12

Geneeriset algoritmit

Teimme luvussa 2 Array-luokkaamme jäsenfunktioita, jotka tukivat operaatioita min(), max(), find() ja sort(). Vakiovektoriluokassa ei kuitenkaan ole näitä ilmeisen perustavaa laatua olevia operaatioita. Jotta löytäisimme minimiarvon ja maksimiarvon vektorin elementeistä, pitää käynnistää jokin *geneerisistä algoritmeista*: "algoritmeista", koska ne toteuttavat yleisen operaation kuten min(), max(), find() ja sort(); "geneerisistä", koska ne toimivat lukuisilla säiliötyypeillä — ei vain esimerkiksi vektori- ja listatyypeillä, vaan myös sisäisellä taulukkotyypillä. Säiliö sidotaan geneeriseen algoritmiin iteraattoriparilla (käsittelimme iteraattorit lyhyesti kohdassa 6.5), joka ilmaisee läpikäytävät elementit. Erityisillä *funktio-olioilla* voimme korvata oletusoperaattoreiden merkityksen geneerisissä algoritmeissa. Geneeriset algoritmit, funktio-oliot ja iteraattorien yksityiskohtainen käsittely muodostavat tämän luvun aiheen.

12.1 Yleiskatsaus

Jokainen geneerinen algoritmi on toteutettu riippumattomaksi yksittäisistä säiliötyypeistä. Koska algoritmin tyyppiriippuvuus on poistettu, yksi malli-ilmentymä voi toimia kaikilla säiliötyypeillä yhtä hyvin kuin sisäisellä taulukkotyypillä. Mietitäänpä find()-algoritmia. Koska se on riippumaton säiliöstä, sitä käytetään seuraavia yleisiä vaiheita noudattaen ja olettaen, että kokoelma on lajittelematon:

1. Tutki jokainen elementti vuorollaan.
2. Jos elementti on yhtäsuuri kuin arvo, jota etsimme, palauta silloin elementin positio kokoelmassa.
3. Muussa tapauksessa tutki seuraava elementti ja toista vaihe 2, kunnes arvo löytyy tai kaikki elementit on tutkittu.
4. Jos olemme saavuttaneet kokoelman lopun emmekä ole löytäneet arvoa, palauta jokin arvo, joka ilmaisee, että arvo ei sisälly kokoelmaan.

Ilmaisemamme algoritmi on riippumaton säiliön tyypistä, johon sitä käytetään tai arvon tyypistä, johon etsintä on johtanut. Algoritmin vaatimukset ovat seuraavat:

1. Tarvitsemme jonkin tavan käydä kokoelma läpi. Tähän kuuluu ilmaisu siirtymisestä seuraavaan elementtiin ja sen havaitseminen, kun seuraava käsiteltävä elementti edustaa kokoelman loppua. Ratkaisemme tämän sisäisellä taulukkotyypillä (paitsi C-tyylisellä merkkijonolla) tavallisesti välittämällä kaksi argumenttia: osoittimen ensimmäiseen elementtiin ja läpikäytävien elementtien lukumäärän. C-tyylisillä merkkijonoilla elementtien lukumäärä ei ole tarpeellinen, sillä merkkijonon loppu ilmaistaan null-merkillä.
2. Jokaista säiliön elementtiä pitää pystyä vertaamaan etsimäämme arvoon. Tyypillisesti tämä ratkaistaan joko käyttämällä yhtäsuuruusoperaattoria, joka liittyy arvon taustalla olevaan tyyppiin, tai välittämällä osoitin funktioon, joka tekee vertailuoperaation.
3. Tarvitsemme yleisen tyypin, joka edustaa sekä elementin positiota säiliössä että tietoa "*ei positiota*", ellei elementtiä löydy. Tyypillisesti palautamme elementin indeksin, arvon -1, osoittimen elementtiin tai arvon 0.

Geneeriset algoritmit ratkaisevat ensimmäisen ongelman eli säiliön läpikäymisen iteraattori-abstraktion avulla. Iteraattorit ovat osoittimen yleistyksiä. Ne tukevat vähintään lisäysoperaattoria, jolla päästään seuraavaan elementtiin, käänteisoperaattoria, jolla varsinaista elementtiä käsitellään ja yhtäsuuruus- sekä erisuuruusoperaattoreita, joilla päätellään, ovatko kaksi iteraattoria yhtäsuuria. Läpikäytävien elementtien alue merkitään iteraattoriparilla: `first`-iteraattorilla, joka osoittaa ensimmäiseen operoitavaan elementtiin ja `last`-iteraattorilla, joka osoittaa yhden yli viimeisen operoitavan elementin. Elementti, johon `last` osoittaa, ei itse kuulu operoitavien joukkoon; se toimii aivan kuin *tuntomerkkinä*, että läpikäynti voi päättyä. Se toimii myös paluuarvona ja ilmaisee positiota "ei mihinkään". Jos arvo löytyy, palautetaan iteraattori, joka ilmaisee tuon position.

Geneeriset algoritmit ratkaisevat vaatimuksen 2 eli arvon vertailun kahdella eri versiolla jokaisesta algoritmista: yksi, joka käyttää elementin taustalla olevan tyypin yhtäsuuruusoperaattoria ja toinen, joka käyttää funktio-oliota eli osoitinta funktioon, joka toteuttaa vertailun (funktio-oliot käsitellään kohdassa 12.3). Tässä on esimerkiksi `find()`-algoritmin yleinen toteutus, jossa käytetään taustalla olevan elementin tyypin yhtäsuuruusoperaattoria:

```
template < class ForwardIterator, class Type >
ForwardIterator
find( ForwardIterator first, ForwardIterator last, Type value )
{
    for ( ; first != last; ++first )
        if ( value == *first )
            return first;
    return last;
}
```

ForwardIterator on yksi viidestä iteraattorikategoriasta, jotka on esimääritelty vakiokirjastoon. ForwardIterator tukee osoittamansa elementin sekä lukemista että kirjoittamista. (Nuo viisi kategoriaa esitellään kohdassa 12.4.)

Algoritmit saavuttavat tyyppiriippumattomuutensa siten, että ne eivät koskaan käsittele suoraan säiliön elementtejä; sen sijaan elementtien kaikki käsittely ja läpikäynti toteutetaan iteraattorien avulla. Todellinen säiliötyyppi (on se sitten säiliötyyppi tai sisäinen taulukko) on tuntematon. Jotta sisäisiä taulukoita pystyttäisiin tukemaan, voidaan geneerisille algoritmeille välittää tavallisia osoittimia kuten myös iteraattoreita. Tässä on esimerkki, jossa find()-algoritmia käytetään sisäiselle int-tyyppiselle taulukolle:

```
#include <algorithm>
#include <iostream>

int main()
{
      int search_value;
      int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };

      cout << "enter search value: ";
      cin  >> search_value;

      int *presult = find( &ia[0], &ia[6], search_value );

      cout << "The value " << search_value
    << ( presult == &ia[6]
                ? " is not present" : " is present" )
          << endl;
}
```

Jos palautetun osoittimen osoite on yhtä kuin osoite &ia[6] (yhden yli ia:n viimeisen elementin), on etsintä epäonnistunut; muussa tapauksessa arvo löytyi.

Yleensä, kun elementtitaulukon osoite välitetään geneeriselle algoritmille, voimme kirjoittaa joko

```
int *presult = find( &ia[0], &ia[6], search_value );
```

tai vähemmän pakottaen

```
int *presult = find( ia, ia+6, search_value );
```

Jos haluamme välittää osa-alueen, muokkaamme yksinkertaisesti algoritmille välitettävien osoitteiden indeksejä. Esimerkiksi tässä find()-algoritmin käynnistyksessä etsitään vain toisesta ja kolmannesta elementistä (muista, että elementit on numeroitu alkaen nollasta):

```
// etsitään vain elementeistä ia[1] ja ia[2]
int *presult = find( &ia[1], &ia[3], search_value );
```

Tässä käytetään vektorisäiliötä find()-algoritmissa:

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

int main()
{
      int search_value;
      int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
      vector<int> vec( ia, ia+6 );

      cout << "enter search value: ";
      cin  >> search_value;

      vector<int>::iterator presult;
      presult = find( vec.begin(), vec.end(), search_value );

      cout << "The value " << search_value
    << ( presult == vec.end()
                ? " is not present" : " is present" )
         << endl;
}
```

Samalla tavalla seuraavassa on käytetty listasäiliötä find()-algoritmissa:

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

int main()
{
     int search_value;
     int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
     list<int> ilist( ia, ia+6 );

     cout << "enter search value: ";
     cin >> search_value;

     list<int>::iterator presult;
     presult = find( ilist.begin(), ilist.end(), search_value );

     cout << "The value " << search_value
   << ( presult == ilist.end()
              ? " is not present" : " is present" )
        << endl;
}
```

Seuraavassa kohdassa käymme läpi ohjelmasuunnittelun, jossa käytetään lukuisia geneerisiä algoritmeja. Tuon kohdan jälkeen esittelemme funktio-oliot. Kohdassa 12.4 esitellään joitakin lisätietoja iteraattorista. Esittelemme laajasti geneerisiä algoritmeja kohdassa 12.5 — yksityiskohtaiset kuvaukset jokaisesta algoritmista on koottu tämän kirjan liitteeseen. Päätämme tämän luvun pohdiskeluun, milloin geneeristen algoritmien käyttö ei ole sopivaa.

Harjoitus 12.1

Eräs kritiikki geneeristen algoritmien suunnittelua kohtaan on, että ne lisäävät tarkkuusvaatimuksia ohjelmoijalle. Esimerkiksi kelpaamaton iteraattori tai -pari, joka ilmaisee kelpaamatonta aluetta, johtaa ohjelman epävakaaseen tilaan. Kuinka oikeutettua tämä kritiikki on? Tulisiko algoritmien käyttö rajoittaa vain kokeneimmille ohjelmoijille? Tulisiko yleensä ohjelmoijat suojata mahdollisilta virhealttiilta rakenteilta kuten geneerisiltä algoritmeilta, osoittimilta ja pakotetuilta tyyppimuunnoksilta?

12.2 Geneeristen algoritmien käyttö

Mietitäänpä seuraavaa ohjelmoijan tehtävää. Sanokaamme, että ajattelimme kirjoittaa lasten kuvakirjan ja haluaisimme saada sellaisille kirjoille sopivan sanavarastotason. Ideamme on seuraava: luemme tekstiä joistakin lasten kuvakirjoista ja tallennamme tekstin yksittäisiin merkkijonovektoreihin (tiedämme jo, kuinka se tehdään — katso kohta 6.7). Näin me sen teemme:

1. Tee kopio jokaisesta vektorista.
2. Yhdistä viisi vektoria yhdeksi isoksi vektoriksi.
3. Lajittele iso vektori aakkosjärjestykseen.
4. Poista kaikki tuplasanat.
5. Lajittele se jälleen sanojen pituuden mukaan.
6. Laske niiden sanojen lukumäärä, jotka ovat pitempiä kuin kuusi merkkiä (pituus on oletettavasti monimutkaisuuden mitta ainakin sanavaraston ehdoilla).
7. Poista merkitykseltään neutraalit sanat (kuten and, if, or, but jne.).
8. Tulosta vektori.

Toivottavasti tämä kuulostaa ainakin yhden luvun mittaiselta tehtävältä. Kuitenkin geneerisiä algoritmeja käyttäen voimme pienentää tehtävän työn tämän luvun alikohdan kokoiseksi.

