toisen jonon aeiou. find_first_of() palauttaa ForwardIterator:in vokaalijonon ensimmäiseen elementin ilmentymään, joka tässä tapauksessa on ensimmäinen e-kirjain. Ellei ensimmäinen jono sisällä yhtään toisen jonon elementtiä, palautetaan last1. Ensimmäisessä versiossa käytetään taustalla olevan tyypin yhtäsuuruusoperaattoria. Toisessa versiossa käytetään binääroperaatiota pred.

```
#include <algorithm>
#include <vector>
#include <string>
#include <iostream.h>

int main()
{
    string s_array[] = { "Ee", "eE", "ee", "Oo", "oo", "ee" };
    string to_find[] = { "oo", "gg", "ee" };

    // palauttaa jonon "ee" ensimmäisen esiintymän -- &s_array[2]
    string *found_it =
        find_first_of( s_array, s_array+6,
                       to_find, to_find+3 );

    // generoi:
    // found it: ee
    //       &s_array[2]:    0x7fff2dac
```

```
    //      &found_it:     0x7fff2dac

    if ( found_it != &s_array[6] )
        cout << "found it: "     << *found_it   << "\n\t"
             << "&s_array[2]:\t" << &s_array[2] << "\n\t"
             << "&found_it:\t"   << found_it    << "\n\n";

    vector< string, allocator > svec( s_array, s_array+6);
    vector< string, allocator > svec_find( to_find, to_find+3 );

    // palauttaa esiintymän "oo" -- svec.end()-2
    vector< string, allocator >::iterator found_it2;

    found_it2 = find_first_of(
                        svec.begin(), svec.end(),
                        svec_find.begin(), svec_find.end(),
                        equal_to<string>() );

    // generoi:
    // found it, too: oo
    //      &svec.end()-2:  0x100067b0
    //      &found_it2:     0x100067b0

    if ( found_it2 != svec.end() )
        cout << "found it, too: "  << *found_it2   << "\n\t"
             << "&svec.end()-2:\t" << svec.end()-2 << "\n\t"
             << "&found_it2:\t"    << found_it2    << "\n";
}
```

for_each()

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

for_each() käyttää func-funktiota jokaiseen elementtiin vuorollaan alueella [first,last). func ei voi kirjoittaa elementteihin (ne ovat syöttöoperaattoreita (IinputIterator), joten ne eivät voi taata sijoitustukea). Jos haluamme muokata elementtejä, pitää käyttää transform()-algoritmia. func saattaa palauttaa arvon, mutta tuo arvo jätetään huomiotta.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
```

```
    vector< int, allocator > ivec;

    for ( int ix = 0; ix < 10; ix++ )
        ivec.push_back( ix );

    void (*pfi)( int ) = print_elements;
    for_each( ivec.begin(), ivec.end(), pfi );

    return 0;
}
```

generate()

```
template< class ForwardIterator, class Generator >
void
generate( ForwardIterator first,
          ForwardIterator last, Generator gen );
```

generate() täyttää jonon alueelta [first,last) käynnistämällä toistuvasti gen:in, joka voi olla joko funktio-olio tai osoitin funktioon.

```
#include <algorithm>
#include <list>
#include <iostream.h>

int odd_by_twos() {
    static int seed = -1;
    return seed += 2;
}

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;

    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // generoi:
    // elements within list the first invocation:
    // 1 3 5 7 9 11 13 15 17 19

    cout << "elements within list the first invocation:\n";
    for_each( ilist.begin(), ilist.end(), pfi );

    generate( ilist.begin(), ilist.end(), odd_by_twos );

    // generoi:
```

```
    // elements within list the second iteration:
    // 21 23 25 27 29 31 33 35 37 39
    cout << "\n\nelements within list the second iteration:\n";
    for_each( ilist.begin(), ilist.end(), pfi );

    return 0;
}
```

generate_n()

```
template< class ForwardIterator,
          class Size, class Generator >
void
generate_n( OutputIterator first, Size n, Generator gen );
```

generate_n() täyttää jonon alkaen kohdasta first ja käynnistää toistuvasti n kertaa gen:in, joka voi olla joko funktio-olio tai osoitin funktioon.

```
#include <algorithm>
#include <iostream.h>
#include <list>

class even_by_twos {
public:
    even_by_twos( int seed = 0 ) : _seed( seed ){}
    int operator()() { return _seed += 2; }
private:
    int _seed;
};

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    list< int, allocator > ilist( 10 );
    void (*pfi)( int ) = print_elements;

    generate_n( ilist.begin(), ilist.size(), even_by_twos() );

    // generoi:
    // generate_n with even_by_twos():
    // 2 4 6 8 10 12 14 16 18 20

    cout << "generate_n with even_by_twos():\n";
    for_each( ilist.begin(), ilist.end(), pfi ); cout << "\n";

    generate_n(ilist.begin(),ilist.size(),even_by_twos(100));

    // generoi:
```

```
    // generate_n with even_by_twos( 100 ):
    // 102 104 106 108 110 112 114 116 118 120

    cout << "generate_n with even_by_twos( 100 ):\n";
    for_each( ilist.begin(), ilist.end(), pfi );
}
```

includes()

```
template< class InputIterator1, class InputIterator2 >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2 );

template< class InputIterator1, class InputIterator2,
    class Compare >
bool
includes( InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          Compare comp );
```

includes() päättelee, sisältyvätkö jonon [first1,last1) elementit jonoon [first2,last2). Ensimmäinen versio olettaa, että jonot on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio käyttää comp:ia päätelläkseen elementtien järjestyksen.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    int ia1[] = { 13, 1, 21, 2, 0, 34, 5, 1, 8, 3, 21, 34 };
    int ia2[] = { 21, 2, 8, 3, 5, 1 };

    // includes pitää välittää lajitelluille säiliöille
    sort( ia1, ia1+12 ); sort( ia2, ia2+6 );

    // generoi: every element of ia2 contained in ia1? true

    bool res = includes( ia1, ia1+12, ia2, ia2+6 );
    cout << "every element of ia2 contained in ia1? "
         << (res ? "true" : "false") << endl;

    vector< int, allocator > ivect1( ia1, ia1+12 );
    vector< int, allocator > ivect2( ia2, ia2+6 );

    // lajittele laskevaan järjestykseen
    sort( ivect1.begin(), ivect1.end(), greater<int>() );
    sort( ivect2.begin(), ivect2.end(), greater<int>() );
```

```
    res = includes( ivect1.begin(), ivect1.end(),
                    ivect2.begin(), ivect2.end(),
                    greater<int>() );

    // generoi:
    // every element of ivect2 contained in ivect1? true

    cout << "every element of ivect2 contained in ivect1? "
         << (res ? "true" : "false") << endl;
}
```

inner_product()

```
template < class InputIterator1, class InputIterator2,
           class Type >
Type
inner_product(
    InputIterator1 first1, InputIterator1 last,
    InputIterator2 first2, Type init );

template < class InputIterator1, class InputIterator2,
           class Type,
           class BinaryOperation1, class BinaryOperation2 >
Type
inner_product(
    InputIterator1 first1, InputIterator1 last,
    InputIterator2 first2, Type init,
    BinaryOperation1 op1, BinaryOperation2 op2 );
```

inner_product():in ensimmäinen versio kerää kahden jonon arvojen tulon ja lisää ne vuorollaan init:in määräämään alkuarvoon. Ensimmäinen jono merkitään alueella [first1,last). Toinen jono alkaa kohdasta first2 ja sitä kasvatetaan ensimmäisen jonon askelilla. Olkoot esimerkiksi kaksi jonoa {2,3,5,8} ja {1,2,3,4}. Tulos on seuraavien tuloparien summa:

```
2*1 + 3*2 + 5*3 + 8*4
```

Jos asetamme alkuarvon 0, on tulos 55.

Toinen versio käyttää binäärioperaatiota op1 oletusarvoisen yhteenlaskuoperaation tilalla ja binäärioperaatiota op2 oletusarvoisen kertomisoperaation tilalla. Jos esimerkiksi käytämme edellistä jonoa ja määritämme vähennyslaskun operandille op1 ja yhteenlaskun operandille op2, tulos on silloin seuraavien yhteenlaskuparien erotus:

```
(2+1) – (3+2) – (5+3) – (8+4)
```

inner_product() on eräs numeerisista algoritmeista. Jotta sitä voi käyttää, pitää ottaa mukaan <numeric>-otsikkotiedosto.

```
#include <numeric>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] =  { 2, 3, 5, 8 };
    int ia2[] = { 1, 2, 3, 4 };

    // kerro kahden taulukon elementtiparit
    // ja lisää sitten alkuarvoon 0

    int res = inner_product( &ia[0], &ia[4], &ia2[0], 0);

    // generoi: inner product of arrays: 55
    cout << "inner product of arrays: "
        << res << endl;

    vector<int, allocator> vec(  ia,  ia+4 );
    vector<int, allocator> vec2( ia2, ia2+4 );

    // lisää kahden vektorin elementtiparit
    // vähennä sitten alkuarvosta 0

    res = inner_product( vec.begin(), vec.end(),
                         vec2.begin(), 0,
                         minus<int>(), plus<int>() );

    // generoi: inner product of vectors: -28
    cout << "inner product of vectors: "
        << res << endl;

    return 0;
}
```

inplace_merge()

```
template< class BidirectionalIterator >
void
inplace_merge( BidirectionalIterator first,
           BidirectionalIterator middle,
           BidirectionalIterator last );

template< class BidirectionalIterator, class Compare >
void
inplace_merge( BidirectionalIterator first,
```

```
                BidirectionalIterator middle,
                BidirectionalIterator last, Compare comp );
```

inplace_merge() yhdistää kaksi lajiteltua, peräkkäistä syöttöjonoa merkityiltä alueilta [first,middle) ja [middle,last). Tuloksena syntyvä jono peittää nuo kaksi aluetta alkaen kohdasta first. Ensimmäinen versio käyttää elementtien lajitteluun taustalla olevan tyypin pienempi kuin -operaattoria. Toinen versio lajittelee elementit binäärivertailun avulla, jonka ohjelmoija välittää.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

