sen sijaan ollaan kirjoittamassa alusta alkaen kirjaston lainausrekisteriä ja tiedetään, että kaikki ohjelmassa esiintyvät kirjat ovat palautuspäivämäärällisiä kirjaston kirjoja, kannattaa ehkä suoraan lisätä palautuspäivämäärän käsittely luokan `Kirja` sisälle ja kenties nimetä luokka uudelleen.

Jos sen sijaan ohjelmassa tarvitaan sekä tavallisia kirjoja että kirjaston kirjoja, kahden luokan ja periytymisen käyttö kannattaa. Tavallinen kirjaolio ainakin vaatii vähemmän muistia kuin kirjastonkirjaolio, jonka täytyy myös muistaa päivämäärä. Lisäksi kirjaston kirjaa on vaikea käyttää tavallisena kirjana, koska esimerkissä uutta kirjaa luotaessa rakentajalle täytyy aina antaa palautuspäivämäärä.

6.5 C++: Virtuaalifunktiot ja dynaaminen sitominen

Joskus aliluokan olion on tarpeen suorittaa kantaluokasta perimänsä palvelu hieman kantaluokasta poikkeavalla tavalla. Toisella tavalla ilmaistuna aliluokka saattaa haluta periä kantaluokalta vain jäsenfunktion ulkoisen rajapinnan, mutta ei toteutusta. C++ tarjoaa aliluokalle mahdollisuuden tarjota oman toteutuksensa kantaluokalta perimälleen jäsenfunktiolle, jos kyseinen jäsenfunktio on kantaluokassa määritelty **virtuaaliseksi**. Tällaista virtuaalista jäsenfunktiota kutsutaan yleensä lyhyesti **virtuaalifunktioksi** (*virtual function*).

6.5.1 Virtuaalifunktiot

Jäsenfunktio määritellään kantaluokan esittelyssä virtuaaliseksi lisäämällä jäsenfunktion eteen avainsana **`virtual`**. Tämän jälkeen kantaluokasta periytettävillä aliluokilla on kaksi mahdollisuutta:

- Hyväksyä kantaluokan tarjoama jäsenfunktion toteutus. Tällöin aliluokan ei tarvitse tehdä mitään eli kantaluokan toteutus periytyy automaattisesti myös aliluokkaan.
- Kirjoittaa oma toteutuksensa perimälleen jäsenfunktiolle. Tässä tapauksessa aliluokan esittelyssä esitellään jäsenfunktio uudelleen, ja sen jälkeen aliluokan toteutuksessa kirjoitetaan jäsenfunktiolle uusi toteutus aivan kuin normaalille aliluokan jä-

senfunktiolle. Aliluokan esittelyssä avainsanan **`virtual`** toistaminen ei ole pakollista, mutta kylläkin hyvän ohjelmointityylin mukaista.

Olio-ohjelmoinnin kannalta on tärkeää, että muutettu jäsenfunktion toteutus tarjoaa kantaluokan kannalta ***saman palvelun*** kuin alkuperäinenkin. On myös huomattava, että aliluokka voi muuttaa vain virtuaalifunktion toteutusta, ei esimerkiksi paluutyyppiä tai parametrien lukumäärää tai tyyppiä. ISO C++ sallii nykyisin, että jos alkuperäinen paluutyyppi on osoitin tai viite luokkaan, niin uuden jäsenfunktion paluutyyppi voi olla osoitin tai viite alkuperäisen paluutyypin aliluokkaan. Tälle paluutyypin *kovarianssille* (*covariance*) ei kuitenkaan ole kovin usein käyttöä (mutta sitä tullaan kyllä käyttämään aliluvussa 7.1.3).[♂]

Listauksessa 6.5 seuraavalla sivulla on kaksi aiemmin esitellyn luokan `Kirja` jäsenfunktiota, jotka luokka määrittelee virtuaalisiksi, joten aliluokilla on mahdollisuus toteuttaa ne omalla tavallaan. Listauksessa 6.6 sivulla 162 on luokka `KirjastonKirja` määritellyt uudelleen jäsenfunktion `tulostaTiedot` niin, että se tulostaa myös palautuspäivämäärän. Sen sijaan jäsenfunktion `sopiikoHakusana` luokka perii kantaluokalta sellaisenaan toteutusta myöten.