Funktiomme argumentti on vektori, joka muodostuu merkkijonovektoreita sisältävästä vektorista. Hyväksymme sen osoittimena ja testaamme aluksi, että sen arvo ei ole null:

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

typedef vector<string> textwords;
void process_vocab( vector<textwords>*pvec )
{
      if ( ! pvec ) {
            // anna varoitusilmoitus
               return;
      }

      // ...
}
```

Ensimmäiseksi haluamme luoda yhden vektorin, joka muodostuu yksittäisten vektoreiden elementeistä. Voimme tehdä sen käyttämällä geneeristä copy()-algoritmia kuten seuraavassa (mukaan pitää ottaa algorithm- ja iterator-vakio-otsikkotiedostot):

```
#include <algorithm>
#include <iterator>

void process_vocab( vector< textwords >*pvec )
{
      // ...
      vector< string > texts;

      vector<textwords>::iterator iter = pvec->begin();
      for ( ; iter != pvec->end(); ++iter )
        copy( (*iter).begin(), (*iter).end(),
                  back_inserter( texts ));

    // ...
}
```

copy()-algoritmi saa kahdessa ensimmäisessä argumentissa iteraattoriparin, joka ilmaisee kopioitavan elementtialueen. Kolmas argumentti on iteraattori, joka ilmaisee, mistä kopioitujen elementtien sijoittaminen alkaa. back_inserter-funktiota sanotaan *iteraattorisovittimeksi* (*iterator adaptor*), sillä se saa aikaan elementtien kopioitumisen sille välitetyn vetkoriargumentin loppuun. (Katsomme iteraattorisovittimia tarkemmin kohdassa 12.4.)

unique() poistaa duplikaattiarvot säiliöstä, mutta vain ne, jotka ovat toistensa vieressä. Esimerkiksi jonon 01123211 tuloksena on 012321 eikä 0123. Jotta saavuttaisimme jälkimmäisen tuloksen, pitää vektori ensiksi lajitella (sort()); tämä tarkoittaa, että jonosta 01111223 tulee 0123. No, ainakin melkein. Todellisuudessa tulos on 01231223.

unique() käyttäytyy hieman omituisesti: sen käsittelemän säiliön koko ei muutu. Sen sijaan jokainen yksittäinen elementti sijoitetaan vuorollaan ensimmäiseen vapaaseen elementtiin ensimmäisestä alkaen. Esimerkissämme fyysinen tulos on 01231223; jono 1223 edustaa algoritmista *tähteiksi* jäänyttä osaa. unique() palauttaa iteraattorin, joka osoittaa tämän tähteiksi jääneen osan alkuun. Tyypillisesti tämä iteraattori välitetään sitten säiliön vastaavalle erase()-operaatiolle, jolla kelpaamattomat elementit poistetaan. (Koska sisäinen taulukko ei tue erase()-operaatiota, unique()-algoritmien perhe ei sovi kovin hyvin sille.) Tässä on tämä osa funktiostamme:

```
void process_vocab( vector< textwords >*pvec )
{
    // ...
    // lajittele tekstielementit
    sort( texts.begin(), texts.end() );

    // poista kaikki duplikaattielementit
    vector<string>::iterator it;
    it = unique( texts.begin(), texts.end() );
    texts.erase( it, texts.end() );

    // ...
}
```

Tässä on tulostusmalli tekstistä (texts), jossa on yhdistetty kaksi pientä tekstitiedostoa sort()-funktion jälkeen, mutta ennen unique():n käynnistämistä:

```
a a a a alice alive almost
alternately ancient and and and and and and
and as asks at at beautiful becomes bird
bird blows blue bounded but by calling coat
daddy daddy daddy dark darkened darkening distant each
either emma eternity falls fear fiery fiery flight
flowing for grow hair hair has he heaven,
held her her her her him him home
houses i immeasurable immensity in in in in
inexpressibly is is is it it it its
journeying lands leave leave life like long looks
magical mean more night, no not not not
now now of of on one one one
passion puts quite red rises row same says
she she shush shyly sight sky so so
star star still stone such tell tells tells
that that the the the the the the
the there there thing through time to to
to to trees unravel untamed wanting watch what
when wind with with you you you you
your your
```

Sen jälkeen, kun unique():ta on sovellettu ja käynnistetty erase(), tekstivektori näyttää tältä:

```
a alice alive almost alternately ancient
and as asks at beautiful becomes bird blows
blue bounded but by calling coat daddy dark
darkened darkening distant each either emma eternity falls
fear fiery flight flowing for grow hair has
he heaven, held her him home houses i
immeasurable immensity in inexpressibly is it its journeying
lands leave life like long looks magical mean
more night, no not now of on one
passion puts quite red rises row same says
she shush shyly sight sky so star still
stone such tell tells that the there thing
through time to trees unravel untamed wanting watch
what when wind with you your
```

Seuraava tehtävämme on lajitella merkkijonot pituuksiensa mukaan. Jotta voisimme sen tehdä, emme käytä sort()-, vaan stable_sort()-algoritmia. stable_sort() säilyttää yhtäsuurien elementtien suhteellisen paikan; kun elementit ovat samanpituisia, niiden aakkosjärjestys ylläpidetään. Jotta voisimme lajitella pituuden perusteella, teemme oman pienempi kuin -vertailuoperaation. Eräs tapa sen tekemiseen on seuraava:

```
bool less_than( const string & s1, const string & s2 )
{
    return s1.size() < s2.size();
}

void process_vocab( vector< textwords >*pvec )
{
    // ...
    // lajittele tekstielementit pituuden mukaan
    // ja säilytä elementtien aikaisempi järjestys
    stable_sort( texts.begin(), texts.end(), less_than );

    // ...
}
```

Vaikka työ saadaan tällä tehtyä, se on huomattavasti tehottomampi kuin voisimme toivoa. less_than() on toteutettu yhdellä lauseella. Normaalisti se käynnistettäisiin välittömänä funktiona. Kuitenkin välittämällä se osoittimena funktioon, estämme sen, että siitä tehtäisiin välitön. Vaihtoehtoinen strategia säilyttää operaation välittömyys on käyttää *funktio-oliota*. Esimerkiksi:

```
// funktio-olio -- operaatio on
// toteutettu operator():in ilmentymänä
class LessThan {
public:
    bool operator()( const string & s1, const string & s2 )
            { return s1.size() < s2.size(); }
};
```

Funktio-olio on luokka, jossa kutsuoperaattori (()) on ylikuormitettu. Kutsuoperaattorin runko toteuttaa funktion tehtävän eli pienempi kuin -vertailun. Kutsuoperaattorin määrittely näyttää hieman kummalliselta aluksi johtuen kaksista suluista. Kohta

```
operator()
```

ilmoittaa kääntäjälle, että ylikuormitamme kutsuoperaattoria. Toinen sulkupari

```
( const string & s1, const string & s2 )
```

ilmaisee muodollisia parametreja, jotka välitetään kutsuoperaattorin ylikuormitetulle ilmentymälle. Jos vertaamme tätä määrittelyä aikaisempaan less_than()-määrittelyyn, huomaamme, että vaikka korvaamme less_than:in operator():illa, nämä kaksi ovat täsmälleen samanlaisia.

Funktio-olio määritellään samalla tavalla kuin tavallinen luokkaolio, vaikka tässä tapauksessa emme ole määritelleet muodostajaa (ei ole tietojäseniä, joita alustaa):

```
LessThan lt;
```

Jotta voisimme käynnistää ylikuormitetun kutsuoperaattorin jäsenen ilmentymän, käytämme yksinkertaisesti kutsuoperaattoria luokkaolioomme ja varustamme sen vaadituin argumentein. Esimerkiksi:

```
string st1( "shakespeare" );
string st2( "marlowe" );

// käynnistää: lt.operator()( st1, st2 );
bool is_shakespeare_less = lt( st1, st2 );
```

Tässä on uudelleentoteutuksemme process_vocab()-funktiosta, jossa välitämme nimeämättömän LessThan-funktio-olion stable_sort()-algoritmille:

```
void process_vocab( vector< textwords >*pvec )
{
    // ...
    stable_sort( texts.begin(), texts.end(), LessThan() );

    // ...
}
```

stable_sort()-algoritmissa ylikuormitettu operaattori laajennetaan välittömäksi. (stable_sort() pystyy hyväksymään kolmannen argumentin, joka on joko osoitin less_than()-funktioon tai LessThan-luokkaolio, koska argumentti on mallimekanismin tyyppiparametri. Katsomme funktio-olioita tarkemmin kohdassa 12.3.)

Tässä on stable_sort()-lajittelun tulos:

```
a i
as at by he in is it no
of on so to and but for has
her him its not now one red row
she sky the you asks bird blue coat
dark each emma fear grow hair held home
life like long mean more puts same says
```

```
star such tell that time what when wind
with your alice alive blows daddy falls fiery
lands leave looks quite rises shush shyly sight
still stone tells there thing trees watch almost
either flight houses night, ancient becomes bounded calling
distant flowing heaven, magical passion through unravel untamed
wanting darkened eternity beautiful darkening immensity journeying
alternately
immeasurable inexpressibly
```

Seuraava tehtävämme on laskea niiden sanojen lukumäärä, joiden pituus on suurempi kuin kuusi merkkiä. Voimme toteuttaa sen geneerisellä algoritmilla count_if() ja toisella funktio-oliolla GreaterThan. Se on hieman monimutkaisempi funktio, koska olemme yleistäneet sen niin, että se sallii käyttäjälle koon antamisen, jonka mukaan verrataan. Tallennamme koon luokan tietojäseneen ja alustamme sen luokan muodostajalla. Oletusarvoisesti se alustetaan arvolla 6:

```
#include <iostream>

class GreaterThan {
public:
    GreaterThan( int sz = 6 ) : _size( sz ){}
    int size() { return _size; }

    bool operator()( const string & s1 )
        { return s1.size() > _size;   }

private:
    int _size;
};
```

Seuraavassa on esimerkki, kuinka voisimme käyttää sitä:

```
void process_vocab( vector< textwords >*pvec )
{
    // ...
    // laske merkkijonojen lukumäärä, jotka ovat pitempiä kuin 6 merkkiä
    int cnt = count_if( texts.begin(), texts.end(),
                        GreaterThan() );

    cout << "Number of words greater than length six are "
         << cnt << endl;

    // ...
}
```

Tässä on ohjelmamme tulostusosuus:

```
Number of words greater than length six are 22
```

remove() käyttäytyy samoin kuin unique() siinä, että se ei varsinaisesti muuta säiliön kokoa, vaan erottaa elementit niihin, jotka jäävät jäljelle (kopioimalla ne vuorollaan säiliön eteen) ja

niihin, jotka poistetaan (jotka jäävät loppuun). Se palauttaa iteraattorin, joka osoittaa ensimmäiseen poistettavaan elementtiin. Seuraavassa näemme, kuinka voisimme käyttää sitä yleisten sanojen poistamiseen, joiden emme halua jäävän vektoriin:

```
void process_vocab( vector< textwords >*pvec )
{
    // ...

