5

Lauseet

C++-ohjelman pienin itsenäinen yksikkö on *lause* (*statement*). Tavallisessa kielessä vastaava rakenne on myös lause. Aivan kuten tavallisen kielen lauseet päättyvät pisteeseen, C++:n lauseet päättyvät yleensä puolipisteeseen. Täten lausekkeesta kuten `ival+5` tulee *yksinkertainen lause* (*simple statement*), kun se päätetään puolipisteellä. *Yhdistetty lause* (*compound statement*) on joukko yksinkertaisia lauseita, jotka on sijoitettu aaltosulkuparin sisään. Oletusarvoisesti lauseet suoritetaan siinä järjestyksessä kuin ne esiintyvät. Paitsi yksinkertaisimmissa ohjelmissa, ei peräkkäinen ohjelman suoritus kuitenkaan ole aina riittävä ongelmille, joita meidän pitää ratkaista. Erityiset *ohjelman kulkua ohjaavat* ohjelmalauseet, jotka perustuvat lausekkeen totuusarvoon, mahdollistavat ehdollisen tai toistetun suorituksen yksinkertaisille tai yhdistetyille lauseille. Ehdollista suoritusta tukevat *if-*, *if-else-* ja *switch*-lauseet. Toistosuoritusta tukevat *while-*, *do-while- ja for*-lauseet. Näitä jälkimmäisiä lauseita kutsutaan usein *silmukoiksi*. Tässä luvussa käsitellään yksityiskohtaisesti C++:n tukemia ohjelmalauseita.

5.1 Yksinkertaiset ja yhdistetyt lauseet

Yksinkertaisin ohjelmalauseen muoto on tyhjä lause eli null-lause. Sen muoto on seuraava (yksi puolipiste):

```
; // null-lause
```

Tyhjä lause on hyödyllinen silloin, kun kielen syntaksi vaatii lausetta tiettyyn kohtaan, mutta ohjelman logiikka ei. Esimerkiksi seuraavassa `while`-lauseessa kaikki tarvittava käsittely, joka tarvitaan C-tyylisen merkkijonon kopioimiseen toiseen, on toteutettu lauseen ns. *ehto*-osassa (tarkoittaa sulkujen sisällä). While-silmukan muoto kuitenkin vaatii, että ehdon jälkeen tulee lause. Koska mitään lisätyötä ei tarvita, täytämme syntaktisen vaatimuksen tyhjällä lauseella:

```
while ( *string++ = *inBuf++ )
        ; // tyhjä lause
```

Vahingossa jäänyt tarpeeton tyhjä lause ei aiheuta käännösvirhettä. Esimerkiksi seuraava

```
ival = dval + sval;; // ok: ylimääräinen tyhjä lause
```

muodostuu kahdesta lauseesta: lausekelauseesta, joka tekee sijoituksen ival-olioon ja tyhjästä lauseesta.

Yksinkertainen lause muodostuu yhdestä lauseesta. Esimerkiksi:

```
// yksinkertaisia lauseita

int ival = 1024;  // esittelylause
ival;                        // lausekelause
ival + 5;              // toinen lausekelause
ival = ival + 5;  // sijoituslause
```

Ehdolliset ja silmukointilauseet sallivat syntaktisesti vain yhden suoritettavan lauseen; käytännössä tämä on harvoin riittävä. Ohjelman logiikka usein vaatii, että täytyy suorittaa kaksi tai useampia lauseita. Sellaisissa tilanteissa käytetään *yhdistettyä lausetta* (*compound statement*) yhden lauseen sijasta. Esimerkiksi:

```
if ( ival0 > ival1 )
{
   // yhdistetty lause muodostuu yhdestä
   // esittelystä ja kahdesta sijoituslauseesta

   int temp = ival0;
   ival0 = ival1;
   ival1 = temp;
}
```

Yhdistetty lause on joukko lauseita, jotka on sijoitettu aaltosulkujen sisälle. Yhdistettyä lausetta kohdellaan kuin yhtä yksikköä ja se voi esiintyä missä tahansa ohjelmassa, jossa yksi lausekelause voi esiintyä. Syntaksiin on lisätty sellainen kätevä piirre, että yhdistettyä lausetta ei tarvitse päättää puolipisteellä.

Tyhjä yhdistetty lause on samanarvoinen tyhjän lauseen kanssa ja toimii vaihtoehtoisena syntaksina tyhjän lauseen käytölle, kuten seuraavassa:

```
while ( *string++ = *inBuf++ )
    {} // yhtä kuin tyhjä lause
```

Yhdistetty lause, joka sisältää yhden tai useamman esittelylauseen, kuten edellisessä esimerkissä, kutsutaan myös *lohkoksi* eli *lauselohkoksi* (*statement block*). Lohko esittelee paikallisen viittausalueen ohjelmassa; lohkossa esitellyt tunnukset kuten esimerkin temp ovat näkyviä vain tuossa lohkossa. Lohkoja, viittausaluetta ja olioiden elinaikaa katsotaan tarkemmin luvussa 8.

5.2 Esittelylause

C++:ssa olion määrittelyä kohdellaan kuten tässä

```
int ival;
```

kielen lauseena (kutsutaan *esittelylauseeksi* (*declaration statement*), vaikka *määrittelylause* (*definition statement*) on tarkempi tässä tapauksessa) ja voidaan yleensä sijoittaa ohjelmaan mihin tahansa, jossa lause on sallittu. Mietitäänpä esimerkiksi seuraavaa ohjelmaa (esittelylauseet on numeroitu seuraavasti: //#n, jossa n on numeroitu peräkkäin alkaen arvosta 1):

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

int main()
{
    string fileName; // #1

    cout << "Ole hyvä ja anna avattavan tiedoston nimi: ";
    cin >> fileName;

    if ( fileName.empty() ) {
     // kyllä, huolimatonta, mutta meillä on eräs syy näyttää:
     cerr << "Tiedostonimi on tyhjä, lopetetaan. Hei!\n";
     return -1;
    }

    ifstream inFile( fileName.c_str() ); // #2
    if ( ! inFile ) {
     cerr << "Tiedostoa ei pysty avaamaan, lopetetaan. Hei!\n";
     return -2;
    }

    string inBuf;          // #3
    vector< string > text; // #4

    while ( inFile >> inBuf ) {
       for ( int ix = 0; ix < inBuf.size(); ++ix ) // #5
            // ch on tässä tapauksessa tarpeeton,
            // mutta kuvaa jälleen asiaa
            if (( char ch = inBuf[ix] ) == '.' ) { // #6
                 ch = '_';
                 inBuf[ix] = ch;
            }
        text.push_back( inBuf );
    }
    if ( text.empty() )
       return 0;

    // yksi esittelylause, kaksi määrittelyä
    vector<string>::iterator iter = text.begin(), // #7
                             iend = text.end();
```

```
    while ( iter != iend ) {
        cout << *iter << '\n';
        ++iter;
    }

    return 0;
}
```

Ohjelma sisältää seitsemän esittelylausetta ja kahdeksan oliomäärittelyä. Esittelylauseet tuovat esille *esittelyiden paikallisuuden* — tarkoittaa, että esittelylauseet esiintyvät määriteltyjen olioiden ensimmäisen käytön lähialueella.

1970-luvulla tietokoneiden ohjelmointikielten suunnittelufilosofiat korostivat piirrettä, että kaikki oliot määriteltäisiin ohjelman, funktion tai lauselohkon alussa ennen yhtäkään ohjelmalausetta. (Esimerkiksi C:ssä olion määrittelyä ei pidetä kielen lauseena ja lohkossa kaikkien olioiden määrittelyiden pitää olla ennen yhtäkään ohjelmalausetta. C-ohjelmoijat ovat totuttautuneet pakon sanelemana määrittelemään kaikki oliot jokaisen lohkon alkuun.) Osittain tämä oli reaktio virhealttiille olion “lennossa” määrittelylle, jota FORTRAN tukee.

Koska olion määrittely on kielen lause, voidaan olioiden määrittelyjä sijoittaa minne tahansa, jossa kielen lauseita voi esiintyä. Syntaksin kannalta tämä on se, joka saa esittelyiden paikallisuuden mahdolliseksi.

Onko se välttämätöntä? Sisäisille tyypeille kuten kokonaisluvuille ja liukuluvuille esittelyiden paikallisuus on pääasiassa henkilökohtainen mieltymysasia. Kieli rohkaisee tähän sallimalla esittelyitä if-, else-if-, switch-, while- ja for-silmukan ehto-osassa (edellisessä esimerkissä on kaksi sellaista tapausta). Ne, jotka suosivat paikallisia esittelyitä, uskovat, että se saa ohjelmat helpommin luettaviksi.

Esittelyiden paikallisuus tulee välttämättömäksi luokkaolioiden ja niiden muodostajien sekä tuhoajien määrittelyissä. Kun sijoitamme näitä luokkaolioita funktion tai lauselohkon alkuun, tapahtuu kaksi asiaa:

1. Luokkaolioihin liittyvät muodostajat käynnistetään ennen kuin itse funktiossa tai lauselohkossa tehdään mitään. Esittelyn paikallisuus mahdollistaa, että voimme vähentää alustuksen määrää funktiossa tai lauselohkossa.
2. Ehkä vielä tärkeämpää on tämä: funktio tai lauselohko usein päättyy ennen kuin jokainen sen sisältämä lause on suoritettu. Esimerkiksi aikaisemmassa ohjelmassamme tuodaan esille kaksi epänormaalia päättymiskohtaa: virheet tiedostonimen saamisessa ja käyttäjän määrittelemän tiedoston avaamisessa. Kun määritellään luokan oliot kuten `inBuf` ja `text` ennen päättymiskohtia, voi se johtaa tarpeettomien muodostaja-tuhoaja-parien suorittamiseen. Kun luokkaolioita tai suorituksen kannalta raskaita muodostajia ja tuhoajia on paljon, voimme vaikuttaa epäsuotuisasti ohjelmiemme suorituksenaikaiseen tehokkuuteen. Tulos on yhä oikein, mutta suorituskyvystä tulee silloin tällöin riittämätön. (Tästä syystä C-ohjelmoijaekspertit, jotka

ovat tottuneet sijoittamaan määrittelyt funktioiden ja lauselohkojen alkuun, saattavat joskus huomata kirjoittamiensa C++-ohjelmien suoriutuvan tehtävistään tehottomammin kuin niiden vastaavat C-kielellä kirjoitetut versiot.)

Esittelylause voi muodostua yhdestä tai useammasta olion määrittelystä. Esimerkiksi ohjelmassamme määrittelemme kaksi vektori-iteraattoria samassa esittelylauseessa:

```
// yksi esittelylause, kaksi määrittelyä
vector<string>::iterator iter = text.begin(),
                         iend = text.end();
```

Seuraavat esittelylauseparit ovat samanarvoisia:

```
// samanarvoiset esittelylauseet
vector<string>::iterator iter = text.begin();
vector<string>::iterator iend = text.end();
```

Vaikka kumman tahansa valitseminen on pääosin henkilökohtainen makuasia, useiden oliomäärittelyiden sijoittaminen yhteen esittelylauseeseen on virhealttiimpaa, kun sekoitamme olioita, osoittimia ja viittauksia. Esimerkiksi seuraavassa esittelylauseessa ei ole selvää, onko käyttäjä aikonut määritellä yhden osoittimen ja yhden olion vai yksinkertaisesti määritellyt väärin toisen osoittimen olioksi (tunnuksen nimi vihjaa, että toinen määrittely on virheellinen):

```
// aikoiko ohjelmoija määritellä näin?
string *ptr1, ptr2;
```

Erilliset esittelylauseet jättävät vähemmän tilaa virheelle tässä tapauksessa:

```
string *ptr1;
string *ptr2;
```

Omassa koodissamme pyrimme ryhmittämään määrittelyt niin, että ne perustuvat olion aiottuun käyttöön. Esimerkiksi seuraavassa esittelylauseparissa

```
int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;
int charCnt=0, wordCnt=0;
```

ne oliot, joita aiotaan käyttää englannin kielen vokaaleille, on ryhmitetty yhteen esittelylauseeseen. Ne, jotka on tarkoitettu merkkien ja sanojen laskemiseen, on ryhmitetty toiseen esittelylauseeseen. Vaikka tämä lähestymistapa näyttää meistä täysin järkevältä, emme olisi voineet puolustella sen loogisuutta paremmin.

Harjoitus 5.1

Kuvitellaan, että sinusta on juuri tehty pienen ohjelmointiprojektin vetäjä ja haluat, että kaikki koodi noudattaa yhtenäistä esittelyjen menettelytapaa. Määrittele selkeästi ja puolustele esittelysäännöt, joita projektin haluat noudattavan.

Harjoitus 5.2

Kuvitellaan, että sinut on juuri sijoitettu harjoituksen 5.1 ryhmään. Olet täysin eri mieltä, et vain

ilmoitetusta esittelypolitiikasta, vaan myös kaikesta esittelypolitiikasta yleensä. Määrittele syysi selkeästi ja puolustele niitä.

5.3 If-lause

If-lause mahdollistaa lauseen tai lauselohkon ehdollisen suorituksen perustuen siihen, onko määritetty lauseke tosi. If-lauseen syntaktinen muoto on seuraava:

```
if ( ehto )
   lause
```

ehto-lausekkeen pitää olla sulkujen sisällä. Se voi olla joko lauseke, kuten tässä

```
if ( a + b > c ) { ... }
```

tai alustettu esittelylause, kuten tässä

```
if ( int ival = compute_value() ) { ... }
```

ehto-lausekkeessa määritelty olio on näkyvissä vain siihen liittyvässä lauseessa tai lauselohkossa. Esimerkiksi yritys käsitellä ival-oliota if-lauseen jälkeen saa aikaan käännöksenaikaisen virheen:

```
if ( int ival = compute_value() )
{
      // ival on näkyvissä vain
      // tämän if-lauselohkon sisällä
}