Kirjaston kirjan tietojen tulostuksen toteutuksessa täytyy saada jotenkin tulostettua myös kantaluokkaosassa olevat kirjan tiedot. Tämän toteuttaa kantaluokan versio funktiosta `tulostaTiedot`, joten aliluokan uusi versio kutsuu sitä rivillä 29. Pelkkä kutsu `tulostaTiedot(virta)` kutsuisi rekursiivisesti aliluokan funktiota itseään, joten kyseisellä rivillä täytyy kertoa erikseen, että halutaan kutsua kantaluokan versiota funktiosta. Tämä tehdään lisäämällä funktion eteen määre `Kirja::`, joka määrää minkä luokan versiota kutsutaan.

6.5.2 Dynaaminen sitominen

Periytymishierarkiat ja virtuaalifunktiot saavat aikaan sen, että jäsenfunktion toteutus saattaa olla luokkahierarkiassa alempana, kuin mis-

[♂] Periaatteessa oliokieli voi sallia vastaavantyyppisen asian myös jäsenfunktion parametreissa, mutta tällöin periytymissuhteen täytyy mennä toiseen suuntaan – periytetyn luokan parametrit ovat kantaluokan parametrien kantaluokkatyyppiä. C++ ei kuitenkaan salli tätä **kontravarianssia** (*contravariance*), eivätkä salli monet muutkaan oliokielet. Näistä asioista voi halutessaan lukea mm. kirjasta “Theory of Objects” [Abadi ja Cardelli, 1996].

```
.......................  Kantaluokan esittelyssä  .......................
 1  class Kirja
 2  {
          ⋮
 8    virtual void tulostaTiedot(std::ostream& virta) const;
 9    virtual bool sopiikoHakusana(std::string const& sana) const;
10
11  private:
12    void tulostaVirhe(std::string const& virheteksti) const;
          ⋮
15  };
.....................  Kantaluokan toteutuksessa  .....................
19  void Kirja::tulostaVirhe(string const& virheteksti) const
20  {
21    cerr << "Virhe: " << virheteksti << endl;
22    cerr << "Kirjassa: ";
23    tulostaTiedot(cerr);
24    cerr << endl;
25  }
26
27  void Kirja::tulostaTiedot(ostream& virta) const
28  {
29    virta << tekija_ << " : \"" << nimi_ << "\"";
30  }
31
32  bool Kirja::sopiikoHakusana(string const& sana) const
33  {
34    return nimi_.find(sana) != string::npos ||
35      tekija_.find(sana) != string::npos; // Löytyikö nimestä tai tekijästä
36  }
```

LISTAUS 6.5: Luokan `Kirja` virtuaalifunktiot

sä jäsenfunktio on alunperin otettu mukaan rajapintaan. Tämä ja periytymisen “aliluokan-olio-kuuluu-kantaluokkaan” (*is-a*) -ilmiö saavat aikaan sen, että kääntäjä ei kaikissa tapauksissa pysty vielä käännösaikana päättelemään, mitä rajapintafunktion toteutusta on tarkoitus kutsua, vaan päätös tästä siirtyy ajoaikaiseksi. Tästä käytetään nimitystä **dynaaminen sitominen** (*dynamic binding*).