/*
 * generoi:
ia sorted into two subarrays:
12 15 17 20 23 26 29 35 40 51 10 16 21 41 44 54 62 65 71 74

ia inplace_merge:
10 12 15 16 17 20 21 23 26 29 35 40 41 44 51 54 62 65 71 74

ivec sorted into two subvectors:
51 40 35 29 26 23 20 17 15 12 74 71 65 62 54 44 41 21 16 10

ivec inplace_merge:
74 71 65 62 54 51 44 41 40 35 29 26 23 21 20 17 16 15 12 10
*/

int main()
{
    int ia[] = { 29,23,20,17,15,26,51,12,35,40,
                 74,16,54,21,44,62,10,41,65,71 };

    vector< int, allocator > ivec( ia, ia+20 );
    void (*pfi)( int ) = print_elements;

    // laita kaksi alijonoa lajiteltuun järjestykseen
    sort( &ia[0], &ia[10] );
    sort( &ia[10], &ia[20] );

    cout << "ia sorted into two sub-arrays: \n";
    for_each( ia, ia+20, pfi ); cout << "\n\n";

    inplace_merge( ia, ia+10, ia+20 );

    cout << "ia inplace_merge:\n";
```

```
    for_each( ia, ia+20, pfi ); cout << "\n\n";

    sort( ivec.begin(),    ivec.begin()+10, greater<int>() );
    sort( ivec.begin()+10, ivec.end(),      greater<int>() );

    cout << "ivec sorted into two sub-vectors: \n";
    for_each( ivec.begin(), ivec.end(), pfi ); cout << "\n\n";

    inplace_merge( ivec.begin(), ivec.begin()+10,
                   ivec.end(),   greater<int>() );

    cout << "ivec inplace_merge:\n";
    for_each( ivec.begin(), ivec.end(), pfi ); cout << endl;
}
```

iter_swap()

```
template <class ForwardIterator1, class ForwardIterator2>
void
iter_swap ( ForwardIterator1 a, ForwardIterator2 b );
```

iter_swap() vaihtaa keskenään arvot, jotka sisältyvät elementteihin, joihin kaksi eteenpäin-iteraattoria (ForwardIterator), a ja b, osoittavat.

```
#include <algorithm>
#include <list>
#include <iostream.h>

int main()
{
    int ia[]  = { 5, 4, 3, 2, 1, 0 };
    list< int,allocator > ilist( ia, ia+6 );

    typedef list< int, allocator >::iterator iterator;
    iterator iter1 = ilist.begin(), iter2,
             iter_end = ilist.end();

    // kuplalajittele lista ...
    for ( ; iter1 != iter_end; ++iter1 )
        for ( iter2 = iter1; iter2 != iter_end; ++iter2 )
            if ( *iter2 < *iter1 )
                iter_swap( iter1, iter2 );

    // generoi tulostuksen:
    // ilist after bubble sort using iter_swap():
// { 0 1 2 3 4 5 }

    cout << "ilist afer bubble sort using iter_swap(): { ";
    for ( iter1 = ilist.begin(); iter1 != iter_end; ++iter1 )
        cout << *iter1 << " ";
```

```
        cout << "}\n";
    }
```

lexicographical_compare()

```
    template <class InputIterator1, class InputIterator2 >
bool
    lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2 );

    template < class InputIterator1, class InputIterator2,
         class Compare >
bool
    lexicographical_compare(
        InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        Compare comp );
```

lexicographical_compare() vertailee kahden jonon elementtipareja, jotka on yksilöity alueilla [first1,last1) ja [first2,last2). Vertailu jatkuu, kunnes jompikumpi elementtipareista ei täsmää, pari [last1,last2] saavutetaan tai joko last1 tai last2 saavutetaan (jos jonot eivät ole yhtä pitkiä). Kun kohdataan ensimmäinen pari, joka ei täsmää, tapahtuu seuraavaa:

- Jos ensimmäisen jonon elementti on pienempi, palauta arvo tosi; muussa tapauksessa palauta epätosi.
- Jos on kohdattu last1, mutta ei last2, palauta tosi.
- Jos on kohdattu last2, mutta ei last1, palauta epätosi.
- Jos sekä last1 että last2 on kohdattu (kaikki elementit täsmäävät), palauta epätosi; eli jos ensimmäinen jono ei ole aakkosten mukaan pienempi kuin toinen jono.

Olkoot esimerkiksi seuraavat kaksi jonoa:

```
string arr1[] = { "Piglet", "Pooh", "Tigger" };
string arr2[] = { "Piglet", "Pooch", "Eeyore" };
```

Algoritmi täsmää ensimmäiseen elementtipariin, mutta ei toiseen. Pooh on suurempi kuin Pooch, koska c on aakkosten mukaan pienempi kuin h (ajattele vertailua niin kuin hakemiston sanaluetteloa). Algoritmi päättyy tässä vaiheessa (kolmatta elementtiparia ei koskaan vertailla). Vertailun tulos on false (epätosi).

Algoritmin toinen versio saa väittävän (predikaatti-) vertailuolion taustalla olevan elementtityypin pienempi kuin -operaattorin sijasta.

```
#include <algorithm>
#include <list>
#include <string>
#include <assert.h>
#include <iostream.h>
```

```
class size_compare {
public:
    bool operator()( const string &a, const string &b ) {
        return a.length() <= b.length();
    }
};

int main()
{
    string arr1[] = { "Piglet", "Pooh", "Tigger" };
    string arr2[] = { "Piglet", "Pooch", "Eeyore" };

    bool res;

    // saa arvon epätosi toisen elementin kohdalla
    // Pooch on pienempi kuin Pooh
    // epätosi saataisiin myös kolmannen elementtiparin kohdalla

    res = lexicographical_compare( arr1, arr1+3,
                                   arr2, arr2+3 );

    assert( res == false );

    // saa arvon tosi: jokaisen ilist2-elementin
    // pituus on pienempi tai yhtäsuuri kuin
    // vastaavan ilist1-elementin

    list< string, allocator > ilist1( arr1, arr1+3 );
    list< string, allocator > ilist2( arr2, arr2+3 );

    res = lexicographical_compare(
          ilist1.begin(), ilist1.end(),
          ilist2.begin(), ilist2.end(), size_compare() );

    assert( res == true );

    cout << "ok: lexicographical_compare succeeded!\n";
}
```

lower_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
lower_bound( ForwardIterator first,
        ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
ForwardIterator
lower_bound( ForwardIterator first,
        ForwardIterator last, const Type &value,
```

```
                Compare comp );
```

lower_bound() palauttaa iteraattorin, joka osoittaa sen alueen lajitellun jonon ensimmäiseen positioon, joka on merkitty arvoilla [first,last), johon value voidaan lisätä sotkematta säiliön lajiteltua järjestystä. Tämä positio merkitsee arvoa, joka on suurempi tai yhtäsuuri kuin value. Olkoon esimerkiksi seuraava jono:

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

Kun lower_bound()-algoritmia kutsutaan arvolla 21, se palauttaa iteraattorin, joka osoittaa arvoon 22. Kun kutsutaan lower_bound()-algoritmia arvolla 22, se palauttaa myös iteraattorin, joka osoittaa arvoon 22. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio lajittelee elementit comp:in perusteella.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    sort( &ia[0], &ia[12] );

    int search_value = 18;
    int *ptr = lower_bound( ia, ia+12, search_value );

    // generoi:
    // The first element 18 can be inserted in front of is 19
    // The previous value is 17

    cout << "The first element "
        << search_value
        << " can be inserted in front of is "
        << *ptr << endl
        << "The previous value is "
        << *(ptr-1) << endl;

    vector< int, allocator > ivec( ia, ia+12 );

    // lajittele laskevaan järjestykseen...
    sort( ivec.begin(), ivec.end(), greater<int>() );

    search_value = 26;
    vector< int, allocator >::iterator iter;

    // täytyy kertoa sopiva lajittelusuhde
    // tässä tapauksessa ...

    iter = lower_bound( ivec.begin(), ivec.end(),
                        search_value, greater<int>() );
```

```
    // generoi:
    // The first element 26 can be inserted in front of is 26
    // The previous value is 29

    cout << "The first element "
        << search_value
        << " can be inserted in front of is "
        << *iter << endl
        << "The previous value is "
        << *(iter-1) << endl;
}
```

max()

```
template< class Type >
const Type&
max( const Type &aval, const Type &bval );

template< class Type, class Compare >
const Type&
max( const Type &aval, const Type &bval, Compare comp );
```

max() palauttaa pienemmän kahdesta elementistä aval ja bval. Ensimmäinen versio käyttää vastaavan Type:n suurempi kuin -operaattoria; toinen versio käyttää vertailuoperaatiota comp.

max_element()

```
template< class ForwardIterator >
ForwardIterator
max_element( ForwardIterator first,
        ForwardIterator last );

template< class ForwardIterator, class Compare >
ForwardIterator
max_element( ForwardIterator first,
        ForwardIterator last, Compare comp );
```

max_element() palauttaa iteraattorin, joka osoittaa jonon sisältämän suurimman elementin arvoon alueella, joka on merkitty arvoilla [first,last). Ensimmäinen versio käyttää taustalla olevan elementtityypin suurempi kuin -operaattoria; toinen versio käyttää vertailuoperaatiota comp.

min()

```
template< class Type >
const Type&
min( const Type &aval, const Type &bval );

template< class Type, class Compare >
const Type&
min( const Type &aval, const Type &bval, Compare comp );
```

min() palauttaa pienemmän kahdesta elementistä aval ja bval. Ensimmäinen versio käyttää Type:n vastaavaa pienempi kuin -operaattoria; toinen versio käyttää vertailuoperaatiota comp.

min_element()

```
template< class ForwardIterator >
ForwardIterator
min_element( ForwardIterator first,
        ForwardIterator last );
template< class ForwardIterator, class Compare >
ForwardIterator
min_element( ForwardIterator first,
        ForwardIterator last, Compare comp );
```

min_element() palauttaa iteraattorin, joka osoittaa pienimmän arvon sisältävään elementtiin jonon alueelta, joka on merkitty [first,last). Ensimmäinen versio käyttää taustalla olevan elementtityypin pienempi kuin -operaattoria; toinen versio käyttää vertailuoperaatiota comp.