// virhe: ival ei ole näkyvissä
if ( ! ival ) ...
```

Kuvataksemme if-lauseen käyttöä, toteuttakaamme min()-funktio, joka palauttaa pienimmän arvon, jonka int-tyyppisten elementtien vektori sisältää. Lisäksi pidämme yllä pienimmän arvon esiintymislukumäärää vektorissa. Jokaiselle vektorin elementille pitää tehdä seuraavaa:

- Vertaaa elementtiä nykyiseen minimiarvoon.
- Jos se on pienempi kuin minimiarvo, sijoita tämä elementti uudeksi minimiarvoksi ja aseta laskurin arvoksi 1.
- Jos se on yhtä kuin minimiarvo, kasvata laskuria arvolla 1.
- Muussa tapauksessa älä tee mitään.
- Tutkittuasi jokaisen elementin, palauta arvo ja esiintymien lukumäärä käyttäjälle.

Tarvitaan kaksi if-lausetta:

```
if ( minVal > ivec[ i ]  ) ... // uusi minVal
if ( minVal == ivec[ i ] ) ... // toinen esiintymä
```

Jonkin verran yleinen ohjelmointivirhe on käyttää if-lausetta ilman yhdistettyä lausetta, kun ehtolause arvioidaan todeksi ja pitäisi suorittaa useampi lause. Tämän virheen keksiminen

on erittäin hankalaa, koska ohjelman teksti näyttää oikealta. Esimerkiksi:

```
if ( minVal > ivec[ i ] )
    minVal = ivec[ i ];
    occurs = 1; // tämä ei ole osa if-lausetta
```

Päinvastoin kuin ohjelmoija oli tarkoittanut ja sisentänyt,

```
occurs = 1;
```

ei tätä pidetä osana if-lausetta, vaan se suoritetaan ilman ehtoja if-lauseen arvioinnin jälkeen. Joten esiintymien lukumäärä asetetaan aina arvoon 1. Tässä on if-lause niin kuin ohjelmoija oli sen tarkoittanut (vasemman aaltosulun sijoituspaikka on loputon kiistelynaihe!):

```
if ( minVal > ivec[ i ] )
{
    minVal = ivec[ i ];
    occurs = 1;
}
```

Toinen if-lauseemme näyttää tältä:

```
if ( minVal == ivec[ i ] )
    ++occurs;
```

Huomaa, että if-lauseiden järjestys on merkitsevä. Funktiomme on aina yhden arvon väärässä, jos sijoitamme lauseet seuraavaan järjestykseen:

```
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
}

// potentiaalinen virhe, jos minVal:iin
// on juuri sijoitettu ivec[i]
if ( minVal == ivec[ i ] )
    ++occurs;
```

Molempien lauseiden suoritus samalla arvolla ei ole vain potentiaalisesti vaarallista, vaan myös turhaa. Sama elementti ei voi olla sekä pienempi että yhtäsuuri kuin minVal. Jos toinen ehdoista on tosi, voidaan toinen jättää turvallisesti huomiotta. If-lause mahdollistaa tällaisen joko-tai-ehdon käsittelyn else-lauseella.

If-else-lauseen syntaktinen muoto on seuraava:

```
if ( ehto )
    lause1
else
    lause2
```

Jos ehto on tosi, suoritetaan lause1; muussa tapauksessa suoritetaan lause2. Esimerkiksi:

```
if ( minVal == ivec[ i ] )
    ++occurs;
```

```
else
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
}
```

Tässä esimerkissä lause2 on itse if-lause. Jos minVal on pienempi kuin elementti, ei tehdä mitään toimenpiteitä.

Seuraavassa esimerkissä suoritetaan yksi lause kolmesta aina:

```
if ( minVal < ivec[ i ] )
    {} // tyhjä lause
else
if ( minVal > ivec[ i ] ) {
    minVal = ivec[ i ];
    occurs = 1;
}
else // minVal == ivec[ i ]
    ++occurs;
```

Tässä if-else-lause tuo esille potentiaalisen moniselitteisyyden, jota kutsutaan *"roikkuvan" else*-lauseen ongelmaksi. Tämä ongelma tulee esille, kun lause sisältää enemmän if-lauseita kuin else-lauseita. Silloin nousee esille kysymys: mihin if-lauseeseen ylimääräinen else-lause kuuluu? Esimerkiksi:

```
if ( minVal <= ivec[ i ] )
    if ( minVal == ivec[ i ] )
        ++occurs;
else {
    minVal = ivec[ i ];
    occurs = 1;
}
```

Sisennys ilmaisee ohjelmoijan uskomusta, että else-lauseen pitäisi kuulua ulompaan if-lauseeseen. C++:ssa kuitenkin roikkuvan else-lauseen moniselitteisyys on ratkaistu yhdistämällä else-lause viimeisimpään täsmäämättömään if-lauseeseen. Tässä tapauksessa if-lauseen arviointi on seuraava:

```
if ( minVal <= ivec[ i ] ) {
   // roikkuvan else-lauseen ratkaisun vaikutus
   if ( minVal == ivec[ i ] )
       ++occurs;
   else { minVal = ivec[ i ]; occurs = 1; }
}
```

Eräs menetelmä roikkuvan else-lauseen oletuskäyttäytymisen kiertämiseksi on sijoittaa viimeinen esiintyvä if-lause yhdistettyyn lauseeseen:

```
if ( minVal <= ivec[ i ] ) {
    if ( minVal == ivec[ i ] )
        ++occurs;
```

```
}
else { minVal = ivec[ i ]; occurs = 1; }
```

Jotkut koodaustyylit suosittelevat yhdistettyjen lauseiden aaltosulkujen käyttöä aina mahdollisten sekaannusten ja myöhempien koodimuutosten virheiden välttämiseksi.

Tässä on ensimmäinen toteutuksemme min()-funktiosta. Toinen argumentti, occurs, tulee sisältämään minimiarvojen lukumäärän; asetamme tämän funktiossa,

kuten myös päättelemme ja palautamme todellisen minimiarvon. Käytämme for-silmukkaa elementtien läpikäymiseen. (Valitettavasti toteutuksemme sisältää loogisen virheen — huomaatko sen?)

```
#include <vector>

int min( const vector<int> &ivec, int &occurs )
{
    int minVal = 0;
    occurs = 0;

    int size = ivec.size();

    for ( int ix = 0; ix < size; ++ix ) {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
        if ( minVal > ivec[ ix ] ) {
            minVal = ivec[ ix ];
            occurs = 1;
        }
    }

    return minVal;
}
```

Yleensä funktio palauttaa vain yhden arvon. Meidän vaatimuksemme on kuitenkin palauttaa, ei vain minimiarvoa, vaan myös sen esiintymien lukumäärä vektorissa. Lisäämme toteutukseemme viittausparametrin, jonka kautta välitämme toisen arvon (katso kohdasta 7.3 viittausparametrien käsittely). Kaikki sijoitukset occur-olioon min()-funktiossa vaikuttavat sen olion arvoon, joka todellisuudessa välitettiin argumenttina. Esimerkiksi:

```
int main()
{
    int occur_cnt = 0;
    vector< int > ivec;

    // ... täytä ivec

    // occur_cnt sisältää esiintymien lukumäärän, jotka on asetettu min()-funktiossa
    int minval = min( ivec, occur_cnt );
```

```
    // ...
}
```

Vaihtoehtoinen ratkaisu on käyttää olioparia (katso kohdasta 3.14 parityypin käsittely), joka sisältää kaksi kokonaislukuoliota: minimiarvon ja esiintymien lukumäärän. Funktio palauttaa sitten tämän olioparin ilmentymän. Esimerkiksi:

```
// vaihtoehtoinen toteutus
// palauttaa olioparin ...

#include <utility>
#include <vector>

typedef pair<int,int> min_val_pair;

min_val_pair
min( const vector<int> &ivec )
{
    int minVal = 0;
    int occurs = 0;

    // sama paluuseen saakka ...

    return make_pair( minVal, occurs );
}
```

Valitettavasti molemmat toteutuksemme min()-funktiosta ovat virheellisiä. Näetkö, mikä ongelma on? Aivan oikein: alustamme minVal-olion arvolla 0. Jos taulukon pienin arvo on suurempi kuin 0, toteutuksemme ei löydä sitä, vaan sen sijaan palauttaa arvon 0 ja occurs-olio asetetaan arvoon 0.

minVal-olion paras ensimmäinen arvo on taulukon ensimmäisen elementin arvo:

```
int minVal = ivec[0];
```

Tämä takaa, että min() palauttaa aina taulukon pienimmän arvon. Vaikka tämä korjaa ohjelmamme virheen, se tuo esille pienen tehottomuuden. Tässä on koodin kyseinen osa. Näetkö, mikä pienen tehokkuuden hinta on?

```
// min()-funktion uudistettu alkuosa
// tuo esille pienen tehottomuuden ...

int minVal = ivec[0];
occurs = 0;

int size = ivec.size();

for ( int ix = 0; ix < size; ++ix )
{
        if ( minVal == ivec[ ix ] )
            ++occurs;

    // ...
```

Koska ix alustetaan arvolla 0, silmukan ensimmäinen iteraatio huomaa aina, että minVal on yhtä kuin ivec[0]: arvo, jolla olemme alustaneet minVal-olion. Alustamalla ix:n arvolla 1 voimme välttää tarpeettoman vertailun minVal-olioon ja uudelleensijoituksen. Myönnettäköön, että tämä on pieni parannus, mutta se tuo esille vielä toisen virheen ohjelmastamme (ehkä meidän olisi pitänyt jättää asiat niin kuin ne olivat!). Näetkö, mitä on pielessä uudistetussa ohjelmassamme?

```
// min()-funktion uudistettu alkuosa
// tuo esille ohjelmavirheen ...

int minVal = ivec[0];
occurs = 0;

int size = ivec.size();

for ( int ix = 1; ix < size; ++ix )
{
    if ( minVal == ivec[ ix ] )
        ++occurs;
    // ...
```

Jos osoittautuu, että ivec[0] on minimiarvo, silloin ei koskaan aseteta occurs-oliota arvoon 1! Korjaus on tietysti helppo, mutta vasta sitten, kun näemme ensin sen välttämättömyyden:

```
int minVal = ivec[0];
occurs = 1;
```

Valitettavasti virheet kuten tämä eivät ole ollenkaan epätavallisia. Ohjelmoijina tulemme tekemään virheitä jossakin vaiheessa — ajoittain jopa typeriä virheitä. Sellaista sattuu. Tärkeä asia on hyväksyä, että virheitä tapahtuu ja olla varuillaan niiden varalta testaamalla ja katsomalla aina uudelleen koodia niin usein kuin tilanne sallii.

Seuraavassa ohjelmassa kokeillaan min()-funktiomme toteutusta:

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

int min( const vector< int > &ivec, int &occurs )
{
    int minVal = ivec[ 0 ];
    occurs = 1;

    int size = ivec.size();

    for ( int ix = 1; ix < size; ++ix )
    {
        if ( minVal == ivec[ ix ] )
            ++occurs;
        else
        if ( minVal > ivec[ ix ] ){
            minVal = ivec[ ix ];
            occurs = 1;
        }
    }

    return minVal;
}

int main()
{
    int ia[] = { 9,1,7,1,4,8,1,3,7,2,6,1,5,1 };
    vector<int> ivec( ia, ia+14 );

    int occurs = 0;
    int minVal = min( ivec, occurs );

    cout << "Minimiarvo: " << minVal
        << " esiintyy: " << occurs << " kertaa.\n";

    return 0;
}
```

Ohjelma generoi seuraavan tulostuksen:

```
Minimiarvo: 1 esiintyy: 5 kertaa.
```

Ehdollinen operaattori tarjoaa kätevän lyhennetyn merkintätavan yksinkertaiselle if-else-testille. Esimerkiksi seuraava min()-funktiomalli

```
template <class valueType>
inline const valueType&
min( valueType &val1, valueType &val2 )
{
   if ( val1 < val2 )
            return val1;
   return val2;
}
```

voidaan kirjoittaa vaihtoehtoisesti

```
template <class valueType>
inline const valueType&
min( valueType &val1, valueType &val2 )
{
   return ( val1 < val2 ) ? val1 : val2;
}
```

Pitkät if-else-lauseiden ketjut kuten seuraavassa voivat olla usein vaikeita lukea ja altistavat virheille muokkausten yhteydessä:

```
if ( ch == 'a' ||
     ch == 'A' )
       ++aCnt;
else
if ( ch == 'e' ||
     ch == 'E' )
       ++eCnt;
else
if ( ch == 'i' ||
     ch == 'I' )
       ++iCnt;
else
if ( ch == 'o' ||
     ch == 'O' )
       ++oCnt;
else
if ( ch == 'u' ||
     ch == 'U' )
       ++uCnt;
```

Vaihtoehtoinen rakenne if-else-lauseiden ketjuttamiselle on switch-lause, edellyttäen, että testattavat arvot ovat vakiolausekkeita kuten edellä testatut merkkivakiot. Switch-lause on seuraavan kohdan aiheena.

Harjoitus 5.3

Korjaa seuraavat:

```
(a) if ( ival1 != ival2 )
        ival1 = ival2
    else ival1 = ival2 = 0;

(b) if ( ival < minval )
        minval = ival;
        occurs = 1;

(c) if ( int ival = get_value())
        cout << "ival = "
             << ival << endl;

    if ( ! ival )
        cout << "ival = 0\n";

(d) if ( ival = 0 )
        ival = get_value();

(e) if ( ival == 0 )
    else ival = 0;
```

Harjoitus 5.4

Muuta min()-funktion argumenttiluettelon occurs-olion esittelyä niin, että sen argumenttityyppi ei ole viittaus, ja aja ohjelma uudelleen. Kuinka ohjelman käyttäytyminen muuttuu?