Listauksessa 6.7 sivulla 163 on funktio `tulostaKirjat`, joka ottaa parametrikseen taulukollisen kirjaosoittimia. Koska jokainen kirjaston kirja on periytymishierarkian mukaisesti myös kirja, tällainen taulukko voi sisältää todellisuudessa osoittimia sekä kirjoihin että

```
.............................  Aliluokan esittelyssä  .............................
   1  class KirjastonKirja : public Kirja
   2  {
             ⋮
   8     virtual void tulostaTiedot(std::ostream& virta) const;
             ⋮
  11  };
...........................  Aliluokan toteutuksessa  ...........................
  27  void KirjastonKirja::tulostaTiedot(ostream& virta) const
  28  {
  29     Kirja::tulostaTiedot(virta); // Erikseen pyydetään kantaluokan palvelua
  30     virta << ", palautus " << palpvm_;
  31  }
```

LISTAUS 6.6: Luokan `KirjastonKirja` virtuaalifunktiot

kirjastonkirjoihin, kuten listauksen riveiltä 24–28 näkyy. Mikään ei tietysti estä muita ohjelman osia periyttämästä `Kirja`-luokasta omia erikoistettuja kirjaluokkiaan, joita ne voivat myös lisätä taulukkoon.

Koska kirjan jäsenfunktio `tulostaTiedot` on virtuaalinen, voidaan sen toteutus määritellä uudelleen missä tahansa periytetyssä luokassa. Niinpä rivillä 14 kääntäjä tietää vain, että siinä kutsutaan *jotain* jäsenfunktion `tulostaTiedot` toteutusta. Kääntäjällä ei kuitenkaan kyseisellä rivillä ole mitään tietoa edes siitä, mitä toteutuksia jäsenfunktiolla voi olla, puhumattakaan siitä, että kääntäjä pystyisi valitsemaan näistä oikean. Niinpä kääntäjä tuottaa kyseiseen kohtaan ohjelmaa koodin, joka *ensin tarkastaa osoittimen päässä olevan olion todellisen luokan* ja vasta sen jälkeen kutsuu sille sopivaa jäsenfunktion toteutusta.

Kun nyt funktiota kutsutaan rivillä 30, tapahtuu seuraavaa: Parametrina annetun taulukon ensimmäinen alkio osoittaa luokan `Kirja` olioon. Rivin 14 koodi kutsuu tämän vuoksi jäsenfunktiota `Kirja::tulostaTiedot`. Silmukan seuraavalla kierroksella taulukon toinen alkio osoittaakin luokan `KirjastonKirja` olioon. Tällöin *täsmälleen sama* ohjelman rivi 14 kutsuukin jäsenfunktiota `KirjastonKirja::tulostaTiedot`, koska se on oikea toteutus tämäntyyppiselle oliolle.

Tällä tavoin dynaaminen sitominen mahdollistaa sen, että sama jäsenfunktiokutsu käyttäytyy *eri tavalla* riippuen siitä, minkä tyyppi-

```
10 void tulostaKirjat(vector<Kirja*> const& kirjat)
11 {
12   for (unsigned int i = 0; i != kirjat.size(); ++i)
13   {
14     kirjat[i]->tulostaTiedot(cout);
15     cout << endl;
16   }
17 }
18
19 int main()
20 {
21   vector<Kirja*> kirjaHylly;
22
23   // Huom! Alla olevasta puuttuu muistin loppumiseen varautuminen
24   kirjaHylly.push_back(
25     new Kirja("Axiomatic", "Greg Egan")); // [Egan, 1995]
26   kirjaHylly.push_back(
27     new KirjastonKirja("Matemaattisia olioita", "Leena Krohn",
28                        Paivays(31,10,1999))); // [Krohn, 1992]
29
30   tulostaKirjat(kirjaHylly); // Tulostetaan kirjat
31
32   for (unsigned int i = 0; i != kirjaHylly.size(); ++i)
33   {
34     delete kirjaHylly[i]; kirjaHylly[i] = 0;
35   }
36 }
```

LISTAUS 6.7: Dynaaminen sitominen C++:ssa

nen olio osoittimen tai viitteen päässä on. Jos kantaluokasta `Kirja` periytetään jossain vaiheessa jokin toinen kirjaluokka, esimerkiksi `MyytavaKirja`, funktioon `tulostaKirjat` ei tarvitse tehdä mitään muutoksia, vaan se osaa automaattisesti kutsua myös uuden kirjan tulostusjäsenfunktiota. Tässä mielessä funktion koodi on yleiskäyttöinen eli **geneerinen**.

