Luku 7

Lisää olioiden elinkaaresta

— Keitä ovat he? hän kysyi. — Nimenomaan kenen luulet koettavan murhata sinut?
– Jokaisen heistä, Jossarian sanoi hänelle.
. . . Clevinger luuli olevansa oikeassa, mutta Jossarianilla oli todisteita, sillä vieraat ihmiset, joita hän ei tuntenut, ampuivat häntä tykeillä joka kerta kun hän oli noussut ilmaan pudottamaan pommeja heidän niskaansa, eikä se ollut hauskaa.

– CATCH-22 *[Heller, 1961]*

Tavallisia perustietotyyppejä käytettäessä tulee usein vastaan tilanne, jossa muuttujasta halutaan tehdä kopio tai vastaavasti yhden muuttujan arvo halutaan sijoittaa toiseen. Sama tilanne toistuu joskus olioiden kanssa, varsinkin jos oliot edustavat abstrakteja tietotyyppejä, kuten päiväyksiä tai kompleksilukuja. Olio-ohjelmoinnissa sijoituksen ja kopioinnin merkitys ei kuitenkaan ole yhtä selvä kuin perinteisessä ohjelmoinnissa, joten niitä on syytä käsitellä tarkemmin.

C++:ssa varsinkin kopioimisen merkitys korostuu entisestään, koska kääntäjä itse tarvitsee olioiden kopiointia esimerkiksi välittäessään olioita tavallisina arvoparametreina tai palauttaessaan niitä paluuarvoina. Koska olioiden kopioiminen ja sijoitus saattaa olla hyvin raskas operaatio, on tärkeätä että ohjelmoija tietää, mitä kaikkea niihin liittyy ja missä tapauksissa kopioiminen on automaattista.

7.1 Olioiden kopiointi

Olioiden kopiointia tarvitaan useisiin eri tarkoituksiin. Joskus oliosta tulee tarve tehdä varmuuskopio, joskus taas taulukkoon halutaan tallettaa itsenäinen kopio oliosta, ei pelkästään viitettä alkuperäiseen olioon. Olio-ohjelmoinnin kannalta oleellinen kysymys kopioinnissa kuitenkin on: *"Mikä oikein on kopio?"*

Perustyyppien tapauksessa kopion määritteleminen ei ole vaikeaa, koska kopio voidaan luoda yksinkertaisesti luomalla uusi muuttuja ja kopioimalla vanhan muuttujan muisti uuteen bitti kerrallaan. Olioiden tapauksessa tämä ei kuitenkaan riitä, koska olion tilaan voi kuulua paljon muutakin kuin olion jäsenmuuttujat.

Yksi tapa määritellä olioiden kopiointi olisi sanoa, että uuden olion tulee olla identtinen vanhan kanssa. Tämäkin määritelmä tuottaa kuitenkin ongelmia: jos kyseessä kerran ovat *täysin* identtiset oliot, miten ne voi erottaa toisistaan, toisin sanoen miten tällöin tiedetään, että kyseessä todella on kaksi *eri* oliota? Jos oliot olisivat täysin identtiset, pitäisi periaatteessa toisen muuttamisen muuttaa myös toista, mikä tuskin on yleensä toivottavaa.

Parempi tapa määritellä kopiointi on sanoa, että oliota kopioitaessa uuden olion ja vanhan olion *arvojen* tai *tilojen* täytyy olla samat. Tässä tilalla ei tarkoiteta yksinkertaisesti jäsenmuuttujia vaan olion tilaa korkeammalla abstraktiotasolla ajatellen. Tällä tavoin ajateltuna esimerkiksi päiväysolion tila on se päiväys, jonka se sisältää. Vastaavasti merkkijono-olion tila on sen sisältämä teksti ja dialogi-ikkunan tila sisältää myös ruudulla näkyvän ikkunan. Näin ajateltuna olion tilan käsite ei enää riipu sen sisäisestä toteutuksesta vaan siitä, mitä olio ohjelmassa edustaa.

Edellä esitetty olioiden kopioinnin määritelmä tarkoittaa, että eri tyyppisiä olioita kopioidaan hyvin eri tavalla. Kompleksilukuolion voi kenties kopioida yksinkertaisesti muistia kopioimalla, kun taas merkkijonon kopiointi saattaa toteutuksesta riippuen vaatia ylimääräistä muistinvarausta olion ulkopuolelta ja muita toimenpiteitä. Näin kääntäjä ei pysty automaattisesti kopioimaan olioita, vaan luokan tekijän täytyy itse määritellä, mitä kaikkea oliota kopioitaessa täytyy tehdä.

Kaikkia olioita ei edes ole järkevää kopioida. Esimerkiksi hissin moottoria ohjaavan olion kopioiminen on todennäköisesti järjetön toimenpide, koska se vaatisi periaatteessa oliomallinnuksen mukai-

sesti myös itse fyysisen moottorin kopioimista. Tämän vuoksi olisi hyvä, jos luokan kirjoittaja voisi myös halutessaan *estää* kyseisen luokan olioiden kopioinnin kokonaan.

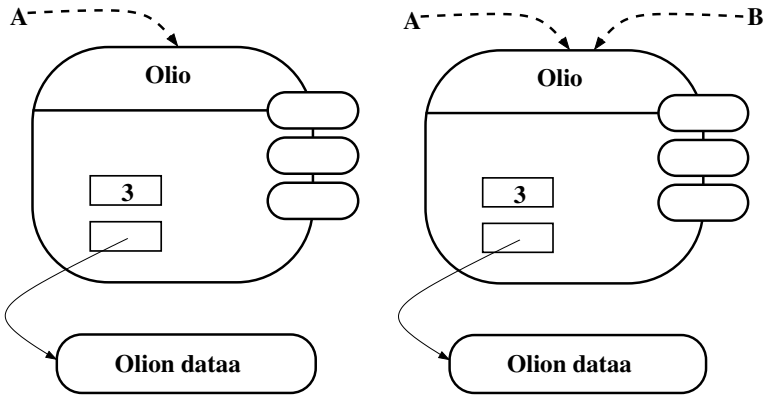
7.1.1 Erilaiset kopiointitavat

Kirjallisuudessa jaotellaan erilaiset olioiden kopiointitavat usein kolmeen kategoriaan: **viitekopiointiin**, **matalakopiointiin** ja **syväkopiointiin**. Tavallisesti olioiden kopiointi onkin mielekästä tehdä jollain edellä mainituista tavoista, mutta todellisuus on jälleen kerran teoriaa ihmeellisempää. Joskus saattaa tulla tarve kopioida osia oliosta yhdellä tavalla ja toisia osia toisella. Perusperiaatteiltaan tämä jaottelu on kuitenkin järkevä, joten tässä aliluvussa käydään läpi kaikki kolme kopiointitapaa.

Viitekopiointi

Viitekopiointi (*reference copy*) on kaikkein helpoin kopiointitavoista. Siinä ei yksinkertaisesti luoda ollenkaan uutta oliota, vaan "uutena oliona" käytetään *viitettä* vanhaan olioon. Kuva 7.1 havainnollistaa tätä. Kuvan tilanteessa viitteen B päähän "kopioidaan" olio A.

Viitekopiointia käytetään erityisesti oliokielissä, jossa itse muuttujat ovat aina vain viitteitä olioihin, jotka puolestaan luodaan dy-



(a) Ennen kopiointia (b) Kopioinnin jälkeen

KUVA 7.1: Viitekopiointi

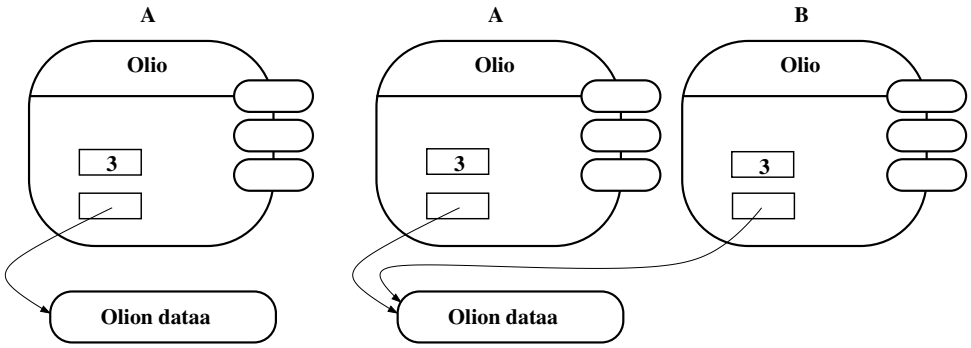
naamisesti. Tällaisia kieliä ovat esimerkiksi Java ja Smalltalk. Kun näissä kielissä luodaan uusi muuttuja ja alustetaan se olemassa olevalla muuttujalla, viittaavat molemmat muuttujat tämän jälkeen *samaan* olioon. C++:ssa sen sijaan viitekopiointia käytetään vain, kun erikseen luodaan viitteitä olioiden sijaan.