5.4 Switch-lause

Syvään sisäkkäin laitetut if-else-lauseet voivat usein olla syntaktisesti oikein, mutta eivät välttämättä ilmaise ohjelmoijan tarkoittamaa logiikkaa. Sisentämättömät else-if-vastaavuudet voivat jäädä huomaamatta. Lauseiden muokkausta on myös vaikeaa saada kerralla oikein. Vaihtoehtona menetelmälle, jossa voidaan valita jokin keskenään poissulkevista arvoista, on C++-kielessä switch-lause.

Kuvataksemme switch-lauseen käyttöä, miettikäämme seuraavaa ongelmaa: meitä on pyydetty laskemaan jokaisen viiden (englanninkielen) vokaalin esiintymät tekstin satunnaisista osista. (Perinteisesti tiedetään, että *e* on useimmin esiintyvä vokaali englanninkielessä.) Ohjelmamme logiikka näyttää tältä:

- Lue jokainen merkki vuorollaan, kunnes luettavia merkkejä ei enää ole.
- Vertaa jokaista merkkiä vokaaleihin.
- Jos merkki on joku vokaaleista, lisää arvo 1 tuon vokaalin laskuriin.
- Näytä tulokset.

Ohjelmaa käytettiin erään englanninkielisen tekstin analysointiin. Tässä on tulostus, joka vahvistaa, että tutkimustieto, joka koskee *e*-vokaalin esiintymistä, pitää paikkansa:

```
aCnt: 394
eCnt: 721
iCnt: 461
oCnt: 349
uCnt: 186
```

Switch-lause muodostuu seuraavista komponenteista:

1. switch-avainsanasta ja suluissa olevasta lausekkeesta, joka arvioidaan. Tässä tapauksessa lauseke on luettu merkki. Esimerkiksi:

```
char ch;
while ( cin >> ch )
        switch( ch )
```

2. case-otsikoista, jotka muodostuvat case-avainsanasta ja vakiolausekkeesta, jota verrataan switch-lausekkeen tulokseen; sekä kaksoispisteestä. Tässä tapauksessa jokainen case-otsikko edustaa yhtä englannin kielen viidestä vokaalista. Esimerkiksi:

```
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
```

3. Lausesarjoista, jotka liittyvät yhteen tai useampaan case-otsikkoon. Kun laskemme esimerkiksi jokaisen vokaalin lukumäärää, käytämme sijoituslausetta, joka kasvattaa vokaalin lukumäärää arvolla 1.
4. Valinnaisesta default-otsikosta. default-otsikko toimii eräänlaisena else-lauseena. Ellei switch-lausekkeen arvo täsmää yhdenkään case-otsikon arvon kanssa, suoritetaan default -otsikon jälkeen olevat lauseet. Jos esimerkiksi haluaisimme laskea merkkejä, jotka eivät ole vokaaleita, lisäisimme seuraavan default-otsikon ja lauseen:

```
default: // kaikki muut, jotka eivät ole vokaaleita
    ++non_vowel_cnt;
```

Arvon, joka on case-avainsanan jälkeen, pitää olla kokonaistyyppinen vakiolauseke. Esimerkiksi seuraava johtaa käännöksenaikaisiin virheisiin:

```
// kelpaamattomia case-otsikon arvoja
case 3.14: // muu kuin kokonaisluku
case ival: // muu kuin vakio
```

Lisäksi kahdella case-otsikolla ei saa olla samoja arvoja; jos on, saa se aikaan käännöksenaikaisen virheen.

switch-lauseke voi olla kuinka monimutkainen lauseke tahansa mukaan lukien funktion kutsun paluuarvo. switch-lausekkeen tuloksen arvoa verrataan jokaiseen case-otsikon arvoon, kunnes

vastaavuus löytyy tai kaikki otsikot on tutkittu. Jos lauseke vastaa jonkin case-otsikon arvoa, suoritus alkaa tuon otsikon jälkeisestä ensimmäisestä lauseesta. Ellei vastaavuutta löydy, suoritus alkaa default-otsikon jälkeisestä ensimmäisestä lauseesta, jos sellainen on mukana; muussa tapauksessa ohjelma jatkuu switch-lauseen jälkeisestä ensimmäisestä lauseesta.

Yleinen väärinkäsitys on, että vain ne lauseet suoritetaan, jotka liittyvät yhteen case-otsikkoon. Sen sijaan suoritus alkaa siitä, mutta jatkuu yli case-rajojen switch-lauseen loppuun saakka. Tämän väärinymmärrys on tae siitä, että olemme tehneet myös ohjelmamme väärin. Tässä on esimerkkinä virheellinen toteutus vokaaleja laskevasta switch-ohjelmastamme:

```
#include <iostream>
int main()
{
   char ch;
   int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0;

     while ( cin >> ch )
            // varoitus: tahallaan väärin!
            switch ( ch ) {
                  case 'a':
                        ++aCnt;
                  case 'e':
                        ++eCnt;
                  case 'i':
                        ++iCnt;
                  case 'o':
                        ++oCnt;
                  case 'u':
                        ++uCnt;
            }

     cout << "Vokaalin a lukumäärä: \t" << aCnt << '\n'
          << "Vokaalin e lukumäärä: \t" << eCnt << '\n'
          << "Vokaalin i lukumäärä: \t" << iCnt << '\n'
          << "Vokaalin o lukumäärä: \t" << oCnt << '\n'
          << "Vokaalin u lukumäärä: \t" << uCnt << '\n';
}
```

Jos ch sisältää arvon i, suoritus alkaa otsikon case ‘i’ jälkeen. Arvoa iCnt kasvatetaan. Suoritus ei kuitenkaan pysähdy tässä, vaan jatkaa yli case-rajojen switch-lauseen sulkevaan aaltosulkuun saakka. Myös arvoja oCnt ja uCnt kasvatetaan. Jos ch sisältää arvon, silloin kasvatetaan myös arvoja e, eCnt, iCnt, oCnt ja uCnt.

Ohjelmoijan pitää eksplisiittisesti kertoa kääntäjälle, että se pysäyttää lauseiden suorituksen switch-lauseessa. Tämä tapahtuu määrittämällä break-lause jokaisen suoritettavan yksikön jälkeen switch-lauseessa. Useimmissa tilanteissa case-otsikon viimeinen lause on break.

Kun break-lause kohdataan, switch-lause päättyy. Kontrolli siirtyy lauseeseen, joka esiintyy switch-lauseen päättävän aaltosulun jälkeen. Esimerkissämme kontrolli välitetään tulostuslauseelle. Korjattu switch-lauseemme on seuraava:

```
switch ( ch ) {
   case 'a':
            ++aCnt;
            break;
   case 'e':
            ++eCnt;
            break;
   case 'i':
            ++iCnt;
            break;
   case 'o':
            ++oCnt;
            break;
   case 'u':
            ++uCnt;
            break;
}
```

case-otsikko, josta jätetään tahallaan break-lause pois, tulisi useimmissa tapauksissa varustaa kommentilla, joka ilmaisee poisjättämisen olevan tarkoituksella tehty. Ohjelmiamme ei vain käännetä ja suoriteta, vaan jälkeemme tulevat ohjelmoijat myös lukevat niitä yhä uudelleen korjatakseen tai laajentaakseen alkuperäisiä toteutuksiamme. Koodia, joka on vastoin odotettua käyttöä, on erityisen vaikeaa ymmärtää, koska emme useinkaan ole varmoja, onko poikkeava menettely tarkoituksellista ja oikein vai huolimattomuutta ja mahdollinen virhe. Kommentti, joka ilmaisee ohjelmoijan aikomusta tällaisissa tapauksissa, parantaa merkittävästi koodin ylläpidettävyyttä.

Milloin ohjelmoija saattaisi haluta jättää break-lauseen pois case-otsikosta ja sallisi ohjelman *valuvan läpi* (*fall through*) useiden case-otsikoiden? Eräs tilanne on se, kun kaksi tai useampi arvo käsitellään samalla toimenpidejoukolla. Tämä on välttämätöntä, koska case-otsikkoon voidaan liittää vain yksi arvo. Siitä syystä ilmaisemme arvoalueen tyypillisesti pinoamalla case-otsikot toistensa perään. Jos esimerkiksi haluaisimme vain laskea kaikki kohdatut vokaalit yksittäisten sijasta, voisimme kirjoittaa seuraavasti:

```
int vowelCnt = 0;
// ...
switch ( ch )
{
    // mikä tahansa vokaalien a,e,i,o,u esiintyminen
    // saa aikaan vowelCnt-olion kasvattamisen
```

```
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

Vaihtoehtoisesti jotkut ohjelmoijat pitävät parempana case-otsikoiden ryhmittelyä korostaakseen, että ne edustavat arvoalueita, joita vastaan verrataan:

```
switch ( ch )
{
    // vaihtoehtoinen sallittu syntaksi
    case 'a': case 'e':
    case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

Juuri toteuttamassamme vokaaleja laskevassa ohjelmassamme on vielä yksi ongelma. Kuinka esimerkiksi ohjelma selviytyy seuraavasta syötöstä?

```
UNIX
```

Isoja kirjaimia *U* ja *I* ei tunnisteta vokaaleiksi. Ohjelmamme ei onnistu laskemaan vokaalien lukumäärä, jos kohdalle sattuu iso kirjain. Tässä on korjattu switch-lauseemme, joka myös käyttää hyväkseen läpivalumista:

```
switch ( ch ) {
   case 'a': case 'A':
          ++aCnt;
          break;
   case 'e':  case 'E':
          ++eCnt;
          break;
   case 'i':  case 'I':
          ++iCnt;
          break;
   case 'o':  case 'O':
          ++oCnt;
          break;
   case 'u':  case 'U':
          ++uCnt;
          break;
}
```

default-otsikko on samanarvoinen ehdottoman else-lauseen kanssa. Ellei yksikään case-otsikko vastaa switch-lausekkeen arvoa ja jos default-otsikko on mukana, suoritetaan default-otsikon jälkeiset lauseet. Esimerkkinä lisätkäämme default-otsikko switch-lauseeseen, joka laskee konsonanttien lukumäärää:

```
#include <iostream>
#include <ctype.h>
int main()
{
     char ch;
     int aCnt=0, eCnt=0, iCnt=0, oCnt=0, uCnt=0,
     consonantCnt = 0;

     while ( cin >> ch )
       switch ( ch )
    {
            case 'a': case 'A':
                 ++aCnt;
                 break;
            case 'e':  case 'E':
                 ++eCnt;
                 break;
            case 'i':  case 'I':
                 ++iCnt;
                 break;
            case 'o':  case 'O':
                 ++oCnt;
                 break;
            case 'u':  case 'U':
                 ++uCnt;
                 break;
            default:
                 if ( isalpha( ch ))
                     ++consonantCnt;
                 break;
        }

     cout << "Vokaalin a lukumäärä: \t" << aCnt << '\n'
          << "Vokaalin e lukumäärä: \t" << eCnt << '\n'
          << "Vokaalin i lukumäärä: \t" << iCnt << '\n'
          << "Vokaalin o lukumäärä: \t" << oCnt << '\n'
          << "Vokaalin u lukumäärä: \t" << uCnt << '\n'
          << "Konsonanttien lukumäärä: \t" << consonantCnt << '\n';
}
```

isalpha() on C-standardin kirjastorutiini; sen arvo on true, jos sen argumentti on aakkosten kirjain. Jotta sitä voi käyttää, ohjelmoijan pitää ottaa mukaan järjestelmän otsikkotiedosto ctype.h. (Katsomme ctype.h-rutiineja tarkemmin luvussa 6.)

Vaikka ei olekaan aivan välttämätöntä määrittää break-lausetta switch-lauseen viimeiseksi lauseeksi, on se kuitenkin turvallisinta. Jos myöhemmin lisätään yksi case-otsikko switch-lauseen loppuun, break-lauseen puuttuminen voi johtaa molempien case-otsikoiden suorittamiseen.

Esittelylause voidaan sijoittaa switch-lauseen ehto-osaan kuten seuraavassa:

```
switch( int ival = get_response() )
```

ival alustetaan ja tuosta alkuarvosta tulee arvo, jota vastaan jokaista case-otsikkoa verrataan. ival näkyy koko switch-lauseen sisällä, mutta ei sen ulkopuolelle.

Esittelylause, joka liittyy case- tai default-otsikkoon, ei ole kelvollinen, ellei sitä sijoiteta lauselohkoon. Esimerkiksi seuraava johtaa käännöksenaikaiseen virheeseen:

```
case illegal_definition:
    // virhe: esittelylause pitää
    // sijoittaa lauselohkoon

    string file_name = get_file_name();
    // ...
    break;
```

Ellei määrittelyä sijoitettaisi lauselohkoon, olisi se näkyvissä kaikissa case-otsikoissa, mutta alustettaisiin vain, jos se case-otsikko suoritettaisiin, jossa sen määrittely on. Vaatimus lauselohkosta takaa, että nimi on näkyvissä — ja siten voidaan käyttää — vain siellä, missä voidaan taata sen alustus. Jotta ohjelmamme kääntyisi, pitää case-otsikko toteuttaa uudelleen esittelemällä lauselohko kuten seuraavassa:

```
case ok:
  {
      // ok: esittelylause lauselohkossa

      string file_name = get_file_name();
      // ...
      break;
  }
```

Harjoitus 5.5

Muokkaa vokaalienlaskentaohjelmaamme niin, että se laskee myös luettujen tyhjien merkkien, sarkainmerkkien ja rivinvaihtomerkkien lukumäärän.

Harjoitus 5.6

Muokkaa vokaalien laskentaohjelmaamme niin, että se laskee myös seuraavien kaksoismerkkien esiintymisten lukumäärän: ff, fl ja fi.

Harjoitus 5.7

Jokainen seuraavista tuo esille yleisen ohjelmointivirheen switch-lauseen käytön yhteydessä. Yksilöi ja korjaa jokainen virhe.