Dynaaminen sitominen toimii myös, jos virtuaalifunktiota kutsutaan kantaluokan omasta jäsenfunktiosta. Listauksessa 6.5 sivulla 161 oli määritelty riveillä 19–25 luokan sisäinen jäsenfunktio `tulostaVirhe`, jota kirjaluokan jäsenfunktiot voivat käyttää virheilmoitusten tulostamiseen. Tämä koodi kutsuu jäsenfunktiota `tulostaTiedot`. Koska tämä jäsenfunktio on virtuaalinen, käytetään sen kutsumiseen dynaamista sitomista myös luokan omien jäsen-

funktioiden koodissa.

Kun nyt virheilmoitusfunktiota kutsutaan jollekin kirjaoliolle tai siitä periytetylle oliolle, tutkitaan `tulostaTiedot`-kutsun yhteydessä, *mikä on olion itsensä todellinen luokka*. Tämän jälkeen kutsutaan tämän todellisen luokan määräämää tulostusfunktiota. Näin kantaluokan omat jäsenfunktiot voivat käyttää hyväkseen aliluokissa määriteltyjä virtuaalifunktioiden toteutuksia, vaikka niillä ei edes ole mitään tietoa siitä, millaisia aliluokkia ohjelmassa on olemassa!

6.5.3 Olion tyypin ajoaikainen tarkastaminen

Kantaluokkaosoittimen päässä olevalle oliolle voi kutsua vain kantaluokan rajapinnassa olevia funktioita, vaikka osoittimen päässä todellisuudessa olisikin aliluokan olio. Tämä johtuu siitä, että käännösvaiheessa kääntäjällä ei ole mitään mahdollisuutta varmistua siitä, minkä tyyppinen olio kantaluokkaosoittimen päässä on. Normaalisti kantaluokan rajapinnan käyttäminen riittää ohjelmalle, ja dynaaminen sitominen mahdollistaa sen, että aliluokan olio käyttäytyy sille ominaisella tavalla.

Joskus tulee kuitenkin tarve päästä käsiksi aliluokan rajapintaan. Jos aliluokan olio on kuitenkin kantaluokkaosoittimen päässä, ei aliluokan rajapinta ole näkyvissä. Ainoa vaihtoehto on luoda osoitin aliluokkaan ja laittaa se osoittamaan kantaluokkaosoittimen päässä olevaan olioon. Tämän jälkeen aliluokan rajapintaan pääsee käsiksi käyttämällä saatua uutta osoitinta.

ISO C++:ssä on olion tyypin ajoaikaista tutkimista varten ominaisuus, jota yleisesti kutsutaan nimellä **RTTI** (*Run-Time Type Information*). Jotta olion tyyppiä voisi tutkia ajoaikana, täytyy olion luokassa olla vähintään yksi virtuaalinen jäsenfunktio. Tämä vaatimus toteutuu käytännössä aina, koska jokaisen kantaluokan purkajan tulisi aina olla virtuaalinen ja tämä virtuaalisuus periytyy myös aliluokkiin.

Kantaluokkaosoittimesta aliluokkaosoittimeksi

Muunnos kantaluokkaosoittimesta aliluokkaosoittimeksi onnistuu tyyppimuunnoksella `dynamic_cast<Aliluokka*>(kluokkaosoitin)`. Sen toiminta on kaksivaiheinen. Ensin tarkastetaan, että kantaluokkaosoittimen päässä oleva olio *todella on aliluokan olio* tai jonkin aliluokasta edelleen periytetyn jälkeläisluokan olio. Näin varmistetaan,

että olion voi turvallisesti sijoittaa aliluokkaosoittimen päähän. Kantaluokkaosoitinhan saattaa myös osoittaa kantaluokan tai jonkin toisen kantaluokasta periytetyn aliluokan olioon.