Viitekopioinnin hyvänä puolena on sen nopeus. “Kopion” luominen ei käytännössä vaadi ollenkaan aikaa, koska mitään kopioimista ei tarvitse tehdä. Viitekopiointi toimiikin hyvin niin kauan, kun olion arvoa ei muuteta. Mikäli sen sijaan oliota muutetaan toisen muuttujan kautta, vaikuttaa muutos tietysti myös toiseen muuttujaan. Tällainen käyttäytyminen on varsin haitallista, koska yksi tärkeä kopioinnin syy on tehdä oliosta “varmuuskopio”, joka säilyttää arvonsa.

Matalakopiointi

Matalakopioinnissa (*shallow copy*) itse *oliosta* ja sen *jäsenmuuttujista* tehdään kopiot. Jos kuitenkin jäsenmuuttujina on viitteitä tai osoittimia olion *ulkopuolisiin* tietorakenteisiin, ei näitä tietorakenteita kopioida vaan matalakopioinnin lopputuloksena molemmat oliot jakavat samat tietorakenteet. Kuva 7.2 havainnollistaa matalakopioinnin vaikutuksia.

Ohjelmointikielten toteutuksen kannalta matalakopiointi on selkeä operaatio, koska siinä kopioidaan aina kaikki olion jäsenmuuttujat eikä mitään muuta. Tästä syystä esimerkiksi C++ käyttää oletusar-



(a) Ennen kopiointia (b) Kopioinnin jälkeen

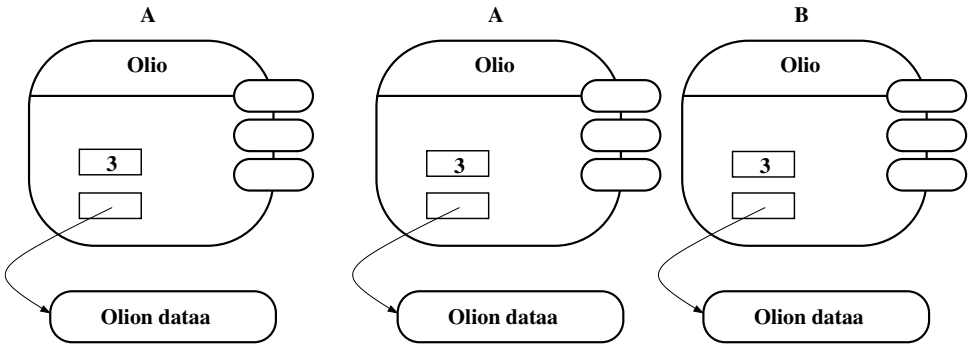
KUVA 7.2: Matalakopiointi

voisesti matalakopiointia, jos luokan kirjoittaja ei muuta määrää. Kopioinnin tuloksena on ainakin päällisin puolin kaksi oliota, ja aika monessa tapauksessa matalakopiointi onkin hyväksyttävä menetelmä. Yleensä viitekopiointia käyttävissä oliokielissä on myös jokin tapa matalakopiointiin. Esimerkiksi Javassa jokaisesta luokasta löytyy jäsenfunktio `clone`, joka oletusarvoisesti tuottaa oliosta matalakopioidun kopion. Samoin Smalltalk:n olioita voi pyytää suorittamaan toiminnon `copy`, joka tekee matalakopioinnin.

Ongelmaksi matalakopioinnissa muodostuu, että usein osa olion tilaan kuuluvasta tiedosta sijaitsee olion ulkopuolella viitteiden ja osoittimien päässä. Jotta kopioituun olioon tehdyt muutokset eivät heijastuisi alkuperäiseen olioon ja päinvastoin, täytyisi myös nämä olion ulkopuoliset tietorakenteet kopioida. C++:ssa kaikki olion dynaamisesti luomat oliot ja tietorakenteet sijaitsevat aina olion ulkopuolella, joten matalakopiointi ei yksinkertaisuudestaan huolimatta ole riittävä läheskään kaikille luokille.

Syväkopiointi

Syväkopiointi (*deep copy*) on kopiointimenetelmä, jossa olion ja sen jäsenmuuttujien lisäksi kopioidaan myös ne olion tilaan kuuluvat oliot ja tietorakenteet, jotka sijaitsevat olion ulkopuolella. Tämä näkyy kuvassa 7.3.



(a) Ennen kopiointia (b) Kopioinnin jälkeen

KUVA 7.3: Syväkopiointi

Olioiden kannalta syväkopiointi on ehdottomasti paras kopiointitapa, koska siinä luodaan kopio kaikista olion tilaan kuuluvista asioista. Näin kopioinnin jälkeen uusi olio ja alkuperäinen olio ovat täysin erilliset. Ohjelmointikielen kannalta syväkopiointi on kuitenkin ongelmallinen. Varsin tyypillisesti oliot sisältävät osoittimia myös sellaisiin olioihin ja tietorakenteisiin, jotka *eivät* varsinaisesti ole osa olion tilaa ja joita *ei* tulisi kopioida. Esimerkiksi kirjaston kirjat saattaisivat sisältää osoittimen siihen kirjastoon, josta ne on lainattu. Kirjan tietojen kopioiminen ei kuitenkaan saisi aiheuttaa koko kirjaston kopioimista.

Kääntäjän kannalta tilanne on ongelmallinen, koska useimmissa ohjelmointikielissä ei ole mitään tapaa ilmaista, mitkä osoittimet osoittavat olion tilaa sisältäviin tietoihin ja mitkä osoittavat olion ulkopuolisiin asioihin. Tämän vuoksi lähes mikään ohjelmointikieli ei automaattisesti tue syväkopiointia. Poikkeuksen tekee Smalltalk, jossa oliolta löytyy myös palvelu `deepCopy`.

Yleensä oliokielissä annetaan ohjelmoijalle itselleen mahdollisuus kirjoittaa syväkopioinnille toteutus, jota kieli sitten osaa automaattisesti käyttää. C++:ssa ohjelmoija kirjoittaa luokalle **kopiorakentajan**, joka suorittaa kopioinnin ohjelmoijan sopivaksi katsomalla tavalla. Samoin Javassa luokan kirjoittaja voi toteuttaa luokalle oman `clone`-jäsenfunktion, joka matalakopioinnin sijaan suorittaa sopivanlaisen syväkopioinnin. Näin kääntäjä tarvitsee syväkopioinnin toteutukseen apua luokan kirjoittajalta.

7.1.2 C++: Kopiorakentaja

C++:ssa olio kopioidaan käyttämällä **kopiorakentajaa** (*copy constructor*). Kopiorakentaja on rakentaja, joka ottaa parametrinaan viitteen toiseen samantyyppiseen olioon. Ideana on, että kun oliosta halutaan tehdä kopio, tämä olio luodaan kopiorakentajaa käyttäen. Tällöin kopiorakentaja voi alustaa uuden olion niin, että se on kopio parametrina annetusta alkuperäisestä oliosta. Kääntäjä käyttää itsekin automaattisesti kopiorakentajaa olion kopioimiseen tietyissä tilanteissa, joita käsitellään tarkemmin aliluvussa 7.3.

Listaus 7.1 seuraavalla sivulla näyttää osan yksinkertaisen merkkijonoluokan esittelystä ja sen kopiorakentajan toteutuksen. Kopiorakentaja alustaa uuden merkkijonon koon suoraan alkuperäisen merkkijonon vastaavasta jäsenmuuttujasta. Sen jälkeen se varaa tarvittaes-

sa tilaa uuden merkkijonon merkeille ja kopioi ne yksi kerrallaan vanhasta. Näin kopiorakentaja ei orjallisesti kopioi jäsenmuuttujia, vaan merkkijono-olion "syvimmän olemuksen" eli itse merkit.

Periytyminen ja kopiorakentaja

Periytyminen tuo omat lisänsä kopion luomiseen. Aliluokan olio koostuu useista osista, ja kantaluokan osilla on jo omat kopiorakentajansa, joilla kopion kantaluokkaosat saadaan alustetuksi. Aliluokan olion kopioiminen onkin jaettu eri luokkien kesken samoin kuin rakentajat yleensä: aliluokan kopiorakentajan vastuulla on kutsua kantaluokan kopiorakentajaa ja lisäksi alustaa aliluokan osa olioista kopioksi alkuperäisestä. Listaus 7.2 seuraavalla sivulla näyttää esimerkin päivätystä merkkijonoluokasta, joka on periytetty luokasta `Mjono`.

mjono.hh

```
 1  class Mjono
 2  {
 3  public:
 4    Mjono(char const* merkit);
 5    Mjono(Mjono const& vanha); // Kopiorakentaja
 6    virtual ~Mjono();
          ⋮
11  private:
12    unsigned long koko_;
13    char* merkit_;
14  };
```

mjono.cc

```
 1  Mjono::Mjono(Mjono const& vanha) : koko_(vanha.koko_), merkit_(0)
 2  {
 3    if (koko_ != 0)
 4    { // Varaa tilaa, jos koko ei ole nolla
 5      merkit_ = new char[koko_ + 1];
 6      for (unsigned long i = 0; i != koko_; ++i)
 7        { merkit_[i] = vanha.merkit_[i]; } // Kopioi merkit
 8      merkit_[koko_] = '\0'; // Loppumerkki
 9    }
10  }
```

