6

Abstraktit säiliötyypit

Tämä luku on laajennus ja päätös luvuille 3 ja 4. Jatkamme tyyppien käsittelyä, joka aloitettiin luvussa 3, esittelemällä lisätietoja string- ja vektorityypeistä kuten myös muita C++-vakiokirjaston säiliötyyppejä. Lisäksi jatkamme luvussa 4 aloitettua operaattoreiden ja lausekkeiden käsittelyä esittelemällä operaatioita, joita tuetaan säiliötyyppien olioille.

Jonosäiliö (sequence container) pitää sisällään järjestetyn kokoelman yhdentyyppisiä elementtejä. Kaksi tärkeintä jonosäiliötä ovat vektori- ja listatyyppi. (Kolmas jonosäiliö, pakka (deque) — lausutaan englanniksi "deck" — omaa samanlaisen käyttäytymisen kuin vektori, mutta on erikoistunut tukemaan tehokasta ensimmäisen elementtinsä lisäystä ja poistoa. Pakkaa pidetään vektoria parempana esimerkiksi jonon toteutuksessa, joka on abstraktio, jossa ensimmäinen elementti haetaan joka kerta. Kun tekstissä kuvaamme vektorin tukemia operaatioita, myös pakka tukee niitä.)

Assosiatiivinen säiliö tukee tehokasta elementin hakua. Kaksi tärkeintä assosiatiivista säiliötyyppiä ovat assosiatiivinen taulukko (map) ja joukko (set). Assosiatiivinen taulukko on avain-arvopari: avainta käytetään hakuun ja arvo sisältää tiedon, jota haluamme käyttää. Assosiatiivinen taulukko tukee esimerkiksi puhelinluetteloa hyvin: avain on yksilön nimi ja arvo on siihen liittyvä puhelinnumero.

Assosiatiivinen joukko sisältää yhden avainarvon ja se tukee tehokasta kyselyä, jos se on mukana. Esimerkiksi tekstinkyselyjärjestelmä voisi muodostaa joukon sanoja, jotka jätetään pois kuten *the*, *and*, *but* jne., kun tehdään tekstin sanoista tietokantaa. Ohjelma lukisi tekstin jokaisen sanan vuorollaan ja tarkistaisi, kuuluuko se pois jätettäviin sanoihin, ja hylkäisi sanan tai lisäisi sen tietokantaan riippuen kyselyn tuloksesta.

Sekä taulukko että joukko voivat sisältää vain yhden esiintymän jokaisesta avaimesta. Monitaulukko ja monijoukko tukevat useamman samanlaisen avaimen esiintymää. Puhelinluettelomme esimerkiksi tulisi tukea yhden yksilön useita listauksia. Eräs menetelmä sen toteuttamiseksi on assosiatiivisen monitaulukon käyttö.

Seuraavissa kappaleissa katsomme säiliötyyppejä yksityiskohtaisesti perustellen niiden käyttöä vähitellen kasvavalla pienellä tekstinkyselyohjelmalla.

6.1 Tekstinkyselyjärjestelmämme

Mistä tekstinkyselyjärjestelmä muodostuu?

1. Käyttäjän ilmaisemasta mielivaltaisesta tekstitiedostosta.
2. Boolean-kyselypiirteestä, jossa käyttäjä voi etsiä sanaa tai vierekkäisten sanojen jonoa tekstistä.

Jos sana tai vierekkäiset sanat löytyvät, näytetään jokaisen sanan esiintymien lukumäärä ja ja sanojen jono. Jos käyttäjä haluaa, lause tai lauseet, johon sana tai sanojen jono kuuluu, näytetään myös. Jos esimerkiksi käyttäjä haluaisi etsiä kaikki viittaukset joko sanoihin Civil War tai Civil Rights, kysely voisi näyttää tältä[1]:

```
Civil && ( War || Rights )
```

Kyselyn tulos voisi näyttää tältä:

```
Civil: 12 occurrences
War:    48 occurrences
Rights: 1 occurrence

Civil && War: 1 occurrence
Civil && Rights: 1 occurrence

(8) Civility, of course, is not to be confused with
Civil Rights, nor should it lead to Civil War.
```

jossa (8) ilmaisee tekstin lauseen numeroa. Järjestelmämme pitää olla tarpeeksi fiksu, niin ettei se näytä samaa lausetta useaan kertaan. Lisäksi useammat lauseet tulisi näyttää nousevassa järjestyksessä (lause 7 tulisi siis aina näyttää ennen lausetta 9).

1. Huomaa: jotta voisimme yksinkertaistaa toteutusta, vaadimme tyhjän merkin erottamaan jokaista sanaa mukaan lukien sulut ja Boolean-operaattorit. Niin, että ohjelmamme ei ymmärrä

```
(War || Rights)
```

(tyhjä merkki on välttämätön ennen W-kirjainta) eikä

```
Civil&&(War||Rights)
```

Vaikka tämä on kohtuutonta tosielämän järjestelmille, joissa käyttäjän mukavuus menee aina toteuttajien mukavuuden edelle, uskomme, että se on hyväksyttävämpää pääosin opetukseen tarkoitetussa tekstissä kuten tässä.

Mitä tehtäviä ohjelmamme pitää tukea?

1. Sen pitää sallia käyttäjän ilmaista avattavan tekstitiedoston nimi, avata se ja lukea teksti.
2. Sen pitää järjestää sisäisesti tekstitiedosto niin, että se voi yksilöidä jokaisen sanan ilmentymien lukumäärän lauseen ehdoilla ja sen sijaintipaikan tuossa lauseessa.
3. Sen pitää tukea jonkin muotoista Boolean-kyselykieltä. Meidän tapauksessamme se tukee seuraavaa:

   && Molemmat sanat eivät ainoastaan löydy, vaan ovat vierekkäin rivillä.

   || Toinen tai molemmat sanat löytyvät riviltä.

   ! Sanaa ei löydy riviltä.

   () Tarkoittaa kyselyn aliryhmitystä.

Täten voidaan kirjoittaa

```
Lincoln
```

kun halutaan etsiä lauseet, joissa sana *Lincoln* esiintyy, tai kirjoittaa

```
! Lincoln
```

etsiäksemme kaikki lauseet, joissa sana *Lincoln* ei esiinny, tai kirjoittaa

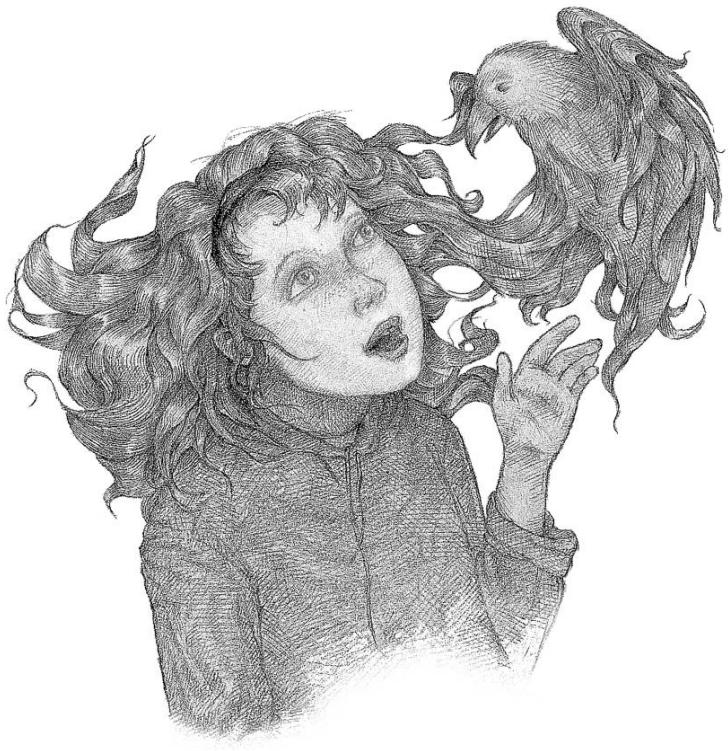
```
( Abe || Abraham ) && Lincoln
```

rajoittaaksemme valittavia lauseita niihin, joissa eksplisiittisesti viitataan sanoihin Abe Lincoln tai Abraham Lincoln.

Teemme kaksi toteutusta järjestelmästämme. Tässä luvussa teemme toteutuksen, joka ratkaisee sanojen ja niihin liittyvien rivi- ja sarakepaikkojen haku- ja tallennusongelman assosiatiivisella taulukolla. Kokeillaksemme tätä ratkaisua, teemme yksittäisen sanan kyselyjärjestelmän. Luvussa 17 teemme täydellisen kyselyjärjestelmän toteutuksen, joka tukee vertailuoperaattoreita, joista puhuimme edellisissä kappaleissa. Viivytämme sen toteutusta sinne saakka, koska ratkaisuun liittyy oliokeskeisen Query-luokkahierarkian käyttöä.

Käsiteltävänä tekstinä käytämme seuraavan sivun julkaisematonta lastentarinaa, jonka Stan on kirjoittanut[2]:

2. Kuvitus: Elena Driskill, uudelleenpainos luvalla.



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?"

10

Käsittelymme lopussa tekstin sisäinen tallennus, joka tukee yksittäisiä kyselyjä, näyttää tältä. (Tähän liittyy yksittäisten tekstirivien lukemista, niiden erottelua yksittäisiin sanoihin,

välimerkkien poistamiesta, isojen kirjainten vaikutuksen poistamisesta, minimaalista tukea loppuliitteille ja merkitykseltään neutraalien sanojen kuten *and*, *a* ja *the* poistamiesta.)

```
alice ((0,0))
alive ((1,10))
almost ((1,9))
ask ((5,2))
beautiful ((2,7))
bird ((2,3),(2,9))
blow ((1,3))
daddy ((0,8),(3,3),(5,5))
emma ((0,1))
fiery ((2,2),(2,8))
flight ((2,5))
flowing ((0,4))
hair ((0,6),(1,6))
has ((0,2))
like ((2,0))
long ((0,3))
look ((1,8))
magical ((3,0))
mean ((5,4))
more ((4,12))
red ((0,5))
same ((4,5))
say ((0,9))
she ((4,0),(5,1))
shush ((3,4))
shyly ((5,0))
such ((3,8))
tell ((2,11),(4,1),(4,10))
there ((3,5),(5,7))
thing ((3,9))
through ((1,4))
time ((4,6))
untamed ((3,2))
wanting ((4,7))
wind ((1,2))
```

Seuraavassa on malli kyselyistunnosta, jossa käytetään tässä luvussa toteutettua ohjelmaa (käyttäjän kirjoitus on kursivoitu):

```
please enter file name: alice_emma

enter a word against which to search the text.
to quit, enter a single character ==> alice

alice occurs 1 time:
```

```
        ( line 1 ) Alice Emma has long flowing red hair. Her Daddy says

enter a word against which to search the text.
to quit, enter a single character ==> daddy

daddy occurs 3 times:

        ( line 1 ) Alice Emma has long flowing red hair. Her Daddy says
        ( line 4 ) magical but untamed. "Daddy, shush, there is no such thing,"
        ( line 6 ) Shyly, she asks, "I mean, Daddy, is there?"

enter a word against which to search the text.
to quit, enter a single character ==> phoenix

Sorry. There are no entries for phoenix.

enter a word against which to search the text.
to quit, enter a single character ==> .
Ok, bye!
```

Jotta voisimme helposti toteuttaa tämän ohjelman, meidän pitää käsitellä vakiokirjaston säiliötyypit yksityiskohtaisesti kuten myös vilkaista uudelleen string-luokkaa, joka esiteltiin luvussa 3.

6.2 Vektori vai lista?

Eräs ensimmäisistä asioista, jotka ohjelmamme tulee tehdä, on tallentaa tuntematon määrä sanoja tekstitiedostosta. Sanat puolestaan tallennetaan merkkijono-olioina. Ensimmäinen kysymyksemme on: tulisiko sanat tallettaa jonosäiliöön vai assosiatiiviseen säiliöön?

Jossain vaiheessa pitää tukea kyselyä, jossa kysytään sanan läsnäoloa, ja jos sana löytyy, pitää hakea sen esiintymät tekstistä. Koska sekä etsimme että haemme arvoa, tukisi assosiatiivinen taulukkotyyppi tätä soveliaimmin.

Kuitenkin ennen sitä täytyy yksinkertaisesti tallentaa syöttöteksti käsittelyä varten — tarkoittaa välimerkkien poistoa, loppuliitteiden käsittelyä jne. Tähän esikäsittelyvaiheeseen vaaditaan jonosäiliö assosiatiivisen säiliön sijasta. Kysymys kuuluu: tulisiko sen olla vektori vai lista?

Jos olet ohjelmoinut C:llä tai C++-esistandardilla, valintasi peukalosääntö menee jokseenkin näin: jos tallennettavien elementtien lukumäärä tiedetään käännöshetkellä, käytä taulukkoa. Jos tallennettavien elementtien määrä on tuntematon tai todennäköisesti vaihtelee paljon, käytä silloin listaa ja varaa dynaamisesti muistia jokaiselle oliolle ja liitä ne vuorollaan listaan.

Tämä peukalosääntö ei kuitenkaan päde jonosäiliötyypeille: vektori, pakka ja lista kasvavat dynaamisesti. Kriteerit valinnalle näiden kolmen kohdalla liittyvät pääosin lisäyspiirteisiin ja sen jälkeisiin elementtien käsittelyvaatimuksiin.

Vektori edustaa yhtenäistä muistialuetta, johon jokainen elementti tallennetaan vuorollaan.

Vektorin hajakäsittely — tarkoittaa elementtien 5, sitten 15, sitten 7 jne. käsittelyä — on erittäin tehokasta, koska jokainen käsittely tapahtuu kiinteällä siirtymällä vektorin alusta. Elementin lisäys minne muualle tahansa kuin vektorin loppuun on kuitenkin tehotonta, koska se vaatii jokaisen elementin kopioimisen lisättävästä elementistä oikealle. Samalla tavalla minkä tahansa muun kuin viimeisen elementin poisto on tehotonta, koska jokainen elementti poistettavasta elementistä oikealle pitää kopioida. Tämä voi olla erityisen raskasta suurille, monimutkaisille luokkaolioille. (Myös pakka edustaa yhtenäistä muistialuetta; kuitenkin toisin kuin vektori, se tukee myös tehokasta ensimmäisen elementtinsä poistoa ja lisäystä. Se saavuttaa tämän kaksitasoisella taulukkorakenteella, jossa yksi taso edustaa varsinaista säiliötä ja toinen taso osoittaa säiliön alkua ja loppua.)

Lista edustaa epäyhtenäistä muistialuetta, joka on kahteen suuntaan linkitetty osoitinparin avulla. Ne osoittavat elementtejä eteen sekä taakse ja mahdollistavat läpikäynnin sekä taakseettä eteenpäin. Elementtien lisäys ja poisto listan mihin tahansa paikkaan on tehokasta: osoittimet pitää asettaa uudelleen, mutta elementtejä ei tarvitse siirtää kopioimalla. Toisaalta hajakäsittelyä ei tueta kovin hyvin: elementin käsittely vaatii välillä olevien elementtien läpikäynnin. Lisäksi kaksi osoitinta per elementti saa aikaan muistin kuormitusta.

Tässä on joitakin valintaperusteita jonosäiliön valintaan:

- Jos vaadimme säiliöltä hajakäsittelyä, on vektori selkeä valinta listan sijasta.
- Jos tiedämme tallennettavien elementtien lukumäärän, on vektori jälleen parempi vaihtoehto listan sijasta.
- Jos meidän pitää lisätä ja poistaa elementtejä muualta kuin säiliön alusta tai lopusta, on lista selkeä valinta vektorin sijasta.
- Ellei meidän tarvitse lisätä tai poistaa elementtejä säiliön alusta, on vektori parempi pakan sijasta.

Mitä sitten, jos meidän pitää hajakäsitellä, -lisätä ja -poistaa elementtejä? Vertailtavana on hajakäsittelyn hinta vastaan yhtenäisten elementtien kopiointi vasemmalle tai oikealle. Yleensä ohjelman pääoperaation (etsintä tai lisäys) tulisi vaikuttaa säiliötyypin valintaan. (Tämän päätöksen tekeminen voi vaatia molempien säiliötyyppien suorituskyvyn profilointia.) Ellei kummankaan suorituskyky ole tyydyttävä, voi olla välttämätöntä suunnitella oma, monimutkaisempi tietorakenne.

Kuinka päätämme, minkä säiliön valitsemme, kun emme tiedä tallennettavien elementtien lukumäärää (säiliö tulee kasvamaan dynaamisesti) ja kun ei ole tarvetta hajakäsittelylle tai lisäykselle muualle kuin loppuun? Onko lista tai vektori tässä tapauksessa merkittävästi tehokkaampi? (Lykkäämme vastausta tähän kysymykseen seuraavaan kohtaan saakka.)

Lista kasvaa suoraviivaisella tavalla: joka kerta, kun uusi olio lisätään listaan, niiden kahden olion, joiden väliin uusi olio jää, toisen osoitin eteenpäin ja toisen olion osoitin taaksepäin asetetaan uudelleen osoittamaan uuteen olioon. Uuden olion osoittimet eteen- ja taaksepäin vuorostaan alustetaan osoittamaan näitä kahta elementtiä. Lista sisältää tarvittavaa muistia ainoastaan sisältämilleen elementeille. Kuormitus on kaksinkertainen: kaksi lisäosoitinta jokai-

selle arvolle ja arvon epäsuora käsittely osoittimen kautta.

Dynaamisen vektorin esitystapa ja kuormitus ovat monimutkaisempia. Katsomme sitä seuraavassa kohdassa.

Harjoitus 6.1

Mikä on kaikkein sopivin — vektori, pakka vai lista — seuraaviin ohjelmointitehtäviin (vai onko mikään hyvä?)

a. Lue tuntematon määrä sanoja tiedostosta, jonka tarkoituksena on generoida satunnaisia englannin kielen lauseita.
b. Lue kiinteä lukumäärä sanoja ja lisää ne säiliöön aakkosjärjestyksessä sitä mukaa, kun ne luetaan.
c. Lue tuntematon määrä sanoja. Lisää aina uudet sanat loppuun. Poista seuraava arvo alusta.
d. Lue tuntematon määrä kokonaislukuja tiedostosta. Lajittele numerot ja tulosta ne vakiotulostukseen.

6.3 Kuinka vektori kasvaa itsestään

Jotta vektori kasvaisi dynaamisesti, sen pitää varata muistia uudelle jonolle, kopioida vuorostaan vanhan jonon elementit ja vapauttaa niiden vanha muisti. Lisäksi, jos elementit ovat luokan olioita, kopiointi tai muistin vapautus voi vaatia kopiointimuodostajan ja tuhoajan käynnistyksen jokaiselle elementille vuorollaan. Koska lista vain yksinkertaisesti linkittää uudet elementit joka kerta, kun säiliö kasvaa, näyttää siltä, että lista on tehokkaampi näistä kahdesta säiliötyypistä sen tukiessa dynaamista kasvua. Mutta käytännössä näin ei ole asianlaita. Katsotaanpa, miksi.

Ollakseen edes vähänkään tehokas, ei vektori voi todellisuudessa kasvaa aina uudelleen jokaisen lisäyksen yhteydessä. Sen sijaan, kun vektorin pitää kasvattaa itseään, se varaa lisäkapasiteettia yli sen välittömän tarpeen — se pitää tätä muistia varastona. (Varatun lisäkapasiteetin täsmällinen määrä riippuu toteutuksen määrittelystä.) Tämä strategia mahdollistaa säiliölle merkittävästi tehokkaamman uudelleenkasvun — jopa niin, että itse asiassa pienille olioille vektori osoittautuu kasvavan tehokkaammin kuin lista. Katsokaamme joitakin esimerkkejä C++-vakiokirjaston Rogue Wave -toteutuksesta. Mutta tehkäämme aluksi selväksi ero säiliön kapasiteetin ja koon välillä.