Jos kantaluokkaosoittimen päässä on väärän tyyppinen olio, palautetaan tyhjä osoitin 0. Jos kantaluokkaosoittimen päässä on oikean tyyppinen olio, palautetaan kyseiseen olioon osoittava aliluokkaosoitin. Tällä tavoin **dynamic_cast**-muunnoksen avulla ohjelma voi samalla kertaa tarkastaa, että olio todella on oikeaa tyyppiä, ja vielä saada olion oikeantyyppisen osoittimen päähän. Listaus 6.8 sisältää esimerkin tyyppimuunnoksen käytöstä.

dynamic_cast-muunnosta voi käyttää myös olioviitteisiin, siis tuottamaan aliluokkaviitteen, joka viittaa samaan olioon kuin annettu kantaluokkaviite. Tässä tapauksessa ainoa ero osoitinmuunnokseen on, että jos kantaluokkaviitteen päässä on väärän tyyppinen olio, **dynamic_cast** heittää poikkeuksen (jonka tyyppi on `std::bad_cast`). Syynä poikkeuksen heittoon on, että ei ole olemassa “tyhjää viitettä”, joka voitaisiin palauttaa virhetilanteessa.

Olion luokan selvittäminen

Edellä mainittu **dynamic_cast** on kätevä, kun halutaan päästä käsiksi aliluokan laajennettuun rajapintaan silloin, kun aliluokan olio on kantaluokkaosoittimen päässä. Samoin **dynamic_cast** on riittävä, jos vain halutaan testata, toteuttaako kantaluokkaosoittimen päässä ole-

```
1  bool myohassako(Kirja* kp, Paivays const& tanaan)
2  {
3     KirjastonKirja* kkp = dynamic_cast<KirjastonKirja*>(kp);
4     if (kkp != 0)
5     { // Jos tultiin tänne, kirja on kirjastonkirja
6        return kkp->onkoMyohassa(tanaan);
7     }
8     else
9     { // Jos tultiin tänne, kirja ei ole kirjastonkirja
10       return false; // Ei siis ole myöhässä
11    }
12 }
```

LISTAUS 6.8: Olion tyypin ajoaikainen tarkastaminen

va olio tietyn aliluokan rajapinnan eli kuuluuko olio tiettyyn aliluokkaan tai johonkin siitä periytettyyn jälkeläisluokkaan.

Joissain ***erittäin harvinaisissa tilanteissa*** tulee kuitenkin tarve saada selville, mihin luokkaan olio kuuluu, ja kenties vielä tallettaa tämä tieto johonkin tietorakenteeseen. Tähän **dynamic_cast** ei kelpaa, koska siltä voi vain kysyä, kuuluuko olio *tiettyyn* luokkaan tai sen jälkeläisluokkaan. Tällaista luokan kysymistä varten C++:stä löytyy operaattori **typeid** ja luokka type_info.

typeid-mekanismin käyttöön tulisi kuitenkin suhtautua *erittäin* suurella varauksella. Lähes kaikissa tapauksissa virtuaalifunktioiden (tai joskus kenties **dynamic_cast**in) käyttö on parempi, uudelleenkäytettävämpi ja selkeämpi vaihtoehto. Esimerkiksi toteutus, jossa funktiossa kysytään **typeid**:llä luokan tyyppi ja sitten **if**-lauseilla valitaan sopiva koodi, ei ole hyvää suunnittelua. Se vaatisi, että luokkia lisättäessä lisätään funktioon aina uusi vaihtoehto jokaista uutta luokkaa varten. Sen sijaan kannattaa lisätä luokkien yhteiseen kantaluokkaan virtuaalifunktio, jonka toteutukset periytetyissä luokissa sitten suorittavat halutun toiminnallisuuden. Näin saadaan kaikki luokkaan liittyvät asiat kapseloitua samaan paikkaan.

Luokka type_info esitellään komennolla **#include** <typeinfo>. Luokan oliot "edustavat" jokainen jotain ohjelman luokkaa. Olion luokan saa selville lausekkeella **typeid**(olio), joka palauttaa sellaisen type_info-olion, joka edustaa olion luokkaa. Lauseke **typeid**(Luokka) palauttaa taas annettua *luokkaa* vastaavan type_info-olion.