```
// kuvaa näiden käyttöä: max(), min(), max_element(), min_element()

#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
   int ia[] = { 7, 5, 2, 4, 3 };
   const vector< int, allocator > ivec( ia, ia+5 );

   int mval = max( max( max( max( ivec[4], ivec[3]),
                          ivec[2]),ivec[1]),ivec[0]);

   // tulostus: the result of nested invocations of max() is: 7
   cout << "the result of nested invocations of max() is: "
        << mval << endl;

   mval = min( min( min( min( ivec[4], ivec[3]),
                       ivec[2]),ivec[1]),ivec[0]);

   // tulostus: the result of nested invocations of min() is: 2
   cout << "the result of nested invocations of min() is: "
      << mval << endl;

   vector< int, allocator >::const_iterator iter;
   iter = max_element( ivec.begin(), ivec.end() );

   // tulostus: the result of invoking max_element() is also: 7
   cout << "the result of invoking max_element() is also: "
      << *iter << endl;
```

```
    iter = min_element( ivec.begin(), ivec.end() );

    // tulostus: the result of invoking min_element() is also: 2
    cout << "the result of invoking min_element() is also: "
         << *iter << endl;
}
```

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 );
```

merge() yhdistelee kaksi lajiteltua jonoa, jotka on merkitty alueilta [first1,last1) ja [first2,last2) yhteen lajiteltuun jonoon, joka alkaa kohdasta result. Palautettu tulostusiteraattori (Output-Iterator) osoittaa yhden yli viimeisen kopioidun elementin uudessa jonossa. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria elementtien lajitteluun; toinen versio lajittelee elementin comp:in perusteella.

```
#include <algorithm>
#include <vector>
#include <list>
#include <deque>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
    int ia[] =  {29,23,20,22,17,15,26,51,19,12,35,40};
    int ia2[] = {74,16,39,54,21,44,62,10,27,41,65,71};

    vector< int, allocator > vec1( ia,  ia +12 ),
                             vec2( ia2, ia2+12 );
    int ia_result[24];
    vector<int,allocator> vec_result(vec1.size()+vec2.size());
```

```
    sort( ia,  ia +12 );
    sort( ia2, ia2+12 );

    // generoi:
    // 10 12 15 16 17 19 20 21 22 23 26 27 29 35
    //          39 40 41 44 51 54 62 65 71 74

    merge( ia, ia+12, ia2, ia2+12, ia_result );
    for_each( ia_result, ia_result+24, pfi ); cout << "\n\n";

    sort( vec1.begin(), vec1.end(), greater<int>() );
    sort( vec2.begin(), vec2.end(), greater<int>() );

    merge( vec1.begin(), vec1.end(),
        vec2.begin(), vec2.end(),
        vec_result.begin(), greater<int>() );

    // generoi:
    // 74 71 65 62 54 51 44 41 40 39 35 29 27 26 23 22
    //                   21 20 19 17 16 15 12 10
    for_each( vec_result.begin(), vec_result.end(), pfi );
    cout << "\n\n";
}
```

mismatch()

```
template< class InputIterator1, class InputIterator2 >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1,
        InputIterator1 last, InputIterator2 first2 );

template< class InputIterator1, class InputIterator2,
        class BinaryPredicate >
pair<InputIterator1, InputIterator2>
mismatch( InputIterator1 first1, InputIterator1 last,
        InputIterator2 first2, BinaryPredicate pred );
```

mismatch() vertailee kahta jonoa rinnakkain ja yksilöi ensimmäisen position, jossa elementit eivät täsmää. Palautetaan iteraattoripari, joka yksilöi ensimmäisen position, jossa elementit eivät täsmää. Jos kaikki elementit täsmäävät, silloin palautetaan iteraattori kummankin säiliön viimeiseen (last) elementtiin. Kun on esimerkiksi jonot meet ja meat, niin palautettaat iteraattorit osoittavat kolmanteen elementtiin. Oletusarvoisesti käytetään yhtäsuuruusoperaattoria vertailuun; toinen versio mahdollistaa käyttäjän määrittää vaihtoehtoinen vertailuoperaatio. Jos toinen jono sisältää enemmän elementtejä kuin ensimmäinen, jätetään nuo elementit huomiotta. Jos toinen jono sisältää vähemmän elementtejä kuin ensimmäinen, on suorituksenaikainen käyttäytyminen tuntematon.

```
#include <algorithm>
#include <list>
```

```
#include <utility>
#include <iostream.h>

class equal_and_odd{
public:
    bool operator()( int ival1, int ival2 )
    {
       // ovatko kaksi arvoa yhtäsuuria ja molemmat nollia tai
       // parittomia?
       return ( ival1 == ival2 &&
            ( ival1 == 0 || ival1%2 ));
    }
};

int main()
{
    int ia[] =  { 0,1,1,2,3,5,8,13 };
    int ia2[] = { 0,1,1,2,4,6,10   };

    pair<int*,int*> pair_ia = mismatch( ia, ia+7, ia2 );

    // generoi: first mismatched pair: ia: 3 and ia2: 4
    cout << "first mismatched pair: ia: "
        << *pair_ia.first << " and ia2: "
        << *pair_ia.second << endl;

    list<int,allocator> ilist(  ia,  ia+7  );
    list<int,allocator> ilist2( ia2, ia2+7 );

    typedef list<int,allocator>::iterator iter;
    pair< iter,iter > pair_ilist =
          mismatch( ilist.begin(),  ilist.end(),
                    ilist2.begin(), equal_and_odd() );

    // generoi:
    // first mismatched pair either not equal or not odd:
    //              ilist: 2 and ilist2: 2

    cout << "first mismatched pair either not equal "
        << "or not odd: \n\tilist: "
        << *pair_ilist.first << " and ilist2: "
        << *pair_ilist.second << endl;
}
```

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 );
```

next_permutation() saa permutaation, joka on merkitty alueelta [first,last) ja järjestää sen uudelleen seuraavaan permutaatioon (katso kohdasta 12.5 käsittely, kuinka edellinen permutaatio on päätelty). Ellei seuraavaa permutaatiota ole, se palauttaa arvon epätosi; muussa tapauksessa se palauttaa arvon tosi. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria permutaation päättelyyn; toinen versio lajittelee elementit comp:in mukaan. Seuraavat next_permutation()-käynnistykset generoivat koko permutaatiojoukon vain, jos merkkijono on lajiteltu. Jos esimerkiksi seuraavassa ohjelmassa emme lajittelisi musil:ia ilmsu:un, emme saisi täyttä permutaatiojoukkoa.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

void print_char( char elem ) { cout << elem ; }
void (*ppc)( char ) = print_char;

/* generoi:
ilmsu  ilmus  ilsmu  ilsum  ilums  ilusm  imlsu  imlus
imslu  imsul  imuls  imusl  islmu  islum  ismlu  ismul
isulm  isuml  iulms  iulsm  iumls  iumsl  iuslm  iusml
limsu  limus  lismu  lisum  liums  liusm  lmisu  lmius
lmsiu  lmsui  lmuis  lmusi  lsimu  lsium  lsmiu  lsmui
lsuim  lsumi  luims  luism  lumis  lumsi  lusim  lusmi
milsu  milus  mislu  misul  miuls  miusl  mlisu  mlius
mlsiu  mlsui  mluis  mlusi  msilu  msiul  msliu  mslui
msuil  msuli  muils  muisl  mulis  mulsi  musil  musli
silmu  silum  simlu  simul  siulm  siuml  slimu  slium
slmiu  slmui  sluim  slumi  smilu  smiul  smliu  smlui
smuil  smuli  suilm  suiml  sulim  sulmi  sumil  sumli
uilms  uilsm  uimls  uimsl  uislm  uisml  ulims  ulism
ulmis  ulmsi  ulsim  ulsmi  umils  umisl  umlis  umlsi
umsil  umsli  usilm  usiml  uslim  uslmi  usmil  usmli
*/

int main()
{
    vector<char,allocator> vec(5);

    // merkkijono: musil
    vec[0] = 'm'; vec[1] = 'u'; vec[2] = 's';
    vec[3] = 'i'; vec[4] = 'l';

    int cnt = 2;
    sort( vec.begin(), vec.end() );
```

```
    for_each( vec.begin(), vec.end(), ppc ); cout << "\t";

    // generoi kaikki "musil"-permutaatiot
    while( next_permutation( vec.begin(), vec.end()))
    {
        for_each( vec.begin(), vec.end(), ppc );
        cout << "\t";

        if ( ! ( cnt++ % 8 )) {
            cout << "\n";
            cnt = 1;
        }
    }

    cout << "\n\n";
    return 0;
}
```

nth_element()

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

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

nth_element() järjestää jonon uudelleen alueelta [first,last) niin, että kaikki elementit, jotka ovat pienempiä kuin nth-elementti, ovat ennen sitä ja kaikki elementit, jotka ovat suurempia, ovat sen jälkeen. Olkoon esimerkiksi taulukko

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

niin nth_element()-algoritmin käynnistys siten, että nth on seitsemäs elementti (sen arvo on 26),

```
nth_element( &ia[0], &ia[6], &ia[12] );
```

johtaa jonoon, jonka vasemmalla puolella on seitsemän elementtiä, jotka ovat pienempiä kuin 26 ja neljä elementtiä, jotka ovat suurempia kuin 26, ovat sen oikealla puolella: {23,20,22,17,15,19,12,26,51,35,40,29}, vaikka nth-elementin kummallakin puolella olevien elementtien ei taata olevan missään tietyssä järjestyksessä. Ensimmäinen versio käyttää vertailuun taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio järjestää elementit binärivertailuoperaation avulla, jonka ohjelmoija välittää.

```
#include <algorithm>
#include <vector>
```

```
#include <iostream.h>

/*
 * generoi:
original order of the vector: 29 23 20 22 17 15 26 51 19 12 35 40
sorting vector based on element 26
12 15 17 19 20 22 23 26 51 29 35 40
sorting vector in descending order based on element 23
40 35 29 51 26 23 22 20 19 17 15 12
*/

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    vector< int,allocator > vec( ia, ia+12 );
    ostream_iterator<int> out( cout," " );

    cout << "original order of the vector: ";
    copy( vec.begin(), vec.end(), out ); cout << endl;

    cout << "sorting vector based on element "
        << *( vec.begin()+6 ) << endl;

    nth_element( vec.begin(), vec.begin()+6, vec.end() );
    copy( vec.begin(), vec.end(), out ); cout << endl;

    cout << "sorting vector in descending order "
        << "based on element "
        << *( vec.begin()+6 ) << endl;

    nth_element( vec.begin(), vec.begin()+6,
            vec.end(),   greater<int>() );

    copy( vec.begin(), vec.end(), out ); cout << endl;
}
```

partial_sort()

