8

Viittausalue ja elinaika

Tämä luku vastaa kahteen tärkeään kysymykseen, jotka koskevat C++:n esittelyitä: missä voidaan esittelyssä esille tullutta nimeä käyttää? Milloin ohjelmalle on turvallista käyttää oliota tai käynnistää funktio; tarkoittaa, mikä on esittelyssä esille tuodun suorituksenaikaisen olion elinaika? Vastataksemme ensimmäiseen kysymykseen, esittelemme viittausalueet ja katsomme, kuinka ne rajoittavat sitä, missä nimiä voidaan käyttää ohjelman tekstitiedostossa. Luvussa esitetään monia C++-viittausalueita: globaali ja paikallinen viittausalue kuten myös pitemmälle menevä aihe nimiavaruuksien viittausalue, joka esitellään luvun lopussa. Vastataksemme toiseen kysymykseen, kuvaamme, kuinka esittelyt tuovat esille oliot ja funktiot (kohteet, jotka kestävät koko ohjelman ajan), paikalliset oliot (oliot, jotka kestävät osan ohjelman ajasta) ja dynaamisesti varatut oliot (oliot, joiden elinaikaa kontrolloi ohjelmoija). Tutkimme myös tiettyjä suorituksenaikaisia ominaisuuksia, jotka liittyvät näihin olioihin ja funktioihin.

8.1 Viittausalue

Jokaisen nimen C++-ohjelmassa pitää viitata yksilölliseen kohteeseen (olioon, funktioon, tyyppiin tai malliin). Tämä ei tarkoita, että nimeä voidaan käyttää vain kerran C++-ohjelmassa: nimeä voidaan käyttää uudelleen viittaamaan eri kohteeseen edellyttäen, että on olemassa jokin *asiayhteys*, jonka avulla nimen eri merkitykset voidaan erottaa. Yleisin asiayhteys, jota käytetään nimien merkityksien erottamiseen, on *viittausalue* (*scope*). C++ tukee kolmea viittausalueen muotoa: *paikallinen* viittausalue, *nimiavaruuden* viittausalue *ja luokan* viittausalue.

Paikallinen viittausalue on osa ohjelmatekstiä, joka sijaitsee funktion määrittelyssä (tai funktiolohkossa). Jokainen funktio edustaa eri paikallista viittausaluetta. Funktiossa jokainen yhdistetty lause (eli lohko) edustaa myös eri paikallista viittausaluetta.

Nimiavaruuden viittausalue on osa ohjelmatekstiä, joka ei sisälly funktion esittelyyn, funktion määrittelyyn tai luokan määrittelyyn. Ohjelman ulointa nimiavaruuden viittausaluetta kut-

sutaan *globaaliksi viittausalueeksi* eli *globaalin nimiavaruuden viittausalueeksi*. Globaalille viittausalueelle voidaan määritellä olioita, funktioita, tyyppejä ja malleja. Ohjelmoija voi määritellä *käyttäjän esittelemiä* nimiavaruuksia sisäkkäin globaaliin viittausalueeseen käyttämällä *nimiavaruuden määrittelyjä*. Jokainen käyttäjän esittelemä nimiavaruus on eri viittausalueella ja ne ovat erillään globaalista viittausalueesta. Kuten globaali viittausalue, myös käyttäjän esittelemät nimiavaruudet voivat sisältää esittelyitä ja määrittelyjä olioille, funktioille, tyypeille ja malleille kuten myös sisäkkäisille käyttäjän esittelemille nimiavaruuksille. Käyttäjän esittelemät nimiavaruudet käsitellään kohdissa 8.5 ja 8.6.

Jokainen luokan määrittely tuo esille erillisen *luokan viittausalueen*. Luokkien määrittelyt ja viittausalueet kuvataan luvussa 13.

Nimi voi viitata eri kohteisiin eri viittausalueilla ilman ristiriitoja. Esimerkiksi seuraavassa ohjelmassa on neljä kohdetta nimeltään s1:

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

// vertaa merkkijonoja s1 ja s2 aakkosten mukaan
int lexicoCompare( const string &s1, const string &s2 )
    { ... }

// vertaa merkkijonojen s1 ja s2 kokoa
int sizeCompare( const string &s1, const string &s2 )
    { ... }

typedef int (*PFI)( const string &, const string & );
// lajittele merkkijonotaulukko
void sort( string *s1, string *s2, PFI compare =lexicoCompare )
    { ... }

string s1[10] = { "a", "light", "drizzle", "was", "falling",
            "when", "they", "left", "the", "school" };

int main()
{
    // kutsu sort()-funktiota -- käytä vertailuun oletusargumenttia
    // viittaa globaaliin taulukkoon s1
    sort( s1, s1 + sizeof(s1)/sizeof(s1[0]) - 1 );

    // näytä lajiteltu taulukko
    for ( int i = 0; i < sizeof(s1) / sizeof(s1[0]); ++i )
        cout << s1[ i ].c_str() << "\n\t";
}
```

Koska funktioiden lexicoCompare(), sizeCompare() ja sort() määrittelyillä on erilainen globaali viittausalue, jokaisessa näillä viittausalueilla voidaan määritellä muuttuja nimeltään s1.

Esitelty nimi on mahdollisesti *näkyvä* tuosta kohdasta sen viittausalueen loppuun, jossa se on esitelty (mukaan lukien sisäkkäiset viittausalueet). Siten lexicoCompare()-funktion parametrin

nimi s1 on näkyvissä viittausalueensa loppuun eli lexicoCompare()-funktion määrittelyn loppuun. Globaalin s1-taulukon nimi on näkyvissä esittelypisteestään tiedoston loppuun, mukaan lukien sisäkkäiset viittausalueet kuten main()-funktion määrittely.

Yleensä nimi pitää esitellä viittaamaan vain yhteen kohteeseen viittausalueen sisällä. Jos esimerkiksi seuraava esittely lisätään edelliseen esimerkkiin, saadaan aikaan käännöksenaikainen virheilmoitus s1-taulukon esittelyn jälkeen:

```
void s1(); // virhe: nimen s1 uudelleenesittely
```

Ylikuormitetut funktiot ovat poikkeus tähän sääntöön: on mahdollista määritellä useampi kuin yksi funktio samalla nimellä samalla viittausalueella edellyttäen, että jokaisella funktiolla on erilainen parametriluettelo. Luvussa 9 käsitellään ylikuormitettuja funktioita.

C++:ssa nimi pitää esitellä ennen kuin sitä voidaan käyttää lausekkeissa. Ellei s1:lle löydy esittelyä ennen sen käyttöä main()-funktiossa, se saa aikaan käännösvirheen. *Nimiresoluutio* on prosessi, jossa lausekkeessa käytetty nimi liitetään esittelyyn. Se on prosessi, jossa nimelle annetaan merkitys. Tämä prosessi riippuu siitä, miten nimeä käytetään ja millä viittausalueella nimeä käytetään. Läpi tämän kirjan käsittelemme nimiresoluutiota useissa yhteyksissä. Paikallisen viittausalueen nimiresoluutiota käsitellään seuraavassa alikohdassa, funktiomallin määrityksen nimiresoluutiota kohdassa 10.9, luokan viittausalueen nimiresoluutiota luvun 13 lopussa ja luokkamallin määrittelyn nimiresoluutiota kohdassa 16.12.

Viittausalueet ja nimiresoluutio ovat käännöksenaikaisia käsityksiä; ne koskevat jotain tiettyä osaa ohjelman tekstistä. Nämä käsitykset antavat merkityksen ohjelmatekstille lähdetiedostossa. Kääntäjä tulkitsee ohjelmatekstiä, jota se lukee viittausalue- ja nimiresoluutiosääntöjen mukaisesti.

8.1.1 Paikallinen viittausalue

Paikallinen viittausalue on osa ohjelmatekstiä, joka sisältyy funktion määrittelyyn (eli funktiolohkoon). Jokainen funktio edustaa eri paikallista viittausaluetta. Funktiossa jokainen yhdistetty lause (eli lohko) edustaa myös omaa paikallista viittausaluetta. Paikallisten lohkojen viittausalueita voidaan laittaa sisäkkäin. Esimerkiksi seuraava funktio määrittelee kaksi tasoa paikallisia viittausalueita, jotka tekevät kokonaislukujen binäärihaun lajitellusta vektorista:

```
const int notFound = -1; // globaali viittausalue

int binSearch( const vector<int> &vec, int val )
{ // paikallinen viittausalue: taso 1
    int low = 0;
    int high = vec.size() - 1;

    while ( low <= high )
    { // paikallinen viittausalue: taso 2
        int mid = ( low + high ) / 2;
        if ( val == vec[ mid ] ) return mid;
        if ( val < vec[ mid ] )
```

```
            high = mid - 1;
        else low = mid + 1;
    }
    return notFound; // paikallinen viittausalue: taso 1
}
```

Ensimmäinen paikallinen viittausalue on binSearch()-funktion runko. Tässä ensimmäisessä paikallisessa viittausalueessa esitellään funktion parametrit vec ja val. Se esittelee myös muuttujat low ja high. binSearch()-funktion sisällä while-silmukka määrittelee sisäkkäisen, paikallisen viittausalueen. Tämä sisäkkäinen paikallinen viittausalue esittelee yhden muuttujan, kokonaisluvun mid. Tämä sisäkkäinen paikallinen viittausalue käyttää funktion parametreja vec ja val ja paikallisia muuttujia high ja low. Globaali viittausalue sulkee sisäänsä molemmat paikalliset viittausalueet. Se esittelee yhden kokonaislukuvakion: notFound.

Funktion parametrien vec ja val nimet kuuluvat funktion rungon ensimmäiselle paikalliselle viittausalueelle. Näitä nimiä ei voi esitellä uudelleen tässä ensimmäisessä paikallisessa viittausalueessa. Esimerkiksi:

```
int binSearch( const vector<int> &vec, int val )
{ // paikallinen viittausalue: taso 1
    int val; // virhe: virheellinen val-nimen uudelleenesittely
    // ...
```

Parametrien nimiä voidaan käyttää binSearch()-funktion rungossa kuten myös while-silmukan sisäkkäisellä viittausalueella. Funktion parametreihin vec ja val ei voida viitata binSearch()-funktion rungon ulkopuolelta.

Paikallisen viittausalueen nimiresoluutio etenee seuraavasti: esittelyä etsitään nimen välittömältä viittausalueelta, jossa sitä on käytetty. Jos esittely löytyy, nimi on ratkaistu; ellei löydy, esittelyä etsitään sitä ympäröivältä viittausalueelta. Tämä prosessi etenee, kunnes joko esittely löytyy tai globaali viittausalue on etsitty. Jos tapahtuu jälkimmäinen, eikä nimelle löydy esittelyä, se saa aikaan virheen.

Järjestyksen vuoksi, jolla viittausalueet etsitään nimiresoluution aikana, saman nimen esittely sisäkkäisessä viittausalueessa *piilottaa* (*peittää, kumoaa*) sitä ympäröivän viittausalueen esittelyn. Jos aikaisemmassa esimerkissä low-muuttuja olisi esitelty globaalilla viittausalueella ennen binSearch()-funktion määrittelyä, viittaisi low-muuttujan käyttö while-silmukan sisäkkäisellä paikallisella alueella yhä paikalliseen low-muuttujan esittelyyn; paikallinen esittely piilottaisi globaalin esittelyn. Esimerkiksi:

```
int low;

int binSearch( const vector<int> &vec, int val )
{
    // low-muuttujan paikallinen esittely
    // piilottaa globaalin viittausalueen esittelyn
    int low = 0;
    // ...
```

```
    // low on paikallinen muuttuja
    while ( low <= high )
    { // ...
    }
    // ...
}
```

Jotkut lauseet sallivat muuttujan määrittelyn ohjausrakenteessaan. Esimerkiksi for-silmukka sallii muuttujan määrittelyn alustuslauseessaan:

```
for ( int index = 0; index < vecSize; ++index )
{
    // index on näkyvissä vain täällä
    if ( vec[ index ] == someValue )
        break;
}
// virhe: index ei näy täällä
if ( index != vecSize ) // elementti löytyi
```

Muuttujat, kuten index, jotka on määritelty for-silmukan alustuslauseessa, ovat näkyvissä vain itse for-silmukan paikallisella viittausalueella ja sen sisältämissä paikallisissa viittausalueissa (asia on näin C++-standardissa, mutta oli eri tavalla C++-esistandardissa) aivan kuten forlause olisi kirjoitettu näin:

```
// Esitys kääntäjän muuntamisesta
{ // näkymätön yhdistetty lause
    int index = 0;
    for ( ; index < vecSize; ++index )
    {
        // ...
    }
}
```

Tämä estää ohjelmoijaa käsittelemästä ohjausmuuttujaa silmukan paikallisen viittausalueen ulkopuolella. Jos ohjelmoija haluaa testata index-ohjausmuuttujaa päätelläkseen, löytyikö arvo, pitää koodikatkelma kirjoittaa seuraavasti:

```
int index = 0;
for ( ; index < vecSize; ++index )
{
    // ...
}
// ok: index on näkyvissä täällä
if ( index != vecSize ) // elementti löytyi
```

Koska for-silmukan alustuslauseessa esitelty muuttuja on paikallinen itse silmukalle, muuttujan nimeä voidaan käyttää uudelleen muiden for-silmukoiden ohjausrakenteissa samalla paikallisella viittausalueella. Esimerkiksi:

```
void fooBar( int *ia, int sz )
{
    for (int i=0; i<sz; ++i) ... // ok
    for (int i=0; i<sz; ++i) ... // ok: eri i
    for (int i=0; i<sz; ++i) ... // ok: eri i
}
```

Samalla tavalla muuttuja voidaan esitellä if- tai switch-lauseen ehdossa kuten while- ja for-silmukan ehdossa. Esimerkiksi:

```
if ( int *pi = getValue() )
{
    // pi != 0 -- on OK käyttää merkintää *pi tässä
    int result = calc(*pi);
    // ...
}
else
{
    // pi näkyvyssä myös täällä
    // pi == 0
    cout << "error: getValue() failed" << endl;
}
```

Muuttujat kuten pi, jotka on määritelty if-lauseen ehdossa, ovat näkyvissä vain if-lauseessa, siihen liittyvässä else-lauseessa ja näiden lauseiden sisäkkäisillä viittausalueilla. Ehdon arvo on muuttujan arvo heti, kun se on alustettu. Jos pi alustetaan arvolla 0 eli osoittimen nolla-arvolla, on ehto epätosi ja if-lauseen else-osa suoritetaan. Jos pi alustetaan millä muulla arvolla tahansa kuin nolla-arvolla, ehto on tosi ja if-osa suoritetaan. If-lausetta, switch-lausetta, for-silmukkalausetta ja while-silmukkalausetta käsitellään luvussa 5.

Harjoitus 8.1

Yksilöi seuraavasta koodiesimerkistä eri viittausalueet. Mitkä seuraavista ix:n esittelyistä ovat virheellisiä, vai onko yksikään? Selitä miksi.

```
int ix = 1024;
int ix();

void func( int ix, int iy ) {
    int ix = 255;

    if ( int ix = 0 ) {
        int ix = 79;
        {
            int ix = 89;
        }
    }
    else {
        int ix = 99;
```

```
    }
}
```

Harjoitus 8.2

Mihin esittelyihin seuraavassa koodiesimerkissä ix ja iy viittaavat?