LISTAUS 7.1: Esimerkki kopiorakentajasta

pmjono.hh

```
1  class PaivattyMjono : public Mjono
2  {
3  public:
4    PaivattyMjono(char const* merkit, Paivays const& paivays);
5    PaivattyMjono(PaivattyMjono const& vanha); // Kopiorakentaja
6
7    virtual ~PaivattyMjono();
         ⋮
15 private:
16   Paivays paivays_;
17 };
```

pmjono.cc

```
1  // Olettaa, että Paivays-luokalla on kopiorakentaja
2  PaivattyMjono::PaivattyMjono(PaivattyMjono const& vanha)
3    : Mjono(vanha), paivays_(vanha.paivays_)
4  {
5  }
```

LISTAUS 7.2: Kopiorakentaja aliluokassa

On huomattava, että jos aliluokan kopiorakentajassa *unohdetaan* kutsua kantaluokan kopiorakentajaa, pätevät normaalit C++:n säännöt, jotka sanovat että tällöin kutsutaan kantaluokan *oletusrakentajaa*. Oletusrakentaja puolestaan alustaa "kopion" kantaluokkaosan oletusarvoonsa, eikä olio näin kopioidu kunnolla! Jotkin kääntäjät antavat varoituksen, kun ne joutuvat kutsumaan kopiorakentajasta kantaluokan oletusrakentajaa, mutta itse C++-standardi ei vaadi tällaista varoitusta. Jos kantaluokalla ei ole oletusrakentajaa, ongelmia ei tule, koska tällöin kääntäjä antaa virheilmoituksen, jos kantaluokan rakentajan kutsu unohtuu aliluokan kopiorakentajasta.

Kääntäjän luoma oletusarvoinen kopiorakentaja

Jos luokalle ei ole määritelty kopiorakentajaa, kääntäjä olettaa luokan olevan niin yksinkertainen, että voidaan käyttää matalakopiointia eli kopioiminen voidaan suorittaa alustamalla kopion jäsenmuuttujat alkuperäisen olion jäsenmuuttujista. Niinpä kääntäjä kirjoittaa automaattisesti luokkaan tällaisen oletusarvoisen kopiorakentajan, jos luokasta ei löydy omaa kopiorakentajaa. Esimerkiksi aiemmin tässä teoksessa esiintyneellä Paivays-luokalla ei ollut kopiorakentajaa,

mutta listauksen 7.2 rivillä 3 voidaan alustuslistassa silti luoda kopio päiväysoliosta juuri tätä oletusarvoista kopiorakentajaa käyttäen.

Periaatteessa oletusarvoisen kopiorakentajan olemassaolo helpottaa yksinkertaisten ohjelmien tekemistä, koska aloittelevan ohjelmoijan ei tarvitse miettiä olioiden kopioimista. Käytäntö kuitenkin osoittaa, että noin 90 %:ssa tosielämän olioista kopiointiin liittyy muutakin kuin jäsenmuuttujien kopiointi. Erityisesti tämä tulee esille, jos luokassa on dataa osoittimien päässä, kuten listauksen 7.1 merkkijonoluokassa, jossa oletusarvoinen kopiorakentaja aiheuttaisi ohjelmaan vakavia toimintavirheitä. Tämän vuoksi ***jokaiseen luokkaan tulisi erikseen kirjoittaa kopiorakentaja*** eikä luottaa oletusarvoisen kopiorakentajan toimintaan.

Kopioinnin estäminen

Kääntäjän automaattisesti kirjoittamasta oletusarvoisesta kopiorakentajasta on haittaa, jos luokan olioita ei ole järkevää kopioida. Tällöinhän luokan kirjoittaja ei halua kirjoittaa omaa kopiorakentajaansa, mutta kääntäjä siitä huolimatta tarjoaa oletusarvoisen kopiorakentajan. Kopioinnin estäminen vaatiikin luokan kirjoittajalta lisäkikkoja.[†]

Kopioinnin estäminen tapahtuu esittelemällä kopiorakentaja luokan *private-puolella*. Tällöin luokan ulkopuolinen koodi ei pääse kutsumaan kopiorakentajaa eikä näin ollen kopioimaan olioita. Koska kopiorakentaja on kuitenkin esitelty, kääntäjä ei yritä tuputtaa oletusarvoista kopiorakentajaa. Näin kaikki ulkopuoliset kopiointiyritykset aiheuttavat käännösvirheen.

Edellä esitetty kikka kuitenkin ratkaisee vasta puolet ongelmasta. Luokan omat jäsenfunktiot pääsevät nimittäin tietysti käsiksi private-osaan ja voivat näin kutsua kopiorakentajaa. Tämä estetään sillä, että kopiorakentajan esittelystä huolimatta *kopiorakentajalle ei kirjoiteta ollenkaan toteutusta*. Jos nyt luokan oma koodi yrittää kopioida luokan olioita, linkkeri huomaa objektitiedostojen linkitysvaiheessa, että kopiorakentajalle ei löydy koodia ja aiheuttaa virheilmoituksen. Eri asia sitten on, kuinka helppo linkkerin virheilmoituksesta on päätellä, mikä on mennyt vikaan ja missä tiedostossa olevasta koodista vika aiheutuu.

[†] Kopioinnin — ja myöhemmin sijoituksen — estämisen vaikeus C++:ssa on yksi kielen suurimpia munauksia, ja estämiseen käytettävä kikka vastaavasti yksi rumimpia tekniikoita, joihin jopa tunnollinen ohjelmoija joutuu turvautumaan.

Vaikka yllä esitetty kikka onkin esteettisyydeltään kyseenalainen, on se ainoa tapa estää olioiden kopioiminen. Sitä tulisikin käyttää aina, kun luokan olioiden kopioiminen ei ole mielekästä.

7.1.3 Kopiointi ja viipaloituminen

Periytyminen tarkoittaa, että jokainen aliluokan olio on myös kantaluokan olio. Ikävä kyllä, tämä suhde ei ole ihan niin helppo ja yksinkertainen, kuin mitä voisi toivoa. Esimerkiksi olioiden kopioiminen tuottaa tietyissä tapauksissa ongelmia.

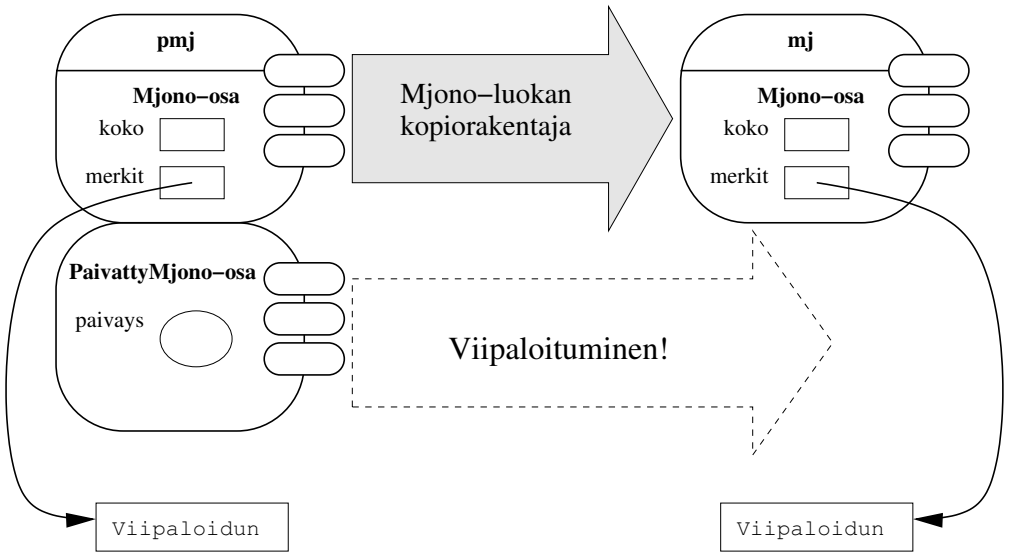
Kun C++:ssa luodaan kopiorakentajalla oliosta kopio, täytyy kopiota luovan ohjelmanosan tietää, minkä luokan oliota ollaan luomassa. Saman voi sanoa myös niin, että C++:ssa ei ole suoraan mitään tapaa luoda oliota ilman, että käännösaikana tiedetään sen tyyppi (tämä koskee sekä kopiointia että muutakin olion luomista). Tämä vaatimus tulee siitä, että kääntäjän on käännösaikana osattava varata jokaiselle oliolle riittävästi muistia, kutsua oikeaa rakentajaa yms.

Tämä rajoitus tarkoittaa, että kantaluokkaosoittimen tai -viitteen päässä olevasta oliosta ei voi luoda kunnollista kopiota, ellei osoittimen päässä olevan olion tyypistä voida olla varmoja jo käännösaikana. Kuten luvussa 6 todettiin, periytymistä käytettäessä on varsin tavallista, että kantaluokkaosoittimen päässä voi olla kantaluokkaolion lisäksi minkä tahansa periytetyn luokan olio.