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

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

partial_sort() lajittelee niin monta elementtiä, kuin voidaan sijoittaa alueelle [first,middle). Elementit, jotka on tallennettu alueelle [middle,last), ovat lajittelematta, mutta jäävät varsinaisesti lajitellun jonon ulkopuolelle. Olkoon esimerkiksi taulukko

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
```

Kun partial_sort() käynnistetään ja merkitään kuudes elementti middle:ksi,

```
stable_sort( &ia[0], &ia[5], &ia[12] );
```

se johtaa jonoon, jossa viisi pienintä elementtiä on lajiteltu (tarkoittaa middle-first elementtejä): {12,15,17,19,20,29,23,22,26,51,35,40}. Elementtejä väliltä middle - last-1 ei laiteta mihinkään erityiseen järjestykseen, vaikka niiden arvot jäävät varsinaisesti lajitellun jonon ulkopuolelle. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio lajittelee elementin comp:in perusteella.

partial_sort_copy()

```
template< class InputIterator, class RandomAccessIterator >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last );

template< class InputIterator, class RandomAccessIterator,
          class Compare >
RandomAccessIterator
partial_sort_copy( InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last,
                   Compare comp );
```

partial_sort_copy() käyttäytyy samoin kuin partial_sort() paitsi, että se kopioi sen osittain lajitellun jonon säiliöön, joka on merkitty alueella [result_ first,result_last) (joten tulos on täysin lajiteltu jono edellyttäen, että ilmaisemme erillisen säiliön, johon kopioidaan). Jos on esimerkiksi kaksi taulukkoa

```
int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
int ia2[5];
```

niin partial_sort_copy()-käynnistys, jossa middle tarkoittaa kahdeksatta elementtiä

```
stable_sort( &ia[0],  &ia[7], &ia[12],
             &ia2[0], &ia[5] );
```

täyttää ia2:n viidellä lajitellulla elementillä: {12,15,17,19,20}. Kaksi lajiteltua lisäelementtiä jäävät käyttämättä.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
/*
```

```
 * generoi:
  original order of vector: 69 23 80 42 17 15 26 51 19 12 35 8
  partial sort of vector: seven elements
  8 12 15 17 19 23 26 80 69 51 42 35
  partial_sort_copy() of first seven elements
  of vector in descending order
  26 23 19 17 15 12 8
 */

int main()
{
     int ia[] = {69,23,80,42,17,15,26,51,19,12,35,8 };
     vector< int,allocator > vec( ia, ia+12 );
     ostream_iterator<int> out( cout," " );

     cout << "original order of vector: ";
     copy( vec.begin(), vec.end(), out ); cout << endl;

     cout << "partial sort of vector: seven elements\n";
     partial_sort( vec.begin(), vec.begin()+7, vec.end() );
     copy( vec.begin(), vec.end(), out ); cout << endl;

     vector< int, allocator > res(7);
     cout << "partial_sort_copy() of first seven elements\n\t"
          << "of vector in descending order\n";

     partial_sort_copy( vec.begin(), vec.begin()+7, res.begin(),
                        res.end(), greater<int>() );
     copy( res.begin(), res.end(), out ); cout << endl;
}
```

partial_sum()

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

template < class InputIterator, Class OutputIterator,
           class BinaryOperation >
OutputIterator
partial_sum(
          InputIterator first, InputIterator last,
          OutputIterator result, BinaryOperation op );
```

Ensimmäinen partial_sum()-versio luo uuden elementtijonon, jossa jokainen uusi elementti edustaa kaikkien edellisten elementtien summaa sen omaan positioon saakka alueella, joka on merkitty arvoilla [first,last). Olkoon esimerkiksi jono {0,1,1,2,3,5,8}, niin uusi jono on {0,1,2,4,7,12,20}. Esimerkiksi neljäs elementti on osasumma sen kolmesta edellisestä arvosta

(0,1,1) plus sen oma (2), joka johtaa arvoon 4.

Toinen versio käyttää binäärioperaatiota, jonka ohjelmoija välittää. Olkoon esimerkiksi jono {1,2,3,4} ja välittäkäämme sille times<int>-funktio-olio. Tulosjono on {1,2,6,24}. Molemmissa versioissa tulostusiteraattori (OutputIterator) osoittaa yhden yli uuden jonon viimeisen elementin.

partial_sum() on eräs aritmeettisista algoritmeista ja edellyttää <numeric>-vakio-otsikkotiedoston mukaanoton.

```
#include <numeric>
#include <vector>
#include <iostream.h>

/*
 * generoi:
   elements: 1 3 4 5 7 8 9
   partial sum of elements:
   1 4 8 13 20 28 37
   partial sum of elements using times<int>():
   1 3 12 60 420 3360 30240
*/

int main()
{
      const int ia_size = 7;
      int ia[ ia_size ] = { 1, 3, 4, 5, 7, 8, 9 };
      int ia_res[ ia_size ];

      ostream_iterator< int  > outfile( cout, " "  );
      vector< int, allocator > vec( ia, ia+ia_size );
      vector< int, allocator > vec_res( vec.size() );

      cout << "elements: ";
      copy( ia, ia+ia_size, outfile ); cout << endl;

      cout << "partial sum of elements:\n";
      partial_sum( ia, ia+ia_size, ia_res );
      copy( ia_res, ia_res+ia_size, outfile ); cout << endl;

      cout << "partial sum of elements using times<int>():\n";
      partial_sum( vec.begin(), vec.end(), vec_res.begin(),
                  times<int>() );

      copy( vec_res.begin(), vec_res.end(), outfile );
      cout << endl;
}
```

partition()

```
template < class BidirectionalIterator, class UnaryPredicate >
```

```
BidirectionalIterator
partition( BidirectionalIterator first,
        BidirectionalIterator last, UnaryPredicate pred );
```

partition() lajittelee elementit uudelleen merkityltä alueelta [first,last). Kaikki elementit, jotka saavat arvon tosi läpäistyään unaaripredikaatin pred, sijoitetaan ennen elementtejä, jotka saavat arvon epätosi. Olkoon esimerkiksi jono {0,1,2,3,4,5,6} ja predikaatti, joka testaa elementtien parillisuutta, niin tosi- ja epätosielementtien alueet ovat {0,2,4,6} ja {1,3,5}. Vaikka on taattu, että kaikki parilliset elementit sijoittuvat ennen yhtäkään parittomista elementeistä, ei elementtien suhteellisen position taata säilyvän uudelleenjärjestelyssä. Arvo 4 voidaan siis sijoittaa ennen arvoa 2 tai 5 ennen arvoa 3. On taattu, että stable_partition(), joka käsitellään myöhemmin, säilyttää elementtien suhteellisen järjestyksen säiliössä.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

class even_elem {
public:
   bool operator()( int elem )
       { return elem%2 ? false : true; }
};

/*
 * generoi:
  original order of elements:
  29 23 20 22 17 15 26 51 19 12 35 40
  partition based on whether element is even:
  40 12 20 22 26 15 17 51 19 23 35 29
  partition based on whether element is less than 25:
  12 23 20 22 17 15 19 51 26 29 35 40
*/

int main()
{
     const int ia_size = 12;
     int ia[ia_size]   = { 29,23,20,22,17,15,26,51,19,12,35,40 };

     vector< int, allocator > vec( ia, ia+ia_size );
     ostream_iterator< int >  outfile( cout, " "  );

     cout << "original order of elements: \n";
     copy( vec.begin(), vec.end(), outfile ); cout << endl;

     cout << "partition based on whether element is even:\n";
     partition( &ia[0], &ia[ia_size], even_elem() );
     copy( ia, ia+ia_size, outfile ); cout << endl;

     cout << "partition based on whether element "
```

```
            << "is less than 25:\n";

        partition( vec.begin(), vec.end(), bind2nd(less<int>(),25) );
        copy( vec.begin(), vec.end(), outfile ); cout << endl;
    }
```

prev_permutation()

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

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

prev_permutation() saa permutaation, joka on merkitty alueelle [first,last). Algoritmi järjestää sen uudelleen edellisen permutaation mukaiseen järjestykseen (katso kohdasta 12.5, kuinka edellinen permutaatio päätellään). Ellei edellistä permutaatiota ole, se palauttaa arvon epätosi; muussa tapauksessa se palauttaa arvon tosi. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria edellisen permutaation päättelyyn; toinen versio lajittelee elementit binäärivertailuoperaation perusteella, jonka ohjelmoija välittää.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

// generoi:
// n d a  n a d  d n a  d a n  a n d  a d n

int main()
{
    vector< char, allocator > vec( 3 );
    ostream_iterator< char > out_stream( cout, " " );

    vec[0] = 'n'; vec[1] = 'd'; vec[2] = 'a';
    copy( vec.begin(), vec.end(), out_stream ); cout << "\t";

    // generoi kaikki "dan"-permutaatiot
    while( prev_permutation( vec.begin(), vec.end() )) {
        copy( vec.begin(), vec.end(), out_stream );
        cout << "\t";
    }

    cout << "\n\n";
}
```

random_shuffle()

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

template< class RandomAccessIterator,
      class RandomNumberGenerator >
void
random_shuffle( RandomAccessIterator first,
        RandomAccessIterator last,
        RandomNumberGenerator rand );
```

random_shuffle() järjestää elementit uudelleen satunnaisesti merkityllä alueella [first,last). Toinen versio saa satunnaislukuja generoivan funktio-olion tai osoittimen funktioon. Oletus on, että rand palauttaa double-tyyppisen arvon väliltä [0,1].

```
#include <algorithm>
#include <vector>
#include <iostream.h>

int main()
{
    vector< int, allocator > vec;
    for ( int ix = 0; ix < 20; ix++ )
        vec.push_back( ix );

    random_shuffle( vec.begin(), vec.end() );

    // generoi:
    // random_shuffle of sequence of values 1 .. 20:
    // 6 11 9 2 18 12 17 7 0 15 4 8 10 5 1 19 13 3 14 16
    cout << "random_shuffle of sequence of values 1 .. 20:\n";
    copy( vec.begin(), vec.end(),
        ostream_iterator< int >( cout," " ));
}
```

remove()

```
template< class ForwardIterator, class Type >
ForwardIterator
remove( ForwardIterator first,
      ForwardIterator last, const Type &value );
```

remove() poistaa kaikki value-ilmentymät määritetyltä alueelta [first,last). remove() (kuten myös remove_if()) ei todellisuudessa poista täsmänneitä elementtejä säiliöstä (säiliön koko säilyy). Sen sijaan jokainen täsmäämätön elementti sijoitetaan vuorollaan ensimmäiseen vapaaseen paikkaan alkaen kohdasta first. Palautettu eteenpäiniteraattori (ForwardIterator) osoittaa yhden yli uuden alueen elementtien. Mietitäänpä esimerkiksi jonoa {0,1,0,2,0,3,0,4}. Sanokaamme, että

haluamme poistaa kaikki 0-arvot. Tulosjono on {1,2,3,4,0,3,0,4}. Arvo 1 kopioidaan ensimmäiseen “lokeroon”, arvo 2 toiseen, arvo 3 kolmanteen ja arvo 4 neljänteen lokeroon. Arvo 0 viidennessä lokerossa edustaa “tähteitä”. Palautettu ForwardIterator osoittaa arvoon 0 lokerossa 5. Tyypillisesti tämä iteraattori välitetään sitten erase()-algoritmille, joka poistaa kelpaamattomat elementit. (Sisäinen taulukko ei sovi remove()- ja remove_if()-algoritmeille, koska sen kokoa ei voi muuttaa helposti. Tästä syystä remove_copy()- ja remove_copy_if()-algoritmeja on parempi käyttää taulukon kanssa.)