(a)
```
switch ( ival ) {
        case 'a': aCnt++;
        case 'e': eCnt++;
        default: iouCnt++;
}
```

(b)
```
switch (ival ) {
        case 1:
            int ix = get_value();
            ivec[ ix ] = ival;
            break;
        default:
            ix = ivec.size()-1;
            ivec[ ix ] = ival;
}
```

(c)
```
switch (ival ) {
        case 1, 3, 5, 7, 9:
            oddcnt++;
            break;
        case 2, 4, 6, 8, 10:
            evencnt++;
            break;
}
```

(d)
```
int ival=512 jval=1024, kval=4096;
int bufsize;
// ...
switch( swt ) {
        case ival:
            bufsize = ival * sizeof( int );
            break;
        case jval:
            bufsize = jval * sizeof( int );
            break;
        case kval:
            bufsize = kval * sizeof( int );
            break;
}
```

(e)
```
enum { illustrator = 1, photoshop, photostyler = 2 };
switch (ival) {
        case illustrator:
            --illus_license;
```

```
            break;
        case photoshop:
            --pshop_license;
            break;
        case photostyler:
            --pstyler_license;
            break;
    }
```

5.5 For-lause

Kuten olemme nähneet, hyvin monet ohjelman toiminnoista liittyvät jonkin lausejoukon toistuvaan suoritukseen niin kauan, kun jokin ehto pysyy totena. Esimerkkejä: niin kauan, kun emme kohtaa tiedoston loppua, lue ja käsittele tiedoston seuraava elementti; vektorin jokaisella indeksillä, jonka arvo ei ole yhtä kuin yksi yli vektorin viimeisen elementin, hae ja käsittele vektorin elementti jne. Kielessä on kolme silmukan ohjauslausetta tukemassa yksittäisen lauseen tai lausejoukon toistuvaa suoritusta niin kauan, kun määritetty ehto on tosi. Olemme jo nähneet lukuisia esimerkkejä sekä for- että while-silmukasta.

For- ja while-silmukka alkaa sen ehdon totuusarvon testauksella. Tämä merkitsee, että koko silmukka ja siihen liittyvä lause tai lauselohko voivat jäädä suorittamatta. Kolmas silmukkarakenne, do while -silmukka, takaa, että lause tai lauselohko suoritetaan ainakin kerran — ehto testataan suorituksen jälkeen. Tässä kohdassa katsomme for-silmukkaa yksityiskohtaisesti. While-silmukka on pääaiheena kohdassa 5.6 ja do while -silmukkaa tutkimme kohdassa 5.7.

For-silmukkaa käytetään yleisimmin kiinteäpituisen tietorakenteen kuten taulukon tai vektorin läpikäymiseen. Esimerkiksi:

```
#include <vector>
int main()
{
    int ia[10];

    for ( int ix = 0; ix < 10; ++ix )
            ia[ ix ] = ix;

    vector<int> ivec( ia, ia+10 );
    vector<int>::iterator iter = ivec.begin();

    for ( ; iter != ivec.end(); ++iter )
            *iter *= 2;

    return 0;
}
```

For-silmukan syntaktinen muoto on seuraava:

```
for ( alustuslause; ehto; lauseke )
```

lause

alustuslause voi olla joko esittelylause tai lauseke. Yleensä sitä käytetään alkuarvon alustamiseen tai sijoitukseen, jota kasvatetaan silmukan suorituksen aikana. Jos alustus on tarpeeton tai tapahtuu jossain muualla, alustuslause voidaan jättää pois kuten edellisen esimerkin toisessa for-silmukassa. Puolipiste on kuitenkin välttämätön ja ilmoittaa, että lause puuttuu (tai toimii pikemminkin tyhjänä lauseena). Seuraavat ovat sallittuja alustuslauseen ilmentymiä:

```
// edellyttää, että index ja iter on määritelty jossain muualla
for ( index = 0; ...
for ( ; /* tyhjä alustuslause */ ...
for ( iter = ivec.begin(); ...
for ( int lo = 0, hi = max; ...
for ( char *ptr = getStr(); ...
```

Silmukan ehto toimii ohjaavana tekijänä, sillä lause suoritetaan niin monta kertaa, kun ehdon arvo on true. Silmukan lause voi olla joko yksi lause tai yhdistetty lause. Jos ehto on false jo ensimmäisellä kerralla, lause jää suorittamatta. Seuraavat ovat sallittuja ehtoja:

```
(...; index < arraySize; ... )
(...; iter != ivec.end(); ... )
(...; *st1++ = *st2++; ... )
(...; char ch = getNextChar(); ... )
```

For-silmukan lauseke arvioidaan jokaisen silmukan jälkeen. Sitä käytetään yleensä niiden muuttujien muokkaamiseen, jotka on alustettu alustuslauseessa ja testattu ehdossa. Jos ehdon ensimmäinen arviointi saa arvon epätosi, lause jää suorittamatta. Seuraavat ovat sallittuja lauseen esiintymiä:

```
( ...; ...; ++index )
( ...; ...; ptr = ptr->next )
( ...; ...; ++i, --j, ++cnt )
( ...; ...; ) // tyhjä esiintymä
```

Jos on kirjoitettu seuraava for-silmukka,

```
const int sz = 24;
int ia[ sz ];
vector<int> ivec( sz );

for ( int ix = 0; ix < sz; ++ix ) {
    ivec[ ix ] = ix;
    ia[ ix ] = ix;
}
```

arviointijärjestys on seuraava:

1. Silmukan alussa suoritetaan alustuslause kerran. Tässä esimerkissä ix määritellään ja alustetaan arvolla 0.
2. Seuraavaksi suoritetaan ehto. Jos sen arvo on true, suoritetaan yhdistetty lause. Tässä esimerkissä niin kauan, kun ix on pienempi kuin sz, sijoitetaan ix vektoriin ivec[ix] ja taulukkoon ia[ix]. Kun ehdon arvo on false, silmukka päättyy. Jos heti alussa ehdon arvo on false, se johtaa siihen, että yhdistettyä lausetta ei koskaan suoriteta.
3. Sitten suoritetaan lauseke. Tyypillisesti tämä saa aikaan muuttujan (muuttujien) muokkauksen, jotka on alustettu alustuslauseessa ja testattu ehdossa. Tässä esimerkissä oliota ix kasvatetaan arvolla 1.

Nämä kolme vaihetta edustavat silmukan yhtä täydellistä toistokertaa. Sitten toistetaan kohta 2, jonka jälkeen kohta 3, kunnes ehdon arvoksi tulee false; ix ei ole enää pienempi kuin sz.

Silmukan alustuslauseessa voidaan määritellä useita olioita. Kuitenkin voi olla vain yksi esittelylause, joten kaikkien olioiden on oltava samaa yleistä tyyppiä. Esimerkiksi:

```
for ( int ival = 0, *pi = &ia, &ri = val;
    ival < size;
    ++ival, ++pi, ++ri )
        // ...
```

Olion määrittely silmukan ehto-osassa on vaikeampi järjestää: sen arvon täytyy jossakin vaiheessa tulla epätodeksi tai silmukka ei pääty koskaan. Tässä on hieman keksitty esimerkki:

```
#include <iostream>

int main()
{
   for ( int ix = 0;
       bool done = ix == 10;
       ++ix )
           cout << "ix: " << ix << endl;
}
```

Niiden kaikkien olioiden näkyvyys, jotka on määritelty silmukan ehto-osassa, rajoittuu silmukan runkoon. Esimerkiksi iter-olion testi, joka on silmukan jälkeen, saa aikaan käännöksenaikaisen virheen[1]:

```
int main()
{
   string word;
   vector< string > text;
   // ...
   for ( vector< string >::iterator
                    iter = text.begin(),
              iter_end = text.end();
         iter != text.end(); ++iter )
   {
        if ( *iter == word )
             break;
        // ...
   }

   // virhe: iter ja iter_end eivät ole näkyvissä
   if ( iter != iter_end )
       // ...
}
```

Harjoitus 5.8

Mitkä seuraavista for-silmukoiden esittelyistä ovat virheellisiä, vai onko yksikään?

```
(a) for ( int *ptr = &ia, ix = 0;
          ix < size && ptr != ia+size;
          ++ix, ++ptr )
             // ...

(b) for ( ; ; ) {
          if ( joku_ehto )
              break;
          // ...
    }
```

1. Niiden olioiden näkyvyys, jotka määriteltiin C++:n esistandardissa init-statement-lauseessa, laajeni funktioon tai lauselohkoon itse silmukka mukaan lukien. Esimerkiksi seuraavien for-silmukoiden ollessa samassa lauselohkossa

```
{
   // sallittu C++-standardissa
   // ei sallittu C++:n esistandardissa, koska  ival on määritelty kahdesti
   for ( int ival = 0; ival < size; ++ival ) // ...
   for ( int ival = size-1; ival >= 0; --ival ) // ...
}
```

C++-esistandardissa, ival saa aikaan käännöksenaikaisen virheen, koska se on määritelty kaksi kertaa, kun taas C++-standardissa molemmat ival-ilmentymät ovat paikallisia vastaaville for-silmukoille. Tällöin koodikatkelma on täysin sallittu.

```
(c) for ( int ix = 0; ix < sz; ++ix )
        // ...

    if ( ix != sz )
        // ...

(d) int ix;
    for ( ix < sz; ++ix )
        // ...

(e) for ( int ix = 0; ix < sz; ++ix, ++ sz )
        // ...
```

Harjoitus 5.9

Sinua on pyydetty suunnittelemaan tyyliohje for-silmukalle projektikäyttöön. Selitä ja kuvaa käyttösäännöt jokaiselle silmukan kolmelle osalle, jos niitä on. Jos olet vahvasti käyttösääntöjä vastaan — tai ainakin for-silmukan käytön yhteydessä — selitä ja kuvaa, miksi.

Harjoitus 5.10

Jos on kirjoitettu funktion esittely

```
bool is_equal( const vector<int> &v1,
               const vector<int> &v2 );
```

kirjoita funktion runko, joka päättelee, ovatko kaksi vektoria yhtäsuuria. Jos vektoreiden pituudet ovat erisuuria, vertaa pienemmän vektorin elementtien lukumäärää. Jos esimerkiksi on annettu vektorit (0,1,1,2) ja (0,1,1,2,3,5,8), is_equal() palauttaa arvon true. v1.size() ja v2.size() palauttavat vektorien koot.

5.6 While-lause

While-silmukan syntaktinen muoto on seuraava:

```
while ( ehto )
    lause
```

Lause tai lauselohko suoritetaan niin monta toistokertaa, kuin ehdon arvo on true. Järjestys on seuraava:

1. Arvioi ehto.
2. Suorita lause, jos ehto on tosi.

Jos ehdon ensimmäisen arvioinnin tulos on false, ei lausetta koskaan suoriteta.

While-silmukan ehto voi olla lauseke kuten jokin seuraavista:

```
bool quit = false;
// ...
while ( ! quit ) {
        // ...
        quit = do_something();
}
string word;
while ( cin >> word ) { ... }
```

tai se voi olla alustettu määrittely kuten tässä:

```
while ( symbol *ptr = search( name )) {
    // do something
}
```

Tässä jälkimmäisessä tapauksessa ptr on näkyvissä vain lauselohkon sisällä, joka liittyy while-silmukkaan aivan kuten if-, switch- ja for-lauseissa.

Tässä on esimerkki while-silmukasta, joka käy läpi elementtikokoelman, jota osoittaa osoitinpari:

```
int sumit( int *parray_begin, int *parray_end )
{
    int sum = 0;

    if ( ! parray_begin || ! parray_end )
        return sum;

    while ( parray_begin != parray_end )
            // lisää arvo sum-olioon
            // kasvata sitten osoitinta seuraavaan elementtiin
            sum += *parray_begin++;

    return sum;
}

int ia[6] = { 0, 1, 2, 3, 4, 5 };
int main()
{
    int sum = sumit( &ia[0], &ia[ 6 ] );
    // ...
}
```

Jotta sumit() toimisi oikein, pitää molempien osoittimien osoittaa *saman* taulukon elementteihin (parray_end-taulukkoa voidaan osoittaa turvallisesti yksi yli sen viimeisen elementin). Elleivät ne osoita, on sumit()-funktion käyttäytyminen *tuntematon*. (Se voi parhaassa tapauksessa palauttaa merkityksettömän arvon.) Valitettavasti kielessä ei ole tapaa taata, että molemmat osoittimet osoittavat samaa taulukkoa. Kuten tulemme näkemään luvussa 12, vakiokirjaston geneeriset algoritmit on toteutettu hyväksymään säiliön ensimmäisen ja viimeisen elementin.

Harjoitus 5.11

Mitkä seuraavista while-silmukan esittelyistä ovat virheellisiä, vai onko yksikään?

```
(a) string bufString, word;
   while ( cin >> bufString >> word )
         // ...

(b) while ( vector<int>::iterator iter != ivec.end() )
         // ...

(c) while ( ptr = 0 )
          ptr = find_a_value();

(d) while ( bool status = find( word )) {
         word = get_next_word();
         if ( word.empty() )
             break;
         // ...
   }

   if ( ! status )
       cout << "Ei löytynyt yhtään sanaa\n";
```

Harjoitus 5.12

While-silmukka on erityisen hyvä suorittamiseen, kun jokin ehto on voimassa. Esimerkiksi “niin kauan, kun tiedoston loppua ei kohdata, lue seuraava arvo”. For-silmukkaa pidetään yleisesti askelsilmukkana: indeksi käy läpi kokoelman arvoalueen. Kirjoita jokaisesta silmukasta tyypillinen käyttötapa ja kirjoita sitten jokainen uudestaan käyttäen toisen silmukan rakennetta. Jos sinun pitäisi ohjelmoida vain yhdellä silmukalla, minkä rakenteen valitsisit? Miksi?