```
int ix = 1024;

void func( int ix, int iy ) {
    ix = 100;

    for( int iy = 0 ; iy < 400; iy += 100 ) {
        iy += 100;
        ix = 300;
    }
    iy = 400;
}
```

8.2 Globaalit oliot ja funktiot

Kun funktio esitellään globaalilla viittausalueella, sitä kutsutaan *globaaliksi funktioksi*. Kun muuttuja esitellään globaalilla viittausalueella, sitä kutsutaan *globaaliksi olioksi*. Globaali olio on suorituksenaikainen kohde, joka on olemassa koko ohjelman keston ajan. Sen muistialueen *elinaika*, jossa olio sijaitsee, alkaa ohjelman käynnistyksen yhteydessä ja loppuu ohjelman päättyessä.

Globaalilla funktiolla, jota kutsutaan tai jonka osoite on otettu, pitää olla määrittely. Samalla tavalla globaalilla oliolla, jota käytetään ohjelmassa, pitää olla määrittely. Globaalit oliot ja muut kuin välittömät globaalit funktiot saa määritellä ohjelmassa vain kerran. Välittömiä eli inline-funktioita voidaan määritellä ohjelmassa useammin kuin kerran, kunhan määrittelyt ovat täsmälleen samanlaisia. Tätä vaatimusta, että globaaleilla olioilla ja funktioilla voi olla vain yksi määrittely tai täsmälleen sama määrittely useammin ohjelmassa, kutsutaan *yhden määrittelyn säännöksi* (*one definition rule*, ODR). Tässä kohdassa näemme, kuinka globaaleja olioita ja funktioita esitellään ja määritellään ohjelmiimme, jotka noudattavat ODR-sääntöä.

8.2.1 Esittelyt vastaan määrittelyt

Kuten näimme luvussa 7, funktion *esittely* määrittää funktion nimen kuten myös paluutyypin ja parametriluettelon. Tämän tiedon lisäksi funktion *määrittely* tuottaa funktion rungon lausejoukkoineen aaltosulkujen sisälle. Funktio pitää esitellä ennen kuin sitä voidaan käyttää. Esimerkiksi:

```
// funktion calc() esittely
// määrittely on tehty toiseen tiedostoon
void calc(int);

int main()
{
```

```
    int locl = get(); // virhe: funktiota get() ei ole esitelty
    calc(locl);       // ok: funktion calc() esittely on löytynyt
    // ...
}
```

Olion määrittelyllä on seuraavat kaksi muotoa:

```
tyyppi_määre  olion_nimi;
tyyppi_määre  olion_nimi  =  alustaja;
```

Esimerkiksi seuraavassa on olion obj1 määrittely. Tässä määrittelyssä obj1 alustetaan arvolla 97:

```
int obj1 = 97;
```

Seuraavassa on olion obj2 määrittely, vaikka alustajaa ei ole määritetty:

```
int obj2;
```

On taattu, että olion muistialue, joka on määritelty globaalille viittausalueelle ilman eksplisiittistä alustajaa, alustetaan alkuarvolla 0. Täten seuraavien kahden muuttujan, var1 ja var2, alkuarvot ovat 0:

```
int var1 = 0;
int var2;
```

Ohjelmassa saa olla vain yksi määrittely globaalille oliolle. Koska olio pitää esitellä tiedostossa ennen kuin sitä voidaan käyttää, pitää olla mahdollista, että ohjelma, joka muodostuu useista tiedostoista, pystyy esittelemään olion tiedostossa määrittelemättä sitä. Kuinka vain yksinkertaisesti esittelemme olion?

Avainsana extern mahdollistaa metodin olion esittelylle ilman, että tämä pitää määritellä. Se lupaa itse asiassa samalla tavalla kuin funktion esittely, että olio on määritelty jossain muualla, joko jossain tämän tekstitiedoston yhteydessä tai ohjelman jossain toisessa tekstitiedostossa. Esimerkiksi

```
extern int i;
```

on “lupaus” ohjelmalle, että jossakin on olemassa määrittely kuten tässä

```
int i;
```

Extern-esittely ei saa aikaan muistinvarausta. Se voi esiintyä useita kertoja saman ohjelman samassa tiedostossa tai eri tiedostoissa. Tyypillisesti esittely esiintyy kuitenkin vain kerran julkisessa otsikkotiedostossa, joka on otettu mukaan tarpeen vaatiessa tiedostoihin, jotka viittaavat globaaliin olioon.

```
// otsikkotiedosto
extern int obj1;
extern int obj2;
```

```
// tekstitiedosto
int obj1 = 97;
int obj2;
```

Globaalin olion esittelyä, jossa määritetään sekä extern-avainsana että eksplisiittinen alustaja, pidetään tuon olion määrittelynä. Muistitila varataan ja kaikki tuon olion seuraavat määrittelyt saavat aikaan virheen. Esimerkiksi:

```
extern const double pi = 3.1416; // määrittely
const double pi; // virhe: pi:n uudelleen määrittely
```

Avainsana extern voidaan määrittää myös funktion esittelyssä. Sen ainoa vaikutus on saada esittelyn implisiittinen "määritelty jossain muualla" -luonne eksplisiittiseksi. Sellaisen esittelyn muoto on seuraava:

```
extern void putValues( int*, int );
```

8.2.2 Välitiedoston esittelyiden täsmäys

Olion tai funktion esittelyn eräs ajoittainen ongelma on, että eri tiedostoissa olevat esittelyt voivat erota toisistaan tai muuttua ajan mittaan. C++:ssa on tuki, joka auttaa havaitsemaan eroja funktioiden esittelyiden välillä eri tiedostoissa.

Esimerkiksi tiedostossa token.C funktio addToken() on määritelty saamaan yhden unsigned char -tyyppisen parametrin. Tiedostossa lex.C, jossa sitä kutsutaan, addToken() on esitelty saamaan yhden char-tyyppisen parametrin.

```
// ---- tiedostossa token.C ----
int addToken( unsigned char tok ) { /* ... */ }

// ---- tiedostossa lex.C ----
extern int addToken( char );
```

Kun lex.C-tiedostossa kutsutaan funktiota addToken(), se saa aikaan virheen linkitysvaiheessa. Jos ohjelma olisi linkittynyt virheittä, olisi seuraava skenaario mahdollinen: Sun Sparc -työasemassa käännetty ja testattu ohjelma suoritetaan virheittä. Ohjelma lähetetään kentälle, jossa käytetään IBM 390 -konetta. Ohjelma kääntyy ongelmitta. Mutta valitettavasti, kun ohjelma ensimmäisen kerran suoritetaan, se epäonnistuu surkeasti. Ei edes yksinkertaisin testiohjelma toimi. Mitä on voinut tapahtua?

Tässä on osa esittelyistä:

```
const unsigned char INLINE = 128;
const unsigned char VIRTUAL = 129;
```

Funktion addToken() kutsu näyttää tältä:

```
curTok = INLINE;
// ...
addToken( curTok );
```

char-merkit on toteutettu yhdessä koneessa etumerkillisenä ja etumerkittömänä toisessa. Funktion addToken() virheellinen esittely saa aikaan jokaisella parametrilla, jonka arvo on suurempi kuin 127, ylivuodon koneessa, jossa char-tyyppi on etumerkillinen. Jos koodin olisi sallittu kääntyä ja linkittyä, siitä olisi todennäköisesti aiheutunut suorituksen aikana pahoja seuraamuksia.

C++:ssa on mekanismi, jolla funktion parametrien tyypit ja niiden lukumäärä koodataan funktion nimeen. Tätä mekanismia kutsutaan *tyyppiturvalliseksi linkitykseksi*. Tyyppiturvallinen linkitys auttaa toteutusta sieppaamaan funktioiden esittelyiden eroavaisuudet eri tiedostoissa. Edellisessä esimerkissä unsigned char -tyyppinen parametri ja char-tyyppinen parametri olivat erilaisia tyypeiltään. Tyyppiturvallisen linkityksen vuoksi funktio addToken(), joka on esitelty tiedostossa lex.C, aiheuttaa virheen tuntemattomana funktiona. Tiedostossa token.C oleva määrittely katsotaan jonkun muun funktion määrittelyksi.

Tyyppiturvallinen linkitysmekanismi tekee jossain määrin tyyppitarkastuksia välitiedoston funktiokutsuille. Tyyppiturvallinen linkitys on tarpeellinen tuki myös ylikuormitetuille funktioille. Käsittelemme tyyppiturvallista linkitystä lisää ylikuormitettujen funktioiden esittelyn yhteydessä luvussa 9.

Saman olion tai funktion esittelyiden muunkaltaisia eroavaisuuksia eri tiedostoissa ei ehkä siepata käännöksen tai linkityksen aikana. Koska kääntäjä käsittelee yhden tiedoston kerrallaan, ei se tavallisesti havaitse tyyppivirheitä tiedostojen välillä. Nämä tyyppivirheet voivat olla vakavien ohjelmavirheiden lähteitä. Esimerkiksi välitiedoston virheellisiä olioiden esittelyitä tai funktioiden paluutyyppejä ei havaita. Seuraavankaltaiset virheet paljastavat itsensä vain suorituksenaikaisissa poikkeuksissa tai ohjelman virheellisessä tulostuksessa:

```
// tiedostossa token.C
unsigned char lastTok = 0;
unsigned char peekTok() { /* ... */ }

// tiedostossa lex.C
extern char lastTok;    // yhden olion historia
extern char peekTok();  // yhden olion haku
```

Kurinalainen otsikkotiedostojen käyttö on peruseste tämänkaltaisten välitiedoston esittelyiden täsmäämättömyysvirheiden syntymiselle. Tämä on seuraavan alikohdan aiheena.

8.2.3 Muutama sana otsikkotiedostoista

Otsikkotiedosto toimii keskitettynä paikkana kaikille extern-olioiden esittelyille, funktioiden esittelyille ja välittömien funktioiden määrittelyille; tätä sanotaan esittelyiden paikallistamiseksi eli *lokalisoinniksi*. Tiedostot, joiden pitää käyttää tai määritellä olio tai funktio, ottavat mukaan (*include*) otsikkotiedoston (tai tiedostoja).

Otsikkotiedostot toimivat vartijoina kahdessa eri mielessä. Ensiksi taataan, että kaikki tiedostot sisältävät samanlaisen esittelyn globaalista oliosta tai funktiosta. Toiseksi, jos esittely vaatii ylläpitoa, tarvitaan vain yksi muutos otsikkotiedostoon. Esittelyn päivityksen unohtami-

nen tietystä tiedostosta ei ole enää mahdollinen. addToken()-esimerkissä on otsikkotiedosto token.h kuten seuraavassa:

```
// ----- token.h -----
typedef unsigned char uchar;
const uchar INLINE = 128;
// ...
const uchar LT = ...;
const uchar GT = ...;

extern uchar lastTok;
extern int addToken( uchar );
inline bool is_relational( uchar tok )
    { return (tok >= LT && tok <= GT); }

// ----- lex.C -----
#include "token.h"
// ...

// ----- token.C -----
#include "token.h"
// ...
```

Otsikkotiedostoja suunniteltaessa tulisi olla huolellinen. Niiden sisältämien esittelyiden tulisi kuulua loogisesti yhteen. Otsikkotiedoston kääntämiseen kuluu aikaa. Jos se on liian suuri tai sisältää liian monta yhteensopimatonta elementtiä, voivat ohjelmoijat olla haluttomia kustantamaan käännökseen menevää aikaa sen mukaanottamisella. Jotta otsikkotiedostojen kääntämiseen menisi vähemmän aikaa, on joissakin C++-toteutuksissa tuki *esikäännetyille* otsikkotiedostoille. Katsopa C++-toteutuksesi hakukäsikirjasta, kuinka luot esikäännettyjä otsikkotiedostoja tavallisista C++-otsikkotiedostoista. Jos sovelluksessasi on isoja otsikkotiedostoja, esikäännettyjen otsikkotiedostojen käyttö tavallisten sijasta voi vähentää merkittävästi sovelluksesi käännösaikaa.

Toinen näkökohta on, että otsikkotiedoston ei tulisi koskaan sisältää muun kuin välittömän funktion tai olion määrittelyä. Esimerkiksi jokainen seuraavista edustaa sellaista määrittelyä eikä niiden tulisi olla otsikkotiedostossa:

```
extern int ival = 10;
double fica_rate;
extern void dummy() {}
```

Vaikka ival on esitelty määreellä extern, sen eksplisiittinen alustaminen tekee siitä todellisuudessa määrittelyn. Samalla tavalla, vaikka dummy() on eksplisiittisesti esitelty määreellä extern, tarkoittaa tyhjä aaltosulkupari tuon funktion määrittelyä. Vaikka muuttujaa fica_rate ei ole eksplisiittisesti alustettu, myös sitä pidetään todellisuudessa määrittelynä C++:ssa, koska extern-avainsana puuttuu. Yhdenkin näistä määrittelyistä mukaan ottaminen saman ohjelman kahteen tai useampaan tiedostoon saa aikaan linkitysvirheen, jossa valitetaan useasta määrittelystä.

Edellä esitetyssä token.h-otsikkotiedostossa sekä INLINE-vakio että välitön is_relational()-funktio näyttävät rikkovan tätä sääntöä. Ne eivät kuitenkaan sitä tee. Vaikka ne ovat määrittelyitä,

symbolisten vakioiden ja välittömien funktioiden määrittelyt ovat erityislaatuisia. Symbolisia vakioita ja välittömiä funktioita voidaan määritellä useita kertoja.

Aina, kun on mahdollista, symbolisen vakion arvo korvaa sen nimen esiintymän ohjelman käännöksen aikana. Tätä korvaamisprosessia kutsutaan *vakioiden kokoamiseksi*. Kääntäjä korvaa esimerkiksi INLINE-nimen arvolla 128 aina, kun symbolista vakiota INLINE on käytetty tiedostossa. Jotta kääntäjä kykenisi korvaamaan vakion nimen arvollaan, vakion määrittelyn (alustajansa arvo) pitää olla näkyvissä aina siellä, missä vakiota käytetään. Tästä syystä symbolinen vakio voidaan määritellä eri tiedostoissa samassa ohjelmassa. Siten ihannetapauksessa, vaikka alustettu vakio voidaan ottaa mukaan moniin eri tiedostoihin, vakion kokoaminen tekisi sen tarpeettomaksi, jos edes yksikin määrittely esiintyisi suorituskelpoisessa ohjelmassa.

Kuitenkin joissakin tapauksissa symbolisen vakion kokoaminen ei ole mahdollista. Sellaisissa tapauksissa on parempi siirtää vakion alustus yhteen ohjelmatekstitiedostoon. Se voidaan tehdä esittelemällä eksplisiittisesti vakio extern. Esimerkiksi:

```
// ----- otsikkotiedosto -----
const int buf_chunk = 1024;
extern char *const bufp;

// ----- ohjelman tekstitiedosto -----
char *const bufp = new char[buf_chunk];
```

Vaikka bufp on esitelty const-määreellä, ei sen arvoa voida laskea käännöksen aikana (sen alustaja on new-lauseke, joka vaatii kirjastofunktion käynnistämisen). Jos se olisi alustettu otsikkotiedostossa, olisi bufp määritelty jokaisessa tiedostossa, joka olisi ottanut sen määrittelyn mukaan. Tämä ei vain tuhlaisi muistia, vaan olisi myös todennäköisesti vastoin ohjelmoijan aikomuksia.

Symbolinen vakio on mikä tahansa olio, joka on const-tyyppinen. Huomaatko, miksi seuraava esittely otsikkotiedoston sisälle sijoitettuna saa aikaan linkitysvirheen, kun se otetaan mukaan ohjelmaan kahteen eri tiedostoon?

```
// hups: ei tulisi olla otsikkotiedostossa
const char* msg = "?? oops: error: ";
```

Ongelmana on, että msg ei ole vakio; sen sijaan se on osoitinmuuttuja, joka viittaa vakioarvoon. Vakio-osoittimen kelvollinen esittely näyttää tältä (katso luvusta 3 osoitinten esittelyiden täydellinen käsittely):

```
const char *const msg = "?? oops: error: ";
```

Tämä vakio-osoittimen määrittely voi esiintyä useissa tiedostoissa.

Samanlainen tilanne kuin symbolisilla vakioilla voi tapahtua myös välittömillä funktioilla. Jotta kääntäjä kykenisi laajentamaan funktion rungon “välittömästi” kohtaan, jossa se käynnistetään, sen pitää nähdä välittömän funktion määrittely. (Välittömät funktiot on esitelty kohdassa 7.6.) Tästä syystä välitön funktio, jota tarvitaan useammassa kuin yhdessä tiedostossa, pitää määritellä otsikkotiedostoon. Välittömäksi määrittäminen on kuitenkin vain vinkki, että

funktion pitäisi olla välitön. Se, tekeekö kääntäjä funktiosta välittömän — yleensä tai joissakin kutsuissa ohjelman aikana — vaihtelee toteutuksittain. Ellei kääntäjä pysty tekemään funktiosta välitöntä sen käynnistyskohdassa, se generoi funktiolle määrittelyn suorituskelpoiseen tiedostoon. Jos saman funktion määrittely generoidaan useampaan kuin yhteen tiedostoon, siitä saattaa olla seurauksena tarpeettoman suuri suorituskelpoinen tiedosto.

Useimmat kääntäjät antavat varoitusilmoituksen, jos jokin seuraavista tapauksista on voimassa (yleensä tämä vaatii varoitusilmoitusten käyttöönoton kääntäjään).

1. Funktion määrittely välittömäksi on luonnostaan mahdotonta. Kääntäjä voi esimerkiksi valittaa, että funktio on liian monimutkainen välittömäksi. Tässä tapauksessa kirjoita funktio uudelleen, jos se on mahdollista; muussa tapauksessa poista välittömäksi määrittäminen ja sijoita funktion määrittely ohjelmatekstitiedostoon.
2. Tiettyä funktion kutsua ei voida tehdä välittömäksi. Esimerkiksi alkuperäisessä C++-toteutuksessa AT&T (cfront) toisen välittömän funktion kutsua samassa lausekkeessa ei tehdä välittömäksi. Tässä tapauksessa lauseke voidaan kirjoittaa uudelleen niin, että näitä kahta välitöntä funktiota kutsutaan erikseen kahdessa eri lausekkeessa.

Ennen kuin funktio esitellään välittömäksi, sen suorituksenaikainen käyttäytyminen pitää analysoida. Varmistuminen siitä, että funktio todella tehdään välittömäksi, on välttämätön osa ohjelmien kirjoittamista. Suosittelemme, että funktiota, jota ei luonnostaan voida tehdä välittömäksi, ei esiteltäisi sellaiseksi eikä sijoitettaisi otsikkotiedostoon.

Harjoitus 8.3

Yksilöi, mitkä seuraavista ovat esittelyitä ja mitkä määrittelyitä ja selitä, miksi.