Kapasiteetti on elementtien kokonaislukumäärä, jotka voidaan lisätä säiliöön, ennen kuin sen pitää kasvaa uudelleen itsekseen. (Kapasiteetti liittyy ainoastaan säiliöön, jossa muistitila on yhtenäinen: esimerkiksi vektori, pakka tai string. Lista ei vaadi kapasiteettia.) Saadaksemme selville vektorin kapasiteetin, käynnistämme sen `capacity()`-operaation. Toisaalta koko on elementtien lukumäärä, jonka säiliö sillä hetkellä sisältää. Kun haluamme hakea säiliön nykyisen koon, käynnistämme sen `size()`-operaation. Esimerkiksi:

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

int main()
{
    vector< int > ivec;
    cout << "ivec: size: " << ivec.size()
         << " capacity: "  << ivec.capacity() << endl;

    for ( int ix = 0; ix < 24; ++ix ) {
      ivec.push_back( ix );
      cout << "ivec: size: " << ivec.size()
           << " capacity: "  << ivec.capacity() << endl;
    }
}
```

Rogue Wave -toteutuksessa ivec-vektorin määrittelyn jälkeen sekä koko että kapasiteetti ovat 0. Kuitenkin ensimmäisen elementin lisäyksen jälkeen vektorin kapasiteetti on 256 ja sen koko on 1. Tämä tarkoittaa, että voidaan lisätä 256 elementtiä, ennen kuin sen täytyy kasvaa uudelleen itsekseen. Kun lisäämme 256. elementin, vektori kasvaa itsekseen uudelleen seuraavalla tavalla: se varaa kaksin verroin nykyisen kapasiteetin, kopioi nykyiset arvot uuteen varattuun muistialueeseen ja vapauttaa edellisen muistinsa. Kuten hetken kuluttua näemme, mitä monimutkaisempi tietotyyppi on, sitä tehottomampi vektorista tulee verrattuna listaan. Taulukosta 6.1 nähdään monia tietotyyppejä sekä niiden kokoja ja alkukapasiteetti niihin liittyvissä vektoreissa:

Taulukko 6.1Koko, kapasiteetti ja muutamia tietotyyppejä

Tietotyyppi	Koko tavuina	Kapasiteetti ensimmäisen lisäyksen jälkeen
int	4	256
double	8	128
yksinkertainen luokka #1	12	85
string	12	85
suuri, yksinkertainen luokka	8 000	1
laaja, monimutkainen luokka	8 000	1

Kuten näet, Rogue Wave -toteutuksessa elementtien oletuskapasiteetti, joka on melkein tai tasan 1 024 tavua, varataan ensimmäisessä lisäyksessä; sitten se kerrotaan kahdella jokaisessa uudelleenvarauksessa. Kun on suuri tietotyyppi ja pieni kapasiteetti, muistin uudelleenvarauk-

sesta ja elementtien kopioinnista tulee kuormitusta vektorin käytön yhteydessä. (Kun puhumme monimutkaisesta luokasta, tarkoitamme luokkaa, jossa on sekä kopiointimuodostaja että kopioinnin sijoitusoperaattori.) Taulukossa 6.2 näkyy aika sekunteina, kun lisätään kymmenen miljoonaa edellä mainitun tyyppistä elementtiä sekä listaan että vektoriin. Taulukosta 6.3 näkyy aika, kun lisätään 10 000 elementtiä (suurempi elementin koko oli liian hidas).

Taulukko 6.2 Aika sekunteina, kun lisätään 10 000 000 elementtiä

Tietotyyppi	Lista	Vektori
int	10.38 s	3.76 s
double	10.72 s	3.95 s
yksinkertainen luokka	12.31 s	5.89 s
string	14.42 s	11.80 s

Taulukko 6.3 Aika sekunteina, kun lisätään 10 000 elementtiä

Tietotyyppi	Lista	Vektori
suuri yksinkertainen luokka	0.36 s	2.23 s
suuri monimutkainen luokka	2.37 s	6.70 s

Kuten näet, pienille tietotyypeille vektori toimii merkittävästi paremmin kuin lista, kun taas suurille olioille pätee päinvastainen: lista toimii huomattavasti paremmin. Tämä ero johtuu vektorin tarpeesta kasvaa uudelleen ja elementtien kopioinnista. Tietotyypin koko ei kuitenkaan ole ainoa valintaperuste, joka vaikuttaa säiliön suorituskykyyn. Myös tietotyypin monimutkaisuus vaikuttaa elementin lisäyksen suorituskykyyn. Miksi?

Joko listan tai vektorin elementin lisäys vaatii sellaisen luokan kopiointimuodostajan käynnistyksen, jossa se on määritelty. (Kopiointimuodostaja alustaa yhden luokkaolion toisella saman luokan tyypillä — katso kohdasta 2.2 perustiedot ja kohdasta 14.5 tarkemmat tiedot.) Tämä selittää eron yksinkertaisen luokan ja string-luokan listaan lisäyksessä. Yksinkertaisen luokan oliot ja laajan yksinkertaisen luokan oliot lisätään biteittäisellä kopioinnilla (olion bitit kopioidaan toisen olion bitteihin), kun taas string-luokan oliot ja suurten, monimutkaisten luokkien oliot lisätään käynnistämällä merkkijonon kopiointimuodostaja.

Lisäksi vektorin pitää kuitenkin käynnistää kopiointimuodostaja jokaiselle elementilleen muistinsa uudelleenvarauksessa. Lisäksi aikaisemman muistin vapautus vaatii vastaavan luokan tuhoajan käynnistyksen jokaiselle elementille (jälleen: katso muodostajasta perustiedot kohdasta 2.2). Mitä useammin vektorin pitää kasvaa uudelleen itsestään, sitä raskaammaksi elementtien lisäys tulee.

Eräs ratkaisu on tietysti vaihtaa vektori listaan, kun vektorista tulee liian raskas. Usein parempi ratkaisu on tallentaa luokan suuret tai monimutkaiset oliot epäsuorasti osoittimella. Kun esimerkiksi tallennamme monimutkaisen luokan olion osoittimella, kuorma 10 000 elementin lisäyksestä vektoriin vähenee dramaattisesti 6.70 sekunnista 0.82 sekuntiin. Miksi? Kapasiteetti kasvaa arvosta 1 arvoon 256, joten uudelleenvarausten lukumäärä putoaa huomattavasti. Toiseksi kopiointi ja luokan olion osoittimen varauksen poisto ei vaadi luokan kopiointimuodostajaa eikä tuhoajaa.

reserve()-operaatio mahdollistaa, että ohjelmoija voi asettaa säiliön kapasiteettiin eksplisiittisen arvon. Esimerkiksi

```
int main {
    vector< string > svec;
    svec.reserve( 32 ); // asettaa kapasiteetin arvoon 32
    // ...
}
```

johtaa siihen, että svec-vektorin koko on nolla elementtiä, mutta kapasiteetti on 32. Se, mitä havaitsimme kokeilemalla oli kuitenkin, että vektorin oletuskapasiteetin säätäminen muuhun kuin arvoon 1 näytti aina aiheuttavan suorituskyvyn alentumista. Esimerkiksi sekä string- että double-vektorien kapasiteetin kasvatus reserve()-funktion avulla johti huonompaan suorituskykyyn. Toisaalta suuren, monimutkaisen luokan kapasiteetin kasvatus sai aikaan merkittävää suorituskyvyn parannusta kuten taulukosta 6.4 nähdään.

Taulukko 6.4 Aika sekunteina, kun lisätään 10 000 elementtiä säätämällä kapasiteettia*

Kapasiteetti	Aika sekunteina
oletusarvo 1	6.70 s
4 096	5.55 s
8 192	4.44 s
10 000	2.22 s
*(Muu kuin yksinkertainen luokka: 8 000 tavua sekä kopiointimuodostajalle että tuhoajalle)	

Tekstinkyselyjärjestelmämme käyttää vektoria, joka tulee sisältämään string-oliomme oletuskapasiteetillaan. Vaikka vektori kasvaa dynaamisesti, kun lisäämme siihen tuntemattoman määrän merkkijonoja kuten ajanottomme näyttää, se toimii silti hieman paremmin kuin lista. Ennen kuin menemme varsinaiseen toteutukseemme, katsokaamme vielä uudelleen, kuinka voimme määritellä säiliöolion.

Harjoitus 6.2

Selitä vektorin kapasiteetin ja koon välinen ero. Miksi on tarpeellista tukea säiliön kapasiteettiajatusta, joka tallentaa elementit yhtenäisesti, mutta ei esimerkiksi listan?

Harjoitus 6.3

Miksi on tehokasta tallentaa suuri, monimutkaisen luokan olio osoittimella, mutta tehotonta kokonaislukuolioiden kokoelman yhteydessä?

Harjoitus 6.4

Kumpi on sopivampi säiliötyyppi seuraavissa tilanteissa, lista vai vektori? Jokaisessa tapauksessa lisätään tuntematon määrä elementtejä. Perustele vastauksesi.

(a) Kokonaislukuarvot
(b) Suuren, monimutkaisen luokan olioiden osoittimet
(c) Suuren, monimutkaisen luokan oliot

6.4 Jonosäiliön määrittely

Jotta säiliöolio voidaan määritellä, pitää ensiksi ottaa mukaan siihen liittyvä otsikkotiedosto, joka on jokin näistä:

```
#include <vector>
#include <list>
#include <deque>
#include <map>
#include <set>
```

Määrittely alkaa säiliöolion nimellä ja sen sisältämien elementtien todellisella tyypillä[3]. Esimerkiksi

```
vector< string > svec;
list< int >      ilist;
```

määrittelee svec:in tyhjäksi string-olioiden vektoriksi ja ilist:in tyhjäksi int-tyyppisten olioiden listaksi. Sekä svec että ilist ovat tyhjiä. Jotta voimme olla varmoja siitä, voimme käynnistää empty()-operaattorin. Esimerkiksi:

```
if ( svec.empty() != true )
      ; // hups, jotain vinossa
```

Elementin lisäyksen yksinkertaisin menetelmä on push_back(), joka lisää elementin säilön loppuun. Esimerkiksi

3. Toteutukset, jotka eivät nykyisellään tue malliparametreja, vaativat toisen argumentin, joka määrittää *allocator*-luokan. Näissä toteutuksissa edellä olevat kaksi määrittelyä esitellään seuraavasti:

```
vector< string, allocator > svec;
list< int, allocator >      ilist;
```

Abstraktio muistin dynaamisesta varaamisesta ja vapauttamisesta kapseloidaan Allocator-luokkaan. Se on esimääritelty vakiokirjastoon ja käyttää new- ja delete-operaattoreita. Allocator-luokka toimii kahdessa eri tarkoituksessa: suojaa säiliöitä muistinvarausstrategian yksityiskohdilta, yksinkertaistaa säiliön toteutusta ja saa mahdolliseksi sen, että ohjelmoija voi toteuttaa ja/tai määritellä vaihtoehtoisia muistinvarausstrategioita kuten jaettu muistinkäyttö.

```
string text_word;
while ( cin >> text_word )
        svec.push_back( text_word );
```

lukee yhden merkkijonon kerrallaan text_word-olioon vakiosyötöstä. Sitten push_back() lisää kopion text_word-merkkijonosta svec-vektoriin. Listasäiliö tukee myös menetelmää push_front(), joka lisää uuden elementin listan alkuun. Oletetaan esimerkiksi, että meillä on seuraava sisäinen int-tyyppinen taulukko:

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

Kun käytetään push_back()-menetelmää

```
for ( int ix = 0; ix < 4; ++ix )
        ilist.push_back( ia[ ix ] );
```

se luo jonon 0,1,2,3, kun taas menetelmä push_front()

```
for ( int ix = 0; ix < 4; ++ix )
        ilist.push_front( ia[ ix ] );
```

se luo jonon 3,2,1,0 listaan ilist[4].

Voimme vaihtoehtoisesti määritellä säiliölle eksplisiittisen koon. Koko voi olla joko vakio tai muuttujalauseke:

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

extern int get_word_count( string file_name );
const int list_size = 64;

list< int > ilist( list_size );
vector< string > svec(get_word_count(string("Chimera")));
```

Jokainen säiliön elementti alustetaan tyyppinsä mukaisella oletusarvolla. Kokonaisluvuille oletusarvo on 0, jolla jokainen elementti alustetaan. String-luokan elementit alustetaan oletusmuodostajalla.

Sen sijaan, että alustaisimme jokaisen elementin vastaavalla oletusarvollaan, voimme määrittää arvon jokaisen elementin alustamiseksi. Esimerkiksi:

```
list< int > ilist( list_size, -1 );
vector< string > svec( 24, "pooh" );
```

Alkuarvon antamisen lisäksi voimme muuttaa säiliön kokoa uudelleen resize()-operaatiolla. Kun esimerkiksi kirjoitamme

```
svec.resize( 2 * svec.size() );
```

4. Jos push_front()-funktion käyttö on hallitsevaa, pakka toimii merkittävästi tehokkaammin kuin vektori.

kaksinkertaistamme svec-vektorin nykyisen koon. Jokainen uusi elementti alustetaan vastaavalla tyyppinsä mukaisella oletusarvollaan. Jos haluamme alustaa jokaisen uuden elementin jollain toisella arvolla, voimme määrittää tuon arvon toisena argumenttina:

```
// alusta jokainen uusi elementti merkkijonolla "piglet"
svec.resize( 2 * svec.size(), "piglet" );
```

Muuten, mikä on svec-vektorin alkuperäisen määrittelyn kapasiteetti? Sen alkuperäinen koko on 24 elementtiä. Mikä on se todennäköinen alkukapasiteetti? Oikein — svec-vektorin kapasiteetti on myös 24. Yleensä vektorin vähimmäiskapasiteetti on sen nykyinen koko. Kun tuplaamme vektorin koon, yleensä tuplaamme sen kapasiteetin.

Voimme myös alustaa uuden säiliöolion olemassaolevan säiliöolion kopiolla. Esimerkiksi:

```
vector< string > svec2( svec );
list< int >      ilist2( ilist );
```

Jokainen säiliö tukee vertailuoperaattoreita, joita vastaan säiliöitä voidaan verrata: yhtäsuuruus, erisuuruus, pienempi kuin, suurempi kuin, pienempi tai yhtäsuuri kuin ja suurempi tai yhtäsuuri. Vertailu tapahtuu kahden säiliön elementeille pareittain. Jos kaikki elementit ovat yhtäsuuria ja molemmat säiliöt sisältävät yhtä monta elementtiä, ovat säiliöt yhtäsuuria; muussa tapauksessa ne ovat erisuuria. Ensimmäinen erisuuri elementti päättää kahden säiliön pienempi kuin- tai suurempi kuin -suhteesta. Esimerkiksi tässä on tulostus ohjelmasta, joka vertailee viittä vektoria:

```
ivec1: 1 3 5 7 9 12
ivec2: 0 1 1 2 3 5 8 13
ivec3: 1 3 9
ivec4: 1 3 5 7
ivec5: 2 4

// ensimmäinen erisuuri elementti: 1, 0
// ivec1 on suurempi kuin ivec2
ivec1 < ivec2 // epätosi
ivec2 < ivec1 // tosi

// ensimmäinen erisuuri elementti 5, 9
ivec1 < ivec3 // tosi

// kaikki elementit yhtäsuuria, mutta vektorissa ivec4 on vähemmän elementtejä,
// joten ivec4 on pienempi kuin ivec1
ivec1 < ivec4 // epätosi

// ensimmäinen erisuuri elementti: 1, 2
ivec1 < ivec5 // tosi

ivec1 == ivec1 // tosi
ivec1 == ivec4 // epätosi
ivec1 != ivec4 // tosi
```

```
ivec1 > ivec2 // tosi
ivec3 > ivec1 // tosi
ivec5 > ivec2 // tosi
```

Säiliöiden tyypeille, joita voimme määritellä, on olemassa kolme rajoitusta (käytännössä ne liittyvät ainoastaan käyttäjän määrittelemiin luokkatyyppeihin).

- Elementin tyypin pitää tukea yhtäsuuruusoperaattoria.
- Elementin tyypin pitää tukea pienempi kuin -operaattoria (kaikki aikaisemmin käsittelemämme vertailuoperaattorit toteutetaan käyttäen näitä kahta operaattoria).
- Elementin tyypin pitää tukea oletusarvoa (jälleen: luokkatyypin yhteydessä puhutaan oletusmuodostajasta).

Kaikki esimääritellyt tietotyypit mukaan lukien osoittimet täyttävät nämä ehdot kuten myös kaikki C++-vakiokirjaston luokkatyypit.

Harjoitus 6.5

Selitä, mitä seuraava ohjelma tekee:

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

int main()
{
    vector<string> svec;
    svec.reserve( 1024 );

    string text_word;
    while ( cin >> text_word )
          svec.push_back( text_word );

    svec.resize( svec.size()+svec.size()/2 );
    // ...
}
```

Harjoitus 6.6

Voiko säiliön kapasiteetti olla pienempi kuin sen koko? Onko toivottavaa, että sen koko on yhtä kuin kapasiteetti? Alussa? Elementin lisäämisen jälkeen? Miksi tai miksi ei?

Harjoitus 6.7

Jos harjoituksessa 6.5 ohjelma lukee 256 sanaa, mikä on sen todennäköinen kapasiteetti koon uudelleenmuodostumisen jälkeen? Entä sitten, jos se lukee 512? 1000? 1048?

Harjoitus 6.8

Jos on tehty seuraavat luokkamääritykset, mitä niistä ei voi käyttää vektorin määrittelyyn?

(a)
```
class cl1 {
  public:
    cl1( int=0 );
    bool operator==();
    bool operator!=();
    bool operator<=();
    bool operator<();
        // ...
};
```

(b)
```
class cl2 {
    public:
        cl2( int=0 );
        bool operator!=();
        bool operator<=();
        // ...
    };
```

(c)
```
class cl3 {
  public:
    int ival;
};
```

(d)
```
class cl4 {
    public:
        cl4( int, int=0 );
        bool operator==();
        bool operator==();
        // ...
    };
```

6.5 Iteraattorit

Iteraattori tarjoaa yleisen menetelmän jokaisen elementin peräkkäinkäsittelylle mille tahansa jono- tai assosiatiiviselle säiliötyypille. Olkoon esimerkiksi iter iteraattori mille tahansa säiliötyypille. Silloin

```
++iter;
```

vie iteraattoria eteenpäin säiliön seuraavan elementin osoitteeseen ja

```
*iter;
```

palauttaa iteraattorin osoittaman elementin arvon.

Jokaisessa säiliötyypissä on sekä begin()- että end()-jäsenfunktio.

- begin() palauttaa iteraattorin, joka osoittaa säiliön ensimmäiseen elementtiin.
- end() palauttaa iteraattorin, joka osoittaa yhden yli säiliön viimeisestä elementistä.

Käydäksemme läpi minkä tahansa säiliötyypin elementit, kirjoitamme

```
for ( iter = container.begin();
    iter != container.end(); ++iter )
        do_something_with_element( *iter );
```

Iteraattorin esittely voi näyttää hieman pelottavalta mallin ja sisäkkäisen luokkasyntaksin takia. Tässä on esimerkiksi iteraattoriparin esittely vektorille, joka sisältää merkkijonoelementtejä:

```
// vector<string> vec;
vector<string>::iterator iter = vec.begin();
vector<string>::iterator iter_end = vec.end();
```

iterator on typedef-nimi, joka on määritelty vektoriluokassa. Syntaksi

```
vector<string>::iterator
```

viittaa vektoriluokan iterator-typedef-nimeen, joka pitää sisällään merkkijonotyyppisiä elementtejä.

Jos haluamme tulostaa jokaisen merkkijonoelementin vakiotulostukseen, kirjoitamme

```
for( ; iter != iter_end; ++iter )
   cout << *iter << '\n';
```

jossa *iter saa tietysti todellisen merkkijono-olion arvon.

Jokainen säiliö määrittelee *iterator*-tyypin lisäksi *const_iterator*-tyypin. Jälkimmäinen on tarpeellinen, kun halutaan käydä läpi säiliö, joka on const. Const_iterator sallii vain-luku-käsittelyn säiliön elementeille. Esimerkiksi:

```
#include <vector >
void even_odd( const vector<int> *pvec,
                        vector<int> *pvec_even,
                        vector<int> *pvec_odd )
{
```

```
    // pitää määritellä const_iterator pvec-vektorin läpikäyntiä varten
    vector<int>::const_iterator c_iter = pvec->begin();
    vector<int>::const_iterator c_iter_end = pvec->end();

    for ( ; c_iter != c_iter_end; ++c_iter )
            if ( *c_iter % 2 )
                pvec_even->push_back( *c_iter );
            else pvec_odd->push_back( *c_iter );
}
```

Mitä, jos lopuksi haluamme katsoa joitakin elementtien alijoukkoja — ehkä joka toista tai kolmatta elementtiä — tai aloittaa elementtien läpikäynnin keskeltä? Voimme siirtyä iteraattorin nykyisestä positiosta käyttämällä skalaarista aritmetiikkaa. Esimerkiksi:

```
vector<int>::iterator iter = vec->begin()+vec.size()/2;
```

asettaa iter-iteraattorin osoittamaan vec-vektorin keskimmäiseen elementtiin, kun taas

```
iter += 2;
```

vie iter-iteraattoria kaksi elementtiä eteenpäin.

Iteraattoriaritmetiikka toimii ainoastaan vektorin ja pakan yhteydessä eikä listan, koska listan elementtejä ei ole tallennettu yhtenäisesti muistiin. Esimerkiksi

```
ilist.begin() + 2;
```

ei ole oikein, koska kahden elementin eteneminen listassa vaatii sisäisen *next*-osoittimen käyttämisen kahdesti. Vektorin ja pakan yhteydessä kahden elementin eteneminen vaatii kahden elementin koon lisäämisen nykyiseen osoitearvoon (kohdassa 3.3 on käsitelty osoitinaritmetiikkaa).

Säiliöolio voidaan alustaa myös iteraattoriparilla, joka merkitsee ensimmäisen ja yhden yli viimeisen kopioitavan elementin. Oletetaan esimerkiksi, että meillä on

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

int main()
{
    vector<string> svec;
    string intext;

    while ( cin >> intext )
          svec.push_back( intext );

    // käsittele svec ...
}
```

Voimme määritellä uuden vektorin, johon kopioidaan kaikki tai osa svec-vektorin elementeistä:

```
int main()
{
    vector<string> svec;
    // ...

    // alusta svec2 koko svec-vektorilla
    vector<string> svec2( svec.begin(), svec.end() );

    // alusta svec3 svec-vektorin ensimmäisellä puoliskolla
    vector<string>::iterator it =
                            svec.begin() + svec.size()/2;
    vector<string>  svec3( svec.begin(), it );

    // käsittele vektorit ...

}
```

Kun käytämme erityistä istream_iterator-tyyppiä (käsitellään yksityiskohtaisesti kohdassa 12.4.3), voimme lisätä suoremmin tekstiä svec-vektoriin:

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

int main()
{
    // syötevirran iteraattori sidottu vakiosyötteeseen
    istream_iterator<string> infile( cin );

    // syötevirran iteraattori ilmaisemassa sen loppua (end-of-stream)
    istream_iterator<string> eos;

    // alusta svec arvoilla, jotka on annettu cin:in kautta;
    vector<string> svec( infile, eos );

    // käsittele svec
}
```

Iteraattoriparin lisäksi voidaan käyttää kahta osoitinta sisäiseen taulukkoon elementtien alueen merkitsemiseen. Oletetaan esimerkiksi, että meillä on seuraava string-olioiden taulukko:

```
#include <string>
string words[4] = {
    "stately", "plump", "buck", "mulligan"
};
```

Voimme alustaa merkkijonovektorin välittämällä osoittimen words-taulukon ensimmäiseen elementtiin ja toisen osoittimen *yhden yli* viimeisen elementin:

```
vector< string > vwords( words, words+4 );
```

Toinen osoitin toimii pysäyttävänä ehtona. Oliota, jota se osoittaa (usein yhden yli säiliön tai taulukon viimeisen olion yli), ei sisällytetä kopioitaviin tai läpikäytäviin elementteihin.

Samalla tavalla voimme alustaa int-elementtien listan kuten seuraavassa:

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

Kohdassa 12.4 katsomme iteraattoreita vielä hieman yksityiskohtaisemmin. Tässä vaiheessa olemme tuoneet iteraattoreita tarpeeksi esille, jotta voimme käyttää niitä tekstinkyselyjärjestelmämme toteutuksessa. Mutta ennen kuin voimme palata siihen, meidän pitää katsoa uudelleen joitakin lisäoperaattoreita, joita säiliötyypit tukevat.

Harjoitus 6.9

Mitkä seuraavista iteraattoreista ovat virheellisiä, vai onko yksikään?

```
const vector< int > ivec;
vector< string >    svec;
list< int >         ilist;

(a) vector<int>::iterator    it = ivec.begin();
(b) list<int>::iterator      it = ilist.begin()+2;
(c) vector<string>::iterator it = &svec[0];
(d) for ( vector<string>::iterator
                it = svec.begin(); it != 0; ++it )
          // ...
```

Harjoitus 6.10

Mitkä seuraavista iteraattoreiden käyttötavoista ovat virheellisiä, vai onko yksikään?

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
   "Fort Sumter", "Manassas", "Perryville", "Vicksburg",
   "Meridian", "Chancellorsville" };

(a) vector<string> svec( sa, &sa[6] );
(b) list<int> ilist(  ia+4, ia+6 );
(c) list<int> ilist2( ilist.begin(), ilist.begin()+2 );
(d) vector<int> ivec( &ia[0], ia+8 );
(e) list<string> slist( sa+6, sa );
(f) vector<string> svec2( sa, sa+6 );
```

6.6 Jonosäiliöiden operaatioita

push_back()-menetelmä on kätevä ja lyhyt ilmaisu yksittäisen elementin lisäämiselle jonosäiliön loppuun. Mutta mitä, jos haluamme lisätä elementin säiliöön johonkin muuhun kohtaan? Tai jos haluamme lisätä elementtijonon säiliön loppuun tai johonkin muuhun kohtaan? Näissä tapauksissa käytämme yleisempiä lisäysmenetelmiä.

Kun haluamme lisätä elementin esimerkiksi säiliön alkuun, kirjoitamme seuraavasti:

```
vector< string > svec;
list< string > slist;
string spouse( "Beth" );

slist.insert( slist.begin(), spouse );
svec.insert(  svec.begin(), spouse  );
```

Tässä insert()-funktion ensimmäinen argumentti on positio (iteraattori, joka osoittaa johonkin säiliön kohtaan) ja toinen argumentti on lisättävä arvo. Arvo lisätään iteraattorin osoittaman kohdan eteen. Hajakäsittelyä enemmän muistuttava lisäys ohjelmoitaisiin seuraavasti:

```
string son( "Danny" );

list<string>::iterator iter;
iter = find( slist.begin(), slist.end(), son );

slist.insert( iter, spouse );
```

Tässä find() joko palauttaa position säiliöön, josta elementti löytyi, tai palauttaa säiliön end()-iteraattorin, joka ilmaisee, että etsintä epäonnistui. (Palaamme find()-funktioon seuraavan kohdan lopussa.) Kuten olet saattanut arvata, push_back()-menetelmä on lyhennetty ilmaisutapa seuraavalle kutsulle:

```
// samanarvoinen kuin: slist.push_back( value );
slist.insert( slist.end(), value );
```

Toinen insert()-metodin muoto tukee tietyn elementtimäärän lisäystä johonkin kohtaan. Jos esimerkiksi haluamme lisätä kymmenen *Anna*a vektorin alkuun, teemme seuraavasti:

```
vector<string> svec;
...
string anna( "Anna" );
svec.insert( svec.begin(), 10, anna );
```

Viimeinen insert()-metodin muoto tukee elementtialueen lisäystä säiliöön. Jos esimerkiksi on annettu seuraava merkkijonotaulukko

```
string sarray[4] = { "quasi", "simba", "frollo", "scar" };
```

voimme lisätä kaikki tai osan elementeistä merkkijonovektoriimme:

```
svec.insert( svec.begin(), sarray, sarray+4 );
svec.insert( svec.begin() + svec.size()/2,
        sarray+2, sarray+4);
```

Voimme vaihtoehtoisesti merkitä lisättävän arvoalueen iteraattoriparilla kummasta tahansa merkkijonosta:

```
// lisää svec:in sisältämät elementit
// alkaen svec_two:n keskeltä
svec_two.insert( svec_two.begin() + svec_two.size()/2,
                        svec.begin(), svec.end() );
```

tai yleisemmin minkä tahansa merkkijono-olioiden säiliön[5]:

```
list< string > slist;

// ...

// lisää svec:in sisältämät elementit
// stringVal:in sijaintipaikan eteen slist-listassa
list< string >::iterator iter =
      find( slist.begin(), slist.end(), stringVal );
slist.insert( iter, svec.begin(), svec.end() );
```

6.6.1 Poistaminen

Säiliön elementin poistamisen yleinen muoto on erase()-metodipari: toinen poistaa yhden elementin ja toinen iteraattoriparin merkitsemän elementtialueen. Pikamenetelmä säiliön viimeisen elementin poistamiselle on pop_back()-metodi.

Kun haluat esimerkiksi poistaa elementin säiliöstä, käynnistät yksinkertaisesti erase()-metodin iteraattorilla, joka ilmaisee elementin position. Seuraavassa koodikatkelmassa käytämme geneeristä find()-algoritmia iteraattorin hakemiseen elementille, jonka haluamme poistaa, ja jos elementti löytyy listasta, välitämme sen position erase()-metodille.

```
string searchValue( "Quasimodo" );
list< string >::iterator iter =
    find( slist.begin(), slist.end(), searchValue );

if ( iter != slist.end() )
   slist.erase( iter );
```

Kun haluamme poistaa säiliön kaikki elementit tai iteraattoriparin merkitsemän alijoukon, voimme tehdä sen seuraavasti:

```
// poista säiliön kaikki elementit
```

5. Tämä viimeinen muoto vaatii, että kääntäjäsi tukee jäsenfunktiomallia. Ellei kääntäjäsi vielä tue tätä C++-standardia, silloin säiliön olioiden pitää olla samaa tyyppiä, kuten kaksi vektoria tai kaksi listaa, jotka sisältävät samaa elementtityyppiä.

```
slist.erase( slist.begin(), slist.end() );

// poista iteraattoreiden merkitsemä alue
list< string >::iterator first, last;

first = find( slist.begin(), slist.end(), val1 );
last  = find( slist.begin(), slist.end(), val2 );

// ... tarkista ensimmäisen ja viimeisen kelvollisuus

slist.erase( first, last );
```

Lopuksi täydennämme push_back()-metodia, joka lisää elementin säiliön loppuun, pop_back()-metodilla, joka poistaa säiliön viimeisen elementin — se ei kuitenkaan palauta elementtiä; se yksinkertaisesti poistaa sen. Esimerkiksi:

```
vector< string >::iterator iter = buffer.begin();
for ( ; iter != buffer.end(), iter++ )
{
        slist.push_back( *iter );
        if ( ! do_something( slist ))
                slist.pop_back();
}
```

6.6.2 Sijoitus ja vaihtaminen keskenään

Mitä tapahtuu, kun sijoitamme säiliön toiseen? Sijoitusoperaattori kopioi oikeanpuoleisen säiliöolion elementit vasemmanpuoleiseen säiliöön. Mitä sitten, jos säiliöt ovat erikokoisia? Esimerkiksi:

```
// svec1 sisältää 10 elementtiä
// svec2 sisältää 24 elementtiä
// sijoituksen jälkeen molemmat sisältävät 24 elementtiä
svec1 = svec2;
```

Sijoituksen kohde, joka on esimerkissämme svec1, sisältää nyt saman määrän elementtejä kuin säiliö, josta elementit kopioitiin (esimerkissämme svec2). svec1-vektorin aikaisemmat elementit poistetaan. (Vektorin svec1 tapauksessa string-tuhoajaa käytetään jokaiselle kymmenelle merkkijonoelementille.)

swap()-metodin voidaan ajatella olevan täydennyksenä sijoitusoperaattorille. Kun kirjoitamme

```
slist1.swap( slist2 );
```

slist1 sisältää nyt 24 merkkijonoelementtiä, jotka kopioitiin käyttäen string-sijoitusoperaattoria, mikä on sama, kuin jos olisimme kirjoittaneet

```
slist1 = slist2;
```

Erona on, että slist2 sisältää nyt kopion kymmenestä elementistä, jotka alun perin sijaitsivat listassa slist1. Jälleen: jos säiliöiden koot eivät ole samoja, kopioinnin kohteena olevan säiliön kokoa muutetaan lähteenä olevan säiliön mukaiseksi.

6.6.3 Geneeriset algoritmit

Ne operaatiot, joita kuvattiin edellisissä kohdissa, ovat itse asiassa kaikki ne operaatiot, jotka vektori ja pakka voivat tarjota. Myönnettäköön, että se on aika ohut rajapinta ja siitä puuttuu perusoperaatioita kuten find(), sort(), merge() jne. Käsitteellisesti ajatus on jakaa säiliöille yleiset operaatiot geneerisiksi algoritmeiksi, joita voidaan käyttää kaikille säiliötyypeille, kuten myös sisäänrakennetuille taulukkotyypeille. (Geneerisiä algoritmeja käsitellään tarkemmin luvussa 12 ja tämän kirjan liitteessä.) Geneeriset algoritmit on sidottu tiettyyn säiliöön iteraattoriparin kautta. Seuraavassa on esimerkki, kuinka käynnistämme geneerisen find()-algoritmin listalle, vektorille ja taulukolle, jotka ovat kaikki eri tyyppisiä:

```
#include <list>
#include <vector>

int ia[ 6 ] = { 0, 1, 2, 3, 4, 5 };
vector<string> svec;
list<double> dlist;

// tähän liittyvä otsikkotiedosto
#include <algorithm>

vector<string>::iterator viter;
list<double>::iterator liter;
int *pia;

// find() palauttaa iteraattorin elementtiin, jos se löytyi
// taulukon yhteydessä se palauttaa osoittimen ...
pia =   find( &ia[0], &ia[6], some_int_value );
liter = find( dlist.begin(), dlist.end(), some_double_value );
viter = find( svec.begin(),  svec.end(),  some_string_value );
```

Listan säiliötyypillä on lisäoperaatioita, kuten merge() ja sort(), koska se ei tue elementeilleen hajakäsittelyä. Näistä keskustellaan kohdassa 12.6. Palatkaamme nyt tekstinkyselyjärjestelmäämme.

Harjoitus 6.11

Kirjoita ohjelma, joka hyväksyy seuraavat määrittelyt:

```
int ia[]  = { 1, 5, 34 };
int ia2[] = { 1, 2, 3 };
int ia3[] = { 6, 13, 21, 29, 38, 55, 67, 89 };
vector<int> ivec;
```

Käytä useita lisäysoperaatioita ja sopivia arvoja taulukoista ia2 ja ia3 sekä muokkaa ivec-vektoria niin, että se sisältää jonon

```
{ 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 }
```

Harjoitus 6.12

Kirjoita ohjelma, joka hyväksyy seuraavat määrittelyt:

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
list<int> ilist( ia, ia+11 );
```

Käytä yhden iteraattorin muotoa erase()-metodista ja poista kaikki ilist-listan sisältämät parittomasti numeroidut elementit.

6.7 Tekstirivien tallentaminen

Ensimmäinen tehtävämme on lukea tekstitiedosto, jota käyttäjämme haluavat kysellä. Meidän täytyy hakea seuraava tieto: jokainen sana tietysti, mutta lisäksi jokaisen sanan sijaintipaikka — eli rivi ja sanan positio tuolla rivillä. Lisäksi meidän pitää tallentaa tekstirivi rivinumeron kera näyttääksemme rivit, jotka vastaavat kyselyä.

Kuinka haemme tekstin jokaisen rivin? Vakiokirjasto tukee getline()-funktiota, joka esitellään seuraavasti:

```
istream&
getline( istream &is, string str, char delimiter );
```

getline() lukee istream-syöttövirtaa ja lisää merkit (tyhjät merkit mukaan lukien) string-olioon, kunnes kohdataan joko erotin tai tiedoston loppu, tai kunnes luetun merkkijonon pituus on yhtä kuin string-olion max_size()-arvo, jolloin lukuoperaatio epäonnistuu.

Jokaisen getline()-kutsun jälkeen lisäämme str-merkkijonon string-tyyppiseen vektoriin, joka edustaa tekstiä. Tässä on yleinen toteutus[6].

6. Se on käännetty toteutuksessa, joka ei tue oletusmallin parametriarvoja, joten meiltä vaaditaan eksplisiittisesti varaajaa (allocator):

```
vector< string, allocator > *lines_of_text;
```

Täysin yhteensopivassa C++-standardin toteutuksessa meidän tarvitsee vain määrittää elementin tyyppi:

```
vector< string > *lines_of_text;
```

Olemme laittaneet sen funktioon nimeltään retrieve_text(). Lisätäksemme kerättyä informaatiota, olemme määritelleet arvoparin, johon tallennetaan rivinumero ja pisimmän rivin pituus. (Koko ohjelma on listattuna kohdassa 6.14.)

```
// palautusarvo on osoitin string-vektoriimme
vector<string,allocator>*
retrieve_text()
{
    string file_name;

    cout << "please enter file name: ";
    cin  >> file_name;

    // avaa tekstitiedosto lukemista varten ...
    ifstream infile( file_name.c_str(), ios::in );
    if ( ! infile ) {
        cerr << "oops! unable to open file "
             << file_name << " -- bailing out!\n";
        exit( -1 );
    }
    else cout << '\n';

    vector<string, allocator> *lines_of_text =
                    new vector<string,allocator>;
    string textline;

    typedef pair<string::size_type, int> stats;
    stats maxline;
    int   linenum = 0;

    while ( getline( infile, textline, '\n' )) {
            cout << "line read: " << textline << '\n';

              if ( maxline.first < textline.size() ) {
                    maxline.first = textline.size();
              maxline.second = linenum;
              }

            lines_of_text->push_back( textline );
          linenum++;
    }

    return lines_of_text;
}
```

Ohjelman tulostus näyttää seuraavalta. (Valitettavasti rivit taittuvat tekstisivun koon takia. Olemme sisentäneet käsin toista riviä parantaaksemme luettavuutta.)

```
please enter file name: alice_emma

line read: Alice Emma has long flowing red hair. Her Daddy says
line read: when the wind blows through her hair, it looks almost
      alive,
line read: like a fiery bird in flight. A beautiful fiery bird, he
      tells her,
line read: magical but untamed. "Daddy, shush, there is no such
      thing,"
line read: she tells him, at the same time wanting him to tell her
      more.
line read: Shyly, she asks, "I mean, Daddy, is there?"

number of lines: 6
maximum length: 66
longest line: like a fiery bird in flight. A beautiful fiery bird,
       he tells her,
```

Nyt, kun jokainen tekstirivi on tallennettu merkkijonona, jokainen rivi pitää jakaa yksittäisiksi sanoiksi. Poistamme ensiksi jokaisesta sanasta välimerkit. Mietitäänpä esimerkiksi tätä seuraavaa riviä *Finnegans Wake*n kappaleesta *Anna Livia Plurrabelle*.