    static string rw[] = { "and", "if", "or", "but", "the" };
    vector< string > remove_words( rw, rw+5 );

    vector<string>::iterator it2 = remove_words.begin();
    for ( ; it2 != remove_words.end(); ++it2 ) {
        // vain näyttääksemme toisen muodon count():ista
        int cnt = count( texts.begin(), texts.end(), *it2 );
          cout << cnt << " instances removed:  "
              << (*it2) << endl;

          texts.erase(
                remove(texts.begin(),texts.end(),*it2 ),
                texts.end()
          );
      }

      // ...
}
```

Tässä on remove():n tulos:

```
1 instances removed:  and
0 instances removed:  if
0 instances removed:  or
1 instances removed:  but
1 instances removed:  the
```

Lopuksi haluaisimme näyttää vektorin sisällön. Eräs tapa tehdä se on käydä elementit läpi ja näyttää jokainen vuorollaan, mutta koska se tapa ei käytä hyväkseen geneerisiä algoritmeja, se ei ole sopiva ratkaisuksi tässä yhteydessä. Sen sijaan pidämme parempana näyttää for_each()-algoritmin käyttöä vektorin elementtien tulostamisessa. for_each() käyttää osoitinta funktioon tai funktio-oliota jokaiseen säiliön elementtiin, jota iteraattoripari ilmaisee. Tapauksessamme funktio-olio PrintElem tulostaa elementit vakiotulostukseen:

```
class PrintElem {
public:
    PrintElem( int lineLen = 8 )
        : _line_length( lineLen ), _cnt( 0 )
    {}

    void operator()( const string &elem )
    {
```

```
            ++_cnt;
            if ( _cnt % _line_length == 0 )
              { cout << '\n'; }

          cout << elem << " ";
      }

private:
      int _line_length;
      int _cnt;
};

void process_vocab( vector< textwords >*pvec )
{
       // ...

       for_each( texts.begin(), texts.end(), PrintElem() );
}
```

Siinä se on. Olemme saaneet ohjelmamme valmiiksi tekemättä tuskin enempää kuin kytkeneet yhteen muutaman geneerisen algoritmin käynnistyksen. Mukavuudeksenne olemme listanneet koko ohjelmatekstin yhdessä main()-funktion kanssa suoritusta varten (se sisältää erikoisia iteraattorityyppejä, jotka käsitellään kohdassa 12.4). Olemme listanneet varsinaisen suoritetun koodin, joka ei ole täysin C++-standardin mukainen. Erityisesti käytettävissä olevat toteutukset count()- ja count_if()-algoritmeista edustavat vanhempia versioita, jotka eivät palauta tulosta, vaan vaativat lisäargumentin, johon arvo sijoitetaan. Lisäksi iostream-kirjasto heijastelee esistandardin toteutusta ja vaatii iostream.h-otsikkotiedoston käyttöä.

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

// esistandardin syntaksi <iostream>:ille
#include <iostream.h>

class GreaterThan {
public:
      GreaterThan( int sz = 6 ) : _size( sz ){}
      int size() { return _size; }

      bool operator()( const string &s1 )
                { return s1.size() > _size; }
private:
      int _size;
};

class PrintElem {
public:
```

```
    PrintElem( int lineLen = 8 )
        : _line_length( lineLen ), _cnt( 0 )
    {}

    void operator()( const string &elem )
    {
        ++_cnt;
        if ( _cnt % _line_length == 0 )
          { cout << '\n'; }

        cout << elem << " ";
    }

private:
    int _line_length;
    int _cnt;
};

class LessThan {
public:
    bool operator()( const string & s1,
                const string & s2 )
    {   return s1.size() < s2.size();  }
};

typedef vector<string, allocator> textwords;
void process_vocab( vector<textwords, allocator>*pvec )
{
    if ( ! pvec ) {
        // anna varoitusilmoitus
        return;
    }

    vector< string, allocator > texts;

    vector<textwords, allocator>::iterator iter;
    for ( iter = pvec->begin(); iter != pvec->end(); ++iter )
        copy( (*iter).begin(), (*iter).end(),
              back_inserter( texts ));

    // lajittele tekstielementit
    sort( texts.begin(), texts.end() );

    // ok, katsotaanpa, mitä meillä on
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n"; // just to separate display output

    // poista kaikki duplikaattielementit
```

```
    vector<string, allocator>::iterator it;
    it = unique( texts.begin(), texts.end() );
    texts.erase( it, texts.end() );

    // ok, katsotaanpa, mitä meillä nyt on
    for_each( texts.begin(), texts.end(), PrintElem() );
    cout << "\n\n";

    // lajittele elementit perustuen oletuspituuteen 6
    // stable_sort() estää samanlaisten elementtien järjestelyn …
    stable_sort( texts.begin(), texts.end(), LessThan() );
    for_each( texts.begin(), texts.end(), PrintElem() );

    cout << "\n\n";

    // laske niiden merkkijonojen lukumäärä, jotka ovat pitempiä kuin 6
    int cnt = 0;

    // vanhentunut count-muoto -- standardi muuttaa tämän
    count_if( texts.begin(), texts.end(), GreaterThan(), cnt );

    cout << "Number of words greater than length six are "
         << cnt << endl;

    static string rw[] = { "and", "if", "or", "but", "the" };
    vector<string,allocator> remove_words( rw, rw+5 );

    vector<string, allocator>::iterator it2 =
          remove_words.begin();

    for ( ; it2 != remove_words.end(); ++it2 )
    {
        int cnt = 0;

        // vanhentunut count-muoto -- standardi muuttaa tämän
        count( texts.begin(), texts.end(), *it2, cnt );

        cout << cnt    << " instances removed:  "
             << (*it2) << endl;

        texts.erase(
            remove(texts.begin(),texts.end(),*it2),
            texts.end()
        );
    }

    cout << "\n\n";
    for_each( texts.begin(), texts.end(), PrintElem() );
}
```

```
// difference_type on sellainen, joka pystyy sisältämään tuloksen,
// jossa kaksi säiliön iteraattoria vähennetään toisistaan
// -- tässä tapauksessa merkkijonovektorista ...
// tavallisesti tämä käsitellään oletusarvoisesti ...

typedef vector<string,allocator>::difference_type diff_type;

// esistandardin otsikkosyntaksi <fstream>:ille
#include <fstream.h>

main()
{
    vector<textwords, allocator> sample;
    vector<string,allocator> t1, t2;
    string t1fn, t2fn;

    // pyydä syöttötiedostot käyttäjältä ...
    // pitäisi tehdä joitakin virhetarkistuksia todellisissa ohjelmissa
    cout << "text file #1: "; cin >> t1fn;
    cout << "text file #2: "; cin >> t2fn;

    // avaa tiedostot
    ifstream infile1( t1fn.c_str());
    ifstream infile2( t2fn.c_str());

    // erikoinen iteraattorimuoto
    // tavallisesti diff_type tulee oletusarvoisesti ...
    istream_iterator< string, diff_type > input_set1( infile1 ),
                                  eos;
    istream_iterator< string, diff_type > input_set2( infile2 );

    // erikoinen iteraattorimuoto
    copy( input_set1, eos, back_inserter( t1 ));
    copy( input_set2, eos, back_inserter( t2 ));

    sample.push_back( t1 ); sample.push_back( t2 );
    process_vocab( &sample );
}
```

Harjoitus 12.2

Sanan pituus ei ole ainoa eikä välttämättä paras tekstiosan monimutkaisuuden mittari. Toinen mahdollinen testi on lauseen pituus. Kirjoita ohjelma, joka lukee sisään joko tekstitiedoston tai tekstiä vakiosyötöstä, rakentaa merkkijonovektorin jokaisesta lauseesta ja välittää jokaisen vektorin count()-algoritmille. Näytä lauseet monimutkaisuusjärjestyksessä. Mielenkiintoinen tapa tehdä tämä on tallentaa jokainen lause yhtenä suurena merkkijonona toiseen merkkijonovektoriin ja sitten välittää vektori sort()-algoritmille funktio-oliona, joka tekee pienempi kuin -ver-

tailun pienemmän merkkijonon perusteella. (Jos haluat lukea tarkempia kuvauksia tietystä geneerisestä algoritmista tai nähdä lisäesimerkin sen käytöstä, katso tämän kirjan liitteestä, jossa algoritmit on lueteltu aakkosjärjestyksessä.)

Harjoitus 12.3

Luotettavampi testi tekstiosan vaikeusasteesta on lauseiden rakenteellinen monimutkaisuus. Anna jokaiselle pilkulle 1 piste, puolipisteelle tai kaksoispisteelle 2 pistettä ja jokaiselle ajatusviivalle 3 pistettä. Muokkaa harjoituksen 12.2 ohjelmaa niin, että se laskee jokaisen lauseen monimutkaisuuden. Käytä count_if()-algoritmia, jotta voit päätellä välimerkit vektorin lauseesta. Näytä lauseet monimutkaisuusjärjestyksessä.

12.3 Funktio-oliot

min()-funktiomme on hyvä esimerkki mallimekanismin sekä tehosta että rajoituksesta.

```
template <typename Type>
const Type&
min( const Type *p, int size )
{
    Type minval = p[ 0 ];
    for ( int ix = 1; ix < size; ++ix )
        if ( p[ ix ] < minval )
            minval = p[ ix ];
    return minval;
}
```

Teho tulee kyvystä määritellä yksittäinen min()-ilmentymä, joka voidaan instantioida lukemattomista eri tyypeistä. Rajoitus on, että vaikka min() voidaan instantioida lukemattomista tyypeistä, se ei ole sovelias kaikille tyypeille.

Rajoitus keskittyy pienempi kuin -operaattorin käyttöön. Jossakin tapauksessa taustalla oleva tyyppi ei ehkä tue pienempi kuin -operaattoria. Esimerkiksi Image-luokassa ei ole pienempi kuin -operaattorin toteutusta, vaikka voisimme olla siitä tietämättömiä ja haluta löytää pienimmän kehysnumeron Image-olioiden taulukosta. Yritys instantioida min() Image-luokan taulukolla johtaa kuitenkin käännöksenaikaiseen virheeseen:

```
error: invalid types applied to the < operator: Image < Image
```

Toisessa tapauksessa pienempi kuin -operaattori on olemassa, mutta sen merkitys ei sovi tähän. Jos esimerkiksi haluamme löytää pienimmän merkkijonon, mutta haluamme ottaa huomioon vain kirjaimet emmekä halua ottaa huomioon isoja ja pieniä kirjaimia, silloin pienempi kuin -operaattori tukee vääriä asioita.

Perinteinen ratkaisu on parametroida vertailuoperaattori, joka tässä tapauksessa tarkoittaa osoittimen esittelyä funktioon, joka saa kaksi argumenttia ja palauttaa bool-tyyppisen arvon:

```
template < typename Type,
           bool (*Comp)(const Type&, const Type&)>
const Type&
```

```
min( const Type *p, int size, Comp comp )
{
      Type minval = p[ 0 ];
      for ( int ix = 1; ix < size; ++ix )
            if ( Comp( p[ ix ], minval ))
                minval = p[ ix ];
      return minval;
}
```

Tämä ratkaisu yhdessä ensimmäisen toteutuksemme kanssa, jossa käytettiin sisäistä pienempi kuin -operaattoria, tarjoaa yleistuen mille tahansa tyypille, mukaan lukien Image-luokkamme, jotka meidän tulisi toteuttaa kahden kuvan (Image) ehdoilla. Pääasiallisin suorituskykyyn liittyvä haitta osoittimesta funktioon on, että sen epäsuora käynnistys estää sen tekemisen välittömäksi.