Ongelmalliseksi tilanteen tekee se, että kääntäjän kannalta jokainen aliluokan olio *on* myös kantaluokan olio. Niinpä aliluokan olio kelpaa parametriksi kantaluokan kopiorakentajalle, koska se ottaa parametrinaan viitteen kantaluokkaan. Niinpä koodissa

```
PaivattyMjono pmj("Viipaloidun", jokupaivays);
Mjono mj(pmj);
```

kutsutaan oliota mj luotaessa Mjono-luokan kopiorakentajaa ja sille annetaan viite pmj:hin parametrina. Kopiorakentajan koodissa tämä parametri näyttää tavalliselta Mjono-oliolta, joten kopiorakentaja luo kopion muuttujan pmj *kantaluokkaosasta!* Sen sijaan aliluokan lisäämään päiväykseen ei kosketa lainkaan, koska Mjono-luokan kopiorakentaja ei ole siitä kuullutkaan! Kuva 7.4 seuraavalla sivulla havainnollistaa tilannetta. Koodia katsottaessa on tietysti selvää, ettei mj voi olla kopio pmj:stä, koska se ei edes ole oikeaa tyyppiä. Koodi kuitenkin kelpaa kääntäjälle, koska se noudattaa periytymisen sääntöjä.



KUVA 7.4: Viipaloituminen olion kopioinnissa

Tätä ilmiötä, jossa oliota kopioitaessa kopioidaankin erehdyksessä vain olion kantaluokkaosa, kutsutaan nimellä **viipaloituminen** (*slicing*) [Budd, 2002, luku 12]. Viipaloituminen esiintyy paitsi kopioimisessa myös sijoittamisessa (josta tarkemmin aliluvussa 7.2.3). Ilmiö osoittaa, kuinka ongelmallinen “aliluokan olio on kantaluokan olio” -suhde voi olla — aliluokan olion tulee olla kuin kantaluokan olio, *paitsi että* siitä ei pitäisi voida tehdä kantaluokkakopiota.

Viipaloitumisen esiintyminen

Viipaloituminen on C++:ssa vaarana kulissien takana monessa tilanteessa. Esimerkiksi allaolevassa koodissa tapahtuu viipaloituminen:

```
vector<Mjono> mjonovektori;
PaivattyMjono pmj("Metsään mennään", tanaan);
mjonovektori.push_back(pmj);
```

Koodissa vektorin loppuun lisätään *kopio* `push_back`:lle annetusta oliosta. `Mjono`-vektori voi sisältää vain `Mjono`-olioita, joten vektoriin lisätään viipaloitunut kopio `pmj`:n kantaluokkaosasta. Tämän vuoksi C++:ssa periytymistä ja polymorfismia käytettäessä vektoreihin ja muihin tietorakenteisiin tulee aina tallettaa *osoittimia* olioihin, ei itse

olioita. Viitteet puolestaan eivät kelpaa STL:n tietorakenteiden kuten vektorin alkioiksi. Tätä käsitellään aliluvussa 10.2 sivulla 311.

C++ kopioi olioita automaattisesti myös muissa tilanteissa, ja silloinkin viipaloituminen on vaarana. Tällaisia tilanteita ovat olioiden välittäminen arvoparametreina ja paluuarvoina, joista kerrotaan aliluvussa 7.3. Näissäkin tilanteissa ongelma ilmenee, kun parametri tai paluuarvo on kantaluokkatyyppiä, mutta ohjelmassa välitetään aliluokan olioita. Tällöin aliluokan oliosta välittyy vain viipaloitunut kantaluokkaosan kopio.

Viipaloituminen ja muut oliokielet

Kopioitumisen yhteydessä viipaloituminen on oliokielissä lähes yksinomaan C++:n ongelma. Tämä saattaa vaikuttaa yllättävältä — aiheutuuhan viipaloituminen periytymisestä, joka toimii samalla tavalla lähes kaikissa oliokielissä. Syy ongelman C++-keskeisyyteen juontaa juurensa tämän aliluvun alussa mainitusta C++:n vaatimuksesta, että olion (myös kopion) tyyppi täytyy tietää oliota luotaessa.

Suurimmassa osassa muita oliokieliä olioita käsitellään aina C++:n osoittimia muistuttavien mekanismien kautta (joita näissä kielissä usein kutsutaan viitteiksi). Näin tehdään muuan muassa Javassa ja Smalltalkissa. Näissä kielissä millään olioilla ei ole esimerkiksi omaa nimeä lainkaan, vaan niihin viitataan aina nimetyn olioviitteen kautta. Tähän liittyy myös se luvussa 3 mainittu asia, että olioiden elinkaari ei ole staattisesti määrätty, vaan roskienkeruu pitää huolen olioiden tuhoamisesta. C++:n näkökulmasta voitaisiin sanoa, että Javassa ja Smalltalkissa oliot luodaan *aina* dynaamisesti **new**'tä vastaavalla mekanismilla.

Aliluvussa 7.1.1 todettiin, että Javassa ja Smalltalkissa oliot kopioidaan kutsumalla kopioitavan olion kopiointipalvelua (Javan `clone`, Smalltalkin `copy` tai `deepCopy`). Yllättävää kyllä, tämä seikka yhdistettynä siihen, että olioita käsitellään aina viitteiden kautta, estää viipalointiongelman syntymisen kopioinnin yhteydessä. Kun nimittäin olio itse suorittaa kopioinnin, se pystyy luomaan oikeantyyppisen kopio-olion. Olio itse tietää oman todellisen tyyppinsä, joten se voi luoda oikeantyyppisen kopion itsestään, vaikka kopiointikutsu tehtäisiinkin kantaluokkaviitteen kautta.

Java ja Smalltalk eivät lisäksi tunne olioiden välittämistä arvoparametreina tai palauttamista paluuarvoina, vaan parametrit ja paluuar-

vot ovat aina olioviitteitä (osoittimia). Samoin tietorakenteisiin talletetaan aina olioviitteitä, koska muuta vaihtoehtoa ei kielissä ole. Niinpä kopiointiviipaloitumista ei näissä kielissä pääse syntymään muuallakaan. Sen sijaan viipaloituminen saattaa kyllä aiheuttaa ongelmia muualla, esimerkiksi sijoituksessa (tätä käsitellään aliluvussa 7.2.3).

Viipaloitumisen kiertäminen C++:ssa

Viipaloituminen ei muodostu C++:ssa ongelmaksi kovinkaan usein, koska varsin usein olioita kuljetetaan ohjelmassa osoittimia ja viitteitä käyttäen ilman, että niitä täytyy missään vaiheessa kopioida. Kopiointitapauksissakin tiedetään usein jo ohjelmointivaiheessa olion todellinen tyyppi, eikä viipaloitumista pääse tapahtumaan.

Aina silloin tällöin ohjelmissa tulee kuitenkin vastaan tilanne, jossa kantaluokkaosoittimen päässä oleva olio pitäisi pystyä kopioimaan, eikä oliosta tiedetä tarkasti, mihin luokkaan se kuuluu (paitsi että se on joko kantaluokan tai jonkin siitä periytetyn luokan olio). Tällöin ongelman voi ratkaista matkimalla muiden oliokielten kopiointitapaa ja antamalla kopioitavan olion kloonata itsensä. Kirjoitetaan kantaluokkaan virtuaalifunktio `kloonaa`, joka palauttaa osoittimen dynaamisesti luotuun kopioon oliosta. Tämä `kloonaa`-jäsenfunktio sitten toteutetaan jokaisessa hierarkian luokassa niin, että se luo **`new`**'llä kopion itsestään kopiorakentajaa käyttäen.

Listaus 7.3 seuraavalla sivulla näyttää esimerkin tällaisesta kloonaamisesta. Viipaloitumista ei pääse tapahtumaan, koska dynaaminen sitominen pitää huolen siitä, että oliolle kutsutaan aina sen omaa `kloonaa`-toteutusta, vaikka itse kutsu olisikin tapahtunut kantaluokkaosoittimen tai -viitteen kautta. Haittana tässä kopiointitavassa on se, että jokainen kopio luodaan dynaamisesti. Tästä johtuen kopiota ei tuhota automaattisesti, vaan kopioijan tulee itse muistaa tuhota kopio **`delete`**llä, kun sitä ei enää tarvita. Lisäksi tyypillisesti olioiden dynaaminen luominen vie hieman enemmän muistia ja on vähän hitaampaa kuin staattinen. Tästä ei kuitenkaan yleensä aiheudu käytännössä tehokkuushaittaa.