```
"For every tale there's a telling,
and that's the he and she of it."
```

Se muodostuu seuraavista yksittäisistä merkkijonoista, joihon on upotettu välimerkkejä:

```
"For
there's
telling,
that's
it."
```

Näistä merkkijonoista pitäisi tulla

```
For
there
telling
that
it
```

Joku voisi väittää, että sanasta

```
there's
```

pitäisi muodostua

```
there is
```

mutta olemme itse asiassa menossa toiseen suuntaan: jätämme huomiotta semanttisesti neutraalit sanat kuten *is*, *that*, *and*, *it*, *the* jne. Joten kyselyymme tulevat *Finnegans Wake*sta mukaan vain seuraavat aktiiviset sanat:

```
tale
telling
```

(Toteutamme tämän käyttäen poisjäävien sanojen joukkoa, jota käsittelemme yksityiskohtaisesti myöhemmin joukkosäiliötyypin (set) yhteydessä.)

Välimerkkien poistamisen lisäksi täytyy poistaa isot kirjaimet ja tehdä joitakin pieniä muutoksia loppuliitteille. Isoista kirjaimista tulee ongelma kuten seuraavissa tekstiriveissä:

```
Home is where the heart is.
A home is where they have to let you in.
```

Selvää on, että kyselyn, jossa etsitään sanaa home, pitää löytää molemmat kohdat.

Loppuliitteet muodostavat havaitsemisen monimutkaisemman ongelman kuten sen, että *dog* ja *dogs* edustavat samaa substantiivia ja että *love*, *loves*, *loving* ja *loved* edustavat samaa verbiä.

Meidän tarkoituksemme on seuraavissa kappaleissa vierailla uudelleen vakiokirjaston string-luokassa ja kokeilla sen laajaa kokoelmaa merkkijonokäsittelyn operaatioista. Tämän matkan aikana kehitämme edelleen tekstinkyselyjärjestelmäämme.

6.8 Alimerkkijonon löytäminen

Ensimmäinen tehtävämme on erottaa tekstiriviä edustava merkkijono yksittäisiin sanoihinsa. Teemme sen etsimällä jokaisen upotetun tyhjän merkin. Jos esimerkiksi on olemassa rivi

```
Alice Emma has long flowing red hair.
```

ja kun merkitsemme kuusi tyhjää merkkiä, voimme tunnistaa seitsemän alimerkkijonoa, jotka edustavat tekstiriviä. Tämän tehdäksemme käytämme joitain string-luokan tukemia find()-funktioita.

String-luokassa on joukko etsintäfunktioita, joista jokainen on nimetty *find*-sanan muunnoksella. find() on suoraviivaisin esiintymä: kun sille annetaan merkkijono, se palauttaa joko indeksin position ensimmäisestä merkkijonoa vastaavasta alimerkkijonosta tai erityisen arvon

```
string::npos
```

joka ilmaisee, että vastaavuutta ei löytynyt. Esimerkiksi:

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

int main() {
    string name( "AnnaBelle" );
    int pos = name.find( "Anna" );
    if ( pos == string::npos )
            cout << "Anna not found!\n";
    else cout << "Anna found at pos: " << pos << endl;
}
```

Vaikka palautetun indeksin tyyppi on melkein aina int, siirrettävämpi ja oikeampi esittely

käyttää tyyppiä

```
string::size_type
```

find()-funktion palauttamalle indeksiarvolle. Esimerkiksi:

```
string::size_type pos = name.find( “Anna” );
```

find()-funktiolla ei ole täsmälleen tarvitsemaamme toimintoa; kuitenkin find_first_of()-funktiolla on. find_first_of() palauttaa indeksin position sen merkkijonon ensimmäiseen merkkiin, joka sisältää minkä tahansa merkin etsitystä merkkijonosta. Esimerkiksi seuraava paikantaa ensimmäisen numeerisen arvon merkkijonosta:

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

int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );

    string::size_type pos = name.find_first_of( numerics );
    cout << "found numeric at index: "
         << pos      << "\telement is "
         << name[pos] << endl;
}
```

Tässä esimerkissä pos asetetaan arvoon 1 (muista, että merkkijonon elementit on indeksoitu alkaen arvosta 0).

Tämä ei kuitenkaan tee aivan sitä, mitä haluaisimme. Meidän pitää löytää kaikki esiintymät eikä vain ensimmäistä. Voimme tehdä sen antamalla toisen argumentin, joka ilmaisee, mistä merkkijonon indeksin positiosta etsintä aloitetaan. Tässä on uudelleen kirjoittamamme merkkijonon "r2d2" etsintä. Silti se ei ole vieläkään aivan oikein. Näetkö, mikä siinä on väärin?

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

int main() {
    string numerics( "0123456789" );
    string name( "r2d2" );

    string::size_type pos = 0;

    // jotain väärin toteutuksessa!
    while (( pos = name.find_first_of( numerics, pos ))
            != string::npos )
                cout << "found numeric at index: "
                     << pos      << "\telement is "
                     << name[pos] << endl;
}
```

pos aloittaa silmukan alkuarvolla 0. Merkkijonosta aletaan etsiä alkaen positiosta 0. Löy-

tyminen tapahtuu indeksillä 1. pos saa arvokseen sijoituksen tuloksena tuon arvon. Koska se ei ole yhtäsuuri kuin npos, silmukan runko suoritetaan. Toinen find_first_of() suoritetaan pos-arvolla 1. Hups! Positio 1 täsmää toisella, kolmannella ja neljännellä kerralla jne.: olemme ohjelmoineet itsemme ikuiseen silmukkaan. Meidän pitää kasvattaa pos-arvoa yhden yli löytyneen elementin ennen jokaista seuraavaa silmukan toistoa:

```
// ok: korjattu silmukkatoisto
while (( pos = name.find_first_of( numerics, pos ))
        != string::npos )
{
        cout << "found numeric at index: "
                << pos      << "\telement is "
                << name[pos] << endl;

        // siirrä yksi yli löytyneen elementin
        ++pos;
}
```

Jotta löytäisimme tyhjät merkit tekstirivistämme, korvaamme yksinkertaisesti argumentin numerics merkkijonolla, joka sisältää tyhjät merkit, joita saatamme kohdata. Jos kuitenkin olemme varmoja, että on käytetty vain välilyöntejä, voimme antaa eksplisiittisesti yhden merkin. Esimerkiksi:

```
// koodikatkelma
while (( pos = textline.find_first_of( ' ', pos ))
        != string::npos )
            // ...
```

Kun merkitsemme jokaisen sanan pituuden, esittelemme toisen paikkasidonnaisen olion kuten seuraavassa:

```
// koodikatkelma

// pos: indeksi yhden yli sanan
// prev_pos: indeksi sanan alkuun

string::size_type pos = 0, prev_pos = 0;

while (( pos = textline.find_first_of( ' ', pos ))
        != string::npos )
{
            // tee jotain merkkijonolle
            // säädä nyt position merkitsijöitä
            prev_pos = ++pos;
}
```

Jokaisella silmukkamme toistokerralla prev_pos indeksoi sanan alkua ja pos sisältää indeksin *yhden yli* sanan (välilyönnin positio). Jokaisen tunnistetun sanan pituus on siten ilmaistu lausekkeella

```
pos - prev_pos; // laskee sanan pituuden
```

Nyt, kun olemme tunnistaneet sanan, se pitää kopioida ja sitten työntää merkkijonovektoriin. Eräs sanan kopioinnin strategia on silmukoida textline läpi positiosta prev_pos arvoon yksi vähemmän kuin pos, ja kopioida jokainen merkki vuorollaan eli erotella kahden indeksin merkitsemä alimerkkijono. Sen sijaan, että tekisimme sen itse, substr()-merkkijono-operaatio tekee sen puolestamme:

```
// koodikatkelma
vector<string> words;

while (( pos = textline.find_first_of( ' ', pos ))
        != string::npos )
{
        words.push_back( textline.substr(
                prev_pos, pos-prev_pos));
        prev_pos = ++pos;
}
```

substr()-operaatio generoi alimerkkijonon kopion olemassaolevasta merkkijono-oliosta. Sen ensimmäinen argumentti ilmaisee aloituspositiota merkkijonossa. Valinnainen toinen argumentti ilmaisee alimerkkijonon pituutta (jos jätämme pois toisen argumentin, kopioidaan loppuosa merkkijonosta).

Toteutuksessamme on eräs virhe: se epäonnistuu jokaisen tekstirivin viimeisen sanan lisäyksessä. Näetkö, miksi? Mietitäänpä riviä

```
seaspawn and seawrack
```

Kaksi ensimmäistä sanaa on merkitty välilyönnillä. Näiden kahden välilyönnin positiot palautetaan vuorollaan kahdella find_first_of()-käynnistyksellä. Kolmas käynnistys ei kuitenkaan löydä välilyöntiä; se asettaa pos-arvon arvoon string::npos, joka lopettaa silmukan. Viimeisen sanan käsittelyn pitää tapahtua siten silmukan päätyttyä.

Tässä on koko toteutus lokalisoituna funktioon, jonka olemme nimenneet separate_words(). Sen lisäksi, että tallennamme jokaisen sanan merkkijonovektoriin, olemme laskeneet jokaisen sanan rivin ja sarakkeen position. (Tarvitsemme tätä tietoa myöhemmin, kun tuemme paikkasidonnaista tekstikyselyä.)

```
typedef pair<short,short> location;
typedef vector<location>  loc;
typedef vector<string>    text;
typedef pair<text*,loc*>  text_loc;

text_loc*
separate_words( const vector<string> *text_file )
{
   // words: sisältää yksittäisten sanojen kokoelman
   // locations: sisältää niihin liittyvät rivi/saraketiedot
   vector<string>   *words = new vector<string>;
   vector<location> *locations = new vector<location>;
```

```
    short line_pos = 0; // current line number

    // silmukoi läpi jokainen tekstirivi
    for ( ; line_pos < text_file->size(); ++line_pos )
    {
                // textline: nykyinen käsiteltävä tekstirivi
                // word_pos: nykyinen tekstirivin sarakepositio
                short  word_pos = 0;
                string textline = (*text_file)[ line_pos ];

                string::size_type pos = 0, prev_pos = 0;

                while (( pos = textline.find_first_of( ' ', pos ))
                        != string::npos )
       {
          // tallenna nykyisen sanan alimerkkijonon kopio
          words->push_back(
               textline.substr( prev_pos, pos - prev_pos ));

          // tallenna rivi/saraketieto parina
          locations->push_back(
               make_pair( line_pos, word_pos ));

          // tallenna positiotietoa seuraavaan toistoon
          ++word_pos; prev_pos = ++pos;
        }

        // käsittele nyt rivin viimeinen sana
        words->push_back(
             textline.substr( prev_pos, pos - prev_pos ));

        locations->push_back(
                make_pair( line_pos, word_pos ));

      }

    return new text_loc( words, locations );
}
```

Ohjelmamme kulku tähän mennessä on seuraava:

```
int main()
{
   vector<string> *text_file = retrieve_text();
   text_loc *text_locations  = separate_words( text_file );
   // ...
}
```

Osittainen tuloste separate_words() syötteestämme text_file näyttää tältä:

```
textline: Alice Emma has long flowing red hair. Her Daddy says

eol: 52 pos: 5 line: 0 word: 0 substring: Alice
eol: 52 pos: 10 line: 0 word: 1 substring: Emma
eol: 52 pos: 14 line: 0 word: 2 substring: has
eol: 52 pos: 19 line: 0 word: 3 substring: long
eol: 52 pos: 27 line: 0 word: 4 substring: flowing
eol: 52 pos: 31 line: 0 word: 5 substring: red
eol: 52 pos: 37 line: 0 word: 6 substring: hair.
eol: 52 pos: 41 line: 0 word: 7 substring: Her
eol: 52 pos: 47 line: 0 word: 8 substring: Daddy
last word on line substring: says

...

textline: magical but untamed. "Daddy, shush, there is no such thing,"
eol: 60 pos: 7 line: 3 word: 0 substring: magical
eol: 60 pos: 11 line: 3 word: 1 substring: but
eol: 60 pos: 20 line: 3 word: 2 substring: untamed.
eol: 60 pos: 28 line: 3 word: 3 substring: "Daddy,
eol: 60 pos: 35 line: 3 word: 4 substring: shush,
eol: 60 pos: 41 line: 3 word: 5 substring: there
eol: 60 pos: 44 line: 3 word: 6 substring: is
eol: 60 pos: 47 line: 3 word: 7 substring: no
eol: 60 pos: 52 line: 3 word: 8 substring: such
last word on line substring: thing,"

...

textline: Shyly, she asks, "I mean, Daddy, is there?"
eol: 43 pos: 6 line: 5 word: 0 substring: Shyly,
eol: 43 pos: 10 line: 5 word: 1 substring: she
eol: 43 pos: 16 line: 5 word: 2 substring: asks,
eol: 43 pos: 19 line: 5 word: 3 substring: "I
eol: 43 pos: 25 line: 5 word: 4 substring: mean,
eol: 43 pos: 32 line: 5 word: 5 substring: Daddy,
eol: 43 pos: 35 line: 5 word: 6 substring: is
last word on line substring: there?"
```

Ennen kuin lisäämme tekstinkyselyrutiinit, katsokaamme lyhyesti loput string-luokan tukemat etsintäfunktiot. `find()`- ja `find_first_of()`-etsintäfunktioiden lisäksi string-luokka tukee lukuisia muita etsintäoperaatioita: `rfind()` etsii viimeistä — oikeanpuoleisinta — ilmaistua alimerkkijonon esiintymää. Esimerkiksi:

```
string river( "Mississippi" );

string::size_type first_pos = river.find( "is" );
string::size_type last_pos = river.rfind( "is" );
```

find() palauttaa indeksin 1, joka ilmaisee ensimmäisen "is"-merkkijonon alkua, kun taas rfind() palauttaa indeksin 4, joka ilmaisee viimeisen "is"-merkkijonon alkua.

find_first_not_of() etsii merkkijonosta ensimmäisen merkin, joka ei täsmää yhteenkään etsittävän merkkijonon elementtiin. Kun etsimme esimerkiksi ensimmäisen muun kuin numeerisen merkin merkkijonosta, kirjoitamme

```
string elems( "0123456789" );
string dept_code( "03714p3" );

// palauttaa indeksin merkkiin 'p'
string::size_type pos = dept_code.find_first_not_of(elems);
```

find_last_of() etsii merkkijonosta viimeisen merkin, joka täsmää mihin tahansa etsittävän merkkijonon elementtiin. find_last_not_of() etsii merkkijonosta viimeisen merkin, joka ei täsmää yhteenkään etsittävän merkkijonon elementtiin. Jokainen näistä operaatioista voi saada valinnaisen argumentin, joka ilmaisee position, josta etsintä aloitetaan.

Harjoitus 6.13

Kirjoita ohjelma, joka etsii merkkijonosta

```
"ab2c3d7R4E6"
```

jokaisen numeerisen merkin ja sitten jokaisen aakkosmerkin käyttäen ensin operaatiota find_first_of() ja sitten find_first_not_of().

Harjoitus 6.14

Kirjoita ohjelma, jossa on merkkijono

```
string line1 = "We were her pride of 10 she named us --";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";

string sentence = line1 + line2 + line3;
```

ja joka laskee lauseen sanojen lukumäärän ja yksilöi suurimman ja pienimmän sanan. Jos useampi kuin yksi sana on joko suurempi tai pienempi, pidä lukua niistä.

6.9 Välimerkkien käsittely

Nyt, kun olemme erotelleet jokaisen rivin yksittäisiksi sanoiksi, pitää poistaa kaikki välimerkit, joita on voinut jäädä sanoihin. Esimerkiksi seuraava rivi

```
magical but untamed. "Daddy, shush, there is no such thing,"
```

erotellaan näin:

```
magical
but
untamed.
```

```
"Daddy,
shush,
there
is
no
such
thing,"
```

Miten voimme poistaa tarpeettomat välimerkit? Ensiksi määrittelemme merkkijonon, jossa on kaikki poistettavaksi haluamamme välimerkit:

```
string filt_elems( "\",.;:!?)(\\" );
```

(\" ja \\ ilmaisevat, että ensimmäistä lainausmerkkiä ja toista takakenoa kohdellaan literaalielementteinä lainatussa merkkijonossa eikä sen loppuna tai jatkomerkkinä seuraavalle riville.)

Seuraavaksi käytämme find_first_of()-operaatiota löytääksemme täsmäävän elementin merkkijonostamme, jos niitä on yhtään:

```
while (( pos = word.find_first_of( filt_elems, pos ))
        != string::npos )
```

Lopuksi pitää käyttää erase()-metodia merkkijonon välimerkille:

```
word.erase(pos,1);
```

Tämän erase()-operaatioversion ensimmäinen argumentti ilmaisee merkkijonosta position, josta merkkien poisto aloitetaan. Valinnainen toinen argumentti ilmaisee poistettavien merkkien lukumäärän. Esimerkissämme poistamme yhden merkin, joka sijaitsee positiossa pos. Jos jätämme pois toisen argumentin, erase() poistaa kaikki merkit positiosta pos merkkijonon loppuun saakka.

Tässä on koko listaus funktiosta filter_text(). Se saa kaksi argumenttia: osoittimen merkkijonovektoriimme, joka sisältää tekstin, ja string-olion, joka sisältää suodatettavat elementit.

```
void
filter_text( vector<string> *words, string filter )
{
    vector<string>::iterator iter = words->begin();
    vector<string>::iterator iter_end = words->end();

    // ellei käyttäjä anna suodatinta, laitetaan minimaalinen oletusjoukko
    if ( ! filter.size() )
        filter.insert( 0, "\"." );

    while ( iter != iter_end ) {
            string::size_type pos = 0;

            // poista jokainen löytynyt elementti
            while (( pos = (*iter).find_first_of( filter, pos ))
                    != string::npos )
                (*iter).erase(pos,1);
            iter++;
```

```
        }
    }
```

Näetkö, miksi emme kasvata pos-positiota jokaisella silmukan toistokerralla eli miksi seuraava on virheellinen?

```
while (( pos = (*iter).find_first_of( filter, pos ))
        != string::npos )
{
        (*iter).erase(pos,1);
        ++pos; // ei oikein ...
}
```

pos edustaa positiota merkkijonossa. Jos esimerkiksi on merkkijono

```
thing,"
```

ensimmäinen silmukan toistokerta sijoittaa positioon pos arvon 5, joka on pilkun positio. Sen jälkeen, kun poistamme pilkun, merkkijonosta tulee

```
thing"
```

Positiossa 5 on nyt lainausmerkki. Jos olisimme kasvattaneet pos-arvoa, emme olisi tunnistaneet emmekä poistaneet välimerkkiä.

Näin käynnistämme filter_text()-funktion pääohjelmastamme:

```
string filt_elems( "\",.;:!?)(\\/" );
filter_text( text_locations->first, filt_elems );
```

Ja lopuksi tässä on mallitulostus joistakin tekstimme merkkijonoista, joista on löytynyt yksi tai useampi suodatettava elementti:

```
filter_text: untamed.
found! : pos: 7.
after: untamed

filter_text: "Daddy,
found! : pos: 0"
after: Daddy,
found! : pos: 5,
after: Daddy

filter_text: thing,"
found! : pos: 5,
after: thing"
found! : pos: 5"
after: thing

filter_text: "I
found! : pos: 0"
after: I
```

```
filter_text: Daddy,
found! : pos: 5,
after: Daddy

filter_text: there?"
found! : pos: 5?
after: there"
found! : pos: 5"
after: there
```

Harjoitus 6.15

Kirjoita ohjelma, jossa on merkkijono

```
"/.+(STL).*$1/"
```

ja joka poistaa kaikki muut merkit paitsi STL käyttäen aluksi metodia erase(pos,count) ja sitten erase(iter,iter).

Harjoitus 6.16

Kirjoita ohjelma, joka hyväksyy nämä määrittelyt:

```
string sentence( "kind of" );
string s1( "whistle" );
string s2( "pixie" );
```

Käyttäen lukuisia merkkijonon lisäysfunktioita, muodosta merkkijono sentence, jonka arvo on

```
"A whistling-dixie kind of walk."
```

6.10 Merkkijono jossain muussa muodossa

Eräs hankala yksityiskohta on tarpeemme havaita tekstinkyselyjärjestelmällämme sama sana eri aikamuodoissa kuten cry, cries ja cried, luvuissa, kuten baby ja babies ja kirjainten koon perusteella, kuten home ja Home. Kaksi ensimmäistä tapausta kuuluvat sanan loppuliiteongelmaan. Vaikka tähän tekstiin eivät kuulu yleiset loppuliitteisiin kuuluvat ongelmat, seuraava pieni malliratkaisu on hyvää harjoitusta string-luokan operaatioista.