Vaihtoehtoinen parametrointistrategia on, että osoitin funktioon onkin funktio-olio (näimme useita esimerkkejä edellisessä kohdassa). Funktio-olio on luokka, joka ylikuormittaa funktion kutsuoperaattoria (operator()). Operaattori kapseloi sen, mikä normaalisti toteutettaisiin funktiona. Funktio-olio välitetään tyypillisesti argumenttina geneeriselle algoritmille, vaikka voimme määritellä myös yksittäisiä funktio-olioita. Jos esimerkiksi AddImages-luokka olisi määritelty funktio-olioksi, joka saa kaksi kuvaa, yhdistää ne (lisää ne yhteen) ja palauttaa uuden kuvan, voisimme esitellä sen ilmentymän seuraavasti:

```
AddImages AI;
```

Jotta saisimme funktio-olion tekemään tämän operaation, käytämme kutsuoperaattoria antamalla tarvittavat Image-luokan operandit. Esimerkiksi:

```
Image im1("foreground.tiff"), im2("background.tiff");
// ...

// käynnistää: Image AddImages::operator()(const Image&,const Image&);
Image new_image = AI( im1, im2 );
```

Funktio-olion etu verrattuna funktio-osoittimeen on kaksinkertainen. Ensiksi, jos ylikuormitettu kutsuoperaattori on välitön funktio, kääntäjä pystyy tekemään sen välittömäksi ja saa aikaan merkittävän tehokkuuden kasvun. Toiseksi funktio-olio voi sisältää minkä tahansa määrän lisätietoa, joko välivarastoituja tuloksia tai tietoa nykyisen operaation auttamiseksi.

Tässä on uudistettu toteutuksemme min()-algoritmista (huomaa, että osoitin funktioon voidaan välittää myös käyttäen tätä esittelyä, mutta tarkistamatta sen prototyyppiä):

```
template < typename Type,
           typename Comp >
const Type&
min( const Type *p, int size, Comp comp )
{
      Type minval = p[ 0 ];
      for ( int ix = 1; ix < size; ix++ )
            if ( Comp( p[ ix ], minval ))
                minval = p[ ix ];
      return minval;
}
```

Geneeriset algoritmit tarjoavat yleensä tukea molemmille operaation käyttömuodoille: sisäisen (tai mahdollisesti ylikuormitetun) operaattorin käyttöä ja osoitinta funktioon tai funktio-olion käyttöä operaation suorittamiseksi.

Mistä funktio-oliot tulevat? Yleensä on olemassa kolme lähdettä funktio-olioille.

- Vakiokirjastossa on esimääritelty joukko aritmeettisia, vertailu- ja loogisia funktio-olioita.
- Esimääritellyillä funktiosovittimilla voimme erikoistaa eli laajentaa esimääriteltyjä funktio-olioita (tai tarvittaessa mitä tahansa funktio-oliota).
- Voimme määritellä omia funktio-olioita, joita voidaan välittää geneerisille algoritmeille, ja ehkä käyttää niihin funktiosovittimia.

Tässä kohdassa katsomme jokaista näitä funktio-olion lähdettä vuorollaan.

12.3.1 Esimääritellyt funktio-oliot

Esimääritellyt funktio-oliot on jaettu aritmeettisiin, vertailu- ja loogisiin operaatioihin. Jokainen olio on luokkamalli, joka on parametroitu operandien tyypeillä. Jotta niitä voidaan käyttää, pitää ottaa mukaan seuraava otsikkotiedosto:

```
#include <functional>
```

Esimerkiksi funktio-olio, joka tukee yhteenlaskua, on luokkamalli nimeltään plus. Jos määrittelisimme ilmentymän, joka lisäisi yhteen kaksi kokonaislukua, kirjoittaisimme

```
#include <functional>
plus< int > intAdd;
```

Kun haluamme käynnistää yhteenlaskuoperaation, käytämme ylikuormitettua kutsuoperaattoria intAdd aivan kuten teimme edellisessä kohdassa AddImage-luokallemme:

```
int ival1 = 10, ival2 = 20;

// sama kuin int sum = ival1 + ival2;
int sum = intAdd( ival1, ival2 );
```

Plus-luokkamallin toteutus käynnistää lisäysoperaattorin, joka liittyy sen int-parametrin tyyppiin. Tätä ja muita esimääriteltyjen luokkien funktio-olioita käytetään pääasiallisesti argumentteina geneerisille argumenteille, usein oletusoperaation korvaamiseksi. Esimerkiksi sort() lajittelee säiliön elementit oletusarvoisesti nousevaan järjestykseen käyttäen taustalla olevan elementin tyypin pienempi kuin -operaattoria. Kun haluamme lajitella säiliön laskevaan järjestykseen, välitämme esimääritellyn luokkamallin greater, joka käynnistää taustalla olevan elementtityypin suurempi kuin -operaattorin:

```
vector< string > svec;
// ...
sort( svec.begin(), svec.end(), greater<string>() );
```

Esimääritellyt funktio-oliot on lueteltu seuraavissa alikohdissa ja ne on jaettu aritmeettiseen, vertailu- ja loogiseen kategoriaan. Jokainen on kuvattu sekä nimettynä että nimeämättömänä funktiolle välitettynä oliona. Käytämme seuraavia oliomäärittelyjä ja yksinkertaista luokan määrittelyä (operaattorin ylikuormitusta käsitellään tarkemmin luvussa 15):

```
class Int {
public:
    Int( int ival = 0 ) : _val( ival ){}

    int operator-()        { return -_val;      }
    int operator%(int ival) { return _val % ival; }

    bool operator<(int ival){ return _val < ival; }
    bool operator!()       { return _val == 0;   }
private:
    int _val;
};

vector< string > svec;
string  sval1, sval2, sres;
complex cval1, cval2, cres;
int     ival1, ival2, ires;
Int     Ival1, Ival2, Ires;
double  dval1, dval2, dres;
```

Lisäksi määrittelemme seuraavat kaksi funktiomallia, joille välitämme monia nimeämättömiä funktio-olioita:

```
template <class FuncObject, class Type>
   Type UnaryFunc( FuncObject fob, const Type &val )
      { return fob( val ); }

template <class FuncObject, class Type>
```

```
Type BinaryFunc( FuncObject fob,
        const Type &val1, const Type &val2 )
    { return fob( val1, val2 ); }
```

12.3.2 Aritmeettiset funktio-oliot

Esimääritellyt aritmeettiset funktio-oliot tukevat yhteenlaskua, vähennyslaskua, kertolaskua, jakolaskua, jakojäännöstä ja negaatiota. Käynnistetty operaattori on ilmentymä, joka liittyy Type-tyyppiin. Luokkatyypille, jossa operaattorin ylikuormitettuja ilmentymä on, käynnistetään tuo ilmentymä.

- Yhteenlasku: plus<Type>

```
plus<string> stringAdd;

// käynnistää: string::operator+()
sres = stringAdd( sval1, sval2 );
dres = BinaryFunc( plus<double>(), dval1, dval2 );
```

- Vähennyslasku: minus<Type>

```
minus<int> intSub;
ires = intSub( ival1, ival2 );
dres = BinaryFunc( minus<double>(), dval1, dval2 );
```

- Kertolasku: multiplies<Type>

```
multiplies<complex> complexMultiplies;
cres = complexMultiplies( cval1, cval2 );
dres = BinaryFunc( multiplies<double>(), dval1, dval2 );
```

- Jakolasku: divides<Type>

```
divides<int> intDivides;
ires = intDivides( ival1, ival2 );
dres = BinaryFunc( divides<double>(), dval1, dval2 );
```

- Jakojäännös: modulus<Type>

```
modulus<Int> IntModulus;
Ires = IntModulus( Ival1, Ival2 );
ires = BinaryFunc( modulus<int>(), ival2, ival1 );
```

- Negaatio: negate<Type>

```
negate<int> intNegate;
ires = intNegate( ires );
Ires = UnaryFunc( negate<Int>(), Ival1 );
```

12.3.3 Vertailufunktio-oliot

Esimääritellyt vertailufunktio-oliot tukevat vertailuja yhtäsuuruus, erisuuruus, suurempi kuin, suurempi tai yhtäsuuri, pienempi kuin ja pienempi tai yhtäsuuri.

- Yhtäsuuruus: equal_to<Type>

```
equal_to<string> stringEqual;
sres = stringEqual( sval1, sval2 );
ires = count_if( svec.begin(), svec.end(),
                        equal_to<string>(), sval1 );
```

- Erisuuruus: not_equal_to<Type>

```
not_equal_to<complex> complexNotEqual;
cres = complexNotEqual( cval1, cval2 );
ires = count_if( svec.begin(), svec.end(),
                        not_equal_to<string>(), sval1 );
```

- Suurempi kuin: greater<Type>

```
greater<int> intGreater;
ires = intGreater( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
            greater<string>(), sval1 );
```

- Suurempi tai yhtäsuuri: greater_equal<Type>

```
greater_equal<double> doubleGreaterEqual;
dres = doubleGreaterEqual( dval1, dval2 );
ires = count_if( svec.begin(), svec.end(),
                        greater_equal<string>(), sval1 );
```

- Pienempi kuin: less<Type>

```
less<Int> IntLess;
Ires = IntLess( Ival1, Ival2 );
ires = count_if( svec.begin(), svec.end(),
                        less<string>(), sval1 );
```

- Pienempi tai yhtäsuuri: less_equal<Type>

```
less_equal<int> intLessEqual;
ires = intLessEqual( ival1, ival2 );
ires = count_if( svec.begin(), svec.end(),
                        less_equal<string>(), sval1 );
```

12.3.4 Loogiset funktio-oliot

Esimääritellyt loogiset funktio-oliot tukevat seuraavia: looginen JA (palauttaa arvon tosi, jos sen molemmat operandit ovat tosia — käyttää &&-operaattoria Type:n kanssa), looginen TAI (palauttaa arvon tosi, jos jompikumpi sen parametreista on tosi — käyttää ||-operaattoria Type:n kanssa) ja looginen EI (palauttaa arvon tosi, jos sen arvo on epätosi — käyttää !-operaattoria Type:n kanssa).

- Looginen Ja: logical_and<Type>

```
logical_and<int> intAnd;
ires = intAnd( ival1, ival2 );
```

```
dres = BinaryFunc( logical_and<double>(), dval1, dval2 );
```

- Looginen Tai: logical_or<Type>

```
logical_or<int> intSub;
ires = intSub( ival1, ival2 );
dres = BinaryFunc( logical_or<double>(), dval1, dval2 );
```

- Looginen Ei: logical_not<Type>

```
logical_not<Int> IntNot;
Ires = IntNot( Ival1, Ival2 );
dres = UnaryFunc( logical_not<double>(), dval1 );
```

12.3.5 Funktio-olioiden funktiosovittimet

Vakiokirjastossa on myös joukko funktiosovittimia, joilla erikoistetaan ja laajennetaan sekä unaarisia että binäärisiä funktio-olioita. Sovittimet ovat erikoisluokkia ja ne on jaettu seuraaviin kahteen kategoriaan:

1. Sitojat: sitoja on funktiosovitin, joka konvertoi binäärifunktio-olion unaariolioksi sitomalla yhden argumenteista tietyksi arvoksi. Jos esimerkiksi haluamme laskea säiliön kaikki ne elementit, jotka ovat pienempiä tai yhtäsuuria kuin 10, välitämme count_if()-funktiolle pienempi tai yhtäsuuri -funktio-olion, jonka yksi argumentti on sidottu arvoon 10. Katsomme seuraavassa kohdassa, kuinka teemme sen.
2. Negaattorit: negaattori on funktiosovitin, joka asettaa funktio-olion totuusarvon päinvastaiseksi. Jos esimerkiksi haluamme laskea säiliön kaikki ne elementit, jotka ovat suurempia kuin 10, välitämme count_if()-funktiolle negaattorin pienempi tai yhtäsuuri -funktio-oliostamme, jonka yksi arvo on sidottu arvoon 10. Tietysti tässä tapauksessa on huomattavasti suoraviivaisempaa välittää yksinkertaisesti greater-funktio-olion sitoja, jonka yksi argumentti on sidottu arvoon 10.