Harjoitus 5.13

Kirjoita pieni funktio, joka lukee peräkkäin merkkijonoja vakiosyötöstä, kunnes joko sama sana esiintyy peräkkäin tai kaikki sanat on luettu. Käytä while-silmukkaa, joka lukee tekstiä yksi sana kerrallaan. Käytä break-lausetta silmukan päättämiseen, jos sana esiintyy peräkkäin. Tulosta sana, jos se esiintyy peräkkäin tai tulosta sitten ilmoitus, joka sanoo, että yhtään sanaa ei toistettu.

5.7 Do while -lause

Kuvitellaan, että meitä on pyydetty kirjoittamaan vuorovaikutteinen ohjelma, joka konvertoi maileja kilometreiksi. Ohjelman pääpiirteet näyttävät tältä:

```
int val;
bool more = true;  // tekaistu arvo silmukan käynnistämiseksi

while ( more )
{
   val = getValue();
   val = convertValue(val);
   printValue(val);
   more = doMore();
}
```

Ongelma tässä on, että silmukan ohjaus asetetaan silmukan sisällä. For- ja while-silmukoiden yhteydessä ei runkoa koskaan suoriteta, ellei silmukan ohjausta arvioida todeksi. Tämä tarkoittaa, että meidän pitää laittaa alkuarvo silmukan käynnin aloittamiseksi. Vaihtoehtoisesti voimme käyttää do while -silmukkaa. Do while -silmukka takaa, että lause suoritetaan ainakin kerran. Do while -silmukan syntaktinen muoto on seuraava:

```
do
      lause
while ( ehto );
```

Silmukan lause suoritetaan ennen kuin ehto arvioidaan. Jos ehto arvioidaan epätodeksi (false), silmukka päättyy. Ohjelmamme pääpiirteet näyttävät nyt tältä:

```
do
{
   val = getValue();
   val = convertValue(val);
   printValue(val);
} while ( doMore() );
```

Toisin kuin muissa silmukkalauseissa, do while -silmukan ehto ei tue olion määrittelyä. Emme siis voi kirjoittaa

```
// virhe: esittelylausetta ei tueta
// do while -silmukan ehdossa

do {
   // ...
   mumble( foo );
} while ( int foo = get_foo() ) // virhe
```

koska ehtoa ei arvioida, ennen kuin lause tai lauselohko on ensiksi suoritettu.

Harjoitus 5.14

Mitkä seuraavista do-while-silmukoista ovat virheellisiä, vai onko yksikään?

```
(a) do
      string rsp;
      int val1, val2;
      cout << "anna kaksi lukua: ";
      cin  >> val1 >> val2;
      cout << "Lukujen " << val1
          << " ja “ << val2
          << " summa = " << val1 + val2 << "\n\n"
          << "Lisää? [kyllä][ei] ";
      cin  >> rsp;
   while ( rsp[0] != 'e' );

(b) do {
      // ...
   } while ( int ival = get_response() );

(c) do {
      int ival = get_response();
      if ( ival == some_value() )
          break;
   } while ( ival );

   if ( !ival )
       // ...
```

Harjoitus 5.15

Kirjoita pieni ohjelma, joka pyytää käyttäjältä kaksi merkkijonoa ja tulostaa, kumpi merkkijonoista on aakkosissa pienempi kuin toinen (tarkoittaa, että tulee aakkosissa ennen toista). Jatka pyytämistä käyttäjältä, kunnes hän vastaa “quit”. Käytä string-tyyppiä, string-luokan pienempi kuin -operaattoria ja do while -silmukkaa.

5.8 Break-lause

Break-lause lopettaa lähimmän päättyvän while-, do while-, for- tai switch-lauseen. Suoritus jatkuu välittömästi päätetyn lauseen jälkeisestä lauseesta. Esimerkiksi seuraava funktio etsii kokonaislukutaulukosta tietyn arvon ensimmäistä esiintymää. Jos se löytyy, funktio palauttaa sen indeksin; muussa tapauksessa funktio palauttaa arvon –1. Tämä on toteutettu seuraavasti:

```
// onko arvo ia:ssa? palauta indeksi, muussa tapauksessa -1

int search( int *ia, int size, int value )
{
    // varmista, että ia != 0 ja että size > 0 ...

      int loc = -1;

    for ( int ix = 0; ix < size; ++ix ) {
```

```
        if ( value == ia[ ix ] ) {
                        // ok: löytyi!
                        // aseta sijainti ja jätä silmukka
            loc = ix;
            break;
        }
    } // for-silmukan loppu

    // break saa kontrollin siirtymään tähän ...
    return loc;
}
```

Esimerkissämme break pysäyttää for-silmukan. Suoritus jatkuu return-lauseesta, joka on heti for-silmukan jälkeen. Tässä esimerkissä break pysäyttää for-silmukan eikä if-lausetta, jonka sisällä se tapahtuu. Break-lauseen esiintyminen if-lauseessa, joka ei kuulu switch- tai silmukkalauseeseen, saa aikaan käännöksenaikaisen virheen. Esimerkiksi:

```
// virhe: break-lausetta ei voi käyttää tässä
if ( ptr ) {
    if ( *ptr == "quit" )
        break;
    // ...
}
```

Yleensä break-lausetta saa käyttää vain silmukka- tai switch-lauseessa.

Kun break-lause suoritetaan sisäkkäisessä switch- tai silmukkalauseessa, se ei vaikuta siihen silmukkaan tai switch-lauseeseen, jonka sisällä se itse oli. Esimerkiksi:

```
while ( cin >> inBuf )
{
   switch( inBuf[ 0 ] ) {
   case '-':
      for ( int ix = 1; ix < inBuf.size(); ++ix ) {
           if ( inBuf[ ix ] == ' ' )
               break; // #1
           // ...
      // ...
      }
      break; // #2
   case '+':
      // ...
   }
}
```

break-lause, joka on otsikoitu //#1, pysäyttää for-silmukan yhdysmerkin case-otsikossa, mutta ei pysäytä switch-lausetta. Samalla tavalla break-lause, joka on otsikoitu //#2, päättää switch-lauseen inBuf:in ensimmäisellä merkillä, mutta ei pysäytä while-silmukkaa, joka lukee yhden merkkijonon kerrallaan vakiosyötöstä.

5.9 Continue-lause

Continue-lause saa lähimmän silmukan nykyisen toiston päättymään. Suoritus jatkuu ehdon arvioinnista. Toisin kuin break-lause, joka lopettaa silmukan, continue-lause lopettaa ainoastaan nykyisen toiston. Esimerkiksi seuraava koodikatkelma lukee ohjelmatekstitiedostoa yhden sanan kerrallaan. Jokainen sana, joka alkaa alaviivalla, käsitellään; muussa tapauksessa nykyinen silmukkatoisto lopetetaan:

```
while ( cin >> inBuf ) {
    if ( inBuf[0] != '_' )
        continue; // lopeta tämä toisto

    // yhä täällä ? käsittele merkkijono ...
}
```

Continue-lause voi esiintyä vain silmukkalauseessa.

5.10 Goto-lause

Goto-lauseen avulla voidaan haarautua ilman ehtoja saman funktion sisällä johonkin otsikkolauseeseen. Sen käyttöä kehotetaan välttämään nykyisen hyvän ohjelmointitavan ajattelutavan mukaisesti.

Goto-lauseen syntaktinen muoto on

```
goto otsikko;
```

jossa otsikko on käyttäjän antama tunnus. Otsikkolausetta voidaan käyttää vain goto-lauseen kohteena ja sen pitää päättyä kaksoispisteeseen. Otsikkolause ei saa olla juuri ennen sulkevaa oikeanpuoleista aaltosulkua. Tyhjä lause on tyypillinen menettelytapa tämän rajoituksen käsittelyyn. Esimerkiksi:

```
end: ; //tyhjä lause
}
```

goto-lauseella ei voida hypätä ylitse esittelylauseen, joka ei ole lauselohkon sisällä. Esimerkiksi seuraava johtaa käännöksenaikaiseen virheeseen:

```
int oops_in_error()
{
   // muminaa ...
   goto end;

   // virhe: goto tehdään esittelylauseen yli
   int ix = 10;
   // ... koodi, joka käyttää ix:ää
end: ;
}
```

Korjatussa toteutuksessa sijoitetaan esittelylause ja kaikki sitä hyödyntävät lauseet lause-

lohkoon. Esimerkiksi:

```
int ok_its_fixed()
{
   // muminaa ...
   goto end;

   {
      // ok: esittelylause lauselohkossa
      int ix = 10;
      // ... koodi, joka käyttää ix:ää
   }
end: ;
}
```

Perustelut ovat samat kuin esittelylauseilla switch-lauseessa: kääntäjän tarve käynnistää luokkaolioille muodostaja/tuhoajapareja. Lauselohko takaa, että sekä muodostaja että tuhoaja suoritetaan tai jätetään huomiotta ja että oliot ovat näkyvissä vain siellä, missä niiden alustukset on tehty.

Hyppy taaksepäin yli alustetun oliomäärittelyn on kuitenkin sallittua. Miksi? Olion alustuksen yli hyppääminen on ohjelmointivirhe; olion alustaminen useita kertoja on silti oikein, vaikkakin tehotonta. Esimerkiksi:

```
// hyppy taaksepäin yli esittelylauseen on ok

void
mumble( int max_size )
{
 begin:
   int sz = get_size();
   if ( sz <= 0 ) {
         // anna varoitus ...
         goto end;
   }
   else
   if ( sz > max_size )
         // ota uusi koon arvo
         goto begin;
```

```
    { // ok: hypätty koko lohkon yli
            int *ia = new int[ sz ];
            doit( ia, sz ); // mitä se ikinä onkin ...
            delete [] ia;
    }

  end:
    ;
}
```

Goto-lause on kaikkein paheksutuin piirre nykyisissä ohjelmointikielissä. Goto-lauseen käyttö johtaa usein siihen, että ohjelmankulkua on vaikea ymmärtää ja ylläpitää. Useimmat sen käyttötilanteista voidaan korvata ehdollisella lauseella tai silmukkalauseella. Jos huomaat käyttäväsi goto-lausetta, suosittelemme, että se ei ulotu koko ohjelman alueelle.

5.11 Esimerkki linkitetystä listasta

Olemme päättäneet luvut 3 ja 4 suunnittelemalla ja toteuttamalla luokan, joka tuo esille ja jonka avulla voimme harjoitella C++:n luokkamekanismia. Samalla tavalla päätämme tämän luvun toteuttamalla yhteen suuntaan linkitetyn listan luokan. (Luvussa 6 katsomme kahteen suuntaan linkitetyn listan säiliöluokan, joka on vakiokirjastossa.) Ensikertalainen lukija saattaa haluta hypätä tämän kohdan yli ja palata tähän uudelleen luettuaan luvun 13. (Lukijan, joka jatkaa tämän kohdan läpi, oletetaan lukeneen joko kohdan 2.3 tai 3.15 ja siten omaavan jonkin verran tietoa luokkamekanismin syntaksista ja termeistä, kuten esimerkiksi, mikä muodostaja on jne. Ellei näin ole, suosittelemme, että luet ensin niistä toisen tai mieluummin molemmat kohdat.)

Lista on jono elementtejä, joista jokainen sisältää tietyn tyyppisen arvon ja osoitteen — joka voi olla myös null-arvo — listan seuraavaan elementtiin. Lista voi olla tyhjä eli se voi olla olemassa ilman jäseniä. Lista ei voi olla täynnä, vaikka yritys lisätä uusi jäsen listaan voi epäonnistua, kun ohjelman vapaan muistin varasto on loppunut.

Mitä ovat ne operaatiot, joita listaluokkamme pitää tukea? Käyttäjän pitää pystyä sekä *lisäämään* että *poistamaan* jäsen kuten myös hakemaan — *etsimään* — jäsen. Käyttäjän pitää pystyä kysymään listan *koko*, *näyttämään* se ja vertaamaan kahden listan *yhtäsuuruutta*. Lisäksi tulemme tukemaan listan elementtien järjestämistä *päinvastaisesti* ja kahden listan *yhdistämistä*.

size()-funktion yksinkertaisin toteutus on käydä lista läpi ja laskea elementtien lukumäärä. Jonkin verran monimutkaisempi toteutus on tallentaa listan koko tietojäseneen. Toisesta size()-funktion toteutuksesta tulee merkittävästi tehokkaampi: se palauttaa yksinkertaisesti siihen liittyvän jäsenen. Monimutkaisuuden lisäys tulee tarpeesta ylläpitää jäsentä jokaisessa elementin lisäyksessä ja poistossa.

Olemme päättäneet ylläpitää ja tallentaa elementtien lukumäärän _size-tietojäseneen. Oletuksemme on, että käyttäjän kysely listan koosta on usein toistuvaa ja pitää siten olla välttämättä nopea. (Eräs etu julkisen rajapinnan erottamisessa yksityisestä on, että jos oletuksemme on

väärä, voimme tehdä uuden toteutuksen ilman, että vaatisimme muutoksia size()-funktiota käyttävään ohjelmaan edellyttäen, että säilytämme saman palautustyypin ja parametriluettelon.)

Yleiselle insert()-operaatiollemme annamme kaksi parametria: osoittimen olemassaolevan listan elementtiin ja uuden arvon. Uusi arvo lisätään olemassaolevan elementin perään. Jos esimerkiksi on lista

```
1 1 2 3 8
```

ja käynnistetään

```
mylist.insert( pointer_to_3, 5 );
```

muuttuu lista seuraavasti:

```
1 1 2 3 5 8
```

Jotta tätä voitaisiin tukea, pitää käyttäjälle antaa keino saada tietyn jäsenen osoite, kuten edellisessä esimerkissä elementin 3. Eräs menetelmä on find()-operaatio. Esimerkiksi:

```
pointer_to_3 = mylist.find( 3 );
```

find() saa etsittävän arvon vaadittuna ensimmäisenä argumenttina. Jos elementti löytyy, find() palauttaa osoittimen tuohon elementtiin; muussa tapauksessa find() palauttaa arvon 0.