remove_copy()

```
template< class InputIterator, class OutputIterator,
          class Type >
OutputIterator
remove_copy( InputIterator first, InputIterator last,
             OutputIterator result, const Type &value );
```

remove_copy() kopioi kaikki täsmäämättömät elementit säiliöön, joka on määritetty tulostusiteraattorilla result. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun elementin. Alkuperäinen säiliö jää muuttumattomaksi.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* generoi:
   original vector sequence:
   0 1 0 2 0 3 0 4 0 5
   vector after remove, without applying erase():
   1 2 3 4 5 3 0 4 0 5
   vector after erase():
   1 2 3 4 5
array after remove_copy():
   1 2 3 4 5
 */

int main()
{
    int value = 0;
    int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

    vector< int, allocator > vec( ia, ia+10 );
    ostream_iterator< int >  ofile( cout," ");
    vector< int, allocator >::iterator vec_iter;

    cout << "original vector sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vec_iter = remove( vec.begin(), vec.end(), value );
```

```
    cout << "vector after remove, without applying erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    // poista kelpaamattomat elementit säiliöstä
    vec.erase( vec_iter, vec.end() );

    cout << "vector after erase():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    int ia2[5];
    vector< int, allocator > vec2( ia, ia+10 );
    remove_copy( vec2.begin(), vec2.end(), ia2, value );

    cout << "array after remove_copy():\n";
    copy( ia2, ia2+5 ofile ); cout << endl;
}
```

remove_if()

```
template< class ForwardIterator, class Predicate >
ForwardIterator
remove_if( ForwardIterator first,
        ForwardIterator last, Predicate pred );
```

remove_if() poistaa kaikki elementit alueelta, joka on merkitty arvoilla [first, last) ja jossa pred saa arvon tosi. remove_if() (kuten myös remove()) ei todellisuudessa poista täsmänneitä elementtejä säiliöstä. Sen sijaan jokainen täsmäämätön elementti sijoitetaan vuorollaan ensimmäiseen vapaaseen paikkaan alkaen kohdasta first. Palautettu ForwardIterator osoittaa yhden yli uuden elementtialueen. Tyypillisesti tämä iteraattori välitetään sitten erase()-algoritmille, joka varsinaisesti poistaa kelpaamattomat elementit. (remove_copy_if() sopii paremmin sisäisten taulukoiden käyttöön.)

remove_copy_if()

```
template< class InputIterator, class OutputIterator,
          class Predicate >
OutputIterator
remove_copy_if( InputIterator first, InputIterator last,
                OutputIterator result, Predicate pred );
```

remove_copy_if() kopioi kaikki täsmäämättömät elementit säiliöön, joka on määritetty tulostusoperaattorilla result. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun elementin. Alkuperäinen säiliö ei muutu.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* generoi:
  original element sequence:
  0 1 1 2 3 5 8 13 21 34
  sequence after applying remove_if < 10:
  13 21 34
  sequence after applying remove_copy_if even:
  1 1 3 5 13 21
*/

class EvenValue {
public:
    bool operator()( int value ) {
        return value % 2 ? false : true; }
};

int main()
{
    int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };

    vector< int, allocator >::iterator iter;
    vector< int, allocator >  vec( ia, ia+10 );
    ostream_iterator< int >   ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    iter = remove_if( vec.begin(), vec.end(),
                bind2nd(less<int>(),10) );
    vec.erase( iter, vec.end() );

    cout << "sequence after applying remove_if < 10:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vector< int, allocator > vec_res( 10 );
```

```
    iter = remove_copy_if( ia, ia+10,
                    vec_res.begin(), EvenValue() );

    cout << "sequence after applying remove_copy_if even:\n";
    copy( vec_res.begin(), iter, ofile ); cout << '\n';
}
```

replace()

```
template< class ForwardIterator, class Type >
void
replace( ForwardIterator first, ForwardIterator last,
         const Type& old_value, const Type& new_value );
```

replace() korvaa kaikki old_value-ilmentymät new_value-arvolla alueelta, jonka iteraattoripari [first, last) määrittää.

replace_copy()

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

replace_copy() tekee saman operaation kuin replace(), paitsi että uusi jono kopioidaan säiliöön alkaen kohdasta result. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun arvon. Alkuperäinen jono ei muutu.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/* generoi:
  original element sequence:
  Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
  sequence after applying replace():
  Christopher Robin Pooh Piglet Tigger Eeyore
  sequence after applying replace_copy():
  Christopher Robin Mr. Winnie the Pooh Piglet Tigger Eeyore
 */

int main()
{
    string oldval( "Mr. Winnie the Pooh" );
    string newval( "Pooh" );

    ostream_iterator< string >  ofile( cout, " " );
    string sa[] = {
```

```
        "Christopher Robin", "Mr. Winnie the Pooh",
        "Piglet", "Tigger", "Eeyore"
    };

    vector< string, allocator > vec( sa, sa+5 );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    replace( vec.begin(), vec.end(), oldval, newval );

    cout << "sequence after applying replace():\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vector< string, allocator > vec2;
    replace_copy( vec.begin(), vec.end(),
                  inserter( vec2, vec2.begin() ),
                  newval, oldval );

    cout << "sequence after applying replace_copy():\n";
    copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
}
```

replace_if()

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

replace_if() korvaa kaikki elementit määritetyltä alueelta [first, last) arvolla new_value, joilla pred saa arvon tosi.

replace_copy_if()

```
template< class ForwardIterator, class OutputIterator,
          class Predicate, class Type >
OutputIterator
replace_copy_if( ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 Predicate pred, const Type& new_value );
```

replace_copy_if() käyttäytyy samoin kuin replace_if(), paitsi että uusi jono kopioidaan säiliöön alkaen kohdasta result. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun elementin. Alkuperäinen jono ei muutu.

```
#include <algorithm>
#include <vector>
#include <iostream.h>
```

```
/*
 * generoi:
  original element sequence:
  0 1 1 2 3 5 8 13 21 34
  sequence after applying replace_if < 10 with 0:
  0 0 0 0 0 0 0 13 21 34
  sequence after applying replace_if even with 0:
  0 1 1 0 3 5 0 13 21 0
 */

class EvenValue {
public:
    bool operator()( int value ) {
        return value % 2 ? false : true; }
};

int main()
{
    int new_value = 0;

    int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    vector< int, allocator > vec( ia, ia+10 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( ia, ia+10, ofile ); cout << '\n';

    replace_if( &ia[0], &ia[10],
            bind2nd(less<int>(),10), new_value );

    cout << "sequence after applying replace_if < 10 with 0:\n";
    copy( ia, ia+10, ofile ); cout << '\n';

    replace_if( vec.begin(), vec.end(),
            EvenValue(), new_value );

    cout << "sequence after applying replace_if even with 0:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

reverse()

```
template< class BidirectionalIterator >
void
reverse( BidirectionalIterator first,
        BidirectionalIterator last );
```

reverse() kääntää elementtien järjestyksen päinvastaiseksi alueelta, joka on merkitty arvoilla [first,last). Jos on esimerkiksi jono {0,1,1,2,3}, niin päinvastainen järjestys on {3,2,1,1,0}.

reverse_copy()

```
template< class BidirectionalIterator, class OutputIterator >
OutputIterator
reverse_copy( BidirectionalIterator first,
        BidirectionalIterator last, OutputIterator result );
```

reverse_copy() käyttäytyy samoin kuin reverse(), paitsi että uusi jono kopioidaan säiliöön alkaen kohdasta result. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun elementin. Alkuperäinen jono ei muutu.

```
#include <algorithm>
#include <list>
#include <string>
#include <iostream.h>

/*
 * generoi:
   Original sequence of strings:
      Signature of all things I am here to
      read seaspawn and seawrack that rusty boot

   Sequence after reverse() applied:
      boot rusty that seawrack and seaspawn read to
      here am I things all of Signature
 */

class print_elements {
public:
    void operator()( string elem ) {
        cout << elem
            << ( _line_cnt++%8 ? " " : "\n\t" );
    }

    static void reset_line_cnt() { _line_cnt = 1; }

private:
    static int _line_cnt;
};

int print_elements::_line_cnt = 1;

int main()
{
    string sa[] = { "Signature", "of", "all", "things",
                "I", "am", "here", "to", "read",
                "seaspawn", "and", "seawrack", "that",
                "rusty", "boot"
    };
```

```
    list< string, allocator > slist( sa, sa+15 );

    cout << "Original sequence of strings:\n\t";
    for_each( slist.begin(), slist.end(), print_elements() );
    cout << "\n\n";

    reverse( slist.begin(), slist.end() );

    print_elements::reset_line_cnt();

    cout << "Sequence after reverse() applied:\n\t";
    for_each( slist.begin(), slist.end(), print_elements() );
    cout << "\n";

    list< string, allocator > slist_copy( slist.size() );
    reverse_copy( slist.begin(), slist.end(),
                  slist_copy.begin() );
}
```

rotate()

```
template< class ForwardIterator >
void
rotate( ForwardIterator first,
      ForwardIterator middle, ForwardIterator last );
```

rotate() siirtää alueelta [first,middle) elementit säiliön loppuun. Elementistä, johon middle osoittaa, tulee säiliön ensimmäinen elementti. Esimerkiksi sana "hissboo" muuttuu 'b'-elementin ympäripyöräytyksen jälkeen tällaiseksi: "boohiss".