Vakiokirjastossa on kaksi esimääriteltyä sitojasovitinta: bind1st ja bind2nd. Kuten saatat arvata, bind1st sitoo arvon binäärifunktio-olion ensimmäiseen argumenttiin ja bind2nd toiseen. Jos haluamme laskea esimerkiksi säiliön kaikki elementit, jotka ovat pienempiä tai yhtäsuuria kuin 10, välitämme count_if()-funktiolle seuraavaa:

```
count_if( vec.begin(), vec.end(),
        bind2nd( less_equal<int>(), 10 ));
```

Vakiokirjastossa on kaksi esimääriteltyä negaattorisovitinta: not1 ja not2. Jälleen kuten saatat arvata, not1 kääntää päinvastaiseksi unaarisen funktio-olion totuusarvon ja not2 binäärisen funktio-olion totuusarvon. Jos haluaisimme kääntää päinvastaiseksi less_equal-funktio-oliomme sitomisen, kirjoittaisimme seuraavasti:

```
count_if( vec.begin(), vec.end(),
          not1( bind2nd( less_equal<int>(), 10 )));
```

Näemme lisää käyttöesimerkkejä sekä sitojista että negaattoreista kirjan liitteessä.

12.3.6 Funktio-olion toteutus

Olemme jo määritelleet useita funktio-olioita, jotka tukevat kohdan 12.2. ohjelmatoteutusta. Tässä kohtaa käymme läpi vaiheet, joilla määritellään funktio-olioita. (Luku 13 kattaa yksityiskohtaisesti luokan yleisen määrittelyn; luku 15 käsittelee operaattorin ylikuormitusta.)

Yksinkertaisin muoto funktio-olion luokan määrittelystä koostuu ylikuormitetusta funktion kutsuoperaattorista. Tässä on esimerkiksi unaarinen funktio-olio, joka päättelee, onko arvo pienempi tai yhtäsuuri kuin 10:

```
// funktio-olion luokan yksinkertaisin muoto
class less_equal_ten {
public:
    bool operator() ( int val )
        { return val <= 10;  }
};
```

Voimme nyt käyttää tätä oliota samalla tavalla kuin käytämme esimääriteltyjä funktio-olioita. Tässä on esimerkki uudistetusta count_if()-algoritmin käynnistyksestä, jossa käytetään funktio-oliomme:

```
count_if( vec.begin(), vec.end(), less_equal_ten() );
```

Tämä luokka on melko rajoitettu. Käytämme negaattoria laskeaksemme säiliön kaikki elementit, jotka ovat suurempia kuin 10:

```
count_if( vec.begin(), vec.end(),
          not1(less_equal_ten()));
```

Vaihtoehtoisesti voimme laajentaa toteutustamme sallimalla, että käyttäjä voi antaa arvon, jota verrataan säiliön jokaiseen elementtiin. Eräs tapa tämän toteuttamiseksi on esitellä tietojäsen, johon verrattava arvo tallennetaan, sekä muodostaja, joka alustaa jäsenen käyttäjän määrittämällä arvolla:

```
class less_equal_value {
public:
    less_equal_value( int val ) : _val( val ){}
    bool operator() ( int val ) { return val <= _val; }

private:
    int _val;
};
```

Voimme nyt käyttää tätä oliota ja määrittää minkä tahansa kokonaislukuarvon. Esimerkiksi seuraava käynnistys laskee niiden elementtien lukumäärän, jotka ovat pienempiä tai yhtäsuuria kuin 25.

```
count_if( vec.begin(), vec.end(), less_equal_value( 25 ));
```

Vaihtoehtoinen luokan toteutus ilman muodostajaa olisi sen parametrointi verrattavalla arvolla. Esimerkiksi:

```
template < int _val >
class less_equal_value {
public:
    bool operator() ( int val ){ return val <= _val; }
};
```

Tässä on esimerkki, kuinka käynnistäisimme tämän luokan laskemaan niiden elementtien lukumäärän, jotka ovat pienempiä tai yhtäsuuria kuin 25.

```
count_if( vec.begin(), vec.end(), less_equal_value<25>());
```

Näemme lisää esimerkkejä omien funktio-olioiden määrittelyistä kirjan liitteessä, jossa geneeristen algoritmien käyttöesimerkkejä esitellään.

Harjoitus 12.4

Käyttäen esimääriteltyjä funktio-olioita ja funktiosovittimia luo funktio-olio, joka tekee seuraavaa:

(a) Etsii kaikki arvot, jotka ovat suurempia kuin 1024.
(b) Etsii kaikki merkkijonot, jotka eivät ole yhtä kuin "pooh".
(c) Kertoo kaikki arvot 2:lla.

Harjoitus 12.5

Määrittele funktio-olio, joka vertailee kolmea oliota ja palauttaa keskimmäisen arvon. Määrittele funktio, joka tekee saman operaation. Näytä esimerkkejä, joissa käytetään jokaista oliota suoraan ja välittämällä jokaista funktioon. Vertaile ja etsi vastakohtia kummankin käyttäytymisestä.

12.4 Paluu iteraattoreihin

Seuraava funktiomallin toteutus ei käänny. Huomaatko, miksi?

```
// tämä ei käänny näin toteutettuna
template < typename type >
int
count( const vector< type > &vec, type value )
{
    int count = 0;

    vector< type >::iterator iter = vec.begin();
    while ( iter != vec.end() )
        if ( *iter == value )
            ++count;

    return count;
}
```

Ongelma on, että vec on const-viittaus, mutta yritämme sitoa ei-const-iteraattorin siihen. Jos

tämä olisi sallittua, ei mikään estäisi meitä muokkaamasta vektorin elementtejä iteraattorin kautta. Jotta kieli voisi sen estää, se vaatii, että iteraattorin, joka on sidottu const-vektoriin, pitää olla const-iteraattori. Teemme sen seuraavasti:

```
// ok: tämä kääntyy virheittä
vector< type >::const_iterator iter = vec.begin();
```

Tämä vaatimus, että const-säiliö pitää sitoa vain const-iteraattoriin, noudattaa samaa käytäntöä, kun sallitaan const-osoittimen osoittaa vain const-taulukkoa. Molemmissa tapauksissa kieli etsii takuita, että const-säiliön sisältö ei muutu.

Sekä begin()- että end()-operaatiot on ylikuormitettu ja ne palauttavat joko const- tai ei-const-iteraattorin riippuen säiliön const-piirteistä. Esimerkiksi seuraavassa esittelyparissa

```
vector< int > vec0;
const vector< int > vec1;
```

begin()- ja end()-käynnistykset vektorilla vec0 palauttavat ei-const-iteraattorin, kun taas samat käynnistykset vektorilla vec1 palauttavat const-iteraattorin:

```
vector< int >::iterator iter0 = vec0.begin();
vector< int >::const_iterator iter1 = vec1.begin();
```

On tietysti aina sallittua sijoittaa ei-const-iteraattori const-iteraattoriin. Esimerkiksi:

```
// ok: ei-const-iteraattorin alustaminen const:illa
vector< int >::const_iterator iter2 = vec0.begin();
```

12.4.1 Lisäysiteraattorit

Tässä on toinen koodikatkelma, jossa on vakava ja vaikeasti havaittava ongelma. Huomaatko sen?

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };
vector< int > ivec( ia, ia+8 ), vres;
// ...

// johtaa tuntemattomaan suorituksenaikaiseen käyttäytymiseen...
unique_copy( ivec.begin(), ivec.end(), vres.begin() );
```

Ongelma tässä on, että vres:ille ei ole varattu muistia, jotta siihen mahtuisi yhdeksän kokonaislukuarvoa, jotka kopioidaan siihen ivec-vektorista. unique_copy()-algoritmi käyttää sijoitusta jokaisen elementin arvon kopioimiseen, mutta sijoitus epäonnistuu, koska vres:issä ei ole tilaa.

Eräs strategia olisi tehdä kaksi versiota unique_copy()-algoritmista: yksi, joka sijoittaa elementit ja toinen, joka lisää ne. Lisäysilmentymä voisi puolestaan tukea tarpeen tullen vaihtoehtoisia ilmentymiä, jotka lisäävät elementit säiliön alkuun, loppuun tai johonkin muuhun kohtaan.

Vaihtoehtoinen strategia, joka on lainattu vakiokirjastosta, on määritellä kolme lisäysiteraattorin sovitinfunktiota, jotka palauttavat erikoislisäysiteraattorit:

- back_inserter(), joka saa aikaan säiliön push_back()-lisäysoperaation käynnistyksen sijoitusoperaattorin sijasta. Argumentti back _inserter():ille on itse säiliö. Voimme esimerkiksi korjata unique_copy()-käynnistyksemme kirjoittamalla

```
// ok: nyt unique_copy() tekee lisäyksen käyttäen operaatiota vres.push_back() ...
unique_copy( ivec.begin(), ivec.end(),
        back_inserter( vres );
```

- front_inserter(), joka saa aikaan säiliön push_front()-lisäysoperaation käynnistyksen sijoitusoperaattorin sijasta. Argumentti front_inserter():ille on myös itse säiliö. Huomaa kuitenkin, että vektoriluokka ei tue push_front()-operaatiota ja sen käyttäminen vektorille on virhe:

```
// hups, virhe:
// vektori ei tue push_front()-operaatiota
// käytä joko pakka- tai listasäiliötä
unique_copy( ivec.begin(), ivec.end(),
        front_inserter( vres );
```

- inserter(), joka saa aikaan säiliön insert()-lisäysoperaation käynnistyksen sijoitusoperaattorin sijasta. inserter() saa kaksi argumenttia: itse säiliön ja iteraattorin säiliöön, joka ilmaisee paikan, josta lisäyksen tulisi alkaa. Esimerkiksi:

```
unique_copy( ivec.begin(), ivec.end(),
        inserter( vres, vres.begin() );
```

  Iteraattori, joka ilmaisee lisäyskohdan alkua, ei jää pysyväksi, vaan sitä kasvatetaan jokaisen lisätyn elementin jälkeen niin, että jokainen elementti lisätään vuorollaan. On kuin olisimme kirjoittaneet

```
vector< int >::iterator iter = vres.begin(),
          iter2 = ivec.begin();

for ( ; iter2 != ivec.end(); ++iter, ++iter2 )
      vres.insert( iter, *iter2 );
```

12.4.2 Käänteisiteraattorit

begin()- ja end()-operaatiot palauttavat vastaavasti iteraattorin säiliön ensimmäiseen elementtiin ja yhden yli säiliön viimeisen elementin. On myös mahdollista palauttaa *käänteisiteraattori* (*reverse iterator*): sellainen, joka käy säiliön läpi viimeisestä elementistä ensimmäiseen. Operaatiot, jotka tukevat sellaista mahdollisuutta kaikilla tyypeillä, ovat rbegin() ja rend(). Kuten tavallisilla iteraattoreilla, myös näillä on sekä const- että ei-const-ilmentymät.