Kahta type_info-luokan oliota voi vertailla keskenään normaaleilla operaattoreilla == ja !=. Oliot ovat keskenään yhtä suuria, jos ne edustavat samaa luokkaa, muuten erisuuria. Osoittimia näihin olioihin voi sitten ohjelmassa käyttää vertailuavaimina, jos esimerkiksi jostain tietorakenteesta pitää etsiä haluttuun luokkaan kuuluva olio. Luokan type_info rajapinnassa on myös jäsenfunktio name, joka palauttaa oliota vastaavaa luokkaa kuvaavan merkkijonon. Tämän tarkka muoto on kääntäjäkohtainen eikä välttämättä pelkkä luokan nimi.

Sen testaaminen, onko osoittimen kp päässä oleva olio kirjaston kirja, käy periaatteessa vertailulla

```
if (typeid(*kp) == typeid(KirjastonKirja))
```

Tämä kuitenkin testaa vain, onko osoittimen päässä *täsmälleen* luokkaan KirjastonKirja kuuluva olio. Olio-ohjelmoinnin mukaan myös

jokainen tästä luokasta *periytetty* olio on kirjaston kirja, joten tällainen täsmällinen testaus ei yleensä ole se mitä halutaan. Tämän vuoksi **dynamic_cast** on yleensä oikea vaihtoehto tällaisiin testeihin ja **typeid**-operaattorin käyttö kannattaa jättää tilanteisiin, joissa tieto olion luokasta talletetaan jonnekin tai välitetään parametrina. Listaus 6.9 näyttää esimerkin koodista, joka etsii taulukosta ensimmäisen annettuun luokkaan kuuluvan olion.

6.5.4 Ei-virtuaalifunktiot ja peittäminen

Dynaamista sitomista käytettäessä kääntäjän on osattava jäsenfunktiokutsun yhteydessä tuottaa ylimääräinen koodi, joka tarkastaa olion todellisen tyypin ajoaikana. Tämän vuoksi kääntäjän on käsiteltävä

```
1  #include <typeinfo>
2  using std::type_info;

       ⋮
13 vector<Kirja*> ktaulukko;
14
15 Kirja* haeEnsimmainen(type_info const& kirjanTyyppi)
16 {
17   for (unsigned int i = 0; i < ktaulukko.size(); ++i)
18   {
19     if (typeid(*ktaulukko[i]) == kirjanTyyppi)
20     {
21       return ktaulukko[i];
22     }
23   }
24   return 0; // Ei löytynyt
25 }
26
27 int main()
28 {
29   // Etsitään ensimmäinen Kirjastonkirja
30   Kirja* kp = haeEnsimmainen(typeid(KirjastonKirja));
31   if (kp != 0)
32   {
33     kp->tulostaTiedot(cout);
34   }
35 }
```

LISTAUS 6.9: Esimerkki **typeid**-operaattorin käytöstä

virtuaalifunktioita ja tavallisia jäsenfunktioita eri tavalla — tavallisen jäsenfunktiokutsun yhteydessähän kääntäjä tietää jo käännösaikana, minkä luokan jäsenfunktiota kutsutaan.

C++ antaa ohjelmoijalle myös mahdollisuuden määritellä periytetyssä luokassa uudelleen kantaluokan *ei-virtuaalisia* jäsenfunktioita. Tällaisten jäsenfunktioiden kutsumisessa ei kuitenkaan käytetä dynaamista sitomista vaan aliluokan uusi toteutus **peittää** (*hide*) kantaluokan toteutuksen (tarkasti ottaen *kaikki* kantaluokan samannimiset jäsenfunktiot) *aliluokassa*. Tämä tarkoittaa sitä, että mikäli kyseistä jäsenfunktiota kutsutaan suoraan aliluokan oliolle, kutsutaan aliluokan toteutusta. Jos taas kutsu tapahtuu kantaluokkaosoittimen tai -viitteen kautta, kutsutaan *kantaluokan* toteutusta. Näin kutsuttava jäsenfunktio riippuu täysin siitä, *miten* jäsenfunktiota ohjelmassa kutsutaan.