```
(a) extern int ix = 1024;
(b) int iy;
(c) extern void reset( void *p ) { /* ... */ }
(d) extern const int *pi;
(e) void print( const matrix & );
```

Harjoitus 8.4

Mitkä seuraavista esittelyistä ja määrittelyistä sijoittaisit otsikkotiedostoon? Ohjelmatekstitiedostoon? Perustele, miksi.

```
(a) int var;
(b) inline bool is_equal( const SmallInt &, const SmallInt & ) { }
(c) void putValues( int *arr, int size );
(d) const double pi = 3.1416;
(e) extern int total = 255;
```

8.3 Paikalliset oliot

Muuttujan esittely paikallisella viittausalueella saa aikaan *paikallisen olion*. On olemassa kolmenlaisia paikallisia olioita: *automaattisia olioita, rekisteriolioita* ja *paikallisia staattisia olioita*. Se, mikä erottaa nämä oliot toisistaan, on muistitilan elinaika ja ominaisuudet, joissa oliot sijaitsevat. Muistialue, jossa automaattinen olio sijaitsee, kestää siitä hetkestä, kun funktio käynnistetään ja olio esitellään aina siihen saakka, kunnes funktio päättyy. Rekisteriolio on automaattinen olio, joka tukee olion arvon nopeaa lukemista ja tallentamista. Paikallinen staattinen olio sijaitsee muistissa, joka kestää ohjelman koko suorituksen ajan. Tässä kohdassa tutkimme näiden kolmen eri paikallisen olion ominaisuuksia.

8.3.1 Automaattiset oliot

Automaattisen olion muistitila varataan sillä hetkellä, kun käynnistetään funktio, jossa se on määritelty. Automaattisen olion muistitila varataan ohjelman suorituksenaikaisesta pinosta ja se on osa funktion aktivointitietuetta. Automaattisten olioiden sanotaan omaavan *automaattisen muistitilan keston* eli *automaattisen ulottuvuuden*. Alustamaton automaattinen olio sisältää satunnaisen bittikuvion, joka on jäänyt muistitilan edelliseltä käyttökerralta. Sen arvoa kutsutaan *määrittämättömäksi*.

Funktion päättymisen yhteydessä aktivointitietue vedetään pois suorituksenaikaisesta pinosta. Itse asiassa automaattisten olioiden varaama muisti vapautetaan. Olion elinaika päättyy funktion päätyttyä. Sen sisältämä mikä tahansa arvo hävitetään.

Koska automaattisiin olioihin liittyvä muisti vapautetaan funktion päätyttyä, automaattisen olion osoitetta tulisi käyttää varoen. Automaattisen olion osoitetta ei tulisi koskaan palauttaa funktion arvona, koska funktion päätyttyä osoite viittaa kelvottomaan muistialueeseen. Esimerkiksi:

```
#include "Matrix.h"

Matrix* trouble( Matrix *pm )
{
    Matrix res;

    // tee jotain käyttäen pm:ää
    // sijoita tulos res-olioon

    return &res; // paha!
}

int main()
{
    Matrix m1;
    // ...
```

```
    Matrix *mainResult = trouble( &m1 );
    // ...
}
```

mainResult asetetaan osoittamaan automaattiseen Matrix-olioon res. Valitettavasti res:in muistialue vapautetaan trouble()-funktion päätyttyä. Palattaessa main()-funktioon mainResult osoittaa teknisesti varaamattomaan muistiin. (Tässä esimerkissä osoite voi silti säilyä kelvollisena, koska emme ole vielä käynnistäneet jotain toista funktiota, joka käyttäisi kokonaan tai osittain aktivointitietuetta, jolla tuettiin trouble()-funktiota. Sellaisen virheen tekeminen on vaikea havaita.) mainResult:in käyttö myöhemmin main()-funktiossa voi johtaa yllättäviin tuloksiin.

Kuitenkin main()-funktion automaattisen olion m1-osoittimen välittäminen funktiolle trouble() on aina turvallista. On taattu, että main()-funktion muistitila säilyy pinossa trouble()-funktion kutsun ajan ja m1:n muistitila säilyy käytettävissä trouble()-funktion kutsun ajan.

Kun automaattisen olion osoite on tallennettu osoittimeen, jonka elinaika on pitempi kuin automaattisen olion, sanotaan sen olevan *roikkuva osoitin.* Tämä on vakava ohjelmoijan virhe, koska osoitetun muistin sisältö on arvaamaton. Jos tuon osoitteen bitit ovat jollain tavalla asiallisessa järjestyksessä (niin, että ohjelma ei generoi esimerkiksi segmenttivirhettä), ohjelma voi mennä loppuun saakka, mutta antaa virheellisiä tuloksia.

8.3.2 Automaattiset oliot rekisterissä

Automaattiset oliot, joita käytetään paljon funktiossa, voidaan esitellä avainsanalla register. Jos se on mahdollista, kääntäjä lataa olion koneen rekisteriin. Ellei se ole mahdollista, olio säilyy muistissa. Taulukoiden indeksit ja silmukan sisällä esiintyvät osoittimet ovat hyviä kandidaatteja rekisteriolioille.

```
for ( register int ix = 0; ix < sz; ++ix ) // ...
for ( register int *p = array ; p < arraySize; ++p ) // ...
```

Parametri voidaan myös esitellä rekisterimuuttujaksi:

```
bool find( register int *pm, int val ) {
    while ( *pm )
        if ( *pm++ == val ) return true;
    return false;
}
```

Rekisterimuuttujat voivat kasvattaa funktion nopeutta, jos muuttujia käytetään erittäin paljon.

register-avainsanan käyttö on vain vinkki kääntäjälle. Jotkut kääntäjät voivat olla huomioimatta tätä vinkkiä ja käyttää rekisterin varausalgoritmeja keksiäkseen parhaat kandidaatit sijoitettaviksi käytettävissä oleviin koneen rekistereihin. Koska kääntäjä on tietoinen koneen arkkitehtuurista, jossa ohjelmaa ajetaan, se kykenee usein paljon tietoisempiin päätöksiin valitessaan sisältöjä koneen rekistereihin.

8.3.3 Staattiset paikalliset oliot

On mahdollista esitellä funktion määrittelyssä tai funktion yhdistetyssä lauseessa paikallinen olio, joka säilyy koko ohjelman keston ajan. Kun paikallisen olion arvon pitää säilyä funktion käynnistyksestä toiseen, ei tavallista automaattista oliota voi käyttää. Automaattisen olion arvo häviää joka kerta, kun funktio päättyy.

Tässä tapauksessa ratkaisu on esitellä paikallinen olio staattisena avainsanalla static. *Staattisella paikallisella oliolla* on *staattinen muistitilan kesto* eli *staattinen ulottuvuus*. Vaikka sen arvo säilyy funktion käynnistyksestä toiseen, sen nimen näkyvyys säilyy rajoitettuna paikallisella viittausalueella. Staattinen olio alustetaan silloin, kun ohjelman suoritus ensimmäisen kerran menee olion esittelyn läpi. Tässä on esimerkkinä versio gcd()-funktiosta, joka seuraa rekursionsa syvyyttä käyttämällä staattista paikallista oliota:

```
#include <iostream>

int traceGcd( int v1, int v2 )
{
    static int depth = 1;
    cout << "depth #" << depth++ << endl;

    if ( v2 == 0 ) {
        depth = 1;
        return v1;
    }
    return traceGcd( v2, v1%v2 );
}
```

Arvo, joka liittyy staattiseen paikallisen olioon depth, säilyy traceGcd()-funktion käynnistyksestä toiseen. Alustus tehdään vain kerran: kun funktiota traceGcd() kutsutaan ensimmäisen kerran. Seuraavassa pienessä ohjelmassa kokeillaan traceGcd()-funktiota:

```
#include <iostream>
extern int traceGcd(int, int);

int main() {
    int rslt = traceGcd( 15, 123 );
    cout << "gcd of (15,123): " << rslt << endl;
    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavat tulokset:

```
depth #1
depth #2
depth #3
depth #4
gcd of (15,123): 3
```

Ohjelma alustaa alustamattoman, paikallisen staattisen olion arvolla 0. Vastakohtana ovat automaattiset oliot, joilla on satunnaiset arvot, ellei niitä ole eksplisiittisesti alustettu. Seuraava ohjelma kuvaa automaattisen ja staattisen paikallisen olion oletusalustusta ja sitä, mikä vaara piilee, ellei automaattisia olioita alusteta.

```
#include <iostream>

const int iterations = 2;
void func() {
    int value1, value2; // alustamatta
    static int depth;   // implisiittisesti alustettu arvolla 0

    if ( depth < iterations )
        { ++depth; func(); }
    else depth = 0;

    cout << "\nvalue1:\t" << value1;
    cout << "\tvalue2:\t" << value2;
    cout << "\tsum:\t" << value1 + value2;
}

int main() {
    for ( int ix = 0; ix < iterations; ++ix ) func();
    return 0;
}
```

Suorituksen tulostus on seuraava:

```
value1: 0         value2:  74924      sum:  74924
value1: 0         value2:  68748      sum:  68748
value1: 0         value2:  68756      sum:  68756
value1: 148620    value2:  2350       sum:  150970
value1: 2147479844  value2:  671088640   sum:  -1476398812
value1: 0         value2:  68756      sum:  68756
```

Huomaa, että value1 ja value2 ovat alustamattomia automaattisia olioita. Kuten tulostus näyttää, niiden alkuarvot ovat täysin satunnaisia ja niiden yhteenlaskun tulos on arvaamaton. Kuitenkin, vaikka depth on alustamaton, taataan, että sen arvo on 0, mikä varmistaa, että func() käynnistää itsensä rekursiivisesti vain kahdesti.

8.4 Dynaamisesti varatut oliot

Globaalien ja paikallisten olioiden elinaika on tarkasti määritelty. Ohjelmoija ei voi muuttaa niiden elinaikaa millään tavoin. On kuitenkin joskus tarpeellista luoda olioita, joiden elinaikaa ohjelmoija voi kontrolloida, ja joiden varaaminen ja vapauttaminen voi tapahtua tai välttyä tapahtumasta, mikä riippuu ohjelman suorituksen tuloksista. Joku voi esimerkiksi haluta varata merkkijonon virheilmoituksen tekstiä varten vain, jos virhe todella tapahtuu ohjelman suorituksen aikana. Jos ohjelma voi generoida enemmän kuin yhden virheilmoituksen, varattavan merkkijonon koko voi vaihdella tapahtuneen virhetekstin mukaan. Ei tiedetä etukäteen varattavan merkkijonon kokoa, koska se riippuu sen virheen laadusta, joka ohjelman suorituksen aikana voi tapahtua.

Kolmas oliotyyppi mahdollistaa, että ohjelmoija voi kontrolloida täydellisesti, milloin sen varaaminen ja vapauttaminen tapahtuu. Sellaista oliota kutsutaan *dynaamisesti varatuksi olioksi*. Dynaamisesti varattu olio varataan vapaasta muistista, jota kutsutaan ohjelman *vapaavarastoksi*. Ohjelmoija luo dynaamisesti varatun olion new-*lausekkeella* ja päättää sellaisen olion elinajan delete-*lausekkeella*. Dynaamisesti varattu olio on joko yksittäinen olio tai oliotaulukko. Vapaavarastosta varatun taulukon koko voidaan laskea suorituksen aikana.

Tässä kohdassa katsomme dynaamisesti varatuista olioista kolmea eri new-lausekkeen muotoa: yksi tukee yksittäisten olioiden dynaamista varaamista, toinen tukee taulukoiden dynaamista varaamista ja kolmatta muotoa sanotaan new-lausekkeen asemoinniksi. Kun vapaavarasto hupenee loppuun, new-lauseke heittää poikkeuksen; poikkeuksia käsitellään lisää luvussa 11. Luvussa 15 käsittelemme tarkemmin new- ja delete-lausekkeiden käyttöä luokkatyyppien yhteydessä.

8.4.1 Dynaaminen muistinvaraus ja yksittäisten olioiden vapauttaminen

New-lauseke muodostuu avainsanasta new ja tyyppimääreestä. Tämä tyyppimääre voi viitata sisäiseen tyyppiin tai luokkatyyppiin. Esimerkiksi

```
new int;
```

varaa yhden int-tyyppisen olion vapaavarastosta. Samalla tavalla

```
new iStack;
```

varaa yhden iStack-luokan olion.

Sinänsä new-lauseke ei ole kovin hyödyllinen. Kuinka voimme todella käyttää varattua oliota? Vapaavaraston muistista on sellainen piirre, että sieltä varatuilla olioilla ei ole nimeä. New-lauseke ei palauta varsinaista varattua oliota, vaan sen sijaan se palauttaa osoittimen tuohon olioon. Kaikki olion käsittely tapahtuu epäsuorasti osoittimien kautta. Esimerkiksi:

```
int *pi = new int;
```

New-lauseke luo yhden int-tyyppisen olion, johon pi viittaa.

Muistin varaamista vapaavarastosta suorituksen aikana, kuten edellisissä new-lausekkeissa, sanotaan *dynaamiseksi muistinvaraukseksi*. Sanomme, että pi:n osoittama muisti on varattu dynaamisesti.

Toinen vapaavaraston piirre on, että varattu muisti on alustamatonta. Vapaavaraston muisti sisältää satunnaisia bittikuvioita, jotka ovat jääneet ohjelmaamme tuon muistipaikan edellisestä käytöstä. Testi

```
if ( *pi == 0 )
```

epäonnistuu todennäköisesti aina, koska pi:n osoittama olio sisältää satunnaisia bittejä. Tästä syystä suosittelemme, että new-lausekkeella luodut oliot alustetaan. Ohjelmoija voi alustaa edellisen esimerkin mukaisen int-tyyppisen olion seuraavasti:

```
int *pi = new int( 0 );
```

Sulkujen sisällä oleva vakio on alkuarvo, jolla new-lausekkeella luotu olio alustetaan. pi viittaa siten int-tyyppiseen olioon, jonka arvo on 0. Sulkujen sisällä olevaa lauseketta kutsutaan *alustajaksi*. Tämän alustajan ei tarvitse olla vakioarvo. Mikä tahansa lauseke, jonka konversio johtaa int-tyypiksi, on kelvollinen alustaja.

New-lausekkeen operaatioiden järjestys on seuraava: olio varataan vapaavarastosta ja olio alustetaan suluissa olevalla arvolla. Varatakseen olion vapaavarastosta, new-lauseke kutsuu kirjasto-operaattoria new(). Edellinen new-lauseke on karkeasti yhtäkuin seuraava koodikatkelma

```
int ival = 0;    // luo int-tyyppisen olion ja alustaa sen arvolla 0
int *pi = &ival; // nyt osoitin osoittaa olioon
```

paitsi tietysti, että pi:n osoittaman olion on varannut new()-kirjasto-operaattori ja se sijaitsee ohjelman vapaavarastossa. Samalla tavalla

```
iStack *ps = new iStack( 512 );
```

luo iStack-tyyppisen olion, jossa on 512 elementtiä. Kun kyseessä on luokkaolio, suluissa oleva arvo tai arvot välitetään luokan muodostajalle, joka käynnistetään onnistuneen olion varaamisen jälkeen. (Luokkaolioiden dynaamista varaamista käsitellään tarkemmin kohdassa 15.8. Tämän kohdan loppuun saakka keskitytään sisäisiin tyyppeihin.)

Tähän mennessä on esitetty yksi ongelma, joka liittyy new-lausekkeisiin. Vapaavarasto valitettavasti edustaa äärellistä resurssia: jossakin vaiheessa ohjelman suorituksen aikana voimme käytännössä havaita vapaavaraston huvenneen loppuun, mikä johtaa new-lausekkeen epäonnistumiseen. Jos new-lausekkeen kutsuma new() ei saa pyydettyä muistia, se heittää yleensä poikkeuksen nimeltään bad_alloc. (Poikkeuksia käsitellään yleisesti luvussa 11.)

Olion elinaika, johon pi viittaa, loppuu, kun sen sijaintipaikan muisti vapautetaan. Muisti vapautetaan, kun pi on delete-lausekkeen operandina. Esimerkiksi

```
delete pi;
```

vapauttaa muistin, johon pi viittaa, ja lopettaa int-tyyppisen olion elinkaaren. Ohjelmoija kontrolloi sitä, milloin olion elinaika päättyy päättämällä delete-lausekkeen sijoituspaikasta ohjelmassa. Delete-lauseke kutsuu delete()-kirjasto-operaattoria, joka vapauttaa muistin vapaavarastoon. Koska vapaavarasto on äärellinen resurssi, on tärkeää palauttaa varattu muisti vapaavarastoon niin pian kuin emme enää tarvitse sitä.

Kun katsomme edellistä delete-lauseketta, joku voi kysyä, mitä tapahtuu, jos jostain syystä pi on asetettu osoittamaan arvoon 0? Eikö koodin tulisi näyttää tältä?

```
// onko tämä tarpeen?
if ( pi != 0 )
    delete pi;
```

Ei. Kieli takaa, että delete-lauseke ei kutsu delete()-operaattoria, jos osoitinoperandi on asetettu arvoon 0. Nolla-arvon tarkistaminen tässä on tarpeetonta. (Itse asiassa useimmissa toteutuksissa on niin, että jos lisäät osoittimelle eksplisiittisen testin, testi tehdään käytännössä kahdesti.)