```
vector< int > vec0;
const vector< int > vec1;

vector< int >::reverse_iterator r_iter0 = vec0.rbegin();
vector< int >::const_reverse_iterator r_iter1 = vec1.rbegin();
```

Käänteisiteraattori käydään läpi samalla tavalla kuin eteenpäin käytävä iteraattori. Erona on seuraavan (ja edellisen) operaattorin toteutus. Eteenpäin käytävän iteraattorin yhteydessä ++ käsittelee säiliön seuraavaa elementtiä; käänteisiteraattorin yhteydessä se käsittelee edellistä elementtiä. Jos esimerkiksi kävisimme vektorin läpi taaksepäin, kirjoittaisimme

```
// vektorin käänteisiteraattori lopusta alkuun
vector< type >::reverse_iterator r_iter;
for ( r_iter = vec0.rbegin(); // sitoo r_iterIn viimeiseen elementtiin
        r_iter != vec0.rend();  // ei yhtä kuin 1 yli viimeisen elementin
        r_iter++ )              // vähentää! iteraattoria yhdellä elementillä
{ /* ... */ }
```

Vaikkakin saattaa näyttää hämmentävältä, kun lisäys- ja vähennysoperaattorien merkitykset ovat käänteisiä, se antaa ohjelmoijalle mahdollisuuden välittää läpinäkyvästi käänteisiteraattoriparin algoritmille. Jos esimerkiksi lajittelemme vektorimme laskevaan järjestykseen, välitämme yksinkertaisesti sort()-algoritmille käänteisiteraattoriparin kuten seuraavassa:

```
// lajittelee vektorin nousevaan järjestykseen
sort( vec0.begin(), vec0.end() );

// lajittelee vektorin laskevaan järjestykseen
sort( vec0.rbegin(), vec0.rend() );
```

12.4.3 Iostream-iteraattorit

Vakiokirjasto tukee sekä syötön että tulostuksen iostream-iteraattoreita, jotka toimivat yhdessä vakiokirjaston säiliötyyppien ja geneeristen algoritmien kanssa. Istream_iterator-luokka tukee istream-luokan tai jonkin sen johdetun luokan iteraattorioperaatioita kuten ifstream-syöttövirtaa. Samalla tavalla ostream_iterator tukee ostream-luokan tai jonkin sen johdetun luokan iteraattorioperaatioita kuten ofstream-tulostusvirtaa. Kummankin iteraattorin käyttöön tarvitaan mukaan iterator-otsikkotiedosto:

```
#include <iterator>
```

Esimerkiksi seuraavassa ohjelmassa käytämme istream-iteraattoria, joka lukee vakiosyötöstä kokonaislukukokoelman vektoriin ja käyttää sitten ostream-iteraatoria geneerisen unique_copy()-algoritmin kohteena.

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <functional>

/*
 * syöttö:
 * 23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
 *
```

```
 * tulostus:
 * 109 90 89 56 45 34 23 12 8 6
 */

int main()
{
      istream_iterator< int > input( cin );
      istream_iterator< int > end_of_stream;

      vector<int> vec;
      copy ( input, end_of_stream, inserter( vec, vec.begin() ));

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

      ostream_iterator< int > output( cout, " " );
      unique_copy( vec.begin(), vec.end(), output );
}
```

12.4.4 Istream_iterator

Istream_iterator on esitelty käyttäen seuraavaa yleistä muotoa[1]:

```
istream_iterator<Type> tunnus( istream& );
```

jossa Type edustaa mitä tahansa sisäistä tai käyttäjän määrittelemää luokkatyyppiä, jolle syöttöoperaattori on määritelty. Muodostajan argumentti voi olla joko istream-luokan olio kuten cin tai mikä tahansa julkisesti johdettu istream-alityyppi kuten ifstream. Esimerkiksi:

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// lue kompleksiolioita vakiosyötöstä
istream_iterator< complex > is_complex( cin );

// lue merkkijonoja nimetystä tiedostosta
ifstream infile( "C++Primer" );
istream_iterator< string > is_string( infile );
```

1. Jos kääntäjäsi ei vielä tue malliparametrien oletusarvoja, pitää pakottaa toinen argumentti istream_iterator-muodostajalle: säiliön difference_type, johon elementit sijoitetaan. Tyyppi difference_type voi sisältää säiliön kahden iteraattorin vähennyslaskun tuloksen. Esimerkiksi kohdassa 12.2, jossa esittelimme ohjelmaa, jonka kääntäjä ei vielä tue malliparametrien oletusarvoja, kirjoitimme

```
typedef vector<string,allocator>::difference_type diff_type;
istream_iterator< string, diff_type > input_set1( infile1 ), eos;
istream_iterator< string, diff_type > input_set2( infile2 );
```

Täysin C++-standardille yhteensopivalle kääntäjälle kirjoittaisimme vain yksinkertaisesti seuraavan:

```
istream_iterator< string > input_set1( infile1 ), eos;
istream_iterator< string > input_set2( infile2 );
```

Jokainen istream_iterator-olion lisäysoperaattori lukee seuraavan elementin syöttövirrasta käyttämällä operaattoria operator>>(). Jotta voisimme lukea syöttövirran kokonaan istream_iterator-iteraattorilla geneerisen algoritmin kautta, pitää olla iteraattoripari, joka ilmaisee tiedoston alkua ja loppua. Istream_iterator, joka on alustettu istream-oliolla kuten is_string, antaa alkuposition. Jotta voisimme määritellä loppuposition, käytämme erityistä istream_iterator-oletusmuodostajaa:

```
// muodostaa end-of-stream-iteraattorin, joka toimii
// iteraattoriparin loppupään ilmaisijana ...
istream_iterator< string > end_of_stream;

vector<string> text;

// ok: anna iteraattoripari
copy( is_string, end_of_stream,
    inserter( text, text.begin() ));
```

12.4.5 Ostream_iterator

Ostream_iterator esitellään jommallakummalla yleisistä muodoista:

```
ostream_iterator<Type> tunnus( ostream& )
ostream_iterator<Type> tunnus( ostream&, char* erotin )
```

jossa Type edustaa mitä tahansa sisäistä tai käyttäjän määrittelemää luokkatyyppiä, jolle syöttöoperaattori (operator>>) on määritelty. erotin edustaa C-tyylistä merkkijonoa, joka tulostetaan tiedostoon jokaisen elementin perään. Koska se on C-tyylinen merkkijono, sen pitää päättyä null-merkkiin; muussa tapauksessa sen käyttäytyminen on tuntematon (aiheuttaa todennäköisesti ohjelman kaatumisen suorituksen aikana). Ostream-argumentti voi olla joko todellinen ostream-luokkaolio kuten cout tai julkisesti johdetun ostream-luokan alityyppi kuten ofstream. Esimerkiksi:

```
#include <iterator>
#include <fstream>
#include <string>
#include <complex>

// kirjoita kompleksiolioita vakiosyöttöön
// ja erottele jokainen elementti tyhjällä merkillä
ostream_iterator< complex > os_complex( cout, " " );

// kirjoita merkkijonoja nimettyyn tiedostoon
// ja sijoita jokainen omalle rivilleen
ofstream outfile( "dictionary" );
ostream_iterator< string > os_string( outfile, "\n" );
```

Tässä on yksinkertainen esimerkki, kuinka vakiosyöttöä luetaan ja kaiutetaan vakiosyöttöön käyttämällä nimeämättömiä stream-iteraattoriolioita ja geneeristä copy()-algoritmia.

```
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
      copy( istream_iterator< int >( cin ),
        istream_iterator< int >(),
        ostream_iterator< int >( cout ));
}
```

Tässä on lopuksi pieni ohjelma, joka avaa käyttäjän määrittämän tiedoston ja kaiuttaa sen vakiotulostukseen käyttämällä jälleen copy()-algoritmia ja ostream_iterator-iteraattoria:

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


int main()
{
   string file_name;
   cout << "please enter a file to open: ";
   cin >> file_name;

   if ( file_name.empty() || !cin ) {
       cerr << "unable to read file name\n"; return -1;
   }

   ifstream infile( file_name.c_str());
      if ( ! infile ) {
       cerr << "unable to open " << file_name << endl;
       return -2;
   }

   istream_iterator< string > ins( infile ), eos;
   ostream_iterator< string > outs( cout, " " );

   copy( ins, eos, outs );
}
```

12.4.6 Iteraattoreiden viisi kategoriaa

Jotta vakiokirjasto voisi tukea kaikkia geneerisiä algoritmeja, se on määritellyt viisi kategoriaa iteraattoreille, jotka perustuvat niiden tekemille operaatioille: InputIterator, OutputIterator, ForwardIterator, BidirectionalIterator ja RandomAccessIterator. Seuraavassa on lyhyt kuvaus niiden piirteistä.

1. Syöttöiteraattoria (InputIterator) voidaan käyttää säiliön elementtien lukemiseen, mutta säiliöön kirjoittamista ei taata. Syöttöiteraattorilla pitää olla seuraava minimituki (iteraattoreita, joilla on lisätukea, voidaan myös käyttää syöttöoperaattoreina edellyttäen, että ne vastaavat näitä vähimmäisvaatimuksia): kahden iteraattorin sekä yhtäsuuruuden että erisuuruuden testaus, iteraattorin eteenpäin kasvatus osoittamaan seuraavaan elementtiin käyttämällä operaattorin (++) etu- ja loppuliiteilmentymiä ja elementin lukeminen käänteisoperaattorin (*) kautta. Geneerisiin algoritmeihin, jotka vaativat tämän tukitason, kuuluvat find(), accumulate() ja equal(). Mille tahansa algoritmille, joka vaatii syöttöiteraattorin tuen, voidaan välittää mikä tahansa kohdissa 3, 4 ja 5 lueteltu iteraattorikategoria.
2. Tulostusiteraattorin (OutputIterator) voidaan ajatella olevan toiminnaltaan syöttöiteraattorin vastakohta: sitä voidaan käyttää säiliön elementtien kirjoittamiseen, mutta ei taata säiliön lukemista. Syöttöiteraattoreita käytetään yleensä algoritmien kolmantena argumenttina ja ne ilmaisevat kohtaa, josta kirjoituksen tulisi alkaa. Esimerkiksi copy() saa tulostusiteraattorin kolmantena argumenttina. Jälleen, mille tahansa algoritmille, joka vaatii tulostusoperaattorin tuen, voidaan välittää mikä tahansa kohdissa 3, 4 ja 5 lueteltu iteraattorikategoria.
3. Eteenpäiniteraattoria (ForwardIterator) voidaan käyttää säiliön lukemisen ja kirjoittamisen läpikäyntiin yhteen suuntaan (niin, seuraava kategoria tukee kaksisuuntaista läpikäyntiä). Geneerisiin algoritmeihin, jotka vaativat minimissään eteenpäiniteraattorin, kuuluvat adjacent _find(), swap_range() ja replace(). Jälleen jokaiselle algoritmille, joka vaatii eteenpäiniteraattorin tuen, voidaan välittää kohdissa 4 ja 5 luetellut iteraattorikategoriat.
4. Kaksisuuntaisuusiteraattori (BidirectionalIterator) lukee ja kirjoittaa säiliöön molempiin suuntiin. Geneerisiin algoritmeihin, jotka vaativat minimissään kaksisuuntaisuusiteraattoria, kuuluvat in place_merge(), next_permutation() ja reverse().
5. Hajakäsittelyiteraattorilla (RandomAccessIterator) päästään säiliön mihin tahansa positioon vakioaikana. Sen lisäksi se tukee kaikkia kaksisuuntaisuusiteraattorin toimintoja. Geneerisiin algoritmeihin, jotka vaativat hajakäsittelyiteraattorin, kuuluvat binary_search(), sort_heap() ja nth_element().