Tällainen kutsutilanteesta riippuva jäsenfunktion valinta ei ole olio-ohjelmoinnissa lähes koskaan toivottavaa. Se tapahtuu kuitenkin helposti vahingossa silloin, kun *kantaluokan jäsenfunktion esittelystä unohtuu avainsana* ***virtual***. Vaikka avainsana olisikin paikallaan aliluokassa, kantaluokkaosoittimen läpi ei tällöin käytetä dynaamista sitomista ja ohjelma valitsee tällaisissa tilanteissa väärän toteutuksen jäsenfunktiolle.

Koska **virtual**-sanan unohtuminen kantaluokan esittelystä ei aiheuta käännösvirhettä vaan väärän toiminnan, on erittäin tärkeää, että kantaluokissa muistetaan merkitä **virtual**-sanalla kaikki sellaiset jäsenfunktiot, joiden toteutus saatetaan määritellä uudellen aliluokissa! Useimmissa muissa oliokielissä dynaamista sitomista käytetään oletusarvoisesti, joten niissä tällaista vaaratekijää ei ole.

6.5.5 Virtuaalipurkajat

Olio-ohjelmissa tulee varsin usein esiin tilanne, jossa kantaluokkaosoitin laitetaan osoittamaan aliluokan olioon, joka on luotu dynaamisesti **new**'llä. Tällainen tilanne vaatii hieman erikoistoimenpiteitä, jos olio aiotaan ohjelmassa tuhota **delete**llä kantaluokkaosoittimen kautta.

Jotta olioiden tuhoaminen kantaluokkaosoittien kautta toimisi oikein, *kantaluokassa* täytyy merkitä luokan purkaja virtuaaliseksi lisäämällä sen eteen avainsana **virtual**. Sama tehdään hyvän tyylin mukaisesti myös aliluokkien purkajissa, mutta kielen kannalta se ei

enää ole välttämätöntä, vaan purkajan virtuaalisuus periytyy aliluokkiin automaattisesti.

Mikäli olio tuhotaan kantaluokkaosoittimen kautta ilman, että kantaluokan purkaja on virtuaalinen, ohjelman toiminta on C++-standardin mukaan määrittelemätöntä. Käytännössä on mahdollista, että ohjelma kaatuu tai sitten kutsuu tuhottavalle oliolle vääriä purkajia tai toimii muuten väärin. Näiden virheiden vuoksi on tärkeää, että jokaisen **kantaluokan purkaja määritellään virtuaaliseksi!**

Ongelma juontaa juurensa siitä, että kääntäjä ei **`delete`**n koodia tuottaessaan pysty osoittimen tyypistä päättelemään, minkä tyyppinen olio osoittimen päässä on. Historiallisista ja optimointiteknisistä syistä C++ ei oletusarvoisesti yritä päätellä osoittimen päässä olevan olion todellista tyyppiä ajoaikaisesti, vaan ohjelman toiminta on jätetty määrittelemättömäksi. Kantaluokan purkajan virtuaalisuus toimii vinkkinä kääntäjälle, jolloin kääntäjä generoi tuhoamisen yhteyteen koodin, joka tarkastaa olion todellisen tyypin ja tuhoaa sen asianmukaisella tavalla.

Useimmissa muissa oliokielissä aliluokan olion tuhoaminen kantaluokkaviitteen läpi on tehty aina oikein toimivaksi tai olioiden tuhoutuminen tapahtuu aina automaattisesti. Niinpä niissä ei yleensä ole mitään C++:n tapaisia ansoja tässä suhteessa.