Ennen kuin palaamme loppuliitteisiin, ratkaiskaamme kuitenkin ensiksi tämä yksinkertaisempi isojen kirjainten ongelma. Sen sijaan, että yrittäisimme olla fiksuja ja havaitsisimme tietyt tapaukset, me vain korvaamme kaikki isot kirjaimet pienillä kirjaimilla. Toteutuksemme näyttää tältä:

```
void
strip_caps( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator iter = words->begin();
    vector<string,allocator>::iterator iter_end = words->end();
```

```
    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

    while ( iter != iter_end ) {
            string::size_type pos = 0;
            while (( pos = (*iter).find_first_of( caps, pos ))
                    != string::npos )
                (*iter)[ pos ] = tolower( (*iter)[pos] );
            ++iter;
    }
}
```

Funktio

```
tolower( (*iter)[pos] );
```

on C-vakiokirjastosta. Funtkio saa ison kirjaimen ja palauttaa vastaavan pienen kirjaimen. Jotta voisimme käyttää sitä, pitää ottaa mukaan otsikkotiedosto

```
#include <ctype.h>
```

(Tämä tiedosto sisältää myös muiden funktioiden esittelyjä kuten isalpha(), isdigit(), ispunct(), isspace() ja toupper(). Jos haluat nähdä täydellisen luettelon selityksineen, katso julkaisusta [PLAUGER92]. C++-vakiokirjastossa on määritelty ctype-luokka, johon on kapseloituna C-standardikirjaston toimintoja, kuten myös jäsenettömiä funktioita, kuten toupper(), tolower() jne. Jotta niitä voitaisiin käyttää, pitää ottaa mukaan C++-standardin otsikkotiedosto

```
#include <locale>
```

Tätä kirjoitettaessa kyseinen toteutus ei kuitenkaan ole käytettävissä, joten käytämme C-standardin toteutusta.)

Loppuliitteiden käsittelyä on kuitenkin vaikea toteuttaa tiukasti. Silti jopa puutteelliset toteutukset johtavat merkittävään laadun ja koon parannukseen kyseltävien sanakokoelmien suhteen.

Toteutuksemme käsittelee vain sanoja, jotka loppuvat 's':

```
void
suffix_text( vector<string,allocator> *words )
{
    vector<string,allocator>::iterator
                                    iter = words->begin(),
                                    iter_end = words->end();

    while ( iter != iter_end ) {
            // jos 3 tai vähemmän merkkejä, anna olla
            if ( (*iter).size() <= 3 )
                { ++iter; continue; }

            if ( (*iter)[ (*iter).size()-1 ] == 's' )
                suffix_s( *iter );

            // loppuliitteiden lisäkäsittely tulee tähän kuten
```

```
                    // ed, ing, ly

                    ++iter;
        }
}
```

Yksinkertaisen heuristista on se, että ei vaivauduta sanojen suhteen, joissa on vähemmän kuin neljä kirjainta. Tämä säästää meitä sanojen kuten has, its, is käsittelyltä, mutta emme onnistu nappaamaan esimerkiksi sanoja tv tai tvs, jotka ovat saman sanan yksikkö- ja monikkomuoto.

Jos sana loppuu "ies" kuten babies ja cries, pitää "ies" korvata kirjaimella "y":

```
string::size_type pos3 = word.size()-3;

string ies( "ies" );
if ( ! word.compare( pos3, 3, ies )) {
      word.replace( pos3, 3, 1, 'y' );
      return;
}
```

compare() palauttaa arvon 0, jos sen vertailemat kaksi merkkijonoa ovat yhtäsuuria. pos3 yksilöi kohdan word-sanasta, vertailu alkaa. Toinen argumentti, tässä tapauksessa 3, ilmaisee alimerkkijonon pituuttaa positiosta pos3. Kolmas argumentti on varsinainen merkkijono, johon verrataan. (Itse asiassa compare()-versioita on kuusi. Katsomme muita versioita lyhyesti seuraavassa kohdassa.)

replace() korvaa yhden tai useamman merkkijonon merkin yhdellä tai useammalla vaihtoehtoisella merkillä. Esimerkissämme korvaamme kolmemerkkisen merkkijonon "ies" yhdellä "y"-merkillä. (replace()-funktiosta on olemassa kymmenen ylikuormitettua ilmentymää. Katsomme niitä lyhyesti seuraavassa kohdassa.)

Samalla tavalla, jos sana loppuu "ses" kuten sanat *promises* ja *purposes*, pitää poistaa yksinkertaisesti loppuliite "es"[7]:

```
string ses( "ses" );
if ( ! word.compare( pos3, 3, ses )) {
      word.erase( pos3+1, 2 );
      return;
}
```

Jos sana loppuu "ous" kuten oblivious, fulvous ja cretaceous, emme tee mitään. Samalla tavalla, jos sana loppuu "is" kuten genesis, mimesis ja hepatitis, emme tee mitään. (Tämä järjestelmä ei kuitenkaan ole täydellinen. Esimerkiksi kiwis vaatii, että pudotamme lopusta pois merkin 's'.) Myös, jos sana loppuu "ius" kuten genius tai "ss" kuten hiss, lateness tai less, emme tee mitään. Jotta voimme päätellä, teemmekö mitään, käytämme compare()-funktiosta toista muotoa:

```
string::size_type spos = 0;
string::size_type pos3 = word.size()-3;
```

7. On olemassa tietysti poikkeuksia. Esimerkiksi sanasta *crises* tulee heuristisesti *cris*. Hups!

```
// "ous", "ss", "is", "ius"
string suffixes( "oussisius" );

if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||     // ous
    ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||   // ius
    ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) || // ss
    ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))   // is
return;
```

Muussa tapauksessa yksinkertaisesti pudotamme lopusta kirjaimen 's':

```
// poista lopusta 's'
word.erase( pos3+2 );
```

Kelvolliset nimet kuten Pythagoras, Brahms ja maalari Burne-Jones jäävät yleissääntöjen ulkopuolelle. Käsittelemme ne — niin, jätämme sen harjoitukseksi lukijalle, kun esittelemme assosiatiivisen säiliötyypin.

Ennen kuin siirrymme assosiatiivisiin taulukko- ja joukkosäiliöihin, käsittelemme lyhyesti joitakin merkkijonon lisäoperaatioita seuraavassa kohdassa.

Harjoitus 6.17

Ohjelmamme ei käsittele sanojen loppuja, jotka päättyvät kirjaimiin ed kuten surprised, ly kuten surprisingly ja ing kuten surprising. Lisää joku seuraavista loppuliitteiden käsittelijöistä ohjelmaan: (a) suffix_ed(), (b) suffix_ly(), or (c) suffix_ing().

6.11 Merkkijonon lisäoperaatioita

Toinen muoto erase()-funktiosta saa iteraattoriparin, joka ilmaisee poistettavien merkkien alueen. Jos esimerkiksi on merkkijono

```
string name( "AnnaLiviaPlurabelle" );
```

tehkäämme siitä merkkijono "Annabelle":

```
typedef string::size_type size_type;
size_type startPos = name.find( 'L' )
size_type endPos   = name.find_last_of( 'a' );

name.erase( name.begin()+startPos,
        name.begin()+endPos );
```

Merkki, jota toinen iteraattori osoittaa, ei ole poistettavien merkkien alueella.

Lopuksi kolmas muoto saa iteraattorin, joka ilmaisee lähtöposition, josta alkaen se poistaa kaikki merkit merkkijonon loppuun saakka. Esimerkiksi

```
name.erase( name.begin()+4 );
```

jättää name-merkkijonoon arvon "Anna".

insert()-operaatio tukee lisämerkkien lisäystä merkkijonoon ilmaistusta positiosta alkaen. Sen yleinen muoto on

```
string_object.insert( position, new_string );
```

jossa position ilmaisee paikkaa string_object-oliossa, johon uusi merkkijono new_string lisätään. new_string voi olla string-olio, C-tyylinen merkkijono tai yksittäinen merkki. Esimerkiksi:

```
string string_object( "Missisippi" );
string::size_type pos = string_object.find( "isi" );
string_object.insert( pos+1, 's' );
```

insert()-operaatio tukee osan merkitsemistä uudesta merkkijonosta new_string. Esimerkiksi:

```
string new_string ( "AnnaBelle Lee" );
string_object += ' '; // lisää tyhjä merkki

// etsi uudesta merkkijonosta, new_string, alku- ja loppupositio
pos = new_string.find( 'B' );
string::size_type posEnd = new_string.find( ' ' );

string_object.insert(
        string_object.size(),  // positio string_object-oliossa
        new_string, pos,       // alkupositio new_string-oliossa
        posEnd                 // loppupositio new_string-oliossa
)
```

string_object sisältää nyt merkkijonon "Mississippi Belle". Jos haluaisimme lisätä kaiken uudesta merkkijonosta, new_string, alkaen positiosta pos, voisimme jättää pois arvon posEnd.

Olkoon seuraavat kaksi merkkijonoa

```
string s1( "Mississippi" );
string s2( "Annabelle" );
```

Näistä haluaisimme luoda kolmannen merkkijonon, jonka arvo on "Miss Anna". Kuinkahan tekisimme sen?

Eräs metodi on käyttää assign()- ja append()-merkkijono-operaatioita. Niiden avulla voimme vuorollaan kopioida ja yhdistää osia merkkijono-oliosta toiseen. Esimerkiksi:

```
string s3;

// kopioi ensimmäiset 4 merkkiä merkkijonosta s1
s3.assign( s1, 4 );
```

s3 sisältää nyt arvon "Miss".

```
// liitä tyhjä merkki
s3 += ' ';
```

s3 sisältää nyt arvon "Miss ".

```
// yhdistä nyt ensimmäiset 4 merkkiä merkkijonosta s2
s3.append( s2, 4 );
```

s3 sisältää nyt arvon “Miss Anna”. Vaihtoehtoisesti voimme kirjoittaa

```
s3.assign( s1, 4 ).append( ' ' ).append( s2, 4 );
```

Jos haluamme erottaa merkkijonosta osan, joka ei ala alusta, voimme käyttää vaihtoehtoista muotoa, jolle annetaan kaksi kokonaislukuarvoa: alkupositio ja pituus. Position lasketaan alkavan kohdasta 0. Kun haluamme erottaa esimerkiksi merkkijonon "belle" merkkijonosta "Annabelle", määritämme alkupositioksi 4 ja pituudeksi 5:

```
string beauty;

// sijoita "belle" beauty-merkkijonoon
beauty.assign( s2, 4, 5 );
```

Sen sijaan, että antaisimme position ja pituuden, voimme vaihtoehtoisesti antaa iteraattoriparin. Esimerkiksi:

```
// sijoita "belle" beauty-merkkijonoon
beauty.assign( s2, s2.begin()+4, s2.end() );
```

Seuraavassa esimerkissä meillä on kaksi merkkijonoa, jotka edustavat nykyistä ja odottavaa työtä. Meidän pitää vaihtaa niitä keskenään ajoittain projektista toiseen ja takaisin. Esimerkiksi:

```
string current_project( "C++ Primer, 3rd Edition" );
string pending_project( "Fantasia 2000, Firebird segment" );
```

swap()-operaatio vaihtaa kahden merkkijonon arvot keskenään. Jokainen seuraava käynnistys

```
current_project.swap( pending_project );
```

vaihtaa näiden kahden olion merkkijonoarvot keskenään.

Olkoon merkkijono

```
string first_novel( "V" );
```

Tällöin indeksi

```
char ch = first_novel[ 1 ];
```

palauttaa tuntemattoman merkin, koska indeksi on alueen ulkopuolella: first_novel-merkkijonon pituus on 1 ja sitä indeksoidaan arvolla 0. Indeksioperaattorissa ei ole raja-arvotarkistusta emmekä halua sitä “hyvin käyttäytyvään” koodiin kuten seuraavassa:

```
int
elem_count( const string &word, char elem )
{
    int occurs = 0;

    // toimii hyvin: ei tarvita tarkistusta raja-arvojen ylityksestä
    for ( int ix=0; ix < word.size(); ++ix )
        if ( word[ ix ] == elem )
            ++occurs;
    return occurs;
}
```

Kuitenkin mahdollisesti “huonosti käyttäytyvään” koodiin, kuten

```
void
mumble( const string &st, int index )
{
    // potentiaalinen raja-arvovirhe
    char ch = st[ index ];

    // ...
}
```

on olemassa at()-operaatio suorituksenaikaiselle indeksin tarkistukselle. Jos indeksi on kelvollinen, at() palauttaa kyseisen merkkielementin samalla tavalla kuin indeksioperaattori tekee. Jos indeksi on kelvoton, at() heittää out_of_range-poikkeuksen:

```
void
mumble( const string &st, int index )
{
    try {
        char ch = st[ index ];
        // ...
    }
    catch( std::out_of_range ) { ... }
    // ...
}
```

Mitkä tahansa kaksi merkkijonoa, jotka eivät ole yhtäsuuria, ovat leksikografisesti — tarkoittaa aakkosten mukaan — järjestyksessä. Olkoot esimerkiksi seuraavat kaksi merkkijonoa

```
string cobol_program_crash( "abend" );
string cplus_program_crash( "abort" );
```

Tällöin merkkijono-olio cobol_program_crash on leksikografisesti pienempi kuin merkkijono-olio cplus_program_crash, kun verrataan ensimmäistä erisuurta merkkiä: e on ennen kirjainta o englannin kielen aakkosissa.

compare()-merkkijono-operaatio vertailee leksikografisesti kahta merkkijonoa. Olkoon

```
s1.compare( s2 );
```

compare() palauttaa jonkin seuraavista mahdollisista arvoista:

1. Jos s1 on suurempi kuin s2, compare() palauttaa positiivisen arvon.
2. Jos s1 on pienempi kuin s2, compare() palauttaa negatiivisen arvon.
3. Jos s1 on yhtäsuuri kuin s2, compare() palauttaa arvon 0.

Esimerkiksi:

```
cobol_program_crash.compare( cplus_program_crash );
```

palauttaa negatiivisen arvon, kun taas

```
cplus_program_crash.compare( cobol_program_crash );
```

palauttaa positiivisen arvon. String-luokan vertailuoperaattorit (<,>,!=,==,<=,>=) tarjoavat vaihtoehtoisen lyhennetyn ilmaisun compare()-operaatiolle.

Kuudella ylikuormitetun compare()-operaatioiden joukolla voimme merkitä alimerkkijonon joko toisesta tai molemmista vertailtavista merkkijonoista. Esimerkit on esitetty edellisessä kohdassa loppuliitteiden käsittelyn yhteydessä.

replace() tarjoaa meille kymmenen eri tapaa korvata yksi tai useampi merkkijonossa oleva merkki yhdellä tai useammalla vaihtoehtoisella merkillä (olemassaolevien ja korvaavien merkkien lukumäärien ei tarvitse olla samoja). replace()-operaatiolla on kaksi päämuotoa ja niiden muunnelmat perustuvat korvattavien merkkien merkitsemismetodeihin. Eräässä muodossa ensimmäiset kaksi argumenttia muodostavat merkkijoukon aloituspositionin indeksin ja korvattavien merkkien lukumäärän. Toisessa muodossa välitetään iteraattoripari, joka ilmaisee merkkijoukon aloituskohdan ja yhden yli viimeisen korvattavan merkin. Tässä on esimerkki ensimmäisestä muodosta:

```
string sentence(
  "An ADT provides both interface and implementation." );

string::size_type position = sentence.find_last_of( 'A' );
string::size_type length = 3;

// korvaa ADT-arvolla Abstract Data Type
sentence.replace( position, length, "Abstract Data Type" );
```

Ensimmäinen argumentti, position, edustaa aloituskohtaa ja toinen argumentti, length, edustaa positiosta position alkavan merkkijonon pituutta. Joten pituus 3 (eikä 2) edustaa merkkijonoa "ADT". Kolmas argumentti edustaa uutta merkkijonoa. On olemassa lukuisia muunnelmia siitä, kuinka uusi merkkijono määritetään. Esimerkiksi tämä muunnelma saa string-olion C-tyylisen merkkijonon sijasta

```
string new_str( "Abstract Data Type" );
sentence.replace( position, length, new_str );
```

Seuraava muunnelma, joka — myönnettäköön — on tehoton, lisää uudesta merkkijonosta osan, joka on merkitty positiolla ja pituudella.

```
#include <string>
typedef string::size_type size_type;
```

```
// hae 3 sanan positiot
size_type posA = new_str.find( 'A' );
size_type posD = new_str.find( 'D' );
size_type posT = new_str.find( 'T' );

// ok: korvaa T arvolla "Type"
sentence.replace( position+2, 1, new_str, posT, 4 );

// ok: korvaa D arvolla "Data "
sentence.replace( position+1, 1, new_str, posD, 5 );

// ok: korvaa A arvolla "Abstract "
sentence.replace( position, 1, new_str, posA, 9 );
```

Toinen muunnelma korvaa alimerkkijonon yksittäisellä merkillä, jota toistetaan tietty määrä. Esimerkiksi:

```
string hmm( "Some celebrate Java as the successor to C++." );
string::size_type position = hmm.find( 'J' );

// ok: korvataan Java merkkijonolla xxxx
hmm.replace( position, 4, 'x', 4 );
```

Tässä on viimeinen muunnelma, jonka haluaisimme tuoda esille ja jossa käytämme osoitinta merkkitaulukkoon ja pituutta, joka ilmaisee uuden merkkijonon pituuden. Esimerkiksi:

```
const char *lang = "EiffelAda95JavaModula3";
int index[] = { 0, 6, 11, 15, 22 };

string ahhem(
     "C++ is the language for today's power programmers." );

ahhem.replace(0, 3, lang+index[1], index[2]-index[1]);
```

Tässä on esimerkki toisesta muodosta, jossa käytetään iteraattoriparia ilmaisemaan korvauksen kohteena olevaa alimerkkijonoa.

```
string sentence(
  "An ADT provides both interface and implementation." );

// osoittaa merkkijonon ADT merkkiin 'A'
string::iterator start = sentence.begin()+3;

// korvaa ADT arvolla Abstract Data Type
sentence.replace( start, start+3, "Abstract Data Type" );
```

Neljä muuta muunnosta sallivat korvattavan merkkijonon olevan string-olio, jolloin merkki voidaan lisätä N kertaa, iteraattoripari tai C-tyylinen merkkijono, josta voidaan käyttää N merkkiä korvaavana merkkijoukkona.

Tässä on kaikki, mitä haluamme sanoa merkkijono-operaatioista. Jos haluat yksityiskoh-

taisempaa ja täydellisempää tietoa, katso C++-standardin määrittely [ISO-C++97]. (Tätä kirjoitettaessa ei C++-kirjastossa ole tähän liittyvää parempaa tekstiä.)

Harjoitus 6.18

Kirjoita ohjelma, joka hyväksyy seuraavat kaksi merkkijonoa:

```
string quote1( "When lilacs last in the dooryard bloom'd" );
string quote2( "The child is father of the man" );
```

Käytä assign()- ja append()-operaatioita ja luo merkkijono

```
string sentence( "The child is in the dooryard" );
```

Harjoitus 6.19

Kirjoita ohjelma, jossa on merkkijonot

```
string generic1( "Dear Ms Daisy:" );
string generic2( "MrsMsMissPeople" );
```

ja joka toteuttaa funktion

```
string generate_salutation( string generic1,
                            string lastname,
                            string generic2,
                            string::size_type pos,
                            int length );
```

käyttäen replace()-operaatioita, joissa lastname korvaa Daisy:n ja pos indeksoi generic2:ta pituudella length merkkiä ja korvaa Ms:n. Esimerkiksi seuraava

```
string lastName( "AnnaP" );
string greetings =
    generate_salutation( generic1, lastName, generic2, 5, 4 );
```

palauttaa merkkijonon

```
Dear Miss AnnaP:
```

6.12 Tekstin sijaintitaulukon rakentaminen

Tässä kohtaa rakennamme jokaisen yksittäisen sanan sijainnin rivi- ja saraketiedoista kokoelman esitelläksemme ja kokeillaksemme säiliöiden assosiatiivista taulukkotyyppiä eli karttaa (*map*). (Seuraavassa kohdassa muodostamme poisjätettävät sanat esitelläksemme ja kokeillaksemme säiliöiden assosiatiivista joukkotyyppiä (*set*).) Yleensä joukko on hyödyllisempi silloin, kun haluamme yksinkertaisesti tietää, onko arvoa nähty, ja kartta on hyödyllisempi silloin, kun haluamme tallentaa siihen liittyvän arvon (ja ehkä muokata). Molemmissa tapauksissa elementit on tallennettu järjestetysti, jotta ne tukisivat tehokasta tallentamista ja hakua.