Ikävää kloonausratkaisussa on myös se, että *jokaisen* periytetyn luokan on muistettava itse toteuttaa `kloonaa`-jäsenfunktio. Jos tämä unohtuu, periytyy kantaluokan toteutus aliluokkaan, ja viipaloi-

```
 1  class Mjono
 2  {
 3  public:
10    Mjono(Mjono const& m);
11    virtual Mjono* kloonaa() const;
          ⋮
12  };
          ⋮
13  Mjono* Mjono::kloonaa() const
14  {
15    return new Mjono(*this);
16  }
          ⋮
17  class PaivattyMjono : public Mjono
18  {
19  public:
21    PaivattyMjono(PaivattyMjono const& m);
22    virtual PaivattyMjono* kloonaa() const;
          ⋮
23  };
          ⋮
24  PaivattyMjono* PaivattyMjono::kloonaa() const
25  {
26    return new PaivattyMjono(*this);
27  }
          ⋮
28  void kaytakopiota(Mjono const& mj)
29  {
30    Mjono* kopiop = mj.kloonaa(); // Tulos voi olla periytetyn kopio
31    // Täällä sitten käytetään kopiota
32    delete kopiop; kopiop = 0; // Pitää muistaa myös tuhota
33  }
```

LISTAUS 7.3: Viipaloitumisen kiertäminen `kloonaa`-jäsenfunktiolla

tuminen pääsee jälleen tapahtumaan.[8] Abstrakteissa kantaluokissa `kloonaa`-jäsenfunktiota ei voi toteuttaa, koska abstraktista kantaluokasta ei voi luoda olioita (eikä näin ollen myöskään kopiota). Tällaisessa kantaluokassa `kloonaa` esitelläänkin puhtaana virtuaalifunktiona, jolloin sen toteuttaminen jää konkreettisten aliluokkien vastuulle.

Listauksessa 7.3 saattaa ihmetyttää se, että `Mjono`-luokassa `kloonaa`-funktiosta palautetaan `Mjono`-osoitin, kun taas periytetyssä luokassa jäsenfunktion paluutyyppi onkin `PaivattyMjono`-osoitin. Tässä on kyseessä aliluvussa 6.5 mainittu paluutyypin kovarianssi. Se mahdollistaa sen, että kun kloonausta kutsutaan jonkin tyyppisen osoittimen kautta, saadaan paluuarvona samantyyppinen osoitin kopioon.

Kloonausfunktio antaa mahdollisuuden kopiointiin ilman viipaloitumista. Viipaloituminen on kuitenkin edelleen vaarana aiemmin käsitellyissä tietorakenteissa, parametrinvälityksessä ja paluuarvoissa. Näihin vaaroihin paras apu on huolellinen suunnittelu ja ongelman tiedostaminen. Yksi mahdollinen ratkaisu on myös suunnitella ohjelman luokkahierarkiat niin, että kaikki kantaluokat ovat abstrakteja. Tällöin viipaloitumista ei pääse syntymään, koska virheellistä kantaluokkakopiota on mahdotonta tehdä (abstrakteista kantaluokista ei saa tehdä olioita) [Meyers, 1996, Item 33].

7.2 Olioiden sijoittaminen

Olioiden kopioimisen lisäksi on toinenkin tapa saada aikaan kaksi keskenään samanlaista oliota: sijoittaminen. Sijoittaminen on imperatiivisessa ohjelmoinnissa erittäin tavallinen toimenpide, ja se opetetaan useimmilla ohjelmointikursseilla lähes ensimmäisenä. Useimmiten sijoitettavat muuttujat ovat jotain kielen perustyyppiä, mutta olio-ohjelmoinnissa luonnollisesti myös sijoitetaan olioita toisiinsa.

7.2.1 Sijoituksen ja kopioinnin erot

Olioiden sijoittaminen toisiinsa liittyy läheisesti olioiden kopiointiin, jopa siinä määrin, että joissain oliokielissä ei sijoittamista tunneta ol-

[8] Tätä mekaanista `kloonaa`-funktion kopioimista voi jonkin verran helpottaa sopivalla esikääntäjämakrokikkailulla, mutta se menee jo reilusti ohi tämän kirjan aihepiirin. Mutta kyllä **`#define`**:lläkin paikkansa on... ☺

lenkaan vaan se on korvattu kopioinnilla. Sijoituksen ja kopioinnin lopputulos on sama: kaksi oliota, joiden “arvo” on sama. Kopioinnissa kuitenkin luodaan *uusi* olio, joka alustetaan olemassa olevan perusteella, kun taas sijoituksessa on jo olemassa vanha olio, jonka “arvo” muutetaan vastaamaan toista oliota.

Sijoitukseen liittyvät ongelmat liittyvät yleensä juuri olion vanhan sisällön käsittelyyn. Usein sijoituksessa joudutaan ensin vapauttamaan vanhaa muistia ja muuten siivoamaan oliota vähän purkajan tapaan ja tämän jälkeen kopioimaan toisen olion sisältö. Tämä aiheuttaa tiettyjä vaaratilanteita, joita uuden olion luomisessa ei ole.

Erityisen hankalaksi sijoitus tulee, jos ohjelman täytyy varautua sijoituksessa mahdollisesti sattuviin virhetilanteisiin. Jos esimerkiksi halutaan, että sijoituksen epäonnistuessa olion vanha arvo säilyy, sijoitus täytyy tehdä varsin varovaisesti, jotta vanhaa arvoa ei heitetä roskiin liian aikaisin. Näitä ongelmia käsitellään jonkin verran virhekäsittelyn yhteydessä aliluvussa 11.8. Tällaisten vikasietoisten sijoitusten käsittely on kuitenkin varsin hankalaa (sijoituksesta ja virheistä C++:ssa on mainio artikkeli *“The Anatomy of the Assignment Operator”* [Gillam, 1997]).

On myös tapauksia, joissa olion kopioiminen on mahdollista ja sallittua mutta sijoittaminen ei kuitenkaan ole mielekästä. Esimerkkinä voisi olla vaikka piirto-ohjelman ympyräolio: kopion luominen olemassa olevasta ympyrästä on varmasti hyödyllinen ominaisuus, mutta ympyrän sijoittaminen toiseen ympyrään ei ole edes ajatuksena järkevä. Niinpä sijoittamisen ja kopioimisen salliminen tai estäminen on syytä pystyä tekemään toisistaan riippumatta.

7.2.2 C++: Sijoitusoperaattori

C++:ssa olioiden sijoittaminen tapahtuu erityisellä jäsenfunktiolla, jota kutsutaan **sijoitusoperaattoriksi** (*assignment operator*). Kun ohjelmassa tehdään kahden *olion* sijoitus `a = b`, kyseisellä ohjelmarivillä kutsutaan itse asiassa olion `a` sijoitusoperaattoria ja annetaan sille viite olioon `b` parametrina. Sijoitusoperaattorin tehtävänä on sitten tuhota olion `a` vanha arvo ja korvata se olion `b` arvolla. Se, mitä kaikkia operaatioita tähän liittyy, riippuu täysin kyseessä olevasta luokasta, kuten kopioinnissakin.

Sijoitusoperaattorin syntaksi näkyy listauksesta 7.4 seuraavalla sivulla. Sijoitusoperaattori on jäsenfunktio, joten se täytyy esitel-

lä luokan esittelyssä. Sen nimi on "**operator** =" (sanaväli yhtäsuuruusmerkin ja sanan **operator** välissä ei ole pakollinen, joten myös "**operator**=" kelpaa). Merkkijonoluokan sijoitusoperaattori ottaa parametrikseen vakioviitteen toiseen merkkijonoon, joka siis on sijoituslauseessa yhtäsuuruusmerkin oikealla puolella oleva olio, jonka arvoa ollaan sijoittamassa. Tätä operaattoria käyttäen sijoitus `a = b` aiheuttaa jäsenfunktiokutsun `a.operator =(b)`.

Sijoitusoperaattori palauttaa paluuarvonaan viitteen olioon itseensä. Syynä tähän on C-kielestä periytyvä mahdollisuus ketjusijoitukseen `a = b = c`, joka ensin sijoittaa `c`:n arvon `b`:hen ja sen jälkeen `b`:n `a`:han. Kun nyt ensimmäinen sijoitus palauttaa viitteen `b`:hen, ket-

mjono.hh

```
 1  class Mjono
 2  {
 3  public:
10    Mjono& operator =(Mjono const& vanha);

          ⋮

14  };
```

mjono.cc

```
 1  Mjono& Mjono::operator =(Mjono const& vanha)
 2  {

 3    if (this != &vanha)
 4    { // Jos ei sijoiteta itseen
 5      delete[] merkit_; merkit_ = 0; // Vapauta vanha
 6      koko_ = vanha.koko_; // Sijoita koko
 7      if (koko_ != 0)
 8      { // Varaa tilaa, jos koko ei nolla
 9        merkit_ = new char[koko_ + 1];
10        for (unsigned long i = 0; i != koko_; ++i)
11        { // Kopioi merkit
12          merkit_[i] = vanha.merkit_[i];
13        }
14        merkit_[koko_] = '\0'; // Loppumerkki
15      }
16    }
17    return *this;
18  }
```

LISTAUS 7.4: Esimerkki sijoitusoperaattorista

jusijoituksesti aiheutuu lopulta jäsenfunktioketju

```
a.operator =(b.operator =(c))
```

Sijoitusoperaattorin koodissa olion itsensä palauttaminen tapahtuu **this**-osoittimen avulla. Tämä osoitin osoittaa olioon itseensä, joten ***this** on tapa sanoa “minä itse”. Niinpä rivi 17 palauttaa viitteen olioon itseensä.