6.5.6 Virtuaalifunktioiden hinta

Mikään hyvä ei ole koskaan ilmaista. Virtuaalifunktiot ja dynaaminen sitominen tekevät mahdollisiksi todella joustavat ohjelmarakenteet, joissa jäsenfunktion kutsujan ei tarvitse tietää yksityiskohtia siitä, mitä jäsenfunktion toteutusta kutsutaan. Tämä parantaa ohjelman ylläpidettävyyttä, laajennettavuutta sekä luettavuutta. Dynaamisella sitomisella on kuitenkin hintansa.

Mikäli dynaamista sitomista käytetään, ohjelman koodin täytyy aina virtuaalifunktion kutsun yhteydessä tarkastaa kutsun kohteena olevan olion todellinen luokka ja valita oikea versio jäsenfunktion toteutuksesta. Tämä valinta jää lähes aina ajoaikaiseksi, joten valinnan tekeminen hidastaa aina jäsenfunktion kutsumista hiukan. Käytännön testit osoittavat, että jäsenfunktion kutsuminen dynaamista sitomista käyttäen on yleensä noin 4 % hitaampaa kuin normaalisti [Driesen ja Hölzle, 1996]. On kuitenkin muistettava, että mikäli dynaamista sitomista todella tarvitaan ohjelmassa, vaatisi ilman virtu-

aalifunktioita kirjoitettu koodi joka tapauksessa ylimääräistä koodia dynaamisen sitomisen matkimiseen, joten virtuaalifunktioiden hinta on todellisuudessa jonkin verran pienempi.

Suoritusnopeuden lisäksi virtuaalifunktiot vaikuttavat olioiden muistinkulutukseen. Mikäli luokassa tai sen kantaluokassa on yksikin virtuaalifunktio, täytyy luokan olioihin tallettaa jonnekin tieto siitä, minkä luokan olioita ne ovat. Yleensä tämä tapahtuu **virtuaalitaulujen** ja **virtuaalitauluosoittimien** avulla (niistä voi lukea enemmän vaikkapa kirjoista "Inside the C++ Object Model" [Lippman, 1996] ja "The Design and Evolution of C++" [Stroustrup, 1994]). Tämä tieto vie useimmissa kääntäjissä muistia yhden osoittimen verran, joten virtuaalifunktioita käyttävien luokkien olioiden muistinkulutus kasvaa useimmissa järjestelmissä 4 tavun verran.

Tämä lisämuistinkulutus ei riipu luokan virtuaalifunktioiden määrästä, joten olioiden koko ei enää ensimmäisen virtuaalifunktion lisäämisen jälkeen kasva, vaikka virtuaalifunktioita olisikin useita. Koska jokaisella kantaluokalla tulisi joka tapauksessa olla virtuaalipurkaja, ei muiden jäsenfunktioiden määrittely virtuaaliseksi käytännössä vaikuta olioiden muistinkulutukseen.

Olioiden koon kasvun lisäksi kääntäjä joutuu yleensä varaamaan jonkin verran muistia jokaista virtuaalifunktioita sisältävää *luokkaa* varten. Tätä lisämuistia tarvitaan tyypillisesti yhdestä kolmeen osoittimen verran jokaista luokassa olevaa virtuaalifunktiota varten. Koska ylimääräistä muistia varataan kuitenkin vain kerran koko luokkaa kohti eikä sen määrä riipu luokan olioiden määrästä, sen vaikutus ohjelman muistinkulutukseen on yleensä mitätön.

On muistettava, että kääntäjällä on aina lupa optimoida koodia. Edellä mainittu ohjelman hidastuminen on mahdollinen, mutta jos kääntäjä pystyy jo käännösaikana päättelemään, ettei dynaamista sitomista tarvita jossain yhteydessä, se voi tietysti tuottaa myös tehokkaampaa koodia.

6.5.7 Virtuaalifunktiot rakentajissa ja purkajissa

Normaalisti dynaamista sitomista käytetään aina, kun virtuaalifunktiota kutsutaan, ja näin kutsutaan aina "oikeaa" versiota virtuaalifunktiosta. Tästä ovat kuitenkin poikkeuksena rakentajissa ja purkajissa tapahtuvat virtuaalifunktioiden kutsut.