Assosiatiivisessa taulukossa eli kartassa (*map*) käytetään avain/arvoparia: avain toimii in-

deksinä taulukkoon ja arvo tallennettavana sekä haettavana tietona. Ohjelmassamme jokainen merkkijono toimii avaimena ja rivi- ja sarakepaikkojen vektori arvona. Kun käsittelemme paikkavektoria, indeksoimme karttaa käyttäen indeksioperaattoria. Esimerkiksi:

```
string query( "pickle" );
vector< location > *locat;

// palauttaa paikan location<vector>*, johon liittyy sana "pickle"
locat = text_map[ query ];
```

Kartan avaintyyppi — esimerkissämme string — toimii indeksinä. Vastaava location<vector>*-arvo palautetaan.

Jotta karttaa voitaisiin käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <map>
```

Kartan (ja joukon) kaksi pääkäyttötapaa ovat täyttäminen elementeillä ja kysely elementtien olemassaolosta. Seuraavassa alikohdassa katsomme, kuinka voimme määritellä ja lisätä avain-arvopareja. Sen jälkeen katsomme, kuinka saamme tietää, onko elementti läsnä, ja jos on, kuinka haemme sen arvon.

6.12.1 Kartan määrittely ja täyttäminen

Jotta karttaolio voitaisiin määritellä, pitää ilmaista vähintään avain ja arvon tyyppi. Esimerkiksi

```
map<string,int> word_count;
```

määrittelee karttaolion word_count, jota indeksoidaan merkkijonolla ja joka sisältää int-arvon. Samalla tavalla

```
class employee;
map<int,employee*> personnel;
```

määrittelee karttaolion personnel, jota indeksoidaan int-tyypillä (se edustaa yksilöllistä työntekijän numeroa) ja sisältää osoittimen employee-luokan ilmentymään.

Tekstinkyselyjärjestelmämme kartan esittely näyttää tältä:

```
typedef pair<short,short> location;
typedef vector<location>  loc;

map<string,loc*> text_map;
```

Koska tämän kirjoittamisen aikaan käytettävissämme olevat kääntäjät eivät tue malliparametrien oletusarvoja, pitää käyttää seuraavaa laajennettua määrittelyä:

```
map<string,loc*,   // avain/arvopari
   less<string>,  // vertailuoperaattori järjestelyyn
   allocator>     // muistin oletusvaraaja
text_map;
```

Oletusarvo on, että assosiatiiviset säiliötyypit järjestetään käyttäen pienempi kuin -operaat-

toria. Voimme aina kuitenkin korvata sen, kun ilmaisemme vaihtoehtoisen vertailuoperaattorin (katso kohdasta 12.3 funktio-olioista).

Kun kartta on määritelty, on seuraava vaihe sen täyttäminen avain/arvoparien elementeillä. Se, mitä välittömästi haluaisimme kirjoittaa, on seuraava:

```
#include <map>
#include <string>
map<string,int> word_count;

word_count[ string("Anna") ]  = 1;
word_count[ string("Danny") ] = 1;
word_count[ string("Beth") ]  = 1;

// ja niin edelleen ...
```

Kun kirjoitamme

```
word_count[ string("Anna") ] = 1;
```

tapahtuvat seuraavat asiat:

1. Muodostuu nimetön, tilapäinen string-olio, joka alustetaan merkkijonolla "Anna" ja sille välitetään karttaluokkaan liittyvä indeksioperaattori.
2. Kartasta word_count etsitään sanaa "Anna". Sanaa ei löydy.
3. Karttaan word_count lisätään uusi avain-arvopari. Avain on tietysti string-olio, joka sisältää arvon "Anna". Arvo ei kuitenkaan ole 1, vaan 0.
4. Kun lisäys on tehty, arvoksi sijoitetaan sitten 1.

Kun avain lisätään karttaan indeksioperaattorin kautta, sen arvoksi alustetaan taustalla olevan elementin oletusarvo. Sisäisten, aritmeettisten tyyppien oletusarvo on 0.

Itse asiassa, kun käytetään indeksioperaattoria kartan elementtien alustamiseen, se saa aikaan jokaisen arvon alustamisen oletusarvokseen, ja sen jälkeen sijoitetaan eksplisiittinen arvo. Jos elementit ovat luokkaolioita, joille oletusalustus ja sijoitus ovat käsittelyn kannalta merkittäviä, se voi vaikuttaa ohjelmiemme suorituskykyyn, vaikka ohjelman virheettömyyteen se ei vaikuttaisikaan.

Parempana pidetty, vaikkakin syntaktisesti pelottavampi, on seuraava yksittäisen elementin lisäysmetodi:

```
// parempana pidetty yhden elementin lisäysmetodi
word_count.insert(
     map<string,int>::
          value_type( string("Anna"), 1 )
);
```

Kartassa on määriteltynä value_type-tyyppi, joka edustaa vastaavaa avain-arvoparia. Rivien

```
map< string,int >::
      value_type( string(“Anna”), 1 )
```

tarkoitus on luoda oliopari, joka sitten suoraan lisätään karttaan. Luettavuuden vuoksi voimme käyttää typedef-nimeä:

```
typedef map<string,int>::value_type valType;
```

Kun käytämme tätä, lisäyksemme näyttää yksinkertaisemmalta:

```
word_count.insert( valType( string(“Anna”), 1 ));
```

Jos haluamme lisätä alueen avain-arvoelementtejä, voimme käyttää insert()-metodia, jolle annamme iteraattoriparin. Esimerkiksi:

```
map< string, int > word_count;
// ... täytä se

map< string, int > word_count_two;

// lisää kopio kaikista avain-arvopareista
word_count_two.insert(word_count.begin(),word_count.end());
```

Tässä esimerkissä sama vaikutus olisi saavutettu alustamalla ensimmäinen kartta toisella:

```
// alusta kaikkien avain-arvoparien kopiolla
map< string, int > word_count_two( word_count );
```

Käykäämme läpi, kuinka rakentaisimme tekstikarttamme. separate_words(), jota käsiteltiin kohdassa 6.8, luo kaksi vektoria: merkkijonovektorin tekstin jokaiselle sanalle ja paikkavektorin, joka sisältää rivi- ja sarakeparin arvoja. Jokaiselle sanalle merkkijonovektorissa on yhtäpitävä elementti paikkavektorissa, joka pitää sisällään rivi- ja saraketiedon tuolle sanalle. Merkkijonovektori muodostaa siten tekstikarttamme avainarvojen kokoelman. Paikkavektori muodostaa vastaavan arvokokoelman.

separate_words() palauttaa nämä kaksi vektoria olioparina, joka sisältää osoittimen kumpaankin. Argumentti build_word_map()-funktiollemme on tämä oliopari. Palautusarvo on tekstin sijaintikartta — tai pikemminkin osoitin siihen:

```
// typedef saa esittelyt helpommiksi
typedef pair< short,short >  location;
typedef vector< location >   loc;
typedef vector< string >     text;
typedef pair< text*,loc* >   text_loc;

extern map< string, loc* >*
        build_word_map( const text_loc *text_locations );
```

Ensimmäiset valmistelevat vaiheemme on varata muistia tyhjälle kartalle vapaavarastosta ja erottaa merkkijonovektori ja paikkavektori argumenttiparista, joka on välitetty argumenttina:

```
map<string,loc*> *word_map = new map< string, loc* >;

vector<string>   *text_words = text_locations->first;
vector<location> *text_locs  = text_locations->second;
```

Seuraavaksi pitää käydä nämä kaksi vektoria läpi rinnakkain. On olemassa kaksi asiaa, jotka pitää ottaa huomioon:

1. Sanaa ei ole vielä kartassa. Tässä tapauksessa pitää lisätä avain-arvopari.
2. Sana on jo lisätty. Tässä tapauksessa pitää päivittää paikkavektoria lisätiedolla sanan rivistä ja sarakkeesta.

Tässä on toteutuksemme:

```
register int elem_cnt = text_words->size();
for ( int ix = 0; ix < elem_cnt; ++ix )
{
    string textword = ( *text_words )[ ix ];

    // poisjättämisen strategioita: älä lisää karttaan,
    // jos merkkejä on vähemmän kuin 3
    // tai jos löytyy poisjätettävien joukosta
    if ( textword.size() < 3 ||
        exclusion_set.count( textword ))
           continue;

    // päättele, onko sana jo lisätty
    // jos count() palauttaa arvon 0, ei löydy -- lisää se
    if ( ! word_map->count((*text_words)[ix] ))
    {
        loc *ploc = new vector<location>;
        ploc->push_back( (*text_locs)[ix] );
        word_map->insert( value_type( (*text_words)[ix], ploc ));
    }
    else
       // päivitä paikkavektoria sanan tiedoilla
       (*word_map)[(*text_words)[ix]]->push_back((*text_locs)[ix]);
}
```

Syntaktisesti monimutkainen lauseke

```
(*word_map)[(*text_words)[ix]]->push_back((*text_locs)[ix]);
```

on ehkä helpompi ymmärtää, jos jaamme sen yksittäisiin komponentteihinsa:

```
// hae päivitettävä sana
string word = (*text_words)[ix];

// hae paikkavektori
vector<location> *ploc = (*word_map)[ word ];

// hae rivi- ja sarakepari
location loc = (*text_locs)[ix];

// lisää uusi paikkapari
ploc->push_back(loc);
```

Jäljelle jäävä syntaktinen monimutkaisuus on tulosta käsittelemistämme vektorien osoittimista kuin vektoreista itsestään. Jos haluamme käyttää indeksioperaattoria suoraan, emme voi kirjoittaa

```
string word = text_words[ix]; // virhe
```

Sen sijaan pitää käyttää osoitintamme ensin käänteisesti:

```
string word = (*text_words)[ix]; // ok
```

Lopulta build_word_map() palauttaa sisäisen karttamme:

```
return word_map;
```

Näin voisimme käynnistää sen main()-funktiostamme:

```
int main()
{
   // lue teksti sisään ja erottele se
   vector<string,allocator> *text_file = retrieve_text();
   text_loc *text_locations = separate_words( text_file);

   // käsittele sanat
   // ...

   // rakenna sanojen sijaintikartta ja tee kysely
   map<string,loc*,less<string>,allocator>
         *text_map = build_word_map( text_locations );

   // ...
}
```

6.12.2 Kartan elementin etsiminen ja hakeminen

Indeksioperaattorin käyttö on yksinkertaisin metodi arvon hakuun. Esimerkiksi:

```
// map<string,int> word_count;
int count = word_count[ "wrinkles" ];
```

Tämä koodi toimii kuitenkin tyydyttävästi vain, kun kartassa on ilmentymä avaimelle. Ellei ilmentymää löydy, indeksioperaattorin käyttö saa aikaan ilmentymän lisäyksen. Tässä esimerkissä avain-arvopari

```
string( "wrinkles" ), 0
```

lisätään karttaan word_count ja count alustetaan arvoon 0.

On olemassa kaksi karttaoperaatiota sen havaitsemiseksi, onko avainelementti olemassa, ilman, että sen löytymättömyys aiheuttaisi ilmentymän lisäyksen:

1. count(keyValue): count() palauttaa avainarvojen keyValue ilmentymien lukumäärän kartasta. (Tietysti kartassa paluuarvo voi olla joko 0 tai 1.) Jos paluuarvo on nollasta poikkeava, voimme turvallisesti käyttää indeksioperaattoria. Esimerkiksi:

```
int count = 0;
if ( word_count.count( "wrinkles" ))
      count = word_count[ "wrinkles" ];
```

2. find(keyValue): find() palauttaa iteraattorin ilmentymään kartassa, jos ilmentymä löytyy, tai iteraattorin, joka on yhtä kuin end(), jos ilmentymää ei löydy. Esimerkiksi:

```
int count = 0;
map<string,int>::iterator it = word_count.find( "wrinkles" );
if ( it != word_count.end() )
  count = (*it).second;
```

Kartan elementin iteraattori osoittaa oliopariin, jossa ensimmäinen, first, sisältää avaimen ja toinen, second, sisältää arvon (katsomme tätä jälleen seuraavassa alikohdassa).

6.12.3 Kartan läpikäynti

Nyt, kun olemme rakentaneet karttamme, haluaisimme tulostaa sen sisällön. Voimme tehdä sen käymällä läpi elementit, joita ilmaisee iteraattoripari begin() ja end(). Tässä on funktiomme, display_map_text(), joka tekee juuri tuon.

```
void
display_map_text( map<string,loc*> *text_map )
{
   typedef map<string,loc*> tmap;
      tmap::iterator iter = text_map->begin(),
   iter_end = text_map->end();

   while ( iter != iter_end )
   {
         cout << "word: " << (*iter).first << " (";

         int         loc_cnt   = 0;
         loc         *text_locs = (*iter).second;
```

```
        loc::iterator liter     = text_locs->begin(),
                      liter_end = text_locs->end();

        while ( liter != liter_end )
        {
            if ( loc_cnt )
                cout << ',';
            else ++loc_cnt;

                                cout << '(' << (*liter).first
                << ',' << (*liter).second << ')';

            ++liter;
        }

        cout << ")\n";
        ++iter;
    }

    cout << endl;
}
```

Ellei kartassa ole elementtejä, ei ole järkeä vaivautua käynnistämään tulostusfunktiotamme. Eräs menetelmä sen havaitsemiseen, onko kartta tyhjä, on käynnistää size()-funktio:

```
if ( text_map->size() )
    display_map_text( text_map );
```

Mutta sen sijaan, että tarpeettomasti laskisimme kaikki elementit, voimme käynnistää suoremmin empty()-funktion:

```
if ( ! text_map->empty() )
    display_map_text( text_map );
```

6.12.4 Kartta sanamuunnoksille

Tässä on pieni ohjelma, joka kuvaa kartan rakentamista, etsimistä ja läpikäyntiä. Käytämme kahta karttaa ohjelmassamme. Sanoamuunnoskarttamme sisältää kaksi string-tyyppistä elementtiä. Avain edustaa sanaa, joka vaatii erityiskäsittelyä; arvo edustaa muunnosta, jota meidän tulisi käyttää aina, kun kohtaamme tuon sanan. Yksinkertaisuuden vuoksi kovakoodaamme kartan sanaparit (harjoituksen vuoksi voit halutessasi yleistää ohjelmaa niin, että se lukee muunnettavia sanapareja joko vakiosyötöstä tai käyttäjän määrittämästä tiedostosta). Tilastokarttaamme tallennetaan tietoja suoritetuista muunnoksista. Tässä on ohjelmamme.

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

int main()
{
   map< string, string > trans_map;
   typedef map< string, string >::value_type valType;

   // ensimmäinen keino: koodaa käsin muunnoskartta
   trans_map.insert( valType( "gratz", "grateful" ));
   trans_map.insert( valType( "'em",   "them"     ));
   trans_map.insert( valType( "cuz",   "because"  ));
   trans_map.insert( valType( "nah",   "no"       ));
   trans_map.insert( valType( "sez",   "says"     ));
   trans_map.insert( valType( "tanx",  "thanks"   ));
   trans_map.insert( valType( "wuz",   "was"      ));
   trans_map.insert( valType( "pos",   "suppose"  ));

   // ok: näyttäkäämme se
   map< string,string >::iterator it;

   cout << "Here is our transformation map: \n\n";
   for ( it = trans_map.begin();
        it != trans_map.end(); ++it )
        cout << "key: "   << (*it).first << "\t"
             << "value: " << (*it).second << "\n";
   cout << "\n\n";

   // toinen keino: koodaa käsin teksti ...
   string textarray[14]={ "nah", "I", "sez", "tanx", "cuz", "I",
       "wuz", "pos", "to", "not", "cuz", "I", "wuz", "gratz" };

   vector< string > text( textarray, textarray+14 );
   vector< string >::iterator iter;

   // ok: näyttäkäämme se
   cout << "Here is our original string vector:\n\n";
   int cnt = 1;
   for ( iter = text.begin(); iter != text.end(); ++iter, ++cnt )
        cout <<  *iter << ( cnt % 8 ? " " : "\n" );

   cout << "\n\n\n";

   // kartta, joka sisältää tilastotietoja -- muodosta dynaamisesti
   map< string,int > stats;
   typedef map< string,int >::value_type statsValType;
```

```
    // ok: varsinainen karttatyö -- ohjelman sydän
    for ( iter = text.begin(); iter != text.end(); ++iter )
        if (( it = trans_map.find( *iter )) != trans_map.end() )
        {
            if ( stats.count( *iter ))
                stats[ *iter ] += 1;
            else stats.insert( statsValType( *iter, 1 ));
            *iter = (*it).second;
        }

    // ok: näytä muunnettu vektori
    cout << "Here is our transformed string vector:\n\n";
    cnt = 1;
    for ( iter = text.begin(); iter != text.end(); ++iter, ++cnt )
        cout <<  *iter << ( cnt % 8 ? " " : "\n" );
    cout << "\n\n\n";

    // ok: käy läpi tilastokartta
    cout << "Finally, here are our statistics:\n\n";
    map<string,int,less<string>,allocator>::iterator siter;

    for ( siter = stats.begin(); siter != stats.end(); ++siter )
        cout << (*siter).first     << " "
             << "was transformed "
             << (*siter).second
             << ((*siter).second == 1
               ? " time\n" : " times\n" );
}
```

Kun ohjelma suoritetaan, se saa aikaan seuraavan tulostuksen:

```
Here is our transformation map:

key: 'em        value: them
key: cuz        value: because
key: gratz      value: grateful
key: nah        value: no
key: pos        value: suppose
key: sez        value: says
key: tanx       value: thanks
key: wuz        value: was


Here is our original string vector:

nah I sez tanx cuz I wuz pos
to not cuz I wuz gratz


Here is our transformed string vector:
```

```
no I says thanks because I was suppose
to not because I was grateful


Finally, here are our statistics:

cuz was transformed 2 times
gratz was transformed 1 time
nah was transformed 1 time
pos was transformed 1 time
sez was transformed 1 time
tanx was transformed 1 time
wuz was transformed 2 times
```

6.12.5 Elementtien poistaminen kartasta

erase()-operaatiosta on olemassa kolme muunnosta, joilla voidaan poistaa elementtejä kartasta. Kun poistamme yhden elementin, välitämme erase()-funktiolle joko avainarvon tai iteraattorin. Kun poistamme elementtijoukon, välitämme erase()-funktiolle iteraattoriparin. Jos esimerkiksi haluamme sallia, että käyttäjä voi poistaa elementtejä text_map-kartasta, teemme seuraavasti:

```
string removal_word;
cout << "type in word to remove: ";
cin  >> removal_word;

if ( text_map->erase( removal_word ))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

Ennen kuin yritämme poistaa sanan, voimme vaihtoehtoisesti tarkistaa, löytyykö sitä:

```
map<string,loc*>::iterator where;
where = text_map.find( removal_word );

if ( where == text_map->end() )
    cout << "oops: " << removal_word << " not found!\n";
else {
    text_map->erase( where );
    cout << "ok: " << removal_word << " removed!\n";
}
```

Tallennamme toteutuksemme text_map-karttaan jokaiseen sanaan useita paikkoja. Tämä järjestely monimutkaistaa varsinaisten paikka-arvojen tallennusta ja hakua. Vaihtoehtoinen toteutus on lisätä sana jokaiseen paikkaan. Kartta voi sisältää kuitenkin vain yhden esiintymän avainarvosta. Jotta samasta avaimesta voisi tehdä useita esiintymiä, pitää käyttää monikarttaa (*multimap*). Kohdassa 6.15 katsomme monikarttaa, joka on assosiatiivinen säiliötyyppi.

Harjoitus 6.20

Määrittele kartta, jossa indeksi on perheen sukunimi ja avain on lasten nimien vektori. Täytä kartta vähintään kuudella tiedolla. Testaa sen tukea käyttäjän kyselylle, joka perustuu sukunimeen, lapsen lisäämistä yhteen perheeseen ja kolmosien toiseen sekä kaikkien kartan elementtien tulostamista.

Harjoitus 6.21

Laajenna harjoituksen 6.20 karttaa niin, että vektoriin voidaan tallentaa merkkijonopari: lapsen nimi ja syntymäpäivä. Uudista harjoituksen 6.20 toteutusta niin, että se tukee uutta vektoriparia. Testaa muutoksiasi varmistuaksesi sen virheettömyydestä.

Harjoitus 6.22

Luettele vähintään kolme mahdollista sovellusta, joissa karttatyyppiä voitaisiin käyttää. Kirjoita määrittely jokaiselle kartalle ja ilmaise, kuinka elementtejä mahdollisesti lisättäisiin tai haettaisiin.

6.13 Poisjätettävän sanajoukon rakentaminen

Kartta muodostuu avain-arvoparista kuten osoite ja puhelinnumero, jossa avaimena käytetään yksilön nimeä. Joukko (*set*) sen sijaan on yksinkertainen kokoelma avainarvoja. Esimerkiksi yritys voi määritellä joukon bad_checks, joka muodostuu henkilöiden nimistä, jotka ovat käyttäneet kelvotonta maksuvälinettä yrityksessä kahden viime vuoden aikana. Joukko on mitä parhain, kun haluamme yksinkertaisesti tietää, löytyykö jotain tiettyä arvoa. Ennen kuin maksuväline hyväksytään, yritys voi esimerkiksi tehdä kyselyn bad_checks-joukosta nähdäkseen, löytyykö sieltä maksajan nimeä.

Rakennamme tekstinkyselyjärjestelmäämme poisjätettävien sanojen joukon, jotka ovat merkitykseltään neutraaleja sanoja kuten *the*, *and*, *into*, *with*, *but* jne. (Vaikka tämä parantaa merkittävästi sanahakemistomme laatua, ei se johda kyvyttömyyteemme paikallistaa Hamletin puheen ensimmäistä riviä: “To be or not to be”.) Ennen kuin lisäämme sanan karttaamme, tarkistamme, löytyykö sanaa poisjätettävien sanojen joukosta. Jos löytyy, emme lisää sitä karttaan.

6.13.1 Joukon määrittely ja täyttäminen

Jotta assosiatiivista säiliötyyppiä voidaan määritellä tai käyttää, pitää ottaa mukaan siihen kuuluva otsikkotiedosto.