On tärkeää tässä vaiheessa käsitellä elinajan eroa pi:n ja sen olion välillä, johon pi viittaa. Itse pi-osoitin on globaali olio, joka on esitelty globaalilla viittausalueella. Tuloksena on, että pi:n muisti varataan ennen kuin ohjelma käynnistyy, ja se kestää ohjelman loppuun saakka. Tämä eroaa pi:n osoittaman olion elinajasta, joka luodaan, kun new-lauseke kohdataan ohjelman suorituksen aikana. Muisti, johon pi viittaa, varataan dynaamisesti ja tähän sijoittuva olio on dynaamisesti varattu. Tästä syystä pi on globaali osoitin, joka viittaa dynaamisesti varattuun int-tyyppiseen olioon. Kun ohjelman suorituksen aikana kohdataan delete-lauseke, vapautetaan muisti, johon pi viittaa. Kuitenkaan delete-lauseke ei vaikuta pi:lle varattuun muistiin eikä sen sisältöön. Delete-lausekkeen jälkeen pi:tä kutsutaan roikkuvaksi osoittimeksi, joka tarkoittaa, että osoitin osoittaa kelpaamattomaan muistialueeseen. Roikkuva osoitin voi olla sellaisten ohjelmavirheiden lähde, joita on vaikea havaita, ja siksi onkin hyvä ajatus asettaa osoitin arvoon 0 sen osoittaman olion poistamisen jälkeen osoitukseksi siitä, että se ei osoita olioon.

Delete-lauseketta pitää käyttää vain osoittimeen, joka viittaa muistiin, joka on varattu vapaavarastosta new-lausekkeella. Delete-lausekkeen käyttö osoittimeen, joka viittaa vapaavarastosta varattuun muistiin, johtaa todennäköisesti ohjelman tuntemattomaan käyttäytymiseen suorituksensa aikana. Kuten näimme aikaisemmin, ei ole rangaistavaa käyttää delete-lauseketta nolla-arvoiselle osoittimelle — tarkoittaa osoitinta, joka ei viittaa mihinkään olioon. Seuraavassa on esimerkkejä turvallisista ja turvattomista delete-lausekkeista:

```
void f() {
    int i;
    string str = "dwarves";
    int *pi = &i;
    short *ps = 0;
```

```
    double *pd = new double(33);

    delete str; // huono: merkkijono "dwarves" ei ole dynaaminen olio
    delete pi;  // huono: pi viittaa paikalliseen olioon i
    delete ps;  // turvallinen
    delete pd;  // turvallinen
}
```

Seuraavat kolme yleistä ohjelmointivirhettä liittyvät dynaamiseen muistinvaraukseen:

1. Ei käytetä delete-lauseketta ja siten estetään muistin palautus vapaavarastoon. Tätä kutsutaan *muistivuodoksi*.
2. Käytetään delete-lauseketta samaan muistipaikkaan kahdesti. Tämä tapahtuu usein silloin, kun kaksi osoitinta jossain vaiheessa päätyvät osoittamaan samaan dynaamisesti varattuun olioon. Se voi olla erityisen ikävä ongelma jäljitettäväksi. Se mitä tapahtuu, on, että olio tuhotaan jonkin osoittamansa osoittimen kautta. Olion muistitila palautetaan vapaavarastoon ja käytetään uudelleen seuraaville muille olioille. Sitten toinen osoitin, joka osoittaa vanhaa oliota, poistetaan, ja uusi olio on yhtäkkiä korruptoitunut.
3. Oliota luetaan tai siihen kirjoitetaan sen jälkeen, kun se on poistettu. Tämä tapahtuu usein, koska osoitinta, johon delete-lauseketta on käytetty, ei ole asetettu arvoon 0.

Tämänkaltaisia dynaamisesti varatun muistin käsittelyvirheitä on huomattavasti helpompi tehdä kuin jäljittää ja korjata. Jotta ohjelmoija hallitsisi paremmin dynaamisesti varattua muistia, on C++-kirjastossa helpotukseksi auto_ptr-luokkatyypin tuki. Tämä on seuraavan alikohdan aiheena. Sen jälkeen katsomme taulukoiden dynaamista varaamista ja vapauttamista käyttäen new- ja delete-lausekkeiden toista muotoa.

8.4.2 Auto_ptr ◆A

Auto_ptr on C++-vakiokirjaston luokkamalli, joka voi helpottaa ohjelmoijia automatisoimaan yksittäisten dynaamisesti new-operaattorilla varattujen olioiden hallintaa. (Valitettavasti ei ole olemassa vastaavaa tukea new-lausekkeen kautta varattujen taulukoiden hallinnalle. Auto_ptr:ää ei tulisi käyttää taulukoiden tallentamiseen. Jos teet niin, tulokset ovat tuntemattomia.)

Auto_ptr-olio alustetaan osoittamaan new-lausekkeella luotuun, dynaamisesti varattuun olioon. Kun auto_ptr-olion elinaika päättyy, dynaamisesti varattu olio vapautetaan automaattisesti. Tässä alikohdassa katsomme, kuinka liitämme auto_ptr-olion new-lausekkeella luotuun olioon.

Ennen kuin auto_ptr-luokkamallia voidaan käyttää, pitää ottaa mukaan seuraava otsikkotiedosto:

```
#include <memory>
```

Auto_ptr-oliolla on kolme yleistä muotoa:

```
auto_ptr< osoitettava_tyyppi > tunnus( osoitin_jonka_varannut_new );
auto_ptr< osoitettava_tyyppi > tunnus( samantyyppinen_auto_ptr );
auto_ptr< osoitettava_tyyppi > tunnus;
```

osoitettava_tyyppi edustaa new-lausekkeella luodun olion tyyppiä. Katsokaamme jokaista määrittelyä vuorollaan. Useimmissa tapauksissa haluamme suoraan alustaa auto_ptr-olion osoittamaan new-lausekkeen palauttamaan osoitteeseen. Voimme tehdä sen seuraavasti:

```
auto_ptr< int > pi( new int( 1024 ) );
```

pi alustetaan new-lausekkeella luodun olion osoitteella. Tämä olio alustetaan arvolla 1024. Voimme tarkistaa sen olion arvon, johon auto_ptr-olion viittaa, samalla tavalla kuin tekisimme sen tavallisella osoittimella:

```
if ( *pi != 1024 )
    // hups, joitain vinossa
else *pi *= 2;
```

New-lausekkeella luotu olio, johon pi viittaa, tuhoutuu automaattisesti, kun pi:n elinaika päättyy. Jos pi on paikallinen olio, sen viittaama olio tuhoutuu sen lohkon lopussa, jossa se on määritelty. Jos pi on globaali olio, sen viittaama olio tuhoutuu ohjelman lopussa.

Mitä, jos alustamme auto_ptr-olion osoittamaan luokkatyypin olioon kuten vakioon string-tyyppiin? Esimerkiksi:

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );
```

Oletetaan, että nyt haluamme käyttää merkkijono-operaatiota. Tavallisella string-osoittimella tekisimme seuraavaa:

```
string *pstr_type = new string( "Brontosaurus" );
if ( pstr_type->empty() )
    // hups, jotain vinossa
```

Kuinka käyttäisimme empty()-merkkijono-operaatiota auto_ptr-oliolla? Käyttäisimme täsmälleen samaa lähestymistapaa:

```
auto_ptr< string > pstr_auto( new string( "Brontosaurus" ) );
if ( pstr_auto->empty() )
    // hups, jotain vinossa
```

Päämotiivi auto_ptr-luokkamallin takana on tukea samanlaista syntaksia kuin mitä käytetään tavallisten osoitintyyppien kanssa, mutta lisäksi tarjota olion automaattinen tuhoamisen hallinta, johon auto_ptr-olio viittaa. Maalaisjärki voi johdattaa sinut uskomaan, että tämä lisäturvallisuus tulee suorituksenaikaisen tehokkuuden kustannuksella, mutta näin ei ole asianlaita. Koska nämä operaatiot ovat välittömiä (kääntäjä laajentaa ne kutsukohdilleen), auto_ptr-olion käyttö ei ole merkittävästi raskaampaa kuin osoittimen suora käyttö.

Mitä tapahtuu seuraavassa tapauksessa, jossa alustamme pstr_auto2:n pstr_auto:n arvolla, joka on auto_ptr-olio string-olioon?

```
// kuka on vastuussa string-olion tuhoamisesta?
auto_ptr< string > pstr_auto2( pstr_auto );
```

Oletetaan, että alustamme suoraan yhden string-osoittimen toisella kuten tässä

```
string *pstr_type2( pstr_type );
```

Molemmat osoittimet sisältävät string-olion osoitteen vapaavarastosta ja pitää olla huolellinen, että käytetään delete-lauseketta vain toiseen niistä. Toisaalta auto_ptr-luokkamalli tukee omistajuuden ilmaisua.

Kun määrittelemme pstr_auto:n, se havaitsee string-olion omistajuuden, jolla alustamme sen ja se tietää siitä olevansa vastuussa tuon string-olion tuhoamisesta. Tämä on vastuu, jonka omistajuus tuo auto_ptr-oliolle.

Kysymys kuuluu, mitä tapahtuu omistajuuden ehdoilla, kun pstr_auto2 on alustettu osoittamaan samaan olioon kuin pstr_auto? Emme halua molempien auto_ptr-olioiden omistavan samaa kohdeoliota — se aiheuttaa kaikki monituhoamisongelmat, jotka me alun perin halusimme estää auto_ptr-tyypin käytöllä.

Kun yksi auto_ptr-olio on alustettu tai siihen on sijoitettu toinen auto_ptr-olio, vasemmalla puolella oleva alustettu tai siihen sijoitettu auto_ptr-olio saa vapaavarastossa kohteena olevan olion omistajuuden. Oikeanpuoleinen auto_ptr-olio luopuu kaikesta vastuusta. Siten esimerkissämme pstr_auto2 on nyt se, joka tuhoaa string-olion, eikä pstr_auto. pstr_auto:a ei voi sen jälkeen käyttää string-olioon viittaamiseen.

Samalla tavalla tapahtuu sijoitusoperaattorin kanssa. Olkoot seuraavat kaksi auto_ptr-oliota

```
auto_ptr< int > p1( new int( 1024 ) );
auto_ptr< int > p2( new int( 2048 ) );
```

Sijoitusoperaattoria voidaan käyttää auto_ptr-olion kopiointiin toiseen seuraavasti:

```
p1 = p2;
```

Ennen sijoitusta tuhotaan olio, johon p1 viittaa. Sijoituksen jälkeen p1 omistaa int-tyyppisen olion, joka sisältää arvon 2048. p2:ta ei voi enää käyttää tähän olioon viittaamiseen.

Kolmannessa auto_ptr-määrittelyn muodossa luomme auto_ptr-olion, mutta emme alusta sitä osoittimella vapaavaraston olioon. Esimerkiksi:

```
// ei juuri nyt viittaa olioon
auto_ptr< int > p_auto_int;
```

Koska alustamaton p_auto_int ei viittaa olioon, sen sisäinen osoitinarvo on asetettu arvoon 0. Tämä tarkoittaa, että sen käyttäminen käänteisesti johtaisi ohjelman outoon käyttäytymiseen samalla tavoin, kuin jos käyttäisimme käänteisesti osoittimen 0-arvoa:

```
// hups: auto_ptr:n käyttö käänteisesti, joka ei osoita mihinkään olioon
if ( *p_auto_int != 1024 )
     *p_auto_int = 1024;
```

Tavallisia osoittimia voidaan testata 0-arvoa vastaan. Esimerkki:

```
int *pi = 0;
if ( pi != 0 ) ...;
```

Mutta kuinka testata, viittaako auto_ptr-olio taustalla olevaan olioon? get()-operaatio palauttaa kohdeosoittimen, jonka auto_ptr-olio sisältää. Joten päätelläksemme, viittaako auto_ptr-olio johonkin olioon, voimme ohjelmoida seuraavasti:

```
// uudistettu testi takaamaan, että p_auto_int viittaa olioon
if ( p_auto_int.get() != 0 &&
    *p_auto_int != 1024 )
          *p_auto_int = 1024;
```

Jos se ei viittaa mihinkään olioon, kuinka voimme saada sen viittaamaan sellaiseen — eli kuinka voimme asettaa auto_ptr-olion kohdeosoittimen? Voimme tehdä sen reset()-operaatiolla. Esimerkiksi:

```
else
     // ok, asetetaan p_auto_int:in kohdeosoitin
     p_auto_int.reset( new int( 1024 ) );
```

Auto_ptr-olio ei tue ilmaisua, että siihen on määrittelynsä jälkeen suoraan sijoitettu new-lausekkeella luodun olion osoite. Emme voi kirjoittaa

```
void example()
{
     // oletusarvoisesti alustettu arvolla 0
     auto_ptr< int > pi;
     {
          // ei tueta
          pi = new int( 5 );
     }
}
```

Jotta auto_ptr-olio voidaan alustaa uudelleen, pitää käyttää reset()-funktiota. reset()-funktiolle voidaan välittää osoitin tai 0, jos haluamme asettaa auto_ptr-olion pois käytöstä. Jos nykyinen auto_ptr viittaa olioon ja sillä on tuon olion omistajuus, silloin sen osoittama olio tuhotaan ennen kohdeosoittimen uudelleenasetusta. Esimerkiksi:

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );

// tuhoaa Brontosaurus-olion ennen uudelleenasetusta
pstr_auto.reset( new string( "Long-neck" ) );
```

Tässä tapauksessa on tehokkaampaa sijoittaa olemassaolevaan string-olioon ja käyttää assign()-operaatiota kuin tuhota se ja varata toinen uudelleen:

```
// tehokkaampi muoto uudelleenasetukselle tässä tapauksessa

// käytä string-olion assign()-operaatiota uuden arvon asettamiseksi
pstr_auto->assign( "Long-neck" );
```

Eräs ohjelmoinnin kovista asioista on, että yksinkertaisesti oikeiden tulosten saaminen ei aina riitä. Joskus meidän ei ainoastaan tarvitse saada oikeita tuloksia, vaan myös näytölle hyväksyttävällä suorituskyvyllä. Pieni asia kuten string-olion vapauttaminen ja uudelleenvaraaminen, kun assign()-funktion kutsu on riittävä, on esimerkki pienestä hetken yksityiskohdasta, kun taas joissakin tapauksissa se voi johtaa kasaantuvasti suorituskyvyn pullonkaulaksi. Nämä eivät ole yksityiskohtia, joiden pitäisi kiusata, kun yritetään miettiä ohjelman yleisratkaisua, vaan nämä tulevat kokeneille ohjelmoijille osaksi ohjelman sisäistä tarkastuslistaa.

Auto_ptr-luokkamalli tuo paljon turvallisuutta ja kätevyyttä dynaamisesti varatun muistin käsittelyyn. Silti meidän kuitenkin pitää olla huolellisia tai joudumme ikävyyksiin. Mitä voimme tehdä väärin?

1. Meidän pitää olla huolellisia, että emme alusta auto_ptr-oliota tai sijoita siihen osoitinta, jota ei ole varattu new-lausekkeella. Jos teemme niin, delete-lauseketta käytetään osoittimeen, jota ei ole dynaamisesti varattu, mikä johtaa ohjelman arvaamattomaan käyttäytymiseen.
2. Meidän pitää olla huolellisia, ettei kaksi auto_ptr-oliota omista samaa oliota vapaavarastosta. Eräs ilmeinen tapa tehdä tämä virhe on alustaa tai sijoittaa sama osoitin kahteen olioon. Vielä helpompi tapa tehdä tämä erehdys on käyttää get()-operaatiota. Esimerkiksi:

```
auto_ptr< string >
    pstr_auto( new string( "Brontosaurus" ) );
```

```
// hups: nyt molemmat osoittavat samaan olioon
// ja molemmat omistavat sen
auto_ptr< string > pstr_auto2( pstr_auto.get() );
```

release()-operaatiolla voimme alustaa tai sijoittaa auto_ptr-olion kohdeolion toiseen ilman näitä kahta omistajuutta. release() ei vain palauta kohdeolion osoitetta kuten get()-operaatio, vaan myös vapauttaa tuon olion omistajuuden. Seuraavassa on edellinen koodikatkelma kirjoitettuna uudelleen:

```
// ok: molemmat yhä osoittavat samaan olioon,
// mutta pstr_auto ei longer enää omista
auto_ptr< string >
    pstr_auto2( pstr_auto.release() );
```

8.4.3 Taulukoiden dynaaminen varaaminen ja vapauttaminen

New-lausekkeella voidaan varata myös taulukko vapaavarastosta. Siinä tapauksessa new-lausekkeen tyyppimääreen jälkeen pitää olla hakasuluissa taulukon ulottuvuuden koko. Ulottuvuus voi olla kuinka monimutkainen lauseke tahansa. New-lauseke palauttaa osoittimen taulukon ensimmäiseen elementtiin. Esimerkiksi:

```
// varaa yksi int-tyyppinen olio
// ja anna sille alkuarvoksi 1024
int *pi = new int( 1024 );

// varaa 1024 elementin taulukko,
// jonka elementit ovat alustamattomia
int *pia = new int[ 1024 ];

// varaa kaksiulotteinen taulukko, jossa on 4 x 1024 elementtiä
int (*pia2)[ 1024 ] = new int[ 4 ][ 1024 ];
```

pi osoittaa yksittäiseen int-tyyppiseen olioon, joka on alustettu arvolla 1024. pia osoittaa 1024-elementtisen taulukon ensimmäiseen elementtiin. pia2 osoittaa neljän 1024 elementin taulukon ensimmäiseen elementtiin — tarkoittaa, että pia2 osoittaa 1024 elementin taulukkoon.