Sijoitusoperaattorin koodi on yleensä jonkinlainen yhdistelmä purkajan ja kopiorakentajan koodeista. Rivillä 3 olevaan ehtolauseeseen palataan myöhemmin. Rivi 5 tuhoaa merkkijonon vanhan sisällön, ja riveillä 6–15 varataan tilaa kopiolle uudesta merkkijonosta ja kopioidaan merkkijono talteen. Tämän jälkeen sijoitus onkin suoritettu.

Sijoitus itseen

Sijoituksessa on tietysti periaatteessa mahdollista, että joku yrittää sijoittaa olion itseensä, siis tyyliin `a = a`. Vaikka kukaan tuskin tällaista sijoitusta ohjelmaansa kirjoittaakaan, saattaa itseen sijoitus piiloutua ohjelmaan tilanteissa, joissa parametreina tai osoittimien kautta saatuja olioita sijoitetaan toisiinsa. Itseen sijoitus tuntuu täysin vaarattomalta operaatiolta, mutta sijoitusoperaattorin koodi on todella helppo kirjoittaa niin, että itseensä sijoittamisella on vakavat seuraukset.

Normaalisti sijoitusoperaattori toimii vapauttamalla ensin olion vanhaan arvoon liittyvän muistin ja mahdollisesti muut resurssit, jonka jälkeen se varaa uutta muistia ja tekee varsinaisen sijoituksen. Jos nyt oliota ollaan sijoittamassa itseensä, olion vanha ja uusi arvo ovat itse asiassa samat. Tällöin sijoitusoperaattori aloittaa toimintansa tuhoamalla olion arvon, jonka jälkeen se yrittää sijoittaa olion nyt jo tuhottua arvoa takaisin olioon itseensä.

Esimerkin merkkijonoluokan tapauksessa tämä aiheuttaisi sen, että merkkijonon merkit sisältävä muisti vapautetaan, jonka jälkeen sijoitusoperaattori varaa osoittimen päähän uutta muistia ja sen jälkeen tekee “tyhjän” kopioinnin, jossa uuden muistialueen alustamaton sisältö kopioidaan itsensä päälle. Joka tapauksessa olion sisältö on hukassa.

Itseen sijoituksen ongelma jää helposti huomaamatta, koska sijoitusoperaattorin koodi näyttää siltä, kuin meillä olisi kaksi oliota: olio

itse ja viiteparametrina saatu olio. Itseensä sijoituksessa nämä molemmat ovat kuitenkin sama olio.

Ongelmalle on kuitenkin onneksi helppo ratkaisu. Olion sijoittamisen itseensä ei tietenkään pitäisi tehdä mitään, joten sijoitusoperaattorin koodissa pitää vain tarkastaa, ollaanko oliota sijoittamassa itseensä. Jos näin on, voidaan koko sijoitus jättää suorittamatta. Itseen sijoittamisen tarkastamisessa täytyy tutkia, ovatko olio itse ja viiteparametrina saatu olio samat. C++:ssa olioiden samuutta voidaan tutkia osoittimien avulla. Jos kaksi samantyyppistä osoitinta ovat yhtä suuret, osoittavat ne varmasti samaan olioon. Niinpä listauksen 7.4 rivillä 3 tutkitaan, onko olioon itseensä osoittava osoitin **`this`** yhtä suuri kuin osoitin parametrina saatuun olioon. Jos osoittimet ovat yhtä suuret eli ollaan suorittamassa sijoitusta itseen, hypätään koko sijoituskoodin yli ja poistutaan sijoitusoperaattorista.

Periytyminen ja sijoitusoperaattori

Aliluokan sijoituksessa täytyy pitää huolta myös olion kantaluokkaosan sijoittamisesta aivan kuten kopioinnissakin. Kopiorakentajassa tämä tapahtui kantaluokan kopiorakentajaa kutsumalla, sijoituksessa vastaavasti periytetyn luokan sijoitusoperaattorissa tulee kutsua kantaluokan sijoitusoperaattoria. Listaus 7.5 seuraavalla sivulla näyttää päivätyn merkkijonon sijoitusoperaattorin, jossa merkkijono-osan sijoitus tapahtuu rivillä 5 eksplisiittisesti kantaluokan sijoitusoperaattoria kutsumalla. Tämän jälkeen rivillä 7 sijoitetaan aliluokan päiväysjäsenmuuttuja sen omaa sijoitusta käyttäen.

Kääntäjän luoma oletusarvoinen sijoitusoperaattori

Jos luokkaan ei kirjoiteta omaa sijoitusoperaattoria, kääntäjä lisää siihen automaattisesti oletusarvoisen sijoitusoperaattorin, joten tässäkin suhteessa kopiorakentaja ja sijoitusoperaattori muistuttavat toisiaan. Tämä oletusarvoinen sijoitusoperaattori yksinkertaisesti sijoittaa kaikki olion jäsenmuuttujat yksi kerrallaan. Tästä syystä listauksen 7.5 rivin 7 päiväysolion sijoitus onnistuu, vaikka päiväysluokalle ei ole kirjoitettu sijoitusoperaattoria.

Oletusarvoisen sijoitusoperaattorin ongelmat ovat samat kuin oletusarvoisen kopiorakentajankin. Jos luokassa on jäsenmuuttujina osoittimia, ne sijoitetaan normaalia osoittimien sijoitusta käyttäen,

pmjono.hh

```
1  class PaivattyMjono : public Mjono
2  {
3  public:
14    PaivattyMjono& operator =(PaivattyMjono const& vanha);
          ⋮
17 };
```

pmjono.cc

```
1  PaivattyMjono& PaivattyMjono::operator =(PaivattyMjono const& vanha)
2  {
3     if (this != &vanha)
4     { // Jos ei sijoiteta itseen
5        Mjono::operator =(vanha); // Kantaluokan sijoitusoperaattori
6        // Oma sijoitus, oletetaan että Paivays-luokalla on sijoitusoperaattori
7        paivays_ = vanha.paivays_;
8     }
9     return *this;
10 }
```

LISTAUS 7.5: Sijoitusoperaattori periytetyssä luokassa

jolloin sijoituksen jälkeen molempien olioiden osoittimet osoittavat samaan paikkaan ja oliot jakavat osoittimen päässä olevan datan. Useimmissa tapauksissa tämä ei kuitenkaan ole se, mitä halutaan, ja esimerkiksi merkkijonoluokassa oletusarvoinen sijoitusoperaattori toimisi pahasti väärin. Tämän takia ***jokaiseen luokkaan tulisi erikseen kirjoittaa sijoitusoperaattori***.

Sijoituksen estäminen

Jos luokan olioiden sijoitus ei ole mielekästä, voidaan sijoittaminen estää samalla tavoin kuin kopioiminen. Luokan sijoitusoperaattori esitellään luokan esittelyssä private-puolella, jolloin sitä ei päästä kutsumaan luokan ulkopuolelta. Tämän jälkeen sijoitusoperaattorille ei kirjoiteta ollenkaan toteutusta, jolloin sen käyttö luokan sisällä aiheuttaa linkitysaikaisen virheilmoituksen.

Varsin usein sijoituksen ja kopioinnin mielekkyys käyvät käsi kädessä, joten yleensä luokalle joko kirjoitetaan sekä kopiorakentaja että sijoitusoperaattori tai sitten molempien käyttö estetään. Mutta kuten jo aiemmin todettiin, poikkeuksia tähän periaatteeseen on helppo keksiä.

7.2.3 Sijoitus ja viipaloituminen

Aliluvussa 7.1.3 mainittu viipaloitumisongelma ei koske ainoastaan olioiden kopiointia. Koska sijoitus ja kopiointi ovat hyvin lähellä toisiaan, on luonnollista, että viipaloituminen on vaarana myös sijoituksessa. Lähtökohdiltaan tilanne on sama kuin kopioimisessakin: jokainen aliluokan olio on myös kantaluokan olio, joten aliluokan olion voi sijoittaa kantaluokan olioon. Tällöin aliluokasta sijoitetaan ainoastaan kantaluokkaosa ja periyttämällä lisättyä osaa ei huomioida mitenkään. Jälleen kerran kääntäjä ei varoita tästä mitenkään, koska periytymisen tyypityksen kannalta asia on niin kuin pitääkin.