rotate_copy()

```
template< class ForwardIterator, class OutputIterator >
OutputIterator
rotate_copy( ForwardIterator first, ForwardIterator middle,
             ForwardIterator last,  OutputIterator result );
```

rotate_copy() käyttäytyy samoin kuin rotate() paitsi, että ympäripyöräytetty jono kopioidaan säiliön loppuun kohdasta result alkaen. Palautettu OutputIterator osoittaa yhden yli viimeisen kopioidun elementin. Alkuperäinen jono ei muutu.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
  original element sequence:
  1 3 5 7 9 0 2 4 6 8 10
  rotate on middle element(0) ::
```

```
  0 2 4 6 8 10 1 3 5 7 9
  rotate on next to last element(8) ::
  8 10 1 3 5 7 9 0 2 4 6
  rotate_copy on middle element ::
  7 9 0 2 4 6 8 10 1 3 5
 */

int main()
{
    int ia[] = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10 };

    vector< int, allocator > vec( ia, ia+11 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    rotate( &ia[0], &ia[5], &ia[11] );

    cout << "rotate on middle element(0) ::\n";
    copy( ia, ia+11, ofile ); cout << '\n';

    rotate( vec.begin(), vec.end()-2, vec.end() );

    cout << "rotate on next to last element(8) ::\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    vector< int, allocator > vec_res( vec.size() );

    rotate_copy( vec.begin(), vec.begin()+vec.size()/2,
                 vec.end(), vec_res.begin() );

    cout << "rotate_copy on middle element ::\n";
    copy( vec_res.begin(), vec_res.end(), ofile );
    cout << '\n';
}
```

search()

```
template< class ForwardIterator1, class ForwardIterator2 >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2 );

template< class ForwardIterator1, class ForwardIterator2,
          class BinaryPredicate >
ForwardIterator
search( ForwardIterator1 first1, ForwardIterator1 last1,
```

```
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred );
```

Kun search()-algoritmille annetaan kaksi aluetta, se palauttaa iteraattorin, joka osoittaa merkityn alueen [first1,last1) ensimmäiseen positioon, jossa toinen jono esiintyy alijonona. Ellei alijonoa esiinny, palautetaan last1. Esimerkiksi sanassa Mississippi esiintyy alijono iss kaksi kertaa ja search() palauttaa iteraattorin ensimmäisen ilmentymän alkuun. Oletusarvoisesti käytetään yhtäsuuruusoperaattoria elementtien vertailuun; toinen versio mahdollistaa käyttäjän antaa käytettävän vertailuoperaation.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
   Expecting to find the substring 'ate': a t e
   Expecting to find the substring 'vat': v a t
 */

int main()
{
    ostream_iterator< char > ofile( cout, " " );

    char str[ 25 ] = "a fine and private place";
    char substr[]  = "ate";

    char *found_str = search(str,str+25,substr,substr+3);

    cout << "Expecting to find the substring 'ate': ";
    copy( found_str, found_str+3, ofile ); cout << '\n';

    vector< char, allocator > vec( str, str+24 );
    vector< char, allocator > subvec(3);

    subvec[0]='v'; subvec[1]='a'; subvec[2]='t';

    vector< char, allocator >::iterator iter;
    iter = search( vec.begin(), vec.end(),
                   subvec.begin(), subvec.end(),
                   equal_to< char >() );

    cout << "Expecting to find the substring 'vat': ";
    copy( iter, iter+3, ofile ); cout << '\n';
}
```

search_n()

```
template< class ForwardIterator, class Size, class Type >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
        Size count, const Type &value );

template< class ForwardIterator, class Size,
        class Type, class BinaryPredicate >
ForwardIterator
search_n( ForwardIterator first, ForwardIterator last,
        Size count, const Type &value, BinaryPredicate pred );
```

search_n() etsii alijonon value-ilmentymää count-kappaletta jonosta, jonka alue on merkitty arvoilla [first,last). Jos value-ilmentymiä ei löydy count kappaletta, palautetaan last. Jotta löydettäisiin esimerkiksi ss-alijono jonosta Mississippi, value asetettaisiin arvoon 's' ja count arvoon 2. Vaihtoehtoisesti, jotta löydettäisiin kaksi ilmentymää alijonosta ssi, value asetettaisiin arvoon "ssi" ja count jälleen kerran arvoon 2. search_n() palauttaa iteraattorin ensimmäiseen löydettyyn value-elementtiin. Oletusarvoisesti käytetään yhtäsuuruusoperaattoria elementtien vertailuun; toinen versio mahdollistaa käyttäjän antaa käytettävä vertailuoperaatio.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
   Expecting to find two instances of 'o': o o
   Expecting to find the substring 'mou':  m o u
 */

int main()
{
    ostream_iterator< char >  ofile( cout, " " );

    const char blank = ' ';
    const char oh    = 'o';

    char str[ 26 ]  = "oh my a mouse ate a moose";
    char *found_str = search_n( str, str+25, 2, oh );

    cout << "Expecting to find two instances of 'o': ";
    copy( found_str, found_str+2, ofile ); cout << '\n';

    vector< char, allocator > vec( str, str+25 );

    // etsi ensimmäinen jono, jonka kolme ensimmäistä kirjainta
    // eivät ole tyhjiä merkkejä: esimerkiksi mou sanasta mouse
```

```
    vector< char, allocator >::iterator iter;
    iter = search_n( vec.begin(), vec.end(), 3,
                     blank, not_equal_to< char >() );

    cout << "Expecting to find the substring 'mou':  ";
    copy( iter, iter+3, ofile ); cout << '\n';
}
```

set_difference()

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

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

set_difference() muodostaa lajitellun jonon elementeistä, jotka löytyvät ensimmäisestä jonosta (alue merkitty arvoilla [first1,last1)), mutta eivät sisälly toiseen jonoon (alue merkitty arvoilla [first2,last2)). Jos on esimerkiksi kaksi jonoa, {0,1,2,3} ja {0,2,4,6}, niin erotusjoukko on {1,3}. Palautettu OutputIterator osoittaa yhden yli säiliöön sijoitetun viimeisen elementin, joka alkaa kohdasta result. Ensimmäinen versio olettaa, että jonot on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio olettaa, että jonot on lajiteltu käyttäen comp:ia.

set_intersection()

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

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

set_intersection() muodostaa lajitellun jonon elementeistä, jotka löytyvät molemmista jonoista, joiden alueet on merkitty arvoilla [first1,last1) ja [first2,last2). Jos on esimerkiksi kaksi jonoa, {0,1,2,3} ja {0,2,4,6}, niiden leikkaus on {0,2}. Elementti kopioidaan ensimmäisestä jonosta. Palautettu OutputIterator osoittaa yhden yli viimeisen säiliöön sijoitetun elementin, jonka alue alkaa kohdasta result. Ensimmäinen versio olettaa, että jonot on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio olettaa, että jonot on lajiteltu käyttäen comp:ia.

set_symmetric_difference()

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

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

set_symmetric_difference() muodostaa lajitellun jonon elementeistä, jotka löytyvät ensimmäisestä säiliöstä, mutta eivät toisesta, ja jotka löytyvät toisesta, mutta eivät ensimmäisestä. Jos on esimerkiksi kaksi jonoa, {0,1,2,3} ja {0,2,4,6}, niin symmetrinen erotus on {1,3,4,6}. Palautettu OutputIterator osoittaa yhden yli viimeisen säiliöön sijoitetun elementin, jonka alue alkaa kohdasta result. Ensimmäinen versio olettaa, että jonot on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio olettaa, että jonot on lajiteltu käyttäen comp:ia.

set_union()

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

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

set_union() muodostaa lajitellun jonon elementtien arvoista, jotka sisältyvät kahdelle merkitylle alueelle [first1,last1) ja [first2,last2). Jos on esimerkiksi kaksi jonoa, {0,1,2,3} ja {0,2,4,6}, niin niiden yhdiste on {0,1,2,3,4,6}. Jos elementti löytyy molemmista säiliöistä kuten 0 ja 2 esimerkissämme, elementti kopioidaan ensimmäisestä säiliöstä. Palautettu OutputIterator osoittaa yhden yli viimeisen säiliöön sijoitetun elementin alueelta, joka alkaa kohdasta result. Ensimmäinen versio olettaa, että jonot on lajiteltu käyttäen taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio olettaa, että jonot on lajiteltu käyttäen comp:ia.

```
#include <algorithm>
#include <set>
#include <string>
#include <iostream.h>

/*
 * generoi:
  set #1 elements:
      Eeyore Piglet Pooh Tigger

  set #2 elements:
      Heffalump Pooh Woozles

  set_union() elements:
      Eeyore Heffalump Piglet Pooh Tigger Woozles

  set_intersection() elements:
      Pooh

  set_difference() elements:
      Eeyore Piglet Tigger

  set_symmetric_difference() elements:
      Eeyore Heffalump Piglet Tigger Woozles
 */

int main()
{
    string str1[] = { "Pooh", "Piglet", "Tigger", "Eeyore" };
    string str2[] = { "Pooh", "Heffalump", "Woozles" };
    ostream_iterator< string >  ofile( cout, " " );

    set<string,less<string>,allocator> set1( str1, str1+4 );
    set<string,less<string>,allocator> set2( str2, str2+3 );

    cout << "set #1 elements:\n\t";
    copy( set1.begin(), set1.end(), ofile ); cout << "\n\n";

    cout << "set #2 elements:\n\t";
    copy( set2.begin(), set2.end(), ofile ); cout << "\n\n";
```