On olemassa kaksi erityistapausta, joita haluamme insert()-funktion tukevan: lisäyksiä listan alkuun ja loppuun. Näissä tapauksissa käyttäjää vaaditaan määrittämään vain lisättävä arvo:

```
insert_front( value );
insert_end( value );
```

Poistaminen tukee seuraavia operaatioita: yksittäisen arvon poisto, elementin poisto alusta ja kaikkien elementtien poisto:

```
remove( value );
remove_front();
remove_all();
```

display()-operaatio tukee muotoiltua tulostusta sekä listan koolle että yksittäisille elementeille. Tyhjä listä näytetään näin

```
(0)( )
```

Lista, jossa on seitsemän elementtiä, näyttää tältä:

```
(7)( 0 1 1 2 3 5 8 )
```

reverse() yksinkertaisesti järjestää elementit päinvastoin. Esimerkiksi sen jälkeen, kun on käynnistetty

```
mylist.reverse();
```

aikaisempi lista näyttää seuraavalta:

```
(7)( 8 5 3 2 1 1 0 )
```

Yhdistäminen liittää toisen listan ensimmäisen loppuun. Jos on esimerkiksi listat

```
(4)( 0 1 1 2 ) // list1
(4)( 2 3 5 8 ) // list2
```

operaatio

```
list1.concat( list2 );
```

muokkaa list1:n muotoon

```
(8)( 0 1 1 2 2 3 5 8 )
```

Tehdäksemme list1:stä aidon Fibonacci-lukujonon, voimme käyttää remove()-funktiota:

```
list1.remove( 2 );
```

Kun olemme määritelleet listaluokkamme käyttäytymisen, on seuraava vaiheemme edetä sen toteuttamiseen. Olemme päättäneet esittää listaa (*list*) ja listan jäsentä (*list_item*) erillisillä luokka-abstraktioilla. (Rajoitamme toteutuksemme tässä vaiheessa vain yksinkertaisesti int-tyyppiin. Luokkamme on siksi nimetty seuraavasti: ilist ja ilist_item.)

Listamme sisältää jäsenen _at_front, joka osoittaa listan alkuun, jäsenen _at_end, joka osoittaa listan loppuun ja jäsenen _size, joka pitää sisällään listan nykyisen koon. Kun listaolio määritellään alussa, pitää nämä kolme jäsentä alustaa arvolla nolla. Takaamme tämän oletusmuodostajalla:

```
class ilist_item;

class ilist {
public:
   // oletusmuodostaja
   ilist() : _at_front( 0 ),
             _at_end( 0 ), _size( 0 ) {}

   // ...
private:
   ilist_item *_at_front;
   ilist_item *_at_end;
   int _size;
};
```

Tämän avulla voimme määritellä ilist-olioita kuten tässä

```
ilist mylist;
```

mutta ei mitään muuta vielä. Lisätkäämme tuki listan koon kyselylle. Sen jälkeen, kun olemme esitelleet size()-jäsenfunktion luokkamäärittelyn julkiseen osaan, voimme määritellä jäsenfunktion kuten seuraavassa:

```
inline int ilist::size() { return _size; }
```

Nyt voimme kirjoittaa

```
int size = mylist.size();
```

Tässä vaiheessa emme halua sallia, että yksi lista alustettaisiin tai siihen sijoitettaisiin toinen lista (muutamme tätä myöhemmin — se ei vaadi muutoksia käyttäjän ohjelmiin). Estääksemme sekä alustamisen että sijoittamisen, esittelemme ilist-luokan kopiointimuodostajan ja sijoitusoperaattorin yksityisiksi jäseniksi emmekä tee määrittelyä kummallekaan. Tässä on uu-

distettu ilist-luokkamme määrittely:

```
class ilist {
public:
   // määrittelyitä ei näytetä
   ilist();
   int size();
   // ...
private:
   // estetään yhden ilist-jäsenen sijoitus ja alustaminen
   // toisella ilist-jäsenellä
   ilist( const ilist& );
   ilist& operator=( const ilist& );

   // tietojäsenet kuten ennen
};
```

Tätä määrittelyä käyttäen molemmat seuraavista johtavat käännöksenaikaiseen virheeseen, koska main() ei voi käsitellä ilist-luokkamme yksityisiä jäseniä:

```
int main()
{
   ilist yourlist( mylist ); // virhe
   mylist = mylist;          // virhe
}
```

Seuraava tehtävämme on tukea jäsenen lisäystä. Olemme päättäneet esittää jäsenen itsenäisenä luokka-abstraktiona, joka sisältää jäsenen _value, joka puolestaan sisältää arvon ja jäsenen _next , joka vuorostaan pitää sisällään listan seuraavan olion osoitteen, jos sellainen on:

```
class ilist_item {
public:
   // ...
private:
   int        _value;
   ilist_item *_next;
};
```

Ilist-luokan muodostaja vaatii, että arvo pitää määrittää ja valinnaisesti sallii myös määrittää osoittimen ilist_item-jäseneen. Jos sellainen on annettu, osoitin edustaa sitä ilist_item-jäsentä, jonka jälkeen uusi jäsen lisätään. Jos esimerkiksi on olemassa lista

0 1 1 2 5

muodostajan käynnistys

```
ilist ( 3, pointer_to_2 );
```

muokkaa listaa seuraavasti

0 1 1 2 3 5

Tässä on toteutuksemme. (Muista, että toinen item-jäsen on valinnainen; oletusarvo 0 välitetään, jos käyttäjä ei anna arvoa. Oletusarvo määritetään funktion esittelyssä eikä sen

määrittelyssä; tämä kuvataan tarkemmin luvussa 7.)

```
class ilist_item {
public:
   ilist_item( int value, ilist_item *item_to_link_to = 0 );
   // ...
};

inline
ilist_item::
ilist_item( int value, ilist_item *item )
          : _value( value )
{
   if ( !item )
       _next = 0;
   else {
       _next = item->_next;
       item->_next = this;
   }
}
```

Yleiselle ilist-luokan insert()-operaatiolle annetaan lisättävä arvo ja list_item-osoitin, joka ilmaisee jäsenen, jonka jälkeen uusi jäsen lisätään. Tässä on ensimmäinen versiomme (siinä on kaksi ongelmaa — löydätkö ne?):

```
inline void
ilist::
insert( ilist_item *ptr, int value )
{
   new ilist_item( value, ptr );
   ++_size;
}
```

Ensimmäinen ongelma on, että siinä ei ole varmistettu osoittimen sisältävän nollasta poikkeavan arvon. Tuo tilanne pitää havaita ja käsitellä, koska muutoin sellainen tapahtuma aiheuttaa todennäköisesti ohjelman kaatumisen suorituksen aikana. Kuinka meidän pitäisi käsitellä se? Eräs mahdollisuus on keskeyttää ohjelma käynnistämällä C-vakiokirjaston abort()-funktio (sijaitsee C-vakiokirjaston cstdlib-otsikkotiedostossa):

```
#include <cstdlib>
// ...
if ( ! ptr )
       abort();
```

Vaihtoehtoisesti voimme käyttää assert()-makroa. Tämä myös keskeyttää ohjelmamme, mutta ilmoittaa aluksi ehdon, joka laukaisi makron:

```
#include <cassert>

// ...
assert( ptr != 0 );
```

Kolmas vaihtoehto on heittää keskeytys. Esimerkiksi:

```
if ( ! ptr )
    throw "Panic: ilist::insert(): ptr == 0";
```

Yleensä ohjelman keskeyttämistä pitäisi välttää aina, kun se on mahdollista. Itse asiassa ohjelman keskeytys jättää käyttäjän oman onnensa nojaan siksi ajaksi, kun me tai tukiorganisaatio yrittää eristää ja ratkaista ongelmaa.

Jos emme pysty jatkamaan käsittelyä virheen ilmaantumispaikassa, poikkeuksen heittäminen on yleensä parempi vaihtoehto kuin ohjelman keskeyttäminen. Poikkeus siirtää kontrollin ohjelman aikaisempaan osaan; sellaiseen, joka ehkä kykenee ratkaisemaan ongelman.

Toteutuksemme havaitsee ja käsittää tyhjän osoittimen pyynnöksi lisätä arvo listan alkuun:

```
if ( ! ptr )
    insert_front( value );
```

Toteutuksemme toinen vika on enemmän filosofinen kuin vakava virhe. Toteutuksemme _size- ja size()-pari on alustava suunnitelma: vaikka uskomme, että listan koon tallentaminen ja välitön hakeminen parhaiten tyydyttää käyttäjiemme tarpeita, voimme käytännössä joutua korvaamaan sen strategialla, jossa listan koko lasketaan pyydettäessä. Laske-pyydettäessä-toteutuksessa _size-jäsen eliminoidaan. Kirjoittamalla

```
++_size;
```

kytkemme tiukasti insert()-funktion toteutuksen nykyiseen listaluokan toteutukseen. Jos listaluokkamme toteutus muuttuu, ei insert() ole enää oikein, ja sitä pitää myös muuttaa, kuten myös jäseniä insert_front(), insert_end() ja poistoilmentymiä. Sen sijaan, että hajottaisimme listaluokkamme toteutuksen yksityiskohtien luotettavuutta useisiin lisäys- ja poisto-operaatioihin, päätämme kapseloida riippuvuuden funktioparin:

```
inline void ilist::bump_up_size()   { ++_size; }
inline void ilist::bump_down_size() { --_size; }
```

Koska olemme esitelleet parin välittöminä, ei suunnittelumme vaikuta toteutuksemme tehokkuuteen. Tässä on uudistettu toteutuksemme:

```
inline void
ilist::
insert( ilist_item *ptr, int value )
{
   if ( !ptr )
       insert_front( value );
   else {
       bump_up_size();
       new ilist_item( value, ptr );
   }
}
```

Funktioiden insert_front() ja insert_end() toteutus on suhteellisen suoraviivaista. Jokaisen pitää käsitellä tyhjän listan erikoistilanne. Tässä ovat niiden toteutukset:

```
inline void
ilist::
insert_front( int value )
{
   ilist_item *ptr = new ilist_item( value );

   if ( !_at_front )
        _at_front = _at_end = ptr;
   else
   {
       ptr->next( _at_front );
       _at_front = ptr;
   }

   bump_up_size();
}

inline void
ilist::
insert_end( int value )
{
   if ( !_at_end )
        _at_end = _at_front = new ilist_item( value );
   else  _at_end = new ilist_item( value, _at_end );

   bump_up_size();
}
```

find() etsii listasta arvoa. Jos se löytyy, find() palauttaa osoittimen arvoon; muussa tapauksessa se palauttaa arvon 0. Tässä on sen toteutus:

```
ilist_item*
ilist::
find( int value )
{
   ilist_item *ptr = _at_front;
   while ( ptr )
   {
         if ( ptr->value() == value )
             break;

         ptr = ptr->next();
   }

   return ptr;
}
```

Funktiota find() voidaan käyttää seuraavasti:

```
ilist_item *ptr = mylist.find( 8 );
mylist.insert( ptr, some_value );
```

tai tiiviimmin esimerkiksi näin

```
mylist.insert( mylist.find( 8 ), some_value );
```

Ennen kuin voimme kokeilla lisäysoperaatioitamme, tarvitsemme display()-funktiotamme, jotta voimme nähdä, kuinka olemme onnistuneet toteutuksessamme. display()-funktion algoritmi on tarpeeksi yksinkertainen: ensimmäisestä elementistä aloittaen tulostamme jokaisen vuorollaan, kunnes olemme tulostaneet ne kaikki. Näetkö, miksi seuraava for-silmukan suunnitelma epäonnistuu?

```
// hups! ei toimi kunnolla
// tarkoitus: näytä kaikki muut paitsi viimeinen ilist-elementti

for ( ilist_item *iter = _at_front;  // aloita listan alusta
     iter != _at_end;              // lopeta lopussa
     ++iter )                    // siirry seuraavaan jäseneen
       cout << iter->value() << ' ';

// näytä nyt viimeinen elementti
cout << iter->value();
```

Syy siihen, että tämä ei toimi, on se, että listan elementtejä ei ole tallennettu muistiin yhtenäisesti. Osoittimen aritmetiikka

```
++iter;
```

ei siirrä iter:iä ilist-listan seuraavan elementin osoitteeseen. Sen sijaan se lisää yhden ilist_item-olion koon tavuina iter:in osoitteeseen. Meillä ei ole aavistustakaan siitä, mitä oliota, jos yleensä mitään, iter osoittaa kasvatuksen jälkeen, tai päättyykö silmukka koskaan. Jotta siirryttäisiin seuraavaan ilist_item-olioon, pitää iter eksplisiittisesti asettaa uudelleen jokaisen toistokerran jälkeen seuraavaan jäseneen, jota ilist_item-tietojäsen _next osoittaa:

```
iter = iter->_next;
```

Olemme kapseloineet pääsyn _value- ja _next-jäseniin joukolla välittömiä käsittelyfunktioita. Tässä on uudistettu ilist_item-luokkamme määrittely:

```
class ilist_item {
public:
   ilist_item( int value, ilist_item *item_to_link_to = 0 );
```

```
    int         value() { return _value; }
    ilist_item* next()  { return _next;  }


    void next( ilist_item *link ) { _next = link;       }
    void value( int new_value )   { _value = new_value; }

private:
    int       _value;
    ilist_item *_next;
};
```

Tässä on display()-funktiomme toteutus, joka käyttää edellä olevaa ilist_item-luokan määrittelyä:

```
#include <iostream>


class ilist_item {
public:
    void display( ostream &os = cout );
    // ...
};

void
ilist::
display( ostream &os )
{
    os << "\n( " << _size << " )( ";

    ilist_item *ptr = _at_front;
    while ( ptr ) {
          os << ptr->value() << " ";
          ptr = ptr->next();
    }

    os << ")\n";
}
```