Sijoitus tuo viipaloitumiseen vielä lisää kiemuroita. Olion, johon sijoitus tapahtuu, ei välttämättä tarvitse olla kantaluokan olio, vaan se voi olla myös *jonkin toisen* samasta kantaluokasta periytetyn luokan olio. Tällaisen erityyppisten olioiden sijoituksen ei tietenkään pitäisi olla edes mahdollista. Sijoitus ja viipaloituminen tulevat kuitenkin mahdolliseksi, jos sijoittaminen tapahtuu olioon osoittavan kantaluokkaosoittimen tai -viitteen kautta. Kääntäjä valitsee käytettävän sijoitusoperaattorin osoittimen tyypin perusteella, joten valittua tulee kantaluokan sijoitus. Tällöin kantaluokan sijoitusoperaattori sijoittaa erityyppisten olioiden kantaluokkaosat, ja aliluokkien lisäosiin ei kosketa. Listaus 7.6 seuraavalla sivulla näyttää esimerkin molemmista viipaloitumismahdollisuuksista. Kuva 7.5 sivulla 223 havainnollistaa listauksessa tapahtuvaa viipaloitumista.

Koska sijoitusviipaloitumisessa on kysymys väärän sijoitusoperaattorin valitsemisesta, luonnolliselta ratkaisulta saattaisi tuntua sijoitusoperaattorin muuttaminen virtuaaliseksi. Tämä ei kuitenkaan ole mahdollista. Syynä on se, että virtuaalifunktion parametrien tulee pysyä luokasta toiseen samana, kun taas sijoitusoperaattorin parametrin tulee olla aina vakioviite luokkaan itseensä. Toisekseen virtuaalisuus ei muutenkaan ratkaisisi koko ongelmaa, koska sijoituksessa aliluokasta kantaluokkaan kantaluokan sijoitusoperaattori on ainoa mahdollinen.

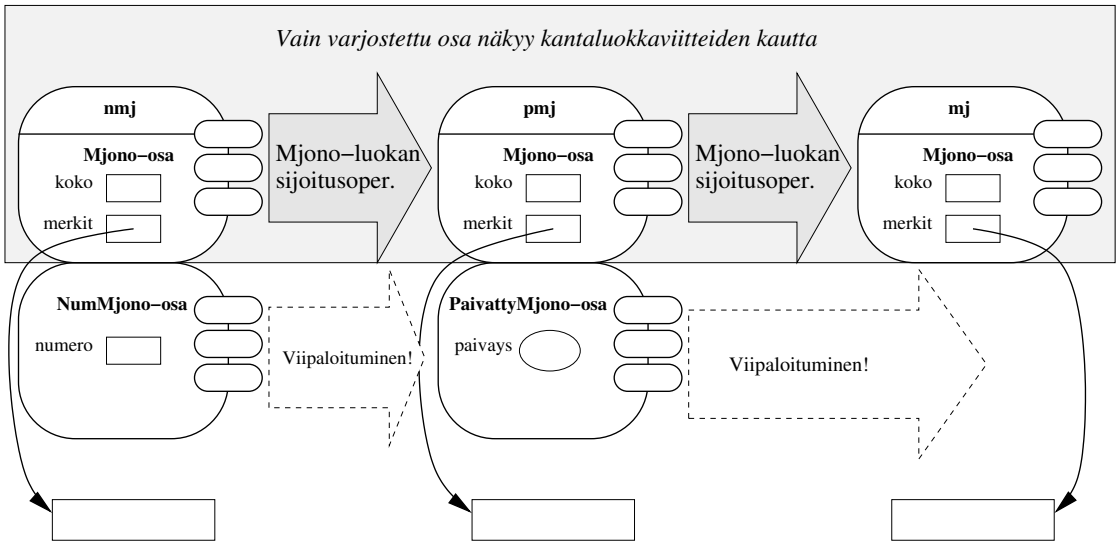
Viipaloitumisvaaran estämiskeinoja C++:ssa on muutama. Helpoin on tehdä luokkahierarkia, jossa kaikki kantaluokat ovat abstrakteja, jolloin niillä ei ole sijoitusoperaattoria ollenkaan. Tällöin viipaloitumista ei voi tapahtua [Meyers, 1996, Item 33]. Toinen ratkaisu on tarkastaa aina sijoituksen yhteydessä, että olio itse ja sijoitettava olio kuuluvat samaan luokkaan kuin suoritettava sijoitusoperaatto-

```
        ⋮
1  class NumMjono : public Mjono
2  {
3  public:
4    NumMjono(char const* merkit, int numero);
        ⋮
5    NumMjono& operator =(NumMjono const& nmj);
6
7  private:
8    int numero_;
9  };
        ⋮
10 void sijoita(Mjono& mihin, Mjono const& mista)
11 {
12   mihin = mista;
13 }
14
15 int main()
16 {
17   Mjono mj("Tavallinen");
18   PaivattyMjono pmj("Päivätty", tanaan);
19   NumMjono nmj("Numeroitu", 12);
20
21   sijoita(pmj, nmj); // Viipaloituminen funktion sisällä!
22   sijoita(mj, pmj);       // Täällä myös!
23 }
```

LISTAUS 7.6: Viipaloitumismahdollisuudet sijoituksessa

ri. Tämä onnistuu listauksen 7.7 seuraavalla sivulla osoittamalla tavalla **typeid**-operaattorin avulla. Tämä tarkastus on kuitenkin vasta ajoaikainen, joten mahdollisten virheiden havaitseminen jää ohjelman testausvaiheeseen.

Sijoitukseen liittyvä viipaloituminen ei sinänsä ole pelkästään C++:n ongelma, vaan se ilmenee myös muissa oliokielissä kuten Javassa ja Smalltalkissa, jos olioita (eikä pelkästään olio-osoittimia) voi sijoittaa toisiinsa. Näissä kielissä ei kuitenkaan yleensä ole erillistä sijoitusoperaattoria, vaan ohjelmoijan täytyy itse kirjoittaa `sijoita`-palvelu tms. Tällaisissa tilanteissa viipaloituminen tulee mahdolliseksi näissäkin kielissä.



KUVA 7.5: Viipaloituminen olioiden sijoituksessa

```
1  #include <typeinfo>
2  Mjono& Mjono::operator =(Mjono const& m)
3  {
4     if (typeid(*this) != typeid(MJono) || typeid(m) != typeid(MJono))
5     {
6        /* Virhetoiminta */
7     }
8     // Tai assert(typeid(*this) == typeid(MJono) && typeid(m) == typeid(MJono));
9     // (ks. aliluku 8.1.4)
10    if (this != &m)
11    {
          ⋮
12    }
13    return *this;
14 }
```

LISTAUS 7.7: Viipaloitumisen estäminen ajoaikaisella tarkastuksella

7.3 Oliot arvoparametreina ja paluuarvoina

Kun C:ssä ja C++:ssa kokonaislukumuuttuja välitetään funktioon normaalina parametrina, käytetään **arvonvälitystä** (*call by value*). Tämä tarkoittaa, että funktioon ei välitetä itse muuttujaa vaan sen *arvo*. Kun nyt C++:ssa halutaan välittää olioita funktioon vastaavanlaisina arvoparametreina, joudutaan jälleen miettimään, mikä oikein on se olion "arvo", joka funktioon välitetään.

Kopioinnin yhteydessä olion arvon käsitettä on jo pohdittu, joten on luonnollista käyttää samaa ideaa tässäkin tapauksessa. Kun olio välitetään funktiolle arvoparametrina, funktion sisälle luodaan *kopio* parametrioliosta käyttämällä luokan kopiorakentajaa. Koska kopiorakentaja tulee määritellä niin, että alkuperäisen olion ja uuden olion "arvot" ovat samat, näin saadaan samalla välitetyksi parametrin arvo funktion sisälle.

Funktion sisällä parametrin kopio käyttäytyy kuten tavallinen funktion paikallinen olio, ja sitä pystyy käyttämään normaalisti parametrin nimen avulla. Parametriin tehtävät muutokset muuttavat tietysti vain kopiota, joten alkuperäinen olio säilyy muuttumattomana. Kun lopulta funktiosta palataan, parametrikopio tuhotaan aivan kuten normaalit paikalliset muuttujatkin.

Olioiden käyttö arvoparametreina on siis täysin mahdollista C++:ssa, kunhan luokalla vain on toimiva kopiorakentaja. Arvoparametrien käyttö ei kuitenkaan yleensä ole suositeltavaa, koska olion kopioiminen ja kopion tuhoaminen funktion lopussa ovat mahdollisesti hyvin raskaita toimenpiteitä. Toinen vaara on, että funktion sisässä muutetaan parametrikopiota ja luullaan, että muutos tapahtuukin parametrina annettuun alkuperäiseen olioon. Viimeiseksi periytyminen ja arvonvälitys saavat aikaan viipaloitumisen vaaran (aliluku 7.1.3). Kaiken tämän vuoksi olioiden parametrinvälityksessä kannattaa aina käyttää viitteitä — ja erityisesti **`const`**-viitteitä — aina kun se on mahdollista.

Kun funktiosta palautetaan olio paluuarvona, tilanne on vielä mutkikkaampi. Olio, joka palautetaan **`return`**-lauseessa, on todennäköisesti funktion paikallinen olio, joten se tuhoutuu heti funktiosta palattaessa. Tämä kuitenkin tarkoittaa sitä, että olio tuhoutuu, ennen kuin paluuarvoa ehditään käyttää kutsujan puolella! Niinpä funktion paluu*arvo* täytyykin saada talteen kutsujan puolelle, ennen kuin funktion paikalliset oliot tuhotaan.