```
    set<string,less<string>,allocator> res;
    set_union( set1.begin(), set1.end(),
               set2.begin(), set2.end(),
               inserter( res, res.begin() ));

    cout << "set_union() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_intersection( set1.begin(), set1.end(),
                      set2.begin(), set2.end(),
                      inserter( res, res.begin() ));

    cout << "set_intersection() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_difference( set1.begin(), set1.end(),
                    set2.begin(), set2.end(),
                    inserter( res, res.begin() ));

    cout << "set_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";

    res.clear();
    set_symmetric_difference( set1.begin(), set1.end(),
                              set2.begin(), set2.end(),
                              inserter( res, res.begin() ));

    cout << "set_symmetric_difference() elements:\n\t";
    copy( res.begin(), res.end(), ofile ); cout << "\n\n";
}
```

sort()

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

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

sort() lajittelee elementit uudelleen alueelta, joka on merkitty arvoilla [first,last) nousevaan järjestykseen käyttäen taustalla olevan tyypin pienempi kuin -operaattoria. Toinen versio lajittelee elementit käyttäen comp:ia. (Jos haluat säilyttää samanlaisten elementtien järjestyksen, käytä stable_sort()-algoritmia sort()-algoritmin sijasta.) Emme esitä esimerkkiohjelmaa sort()-algoritmin käytöstä, koska sitä käytetään monissa muissa esimerkkiohjelmissa kuten binary_search(), equal_range() ja inplace_merge().

stable_partition()

```
template< class BidirectionalIterator, class Predicate >
BidirectionalIterator
stable_partition( BidirectionalIterator first,
                  BidirectionalIterator last,
                  Predicate pred );
```

stable_partition() käyttäytyy täsmälleen samoin kuin partition(), paitsi että stable_partition() takaa säilyttävänsä säiliön elementtien suhteellisen järjestyksen. Tässä on sama ohjelma, jota käytettiin partition()-algoritmin yhteydessä, mutta muokattu käynnistämään stable_partition():

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
   original element sequence:
   29 23 20 22 17 15 26 51 19 12 35 40
   stable_partition on even element:
   20 22 26 12 40 29 23 17 15 51 19
   stable_partition of less than 25:
   23 20 22 17 15 19 12 29 26 51 35 40
 */

class even_elem {
public:
     bool operator()( int elem ) {
          return elem%2 ? false : true;
     }
};

int main()
{
     int ia[] = { 29,23,20,22,17,15,26,51,19,12,35,40 };
     vector< int, allocator > vec( ia, ia+12 );
     ostream_iterator< int >  ofile( cout, " " );

     cout << "original element sequence:\n";
     copy( vec.begin(), vec.end(), ofile ); cout << '\n';
```

```
    stable_partition( &ia[0], &ia[12], even_elem() );

    cout << "stable_partition on even element:\n";
    copy( ia, ia+11, ofile ); cout << '\n';

    stable_partition( vec.begin(), vec.end(),
                bind2nd(less<int>(),25)  );

    cout << "stable_partition of less than 25:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

stable_sort()

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

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

stable_sort() säilyttää säiliön samanlaisten elementtien järjestyksen alueella, joka on merkitty arvoilla [first,last), lajitellessaan ne nousevaan järjestykseen käyttäen taustalla olevan tyypin pienempi kuin -operaattoria. Toinen versio lajittelee elementit comp:in perusteella.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
  original element sequence:
  29 23 20 22 12 17 15 26 51 19 12 23 35 40
  stable sort -- default ascending order:
  12 12 15 17 19 20 22 23 23 26 29 35 40 51
  stable sort: descending order:
  51 40 35 29 26 23 23 22 20 19 17 15 12 12
 */

int main()
{
    int ia[] = { 29,23,20,22,12,17,15,26,51,19,12,23,35,40 };
    vector< int, allocator > vec( ia, ia+14 );
    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
```

```
    stable_sort( &ia[0], &ia[14] );

    cout << "stable sort -- default ascending order:\n";
    copy( ia, ia+14, ofile ); cout << '\n';

    stable_sort( vec.begin(), vec.end(), greater<int>() );

    cout << "stable sort: descending order:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

swap()

```
template< class Type >
void
swap ( Type &ob1, Type &ob2 );
```

swap() vaihtaa arvot keskenään olioista ob1 ja ob2.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
   original element sequence:
   3 4 5 0 1 2
   sequence applying swap() to support bubble sort:
   0 1 2 3 4 5
 */

int main()
{
    int ia[]  = { 3, 4, 5, 0, 1, 2 };
    vector< int, allocator > vec( ia, ia+6 );

    for ( int ix = 0; ix < 6; ++ix )
    for ( int iy = ix; iy < 6; ++iy ) {
    if ( vec[iy] < vec[ ix ] )
    swap( vec[iy], vec[ix] );
    }

    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence:\n";
    copy( ia, ia+6, ofile ); cout << '\n';

    cout << "sequence applying swap() "
        << "to support bubble sort:\n";
```

```
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';
}
```

swap_range()

```
template <class ForwardIterator1, class ForwardIterator2 >
ForwardIterator2
swap_range( ForwardIterator1 first1, ForwardIterator1 last,
        ForwardIterator2 first2 );
```

swap_range() vaihtaa keskenään alueen [first1,last) elementtien arvot niiden arvojen kanssa, jotka alkavat kohdasta first2. Nuo kaksi jonoa voivat olla joko epäyhtenäisiä samassa säiliössä tai kahdessa eri säiliössä. Suorituksenaikainen käyttäytyminen on tuntematon, jos jono, joka on merkitty alkamaan kohdasta first2, on pienempi kuin merkitty alue [first1,last) tai jos nuo kaksi jonoa menevät päällekkäin samassa säiliössä. swap_range() palauttaa iteraattorin toisesta jonosta, joka osoittaa yhden yli viimeisen vaihdetun elementin.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

/*
 * generoi:
  original element sequence of first container:
  0 1 2 3 4 5 6 7 8 9
  original element sequence of second container:
  5 6 7 8 9
  array after swap_ranges() in middle of array:
  5 6 7 8 9 0 1 2 3 4
  first container after swap_ranges() of two vectors:
  5 6 7 8 9 5 6 7 8 9
  second container after swap_ranges() of two vectors:
  0 1 2 3 4
 */

int main()
{
    int ia[]  = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    int ia2[] = { 5, 6, 7, 8, 9 };

    vector< int, allocator > vec( ia, ia+10 );
    vector< int, allocator > vec2( ia2, ia2+5 );

    ostream_iterator< int >  ofile( cout, " " );

    cout << "original element sequence of first container:\n";
    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    cout << "original element sequence of second container:\n";
```

```
    copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';

    // arvojen vaihto keskenään samassa jonossa
    swap_ranges( &ia[0], &ia[5], &ia[5] );

    cout << "array after swap_ranges() in middle of array:\n";
    copy( ia, ia+10, ofile ); cout << '\n';

    // arvojen vaihto keskenään eri säiliöiden välillä
    vector< int, allocator >::iterator last =
            find( vec.begin(), vec.end(), 5 );

    swap_ranges( vec.begin(), last, vec2.begin() );

    cout << "first container after "
        << "swap_ranges() of two vectors:\n";

    copy( vec.begin(), vec.end(), ofile ); cout << '\n';

    cout << "second container after "
        << "swap_ranges() of two vectors:\n";

    copy( vec2.begin(), vec2.end(), ofile ); cout << '\n';
}
```

transform()

```
template< class InputIterator, class OutputIterator,
          class UnaryOperation >
OutputIterator
transform( InputIterator first, InputIterator last,
           OutputIterator result, UnaryOperation op );

template< class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation >
OutputIterator
transform( InputIterator1 first1, InputIterator1 last,
           InputIterator2 first2, OutputIterator result,
           BinaryOperation bop );
```

Ensimmäinen tranform()-versio generoi elementtijonon käynnistäen jokaiselle alueen [first,last) elementille op:n. Jos on esimerkiksi jono {0,1,1,2,3,5} ja funktio-olio Double, joka tuplaa jokaisen elementin, tulosjono on {0,2,2,4,6,10}.

Toinen versio generoi elementtijonon käynnistäen bop:in jokaiselle elementtiparille, joista ensimmäinen on jonon alueelta [first1,last) ja toinen jonosta, joka alkaa kohdasta first2. Suorituksenaikainen käyttäytyminen on tuntematon, jos toinen jono sisältää vähemmän elementtejä kuin ensimmäinen jono. Jos on esimerkiksi jonot {1,3,5,9} ja {2,4,6,8} sekä funktio-olio AddAndDouble, joka lisää kaksi elementtiä ja sitten tuplaa niiden summan, tulosjono on {6,14,22,34}.

Molemmat transform()-versiot sijoittavat tulosjonon säiliöön, joka alkaa kohdasta result. result voi osoittaa johonkin syöttösäiliöistä eli itse asiassa korvata nykyiset elementit transform():in palauttamilla elementeillä. Palautettu OutputIterator osoittaa yhden yli viimeisen tulosjonoon sijoitetun elementin.

```
#include <algorithm>
#include <vector>
#include <math.h>
#include <iostream.h>

/*
 * generoi:
   original array values: 3 5 8 13 21
   transform each element by doubling: 6 10 16 26 42
   transform each element by difference: 3 5 8 13 21
 */

int double_val( int val ) { return val + val; }
int difference( int val1, int val2 ) {
      return abs( val1 - val2 ); }

int main()
{
     int ia[]  = { 3, 5, 8, 13, 21 };
     vector<int, allocator> vec( 5 );
     ostream_iterator<int> outfile( cout, " " );

     cout << "original array values: ";
     copy( ia, ia+5, outfile ); cout << endl;

     cout << "transform each element by doubling: ";
     transform( ia, ia+5, vec.begin(), double_val );
     copy( vec.begin(), vec.end(), outfile ); cout << endl;

     cout << "transform each element by difference: ";
     transform( ia, ia+5, vec.begin(), outfile, difference );
     cout << endl;
}
```

unique()