Yleensä vapaavarastosta varatulle taulukolle ei voida antaa alkuarvoja. (Kohdassa 15.8 tulemme näkemään, kuinka luokkataulukoita varataan vapaavarastosta ja alustetaan luokan oletusmuodostajalla.) Ei ole mahdollista määrittää alustajaa edellä oleviin new-lausekkeisiin taulukoiden elementtien alustamiseksi. Vapaavarastosta luotu taulukko, joka on sisäistä tyyppiä, pitää alustaa for-silmukalla, jossa taulukon elementit alustetaan yksi toisensa jälkeen:

```
for ( int index = 0; index < 1024; ++index )
    pia[ index ] = 0;
```

Dynaamisesti varatun taulukon pääetu on, että sen ensimmäisen ulottuvuuden ei tarvitse olla vakioarvo; tarkoittaa, että ulottuvuuden ei tarvitse olla tiedossa käännöksen aikana kuten pitää olla taulukolla, joka on esitelty määrittelyssä paikallisella tai globaalilla viittausalueella. Tämä tarkoittaa, että voimme varata muistia sen verran kuin ohjelman sen hetkinen tarve vaatii.

Jos esimerkiksi tosielämän C++-ohjelmissa osoitin mahdollisesti viittaa useaan C-tyyliseen merkkijonoon ohjelman suorituksen aikana, C-tyylisille merkkijonoille käytetty muisti, johon osoitin osoittaa, on tyypillisesti varattu dynaamisesti ohjelman suorituksen aikana ja perustuu talletettavan merkkijonon pituuteen. Tämä tekniikka on huomattavasti tehokkaampi kuin kiinteäpituisen taulukon varaaminen näitä merkkijonoja varten, koska tuon taulukon pitää olla tarpeeksi suuri pitämään sisällään suurimman mahdollisen merkkijonon, vaikka suurin osa merkkijonoista ovat todennäköisesti huomattavasti pienempiä. Lisäksi ohjelmamme suoritus ei onnistu, jos yksikin merkkijono on suurempi kuin päättämämme kiinteä koko.

Tässä on esimerkki, kuinka new-lauseketta voidaan käyttää taulukon ensimmäisen ulottuvuuden määrittämiseen suorituksen aikaisena arvona. Oletetaan, että meillä on seuraavat C-tyyliset merkkijonot:

```
const char *noerr = "success";
// ...
const char *err189 = "Error: a function declaration must "
                     "specify a function return type!";
```

New-lausekkeella varattavan taulukon ulottuvuus voidaan määrittää arvolla, joka ratkaistaan suorituksen aikana kuten seuraavassa:

```
#include <cstring>

const char *errorTxt;

if (errorFound)
    errorTxt = err189;
else
    errorTxt = noerr;

int dimension = strlen( errorTxt ) + 1;
char *str1 = new char[ dimension ];

// kopioi teksti virhettä varten str1:een
strcpy( str1, errorTxt );
```

`dimension` voidaan korvata lausekkeella, joka ratkaistaan suorituksen aikana:

```
// tyypillinen todellisen ohjelman idiomi, mutta
// hämmentää joskus aloittelevia ohjelmoijia
char *str1 = new char[ strlen( errorTxt ) + 1 ];
```

Arvon 1 lisääminen `strlen()`-funktion paluuarvoon on välttämätöntä, jotta voidaan järjestää C-tyylisten merkkijonojen lopussa oleva null-merkki. Sen varaamisen unohtaminen on yleinen ohjelmavirhe ja sen jäljittäminen saa aikaan päänsärkyä, koska se tuo itsensä esille epäsuorasti muistin luku- tai kirjoituskorruptiona jossain ohjelman osassa. Miksi? Useimmat rutiinit, jotka käsittelevät C-tyylisiä merkkijonotaulukoita, käyvät taulukkoa läpi, kunnes kohtaavat lopussa olevan null-merkin. Null-merkin poisjääntі johtaa usein vakavaan ohjelmavirheeseen, koska

ohjelma saattaa lukea ja mahdollisesti kirjoittaa muistialueelle, jota sillä ei ole lupaa käsitellä. Tämänkaltaisten virheiden välttämiseksi suosittelemme C++-standardikirjaston string-luokan käyttöä.

Huomaa, että new-lausekkeella varatun taulukon vain ensimmäinen elementti voidaan määrittää lauseketta käyttäen suorituksen aikana. Muiden ulottuvuuksien pitää olla tunnettuja vakioita käännöksen aikana. Esimerkiksi:

```
int getDim();

// varaa kaksiulotteinen taulukko
int (*pia3)[ 1024 ] = new int[ getDim() ][ 1024 ]; //ok

// virhe: taulukon toinen ulottuvuus ei ole vakio
int **pia4 = new int[ 4 ][ getDim() ];
```

Delete-lausekkeella, jota käytetään taulukon muistin vapauttamiseen, on seuraava muoto:

```
delete[] str1;
```

Tyhjä hakasulkupari on välttämätön. Se ilmaisee kääntäjälle, että osoitin osoittaa elementtitaulukkoa vapaavarastossa eikä yhtä oliota. Koska str1:n tyyppi on osoitin char-tyyppiin, ei kääntäjä tajua näkemättä tyhjää hakasulkuparia, että tuhottava muisti on taulukko.

Mitä tapahtuu, jos erehdyksessä jätät pois tyhjän hakasulkuparin? Kääntäjä ei huomaa virhettä, mutta ei ole takuita siitä, että ohjelma toimii oikein (tämä pätee erityisesti luokkatyyppisiin taulukoihin, joilla on omat tuhoajat kuten kuvataan kohdassa 14.4).

Jotta välttäisit dynaamisesti varattujen taulukoiden muistinhallinnan ongelmat, suosittelemme yleisesti, että käytät vakiokirjaston vector- (vektori), list- (lista) tai string-tyyppisiä säiliöitä. Nämä tyypit hallitsevat muistinvarauksen automaattisesti. String-tyyppi esiteltiin kohdassa 3.4, vektori-tyyppi kohdassa 3.10. Säiliötyypit ovat esiteltynä tarkemmin luvussa 6.

8.4.4 Vakio-olioiden dynaaminen varaaminen ja vapauttaminen

Ohjelmoija voi haluta luoda olion vapaavarastoon, mutta estää ohjelmaa muuttamasta olion arvoa sen jälkeen, kun se on alustettu. Voit tehdä tämän luomalla olion vapaavarastoon const-tyyppisenä. Ohjelmoija, joka haluaa luoda const-olion vapaavarastoon, voi käyttää new-lauseketta seuraavasti:

```
const int *pci = new const int(1024);
```

const-oliolla, joka on luotu vapaavarastoon, on muutama erityisominaisuus. Ensiksikin, const-olio pitää alustaa. Jos sulkujen sisältä jätetään alustaja pois, siitä seuraa kääntäjän virheilmoitus (paitsi luokkatyyppisellä oliolla, jolla on muodostaja; siltä alustaja voidaan jättää pois). Toiseksi osoittimen, joka alustetaan new-lausekkeen paluuarvolla, pitää olla osoitin const-tyyppiin. Edellisessä esimerkissä pci on osoitin const int -tyyppiin. Osoitin viittaa const int -olioon, joka on varattu new-lausekkeella.

Mitä merkitsee oliolle olla varattuna vapaavarastosta const-tyyppisenä? Se merkitsee, että heti, kun olio on alustettu, olion arvoa ei voi muuttaa. Vaikka olion arvoa ei voi muuttaa, sen elinaika päättyy delete-lausekkeeseen. Esimerkiksi:

```
delete pci;
```

Vaikka delete-lausekkeen operandi on osoitin const int -tyyppiin, delete-lauseke on kelvollinen ja saa aikaan muistin vapautuksen, johon pci viittaa.

Ei ole mahdollista luoda const-tyyppistä sisäisten tyyppien elementtitaulukkoa vapaavarastosta siitä yksinkertaisesta syystä, että sitä ei voida alustaa, jos se on luotu new-lausekkeella. Kaikki oliot, jotka luodaan const-tyyppisinä vapaavarastoon, pitää alustaa. Ja koska const-taulukkoa ei voida alustaa (paitsi luokkataulukoita), yritys luoda const-tyyppinen taulukko sisäisillä tyypeillä new-lausekkeella johtaa käännösvirheeseen:

```
const int *pci = new const int[100]; // virhe
```

8.4.5 New-lausekkeiden asemointi

On olemassa kolmas new-lausekkeen muoto, jossa ohjelmoija voi pyytää, että luotavien olioiden muisti on etukäteen varattu. Tätä new-lausekkeen muotoa kutsutaan *new-lausekkeen asemoinniksi* (*placement new expression*). Ohjelmoija määrittää muistin osoitteen, johon olio tullaan luomaan new-lausekkeella. Tämän new-lausekkeen muoto on seuraava:

```
new (paikan_osoite) tyyppimäärite
```

paikan_osoitteen pitää olla osoitin. Jotta tätä new-lausekkeen muotoa voidaan käyttää, pitää ottaa mukaan otsikkotiedosto <new>. Tämän piirteen avulla ohjelmoija voi varata etukäteen suuria määriä muistia, joka myöhemmin tulee sisältämään tällä new-lausekkeen muodolla luotuja olioita. Esimerkiksi:

```
#include <iostream>
#include <new>
const int chunk = 16;
class Foo {
public:
    int val()  { return _val; }
    Foo() { _val = 0; }
private:
    int _val;
};

// varaa muisti etukäteen, mutta älä Foo-olioita
char *buf = new char[ sizeof(Foo) * chunk ];

int main() {
    // luo Foo-olio puskuriin (buf)
    Foo *pb = new (buf) Foo;

    // tarkista, että olio sijoitettiin puskuriin
```

```
    if ( pb.val() == 0 )
        cout << "new expression worked!" << endl;

    // pb:tä ei voi käyttää täällä
    delete[] buf;

    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se generoi seuraavan tulostuksen:

```
Operator new worked!
```

New-lausekkeen asemoinnille ei ole olemassa vastaavaa delete-lauseketta. Sellaista delete-lauseketta ei tarvita, koska new-lausekkeen asemointi ei varaa muistia. Edellisessä esimerkissä `pb`-osoittimen osoittama muisti ei ole se, joka pitää tuhota, vaan se muisti, jota `buf` osoittaa. Tämä muisti tuhoutuu ohjelman lopussa, kun merkkipuskuria ei enää tarvita. Koska `buf` viittaa merkkitaulukkoon, on delete-lausekkeella muoto

```
delete[] buf;
```

Kun merkkipuskuri on tuhottu, jokaisen sen sisältämän olion elinaika päättyy. Esimerkissämme `pb` ei enää viittaa kelvolliseen Foo-luokkatyyppiin.

Harjoitus 8.5

Selitä, miksi seuraavat new-lausekkeet ovat virheellisiä.

```
(a) const float *pf = new const float[100];
(b) double *pd = new double[10][getDim()];
(c) int (*pia2)[ 1024 ] = new int[ ][ 1024 ];
(d) const int *pci = new const int;
```

Harjoitus 8.6

Olkoot seuraavat new-lausekkeet. Kuinka tuhoaisit `pa`:n?

```
typedef int arr[10];
int *pa = new arr;
```

Harjoitus 8.7

Mitkä seuraavista delete-lausekkeista ovat mahdollisesti suorituksenaikaisia virheitä, vai onko yksikään? Miksi?

```
int globalObj;
char buf[1000];

void f() {
    int *pi = &globalObj;
    double *pd = 0;
    float *pf = new float(0);
    int *pa = new(buf)int[20];

    delete pi;   // (a)
    delete pd;   // (b)
    delete pf;   // (c)
    delete[] pa; // (d)
}
```

Harjoitus 8.8

Mitkä seuraavista auto_ptr-esittelyistä ovat kelvottomia tai todennäköisesti johtavat jatkossa ohjelmavirheisiin? Selitä, mikä ongelma kussakin on.

```
int ix = 1024;
int *pi = & ix;
int *pi2 = new int( 2048 );

(a) auto_ptr<int> p0(ix);            (b) auto_ptr<int> p1(pi);
(c) auto_ptr<int> p2(pi2);           (d) auto_ptr<int> p3(&ix);
(e) auto_ptr<int> p4(new int(2048)); (f) auto_ptr<int> p5(p2.get());
(g) auto_ptr<int> p6(p2.release());  (h) auto_ptr<int> p7(p2);
```

Harjoitus 8.9

Selitä näiden kahden lauseen ero:

```
int *pi0 = p2.get();
int *pi1 = p2.release();
```

Millaisissa tilanteissa kummankin käynnistys olisi sopivaa?

Harjoitus 8.10

Oletetaan, että meillä on seuraava:

```
auto_ptr< string > ps( new string( "Daniel" ) );
```

Mikä ero on seuraavien kahden assign()-funktion käynnistyksen välillä, vai onko eroa ollenkaan? Kumpi mielestäsi on parempi? Miksi?

```
ps.get()->assign( "Danny" );
ps->assign( "Danny" );
```

8.5 Nimiavaruuksien määrittelyt ◆A

Oletusarvoisesti globaalilla viittausalueella, jota kutsutaan *globaalin nimiavaruuden viittausalueeksi*, jokainen esitelty olio, funktio, tyyppi tai malli esittelee *globaalin olion*. Jokaisella globaalilla oliolla globaalin nimiavaruuden viittausalueella pitää olla yksilöllinen nimi. Esimerkiksi funktiolla ja oliolla ei voi olla sama nimi, oli ne esitelty samassa ohjelmatekstitiedostossa tai ei.

Tämä tarkoittaa, että jotta kirjastoja voidaan käyttää ohjelmissamme, pitää varmistua globaalien olioiden nimistä ohjelmissamme, etteivät ne törmää yhteen kirjaston globaalien olioiden nimien kanssa. Tämä voi olla melko vaikeaa saada väkisin aikaan, jos ohjelma muodostuu useiden eri valmistajien kirjastoista ja niissä esitellään monia nimiä globaalin nimiavaruuden viittausalueella. Kuinka voimme varmistua, kun yhdistelemme näiden eri valmistajien kirjastoja, että ohjelmiemme globaalien olioiden nimet eivät törmää yhteen kirjastoissa esiteltyjen globaalien olioiden nimien kanssa? Tätä nimien törmäystä kutsutaan *globaalin nimiavaruuden saastumisongelmaksi*.

Ohjelmoijat ovat pystyneet välttämään nämä ongelmat tekemällä globaalien olioidensa nimet erittäin pitkiksi varustamalla usein nimet ohjelmiinsa erityisellä merkkijonolla. Esimerkiksi:

```
class cplusplus_primer_matrix { ... };
void inverse( cplusplus_primer_matrix & );
```

Tämä ratkaisu ei kuitenkaan ole ihanteellinen. C++:lla kirjoitetussa ohjelmassa saattaa olla suuri määrä globaaleja luokkia, funktioita ja malleja, jotka ovat näkyvissä koko ohjelmassa. Ohjelmoijille voi olla hankalaa kirjoittaa ohjelmia, joissa käytetään niin pitkiä nimiä.

Nimiavaruudet mahdollistavat sen, että voimme paremmin hallita globaalin nimiavaruuden saastumisongelmaa. Kirjaston tekijä voi määritellä nimiavaruuden nimien piilottamiseksi globaalilta nimiavaruudelta. Esimerkiksi:

```
namespace cplusplus_primer {
    class matrix {  /* ... */  };
    void inverse ( matrix & );
}
```

Nimiavaruus cplusplus_primer on *käyttäjän esittelemä* nimiavaruus (verrattuna globaaliin nimiavaruuteen, joka on esitelty implisiittisesti ja on olemassa ohjelmassa).

Jokainen käyttäjän esittelemä nimiavaruus edustaa eri nimiavaruuden viittausaluetta. Käyttäjän esittelemä nimiavaruuden viittausalue voi sisältää toisia sisäkkäisiä nimiavaruusmäärityksiä kuten myös funktioiden, olioiden, mallien ja tyyppien esittelyitä. Nimiavaruudessa esiteltyjä olioita kutsutaan *nimiavaruuden jäseniksi*. Aivan kuten globaalilla nimiavaruuden viittausalueella, pitää myös käyttäjän esittelemässä nimiavaruudessa jokaisen nimen viitata yksilölliseen olioon tuossa nimi-

avaruudessa. Koska kuitenkin käyttäjän esittelemät eri nimiavaruudet koskevat eri viittausalueita, niissä voi olla keskenään samannimisiä jäseniä.

Nimiavaruuden jäsenen nimi yhdistetään automaattisesti eli *täsmennetään* nimiavaruutensa nimeen. Esimerkiksi cplusplus_primer-nimiavaruudessa esitelty matrix-luokan nimi on cplusplus_primer::matrix ja funktion nimi on cplusplus_primer::inverse().

Cplusplus_primer-nimiavaruuden jäseniä voidaan käyttää ohjelmassa niiden täsmennetyillä nimillä kuten seuraavassa:

```
void func( cplusplus_primer::matrix &m )
{
    // ...
    cplusplus_primer::inverse(m);
    return m;
}
```

Jos toisessa käyttäjän esittelemässä nimiavaruudessa (sanotaan vaikka DisneyFeatureAnimation) on myös matrix-luokka ja haluamme käyttää sitä luokkaa sen luokan sijasta, joka on määritelty cplusplus_primer-nimiavaruudessa, silloin func()-funktiota pitää muokata seuraavasti:

```
void func( DisneyFeatureAnimation::matrix &m )
{
    // ...
    DisneyFeatureAnimation::inverse(m);
    return m;
}
```

Tietysti nimiavaruuden jäseneen viittaaminen täsmennettyä merkintätapa käyttäen kuten tässä

```
namespace_name::member_name
```

voi olla kömpelöä. Tästä syystä jotkut mekanismit kuten *nimiavaruuden aliasnimet, using-esittelyt* ja *using-direktiivit* on tehty helpottamaan nimiavaruuden jäsenien käyttöä ohjelmissa. Esittelemme nämä mekanismit kohdassa 8.6.

8.5.1 Nimiavaruuden määrittelyt

Käyttäjän esittelemän nimiavaruuden määrittely alkaa avainsanalla namespace, jonka jälkeen tulee nimiavaruuden nimi. Tämän nimen pitää olla yksilöllinen nimi sillä viittausalueella, jolla nimiavaruus on määritelty. On virhe, jos yhdelläkin muulla saman nimialueen viittausalueella esitellyllä oliolla on sama nimi kuin nimiavaruudella. Tietysti tämä tarkoittaa, että globaalin nimiavaruuden nimien saastumisongelmaa ei ole eliminoitu. Nimiavaruuksien käyttö vähentää kuitenkin merkittävästi tätä ongelmaa.

Nimiavaruuden nimen jälkeen on lohko esittelyitä aaltosulkujen sisällä ({ }). Kaikki esittelyt, jotka voivat esiintyä globaalin nimiavaruuden viittausalueella, voidaan myös sijoittaa käyttäjän esittelemään nimiavaruuteen: luokat, muuttujat (alustuksineen), funktiot (määrittelyineen) ja mallit. Esittelyn sijoittaminen käyttäjän esittelemään nimiavaruuteen ei muuta sen mer-

kitystä. Ainoa ero on, että käyttäjän esittelemän nimiavaruuden esittelyssä mainitut nimet yhdistetään sen nimiavaruuden nimeen, jossa esittelyt ovat. Esimerkiksi:

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    void inverse ( matrix & );
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
    const double pi = 3.1416;
}
```

Luokan nimi, joka on esitelty cplusplus_primer-nimiavaruudessa, on

```
cplusplus_primer::matrix
```

Funktion nimi on

```
cplusplus_primer::inverse()
```

Vakion nimi on

```
cplusplus_primer::pi
```

Luokan, funktion tai vakion nimi täsmennetään sen nimiavaruuden nimellä, jossa se on esitelty; näiden nimien sanotaan olevan *täsmennettyjä nimiä*.

Nimiavaruuden määrittelyn ei tarvitse olla yhtenäinen. Esimerkiksi edellinen nimiavaruus olisi voitu määritellä seuraavasti:

```
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
}

namespace cplusplus_primer {
    void inverse ( matrix & );
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
}
```

Kaksi edellistä esimerkkiä ovat samanarvoiset: molemmat määrittelevät nimiavaruuden cplusplus_primer, joka sisältää luokan matrix, funktion `inverse()`, vakion `pi` ja operaattorin `operator+()`. Nimiavaruuden määrittely on siten kumulatiivinen.

Seuraavassa

```
namespace namespace_name {
```

määritellään uusi nimiavaruus, jos `namespace_name` ei viittaa aikaisemmin määriteltyyn nimiavaruuteen, tai jos tekee niin, se avaa tuon nimiavaruuden uudelleen uusien esittelyjen lisäämiseksi.

Tosiasia, että nimiavaruuden määrittelyt voivat olla epäyhtenäisiä, on suureksi avuksi, kun haluamme rakentaa kirjaston. Se mahdollistaa, että voimme helposti organisoida kirjaston lähdekoodia rajapinta- ja toteutusosiin. Esimerkiksi:

```
// Tämä osa nimiavaruudesta
// määrittelee kirjaston rajapinnan

namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &m1, const matrix &m2 );
    void inverse ( matrix & );
}

// Tämä osa nimiavaruudesta määrittelee
// kirjaston toteutuksen

namespace cplusplus_primer {
    void inverse ( matrix &m )
        { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
}
```

Nimiavaruuden ensimmäisessä osassa on esittelyt ja määrittelyt, jotka kuvaavat kirjaston rajapinnan: tyyppimäärittelyt, vakioiden määrittelyt ja funktioiden esittelyt. Nimiavaruuden toisessa osassa on kirjaston toteutuksen yksityiskohdat — tarkoittaa funktioiden määrittelyjä.

Se, mikä helpottaa järjestämään kirjastomme lähdekoodia vielä enemmän, on, että saman nimiavaruuden määrittely voi jakautua eri ohjelmatekstitiedostoihin. Nimiavaruuden määrittelyt eri ohjelmatekstitiedostoissa ovat myös kumulatiivisia. Kirjastomme voidaan siten järjestää seuraavasti:

```
// ---- primer.h ----
namespace cplusplus_primer {
    class matrix { /* ... */ };
    const double pi = 3.1416;
    matrix operator+ ( const matrix &m1, const matrix &m2 );
    void inverse( matrix & );
}

// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    void inverse( matrix &m )
        { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }
}
```

Ohjelma, joka käyttäisi kirjastoamme, voisi näyttää tältä:

```
// ---- user.C ----
// määrittelee kirjaston rajapinnan
#include "primer.h"

void func( cplusplus_primer::matrix &m )
{
    // ...
    cplusplus_primer::inverse( m );
    return m;
}
```

Tämä ohjelman järjestely antaa kirjastollemme tarvittavaa modulaarisuutta toteutuksen piilottamiseksi käyttäjiltä ja toisaalta mahdollistaa tiedostojen primer.C ja user.C kääntämisen ja linkittämisen yhdeksi ohjelmaksi saamatta aikaan yhtään käännös- tai linkitysvirhettä.

8.5.2 Viittausalueen operaattori (::)

Käyttäjän esittelemän nimiavaruuden jäsenen nimeen laitetaan automaattisesti etuliitteeksi viittausalueen operaattori (::). Nimiavaruuden jäsenen nimi tarkennetaan nimiavaruutensa nimellä.

Nimiavaruuden jäsenen nimen kuten matrix käyttäminen ilman tarkentavaa nimiavaruuden nimeä on virhe. Kääntäjä ei tiedä, mihin esittelyyn nimi matrix viittaa:

```
// määrittelee kirjaston rajapinnan
#include "primer.h"

// virhe: ei löydä esittelyä matrix:ille
void func( matrix &m );
```

Nimiavaruuden jäsenen nimi piilotetaan nimiavaruuteensa. Ellemme määritä kääntäjälle, mistä nimiavaruudesta esittelyä tulisi etsiä, kääntäjä etsii yksinkertaisesti nykyiseltä viittausalueelta ja kaikilta nykyisiltä sisäkkäisiltä viittausalueilta, joiden sisällä se on löytääkseen esittelyn nimelle. Jos esimerkiksi edellinen ohjelma kirjoitetaan uudelleen näin:

```
// määrittelee kirjaston rajapinnan
#include "primer.h"

class matrix { /* user definition */ };

// ok: löytää globaalin matrix-tyypin
void func( matrix &m );
```

Tällöin matrix-luokan määrittely löytyy globaalilta viittausalueelta ja ohjelma kääntyy asianmukaisesti. Koska nimiavaruuden matrix-jäsenen esittely on piilotettu cplusplus_primer-nimiavaruuteen, nimiavaruuden jäsenen nimi ei törmää yhteen globaalilla viittausalueella esitellyn luokan nimen kanssa. Tässä on syy, miksi sanomme, että nimiavaruudet ratkaisevat nimiava-

ruuksien saastumisongelman: nimiavaruuden jäsenen nimi ei löydy, ellei käyttäjä eksplisiittisesti käytä viittausalueen operaattoria ja nimiavaruuden nimeä. On olemassa muita mekanismeja, joita voidaan käyttää nimiavaruuden jäsenen esittelyn saamiseen näkyväksi nimiavaruutensa ulkopuolella. Näitä mekanismeja kutsutaan *using-esittelyiksi* ja *using-direktiiveiksi*. Ne esitellään seuraavassa kohdassa.

Huomaa, että viittausalueen operaattoria voidaan käyttää myös viittaamaan globaalin nimiavaruuden jäseniin. Koska globaalilla nimiavaruudella ei ole nimeä, ilmaisu

```
::member_name
```

viittaa globaalin nimiavaruuden jäseneen. Tämä voi olla melko hyödyllinen, kun viitataan globaalin nimiavaruuden jäseniin, joiden nimiä peittävät sisäkkäisillä viittausalueilla esitellyt nimet.

Seuraavassa esimerkissä, joka on kehitelty kuvaamaan sitä, kuinka viittausalueen operaattoria voidaan käyttää viittaamaan piilotettuun nimiavaruuden jäseneen, funktio laskee Fibonacci-jonoja. Muuttujasta `max` on kaksi määrittelyä. Globaali esittely ilmaisee jonojen maksimiarvon. Paikallinen esittely ilmaisee sarjojen halutun pituuden. (Muista, että funktion parametrit sijoitetaan funktion paikalliselle viittausalueelle.) Funktion pitää päästä käsiksi molempiin `max`-muuttujan esittelyihin. Kuitenkin jokainen täsmentämätön `max`-muuttujan käyttö viittaa paikalliseen esittelyyn. Jotta globaalia esittelyä voitaisiin hyödyntää, pitää käyttää viittausalueen operaattoria: `::max`. Tässä on toteutuksemme:

```
#include <iostream>
const int max = 65000;
const int lineLength = 12;

void fibonacci( int max )
{
    if ( max < 2 ) return;
        cout << "0 1 ";

    int v1 = 0, v2 = 1, cur;
    for ( int ix = 3; ix <= max; ++ix ) {
        cur = v1 + v2;
        if ( cur > ::max ) break;
        cout << cur << " ";
        v1 = v2;
        v2 = cur;
        if (ix % lineLength == 0) cout << endl;
    }
}
```

Tässä on main()-funktion toteutus, jolla funktiota kokeillaan:

```
#include <iostream>
void fibonacci( int );

int main() {
    cout << "Fibonacci Series: 16\n";
    fibonacci( 16 );
    return 0;
}
```

Kun ohjelma käännetään ja suoritetaan, se tuottaa seuraavan tulostuksen:

```
Fibonacci Series: 16
0 1 1 2 3 5 8 13 21 34 55 89
144 233 377 610
```

8.5.3 Sisäkkäiset nimiavaruudet

Mainitsimme aikaisemmin, että käyttäjän esittelemä nimiavaruus voi sisältää muita nimiavaruuksia. Voimme käyttää sisäkkäisiä nimiavaruuksia parantaaksemme edelleen koodimme järjestelyä kirjastossamme. Esimerkiksi:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // ensimmäinen sisäkkäinen nimiavaruus:
    // määrittelee kirjaston matriisi-osuuden
    namespace MatrixLib {
        class matrix {  /* ... */  };
        const double pi = 3.1416;
        matrix operator+ ( const matrix &m1, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
    // toinen sisäkkäinen nimiavaruus:
    // määrittelee eläintieteellisen osuuden kirjastosta
    namespace AnimalLib {
        class ZooAnimal { /* ... */ };
        class Bear : public ZooAnimal { /* ... */ };
        class Raccoon : public Bear { /* ... */ };
        // ...
    }
}
```

Nimiavaruus cplusplus_primer sisältää kaksi nimiavaruutta: MatrixLib ja AnimalLib.

Nimiavaruutta cplusplus_primer käytetään estämään kirjaston ja globaalin nimiavaruuden nimien yhteentörmäyksiä käyttäjiemme ohjelmissa. Kirjasto on myös järjestetty pienempiin pakkauksiin sisäkkäisine nimiavaruuksineen toisiinsa liittyvien esittelyiden ja määrittelyiden ryhmittämiseksi. Nimiavaruus MatrixLib sisältää primer-kirjaston matriisiosuuden, kun taas AnimalLib sisältää kirjaston ZooAnimal-osuuden.

Sisäkkäisen nimiavaruuden jäsenen esittelyn piilottaa sen nimiavaruuden jäsenen esittely, jonka sisällä se on. Sellaisen jäsenen nimen eteen liitetään automaattisesti ulomman nimiavaruuden nimi kuten myös sisäkkäisen nimiavaruuden nimi. Esimerkiksi sisäkkäisessä MatrixLib-nimiavaruudessa esitellyn luokan nimi on

```
cplusplus_primer::MatrixLib::matrix
```

ja funktion nimi on

```
cplusplus_primer::MatrixLib::inverse
```

Ohjelma voi käyttää sisäkkäisen nimiavaruuden `cplusplus_primer::MatrixLib` jäseniä seuraavasti:

```
#include "primer.h"

// kyllä, tämä on kaameaa...
// esittelemme pian mekanismeja, jotka saavat nimiavaruuden jäsenten
// käytön helpommaksi!
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplus_primer::MatrixLib::inverse( m );
    return m;
}
```

Sisäkkäinen nimiavaruus on sisäkkäinen viittausalue sillä nimiavaruudella, joka sen sisältää. Nimiresoluution aikana sisäkkäiset nimiavaruudet toimivat kuten sisäkkäiset lohkot. Kun esimerkiksi nimeä on käytetty nimiavaruuden määrittelyssä, esittelyä etsitään ympäröivistä nimiavaruuksista. Kun seuraavassa esimerkissä etsitään esittelyä `Type`-tyypille, otetaan huomioon esittelyt ennen `Type`-tyypin käyttöä. Nimiavaruuden MatrixLib esittelyt otetaan huomioon ensin; sitten otetaan huomioon cplusplus_primer-nimiavaruuden esittelyt ja lopuksi globaalin viittausalueen esittelyt.

```
typedef double Type;

namespace cplusplus_primer {
    typedef int Type; // hides ::Type

    namespace MatrixLib {
        int val;

        // Type: löytää esittelyn cplusplus_primer:ista
        int func(Type t) {
            double val; // peittää MatrixLib::val:in
            val = ...;
        }
        // ...
    }
}
```

Olio, joka on esitelty ympäröivässä nimiavaruudessa, piilotetaan samannimiseltä oliolta, joka on esitelty sen sisältämässä nimiavaruudessa. Edellisessä esimerkissä globaalin viittausalueen Type-esittely piilotetaan siltä Type-esittelyltä, joka tapahtuu cplusplus_primer-nimiavaruudessa. Kun nimiavaruudessa MatrixLib käytetty nimi Type on ratkaistu, nimiavaruudessa cplusplus_primer tehty esittely löytyy ja func() esitellään niin, että se saa int-tyyppisen parametrin.

Samalla tavalla olio, joka on esitelty nimiavaruudessa, piilotetaan oliolta, joka on esitelty paikallisella viittausalueella. Edellisessä esimerkissä nimiavaruuden MatrixLib val-muuttujan esittely piilotetaan func()-funktion paikalliselta val-muuttujan esittelyltä. Kun ratkaistaan func()-funktiossa käytettyä val-nimeä, löydetään paikallisen viittausalueen esittely, ja sijoittaminen func()-funktiossa tapahtuu paikalliseen muuttujaan.

8.5.4 Nimiavaruuden jäsenten määrittelyt

Olemme nähneet, että nimiavaruuden jäsenen määrittely voi tapahtua itse nimiavaruuden määrittelyssä. Esimerkiksi matrix-luokka ja pi-vakio on määritelty sisäkkäisen MatrixLib-nimiavaruuden määrittelyssä, kun taas funktioiden operator+() ja inverse() määrittelyt ovat ohjelman jossain myöhemmässä vaiheessa:

```
// ---- primer.h ----
namespace cplusplus_primer {
    // ensimmäinen sisäkkäinen nimiavaruus:
    // määrittelee kirjaston matriisiosuuden
    namespace MatrixLib {
        class matrix {  /* ... */  };
        const double pi = 3.1416;
        matrix operator+ ( const matrix &m1, const matrix &m2 );
        void inverse( matrix & );
        // ...
    }
}
```

On myös mahdollista määritellä kuinka monta tahansa nimiavaruuden jäsentä nimiavaruuden määrittelyn ulkopuolelle. Sellaisessa tapauksessa nimiavaruuden jäsen pitää täsmentää ympäröivien nimiavaruuksien nimillä. Esimerkiksi operator+()-funktio voidaan määritellä globaalilla viittausalueella seuraavasti:

```
// ---- primer.C ----
#include "primer.h"

// globaalin viittausalueen määrittely
cplusplus_primer::MatrixLib::matrix
    cplusplus_primer::MatrixLib::operator+
        ( const matrix& m1, const matrix &m2 )
            { /* ... */ }
```

Tässä määrittelyssä nimi operator+() täsmennetään nimiavaruuksien cplusplus_primer ja Matrix-Lib nimillä. Katso kuitenkin matrix-tyypin käyttöä operator+()-funktion parametriluettelossa. Nimeä ei ole täsmennetty sisäkkäisen nimiavaruuden nimellä cplusplus_primer::MatrixLib. Kuinka näin voi olla?

Funktion operator+() määrittely voi käyttää nimiavaruuden jäsenten nimiä niiden lyhyessä muodossa. Näin siitä syystä, että nimiavaruuden jäsenen määrittely on nimiavaruutensa viittausalueella. Nimiavaruuden MatrixLib jäsenet otetaan huomioon, kun funktiossa operator+() käytettyjä nimiä ratkotaan. Huomaa kuitenkin, että paluutyyppi pitää silti täsmentää. Tämä siitä syystä, että paluutyyppi ei ole funktion määrittelyn viittausalueella. Nimiavaruuden jäseniä voidaan käyttää niiden lyhyessä muodossa vain noudattamalla jäsenen nimeä:

```
cplusplus_primer::MatrixLib::operator+
```

Funktion operator+() määrittely voi käyttää nimiavaruuden jäsenten nimiä niiden lyhyessä muodossa missä tahansa esittelyssä, lausekkeessa, parametriluettelossa tai funktion rungossa. Esimerkiksi paikallinen esittely operator+()-funktiossa voi luoda matrix-luokan olion seuraavasti:

```
// ---- primer.C ----
#include "primer.h"

cplusplus_primer::MatrixLib::matrix
    cplusplus_primer::MatrixLib::operator+
        ( const matrix &m1, const matrix &m2 )
    {
        // esittelee paikallisen muuttujan, joka on tyyppiä:
        // cplusplus_primer::MatrixLib::matrix
        matrix res;

        // laske kahden matrix-olion summa
        return res;
    }
```

Vaikka nimiavaruuden jäsen voidaan määritellä nimiavaruutensa määrittelyn ulkopuolelle, on olemassa rajoituksia, missä tämä määrittely voi esiintyä. Vain nimiavaruudet, joiden sisällä jäsenen esittelyt ovat, voivat sisältää sen määrittelyn. Esimerkiksi operator+() voidaan esitellä globaalilla viittausalueella, nimiavaruudessa cplusplus_primer tai MatrixLib. Nämä ovat ainoat mahdollisuudet. Nimiavaruudessa MatrixLib määrittely näyttäisi tältä:

```
// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    MatrixLib::matrix  MatrixLib::operator+
        ( const matrix &m1, const matrix &m2 ) { /* ... */ }
}
```

Huomaa, että nimiavaruuden jäsen voidaan määritellä nimiavaruutensa määrittelyn ulkopuolelle vain, jos jäsen on aikaisemmin esitelty nimiavaruuden määrittelyssä. Funktion operator+() määrittely, joka juuri nähtiin, olisi virheellinen, ellei sitä olisi edeltänyt seuraava esittely otsikkotiedostossa primer.h:

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix {  /* ... */  };
        // seuraavaa esittelyä ei voi jättää pois
        matrix operator+ ( const matrix &m1, const matrix &m2 );
        // ...
    }
}
```

8.5.5 ODR ja nimiavaruuden jäsenet

Kuten aikaisemmin mainittiin, nimiavaruuden määrittely voi olla epäyhtenäinen ja jakaantua useisiin tiedostoihin. Nimiavaruuden jäsen voidaan siten esitellä monissa tiedostoissa. Esimerkiksi:

```
// primer.h
namespace cplusplus_primer {
    // ...
    void inverse( matrix & );
}
// use1.C
#include "primer.h"
// esittelee funktion cplusplus_primer:: inverse() tiedostossa use1.C

// use2.C
#include "primer.h"
// esittelee funktion cplusplus_primer:: inverse() tiedostossa use2.C
```

Jäsen cplusplus::inverse(), joka on esitelty otsikkotiedoston primer.h kautta tiedostossa use1.C, viittaa samaan funktioon kuin jäsen cplusplus::inverse(), joka on esitelty otsikkotiedoston primer.h kautta tiedostossa use2.C.

Vaikka nimiavaruuden jäsenen nimi on täsmennetty, se on kuitenkin globaali olio. ODR-vaatimus (jota käsiteltiin ensimmäisen kerran kohdassa 8.2), että muut kuin välittömät funktiot ja oliot saa määritellä ohjelmassa vain kerran, käy myös nimiavaruuden jäseniin. Tätä vaatimusta kunnioittaen nimiavaruuksia käyttävä ohjelma järjestetään yleensä seuraavasti:

1. Funktioiden ja olioiden esittelyt, jotka ovat nimiavaruuden jäseniä, sijoitetaan otsikkotiedostoon, joka otetaan mukaan niihin tiedostoihin, joissa nimiavaruuden jäseniä käytetään.

```
// ---- primer.h ----
namespace cplusplus_primer {
    class matrix { /* ... */ };
```

```
    // funktion esittelyitä
    extern matrix operator+ ( const matrix &m1, const matrix &m2 );
    extern void inverse( matrix & );

    // olioiden esittelyitä
    extern bool error_state;
}
```

2. Näiden jäsenten määrittelyjä voi esiintyä toteutustiedostossa.

```
// ---- primer.C ----
#include "primer.h"

namespace cplusplus_primer {
    // funktion määrittelyjä
    void inverse( matrix & )
        { /* ... */ }
    matrix operator+ ( const matrix &m1, const matrix &m2 )
        { /* ... */ }

    // olioiden määrittelyjä
    bool error_state = false;
}
```

Kuten olioiden esittelyissä globaalilla viittausalueella, pitää extern-avainsanaa käyttää määrittämään, että nimiavaruuden jäsen vain esitellään eikä määritellä. extern-avainsanaa voi käyttää myös nimiavaruuden jäsenfunktioiden esittelyissä; kuitenkin kuten globaaleilla funktioilla, extern-avainsanan käyttö on tässä tapauksessa valinnainen.

8.5.6 Nimeämättömät nimiavaruudet

Voit haluta määritellä olion, funktion, luokkatyypin tai jonkin muun olion, joka on näkyvissä vain pienessä ohjelman osassa. Tällä tavalla vähennämme nimiavaruuden saastumisongelmaa edelleen. Koska tiedämme, että tätä oliota käytetään vain rajoitetusti, emme aio uhrata energiaa varmistuaksemme, että oliolla on yksilöllinen nimi eikä se törmää yhteen muiden olioiden nimien kanssa, jotka on esitelty jossain muualla ohjelmassa. Kun esittelemme olion funktiossa tai sisäkkäisessä lohkossa, esittelyssä esille tuotu nimi on näkyvissä vain siinä lohkossa, jossa se on esitelty. Mutta mitä, jos ohjelmoija haluaa olion käytettäväksi useissa funktioissa ilman, että tekisi sen nimen käytettäväksi koko ohjelmassa?

Oletetaan esimerkiksi, että haluamme toteuttaa joukon lajittelufunktioita vektorin double-tyyppisten elementtien lajitteluun:

```
// ----- SortLib.h -----
void quickSort( double *, double * );
void bubbleSort( double *, double * );
void mergeSort( double *, double * );
void heapSort( double *, double * );
```

Kaikki nämä funktiot käyttävät samaa swap()-funktiota vektorin elementtien vaihtamiseen. Emme kuitenkaan halua swap()-funktiota näkyväksi koko ohjelmassa. Haluamme pitää sen lokalisoituna tiedostossa SortLib.C, koska nuo neljä funktiota ovat ainoat funktiot, jotka käynnistävät swap()-funktion. Seuraava ei anna meille sellaisia tuloksia, joita haluaisimme. Huomaatko, miksi?

```
// ----- SortLib.C -----
void swap( double *d1, double *d2 ) { /* ... */ }

// vain seuraavat funktiot käyttävät swap()-funktiota
void quickSort( double *d1, double *d2 ) { /* ... */ }
void bubbleSort( double *d1, double *d2 ) { /* ... */ }
void mergeSort( double *d1, double *d2 ) { /* ... */ }
void heapSort( double *d1, double *d2 ) { /* ... */ }
```

Vaikka swap()-funktio on määritelty SortLib.C-tiedostoon eikä esitelty otsikkotiedostossa SortLib.h, jossa lajittelukirjaston rajapinta on kuvattu, swap()-funktio on esitelty globaalilla viittausalueella. Se on siitä syystä globaali olio eikä sen nimi saa törmätä minkään muun globaalin olion nimen kanssa.

C++:ssa voidaan käyttää *nimeämätöntä nimiavaruutta* olion esittelyyn, joka on paikallinen tiedostolle. Nimeämättömän nimiavaruuden määrittely alkaa avainsanalla namespace. Koska nimiavaruus on nimetön, avainsanan jälkeen ei tule mitään nimeä. Avainsanan namespace jälkeen on lohko esittelyjä aaltosulkujen sisällä. Esimerkiksi:

```
// ----- SortLib.C -----
namespace {
    void swap( double *d1, double *d2 ) { /* ... */ }
}
// lajittelufunktioiden määrittelyjä kuten edellä
```

Funktio swap() on näkyvissä vain tiedostossa SortLib.C. Jos jossain toisessa tiedostossa on nimeämätön nimiavaruus ja sisältää swap()-funktion määrittelyn, on kyseessä eri funktion määrittely. Tosiasia, että on olemassa kaksi määrittelyä swap()-funktioille, ei ole virhe, koska funktiot ovat eri funktioita. Nimeämättömät nimiavaruudet eivät ole kuten muut nimiavaruudet; nimeämättömän nimiavaruuden määrittely on paikallinen tietyssä tiedostossa eikä koskaan jakaudu useisiin tekstitiedostoihin.

Nimeen swap() voidaan viitata sen lyhyessä muodossa tiedostossa SortLib.C noudattaen nimeämättömän nimiavaruuden määrittelyä. Ei ole välttämätöntä käyttää viittausalueen operaattoria nimeämättömän nimiavaruuden jäseniin viittaamiseen.

```
void quickSort( double *d1, double *d2 ) {
    // ...
    double* elem = d1;
    // ...
    // viittaa nimeämättömän nimiavaruuden jäseneen swap()
    swap( d1, elem );
```

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

Nimeämättömän nimiavaruuden jäsenet ovat ohjelman olioita. Funktiota swap() voidaan siten kutsua koko ohjelman keston ajan. Kuitenkin nimeämättömän nimiavaruuden jäsenten nimet ovat näkyvissä vain määrätyssä tiedostossa eivätkä näy muihin ohjelman muodostaviin tiedostoihin.

Ennen nimiavaruuksien esittelyä C++-standardissa oli yleinen ratkaisu tähän esittelyn paikallistamisongelmaan käyttää C:stä perittyä static-avainsanaa. Nimeämättömän nimiavaruuden jäsenellä on samanlaisia ominaisuuksia kuin globaalilla oliolla, joka on esitelty static-tyyppisenä. C:ssä globaali olio, joka on esitelty static-tyyppisenä, on näkyvissä sen tiedoston ulkopuolelle, jossa se on esitelty. Esimerkiksi esittely SortLib.C:ssä voidaan kirjoittaa C-tyylisesti kuten seuraavassa ja antaa swap()-funktiolle samat ominaisuudet:

```
// SortLib.C
// swap() on näkymätön ohjelman muille tiedostoille
static void swap( double *d1, double *d2 ) { /* ... */ }
// lajittelufunktioiden määrittelyt kuten edellä
```

Monet C++-toteutukset tukevat globaaleja, staattisia esittelyitä, vaikkakin on otaksuttu, että kun nimiavaruudet tulevat laajempaan käyttöön eri C++-toteutuksissa, globaalit staattiset esittelyt korvataan nimeämättömän nimiavaruuden jäsenillä.

Harjoitus 8.11

Miksi määrittelisit oman nimiavaruuden ohjelmiisi?

Harjoitus 8.12

Oletetaan, että meillä on seuraava esittely operaattorista operator*(), joka on sisäkkäisen nimiavaruuden cplusplus_primer::MatrixLib jäsen:

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix {  /* ... */  };
        matrix operator* ( const matrix &, const matrix & );
        // ...
    }
}
```

Kuinka määrittelisit tämän operaattorin globaalilla viittausalueella? Tee ainoastaan prototyyppi operaattorin määrittelystä.

Harjoitus 8.13

Selitä, miksi käyttäisit nimeämätöntä nimiavaruutta ohjelmissasi.

8.6 Nimiavaruuden jäsenten käyttö

Myönnettäköön, että aina, kun viitataan nimiavaruuden jäseneen käyttäen täsmennettyä ilmaisua nimiavaruuden_nimi::jäsenen_nimi, se on kömpelöä ja etenkin, jos nimiavaruuden nimi on pitkä. Jos täsmennettyjä nimiä pitäisi käyttää koko ajan, haluaisimme varmaan luoda erittäin lyhytnimisiä nimiavaruuksia, koska niitä on helpompi lukea ja kirjoittaa. Lyhyiden nimiavaruuksien nimien käyttö kuitenkin kasvattaa riskiä, että nimet törmäävät ohjelman muiden globaalien nimien kanssa, joten on parempi, että toimitamme kirjastomme pitkillä nimiavaruuksien nimillä.

Onneksi on olemassa mekanismi, joka helpottaa nimiavaruuksien jäsenten käyttöä ohjelmissamme. Nimiavaruuden aliakset, using-esittelyt ja using-direktiivit ovat mekanismeja, jotka auttavat meitä selviytymään erittäin pitkien nimiavaruuksien nimien epämukavuudesta.

8.6.1 Nimiavaruuden aliakset

Nimiavaruuden aliasta voidaan käyttää liitettäessä lyhyempi synonyymi nimiavaruuden nimeen. Esimerkiksi pitkä nimiavaruuden nimi kuten

```
namespace International_Business_Machines
    { /* ... */ }
```

voidaan liittää lyhyempään synonyymiin kuten seuraavassa:

```
namespace IBM = International_Business_Machines;
```

Nimiavaruuden aliaksen esittely alkaa avainsanalla namespace, jonka jälkeen ilmaistaan (lyhyempi) nimiavaruuden aliasnimi, sijoitusoperaattori ja alkuperäisen nimiavaruuden nimi. On virhe, jos alkuperäisen nimiavaruuden nimeä ei tunneta nimiavaruuden nimenä.

Nimiavaruuden alias voi viitata myös sisäkkäin olevaan nimiavaruuteen. Muistapa kamala func()-funktion määrittely, joka esiteltiin aikaisemmin:

```
#include "primer.h"

// vaikea lukea!
void func( cplusplus_primer::MatrixLib::matrix &m )
{
    // ...
    cplusplus_primer:: MatrixLib::inverse( m );
    return m;
}
```

Nimiavaruuden alias voidaan määritellä viittaamaan sisäkkäiseen nimiavaruuteen cplusplus_primer::MatrixLib, joka saa määrittelyn helpommin luettavaksi:

```
#include "primer.h"
```

```
// lyhempi aliasnimi
namespace mlib = cplusplus_primer::MatrixLib;

// helpompi lukea!
void func( mlib::matrix &m )
{
    // ...
    mlib::inverse( m );
    return m;
}
```

Nimiavaruudella voi olla monia synonyymejä eli aliasnimiä. Kaikkia aliaksia ja alkuperäistä nimiavaruuden nimeä voidaan käyttää vaihtoehtoisesti. Olkoon esimerkiksi, että aliasnimi Lib viittaa nimiavaruuteen cplusplus_primer, silloin func()-funktion määrittely voidaan kirjoittaa seuraavasti samalla, kun säilytetään sama merkitys:

```
// alias viittaa nimiavaruuteen cplusplus_primer
namespace alias = Lib;

void func( cplusplus_primer::matrix &m ) {
    // ...
    alias::inverse( m );
    return m;
}
```

8.6.2 Using-esittelyt

On mahdollista saada nimiavaruuden jäsenen nimi näkyväksi niin, että jäseneen voidaan viitata ohjelmassa käyttämällä sen nimen täsmentämätöntä muotoa eli etuliitettä nimiavaruuden_nimi::. Tämä on mahdollista, jos jäsen on esitelty käyttäen *using-esittelyä*.

Using-esittely alkaa avainsanalla using, jonka jälkeen kirjoitetaan nimiavaruuden jäsenen nimi. Jäsenen nimen using-esittelyssä pitää olla täsmennetty nimi. Esimerkiksi:

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */ };
        // ...
    }
}

// using-esittely nimiavaruuden jäsenelle matrix
using cplusplus_primer::MatrixLib::matrix;
```

Using-esittely esittelee nimen sille viittausalueelle, jossa using-esittely on. Esimerksiksi edellinen using-esittely esittelee matrix-nimen globaalilla viittausalueella. Sen jälkeen, kun using-esittely on kohdattu, matrix-nimen käyttö globaalilla viittausalueella tai sen sisältämillä viittausalueilla viittaa nimiavaruuden jäseneen. Oletetaan esimerkiksi, että seuraava esittely on using-esittelyn jälkeen:

```
void func( matrix &m );
```

Tämä esittelee func()-funktion, jonka parametrityyppi on cplusplus_primer::MatrixLib::matrix.