```
#include <set>
```

Tässä on poisjätettävien sanojen joukko-olion määrittely:

```
set<string> exclusion_set;
```

Yksittäiset elementit lisätään joukkoon käyttäen insert-operaatiota. Esimerkiksi:

```
exclusion_set.insert( "the" );
exclusion_set.insert( "and" );
```

Vaihtoehtoisesti voimme lisätä elementtialueen antamalla iteraattoriparin insert()-operaatiolle. Esimerkiksi tekstinkyselyjärjestelmämme sallii käyttäjän määritellä kartasta poisjätettävien sanojen tiedoston. Ellei käyttäjä anna tiedostoa, täytämme joukon sanojen oletuskokoelmalla:

```
typedef set< string >::difference_type diff_type;
set< string > exclusion_set;

ifstream infile( "exclusion_set" );
if ( ! infile )
{
    static string default_excluded_words[25] = {
        "the","and","but","that","then","are","been",
        "can","can't","cannot","could","did","for",
        "had","have","him","his","her","its","into",
        "were","which","when","with","would"
    };

    cerr << "warning! unable to open word exclusion file! -- "
        << "using default set\n";

    copy( default_excluded_words, default_excluded_words+25,
         inserter( exclusion_set, exclusion_set.begin() ));
}
else {
    istream_iterator<string,diff_type> input_set(infile),eos;
    copy( input_set, eos, inserter( exclusion_set,
         exclusion_set.begin() ));
}
```

Tässä koodikatkelmassa tuodaan esille kaksi elementtiä, joita emme ole vielä nähneet: difference_type ja inserter-luokka. difference_type on tuloksen tyyppi, kun merkkijonojoukkomme kaksi iteraattoria vähennetään toisistaan. istream_iterator käyttää sitä yhtenä parametrinaan.

copy() on tietysti yksi geneerisistä algoritmeista (käsittelemme niitä yksityiskohtaisesti luvussa 12 ja liitteessä). Sen kaksi ensimmäistä argumenttia ovat joko iteraattoreita tai osoittimia, jotka ilmaisevat kopioitavien elementtien aluetta. Kolmas argumentti on joko iteraattori tai osoitin säiliön alkuun, johon elementit kopioidaan.

Ongelma on, että copy() olettaa säiliön koon olevan yhtäsuuri tai suurempi kuin kopioitavien elementtien lukumäärä. Tämä johtuu siitä, että copy() sijoittaa jokaisen elementin vuorollaan; se ei lisää elementtejä. Assosiatiiviset säiliöt eivät kuitenkaan tue koon esisijoitusta. Jotta voimme kopioida elementit poisjätettävien joukkoon, meidän pitää jotenkin saada copy() lisäämään jokainen elementti sijoittamisen sijasta. Luokka inserter tekee juuri sen. (Sitä käsitellään yksi-

tyiskohtaisesti kohdassa 12.4.)

6.13.2 Elementin etsiminen

Kaksi operaatiota, joilla voidaan kysellä joukosta olion olemassaoloa, ovat find() ja count(). find() palauttaa iteraattorin, joka osoittaa löytyneeseen elementtiin tai muussa tapauksessa iteraattorin, joka on yhtä kuin end(). count() palauttaa arvon 1, jos elementti löytyy ja arvon 0, ellei elementtiä löydy. Lisäämme funktioon build_word_map() testin exclusion_set ennen sanan lisäämistä karttaamme:

```
if ( exclusion_set.count( textword ))
    continue;
// ok: lisää sana karttaan
```

6.13.3 Joukon läpikäynti

Jotta voimme kokeilla sanojen sijaintipaikkakarttaamme, toteutamme pienen funktion, jolla voi tehdä yhden sanan kyselyitä (täyden kyselykielen tuki esitellään luvussa 17). Jos sana löytyy, haluamme näyttää jokaisen rivin, jolla sana esiintyy. Sana voi kuitenkin esiintyä useita kertoja samalla rivillä kuten

```
tomorrow and tomorrow and tomorrow
```

ja haluamme varmistua, että näytämme tämän rivin vain kerran.

Eräs strategia vain yhden esiintymän ylläpitämiseksi jokaisesta esiintymästä on käyttää joukkoa, kuten teemme seuraavassa koodikatkelmassa:

```
// hae osoitin paikkavektoriin
loc *ploc = (*text_map)[ query_text ];

// käy läpi paikkojen tietoparit
// lisää jokaisen rivin arvo joukkoon
set< short > occurrence_lines;
loc::iterator liter = ploc->begin(),
              liter_end = ploc->end();

while ( liter != liter_end ) {
      occurrence_lines.insert( occurrence_lines.end(),
                               (*liter).first );
      ++liter;
}
```

Joukko sisältää vain yhden esiintymän jokaisesta avainarvosta. Tästä syystä taataan, että occurrence_lines sisältää rivistä vain yhden ilmentymän, jossa sana esiintyy. Tulostaaksemme nämä tekstirivit, käymme yksinkertaisesti läpi joukon:

```
register int size = occurrence_lines.size();
cout << "\n" << query_text
    << " occurs " << size
    << (size == 1 ? " time:" : " times:")
    << "\n\n";

set< short >::iterator it=occurrence_lines.begin();
for ( ; it != occurrence_lines.end(); ++it ) {
     int line = *it;

     cout << "\t( line "
          << line + 1 << " ) "
          << (*text_file)[line] << endl;
}
```

(query_text()-funktion täydellinen toteutus on esitetty seuraavassa kohdassa.)

Joukko tukee operaatioita size(), empty() ja erase() samalla tavalla kuin karttatyyppi, joka esiteltiin edellisessä kohdassa. Geneerisissä algoritmeissa on lisäksi joukkokohtaisia funktioita kuten set_union() ja set_difference(). (Käytämme näitä luvussa 17 kyselykielemme tukena.)

Harjoitus 6.23

Lisää poisjätettävä joukko sanoille, joiden lopussa olevaa kirjainta 's' ei tulisi poistaa, mutta joille ei ole olemassa yleissääntöä. Esimerkiksi kolme sanaa, jotka tulisi sijoittaa tähän joukkoon, ovat kelvollisia sanoja: Pythagoras, Brahms ja Burne_Jones. Lisää tämän poisjätettävän joukon käsittely kohdan 6.10 suffix_s()-funktioon.

Harjoitus 6.24

Määrittele vektori kirjoista, jotka haluaisit lukea seuraavan kuuden kuukauden aikana ja joukko kirjojen nimistä, jotka olet lukenut. Kirjoita ohjelma, joka valitsee seuraavan kirjan sinulle vektorista edellyttäen, että et ole vielä sitä lukenut. Kun se palauttaa valitsemansa kirjan sinulle, tulisi sen lisätä kirjan nimi lukemiesi kirjojen joukkoon. Jos kuitenkin päädyt laittamaan kirjan sivuun, ohjelmoi tuki sille, että voit poistaa kirjan nimen lukemiesi kirjojen joukosta. Tulosta kuuden kuukauden kuluttua lukemasi kirjat ja ne, joita et ole lukenut.

6.14 Koko ohjelma

Nyt esitellään koko tässä luvussa kehitetty, toimiva ohjelma parilla muutoksella: sen sijaan, että säilyttäisimme proseduraalisen erillisten tietorakenteiden ja funktioiden järjestyksen, olemme esitelleet TextQuery-luokan, joka kapseloi molemmat (katsomme luokan käyttöä tähän yksityiskohtaisemmin tulevissa luvuissa) ja esitämme tekstin kuten se muokattiin käännettäväksi nykyisiin käytettävissä oleviin toteutuksiin. Esimerkiksi iostream-kirjasto noudattaa esistandardin toteutusta. Mallit eivät tue malliparametrien oletusargumentteja. Jotta voisit ajaa ohjelmaa nykyisessä järjestelmässäsi, voi olla tarpeen, että muokkaat esittelyitä sieltä täältä.

```
// vakiokirjaston otsikkotiedostot
#include <algorithm>
#include <string>
#include <vector>
#include <utility>
#include <map>
#include <set>

// esistandardin iostream-kirjaston otsikkotiedosto
#include <fstream.h>

// C-standardin otsikkotiedostoja
#include <stddef.h>
#include <ctype.h>

// typedef saa esittelyt helpommiksi
typedef pair<short,short>          location;
typedef vector<location,allocator>  loc;
typedef vector<string,allocator>    text;
typedef pair<text*,loc*>           text_loc;

class TextQuery {
public:
    TextQuery() { memset( this, 0, sizeof( TextQuery )); }

    static void
      filter_elements( string felems ) { filt_elems = felems; }

    void query_text();
    void display_map_text();
    void display_text_locations();
    void doit() {
          retrieve_text();
          separate_words();
          filter_text();
          suffix_text();
          strip_caps();
          build_word_map();
  }

private:

    void retrieve_text();
    void separate_words();
    void filter_text();
    void strip_caps();
    void suffix_text();
    void suffix_s( string& );
    void build_word_map();
```

```
private:
    vector<string,allocator>    *lines_of_text;
    text_loc                    *text_locations;
    map< string,loc*,
     less<string>,allocator> *word_map;
    static string                                   filt_elems;
};

string TextQuery::filt_elems( "\",.;:!?)(\\/" );

int main()
{
    TextQuery tq;
    tq.doit();
    tq.query_text();
    tq.display_map_text();
}

void
TextQuery::
retrieve_text()
{
    string file_name;

    cout << "please enter file name: ";
    cin  >> file_name;

    ifstream infile( file_name.c_str(), ios::in );
    if ( !infile ) {
        cerr << "oops! unable to open file "
             << file_name << " -- bailing out!\n";
        exit( -1 );
    }
    else cout << "\n";

    lines_of_text = new vector<string,allocator>;
    string textline;

    while ( getline( infile, textline, '\n' ))
          lines_of_text->push_back( textline );
}

void
TextQuery::
separate_words()
{
    vector<string,allocator> *words = new vector<string,allocator>;
    vector<location,allocator> *locations =
```

```
                new vector<location,allocator>;

    for ( short line_pos = 0; line_pos < lines_of_text->size();
            line_pos++ )
    {
            short  word_pos = 0;
            string textline = (*lines_of_text)[ line_pos ];

            string::size_type eol = textline.length();
            string::size_type pos = 0, prev_pos = 0;

            while (( pos = textline.find_first_of( ' ', pos ))
                        != string::npos )
            {
                    words->push_back(
                            textline.substr( prev_pos, pos - prev_pos ));
                    locations->push_back(
                            make_pair( line_pos, word_pos ));

                        word_pos++; pos++; prev_pos = pos;
            }

            words->push_back(
                    textline.substr( prev_pos, pos - prev_pos ));

            locations->push_back(make_pair(line_pos,word_pos));
    }

    text_locations = new text_loc( words, locations );
}

void
TextQuery::
filter_text()
{
    if ( filt_elems.empty() )
        return;

    vector<string,allocator> *words = text_locations->first;

    vector<string,allocator>::iterator iter = words->begin();
    vector<string,allocator>::iterator iter_end = words->end();

    while ( iter != iter_end )
    {
            string::size_type pos = 0;
            while (( pos = (*iter).find_first_of( filt_elems, pos ))
                        != string::npos )
                        (*iter).erase(pos,1);
```

```
            ++iter;
    }
}

void
TextQuery::
suffix_text()
{
    vector<string,allocator> *words = text_locations->first;

    vector<string,allocator>::iterator iter = words->begin();
    vector<string,allocator>::iterator iter_end = words->end();

    while ( iter != iter_end )
    {
        if ( (*iter).size() <= 3 )
                { iter++; continue; }

        if ( (*iter)[ (*iter).size()-1 ] == 's' )
                suffix_s( *iter );

        // loppuliitteiden lisäkäsittely tulee tähän ...

        iter++;
    }
}

void
TextQuery::
suffix_s( string &word )
{
    string::size_type spos = 0;
    string::size_type pos3 = word.size()-3;

    // "ous", "ss", "is", "ius"
    string suffixes( "oussisius" );

    if ( ! word.compare( pos3, 3, suffixes, spos, 3 ) ||
        ! word.compare( pos3, 3, suffixes, spos+6, 3 ) ||
        ! word.compare( pos3+1, 2, suffixes, spos+2, 2 ) ||
        ! word.compare( pos3+1, 2, suffixes, spos+4, 2 ))
                    return;

    string ies( "ies" );
    if ( ! word.compare( pos3, 3, ies ))
    {
        word.replace( pos3, 3, 1, 'y' );
        return;
    }
```

```
    string ses( "ses" );
    if ( ! word.compare( pos3, 3, ses ))
    {
        word.erase( pos3+1, 2 );
        return;
    }

    // poista lopusta 's'
    word.erase( pos3+2 );

    // ole varuillasi näistä: "'s"
    if ( word[ pos3+1 ] == '\'' )
        word.erase( pos3+1 );
}

void
TextQuery::
strip_caps()
{
    vector<string,allocator> *words = text_locations->first;

    vector<string,allocator>::iterator iter = words->begin();
    vector<string,allocator>::iterator iter_end = words->end();

    string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

    while ( iter != iter_end ) {
        string::size_type pos = 0;
        while (( pos = (*iter).find_first_of( caps, pos ))
                != string::npos )
            (*iter)[ pos ] = tolower( (*iter)[pos] );
        ++iter;
    }
}


void
TextQuery::
build_word_map()
{
    word_map = new map< string, loc*, less<string>, allocator >;

    typedef map<string,loc*,less<string>,allocator>::value_type
            value_type;

    typedef set<string,less<string>,allocator>::difference_type
            diff_type;
```

```
set<string,less<string>,allocator> exclusion_set;

ifstream infile( "exclusion_set" );
if ( !infile )
{
     static string default_excluded_words[25] = {
       "the","and","but","that","then","are","been",
       "can","can't","cannot","could","did","for",
       "had","have","him","his","her","its","into",
       "were","which","when","with","would"
     };

     cerr << "warning! unable to open word exclusion file! -- "
      << "using default set\n";

     copy( default_excluded_words, default_excluded_words+25,
      inserter( exclusion_set, exclusion_set.begin() ));
}
else {
     istream_iterator< string, diff_type >
               input_set( infile ), eos;

     copy( input_set, eos,
      inserter( exclusion_set, exclusion_set.begin() ));
}

// käy sanat läpi lisäten avainpareja

vector<string,allocator> *text_words  = text_locations->first;
vector<location,allocator> *text_locs = text_locations->second;

register int elem_cnt = text_words->size();
for ( int ix = 0; ix < elem_cnt; ++ix )
{
     string textword = ( *text_words )[ ix ];

     if ( textword.size() < 3 ||
          exclusion_set.count( textword ))
                    continue;

     if ( ! word_map->count((*text_words)[ix] ))
     {  // ei löydy, lisää se:
       loc *ploc = new vector<location,allocator>;
       ploc->push_back( (*text_locs)[ix] );
       word_map->insert( value_type( (*text_words)[ix], ploc ));
     }
     else (*word_map)[(*text_words)[ix]]->
                         push_back( (*text_locs)[ix] );
}
```

```
}

void
TextQuery::
query_text()
{
    string query_text;

    do {
        cout << "enter a word against which to search the text.\n"
             << "to quit, enter a single character ==>  ";
        cin  >> query_text;

        if ( query_text.size() < 2 ) break;

        string caps( "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
        string::size_type pos = 0;
        while (( pos = query_text.find_first_of( caps, pos ))
                 != string::npos )
               query_text[ pos ] = tolower( query_text[pos] );

        // jos indeksoimme karttaa, query_text lisätään, ellei sitä löydy
        // ei ole ollenkaan sitä, mitä haluaisimme ...

        if ( !word_map->count( query_text )) {
            cout << "\nSorry. There are no entries for "
                 << query_text << ".\n\n";
            continue;
        }

        loc *ploc = (*word_map)[ query_text ];

        set<short,less<short>,allocator> occurrence_lines;
        loc::iterator liter = ploc->begin(),
                      liter_end = ploc->end();

        while ( liter != liter_end ) {
          occurrence_lines.insert(
            occurrence_lines.end(), (*liter).first);
          ++liter;
        }

        register int size = occurrence_lines.size();
        cout << "\n" << query_text
             << " occurs " << size
             << (size == 1 ? " time:" : " times:")
             << "\n\n";

        set<short,less<short>,allocator>::iterator
```

```
            it=occurrence_lines.begin();
        for ( ; it != occurrence_lines.end(); ++it ) {
            int line = *it;

            cout << "\t( line "
                // don't confound user with
                // text lines starting at 0
                << line + 1 << " ) "
                << (*lines_of_text)[line] << endl;
        }

        cout << endl;
    }
    while ( ! query_text.empty() );
    cout << "Ok, bye!\n";
}

void
TextQuery::
display_map_text()
{
    typedef map<string,loc*,less<string>,allocator> map_text;
    map_text::iterator iter = word_map->begin(),
                       iter_end = word_map->end();

    while ( iter != iter_end ) {
        cout << "word: " << (*iter).first << " (";

        int          loc_cnt = 0;
        loc          *text_locs = (*iter).second;
        loc::iterator liter     = text_locs->begin(),
                      liter_end = text_locs->end();

        while ( liter != liter_end )
        {
            if ( loc_cnt )
                cout << ",";
            else ++loc_cnt;

            cout << "(" << (*liter).first
                << "," << (*liter).second << ")";

            ++liter;
        }

        cout << ")\n";
        ++iter;
    }
```

```
    cout << endl;
}

void
TextQuery::
display_text_locations()
{
    vector<string,allocator>   *text_words  = text_locations->first;
    vector<location,allocator> *text_locs   = text_locations->second;

    register int elem_cnt = text_words->size();

    if ( elem_cnt != text_locs->size() )
    {
        cerr << "oops! internal error: word and position vectors "
         << "are of unequal size\n"
         << "words: " << elem_cnt << " "
         << "locs: "  << text_locs->size()
         << " -- bailing out!\n";
        exit( -2 );
    }

    for ( int ix = 0; ix < elem_cnt; ix++ )
    {
        cout << "word: " << (*text_words)[ ix ] << "\t"
            << "location: ("
            << (*text_locs)[ix].first  << ","
            << (*text_locs)[ix].second << ")"
            << "\n";
    }

    cout << endl;
}
```

Harjoitus 6.25

Selitä, miksi pitää käyttää erityistä inserter-iteraattoria poisjätettävien sanojen joukon täyttämisessä. (Se on lyhyesti selitettynä kohdassa 6.13.1 ja käsitellään yksityiskohtaisesti kohdassa 12.4.1.)