Tässä on pieni ohjelma, jossa kokeillaan ilist-luokkaamme niin kuin olemme sen tähän saakka määritelleet.

```
#include <iostream>
#include "ilist.h"

int main()
{
    ilist mylist;

    for ( int ix = 0; ix < 10; ++ix ) {
        mylist.insert_front( ix );
```

```
        mylist.insert_end( ix );
    }

    cout << "Ok: after insert_front() and insert_end()\n";
    mylist.display();

    ilist_item *it = mylist.find( 8 );
    cout << "\n"
        << "Searching for the value 8: found it?"
        << ( it ? " yes!\n" : " no!\n" );

    mylist.insert( it, 1024 );
    cout << "\n"
        << "Inserting element 1024 following the value 8\n";

    mylist.display();

    int elem_cnt = mylist.remove( 8 );
    cout << "\n"
        << "Removed " << elem_cnt << " of the value 8\n";

    mylist.display();

    cout << "\n" << "Removed front element\n";
    mylist.remove_front(); mylist.display();

    cout << "\n" << "Removed all elements\n";
    mylist.remove_all(); mylist.display();
}
```

Se generoi seuraavat tulokset kääntämisen ja suorituksen jälkeen:

```
Ok: after insert_front() and insert_end()

( 20 )( 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )

Searching for the value 8: found it? yes!

Inserting element 1024 following the value 8

( 21 )( 9 8 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )

Removed 2 of the value 8

( 19 )( 9 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )

Removed front element

( 18 )( 1024 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 9 )
```

```
Removed all elements

( 0 )( )
```

Yhtä hyvin kuin käyttäjät lisäävät jäseniä listaan, heidän pitää pystyä myös poistamaan jäseniä listasta. Tuemme kolmea eri piirrettä elementin poistossa:

```
void remove_front();
void remove_all();
int  remove( int value );
```

Tässä on toteutuksemme remove_front()-funktiostamme:

```
inline void
ilist::
remove_front()
{
   if ( _at_front ) {
       ilist_item *ptr = _at_front;
       _at_front = _at_front->next();
       bump_down_size();
       delete ptr;
   }
}
```

remove_all() käynnistää toistuvasti remove_front()-funktiota, kunnes lista on tyhjä:

```
void
ilist::
remove_all()
{
   while ( _at_front )
         remove_front();

   _size = 0;
   _at_front = _at_end = 0;
}
```

Yleinen remove()-funktion toteutus käyttää myös hyödyksi remove_front()-funktiota erikoistilanteessa, jossa poistetaan yksi tai useampi jäsenistä, jotka ovat listan alussa. Muussa tapauksessa käymme listan läpi edellisellä ja nykyisellä osoittimella, poistamme jäsenen ja kytkemme listan uudelleen jokaisella löytämällämme ilmentymällä. Tässä on sen toteutus:

```
int
ilist::
remove( int value )
{
   ilist_item *plist = _at_front;
   int elem_cnt = 0;

   while ( plist && plist->value() == value )
   {
```

```
        plist = plist->next();
        remove_front();
        ++elem_cnt;
    }

    if ( ! plist )
        return elem_cnt;

    ilist_item *prev = plist;
    plist = plist->next();

    while ( plist )
    {
        if ( plist->value() == value )
        {
            prev->next( plist->next() );
            delete plist;
            ++elem_cnt;
            bump_down_size();
            plist = prev->next();
            if ( ! plist )
            {
                _at_end = prev;
                return elem_cnt;
            }
        }
        else
        {
            prev = plist;
            plist = plist->next();
        }
    }

    return elem_cnt;
}
```

Seuraavassa ohjelmassa kokeillaan poisto-operaatioita ja testataan seuraavia tilanteita: (1) kaikki poistettavat jäsenet sijaitsevat listan lopussa, (2) kaikki listan jäsenet poistetaan, (3) jäseniä ei ole yhtään ja (4) jäsenet sijaitsevat sekä listan alussa että lopussa.

```
#include <iostream>
#include "ilist.h"

int main()
{
    ilist mylist;
```

```
cout << "\n-----------------------------------------------\n"
     << "test #1: items at end\n"
     << "-----------------------------------------------\n";

mylist.insert_front( 1 ); mylist.insert_front( 1 );
mylist.insert_front( 1 );

mylist.insert_front( 2 ); mylist.insert_front( 3 );
mylist.insert_front( 4 );

mylist.display();

int elem_cnt = mylist.remove( 1 );
cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
mylist.display();

mylist.remove_all();

cout << "\n-----------------------------------------------\n"
     << "test #2: items at front\n"
     << "-----------------------------------------------\n";

mylist.insert_front( 1 ); mylist.insert_front( 1 );
mylist.insert_front( 1 );
mylist.display();

elem_cnt = mylist.remove( 1 );
cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
mylist.display();

mylist.remove_all();

cout << "\n-----------------------------------------------\n"
     << "test #3: no items present\n"
     << "-----------------------------------------------\n";

mylist.insert_front( 0 ); mylist.insert_front( 2 );
mylist.insert_front( 4 );
mylist.display();

elem_cnt = mylist.remove( 1 );
cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
mylist.display();

mylist.remove_all();

cout << "\n-----------------------------------------------\n"
     << "test #4: items at front and end\n"
     << "-----------------------------------------------\n";
```

```
    mylist.insert_front( 1 ); mylist.insert_front( 1 );
    mylist.insert_front( 1 );

    mylist.insert_front( 0 ); mylist.insert_front( 2 );
    mylist.insert_front( 4 );

    mylist.insert_front( 1 ); mylist.insert_front( 1 );
    mylist.insert_front( 1 );

    mylist.display();

    elem_cnt = mylist.remove( 1 );
    cout << "\n" << "Removed " << elem_cnt << " of the value 1\n";
    mylist.display();
}
```

Kun ohjelma on käännetty ja suoritetaan, ohjelma tulostaa seuraavat tulokset:

```
----------------------------------------------------
test #1: items at end
----------------------------------------------------

( 6 )( 4 3 2 1 1 1 )

Removed 3 of the value 1

( 3 )( 4 3 2 )

----------------------------------------------------
test #2: items at front
----------------------------------------------------

( 3 )( 1 1 1 )

Removed 3 of the value 1

( 0 )( )

----------------------------------------------------
test #3: no items present
----------------------------------------------------

( 3 )( 4 2 0 )

Removed 0 of the value 1

( 3 )( 4 2 0 )
```

```
-----------------------------------------------------
test #4: items at front and end
-----------------------------------------------------

( 9 )( 1 1 1 4 2 0 1 1 1 )

Removed 6 of the value 1

( 3 )( 4 2 0 )
```

Viimeiset kaksi operaatiota, jotka haluaisimme tehdä, ovat yhdistäminen (yhden listan lisääminen toisen perään) ja päinvastaiseen järjestykseen laittaminen (elementtien järjestyksen vaihtaminen). Ensimmäinen toteutuksemme concat()-funktiosta on hieman virheellinen. Näetkö ongelman?

```
void ilist::concat( const ilist &il )
{
   if ( ! _at_end )
        _at_front = il._at_front;
   else _at_end->next( il._at_front );
   _at_end = il._at_end;
}
```

Ongelma on, että kaksi ilist-oliota osoittavat nyt samaan jäsenjoukkoon. Muutokset yhteen ilist-olioon kuten insert() tai remove() vaikuttavat virheellisesti myös toiseen ilist-olioon. Yksinkertaisin ratkaisu tähän ongelmaan on kopioida jokainen jäsen. Korjattu concat()-funktiomme käyttää insert_end()-funktiota:

```
void
ilist::
concat( const ilist &il )
{
      ilist_item *ptr = il._at_front;
      while ( ptr ) {
            insert_end( ptr->value() );
            ptr = ptr->next();
      }
}
```

Tässä on toteutuksemme reverse()-funktiosta:

```
void
ilist::
reverse()
{
   ilist_item *ptr = _at_front;
   ilist_item *prev = 0;

   _at_front = _at_end;
   _at_end = ptr;
```

```
    while ( ptr != _at_front )
    {
        ilist_item *tmp = ptr->next();
        ptr->next( prev );
        prev = ptr;
        ptr = tmp;
    }

    _at_front->next( prev );
}
```

Seuraavassa pienessä ohjelmassa kokeillaan toteutustamme:

```
#include <iostream>
#include "ilist.h"

int main()
{
    ilist mylist;

    for ( int ix = 0; ix < 10; ++ix )
        { mylist.insert_front( ix ); }

    mylist.display();

    cout << "\n" << "reverse the list\n";
    mylist.reverse(); mylist.display();

    ilist mylist_too;
    mylist_too.insert_end( 0 ); mylist_too.insert_end( 1 );
    mylist_too.insert_end( 1 ); mylist_too.insert_end( 2 );
    mylist_too.insert_end( 3 ); mylist_too.insert_end( 5 );

    cout << "\n" << "mylist_too:\n";
    mylist_too.display();

    mylist.concat( mylist_too );
    cout << "\n" << "mylist after concat with mylist_too:\n";
    mylist.display();

}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
( 10 )( 9 8 7 6 5 4 3 2 1 0 )

reverse the list

( 10 )( 0 1 2 3 4 5 6 7 8 9 )
```

```
mylist_too:

( 6 )( 0 1 1 2 3 5 )

mylist after concat with mylist_too:

( 16 )( 0 1 2 3 4 5 6 7 8 9 0 1 1 2 3 5 )
```

Toisaalta suunnitelmamme ja toteutuksemme on valmis. Emme ole ainoastaan toteuttaneet tarpeellisiksi tunnistamiamme operaatioita, vaan olemme myös testanneet ne varmistuaksemme jonkin tason virheettömyydestä. Puutteet eivät kohdistu niihin, jotka olemme tehneet, vaan niihin, joita emme ole tehneet.

Ilist-luokkamme vakavin puute on se, että käyttäjät eivät pysty käymään läpi listan elementtejä. Toteutuksemme mukaisesti luokkamme ei yksinkertaisesti tue sitä ja koska kapseloimme toteutuksen, ei käyttäjällä ole mitään keinoa suoraan tehdä sitä.

Toinen puute on se, että luokkamme ei tue yhden ilist-luokan olion alustusta tai sijoitusta toisella ilist-luokan oliolla. Vaikka tämä päätös oli tarkoituksellinen osaltamme, ei se tee sitä harmittomammaksi käyttäjillemme. Korjatkaamme jokainen puute vuorollaan alkaen alustamisesta ja kopioinnista.

Jotta yhden ilist-olion voisi alustaa toisella, pitää määritellä ilist-luokalle kopiointimuodostaja. Syy siihen, miksi alun perin emme sallineet tätä, on siinä, että oletuskäyttäytyminen on täysin väärin listaluokallemme (yleensä se on väärin kaikille luokille, jotka sisältävät osoitinjäseniä) ja on parempi kiusata käyttäjää olemalla tekemättä jotain toimintoa sen sijaan, että tekisi toiminnon väärin ja rikkoisi käyttäjän ohjelman. (Syy siihen, miksi oletuskäyttäytyminen on väärin, selitetään kohdassa 14.5.) Kopiointimuodostaja käyttää hyväkseen insert_end()-funktiota:

```
ilist::ilist( const ilist &rhs )
{
   ilist_item *pt = rhs._at_front;
   while ( pt ) {
      insert_end( pt->value() );
      pt = pt->next();
   }
}
```

Kopioinnin sijoitusoperaattorin pitää yksinkertaisesti poistaa kaikki jäsenet remove_all()-funktiolla ja sitten lisätä toisen ilist-olion arvot insert_end()-funktiolla. Koska lisäyksen koodi on samanlainen molemmissa tapauksissa, voimme jakaa sen insert_all()-jäsenen avulla:

```
void ilist::insert_all( const ilist &rhs )
{
   ilist_item *pt = rhs._at_front;
   while ( pt ) {
      insert_end( pt->value() );
      pt = pt->next();
   }
}
```

Sitten toteutamme kopiointimuodostajan ja kopioinnin sijoitusoperaattorin seuraavasti:

```
inline ilist::ilist( const ilist &rhs )
   : _at_front( 0 ), _at_end( 0 )
   { insert_all( rhs ); }

inline ilist&
ilist::operator=( const ilist &rhs ) {
   remove_all();
   insert_all( rhs );
   return *this;
}
```

Lopuksi käyttäjän pitää pystyä käymään läpi ilist-luokan yksittäiset elementit. Eräs strategia tukea tätä on yksinkertaisesti antaa pääsy _at_front:iin:

```
ilist_item *ilist::front() { return _at_front(); }
```

Sitten käyttäjällä on mahdollisuus toteuttaa samanlainen yleinen silmukkaratkaisu, kuin minkä teimme aikaisemmin:

```
ilist_item *pt = mylist.front();
while ( pt ) {
      do_something( pt->value() );
      pt = pt->next();
}
```

Vaikka tällä saa homman tehtyä, ei se ole meidän paras ratkaisumme. Sen sijaan pidämme parempana ratkaisuna tukea yleisempää käsitettä säiliön elementtien läpikäynnistä. Tässä kohtaa tuemme vähintään muotoa

```
for ( ilist_item *iter = mylist.init_iter();
    iter;
    iter = mylist.next_iter() )
        do_something( iter->value() );
```

(Luvuissa 6 ja 12 katsomme iteraattorityyppejä, joita on määritelty vakiokirjaston säiliötyypeille ja geneerisille algoritmeille. (Kävimme iteraattorit lyhyesti läpi kohdassa 2.8.))