Kartta (map), joukko (set) ja lista (list) ylläpitävät kaksisuuntaisuusiteraattoreita. Tämä tarkoittaa käytännössä, että niitä ei voida käyttää sellaisten geneeristen algoritmien kanssa, jotka vaativat hajakäsittelyiteraattoria, kuten sort_heap() ja nth_element(); katsomme kohdassa 12.6 vaihtoehtoisia operaatioita, joita listalle on käytettävissä. Vektori (vector) ja pakka (deque) ylläpitävät hajakäsittelyiteraatioita, jolloin niitä voidaan käyttää kaikissa geneerisissä algoritmeissa.

Harjoitus 12.6

Selitä, miksi kaikki seuraavat ovat virheellisiä. Yksilöi, mitkä virheet saadaan kiinni käännöksen aikana.

(a)
```
const vector<string> file_names( sa, sa+6 );
vector<string>::iterator it = file_names.begin()+2;
```

(b)
```
const vector<int> ivec;
fill( ivec.begin(), ivec.end(), ival );
```

(c)
```
sort( ivec.begin(), ivec.rend() );
```

(d)
```
list<int> ilist( ia, ia+6 );
binary_search( ilist.begin(), ilist.end() );
```

(e)
```
sort( ivec1.begin(), ivec2.end() );
```

Harjoitus 12.7

Kirjoita ohjelma, joka lukee kokonaislukuja vakiosyötöstä ja käyttää siihen istream_iterator-iteraattoria. Kirjoita parittomat numerot yhteen tiedostoon käyttäen ostream_iterator-iteraattoria. Jokainen arvo tulisi erottaa toisista tyhjällä merkillä. Kirjoita parilliset numerot toiseen tiedostoon käyttäen myös ostream_iterator-iteraattoria. Jokainen näistä arvoista tulisi sijoittaa omalle riville.

12.5 Geneeriset algoritmit

Kaikkien geneeristen algoritmien kaksi ensimmäistä argumenttia (lukuunottamatta kourallista välttämättömiä poikkeuksia, jotka vahvistavat sääntöä) ovat iteraattoripari, jota kutsutaan yleisesti ensimmäiseksi (first) ja viimeiseksi (last) ja ne ilmaisevat operoitavan säiliön tai sisäisen taulukon elementtialuetta. Elementtialueen ilmaisu (kutsutaan joskus *vasemmalta mukaan -jaksoksi*) kirjoitetaan usein näin

```
// luetaan: mukaan kuuluu ensimmäinen ja
// jokainen elementti siitä eteenpäin lukuunottamatta viimeistä
[ first, last )
```

Tämä ilmaisee, että alue alkaa kohdasta first ja jatkuu siitä eteenpäin, mutta siihen ei kuulu last. Kun

```
first == last
```

sanotaan alueen olevan tyhjä.

Iteraattoriparin vaatimus on, että pitää olla mahdollista saavuttaa last alkaen kohdasta first kasvatusoperaattoria toistamalla. Kääntäjä ei voi kuitenkaan itse pakottaa tähän; tämän vaatimuksen täyttämättä jättäminen johtaa tuntemattomaan suorituksenaikaiseen käyttäytymiseen — usein ohjelman muistivedokseen eli *dumppiin*.

Jokaisen algoritmin esittelyssä ilmaistaan iteraattorituen minimikategoria, jonka se vaatii (katso kohdasta 12.4 lyhyt kuvaus viidestä iteraattorikategoriasta). Esimerkiksi find() — joka toteuttaa säiliön yksisuuntaisen, vain-luku-läpikäynnin — vaatii minimissään syöttöiteraattorin. Sille voidaan välittää myös eteenpäin-, kaksisuuntaisuus- tai hajakäsittelyiteraattori. Tulostusoperaattorin välittäminen sille johtaa virheeseen. Virheitä, jotka aiheutuvat sopimattoman iteraattorikategorian välittämisestä algoritmille, ei taata havaittavan käännöksen aikana, koska iteraattorikategoriat eivät ole todellisia tyyppejä. Sen sijaan ne ovat tyyppiparametreja, jotka välitetään funktiomallille.

Jotkut algoritmit tukevat useita versioita; yksi käyttää sisäistä operaattoria ja toinen hyväksyy joko funktio-olion tai osoittimen funktioon tarjoten vaihtoehtoisen toteutuksen tuosta operaattorista. Esimerkiksi unique() vertaa oletusarvoisesti kahta vierekkäistä elementtiä käyttäen säiliön taustalla olevan tyypin yhtäsuuruusoperaattoria. Jos taustalla olevalla tyypillä ei kuitenkaan ole yhtäsuuruusoperaattoria tai jos haluamme määritellä elementin yhtäsuuruuden eri tavalla, välitämme joko funktio-olion tai osoittimen funktioon funktiolle, joka suorittaa halutut toimenpiteet. Toiset algoritmit on kuitenkin erotettu kahteen yksilöllisesti nimettyyn ilmentymään, jotka päättyvät (*väittävään*) predikaattiliitteeseen _if kuten find_if(). On olemassa esimerkiksi replace()-ilmentymä, jossa käytetään sisäistä yhtäsuuruusoperaattoria ja replace_if()-ilmentymä, joka saa predikaattina funktio-olion tai osoittimen funktioon..

Niille algoritmeille, jotka muokkaavat operoimaansa säiliötä, on yleensä olemassa kaksi versiota: paikalleen muuttava versio, joka muuttaa sen kanssa käytettyä säiliötä ja versio, joka palauttaa säiliöstä kopion, johon muutokset kohdistuvat. On olemassa esimerkiksi replace()- ja replace_copy()-algoritmit. Kopiointiversion nimi sisältää aina sanan _copy. Kaikilla säiliöönsä muuntavilla algoritmeilla ei kuitenkaan ole kopiointiversiota. Esimerkiksi sort()-algoritmeilla ei ole kopiointi-ilmentymää. Jos siinä tapauksessa haluamme algoritmin kopioivan, meidän pitää tehdä ja välittää kopiointi itse.

Jotta jotain geneerisistä algoritmeista voitaisiin käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <algorithm>
```

Jos haluamme käyttää jotain neljästä numeerisesta algoritmista — adjacent_difference(), accumulate(), inner_product() tai partial_sum() — pitää ottaa mukaan otsikkotiedosto

```
#include <numeric>
```

Algoritmit on jaettu seuraaviin yhdeksään kategoriaan (kategoriat ovat mukavuussyistä algoritmien esittelyjärjestyksessä eikä niillä ole virallista yhteyttä vakiokirjastoon). Jokainen algoritmi on käsitelty ja kuvattu aakkosjärjestyksessä tämän kirjan liitteessä.

12.5.1 Etsintäalgoritmit

13 etsintäalgoritmia tarjoavat useita strategioita sen päättelyyn, löytyykö jokin arvo säiliöstä. Algoritmeilla equal_range(), lower_bound() ja upper_bound() voidaan tehdä binäärihakuja. Ne ilmaisevat, mihin kohtaan säiliötä arvo voidaan lisätä ja silti säilyttää sen lajittelujärjestys.

adjacent_find(), binary_search(), count(), count_if(), equal_range(),
find(), find_end(), find_first_of(), find_if(), lower_bound(),
upper_bound(), search(), search_n()

12.5.2 Lajittelualgoritmit ja yleiset järjestelyalgoritmit

14 lajittelualgoritmia ja yleistä järjestelyalgoritmia tarjoavat useita strategioita säiliön elementtien järjestelyyn. Osittamalla voidaan säiliöt jakaa kahteen ryhmään. Ensimmäinen ryhmä muodostuu niistä elementeistä, jotka tyydyttävät ehdon; toinen ryhmä niistä, jotka eivät tyydytä. Voimme osittaa säiliön esimerkiksi sen perusteella, ovatko elementit parittomia vai parillisia tai alkaako sana isolla kirjaimella vai ei. Stabiili (*stable*) algoritmi säilyttää sellaisten elementtien suhteellisen järjestyksen, jotka ovat samanarvoisia tai täyttävät jonkin ehdon samalla tavalla. Esimerkiksi merkkijonojen

{ "pshew", "honey", "Tigger", "Pooh" }

stabiili ositus, joka perustuu siihen, alkaako sana isolla kirjaimella, generoi merkkijonot niin, että ne säilyttävät suhteellisen järjestyksensä:

{ "Tigger", "Pooh", "pshew", "honey" }

Tätä ei taata algoritmin stabiilittomalle ilmentymälle. (Huomaa, että lajittelualgoritmeja ei voida käyttää listan tai assosiatiivisten säiliöiden kuten joukon tai kartan yhteydessä.)

inplace_merge(), merge(), nth_element(), partial_sort(),
partial_sort_copy(), partition(), random_shuffle(), reverse(),
reverse_copy(), rotate(), rotate_copy(), sort(), stable_sort(),
stable_partition()

12.5.3 Poisto- ja korvausalgoritmit

15 poisto- ja korvausalgoritmia tarjoavat useita strategioita korvata tai poistaa yksi elementti tai elementtialue. unique() poistaa vierekkäiset, samanlaiset elementit. iter_swap() vaihtaa keskenään iteraattoriparin osoittamien elementtien arvot; se ei muuta itse iteraattoreita.

copy(), copy_backwards(), iter_swap(), remove(), remove_copy(),
remove_if(), remove_if_copy(), replace(), replace_copy(),
replace_if(), replace_copy_if(), swap(), swap_range(), unique(),
unique_copy()

12.5.4 Permutaatioalgoritmit

Mietipä seuraavaa kolmen merkin sarjaa: {a,b,c}. On olemassa kuusi mahdollista permutaatiota tälle sarjalle: abc, acb, bac, bca, cab ja cba. Lisäksi nämä permutaatiot on järjestetty aakkosten mukaan pienempi kuin -operaattorilla. Tarkoittaa, että abc on ensimmäinen permutaatio. Miksi? Siksi, koska jokainen elementti on pienempi kuin elementti, joka tulee sen jälkeen. acb on seuraava permutaatio, koska sitä ankkuroi a, joka on sarjan pienin elementti. Samalla tavalla permutaatiot, joita b ankkuroi, joka on toiseksi pienin elementti, tulevat ennen niitä, joita c ankkuroi. Kahdesta permutaatiosta, bac ja bca, tulee bac ennen bca:ta, koska sarja ac on pienempi kuin sarja ca. Permutaatiosta bca voimme sanoa, että sitä edellinen permutaatio on bac ja sitä seuraava permutaatio on cab. Sarjalla abc ei ole edellistä permutaatiota eikä sarjalla cba seuraavaa permutaatiota.

next_permutation(), prev_permutation()

12.5.5 Numeeriset algoritmit

Seuraavilla neljällä algoritmilla voidaan tehdä numeerisia operaatioita säiliölle. Jotta niitä voidaan käyttää, pitää ottaa mukaan otsikkotiedosto <numeric>.

accumulate(), partial_sum(), inner_product(), adjacent_difference()

12.5.6 Generointi- ja muunnosalgoritmit

Kuudella generointi- ja muunnosalgoritmilla voidaan joko täyttää uusi sarja tai korvata olemassaoleva sarja arvokokoelmalla.

fill(), fill_n(),for_each(), generate(), generate_n(), transform().

12.5.7 Vertailualgoritmit

Seitsemän vertailualgoritmia tarjoavat useita strategioita säiliöiden vertailuun (min() ja max() vertailevat yksinkertaisesti kahta elementtiä). lexicographical_compare() käyttää aakkosjärjestystä (katso lisää liitteestä permutaatioiden käsittelyn lopusta).

equal(), includes(), lexicographical_compare(), max(), max_element(),
min(), min_element(), mismatch()

12.5.8 Asetusalgoritmit

Neljällä asetusalgoritmilla (*set*) voidaan tehdä yleisiä asetusoperaatioita mille tahansa säiliötyypille. *Union* lajittelee kahden säiliön elementit. *Intersection* luo lajitellun elementtijonon elementeistä, jotka löytyvät molemmista säiliöistä. *Difference* luo lajitellun elementtijonon elementeistä, jotka löytyvät ensimmäisestä säiliöstä, mutta eivät toisesta. *Symmetric difference* luo lajitellun elementtijonon elementeistä, jotka ovat olemassa jommassakummassa säiliössä, mutta eivät molemmissa.

set_union(), set_intersection(), set_difference(),
set_symmetric_difference()

12.5.9 Kekoalgoritmit

Keko (*heap*) on binääripuun muoto, joka esitetään taulukkona. Vakiokirjastossa on max-heap-esitystapa, jossa jokaisen solmun avaimen arvo on suurempi tai yhtäsuuri kuin sen alisolmujen avaimet.

make_heap(), pop_heap(), push_heap(), sort_heap()

12.6 Milloin geneerisiä algoritmeja ei tulisi käyttää?

Assosiatiiviset säiliöt kuten kartat ja joukot ylläpitävät sisäisesti elementtinsä järjestettynä, jotta niitä voitaisiin nopeasti etsiä ja hakea arvoja. Siksi assosiatiivisille säiliöille ei ole sallittua käyttää geneerisiä algoritmeja kuten sort() ja partition() niiden uudelleenjärjestelyyn. Jos assosiatiivisen säiliön elementit pitää järjestää uudelleen, ne pitää ensin kopioida peräkkäissäiliöön kuten vektoriin tai listaan.

Listasäiliö on kahdesti linkitetty lista: varsinaisen tiedon lisäksi jokainen elementti ylläpitää kahden jäsenen eli seuraavan ja edellisen listaelementin osoitusta. Listan pääetu on tehokas elementin tai elementtialueen lisäys ja poisto mistä tahansa listan kohdasta. Päähaitta on elementtien hajakäsittelyn puute. Vaikka voimme kirjoittaa esimerkiksi