```
set<string> exclusion_set;
ifstream    infile( "exclusion_set" );

// ...

copy( default_excluded_words, default_excluded_words+25,
    inserter( exclusion_set, exclusion_set.begin() ));
```

Harjoitus 6.26

Alkuperäinen toteutuksemme noudattelee proseduraalista ratkaisua — eli globaalisia funktiokokoelmia ja riippumattomia, kapseloimattomia tietorakenteita. Lopullinen ohjelmamme noudattaa vaihtoehtoista ratkaisua, jossa olemme sijoittaneet funktiot ja tietorakenteet TextQuery-luokkaan. Vertaile näitä kahta eri lähestymistapaa. Mitkä ovat kummankin heikkoudet ja vahvuudet?

Harjoitus 6.27

Tässä ohjelmaversiossa käyttäjältä kysytään käsiteltävän tiedoston nimeä. Kätevämpi toteutus antaisi käyttäjän määrittää tiedoston ohjelman komentoriville — katsomme luvussa 7, kuinka tuemme ohjelman komentorivin argumentteja. Mitä muita komentorivin valitsimia tulisi ohjelman tukea?

6.15 Monikartta/monijoukko

Sekä kartta että joukko voivat sisältää vain yhden ilmentymän jokaisesta avaimesta. Monijoukkoon ja monikarttaan voidaan tallentaa useita avaimen ilmentymiä. Esimerkiksi puhelinluettelosta joku voisi haluta listauksen jokaisesta puhelinnumerosta nimineen. Kirjailijalta saatavilla olevista teksteistä voisi tuottaa erillisen listauksen kirjojen nimien perusteella tai tekstistä erillisen listauksen sanojen esiintymispaikoista. Jotta monikarttaa tai monijoukkoa voidaan käyttää, pitää ottaa mukaan niihin liittyvä map- tai set-otsikkotiedosto.

```
#include <map>
multimap< key_type, value_type > multimapName;

// indeksoitu merkkijonolla, sisältää list< string >
multimap< string, list< string > > synonyms;

#include <set>
multiset< type > multisetName;
```

Sekä monikartalle että monijoukolle on käytettävissä eräs läpikäyntistrategia, jossa käytetään kombinaatiota iteraattorista, jonka `find()` palauttaa (osoittaa ensimmäiseen ilmentymään) ja arvosta, jonka `count()` palauttaa. (Tämä toimii, koska taataan, että ilmentymät esiintyvät yhtenäisesti säilössä.) Esimerkiksi:

```
#include <map>
#include <string>

void code_fragment()
{
    multimap< string, string > authors;
    string search_item( "Alain de Botton" );
    // ...
```

```
    int number = authors.count( search_item );
    multimap< string,string >::iterator iter;

    iter = authors.find( search_item );
    for ( int cnt = 0; cnt < number; ++cnt, ++iter )
        do_something( *iter );

    // ...
}
```

Hieman elegantimpi, vaihtoehtoinen strategia on käyttää iteraattoriparin arvoja, jotka palauttaa erityinen monijoukon ja monikartan operaatio, equal_range(). Jos arvo löytyy, ensimmäinen iteraattori osoittaa ensimmäisen arvon esiintymään ja toinen osoittaa yhden yli viimeisen arvon esiintymän. Jos viimeinen esiintymä on monijoukon viimeinen elementti, toisen iteraattorin arvoksi asetetaan end(). Esimerkiksi:

```
#include <map>
#include <string>
#include <utility>

void code_fragment()
{
    multimap< string, string > authors;
    // ...
    string search_item( "Haruki Murakami" );

    while ( cin && cin >> search_item )
        switch ( authors.count( search_item ))
        {
            // ei löydy, siirry seuraavaan
            case 0:
                break;

            // yksi löytyi. tavallinen find()
            case 1: {
                multimap< string,string >::iterator iter;
                iter = authors.find( search_item );
                // tee jotain elementille
                break;
            }
            // löytyi useita ilmentymiä ...
            default:
            {
                typedef multimap< string,string >::iterator iterator;
                pair< iterator, iterator > pos;

                // pos.first osoittaa ensimmäistä ilmentymää
                // pos.second osoittaa positioon, jossa
                //     ei enää ole arvoa
```

```
                    pos = authors.equal_range( search_item );
                    for ( ; pos.first != pos.second; pos.first++ )
                        // tee jotain jokaiselle elementille
            }
          }
      }
```

Lisäys ja poisto ovat samanlaisia kuin yksinkertaisemmilla kartan ja joukon assosiatiivisilla säiliötyypeillä. equal_range() on hyödyllinen annettaessa tarpeellinen iteraattoripari, joka ilmaisee poistettavaa usean elementin aluetta. Esimerkiksi:

```
#include <multimap>
#include <string>

typedef multimap< string, string >::iterator iterator;
pair< iterator, iterator > pos;
string search_item( "Kazuo Ishiguro" );

// authors on monikartta: multimap<string,string>
// tämä on yhtä kuin
// authors.erase( search_item );
pos = authors.equal_range( search_item );
authors.erase( pos.first, pos.second );
```

Lisäys saa aikaan uuden elementin joka kerta. Esimerkiksi:

```
typedef multimap<string,string>::value_type valType;
multimap<string,string> authors;

// esittelee ensimmäisen Barth-avaimen
authors.insert( valType(
  string( "Barth, John" ),
  string( "Sot-Weed Factor" )));

// esittelee toisen Barth-avaimen
authors.insert( valType(
  string( "Barth, John" ),
  string( "Lost in the Funhouse" )));
```

Eräs monikartan elementin käsittelyä rajoittava asia on indeksioperaattorin tuen puuttuminen. Esimerkiksi seuraava

```
authors[ "Barth, John" ]; // virhe: monikartta
```

johtaa käännöksenaikaiseen virheeseen.

Harjoitus 6.28

Toteuta kohdan 6.14 tekstinkyselyohjelma uudelleen käyttämään monikarttaa, jossa jokainen paikka lisätään erikseen. Millaisia ovat suorituskyvyn ja suunnittelun piirteet näissä kahdessa ratkaisussa? Kumpaa pidät parempana ratkaisuna? Miksi?

6.16 Pino

Kohdassa 4.5 kuvasimme lisäys- ja vähennysoperaattoreita toteuttamalla pinoabstraktion. Yleensä pinot tarjoavat tehokkaan ratkaisun ongelmaan, jossa halutaan pitää yllä nykyistä tilaa, kun useita eri tiloja voi tapahtua yhtä aikaa ohjelman suorituksen aikana. Koska pino on tärkeä tietoabstraktio, C++-vakiokirjastossa on sille luokkatoteutus. Jotta sitä voidaan käyttää, pitää ottaa mukaan siihen liittyvä otsikkotiedosto:

```
#include <stack>
```

Vakiokirjaston pino on toteutettu hieman eri tavalla kuin meidän versiomme, niin että ylimmän elementin käsittely ja poisto ovat vastaavasti erillisessä top()- ja pop()-operaatioparissa. Pinosäiliön tukemien operaatioiden täydellinen luettelo on seuraavassa:

Taulukko 6.5 Operaatiot, joita pinosäiliö tukee

Operaatio	Toiminto
empty()	palauttaa arvon true, jos pino on tyhjä; muutoin false.
size()	palauttaa pinon elementtien lukumäärän.
pop()	poistaa pinon ylimmän elementin, muttei palauta sitä.
top()	palauttaa pinon ylimmän elementin, muttei poista sitä.
push(item)	sijoittaa uuden ylimmän elementin pinoon.

Seuraavassa ohjelmassa kokeillaan näitä viittä eri pino-operaatiota:

```
#include <stack>
#include <iostream>

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

```
// täytä pino
int ix = 0;
stack< int > intStack;
for ( ; ix < ia_size; ++ix )
        intStack.push( ia[ ix ] );

int error_cnt = 0;
if ( intStack.size() != ia_size ) {
        cerr << "oops! invalid intStack size: "
                << intStack.size()
                << "\t expected: " << ia_size << endl;
        ++error_cnt;
}

int value;
while ( intStack.empty() == false )
{
            // lue pinon ylin elementti
            value = intStack.top();

            if ( value != --ix ) {
                 cerr << "oops! expected " << ix
                     << " received " << value
                     << endl;
                 ++error_cnt;
            }

            // hae ylin elementti ja toista
            intStack.pop();
}

cout << "Our program ran with "
        << error_cnt << " errors!" << endl;
}
```

Seuraava

```
stack< int > intStack;
```

esittelee intStack:in tyhjäksi kokonaislukuelementtien pinoksi. Pinotyyppiä sanotaan *säiliösovittimeksi* (*container adapter*), koska se käyttää hyväkseen pinoabstraktioon taustalla olevaa säiliökokoelmaa. Oletusarvo on, että pino toteutetaan pakan (*deque*) säiliötyypillä, koska pakalla on tehokkaat lisäys- ja poisto-operaatiot säiliön alkupäälle, eikä vektori tue sitä. Jos haluaisimme korvata oletusarvon, voisimme määritellä pino-olion, joka saa eksplisiittisen jonosäiliötyypin toisena argumenttinaan. Esimerkiksi:

```
stack< int, list<int> > intStack;
```

Pinon elementit lisätään arvon perusteella; jokainen olio kopioidaan taustalla olevaan säiliöön. Suurilla tai monimutkaisilla olioilla tämä lähestymistapa voi osoittautua liian raskaaksi etenkin, jos vain luemme elementtejä. Vaihtoehtoinen tallennusstrategia on määritellä osoitinpino. Esimerkiksi:

```
#include <stack>

class NurbSurface { /* muminaa... */ };
stack< NurbSurface* > surf_Stack;
```

Kahta samantyyppistä pinoa voidaan verrata, ovatko ne yhtäsuuria, erisuuria, toinen pienempi kuin toinen, toinen suurempi kuin toinen, toinen pienempi tai yhtäsuuri kuin toinen ja toinen suurempi tai yhtäsuuri kuin toinen, edellyttäen, että taustalla oleva elementtityyppi tukee yhtäsuuri kuin- ja pienempi kuin -operaattoreita. Näissä operaatioissa jokaista elementtiä verrataan vuorollaan. Ensimmäinen erisuuri elementtipari päättää pienempi kuin- tai suurempi kuin -vertailusuhteen.

Kuvaamme pinon käyttöä ohjelmassa kohdassa 17.7, kun tuemme monimutkaisia käyttäjän kyselyitä tekstistä, kuten

```
Civil && ( War || Rights )
```

6.17 Jono ja prioriteettijono

Jonoabstraktio tuo esille tallennuksen ja hakemisen FIFO-periaatteen — ensiksi sisään, ensiksi ulos (*first in, first out*). Jonoon lisättävät oliot sijoitetaan taakse. Seuraava haettava olio otetaan jonon edestä. Vakiokirjastossa on kaksi erityylistä jonoa: FIFO-jono, jota kutsumme yksinkertaisesti jonoksi (*queue*) ja prioriteettijono (*priority queue*).

Prioriteettijono mahdollistaa, että käyttäjä voi priorisoida jonossa olevia olioita. Sen sijaan, että vasta lisätty olio laitettaisiin jonon hännille, olio sijoitetaankin niiden olioiden edelle, joilla on alhaisempi prioriteetti. Käyttäjä, joka määrittelee prioriteettijonon, päättää, kuinka prioriteetti määrätään. Todellisen elämän esimerkki prioriteettijonosta on lentokentän lähtöselvitysjono. Ne, joiden lento lähtee 15 minuutin sisällä, siirretään yleensä jonon eteen niin, että he ehtivät lähtöselvitysprosessista ennen koneen lähtöä. Ohjelmointiesimerkki prioriteettijonosta on käyttöjärjestelmän ajastin, joka päättelee, mikä lukuisista odottavista prosesseista tulisi suorittaa seuraavaksi.

Jotta jonoa tai prioriteettijonoa voidaan käyttää, pitää ottaa mukaan niihin liittyvä otsikkotiedosto:

```
#include <queue>
```

Jono- ja prioriteettijonosäiliöiden tukemien operaatioiden täydellinen luettelo on nähtävissä taulukossa 6.6:

Taulukko 6.6 Operaatiot, joita jono ja prioriteettijono tukevat

Operaatiot	Toiminto
empty()	palauttaa arvon true, jos jono on tyhjä; muutoin false.
size()	palauttaa jonossa olevien elementtien lukumäärän.
pop()	poistaa elementin jonon edestä, muttei poista sitä. Kun kyseessä on prioriteettijono, alun elementtiä edustaa se, jolla on korkein prioriteetti.
front()	palauttaa elementin jonon edestä, muttei poista sitä. Tätä voidaan käyttää vain yksinkertaiseen jonoon.
back()	palauttaa elementin jonon takaa, muttei poista sitä. Tätä voidaan käyttää vain yksinkertaiseen jonoon.
top()	palauttaa suurimman prioriteetin omaavan elementin prioriteettijonosta, muttei poista sitä. Tätä voidaan käyttää vain prioriteettijonoon.
push(item)	sijoittaa uuden elementin jonon taakse. Kun kyseessä on prioriteettijono, olion järjestys perustuu sen prioriteettiin.

Prioriteettijonon elementit järjestetään suurimmasta pienimpään siten, että suurimmalla on korkein prioriteetti. Oletusarvo on, että elementtien prioriteetti toteutetaan taustalla olevan elementtityypin pienempi kuin -operaattorilla. Jos haluaisimme korvata oletusarvoisen pienempi kuin -operaattorin, voimme eksplisiittisesti määrätä funktion tai funktio-olion käytettäväksi prioriteettijonon elementtien järjestämiseen (Kohdassa 12.3 selitetään ja kuvataan tätä enemmän).

6.18 Palaaminen iStack-luokkaamme

iStack-luokka, joka esiteltiin kohdassa 4.15, sisältää rajoituksen kahdessa mielessä:

1. Se tukee ainoastaan yhtä tyyppiä: int-tyyppiä. Pitäisimme parempana, jos se tukisi kaikkia elementtityyppejä. Voimme tehdä sen muuttamalla toteutuksemme tukemaan yleistä Stack-luokkamallia.
2. Sen pituus on kiinteä. Tämä on ongelmallista kahdessa mielessä: pinomme voi täyttyä, jolloin siitä tulee käyttökelvoton, ja jotta voimme estää sen, päädymme varaamaan keskimäärin huomattavasti enemmän muistia kuin on tarve. Ratkaisu on tukea dynaamisesti kasvavaa pinoa. Voimme tehdä tämän käyttämällä hyödyksi suoraan taustalla olevan vektoriolion dynaamisuutta.

Ennen kuin aloitamme, tässä on alkuperäinen iStack-luokkamme määrittely:

```
#include <vector>

class iStack {
public:
   iStack( int capacity )
         : _stack( capacity ), _top( 0 ) {};

   bool pop( int &value );
   bool push( int value );

   bool full();
   bool empty();
   void display();

   int size();

private:
   int _top;
   vector< int > _stack;
};
```

Muuttakaamme luokkaa niin, että se tukee dynaamista muistinvarausta. Itse asiassa tämä tarkoittaa, että elementtejä pitää lisätä ja poistaa sen sijaan, että indeksoisimme kiinteänkokoista vektoria. Tietojäsen `_top` ei enää ole tarpeellinen; funktioiden `push_back()` ja `pop_back()` käyttö hoitaa ylimmän elementin automaattisesti. Tässä on uudistettu toteutuksemme `pop()`- ja `push()`-operaatioista:

```
bool iStack::pop( int &top_value )
{
   if ( empty() )
       return false;
   top_value = _stack.back(); _stack.pop_back();
   return true;
}

bool iStack::push( int value )
{
   if ( full() )
       return false;
   _stack.push_back( value );
   return true;
}
```

`empty()`, `size()` ja `full()` pitää myös toteuttaa uudelleen — tässä versiossa kytkettynä tiukemmin taustalla olevaan vektoriin:

```
inline bool iStack::empty(){ return _stack.empty(); }
inline bool iStack::size() { return _stack.size();  }
inline bool iStack::full() {
     return _stack.max_size() == _stack.size();   }
```

display() vaatii hieman muutoksia, kun _top poistetaan silmukan loppuehdosta:

```
void iStack::display()
{
   cout << "( " << size() << " )( bot: ";
   for ( int ix = 0; ix < size(); ++ix )
       cout << _stack[ ix ] << " ";
   cout << " :top )\n";
}
```

Ainoa merkittävä suunnittelupäätös koskee uudistettua iStack-muodostajaa. Suoraan sanoen muodostajamme ei enää tarvitse tehdä mitään, jolloin seuraava tyhjä muodostin on riittävä uudistetulle iStack-luokallemme:

```
inline iStack::iStack() {}
```

Se ei kuitenkaan ole riittävää käyttäjillemme. Tässä vaiheessa olemme säilyttäneet täsmälleen alkuperäisen rajapinnan, joten mitään olemassaolevaa, käyttäjää koskevaa koodia ei tarvitse kirjoittaa uudelleen. Jotta tämä olisi täysin yhteensopiva alkuperäisen iStack-rajapinnan kanssa, pitää säilyttää muodostaja, jolle annetaan yksi argumentti, vaikka emme vaadikaan sitä, kuten alkuperäinen toteutuksemme teki. Muokkaamamme rajapinta hyväksyy, mutta ei vaadi, yhden argumentin, joka on tyyppiä int:

```
class iStack {
public:
   iStack( int capacity = 0 );
   // ...
};
```

Mitä me teemme argumentille, jos sellainen annetaan? Käytämme sitä vektorin kapasiteetin asettamiseen:

```
inline iStack::iStack( int capacity )
{
   if ( capacity )
       _stack.reserve( capacity );
}
```

Muunnos mallittomasta luokasta luokkamalliksi on vielä helpompaa; osittain siksi, koska taustalla oleva vektori jo kuuluu luokkamalliin. Tässä on uudistettu luokkamäärittelymme:

```
#include <vector>

template <class elemType>
class Stack {
public:
   Stack( int capacity=0 );

   bool pop( elemType &value );
   bool push( elemType value );
```

```
    bool full();
    bool empty();
    void display();

    int size();

  private:
    vector< elemType > _stack;
  };
```

Jotta säilyttäisimme yhteensopivuuden olemassaolevien ohjelmien kanssa, jotka käyttävät aikaisempaa iStack-luokkamme toteutusta, teemme seuraavan typedef-nimen:

```
typedef Stack<int> iStack;
```

Jätämme jäsenoperaatioiden uudistamisen harjoitukseksi.

Harjoitus 6.29

Toteuta peek()-funktio uudelleen (harjoitus 4.23 kohdassa 4.15) dynaamiseen Stack-luokkamalliimme.

Harjoitus 6.30

Uudista Stack-luokkamallimme jäsenoperaatiot. Suorita kohdan 4.15 testiohjelma uutta toteutusta vastaan.

Harjoitus 6.31

Käytä kohdan 5.11.1 List-luokkamallia hyväksi ja kapseloi Stack-luokkamallimme nimiavaruuteen Primer_Third_Edition.