Using-esittely käyttäytyy kuten mikä tahansa esittely: sillä on viittausalue ja sen esittelemä nimi on näkyvissä using-esittelystä eteenpäin aina sen viittausalueen loppuun, josta esittely löytyy. Using-esittely voi esiintyä globaalilla viittausalueella kuten myös millä tahansa nimiavaruuden viittausalueella. Using-esittely voi esiintyä myös paikallisella viittausalueella. Kuten minkä tahansa muun esittelyn, myös using-esittelyn nimellä on nämä ominaisuudet:

- Sen pitää olla yksilöllinen viittausalueellaan.
- Se piilottaa saman nimen, joka on esitelty sitä ympäröivällä viittausalueella.
- Sen peittää samanniminen esittely, joka on sen sisältämällä viittausalueella.

Esimerkiksi:

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // muita esittelyitä
}
int bj = 0;

void manip() {
    using blip::bi; // bi funktiossa manip() viittaa tähän: blip::bi
    ++bi;          // aseta blip::bi arvoon 17

    using blip::bj; // peittää bj:n, joka on esitelty globaalilla viittausalueella
                    // bj funktiossa manip() viittaa tähän: blip::bj
    ++bj;          // aseta blip::bj arvoon 16

    int bk;        // esittely bk:n paikallisella viittausalueella
    using blip::bk; // virhe: bk:n uudelleenesittely manip():ssa
}

int wrongInit = bk; // virhe: bk ei näy tänne
                    // pitäisi olla blip::bk
```

Using-esittelyt manip()-funktiossa sallivat sen viittaukset blip-nimiavaruuden jäseniin niiden lyhyessä muodossa. Using-esittelyt eivät näy manip()-funktion ulkopuolelle ja käyttäjä voi käyttää lyhyitä nimiä vain manip()-funktiossa. Funktion ulkopuolella pitää käyttää täsmennettyjä nimiä.

Using-esittelyt helpottavat nimiavaruuden jäsenien käyttöä. Using-esittely esittelee vain yhden nimiavaruuden jäsenen kerrallaan. Se mahdollistaa, että voimme eritellä tarkoin, mitä nimiä käytämme ohjelmissamme. Using-esittelyä käytetään tietyllä viittausalueella ja se mahdollistaa, että voimme määrittää täsmällisesti, missä ohjelmassamme nimiavaruuden jäsenten nimet ovat näkyvissä lyhyessä muodossaan. Seuraavassa alikohdassa näemme, kuinka kaikki nimiavaruuden jäsenten nimet voidaan esitellä viittausalueella kerralla.

8.6.3 Using-direktiivit

Nimiavaruudet esiteltiin C++-standardin mukana. Sitä ennen C++-toteutukset eivät tukeneet nimiavaruuksia ja siitä seurauksena esistandardin kirjastot eivät kietoneet globaaleja esittelyitään nimiavaruuksiin. Merkittävä osa C++-koodista ja monet sovellukset kirjoitettiin ennen kuin nimiavaruudet tulivat käytettäviksi lukuisiin C++-toteutuksiin. Sulkemalla kirjaston sisällön nimiavaruuteen saamme mahdollisesti vanhat sovellukset käyttämään kirjaston vanhempia versioita. Jos kiedomme kirjaston sisällön nimiavaruuteen, kaikista kirjaston nimistä tulee täsmennettyjä, mikä tarkoittaa, että laitamme nimen eteen nimiavaruuden nimen ja viittausalueen operaattorin. Kaikki sovellukset, jotka käyttävät kirjaston nimiä niiden lyhyessä muodossa, lakkaavat toimimasta.

Voimme käyttää using-esittelyitä, jotta saamme kirjaston nimiä näkyviin ohjelmiemme käyttöön. Olettakaamme esimerkiksi, että primer.h-tiedosto sisältää uuden version kirjastosta, jossa globaalit esittelyt on kiedottu cplusplus_primer-nimiavaruuteen. Haluamme saada nopeasti ohjelmamme toimimaan uuden kirjaston kanssa. Voimme käyttää kahta using-esittelyä saadaksemme matrix-luokan ja inverse()-funktion esittelyt näkyviin cplusplus_primer-nimiavaruudesta .

```
#include "primer.h"
using cplusplus_primer::matrix;
using cplusplus_primer::inverse;

// using-esittelyn vuoksi
// nimiä matrix ja inverse voidaan käyttää ilman täsmennystä
void func( matrix &m ) {
    // ...
    inverse( m );
    return m;
}
```

Jos kirjasto on melko suuri ja jos sovellus käyttää monia kirjaston nimiä, kirjaston uuden version taaksepäin sovittaminen voi vaatia monia using-esittelyitä. Kaikkien tarpeellisten using-esittelyiden lisääminen vain siksi, että vanha koodi kääntyisi ja voitaisiin suorittaa kuten ennen, voi olla vaivalloista ja virhealtista. Tämän ongelman ratkaisemiseksi voidaan käyttää *using-direktiivejä* ja siten helpottaa siirtymistä eri kirjastoversioihin, jotka käyttävät nimiavaruuksia ensimmäistä kertaa.

Using-direktiivi alkaa avainsanalla using, jonka jälkeen kirjoitetaan avainsana namespace ja nimiavaruuden nimi. On virhe, jos nimi ei viittaa aikaisemmin määriteltyyn nimiavaruuden nimeen. Using-direktiivi mahdollistaa, että voimme saada kaikki tietyn nimiavaruuden nimet näkyviin niiden lyhyessä muodossa. Näitä jäseniä voidaan sitten käyttää ilman vaatimusta, että niiden nimet pitää täsmentää. Esimerkiksi edellä oleva koodi voidaan kirjoittaa seuraavasti uudelleen:

```
#include "primer.h"
// using-direktiivi: kaikista cplusplus_primer-nimiavaruuden jäsenistä tulee näkyviä
```

```
using namespace cplusplus_primer;

// nimiä matrix ja inverse voidaan käyttää ilman täsmennystä
void func( matrix &m ) {
    // ...
    inverse( m );
    return m;
}
```

Using-direktiivi saa nimiavaruuden jäsenten nimet näkyviin aivan kuin ne olisi esitelty nimiavaruuden ulkopuolella siinä paikassa, jossa nimiavaruuden määrittely sijaitsee. Using-direktiivin käytöstä johtuen on esimerkiksi niin kuin cplusplus_primer-nimiavaruuden jäsenet olisi esitelty globaalilla viittausalueella juuri ennen func()-funktion määrittelyä. Using-direktiivi ei esittele paikallisia aliaksia nimiavaruuden jäsenten nimille. Sen sijaan se vaikuttaa nimiavaruuden jäsenten nostamiseen viittausalueelle, joka sisältää nimiavaruuden määrittelyn. Koodi kuten

```
namespace A {
    int i, j;
}
```

näyttää tältä

```
int i, j;
```

koodille, jonka viittausalueella seuraava using-esittely on

```
using namespace A;
```

Katsokaamme esimerkkiä vastakohtana using-esittelyn (säilyttää nimiavaruuden viittausalueen, mutta liittää paikallisen synonyymin jäsenen nimeen) ja using-direktiivin vaikutukselle (poistaa nimiavaruuden kokonaan):

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // muut esittelyt
}
int bj = 0;

void manip() {
    using namespace blip; // using-direktiivi -
                          // törmäys ::bj:n ja blip::bj:n välillä
                          // havaitaan vain, kun bj:tä käytetään

    ++bi;                 // asettaa blip::bi:n arvoksi 17
    ++bj;                 // virhe: moniselitteinen
                          // globaali bj vai blip::bj?
    ++::bj;               // ok: asettaa globaalin bj:n arvoksi 1
    ++blip::bj;           // ok: asettaa blip::bj:n arvoksi 16

    int bk = 97;          // paikallinen bk peittää blip::bk:n
```

```
    ++bk;           // asettaa paikallisen bk:n arvoksi 98
}
```

Ensimmäinen asia, joka pitäisi huomata, on, että using-direktiivit ovat viittausalueilla. Using-direktiivi manip()-funktiossa toimii vain tuon funktion lohkossa. Funktiolle manip() blip-nimiavaruuden jäsenet näkyvät aivan kuin ne olisi esitelty globaalilla viittausalueella. Täten manip()-funktio voi viitata näiden jäsenten nimiin käyttäen niiden lyhyttä muotoa. Koodin, joka sijaitsee manip()-funktion ulkopuolella, pitää käyttää täsmennettyjä nimiä.

Toinen merkille pantava asia on, että moniselitteisyysvirheet, joita using-direktiivit voivat aiheuttaa, havaitaan silloin, kun nimeä käytetään, eikä silloin, kun using-direktiivi kohdataan. Esimerkiksi jäsen bj näkyy funktiolle manip() aivan kuin se olisi esitelty blip-nimiavaruuden ulkopuolella, globaalilla viittausalueella sillä paikalla, jossa nimiavaruuden määrittely sijaitsee. Globaalilla viittausalueella on kuitenkin jo muuttuja nimeltään bj. Nimen bj käyttö funktiossa manip() on siten moniselitteinen: nimi viittaa sekä globaaliin muuttujaan että blip-nimiavaruuden jäseneen. Using-direktiivi ei kuitenkaan ole virheellinen. Vain silloin, kun bj:tä käytetään manip()-funktiossa, moniselitteisyysvirhe havaitaan. Ellei bj:tä olisi koskaan käytetty manip()-funktiossa, ei virheilmoitusta olisi annettu.

Kolmas merkille pantava asia on, että using-direktiivien käyttö ei vaikuta täsmennettyjen nimien käyttöön. Kun manip() viittaa ::bj:hin, vain globaalilla viittausalueella esitelty muuttuja otetaan huomioon. Kun manip() viittaa blip::bj:hin, vain blip-nimiavaruudessa esitelty muuttuja otetaan huomioon.

Viimeinen merkille pantava asia on, että koska nimiavaruuden jäsenet näkyvät aivan kuin ne olisi esitelty nimiavaruuden ulkopuolella, *sillä paikalla, jossa nimiavaruus on määritelty*, jäsenet näkyvät manip()-funktiolle aivan, kuin ne olisi esitelty globaalilla viittausalueella. Tämä tarkoittaa, että paikalliset esittelyt manip()-funktiossa voivat peittää joitakin nimiavaruuden jäsenten nimiä. Paikallinen muuttuja bk peittää nimiavaruuden jäsenen blip::bk. Viittaus muuttujaan bk funktiossa manip() ei ole moniselitteinen; se viittaa paikalliseen muuttujaan bk.

Using-direktiivejä on helppo käyttää: yhdellä using-direktiivillä kaikki jäsenten nimet ovat näkyvissä. Vaikka tämä voi näyttää yksinkertaiselta ratkaisulta, sen ylikäyttö voi johtaa omaan ongelmaan. Jos sovellus käyttää monia kirjastoja ja jos näiden kirjastojen nimet on tehty näkyviksi using-direktiiveillä, olemme palanneet lähtöruutuun ja globaalin nimiavaruuden saastumisongelma ilmestyy jälleen. Esimerkiksi:

```
namespace cplusplus_primer {
    class matrix { };
    // muuta mukavaa ...
}
namespace DisneyFeatureAnimation {
    class matrix { };
    // tänne myös ...
}
using namespace cplusplus_primer;
using namespace DisneyFeatureAnimation;
```

```
matrix m; // virhe, moniselitteinen:
// cplusplus_primer:in vai DisneyFeatureAnimation:in?
```

Using-direktiivien aiheuttamat nimien moniselitteisyysvirheet havaitaan vasta siinä vaiheessa, kun nimiä käytetään. Edellisessä esimerkissä moniselitteisyysvirhe havaitaan, kun matrix-nimeä käytetään. Tämä myöhäinen havainto voi aiheuttaa yllätyksiä käyttäjille. Virheitä voi tulla esiin myöhemmin vaikka otsikkotiedostoja ei ole muutettu eikä uusia esittelyjä ole lisätty ohjelmaan. Virheet tulevat esille, kun yhtäkkiä päätämme käyttää kirjaston uusia piirteitä.

Using-direktiivit voivat olla erittäin hyödyllisiä, kun siirretään sovelluksia uusiin kirjastoversioihin, jotka on kiedottu nimiavaruuksiin. Monien using-direktiivien käyttö voi kuitenkin aiheuttaa globaalin nimiavaruuden saastumisongelman ilmestymisen uudelleen. Tätä ongelmaa voidaan minimoida korvaamalla using-direktiivejä valikoivammilla using-esittelyillä. Valikoivampien using-esittelyiden moniselitteisyysvirheet havaitaan esittelyhetkellä. Siitä syystä suosittelemme using-esittelyitä using-direktiivien sijasta, jotta globaalin nimiavaruuden nimien saastumisongelmaa voitaisiin paremmin hallita ohjelmissa.

8.6.4 Vakio nimiavaruus, std

Kaikki C++-vakiokirjaston komponentit on esitelty ja määritelty nimiavaruudessa nimeltään std. Jokainen funktio, olio ja luokkamalli, joka on esitelty vakio-otsikkotiedostossa kuten <vector> tai <iostream>, on esitelty std-nimiavaruudessa.

Jos kirjaston kaikki komponentit on esitelty std-nimiavaruudessa, mikä virhe liittyy kirjaston komponenttien nimiin, joita on käytetty seuraavassa kohdan 6.5 esimerkissä ?

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

int main()
{
    // syöttövirran iteraattori sidottu vakiosyöttöön
    istream_iterator<string> infile( cin );

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

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

    // käsittele svec
}
```

Aivan oikein — koodikatkelma ei käänny, koska std-nimiavaruuden jäseniä ei voi käsitellä ilman täsmennettyjä nimiä. Jotta voimme korjata tämän, teemme jonkin seuraavista asioista:

- Korvaamme esimerkin std-nimiavaruuden jäsenten nimet sopivasti täsmennetyillä nimillä.
- Käytämme using-esittelyitä saadaksemme std-nimiavaruuden jäsenet näkyviksi esimerkissämme.
- Käytämme using-direktiiviä saadaksemme kaikki std-nimiavaruuden jäsenet näkyviksi.

std-nimiavaruuden jäsenet, joita esimerkissä on käytetty, ovat seuraavat: istream_iterator-luokkamalli, ohjelman cin-vakiosyöttö, string-luokka ja vector-luokkamalli.

Yksinkertaisin ratkaisu on lisätä using-direktiivi #include-direktiivien jälkeen kuten seuraavassa:

```
using namespace std;
```

Using-direktiivi saa kaikki std-nimiavaruuden jäsenet näkyviksi esimerkissämme. Kuitenkin std-nimiavaruudessa on esitelty monia asioita. Pidämme using-esittelyitä parempina, jotta voimme vähentää tulevaisuuden nimikonflikteja, kun lisäämme uusia globaaleja esittelyitä ohjelmaamme.

Tarvittavat using-esittelyt esimerkkiohjelmamme kääntämiseksi ilman virheitä ovat seuraavat:

```
using std::istream_iterator;
using std::string;
using std::cin;
using std::vector;
```

Mutta minne sijoittaisimme nämä using-esittelyt? Jos ohjelmamme muodostuu monista tiedostoista, voi olla kätevää luoda otsikkotiedosto, joka sisältäisi kaikki ohjelmamme tarvitsemat std-nimiavaruuden jäsenten using-esittelyt. Tämä otsikkotiedosto otettaisiin mukaan C++-vakiokirjaston otsikkotiedostojen jälkeen ohjelmatekstitiedostoihin.

Pitääksemme koodiesimerkit lyhyinä ja koska monet esimerkit voidaan kääntää toteutuksissa, jotka eivät tue nimiavaruuksia, olemme tässä kirjassa jättäneet listaamatta eksplisiittisesti using-esittelyt, joita tarvitaan koodiesimerkkien kääntämiseen asianmukaisesti. On oletettu, että koodiesimerkeissä on käytetty std-nimiavaruuden jäsenille using-esittelyitä.

Harjoitus 8.14

Selitä erot using-esittelyiden ja using-direktiivien välillä.

Harjoitus 8.15

Kohdan 6.14 esimerkkiin liittyen, kirjoita tarvittavat using-esittelyt, joilla saat std-nimiavaruuden jäsenen näkyviin tässä esimerkissä.

Harjoitus 8.16

Tutki seuraavaa koodikatkelmaa:

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;

//1
void manip() {
    //2

    double dvar = 3.1416;

    int iobj = limit + 1;

    ++ivar;
    ++::ivar;
}
```

Mitkä ovat esittelyiden ja lausekkeiden vaikutukset tässä koodikatkelmassa, jos Exercise-nimiavaruuden kaikkien jäsenten using-esittelyt sijoitetaan kohtaan //1 tai kohtaan //2? Vastaa sitten samaan kysymykseen, mutta korvaa using-esittely Exercise-nimiavaruuden using-direktiivillä.