```
template< class ForwardIterator >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last );

template< class ForwardIterator, class BinaryPredicate >
ForwardIterator
unique( ForwardIterator first,
        ForwardIterator last, BinaryPredicate pred );
```

Kaikki peräkkäiset elementtiryhmät, jotka sisältävät joko saman arvon (käyttäen taustalla olevan tyypin yhtäsuuruusoperaattoria) tai saavat arvon tosi, kun ne välitetään pred-binääripredikaatille, kootaan yhdeksi elementiksi. Siten esimerkiksi sanan mississippi *semanttinen* tulos on "misisipi". Huomaa, että koska kolme i:tä eivät ole peräkkäin, niitä ei koota yhteen. Samalla tavalla kahta s-paria ei koota yhdeksi ilmentymäksi, koska ne eivät ole peräkkäin. Voidaksemme taata, että kaikki tuplaelementit kootaan yhdeksi, meidän pitäisi lajitella säiliö ensin.

Itse asiassa unique() käyttäytyy hieman samalla tavalla kuin remove()-algoritmi. Molemmissa tapauksissa säiliön todellinen koko ei muutu. Jokainen tuplaelementti sijoitetaan vuorollaan seuraavaan vapaaseen paikkaan, joka alkaa kohdasta first.

Tästä syystä esimerkkimme *fyysinen* tulos on misisipippi, jossa merkkijono ppi edustaa algoritmin niin sanottuja "tähteitä". Palautettu ForwardIterator osoittaa tähteiden alkuun. Tyypillisesti tämä iteraattori välitetään sitten erase()-algoritmille, joka tuhoaa kelpaamattomat elementit. (Koska sisäinen taulukko ei tue erase()-operaatiota, ei unique() ole oikein sovelias sille; unique_copy() on sopivampi.)

unique_copy()

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

template< class InputIterator, class OutputIterator,
          class BinaryPredicate >
OutputIterator
unique_copy( InputIterator first,   InputIterator last,
             OutputIterator result, BinaryPredicate pred );
```

unique_copy() kopioi yhden ilmentymän jokaisesta peräkkäisestä elementtiryhmästä, joka joko sisältää saman arvon (käyttäen taustalla olevan tyypin yhtäsuuruusoperaattoria) tai saa aikaan arvon tosi, kun se välitetään pred-binääripredikaatille (katso kuvaus unique()-algoritmista). Jotta voitaisiin taata, että kaikki tuplaelementit poistetaan, pitää säiliö ensin lajitella. Palautettu OutputIterator osoittaa kohdesäiliön loppuun.

```
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;
void (*pfs)( string ) = print_elements;

int main()
```

```
{
    int ia[] = { 0, 1, 0, 2, 0, 3, 0, 4, 0, 5 };

    vector<int,allocator> vec( ia, ia+10 );
    vector<int,allocator>::iterator vec_iter;

    // tuloksena muuttumaton jono: 0:t eivät ole peräkkäin
    // generoi: 0 1 0 2 0 3 0 4 0 5
    vec_iter = unique( vec.begin(), vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // lajittele vektori: 0 0 0 0 0 1 2 3 4 5
// then apply unique:
    // generoi: 0 1 2 3 4 5 2 3 4 5

    sort( vec.begin(), vec.end() );
    vec_iter = unique( vec.begin(), vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // poista kelpaamattomat elementit säiliöstä
    // generoi: 0 1 2 3 4 5

    vec.erase( vec_iter, vec.end() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    string sa[] = { "enough", "is", "enough",
                    "enough", "is", "good"
    };

    vector<string,allocator> svec( sa, sa+6 );
    vector<string,allocator> vec_result( svec.size() );
    vector<string,allocator>::iterator svec_iter;

    sort( svec.begin(), svec.end() );
    svec_iter = unique_copy( svec.begin(), svec.end(),
                             vec_result.begin() );

    // generoi: enough good is
    for_each( vec_result.begin(), svec_iter, pfs );
    cout << "\n\n";
}
```

upper_bound()

```
template< class ForwardIterator, class Type >
ForwardIterator
upper_bound( ForwardIterator first,
             ForwardIterator last, const Type &value );

template< class ForwardIterator, class Type, class Compare >
```

```
ForwardIterator
upper_bound( ForwardIterator first,
        ForwardIterator last, const Type &value,
        Compare comp );
```

upper_bound() palauttaa iteraattorin, joka osoittaa lajitellun, iteraattoriparilla [first,last) merkityn jonon viimeiseen positioon, johon value-arvo voidaan lisätä rikkomatta säiliön lajiteltua järjestystä. Tuo positio osoittaa arvoon, joka on suurempi kuin value. Olkoon esimerkiksi seuraava jono

```
int ia[] = {12,15,17,19,20,22,23,26,29,35,40,51};
```

niin upper_bound()-kutsu arvolla 21 palauttaa iteraattorin, joka osoittaa arvoon 22. Kun kutsutaan upper_bound()-algoritmia arvolla 22, palautetaan iteraattori, joka osoittaa arvoon 23. Ensimmäinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria; toinen versio lajittelee elementit comp:n perusteella.

```
#include <algorithm>
#include <vector>
#include <assert.h>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

void (*pfi)( int ) = print_elements;

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40};
    vector<int,allocator> vec(ia,ia+12);

    sort(ia,ia+12);
    int *iter = upper_bound(ia,ia+12,19);
    assert( *iter == 20 );

    sort( vec.begin(), vec.end(), greater<int>() );
    vector<int,allocator>::iterator iter_vec;

    iter_vec = upper_bound( vec.begin(), vec.end(),
                    27, greater<int>() );

    assert( *iter_vec == 26 );

    // generoi: 51 40 35 29 27 26 23 22 20 19 17 15 12
    vec.insert( iter_vec, 27 );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
}
```

Kekoalgoritmit

Vakiokirjaston keko on *maksimikeko* (*max-heap*). Maksimikeko on binääripuun muoto, joka esitetään taulukkona, jossa jokaisen solmun avainarvo on joko suurempi tai yhtäsuuri kuin sen alisolmujen avainarvot (katso julkaisusta [SEDGEWICK88] aiheen koko käsittely). (Vaihtoehtoinen esitystapa on *minimikeko* (*min-heap*), jossa jokaisen solmun avaimen arvo on pienempi tai yhtäsuuri kuin sen alisolmujen avainarvot.) Vakiokirjaston esitystavassa suurimman avaimen arvo (ajattele sen olevan puun juuri) on aina taulukon alku. Esimerkiksi seuraava kirjainten jono täyttää keon vaatimukset:

```
kirjainten jono täyttää keon vaatimukset
X T O G S M N A E R A I
```

Tässä esimerkissä X on juurisolmu, jonka vasemmalla puolella on alisolmu T ja oikealla puolella alisolmu O. Huomaa, että alisolmujen ei tarvitse olla lajitellussa järjestyksessä (tarkoittaa, että vasemman alisolmun ei tarvitse olla pienempi kuin oikean alisolmun). G ja S ovat T:n alisolmuja, kun taas M ja N ovat O:n alisolmuja. Samalla tavalla A ja E ovat G:n alisolmuja, R ja A ovat S:n alisolmuja, I on M:n vasen solmu ja N on lehti ilman alisolmuja.

On olemassa neljä geneeristä kekoalgoritmia, jotka tukevat keon luontia ja käsittelyä: make_heap(), pop_heap(), push_heap() ja sort_heap(). Viimeiset kolme algoritmia edellyttävät, että iteraattoriparilla merkitty jono edustaa todellista kekoa (ellei näin ole, on suorituksenaikainen käyttäytyminen tuntematon). Huomaa, että koska listasäiliö ei tue hajakäsittelyä, sitä ei voi käyttää keon kanssa. Vaikka sisäistä taulukkoa voidaan käyttää keon tukena, on pop_heap()- ja push_heap()-algoritmeja vaikea käyttää niiden kanssa, koska nuo kaksi algoritmia vaativat taulukon koon muuttamismahdollisuuden. Esittelemme lyhyesti jokaisen näistä neljästä algoritmista ja kuvaamme sitten niiden käyttöä pienellä ohjelmalla.

make_heap()

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

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

make_heap() sijoittaa kekoon elementit merkityltä alueelta [first,last). Kaksiargumenttinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria lajitteluun; toinen versio lajittelee elementit comp:in perusteella.

pop_heap()

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

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

pop_heap() ei itse asiassa vedä suurinta elementtiä keosta, vaan järjestää keon uudelleen. Se vaihtaa keskenään arvot first ja last-1 ja sitten sijoittaa jonon uudelleen kekoon käyttäen aluetta [first,last-1). Voimme sitten käsitellä "vedettyä" elementtiä käyttäen säiliön jäsenoperaatiota back() tai todella poistaa sen käyttäen pop_back():iä. Kaksiargumenttinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria lajitteluun; toinen versio lajittelee elementit comp:in perusteella.

push_heap()

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

push_heap() olettaa, että jono, joka on merkitty alueella [first,last-1), on jo kelvollinen keko ja että kekoon lisättävä uusi elementti on positiossa last-1. Se sijoittaa jonon uudelleen kekoon käyttäen aluetta [first,last). Ennen kuin push_heap() käynnistetään, pitää lisätä elementti säiliön loppuun; ehkä käyttäen push_back()-operaattoria (tämä on kuvattu seuraavassa ohjelmaesimerkissä). Kaksiargumenttinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria lajitteluun; toinen versio lajittelee elementit comp:in perusteella.

sort_heap()

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

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

sort_heap() lajittelee jonon alueella [first,last). Se olettaa, että jono on kelvollinen keko (sen käyttäytyminen on muutoin tuntematon). (Tietystikään lajiteltu keko ei ole enää kelvollinen keko!) Kaksiargumenttinen versio käyttää taustalla olevan tyypin pienempi kuin -operaattoria lajitteluun; toinen versio lajittelee elementit comp:in perusteella.

```
#include <algorithm>
#include <vector>
#include <iostream.h>

template <class Type>
void print_elements( Type elem ) { cout << elem << " "; }

int main()
{
    int ia[] = {29,23,20,22,17,15,26,51,19,12,35,40 };
    vector< int, allocator > vec( ia, ia+12 );


    // generoi: 51 35 40 23 29 20 26 22 19 12 17 15
    make_heap( &ia[0], &ia[12] );
    void (*pfi)( int ) = print_elements;
    for_each( ia, ia+12, pfi ); cout << "\n\n";

    // generoi: 12 17 15 19 23 20 26 51 22 29 35 40
    // minimikeko: juuri on pienin elementti

    make_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";

    // generoi: 12 15 17 19 20 22 23 26 29 35 40 51
    sort_heap( ia, ia+12 );
    for_each(  ia, ia+12, pfi ); cout << "\n\n";

    // lisää uusi, pienin elementti:
    vec.push_back( 8 );

    // generoi: 8 17 12 19 23 15 26 51 22 29 35 40 20
    // tulisi sijoittaa uusin, pienin elementti juureksi

    push_heap( vec.begin(), vec.end(), greater<int>() );
    for_each(  vec.begin(), vec.end(), pfi ); cout << "\n\n";
```

```
    // generoi: 12 17 15 19 23 20 26 51 22 29 35 40 8
    // tulisi korvata pienin elementti toiseksi pienimmällä

    pop_heap( vec.begin(), vec.end(), greater<int>() );
    for_each( vec.begin(), vec.end(), pfi ); cout << "\n\n";
}
```