Sana “arvo” esiintyy tässäkin yhteydessä, joten ongelma ratkeaa kopioinnilla. Kun funktiosta palautetaan **return**-lauseella olio, funktion *kutsujan* puolelle luodaan nimetön **väliaikaisolio** (*temporary object*), joka alustetaan kopiorakentajalla kopioksi **return**-lauseessa esiintyvästä oliosta. Tämän jälkeen funktiosta voidaan palata ja tuhota kaikki paikalliset oliot, mukaanlukien alkuperäinen paluuarvo. Kutsujan koodissa väliaikaisolio toimii funktion “paluuarvona”, ja jos paluuarvo esimerkiksi sijoitetaan talteen toiseen olioon, väliaikaisolio toimii sijoituksen alkuarvona. Kun väliaikaisoliota ei lopulta enää tarvita, se tuhotaan automaattisesti. Näin väliaikaisoliot eivät voi aiheuttaa muistivuotoja.

C++ antaa kääntäjälle olioita palautettaessa mahdollisuuden optimointeihin ja tehokkaan koodin tuottamiseen. Paluuarvojen tapauksessa kääntäjä voi esimerkiksi käyttää väliaikaisolion sijaan jotain tavallista oliota, jos se huomaa että väliaikaisolio välittömästi sijoitettaisiin tai kopioitaisiin johonkin toiseen olioon.

Kuten arvoparametrienkin tapauksessa, olioiden välittäminen paluuarvona onnistuu C++:ssa ilman erityisiä ongelmia, kunhan luokan kopiorakentaja toimii oikein. Paluuarvon kopiointi ja tuhoaminen aiheuttavat kuitenkin jälleen sen, että olion palauttaminen saattaa olla hyvin raskas operaatio. Myös viipaloituminen on vaarana, jos kantaluokkaolion palauttavasta funktiosta yritetäänkin palauttaa aliluokan oliota (aliluku 7.1.3). Olioiden palauttamista paluuarvoina kannattaakin välttää, jos mahdollista. Olion palauttamiselle ei kuitenkaan ole

```
1  Paivays kuukaudenAlkuun(Paivays p)
2  {
3     Paivays tulos(p);
4     tulos.asetaPaiva(1); // Kuukauden alkuun
5     return tulos;
6  }
7
8  int main()
9  {
10    Paivays a(13, 10, 1999);
11    Paivays b(a); // Kopiorakentaja
12    b = kuukaudenAlkuun(a);
13 }
```

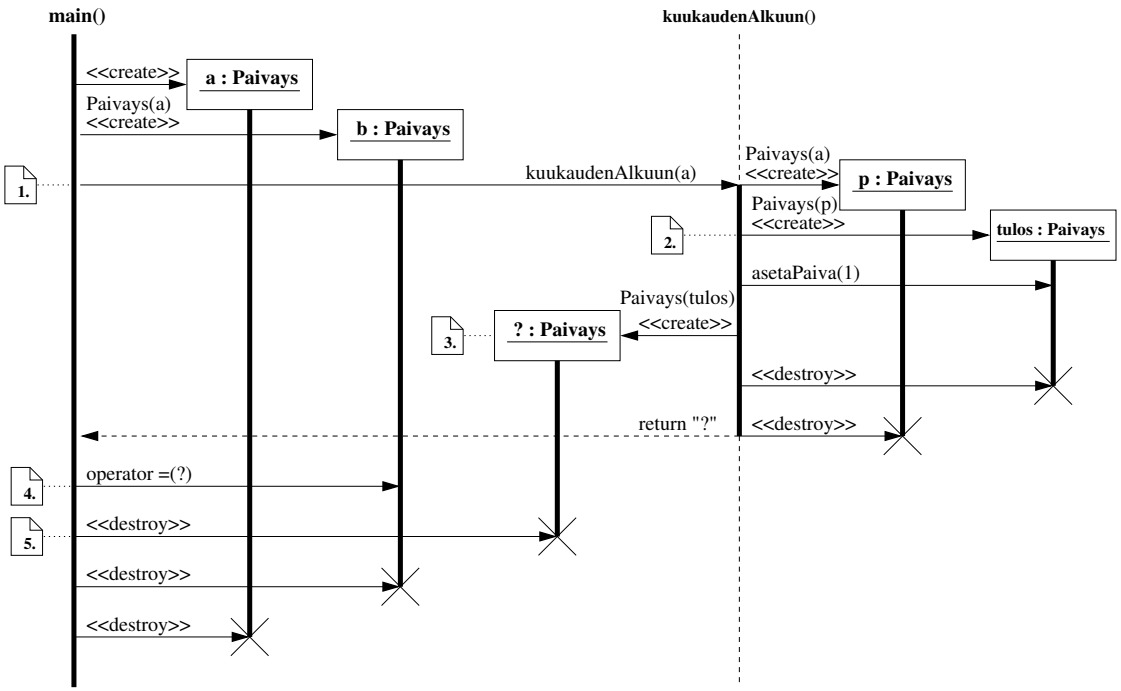
LISTAUS 7.8: Olio arvoparametrina ja paluuarvona

aivan yhtä itsestäänselvää vaihtoehtoa kuin viitteet parametrinvälityksen tapauksessa.

Ei pidä koskaan tehdä sitä virhettä, että yrittää välttää olion palauttamista paluuarvona käyttämällä viitettä samalla tavoin kuin arvoparametrien yhteydessä. Jos ohjelma palauttaa viitteen paikalliseen olioon, viitteen päässä oleva olio tuhoutuu funktiosta palattessa. Tällöin ohjelmaan jää viite olemattomaan olioon, jonka käyttäminen todennäköisesti aiheuttaa varsin vinkeitä virhetilanteita. Aliluvussa 11.7 esiteltävät automaattiosoittimet ovat yksi vaihtoehto, mutta usein paluuarvon korvaaminen viiteparametrilla on helpoin ratkaisu.

Listauksessa 7.8 edellisellä sivulla on funktio, joka käyttää olioita sekä arvoparametrina että paluuarvona. Kuva 7.6 näyttää vaihe kerrallaan, mitä funktiota kutsuttaessa tapahtuu. Funktiokutsun vaiheet ovat seuraavat:

1. Funktiokutsun yhteydessä parametrista `a` tehdään automaatti-



KUVA 7.6: Oliot arvoparametreina ja paluuarvoina

sesti kopiorakentajaa käyttäen kopio kutsuttavan funktion puolelle. Näin funktion parametri `p` saa saman "arvon" kuin `a`.

2. Funktion sisällä luodaan paluuarvoa varten olio `tulos` kopioimalla se parametrista `p`. Tämän jälkeen tulosolion päiväys siirretään kuukauden alkuun.

3. Funktio palauttaa paluuarvonaan olion `tulos`. Siitä luodaan kopiorakentajalla automaattisesti nimetön väliaikaiskopio kutsujan puolelle. Tämän jälkeen funktiosta poistutaan ja sekä `tulos` että `p` tuhotaan.

4. Funktion paluuarvo sijoitetaan olioon `b`. Tässä käytetään sijoitusoperaattoria, jolle annetaan parametriksi äsken saatu väliaikaisolio. Näin funktion paluuarvo saadaan talteen olioon `b`.

5. Rivin 12 lopussa väliaikaisoliota ei enää tarvita, joten se tuhoutuu automaattisesti.

7.4 Tyyppimuunnokset

Tyyppimuunnos (*type cast*) on operaatio, jota ohjelmoinnissa tarvitaan silloin tällöin, kun käsiteltävä tieto ei ole jotain operaatiota varten oikean tyyppistä. Olio-ohjelmoinnin kannalta suomenkielinen termi "tyyppi*muunnos*" on harhaanjohtava. Jos tyyppiä **`int`** oleva muuttuja `i` "muunnetaan" liukuluvuksi **`double`**, muuttujan `i` tyyppi ei tietenkään muutu miksikään, vaan se on edelleenkin kokonaislukumuuttuja. Tarkempaa olisikin sanoa, että tyyppimuunnoksessa kokonaislukumuuttujan `i` *arvon* perusteella tuotetaan *uusi liukulukuarvo,* joka jollain tavalla vastaa muuttujan `i` arvoa. Tässä suhteessa tyyppimuunnos muistuttaa suuresti kopiointia, jossa siinäkin tuotetaan olemassa olevan olion perusteella uusi olio. Kopioinnissa uusi ja vanha olio ovat samantyyppiset, tyyppimuunnoksessa taas eivät.

Edellisessä esimerkissä vanhan ja uuden arvon vastaavuus tietysti on, että sekä `i`:n että tuotetun liukuluvun numeerinen arvo on sama. On kuitenkin olemassa tyyppimuunnoksia, joissa tämä vastaavuus ei ole yksi-yhteen. Esimerkiksi sekä liukuluvut `3.0` että `3.4` tuottavat kokonaisluvuksi muunnettaessa arvon `3`. Vastaavuus voi olla muutakin