```
vector<string>::iterator vec_iter = vec.begin() + 7;
```

kun haluamme alustaa vec_iter:in vektorin kahdeksannen elementin osoitteella, ei ole sallittua kirjoittaa

```
// virhe: lista ei tue iteraattoriaritmetiikkaa
list<string>::iterator list_iter = slist.begin() + 7;
```

koska listan elementit eivät ole yhtenäisenä. Jotta pääsisimme listan kahdeksanteen elementtiin, pitää välissä olevat linkit käydä läpi.

Koska listasäiliö ei tue hajakäsittelyä, ei geneerisiä algoritmeja merge(), remove(), reverse(), sort() ja unique() kannata käyttää listaolioiden yhteydessä, vaikka yksikään näistä algoritmeista ei pakosta vaadi hajakäsittelyiteraattoria (RandomAccessIterator). Sen sijaan jokaisella algoritmilla on tietyt listan jäsenilmentymät (listakohtaisena splice()-operaationa):

- list::merge() yhdistää lajitellun listan toisen lajitellun listan kanssa.
- list::remove() poistaa elementit, jotka ovat tietyn arvoisia.
- list::remove_if() poistaa elementit, jotka täyttävät jonkin ehdon.
- list::reverse() laittaa listan elementit päinvastaiseen järjestykseen.
- list::sort() lajittelee listan elementit.
- list::splice() siirtää yhden listan elementit tähän listaan.
- list::unique() poistaa elementin vierekkäiset kopiot.

12.6.1 list::merge()

```
void list::merge( list rhs );
template <class Compare>
   void list::merge( list rhs, Compare comp );
```

Kahden järjestetyn listan elementit yhdistetään joko taustalla olevan elementtityypin pienempi kuin -operaattorin perusteella tai käyttäjän määrittämän vertailuoperaation perusteella. (Huomaa, että rhs:n elementit *siirretään* listaolioon, joka käynnistää merge():n; operaation jälkeen rhs on tyhjä.) Esimerkiksi:

```
int array1[ 10 ] = { 34, 0, 8, 3, 1, 13, 2, 5, 21, 1 };
int array2[ 5 ] =  { 377, 89, 233, 55, 144 };

list< int > ilist1( array1, array1 + 10 );
list< int > ilist2( array2, array2+5 );

// merge vaatii, että molemmat listat on järjestetty
ilist1.sort(); ilist2.sort();
ilist1.merge( ilist2 );
```

Kun merge()-operaatiota on käytetty, on ilist2 tyhjä; ilist1 sisältää ensimmäiset 15 elementtiä Fibonacci-jonosta nousevassa järjestyksessä.

12.6.2 list::remove()

```
void list::remove( const elemType &value );
```

remove()-operaatio poistaa kaikki määritetyn arvon (value) mukaiset ilmentymät. Esimerkiksi:

```
ilist1.remove( 1 );
```

12.6.3 list::remove_if()

```
template < class Predicate >
   void list::remove_if( Predicate pred );
```

remove_if()-operaatio poistaa kaikki elementit, jotka täyttävät määrätyn ehdon. Esimerkiksi

```
class Even {
public:
        bool operator()( int elem ) { return ! ( elem % 2 ); }
};

ilist1.remove_if( Even() );
```

poistaa listaolion kaikki parilliset elementit, jotka määriteltiin merge():n yhteydessä.

12.6.4 list::reverse()

```
void list::reverse();
```

reverse()-operaatio kääntää listan elementit päinvastaiseksi.

```
ilist1.reverse();
```

12.6.5 list::sort()

```
void list::sort();
template <class Compare>
  void list::sort( Compare comp );
```

Oletusarvo on, että sort()-operaatio laittaa listan elementit nousevaan järjestykseen taustalla olevan elementtityypin pienempi kuin -operaattorin perusteella. Vaihtoehtoinen vertailuoperaattori voidaan määrittää argumenttina. Esimerkiksi

```
list1.sort();
```

järjestää list1:n nousevaan järjestykseen, kun taas

```
list1.sort( greater<int>() );
```

järjestää list1:n laskevaan järjestykseen käyttämällä suurempi kuin -operaattoria.

12.6.6 list::splice()

```
void list::splice( iterator pos, list rhs );
void list::splice( iterator pos, list rhs, iterator ix );
void list::splice( iterator pos, list rhs,
                   iterator first, iterator last );
```

splice()-operaattori siirtää yhden elementin tai elementtialueen yhdestä listasta toiseen. Tähän on olemassa kolme eri muotoa: siirrä kaikki elementit yhdestä listasta toiseen, siirrä elementtialue yhdestä listasta toiseen ja siirrä yksi elementti yhdestä listasta toiseen. Jokaisessa muodossa tarvitaan iteraattori, joka ilmaisee paikan, johon elementti tai elementtialue lisätään. Elementit siirretään paikkaan välittömästi ennen juuri tuota kohtaa. Esimerkiksi seuraaville kahdelle listalle

```
int array[ 10 ] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
list< int > ilist1( array, array + 10 );
list< int > ilist2( array, array+2 ); // sisältää 0, 1
```

seuraavan splice():n käyttö siirtää ilist1:n ensimmäisen elementin ilist2:een. ilist2 sisältää nyt elementit 0, 1 ja 0, kun taas ilist1 ei enää sisällä arvoa 0:

```
// ilist2.end() ilmaisee paikan siirrettävälle elementille
// siirretyt elementit tulevat juuri ennen tuota paikkaa
// ilist1 ilmaisee listaa, josta elementit siirretään
// ilist1.begin() ilmaisee, mikä elementti siirretään

ilist2.splice( ilist2.end(), ilist1, ilist1.begin() )
```

Seuraavassa splice():n käytön yhteydessä välitetään kaksi iteraattoria, jotka ilmaisevat siirrettävän elementtialueen:

```
list< int >::iterator first, last;

first = ilist1.find( 2 );
last  = ilist1.find( 13 );
ilist2.splice( ilist2.begin(), ilist1, first, last );
```

Tässä tapauksessa siirretään ilist1-listasta elementit 2, 3, 5 ja 8 ilist2-listan eteen. Nyt ilist1 sisältää viisi elementtiä: 1, 1, 13, 21 ja 34. Jotta voisimme siirtää nämä viimeiset elementit ilist2-listaan, voimme käyttää splice()-operaattorin viimeistä muotoa:

```
list< int >::iterator pos = ilist2.find( 5 );
ilist2.splice( pos, ilist1 );
```

ilist1 on nyt tyhjä. Jäljellä olevat viisi elementtiä on siirretty ilist2:een välittömästi ennen elementin positioarvoa 5.

12.6.7 list::unique()

```
void list::unique();
template <class BinaryPredicate>
   void list::unique( BinaryPredicate pred );
```

unique()-operaatio poistaa elementtien vierekkäiset kopiot. Oletusarvo on, että se käyttää taustalla olevan tyypin yhtäsuuruusoperaattoria. Kun esimerkiksi arvoille {0,2,4,6,4,2,0} käytetään unique()-operaatiota, se johtaa täsmälleen samaan seitsemän elementin listaan, koska vierekkäisiä, samanlaisia elementtejä ei ole. Jos ensin lajittelemme listan, joka johtaa tulokseen {0,0,2,2,4,4,6}, on unique()-operaation käytön tuloksena neljä yksilöllistä arvoa {0,2,4,6}.

```
ilist.unique();
```

Toinen unique()-operaation muoto hyväksyy vaihtoehtoisen vertailuoperaattorin. Esimerkiksi

```
class EvenPair {
public:
        bool operator()( int val1, int val2 )
              { return ! ( val2 % val1 );    }
};

ilist.unique( EvenPair() );
```

poistaa kaikki vierekkäiset elementit, joissa toinen elementti jakautuu tasan ensimmäisen kanssa.

Näitä jäsenoperaatioita tulisi käyttää listaolioon mieluummin kuin niiden geneerisiä algoritmikumppaneita. Toiset geneeriset algoritmit kuten find(), transform(), for_each() jne. toimivat yhtä tehokkaasti listaolioilla (jälleen, yksittäiset geneeriset algoritmit on käsitelty yksityiskohtaisesti liitteessä).

Harjoitus 12.8

Toteuta kohdan 12.2 ohjelma uudelleen käyttämällä listaa vektorin sijasta.