Meidän iteraattorimme on hieman enemmän kuin osoitin, koska se muistaa toistokerran nykyisen jäsenen, pystyy palauttamaan seuraavan jäsenen ja havaitsee toistojen päättymisen. Oletusarvoisesti init_iter() alustaa iteraattorin arvolla _at_front. Valinnaisesti käyttäjä voi välittää ilist_item-osoittimen, josta toisto aloitetaan. next_iter() palauttaa listan seuraavan jäsenen tai arvon 0, jos toisto on loppunut. Toteutukseen kuuluu lisäksi ilist_item-osoitin:

```
class ilist {
public:
   // ...
   init_iter( ilist_item *it = 0 );
private:
   // ...
   ilist_item *_current;
```

```
};
```

init_iter() näyttää tältä:

```
inline ilist_item*
ilist::init_iter( ilist_item *it )
{
   return _current = it ? it : _at_front;
}
```

next_iter() siirtää _current-osoitinta seuraavaan jäseneen ja palauttaa sen, ellei toisto pääty. Jos toisto on päättynyt, next_iter() palauttaa arvon 0, kunnes init_iter() asettaa uudelleen _current-osoittimen. Tässä on toteutuksemme:

```
inline ilist_item*
ilist::
next_iter()
{
   ilist_item *next = _current
                  ? _current = _current->next()
                  : _current;

   return next;
}
```

Tukemme tälle läpikäynnille voisi olla ongelmallista, jos jäsen, johon _current osoittaa, olisi poistettu. Ratkaisumme on muokata funktioita remove_front() ja remove() ja testata, onko _current-osoittimen osoittama jäsen poistettu. Jos se on poistettu, _current-osoitinta siirretään osoittamaan seuraavaan jäseneen (tai ei mihinkään jäseneen, jos poistettu jäsen on listan viimeinen). Jos kaikki jäsenet on poistettu, _current-osoitinta ei aseteta osoittamaan mihinkään jäseneen. Tässä on uudistettu remove_front()-funktiomme:

```
inline void
ilist::remove_front()
{
   if ( _at_front ) {
      ilist_item *ptr = _at_front;
      _at_front = _at_front->next();

      // ei haluta current-osoittimen osoittavan poistettuun jäseneen
      if ( _current == ptr )
         _current = _at_front;

      bump_down_size();
      delete ptr;
   }
}
```

Tässä on olennainen osa uudistetusta remove()-funktiosta:

```
while ( plist )
   {
      if ( plist->value() == value )
      {
          prev->next( plist->next() );

          if ( _current == plist )
              _current = prev->next();
```

Mitä, jos jäsen lisätään sen jäsenen eteen, johon _current osoittaa? Siinä tapauksessa emme muokkaa _current-osoitinta. Tahdistaakseen läpikäynnin uudelleen käyttäjän pitää käynnistää init_iter(). Toisaalta, kun alustamme tai kopioimme yhden ilist-luokan olion toiseen, ei _current-osoitinta kopioida, vaan asetetaan osoittamaan tyhjää.

Tässä on pieni ohjelma, jossa kokeillaan kopiointimuodostajaamme ja kopioinnin sijoitus-operaattoria kuten myös läpikäynnin tukeamme:

```
#include <iostream>
#include "ilist.h"

int main()
{
   ilist mylist;

   for ( int ix = 0; ix < 10; ++ix ) {
        mylist.insert_front( ix );
        mylist.insert_end( ix );
   }

   cout << "\n" << "Use of init_iter() and next_iter() "
       << "to iterate across each list item:\n";

   ilist_item *iter;
   for ( iter = mylist.init_iter();
        iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";

   cout << "\n" << "Use of copy constructor\n";

   ilist mylist2( mylist );
   mylist.remove_all();

   for ( iter = mylist2.init_iter();
        iter; iter = mylist2.next_iter() )
        cout << iter->value() << " ";

   cout << "\n" << "Use of copy assignment operator\n";
```

```
    mylist = mylist2;

    for ( iter = mylist.init_iter();
        iter; iter = mylist.next_iter() )
        cout << iter->value() << " ";

    cout << "\n";

}
```

Kun tämä ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
Use of init_iter() and next_iter() to iterate across each list item:
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Use of copy constructor
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
Use of copy assignment operator
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
```

5.11.1 Geneerisen listaluokan muodostaminen

Ilist-luokkamme on pahasti rajoittunut siten, että se voi nykyisellään käsitellä vain int-tyyppisiä elementtejä. Yleisempi ja hyödyllisempi listatyyppi tukisi sekä sisäisiä että luokkatyyppejä. Kuinka muuntaisimme ilist-luokkaamme niin, että se tukisi laajemmin erilaisia elementtityyppejä, ilman, että joutuisimme mittavaan uudelleenohjelmointiin tai koodin tuplaamiseen? Luokan mallimekanismissa on ratkaisu (sitä käsitellään tarkemmin luvussa 16).

Luokkamalli saa parametroinnin avulla erotettua tyyppiriippuvaiset piirteet luokan suunnittelusta — meidän tapauksessamme elementin taustalla olevan tyypin, jonka listamme sisältää. Myöhemmin käyttäjä, joka haluaa tietyntyyppisen listan, antaa todellisen tyypin mallin parametrina. Esimerkiksi:

```
list< string > slist;
```

luo ilmentymän listamalliluokastamme, joka muodostuu string-olioista, kun taas

```
list< int > ilist;
```

luo ilmentymän, joka on samanarvoinen alkuperäisen käsin koodatun ilist-luokkamme kanssa. Käyttämällä luokan mallimäärittelyä voimme tukea rajoittamatonta määrää listan elementtityyppejä yhdellä luokkamallin toteutuksella. Käykäämme läpi, kuinka teemme sen, ja keskittykäämme list_item-luokkaamme.

Luokkamallin määrittely alkaa avainsanalla template, jonka jälkeen tulee parametriluettelo kulmasuluissa. Tyyppiparametri muodostuu joko avainsanasta typename tai class ja tunnuksesta. Esimerkiksi:

```
template <class elemType>
class list_item;
```

Tämä esittelee list_item:in luokkamalliksi yhdellä tyyppiparametrilla. elemType on mikä tahansa tunnus, jonka olemme valinneet nimeämään tyyppiparametrimme. Seuraava esittely on samanarvoinen kuin list_item-luokan esittely:

```
template <typename elemType>
class list_item;
```

Avainsanat typename ja class ovat keskenään toisensa korvaavia. typename on uutta C++-standardissa. Se on mnemonisempi, mutta tällä hetkellä vähemmän tuettu kuin alkuperäinen class-avainsana. Käytämme pääosin class-avainsanaa tästä syystä, ja koska vanhoja tapoja on vaikea muuttaa johdonmukaisesti. Joka tapauksessa tässä on määrittelymme list_item-luokkamallista:

```
template <class elemenType>
class list_item {
public:
   list_item( elemType value, list_item *item = 0 )
         : _value( value ) {
           if ( !item )
               _next = 0;
           else {
               _next = item->_next;
               item->_next = this;
           }
   }

   elemType  value() { return _value; }
   list_item* next() { return _next;  }

   void next(  list_item *link    ) { _next = link;       }
   void value( elemType new_value ) { _value = new_value; }

private:
   elemType   _value;
   list_item *_next;
};
```

Aikaisempi int-esiintymä ilist_item-luokkamme määrittelyssä on korvattu elemType-parametrilla list_item-luokkamallissamme. Kun kirjoitamme

```
list_item<double> *ptr = new list_item<double>( 3.14 );
```

kääntäjä sitoo automaattisesti elemType:n todelliseen double-tyyppiin ja luo list_item-luokan, joka kykenee tukemaan tuon tyypin elementtejä.

Ilist-luokan muunnos list-luokkamalliksi tehdään samalla tavalla. Tässä on luokkamallin toteutus:

```
template <class elemType>
class list {
public:
   list()
     : _at_front( 0 ), _at_end( 0 ), _current( 0 ),
```

```
        _size( 0 ) {}
    list( const list& );
    list& operator=( const list& );
    ~list() { remove_all(); }

    void insert( list_item<elemType> *ptr, elemType value );
    void insert_end( elemType value );
    void insert_front( elemType value );
    void insert_all( const list &rhs );

    int  remove( elemType value );
    void remove_front();
    void remove_all();

    list_item<elemType> *find( elemType value );
    list_item<elemType> *next_iter();
    list_item<elemType>* init_iter( list_item<elemType> *it );

    void display( ostream &os = cout );

    void concat( const list& );
    void reverse();
    int size() { return _size; }

private:
    void bump_up_size()   { ++_size; }
    void bump_down_size() { --_size; }

    list_item<elemType> *_at_front;
    list_item<elemType> *_at_end;
    list_item<elemType> *_current;
    int _size;
};
```

Luokkamallin olioita voidaan käyttää täsmälleen samalla tavalla kuin eksplisiittisesti koodaamiamme ilist-luokkaolioita. Pääetu on kykymme tukea rajoittamatonta määrää listatyyppejä yhdellä luokkamallin määrittelyllä.

Mallit muodostavat C++-ohjelmoinnin tärkeän peruskomponentin. Itse asiassa luvussa 6 me katsomme vakiokirjaston säiliötyyppimallien kokoelmaa. Yllättävää ei ole, että tähän kuuluu listaluokkamalli kuten myös vektoriluokkamalli, jotka näimme jo luvuissa 2 ja 3.

Vakiokirjaston listaluokan olemassaolo tuo esille erään vaikean pulman. Olemme päättäneet kutsua luokkaamme nimellä “list” tähän saakka nimen “ilist” sijasta. Valitettavasti se on ristiriidassa vakiokirjaston listaluokan nimen kanssa. Emme voi nyt käyttää molempia luokkia samassa ohjelmassa. Eräs ratkaisu on tietenkin nimetä listaluokkamme uudelleen ristiriidan poistamiseksi. Tämä toimii tietenkin tässä tapauksessa, koska kyseessä on oma koodimme. Monissa tapauksissa tuo ratkaisu ei ole meille niin selvä.

Yleisempi ratkaisu on C++:n nimiavaruusmekanismi. Nimiavaruuden avulla kirjastojen val-

mistajat voivat kapseloida muutoin globaalit nimet estääkseen nimien yhteentörmäykset. Lisäksi nimiavaruuden merkintätavan avulla voidaan käyttää niitä nimiä ohjelmissamme. Esimerkiksi C++:n vakiokirjasto on pakattu std-nimiavaruuteen. Kolmannen painoksemme koodi olisi myös voitu sijoittaa yksilöllisesti nimettyyn nimiavaruuteen:

```
namespace Primer_Third_Edition
{
   template <typename elemType>
   class list_item{ ... };

   template <typename elemType>
   class list{ ... };

   // ...
}
```

Käyttäjä, joka haluaisi kokeilla listaluokkaamme, voisi kirjoittaa seuraavasti:

```
// listaluokkamme otsikkotiedosto
#include “list.h”

// tee määrittelyt näkyviksi ohjelmalle
using namespace Primer_Third_Edition;

// ok: käsittelee listaluokkaamme
list< int > ilist;

// ...
```

(Nimiavaruudet käsitellään tarkemmin kohdissa 8.5 ja 8.6.)

Harjoitus 5.16

Emme määrittele ilist_item-luokalle tuhoajaa, vaikka se sisältää osoitinjäsenen. Syy on se, että ilist_item-luokka ei varaa muistia oliolle, jota _next osoittaa, joten se ei ole velvollinen sen muistin vapauttamiseen. Yleinen aloittelijan virhe on tehdä ilist-luokalle tuhoaja, kuten seuraavassa:

```
// huono suunnitteluvalinta
ilist_item::~ilist_item()
{
   delete _next;
}
```

Kun katsotaan funktioita remove_all() tai remove_front(), selitä, miksi tuhoajan läsnäolo olisi huonoa suunnittelua.

Harjoitus 5.17

Ilist-luokkamme ei tue kumpaakaan näistä lauseista:

```
void ilist::remove_end();
```

```
void ilist::remove( ilist_item* );
```

Miksi arvelet meidän jättäneen ne pois? Hahmottele algoritmi, joka tukisi näitä kahta operaatiota.

Harjoitus 5.18

Muokkaa find()-funktiota niin, että se saa toisen argumentin, ilist_item*, joka ilmaisee, jos se on asetettu, mistä jäsenen etsintä aloitetaan. Ellei sitä ole asetettu, etsinnän tulisi alkaa listan alusta. (Laittamalla tämän uuden parametrin toiseksi argumentiksi ja määrittämällä nollan oletusargumentiksi, säilytämme alkuperäisen julkisen rajapinnan. Koodia, joka käyttää aiempaa find()-funktion määrittelyä, ei tarvitse muuttaa.)

```
class ilist {
public:
   // ...
   ilist_item* find( int value, ilist_item *start_at = 0 );
   // ...
};
```

Harjoitus 5.19

Käytä tätä uutta versiota find()-funktiosta ja toteuta count()-funktio, joka palauttaa arvon esiintymien lukumäärän ilist-listassa. Kirjoita pieni ohjelma, joka testaa toteutustasi.

Harjoitus 5.20

Uudista insert(int value)-funktiota niin, että se palauttaa ilist_item-osoittimen, jonka se juuri on lisännyt.

Harjoitus 5.21

Käytä uudistettua insert()-funktiota ja toteuta

```
void ilist::
insert(ilist_item *begin, int *array_of_value, int elem_cnt );
```

jossa array_of_value osoittaa ilist-listaan lisättävää arvotaulukkoa, elem_cnt on elementtien lukumäärä taulukossa ja begin ilmaisee, mistä elementtien lisäys aloitetaan. Jos esimerkiksi on seuraavia ilist-arvoja

```
(3)( 0 1 21 )
```

ja seuraava taulukko

```
int ia[] = { 1, 2, 3, 5, 8, 13 };
```

uutta lisäysoperaatiota kutsuttaisiin näin:

```
ilist_item *it = mylist.find( 1 );
mylist.insert(it, ia, 6 );
```

Se muokkaisi mylist-listaa seuraavasti:

(9)(0 1 1 2 3 5 8 13 21)

Harjoitus 5.22

Eräs ongelma funktioiden concat() ja reverse() kanssa on, että molemmat niistä muokkaavat alkuperäistä listaa. Tämä ei ole aina toivottavaa. Tee vaihtoehtoinen operaatiopari, joka palauttaa uuden ilist-olion:

```
ilist ilist::reverse_copy();
ilist ilist::concat_copy( const ilist &rhs );
```